X11 Window System
Core Distribution

Volume II

Integrated Solutions
1140 Ringwood Court
San Jose, California 95131
(408) 943-1902

490244 Rev. A

September 1988

PREFACE

This manual is based on the Core Distribution Documentation from MIT. It is divided into two volumes as follows:

Volume I contains documentation on:

- Protocol
- Xlib
- Intrinsics
- Widgets
- Conventions

Volume II contains documentation on:

- Andrew Toolkit
- Xrlib Toolbox

Although ISI provides documentation and software for Andrew Toolkit and Xrlib Toolbox, ISI does not offer technical support for these packages.

Introducing the Andrew Toolkit

Draft Edition

February 14, 1988

Diane Langston and
Dennis Grantham, Editors

Information Technology Center
Carnegie Mellon University
Pittsburgh, PA 15213

A Joint Computing Venture
between IBM and
Carnegie Mellon University

Table of Contents

Preface

Parts of this Document

This document contains three major parts, each of which contains various sections:

Part 1, **Overview, Installation, and Testing,** is intended for system administrators and experienced UNIX system users. It provides a brief overview of the Andrew Toolkit, followed by extensive installation and testing instructions.

Part 2, A **Programmer's View of the Andrew Toolkit,** presents a series of excerpts from our two-volume *Programmer's Guide to the Andrew Toolkit.* The first section, taken from the Tutorial Volume, provides a brief introduction to object-oriented programming followed by a selected group of tutorial programming examples. The second section, taken from the Reference Manual, provides a brief explanation of the View class, one of the key components of the Andrew Toolkit. The excerpt sections are followed by a third section, containing a quick reference guide, Selected Interfaces to the Andrew Toolkit.

Part 3, **A User's View of Andrew Toolkit Applications,** contains excerpts from *A Guide to Andrew*, a fully-illustrated, tutorial user guide that is widely used on the Carnegie-Mellon University campus and at other Andrew sites.

For Additional Information

For additional information about the Andrew Toolkit and announcements of upcoming workshops and training, please fill out the form below or send electronic mail to:

andrew-toolkit-info@andrew.cmu.edu

Available documentation

The Andrew Toolkit--An Overview

1988 USENIX Conference Paper written by the developers of the Andrew Toolkit.

Available at no cost.

Programmer's Guide to the Andrew Toolkit

Volume I: Tutorial Introduction (250 pages)

Volume II: Reference (360 pages)
Available for $25.00 per printed set.

A Guide to Andrew

An illustrated 140-page user's guide to the essential features of the Andrew System as it operates at Carnegie Mellon University. This document is shrink-wrapped with a three-ring binder.

Available for $10.00

Order form available on the next page.

Andrew Toolkit Information

Information Technology Center

Carnegie-Mellon University

Pittsburgh, PA 15213-3890

(412) 268-6700

___ Please add me to the Andrew Toolkit mailing list.

___ Please send me a copy of the article *The Andrew Toolkit--An Overview*, at no cost.

___ Please send ____ copies of the *Programmer's Guide to the Andrew Toolkit* at $25.00 per set..

___ Please send ____ copies of *A Guide to Andrew* at $10.00 per copy. Please specify whether you want the ___ Andrew File System or ___ Standalone version. The Standalone version will not be available in hard copy until late March. The Andrew File System version, from which the excerpts in this document are taken, is already available.

Name ____

Title ____

Company ____

Address ____

City ____________________ State __________ Zip __________

Phone ____

Electronic Mail ____

Overview, Installation, and Testing

Sections in this Part

This part of the document contains the following sections:

The first section, **A Brief Overview of the Andrew Toolkit**, is condensed from a paper presented at the 1988 USENIX Conference. It briefly describes the design and operation of the Andrew Toolkit, a high-level programming environment for the development of flexible, expandable, and innovative objects and user interface applications on graphic workstations.

The second section, **Installing the Andrew Toolkit**, is intended for system maintainers. It contains a series of instructions for compiling and installing the Andrew Toolkit. Later parts describe how to set up the application environment, followed by a brief startup section for a few applications. For detailed application testing scripts, see the following section.

The third section, **Testing the Andrew Toolkit**, contains the testing scripts used to verify the installation and operation of the Cambridge Window Manager (cwm) and important Andrew Toolkit applications. It is intended for use by system maintainers or knowledgeable users. The section begins with a recap of the actions needed to set up the application environment, then follows with testing scripts for cwm, Typescript, Console, Help and Messages.

A Brief Overview of the Andrew Toolkit

This section is excerpted from a longer article, "The Andrew Toolkit--An Overview," presented at the 1988 USENIX Conference in Dallas, Texas. Reprints of the entire article are available at no cost. Simply fill out the Documentation Order Form at the beginning of this documentation set.

Introduction

UNIX, and its software tools approach to computing, provided a new paradigm for building applications. Portable, general purpose modules could be connected in different ways to create complex applications without duplication of effort. The Andrew Toolkit can be seen as an extension of this concept to the graphic workstation environment. The Toolkit's high-level programming interface provides a framework for developing and combining components into user interface applications. Its emphasis on the use of high-level components provides developers with the freedom to combine existing components into new applications or to create and integrate an unlimited variety of new components and applications into the environment. It also allows for code-sharing between working applications and promotes the use of consistent user interface elements across applications.

The original design of the Andrew Toolkit rose from discussions about the development of a text editor that would allow the user to embed other components such as tables, drawings, rasters, and animations within the text. After an initial examination of the problems involved with this approach, we developed a general architecture which allows for the inclusion of one component inside another. This architecture allows developers to build components that can embed other components without having detailed knowledge about the embedded component. Further, it allows for the development of new components that can be easily embedded in existing components.

The Toolkit provides the usual set of simple components (menu, scroll bars, etc) and a number of higher-level editable components including multi-font text, tables/spreadsheets, drawings, equations, rasters and simple animations. The text and table components are multi-media components that allow the embedding of other components within themselves. The drawing component will soon support this feature as well. The editable components can be run as standalone programs or through a multi-media editing program called EZ.

Current Toolkit Applications

Users of Andrew Toolkit applications can currently compose papers that contain tables, equations, drawings, rasters and animations. The multi-media text and table components use a generic mechanism to include other components. If a new component is developed, it can be included in text or table using that same mechanism. This is an important feature of the system, especially within a university environment. Given our limited resources we knew from the outset that we could not write all the components that were required by the university, because the range of possibilities is limitless. For example, members of the music department will want to include musical scores inside of text just as easily as business students can include charts or spreadsheets. Members of the electrical engineering department will want to include circuit diagrams inside of text, while programmers will want to include formatted sections of code.

In addition to the multi-media editor, we have used the Toolkit to develop a number of other basic applications including a help system, a typescript facility that provides an enhanced interface to the C-shell, a *ditroff* previewer, and a system monitor (console) that displays status information such as the time, date, CPU load and file system information. Since both the mail and help applications use the text component for the display of information, they automatically inherit the multi-media functionality of the text component.

The basic toolkit applications (editor, mail, help, preview, typescript, console) have been in general use since August 1987 on the Carnegie Mellon campus, where the Andrew System is actively used by approximately 3000 people. In January 1988, the user community received the multi-media editing applications, along with a number of new extension packages for the editor. These packages include a C-language programming component, a compile package, a tags package, a spelling checker, a new style editor and a filter mechanism, which gives the user the ability to use standard tools on regions of text contained in a file being edited.

The Andrew Class System

Support for the use of multiple components within applications comes from the Andrew Class System (Class), the object-oriented system on which the Andrew Toolkit is built. Class is a C language-based system, consisting of a small run-time library and a simple preprocessor that only preprocesses class header files. The object-oriented system in Class is similar to the one used in C++.

Class provides an object-oriented environment with single-inheritance. The Class language permits the definition of object methods and class procedures. Object methods are similar to C++ methods, and may be overridden in subclasses. Class procedures are similar to Smalltalk's class methods, but they may not be overridden. C procedures for controlling the initialization and disposition of objects are created by the Class preprocessor.

In addition to providing a high-level language for developing new components, Class can preprocess components and dynamically load and link code needed to support them. Dynamic linking provides a high degree of extendability to Andrew Toolkit applications like the multi-media editor EZ, which can edit a wide variety of components by loading the appropriate code when needed. Further, the dynamic loading facility can be used to add additional components to the basic toolkit without having to rebuild existing applications.

For example, if a member of the music department creates a music component and embeds that component into a text component (or any other component that allows embedding of components), the code for the music component will be dynamically loaded into the application. Except for a slight delay to load the code, the user of the editor is unaware that the music component was not statically loaded. The user is also unaware that the music component was not part of the original system. The editor did not have to be recompiled, relinked, or otherwise modified to use the new music component. Further, all users of the text component automatically acquire the ability to use the music component: it can be sent in a mail messages easily as edited in a document.

The dynamic loading/linking feature of Class also provides a low-level extension language for applications built using the Toolkit. Knowledgeable users can write code, using Class, to gain access to new commands. These commands can be bound either to key sequences or to menus and used to customize an application. When the application initializes, the code is loaded. The user can then execute the additional commands as needed.

In addition, Class's dynamic loading/linking feature gives us the ability to run all of our applications from a single base program. This program, called *runapp*, contains all the basic components of the Toolkit. The code for each application is then dynamically loaded at run time. Since most UNIX systems do not provide shared libraries, a program like *runapp* allows multiple Toolkit applications to share a significant portion of code. This leads to performance improvements number of areas, particularly in reducing paging and virtual memory use, reducing the file size of an application, and reducing the time needed for file fetches.

Basic Toolkit Components: Data Objects and Views

The Andrew Toolkit is based on the development of components that can be used as building blocks for applications or other, more complex components. These components are frequently referred to as *objects*. Two of the most important object types within the Andrew Toolkit are *data objects* and *views*. A complex component, such as an editing package, is normally composed of a view/data object pair. The data object contains the information that is to be displayed, while the view provides display and editing features (a user interface) suited to that type of data object. For example, the text data object contains the actual characters, style information and pointers to any data objects embedded within it. It also provides ways to alter the data, such as inserting characters and deleting characters. The text view contains information such as the current selected piece of text, the portion of the text that is currently visible, and the location of the text. The text view provides methods for drawing the text, handling various input events (mouse, keyboard, menus), and manipulating the visual representation of the text.

The contents of a data object can be saved into a file, but the contents of the view cannot. A default view is created for each data object each time an application is run. The information associated with the view is temporary; changes to the default view setting can exist only while the application is running. Views communicate with the *drawable* object in the Graphics Layer to produce screen and printed output. (See the section about the Graphics Layer for details).

While it is often the case that a view has an underlying data object, there are many cases when a view will be used to solely provide a user interface function. In such a case there is no underlying data object. The scroll bar is one such example. It only adjusts the information contained in another view.

We have made the view/data object distinction to provide a system where multiple views can simultaneously display the information contained in a single data object. Our design is similar to the Model-View-Controller design used in Smalltalk systems. By comparison, our data objects serve as the models, our views are views, and the controller is distributed between the interaction manager (global decisions) and individual views (decisions between children and parent views).

The view-data object brings many advantages. For example, the editor allows a user to edit the same information in more than one window and show the changes in both. This case is handled by having two views of the same type, one in each window, displaying information from the same data object. Similarly, a user might want to have multiple views of the same type on a single data object in one window. A system like Aldus' PageMaker(TM) could be built under the Andrew Toolkit by allowing the user to specify a set of views and their placement on a page. Some of those views (for example, the text views) would be examining different sections of the same data object.

It is also possible to have two different types of views displaying information contained in the one data object. Currently the text view is a display-based text processing system. It can be characterized as a semi-WYSIWYG (What You See Is What You Get) or a "WYSLRN" (What You See Looks Real Neat) view. It displays text with multiple fonts, indentations, etc. but makes no attempt to display the information as it would appear on a piece of paper. This view has been used for the basic text editor as well as the mail and help systems. It is quite useful for editing on a workstation, but not useful when the user wants to format the text for printing. In the case of printing, we plan to provide a full WYSIWYG text view. This paper-based text view will be designed to use the same text data object. So, the user will be able to choose either view, or both, while formatting the final version of a paper. Because both views will use the same data object, changes made in one window will automatically be reflected in the other.

Just as it is possible to have two different views on the same data object in two windows, it is also possible to have two different views on the same data object within the same window. A text component could have two embedded views on the same data object. For example, the user might want to display a table of numbers and a pie chart representing the table. This could be done by having one table data object and two views, a normal table view and a pie chart view.

While there are many advantages to separating information into data objects and views, it does introduce some novel problems. Two of the most important problems solved by the Toolkit have to do with coordinating updates between data objects and views and maintaining a stable view state.

The view/data object separation does not encourage close coordination between the changing of the data object's information and the update of a view's appearance. Since only one view causes the data object to change, and multiple views may have to reflect the change, a delayed update mechanism must be used. When the user issues a command to a view to alter the underlying data object, the view first requests that the data object modify itself and then requests the data object to inform all of its views that it has changed. When a view is informed that the underlying data object has changed, it must determine what the change is and update its visual representation appropriately.

This delayed update mechanism is the trickiest challenge in building a data object/view pair. The developer must create some mechanism by which the view can determine which portion of the data object has changed. This mechanism is normally provided by a set of methods exported by the data object. However, by design, data objects do not have detailed knowledge of a specific type of view. This would be one way to handle the delayed update, but this would preclude the development of other views on the same type of data object.

Retaining a stable (or permanent) state for a view is another problem. In the chart example, the underlying data object is a table of values. When a file displaying the chart is saved, only those values (along with the information that a "chart" is viewing the table) is saved. However, the user may have set certain parameters in the chart, such as the way to label the axes. This information is not part of the table data object and would not be stored in it. Since a view has no permanent state, information kept in the view, such as axes labelling, would also not be saved. There is simply no place to keep this view-specific information.

The Andrew Toolkit provides a solution for both of these problems. The solution consists of two parts: additional data objects and the idea of an observer. In the example above, the chart view would be viewing not a table data object but an auxiliary chart data object. The chart data object would retain information such as axes labelling. In addition, the chart data object would be an observer of the table data object. As information in the table changed, the chart data object would be notified and it, in turn, would notify the chart view. In fact, we do not have specially defined auxiliary data objects. Rather, our update system is based on the observer mechanism, where a data object may be observed by any number of other data objects and views. We have found that this design has permitted a great deal of flexibility and functionality for combining pieces in the Toolkit.

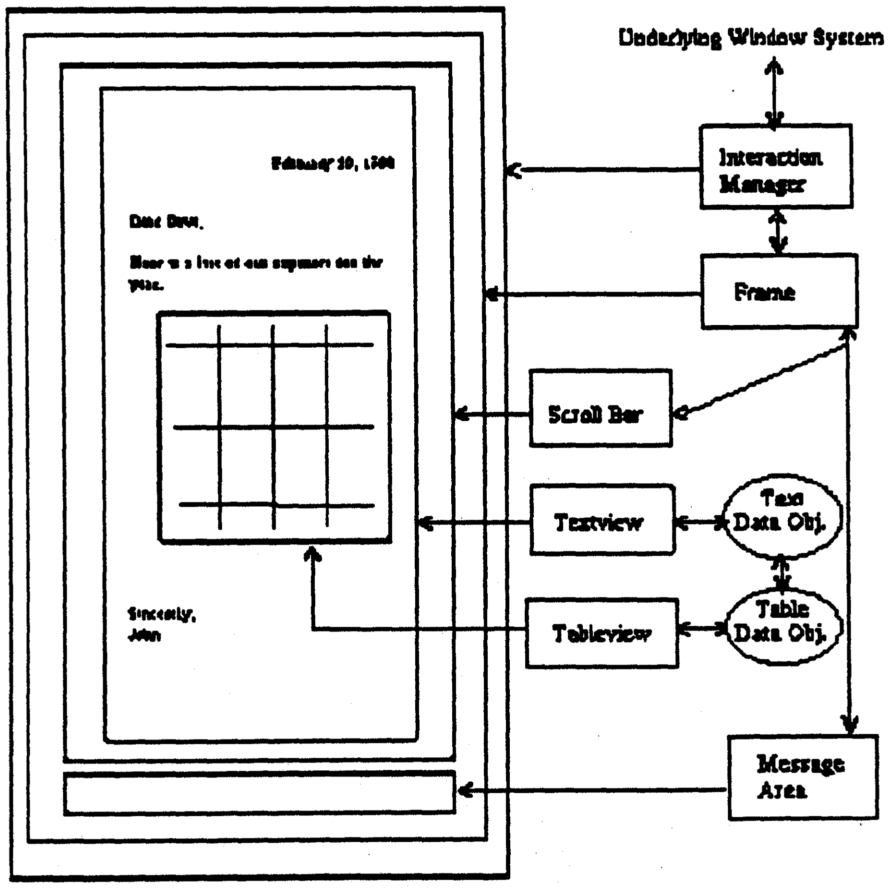
Event Processing: Andrew's View Tree

The flexibility of the Andrew Toolkit's view architecture is based on a number of key concepts and relationships. The Andrew Toolkit differs significantly from other toolkits in its handling of views because it relies on distributed, rather than centralized, control of input events and mediates the relationship of various objects through a series of parent-child relationships. These design elements help to overcome limitations in other systems which tied event handling closely to the physical relationship of components on the screen. They also allow for a higher degree of cooperation and consistency between the various objects in an application. Each application presents the user with a similar set of physical relationships, regardless of the objects being used, but provides an extra level of control and predictability at times when the physical display of elements is insufficient.

As noted earlier, views are used to define the user interface for an application. Generally, this consists of displaying the information in the data object on the screen and handling input events that might change the display. All views for an application are organized into a tree structure, with each child view appearing as a rectangle that is completely contained in its parent view. The *interaction manager* is at the top of the view tree, inside a window provided by the underlying window system. The interaction manager translates input events such as key strokes, mouse events, menu events and exposure events from the window system to the rest of the view tree. It also synchronizes drawing requests between views. By design, the interaction manager has one child view, of arbitrary type, which can be a parent to any number of child views. Child views are related only to their parents, not to each other.

In general, when an event is received by the interaction manager, the event is passed down to the interaction manager's child. That view determines whether it should accept the event or pass it down to one of its children. This process recurs until some view actually handles the event. By passing the event down the view tree, each parent view gets the chance to determine the disposition of the event, using the semantic information that is associated with itself.

Updates to the visual image of an application are handled in a similar fashion. When a view wants to update its image it makes a request to its parent view. That request is usually passed up to the interaction manager which then sends an update event back down the tree. Since a view might be embedded in another view, it does not have complete control over its allocated screen space; its parent view might have placed some other image on top of its image. However, by posting an update request up the tree and having the update event come back down, the parent is able to update its image and the child's image in the appropriate order.



The preceding figure presents a view tree for a window that contains a scrollable text view that contains a table view. The text view is surrounded by a scroll bar, which is surrounded by a frame that provides a message area view at the bottom of the window. The text and table views reference their respective data objects. The lines around the screen image represent the physical area of the image associated with each view.

When a user sends a mouse event to this window, the event is received by the interaction manager and passed down to the frame view. The frame view then determines whether it should handle the event or pass it to one of its children. (Note that the frame physically divides its image into two areas, the scrollable area and the message area, which separated by a thin line.) In order to allow the user to easily drag that line (to resize the message area), the frame allocates a slightly larger area to accept mouse events. That area overlaps the space allocated to the frame's children. If the handling of events was dictated by the screen layout alone, this interaction would be much more difficult to provide and would require more detailed knowledge of the view tree structure to maintain.

The frame accepts the mouse event directly if it is close to the dividing line between its two children. If the mouse event is passed down to the scroll bar view, that view will accept the mouse event if it is over the scroll bar or pass it to its child if it is not. The text view accepts the mouse event if it is not within a child view, such as the table; otherwise it too passes the event down.

As the mouse event works its way down the view tree, the view determining the disposition of the mouse event only needs to be aware of the location of its children and not the children's types. Similarly, the child does not need any knowledge about the type of its parent, or its parent's location in the view tree.

Menus are another place where the parent-child relationship comes into play. All Andrew Toolkit applications are expected to have a generic set of menu options, such as those related to copying and pasting selected regions; switching files in buffers; saving and printing files; and quitting the application. However, the specialized editing objects must also provide specialized menu options, regardless of whether they are the parent or child of another object. Because parent objects retain some control over children, the parents are able to pass along the generic menu options and then allow the children to fill in the specific editing options they need to provide. This cooperative relationship helps to maintain a consistent interface and allows a measure of context-sensitivity, while eliminating the redundancy that would be caused if both objects displayed separate, full sets of menu options.

Window System Independence: The Graphics Layer

The Andrew Toolkit has been designed to be window system independent (and to a great extent operating system independent). It currently runs under both the original ITC/Andrew Window System and under X.11, though it could be ported to other window systems. We chose to make the Andrew Toolkit window system independent because we want to support a consistent set of applications over a diverse set of window systems, ranging from workstations to Macintoshes and PCs. Clearly the personal computer versions of Andrew Toolkit applications would be more limited, but we consider it to be a great advantage for a user to be able to use a low-cost machine for simple tasks, and then easily move to the more powerful machines when required.

As noted above, the view tree provides a general mechanism for handling of events in the Andrew Toolkit. It hides from the developer the specifics of the input model used by the underlying window system. In a similar fashion, the Toolkit uses the Graphics Layer to hide the output model of the specific display medium. The display medium is usually the underlying window system, but can also be a printer.

The Graphics Layer is built using a third type of object, the *drawable*. Each view contains a pointer to a drawable, which contains drawing commands for a particular imaging device and is used for all drawing operations. Separating the view and the drawable will allow us to provide a simple default printing mechanism. When a view receives a print request for a specific type of printer it can temporarily shift its pointer to a drawable for that printer type and do a redraw of its image. We expect to provide this facility in a later version of the Toolkit.

Summary

The Andrew Toolkit provides for the development and use of components, called objects, which can be used separately or combined to produce more complex objects. The Toolkit provides the ability to embed these components within other components in multi-media applications such as the EZ editor. The architecture for embedding components has been designed to strongly encourage programmers to build new components that can be used in new and existing applications. The architecture also encourages programmers to develop objects that can include arbitrary components instead of specific ones.

The separation of information into data objects and views provides a highly modularized structure that supports this building block paradigm. Users are free to work with data objects in ways different than those envisioned by the data object's creator. New views on existing data objects can be created. Existing data objects can also be used as the building blocks for more complex objects.

The Andrew Class System (Class) is an essential element in supporting the Toolkit's building block paradigm. The object-oriented nature of Class allows programmers to easily develop new, specialized objects out of existing objects, using components such as the C-language component for the multi-media editor. The dynamic loading capability of Class allows the Toolkit to be easily extended by a large community of developers without the need to modify or recompile existing objects.

The Toolkit's view architecture provides a higher level of control and consistency across applications than systems which reply only on physical relationships between components. The view tree supports a series of parent-child relationships between views which simplify the implementation of new views by providing a uniform means of communication via the view's parent. Further, the parent-child relationship between objects allows a parent object to pass on certain characteristics to the child or to assume a reasonable measure of control over input events and the display of the child object.

Extendability is another major feature of the Andrew Toolkit. The Toolkit has been designed so that the creator of a data object or view does not have to take any special action to allow that object to be embedded in another object. The data object and view interfaces in the Andrew Class System (Class) have been designed to provide the necessary and sufficient set of methods for two objects to communicate without detailed knowledge of each other. Further, those interfaces make it easy for the author of an object to allow it to include other objects.

Finally, the Toolkit is unique among other toolkits in its potential to provide a common base of applications across a diverse set of machines and window systems. We have spent considerable effort to make the Toolkit window system independent and believe that it will be important in the future to support the same software base on many systems.

Installation Instructions for the Andrew Toolkit

This section explains how to compile and install the Andrew Toolkit (ATK). Following it is a short description on what is needed to start up some of the applications. Detailed documentation on using ATK can be found in /contrib/andrew/Tutorial.ps.Z and /contrib/andrew/Reference.ps.Z, which are compressed postscript files of complete manuals (pages are in reverse order, for those who want to change the order of printing). More information (including simple testing scripts) will be made available soon.

During the build process, the ATK makefile constructs applications, dynamically loadable objects, libraries, fonts, templates, console descriptions, help files and include files. Each of these collections goes into its own directory. During runtime, ATK applications and objects assume that there will be an environment variable called ANDREWDIR that will be the root of all andrew related run-time files. Within the makefiles, the variable DESTDIR is used for the value you expect to use for ANDREWDIR during run time. In particular, the system should install files so that the following directories are used for each category:

${ANDREWDIR}bin - Runnable applications

${ANDREWDIR}dlib/be2 - Dynamic objects

${ANDREWDIR}X11fonts -ATK specific fonts for X

${ANDREWDIR}lib/templates - Style templates for text-oriented objects

${ANDREWDIR}lib/consoles - Console descriptions

${ANDREWDIR}lib/be2 - Compile time libraries

${ANDREWDIR}include/be2 - Compile time include files

${ANDREWDIR}help - Help files

${ANDREWDIR}doc - Doc files

The following applications are installed into ${ANDREWDIR}bin

ez - the multi-media editor

typescript - a fancy view on the unix shell

help - a program for reading help files for the system and unix man pages

preview - previews output files from ditroff.

console - monitors system status

ezprint - prints formatted files (same as reading the file into ez and then selecting the print options). Other insets that are available include:

table - a simple spreadsheet and table editor eq - equations

zip - hierarchical drawings

raster - raster displayer, minimal editing functions

fad - frame animation editor

ctext - a inset for editing C files.

tm - like typescript but also allows termcap style editing.

More extended information about running these programs will be made available soon. It is also available from the help system.

(Note: as of this writing, the console program will try to reset ANDREWDIR to be /usr/andrew because of its need to perform "setuid". As a result, console may not run correctly, or may appear to have very limited functions. We are fixing this problem, but we may not finish in time for the next release. Similarly, text oriented objects may attempt to use /usr/andrew as the basis for finding templates. As an interim measure, you can place a symbolic link from /usr/andrew/ to ${ANDREWDIR}.)

The source directory for the ATK contains two main subdirectories: overhead and be2. It also contains a top level Makefile that builds the entire toolkit. The build procedure has been designed to allow the entire toolkit to be built with changes made only to the top level Makefile. In the process of building the toolkit on a variety of machines, we have discovered a number of problems that will have to be fixed in order for you to build the toolkit on your machine. We intend to fix these problems on the version that is provided on the X.V11R2 tape. The remainder of this section will describe the build process.

Prerequisites

The toolkit is built using a class preprocessor that generates a large number of defines that are resolved by the C preprocessor. We have been using the 4.3 version of the C preprocessor that allows a large number of defines. Earlier versions of the C preprocessor use a static table. When the earlier version runs out of space it reports the error "too much defining". Unfortunately there is no easy solution to this problem. Either you should be using the 4.3 version of the C preprocessor or have the static tables in the earlier C preprocessor increased. The values that need to be changed in the preprocessor (cpp.c) are symsiz (to 7500) and sbfsize (to 125*4096).

Modifications to the Top Level Makefile

The top level Makefile contains a number of defines that you may want to or need to change. These changes should be made after applying the patches described below.

DESTDIR - the location where you want to install the andrew software. The default location is into /usr/andrew/. This is its preferred location (certain programs still will not work if the software is installed in other places - these will be described later). If you choose a different directory you should make sure you include the trailing / at the end of the pathname.

XDIR - the location of your X.11 include and library files.

DEBUG - whether you want to use -g or not. For the time being do not set this flag to -O. It is being used for both the C compiler and the Class preprocessor. We plan to fix this problem soon.

WINDOWSYSTEMS - This controls which window systems are to be included in the base program. It is currently set up for supporting both X11 and the original ITC window system. You should be able to change this definition to be just -DX (We have not tried this at this time).

FCDIR - the directory containing the X11 font compiler. As of the last X11 release the font compiler was not installed anywhere. This will disappear when the font compiler is installed when building X. You will need to set this as a Makefile will copy the file into the andrew build tree for later use.

FC - The location of the X11 font compiler. You will only need to set this variable if the font compiler is not located in ${FCDIR}/fc.

Building the System

Once you have made the appropriate changes to the top-level Makefile and applied the fixes noted below you should type make in the source directory and wait (it take 4 hours on a RT PC). This will compile and install the overhead files and then compile the be2 files. After this has completed you should then type "make install", which will install the be2 files.

Finally you need to install the help files and documentation. The previous procedure has placed the file help.input.common into ${DESTDIR}/lib. This file contains a list of the directories that should be made available by the help system. The names in the files refer to /usr/andrew. If you have chosen to install the andrew system in a different location you should edit that file replacing all occurrences of /usr/andrew with the value you chose for DESTDIR. After making that change you can then execute "make ibmdoc". More information about customizing the help system can be found in the file ${DESTDIR}/doc/help/helpmaint.guide. You should wait until you have the system running and then read the file using ez.

Once you have completed the compilation and installation of the overhead directory you can compile just the be2 directory by using "make be2" from the source directory. This will just do the compilation of the be2 tree. This should be used if you encounter any problems compiling the be2 directories. Once you have completed building be2 in this fashion you should execute "touch compile.timestamp". You can then do the "make install" and the "make ibmdoc".

The original distribution (released on 2/3/88) contains some files that should not be present because they are generated automatically or are irrelevant to the working system. To prevent problems during the make process, these files should be removed. (NOTE: If you pick up the Toolkit after 2/15/88, these files may already have been removed. Watch for official posts on andrew-toolkit-info for updates and details.) Specifically, the files that should be removed before building the system are:

1. overhead/fonts/fontfiles/programs/banzai16.fdb

2. overhead/fonts/fontfiles/programs/life1.fdb

3 .overhead/libmail/y.tab.c

4. be2/table/table12.snf

5. be2/eq/parse.h

6. be2/hz/makehelpindex

7. ibmdoc/vdoc (the entire directory)

8. overhead/vfile (the entire directory)

9. be2/basics/offscreenwindow.[cH]

10. be2/basics/xoffscreenwindow.[cH]

11. be2/basics/wmoffscreenwindow.[cH]

12. be2/console/diskmonitorfreq.c

13. be2/console/errormonitorfreq.c

14. be2/console/mailmonitorfreq.c

15. be2/console/venusmonitorfreq.c

16. be2/console/vmmonitorfreq.c

17. be2/console/diskmonitor.c

18. be2/console/errormonitor.c

19. be2/console/mailmonitor.c

20. be2/console/venusmonitor.c

21. be2/console/vmmonitor.c Some files in the original distribution are outdated and contain bugs that will appear during compilation or execution. These can be fixed by using the patches that are uploaded in the file /contrib/andrew/PATCHES1-36.tar.Z with the patch program distributed on the X.V11R1 tape. Initially, there are 34 patches (numbers 13 and 14 have been deleted) which are described below. As we accumulate more patches, we will try to post them onto zap. Each patch file (which is numbered) is described below:

1. Makefile

Changed definition of location of FC to depend on FCDIR.

Changed definition of IBMDOCDIR to be ./

2. overhead/Makefile

The RINSECOMMAND does not pass down the location of XDIR. Needs to build cmenu and cwm during the compile phase. Remove references to vfile.

3. overhead/cmenu/Makefile

Need to include line -I${XDIR}/include in INCLUDEPATH

4. overhead/cwm/Makefile

Need to include line -I${XDIR}/include in INCLUDEPATH

Need to pass XDIR down to the make of stubs

5. overhead/cwm/stubs/Makefile

Need to include line -I${XDIR}/include in INCLUDEPATH

CC line should be CC=cc not CC=hc

6. overhead/fonts/fdbbdf.c

Definition of MAX must be on own line.

Printing of header information needs to be done upon reaching first character.

7. overhead/fonts/fontfiles/Makefile

Proper location of X font compiler

8. overhead/fonts/fontfiles/adobe/Makefile

Proper location of X font compiler

9. overhead/fonts/fontfiles/programs/Makefile

Proper location of X font compiler

10. overhead/libmail/Makefile

Add in clean of y.tab.c

11. overhead/libmail/valhost.c

Proper handling of h_addr

12. overhead/libwmdir/Makefile

Remove multiple defintions of INSTALL and FINALDIR

Fix defintion of INSTALL

(13 and 14 have been deleted)

15. be2/basics/Makefile

Fix handling of making bogus wmws.do file.

Add in missing libraries in making of xws.do

16. be2/console/mailmonfreq.c

Fix indentation for #ifdef VICE

17. be2/console/vmmon.c

Proper handling of defining MAXUPRC

18. be2/console/vmmonfreq.c

Proper handling of defining MAXUPRC

19. be2/raster/Makefile

Remove multiple defines in Makefile

Setting of XDIR

20. **be2/table/Makefile**

    Fix cleaning of X11FONTS

21. **be2/zip/Makefile**

    Removed extra blank lines

22. **be2/eq/Makefile**

    Added cleaning of parse.h

23. **be2/hz/Makefile**

    Added cleaning of makehelpindex

    Add installation of setuphelp setuphelpmenus.

24. **be2/raster/libraster/Makefile**

    Fix defintions of SRCDIR and INSTALL

25. **be2/basics/im.H**

    New handling of redrawing windows

    Replace use of original vfile code.

26. **be2/basics/im.c**

    New handling of redrawing windows

    Remove reference to mrl code

    Replace use of original vfile code.

27. **be2/basics/wmim.H**

    New handling of redrawing windows

28. **be2/basics/wmim.c**

    New handling of redrawing windows

    Fixed posting of cursors.

    Replace use of original vfile code.

29. **be2/basics/xim.H**

    New handling of redrawing windows

30. **be2/basics/xim.c**

    New handling of redrawing windows

    Handle setting of DISPLAY to unix:0

    Fix posting of menus to take proper number of arguments.

Handle errors when posting menus better.

Remove reference to mrl code

Replace use of original vfile code.

31. **be2/basics/xfontdesc.c**
    Handle searching for fonts in a better fashion.

    Setting MaxHeight when its original value is 0.

32. **be2/basics/application.c**
    Remove automatic aggregate initialization

33. **be2/apps/Makefile**
    Remove use of libvfile.a

34. **be2/apps/runapp.c**
    Added code to cleanup any outatnding vfile entries.

35. **be2/apps/staticloads.c**
    Moved mrl references under ifdef WM.

36. **overhead/cmenu/cmenuActivate.c**
    Fixed NULL reference bug.

37. **overhead/util/Makefile**
    Around line 143 (by Makeclean), there is a control M in the file.

38. **be2/Makefile**
    Get rid of blank line and indent fi at end of COMMAND macro.

39. **be2/basics/Makefile**
    Get rid of three references to vfile.h.

40. **be2/basics/wmim.c**
    Get rid of vfile.h.

41. **be2/basics/xim.c**
    Get rid of vfile.h.

    Get rid of _ in im__vfileclose.

42. **be2/console/diskmon.c**
    #ifdef sunv3 changed to #ifdef sun.

    Location of fs.h moves from sys to ufs in Sun 3.5

This initial release contains a number of bugs that have not yet been fixed. We will be working on these bugs for the V11R2 release.

- Console uses ANDREWDIR for finding consoles and forces ANDREWDIR to be /usr/andrew or /usr/andy (L35)

- Templates are hardwired into living in /usr/andrew/lib/templates? (L34)

- Menus fail if Andrew fonts are not available. (L4)

- Scrolling with Overlapped windows doesn't work well (L16) - X11 server problem

- Missing slashes in the Makefile (L17)

- XGetImage/XPutIMage problems (L21) - X11 server problem

- Dancing menus (L23)

- Error numbers instead of names are reported in console when an error occurs (L28)

- Xgraphic_DrawPoint alters the foreground and transfer function without resetting them (L29) and still does not draw a point (L31)

- Handle XDefaults and preferences in a reasonable fashion. (L33)

- Console does not draw circular clocks (L36)

- typescript and tm seem to have problems when default fonts are not available (L41)

- ctext when cut and pasted into text comes in as a ctext object (L44)

- Redrawing windows that were obscurred sometimes misses by one pixel on the left or bottom. (L46)

- Cursors do not appear on the sun (L58)

- ezprint -t does not produce full output on the sun when writing to stdout (L60)

- cursors leave streaks behind them on the Sun (L62)

- multiple slashes on Ultrix 1.2 causes problems (L63)

- rasters mess up when rubber banding (L14)

- Lines flicker in fad (L74)

- No error message posted when doing font substitution (L75)

--Help displays .CKP files (L90)

Setting up Your Environment for Running Applications

Prior to attempting to run the toolkit applications you should do the following operations:

- If you have installed the software somewhere other than /usr/andrew you should set the environment variable ANDREWDIR to that path.

- Add /usr/andrew/bin or ${ANDREWDIR}/bin if you installed the toolkit elsewhere to your path.

- set the environment variable BE2WM to x11.

- if you are going to be using an overlapping window manager you might also want to define the environment variable UsingOverlappingWindowManager. This attempts to get around a bug in the X11 region code that causes regions to be improperly combined.

The applications will look in a file called "preferences" for most defaults. It is not necessary for you to have such a file to start the applications. Preference options can be found by examining the help information. We are in the process of trying to provide a clean use of both our preferences file and .Xdefaults. We expect this to be available soon.

There are two entries that you might need to set in you .Xdefaults file. They are cwm.MenuFreeze and cmenu.MenuFreeze. They have a default value of 1. Because of problems with GetImage and PutImage you might want to set them both to 0. The problems become apparent when using the menus.

The toolkit comes with a tiling window manager called cwm. By default it divides the screen into two columns. Most windows will appear in the right most column. Typescript, xterm, and console are the exceptions. You may shrink windows to only their title bar by clicking in the title bar with the left button. You may move windows by pressing the right button with the title bar and dragging the window to a new location. You can also click the right button and then move the mouse and click again to get the same effect without having to hold the mouse button down. If press the middle button down (both buttons on a two button mouse) in the title bar you will be a set of menus that provide window manager functions. Cwm will by default bring up a typescript window.

The two basic applications are typescript (to eventually be replaced by tm) and ez. The keybindings for these applications are similar to the emacs key bindings. There are also a set of menus available (by using the middle button). The scroll bar has four different regions. The regions at the top and bottom (end zones) allow you to move to the beginning and end of the file by clicking in the regions with the left button. The white area (the elevator) provides an indication of where you are in the file. You can click in that area with the left button and drag the elevator. It will move you to that location. Finally any other left click will move the part of the image that is to the right of the scroll bar to the top of the screen. A right click will move the part of the image that is at the top of the window down next to where you clicked. The scroll bar is used in many applications (both horizontal and vertical and works similarly in all cases).

We suggest that you bring up a typescript and then type help. This will bring up a help window and you can then explore the documentation. We will be making more detailed documentation available in hardcopy form very soon. A copy of the andrew guide can be found in the directory ${DESTDIR}/doc/andrewguide. You may look at this using ez.

Andrew Toolkit Testing Suite

Introduction

This section assumes that you, or your system maintainer, have followed the instructions in the Installation section and believe that the Andrew Toolkit is ready to be tested. The following pages contain setup instructions and a suite of tests for important Andrew Toolkit applications. These test suites assume that you will be running Andrew Toolkit applications on top of X11 and the Cambridge Window Manager (cwm). It is recommended that you follow the setup instructions and testing suites in order. (Note that a test script for EZ is in preparation, but not available at this time.):

SETTING UP YOUR APPLICATIONS ENVIRONMENT

CAMBRIDGE WINDOW MANAGER (CWM)

TYPESCRIPT

CONSOLE

HELP

MESSAGES

CAVEATS

These test scripts are intended to be used following installation of the Andrew System. Each script sets out a number of procedures which you can use to test the basic features of each program. Because this is our first draft, we have a number of caveats:

Testing of the scripts has been limited, so inaccuracies are possible.

Most scripts are quite brief. Additions, corrections, and changes are likely to occur in subsequent releases.

No illustrations could be distributed with this draft, though we do plan to send them in later releases. Occasionally in these drafts, we refer to figures that are not yet available.

Certain test procedures refer to systems which are connected to the Andrew File System. Obviously, if your machine is not using the Andrew File System, you won't be able to use these tests. Use the contingency tests when they are provided for you.

IF YOU HAVE PROBLEMS

If you run into major problems while testing or using an Andrew Toolkit application, send electronic mail to us at one of the the appropriate bulletin boards. Both bulletin boards are monitored by the Andrew Toolkit development staff, so you can be assured of a quick answer.

For open forum questions, comments, problems, and help from Andrew Toolkit programmers across the country, send a message to:

andrew-toolkit-discussion@andrew.cmu.edu

Messages sent to the discussion bulletin board will be automatically remailed to a large nation-wide mailing list similar in nature to the "xpert" mailing list.

For more official inquiries about the Andrew System to the ITC or CMU, send a message to:

andrew-toolkit-info@andrew.cmu.edu

HOW THE TEST SCRIPTS ARE STRUCTURED

All of the test scripts contain the following sections:

OVERVIEW. The initial overview describes basic terminology and concepts at work in the application. It also explains the assumptions made by the writer of the test script.

CONTENTS. This section outlines the parts of the test procedure.

TEST PROCEDURES. Many of the testing procedures begin with some kind of STARTUP section, which mentions critically important pre-test procedures or configuration instructions. Following the STARTUP section, the actual testing procedures begin. Most testing procedures consist of a series of Action-Result units, which actually direct what the tester must do to test the program. These units are written in an imperative style, and contain some or all of the following fields:

ACTION: Step-by-step description of the actions the tester should perform.

RESULT: Describes the state of the system after each action is performed.

CONTINGENCY (optional) : This describes what to do if the test fails. In many cases, momentary failures can be overcome using the contingency directions. If the contingency directions fail or are nonexistent, the tester should report the problem via electronic mail.

KNOWN PROBLEMS (optional) : Covers the cases of bugs and problems that we know about. These are mentioned mainly to tell the tester how the system "should" behave.

EXCEPTIONS (optional) : Annotated list of "things that might be different." These are not bugs, but situations where the result may not exactly match the above description. These are usually stated in the form "if your system is <uncommon situation>, you may experience <differing result>." In most cases, exceptions are caused when the test system is in a non-default configuration. All of our tests assume a default configuration.

--END OF INTRODUCTION--

Setting up Your Applications Environment

OVERVIEW

This section is excerpted from the end of the section "Installation Instructions for the Andrew Toolkit." If you already followed the instructions there for setting up the environment, this will be a review.

SETUP ACTIONS

Before you attempt to run Andrew Toolkit applications, you, or your system maintainer, should perform the following actions, if you did not do so while completing the Installation section:

ACTION 1. If you have installed the Toolkit somewhere other than in /usr/andrew/ you should set the environment variable ANDREWDIR to that path.

ACTION 2. Add /usr/andrew/bin (or add ${ANDREWDIR}/bin if you have installed the Toolkit in a bin subdirectory other than /usr/andrew/) to your searchpath for executable programs.

ACTION 3. Set the environment variable BE2WM to x11.

ACTION 4. Characteristics of the window system are set in the .Xdefaults file. To overcome problems with the X functions GetImage and PutImage which affect the cwm menus, you need to override the defaults for two cwm preferences. These preferences are cwm.MenuFreeze and cmenu.MenuFreeze. They have a default value of 1. You should set them to the value 0.

The Andrew Toolkit applications will look in a file called "preferences" for most defaults. It is not necessary for you to have such a file to start the applications, because they have their own defaults. Preference options can be found by examining the help information. We are in the process of trying to provide a clean use of both our preferences file and .Xdefaults. We expect this to be available soon.

--END OF SETTING UP APPLICATIONS ENVIRONMENT SECTION--

Cambridge Window Manager (CWM) Test Script

OVERVIEW

Cwm is a tiling window manager designed to be compatible with the Andrew user interface. It divides the screen into several (usually two) columns and fills each column with a stack of windows. Each window is provided with a title bar which provides the application name and is associated with a set of standard menus. Cwm resizes columns and windows within columns by dragging the borders between them. Clicking mouse buttons in title bars can hide windows entirely or can shrink them to display only their title bars.

This test script tests the window manager startup and the use of basic features such as scrollbars, movable windows, titlebars, and menus. You will recognize many of these basic features in the other programs you test.

It also has references to the Typescript and Console programs that are tested in some of the following test suites. Because this is a simplified version of the test suite, no illustrations are available. We plan to make them available in subsequent versions of the test materials as soon as we can.

For all the actions in this script, if an action fails to yield the described results, there is little point in going on. If cwm is not working, things are in bad shape. Therefore, the default contigency entries for all the actions, until otherwise stated, is to quit the test, recheck your installation documentation, or seek outside help. Where needed, we reference documents that you might want to use in problem solving.

CONTENTS

This test set consists of the following:

CWM SETUP

STARTING CWM

CHECKING CURSORS

CHECKING TITLE BARS

CHANGING WINDOW BOUNDARIES

MOVING WINDOWS

CHECKING FOR MENUS

USING CWM MENU OPTIONS

CHECKING BUFFERS

SCROLLING WINDOW CONTENTS

CWM SETUP

ACTION. Login to the system.

ACTION. In order to test the cut and paste buffers, two simple test files are required: 'flong' and 'fshort'. The best way to create these would be with the following commands:

```
ls /lib fshort

ls /etc flong
```

which was done for the examples.

NOTE. There are no startup preferences needed for cwm. If you already have a .Xdefaults file, the preferences that you have set in this file will be used, unless you comment them out. If you don't have a .Xdefaults file, cwm will be started with a set of defaults which should be sufficient.

STARTING CWM

ACTION: At the shell prompt, enter the following command:

```
Xinit cwm
```

RESULTS: If your configuration is correct, cwm starts by displaying a "Welcome to Andrew" window. This window should replaced momentarily by a gray screen on which a Typescript window is created.

CONTINGENCY: The command "Xinit cwm" works if the following things are in place:

a) "Display" variable is correctly set, as described in the Xlib man page. To set this, enter the command:

```
setenv DISPLAY :0
```

b) Your server is called "X". (See Xinit man page for details.)

c) Both "Xinit" and "cwm" are on your standard searchpath. (See Xinit man page for details.)

If you see a "Welcome to Andrew" window, but a Typescript fails to appear, you probably have not set the BE2WM environment variable correctly. To set this variable, enter the command:

```
setenv BE2WM x11
```

See the file "Installation Instructions for the Andrew Toolkit" for details about BE2WM.

CHECKING CURSORS

ACTION: Try moving mouse to different regions of the screen.

RESULTS: Cursor should normally appear as an upwardly curved arrow. When in the scroll bar (area on left hand edge of the Typescript window) it should change to a vertical arrow with points on both ends. Within the title bar (the top of the window), it should become a square box. When moved to the area on top, down below, or between windows, it should change to a vertical or horizontal boundary cursor. A boundary cursor appears as two arrows pointing to a vertical or horizontal bar.

CONTINGENCY: If mouse does not move, quit. If cursor does not change, quit.

KNOWN PROBLEMS: On double screen workstations, if input focus is switched before cwm is initialized, cursor may only move when focus is on wrong screen. Sometimes, even with a single screen, a 6152 behaves like a double screen workstation (VGA and 8514). It is necessary to switch to the 8514 by pressing alt-ScrollLock.

CHECKING TITLE BARS

ACTION: Move cursor into the title bar of a window and click the left mouse button.

RESULTS: Window shrinks to the title barm which then moves to the top of the column.

ACTION: With cursor in title bar and window shrunk, click left button again.

RESULTS: Window reappears and moves down to its former place in the column.

CHANGING WINDOW BOUNDARIES

Widening/Narrowing windows:

ACTION: Move cursor to right edge of Typescript window until cursor is changed to vertical boundary cursor. With the boundary cursor showing, press and hold both buttons (or the middle button on a three-button mouse), drag the boundary to a new horizontal position, and release.

RESULTS: Each time you drag the boundary cursor to a new position and release the button, the window should redraw and widen or narrow to the place you moved the cursor.

Shortening/Heightening windows:

ACTION: Move cursor to bottom edge of Typescript window until the horizontal boundary cursor appears. Press and hold both buttons (or the middle button), drag this cursor to new vertical position, and release. The window's height should be redrawn to the new position of the cursor.

RESULTS: Each time you drag the boundary cursor to a new position and release the button, the window should redraw and shorten or heighten to the place you moved the cursor.

MOVING WINDOWS:

ACTION: With cursor in title bar, click right button.

RESULTS: Window disappears and cursor changes to a bullseye.

ACTION: Move cursor to right hand side of screen and click right button.

RESULTS: Window should reappear in new position in right column of the screen. NOTE: After moving windows, changes to window boundaries should still work properly.

CHECKING FOR MENUS:

ACTION: Move cursor to grey area outside the Typescript and press and hold both buttons (or the middle button), slide the cursor vertically and horizontally across menu cards, and release the button(s) outside the menu cards.

RESULTS: Menu cards should appear and change as the cursor slides across their boundaries. Moving the cursor up and down across the menu options should cause a black selected region to temporarily appear on each one. Releasing the mouse button(s) causes the menu cards to disappear.

ACTION: Move cursor to title bar and examine the title bar menus in the same way.

RESULTS: A set of title bar menus should appear and behave as described above.

ACTION: Move cursor to Typescript window and repeat sliding around menu cards.

RESULTS: A set of Typescript menu cards should appear and behave as described above.

USING CWM MENU OPTIONS

In the following section, we use the following convention when referring to menu options. Menus will be in the format:

WINDOW / (Menu Card) Menu Option

WINDOW describes the window where the cursor should be before displaying the menus. The WINDOW can be CWM (gray space outside a window), Titlebar at the top of program window, or a program name, such as Typescript. (Menu Card) describes the name at the top of the menu card that appears when you press and hold both mouse buttons (or the middle button). Menu Option is the option that should be chosen from the (Menu Card) named in the example. Note that the CWM and Titlebar menu cards are the same. The only difference is that the "This Window" menu card is inoperative in the background area. As such, anytime CWM is specified a Titlebar could be used instead.

Zapping Windows:

ACTION: Select CWM / (All Windows) Zap.

RESULTS: Window manager should exit. (We are doing this now to test whether an exit from cwm via the Zap command works. We may need it later in the event of problems.)

CONTINGENCY: If the window manager does not exit, try using 'ps' and appropriate 'kill's in Typescript. If that doesn't work you might try booting, but running this system is not a good idea.

Restarting CWM

ACTION: If window manager exits successfully, do 'ps' to check for left over processes (should be none) and any error messages output to terminal while window manager was active.

ACTION: Restart window manager, using steps in STARTING CWM above. CWM and a Typescript window should start normally.

Adding a New Typescript:

ACTION: Select CWM / (Expose) New Typescript.

RESULTS: A second Typescript appears.

Checking New Typescript:

ACTION: Try horizontal and vertical edge movements, particularly at horizontal edge between Typescripts.

RESULTS: The windows should fill the column.

ACTION: Move Typescript 2 to right column (right click in titlebar, move to right column and right click).

RESULTS: Typescript 2 now in right column.

ACTION: Try moving the horizontal and vertical boundaries.

RESULTS: The movements should work as above.

Using Enlarge:

ACTION: Using vertical repositioning motion move bottom edge of Typescript 2 to mid-screen and then select Titlebar / (This window) Enlarge.

RESULTS: Window should expand to fill entire vertical column.

Using Hide:

ACTION: Select Typescript2 Titlebar / Hide.

RESULTS: Typescript2 should disappear.

Exposing a Hidden Window:

ACTION: Select CWM / (Expose) Typescript2.

RESULTS: Typescript 2 should reappear.

Zapping one window:

ACTION: Select Typescript 2 Titlebar / (This Window) Zap.

RESULT: Typescript 2 should disappear.

CHECKING BUFFERS:

ACTION: Unshrink the title bar by clicking the left mouse button, if needed. Move cursor to Typescript (arrow cursor) and type

```
echo hi there
```

RESULTS: What you type should appear in bold. Output should appear in unbold.

ACTION: Enter 'cat fshort'.

RESULTS: Contents of fshort should be displayed.

ACTION: Enter command 'cat flong'.

RESULTS: Contents of flong should be displayed.

SCROLLING WINDOW CONTENTS: (This section uses the fshort and flong files you created in the SETUP section earlier.)

Scrolling to top of window contents:

ACTION: Position cursor in speckled area at top of scroll bar and left click.

RESULTS: The top line of the Typescript should be the 'cat fshort' command.

Scrolling to bottom of window contents:

ACTION: Left click in bottom speckled scroll bar area.

RESULTS: Should be at bottom of output (i.e., shell prompt).

Moving a line to the top of a window:

ACTION: Put the cursor into the scrollbar, opposite a line of information. Click the left mouse button.

RESULTS: The line opposite the cursor moves to top of window.

Moving the top line to the cursor position:

ACTION: Put the cursor into the scrollbar. Note the contents of the line at the top of the window. Click the right mouse button.

RESULTS: The line that was at the top of the window moves to be opposite the cursor.

Changing the visible portion of a window:

ACTION: Position cursor in the white box in the scroll bar. (The box represents the visible portion of the text). Press left mouse button button, drag the white box up or down, and release.

RESULTS: Cursor changes to a rounded dot and text scrolls accordingly. (Repeat this step several times, moving up or down.

--END OF CWM TEST SCRIPT--

Typescript Test Script

OVERVIEW

Typescript is an Andrew Toolkit interface to command-line based UNIX programs. It allows users to invoke the Andrew Toolkit's text editing functions while interacting with these programs. Typescript is most commonly used as an interface to the csh. In order to test Typescript, you must have a working version of your window manager, and some way of executing the Typescript. It is assumed that the tester knows how to use scrollbars and the basic toolkit editting functions.

We have tried to make this script a representative test of the Typescript program, but it may have some inaccuracies. At present it also lacks illustrations. We look forward to expanding and illustrating this test script in later releases.

CONTENTS

CRITICAL TESTS

Typescript must pass all of these tests if it is to be useful.

Starting Typescript:

ACTION: If you are running the cwm window manager, or CMU's wm, choose New Typescript from the Expose menu card in the Titlebar menus. Otherwise, run the program Typescript using some other shell interface.

RESULT: After a few seconds, a Typescript window should appear, consisting of a title bar on top, a scrollbar on the left side, a large white area for input in the center, a prompt (usually a % sign) and a message line at the bottom.

Executing a Command:

ACTION: Execute the ls command in the Typescript.

RESULT: You should see your command in bold, output in a plain font.

Command Line Editing:

ACTION: Type several command lines. As you type, execute the editing functions such as delete, control-b, control-f and so on.

RESULT: You should be able to edit commands using the toolkit's editing functions.

Menu Item: Copy

ACTION: Select some text in the Typescript. Choose Copy from the first menu card.

RESULT: The text should be copied into the cut buffer. The next test can help to confirm that if other testing means are unavailable (such as a working ez).

Menu Item: Paste

ACTION: Copy some text into the cut buffer. Choose Paste from the first menu card.

RESULT: The text should appear on the current command line.

Scrolling Backwards:

ACTION: Execute enough shell commands so that output has scrolled off the top of the window. Try to scroll back through this output using the scroll bar.

RESULT: You should be able to at least 10k of previous interaction.

Making typed input visible:

ACTION: After scrolling backwards, start to type a new command.

RESULT: The current line should come into view.

Using Other Programs With Typescript

ACTION: Execute the command

```
typescript sh
```

RESULT: A Typescript should appear in which sh is running rather than csh. (After the new Typescript appears, check for the sh process. Then, you may Zap the Typescript if you wish, using Zap from the Titlebar menus.)

NONCRITICAL TESTS

If one of these tests fails, Typescript is probably still quite useful, though a bug should be reported.

History Mechanism:

ACTION: Type Escape = several times.

RESULT: Previous command lines should appear and be selected. Pressing Enter causes the currently-selected command to be executed.

Menu Item: Move

ACTION: Select some text in the Typescript and choose Move from the first menu card.

RESULT: The text should appear on the command line and should be selected.

Menu Item: Execute

ACTION: Select some previous command line and choose Execute from the first menu card.

RESULT: The text should be copied to the command line and then executed.

Menu Item: Clear

ACTION: Choose Clear from the first menu card.

RESULT: Everything except the current command line should be deleted from the Typescript window.

--END OF TYPESCRIPT TEST SCRIPT--

Console Test Script

OVERVIEW

Console is a program which keeps track of various kinds of status information about your workstation and files. It can display a digital or analog clock, monitor various aspects of your system's performance, notify you about the arrival of mail, warn you about system errors, and much more.

For the time being, the figures that are referenced in this draft (figure 1, figure 2, et al.) are not available. We intend to make these figures available in a future version of this test suite. Sorry about the inconvenience.

CONTENTS

STARTING CONSOLE

ACTION: In Typescript, enter the command

```
console
```

RESULT: After a few seconds you should see a message in the Typescript telling you that console is starting to run, the message should include the version number of the program which is needed for any bug reports. Shortly after that a window should appear on the screen similar to figure 1. Along the top of the console is a clock, a series of icons (for printing, mail, and trouble), and a CPU load graph. Along the bottom is an ErrorLog area. (The details of non-default consoles may vary slightly.)

SENDING TEXT TO /DEV/CONSOLE

ACTION: In Typescript, enter the command

```
'echo foobar  /dev/console
```

RESULT: The word "foobar" followed by a timestamp should appear in the ErrorLog, which is in the lower portion of the Console window - see figure 2.

KNOWN PROBLEMS: Large amounts of output being sent to /dev/console in quick succession may not all appear in the log.

CONTINGENCY: If this does not work, check to see if the /etc/services has an entry for console - i.e.: "console   2018/udp." If it does not have this entry, please add it and repeat this test. The ability for console to capture messages of this sort is fairly important to its functionality.

ACTION: Using the left-mouse-button, click on the instruments (clock, printing icon, mail icon, trouble icon, CPU load graph) in the upper half of the console window.

RESULT: A line of text, telling you something about the instrument you clicked on, should appear in the ErrorLog. - see figure 3. (Repeat this ACTION several times to put more output in the ErrorLog for later tests.)

EXCEPTIONS: Clicking on the space labelled "Load" will produce a quick flash across the entire window - there is no information string (LeftClickString) associated with that portion of the console.

ERRORLOG

ACTION: Click the left-mouse-button in the ErrorLog.

RESULT: A text-caret (^) should appear where you clicked the mouse. - see figure 4.

ACTION: Click the right-mouse-button in the ErrorLog at a different location than where the text-caret is located.

RESULT: The text between where the text-caret was, and where you clicked the right-mouse-button should now be selected in an inverse-video. - see figure 5.

ACTION: Move the cursor until it is below the console window (the cursor should change shape to the boundary moving cursor) - holding down either the left, right, or both buttons, drag the bottom of the console window down until the window is approximately twice as big/high

RESULT: You should now see a scrollbar appear at the left-hand side of the ErrorLog - see figure 6.

CONTINGENCY: If you don't see the scrollbar, make the window even bigger - if the window is as tall as the screen and you still don't see a scrollbar - skip to the next unit and send in a bug report.

ACTION: Using the scrollbar, scroll to the beginning of the ErrorLog.

RESULT: see figure 7.

ACTION: Click anywhere (in or out of the ErrorLog) in the console window with the left-mouse-button.

RESULT: Any text in the ErrorLog that was previously selected, should now be unselected.

USING MAIN CONSOLE MENU OPTIONS

ACTION: With the mouse somewhere within the console window, but not in the title-bar, click-and-hold on the middle (both) mouse-button(s).

RESULT: This should bring up the console menus as they appear in figure 8.

Set Alarm:

ACTION: Select 'Set Alarm' from the Console menus.

RESULT: The console window should have been blanked out and replaces with a prompt for the "time" - figure 9.

ACTION: Enter a time (format: ##:##)

RESULT: The first prompt should be replaced with a second prompt for the "event" - figure 10.

ACTION: Hit return for the default event "Wake Up!!"

RESULT: The console window should redraw itself, and you should see a small bell icon to the right of the clock - figure 11.

Turn Off Alarm:

ACTION: Bring up the menus and select 'Turn Off Alarm'.

RESULT: The bell icon should disappear.

Make Clock Digital:

ACTION: Bring up the menus and select 'Make Clock Digital'.

RESULT: The clock face should be replaced by a digital display of the time - figure 12.

Write Log File:

ACTION: Bring up the menus and select 'Write Log File'.

RESULT: The console widow should have been blanked out and replaced with a prompt for the name of a file to write the log out to, with a default of '/tmp/ConsoleLog' - figure 13.

ACTION: Press Enter to use the default.

RESULT: The console window should redraw itself, and a message in the ErrorLog should tell you that it wrote out the log file. (you can examine this file later to make sure that it contains the contents of four logs (0-3) only the first of which, 0, should have any entries.) Clear All Logs:

ACTION: Bring up the menus and select 'Clear All Logs'.

RESULT: The ErrorLog should now be empty - figure 14.

USING THE 'DISPLAY' & 'CONSOLES' MENUS

ACTION: Bring up the menus and move the cursor over to the left until the second menu card, titled 'Display' is shown.

RESULT: figure 15.

ACTION: Move the cursor further to the left until the third menu card, titled 'Consoles' is shown.

RESULT: figure 16.

ACTION: Going back to the Display menu, select 'Expand Menu', wait a couple of seconds, and then bring the menus up again.

RESULT: figure 17.

KNOWN PROBLEMS: Occasionally you have to select this menu option twice in order for it to take effect.

EXCEPTIONS: Depending on how many consoles have been ported and/or stored in the library of consoles, the number of menus, and contents therein may differ.

ACTION: Selecting one of the "named" consoles off of one of these menus (third menu card and up) should RESULT in the starting up of that console in the same window that we have been using, replacing the Default "Monitor" console. - After having tried this, go back to the Display menu card and select 'Restart Default'.

RESULT: The Default console should again occupy the window.

ACTION: Select another random console from the list, after it has initialized, bring up the menus and go to the Display menu card again. Select 'Read New Console File'.

RESULT: The console window should be blanked out and replaced with a prompt for the name of another console file to read.

ACTION: For this script, simply type in "Default" (no '"'s, capitalization counts)

RESULT: The Default console should once again take its place in the window.

ACTION: Bring up the menus once again and select 'Shrink Menu' from the Display menu card.

RESULT: After doing this, if you bring up the menus again, you should only see the original three menu cards.

INSTRUMENT MONITORING

ACTION: Queue a job for printing.

RESULT: Within a few seconds after the job has been queued, the Print icon should become highlighted, and display a number below it indicating the number of jobs you have in the queue - figure 18.

EXCEPTIONS: On systems which are not connected to the Andrew File System and which do not use the Andrew printing programs, this monitor will not work, in which case it is likely that at startup time it would have announced (via the ErrorLog) that it was terminating monitoring of the PrintDir.

ACTION: Send yourself some mail.

RESULT: Within a few seconds after the mail has been delivered, the Mail icon should become highlighted, and display the number of messages waiting to be read below it - figure 19.

ACTION: In Typescript, enter the command

```
unlog
```

RESULT: Within a few seconds, the Trouble icon should light up, and a message should appear in the ErrorLog, in bold text telling you that you have lost authentication and should use log to refresh your tokens - figure 20.

EXCEPTIONS: If your system is not connected to the Andrew File System, you should be able to approximate this behavior by creating extremely large files (cores if you have them) onto the local hard disk until the hard disk becomes over 95% full. - figure 21.

ACTION: In Typescript, enter the command,

```
touch foo
```

RESULT: The right-most icon (the ITC logo) should become inverse-highlighted for a brief moment, indicating that an Andrew File System store occurred. - see figure 22.

EXCEPTIONS: If your system is not connected to the Andrew File System, this will not happen.

--END OF CONSOLE TEST SCRIPT--

Help Test Script

OVERVIEW

The Help program provides a user interface to the Andrew Help System, an extensive set of help documents for Andrew System applications as well as man pages for UNIX operating system utilities. Help's interface consists of a menu-driven, multi-paneled interface in which read-only versions of documents are shown in the window. Help's keyword index, a critical part of the program, provides quick access to its files.

CONTENTS

HELP STARTUP

ACTION. In Typescript, type "help" at the prompt and press Enter. Or, move the cursor into the title bar of any window, display the menus, and choose the "Help" menu option.

RESPONSE. The Help window should appear, containing the text called "A Guided Tour of Andrew." If you started help from the title bar of a window, the Help window should contain the help file for the program running in that window. In the right column, two menu lists should appear--the top one titled "Overviews" and the bottom one titled "Programs." Each menu list should contain a list of alphabetical entries.

FETCHING FILES WITH KEYWORDS

All Help files are indexed and retrieved with keywords. Keywords are the names of help files or terms associated with a particular help file. The following action-result units let you test various methods of retrieving files with keywords in the Help system.

From the Programs list:

ACTION. Move the mouse cursor over the keyword "console" in the Programs list at the lower right of the Help window. Press the left mouse button.

RESPONSE. The help file for "Console: A System Monitor" should appear in the Help window.

From the Overviews list:

ACTION. Move the mouse cursor over the keyword "Managing Files and Directories" in the Programs list at the top right of the Help window. Press the left mouse button.

RESPONSE. The Overview file called "Managing Files and Directories" should appear in the Help window.

By selecting a keyword from the text:

ACTION. Scroll down in the text for managing files and directories until you see the term "cp". Then, select "cp" and choose "Get Help on Selected Word" from the Help menu.

RESPONSE. The help file "cp.help" should appear in the Help window.

Seeing Unix documentation associated with a keyword:

ACTION. When the cp.help text is shown in the help window, the string (more) should appear in the title bar, indicating that there is additional documentation (probably man pages) that begins with the string "cp". Expose the menus, and, on the top card, you should see an extra menu option called "Show More UNIX Documentation." Choose this menu option.

RESPONSE. An additional help file about cp (probably the cp.1 man page) should appear. (The correct operation of the "Show more UNIX documentation feature depends on whether there is more than one piece of online documentation that begins with the same string. Whenever there is more than one document beginning with the same first word of the filename, the (more) and the "Show More UNIX Documentation" menu options should appear.)

By typing in a keyword:

ACTION. Display the menus in the Help window and choose "Show Help On . . ."

RESPONSE. The following prompt appears in the message line at the bottom of the Help window:

Enter a keyword about which you want more help:

ACTION. Type the word "tour" and press Enter.

RESPONSE. The help file "A Guided Tour to Andrew" should reappear in the Help window.

WORKING WITH HELP PANELS

Expanding the Programs List:

ACTION: Choose "Expand Programs List" from the Help menus.

RESPONSE: After a momentary pause, the list of keywords shown in the Programs panel lengthens to include the names of all help files and man pages available in the help system for that machine. This list is labeled "Full program list." Clicking the left mouse button when the cursor is on any of these names retrieves the file associated with the name.

Shrinking the Programs List:

ACTION: When the Programs List is expanded (the "Full program list" is shown), the "Shrink Programs List" menu option should appear in place of "Expand Programs List." To shrink the Programs List to its original size, choose "Shrink Programs List" from the menus.

RESULT: The original, shortened "Programs" list returns.

Checking the Help History Panel:

ACTION: Choose "Show History Panel" from the Help menus.

RESULT: The Programs panel in the lower right of the Help window is replaced by a panel labeled "History," which contains a listing of the help files that have already been consulted during the session. The history list lengthens with each new file that is consulted.

ACTION: When the History Panel is exposed, the menu option "Show Programs List" should appear in place of "Show History Panel." To bring back the Programs list, choose "Show Programs List" from the menus.

RESULT: The Programs List should reappear in place of the History panel.

Hiding Help Panels:

ACTION: Choose "Hide Help Panels" from the Help menus.

RESULT: All panels at the right side of the Help window are hidden.

Restoring Help Panels

ACTION: When the Help panels are hidden, the menu option "Show Help Panels" appears in place of "Hide Help Panels." Choose "Show Help Panels."

RESPONSE: The Help panels reappear at the right side of the Help window.

SELECTING TEXT, SCROLLING TEXT, SEARCHING IN TEXT, PRINTING TEXT

All these features should work the same in Help as in other Andrew Toolkit programs.

After a word is selected, the menu option "Copy" should appear in the Help window. (The menu option "Cut" does not appear because Help is a read-only window.)

Scrolling should work in the manner described in the CWM testing document.

Searching and Printing should work as described in the forthcoming EZ testing document.

--END OF HELP TEST SCRIPT--

Messages 5.xx Test Script

OVERVIEW

Messages is a program for reading mail and bulletin boards. It is designed specifically to run on Andrew workstations.

This script is an attempt to take you through the various procedures of subscribing to, reading, and sending messages. All preferences and/or Xdefaults will be assumed to be the default. We have tried to make this script a representative test of the Messages program, but it may have some inaccuracies. At the present time, it also lacks figures and illustrations, which are occasionally referenced in the script. We look forward to expanding and illustrating the script in later releases.

Within this text, a number of terms are used. "Messages Window" refers to the entire program window. This will also be referred to as the "program window". "Send Message Window" and "Composition Window" are synonymous and relate to either the separate program Sendmessage or the sub-process of sending messages from within the Messages program. "Captions_Region", "Folder_Region" and "Body_Region" all refer to subdivisions within the "Messages Window."

CONTENTS

SETUP FOR TESTING

ACTION: If you have already used Messages, or if you tend to get a fair amount of Email during the day - you should try to find and use a test account for this exercise. If you can not get access to a test account, you will want to follow the steps below to try and minimize any differences from the results listed. (NOTE - If you have never used Messages, you have no need for these steps.):

```
mv .AMS.prof .AMS.prof.good

mv preferences preferences.good

mv Mailbox Mailbox.good

mv .MESSAGES .MESSAGES.good
```

RESULT: Your are now configured to test Messages as if you had never used it before.

STARTING THE TEST

ACTION: To begin, type "messages" at the Typescript prompt and press Enter.

RESULT: You should get a message in your Typescript saying:

Starting Message (version x.xx), please wait...

You may also see several Console messages indicating that Messages is creating files and/or directories that it needs to begin working. (After Messages is run for the first time, these files and directories will remain in place for future use.)

Then you should get a Messages Window which will be split into three parts (Folder_Region [top], Captions_Region [middle] & Body_Region [bottom]) - see figure 1.

EXCEPTIONS: The "caption" shown in the above referenced figure says "Nathaniel Borenstein", despite the fact that, as you will see later, the message's "From:" line reads "Site.Administrator". On a system connected to the Andrew File System which uses the AMS Delivery System subcomponent, the name "Nathaniel Borenstein" is derived from the authenticated ID of the person who owns the file /usr/andrew/lib/WelcomeMail. (Actually, this file name might be changed, using the AndrewSetup file, but that is the default location for the welcome mail.) Thus, at a site where the Andrew File System is used, it is recommended that this file be owned by the user "postman". At a non-AFS site, the "Site.Administrator" name will be used in the caption automatically.

SUBSCRIBING TO A MESSAGE FOLDER

ACTION: Move the cursor over to the Folder_Region and left-click on the help-icon to the left of a folder-name (mail).

RESULT: A dialog box "What do you want to do with 'mail'?" should appear, containing the choices as shown in figure 2. If you display the menus at this point, they should contain the same choices that are in the dialog box. -- see figure 3.

ACTION: Choose the option to 'Alter susbcription status' by clicking in the dialog box or choosing the menu option.

RESULT: Another dialog box, "Subscribe to Mail?" appears, containing a set of applicable choices -- see figure 4. By default, your current subscription status "Yes," is selected, because you should be automatically subscribed to Mail.

ACTION: Since you want to remain subscribed to mail, left-click again on "Yes" to close the dialogue box.

RESPONSE: The dialogue box disappears.

ACTION: Left-click again on the help-icon next to mail, and this time choose the option 'Explain what it is'.

RESULT: The contents of the Body_Region should be replaced with something like:

Folder name: mail

Folder type: Personal mail

Number of messages: 1

Your subscription status: Subscribed

No explanation of this folder is available, but here is the first message:

Below all of these lines, you will see some data that may look like garbage.

EXCEPTIONS: The last part might contain 'Explanation of this message folder:' if there were an introductory text associated with that folder, as there is with many of the bboards on the Andrew System at CMU.

READING A MESSAGE

ACTION: Click on the message caption 'Welcome to Andrew'.

RESULT: The caption you clicked on will become bold-text, and the contents of that message should now appear in the Body_Region. It should look something like figure 5.

OPTIONAL ACTION: Read the message; it will explain some things about the program and the Andrew System in general. To scroll though the message use the scroll-bar to the left of the Body_Region.

USING MENU OPTIONS

ACTION: Choose 'Append To File' from the menus.

RESULT: You will be prompted in the message-line for the name of the file to append the currently displayed message into. (After checking for the prompt, get rid of it by typing Ctrl-G or pressing Enter to accept the default.)

ACTION: Choose 'Forward To' from the menus.

RESULT: In a few seconds another window will come up (generally causing a redisplay in the original Messages Window as it is sharing the column with it). If expanded it should look something like figure 6 (the screen image includes a formatting bug in the current version at CMU).

ACTION: Choose 'Resend To' from the menus.

RESULT: You will be prompted in the message-line of the Messages Window for an address. (After checking for the prompt, get rid of it by typing Ctrl-G.)

SENDING AND DELIVERY

ACTION: In the Composition Window click on the Reset box in the upper right corner of the Messages window.

RESULT: This will clear the Composition Window for the next step.

ACTION: Type your name or userid in the To: field, enter "test" in the Subject field, and press Return two times.

RESULT: The cursor should bypass the CC: field and move into the body of the Messages Window.

ACTION: Enter some text in the body of the window, and then choose "Send/Post" from the menus.

RESULT: In a minute or two you should see that you have new mail waiting for you (indicated in console).

ACTION: Use the 'Read Mail' menu option from the first menu card in the Messages Window.

RESULT: The caption for the message currently on display should become un-bolded, the Captions_Region should clear and then get filled back in again. The caption for the message you just sent to yourself should appear as the next new message in the Captions Region.

CONTINGENCY: If this does not work, there is probably a configuration problem -- that is, something wrong in the AndrewSetup file. Consult the Andrew Message System Installation and Operation Manual for further information (in the works).

MISCELLANY

Messages and Sendmessage have several dozen other menus and options. They cannot possibly all be described here, nor can they all be tested for every release of the system. However, it is probably a good idea, at this stage in the release testing, to simply browse through the menu items checking out as many of them as time permits. They should all work as documented in the help files.

CLEANING UP

If you previously had used Messages, and had to use the SETUP instructions to move files out of the way to perform this testing, use the following steps to restore those files:

ACTION: Use the following steps:

```
mv .AMS.prof.good .AMS.prof

rm -rf .MESSAGES

mv .MESSAGES.good .MESSAGES

mv preferences.good preferences

mv Mailbox Mailbox.test

mv Mailbox.good Mailbox

mv Mailbox.test/* Mailbox

rmdir Mailbox.test
```

(Note that the Mailbox situation is slightly more complicated than the others, in order to avoid losing mail that was delivered while you were testing.)

--END OF MESSAGES TESTING SCRIPT-- --END OF ANDREW TOOLKIT TEST SUITE--

A Programmer's View of the Andrew Toolkit

This section is composed of selections from the *Programmer's Guide to the Andrew Toolkit*, Volumes I and II. Since this section contains excerpts, not the complete documentation, it is not intended to be a complete instructional text. Instead, it is intended only to illustrate some aspects of Andrew Toolkit programming.

To order a complete copy of the Programmer's Guide to the Andrew Toolkit, use the Documentation Order Form at the beginning of this document.

Sections in this Part

This section contains the following parts:

The first section, **Object Oriented Programming**, is an excerpt from the Tutorial Manual, which is Volume I of the *Programmer's Guide to the Andrew Toolkit*.

The second section, **Examples**, contains a series of examples also excerpted from the Tutorial Manual. Many examples and portions of examples have been edited or omitted to save length in this draft, though the first page of this section contains a complete list of the examples available in the full-length Tutorial Volume. Of the 17 examples in the complete Tutorial Volume, the five we have included here are:

Example 1: Introduction to the classes, program structure and compilation techniques needed to create a stand-alone object.

Example 5: Creating menus

Example 11: Storing and manipulating data

Example 12: Multiple views on a single data object

Example 17: Creating a view that has children

The third section, **View**, is an excerpt from the chapter on the class View, taken from the Reference Manual, which is Volume II of the *Programmer's Guide to the Andrew Toolkit*. The complete Reference Volume contains a full listing of other Andrew Toolkit classes and their associated procedures and methods.

The fourth section, **Selected Interfaces to the Toolkit**, contains a quick-reference list of selected interfaces for the Andrew Toolkit. This list was prepared for the X Window System Conference at MIT in January 1988.

Object-oriented Programming

Objects, Classes, and Instances

The Andrew Toolkit is written within an *object-oriented programming environment.* An *object* in the Andrew Toolkit is simply a structure capable of representing the following, fundamental capabilities of a computer: storing data and carrying out operations on data.

In an object-oriented system, it is useful to group all the objects that share the same data structures or procedures that manipulate the data. Such a group of objects is called a *class*. Objects that are in a class are *instances* of the class.

Suppose an object L1 represents the list *(a b c)* and object L2, the list *(d e f)*. Both can be usefully grouped as instances of a class *list*. Even though each object has different data, they share the same data structure as well as the procedures for operating on their data.

Data, Methods and Class Procedures

Each object has a set of *data, methods*, and *class procedures.*

There are a small number of operations that are generic to all objects within the system. First, there must be operation for creating and destroying an instance of an object. Second, an object must provide an initialization routine to be used to handle subclassing of objects. If B is a subclass of A then when an instance of B is created the initialization procedure for A must be called. These are usually defined as *class procedures* since they are operations carried out on the class only.

Along with the set of general class procedures that each class must provide, a class may provide a set of specific procedures, called *methods*, which are operations that an object carries out on its data. For example, a class for text might have methods for inserting a string, deleting characters, and returning the character at a given position. The *data* for a class is a collection of one or more variables, which may be structured for convenient processing. For the text class, such data might include a variable for character strings, a variable for document length, and another variable for indicating if the text is Read Only or not. The data and methods for a class my be either explicitly written in its class definition, or inherited from its ancestor class or classes (see **Subclasses and superclasses** below). Class procedures, however, are not inherited, and operate only on the class for which they were defined.

Subclasses and Superclasses

Classes can have subclasses and superclasses. When you create a subclass of a class, you do not need to re-specify the data and methods that the subclass shares with its parent class, or superclass. Unless you specify otherwise, via a set of *overrides* in the class declaration, the subclass simply inherits the data and methods from the superclass. Classes do not inherit class procedures since those are specific to the class. In addition to the inherited data structures and operations, a subclass can add new data structures and new methods. It can also override methods that it would otherwise inherit from the superclass.

The relationship between classes go beyond a single parent-child link. A class inherits not only all the data and methods from its immediate parent, but also the data and methods of the parent's parent, and so on through the complete hierarchy of ancestor classes. Of course, a descendant class would not inherit methods that have been overridden by an ancestor.

Requests

An object will carry out one of its operations when another object *requests* it to do so, *i.e.*, when another object invokes one of its methods or class procedures. Each object maintains a list of the set of requests that it has methods or class procedures for carrying out. The requests that any given object understands depend, of course, on what the object is intended to represent. If an object represents a list, then the object should understand requests to add another element to the front of a list, delete an element from a list, etc; if an object represents a text-editor, then the object should understand requests to display text on the screen, respond to a user's mouse click, etc.

When another object issues a request to an object, the methods in the object's class are searched. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found or until the root of the class hierarchy is reached. If the method is not found by that time, an error is reported to the programmer at load time.

The Andrew Toolkit's stratagem for finding methods insures that the most "recent" version of the method code is executed. For example, class A defines a method named *foo*, and class A has a subclass named B, and B has a subclass named C. Ordinarily, if C calls method *foo*, then the Andrew Toolkit searches first in class C, then in class B, then finally in class A, and executes the code for *foo* as defined in class A. But suppose class B has overridden the method *foo* it has inherited from class A. Now, if C calls method *foo*, the code as defined in the override in class B is executed, *not* the original code for *foo* in class A.

Overview of Andrew Toolkit Programming

Programming in any object-oriented language consists of defining objects, analyzing the similarities and differences among the objects and using the analysis to build a classification system, and deciding how the objects will interact. These activities usually result in creating new classes, creating instances of classes, defining new methods and procedures, and specifying sequences of requests among the objects.

Application programmers can design and implement new classes whose objects will eventually appear inside a multi-media editor or another application program. Application programmers must follow a set of guidelines when developing a class. If a programmer follows these guidelines, then instances of the class can be included in any other application program or multi-media editor that has been developed to manage arbitrary objects. An object can be placed inside another object without either one having specific information about the other. The only piece of information that the enclosing object must have about the enclosed object is its name.

Using only the general procedures we can build a mechanism for embedding one object within another and create hierarchies of objects. Consider a document that contains both some text, a line drawing and an equation. The interactions between the text objects, the line drawing object, and the equation object are all done via the general procedures. Thus, you can write your own support packages to work with the general procedures instead of relying on or waiting for such packages to be written by the Andrew system developers.

Examples

This section introduces the essential Andrew Toolkit classes through several example programs. These examples illustrate the program structure needed by any application program that uses the Andrew Toolkit.

Example 1 introduces the classes, program structure and compilation techniques needed to create a stand-alone object, i.e., an object that will be running as an independent application program.

Example 2 shows how to create an object that responds to mouse hits.

Example 3 describes how to make the same object respond to mouse drags.

Example 4 introduces the methods needed to set key bindings.

Example 5 shows how to create menu items.

Example 6 adds a scroll bar to the display.

Example 7 illustrates a way to package an object so that it can be used easily as an independent application or by another object.

Example 8 shows how to display a message in the message line.

Example 9 shows how to use the message line to ask the user a question.

Example 10 describes how to create a view with an associated data object.

Example 11 shows how to store and manipulate data.

Example 12 illustrates multiple views on one data object.

Example 13 shows how to work with simple fonts.

Example 14 describes how to use multiple-font, styled text.

Example 15 shows how to work with styles in documents.

Example 16 introduces the layout package.

Example 17 shows how to create a view that has children.

Example 18 illustrates the dynamic loading mechanism.

Example 1: Drawing 'hello world' in a Window

This section describes how to write a program that defines and uses a very simple class. The program will create a new window and put an object that is an instance of the class in the window. Then the object will draw the string *hello world* in the center of the window. This example program illustrates the following activities:

-- defining a class

-- defining class procedures and methods

-- exporting class procedures and methods

-- importing Andrew Toolkit class procedures and methods

-- setting up a stand-alone application program

-- compiling a class for static linking, i.e., linking before run-time

After reading Example1, you will know the very basic program structure needed to create a stand-alone application program. Example 2 will introduce the changes you need to make in order to build an object that responds to user input, in particular, mouse hits.

The discussion that follows presents a step-by-step description of how to write the example program. If you were to follow the steps, you would produce a program, called *hw*, in four files:

-- a helloworld.H file -- will contain the class definition for the helloworld object, which will be a subclass of the Andrew Toolkit class *view*.

-- a helloworld.c file -- will contain statements that import Andrew Toolkit classes and define the object's methods and class procedures.

-- hw.c -- will contain declarations needed by Andrew Toolkit linking and loading facilities as well as statements that create an instance of *helloworld*, create a window, put the object in the window, and enter an interaction loop.

-- Makefile -- will contain the directions for compiling, linking and loading.

For a complete listing of these files, see *Program Listing for Example 1* at the end of this section. On a first reading of this section, you may find it useful to just skim the program listing, then refer to the listing when needed as you study how to build the program. The source code is available in the directory /usr/andrew/doc/be2/ex1, together with the compiled program.

Although the discussion of the steps refers directly to this example, the information generally applies to the creation of any class that will be used in a stand-alone application.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program on your workstation.

Action 1. To run the program, at the **Typescript** prompt type

/usr/andrew/examples/be2/ex1/hw and press the Enter key.

Response. The program will produce a window with *hello world* centered in the body.

Action 2. Re-shape the program's window.

Response. The object will respond to an update request and redraw *hello world* in the center.

Action 3. Click with a mouse button.

Response. The program will make no response. This simple object does not respond to keyboard or mouse input and it has no menus. Later examples will illustrate how to extend the program so that the view responds to user input.

Action 4. To quit the program, move the cursor to the window title bar, pop-up the menus and choose **Zap** from the *This Window* card.

Response. The program will disappear from the screen.

Creating a class

Deciding on a name for the class

The first step in creating a class is to choose a name for the class. The name is important for two reasons. First, if a user wants to include your object in a multi-media editor, the user will type the class name in order to obtain an instance of it. The name you pick should be unique and not conflict with already existing ones. To see whether the name you pick is on your dynamic object path, in the **Typescript**, type

whichdo *your class name*

The command *whichdo* (which dynamic object) is analogous to the *Berkeley Unix* command *which* (see **help which** for more information).

The second reason the name of your class is important is that the class procedures and methods for the class will all be located in a *.c* file. When the class is compiled, it is easiest if the module name for this *.c* file is the same as your class name. For example, the class procedures and methods for an view named *piechart* should all be located in a file named *piechart.c*.

For Example 1, we chose the class name *helloworld*.

The Andrew Toolkit class 'view'

In the next section, we will define the class *helloworld* to be a subclass of the Andrew Toolkit class *view*. The class *view* is at the heart of the Andrew Toolkit. It provides the methods, class procedures and data structures needed to (1) display a data object in a window, (2) request display updates and respond to such requests, and (3) respond to mouse and keyboard inputs. Thus, to create an application program using the Andrew Toolkit, you will in general want to declare a new class that is a subclass of *view*. To simplify the exposition, Example1 does not include a data object (see Example 10) and does not not respond to mouse and keyboard inputs (see Examples 2-5), but it does respond to requests to update the display.

Defining the class

To create a class, you should declare the class in a .H, or header file, that has the same name as your class. (Note that the file extension must be .H, not .h). Thus, the class declaration for the class *helloworld* is in the file *helloworld.H*.

The general format for a class declaration is

```
class <class name> : <super-class> {
overrides:
    list of inherited methods that this class is going to override
methods:
    list of methods for this class
macromethods:
    list methods for this class that are macros
classprocedures:
    list of class procedures
macros:
    list of class procedures that are macros
data:
    data structures
};
```

The following is the declaration for the class *helloworld*,

```
class helloworld : view {
overrides:
    FullUpdate(enum view_UpdateType type, long left, long top, long width, long right);
};
```

The declaration indicates that a class, named *helloworld*, is to be a subclass of the Andrew Toolkit class *view*. One method, *FullUpdate*, will override the class *view*'s *view_FullUpdate* method, which it otherwise would have inherited. No other methods, special class procedures, or data are necessary in the class declaration for this simple example.

Class hierarchies and inheritance
In the Andrew Toolkit, classes form a hierarchy. Subclasses inherit the methods and data structures of their parents, or superclasses.

Although *helloworld* declares no methods or data, it inherits the methods and data from its parent superclasses. The class *view*, *helloworld*'s immediate parent, is part of the following class hierarchy:

```
basicobject

          observable

                    view

                              helloworld
```

Therefore, *helloworld* inherits the methods and data structures of *basicobject*, *observable* and *view*. For example, *view* provides the method *Hit*, which will be called when the user clicks with the mouse on the left or right button. Because *helloworld* does not override *view_Hit*, it will inherit that method. So, when the user clicks with the mouse in a window containing a *helloworld* object, the Andrew Toolkit *Interaction Manager*, which manages interaction events, will call the method *helloworld* inherits, *view_Hit*. *view_Hit* does nothing in response to a mouse hit, so the *helloworld* object in this example will not respond to mouse hits. Example 3 will show how to override *view's* method so that the class *helloworld's* instances will respond to mouse hits.

In general, you will declare a class as a subclass of a superclass. If you leave the : < *super-class* > slot blank, the superclass defaults to *basicobject*, which is the class at the top of the Andrew Toolkit class hierarchy.

Writing a method for full update requests
Whenever the user takes an action that requires the *helloworld* object's window to be redrawn, the Andrew Toolkit *Interaction Manager* will call the *FullUpdate* method. The method should do whatever it must do to redraw the display. In this example, when *helloworld_FullUpdate* gets called, we would like it to draw the string *hello world* in the center of the window.

To write the *FullUpdate* method, you should declare it in a *.c* file that has the same name as your class. The declaration for the *helloworld's FullUpdate* method is in the file *helloworld.c*.

```
void helloworld__FullUpdate(hw, type, left, top, width, height)
    struct helloworld *hw;
    enum view_UpdateType type;
    long left;
    long top;
    long width;
    long height; {

    int x,y;
    struct rectangle VisualRect;

    helloworld_GetVisualBounds(hw,&VisualRect);
x = rectangle_Left(&VisualRect) +
        rectangle_Width(&VisualRect)/2;
y = rectangle_Top(&VisualRect) +
        rectangle_Height(&VisualRect)/2;

    helloworld_MoveTo(hw,x,y);
    helloworld_DrawString(hw,"hello world",
        graphic_BETWEENTOPANDBASELINE |
        graphic_BETWEENLEFTANDRIGHT);
}
```

Defining versus calling class procedures and methods
Note that when you *define* class procedures or methods, you must use a double underscore; when you *use*, or call, the procedures or methods, you only use a single underscore. For example, if you are creating the class *helloworld* and defining a *FullUpdate* method, you use the double underscore, and write *helloworld__FullUpdate* as above. If you were to call helloworld's *FullUpdate* method, you would use a single underscore, i.e., *helloworld_FullUpdate*. If you look in the source code for the Andrew Toolkit classes (see /usr/andrew/src/be2), all the class procedures and methods are defined with the double underscore, although when you, the application programmer, call the class procedures or methods, you only type one underscore.

The logical and visual rectangle
To understand how to do a *FullUpdate*, it is necessary to introduce some graphic concepts. Each *view* object has an associated *graphic* object in which drawing takes place. The class *graphic* supplies a *logical rectangle* which defines the coordinate system for drawing. The units are pixels. The coordinates of the logical rectangle are initially given by (0,0) at the origin on the rectangle's top left to 2(32) - 1, 2(32) - 1 at its bottom right. You can draw anywhere in this logical unsigned 32-bit coordinate space.

The logical rectangle, *lr*, gives the coordinates that you can scale your drawing to. The class *graphic* also provides a visual rectangle, *vr*, that gives the coordinates for what is potentially visible in the window. To understand *lr* and *vr*, suppose that a multi-media editor were displaying some text and a drawing, and that the drawing happens to be at the very bottom of the window. If the graphic's rectangle, *lr*, falls outside the window boundary, and the drawing is relative to *lr*, then the drawing will be clipped to the graphic's visual rectangle, *vr*.

The part of the drawing that falls within the coordinates given by the visual rectangle normally corresponds to what the user sees on the screen, except if another drawing were laid over the first (e.g., a message box could be temporarily laid on top of all or part of the drawing).

The parameters and method definition
The *FullUpdate* method has six parameters: *hw*, a pointer to the helloworld object that needs to be redrawn; *type*, the type of redraw and *left, top, width, height*, the limits of what needs to be redrawn. The last five parameters allow you to optimize the redraw (see *FullUpdate* in the section, View); we will ignore them in this example.

Updating the visual rectangle
The window is a graphic rectangle defined by a *width, height, left* and *top*. The statement, *helloworld_GetVisualBounds(hw, &VisualRect)* sets *VisualRect* to the dimensions of *hw*'s visual rectangle, *i.e.*, that part of *hw*'s graphic rectangle that is potentially visible in the window. The statements that assign values to *x* and *y* calculate the horizontal and vertical centerpoints of the visual rectangle. The values of *left* plus the *width* divided by two yield the horizontal center point; the values of *top* plus *height* divided by two yield the vertical.

An important note on calling conventions
GetVisualBounds is a method that is defined in the class *graphic*. As mentioned previously, the *helloworld* class inherits methods from that class as well as from the classes *observable* and *view*. But when you use a method defined in another class, you should call it with the class name of its first actual parameter, not the class name of the method itself. This is because an intervening class may have redefined the method or may redefine the method in the future. Thus, to insure that you use the method appropriate to your place in the class hierarchy, you call *helloworld_GetVisualBounds*, not *graphic_GetVisualBounds*, since the first parameter to the method is *hw*, even though the method was defined in *graphic*.

In general, for any object *x* in *classx*, (*i.e., struct classx *x*), you call *classx_methodname (x, <other parameters>) in order to call either a method for object x or a method for any of x*'s superclasses. Thus, for object *hw* in the class *helloworld*, you write *helloworld_GetVisualBounds (hw, <other parameters>) to call the graphic method GetVisualBounds*.

Moving the graphic point and drawing
helloworld_MoveTo moves the graphic drawing point to coordinates (x,y), the center of *hw*'s visual rectangle. The final statement, *helloworld_DrawString*, actually draws *hello world* in the window at the center point. The alignment options BETWEENTOPANDBASELINE and BETWEENLEFTANDRIGHT is used to center the string around the center coordinate (x,y). Other alignment options are available to

start the string at point (x,y), to end the string at (x,y), etc. See *graphic_DrawString* in the Reference Guide.

Importing Andrew Toolkit exports and exporting methods and class procedures

The file *helloworld.c*, the program file for the *helloworld*'s class procedures and methods, begins by importing the constructs it needs from the Andrew Toolkit library and exporting its class procedures and methods:

```
        #include <class.h>
        #include "helloworld.eh"
```

helloworld.c must #include *class.h* to access the Andrew Toolkit class facilities, and export *helloworld.eh*, the file containing the declaration of exported class procedures and methods. *helloworld.eh* is created automatically from *helloworld.H* by a *class* precompiler.

The file *hw.c*, which will contain *main()*, must import all the classes that it uses directly or indirectly. The following classes are candidates for inclusion in any application:

```
        #include "class.h"

        #include "xgraphic.ih"
        #include "xfontdesc.ih"
        #include "xim.ih"
        #include "xcursor.ih"
        #include "xws.ih"

        #include "updatelist.ih"
        #include "keyrec.ih"
        #include "proctable.ih"
        #include "keystate.ih"
        #include "keymap.ih"
        #include "menulist.ih"
        #include "fontdesc.ih"
        #include "mrl.ih"
        #include "im.ih"
        #include "buffer.ih"
        #include "mark.ih"
        #include "nestedmark.ih"
        #include "rectanglelist.ih"
        #include "text.ih"
        #include "environment.ih"
        #include "tree23int.ih"
        #include "stylesheet.ih"
        #include "style.ih"
        #include "device.ih"
        #include "context.ih"
        #include "textview.ih"
```

```
#include "frameview.ih"
#include "msghandler.ih"
#include "message.ih"
#include "framemessage.ih"
#include "cursor.ih"
#include "filetype.ih"
#include "frame.ih"
#include "framecmds.ih"
#include "scroll.ih"
#include "event.ih"
#include "search.ih"
#include "init.ih"
#include "environ.ih"
#include "dictionary.ih"
#include "viewreference.ih"
#include "bind.ih"
#include "completion.ih"
#include "atom.ih"
#include "atomlist.ih"
#include "namespace.ih"
#include "rm.ih"
```

For the program *helloworld*, the *X* classes are included because this example is to be displayed on top of the Andrew window manager. If the program is to be displayed on an X window system, then corresponding X classes must also be included. Other classes are included because *helloworld* uses them directly (e.g., *class.h, im.ih)* or indirectly through its superclasses (e.g., *observable.ih, graphic.ih, view.ih*). For example, *fontdesc and region* are used by *graphic; xfontdesc* is used by *xgraphic*; *keyrec, keystate, keymap*, and *menulist* are used by *view*; *updatelist, bind, message, msghandler, and event are* used by *im.*

In general, you must include all the classes that your class uses directly or indirectly; the list of included classes should correspond to the list of static entries in your main program (see next section). If you leave one out, when you run the program you will get the following run-time error message in your Typescript:

Could not find the class methods for <class name> (xxxxx) - exiting

where <class name> will be the name of a specific class that must be included.

In the program examples in this tutorial, the included classes are collected in a file named *staticload.h*, which is then #included:

```
#include <class.h>

#include "staticload.h"

#include "im.ih"
#include "helloworld.ih"
```

If you want to set up your own program so that it includes the file *staticload.h*, you can copy it from *usr/andrew/doc/be2/ex1*.

Setting up as a stand-alone application program

```
main()
{
    struct helloworld *hw;
    struct im *im;

    class_Init(".:/usr/andrew/dlib/be2");

    systemStaticEntries;
    helloworld_StaticEntry;

    hw=helloworld_New();
    im=im_Create(NULL);
    im_SetView(im,hw);

    im_KeyboardProcessor();

    exit(0);
}
```

To set up as a stand-alone application program, you should create a main program that declares information needed for linking and loading, creates an instance of the class, creates a window, puts the object in the window, and enters an interaction loop.

Setting the dynamic object path
main(), defined in *hw.c*, will normally begin with a call to *class_Init (".:/usr/andrew/dlib/be2")* which sets up the Andrew Toolkit linking facilities to look for objects in the current directory and the dynamic object library, */usr/andrew/dlib/be2.*

Specifying static loading
The call to *systemStaticEntries* specifies that the Andrew Toolkit classes should be loaded statically, i.e., at compile time rather than run-time. Likewise, the call *helloworld_StaticEntry* specifies that the class *helloworld* should be loaded statically. Your programs should make calls to *StaticEntry* for all classes that you know you will need in advance, because the program will be significantly smaller and slightly faster.

It is possible to compile and run this program with no static entries. If the main file had no *StaticEntry* calls, the necessary objects would be located in the dynamic object library, and loaded dynamically. However, you should nevertheless load objects you know you will use statically, for two reasons. First, as mentioned above, your program will be more efficient. Second, if a server is down, the program will not be able to dynamically load; if all objects were dynamically loaded, then the program would not run at all.

It is possible to increase the efficiency of your program even further by differentiating between classes that are used directly and indirectly. You can define a *class_StaticEntryOnly* that includes all classes that are not used directly. In this case, the only classes used directly are im and helloworld, so everything else (except class.h) could be included under *StaticEntryOnly*. This speeds up the compiling process since only the portion of each class needed for a static link is actually loaded.

Creating instances of classes
Once the necessary classes have been loaded, the call to *helloworld_New* creates an instance of the class *helloworld*. *im_Create(NULL)* creates an *Interaction Manager (im)* view and, as a side effect, creates a window using whatever window manager is running on the workstation.

The view tree
The *im* view is the root of the window's view tree. Recall that views represent space on your computer's display screen. Views are organized into trees. *im_SetView (hw, im)* puts *hw* in the view hierarchy with *im*, which happens to be the root view, as the parent. The screen space assigned to a view is always totally included within the screen space allocated to its parent view. Two sibling views, however, may find their screen space overlapping in any way, as long as both siblings are contained within their parent's screen space allocation.

It is important to understand the difference between the Andrew Toolkit class hierarchy and a view hierarchy. The class hierarchy defines a tree of parent-child relationships among classes; the view hierarchy defines a tree of parent-child relationships among *view objects*, i.e., actual instances of the class *view*. In the class hierarchy, defined through the statements class <class name> : <super-class>, both the Andrew Toolkit class *im* and the class *helloworld* are subclasses of the class *view;* in the view hierarchy, defined by calls to *im_SetView*, the view *hw*, an instance of class *helloworld*, is a child of the view *im*, an instance of the class *im*.

The interaction loop
The call to *im_KeyboardProcessor()* enters the *Interaction Manager*'s interaction loop that mediates communication between a user and the application and views in the view tree and the user. Note that *SetView* must occur after both *helloworld* and *im* are created, but before the interaction loop begins. You cannot attach a view to a view tree until the root of the view tree (im) and the view itself (helloworld) exist. The call to *im_KeyboardProcessor* must come last, since once the interaction loop is entered, subsequent calls will not be processed until the loop is exited.

Compiling for static linking

```
SRCDIR=/usr/andrew/
WINDOWDEFINES=-DX
WINDOWINCLUDES=-I/usr/include/x11
WINDOWLIBS=/usr/lib/libcwm.a /usr/lib/libcmenu.a \
/usr/lib/liboldX.a /usr/lib/libX11.a
CC=cc

INCLUDES= -I. -I${SRCDIR}include/be2 -I${SRCDIR}include
${WINDOWINCLUDES}
CLASSFLAGS=${INCLUDES}
CFLAGS= -g  ${WINDOWDEFINES} ${INCLUDES}

LIBPATH=/usr/andrew/lib/be2
LIBS=${LIBPATH}/libframe.a ${LIBPATH}/libtext.a ${LIBPATH}/libsupport.a
${LIBPATH}/libsupportviews.a ${LIBPATH}/libbasics.a ${SRCDIR}/lib/libclass.a
${WINDOWLIBS} ${SRCDIR}/lib/liberrors.a ${SRCDIR}/lib/libvfile.a
${SRCDIR}/lib/libplumber.a ${SRCDIR}/lib/libutil.a

.SUFFIXES: .ih .eh .H

.H.ih:
    class ${CLASSFLAGS} $*.H
.H.eh:
    class ${CLASSFLAGS} $*.H

hw: hw.o helloworld.o ${LIBS}
    ${CC} ${CFLAGS} -o hw hw.o helloworld.o ${LIBS} -lm
hw.o: helloworld.ih
helloworld.o: helloworld.eh
helloworld.ih:
helloworld.eh:
```

The *Makefile* for a class that will stand-alone is for the most part, like any other *Makefile* (see *make* in the online help pages). This particular Makefile finds appropriate sources and libraries, then creates *helloworld.ih, helloworld.eh,* and *helloworld.o* from the information in *helloworld.c,* and then creates the executable.

Specifying a window manager
At the present time, the Andrew Toolkit will run on top of the the window managment system X.11 and on the Andrew window manager, *wm*. In the Makefile above, the statement WINDOWDEFINES=-DX specifies that this application will be running on top of X.11. When running on top of X.11, the include directory is specified by the statement

```
WINDOWINCLUDES=-I/usr/include/x11
```

and the library path is specified by the statement

```
WINDOWLIBS=/usr/lib/libcwm.a /usr/lib/libcmenu.a \
/usr/lib/liboldX.a  /usr/lib/libX11.a
```

To run on the Andrew window manger, *wm*, you must specify WINDOWDEFINES=-DWM and specify the following include and library paths:

```
WINDOWINCLUDES=-I/usr/andrew/include
WINDOWLIBS=/usr/andrew/lib/libwm.a
```

Compiling the program
To compile the program using this *Makefile*, you should have the files *staticload.h helloworld.H*, *helloworld.c*, *hw.c* and the *Makefile* in a single directory in which you have read, write and list permissions. You may copy these files from /usr/andrew/doc/be2/ex1.

Go to the directory in which you have put the files (make it the current directory). Then at the Typescript prompt, type

```
make
```

and press the Enter key.

To run after compilation, type

```
hw
```

and press the Enter key.

Example 5: Menus

Example 5 illustrates how to create an object that has menus. The Andrew Toolkit provides two classes--*menulist* and *bind*--that together allow objects to bind lists of menus to command procedures. Menu lists provide stacks of menu cards that a user can pop-up within the body of your program's windows. Each menu list corresponds to one stack of menu cards. Your program can have only one stack of menu cards per window, but can have any number of stacks that it posts at different times. There can be any number of cards in a stack. Each card in the stack can have a title and any number of menu items. If you use the methods that *menulist* and *bind* provide to bind a menu item to a command procedure, then when the user chooses that menu item, the *Interaction Manager* will invoke the procedure. For example, if you bind the menu item Save to the procedure *Write_Buffer_to_File,* then when the user chooses the menu item Save, the *Interaction Manager* will invoke *Write_Buffer_to_File.*

The example program in this section will build upon the program in Example 4. Like Example 4, this program draws *hello world* initially at the center of the window. Then, if the user clicks on the left or the right mouse button, the program draws *hello world* centered at the location of the mouse cursor when the user lets up on the button. Like Example 4, the program also responds to two keys: ctrl-c to center *hello world,* and ctrl-i to invert the screen. In addition, the program posts a new menu card titled **Hello World** with two menu items, **Center** and **Invert.** If the user pops-up the **Hello World** menu card and chooses **Center,** the program centers *hello world* in the window; if the user chooses the menu item **Invert,** the program inverts the screen, from white to black and black to white.

The discussion that follows presents a step-by-step description of how to modify the *helloworld* class in Example 4 to produce Example 5. If you were to follow the steps, you would produce a program, called *hw,* in four files:

-- a helloworld.H file -- will contain the class definition for *helloworld.* It will be exactly the same as Example 4.

-- a helloworld.c file -- will contain statements that import Andrew Toolkit classes and define the object's methods and class procedures. We will add to Example 4's class procedure for initializing the menulist and for binding menu items to command procedures.

-- hw.c -- will contain declarations needed by Andrew Toolkit linking and loading facilities as well as statements that create an instance of *helloworld,* create a window, put the object in the window, and enter an interaction loop. This will be the same as Example 4.

-- Makefile -- will contain the directions for compiling, linking and loading. This will be exactly the same as Example 4.

For a complete listing of these files, see *Program Listing for Example 5* at the end of this section. On a first reading of this section, you may find it useful to just skim the program listing, then refer to the listing when needed as you study how to build the program. The source code is available in the directory /usr/andrew/doc/be2/ex5.

Overview of menu lists

Menus

Each menu list contains a stack of menu cards. There are methods for adding menu cards to the stack and deleting them, adding menu items to a card and deleting them, and deleting all the cards from the stack.

Menu chains

In the Andrew Toolkit, an object can contain other objects. For example, a multi-media editor may contain a drawing editor. Since an object may not know what other objects it will contain in advance, there must be a protocol that allows objects to coordinate the menu items that each object may add or delete from the stack of menus. For example, if the multi-media editor and the drawing editor both add the menu item Quit, there must be a way to decide which menu item takes priority. *Menu chains* establish the priority of menu items. Menu chains work according to the following protocol: Each object posts the menus that it wants. If the object is contained in another object, the parent object determines the placement of the child's menu list in a menu chain. If a menu list *ml(j)* occurs after the menu list *ml(i)* on the menu chain, then *m(j)*'s menus take precedence over *(ml)*'s. Thus, a parent object can decide that its menus should have precedence over its child's menus, or that its child's menus should have precedence. The usual way to decide precedence is based on input focus: the menus of the object with the input focus should have precedence.

Flow of control for menus

An object will normally define its menus when it is first loaded. In this case, the object's class procedure, *InitializeClass*, will create a menu list and add its menu items to the menus. Normally, the object will create all the menu lists it will ever use. At this point, the menu items are defined, but they are not yet visible to the user.

Whenever an instance of the class is created, its *InitializeObject* method should normally not define the menus, but should *duplicate* the menus of its class. The duplication saves space.

Most application programs start up by creating a set of views and asking the *Interaction Manager* to make one of the views the input focus. A user can request a view to make itself the input focus by clicking in its visual rectangle. If a view wishes to respond to such user requests, it should request the *Interaction Manager* to give it the input focus upon receiving the *DownTransition* in its *Hit* method.

When an object receives the input focus, it should post the *menulist* that it wants to have displayed to the user. (At the present time, if the object does not want menus, it should normally post a NULL menulist so that its parent menus are posted.) When a child posts menus, its parent's *post menu* method is called. If a parent object wants to control which menus appear, it should save all its children's menu lists when it creates the children. After the parent has created all its children, it should create its own menu list, and chain its children's menu lists into the menu chain according to the priority it desires. Then in its *post menu* method, it should post the chained list. The menu list that the top-most object posts defines the complete set of menu items that the user will see.

The class definition

If you are creating a subclass of view that will have menus, you must add a pointer to a menulist structure to the class's data definition.

For example, the following is the new class declaration for the example class *helloworld*:

```
class helloworld : view {
overrides:
    FullUpdate(enum view_UpdateType type, long left, long top, long
width, long right);
    Update();
    Hit (enum view_MouseAction action, long x, long y, long numberOfClicks)
returns struct helloworld *;
    ReceiveInputFocus();
    LoseInputFocus();
classprocedures:
    InitializeClass() returns boolean;
data:
    struct menulist *menulist;
    struct keystate *keystate;
    long x,y;
    boolean blackOnWhite;
    boolean newBlackOnWhite;
    long distX,distY;
    long newX, newY;
    boolean HaveDownTransition;
    boolean haveInputFocus;
};
```

Everything is the same as Example 4, except the data declaration *struct menulist *menulist*: *helloworld* will store its menu list in *menulist*.

Note that any class that uses menus must also override *ReceiveInputFocus* and *LoseInputFocus*.

Describing the menus
The description of the menus should be stored in a NULL terminated array of *struct bind_Description*. The structure of a bind description array is described in detail in Example 4. Each entry in the array should be a description of a single menu item binding.

For example, the two menu items for helloworld are described by the following array, *helloworldBindings* in *helloworld.c*:

```
        static struct bind_Description helloworldBindings[]={
            {"helloworld-center", "\003",0, "Hello World,Center",0,0, Center,
        "Center the helloworld string."},
            {"helloworld-invert", "\011",0, "Hello World,Invert",0,0, Invert,
        "Invert the helloworld string."},
            NULL
        };
```

The first entry in the array describes the binding of the menu item Center on the menu card Hello World to the procedure *Center*. The item in the first entry, *"helloworld-center,"* is the user invocation name, discussed in Example 4. The next two items, "\003",0, are the keyboard descriptions (see Example 4). These could be NULL,0 if you did not want key bindings at all. The next item, *"Hello World, Center"*, is the menu string. The part before the comma, *Hello World*, specifies the menu card and the part after the comma, *Center*, specifies the menu item. If no card name appears, menu items would appear on the front menu card. The menu items are listed on a card in the order they are added to the menu list and without blank lines between the menu items. Cards are stacked in the order in which they are posted. More complicated menu string formats allow you to control the order and spacing of menu items on the cards (see **Menus, Volume 2**) and the stacking of cards. The next item, 0, represents the data to be passed to the procedure upon the user selecting the menu item Center; in this example, none. The next item, 0, is the menu mask; again, in this example, none. The remaining items are the same as Example 4. Likewise, the second entry in the array binds the menu item **Invert** on the card **Hello World** to the procedure *Invert*. The third entry in the array is NULL and indicates the end of the description of the key bindings.

When you run the program and pop-up the menus, you will find that the menu card "Hello World" is the second card in the stack of menus. The first card will be unnamed, and will have the menu item Quit on it. The first card is posted by the *Interaction Manger, im, helloworld's* parent in the view tree. Parents in the view tree can choose to override menus that a child posts or add to those menus. Normally, parents give precedence to the menus of the view with the input focus.

Note that the actual command functions, *Center* and *Invert*, were discussed in Example 4. They remain exactly the same.

Creating the menu list for the class
A *menulist* for the class represents the set of bindings from the menu items to the functions to be performed. The same *menulist* can normally be shared among all instances of a view. Thus, like *keymap*, a *menulist* should normally be created in *InitializeClass*, a class procedure that is called only once--the first time the class is loaded. Likewise, the structure declaration for the *menulist* can be done in the module rather than in the class data.

In general, to create menus, you should declare an *InitializeClass* procedure that creates a new *menulist* and associates the menu bindings for the class with the newly created *menulist*.

For example, the following creates a *menulist* and associates the *menulist* with the menu descriptions for the view *helloworld* in the file *helloworld.c*:

```
static struct menulist *helloworldMenulist;


boolean helloworld__InitializeClass(classID)
    struct classheader *classID;
{
    helloworldMenulist=menulist_New();
    helloworldKeymap=keymap_New();
    bind_BindList(helloworldBindings,
        helloworldKeymap,helloworldMenulist,&helloworld_classinfo);
    return TRUE;
}
```

The line *static struct menulist *helloworldMenulist* declares the menulist structure for this class.

The statement *helloworldMenulist=menulist_New()* creates a new menulist and stores it in *helloworldMenulist*. The statement *bind_BindList (helloworldBindings, helloworldKeymap, helloworldMenulist, &helloworld_classinfo)* associates the menu descriptions in *helloworldBindings* with the newly created menulist, *helloworldMenulist*. The method *bind_BindList* is described in Example 4. The second parameter, *helloworldKeymap* could be NULL for a class with no key bindings.

If you are creating a subclass of view and your application requires dynamic menu items, then you can create multiple menu lists for the class; alternatively, you can use menu masks (see Menus, Volume 2).

Duplicating the menu list for the object
Most objects in the same class can share menus. To share menus, when an instance of an object is created, its *InitializeObject* method should not create the menus, but should *duplicate* the menus of its class. The duplicated menus should be stored as part of the view's data for later use in *ReceiveInputFocus.* In addition, a view with menus should set a flag in its *InitializeObject* method to indicate that it does not have the input focus.

For example, the following *InitializeObject* method in *helloworld.c* creates a *menulist* for the object by duplicating the one for the class and associates the *menulist* for the class, *helloworldMenulist*, with *hw*, an instance of the view *helloworld*:

```
boolean helloworld__InitializeObject(hw)
    struct helloworld *hw;
{
    hw->x = POSUNDEF;
    hw->y = POSUNDEF;
    hw->blackOnWhite = TRUE;
    hw->newBlackOnWhite = TRUE;
    hw->HaveDownTransition = FALSE;
    hw->keystate = keystate_Create(hw, helloworldKeymap);
    hw->menulist = menulist_DuplicateML(helloworldMenulist, hw);
    hw->haveInputFocus = FALSE;
    return TRUE;
}
```

The statement *hw->menulist = menulist_DuplicateML(helloworldMenulist, hw)* creates a duplicate of the class helloworld's *menulist*, *helloworldMenulist*, associates the *menulist* with the view, *hw*, and stores a pointer to the newly duplicated *menulist* in *hw->menulist*.

User requests for the input focus
By convention, a user can request a *view* to make itself the input focus by clicking within its space on the screen. If a *view* wishes to respond to such user requests, it should request the *Interaction Manager* to give it the input focus upon receiving the *DownTransition* in its *Hit* method. Normally, if you are building a *view* that will have menus, then in the *view*'s *Hit* method, you should request the input focus by calling *WantInputFocus*. In Example 5, the *helloworld_Hit* method does not need to be modified from Example 4, since it requests the input focus in order to do keyboard input.

Receiving the input focus
When the *Interaction Manager* gives a view the input focus, it notifies the view by calling a method, *ReceiveInputFocus.* If you are writing a view that has menus, its *ReceiveInputFocus* method must do two things. First, it should set a flag that indicates it has the input focus. In general, you will need to write other methods (*e.g.*, Hit, Update, etc.) so that they test this flag and act accordingly (see Example 4).

Second, the view should post its *menulist* to its parent. The *menulist* provides the *Interaction Manager* with the information it needs to post menus appropriately for the view.

For example, the following, in *helloworld.c*, is *helloworld*'s *ReceiveInputFocus* method:

```
void helloworld__ReceiveInputFocus(hw)
    struct helloworld *hw;
{
    hw->haveInputFocus = TRUE;
    hw->keystate-next = NULL;
    helloworld_PostKeyState(hw, hw->keystate);
    helloworld_PostMenus(hw, hw->menulist);
}
```

The statement *hw->haveInputFocus = TRUE sets a flag to indicate that hw* has the input focus. The statement *helloworld_PostMenus (hw, hw->menulist)* posts the *menulist hw->menulist* for *hw* to the *Interaction Manager.*

If you are creating a view that is overriding view's *ReceiveInputFocus* method, even if your view does not have a *menulist* it should post a *menulist* of NULL.

Losing the input focus
When the *Interaction Manager* takes the input focus away from a view, it notifies the view by calling another method, *LoseInputFocus.* A view's *LoseInputFocus* method should set a flag indicating that it no longer has the input focus, then do whatever it needs to do before losing the input focus. For example, if the view needs to de-highlight itself, it should request an update from it parent in order to de-highlight.

The *LoseInputFocus* method for Example 5 is exactly the same as Example 4:

```
void helloworld__LoseInputFocus(hw)
    struct helloworld *hw;
{
    hw->haveInputFocus = FALSE;
}
```

Requesting the input focus upon start-up

If you are creating an application that has menus and you want the menus to appear immediately, then it should request the input focus after putting the object in the view tree and before entering the keyboard processor interaction loop. If the input focus is not requested, the menus will not appear until the first call to *FullUpdate,* which occurs after a mouse action.

The following, in *hw.c*, requests the input focus for *hw* upon start-up:

```
hw = helloworld_New();
im = im_Create(NULL);
im_SetView(im, hw);
helloworld_WantInputFocus(hw, hw);
im_KeyboardProcessor();
exit(0);
```

This is exactly the same as Example 4.

Importing Andrew Toolkit procedures

```
#include <class.h>
#include "helloworld.eh"

#include "graphic.ih"
#include "rectangle.ih"
#include "keymap.ih"
#include "keystate.ih"
#include "bind.ih"
#include "menulist.ih"
```

The import/export declarations are exactly the same with the additional include of *menulist.ih* to import the menu list methods.

Example 11: Data Streams

Example 11 illustrates how to store data and read it from a file. We will modify the previous example to store (i.e. write) the coordinates of the hello world string in a file. Then, when the file is read in, the string will be redrawn at the stored coordinate position.

Overview of the data stream

All Andrew Toolkit applications must store data in files in a datastream format. The datastream is a standard representation for data objects in files. It consists of a standard header, a object-specific part, and a standard trailer. The header has the format:

```
    \begindata{
classname
    ,
objid \
    }
```

where *classname* is the name of the class, and *objid* is an integer unique within the datastream. A newline following the header is not considered part of the data.

The format of the enddata part is similar:

```
    \enddata{
classname,
    ,
objid
    }
```

If either of the strings "\begindata" or "\enddata" occur within the object-specific part that is not actually the header for a sub-object, the backslash in front of the string should be escaped with another backslash. In general, escaping all backslashes with a backslash is safe.

Defining the helloworld data object class

```
#define POSUNDEF -1

class helloworld: dataobject {
overrides:
    Read(FILE *file,long id) returns long;
    Write(FILE *file,long writeId,int level) returns long;
data:
    long x,y;
    boolean blackOnWhite;
};
```

The second major function of the data object class is the abstraction of permanent storage. Subclasses of dataobject can override the *Read* and *Write* methods with procedures to read and write its associated data, enabling any code that deals with dataobjects in general to also deal with this specific class.

Writing a data object to a file
The *Write* method must write the contents of a dataobject to the given *stdio* stream. The *writeId* parameter is used to make sure we are not rewriting the same thing we just wrote (so only if it's different than our stored copy of the last *writeId* do we actually write anything). The level parameter indicates whether or not this is the "toplevel" object being written (i.e., not a subobject within another object). It is the toplevel object if it is zero (so an object that is writing out sub-objects within itself should call their *Write* methods with a non-zero level). This distinction is important if an object has both a datastream and a non-datastream representation (e.g., text)-- if an object is not the toplevel object, it *must* write itself out in datastream format, as it cannot make any assumptions about whether the objects writing it out can handle either format.

The *Write* method should return its object's *UniqueID*.

```
long helloworld__Write(hw,file,writeId,level)
struct helloworld *hw;
FILE *file;
long writeId;
int level;
{
    if(writeId!=helloworld_GetWriteID(hw)){ /* only write a given version
once */
        helloworld_SetWriteID(hw,writeId);
        fprintf(file,"\\begindata{%s,%d}\n",
            class_GetTypeName(hw), helloworld_UniqueID(hw));
        fprintf(file,"%d %d %d\n",hw->x,hw->y,hw->blackOnWhite);
        fprintf(file,"\\enddata{%s,%d}\n",
            class_GetTypeName(hw), helloworld_UniqueID(hw));
    }

    return helloworld_UniqueID(hw);
}
```

The *helloworld Write* method only writes in the datastream format, so it just checks to make sure this is not a duplicate, and then writes the *begindata* header, the 3 numbers that represent the *helloworld* state, and the *enddata* trailer.

Reading a data object from a file
The *Read* method takes a *stdio* file pointer and an *id*. The *id* is either zero, indicating that no data stream header was found, or an integer guaranteed to be unique within the data stream. In the case that the *id* is zero, some dataobjects may try and to read non-datastream formats, like a text object reading a normal text file or a raster object reading a graphics file; it probably should not try to read the *enddata* either. Otherwise the stream is assumed to be one written by the object's *Write* routine, with the file pointer right after the \begindata{...}. No version checking is done, although a particular object may implement its own.

```
long helloworld__Read(hw,file,id)
struct helloworld *hw;
FILE *file;
long id;
{
    char buf[100];

    helloworld_SetID(hw,helloworld_UniqueID(hw));

    if(fgets(buf,sizeof(buf),file)==NULL)
        return dataobject_PREMATUREEOF;
    /* the %hd tells sscanf that blackOnWhite is a short, not an int*/
    if(sscanf(buf,"%d %d %hd\n",&hw->x,&hw->y,&hw->blackOnWhite)<3)
        return dataobject_PREMATUREEOF;

    if(fgets(buf,sizeof(buf),file)==NULL) /* read in the \enddata{...}*/
        return dataobject_MISSINGENDDATAMARKER;

        return dataobject_NOREADERROR;
}
```

The *helloworld_Read* procedure just tries to read two lines, the object-specific data, and the enddata trailer. If it can't read all the data, it returns a premature EOF error; if it can't read the *enddata*, it returns a missing-enddata error.

The view I/O interface

The *helloworldview* class has two procedures that are on the *helloworldview* menus which call the appropriate procedure to write or read the *helloworld* dataobject associated with the view. They use the message string input facilities discussed in example 9 to read in a filename from the user.

Writing

```
static void writeHW(hwv,key)
struct helloworldview *hwv;
long key;
{
    char file[100], msgBuf[100];
    FILE *fp;

    message_AskForString(hwv,0,"Write file: ",NULL,file,sizeof(file));
    fp=fopen(file,"w");
    if(fp==NULL){
        sprintf(msgBuf,"Couldn't open %s for writing.",file);
        message_DisplayString(hwv,1,msgBuf);
    }else{
        struct helloworld *hw=
          (struct helloworld *)hwv->header.view.dataobject;

        helloworld_Write(hw,fp,im_GetWriteID(),0);
        fclose(fp);
    }
}
```

This routine simply prompts the user for the filename, opens a file, calls the *helloworld_Write* procedure on its dataobject, and closes the file.

Reading

```
static void readHW(hwv,key)
struct helloworldview *hwv;
long key;
{
    char file[100], msgBuf[100];
    FILE *fp;

    message_AskForString(hwv,0,"Read file: ",NULL,file,sizeof(file));
    fp=fopen(file,"r");
    if(fp==NULL){
        sprintf(msgBuf,"Couldn't open %s for reading.", file);
        message_DisplayString(hwv,1,msgBuf);
```

```
            }else{
                char header[100];
                if(fgets(header,sizeof(header),fp)==NULL){
                    sprintf(msgBuf,"Premature end-of-file in %s.",file);
                    message_DisplayString(hwv,1,msgBuf);
                }else{
                    char name[20];
                    int id;

            if(sscanf(header,"\begindata{%[^,],%d}\n",name,&id)!=2){
                        sprintf(msgBuf,
                            "%s doesn't contain a valid datastream header.",
                            file);
                        message_DisplayString(hwv,1,msgBuf);
                    }else{
                        struct helloworld *hw=
                            (struct helloworld *)hwv->header.view.dataobject;

                        if(strcmp(name,class_GetTypeName(hw))!=0){
                            sprintf(msgBuf,
                                "%s doesn't contain a helloworld dataobject.",
                                file);
                            message_DisplayString(hwv,1,msgBuf);
                        }else{
                            /* FINALLY, read the object in... */
                            helloworld_Read(hw,fp,id);
                            fclose(fp);
                            helloworld_NotifyObservers(hw,0);
                        }
                    }
                }
            }
        }
```

This is slightly more complicated than the *writeHW* procedure, as it must also parse the first line of the file it opens and make sure its both a valid *begindata* marker, and that the class that it is really a *helloworld* dataobject. If so, it calls the *helloworld_Read* method with the *id* from the *begindata* header.

Binding the functions

```
static struct bind_Description helloworldviewBindings[]={
    {"helloworldview-center", "\003",0, "Hello World,Center",0, Center,
"Center the helloworldview string."},
    {"helloworldview-invert", "\011",0, "Hello World,Invert",0, Invert,
"Invert the helloworldview string."},
    {"helloworldview-relocate", "\022",0, "Hello World,Relocate",0,
relocate, "Relocate the helloworld string."},
    {"helloworldview-read", NULL,0, "Hello World,Read",0, readHW,
      "Read in a new hello world."},
    {"helloworldview-write", NULL,0, "Hello World,Write",0, writeHW,
      "Write out the current hello world to a file."},
    NULL
};
```

The bold face section is added to bind the readHW and writeHW functions for the program. Note that these are menu commands only; there are no key bindings for the functions.

Example 12: Multiple Views on One Data Object

Example 12 illustrates how to have more than one view onto a single data object. We will modify the program from Example 11 to produce two windows, each with *hello world* in the center. The two *hello world* strings are two views onto the same data object. Thus, when the *helloworld* data object is changed in one window, (i.e., when the user moves the *hello world* text string with mouse, menu, or key commands), the data object in the other window reacts in the same way since it is the same data object. However, you can look at different parts of the view in the windows: if you move the scroll bars in one window, you will be seeing a different portion of the view in that window, but the view in the other window will not change.

Creating multiple windows

With the split between the dataobject and the view, it is very simple to have multiple windows open on the same dataobject. The general procedure is to simply create another view, set its dataobject to the same one as the previous view and create an im, frame, etc., to put the new view in. (You can't just point another im at the original view).

```
void makeWindow(dataObj)
struct helloworld *dataObj;
{
    struct helloworldview *hwv=helloworldview_New();
    struct frame *frame=frame_New();
    struct im *im=im_Create(NULL);

    helloworldview_SetDataObject(hwv,dataObj);
    frame_SetView(frame,helloworldview_GetApplicationLayer(hwv));
    im_SetView(im,frame);

    /* the last one called will end up with the focus */
    helloworldview_WantInputFocus(hwv,hwv);
}
```

In this example, the procedure *makeWindow* simply encapsulates the procedure necessary to display a view on the given dataobject, by creating a frame, an im, and a view pointing at the given dataobject.

```
main()
{
    struct helloworld *hw;
    class_Init(".:/usr/andrew/dlib/be2");

    systemStaticEntries;
    helloworld_StaticEntry;
    helloworldview_StaticEntry;

    hw=helloworld_New();

    makeWindow(hw);
    makeWindow(hw); /* 2nd window */
```

```
        im_KeyboardProcessor();

        exit(0);
    }
```

The main procedure now uses the *makeWindow* procedure to create a window on the *helloworld* data object, and makes a second call to create a second window.

Example 17: Creating a View that has Children

The perspective on the Andrew Toolkit interaction up to now has been from that of the child view: Examples 1-16 illustrated how to build a class that would respond appropriately if included as a child in other views. This example shifts to the parent's perspective: It illustrates how to build a view that includes a child.

This example also illustrates a principle that guided much of the Andrew system development: When application programmers are implementing new views, they should use views that already exist as building blocks for their own views whenever possible. For example, the Andrew mail program uses a *textview* as a building block. The mail view builds on top of the *textview* code without changing *textview* at all, but simply adds new commands to implement functionality needed for mail. To illustrate this principle, Example 17 will use a *textview* as its child view. Of course, a view's child can also be a completely new view. Using an already existing view simply saves effort.

Example 17 works similarly to Example 16. Instead of displaying the string, "hello world," however, it displays a *textview* with a *text* data object that contains the text "hello world." The user interface is the same, with the exception that mouse hits inside of the *textview* go to the *textview*, so the user must click outside of the *textview* rectangle to drag *helloworld* around, for instance.

The example illustrates the following activities:

creating a child view at the appropriate time

allocating screen space for a child

inserting the child into the view tree

passing requests (e.g., mouse hits) to your child

passing your child's requests (e.g., update, input focus) up the view tree

Although the discussion refers directly to this example, which illustrates a view with a single child, the information applies generally to the creating of any view with any number of children.

Creating a view from component views

You should consider creating a new view from existing building blocks whenever part of your application's user interface shares significant similarities to an existing view's user interface. For example, the user interface for the body of a mail message is similar to the user interface provided by *textview*, so it makes sense to build the body of a mail message from *textview*. A new view that you build from an existing component can override the component's user interface behavior, so the user interface can differ somewhat from the component's. A rule of thumb for deciding how different the two can be is the following: If you are considering creating a new view from an existing building block, but find yourself needing to make changes to large numbers of the building block's methods, you will probably be better off implementing a new view altogether; on the other hand, if there are large numbers of methods that you can use as is, or you can use by simply adding new methods, the building block approach will save you effort.

Defining the data object

If you are building a view that is going to have a child views and the children have associated data objects, you must declare the *dataobject* class procedures *InitializeObject* and *FinalizeObject*. These class procedures are the places where you will create the child views' data objects and delete them, respectively. You must also modify the *dataobject*'s data definition to add pointers to the child views' data objects.

For example, the following is the new class declaration for the example dataobject, *helloworld*:

```
class helloworld: dataobject {
overrides:
    Read(FILE *file,long id) returns long;
    Write(FILE *file,long writeId,int level) returns long;
classprocedures:
    InitializeObject(struct helloworld *hw) returns boolean;
FinalizeObject(struct helloworld *hw);}
data:
    long x,y;
    boolean blackOnWhite;
  struct text *text;
};
```

In previous examples, to display ''hello world'' we used a constant string that was cheap to generate, so we did not need to explicitly store it. But in this example, to display "hello world" we will use a child view, *textview*, whose data object is a *text*. The statement *struct text *text* declares a pointer to it.

In previous examples, *FinalizeObject* was not declared, since it had no work to do. In this example, it will free the space allocated to the child view's data object, *text*.

Creating the children's data objects

When you are creating a new view with children, you must create the children's data objects and keep pointers to them in the new view's associated data object. You should also initialize the children's data objects. These activities should be done in the associated dataobject's *InitializeObject* class procedure.

For example, the following is the *InitializeObject* procedure for *helloworld*:

```
boolean helloworld__InitializeObject(classID,hw)
struct classheader *classID;
struct helloworld *hw;
{
    hw->x = POSUNDEF;
    hw->y = POSUNDEF;
    hw->blackOnWhite = TRUE;

  hw-text=text_New();

  text_InsertCharacters(hw->text,
                              0,"Hello world!",sizeof("Hello world!")-1);
  text_AddStyle(hw->text,0,5,bold);
 text_AddStyle(hw->text,6,5,italic);
    return TRUE;
}
```

In this example, the *helloworldview* will have a single child, a *textview*. The statement *hw->text=text_New()* creates an instance of *textview*'s data object, *text*, and stores a pointer to it in *hw->text*.

The remaining statements, *text_InsertCharacters* and *text_AddStyle*, initialize the child data object. It is analogous to the use with text objects previously.

Deleting the children's data objects

When a parent object with children is deleted, it should free up the children's storage, provided of course that there are no other pointers to the children. The freeing of storage should be done in the parent's *FinalizeObject* class procedure by calls to the children's *Destroy* or *Finalize* procedures.

For example, the following is helloworld's FinalizeObject procedure:

```
void helloworld__FinalizeObject(classID,hw)
struct classheader *classID;
struct helloworld *hw;
{
    text_Destroy(hw->text);
}
```

The statement *text_Destroy(hw->text)* frees the storage allocated to the text object pointed to by *hw->text*.

Writing a data object with children to a file

The *Write* method must write the contents of the data object to a file. When a view/data object is constructed from component views and data objects, the parent data object's *Write* method must call the children's *Write* methods.

For example, the following is Example 17's *Write* method:

```
long helloworld__Write(hw,file,writeId,level)
struct helloworld *hw;
FILE *file;
long writeId;
int level;
{
    if(writeId!=helloworld_GetWriteID(hw)){ /* only write a given version once */
        helloworld_SetWriteID(hw,writeId);
        fprintf(file,"\begindata{%s,%d}\n",
            class_GetTypeName(hw), helloworld_UniqueID(hw));
        fprintf(file,"%d %d %d\n",hw->x,hw->y,hw->blackOnWhite);
        text_Write(hw->text,file,writeId,level++);}
        fprintf(file,"\enddata{%s,%d}\n",
            class_GetTypeName(hw), helloworld_UniqueID(hw));
    }

    return helloworld_UniqueID(hw);
}
```

The *Write* method is the same as Example 11, with the exception of the statement *text_Write (hw->text,file,writeId,level++)*. Like before, the method writes out the position and color (black or white) of the helloworld dataobject, *hw*, but the statement *text_Write* writes out the text object, *hw->text*, as well--inside the helloworld dataobject.

Recall that the *level* parameter in the *Write* method indicates whether or not the "top level" object is being written (i.e., not a sub-object within another object in the datastream object hierarchy). Note that the level parameter must be incremented in the call to *text_Write*, since the object being written is not a top level object. This is especially important for text, which does not write in the datastream format if it is the top-level object.

Reading a data object with children from a file

The *Read* method for a data object with children is similar to the *Read* methods already discussed, but the method must also read in any embedded data objects. To read embedded objects, the method must scan for the \begindata header that the object should have written when its *Write* routine was called. If the header exists, then calling the *Read* method for the object will read the object, up to and including its \enddata marker. After the embedded object returns success, the parent's *Read* method should read any other data, including its own \enddata marker.

For example, the following is Example 17's *Read* method:

```
    long helloworld__Read(hw,file,id)
    struct helloworld *hw;
    FILE *file;
    long id;
    {
        char buf[100],classNameBuf[100];
        long retVal,textObjId;

        helloworld_SetID(hw,helloworld_UniqueID(hw));

        if(fgets(buf,sizeof(buf),file)==NULL ||
         /* the %hd tells scanf that blackOnWhite is a short, not an int*/
           sscanf(buf,"%d %d %hd\n",&hw->x,&hw->y,&hw->blackOnWhite)<3||
           fgets(buf,sizeof(buf),file)==NULL ||

    sscanf(buf,"\begindata{%[^,],%d}\n",classNameBuf,&textObjId)<2)
                retVal=dataobject_PREMATUREEOF;
        else if(strcmp(classNameBuf,"text")!=0)
                retVal=dataobject_BADFORMAT;
        else{
                retVal=text_Read(hw->text,file,id);
                if(retVal==dataobject_NOREADERROR)
                    if(fgets(buf,sizeof(buf),file)==NULL) /* read the \enddata{...}*/
                        retVal=dataobject_MISSINGENDDATAMARKER;
        }
        return retVal;
    }
```

This method is similar to previous *Read* methods, but after reading in the *x, y, & blackOnWhite* values correctly, the method attempts to read in the embedded text object. The statement *sscanf(buf,"\\begindata{%[^,],%d}\n",classNameBuf,&textObjId))* scans for the \begindata header, returning *dataobject_PREMATUREEOF* upon an error. If it exists, the statement *if (strcmp(classNameBuf,"text")!=0)* checks that the object is of type text as we expect it in this example to be, returning *dataobject_BADFORMAT* upon an error. If it is of type text, the statement *retVal=text_Read(hw->textfile,id)* reads the text object, up to an including its \enddata marker. If the text_Read was successful (i.e., *if(retVal==dataobject_NOREADERROR)* succeeds) the statement *if(fgets(buf,sizeof(buf),file)==NULL)* performs a quick "check" for an \enddata marker, returning *dataobject_MISSINGENDDATAMARKER* upon a failure, success otherwise.

Defining the view

If you are building a view that is going to have child views, you must override the *view* methods *LinkTree* and *SetDataObject.* You must write a LinkTree method that links your child views into the view tree; you must write a SetDataObject method that associates the child views with their data objects. You must also modify the *view*'s data definition to add a pointers to the child views.

For example, the following are the new elements in the class declaration for the example view, *helloworldview*:

```
class helloworldview: view {
overrides:
    SetDataObject(struct helloworld *hw);
        FullUpdate(enum view_UpdateType type, long left, long top, long
          width, long right);
        Update();
        Hit (enum view_MouseAction action, long x, long y, long numberOfClicks)
          returns struct view *;
        ReceiveInputFocus();
        LoseInputFocus();
        GetInterface(int type) returns char *;
        GetApplicationLayer() returns struct view *;
        DeleteApplicationLayer(struct view *);
    LinkTree(struct view *parent);
classprocedures:
        InitializeClass() returns boolean;
data:
        struct keystate *keystate;
        struct menulist *menulist;
        boolean movingString;
        boolean haveInputFocus;
        long hitX,hitY;
        int x,y;
        boolean blackOnWhite;
```

```
            long frameX, frameY;
            long newFrameX, newFrameY;
            int grWidth,grHeight;
        struct textview *textview;
        struct view *view;
    };
```

The helloworld view must now have a textview to display its dataobject's text dataobject. Since we would like a scrollable textview, we also keep a pointer to the textview's application layer, view.

Creating the children's views

If you are creating a new view with children, you must create the children's views. This activity should be done in the parent view's *InitializeObject* class procedure.

For example, the following is the *InitializeObject* procedure for *helloworldview*:

```
    boolean helloworldview__InitializeObject(classID,hwv)
    struct classheader *classID;
    struct helloworldview *hwv;
    {
            hwv->haveInputFocus=FALSE;
            hwv->movingString=FALSE;
            hwv->keystate=keystate_Create(hwv,helloworldviewKeymap);
            hwv->menulist=menulist_DuplicateML(helloworldviewMenulist,hwv);
            hwv->newFrameX=0;
            hwv->newFrameY=0;
        hwv->textview=textview_New();
        hwv->view=textview_GetApplicationLayer(hwv-textview);
            return TRUE;
    }
```

The statement *hwv->textview=textview_New()* creates helloworld view's child, a *textview* object, and assigns a pointer to it in *hwv->textview*. The statement *hwv->view=textview_GetApplicationLayer(hwv->textview)* creates an application layer for the *hwv->textview*, and assigns a pointer to in in *hwv->view*.

Deleting the children's views

When a parent view with children is deleted, it should free up the children's storage, provided of course that there are no other pointers to the children. The freeing of storage should be done in the parent's *FinalizeObject* class procedure by calls to the children's *Destroy* or *Finalize* procedures.

For example, the following is *helloworldview*'s FinalizeObject procedure:

```
    void helloworldview__FinalizeObject(classID,hwv)
    struct classheader *classID;
    struct helloworldview *hwv;
    {
        textview_DeleteApplicationLayer(hwv->textview,hwv->view);
        textview_Destroy(hwv->textview);
    }
```

The statement *textview_DeleteApplicationLayer(hwv->textview,hwv->view)* frees the storage allocated for *hwv->view*. The statement *textview_Destroy(hwv->textview)* frees the storage allocated for *hwv->textview*.

Inserting the children into the view tree

When building a new view out of other component views, your new view must insert the child views in the view tree, and allocate screen space to these child views. The insertion of child views under a parent view in the view tree is done with the view method, *LinkTree*. You must override the method and write a method that calls *LinkTree* for each of its children, and then calls *super_LinkTree*.

For example, for *helloworldview*:

```
    void helloworldview__LinkTree(hwv,parent)
    struct helloworldview *hwv;
    struct view *parent;
    {
        view_LinkTree(hwv->view,hwv);
        super_LinkTree(hwv,parent);
    }
```

The statement *view_LinkTree (hwv->view, hwv)* links *hwv*'s child, *hwv->view*, into the view tree. The statement *super_LinkTree (hwv,parent)* links *hwv* itself into the view tree.

This example links the child into the view tree and leaves it. For applications that require dynamic view hierarchies, the Andrew Toolkit provides a method for taking children out of the view tree, *view_UnlinkTree* (see View, Vol. 2).

Associating the child views with their data objects

When building a new view out of other component views, your new view must associate the child views with their data objects. The association of the child views with their data objects must be done with the view method, *SetDataObject*. You must override the method and write a method that calls *SetDataObject* for each of its children, and then calls *super_SetDataObject*.

For example, for *helloworldview*:

```
    void helloworldview__SetDataObject(hwv,hw)
    struct helloworldview *hwv;
    struct helloworld *hw;
    {
        hwv->x=hw->x;
        hwv->y=hw->y;
        hwv->blackOnWhite=hw->blackOnWhite;
        textview_SetDataObject(hwv->textview,hw->text);
        super_SetDataObject(hwv,hw);
    }
```

The first three statements initialize view data from the data object. (Such initializations cannot be done in the view's *InitalizeObject* method, since the view does not have a handle on its data object there.) The statement *textview_SetDataObject(hwv->textview,hw->text)* associates *hwv*'s child, *hwv->textview*, with its dataobject, *hw->text*. The statement *super_SetDataObject(hwv,hw)* associates *hwv* itself with its dataobject, *hw*.

Writing the method for full update requests

When building a new view out of child views, your new view must call the FullUpdate method for all its children within its FullUpdate method. Before calling a child's FullUpdate method, the parent should:

optionally negotiate with the child about its desired size (see *view_DesiredSize*, View, Vol. 2)

allocate space to the child view with *view_InsertView*

You have the option to initiate a negotiation with the child view about how much size it wants, given a particular space offer. In this case you can call the child view's *view_DesiredSize* method, giving it an offer of a certain amount of space, and requesting that it tell you how much space it actually needs.

A view's *DesiredSize* procedure can be called anytime after view_LinkTree has been called on the view. This means that the view's parent pointer points to the actual view's parent. This enables the view to discover what type of display it will be displayed upon, and to make calls that return font dimensional information. However, the view's space allocation has not yet been set, or more precisely, may change again. The *view_DesiredSize* procedure is an optional procedure and if no such procedure is provided by the view's author, the default action is for the view to claim all of the screen space it is offered. Note that a parent view does not have a responsibility to call a child view's *DesiredSize*: the parent view may already know how much screen space to allocate to the child based upon the parent's own dimensions, for example. This is the case in Example 17, below.

To allocate space to a child view, you call view_InsertView which sets the child's local bounds rectangle and its visible rectangle. The local bounds rectangle defines the space allocated to the child view, while the visible rectangle indicates how much of this space is actually visible on the screen (assuming this can be expressed as a single rectangle). Normally you would make this call at the end of a size negotiation and before calling the child's *FullUpdate* method.

Once a child's *DesiredSize* method is optionally called, and the child's *InsertView* and *FullUpdate* methods invoked, the child view is essentially "born," that is, the child will take its place, if any, on the screen display.

The following is the *FullUpdate* method for *helloworldview*:

```
void helloworldview__FullUpdate(hwv,type,left,top,width,height)
struct helloworldview *hwv;
enum view_UpdateType type;
long left;
long top;
long width;
long height;
{
    struct helloworld *hw=(struct helloworld *)hwv->header.view.dataobject;
    struct rectangle myVisualRect,rec;
```

```
        helloworldview_GetVisualBounds(hwv,&myVisualRect);
        hwv->grWidth=rectangle_Width(&myVisualRect);
        hwv->grHeight=rectangle_Height(&myVisualRect);

        if (hwv->newFrameX+hwv->grWidthTOTALSIZE)
                    hwv->newFrameX=TOTALSIZE-hwv->grWidth;
        if (hwv->newFrameY+hwv->grHeightTOTALSIZE)
                    hwv->newFrameY=TOTALSIZE-hwv->grHeight;

        hwv->frameX=hwv->newFrameX;
        hwv->frameY=hwv->newFrameY;

        if(hw->x==POSUNDEF){
            hw->x=hwv->frameX+(hwv->grWidth-WIDTH)/2;
            hw->y=hwv->frameY+(hwv->grHeight-HEIGHT)/2;
        }

        hwv->x=hw->x;
        hwv->y=hw->y;
        hwv->blackOnWhite=hw->blackOnWhite;
    /* rec is the rectangle in which the  child view */
    /* will be displayed, plus a one pixel border. */
    rectangle_SetRectSize(&rec,
            hwv->x-hwv->frameX-1,hwv->y-hwv->frameY-1,WIDTH+1,HEIGHT+1);

helloworldview_SetTransferMode(hwv,graphic_COPY);
        if(hw->blackOnWhite){
                helloworldview_FillRect(hwv,
                    &myVisualRect,helloworldview_WhitePattern(hwv));
        /* if on white background, draw a rectangle around it */
            helloworldview_DrawRect(hwv,&rec);
    }else{
        helloworldview_FillRect(hwv,
                    &myVisualRect,helloworldview_BlackPattern(hwv));
/* Most views expect a white background,*/
/* so make one for the sub-view */
        helloworldview_FillRect(hwv,&rec,helloworldview_WhitePattern(hwv))
    }

/* Get rid of the border pixel */
```

```
rec.top++;
rec.left++;
rec.width--;
rec.height--;

/* InsertView sets the child view's coordinate system */

view_InsertView(hwv->view,hwv,&rec);

/* FullUpdate draws the child view */

view_FullUpdate(hwv->view,view_FullRedraw,0,0,WIDTH,HEIGHT);
}
```

Rather than negotiate about size, the statement *rectangle_SetRectSize (&rec, hwv->x-hwv->frameX-1, hwv->y-hwv->frameY-1, WIDTH+1, HEIGHT+1)* calculates a size for the *hwv*'s child based on *hwv*'s size.

The statement *view_InsertView (hwv->view, hwv, &rec)* sets *hwv*'s child, *hwv->view*, to have the logical and visible coordinates specified by *&rec*.

The statement *view_FullUpdate(hwv->view,view_FullRedraw,0,0,{WIDTH,HEIGHT)* calls the child's *FullUpdate* method.

Updating the screen partially

If you are creating a view that will have children, then you may need to handle partial updates to the children. The following *Update* method from Example 17 is illustrative:

```
void helloworldview__Update(hwv)
struct helloworldview *hwv;
{
        struct helloworld *hw=(struct helloworld *)hwv>-header.view.dataobject;

        helloworldview_SetTransferMode(hwv, graphic_COPY);
    if(hwv->x!=hw->x ||
        hwv->y!=hw->y ||
        hwv->frameX!=hwv->newFrameX  ||
        hwv->frameY!=hwv->newFrameY ||
        hwv->blackOnWhite!=hw->blackOnWhite){
                        struct rectangle rec;

                        if(hwv->x!=hw->x ||
                            hwv->y!=hw->y){
                                static char buf[100];
                                sprintf(buf,"Hello world at (%d,%d)",hw->x,
                                                    hw->y);
                                message_DisplayString(hwv,0,buf);
    }

    if(hw->blackOnWhite!=hwv->blackOnWhite){
        struct rectangle vr;
        helloworldview_GetVisualBounds(hwv,&vr);
    if(hw-blackOnWhite)
            helloworldview_FillRect(hwv,
                                    &vr,
                                    helloworldview_WhitePattern(hwv));
    else
            helloworldview_FillRect(hwv,
                                    &vr,
                                    helloworldview_BlackPattern(hwv));
    hwv->blackOnWhite=hw->blackOnWhite;
    }
    /* includes 1 pixel border */
    rectangle_SetRectSize(&rec,
                                hwv->x-hwv->frameX-1,hwv->y-hwv>-frameY-1,
                                WIDTH+2,HEIGHT+2);

/* erase the old child view */

            if(hw->blackOnWhite)
                helloworldview_FillRect(hwv,
                                    &rec,
                helloworldview_WhitePattern(hwv));
                    else
                        helloworldview_FillRect(hwv,
                                        &rec
                helloworldview_BlackPattern(hwv));

    hwv->x=hw->x;
    hwv->y=hw->y;
    hwv->frameX=hwv->newFrameX;
    hwv->frameY=hwv->newFrameY;

/* Re-set the child view's size and coordinates */
        rectangle_SetRectSize(&rec,
                                    hwv->x-hwv->frameX,hwv->y-hwv->frameY,
                                    WIDTH,HEIGHT);
            view_InsertView(hwv>-view,hwv,&rec);

        /* Completely redraw it, since it's moved */
        /* (not as efficient as raster-op-ing it to the new place, but much easier) */
            if(hw->blackOnWhite)
                helloworldview_DrawRectSize(hwv,
                                            hwv->x-hwv->frameX-1,
                                            hwv->y-hwv>-frameY-1,
                                            WIDTH+1,HEIGHT+1);
            view_FullUpdate(hwv-view,view_FullRedraw,0,0,WIDTH,HEIGHT);

        /* If the view hasn't moved, simply let it do
            any updating that it needs to */
        }else
            view_Update(hwv-view);
    }
```

Passing mouse events to children

In general, it is up to the parent view to pass interesting events down to its children by calling the child's specialized methods.

There is also one standard view method that a parent view often invokes in a child view: the *Hit* method. In particular, quite often, when a mouse hit occurs in a region of the screen allocated to a child view, the correct thing to do is simply pass the mouse hit to the child view. Example 17's Hit method illustrates this strategy:

```
    struct view *helloworldview__Hit(hwv,action,x,y,numberOfClicks)
    struct helloworldview *hwv;
    enum view_MouseAction action;
    long x;
    long y;
    long numberOfClicks;
    {
            struct helloworld *hw=(struct helloworld *)hwv->header.view.dataobject;
    /* We send mouse hits to the child, hwv->view */
    /* if they fall within its visible rectangle.*/
        if(!hwv->movingString &&
            x>=(hwv->x-hwv->frameX) && x<(hwv->x-hwv->frameX+WIDTH) &&
            y>=(hwv->y-hwv->frameY) && y(hwv->y-hwv->frameY+HEIGHT)){

                view_Hit(hwv->view,
                    action,
                    x-(hwv->x-hwv->frameX),
                    y-(hwv->y-hwv->frameY),
                    numberOfClicks);
                return;
    }

    if(hwv->movingString)
            switch(action){
                case view_RightUp:
                    hwv->movingString=FALSE;
                    /* fall through */
                case view_RightMovement:
                    hw->x+=x-hwv->hitX;
                    hw->y+=y-hwv->hitY;
                    hwv->hitX=x;
                    hwv->hitY=y;
                    break;
                case view_LeftUp:
                    hwv->movingString=FALSE;
                    hw->x=x+hwv>-frameX;
                    hw->y=y+hwv->frameY;
                    break;
                case view_LeftMovement:
                    /* do nothing */
                    break;
                default:
                    /* re-synchronize mouse */
                    hwv->movingString=FALSE;
        }

    if(!hwv->movingString)
            switch(action){
                case view_RightDown:
                    hwv->hitX=x;
                    hwv->hitY=y;
                    /* fall through */
                case view_LeftDown:
                    hwv->movingString=TRUE;
                    helloworldview_WantInputFocus(hwv,hwv);
                    break;
        }

    helloworld_NotifyObservers(hw,0);
    return (struct view *)hwv;
    }
```

Note that two of the parameters to the *Hit* method are the *x* and *y* coordinates of the mouse hit, and these coordinates are expressed relative to the parent view's local bounds rectangle, so the parent view's *Hit* method must subtract the difference in origins between its view and the child's view before calling the child's *Hit* method. Don't forget that the child's *Hit* method returns a value, so you should return this value as the value of your own *Hit* method.

Menus and keystates

The child's *Hit* method may request the input focus. If granted the input focus, the child view's *ReceiveInputFocus* method may post menus or keystates. The parent view has full control over these possibilities. In particular, the parent view may intercept, and possibly ignore, requests for the input focus from child views by providing a *view_WantInputFocus* method. This method will be invoked every time a child (or any descendant) of the parent view requests the input focus. The first parameter to this method will be the parent, and the second parameter will be the view that desires the input focus. The parent method can discard the request by simply returning, in which case the descendant will never receive the input focus. Alternatively, the parent can permit the request to continue up the view tree by invoking its own parent's *view_WantInputFocus* method with the first parameter equal to its parent's view and the second parameter being the view desiring the input focus.

Once a child view is granted the input focus by Andrew Toolkit, its *ReceiveInputFocus* method will be called. In this method, a child view may post menus or keystates (providing key-to-command binds). Again, a parent view may intercept these requests and modify them. To intercept menu posting requests from its children, a parent should provide a *PostMenus* method; to intercept keystate posting requests from your children, the parent should provide a *PostKeyState* method. Note that if you do not provide *PostMenus* or *PostKeyState* methods, defaults will be provided that allow you children's posts to proceed up the view tree unimpeded.

Your view's *PostMenus* method is called with two parameters, the first being a pointer to your view and the second being the menu list being posted by one of your child views. Your *PostMenus* method has several options. It can discard the request by simply returning, which will result in the child's menus being discarded. It can allow the request by calling its own parent's *PostMenus* method with the same menu list. Finally, your view can change the menulist being posted, by either making a copy of the menulist with changes of your own, chaining your own menulist changes to the original menulist or completely replacing the menulist with a new menulist of your own design (see Menu List, Vol. 2).

Your view's *PostKeyState* method is called with two parameters, a pointer to your view and a pointer to the keystate being posted. Your view can allow the post to continue unimpeded by invoking its own parent with the same keystate structure. Your view can discard the keystate by simply returning, in which case the last keystate actually posted will remain in effect. Or your view can post its own keystate, by invoking its parent's *PostKeyState* method with a new second parameter. Keystates can be threaded together in a list, just like menulists (Keystate, Vol. 2).

View

The class *view* provides an interface to the underlying window system. As an Andrew Toolkit application programmer, you should understand why applications programs should not interact directly with the underlying window system: application programs must often be ported to other workstations that will eventually be deployed. The workstations often have similar, but not identical, display capabilities and the hardware can be significantly different; the underlying window system may be different. To insure portability of your application, you should never make direct calls to the underlying window system; you should always use the *view* class procedures and methods; doing so will maximize the device independence and portability of your application program.

Overview of View

The associated data object

A *view* may have an associated *dataobject* (see the section, **Data Object**). Typically, an instance of *view* displays a *dataobject* in a rectangular area on the workstation display. Usually the data object is displayed because a user wishes to view or change the data. The class *view* is responsible for managing the display of the data and the interaction with the user; the class *dataobject* is responsible for maintaining the data, including storing and manipulating it. In general, views represent the input/output interface to data objects; data objects represent memory or permanent storage.

Together with the class *dataobject*, the class *view* provides an architecture that supports multiple views of a single data object. For example, a document may contain both a graphical and a tabular view of the same underlying numerical data. When the user changes the numbers in the table, the table and the graph can each adjust accordingly. Likewise, a text editor may display the same document in more than one window with changes reflected appropriately in each.

Subclassing view

Suppose that *spreadsheetview* is a sub-class of the class *view*. Then

```
struct spreadsheetview *ssv;
ssv = spreadsheetview_New();
```

creates an instance of *spreadsheetview*; *ssv* points to the newly created instance. Since *ssv* is a *view*, it inherits *view*'s data and methods. The writer of spreadsheetview must override many of the view methods.

When the spreadsheetview_New is called it initializes all the data associated with an instance of spreadsheetview. This includes the data associated with view. This is done by the system first calling view_InitializeObject followed by spreadsheetview_InitializeObject. The writer of the spreadsheetview class only provides the spreadsheetview_InitializeObject. The InitializeObject routines are called automatically. The programmer never calls them directly.

```
class spreadsheetview: view {

classprocedures:
    InitializeObject(struct spreadsheetview *view) returns boolean;
    FinalizeObject(struct spreadsheetview *view);
data:
    boolean HasInputFocus;
    short nLines;
    short nRows;
    struct keystate *keystate;
};
```

Because spreadsheet has data, you must declare an InitilalizeObject:

```
boolean spreadsheetview__InitializeObject(classID, ssv)
struct classheader *classID;   /* This parameter is ignored */
struct spreadsheetview *ssv; {

    ssv->HasInputFocus = FALSE;
    ssv->nLines=0;
    ssv->nRows=0;
    ssv->keystate = keystate_Create(ssv, ssvKeymap);
    return TRUE;
}
```

Similarly, to delete an instance of spreadsheetview its Destroy routine is called. This will automatically call the FinalizeObject procedure for spreadsheetview, if it is declared as a classprocedure. The programmer never calls the FinalizeObject procedure directly. The FinalizeObject procedure is normally used to delete any allocated memory associated with the instance being deleted. In this case the FinalizeObject procedure would have to destroy its keystate.

```
void spreadsheetview__FinalizeObject(classID, ssv)
struct classheader *classID;
struct spreadsheet *ssv; {

if (ssv-keystate != NULL)
    keystate_Destroy(ssv-keystate);
}
```

Building the view tree

Views are organized in a tree structure. Prior to any events being sent to the view it must be added to a tree. There is a view tree associated with window opened on the display. For each window there is an interaction manager(im) view, that is the root of the view tree. All views in the view tree have a pointer to the im view that is the root of the tree.

Adding a view into the view tree

```
void view_LinkTree(childview, parentview)
struct view *childview;
struct view *parentview;
```

Method description. *view_LinkTree* is used to link *childview* (and its descendants) into *parentview*'s view tree. This sets the parent pointer in the child to point to the parent. It sets *childview*'s im pointer to be the same as *parentview*'s im pointer. It also associates a graphic with the child that is of the same type as the parent.

view_LinkTree can also be called with a NULL parentview. This is used to invalidate the graphic and im pointer associated with *childview*. When *parentview* is NULL the parent field for the view should not be reset.

Usage. Before drawing or passing events to a child view, the parent should make a call to the child's LinkTree method. When the child receives a call to its LinkTree method it must call the LinkTree method for all of its children. When the entire view tree is built, each view will have pointers to graphic objects of the type defined by the im object.

Removing a view from the view tree

```
void view_UnlinkTree(rview)
struct view *rview;
```

Method description. *view_UnlinkTree* is used to remove *rview* (and its descendants) from its current view tree. This method begins a notification procedure that notifies other views in the view tree that this subtree is being removed. It also must invalidate the graphics objects associated with each of its children. The default UnlinkTree method calls *view_UnlinkNotification* on its parent, sets its parent pointer to NULL, and then calls *LinkTree* on itself with a NULL parent parameter.

In most cases this method need not be overridden.

Usage. Whenever a view is being removed from a view tree it must be unlinked using this method.

Negotiating about the size of a view

Specifying the desired size of the view

```
enum view_DSattributes view_DesiredSize(childview, width, height,
pass, desiredWidth, desiredHeight)
struct view *childview;
long width;
long height;
enum view_DSpass pass;
long *desiredWidth;
long *desiredHeight;
```

Method description. *view_DesiredSize* is used to conduct size negotiations between a parent and child view. The parameters for *height* and *width* are filled with the parent's preferred values; the *pass* parameter tells the child which parameters are flexible and which are fixed. The *desiredWidth* and *desiredHeight* parameters are for the child to pass back what it would like for its size. *view_DesiredSize* may be called several times as the parent negotiates with its child. The parameter *pass* can take on one of three values:

view_NOSET - if both dimensions are flexible.

view_WIDTHSET - if the width can not be changed.

view_HEIGHTSET - if the height can not be changed.

In response to these three parameters, *view_DesiredSize* sets the *desiredWidth* and *desiredHeight*.

Return value. The return value is a *DSattributes* and can be set to an OR-ed combination of any of the following values:

view_FIXED - the size is fixed, that is, not flexible. (=0)

view_WIDTHSMALLER -the width could be made smaller. (=2)

view_WIDTHLARGER - the width could be made larger. (=1)

view_WIDTHFLEXIBLE - the width is flexible. (=3)

view_HEIGHTSMALLER - the height could be made smaller. (=8)

view_HEIGHTLARGER - the height could be made larger. (=4)

view_HEIGHTFLEXIBLE - the height is flexible. (=12)

view_DesiredSize returns view_HEIGHTFLEXIBLE | view_WIDTHFLEXIBLE. If you are creating a subclass of the class *view* named *x* and you want to negotiate the size of *x*, you should return a value that is appropriate.

The default method sets *desiredWidth* and *desiredHeight* to *width* and *height* respectively. It returns that both values are flexible.

Usage. The process of negotiating between the parent's proposal and the child's request will normally take place during the call to the *FullUpdate* of the parent. The actual size and position of the view *childview* must be set prior to the parent calling the *FullUpdate* procedure of *childview*.

Selected Interfaces to the Andrew Toolkit

Part 1: Data Object

void **New**(do)

void **Read**(do,file, id)

SetID(id)

GetID(id)

long **Write**(do, file, writeID, level)

UniqueID(id)

GetWriteID(id)

SetWriteID(id)

long **GetModified**(do)

void **SetModified**(do)

char * **ViewName**(do)

void **SetAttributes**(do, attributes)

void **SetDataObject**(viewer, dataobj)

void **LinkTree**(childview, parentview)

void **UnlinkTree**(rview)

void **UnlinkNotification**(v, unlinkedview)

boolean **IsAncestor**(v, ancestor)

void **InsertView**(childv, parentv, enclosedRectangle)

enum view_DSattributes **DesiredSize**(childview, width, height,

pass, desiredWidth, desiredHeight)

void **GetOrigin**(childview, width, height, originX, originY)

void **FullUpdate**(viewer, type, left, top, width, height)

void **WantUpdate**(view,updateView)

void **Update**(viewer)

void **ObservedChanged**(viewer,changedobj, status)

void **WantInputFocus**(viewer, requester)

void **ReceiveInputFocus**(viewer)

void **LoseInputFocus**(viewee)

struct view * **Hit**(v, action, x, y, numberOfClicks)

void **PostKeyState**(viewer, kstate)

void **PostMenus**(viewer, menulist)

void **PostCursor**(v, rect, c)

void **RetractCursor**(v, c)

void **RetractViewCursors**(view, requestor)

struct basicobject * **WantHandler**(viewer, handlerName)

void **PostDefaultHandler**(viewer, handlerName, handler)

struct view ***GetApplicationLayer**(viewer)

void **DeleteApplicationLayer**(viewer)

void **Print**(viewer, file, processor, finalFormat, topLevel)

char ***GetInterface**(viewer, type)

void **SetName**(view, name)

struct atomlist * **GetName**(view)

struct atomlist * **GetClass**(view)

short **GetParameter**(view, name, type, data)

short **GetResource**(view, name, class, type, data)

void **GetManyParameters**(view, resources, name, class)

void **PostResource**(view, path, type, data)

Part 3: View Graphic Methods

struct graphic ***CreateGraphic**(classID)

void **InsertGraphic**(EnclosedGraphic, EnclosingGraphic, EnclosedRectangle)

void **MoveToPt**(grphc,NewPosition)

void **MoveTo**(grphc, NewX, NewY)

void **Move**(grphc, DeltaX, DeltaY)

void **DrawLineToPt**(grphc,LineEnd)

void **DrawLineTo**(grphc, XEnd, YEnd)

void **DrawLine**(grphc, DeltaX, DeltaY)

void **SetLineWidth**(grphc,NewLineWidth)

short **GetLineWidth**(grphc)

void **SetFont**(grphc, ChosenFont)

struct fontdesc ***GetFont**(grphc)

void **DrawText**(grphc, Text, TextLength, Operation)

void **DrawRect**(grphc, Rect)

void **DrawPolygon**(grphc, PointArray, PointCount)

void **DrawOval**(grphc, Rect)

void {**DrawArc**(grphc, EnclRect, StartAngle, OffsetAngle)

void **FillRect**(grphc, Rect, Tile)

void **FillPolygon**(grphc,PointArray, PointCount, Tile)

void **FillOval**(grphc, Rect, Tile)

void **FillArc**(grphc, EnclRect,StartAngle, OffsetAngle,Tile)

void **GetLogicalBounds**(grphc,rect)

long **GetLogicalRight**(grphc)

long **GetLogicalBottom**(grphc)

void **GetEnclosedBounds**(grphc,rect)

long **GetEnclosedRight**(grphc)

long **GetEnclosedBottom**(grphc)

void **GetVisualBounds**(grphc,rect)

long **GetVisualRight**(grphc)

long **GetVisualBottom**(grphc)

void **SetClippingRect**(grphc, rect)

void **ClearClippingRect**(grphc)

void **GetClippingRect**(grphc,rect)

void **SetTransferMode**(grphc,NewTransferMode)

short **GetTransferMode**(grphc)

void **FlushGraphics**(grphc)

struct pixmap * **WhitePattern**(grphc)

struct pixmap * **BlackPattern**(grphc)

struct pixmap * **GrayPattern**(grphc,IntensityNum, IntensityDenom) struct text ***New**()

void **Clear**(txt)

void **Read**(txt, file, id)

long **ReadSubString**(txt, pos, file, quoteCharacters)

long **Write**(txt, file, writeID, level)

void **WriteSubString**(txt, pos, n, file, quoteCharacters)

boolean **ReplaceCharacters**(txt, pos, len, replacementString,

replacementLen)

void **SetAttributes**(txt, attributes)

long **HandleDataObject**(txt, pos, dop, file)

long **HandleKeyWord**(txt, pos, keyword, file)

void **ObservedChanged**(txt, changed, value)

struct viewreference ***InsertObject** (txt, pos, name,viewname)

struct environment ***AddStyle**(txt, pos, len, style)

struct environment ***AddView**(txt, pos, viewtype, dataobject)

void **SetEnvironmentStyle**(txt, envptr, styleptr)

void **SetGlobalStyle**(txt, styleptr)

struct style ***GetGlobalStyle**(txt)

void **ApplyEnvironment**(classID, sv, defaultStyle, env,devicename, contextname)

struct textview ***New**()

void **SetDataObject**(txtv,txt)

void **FullUpdate** (txtv, type, left, top, width, height)

Update (txtv)

ReceiveInputFocus(txtv)

LoseInputFocus(txtv)

struct view ***Hit**(txtv, action, x, y, numberOfClicks)

WantUpdate(txtv, descendant)

void **WantNewSize**(txtv,requestor)

void **ObservedChanged**(txtv, changed, value)

void **SetTopPosition**(txtv, newTopPosition)

long **GetTopPosition**(txtv)

void **SetBorder**(txtv, xBorder, yBorder)

void **GetClickPosition**(txtv, position, numberOfClicks, action,

startLeft, startRight, leftPos, rightPos)

long **MoveBack**(txtv, pos, units, type, distMoved, linesAdded)

long **MoveForward**(txtv, pos, units, type, distMoved, linesAdded)

void **SetDefaultStyle**(txtv, styleptr)

struct style ***GetDefaultStyle**(txtv)

Part 5: Table

void **ChangeSize**(nrows, ncols)

struct table ***CopyData**(chunk, destroy)

void **InsertData**(T, Chunk chunk)

void **WriteASCII** (f, rowdata, coldata)

struct table ***ReadASCII**(f, rowdata, coldata)

void **FormatCell**(cell, buff)

void **ParseCell** (struct cell * ell, buff)

void **ChangeThickness**(dim, i, thickness)

void **FindBoundary**(chunk)

void **SetInterior**(chunk, color)

void **SetBoundary**(chunk, color)

void **ReEval**(r, c)

void **SetFormat**(ch, chunk)

void **SetPrecision**(precision, chunk)

void **Imbed**(name, chunk)

void **Lock**(ch, chunk)

Part 6: Raster

struct raster ***Create**(rast, width, height)

Resize(rast, width, height)

Clear(rast)

long **GetWidth**(rast)

long **GetHeight**(rast)

long **GetPixel**(rast, x, y)

SetPixel(rast, x, y, pixelValue)

long **GetRow**(rast, x, y, length, dest)

long **SetRow**(rast, x, y, length, src)

long **GetColumn**(rast, x, y, length, dest)

long **SetColumn**(rast, x, y, length, src)

long **ReadSubRaster**(rast, file, r)

long **WriteSubRaster**(rast, file, objectid, subraster)

PaintSubraster(rast, subrect, byte)

InvertSubraster(rast, subrect)

MirrorLRSubraster(rast, subrect)

MirrorUDSubraster(rast, subrect)

GetRotatedSubraster(rast, subrect, target)

boolean **GetResized**(rast)

boolean **GetChanged**(rast)

A User's View of Andrew Toolkit Applications

The following section contains excerpts from *A Guide To Andrew*, an illustrated user's manual that is widely used on the Carnegie Mellon University campus and at other Andrew sites.

The following examples from the Guide assume that you are running applications over a network with a workstation connected to the Andrew File System. They should accurately portray the behavior of most applications in most circumstances, but some things will inevitably be different. For example:

> If you are running on a standalone workstation, you will notice a number of differences, chiefly relating to program startups and the fetching, storing, and printing of mail and files.
>
> This documentation contains many examples of the way the system runs at CMU. Because your system and the support people at your site are different, these examples will be inappropriate for you. For example, CMU has services such as "Advisor" and "Help Comments" to which users can send questions about Andrew. CMU also has a user consultant staff in public workstation clusters around campus. Occasionally, we refer to these services. Though the examples may be inappropriate as written, we hope you can think of ways to adapt them to your circumstances.
>
> Originally at CMU, the Help system and *A Guide to Andrew* were designed to work together, with references in the Guide intended as lead-ins to the more advanced information found in the Help files. However, some changes were made to the released versions of the Help files, so we are not certain that all of the references in here work the way we intended, though we are confident that most them do.

You can order a complete copy of this guide now if you want one. A new version of the Guide, intended for standalone machines, is in preparation and should be available in April. See the section "For Additional Information" for an order form.

Sections in this Part

This part of the document contains the following sections:

1. Introduction
2. Logging In
3. Using Windows
4. Pop-Up Menus
5. Ending an Andrew Session
6. The Scrollbar

7. Selected Regions

8. Copying and Pasting

9. The Help System

10. Writing and Printing

Sections 11, 12, and 13, have been omitted. These sections were titled, "Working with Files and Directories," "Protecting Files and Directories," and "Checking Your Space Allocation." Some references to these sections may remain in the excerpts. We apologize for any inconvenience.

14. Mail and Bulletin Boards

1. Introduction

What is Andrew?

The Andrew System, or *Andrew*, is a set of tools that you can use to write and edit documents, send and receive mail, read bulletin boards, write your own programs, and do many other activities. The tools that you use on Andrew, such as the editor and the mail system, are computer programs designed to help you accomplish the work you need to do. Most people use Andrew on a special kind of computer, an *Andrew workstation* with a large screen such as an IBM PC-RT, a DEC MicroVax, or a Sun Workstation. On an Andrew workstation, you can use more than one Andrew program at once because each program appears in its own *window* or area of the screen. You can learn more about windows and the programs that appear in them in the following tutorials.

Many of the Andrew programs, such as the editor (called "EZ") and the mail programs ("Messages" and "SendMessage"), rely on the same basic operations and are easy to learn because of their similarities. All the programs use information that is stored on a *network* that connects hundreds of workstations across the CMU campus. The *Andrew File System* uses the network to store and retrieve thousands of pieces of electronic information each day. Because Andrew workstations are connected by the network in special ways, any *files* or documents that you create and all the Andrew programs are available from any Andrew workstation anywhere on campus.

Andrew uses versions of the *UNIX*(tm)* operating system to perform many of its most basic operations, like copying or deleting files. UNIX is a powerful operating system that allows for *multi-processing*, or for many activities to be taking place inside the workstation at once, instead of occurring one after the other as they do on many other kinds of computers. This guide contains some introductory information about UNIX and how to use it for managing your files.

Andrew was developed by the Information Technology Center at CMU as part of a joint effort between CMU and IBM.

* UNIX(tm) is a registered trademark of AT & T.

If You Have Problems using Andrew

If you have any problems using Andrew, you should send electronic mail to the Andrew user called *advisor*; your mail will be answered within two working days and an effort will be made to correct whatever problems you are having. (You can use the *Messages* program to send mail. See page 89 of this guide for more information about Messages.) User consultants are available at the workstation clusters around campus, such as UCC 100 and Porter Hall 217, and you should ask a user consultant for help if you need it. If you cannot contact advisor or a user consultant, you can call a user consultant at 268-2667 from 9 a.m. to 9 p.m. Monday-Friday for assistance.

Here is a quick reference chart of people who can help you and how contact them:

advisor	by electronic mail
user consultant	in workstation clusters
user consultant (phone)	at 268-2667, 9 a.m.-9 p.m., M-F

2. Logging In

Identifying Your Workstation

Andrew currently runs on four different computers: the IBM RT-PC workstation, the Microvax workstation from Digital, and two kinds of Sun workstations, the Sun 2 and the Sun 3. Before you log in, check to see which workstation you are working on. The name of the manufacturer of the workstation will usually appear on the front of the workstation.

Parts of some workstations are slightly different from the same parts on other workstations. Some keyboards, for example, have a key labelled "Return" while others have the same key labelled "Enter." In this guide we refer to the "Enter" key, but if your keyboard only has a "Return" key, you should use that instead.

The mouse is also different for different workstations. (The mouse is the hand-sized box connected to your workstation by a thin cable. You can place it on either the left or right side of the workstation to use it.) While the IBM RT-PC has a two-button mouse, the Sun and Microvax workstations each have a three-button mouse. The instructions in this guide are geared to the two-button mouse because the IBM RT-PC is the most common Andrew workstation on campus. The three-button mouse works the same as the two-button mouse, except that instead of pressing both keys on a two-button mouse for some operations, you press the middle key on the three-button mouse.

At a glance, here is the difference between using the two- and three-button mouse:

2-Button Mouse	**3-Button Mouse**
both buttons	center button

Logging In

To log into an Andrew workstation, you must have an Andrew user ID (login name) and a password. If you do not have a user ID and a password, check with an account administrator in your department or the teacher in your Andrew class to find out what they are.

If the **login:** prompt does not currently appear on your workstation screen, press the Enter key to make it appear. If it still does not appear, ask a user consultant to help you.

To log in:

Action. At login: type your user ID. If you make a mistake, use the Backspace or Delete key to correct it; then press Enter.

Response. The word "Password:" appears.

Action. At the Password: prompt, type your password, then press Enter. Your password does not appear on the screen when you type it.

Response. A "Wecome to Andrew" message appears.

The "Welcome to Andrew" message does not stay on the screen long. It is replaced by the windows on the initial screen described on the next page.

If you cannot log in

If you see the message "Login incorrect" when you try to log in, you may have mistyped your user ID or password. Try again. If you cannot log in after four tries, go back to the person who told you your user ID and password, and tell them what happened. That person will help you find out what is wrong.

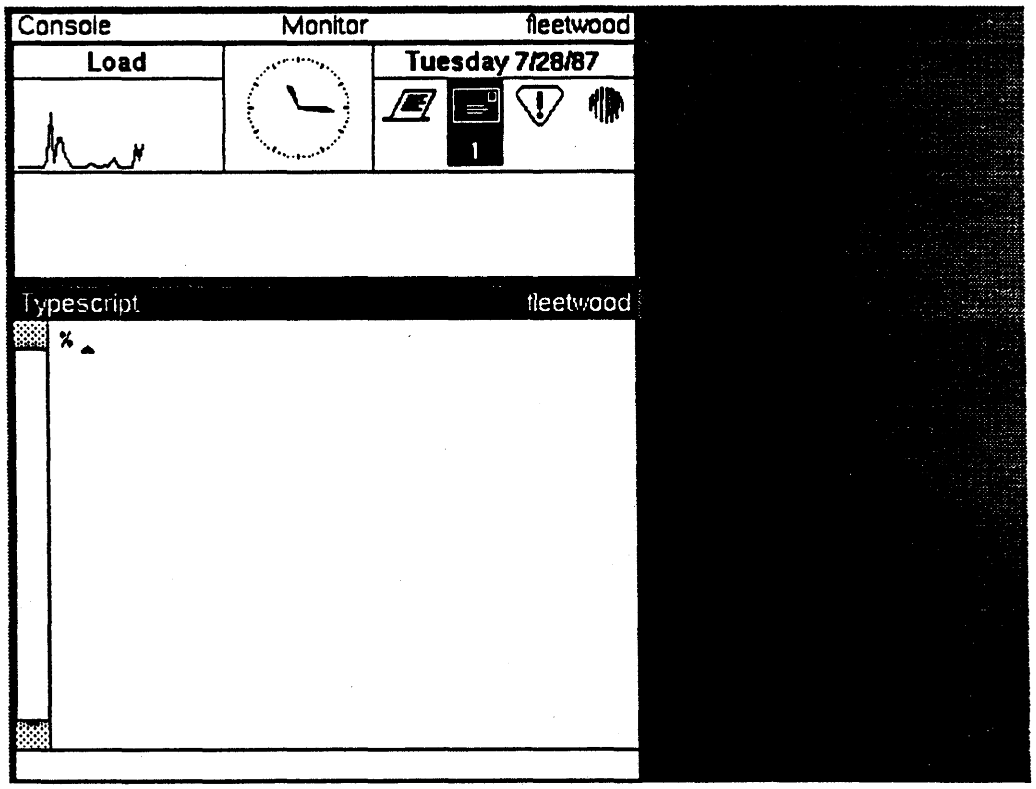
The Initial Screen

After the "Welcome to Andrew" message appears, the workstation screen becomes gray and two windows, Console and Typescript, appear on the left side with a gray area to the right. There may be messages in the Console window. You can ignore them for now.

If the bar at the top of the Typescript window on your screen is not darkened like the one in the illustration below, use the mouse to move the mouse cursor (the arrow on the screen) into the Typescript window. The bar at the top of the window, called the ***title bar***, will become darkened and the letters that you type in the next exercise will appear in the Typescript window.

If the screen goes dark

Do not be alarmed if while doing the exercises you stop working to read for a few moments and the workstation screen becomes dark. If an Andrew workstation is idle for 10 minutes or more, the screen will dim and an icon will move around the dark screen to remind you that you are still logged in. None of your work will be disrupted. Move the mouse to brighten the screen again.



Moving the Mouse

As you have seen, the mouse is the hand-sized box connected to your workstation by a thin cable. Sliding the mouse across the surface of the table beside your workstation or across a mouse pad slides the mouse cursor in the same direction across your computer screen. Moving the mouse without pressing any of the mouse buttons only moves the cursor; it does not affect any information on the screen except to make different windows active.

Action. Move the mouse cursor around the screen slowly. If you are using an IBM RT-PC or a Microvax workstation, you can slide the mouse on the table next to your workstation to move the mouse cursor around the screen.

If you are using a Sun workstation, you need to keep the mouse on the *mouse pad* beside the workstation when you slide it. If you come to the edge of the mouse pad but want to move the mouse cursor further around the screen, lift the mouse, put it back on the pad, and begin sliding it again.

Response. The movement of the mouse cursor on the screen corresponds to your movement of the mouse on the table or mouse pad. As you move your mouse cursor, notice two things:

1. Different windows become active. When the mouse cursor is in a window, that window's title bar becomes darkened. A darkened title bar means that the window is active and ready to receive input from what you type or from the pop-up menus. When the mouse cursor leaves one window to enter another one, the title bar on the first window becomes light again, while the title bar on the second window darkens.

2. The mouse cursor changes its shape. As the mouse cursor moves across different areas of the screen, different mouse cursors appear. Each mouse cursor has a special significance and tells you about the different actions that can be taken in that area.

The next page contains a reference list of the different mouse cursors that you might see as you work with Andrew.

The Mouse Cursors

As you move the mouse cursor around the screen, here are some cursors you might see. Don't be alarmed if you do not see all of them at any one time; some of them appear under special circumstances.

Curved arrow cursor. As you move it into a window, the cursor activates the window. You re-position the text caret by pointing and clicking with left mouse button.

Straight arrow cursor. Appears on menus and in Help and Messages. Use it to point and choose an item from a list.

Double arrow cursor. Appears when the mouse cursor is in the scrollbar (the bar on the left side of some windows). Click left mouse button to move the text forward and see text that is not currently displayed in window.

Clockface cursor. Tells you to wait and stop all work within that window until the clockface goes away. You can move the cursor to another window and work.

Small-window cursor. Appears when you move the mouse cursor into the title bar of the window (the bar at the top of the window). When the small-window cursor appears, you can use the title bar menus.

Target cursor. Appears when you move the mouse cursor into the title bar and click the right mouse button. The window disappears. Move the target cursor to a different part of the screen and click the right mouse button again. The window reappears in this new position.

Boundary cursor. Appears when you move the mouse cursor into the boundary between two windows or two areas of the same window. Press and hold any mouse button and slide the mouse to change the length or width of windows. You can learn more about the boundary cursor in the next few exercises.

Console: An Information Window

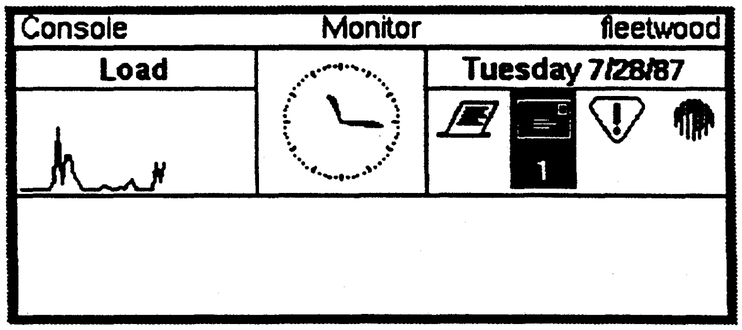
The Console window in the upper left corner of the screen keeps track of various kinds of information about your workstation.

As you work, the Load graph changes to represent how busy your workstation is, or how much of the computer's processing power you are using at any given moment. The higher the load, the longer some activities will take, such as opening new windows. If the load goes to 100% and stays for more than a few minutes, ask a user consultant to look at the workstation you are using.

The clock shows the current time and the current date appears to the right of the clock. The icons or pictures under the date darken when certain kinds of activity occur:

> **Print icon:** darkens when you have sent something to the printer. The number of files remaining to be printed appears under the icon. Becomes light again when the all the files are printed.
>
> **Letter icon:** darkens when you have new mail. The number of new mail messages appears under the icon. Becomes light again when you use any mail-reading program on Andrew. (You can learn about one mail-reading program, Messages, later in this guide.)
>
> **Trouble icon:** darkens when your workstation has a problem that you should correct before continuing to work. Move the mouse cursor into the trouble icon and click the left mouse button for more information. Becomes light again when the problem is corrected.
>
> **File system icon:** darkens when file system activity takes place on your workstation. Becomes light again when the file system activity is complete.

The file system icon darkens frequently as your workstation retrieves programs and files from other workstations on the network. When the file system icon darkens, you know that the command you issued is still being processed. When the file system icon becomes light again, you can expect your command to be completed soon.



Using Console

Most people simply use Console by watching for changes in the icons. However, the icons in Console supply more information if you click on them.

To find out more about the status of your workstation:

> **Action.** Move the mouse cursor into any of the icons in the Console and click the left mouse button.
>
> **Response.** Information about the function that the icon represents appears in the body of the Console window.

From time to time messages appear in the body of the Console, advising you about processing that your workstation is doing or errors that have occurred in parts of the network. Many of the messages that you see in the Console are only informative for the moment. You may wish to clear them periodically so that you will notice when new ones appear.

To clear the Console window of messages that appear there:

> **Action.** Mouse the mouse cursor into the body of the Console window and click the left mouse button several times.
>
> **Response.** The messages disappear off the top of the body of the Console window, one line at a time.

You can change several things about the Console, like whether the clock is analog or digital, by using its pop-up menus. You can learn more about the pop-up menus on Andrew in the exercises that follow. A list of the menu items in Console and what they do appears in the last section of this guide.

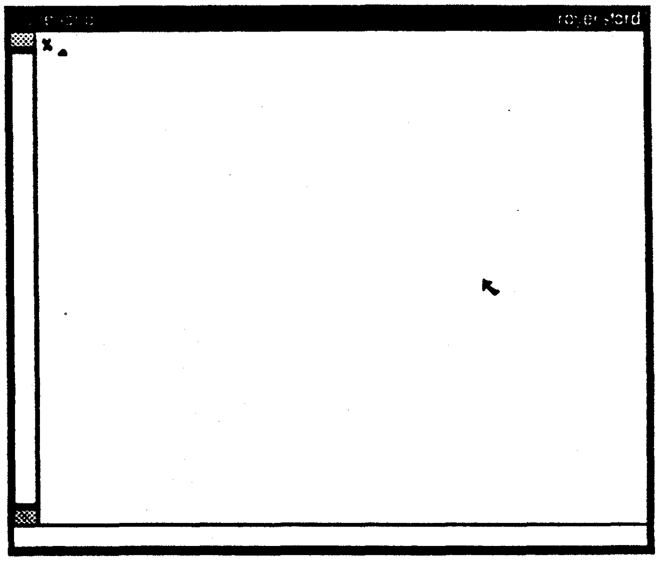
Typescript: The Command Window

In UNIX terms, Typescript is an interface to the UNIX shell or the part of UNIX that interprets commands. In more straightforward terms, Typescript is the window where you type commands to tell the computer what to do. Because it is a command window, you need a Typescript window for your entire session when you log into Andrew, so one is provided for you automatically or by default.

The Typescript window has some features that are common to many windows on Andrew. The parts of the window are labeled below and explained here.

- **Title bar** --shows you the name of the program running in a window (at the extreme left of the title bar). The center of the title bar is reserved for special information. For example, in the Console window, the name of the particular Console being used is presented ("Monitor" is the default). The extreme right of the title bar shows the name of your workstation.
- **Percent sign** -- the Typescript prompt. When you see a percent sign with nothing beside it in a Typescript window, you know that you can type a command there.
- **Text caret** -- a small upward pointing arrow indicating where the next letters you type will appear.
- **Mouse cursor** -- the object that moves around the screen as you move the mouse.
- **Body** -- the area in which you do your work or information appears.
- **Scrollbar** -- the part of the window you use to bring other sections of a long document into view inside a window.
- **Message area** -- the place where messages are displayed when you are working inside a window. For example, when you save a document using EZ, a message appears the message area stating that the document has been written into a file.

Only the percent sign (prompt) is unique to the Typescript window. Any window that has a scrollbar shares the same basic features as the Typescript window.



3. Using Windows

About Windows

A window is a box-shaped area of the computer screen in which one program runs or you work on one kind of task. Most windows on Andrew have similar parts that work the same way, such as the title bar. Typescript and Console each appear in a window in the illustration below.

On Andrew, windows appear in a *tiled* or *tiling* format. That is, one window never overlaps another. If your screen is filled with windows and you open another one, the existing windows become smaller to make room for the new window. Typescript and Console windows always appear on the left side of the screen (in the left column, in other words). Most other windows, however, appear in the wider right column. Once the windows appear, you can change their shapes or move them wherever you like on the screen.

Andrew has two kinds of windows:

Windows with a scrollbar. Most Andrew windows have a scrollbar along the left side, which means that you can move both backward and forward through the contents of the window. The Typescript window has a scrollbar.

Windows without a scrollbar. Some Andrew windows, such as Console, do not have a scrollbar. You cannot read back over previous information in these windows; only current information, such as the status icons and messages in the Console, appears there.

You can only work in one window at a time, and you must move the mouse cursor into a window before you can work in it.

Opening a Window

There are two common ways to open new windows on Andrew. One way is to use the pop-up menus. You can learn more about the menus in the next section.

Another common way to open new windows is to type commands in the Typescript window. Not every Typescript command starts a new window. However, many of the programs that people use each day on Andrew open a window of their own. The Help program is a good example of a program that creates a new window, and you can use a Help window to learn more about how windows work.

Action. Type

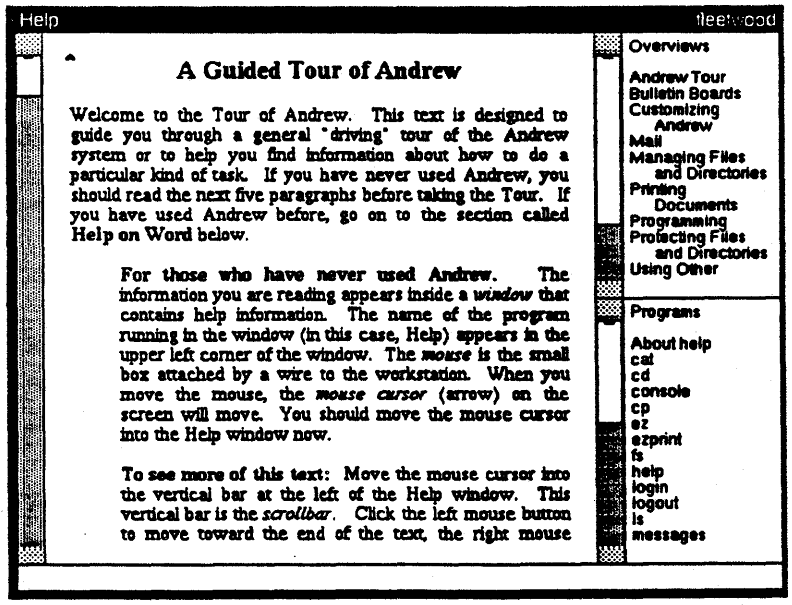
help

in the Typescript window and press the Enter key.

Response. After a few seconds, a Help window appears in the right column.

If a new window does not appear, a problem has occurred on the network or the file system. Ask a user consultant for assistance.

Do not be alarmed if part of the text in the Help window turns dark with white letters. If that happens, you have inadvertently created a *selected region*. Click the left mouse button while the mouse cursor is anywhere inside the main part of the Help window to remove the selected region. You can learn more about selected regions in a later section of this guide.



A Guided Tour of Andrew

Welcome to the Tour of Andrew. This text is designed to guide you through a general "driving" tour of the Andrew system or to help you find information about how to do a particular kind of task. If you have never used Andrew, you should read the next five paragraphs before taking the Tour. If you have used Andrew before, go on to the section called **Help on Word** below.

For those who have never used Andrew. The information you are reading appears inside a ***window*** that contains help information. The name of the program running in the window (in this case, Help) appears in the upper left corner of the window. The ***mouse*** is the small box attached by a wire to the workstation. When you move the mouse, the ***mouse cursor*** (arrow) on the screen will move. You should move the mouse cursor into the Help window now.

To see more of this text: Move the mouse cursor into the vertical bar at the left of the Help window. This vertical bar is the *scrollbar*. Click the left mouse button to move toward the end of the text, the right mouse

Changing the Size of Windows

You can change the sizes of the windows on the screen in two ways, both illustrated here. First, you can change the width or height of a window. Second, you can hide or "shrink" the window so that only the title bar of the window is displayed.

Moving Window Boundaries

To change the width of a window:

Action. Slowly move your mouse cursor into the vertical boundary between the Help window and Typescript. When the mouse cursor turns into a boundary cursor, press and hold any mouse button and drag the cursor to the left; then release the mouse button.

Response. The windows adjust to the new border. In this case, Typescript and Console become thinner while the Help window becomes wider.

You can move the boundary to the right, too, if you wish to practice.

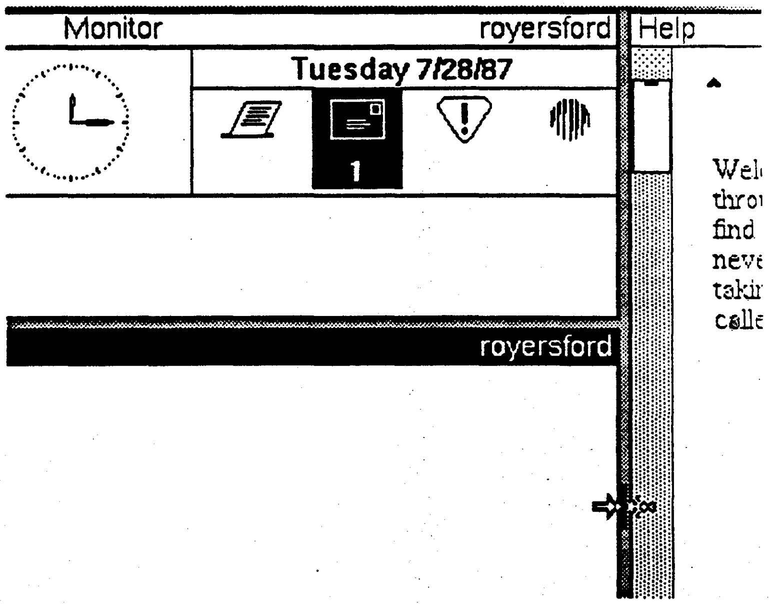
To change the height of a window:

Action. Slowly move your mouse cursor into the horizontal boundary between Console and Typescript. When the boundary cursor appears, press and hold any mouse button; then drag the cursor down a bit and release the mouse button.

Response. The windows adjust to the new border. In this case, Typescript becomes shorter and Console becomes taller.

After you practice moving the window boundaries, move them back into their orginal positions (that is, have the left column occupy about a third of the screen, the right column about two thirds) to make the next exercises a little easier to complete.

You can use these methods to change the width and height of any windows on the screen.



Shrinking and Unshrinking Windows

To shrink a window to its title bar:

Action. Move the mouse cursor into the title bar of the Help window and click the left mouse button.

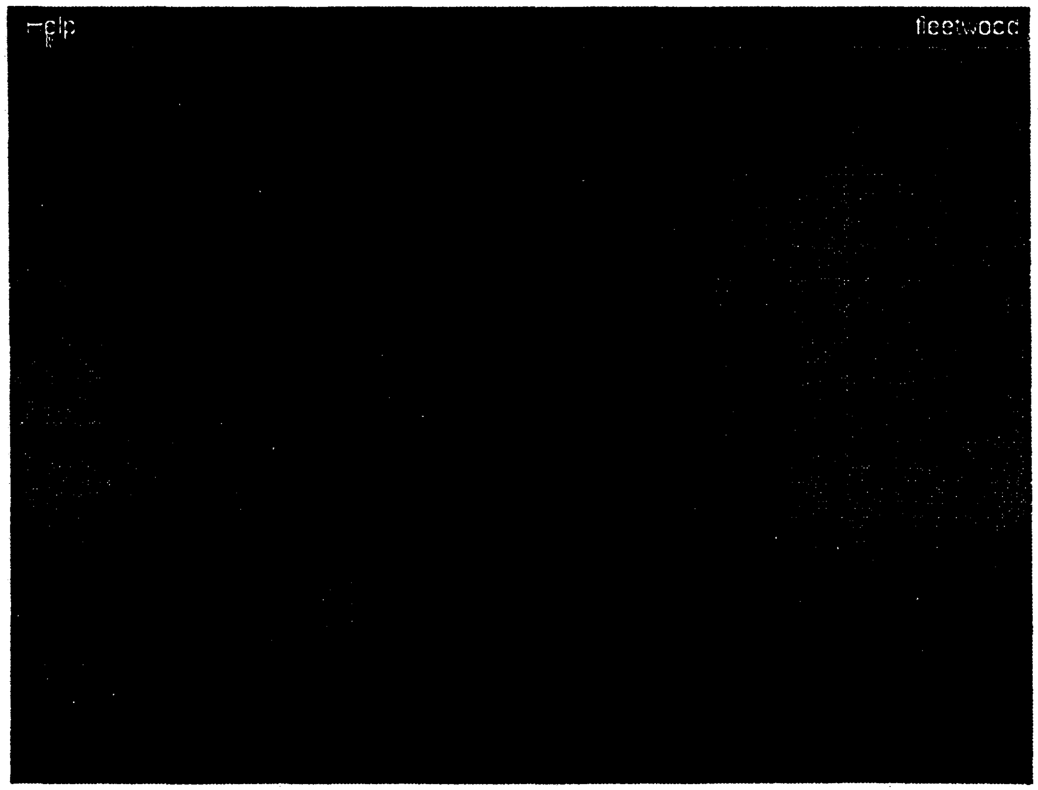
Response. The window shrinks to display only the title bar.

To unshrink a window:

Action. Move the mouse cursor into the title bar of the shrunken Help window and click the left mouse button.

Response. The window reappears on the screen.

You can use these methods to shrink and unshrink any window on the screen.



Moving Windows on the Screen

You can move a window to another place in the same column or to a different column.

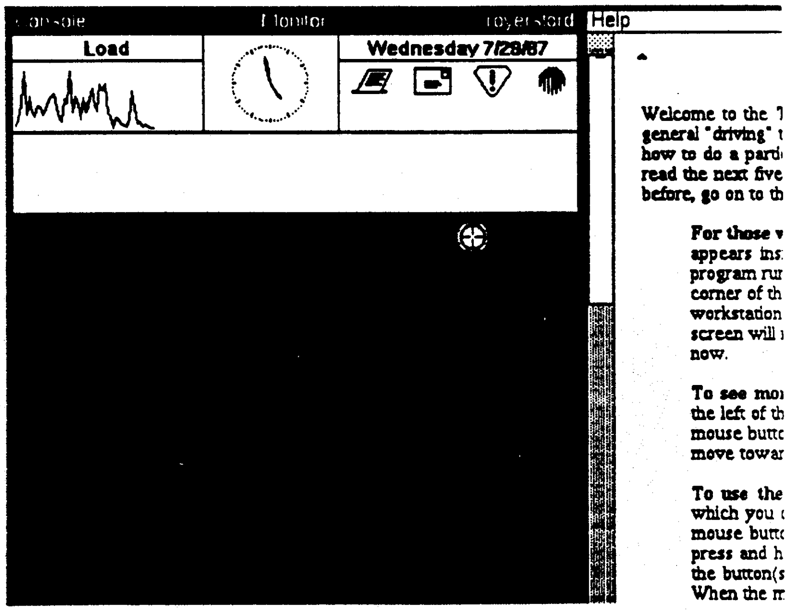
To move a window:

> **Action.** Move the mouse cursor into the title bar of the Typescript window and click the right mouse button.
>
> **Response.** The Typescript window disappears temporarily and the cursor turns into the target cursor, signifying that you are in the process of moving a window. If you click the right mouse button in the title bar of a window at the top of a column, any other windows in that column move toward the top of the column, taking up the space left by the missing window.
>
> **Action.** Move the target cursor into the bottom half of the Help window and click the right mouse button.
>
> **Response.** The Typescript window reappears in the bottom half of the right column.

Although the Typescript window becomes wider when it appears in the right column, nothing else about it changes. You can still do the same things with it as before.

> ***To move a window, move the mouse cursor into the title bar of the window that you want to move and click the right mouse button. Move the cursor to where you want the window to reappear and click the right mouse button again.***

Move the Typescript window back into the left column now so that the next exercises will be easier to complete.



Welcome to the 1
general "driving" t
how to do a parti
read the next five
before, go on to th

For those v
appears ins
program ru
corner of th
workstation
screen will
now.

To see mo
the left of th
mouse butt
move towa

To use the
which you
mouse butt
press and h
the button(s
When the m

Quitting from Windows

You can quit from a window by using the *pop-up menus*, which are explained in the next section. Most windows have a **Quit** menu item used for quitting.

Maximum Number of Windows

With Andrew you can work with more than one window at a time. However, there is sometimes a limit on the number of windows you can use. The limit varies depending on what kind of windows you have active. Usually 10 or 12 is windows is the most you can use at once.

4. Pop-up Menus

Using Menus

Menus are lists of the actions you can take in a given part of the screen, depending on where your mouse cursor is. Most menus look like stacks of cards, with similar actions grouped together on each *menu card*. You use menus on Andrew to do a wide variety of activities, from opening new windows to logging out. In this exercise, you should simply display and examine a set of pop-up menus. You will choose an item in the next exercise.

To make menus appear:

Action. Move your mouse cursor into the Help window, to the right of the scrollbar but not in the title bar. Press both mouse buttons at the same time and hold them. Continue to hold them until the end of this exercise.

Response. The pop-up menus for Help appear. Do not release the mouse buttons.

Action. Still holding down the mouse buttons, slide the cursor up and down the menus and menu items.

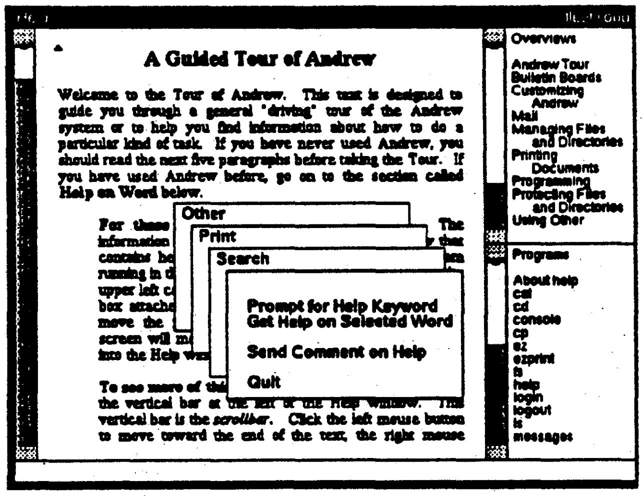
Response. As you move the cursor onto an option on a menu card, the option darkens. As you move up the stack of menus, the top menus move behind the other menus and become gray. As you move onto any portion of a hidden or gray menu, that menu moves to the top of the menu stack. Do not release the mouse buttons yet.

To make menus disappear:

Action. Still holding down the mouse buttons, move the cursor completely off the menus. Release the buttons.

Response. The pop-up menus for Help disappear.

If you choose a menu option by mistake, do not be alarmed. Simply follow the instructions in the next exercise to quit from the Help window.



A Guided Tour of Andrew

Welcome to the Tour of Andrew. This text is designed to guide you through a general "driving" tour of the Andrew system or to help you find information about how to do a particular kind of task. If you have never used Andrew, you should read the next five paragraphs before taking the Tour. If you have used Andrew before, go on to the section called Help on Word below.

For those ... The information ... that contains he... running in th... upper left c... box attache... move the ... screen will m... into the Help w...

To see more of thi... the vertical bar at the left of the Help window. This vertical bar is the *scrollbar*. Click the left mouse button to move toward the end of the text, the right mouse

Choosing a Menu Option

You choose a menu option by making the menus appear, moving the cursor to darken the option that you want, and releasing the mouse buttons. Please do the next exercise carefully so that you choose the Quit menu option and not another one.

To use the pop-up menus to quit from a window

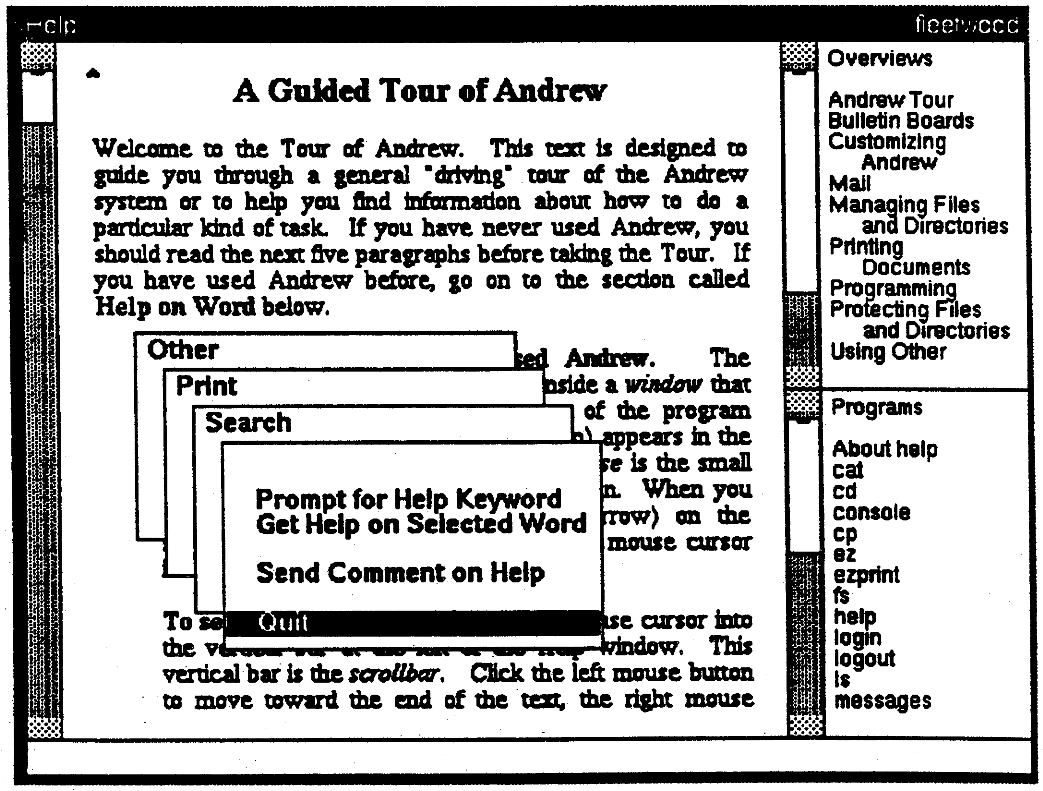
Action. Move the mouse cursor into the Help window if it is not there already, to the right of the scrollbar and into the text of the "Tour of Andrew" document. Press both mouse buttons to display the menus. Continue to hold down the buttons and slowly pull the mouse cursor straight down to point to the last menu option, Quit. When Quit becomes darkened, release the mouse buttons.

Response. After a moment, the Help window disappears.

Use this method to choose any menu option on any menu, not just the Help menus. You should find out what a menu option does before you choose it or you may get unexpected results. Read the next page to learn how to find out about items on different kinds of menus.

If 'Quit' does not close the window

If you chose a menu item by mistake in the last exercise, or if you chose something other than Quit during this exercise, you may have to choose Quit several times before the Help window will disappear. If the window does not appear the first time you choose Quit, try two more times. Wait twenty seconds between tries to give the program time to quit.



A Guided Tour of Andrew

Welcome to the Tour of Andrew. This text is designed to guide you through a general "driving" tour of the Andrew system or to help you find information about how to do a particular kind of task. If you have never used Andrew, you should read the next five paragraphs before taking the Tour. If you have used Andrew before, go on to the section called Help on Word below.

ed Andrew. The
nside a *window* that
of the program
) appears in the
se is the small
n. When you
rrow) on the
mouse cursor

To se use cursor into the v window. This vertical bar is the *scrollbar*. Click the left mouse button to move toward the end of the text, the right mouse

Kinds of Menus

While you are using Andrew, you may see three different kinds of menus depending on where the cursor is when you press both mouse buttons.

Program Menus

The menus that appear when you press both mouse buttons while the cursor is in the body or main part of a window are called *program menus* because they are specific to the program that is running in a given window. For example, you used the program menus in the Help window to quit from Help. The program menus in the Help window are different from the program menus in the Typescript window because each program has menu items that are appropriate for what that program does. One thing that all program menus have in common is a Quit menu option near the bottom of the first menu card. All the program menus for the programs covered in these first parts of the tutorial (Console, Typescript, and Help) are decribed in the section of the guide that explains that program.

Selected Region Menus

The **selected region menus** appear when you press both mouse buttons while you have a selected region in a window that has a scrollbar. You can learn how to create and use selected regions in the next section.

Title Bar Menus

When you press both mouse buttons while the cursor is in the title bar of a window, the *title bar menus* appear. The title bar menus also appear if you press both mouse buttons while the cursor is in the grey or background part of the screen. The title bar menus are the same wherever they appear; whenever you place the cursor in the title bar of any window or in the gray area and press both buttons, you see the same set of title bar menus. Because they always appear, you can always use them to create new windows if something happens to the windows that you have. The next exercises show you more about the title bar menus.

Getting a New Typescript from the Title Bar Menus

Occasionally windows disappear unexpectedly, either because you choose a menu option like **Quit** by mistake or because of a problem with the program that is running in a window. As you have seen, as long as you have a Typescript window, you can start any other window again either by typing a command in the Typescript window or by using the menus in Typescript. But what happens if your Typescript window disappears unexpectedly?

You can always open a new Typescript window from the title bar menus since they appear when the cursor is in the gray area.

Even if you close all your windows by mistake and the screen is gray, the title bar menus are still available. They contain an option called **New Typescript** that opens a new Typescript window.

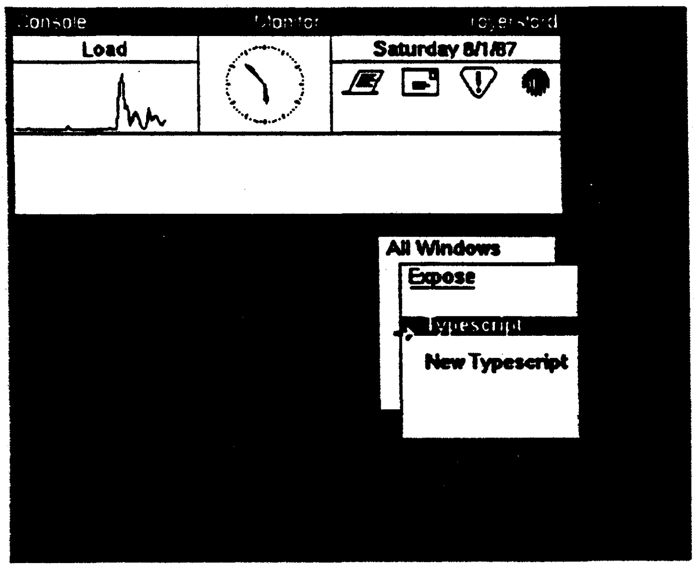
Action. Move the mouse cursor into the Typescript window. Press both mouse buttons to display the program menus for Typescript. Slowly pull the mouse cursor straight down to point to **Quit**. When **Quit** becomes darkened, release the mouse buttons.

Response. The Typescript window disappears.

Action. With the mouse cursor in the gray area, press both mouse buttons to display the title bar menus. Move the cursor back through the menus until the second or **Expose** menu is displayed. Move the mouse cursor down the **Expose** menu to point to the **New Typescript** option. (You may have to move the cursor along the left side of the menu to cause **New Typescript** to become darkened.) When **New Typescript** is darkened, release the mouse buttons.

Response. After a few moments, a new Typescript window appears.

You do not need to know what all the options on the title bar menus do in order to complete this exercise, but if you are curious, each of the items is explained the last section of this guide.



5. Ending an Andrew Session

You do not need to log out if you want to continue working on the exercises in this guide. However, if you would like to take a break, you can log out now and complete the next section later. Always log out when you are finished working with Andrew.

If this is the first time you have used Andrew, you should change your password before you continue.

Changing Your Password

The first time you log in, and every few months after that, you should change your password to something that only you know, so that other users cannot log into your account. A password can be any combination of letters or numbers, including upper and lower case. Choose a password that is at least six characters long, and one that you can remember easily.

To change your password:

Action. Type

```
passwd
```

and press Enter.

Response. Changing password for user. Old password:

Action. Type your current password and press Enter. (The password does not appear as you type it.)

Response. New password for user:

Action. Type your new password and press Enter. (The password does not appear as you type it.)

Response. Retype new password:

Action. Type your new password and press Enter. (The password does not appear as you type it.)

Response. Password changed, it will be in effect in about 1 hour.

Logging Out

There are two ways to log out, depending on what is the last menu option on the last menu card of the title bar menus, the **All Windows card**. Your **All Windows** menu card will probably have **Logout** as its last menu item, but look carefully to be sure.

Quitting from any windows other than Console and Typescript before you log out is a good habit. If you have been editing a document or working with your mail, quitting from each window insures that you will not lose any changes you have been making. If you are going to log out now, quit from the Help window first for practice. See page 18 if you do not remember how to quit from a window.

How to Log Out

If your 'All Windows' menu says Logout:

Action. Move the mouse cursor to point to Logout. When Logout becomes darkened, release the mouse buttons.

Response. All windows disappear from your screen and the 'login:' prompt appears. You are logged out.

If your 'All Windows' menus says Zap:

Action. Move the mouse cursor to point to Zap. When Zap becomes darkened, release the mouse buttons.

Response. All windows disappear from your screen. After a moment, a percent prompt appears at the top of the screen.

Action. At the percent prompt, type

logout

and Press Enter.

Response. The 'login:' prompt appears. You are logged out.

If you are not taking a break now, continue reading.

6. The Scrollbar

As you have seen, some Andrew windows have a scrollbar along the left side for moving backward and forward through the contents of the window. The Help window is a good example of a window with a scrollbar and you can use it to learn more about how a scrollbar works.

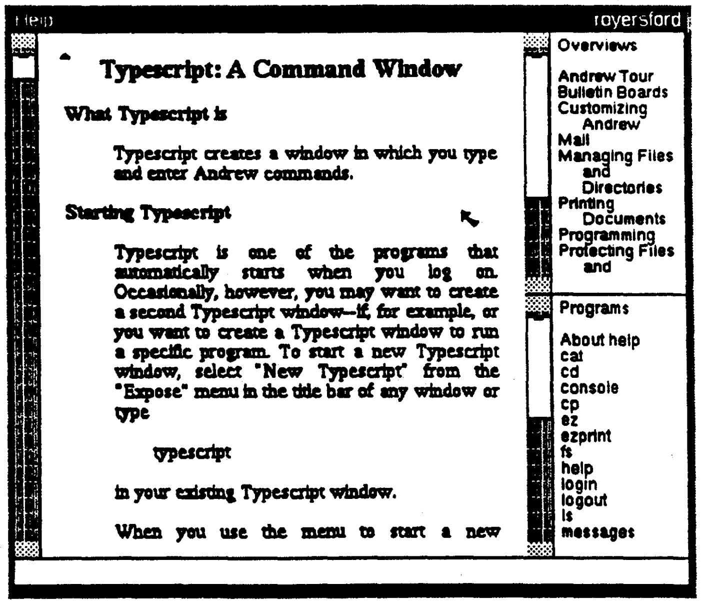
If you chose to take a break before this section, log back in. (See page 4 if you do not remember how to log in.) When the Console and Typescript windows appear, move the mouse cursor into the title bar of the Typescript window and press both mouse buttons to expose the title bar menus. Choose **Help** from the front menu. (See page 18 if you need more information about how to choose a menu option.)

Knowing When to Use the Scrollbar

The help document about Typescript, which is displayed in the Help window, is too long to be completely visible in the window. If you could see the entire document in the window, you would not need to use the scrollbar. How can you know that there is more of a document than you are seeing?

You can tell that a document is longer than the section that appears in the window by looking at the size of the white box in the scrollbar. The smaller the white box, the larger the document.

The scrollbar represents the entire document. The white box represents the part of the document that is currently displayed in the window. In a larger document, only a small percentage of the document can be displayed in the window at any one time. In a very small document, the entire document might be displayed in the window, which means that the white box would take up almost all of the scrollbar and using the scrollbar would have no effect because there would be no more of the document to view.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window—if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

Using the Scrollbar

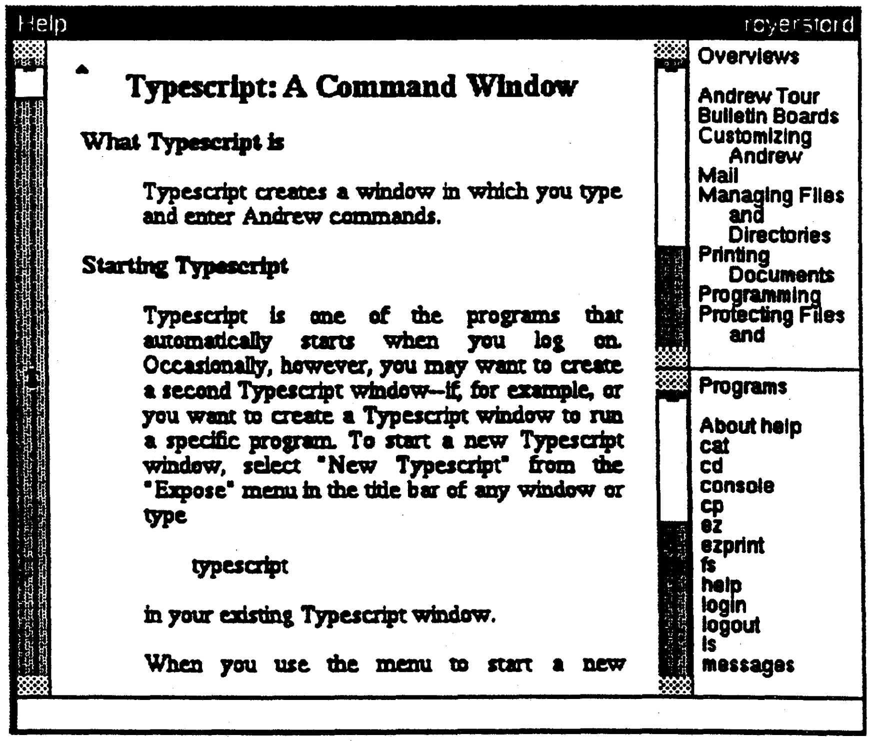
To see the rest of a large document, you can *scroll* through it. Scrolling allows you to move through a document until you have read it all, then go back to an earlier section if needed.

To see more of a document:

Action. Move the mouse cursor into the largest scrollbar of the Help window (the scrollbar on the left). Click the left mouse button.

Response. When the mouse cursor enters the scroll bar, the double arrow cursor appears. As you click the left mouse button, the text of the Help document moves past on your screen. Note that you are moving forward through the document, toward the end.

To scroll forward through your document, move the mouse cursor into the scrollbar and click the right mouse button. To scroll backward, click the right mouse button when the mouse cursor is in the scrollbar.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

Controlling How Far You Move with the Scrollbar

A good general rule for moving with the scrollbar is

The closer the double arrow cursor is to the bottom of the scrollbar (but not in the dotted area at the very bottom), the further you move in the text with each click.

(The dotted areas at the top and bottom of the scrollbar are called the *end zones*. You will use them in the next exercise.) When you click the left mouse button anywhere in the scrollbar except the end zones, the line of the document that is beside the double arrow cursor moves to the top of the window. If you click the right mouse button anywhere in the scrollbar except the end zones, the line that is at the top of the window moves to be beside the double arrow cursor.

Action: Move the double arrow cursor to the bottom of the left scrollbar in the Help window, just above the end zone. Click the left mouse button.

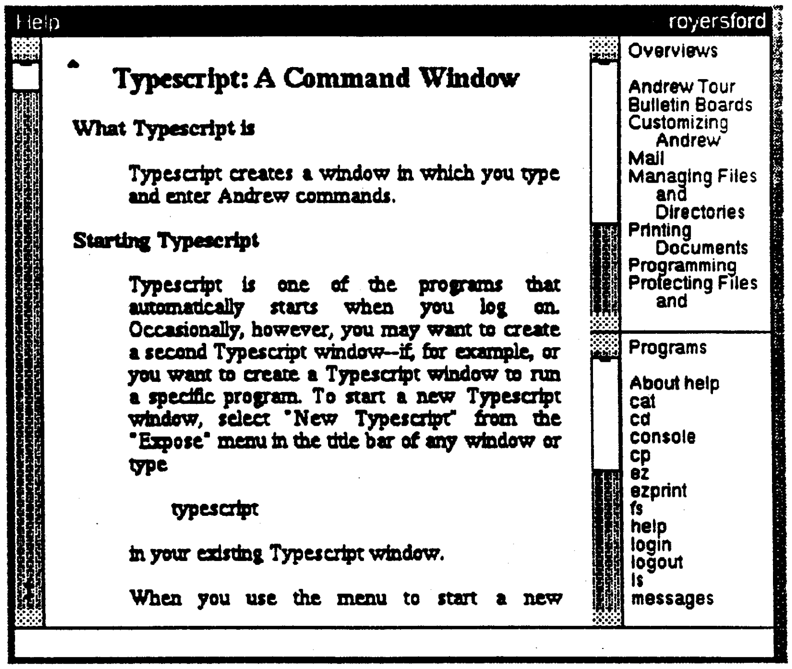
Response: The line that was beside the double arrow cursor moves to the top of the window.

Action: Leave the double arrow cursor at the bottom of the left scrollbar and click the right mouse button.

Response: The line that was at the top of the window moves to be beside the double arrow cursor.

If you click when the cursor is in the end zone by mistake, just move the cursor out of the end zone and try again. To move a shorter distance with each click, move the mouse cursor closer to the top of the scrollbar. Practice a few times until you feel comfortable with clicking in the scrollbar.

You can use the actions explained here to scroll through a document in any window that has a scrollbar, not just a help document.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

More on Controlling How Far You Move

The dotted areas at the top and bottom of the scrollbar or *end zones* represent the beginning and end of the document. Clicking the left mouse button while the double arrow is in the end zone at the bottom makes the end of the document visible; clicking the left mouse button while the mouse cursor is in the end zone at the top of the scrollbar makes the beginning of the document visibile in the window.

To move to the end of a document:

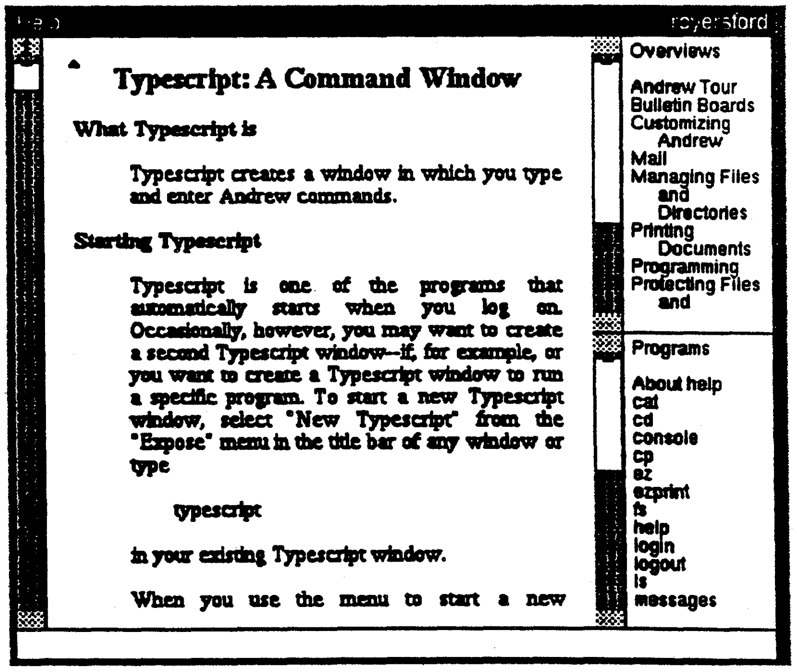
Action. Move the mouse cursor into the dotted area at the bottom of the left scrollbar in the Help window. Click the left mouse button.

Response. The end of the help document about Typescript appears in the Help window.

To move to the beginning of a document:

Action. Move the mouse cursor into the end zone at the top of the scrollbar. Click the left mouse button.

Response. The beginning of the help document about Typescript appears in the Help window.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

7. Selected Regions

Currently, there are two kinds of selected regions on Andrew: one marked by the *text caret* and one marked by a large reverse video box in windows that contain documents. In the future, there may be more kinds of selected regions (such as a selected cell in a spreadsheet, for example). In the next exercises, you can learn how to create and use the selected regions that are available now.

Moving the Text Caret

Many windows that have a scrollbar also have a *text caret*, shaped like small pointer. The text caret has two main purposes:

1. In some windows, such as the Typescript window, the text caret marks the place where the characters that you type appear. (You cannot type into all windows that have scrollbars. Some windows will not allow you to change the documents that they present.)

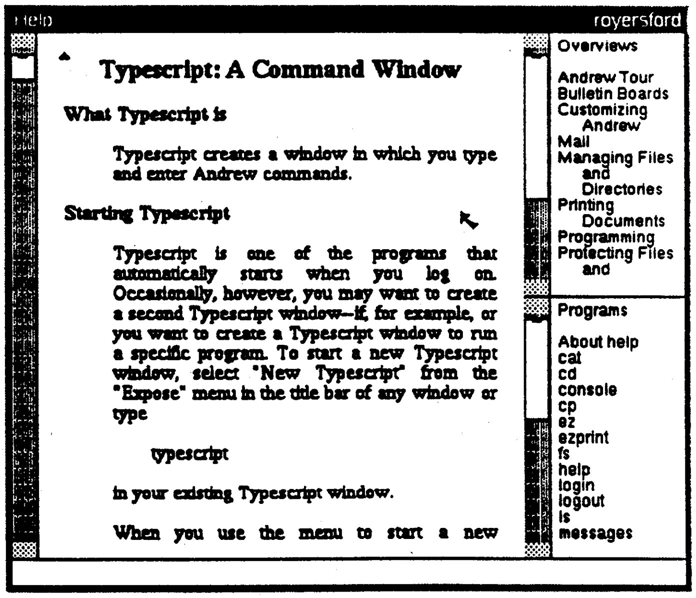
2. In any window with a scrollbar, the text caret marks your current position in a document so that you can create *selected regions* that you can use, for example, to make a copy of part of a document. You can learn more about selected regions in the next few exercises.

To move the text caret:

Action. Move the mouse cursor into the Help window and point the mouse cursor at the exact spot where you want the text caret to appear. Click the left mouse button.

Response. The text caret appears where you pointed the mouse cursor and clicked.

Do this to move the text caret in any window with a scroll bar, not just a Help window.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

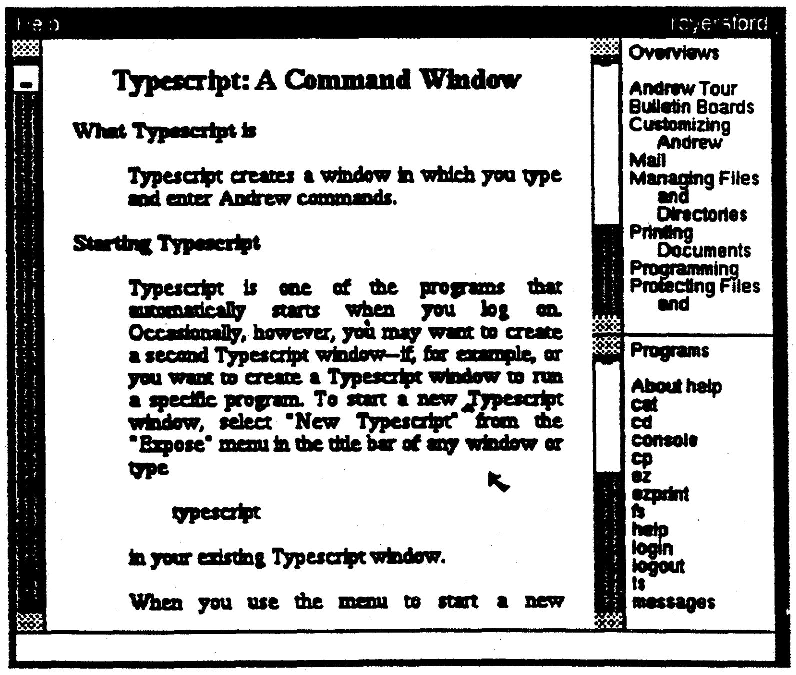
in your existing Typescript window.

When you use the menu to start a new

The Text Caret and the Scrollbar

When you use the scrollbar to bring another part of the document into view, the text caret remains where it was before you moved. The black box inside the scrollbar indicates the location of the text caret in the document. If the text caret is visible in the window, the black box in the scrollbar will appear inside the white box. (Remember that the white box stands for that portion of the document that is visible in the window.) If the text caret is not visible in the window, the black box will appear in a different part of the scrollbar than the white box. (In very long documents, sometimes the black box appears inside the white box even if the text caret is not visible in the window. The white box has a minimum size, so in a long document it may not reflect the position of the text caret in every case.)

Moving the text caret to the part of the document that is visible in the window is easy; just point the mouse cursor at the exact spot where you want the text caret to appear and click the left mouse button, as you did in the previous exercise. However, suppose you do not want to move the text caret, but to look at the portion of the document that contains the text caret. One way of doing that would be to scroll the document (by clicking in the scrollbar) until you come to the portion that contains the text caret. However, there is a faster way, illustrated in the exercise on the next page.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

Moving to the Location of the Text Caret

To make the portion of the document that contains the text caret visible in the window:

Action: Move the mouse cursor into the scrollbar in the Help window and click the left mouse button once or twice.

Response: The white box in the scrollbar moves below the black box that marks the position of the text caret.

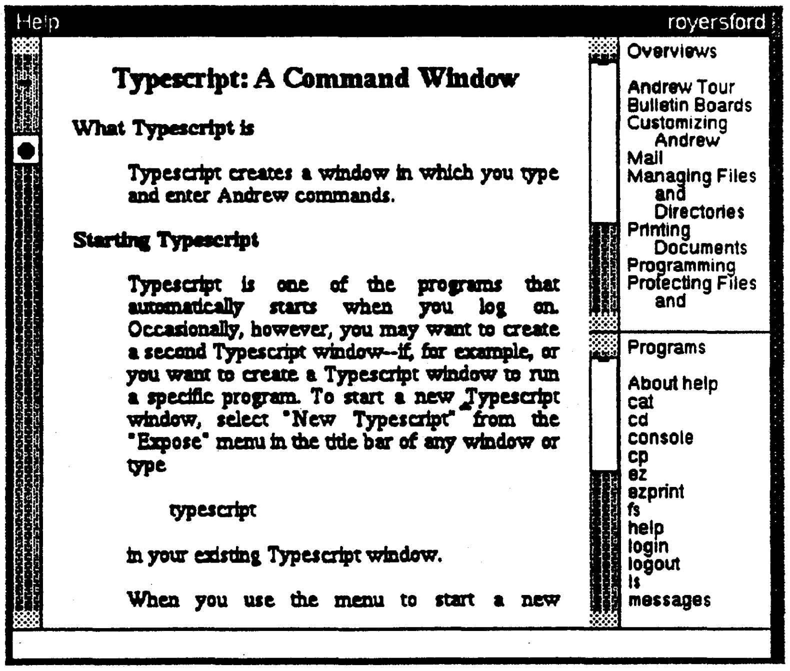
Action: Position the double arrow cursor inside the white box in the scrollbar. Press the left mouse button and hold it down (don't click). Move the mouse to slide the white box until the black box is inside the white box.

Response: The cursor becomes a black circle.

Action: Release the left mouse button.

Response: The portion of the document that contains the text caret becomes visible in the window.

Moving through the document with the scrollbar does not automatically move the text caret. You have to click the left mouse button where you want the text caret to appear.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

Selecting Text

When you want to take action on part of a document, such as making a copy of it, you must first *select* the text by creating a *selected region* around it.

To select text:

Action. Move the mouse cursor into the Help window, point the mouse cursor at the beginning of the text you want to select and click the left mouse button.

Response. The text caret appears at the place where you clicked.

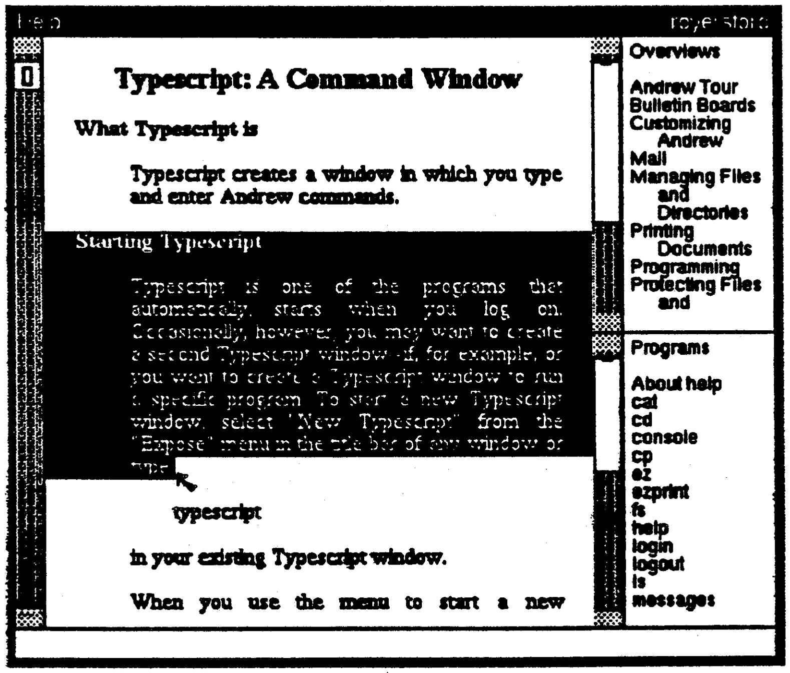
Action. Point the mouse cursor at the end of the text you want to select and click the right mouse button.

Response. The text is surrounded by a box shown in reverse video (white letters on a black background). The region enclosed by the box is called a *selected region.*

You can do this to select text in any window with a scroll bar, not just a Help window. Remember, click the left button to mark the beginning of the selected region, the right button to end the selected region.

Selected Regions and the Scrollbar

As you saw earlier, the position of the text caret in a document is shown in the scrollbar by a black box. When you have a selected region in a document, the black box in the scrollbar is replaced by a lined box that represents the position of the selected region. You cannot have a selected region and a text caret visible in the same document at the same time; if you click the left mouse button to position the text caret while you have a selected region, the region is deselected and the text caret appears where you clicked.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

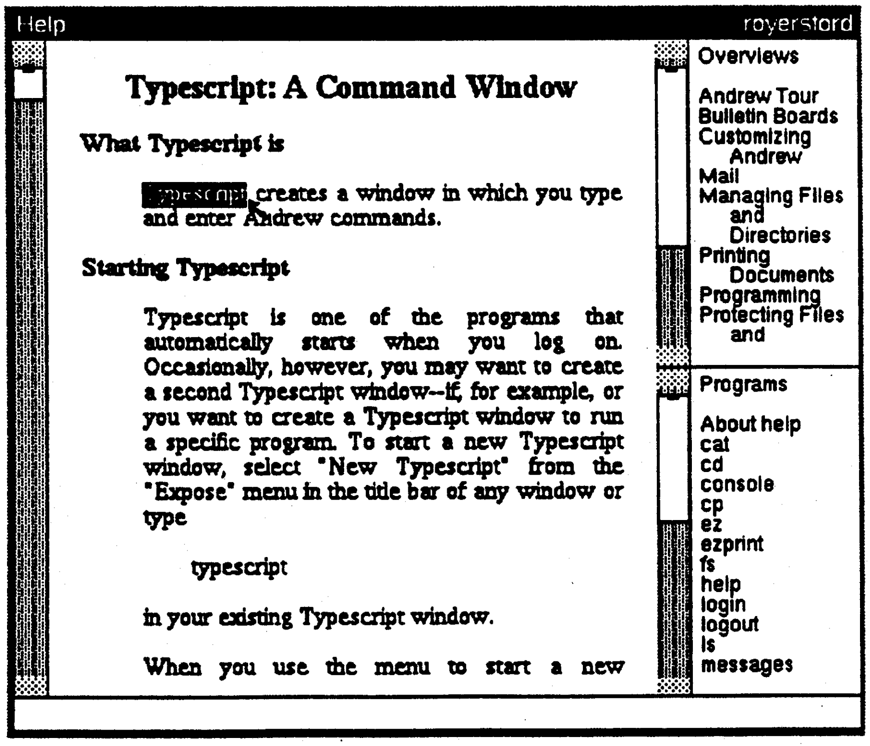
8. Copying and Pasting

You can select a region in any window with a scrollbar and make a *copy* of it to *paste* into another location in the same window or into a different windows. Although you can copy a region from many windows that have a scrollbar, you cannot paste in every window. For example, you cannot change the documents that appear in the Help window, so you cannot paste into the Help window. However, you can always copy a region from the Help window and paste it into the Typescript window, so that you can use commands that you learn about in Help without having to retype them.

To copy a region from the Help window:

Action. Move the mouse cursor into the Help window and select the word **Typescript** (with a Capital "t" as shown in the illustration below). Press both mouse buttons to display the program menus for Help. Choose **Copy.**

Response. Since you are only making a copy of the selected region and not *cutting* or removing it from the window, no change to the window occurs. You cannot tell whether you have successfully copied a region until you try to paste it somewhere.



Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

Pasting a Selected Region

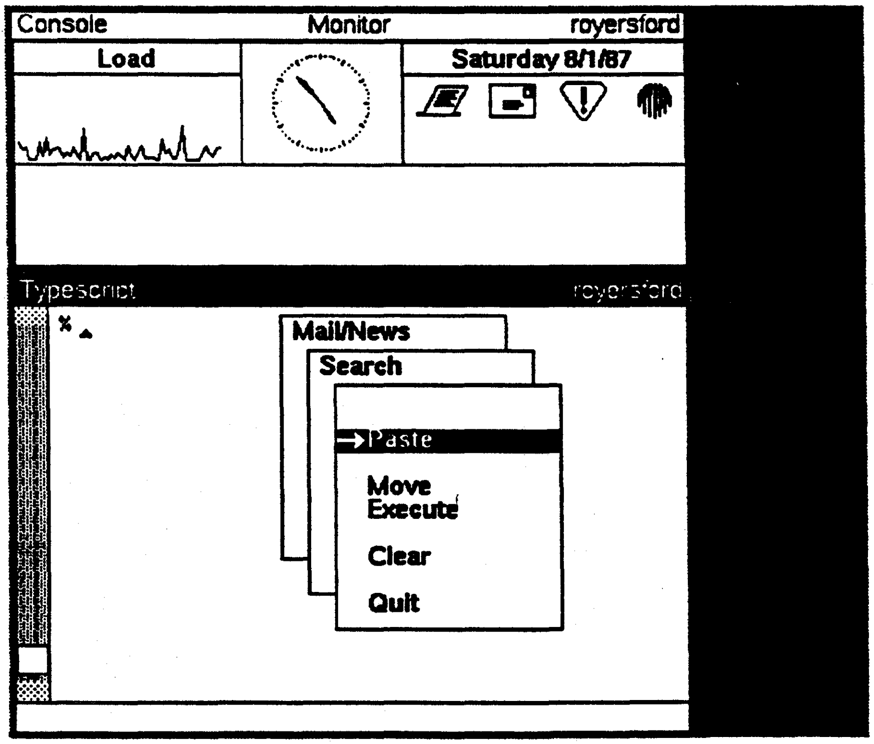
To paste a region into the Typescript window:

Action. Move the mouse cursor into the Typescript window. Press both mouse buttons to display the program menus for Typescript. Choose **Paste**.

Response. The word "Typescript" that you copied from the Help window appears at the percent prompt in the Typescript window.

Don't be alarmed if something other than "Typescript" appears when you choose **Paste**. Use the Backspace key on the keyboard to delete the letters that appear in the Typescript window and try the exercise on the previous page again until the word "Typescript" appears at the percent prompt.

You can use this procedure to copy a region from any window with a scrollbar, not just a Help window, and to paste into any window that you can type characters into.

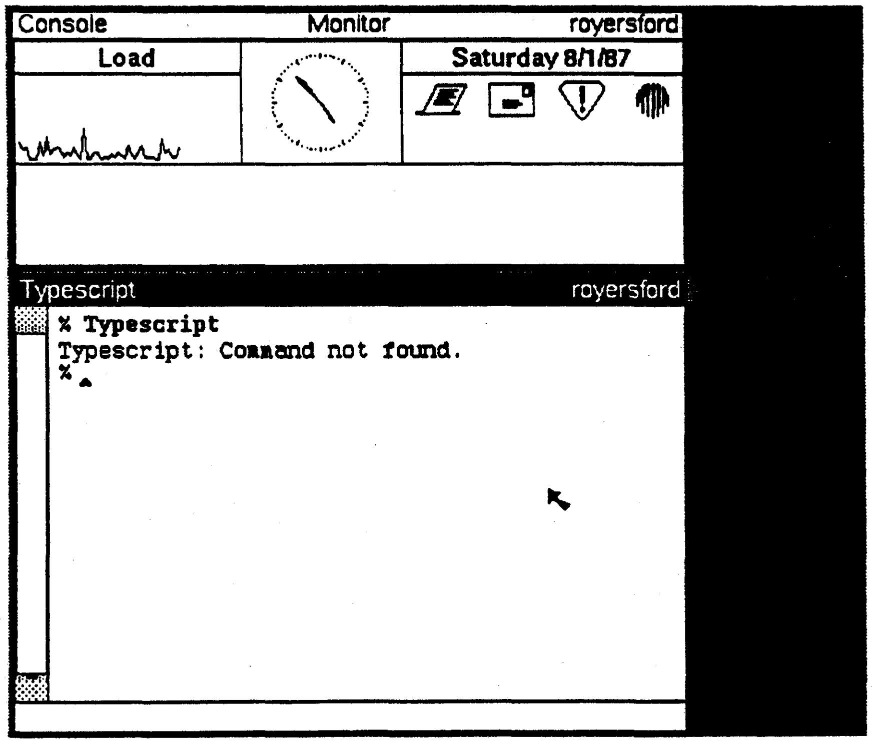


Using Copied Regions as Commands in Typescript

As you are reading help documents, you may come across a command or an example that answers a question that you had. For example, suppose you were wondering how to start a second Typescript window. The Help document about Typescript says that you can type the word "typescript" into your existing Typescript window to start a second Typescript. One way to use commands or examples that you find in the Help window is to copy them from the Help window, paste them into the Typescript window, and press Enter.

> **Action.** Having copied the word "Typescript" from the Help window in the last exercise and pasted it to the percent prompt in the Typescript window, press Enter.
>
> **Response.** The words "Typescript: Command not found" appear in the Typescript window.

The response "Typescript: Command not found" is an *error message* notifying you that Typescript was unable to carry out the command that you gave. The first word, "Typescript," is the command that you gave. The phrase after the colon (:) is the reason that Typescript could not complete the command. In this case, Typescript does not recognize the command "Typescript" as a valid command. The reason is explained in the next exercise.



```
% Typescript
Typescript: Command not found.
%
```

Small and Capital Letters in the Typescript Window

You may wonder why, since you completed the last exercise correctly, you got an error message in the Typescript window instead of having a new Typescript window appear. The answer is that the Typescript window is *case-sensitive* or treats commands that have capital (or *upper case*) letters in them differently from commands that have only small (or *lower case*) letters. If you copy the word "typescript" (with a lower-case "t") from the Help window and paste it into the Typescript window, you can use it as a command successfully.

Action. Move the mouse cursor into the Help window and select the word "typescript" (with a lower-case "t"). Press both mouse buttons to display the program menus for Help and choose Copy. Move the mouse cursor into the Typescript window and choose Paste.

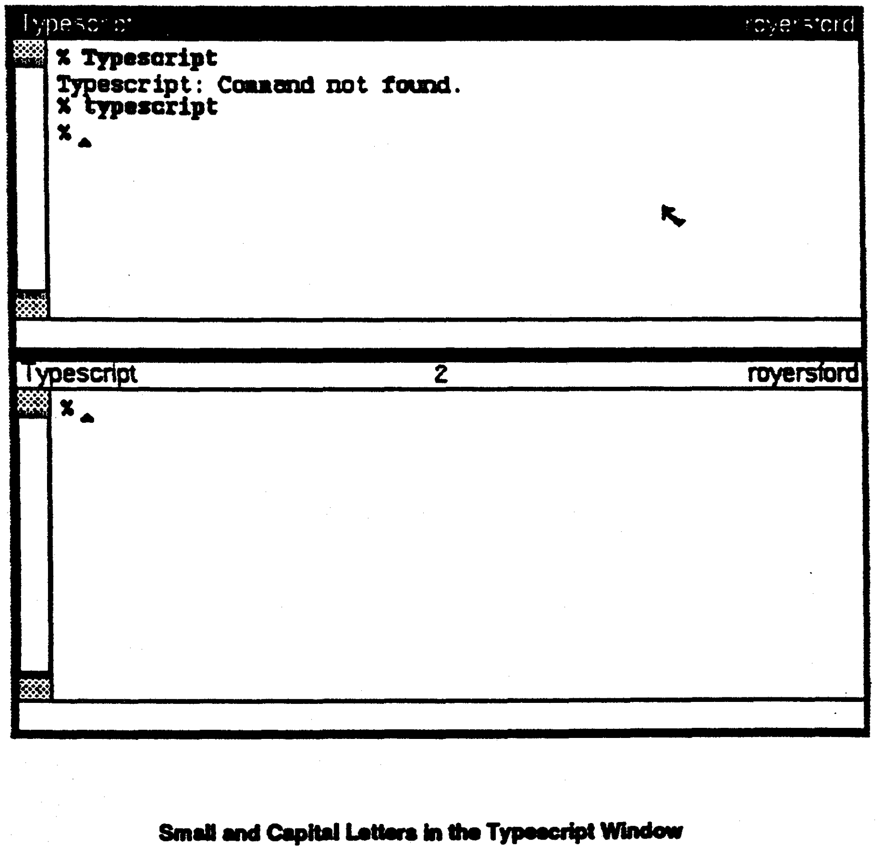
Response. The word "typescript" appears at the percent prompt.

Action. Press Enter.

Response. After a few moments, a second Typescript window appears in the left column.

The Typescript window is case-sensitive, so if you see the error message "Command not found," check the command for extra capital letters. The same error may occur if you misspell a word that you typed.

You do not need to quit from the second Typescript window in order to complete the next exercises, which are about the Help window.



```
% Typescript
Typescript: Command not found.
% typescript
%
```

```
%
```

9. The Help System

You can find out more about any Andrew program or about common tasks by reading help documents.

Using the Panels in the Help Window

To help you find the documents of interest to you, the Help window contains two panels at the far right. The top panel is labeled Overviews and contains a list of help documents about how to do specific tasks on Andrew. The first document in the top panel, the Andrew Tour, is a general introduction to Andrew. It contains a list of all the other Overview files and a summary of what is in each one.

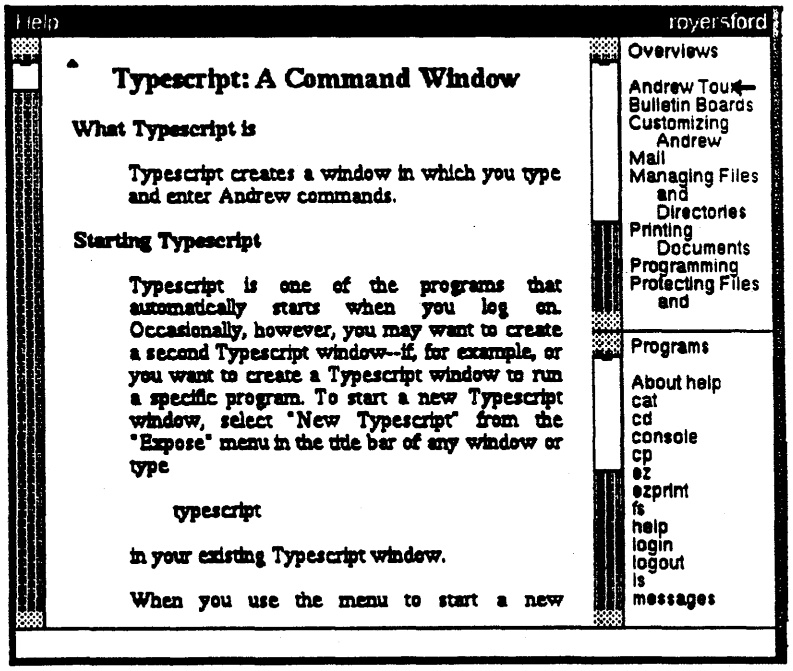
To see the Overview document called the Andrew Tour:

> **Action.** Move the mouse cursor until it points to the words **Andrew Tour** in the Overviews panel and click the left mouse button.
>
> **Response.** After a few seconds, the Andrew Tour document appears.

The Programs panel in Help contains a short list of commonly-used Andrew programs. It works like the Overviews panel. Try it if you wish.

To use the panels in Help, move the mouse cursor to point to the item of interest to you and click the left mouse button.

You do not need to read the any of the help documents in order to continue these exercises. You may wish to return to them later, however. The next exercise explains more about the Help program.



Help royersford

Typescript: A Command Window

What Typescript is

Typescript creates a window in which you type and enter Andrew commands.

Starting Typescript

Typescript is one of the programs that automatically starts when you log on. Occasionally, however, you may want to create a second Typescript window--if, for example, or you want to create a Typescript window to run a specific program. To start a new Typescript window, select "New Typescript" from the "Expose" menu in the title bar of any window or type

typescript

in your existing Typescript window.

When you use the menu to start a new

Overviews

Andrew Tour
Bulletin Boards
Customizing Andrew
Mail
Managing Files and Directories
Printing Documents
Programming
Protecting Files and

Programs

About help
cat
cd
console
cp
ez
ezprint
fs
help
login
logout
ls
messages

Getting Help on a Particular Topic

You may wish to use the Help program when you want to know more about a particular program or feature of Andrew. You can use the pop-up menus in the Help window to request the help document about a particular *keyword* if you know which document is of interest to you.

To get help about a keyword:

Action. Move the mouse cursor into the Help window and press both mouse buttons to display the program menus for Help. Choose **Prompt for Help Keyword.**

Response. A *prompt* appears in the message area at the bottom of the window and the text caret moves to the end of the prompt:

Enter a keyword about which you want more help:

Action. Type the word

help

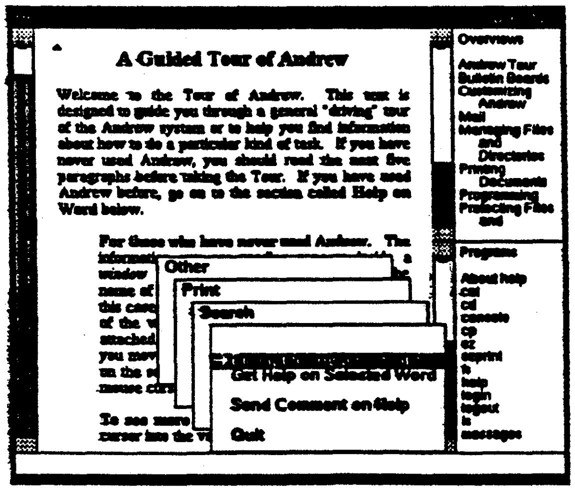
and press Enter.

Response. After a few moments, the help document about the Help program appears.

The Help program has many other useful features and you can learn more about them by reading the help document about Help. You do not need to read it in order to continue doing the exercises in this guide, but you may wish to come back to it later.

If you wish to take a break now, you can complete the next section later. Use **Logout** from the *All Windows* menu card in the title bar menus to log out.

If you do not wish to take a break now, log out and log back in to reset your screen so that only a Console and a Typescript window appear. See page 24 if you need more information about how to log out.


A Guided Tour of Andrew

Welcome to the Tour of Andrew. This text is designed to guide you through a general "driving" tour of the Andrew system or to help you find information about how to do a particular kind of task. If you have never used Andrew, you should read the next five paragraphs before taking the Tour. If you have used Andrew before, go on to the section called Help on Word below.

For those who have never used Andrew. The informati[illegible] a window [illegible] name of [illegible] this case [illegible] of the w[illegible] attached [illegible] you mov[illegible] on the s[illegible] mouse cur[illegible]

To see more [illegible] cursor into the w[illegible]

10. Writing and Printing

About EZ, the Andrew Editor

EZ is an editing program or *editor* designed especially for Andrew. Most people think of an editing program as a *text editor* or a tool that they can use to create, save, format, and print documents or *texts*. At present, EZ is a text editor and you can use it to do all the things listed above. However, EZ is still being developed and extended. In the future, EZ will become a general-purpose editor from which you can edit line drawings, pictures, and even animations in addition to editing documents.

For now, however, EZ is primarily a text editor. The exercises in this section are about creating, changing, and printing documents using EZ. Whenever you work with a document in EZ and you *save* it, that document is stored in the computer as a *file*. A file is an electronic version of your document that stays in the computer so that you can use the document again later.

Note for people who have used text editors before: This tutorial describes how to work with a single file in a single buffer using menu options to format your text. You can also work with multiple buffers in EZ and use keyboard commands to do most operations. To find out more about these facilities in EZ, use the Help program later to learn more about EZ.

The editor called "Emacs" is also available on Andrew. You do not need to know about Emacs in order to complete these exercises, but if you are curious, you can use the Help program later to learn more about Emacs.

(If you chose to take a break before this section, all you need to do now is log back in.)

Creating a Document

Whenever you edit a document on Andrew, you should give EZ a name for the document. EZ will store the document into a file and give the file the name that you supply. In general, to start a session with EZ, you type ez in the Typescript followed by a space and the name of the file you want to edit, then press Enter.

To create a new document:

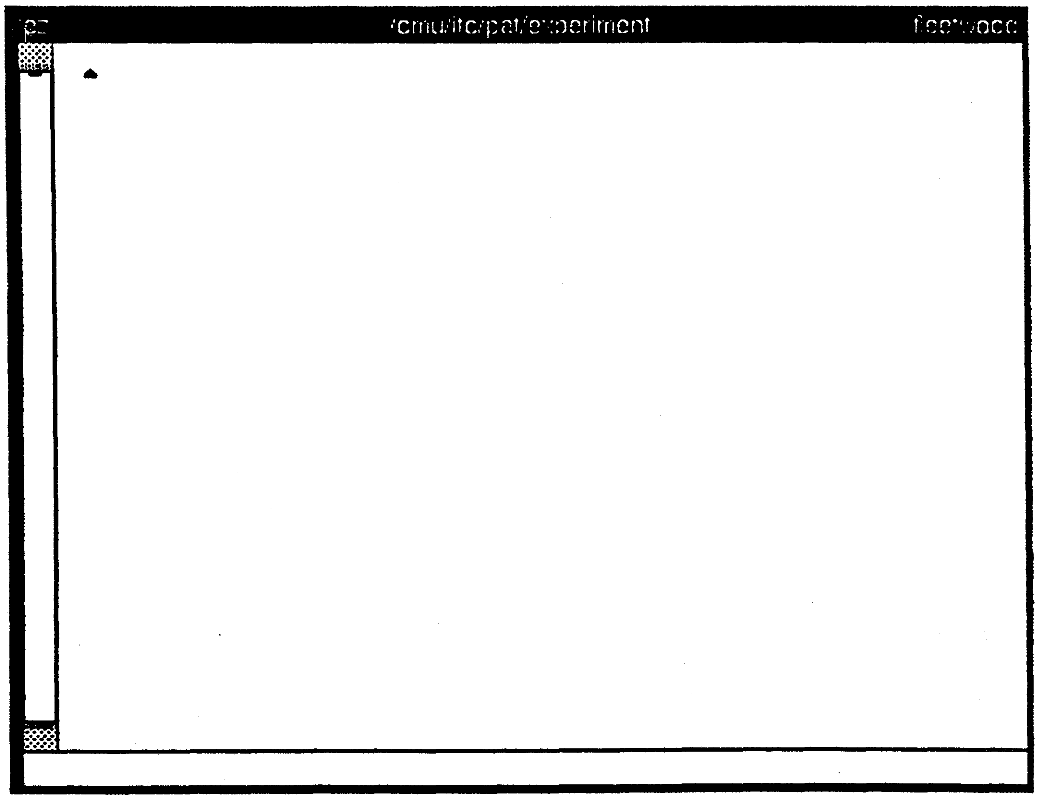
Action. For this exercise, type

ez experiment

and press Enter. (Be sure to type "ez" in lower-case letters.)

Response. A new EZ window appears in the right column of your screen. Since you are creating a new document, the window is empty. If you were editing a document in an existing file, that document would appear in the window.

The full *pathname* to the document that you are editing, "experiment," appears in the title bar of the EZ window. You can learn more about pathnames in the next section.



Typing into a Document

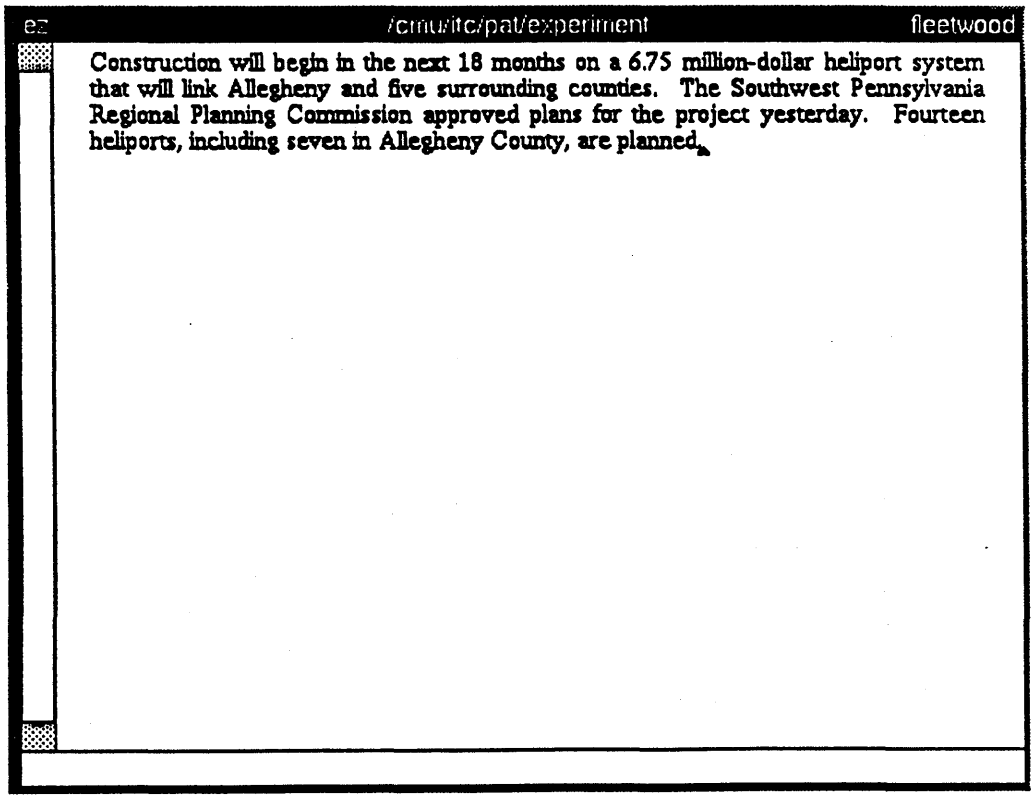
To type into the document:

Action. Move the mouse cursor into the EZ window. Type the following three sentences. (You should not press Enter to make the lines look exactly like these.)

Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Fourteen heliports, including seven in Allegheny County, are planned.

Response. The text caret moves to the right as you type, and the letters you type appear in the EZ window.

When you type enough characters to fill a line on the screen, you do not need to press Enter to start a new line because EZ automatically *wraps* the text. In other words, EZ automatically moves the text caret to the next line so that you can continue typing. If you press Enter to start a new line instead of letting EZ wrap the lines, your text may be oddly formatted when you print it. You only need to press Enter when you want to start a new paragraph or create blank lines.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Fourteen heliports, including seven in Allegheny County, are planned.

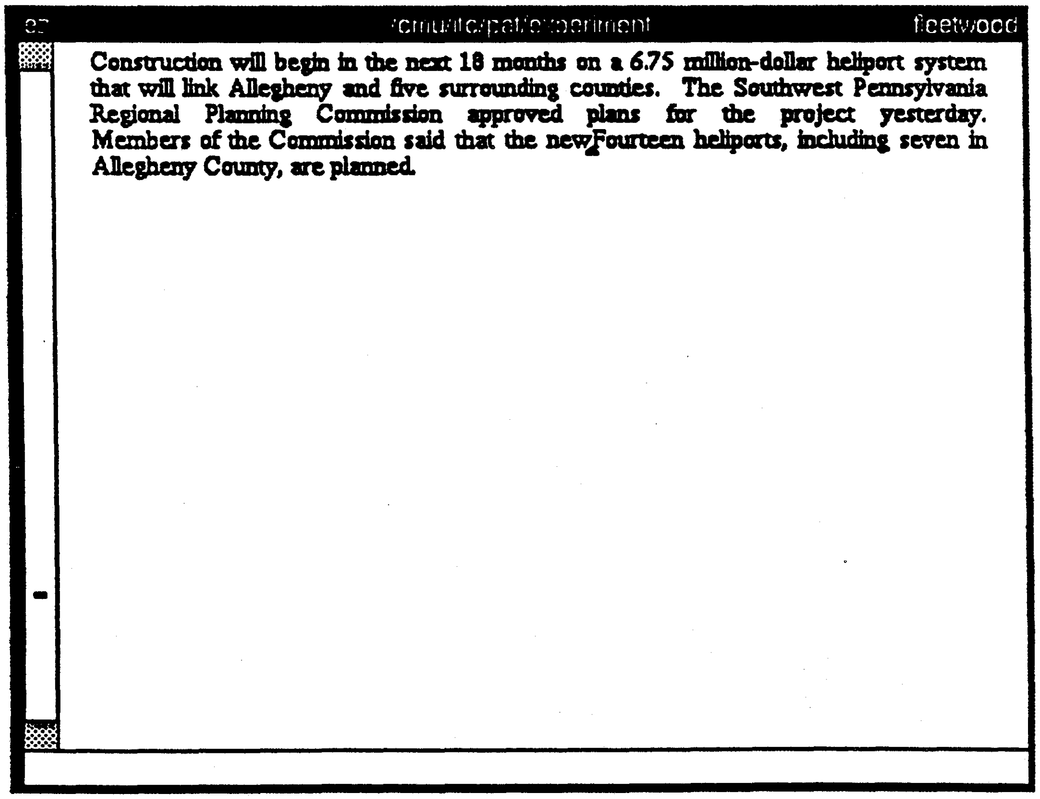
Changing a Document

Changing a document is typically a three-step process. First, move the text caret to the part of the document that you want to change. Then, make whatever changes you wish by inserting, deleting, pasting, or moving text. Finally, save your changes so that your newly-changed version of the document will be stored into a file.

To insert additional text into a document:

> **Action.** Move the text caret to the beginning of the sentence that begins with the word "Fourteen." Type in the following additional text:
>
> **Members of the Commission said that the new heliports represent an exciting expansion of air travel services in southwestern Pennsylvania.**
>
> **Response.** The letters that you type appear just before the text caret.

The illustration below shows what the letters look like when someone is inserting text in EZ. Note that the inserted letters appear just before the text caret and cause the letters after the caret to move to the right and be wrapped to the next line.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Members of the Commission said that the newFourteen heliports, including seven in Allegheny County, are planned.

Saving Your Work

You should save your work frequently when you are making changes to a document with any text editor, including EZ. When you do a save, EZ writes the changes that you have been making to the document into a file. EZ gives the file the name that you supplied when you started your EZ session. Saving the file insures that when you quit from EZ, you will not lose the work that you have been doing. Frequent saves also protect you from losing much work if you make a mistake, such as choosing the wrong menu item, while editing a file.

To save your work:

Action. With the mouse cursor in the EZ window, press both mouse buttons to expose the program menus for EZ. Choose **Save** from the front menu. (Do not be confused if you move the mouse onto one of the other menu cards and see the menu options "Save as" and "Save all." Those options are different from "Save." Move the mouse cursor back onto the front card and choose "Save.")

Response. The words "Wrote file." appear in the message area at the bottom of the window, indicating that your document has been written into a file. The full *pathname* to the file appears in the "Wrote file"message, too.

If you choose another menu option by mistake, do not be alarmed. Some of the options have little effect on the screen, such as the option to send your file to be printed. You can just try again to choose Save unless a new window appears or a prompt or question appears at the bottom of the EZ window. If another window appears, move the cursor into the second window, press both mouse buttons to expose the program menus, and choose Quit. Then move the mouse cursor back into the EZ window and try again to choose Save from the EZ program menus. If a prompt appears at the bottom of the window, press the Ctrl and g keys on the keyboard at the same time to cancel the prompt. The word "Cancelled" or "Punt" will appear in place of the prompt at the bottom of the window and you can try again to choose Save from the EZ menus.

If a save fails:

Occasionally a temporary interruption on the network can keep you from being able to save your file. If that happens, an error message appears in the message area at the bottom of the EZ window informing you that the effort to *write* the document into a file has failed. In most cases, your best bet is simply to stop working on the document, wait a few minutes, and try again to save the file. Most network interruptions are brief (10-15 minutes) and you are able to do a save and continue working afterward without any damage to the file. If you are unable to save your file for more than 20 minutes, there may be a more serious problem. Ask a user consultant to help you.

Automatic Saves, or Checkpoint Files

After you have made changes to a file once, Andrew begins to save your work automatically in checkpoint files. You can tell when your document is being automatically saved because the clockface cursor may appear on the screen in the place of the arrow cursor. At those moments, anything that you type will not appear in the window until after the clockface disappears. In addition, a message appears in the message area at the bottom of the window.

A checkpoint file has the same name as the file in which your document is stored except that the checkpoint file has an *extension* of .CKP. An extension is the part of a file's name that appears after a period or dot. For the file that you have been editing, for example, EZ has been creating a checkpoint file called "experiment.CKP" every two minutes while you work. Every two minutes, the latest version of your document, including all your most recent changes, gets copied into the checkpoint file. When you save the document, the checkpoint file is deleted. If you continue editing after you do a save, a new checkpoint file is created in two minutes.

The purpose of a checkpoint file is to provide a backup copy of your document and minimize the amount of work you might lose if for some reason you quit from EZ without saving your changes (if the EZ window disappears unexpectedly, for example). If something unexpected happens while you are editing a document, you can find the ".CKP" file to retrieve a relatively recent version of the document. In the next section you can learn more about how to locate files such as ".CKP" files in your directory.

Cutting and Pasting Text in EZ Documents

As you saw in an earlier exercise, you can use the **Copy** menu option to make a copy of text from any scrollable window and **Paste** to paste the copy into another window. (Remember that any window you can type into, you can also paste into.) You can copy from most windows that have scrollbars, but some windows, such as the Help window, do have the **Cut** menu option; you can only copy from them.

In EZ, you can *cut* as well as copy a selected region into a document that you are editing. When you make a selected region and choose Cut from the pop-up menus, you remove the selected region from the document. You can then paste the region that you cut into any scrollable window. If you do not move the text caret after you choose Cut, you can paste the region that you cut back to where it was before.

To cut a selected region:

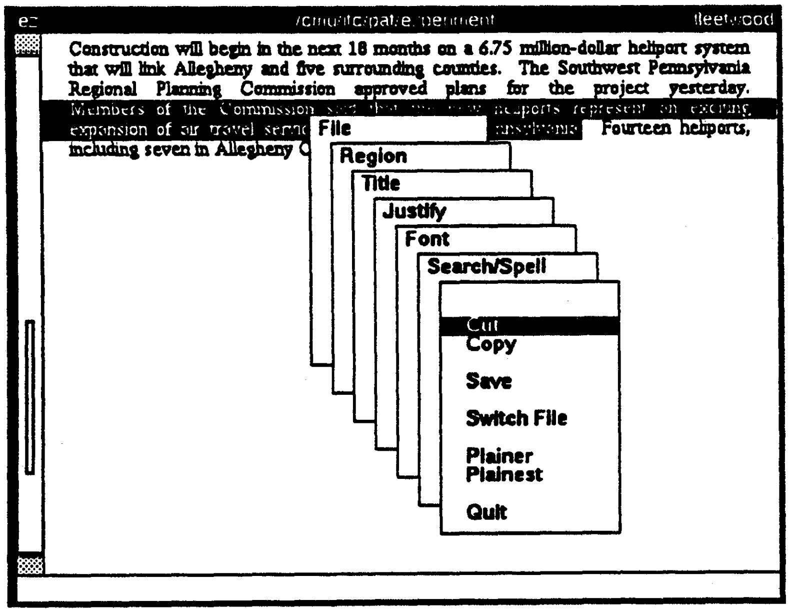
Action. Select the text you want to copy or cut. (Remember that to select text, you move the mouse cursor to where you want the selected region to begin and press the left mouse button, then move the mouse cursor to where you want the selected region to end and press the right mouse button.) When the selected region is darkened, presss both mouse buttons to expose the selected region menus, then choose Cut.

Response. The selected text disappears from the window.

To paste a selected region that you cut or copied:

Action. Move the text caret to a place in your document where you want to paste the selected text (by pointing the mouse cursor and clicking the left mouse button). Expose the program menus and choose **Paste.**

Response. The text you just cut is pasted at the position of the text caret, and is selected. Click the left mouse button to deselect the text.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Members of the Commission [illegible] heliports represent an exciting expansion of air travel serv[illegible] Fourteen heliports, including seven in Allegheny C[illegible]

Cut, Copy, and the Cutbuffer

Each region of text that you cut or copy is stored in a *cutbuffer* (a temporary storage space) until another piece of text is cut or copied. Therefore, the region that you cut most recently is always the one that appears when you use **Paste** to put it back. If you use **Cut** two consecutive times, the region that you cut the first time will no longer be easily available; it is replaced in the cutbuffer by the region that you cut the second time. (More information about the cutbuffer in EZ is available in the help document about EZ. If you are interested in that, use the Help program later to read the help document.)

To discard material that you have cut, simply do not use **Paste** to bring it back. The next time you use Cut or Copy, the material that you previously cut will be replaced.

Adding Styles to Your Text

Once you have created a document, you can use EZ to add *styles* to your text. Styles are special typographic effects, like boldface or italic type. To add styles, you select the region that you want to have a particular style and then choose the appropriate menu option.

To add a style:

Action. Select the sentence in your document that begins with "The Southwest...". Press both mouse buttons to display the selected region menus and choose the **Bold** option from the Font menu.

Response. The selected text changes to boldface type and the region is still selected. Click the left mouse button to deselect the text.

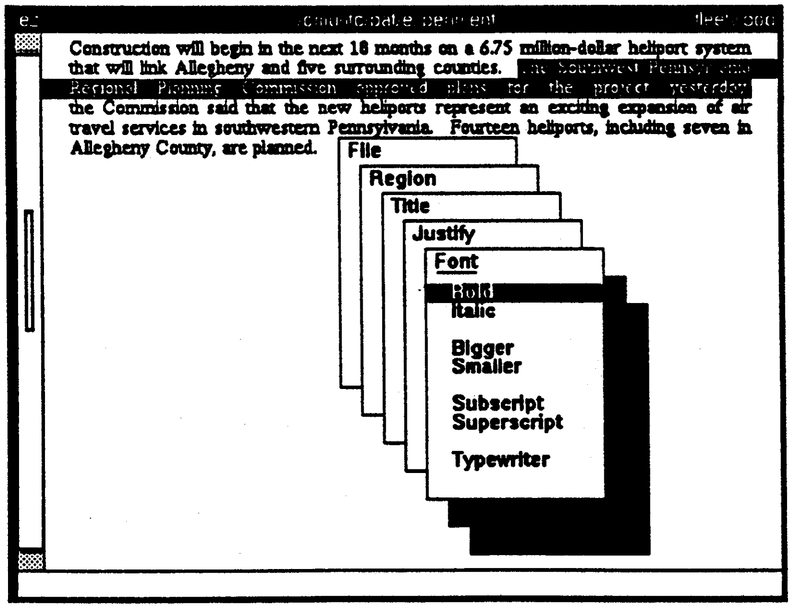
If you change your mind about a style, you can remove the style and return to plain text using the Plainer menu option.

To remove one or more styles:

Action. Select the styled region. Press both mouse buttons to display the selected region menus and choose the **Plainer** menu option.

Response. The bold style is removed from the selected region.

As you may have seen as you did this exercise, it can be difficult to create a selected region that exactly contains the styled text. For this reason, you can also use **Plainer** to remove a style without having to select the text first. You can place the text caret inside a region that has a style and choose **Plainer** from the menu. **Plainer** will automatically find the beginning and end of the style that appears around the text caret and remove the style.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday; the Commission said that the new heliports represent an exciting expansion of air travel services in southwestern Pennsylvania. Fourteen heliports, including seven in Allegheny County, are planned.

Adding More Than One Style to a Region

You can add more than one style to a single piece of text (in other words, you can make a single word both bold and italic if you wish).

To add several styles:

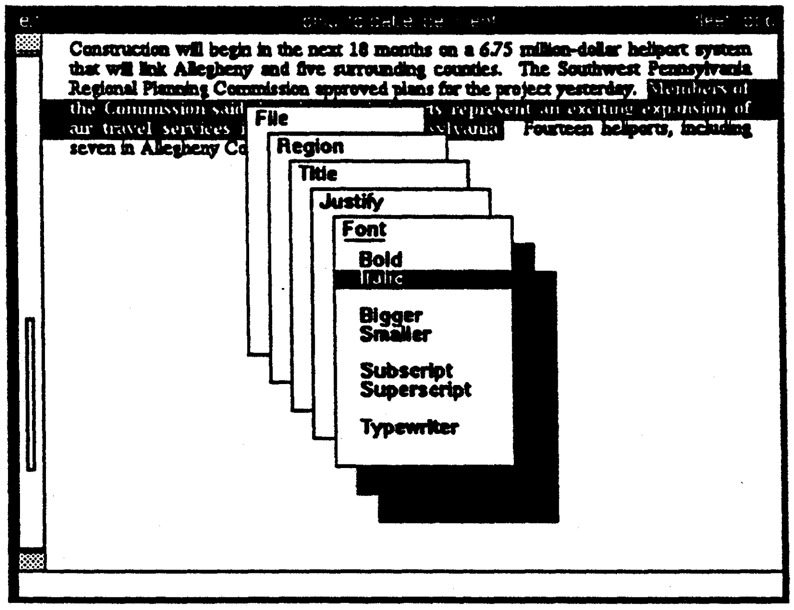
> **Action.** Select the sentence that begins with "Members of..." Press both mouse buttons to display the selected region menus and choose the **Bold** option from the Font menu.
>
> **Response.** The selected region changes to boldface type and the region is still selected.
>
> **Action.** Choose the **Italic** menu option.
>
> **Response.** The selected text changes to bold-italic type and the region is still selected. Click the left mouse button to deselect the text.

You can use the Plainest menu option to remove several styles at once.

To remove several styles:

> **Action.** Select the styled text. Press both mouse buttons to display the selected region menus and choose the **Plainest** menu option.
>
> **Response.** All styles disappear from the selected text.

Another way to remove multiple styles is to use **Plainer** several times. Using **Plainer** on a region that has multiple styles causes the most recent style you added to be removed. If you had a region with three styles added, you would have to use **Plainer** three times to get all the styles removed.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Members of the Commission said ts represent an exciting expansion of air travel services i lvania **Fourteen heliports, including seven in Allegheny Co**

Previewing A Document

In order to work flexibly with Andrew's changeable columns, EZ does not try to present your document exactly as it will appear on the printed page while you are adding new text. Instead, EZ has a Preview menu option to show you at any given moment what your document would look like if you printed it. Previewing a document often saves both time and paper, since it can show formatting errors that may not have been apparent earlier and let you repair them before you actually print the document.

To preview a document:

Action. Choose **Preview** from the program menus in EZ.

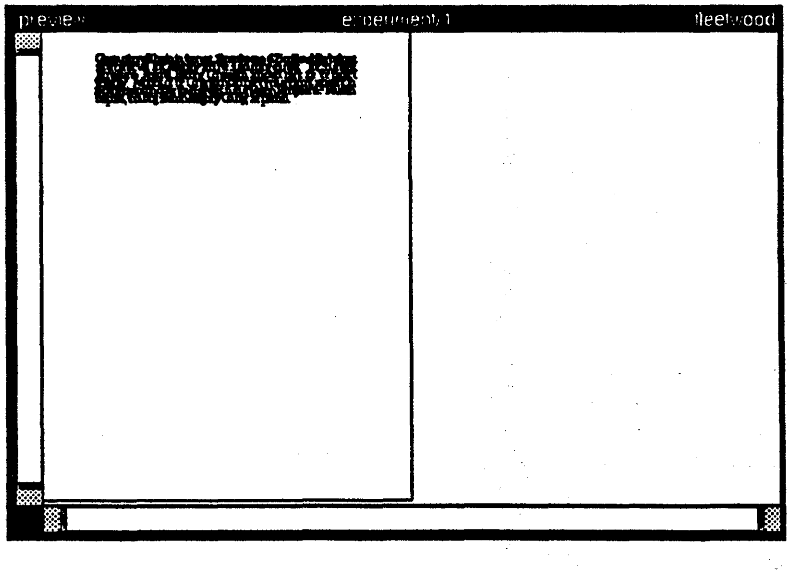
Response. A Preview window appears with the first page of your document displayed, exactly as it will look when printed.

When the Preview window appears, it shows the document to *scale*, or reduced so that the entire page will show in the window. You can use the scaled view to see how blocks of text appear on the screen and to check page breaks, but you usually cannot read the document in the scaled view because the letters are too small. You can use the **Full Size** to show the document in the font in which it will be printed, but only a portion of each page will appear in the window.

The Preview window has two scrollbars (one on the left side and one at the bottom of the window) that allow you to see different parts of the printed page. The horizontal scrollbar works much like the vertical scrollbar; while the vertical scrollbar moves the image of the printed page up and down, the horizontal bar allows you to move the image from side to side. They are of greater use when viewing the image at Full Size.

Viewing additional pages of Preview documents

If your document were more than one page long, you would want to examine more than just the first page. The pop-up menus in Preview contain options that let you see any page you specify and move among the pages easily.



Printing Documents with Preview

If, after checking your document, you find that it is ready to print, you can print it using the Preview menus.

To print from Preview:

Action. With the mouse cursor in the Preview window, press both mouse buttons to display the program menus and choose **Print** from the Preview program menus.

Response. After a moment, the printer icon (in Console) darkens, indicating that your document is waiting or *queued* (pronounced "cued") to be printed. When the printer icon returns to normal, your document has been printed.

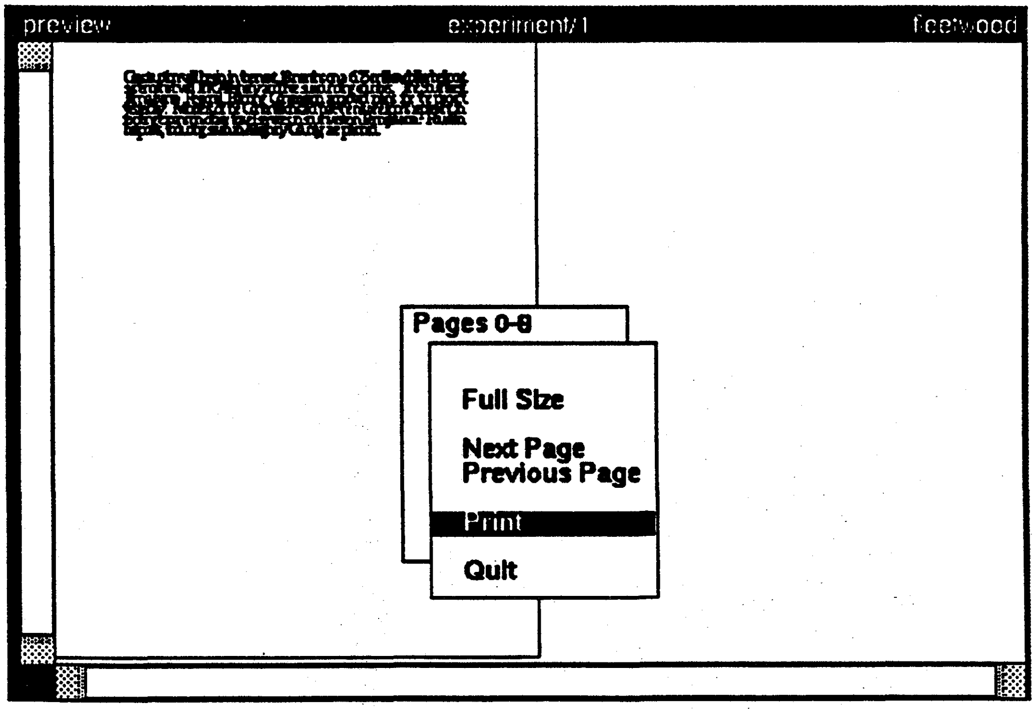
You can choose which printer will be used to print your document with the "Set Printer" option if, for example, the printer that you usually use (your default printer) develops a problem. The default printer for most Andrew users is "pine," which is described later in this section. To see a list of all the printers on campus and their locations, use the Help program to see the help document on "printers." You can also learn more about printing and the print queue (pronounced "cue") in the next few exercises.

Once you have finished examining your document with Preview, you can quit from it using the pop-up menus.

To quit from Preview:

Action. Choose **Quit** from the Preview program menus.

Response. The Preview window disappears.



Printing Documents with EZ

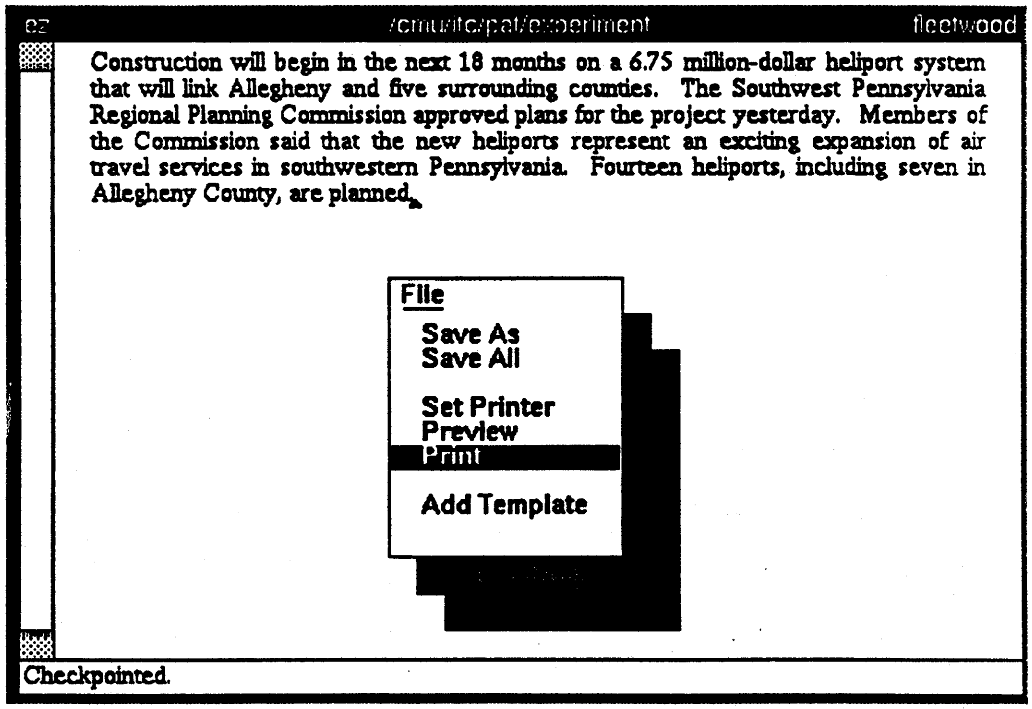
Although it is often convenient to use Preview before you print a document, for shorter documents you may have a good idea of how the document will look without using Preview. In those cases, you can print the document from the EZ window.

Before you continue this exercise, you may want to left-click in the title bar of the EZ window to shrink it, then left-click again to expand it so that the EZ window will take up all the space vacated by the Preview window. (You could also expand the window by using the "Expand" menu option in the title bar menus.)

To print from EZ:

> **Action.** With the mouse cursor in the EZ window, press both mouse buttons to display the program menus. Choose **Print** from the File menu.
>
> **Response.** After a moment, the printer icon (in Console) darkens, indicating that your document is queued to be printed. When the printer icon returns to normal, your document is finished printing.

You can also print documents from the Typescript window using the *ezprint* command if you know the name of the file in which the document is stored. You do not need to know about EZprint in order to complete these exercises, but you can use the Help program later to see the help document about it.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Members of the Commission said that the new heliports represent an exciting expansion of air travel services in southwestern Pennsylvania. Fourteen heliports, including seven in Allegheny County, are planned.

Checkpointed.

Checking the Print Queue

Now that you have queued two files for printing (or at least, queued the same file twice), you may be curious to know more about the print queue. In particular, you may be wondering whether and where you documents will be printed. You can check the *print queue* or the list of files waiting to be printed to find out where your printed copies will be printed and to see how long it will take for your document to be printed. (A *queue* is a line, like a checkout line at a grocery store. Shoppers enter the checkout line at the end and the shoppers who were in line first go through the checkout first, usually. In the same way, when you queue a file for printing from Preview or EZ, your print request is usually added to the end of the print queue and must wait its turn to be printed.)

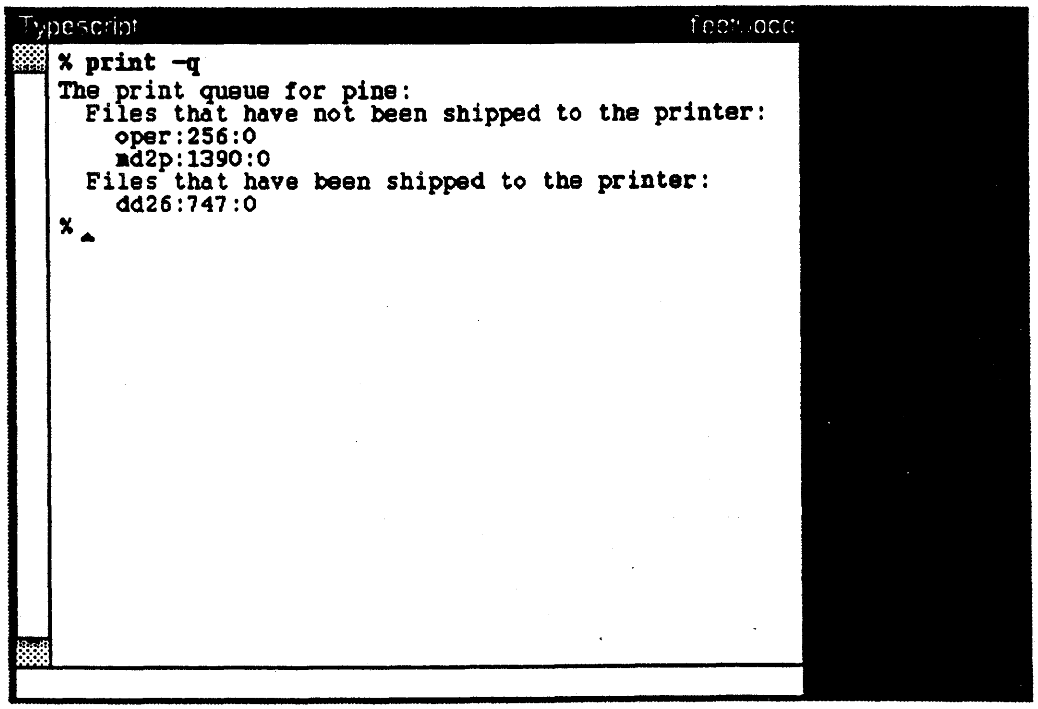
To check the print queue:

Action. In Typescript, type

print -q

and press Enter.

Response. Typescript displays a message similar to the one shown below. Your print is marked with your user ID and a number.



```
% print -q
The print queue for pine:
  Files that have not been shipped to the printer:
    oper:256:0
    md2p:1390:0
  Files that have been shipped to the printer:
    dd26:747:0
%
```

What the Print Queue Tells You

The listing of the print queue that you get by typing "print -q" in Typescript contains two piece of useful information. First, the name of your default printer appears in the first line, which tells you which printer will print your files. In the example above, the print queue is for the printer "pine," which is the most common default printer.

Second, some letters and numbers appear in the listing of the print queue that correspond to each of the files that is currently queued for printing on "pine" appears. The more files are in the queue, the longer it will take for your file to be printed. In the example on the previous page, there are relatively few files queued, so the files will be printed quickly unless the printer breaks down, as printers sometimes do.

About the printer called "pine"

The printer "pine" is located in the machine room on the ground floor of the UCC building. Documents that are printed on "pine" are placed every two hours in the *output bins* or racks across from the machine room. The bins are marked according to the first two letters of an individual's user ID, so "jbLw" would find any printed files that she had queued to "pine" in the bins marked IA-JB.

Other campus printers and their locations are listed in the help document about *printers.* You do not need to know about them in order to complete these exercises, but you can use the Help program to read about them later.

Ending an Editing Session with EZ

To end a session with EZ, you can just choose **Quit** from the program menus. However, if you have made changes to the document that you were editing but have not saved the changes, EZ notices that. If you try to quit when you have unsaved changes left, EZ *prompts* or asks you about whether you really want to quit, to keep you from losing work. Try it and see.

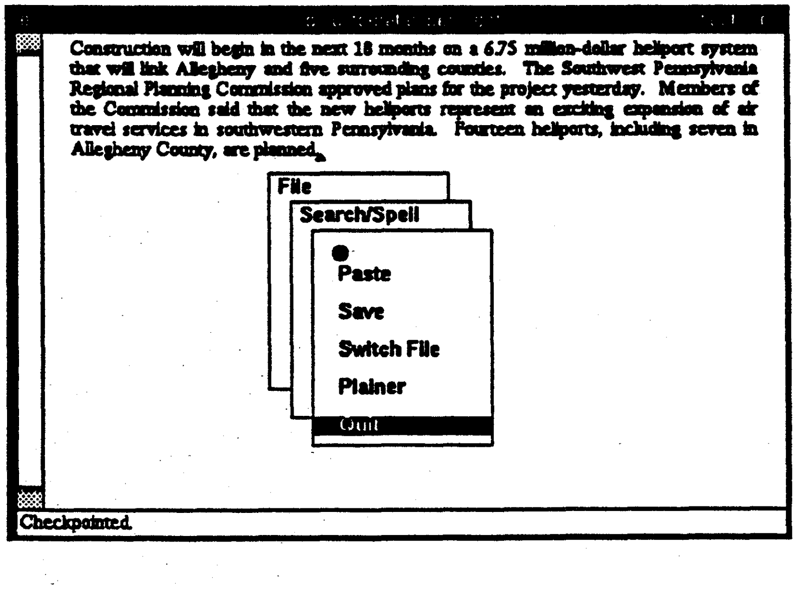
> **Action.** With the mouse cursor in the EZ window, type a few characters. Then, press both mouse buttons to display the program menus and choose **Quit**.
>
> **Response.** The words "You have unsaved changes; do you really want to quit? [no]" appear. The "no" in brackets at the end of the question means that "no" is the *default* answer to the question. If you just press Enter without typing anything, EZ assume that you meant, "No, I don't really want to quit yet" and removes the prompt, leaving the window as it was before you tried to quit.
>
> **Action.** Press Enter.

It is easy to end an editing session with EZ. However, to save yourself the trauma of losing any work, you should get into the habit of saving your document before you quit.

To quit EZ:

> **Action.** From the program menus in EZ, choose **Save**.
>
> **Response.** The words "Wrote file" appear at the bottom of the window to indicate that your document was written to a file. The full *pathname* to the file appears in the "Wrote file" message, too.
>
> **Action.** Choose **Quit** from the program menus.
>
> **Response.** The file is closed, and the window disappears.

If you would like to take a break now, remember to log out.



Construction will begin in the next 18 months on a 6.75 million-dollar heliport system that will link Allegheny and five surrounding counties. The Southwest Pennsylvania Regional Planning Commission approved plans for the project yesterday. Members of the Commission said that the new heliports represent an exciting expansion of air travel services in southwestern Pennsylvania. Fourteen heliports, including seven in Allegheny County, are planned.

Checkpointed.

14. Mail and Bulletin Boards

About Messages

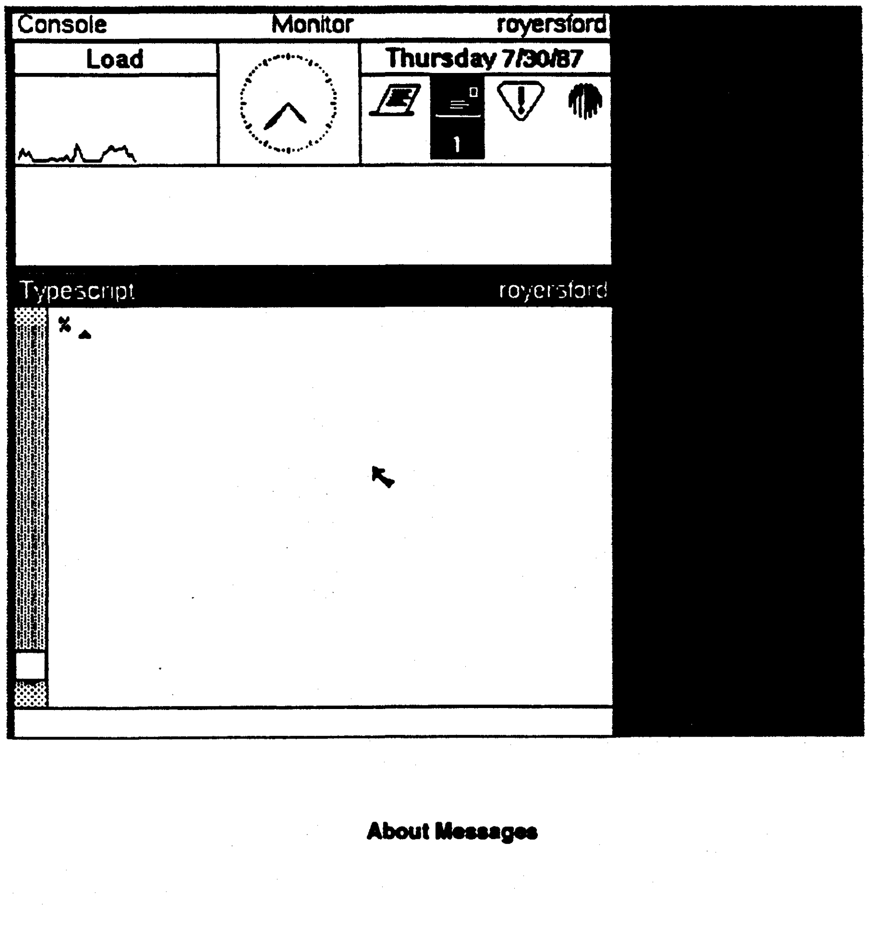
On Andrew, the program for reading electronic mail and bulletin boards is called Messages. SendMessage, its counterpart, is for sending mail to other users and to bulletin boards. soon, you will be able to send and receive multi-media messages using Messages and SendMessage on Andrew workstations.

In this section you can learn the basics of Messages and SendMessage. This includes reading mail and bulletin boards, printing and deleting mail, and sending messages. You can also learn how to change your subscription list, and how to quit the program. Along the way there are pointers to help documents which explain other features of the Andrew message system.

If you chose to take a break before this section, all you need to do is log back in.

How to Tell When You Have New Mail

The Console in the upper left hand corner of the screen tells you when you have mail by darkening its letter icon and showing the number of mail messages you have beneath the icon.



Reading New Mail with Messages

To read your new mail, start the program called Messages which creates a Messages window.

There are actually four ways to start Messages. You can either type one of two commands in the Typescript window, or you can choose one of two Typescript pop-up menu options. This section tells you how to start Messages using a pop-up menu option.

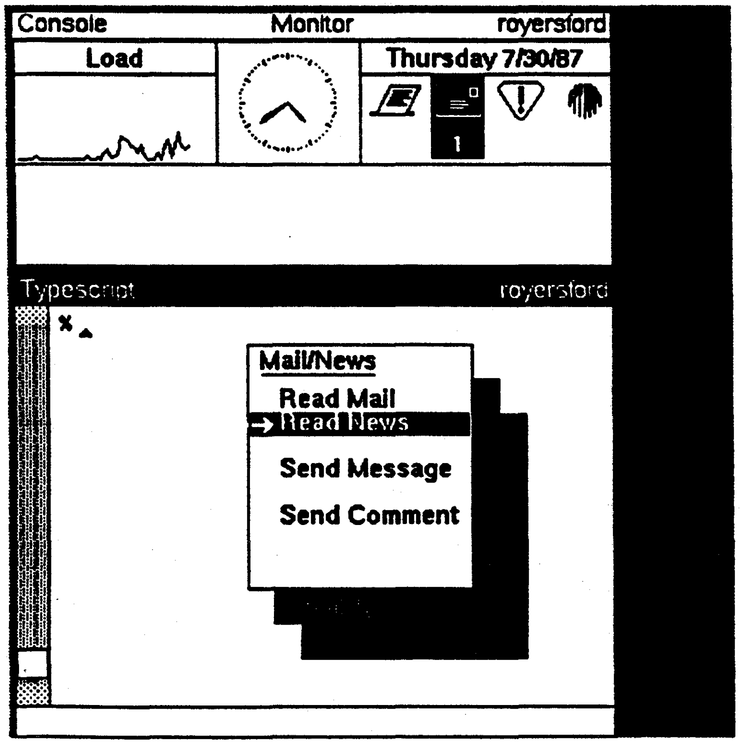
If you ever want to start Messages without the pop-up menus, typing the command **messages** in your Typescript starts Messages reading your mail and bulletin boards. Typing the command **messages -m** starts Messages reading only your mail.

To start Messages using pop-up menus:

> **Action.** Move the mouse cursor into the Typescript window. Choose **Read News** from the *Mail/News* pop-up menu card (see below).
>
> **Response.** In a moment or two, the Messages window (next page) appears on your screen.

Choosing **Read News** starts Messages to read both your mail and bulletin boards to which you are subscribed. Choosing **Read Mail** starts Messages to read only your mail.

Once you start Messages, you do not need to move into the Typescript and run it again to perform a new action. All actions should be performed from within the Messages window.



Using Messages

Think of Messages as a program that lets you use a filing cabinet which holds messages. In your filing cabinet are manila folders. The *folders* hold messages, like mail someone has sent you, or a message on a bulletin board. The folders are named so that you know what type of messages are in them.

You are able to see at a glance which messages are in a folder because each message has a one-line summary called a *caption* and all the captions for a folder are kept together in a list you can scan. The caption tells you the date, subject, sender and number of characters in a message. The caption also has a little marker at the front of it that lets you know if you have already seen the message before.

When you start Messages, the program goes through your filing cabinet looking for folders that have new messages in them. It takes all the folders with new messages out of the cabinet and puts them in a stack. It then takes the top folder off of the stack for you and opens it up so you can see the captions for the new messages in the folder.

This, essentially, is what you see in the Messages window when you start the program. The stack of folders with new messages appears in the top region of the window (although you often have more folders with new messages than you can see). The captions for the first folder on the stack appear in the middle region of the window. When you select a caption to read its message, the message body appears in the lower region of the window.

So, to read messages using Messages, you have to tell the program which folder from the stack you want to look in, and then which message in that folder you want to read. The following sections take you through this process of "selecting" messages and folders.

Selecting a Mail Message

As mentioned on the previous page, Messages "selects" your first folder with new messages so you can see its captions. In this case, your first folder with new messages is your mail folder. This means that you do not have to "select" your mail folder at first. All you have to do is select the message you want to read.

To select a mail message:

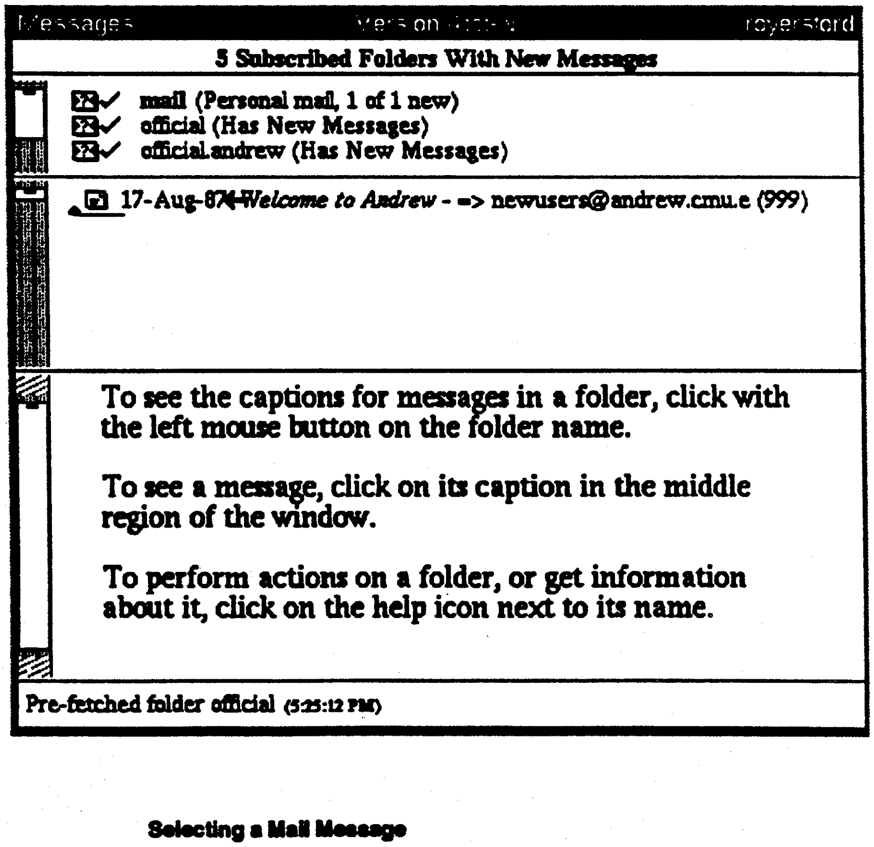
Action. Move the mouse cursor onto the caption of your first mail message. Look at the illustration below, especially **mouse cursor on caption**, to position the mouse cursor. Click once with the left or right mouse button.

Response: Three important things happen:

- The caption becomes bold, indicating that it and the message it represents are "selected."
- The message appears in the body area of the window.
- The mail icon becomes a check mark, indicating you have read that message.

Notice that in the illustration below, the Messages version number in the middle of the title bar is 4.xx. The version of Messages that you use at your workstation has a number instead of two x's.

Messages is sometimes a little slow because it has a lot of work to do. Wait until the clock face cursor returns to an arrow cursor before expecting Messages to respond to your actions. Although at times it may not look like Messages is doing anything, it remembers all the clicks you make and will perform the actions for your clicks in order.



Messages   Version [illegible]   royersford

5 Subscribed Folders With New Messages

✓ mail (Personal mail, 1 of 1 new)
✓ official (Has New Messages)
✓ official.andrew (Has New Messages)

17-Aug-87 *Welcome to Andrew* - => newusers@andrew.cmu.e (999)

To see the captions for messages in a folder, click with the left mouse button on the folder name.

To see a message, click on its caption in the middle region of the window.

To perform actions on a folder, or get information about it, click on the help icon next to its name.

Pre-fetched folder official (5:25:12 PM)

About Messages Folders

Before you go on to select other folders to look in, you need to know a little more about the different types of folders in Messages. There are three types of Messages folders:

Your personal mail folder is like your own folder that is stamped "Top Secret," with the added feature which other people can slip messages into it without looking in it. This means that it can be read only by you, and that mail arriving at your Andrew account is put in your mail folder automatically when you start Messages. Because "mail" is your own personal folder (every Andrew user has their own personal folder called "mail"), you can delete messages from it.

Personal folders are folders that you create for your own use. Like "mail," personal folders are "Top Secret" and can be read only by you. Also like "mail," you can copy messages into and delete messages from your personal folders. Unlike "mail," other people can not slip messages into your personal folders--only you can. You do not have any personal folders, yet, (other than "mail") because you have to create them yourself. For more information on creating and using personal folders, type **help messages-folders** in the Typescript.

Bulletin board folders are not stamped "Top Secret," and can be read by anyone. Generally, you can put messages in bulletin board folders, but you can not delete messages from bulletin boards because they are not really your folders. You can, however, copy messages from bulletin boards into personal folders. Bulletin boards are distinguished from one another by topic. Each bulletin board folder is supposed to contain messages related to its name. "Bulletin board" is often abbreviated "bboard," which is pronounced "bee-board." Messages on them are often referred to as "posts."

You select and read all folders in exactly the same way, and they all appear in the folders region of the Messages window as if there is no difference between them. However, the folders are different in the ways you can use them, so you should keep the three different types of folders clear in your mind, as if you had three different sections in your filing cabinet.

Selecting a Folder

To read messages on a bulletin board you have to do two things. First, you select the bulletin board folder that you want to see. Then, you select messages in the folder to read. The following sections take you through these actions.

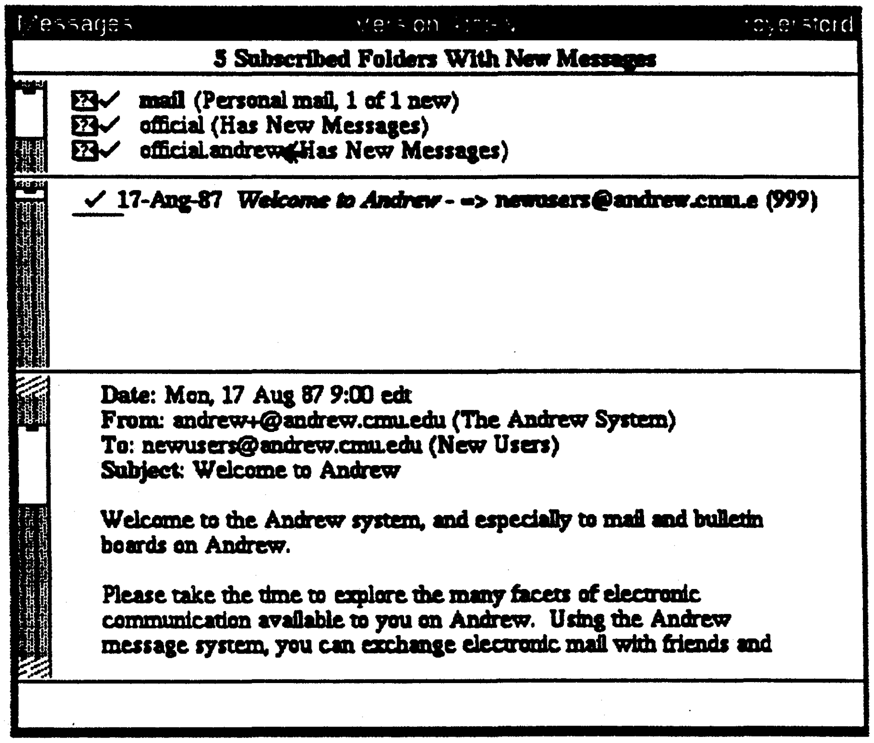
To select a folder:

Action. In the Messages window, move the mouse cursor onto the name of the bulletin board called "official.andrew." Look at the illustration below, especially **mouse cursor on folder name**, to position the mouse cursor. Click once with the left mouse buttom.

Response. Three important things happen:

- The name of the folder becomes bold, indicating that it is "selected."
- The new captions that represent messages in this folder appear in the captions area.
- The "(Has New Messages)" beside the folder name becomes something like "(Official Bboard, 4 of 4 new)."

If you see a dialogue box in the middle of your window, you clicked on the help icon instead of the folder name. Although you can see the messages in a folder by using the help icon, later you can learn a different use of the help icon later. For now, click in the box that says **Forget it -- do nothing**, and select the folder by clicking on its name.



5 Subscribed Folders With New Messages

✓ mail (Personal mail, 1 of 1 new)
✓ official (Has New Messages)
✓ official.andrew (Has New Messages)

✓ 17-Aug-87 *Welcome to Andrew* - => newusers@andrew.cmu.e (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu (The Andrew System)
To: newusers@andrew.cmu.edu (New Users)
Subject: Welcome to Andrew

Welcome to the Andrew system, and especially to mail and bulletin boards on Andrew.

Please take the time to explore the many facets of electronic communication available to you on Andrew. Using the Andrew message system, you can exchange electronic mail with friends and

Selecting a Message

When you selected your first mail message, you were selecting a message in a folder. Now, you can do it again. You can select any number of messages in a folder, in any sequence. The only restriction is that you can only read messages one at a time.

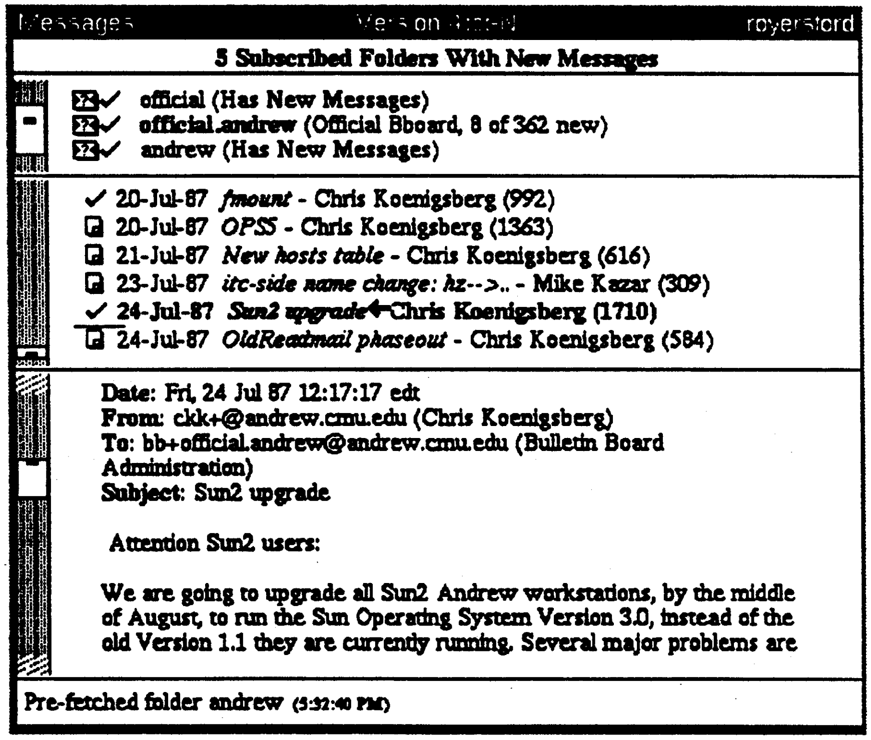
To select a message in a folder:

Action. In the captions area, move the mouse cursor on to the caption for any message. Click once with the left mouse button.

Response. Four important things happen:

- The caption becomes bold, indicating that it and the message it represents are "selected."
- The message appears in the body area of the window.
- The **bboard icon** in the caption becomes a check mark, indicating that you have seen that message.
- The **underline icon** moves down beneath the check next to the selected caption, indicating that the Messages program will treat all the messages above the underline icon as "seen" the next time you run Messages, even if you haven't actually looked at all of the messages above the underline.

Since some bulletin board folders have lots and lots of messages in them, Messages makes it easier to read the new messages by presenting only the new message captions to you. This does not mean that you can not see the old messages. Simply scroll back up in the captions region to see captions for old messages.



5 Subscribed Folders With New Messages

✓ official (Has New Messages)
✓ **official.andrew** (Official Bboard, 8 of 362 new)
✓ andrew (Has New Messages)

✓ 20-Jul-87 *fmount* - Chris Koenigsberg (992)
20-Jul-87 *OPSS* - Chris Koenigsberg (1363)
21-Jul-87 *New hosts table* - Chris Koenigsberg (616)
23-Jul-87 *itc-side name change: hz-->..* - Mike Kazar (309)
✓ **24-Jul-87 *Sun2 upgrade* - Chris Koenigsberg (1710)**
24-Jul-87 *OldReadmail phaseout* - Chris Koenigsberg (584)

Date: Fri, 24 Jul 87 12:17:17 edt
From: ckk+@andrew.cmu.edu (Chris Koenigsberg)
To: bb+official.andrew@andrew.cmu.edu (Bulletin Board Administration)
Subject: Sun2 upgrade

Attention Sun2 users:

We are going to upgrade all Sun2 Andrew workstations, by the middle of August, to run the Sun Operating System Version 3.0, instead of the old Version 1.1 they are currently running. Several major problems are

Pre-fetched folder andrew (5:32:40 PM)

Deleting Mail Messages

Now that you know how to select a folder and select messages to read in a folder, you can move around in your filing cabinet and perform actions on your messages, if you want. For example, once you have read a mail message, you may want to delete the message so that it doesn't clutter up your mail folder. (Don't worry about deleting a message if you want to keep it. You can undelete it in before you quit Messages.)

Remember that you can't delete messages from bulletin boards.

To delete a mail message:

Action. Prepare to delete your first mail message by selecting your mail folder, and then selecting the message in your mail folder. You may need to scroll up in the folders area to see your mail folder. Review the previous two pages if you need to remember how to select folders and messages.

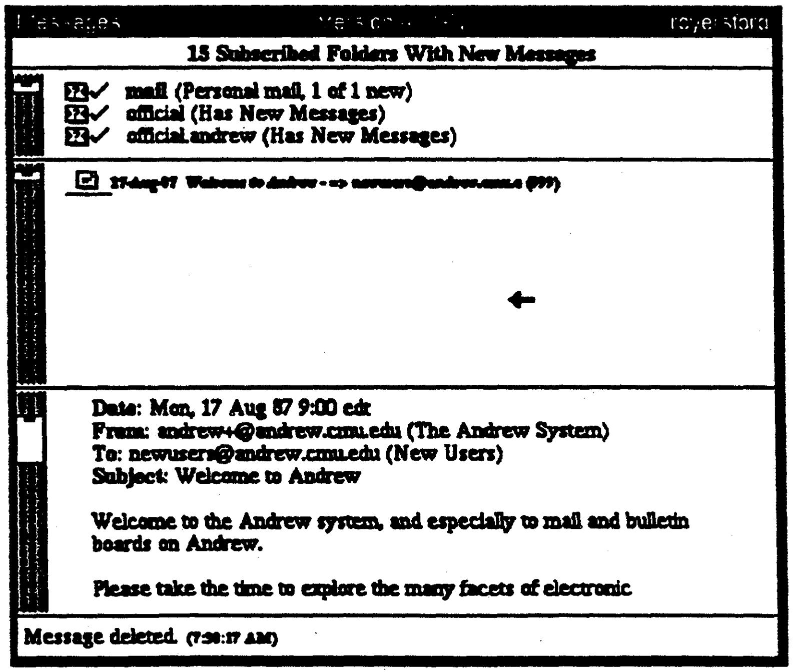
Response. Selecting your mail folder makes the caption appear in the captions area. Selecting the caption makes the message appear in the body area.

Action. In the Messages window with your first mail message selected, choose **Delete** from the front pop-up menu card.

Response. Three important things happen:

- The selected message's caption changes to a smaller type size.
- The caption icon changes to a **deleted mail icon.**
- The message "Message deleted. (time)" appears in the message area.

Although the message is deleted, it is not permanently removed, yet. The message is only permanently removed when the folder is "purged." Folders are purged when you quit the program.



mail (Personal mail, 1 of 1 new)
official (Has New Messages)
official.andrew (Has New Messages)

17-Aug-87 Welcome to Andrew - => newusers@andrew.cmu.e (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu (The Andrew System)
To: newusers@andrew.cmu.edu (New Users)
Subject: Welcome to Andrew

Welcome to the Andrew system, and especially to mail and bulletin boards on Andrew.

Please take the time to explore the many facets of electronic

Message deleted. (7:30:17 AM)

Undeleting Mail Messages

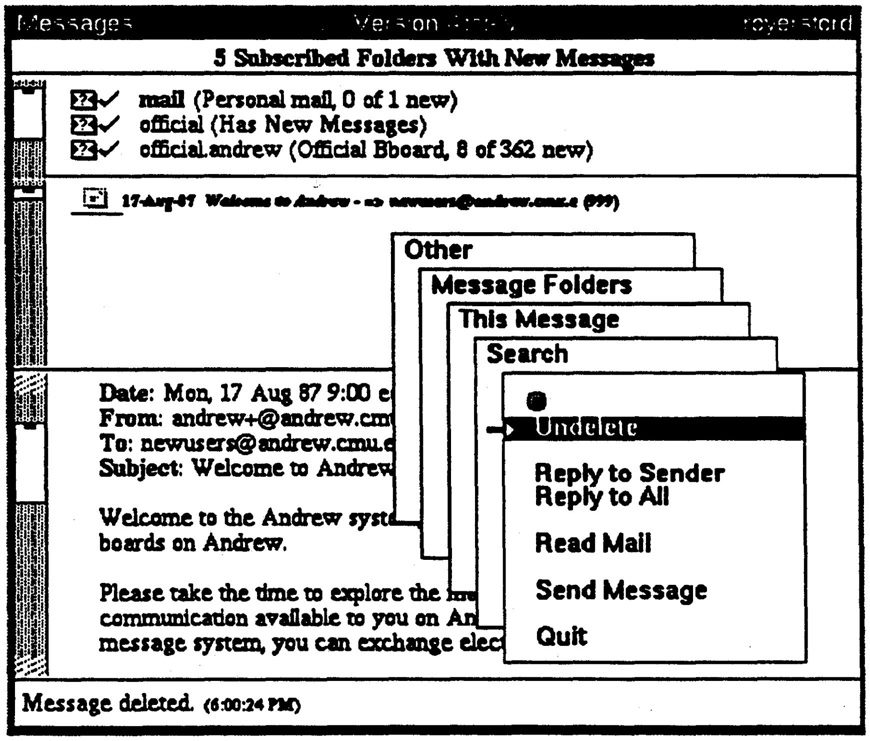
If you decide that you don't want to delete a message, you can undelete it, since the message is not permanently removed until you quit the program. (You can also delete the message again after you undelete it.)

To undelete a mail message:

Action. In the Messages window with your first mail message selected and deleted, choose **Undelete** from the front pop-up menu card.

Response. Three important things happen:

- The selected message's caption changes to the regular type size.
- The deleted mail icon becomes a check icon.
- The message "Message undeleted. (time)" appears in the message area.



5 Subscribed Folders With New Messages

mail (Personal mail, 0 of 1 new)
official (Has New Messages)
official.andrew (Official Bboard, 8 of 362 new)

17-Aug-87 Welcome to Andrew - => newusers@andrew.cmu.e (999)

Date: Mon, 17 Aug 87 9:00 e
From: andrew+@andrew.cm
To: newusers@andrew.cmu.e
Subject: Welcome to Andrew

Welcome to the Andrew syst
boards on Andrew.

Please take the time to explore the
communication available to you on An
message system, you can exchange elec

Message deleted. (6:00:24 PM)

Printing a Message

After you read a message, you may want to print out a copy. This section describes how you can print out a copy of a mail message from your mail folder.

You can print messages from any folder.

To print a message:

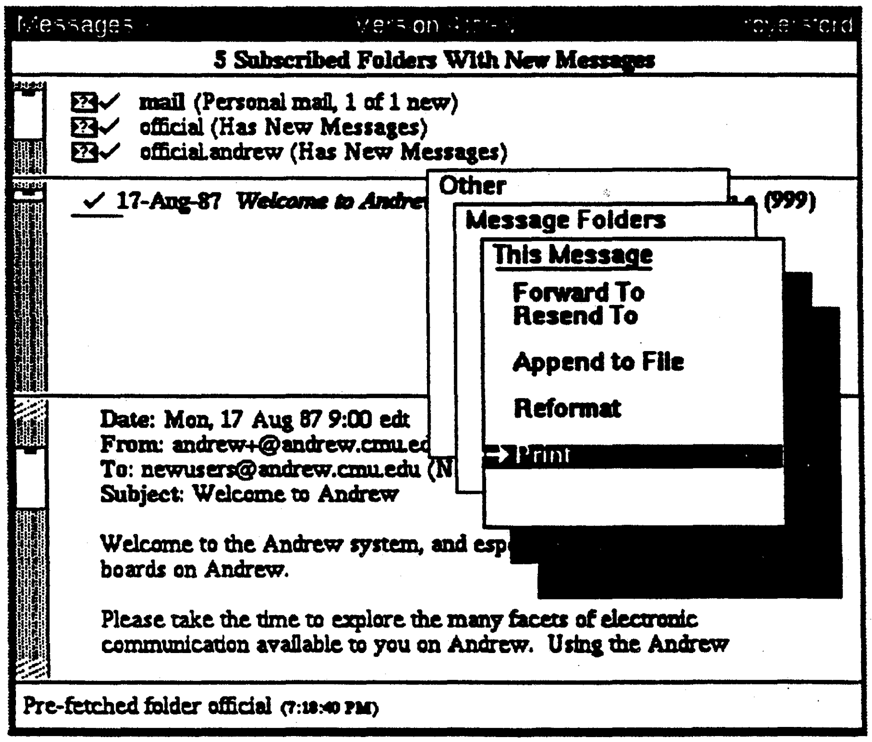
Action. In the Messages window with a message selected, choose **Print** from the *This Message* pop-up menu card.

Response. Three important things happen:

- "Printing message, please wait..." appears in the Messages window.
- The message "Message queued for printing. (time)" appears in the message area.
- In a minute, the printer icon in your Console darkens, indicating that the selected message is queued for printing at your default printer.

When the message is printed, "Messages_You_Wanted_To_Print" appears on the cover sheet. It also appears in the print queue.

You can check the print queue for your message.



5 Subscribed Folders With New Messages

mail (Personal mail, 1 of 1 new)
official (Has New Messages)
official.andrew (Has New Messages)

✓ 17-Aug-87 *Welcome to Andre* (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.ed
To: newusers@andrew.cmu.edu (N
Subject: Welcome to Andrew

Welcome to the Andrew system, and esp
boards on Andrew.

Please take the time to explore the many facets of electronic communication available to you on Andrew. Using the Andrew

Pre-fetched folder official (7:18:40 PM)

Sending Mail with Messages

Now that you know about reading messages with Messages, you need to learn how to send messages with SendMessage. To send a mail message you must to do four things. First, start the SendMessage program. Then, when the Send Message or Message Composition window appears, address the mail, compose the mail, and send the mail. The following sections take you through these actions as you send mail to an imaginary user, **Pat Enright**, who has an Andrew account.

There are actually three ways to start a blank SendMessage window. You can start it from Typescript by either typing **sendmessage**, or choosing **Send Message** from the *Mail/News* pop-up menu card. In this section, you start SendMessage a third way–from within the Messages program.

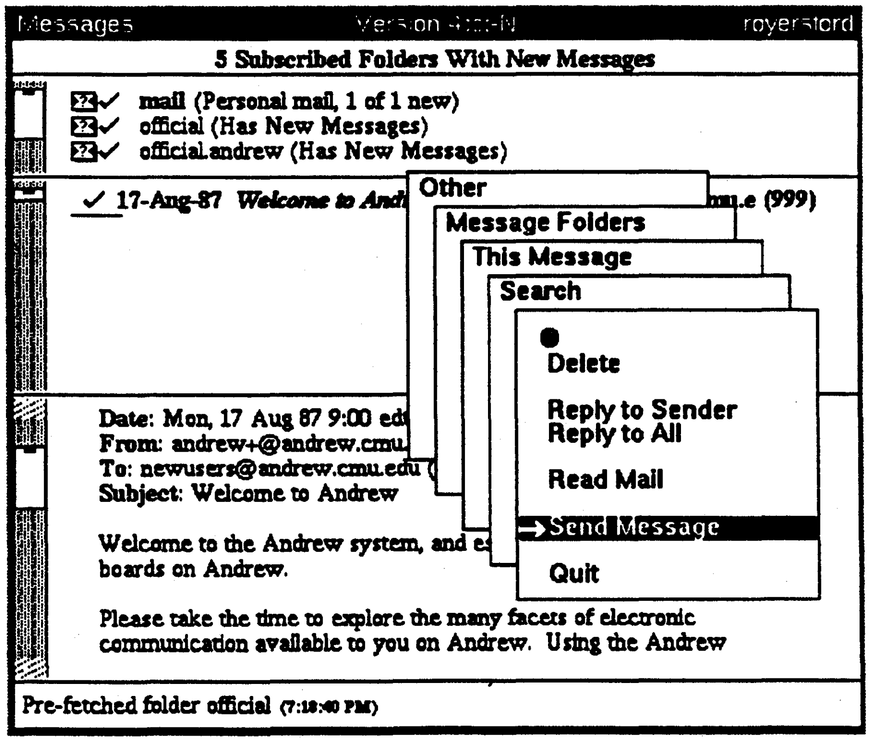
To start SendMessage from within Messages:

Action. In the Messages window, choose **Send Message** from the front pop-up menu card.

Response. In a moment or two, the Message Composition window appears on your screen.

When you choose **Send Message** from within Messages, the Send Message window that appears is called a Message Composition window. It is linked to the Messages program. When you quit Messages, the Message Composition window disappears, also.

You can also create a Message Composition window that is not blank, but already addressed. To do this, choose **Reply to Sender** from the front pop-up menu. How the message is addressed when you use this menu option is explained in the following sections.



5 Subscribed Folders With New Messages

✓ mail (Personal mail, 1 of 1 new)
✓ official (Has New Messages)
✓ official.andrew (Has New Messages)

✓ 17-Aug-87 *Welcome to And* .e (999)

Date: Mon, 17 Aug 87 9:00 ed
From: andrew+@andrew.cmu.
To: newusers@andrew.cmu.edu (
Subject: Welcome to Andrew

Welcome to the Andrew system, and e
boards on Andrew.

Please take the time to explore the many facets of electronic communication available to you on Andrew. Using the Andrew

Pre-fetched folder official (7:18:40 PM)

Addressing the Message

In the Message Composition window, the text caret is automatically positioned at the To: header. This means that the mouse cursor can be anywhere in the Message Composition window when you start typing. You can use the mouse cursor to move the text caret between header lines, and you can use the backspace key to correct any typing errors.

To fill in the To: header:

Action. Move the mouse cursor into the Message Composition window. Type

P Enright

for Pat Enright at the To: header. When you are finished, press Enter.

Response. "P Enright" appears beside the To: header and the text caret moves down to the Subject: header.

SendMessage has an address checking procedure that supplies the full address

userID+@andrew.cmu.edu

for names or user IDs after you send the mail. Thus, if you don't know someone's Andrew user ID, his or her first initial and last name are often good information to type at the To: header. However, once you know someone's Andrew user ID, you can use

userID+

to address your mail. You can also set up an alias file to simplify addressing mail. Type help aliases at the Typescript for more information on aliasing.

To send mail to more than one person, separate their names or user IDs with commas. You can mix names and user IDs, if you want.

Capitals are not important when you are filling in SendMessage headers.

When you choose Reply to Sender to open a Message Composition window, the To: header is automatically filled in with the address of the person who sent you the message that is currently selected in the Messages window.

Filling in the Other SendMessage Headers

To fill in the Subject: header:

Action. Type

try SendMessage

at the Subject: header. When you are finished, press Enter.

Response. "try SendMessage" appears at the Subject: header, and the text caret moves down to the CC: header.

You must type a subject, or your message will not be sent.

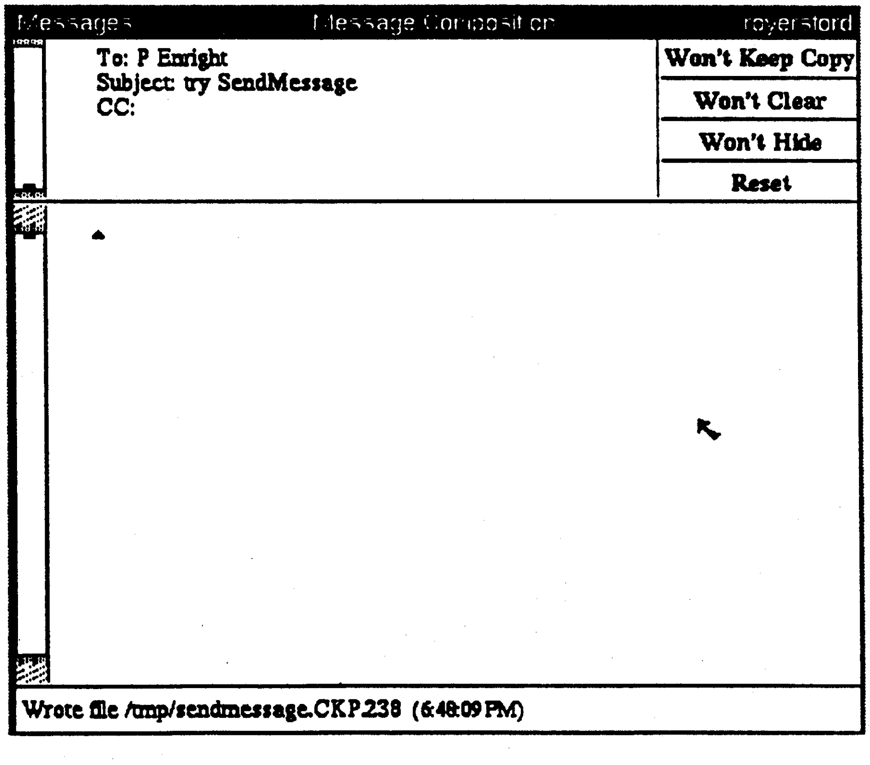
When you choose **Reply to Sender** to open a Message Composition window, the Subject: heder is filled in with the subject of the currently selected message, with "Re:" inserted.

To fill in the CC: header:

A "carbon copy" of the message will be sent to the recipient(s) you specify at this header. The address checking procedure that supplies correct address forms for names and user IDs at the To: header also checks information at this header.

Action. Do not type anything at the CC:. Press Enter to skip over it.

Response. The text caret moves into the body area of the Message Composition window.



To: P Enright
Subject: try SendMessage
CC:

Wrote file /tmp/sendmessage.CKP238 (6:48:09 PM)

Composing the Message

The body area of the Message Composition window is like an EZ document. You can cut and paste to and from the body area of the Message Composition window and other windows.

To compose a message:

> **Action.** With the text caret in the body of the Message Composition window, type anything you would like to send to Pat.
>
> **Response.** What you type appears in the body of the Message Composition window.

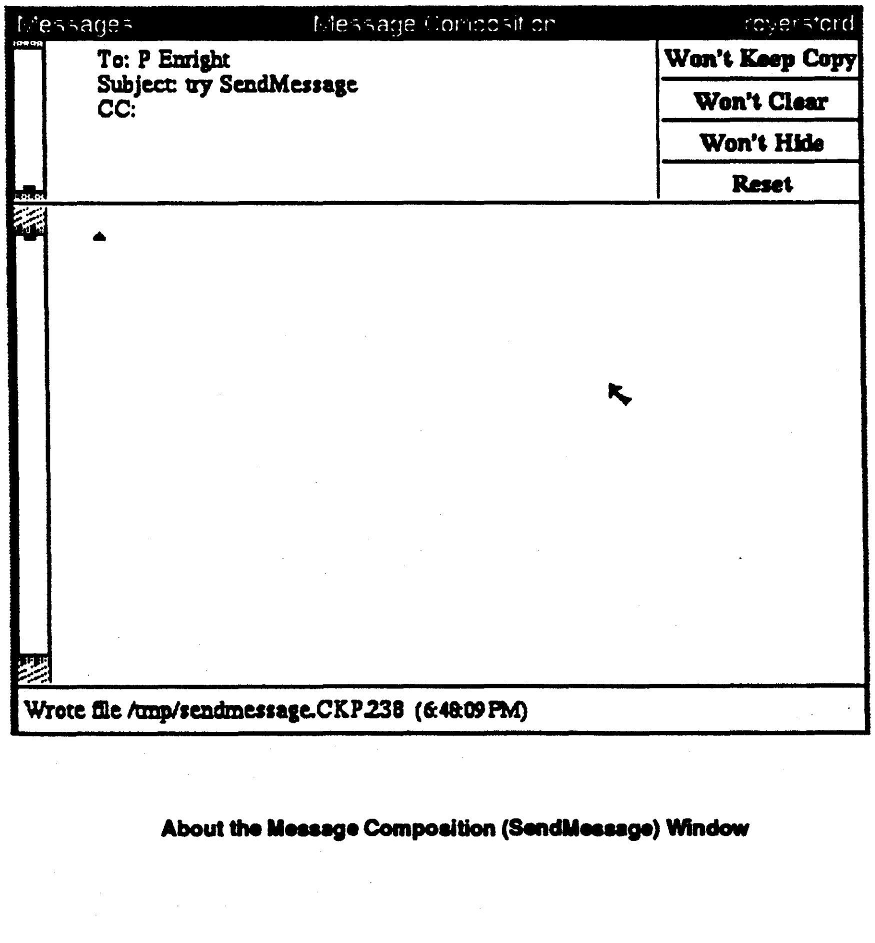
You must type at least two characters in the body of the message or it will not be sent.

About the Message Composition (SendMessage) Window

The rectangular areas on the right side of the Message Composition window are actions that SendMessage performs when sending your mail. When you click with either the left or right mouse button on one of the rectangles (called "buttons"), the button changes to its opposite. Click again, and it changes back.

> **Won't Keep Copy** means that when you send your mail, SendMessage will not send a copy of it to your mail box.
>
> **Won't Clear** means that after you send your mail, the header and body areas of the Message Composition window will not be cleared.
>
> **Won't Hide** means that after you send your mail, the Message Composition window will not shrink to a title bar, and the Messages window will not expand back out to fill the screen.
>
> **Reset** clears the header and body areas of the Message Composition window, and also makes the buttons read **Won't Keep Copy, Won't Clear, and Won't Hide.**

If you find that you use these buttons a lot, and want to change the default values of them (what Reset resets to, and what appears in the initial SendMessage window) choose **Set Options** from the back menu card in Messages. From the dialogue box that appears, choose **Message sending options.** Help appears in the body area of the message window to aid you in making the next appropriate dialogue box choice.



To: P Enright
Subject: try SendMessage
CC:

Wrote file /tmp/sendmessage.CKP238 (6:48:09 PM)

Sending the Message

To send the mail:

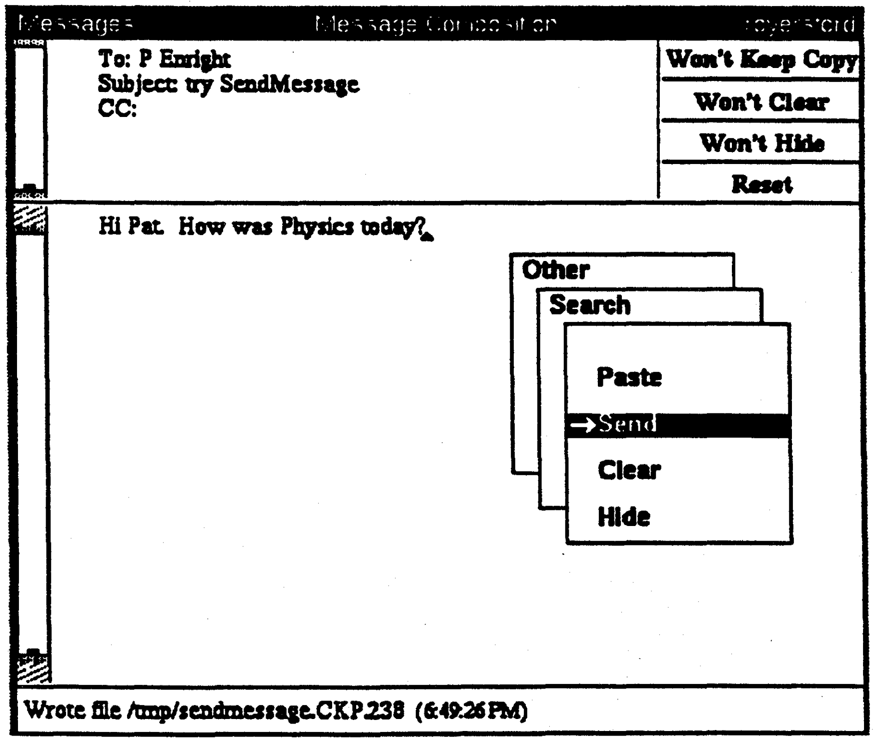
Action. Choose Send from the front pop-up menu card in the Message Composition window.

Response. Two important things happen:

- SendMessage presents you with a series of messages telling you what it is doing.
- When it is finished, the line "Your message has been sent. (time)" appears in the message area of the Message Composition window.

If you log out immediately after sending mail, it may take a while for your mail to reach its destination. In the future, this will not be the case.

If one of the names you typed does not validate, bring the text caret back to the name that is not validating by moving the mouse cursor to the end of the name and left clicking. Check for and correct any spelling errors, or try using a different first initial, or no initial at all, and choose Send again.



To: P Enright
Subject: try SendMessage
CC:

Hi Pat. How was Physics today?

Wrote file /tmp/sendmessage.CKP.238 (6:49:26 PM)

Clearing and Hiding the Message Composition Window

If you want to send another message, you can now clear the Message Composition window. If you don't want to send another message, you may want to hide the Message Composition window so it does not take up space on your screen.

To clear the Message Composition window:

Action. Choose **Clear** from the front pop-up menu in the Message Composition window.

Response. The headers and body areas of the Message Composition are erased. You are now ready to send another message, if you want.

If a dialogue box appears asking if you want to erase the mail you have not sent, this means you did not choose **Send** when sending your last message. Remove the box by clicking on **no** (if you want to send your last piece of unsent mail), or **yes** (if you don't want to send your last piece of unsent mail).

To hide the Message Composition window:

Action. Choose **Hide** from the front pop-up menu in the Message Composition window.

Response. The Message Composition window shrinks to a title bar.

Do not use **Zap** from the title bar menus to hide the Message Composition window. It causes Messages also to be Zapped. In general, avoid using **Zap** from the title bar menus to close windows.

You do not have to clear the Message Composition window before you hide it.

Sending Messages to Bulletin Boards

Sending a message to a bulletin board ("posting") is just like sending mail to someone, except that the address is the name of a bboard instead of a person's name or user ID.

In this section, you can send a message to the bulletin board called "andrew.test." This bulletin board exists for people (like yourself) to learn how to send messages to bboards, so you can type anything you want in the subject and body of your message.

To make a bulletin board post:

Action. If you hid the Message Composition window, reopen it by left clicking on its title bar.

Response. The Message Composition window expands out from the title bar.

Action. Type

andrew.test

at the To: header. Press Enter when you are finished.

Response. "andrew.test" appears at the To: header.

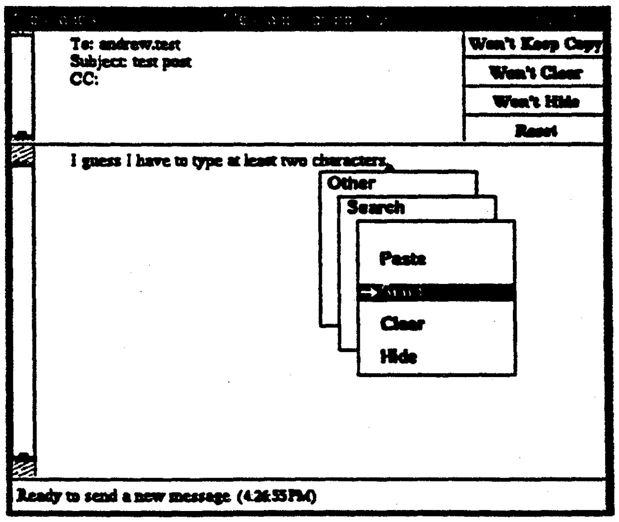
Action. Type a subject, CC the message to someone if you want, and compose your message. Press Enter to move from one header to the next, and into the body of the Message composition window.

Response. What you type appears at the appropriate places.

Action. Choose Send from the front pop-up menu.

Response. Your message is sent to the bboard called andrew.test

If you want more information on posting to bboards, type **help bboards** in the Typescript.


To: andrew.test
Subject: test post
CC:

I guess I have to type at least two characters.

Ready to send a new message. (4:26:55 PM)

About Your Subscription List

Your subscription list is a list of all the folders in your filing cabinet. All new users initially have five folders in their subscription list:

mail
official
official.andrew
andrew
andrew.hints

The first folder, "mail," is your mail folder. The other four folders are bulletin board folders. Remember the difference between your mail folder and bulletin board folders.

You can take folders in and out of your filing cabinet depending on your interests by subscribing and unsubscribing to them. This way, you can control which folders Messages checks for new messages when you start the program.

The folders you can subscribe and unsubscribe to are your mail folder, other personal folders, and bulletin board folders. Your mail and personal folders only exist in your own filing cabinet. Bulletin boards, however, are kept in one huge filing cabinet that everyone can look through. When you have a bulletin board in your subscription list, it is like having an exact copy of the bulletin board folder from the huge filing cabinet in your own personal filing cabinet.

There are many, many bulletin boards available on a wide range of topics. From bulletin boards about Dr. Who, David Letterman, or the Grateful Dead, to discussions of feminism, the latest movies, cars, recipes, music, or current events, there are bulletin boards for everyone.

In the following sections, you can learn how to look through the huge bulletin board filing cabinet, search around in it, read messages from folders in it, and subscribe (or unsubscribe) to bulletin board folders to make your subscription list fit your needs and tastes.

Exposing the All Folders List

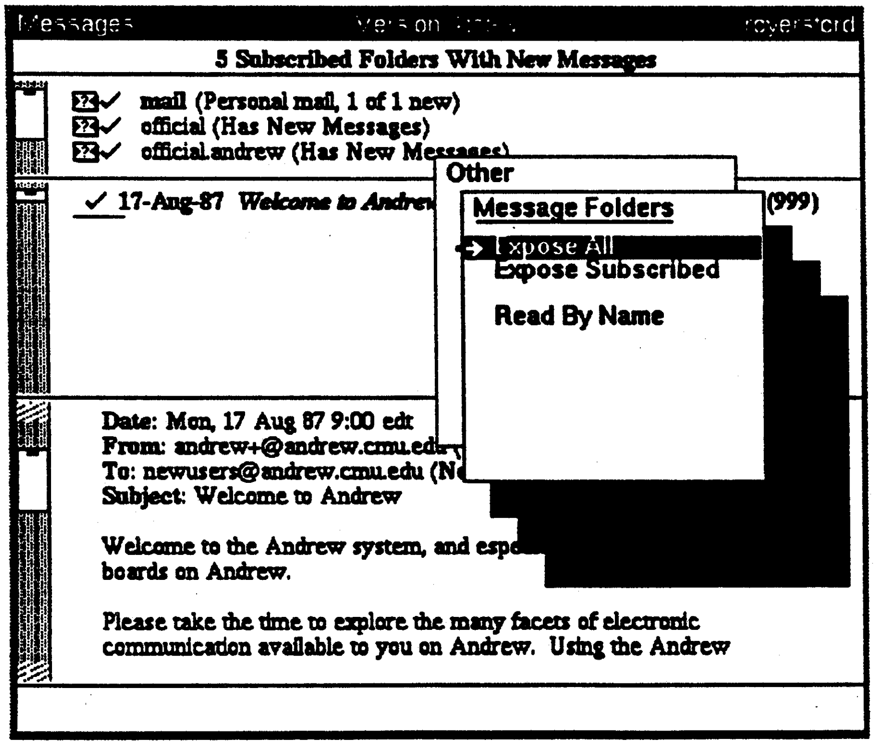
Exposing the All Folders list is like temporarily dumping the huge filing cabinet right into to your personal filing cabinet. When you expose a different list after you are finished looking through the All Folders list, your filing cabinet returns to its normal size, with only your subscriptions in it.

If your Message Composition window is still exposed from sending a message to "andrew.test" you may want to hide it and make the Messages window fill the screen. To do this, choose Hide from the front menu in the Message Composition window. Then, click once on the title bar of the Messages window (to hide the window), and then once again (to expand the window).

To see a list of all the folders available:

> **Action.** In the Messages window, choose **Expose All** from the *Messages Folders* pop-up menu card.
>
> **Response.** In a minute, a list of all the folders available to you (the "All Folders" list) appears in the folders area of the Messages window.

Since the number of folders in the All Folders list is quite large (and growing all the time), you may want to have more room to see it. You can do this by resizing the areas within the Messages window, as explained on the next page.



mail (Personal mail, 1 of 1 new)
official (Has New Messages)
official.andrew (Has New Me

✓ 17-Aug-87 *Welcome to Andrew* (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu
To: newusers@andrew.cmu.edu (N
Subject: Welcome to Andrew

Welcome to the Andrew system, and esp
boards on Andrew.

Please take the time to explore the many facets of electronic communication available to you on Andrew. Using the Andrew

Resizing Areas in the Messages Window

To resize Messages window areas:

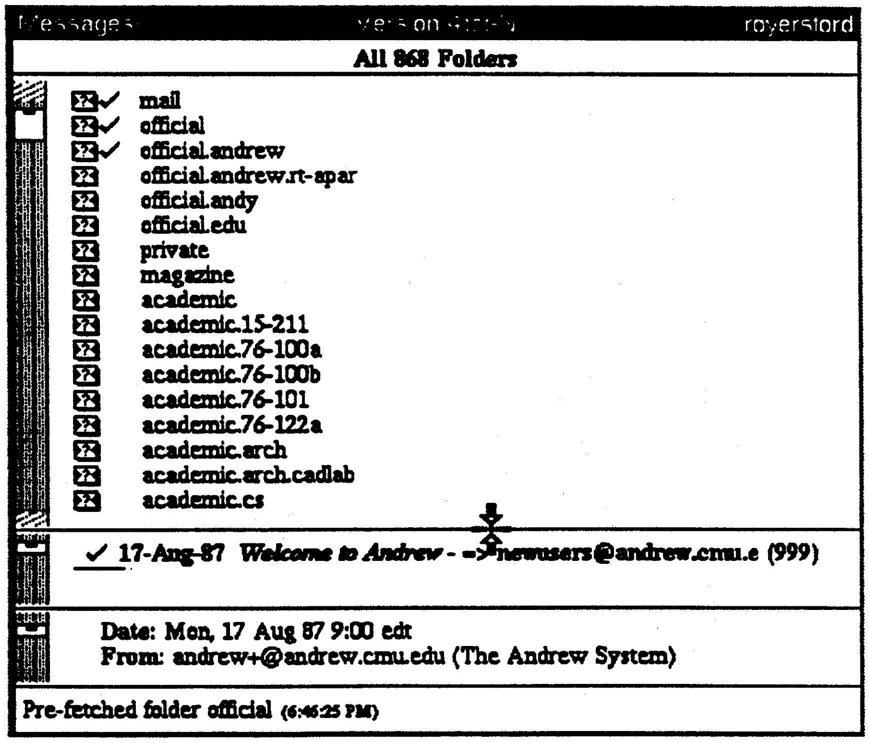
Action. Position the mouse cursor on the boundary between the folders and captions areas of the Messages window.

Response. The cursor becomes a boundary cursor.

Action. Press either the left or right mouse button down. Holding the button down, drag the boundary cursor down to where you would like the new boundary to be. Release the mouse button.

Response. The Messages window redraws with the areas reproportioned to where you released the mouse button.

You can resize any of the areas (up or down) as often as you like, at any time. Even the message area at the bottom of the window can become larger.


Messages version [illegible] royersford

All 868 Folders

✓ mail
✓ official
✓ official.andrew
official.andrew.rt-apar
official.andy
official.edu
private
magazine
academic
academic.15-211
academic.76-100a
academic.76-100b
academic.76-101
academic.76-122a
academic.arch
academic.arch.cadlab
academic.cs

✓ 17-Aug-87 *Welcome to Andrew* - ->newusers@andrew.cmu.e (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu (The Andrew System)

Pre-fetched folder official (6:46:25 PM)

Searching in the All Folders List

There are three ways to search through the All Folders list. You can use the scrollbar, which is like thumbing through all the folders. You can use use options on the *Search* menu card, which is like thumbing for a specific word. Or, you can use the **Read By Name** option on the *Message Folders* menu card, which is like pulling out and opening exactly the folder the you want to see. To see the help document containing explanations of *Search* options and **Read By Name**, type **help messages** in the Typescript.

To see the help document containing an explanation of the structure and organization of the bboards (how they are arranged in the huge filing cabinet), type **help bboards** in the Typescript.

To scroll through the All Folders list:

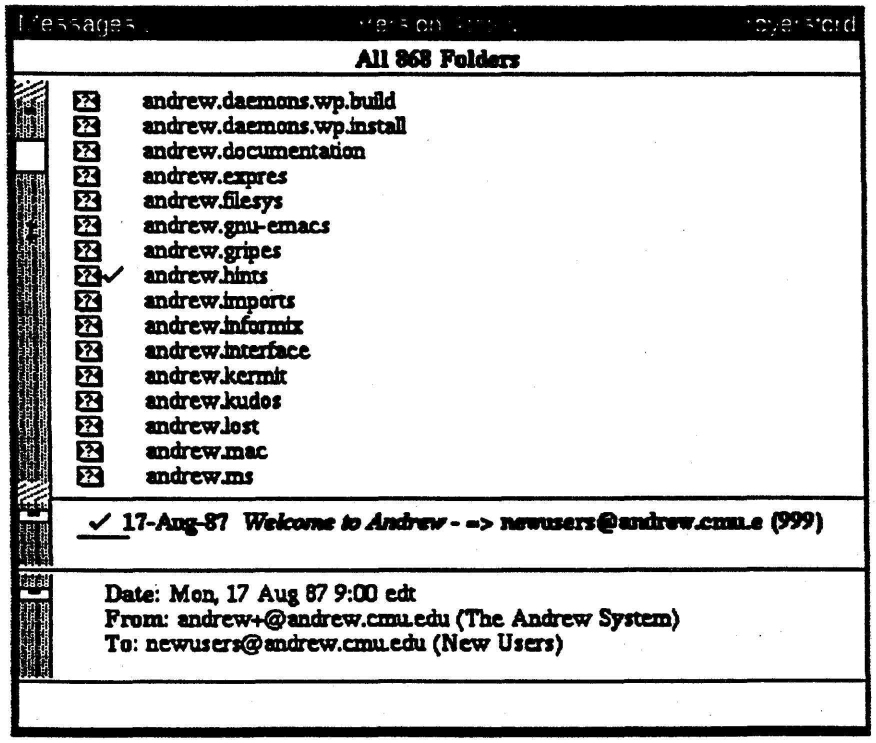
Action. Move the mouse cursor into the scroll bar next to the folders area. Left click until you see the folder

andrew.gripes

in the folders area.

Response. The All Folders list scrolls up.

Notice that the folders to which you are subscribed have a check mark next to them.



All 868 Folders

```
andrew.daemons.wp.build
andrew.daemons.wp.install
andrew.documentation
andrew.expres
andrew.filesys
andrew.gnu-emacs
andrew.gripes
✓ andrew.hints
andrew.imports
andrew.informix
andrew.interface
andrew.kermit
andrew.kudos
andrew.lost
andrew.mac
andrew.ms
```

✓ 17-Aug-87 *Welcome to Andrew* - => newusers@andrew.cmu.e (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu (The Andrew System)
To: newusers@andrew.cmu.edu (New Users)

Reading Messages in Folders

Reading messages in folders in the All Folders list is just like reading messages in folders from your stack of folders with new messages, or from your own subscription list. First you select the folder to read, and then you select messages in the folder.

To select a folder to read:

Action. Now that you have found andrew.gripes, move the mouse cursor on to the name "andrew.gripes" and left click.

Response. Andrew.gripes becomes selected and its captions appear in the captions area.

If a dialogue box appears, you clicked on the help icon. Choose **See the messages** from the dialogue box. Notice that you can see captions by either clicking on the help icon or the name of a folder. It does not matter which way you select a folder, but most people prefer to click on the name because is one less step than clicking on the help icon.

To select messages in the folder:

Action. Just as you selected your initial piece of mail, and a message on official.andrew, move the mouse cursor on to a caption and left click.

Response. The caption becomes selected, and the message appears in the body area.

To see old messages, scroll up in the captions area. Scrolling up in the captions area takes a little longer than scrolling down because Messages has to fetch the old captions before it presents them.

Now that you know how to find and see messages on different bulletin boards, you may want to look around at messages in other folders and then subscribe and unsubscribe to some bboards. Subscribing and unsubscribing is explained in the following sections.

Subscribing to a Folder

In this section, you can subscribe to the bulletin board folder "andrew.gripes." If you do not really want to subscribe to andrew.gripes, do not worry. You can always unsubscribe to it.

To subscribe to a folder:

Action. Click with the left mouse button on the help icon next to the folder called andrew.gripes.

Response. The dialogue box appears.

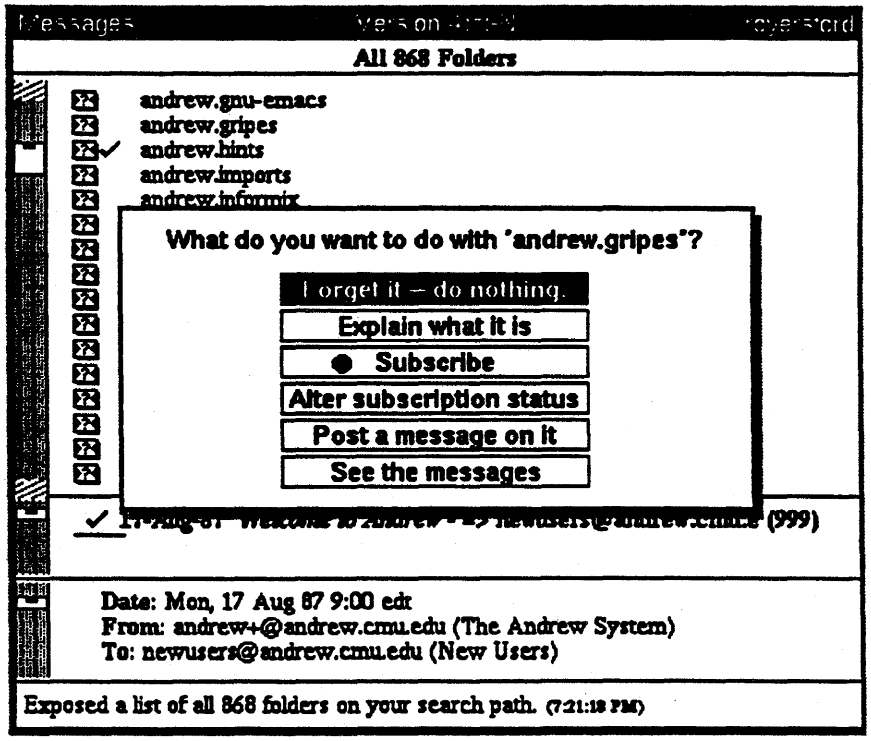
Action. Move the mouse cursor into the box that says Subscribe, and left click.

Response. Two important things happen:

- A check appears next to the folder.
- A message appears in the messages area informing you that you are subscribed to andrew.gripes.

If you click anywhere other than in a specific box, Messages takes the default (darkened) action: Forget it -- do nothing.

Clicking in the folders or captions region will also often select a folder or caption, in addition to making the dialogue box disappear.



✓ (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu (The Andrew System)
To: newusers@andrew.cmu.edu (New Users)

Exposed a list of all 868 folders on your search path. (7:21:18 PM)

Unsubscribing to a Folder

In this section, you can unsubscribe to the bulletin board folder andrew.gripes. If you really do want to be subscribed to andrew.gripes, do not worry. You can always subscribe to it, again.

To unsubscribe to a folder:

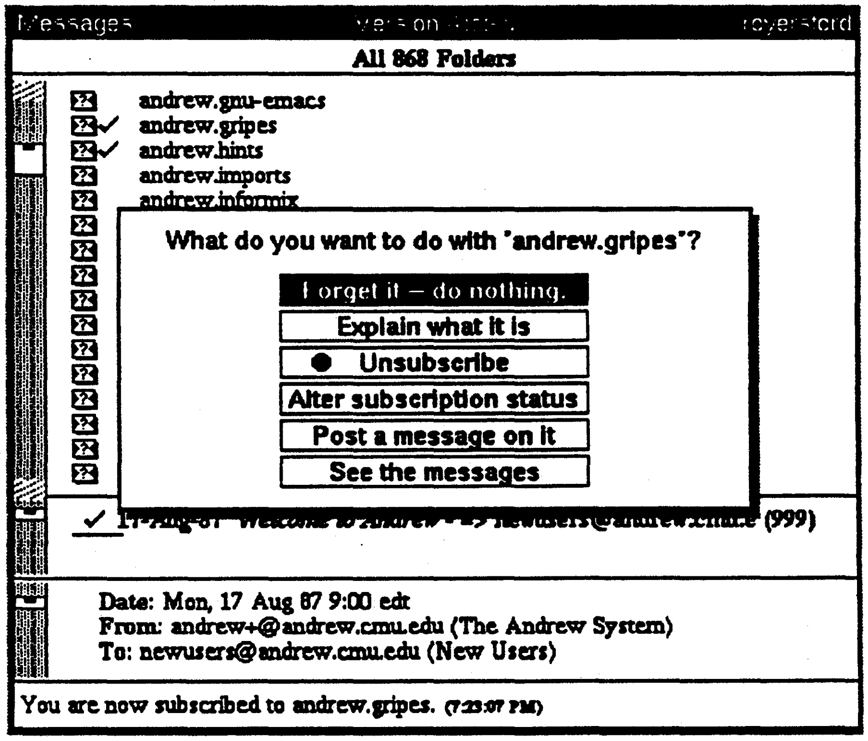
Action. Click on the help icon next to andrew.gripes.

Response. The dialogue box appears.

Action. Move the mouse cursor into the box that says Unsubscribe, and left click.

Response. Two important things happen:

- The check mark next to the folder name disappears.
- A message appears in the message area informing you that you are not subscribed to andrew.gripes.



andrew.gnu-emacs
andrew.gripes ✓
andrew.hints ✓
andrew.imports
andrew.informix

✓ 17-Aug-87 Welcome to Andrew - => newusers@andrew.cmu.e (999)

Date: Mon, 17 Aug 87 9:00 edt
From: andrew+@andrew.cmu.edu (The Andrew System)
To: newusers@andrew.cmu.edu (New Users)

You are now subscribed to andrew.gripes. (7:23:07 PM)

Exposing Your Subscription List

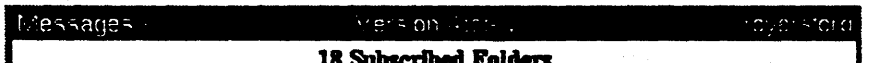
Your personal filing cabinet can grow or shrink as large or small as you want it, so there is no limit on the number of folders you can have in your subscription list.

Notice in the illustration below of a subscription list that the bulletin boards are subscribed to in a relatively "top-down" manner. This means, for example, that if you subscribe to ext.nn.announce.newusers, you should also subscribe to ext.nn.announce. You should subscribe like this because when a new bulletin board is created, its creation is announced on the bboad "above" it. This way, you won't miss the creation of a bboard that is potentially of interest to you.

To see your subscription list:

Action. Choose **Expose Subscribed** from the ***Message Folders*** pop-up menu card.

Response. A list of all the folders you are subscribed to appears in the folders area of the Messages window.



Messages [illegible]

18 Subscribed Folders

```
✓ mail (Personal mail, 1 of 1 new)
✓ official
✓ official.andrew (Official Bboard, 4 of 362 new)
✓ andrew
✓ andrew.hints (Local Bboard, 10 of 862 new)
✓ andrew.ms (Local Bboard, 16 of 1655 new)
✓ hobbies (Local Bboard, 3 of 4 new)
✓ hobbies.cooking (Local Bboard, 7 of 8 new)
✓ magazines (Local Bboard, 9 of 10 new)
✓ magazines.humor.ckk (Local Bboard, 10 of 629 new)
✓ market (Local Bboard, 10 of 206 new)
✓ ext.cc (External Bboard, 3 of 3 new)
✓ ext.cc.cooking (External Bboard, 0 of 1 new)
✓ ext.cc.registrar-information (External Bboard, 3 of 4 new)
✓ ext.nn.news (External Bboard, 0 of 1 new)
✓ ext.nn.news.announce.newusers (External Bboard, 1 of 18 new)
✓ ext.nn.rec.arts.startrek (External Bboard, 42 of 887 new)

✓ 27-Jul-87 Re: African cookbook query - Miriam Nadel@gryphon.CTS (900)

From: mhnadel@gryphon.CTS.COM (Miriam Nadel)
Subject: Re: African cookbook query

You are now subscribed to ext.nn.rec.arts.startrek. (9:32:29 AM)
```

Quitting Messages

To quit Messages:

Action. Move the mouse cursor into the Messages window, and choose **Quit** from the front pop-up menu card.

Response. The Messages window disappears.

If you have deleted messages during the session with Messages, a dialogue box appears when you quit. The box asks if you want to purge (permanently remove) deleted messages.

If you answer **yes**, the deleted messages will be purged (permanently removed), and the program will quit.

If you answer **no**, the deleted messages will remain deleted, but not be purged (permanently removed), and the program will quit.

If you answer **do not quit**, the program will not quit and the deleted messages will not be purged (permanently removed).

Getting Help for Other Mailing Needs

To customize Messages and SendMessage:

Messages and SendMessage have lots of options that are easily set and easily undone. To turn on options, you must do a sequence of things. First, you choose **Set Options** from the back menu card in Messages. Then you can read the help that appears in the body region of the Messages window, and choose dialogue boxes according to which options you want to try.

Some of the more popular options are marking of messages, folder manipulation features and options, the Punt button (in **Display Format Options**), and showing the folders area next to the captions area (also in **Display Format Options**). See the help containing a list of the available options by typing **help messages-customization** in the Typescript.

To send yourself blind carbon copies (bccs) of messages:

See the help on SendMessage by typing **help sendmessage** in the Typescript.

To send mail to users off campus:

See the help on computer networks by typing **help networks** in the Typescript.

To have mail arriving at Andrew sent to other machines:

See the help on forwarding by typing **help forward** in the Typescript.

To simplify addressing mail to people you mail to frequently:

See the help on mail aliases by typing **help aliases** in the Typescript.

To send mail to groups of people:

See the help on mailing and distribution lists by typing **help mailinglists** in the Typescript.

To find out more about how the Andrew message system works:

See the help on the Andrew message system by typing **help ams** in the Typescript. You can also read the bulletin board "andrew.ms" for an on-going discussion of the developing Andrew message system.

Programming With the Xrlib User Interface Toolbox

February 1988

Contents

This page left blank intentionally.

1 Introduction To Xrlib

1.1 How To Use This Manual

This manual is for programmers who intend to use the X window system and the Xrlib development tools to create application programs. The manual contains the following types of information:

- An introduction to the capabilities of the Xrlib user interface library (chapter 1).
- Descriptions of each level of functionality and how to incorporate them into your programs (chapters 2 through 5).
- Man page reference information (appendix A).
- Porting X10 Xrlib Applications to X11 (appendix B).

You should read the rest of this chapter to become familiar with the relationships among the levels of functionality provided by these libraries and to realize that you'll need to build your understanding starting at the lowest levels.

Using the X window system is a matter of using the X library and, if desired, the Xrlib library or other toolboxes. The X11R library (libX11R.a) is described here, while the X11 library (libX11.a) is described in a separate document entitled: 'Xlib - C Language Interface, Protocol Version 11.'

This chapter contains an overview of the X11 and X11R libraries and some examples of their use.

1.2 The X11 Library

The X11 library is the low-level interface to the X server protocol. It contains more than 250 subroutines that control and manipulate windows. The X11 library provides the foundation for the higher-level functionality of the X11R library and provides low-level control of the user interface.

The following paragraphs describe some of the operations provided by the X11 library.

Accessing the Display. An advantage of X is its ability to control windows over a network. The application program can control which display it uses for output and can terminate that connection when necessary.

Creating Windows. Windows can be created and destroyed as required by the program. A program can have several windows active at one time. In addition, the program can define the cursor associated with each window.

Changing Windows. The characteristics of a window can be controlled by the program. For example, a window can be moved, resized, moved to the top or bottom of a "stack" of windows, and have its background pattern changed.

Performing Graphics Operations. Graphics operations provided by the X11 library enable the program to draw lines and polygons, fill areas with patterns, copy and move regions of the window, and perform raster graphics operations. In addition, the program can control the colors of the graphic elements and background.

Working With Text. The program can place text in a window using desired fonts. Fonts are not restricted to alphanumeric characters.

Using the Cut Buffers. Eight cut buffers enable a program to save and retrieve data from a window, allowing cut-and-paste operations.

Accepting User Input. User input can take several forms, such as keyboard input, mouse pointer movement, and mouse button input. The program can define the types of input events that it cares about so that it can respond in the appropriate manner.

Customizing the Operation. Certain user features can be set by the program, including cursor speed in relation to mouse pointer movement, the keyboard's auto-repeat operation, the keyboard's caps-lock operation, and the display behavior after a period of inactivity.

The X11 library is described in detail in a separate document, entitled 'Xlib - C Language Interface, Protocol Version 11.'

1.3 The X11R Library

The Xrlib user interface library is a set of powerful programming tools that enables you to create a graphical user interface for your application programs. The user interface becomes much more than prompts and keyboard input. It becomes a more natural "point-and-select" interaction that makes programs easier to use. It's also a tool that can make your programs compatible with virtually any system that implements the X window system.

Several naming conventions are used in the X11R library descriptions to help you keep track of the functions, structures, and data types you use. The following table describes those conventions using the sample name of FOO:

Name	Description
XFoo	An X function or routine.
XrFoo	An Xrlib function or routine.
XrFOO	An Xrlib define.
xrFoo	An Xrlib data type.

You can think of the X11 and X11R libraries as providing four levels of programming functionality, each building upon the capabilities of the lower levels:

- Dialogs (the highest level).
- Field Editors.
- Intrinsics.
- X11 library primitives (the lowest level).

Dialogs, field editors, and intrinsics are provided in the X11R library–sometimes called a *toolbox*. The X11 library primitives are provided by the X11 library.

In the following sections of this chapter, each of these levels is described briefly, starting with the highest, most powerful level. This should give you an overview of the capabilities of these levels of functionality, highlighting the most powerful capabilities first.

When you actually begin learning the details of programming with the X window system (after you finish this chapter), you'll have to start learning at the lower levels and build upon that knowledge in order to program at the higher levels. For this reason, the remainder of this manual describes Xrlib starting at the lowest levels of functionality and proceeding to the highest.

1.3.1 Dialog Level

Dialogs provide the highest level of communication with the end-user. Xrlib provides three types of dialogs:

- Menu. For presenting and selecting one user choice out of many that may be available.
- Message box. For presenting a message or simple choice to the user.
- Panel. For presenting several types of information in interactive forms and fields and collecting user input.

Each type of dialog presents to the user a separate window that's dedicated to communication with the user. Each of these dialog types are summarized below.

Menus. Menus allow users to view the choices available to them at any one time without having to remember command words or special keys or even select an option. In Xrlib, menus are lists of items (text strings, icons, etc.) displayed in a pop-up window. A menu is displayed with selectable items in bold and unselectable items in gray. Selecting a menu item involves pointing at an item, which highlights it, and then selecting it. The selection process depends upon the human interface model. This process and others are configurable by the user. (For more information about configurations, refer to chapter 4, "Dialogs.")

The selection information is returned to the application through the input *XrInput*() model, which is described in chapter 2.

You can view the information base of an Xrlib menu as a tree. The leaves of the tree represent commands or options. The branches of the tree are paths to sub-menus. A sample menu is shown in figure 1-1.

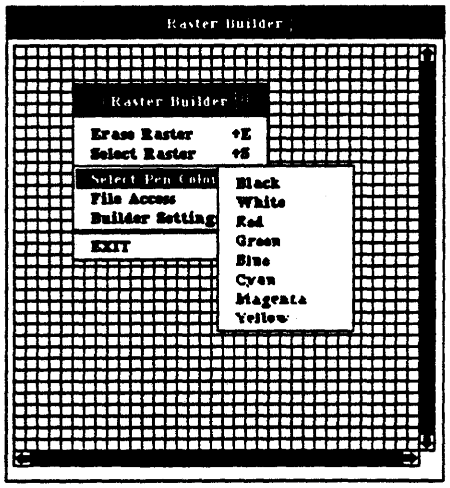
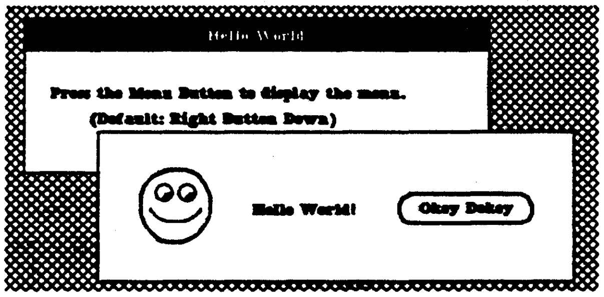


Figure 1-1. Sample Menu

Message Boxes. Many times a program must warn the user or ask a simple multiple choice question. Message boxes provide this capability in an easy-to-use format. When a message window appears on the display, the user must respond to the question or statement displayed by selecting one of the choices.

A message box is made up of an icon (a symbol), some descriptive text, and buttons. Icon placement, text formatting, and button placement is performed automatically by the message box manager. If you need to perform a more complex interaction and have more control over the appearance of the message box, you should use a panel. A sample message box is shown in figure 1-2.



Press the Menu Button to display the menu.
(Default: Right Button Down)

Hello World!

Figure 1-2. Sample Message Box

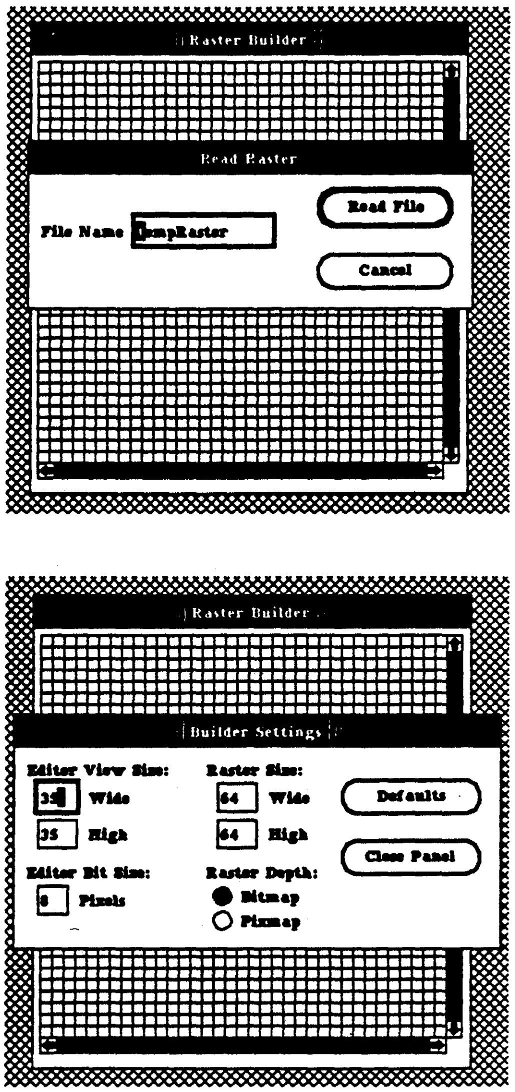
Panels. Panels are generally used to display the state of an application and allow the user to change that state. A panel is a collection of *field editors*, which can manipulate the state of an application, contained in a single window. Field editors provide communication with the user in particular "formats" and are described below.

When a program displays a particular panel, the panel manager sends messages to the field editors in the panel instructing them to create *instances* of themselves, to display themselves, to make themselves active, and so on.

When the user must make changes to the panel, such as entering text or checking a check box, the panel manager passes control to the appropriate field editor. When the desired changes have been made, there is often a button labeled "OK" that signals the panel manager to return control to the application. The changes made are noted by the application, which makes appropriate changes to its own data structures.

The Xrlib concept of what can be in a panel is not limited to just letters and numbers. Because panels use editors to manage their fields, a field can contain almost any kind of information. (Refer to "Field Editor Level" below.) This concept increases your programming ease since many more difficult-to-program user interactions are easily expressed by a panel.

Many applications can be designed easily as panels because Xrlib provides a robust panel structure and an extensible collection of editors to manage fields. Two sample panels are shown in figure 1-3. (For more information about dialogs, refer to chapter 4.)



File Name TempRaster

Editor View Size: 35 Wide 35 High

Editor Bit Size: 8 Pixels

Raster Size: 64 Wide 64 High

Raster Depth: ● Bitmap ○ Pixmap

Figure 1-3. Sample Panels

1.3.2 Field Editor Level

Field editors provide several formats for user input. (Field editors are described in Chapter 4. The chapter also includes illustrations showing each type of field editor.) Each of the following field editors supports one format of user input:

- The titlebar editor creates a "title bar" (with a title and gadget boxes) at the top of a window. See figure 1-4 below.
- The scrollbar editor manages a "scroll bar" that controls scrolling within the window.
- The checkbox editor manages an array of labeled checkboxes with predefined meanings – several can be active at one time. See figure 1-5 below.
- The radio-button editor manages an array of labeled buttons with predefined meanings – only one can be active at one time.
- The push-button editor manages an array of labeled bars that perform predefined actions when selected. See figure 1-6 below.

- The text editor manages a text-input field in which the user may enter or revise a single line of text.
- The static-text editor displays a block of predefined text.
- The raster-select editor manages an array of raster samples – only one can be active at one time. See figure 1-7 below.
- The raster editor manages a raster-image field in which the user may revise a raster pattern.
- The static-raster editor displays a predefined raster pattern, often designed as a symbol.

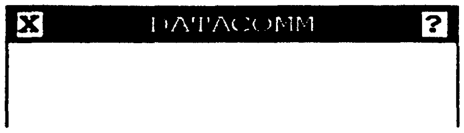


Figure 1-4. Sample Titlebar Editor

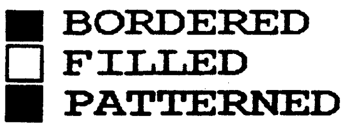


Figure 1-5. Sample Checkbox Editor

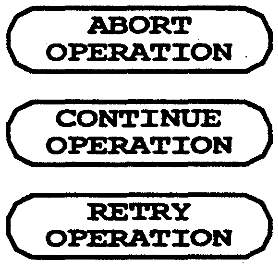


Figure 1-6. Sample Push-Button Editor

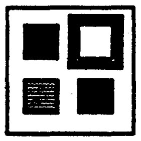


Figure 1-7. Sample Raster-Select Editor

Each of these editors provides and manages a specific format of user interface. When a field editor instance is created by an application program, the editor takes over a rectangular region of a window and displays the instance therein. The editor then processes all keyboard and mouse pointer events that occur within this region.

Whenever an editor instance is modified by a user's action, such as altering the slide box position in a scrollbar instance, the editor returns a block of information to the application program, notifying it of the change. This allows an application to then take any actions it might require regarding the new value of the field editor instance.

An application can also modify the view and behavior of a field editor instance as a whole.

Editors are most often used as the building blocks for creating higher-level user-interface constructs, such as dialogs (panels and message boxes). However, the use of editors is in no way limited to just this. You may incorporate field editors into an application in whatever way you need. In addition, if no editor provides the functionality needed by a program, you can design your own editor, which you can then integrate into the X11R library.

(For more information about field editors, refer to chapters 3 and 5)

1.3.3 Intrinsics Level

Xrlib intrinsics are the lowest level functions and structures of the X11R library. The Xrlib intrinsics provide an intermediate level of control of the system by combining and enhancing the X11 library primitives. The intrinsics provide the base level of tools needed to build a very usable human interface.

The intrinsics provide four main areas of functionality:

- **Input Abstractions.** The intrinsics take control of processing all input and place it into one input stream. Types of input include normal keyboard input, mouse pointer and window input, and special types of input provided by the higher-level functions. By providing this method of input, the application gathers input from one source only, and Xrlib provides the control routines needed to handle the higher levels. The intrinsics can handle any type of input passed by X.
- **Resource Management.** The intrinsics provide an extensible resource manager that allows an application to maintain and access resources. Many types of resources are defined by Xrlib, and the application can create its own special types.
- **Editor Interaction.** Each window can have a set of field editors logically attached to it. This pairing of a window to its editors is needed to allow the "transparent" transfer of control to the editors. The intrinsics provide for this capability through two entry points that can handle individual editors or groups of editors.
- **Geometric Utility Routines.** The intrinsics provide a large set of routines to handle the structures that describe drawn figures. The routines can manipulate points and rectangles, performing operations such as the union and intersection of rectangles.

(For more information about intrinsics, refer to chapter 2.)

1.4 Using the X Window System in an Application

The X11 and X11R libraries reside in the /usr/lib directory. Each has an associated include file which normally is included by any application that uses the libraries.

```
/usr/include/X11/Xlib.h
/usr/include/Xr11/Xrlib.h
```

Before you compile an application, include the following lines at the beginning of each source module that uses any of the the X window system structures or defines. (If you use only X, you need to include only the Xlib.h file. If you use either of the other two libraries, you need to include the Xlib.h file, as well as the Xrlib.h include file.)

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
```

After each module of the application has been successfully compiled, link the modules with the X11 and X11R libraries. This involves adding the following options to the *cc* or *ld* commands, which is the mechanism for linking modules:

```
-lX11R -lX11
```

This option instructs the loader to use the routines present in the library files /usr/lib/libX11R.a, and /usr/lib/libX11.a to resolve any undefined globals and procedures.

Note

The last option must be "-lX11" when using either library. Also note that if an application does not use the X11R library, the "-lX11R" line and options aren't required above.

1.5 Building A Program Using X

To conclude this overview of X, it might be helpful to look at some simple *Hello World* programs. The programs demonstrate the steps necessary to open a window for input and display subwindows. The first program shows how to open a window and display a message in it using the X11 library. The following two programs build upon the first using the X11R library. The last program displays a menu and accepts input from either the keyboard or mouse pointer. With these four examples, you should be able to get an idea of the capabilities and uses of both the X11 and X11R libraries. (The following examples don't use panels. However, you can find examples of programs that use panels in the /usr/contrib/RB directory.)

1.5.1 Sample Program 1

This first sample program uses only the X window library – it doesn't use the X11R library. This simple output-only program creates a window and prints a message in that window. The program does not repaint the message if the message is covered by another window and then exposed. The numbered comments and their corresponding lines of code in the program refer to numbered paragraphs at the end of the program.

```
/*
 *   Hello World using X
 */
#include <X11/Xlib.h>

main()
{
    Display *display;
    int screen;
    GC gc;
    Window windowId;
    Font f;
    unsigned long border, background;
    XSetWindowAttributes wAttribs;

                                    /*
1.  Open the display and default screen specified by the environment
    variable DISPLAY.
                                */
```

```
        if ((display=XOpenDisplay (0)) == 0)
            {
            printf ("cannot create a window on %s\n", getenv ("DISPLAY"));
            exit (1);
            }

        screen = DefaultScreen (display);
        border = BlackPixel (display, screen);
        background = WhitePixel (display, screen);

                                        /*
    2.  Create a window and put it on the display.
                                        */

        windowId = XCreateSimpleWindow (display, RootWindow(display, screen),
                                        50, 50, 400, 200, 3, border, background);

        wAttribs.override_redirect = 1;
        XChangeWindowAttributes (display, windowId, CWOverrideRedirect, &wAttribs);

        XMapWindow (display, windowId);


                                        /*
    3.  Create a Graphics Context.
                                        */

        gc = XCreateGC (display, windowId, 0, 0);


                                        /*
    4.  Display a string. Send "Hello World" to the window.
                                        */

        XDrawString (display, windowId, gc, 100, 80, "Hello World", 11);


                                        /*
    5.  Flush the output to the window.
                                        */

        XFlush (display);

                                        /*
    6.  Sleep, Close Display, and exit.
                                        */

        sleep (10);
        XCloseDisplay (display);

    }
```

1. **Open the Display.** Using the Xlib function *XOpenDisplay*(), you establish a connection to the X server for an accessible display. You need to do this because the X window system is a network service. Since your program is initially independent of a display, it must establish a connection to a display. Using the zero argument to *XOpenDisplay*() indicates that the default display should be determined from the environment variable DISPLAY. The DISPLAY environment variable should be in the format "hostname:display_number.screen_number", where hostname is the name of the machine where the window is to be created, display_number determines the display and screen_number determines the screen to be used on that machine. *XOpenDisplay*() returns

a zero if it is unable to establish a connection. (Refer to chapter 2 under "Opening and Closing the Display" for more information about the *XOpenDisplay*() function.)

2. **Create a window.** Now that a connection is established to the target display, the example creates a child window of the *RootWindow*. Many window attributes are set through this function, including the window's dimensions, location with respect to the *RootWindow*, border width, and colors. Windows are not displayed upon window creation, (that is, they are not "mapped"). Any information sent to the window while it is not mapped will not be displayed.

   XCreateWindow() returns the window id of the created window on success, or zero on failure. The window id is used in most calls to identify a particular window. (Refer to chapter 2 under "Creating and Destroying Windows" for more information about the *XCreateWindow*() function.)

   The call to *XMapWindow*() requests that the X server display ("map") the window identified by the window id argument. It is interesting to note that the example program does not try to test *XMapWindow*() for a returned failure condition. This is indicative of network window calls that return after the X client has queued the function request, but possibly before the server has fulfilled the request. (Refer "Window Functions" in the 'Xlib - C Language Interface, Protocol Version 11' document for more information about the *XMapWindow*() function.)

   The window server may be on a remote machine. If the server fails to fulfill an X window request, then the failure condition may not be detected until a future blocked ("synchronous") function call.

3. **Create a Graphics Context** *XCreateGC*() creates a graphics context. This example creates a graphics context that is set to all default values. A graphics context is required for all X Windows functions that draw either text or graphics to a window. (See "Graphics Resource Functions" in the 'Xlib - C Language Interface, Protocol Version 11' document for more information.)

4. **Display a string.** The call to *XDrawString*() requests that a string be displayed in the window at a particular location using the attributes from the graphics context.

5. **Flush the output to the window.** The call to *XFlush*() sends all output requests to the appropriate X server.

6. **Sleep and then terminate.** The call to *sleep*() is made in order to keep the window visible on the screen for a short while. *XCloseDisplay*() performs cleanup activities and disconnects the display.

1.5.2 Sample Program 2

This example adds a simple title bar to the first "Hello World" example. Again, the numbered paragraphs following the code correspond to the numbered comments. Only code that has been added to the first example is described.

Note that title bars are an X11R library facility. Thus we now need to begin using the X11R library. In general the Xrlib functions are layered upon the X window library. Xrlib provides a set of routines that can greatly enhance the user interface of your programs.

```
/*
 *   add a Title Bar to Hello World
 */
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

/*
 * fill a structure to define a Title Bar
 */
```

```
xrTitleBarInfo titleBarInfo =
{
    0,                         /* windowId */
    {0,0,0,0},                 /* editor rectangle */
    XrVISIBLE | XrSENSITIVE,   /* editor state */
    -1, -1,                    /* editor colors - use defaults */
    NULL,                      /* editor font - use defaults */
    "Hello World",             /* Title name */
    NULL,                      /* gadget1 - unused */
    NULL,                      /* gadget2 - unused */
};

xrWindowData windowData;


main()
{
    Display *display;
    int screen;
    Window windowId;
    XSetWindowAttributes wAttribs;



/* Open the display specified by the environment variable DISPLAY */

    if ((display=XOpenDisplay (0)) == NULL)
        {
        printf ("Cannot open display: %s\n", getenv ("DISPLAY"));
        exit (1);
        }

    screen = DefaultScreen (display);



                                /*
1.  Initialize Xrlib
                                */

    if (XrInit (display, screen, NULL) == FALSE)
        {
        printf ("Could not initialize Xrlib\n");
        exit (1);
        }



/* Create a window and put it on the display */

    windowId = XCreateSimpleWindow (display, RootWindow(display, screen),
                                    50, 50, 400, 200, 3,
                                    BlackPixel(display, screen),
                                    WhitePixel(display, screen));

    wAttribs.override_redirect = TRUE;
    XChangeWindowAttributes (display, windowId, CWOverrideRedirect, &wAttribs);

    XMapWindow (display, windowId);



                                /*
    2.  Register this window with Xrlib.
                                   */

        XrSetRect (&windowData.windowRect, 50, 50, 400, 200);
        windowData.foreTile = BlackPixmap;
        windowData.backTile = WhitePixmap;

        XrInput (windowId, MSG_ADDWINDOW, &windowData);



                                   /*
    3.  Draw the Title Bar.
                                   */

        titleBarInfo.editorWindowId = windowId;
        XrTitleBar (NULL, MSG_NEW, &titleBarInfo);



    /* write the hello world string into the window */

        XSetForeground (display, xrEditorGC1, BlackPixel(display, screen));
        XDrawString (display, windowId, xrEditorGC1, 100, 80, "Hello World", 11);



    /* Send all of those instructions to the X window server to process */

        XFlush (display);



    /* Sleep, close display, and exit */
        sleep (5);
        XCloseDisplay (display);

    }
```

1. **Initialize the X11R Library.** Initialize the X11R library using the *XrInit*() routine. *XrInit*() returns FALSE upon failure. (Refer to chapter 2 for more information about initializing Xrlib.)

2. **Register Your Window With Xrlib.** Windows to be used with the X11R library need to be "registered" with the X11R library. A window becomes registered through the *XrInput*() intrinsic using the MSG_ADDWINDOW message. (MSG_ADDWINDOW is described in chapter 3 under "Input Messages.")

   This call illustrates how a programmer interacts with the X11R library. Generally the function of a library call is determined by the Xrlib function called and the message sent as an argument. Additionally, Xrlib functions most often use pointers to structures to communicate other details to the calling program.

   Notice that it is not necessary to explicitly get a font for this window. The X11R library establishes a default font for you. Of course, you can choose your own font and then refer to it through its appropriate font id.

3. **Draw a Title Bar.** The call to *XrTitleBar*() draws a title bar in the window. A title bar is typically a rectangular banner across the top of a window displaying the name associated with the window. When *XrTitleBar*() is called with the MSG_NEW message, it also expects the first argument to be NULL. The third argument, a pointer to a xrTitleBarInfo structure, conveys other details to

Xrlib about the requested title bar attributes. (Refer to the *XRTITLEBAR*() manual page in appendix A and to chapter 3 for more information about the Titlebar Editor.)

1.5.3 Sample Program 3

This third example adds a simple message box to our growing "Hello World" program. A message box is a simple type of dialog that a user may have with an application. This example adds the use of message boxes and user input upon the previous examples. Once again, the numbered comments and their associated code correspond to the numbered paragraphs at the end of the program. Only new code sections are described.

```
/*
 * add a Message Box to Hello World
 */
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
#include <time.h>

/*
 * fill a structure to define a Title Bar
 */
xrTitleBarInfo titleBarInfo =
{
   0,                         /* windowId */
   {0,0,0,0},                 /* editor rectangle */
   XrVISIBLE | XrSENSITIVE,   /* editor state */
   -1, -1,                    /* editor colors; use defaults */
   NULL,                      /* editor font; use defaults */
   "Hello World",             /* Title name */
   NULL,                      /* gadget1; unused */
   NULL                       /* gadget2; unused */
};

xrWindowData windowData;
xrEditor    * t_bar;
xrEvent       messageEvent;
xrMsgBoxInfo msgInfo;
INT8        * myButtons[] = {"Ready", "Not Ready"};


main()
{
   Display * display;
   int       screen;
   Window    windowId;
   XSetWindowAttributes wAttrs;

   /* Open the display specified by the environment variable DISPLAY */

      if ((display = XOpenDisplay(0)) == NULL)
      {
         printf ("Cannot open display: %s\n", getenv("DISPLAY"));
         exit (1);
      }

      screen = DefaultScreen(display);


   /* Initialize Xrlib */
   if (XrInit (display, screen, NULL) == FALSE)
   {
      printf ("Could not initialize Xrlib\n");
      exit (1);
   }


/* Create a window and put it on the display */

   windowId = XCreateSimpleWindow (display, RootWindow(display, screen),
                                   50, 50, 400, 200, 3,
                                   BlackPixel(display, screen),
                                   WhitePixel(display, screen));
   wAttrs.override_redirect = TRUE;
   XChangeWindowAttributes (display, windowId, CWOverrideRedirect, &wAttrs);
   XMapWindow (display, windowId);


/* Associate this window with Xrlib functionality */

   XrSetRect (&windowData.windowRect, 50, 50, 400, 200);
   windowData.foreTile = BlackPixmap;
   windowData.backTile = WhitePixmap;
   XrInput (windowId, MSG_ADDWINDOW, &windowData);


/* Draw the Title Bar */

   titleBarInfo.editorWindowId = windowId;
   t_bar = XrTitleBar (NULL, MSG_NEW, &titleBarInfo);



                                                        /*
1. Select the type of input for the window.
                                                         */

   XSelectInput (display, windowId, ButtonPressMask | ButtonReleaseMask |
                                    KeyPressMask);


                                                        /*
2. Create and Display a message box
                                                         */

   msgInfo.messageOrigin.x = 50;
   msgInfo.messageOrigin.y = 50;
   msgInfo.relativeTo = windowId;
   msgInfo.messageText = "Are you ready?";
   msgInfo.messageButtons = myButtons;
   msgInfo.numButtons = 2;

   /*
    * Put up a message box; upon return, examine value1:
    *  value1 == 1, iff the user selected the 'not ready' button.
    *  value1 == 0, iff the user hit a key or selected the 'ready' button.
    *  value1 == -1, if the message box times out.
    */

   do
   {
      XrMessageBox (&msgInfo, MSG_EDIT, &messageEvent);
   }  while (messageEvent.value1 != 0);
```

```
                                                        /*
    3. Redraw the Title Bar.
                                                         */

       XrTitleBar (t_bar, MSG_REDRAW, XrREDRAW_ALL);

       /* Write hello world message to the window */

       XSetForeground (display, xrEditorGC1, BlackPixel(display, screen));
       XDrawString (display, windowId, xrEditorGC1, 100, 80, "Hello World", 11);

       /* Send all those instructions to the X window server to process */

       XFlush (display);

       /* Sleep, close the display, and then exit */

       sleep (5);
       XCloseDisplay (display);

}
```

1. **Select the type of input for the window.** In order to receive input from the window, you must specify the types of input you want. For this example we are going to respond to presses and releases of mouse buttons or keystrokes. We call *XSelectInput*() with a window id and a mask describing the desired events. (For more information about *XSelectInput*(), refer to "Event Handling" in chapter 2.)
2. **Create and display a message box.** The message box is created and placed on the display with the call to *XrMessageBox*() with the MSG_EDIT message. The *xrMsgBoxInfo* structure, *msgInfo*, describes the attributes of the requested message box. For example this box displays the string, "Are you ready?", along with two buttons, "Ready" and "Not Ready". The *xrEvent* structure, *messageEvent*, will contain the values returned by the dialog with the message box. In this example we examine the field *messageEvent.value1* to determine what response the user provided to the message box. (Refer to chapter 3 for more information about *Message Boxes.*)
3. **Redraw the title bar.**

Note

We need to redraw the title bar that had been occluded by the message box.

To redraw the title bar we need to call the *XrTitleBar*() routine using the MSG_REDRAW message. The pointer, *t_bar*, was obtained as the return value to the call to *XrTitleBar*() when the MSG_NEW message was used to create the title bar. (Refer to the XRTITLEBAR() manual page in appendix A for more information about the MSG_REDRAW message.)

1.5.4 Sample Program 4

As you have seen, the initial setup of an Xrlib program requires the use of Xlib functions and Xrlib intrinsics. Once some low-level tasks are performed, you can operate at the field editor and dialog levels of the Xrlib model.

The following program displays a menu and accepts input from either the keyboard or mouse pointer. It uses a user defined function to draw the window. (The include files smile.h and frown.h are in the /usr/contrib/HelloWorld11 directory.)

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
#include "HelloWorld.h"
#include "smile.h"
#include "frown.h"

GC        gc1, gc2;
Display * displayPtr;
int       screen;


main (argc, argv)
int argc;
char * argv[];
{
  xrWindowData windowData;
  XEvent  xinput;
  xrEvent * input;
  extern xrPFI DrawWindow();
  Pixmap pixmap;
  XImage * image;
  XSetWindowAttributes wAttr;

   input = (xrEvent *) & xinput;

                                        /*
   1. Open the display
                                        */

      if ((displayPtr = XOpenDisplay (argv[1])) == NULL)
      {
        printf ("Could not open the display\n");
        exit();
      }
      screen = DefaultScreen(displayPtr);


                                        /*
   2. Initialize Xrlib
                                        */

      if (XrInit (displayPtr, screen, NULL) == FALSE)
      {
        printf ("Could not initialize Xrlib\n");
        exit();
      }


                                        /*
   3. Create the window
                                        */

      wAttr.override_redirect = TRUE;
      wAttr.border_pixmap = xrWindowForeground;
      wAttr.background_pixmap = xrWindowBackground;
      if ((windowId = XCreateWindow (displayPtr,
           RootWindow(displayPtr, screen),
           150, 150, 375, 120, xrBorderWidth, DefaultDepth(displayPtr, screen),
           InputOutput, DefaultVisual(displayPtr, screen),
           CWBorderPixmap | CWBackPixmap | CWOverrideRedirect, &wAttr)) == 0)
      {
         printf("The window create failed\n");
         exit();
   }


                                          /*
4. Map the window. Make it appear on the display
                                          */

   XMapRaised (displayPtr, windowId);


                                          /*
5. Set the window's input
                                          */

   XrSetRect (&windowData.windowRect, 150, 150 , 375, 120);
   windowData.foreTile = xrWindowForeground;
   windowData.backTile = xrWindowBackground;

   XrInput (windowId,MSG_ADDWINDOW, &windowData);


                                          /*
6. Select the types of input for the window.
                                          */

  XSelectInput (displayPtr, windowId, ButtonPressMask | ButtonReleaseMask |
               KeyPressMask | ExposureMask);


                                          /*
7. Create the graphics context.
                                          */

   gc1 = XCreateGC (displayPtr, RootWindow(displayPtr, screen), 0, 0);
   _XrSetUpGC (gc1);
   XSetForeground (displayPtr, gc1, xrBackgroundColor);
   XSetBackground (displayPtr, gc1, xrForegroundColor);
   gc2 = XCreateGC (displayPtr, RootWindow(displayPtr, screen), 0, 0);
   _XrSetUpGC (gc2);
   XSetForeground (displayPtr, gc2, xrForegroundColor);
   XSetBackground (displayPtr, gc2, xrBackgroundColor);
   XSetFont (displayPtr, gc2, xrBaseFontInfo->fid);
   XSetGraphicsExposures (displayPtr, gc2, 0);


                                          /*
8. Draw the window
                                          */

   DrawWindow (windowId, MSG_NEW, NULL);


                                          /*
9. Define the cursor for the window
                                          */

   XDefineCursor (displayPtr, windowId, xrDefaultCursor);


                                          /*
10. Set up and create the menu
                                          */
   menu = XrMenu (NULL, MSG_NEW, &menuInfo);
   XrMenu (menu, MSG_ACTIVATEMENU, windowId);


                                        /*
11. Set up the rasters for message boxes from smile.h and frown.h
                                        */

   image = (XImage *)XCreateImage (displayPtr,
           DefaultVisual(displayPtr, screen),
           1, XYBitmap, 0, smile.raster, smile.width, smile.height,
           BitmapPad(displayPtr), 0);
   image->bitmap_unit = 16;
   image->bitmap_pad = 16;
   image->byte_order = LSBFirst;
   image->bitmap_bit_order = LSBFirst;
   pixmap = XCreatePixmap (displayPtr, RootWindow(displayPtr, screen),
                           smile.width, smile.height,
                           DefaultDepth(displayPtr, screen));
   XPutImage(displayPtr, pixmap, gc1, image, 0,0,0,0, smile.width,
             smile.height);
   niceMsgInfo.rasterId = pixmap;
   niceMsgInfo.rasterWidth = smile.width;
   niceMsgInfo.rasterHeight = smile.height;


   image = (XImage *)XCreateImage (displayPtr,
           DefaultVisual(displayPtr, screen),
           1, XYBitmap, 0, frown.raster, frown.width, frown.height,
           BitmapPad(displayPtr), 0);
   image->bitmap_unit = 16;
   image->bitmap_pad = 16;
   image->byte_order = LSBFirst;
   image->bitmap_bit_order = LSBFirst;
   pixmap = XCreatePixmap (displayPtr, RootWindow(displayPtr, screen),
                           frown.width, frown.height,
                           DefaultDepth(displayPtr, screen));
   XPutImage(displayPtr, pixmap, gc1, image, 0,0,0,0,frown.width,
             frown.height);
   meanMsgInfo.rasterId = pixmap;
   meanMsgInfo.rasterWidth = frown.width;
   meanMsgInfo.rasterHeight = frown.height;


                                        /*
12. Input loop
                                        */

   while (1)
   {
     if (XrInput (0, MSG_BLKHOTREAD, input) != FALSE)
     {
       if (input->type == XrXRAY && input->inputType  == XrMENU)
       {
         switch (input->value3)
         {
           case 0:      /* Show the nice message box */
             XrMessageBox (&niceMsgInfo, MSG_EDIT, input);
           break;

           case 1:     /* Show the mean message box */
             XrMessageBox (&meanMsgInfo, MSG_EDIT, input);
           break;
```

```
              case 3:        /* Exit */
                exit();
              break;
            }
          }
        }
      }
}



/*
 * This function handles the initial drawing of the window, which
 * occurs if the message is set to MSG_DRAW.  It also handles the
 * redrawing of the window when an exposure event occurs.
 * Message will contain MSG_REDRAW.
 */

xrPFI
DrawWindow (window, message, data)
Window window;
INT32 message;
XEvent * data;

{
  xrWindowEvent exposeEvent[1];
  xrWindowFunctInfo exposeFunct;
  XEvent    xnullEvent;
  xrEvent * nullEvent;


/* Set up switch to handle the two possible messages */

  switch (message)
  {
    case MSG_NEW:

     /* create the titlebar editor */

    titleBarInfo.editorWindowId = windowId;
    titleBar = XrTitleBar (NULL, MSG_NEW, &titleBarInfo);

    /*
     * Set up the structure necessary to have this function called
     * automatically upon an exposure event.
     */

    exposeEvent[0].type = Expose;
    exposeEvent[0].serial = 0;
    exposeEvent[0].send_event = 0;
    exposeEvent[0].code = 0;
    exposeEvent[0].modifier = 0;

    exposeFunct.processFlag = TRUE;
    exposeFunct.funct = (xrPFI) DrawWindow;
    exposeFunct.instance = (INT32) windowId;
    exposeFunct.message = MSG_REDRAW;
    exposeFunct.eventCount = 1;
    exposeFunct.eventList = &exposeEvent[0];

    XrInput (windowId, MSG_ADDWINDOWFUNCT, &exposeFunct);

    case MSG_REDRAW:
      /* write the text into the window */

        XDrawImageString (displayPtr, window, gc2, 20, 50,
                   "Press the Menu Button to display the menu.", 42);
        XDrawImageString (displayPtr, window, gc2, 70,
                   50 + xrBaseFontInfo->ascent + xrBaseFontInfo->descent + 5,
                   "(Default: Right Button Down)", 28);

        /* Redraw the titlebar */

        XrTitleBar (titleBar, MSG_REDRAW, XrREDRAW_ALL);

        /* Push a null event to be returned to the application */

        nullEvent = (xrEvent *) &xnullEvent;
        nullEvent->type = XrXRAY;
        nullEvent->display = displayPtr;
        nullEvent->inputType = NULL;
        XrInput (0, MSG_PUSHEVENT, nullEvent);
       break;
      }
  }
```

The HelloWorld.h Include File.

```
  Window windowId;    /*  The window identifier for the main window  */


  /*
   *  The declarations for the title bar and the info structure
   *  needed to create the title bar.
   */

  xrEditor * titleBar;
  xrTitleBarInfo titleBarInfo =
  {
     0,                           /* windowId                     */
     {0, 0, 0, 0},          /* editor rectangle             */
     XrVISIBLE | XrSENSITIVE,     /* editor state                 */
     -1, -1,                      /* editor colors - use defaults  */
     NULL,                  /* editor font - use defaults    */
     "Hello World",               /* title name                  */
     NULL,                  /* gadget 1 - unused            */
     NULL                         /* gadget 2 - unused            */
  };


  /*
   *  The declarations for the menu instance, items contained
   *  in the menu, and the menu info structure.
   */

  xrMenu * menu;
  INT8 * menuItems [9] =
  {
     "\\KEHHello Message", /*  item with a keyboad equiv of ^H  */
     "\\KEGGoodbye Message",       /*  item with a keyboad equiv of ^H  */
     "\\=",                        /*  double line seperator item       */
     "Quit"
  };
```

```
xrMenuInfo menuInfo =
{
   "Hello World",                /*  Menu title bar string       */
   menuItems,                    /*  item array declared above   */
   4,                            /*  item count                  */
   NULL,                /*  menu context - unused        */
   0                             /*  id returned with menu input */
};


/*
 *  The declarations for the two message box info structures.
 */

INT8 * niceButton[1] = { "Oky Doky" };

xrMsgBoxInfo niceMsgInfo =
{
   { 240, 245 },                 /*  message box origin  */
   NULL,                /*  relative to the RootWindow  */
   NULL,                /*  use default panel context   */
   0,                            /*  raster height*/
   0,                            /*  raster width*/
   0,                            /*  raster id*/
   "Hello World!",               /*  Text message*/
   niceButton,                   /*  button strings*/
   1                             /*  button count*/
};


INT8 * meanButton[1] = { "Adios" };

xrMsgBoxInfo meanMsgInfo =
{
   { 240, 245 },                 /*  message box origin  */
   NULL,                /*  relative to the RootWindow  */
   NULL,                /*  use default panel context   */
   0,                            /*  raster height*/
   0,                            /*  raster width*/
   0,                            /*  raster id*/
   "Goodbye World!",             /*  Text message*/
   meanButton,                   /*  button strings*/
   1                             /*  button count*/
};
```

1. **Open the display.** Using the Xlib function *XOpenDisplay*(), establish a connection to the server for the specified display. In the example the display name is passed to the program as a command line argument. The function returns a pointer to the display structure. It is important to open the display before you initialize Xrlib because Xrlib sets up the *current* display only.

2. **Initialize Xrlib.** Remember that *XrInit*() must be called before any of the functionality of Xrlib is used. In fact, you must call (only once) *XrInit*() after opening the display. This establishes the allocation functions that Xrlib routines should use for storage allocation. In most cases you will want to use the default allocation functions. To do this, pass the display pointer return by *XOpenDisplay,* a screen index (usually obtained via the *DefaultScreen*() macro) and a NULL to *XrInit*(). Remember that if Xrlib can't be initialized, *XrInit*() returns FALSE.

3. **Create the window.** This part of the program creates the *Hello World* window as a child of the *RootWindow.* Although the window is officially created, it doesn't yet appear on the display. So

we need to 'map' the window.

4. **Map the window.** To make the newly created window appear on the display we must *map* it. Subsequent mappings have no effect.
5. **Set up the window's input.** You can modify the data in the window's structure at this point, but before you actually use the window, you need to establish its input parameters and register it with *XrInput*(). Issuing the add window message to Xrlib causes the window to be registered, which then sets up other routines that need to know about the window.
6. **Select the types of input for the window.** Even though *XrInput*() knows about your window now, it won't receive any input until you specify what types of input you would like to receive. For this example, we simply want to know when the user has pressed or released a mouse button, pressed a key, or exposed this window (for which we must redraw the window). To select the types of input we want, we give to *XSelectInput*() a mask defining those input types.
7. **Create the Graphics Context.** X requires a graphics context when drawing text or graphics on windows. Here we create 2 graphics contexts and then copy the default Xrlib graphics attributes to each one. The only difference between **gc1** and **gc2** is that the foreground and backround colors are reversed.
8. **Draw the Window.** Now that our window is ready to receive input, let's draw it. To do this, we'll use the redraw routine that is typically used whenever a window is exposed.
9. **Define the Cursor for the Window.** When the cursor enters the window, we want it to be customized for our application.
10. **Set up and create the menu.** With our window now displayed, we set up a menu and display it. We will shortly be prepared to handle input from the user.
11. **Set up the rasters for the message boxes.** In this example, we have two raster images – a friendly face and a frowning face. The information for these raster images is stored in the files frown.h and smile.h. These raster files were created using a utility that enables a user to draw an image with a mouse pointer and then store that image.
12. **Input loop.** We are now ready to enter the input loop, performing our user-directed tasks.

    The input loop accepts its input from Xrlib. The function *XrInput*() funnels all input through Xrlib. The specific type of input (MSG_BLKHOTREAD) specifies blocked, hot input. Xrlib supports both blocked and non-blocked input. Blocked input will cause the input routine to wait for input on one of the windows or file descriptors of the application before returning, or to return (before input occurs) if the application sets a timeout. Non-blocked input will return immediately when an input occurs or when there is an indicator that no input has occurred.

    Regular reads and hot reads are the two modes of input available in Xrlib. When input occurs during a hot read, the set of functions attached to the window will be scanned to see if the input matches any of the events in the function's event list. If a match is found, the function will be called. Thus, hot read is the method by which an application calls the editors in a window, posts the menu, or performs any other function which has been added to the called window. Regular reads perform no scanning of the function list.

    After getting an input event, the main loop simply tests for the inputs it wants and acts accordingly.

One of the advantages of using Xrlib is that all input is routed through a single entry point. Additionally, a structure containing pointers to routines can be used to automatically perform specified operations whenever a window is exposed. This frees the programmer from having to test for an exposure by specifying the flow of control when an expose event occurs.

The DrawWindow() Routine. The function *DrawWindow*() defined in this program uses Xrlib intrinsics. This function is not part of Xlib or Xrlib–it's a function the programmer creates. (While going through this part of the example, you might want to refer to chapter 3, "Intrinsics," for more information about the functions described here.) This function does two things: it sets up a new window and provides information about how to redraw a window whenever an expose event occurs, and then draws the window using the redraw routine.

The main program explicitly calls *DrawWindow*() once –when it first draws the window. Subsequently, *DrawWindow*() is invoked automatically when an expose event occurs. Therefore, the function responds to two messages: MSG_NEW and MSG_REDRAW.

When the program calls this function, it passes the MSG_NEW message to it. When drawing the new window, the function creates the title bar editor and then sets up the structure necessary to have this function invoked automatically when an expose event occurs. The function then draws the window and title bar.

The structure for the titlebar editor is declared and initialized in the *HelloWorld.h* include file. (For more information, refer to "XRTITLEBAR()" in Chapter 5.) The include file uses Xrlib field editor functions to set up the title bar editor.

The MSG_NEW is the vehicle for creating a titlebar editor instance in a window. It expects the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled-out instance of the *xrTitleBarInfo* structure. In the function *DrawWindow*() we first supply the titlebar information structure the ID of the window. Then we call *XrTitleBar*() to create the title bar.

The next step is to initialize the structure that provides control information when an expose event occurs. An array called *exposeEvent*[] contains a set of pointers to structures that provide information about each event that we want to see. This array is attached to the window. For each input type you can specify one or more event types. In this example, we are interested only in the window expose event, so our array has only one structure. However, if we wanted to also have, for instance, an event for a keypress of the "A" key, we would initialize the array for two elements, and assign the keycode of the "A" key as the second inputCode.

Now that we have defined the events we want a response for, we define the response. We do this by adding information to the structure *exposeFunct*.

The process flag set to TRUE indicates that we want the automatic calling of the function pointed to by *funct*. The instance is the globally defined *windowid*. Although not required, the *funct* and *instance* values are cast as a precaution.

The message we want to pass to the function is MSG_REDRAW. The *eventCount* field indicates the number of events that cause the function for this window to be invoked, which in this case is one. This number should match the number of events in the *eventList*. The *eventList* is a pointer to the set of events that cause the function for this window to be invoked. Each window can have as many *xrWindowFunct* structures attached to it as you want. When an event occurs, Xrlib searches this set of structures to see if a match can be found. If a match occurs, the function for the event is executed. If no match is found, the event is returned as normal input.

After attaching the function to the event, we want to pass it to Xrlib using the *XrInput*() function. Now, every time the window is exposed, the *DrawWindow*() function executes.

Redrawing the window involves three steps:

1. **Writing the text to the window.** In this case the Xlib function *XDrawImageString*() is used. We supply this function with the window ID, the coordinates of the upper left corner of the first character, a non-null-terminated string followed by the string length, the font to use, and then the colors to use.

The foreground and background colors and font are defined in the XDefaults file, which is scanned when Xrlib is initialized (using *XrInit*()).

2. **Redrawing the title bar.** Using the field editor function *XrTitleBar*(), the specified title bar is redrawn with the only valid option in this case of redraw all.
3. **Pushing a null event onto the event queue.** The calling routine needs to know that an event happened, even if, in this case, no response is required. Pushing this event on the queue ensures that this event will be detected by the calling program. The event pushed is an Xrlib event, indicated by the define XrXRAY. With the input type set to null (and therefore not equal to XrMENU) the calling program identifies this as an Xrlib input with an input type that doesn't match what it's looking for, so it throws the input away.

The sample program is useful as a prototype as you become familiar with Xrlib. You might find it helpful to review this program, looking up the functions used in the manual, and adding your own functionality to the program.

2 Intrinsics

The intrinsics provide an application and higher layers of the X11R library with the group of functions which expand X's input model, handle resource management, set up and control a window's editors, and provide a large set of utility functions. The following sections contain descriptions of all of the intrinsic functions. Included in these sections is the syntax, structures, and messages for each of the functions.

2.1 Graphic Structures and Data Types

Before the functionality of the Xrlib intrinsic level can be described, a few defined types and structures which are used throughout the X11R library need to be defined.

Xrlib uses a set of typedefs which ensure exact element sizes on various architectures:

```
INT8 - 8 bit signed integer.
INT16 - 16 bit signed integer.
INT32 - 32 bit signed integer.

UINT8  - 8 bit unsigned integer.
UINT16 - 16 bit unsigned integer.
UINT32 - 32 bit unsigned integer.

STRING8  - array of 8 bit characters
```

These typedefs are used in Xrlib whenever the size of the storage space could cause portability problems. They should also be used by applications which want to easily port to other systems.

The X11R library uses several structures which represent geometric types:

```
typedef struct
{
   INT16 x;
   INT16 y;
} POINT;
```

The POINT structure defines a location in coordinate space. It is used for positioning graphical objects, such as editors.

```
typedef struct
{
   INT16 width;
   INT16 height;
} SIZE;
```

The SIZE structure defines the horizontal and vertical size of an object.

```
typedef struct
{
   INT16 x;
   INT16 y;
   INT16 width;
   INT16 height;
} RECTANGLE;
```

The RECTANGLE structure defines a rectangular region. It is used by many of the functions within Xrlib to define both the size and location an object is to be placed or moved.

Also defined are a set of constants which are used throughout the X11R library:

```
#define TRUE 1
#define FALSE 0
#define NULL 0
```

2.2 Initializing the X11R Library

Xrlib must be initialized before it can be used by an application. This initialization is provided to the application through one call which has the following syntax:

```
XrInit(displayPtr, screen, allocFuncts);
Display      * displayPtr;
INT32          screen;
xrAllocFuncts * allocFuncts;
```

XrInit() sets up the global information needed by both the Xrlib routines and the application. The types of information initialized include a base font and the resource manager with a set of default resources. XrInit() will return **TRUE** to the application when finished with the initialization. XrInit() can only be called once. Subsequent calls will return **FALSE**.

XrInit() requires a pointer to a display structure returned by a previous XOpenDisplay() call, a screen index number for that display and a pointer to the following xrAllocFuncts structure:

```
typedef struct
{
   char * (*newMalloc)();
   int    (*newFree)();
   char * (*newRealloc)();
   char * (*newCalloc)();
} xrAllocFuncts;
```

newMalloc	contains a pointer to a memory allocation function which Xrlib is to use for allocating memory in place of malloc().
newFree	contains a pointer to a memory free function which Xrlib is to use for freeing allocated memory and will be used in place of free().
newRealloc	contains a pointer to a memory reallocation function which Xrlib is to use for reallocating memory in place of realloc().

newCalloc	contains a pointer to a calloc function which Xrlib is to use for allocating memory in place of calloc(). arch forward

These functions are supplied to Xrlib by the application so that Xrlib can use this alternate set of allocation functions for applications which need to handle special forms of memory management. These new allocation functions must have the same return values as the system-supplied functions.

Applications which use the normal UNIX-supplied allocation functions have the *allocFuncts* parameter set to NULL.

Events used by Xrlib for editor selecting, menu posting and menu item selecting are set up by the first call to XrInit(). These four events and their default values are:

Define	Default	Meaning
XrSELECT	Button1Down	The entry condition for an editor.
XrSELECTUP	Button1Up	The exit condition for an editor.
XrMENUPOST	Button3Down	The condition which displays a menu.
XrMENUITEMSELECT	Button3Up	The condition which selects an item.

XrInit() will also examine the .Xdefaults in the users home directory file to reset any or all of the four default conditions.

XrInit() accesses these four defaults by:

```
string = XGetDefault(DisplayPtr, "X-ray", "define");
```

where the *define string* parameter is one of "Select", "SelectUp", "MenuPost", or "MenuItemSelect". The *string* parameter will be set to point to one of the following strings:

X Default	Meaning
Button1Down	The first button was pressed.
LeftButtonDown	The first button was pressed.
Button1Up	The first button was released.
LeftButtonUp	The first button was released.
Button2Down	The second button was pressed.
MiddleButtonDown	The second button was pressed.
Button2Up	The second button was released.
MiddleButtonUp	The second button was released.
Button3Down	The third button was pressed.
RightButtonDown	The third button was pressed.
Button3Up	The third button was released.
RightButtonUp	The third button was released.
Button4Down	The forth button was pressed.
Button4Up	The forth button was released.
Button5Down	The fifth button was pressed.
Button5Up	The fifth button was released.

The first three buttons are equivalent to the left, middle, and right buttons on a three button mouse.

For example, to reset the XrSELECT parameter to Button2Down, place the following in the .Xdefaults file:

```
X-ray.Select:    Button2Down
```

An additional modifier can be attached to any or all of the buttons. The valid modifiers are Control, Shift, Meta, Modifier1-Modifier5 or None. These modifiers are set through the .Xdefaults file and are as follows:

X Default	Meaning
AllButtonModifier	Sets all of the buttons to one modifier.
Button1DownModifier	Modifier for the first button down.
LeftButtonDownModifier	Modifier for the first button down.
Button1UpModifier	Modifier for the first button up.
LeftButtonUpModifier	Modifier for the first button up.
Button2DownModifier	Modifier for the second button down.
MiddleButtonDownModifier	Modifier for the second button down.
Button2UpModifier	Modifier for the second button up.
MiddleButtonUpModifier	Modifier for the second button up.
Button3DownModifier	Modifier for the third button down.
RightButtonDownModifier	Modifier for the third button down.
Button3UpModifier	Modifier for the third button up.
RightButtonUpModifier	Modifier for the third button up.
Button4DownModifier	Modifier for the forth button down.
Button4UpModifier	Modifier for the forth button up.
Button5DownModifier	Modifier for the fifth button down.
Button5UpModifier	Modifier for the fifth button up.

The first three buttons are equivalent to the left, middle, and right buttons on a three button mouse.

The default condition for all of the above (seventeen) modifiers is **None**.

When a modifier is given for a button it affects the use of that button. For example, if Button1Down is tied to XrSELECT and Button1Down is also modified by Control, an editor will be called only when the first button is pressed while the Control key is being held down.

The X11R library sets up several other default values which are used by editors, panels, and the menu manager. These defaults can also be used by the application through the corresponding Xrlib global variable.

Type	.Xdefault Entry	Xrlib Global	Meaning
Pixmap	WindowForeground	xrWindowForeground	Foreground window tile
Pixmap	WindowBackground	xrWindowBackground	Background window tile
INT32	BorderWidth	xrBorderWidth	Window border width
INT32	ForegroundColor	xrForegroundColor	Foreground color
INT32	BackgroundColor	xrBackgroundColor	Background color
XFontStruct	DefaultFont	xrBaseFontInfo	Default system font

The cursor foreground and background colors are used only in XrInit() to set up the default cursor which is used by the menu and panel managers. These values can be set in the .Xdefaults file, but can not be accessed by the application.

.XDefault Entry	Meaning
CursorForeground	Cursor foreground color
CursorBackground	Cursor background color

The cursor itself can be used by the application and is contained in the global variable xrDefaultCursor.

Refer to the X documentation and the X(1) man page for a description of the .Xdefaults file.

XrInit() sets up the resource manager and adds a number of resources. These resources are, in some cases, used internally by Xrlib functions, and they can also be used by the application. Because some of the resources are used internally, the application should not remove them from the resource manager. However, an application can replace these resources with resources of its own. This will have the effect of changing the appearance of graphical output from Xrlib routines which use the resource.

The following paragraphs detail the set of resources that Xrlib adds to the resource manager. Refer to the "Resource Manager" section below for a description of the structure and how to use each of the resources.

A set of single layer pixmaps (also called bitmaps) and a set of pixmap ids are added to the resource manager. Pixmaps are used for drawing editor instances. The resource type definitions for these two types of resources are **XrTYPE_BITMAP** and **XrTYPE_BITMAPID**. The following table lists a set of defines which can be used as the resource ids to access either of these sets of resources.

Define	Style
XrWHITE	100% foreground
XrBLACK	100% background
XrPERCENT25	25% foreground - 75% background
XrPERCENT50	50% foreground - 50% background
XrPERCENT75	75% foreground - 25% background
XrVERTICAL	Alternating vertical colors
XrHORIZONTAL	Alternating horizontal colors
XrSLANTLEFT	Alternating 45 degree colors
XrSLANTRIGHT	Alternating 45 degree colors

XrInit() initializes a set of global variables which are used by the Xrlib functions and can also be used by the application. None of these variables should be reset by the application.

BlackPixmap	A Black Pixmap constant.
WhitePixmap	A White Pixmap constant.
xrCalloc	The pointer to the calloc() function.
xrErrno	Error code set when a function fails.
xrFree	The pointer to the free() function.
xrMalloc	The pointer to the malloc() function.
xrRealloc	The pointer to the realloc() function.
xrZeroPt	A POINT structure set to {0, 0}.
xrZeroRect	A RECTANGLE structure set to {0, 0, 0, 0}.
_xrCurrentDisplay	The Display pointer for the current display.

_xrCurrentScreen	The INT32 screen index of the current screen.

2.3 The Resource Manager

The resource manager facilitates the control and access of resource objects by allowing resources to be added, retrieved, or removed from an internally- managed list. Resources can take the form of pixmaps, menus, or panels, and Xrlib defines a large set of these resource types. Applications can also define their own resource types and have them managed by the resource manager. (Refer to "Creating New Resource Types" below for a description of what is needed to create new resource types.)

The syntax for calling the resource manager is as follows:

```
XrResource (message, data)
INT32   message;
INT8  * data;
```

message is the command XrResource() is to perform, and *data* contains the information needed by *message* or will contain the return information requested by the application. (The type of structure *data* points to depends upon the message and is described in the "Resource Manager Messages" section below.)

2.3.1 What Is a Resource?

A resource is built from identification, data, and state information.

Resources are ordered and accessed in groups separated by the resource data types. These groupings of resources are referred to as *resource types*. The data for each resource type is represented by a data type and identified by an integer called the *resource type id*. These ids are 16-bit unsigned quantities which allow up to 65000 unique resource types. Xrlib reserves the type ids 64000 - 65000. The type ids of 1 - 63999 are available to the application for the creation of its own types. *Note that a resource type id of "0" is invalid.*

Within a resource type, resources are ordered and identified by a 32 bit signed value called the *resource id*. Xrlib reserves the negative values for resource id's. The application is free to use all of the positive values except zero.

The combination of the resource *type* id and the resource id create the unique identifier needed to define and access the data and state information that make up a resource.

Each resource contains a pointer to a block of memory which contains the data for the resource. The data for each resource *type* is defined by a particular data type. Xrlib defines a set of resource types which can be used by the application. (The "X11R Library Resource Types" section below describes these resource types and the data structure which represents each resource.)

Data and state information for a resource come from a block of memory allocated by an application. A resource whose data is allocated by the application will be referred to as a *memory-based resource*. For memory-based resources, the application must make sure that the resource space is deallocated when the resource is removed from the resource manager.

A resource has several state flags associated with it. Each resource maintains a state field which contains the current state flags. This field is represented by an 8-bit integer with each of its bits defining part of the resource state. The following table lists the bit within the field which contains the flag, the state flag, the define which can be used to access the bit, and a brief description of what the flag means.

Bit	State Flag	Define	Description
0	Lock	XrLOCK	Can the resource be replaced?

Lock A resource within the resource manager will be replaced by a new resource if the new resource has the same type and id. Resources can also be removed from the resource manager by a function other than the one which added the resource. If the lock flag is set, replacement or removal of a resource is inhibited.

2.3.2 X11R Library Resource Types

The following table lists the resources that Xrlib predefines and the structure or variable type that is used to define each resource.

Resource Define	Object Structure
XrTYPE_STRINGS	STRING8
XrTYPE_PIXMAP	xrPixmap
XrTYPE_PIXMAPID	xrPixmapId
XrTYPE_BITMAP	xrPixmap
XrTYPE_BITMAPID	xrPixmapId
XrTYPE_CURSOR	xrCursor
XrTYPE_CURSORID	Cursor
XrTYPE_FONTINFO	XFontStruct
XrTYPE_FONTID	Font
XrTYPE_FUNCTION	xrFunction

The follow list describes each of these resource types:

- XrTYPE_STRING8

  This resource type contains a null-terminated character string.

- XrTYPE_PIXMAP

  This resource type contains the structure and data information needed to define a pixmap. The structure used for this type is defined as follows:

```
typedef struct
{
   INT16 width;
   INT16 height;
   INT8  depth;
   UINT16 * raster;
} xrPixmap;
```

width	defines the width of the pixmap in pixels.
height	defines the height of the pixmap in pixels.
depth	defines the number of bits needed to define each pixel and can only be set to one of the following defines, **XrBIT1**, **XrBYTE1**, **XrBYTE2**, **XrBYTE3**, and **XrBYTE4**.
raster	contains a pointer to the block of memory which contains the pixel image for the pixmap.

- XrTYPE_PIXMAPID

  This resource type contains the size of a pixmap resource and the identifier for the pixmap which has been installed on an X server. Refer to the X documentation for a discussion on how to install a pixmap. The structure used to represent this resource type is defined as follows:

```
typedef struct
{
   INT16 width;
   INT16 height;
   Pixmap pixmapId;
} xrPixmapId;
```

width	contains the width (in pixels) of the installed pixmap.
height	contains the height (in pixels) of the installed pixmap.
pixmapId	contains the id of the installed pixmap.

- XrTYPE_BITMAP

  This resource type contains the structure and data information needed to define a bitmap. A bitmap is a pixmap with a depth of one. The structure for this resource type is the same as for XrTYPE_PIXMAP but the *depth* field must be set to the define **XrBIT1.**

- XrTYPE_BITMAPID

  This resource type contains the size of a bitmap resource and the identifier for the bitmap which has been installed on an X server. (Refer to chapter 2 for information about installing a bitmap.) The structure used to represent this resource type is defined as follows:

```
typedef struct
{
   INT16 width;
   INT16 height;
   Pixmap bitmapId;
} xrBitmapId;
```

width	contains the width (in pixels) of the installed bitmap.
height	contains the height (in pixels) of the installed bitmap.
bitmapId	contains the id of the installed bitmap.

- XrTYPE_IMAGEID This resource type contains the structure and data information needed store X image information within the resource tree. The structure used for this type is defined as follows:

```
typedef struct
{
   INT16    width;
   INT16    height;
   XImage * imageId;
} xrImageId;
```

width	contains the width (in pixels) of the installed image.
height	contains the height (in pixels) of the installed image.

ImageId	contains a pointer to the image structure of the installed image.

- XrTYPE_CURSOR

  This resource type contains the structure and data information needed to define a cursor. The structure used for this type is defined as follows:

```
typedef struct
{
   xrPixmap cursor;
   xrPixmap mask;
   INT16 xoff;
   INT16 yoff;
   INT32 foreground;
   INT32 background;
   INT32 func;
} xrCursor;
```

cursor	contains the pixmap which defines the cursor image. The cursor image is defined by an xrPixmap structure which must contain a Pixmap image.
Mask	contains the pixmap which contains the cursor mask. The mask defines which pixels of the cursor pixmap won't be displayed. It is defined by an xrPixmap structure that must also contain a pixmap image. If the structure has the *raster* field set to NULL, the mask is undefined and all of the pixels of the cursor will be displayed.
xoff	contains x offset which defines what x coordinate, relative to the location of the cursor, will be returned by the pointer when an event occurs.
yoff	contains the y offset which defines what y coordinate, relative to the location of the cursor, will be returned by the pointer when an event occurs.
foreground	contains the foreground color in which the cursor is to be displayed. Any bits in the cursor bitmap which are set to 1 will be displayed in foreground color.
background	contains the background color in which the cursor is to be displayed. Any bits in the cursor bitmap which are set to 0 will be displayed in background color.
func	contains the combination rule to be used for moving the cursor about the display. Refer to the list of defines which can be used for the combination rule.

- XrTYPE_CURSORID

  This resource type contains the identifier for a cursor which has been installed on an X server. (Refer to chapter 2 for a discussion on how to install a cursor.)

- XrTYPE_FONTINFO

  This resource type contains the XFontStruct structure for a font which has been installed on an X server. Refer to chapter 2 for a discussion of the XFontStruct structure and how to load a font.

- XrTYPE_FONTID

  This resource type contains the identifier for a font which has been installed on an X server. Refer to chapter 2 for a discussion on how to install a font.

- XrTYPE_FUNCTION

  This resource type contains a pointer to a function. The structure which contains the function is defined as follows:

```
            typedef INT32 (*xrFunction)();
```

2.3.3 Resource Manager Messages

This section lists and describes each of the messages that XrResource understands.

Several of these messages use the following structure for passing or returning data:

```
typedef struct
{
   UINT16   resourceType;
   INT32    resourceId;
   INT8     resourceState;
   INT32    resourceFile;
   INT8   * resourceObject;
} xrResourceInfo;
```

resourceType	contains the resource type id.
resourceId	contains the resource id.
resourceState	contains the state flags for the resource.
resourceFile	is set to the define **XrMEMORY** for memory-based resources.
resourceObject	contains a pointer to the memory containing the resource.

The following messages provide for the main-line accessing of the resources contained in the resource lists.

- MSG_ADD

  This message adds to the resource list the resource identified by the *data* parameter, which points to an xrResourceInfo structure. If the *resourceObject* field contains a valid pointer, the *resourceFile* field will be set to **XrMEMORY** to indicate the resource is from application memory.

  If the resource type and id fields in *data* match a resource already within the resource list, the resource in the list will be replaced with the new resource. If the location of the old resource is from application memory the *resourceObject* field within *data* will be set to point at the object being replaced. This allows an application to maintain control of an object being replaced in the resource list.

- MSG_REMOVE

  This message removes from the resource list the object identified by the *data* parameter, which points to a filled-out instance of an xrResourceInfo structure. Only the *resourceType* and *resourceId* fields need to be set by the application. When the resource is located in the resource manager's lists, the *resourceFile* field is checked and if it is set to **XrMEMORY**, the *resourceObject* field in *data* will be set to point to the object. If the object is not found in the resource list this message will fail.

- MSG_FIND

  This message locates the requested resource within the resource list. *data* is a pointer to an xrResourceInfo structure. The *resourceType* and *resourceId* fields of the structure identify the resource to be found. When the resource is located, the rest of the fields of the structure will be filled out.

- MSG_SETSTATE

  This message is used to set the state of a resource contained in the resource manager. For this message, *data* is a pointer to an xrResourceInfo structure which has the *resourceType*, *resourceId*, and *resourceState* fields filled out. This message will search for the resource, and when it is found, set its *resourceState* to the new state.

  Note that for memory-based resources, only the lock flag of the resource state is used.

The following two messages provide for the creation or destruction of application-defined resource types. Refer to the "Creating New REsource Types" section below for detailed instructions about how to create a resource.

- MSG_NEWTYPE

  This message allows an application to create a new resource type. *data* is a pointer to the following structure.

```
typedef struct
{
   UINT16 resourceType;
   INT32  (*resourceHandler)();
} xrResourceTypeInfo;
```

resourceType	contains the integer identifier number for the resource type. The number must be between 1 - 63999. High identifier number values for the resource type require more space for the resource type list, therefore, sequential definition of resource types is highly recommended.
resourceHandler	is reserved and should be set to NULL.

- MSG_FREETYPE

  This message destroys and removes a resource type from the resource type list. *data* contains the resource type identifier. This message will fail if the identifier is not in the range of 1 - 63999, if the resource type is not defined, or if there are any resources still attached to the resource type.

2.3.4 Creating New Resource Types

Besides using the X11R library's predefined resource types, the resource manager enables the application writer to create his own resource types. The following paragraphs describe what is required for an application to create a resource type.

Resource types are managed as two ordered arrays of resource type structures. One of the arrays is used for the X11R library's predefined resource types. The other contains the set of types the application has defined. Even though 63999 types can be defined, the application should define the resource types *sequentially* from a type id of 1, or system memory will be wasted.

The structure used for all resource types is as follows.

```
typedef struct
{
   xrResource * resourcePtr;
   INT32 (*resourceHandler)();
} xrResourceType;
```

resourcePtr	contains a pointer to the list of resources of this type.
resourceHandler	is reserved and should be set to NULL.

One of the members of the resource type structure is the *resourcePtr* field. This field points to the head of the set of resources attached to the resource type. The resources are added and accessed through a binary tree which is ordered by the resource id. Every resource in the resource tree is defined by the following xrResource structure:

```
typedef struct _xrResource
{
   INT32   resourceId;
   UINT8   resourceState;
   INT8  * resourceObject;
   INT32   resourceFile;
   struct _xrResource left;
   struct _xrResource right;
} xrResource;
```

resourceId	is the resource identifier which uniquely distinguishes the resource object within its type.
resourceState	contains the state of the resource.
resourceObject	points to the resource object itself.
resourceFile	is set to the define XrMEMORY for memory-based resources.
left	points at the left child resource in the resource tree.
right	points at the right child resource in the resource tree.

2.4 Input

The XrInput() routine handles all requests for input by an application from one or more windows. All input is routed through a single entry point within Xrlib, providing a one-routine interface to all of the types of input an application will need. The input routine also provides the mechanism for routines that handle the higher level Xrlib functions such as editors and menus, to route the results of their actions through the same input stream to the application. The syntax for calling XrInput() is as follows:

```
XrInput (windowId, message, data);
Window   windowId;
INT32    message;
INT8   * data;
```

windowId contains the identification number of the window and is used to set some of the input settings. *message* is the command XrInput() uses to perform the function, and *data* is the information needed to complete the function or will contain any return information generated by XrInput().

The following sections define the structures used by XrInput(), the types of input that XrInput() can generate, and the messages that XrInput() provides.

2.4.1 The XEvent Structure

Xrlib uses the X *XEvent* structure to return input to the application. This structure is defined as follows:

```
    typedef union _XEvent {
      int                       type;
      XAnyEvent                 xany;
      XKeyEvent                 xkey;
      XButtonEvent              xbutton;
      XMotionEvent              xmotion;
      XCrossingEvent            xmotion;
      XFocusChangeEvent         xfocus;
      XExposeEvent              xepose;
      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;
} XEvent;
```

type contains the type of event which has occurred. The event types that can be generated by X are defined in <X11/Xlib.h> and are as follows:

Request	Circumstances
KeyPress	keyboard key pressed
KeyRelease	keyboard key released
ButtonPress	pointer button pressed
ButtonRelease	pointer button released
MotionNotify	pointer moves within window
EnterNotify	pointer entering window
LeaveNotify	pointer leaving window
FocusIn	keyboard focus on window
FocusOut	keyboard focus leaving window
KeymapNotify	keyboard state changed
MouseMoved	pointer moves within window
Expose	full window changed and/or exposed
GraphicsExpose	graphics draw attempted on obscured area
NoExpose	graphics draw successful
VisibilityNotify	window (un)obscured
CreateNotify	a window was created
DestroyNotify	a window was destroyed
UnmapNotify	window was unmapped
MapNotify	a window was mapped
MapRequest	a map window request occurred
ReparentNotify	window has new parent
ConfigureNotify	window configuration changed
ConfigureRequest	a configure window request occurred
GravityNotify	parent resized, moved child
ResizeRequest	a resize window request occurred
CirculateNotify	window changed position in stack
CirculateRequest	a circulate window request occurred
PropertyNotify	window property has changed
SelectionClear	window lost selection ownership
SelectionRequest	selection ownership request occurred
SelectionNotify	response to SelectionRequest
ColormapNotify	the colormap was changed
ClientMessage	a client message was received
MappingNotify	pointer/keyboard mapping changed

XEvent is actually a union of all the different XEvent types. These various types are not the same size and do not contain the same data. The only value that is guaranteed to be reliable for every XEvent structure is the *type* field. To extract the results of the events, the XEvent *type* must be determined and the XEvent structure needs to be cast to the proper type. For example, to process Xrlib, KeyPress and Visibility Events the following code fragment could be used:

```
XEvent event;
UINT32 keycode;
INT32  state;

    /* XrInput() is explained in the pages that follow */
XrInput(NULL, MSG_BLKHOTREAD, &event);

switch (event.type) {
```

```
                case XrXRAY:
                    xrayEvent= (xrEvent *) &event; /* Cast event to Xrlib Event */
                    .
                    .    /* Xrlib Event Processing */
                    .
                    break;
                case KeyPress:
                    keycode=event.xkey.keycode;/* Get Keycode */
                    .
                    .    /* Keyboard Event Processing */
                    .
                    }
                    break;
                case VisibilityNotify:
                    state=event.xvisibility.state;/* Get State   */
                    .
                    .    /* Visibility Event Processing */
                    .
                    }
                    break;
                default:
                    .
                    .    /* Unknown Event Processing */
                    .
                    break;
        }/* end switch */
```

Note that each event structure is cast to the appropriate type immediately after the event type is determined. This is the proper way to process events with Xrlib and X11. (Refer to the "$DOC" document. for a description of the various event structures.)

2.4.2 X11R Library Input Sources

The input model supplied by the XrInput() routine can handle the gathering of input from three types of sources:

- An X window registered with XrInput()
- Other X windows
- Application-supplied file descriptors

An X window is registered with XrInput() through a message to XrInput(). This message will be discussed in the section containing XrInput()'s messages. Registering a window with XrInput() gives the window the capability to generate Xrlib's special types of input. For input that occurs in a registered window, the XEvent structure is coerced into an Xrlib input structure so that the data can be accessed to extract the Xrlib generated input. The Xrlib input structure will be defined in the next section.

Applications can also use Xrlib's input model to gather input from a window which the application has created but has not registered with XrInput(). Input of this form will be returned to the application through XrInput() as normal XEvent input.

Applications can gather input from a file descriptor by providing the file descriptor to XrInput(). The file descriptor will tell Xrlib where to look for input. Input of the types *read, write,* and *exception* can be gathered. When input is pending on a file descriptor, XrInput() will return an Xrlib input structure containing the file descriptor and an indicator of whether the type of input pending is *read, write,* or *exception.* It is then up to the application to service the input request on the file descriptor.

2.4.3 X11R Library Input

Xrlib adds an input-event type to the list of input types defined by X. This can be checked by the application by comparing the field to the define XrXRAY. *X* and *Xrlib* input types can be generated from a registered window. All of the forms of X input can be gathered. The data is extracted from the XEvent structure by looking at the type of input and coercing the structure properly. Data is accessed for Xrlib input in the same manner. When Xrlib input occurs the *type* field of the XEvent structure will be set to XrXRAY. When this occurs, the data contained in the XEvent structure can be extracted by coercing it into Xrlib's input structure. This structure is defined as follows.

```
typedef struct
{
   UINT32          type;
   unsigned long   serial;
   Bool            send_event;
   Display       * display;
   INT32           source;
   INT16           inputCode;
   INT8            inputType;
   INT8            value1;
   INT16           value2;
   INT16           value3;
   POINT           valuePt;
   INT32           valuePtr;
} xrEvent;
```

type	contains a value indicating whether input is being returned from a registered window, or an ordinary X window. The input from a registered window can be tested by comparing the inputCode with the define XrXRAY. If the type does not equal XrXRAY, the input is an X event type and the structure to be used to extract data for the event is defined in the "Xlib - C Language Interface, Protocol Version 11" document.
serial	contains the serial number of the last request processed by the X server. Xrlib ignores this field.
send_event	contains a Boolean true or false value. This value is set to true if the event came from a SendEvent request. Xrlib currently ignores this field.
display	contains a pointer to the display on which this event occurred. When using Xrlib this value should always point to the current display.
source	contains the source of the input. Its value is dependent on the *inputType* field of the xrEvent structure and will be set to either the window id of the window in which the input occurred or to a file descriptor of an input source which the application has defined. This feature will be discussed in a later section.
inputType	contains the type of input that was generated. The set of input types that Xrlib can generate are defined as follows.

Type	Meaning
XrEDITOR	Input from an editor
XrMENU	Input from a menu
XrPANEL	Input from a panel
XrMESSAGEBOX	Input from a message box
XrFILE	Input from a file descriptor
XrTIMEOUT	Input timed out

The rest of the fields are used in different ways by the various types of input. Refer to the input-generating functions to see how these fields are defined.

2.4.4 Input Messages

The XrInput() functions are broken into four sets of messages, each of which handle a different aspect of the XrInput() routine.

The following messages initialize and set up windows for processing Xrlib input:

- MSG_ADDWINDOW

  If an application wants to gather Xrlib type input or place editors within a field it has created, Xrlib needs to be told about the window. Issuing this message causes a window to be registered with XrInput() which then sets up the other routines that need to know about the window. To register a window, *windowId* should be set to the id of the window to be registered and *data* should point to an xrWindowData structure which is defined as follows:

```
typedef struct
{
   RECTANGLE windowRect;
   Pixmap    foreTile;
   Pixmap    backTile;
} xrWindowData;
```

windowRect	contains the size and location of the window on the display.
foreTile	contains the foreground tile of the window. It should be set to the same tile as the border tile used to create the window.
backTile	contains the background tile of the window and should be set to the same tile as the background tile used to create the window.

- MSG_REMOVEWINDOW

  When an application destroys a window, it should call XrInput() so that the window will be removed from XrInput()'s tables. For this message, *windowId* should be set to the id of the window to be removed. *data* is unused and can be set to NULL.

- MSG_SETWINDOWDATA

  If an application changes the size or tiles of a window, XrInput() needs to be given the data to modify its tables. For this message, *windowId* should be set to the id of the window in which the data is to be set and *data* should be set to point to an xrWindowData structure which contains the needed information.

- MSG_ADDWINDOWFUNCT

The X11R library's editors and managers can be called automatically upon a particular input or inputs. An example of this is the menu manager displaying a menu upon an XrMENUPOST event input. To accomplish this function, XrInput() maintains a set of information for each registered window. This message is the means by which a particular window gets the capabilities described above. For this message, *data* is a pointer to the following structure:

```
typedef struct
{
   INT8  processFlag;
   xrPFI (*funct)();
   INT32 instance;
   INT32 message;
   INT32 eventCount;
   xrWindowEvent * eventList;
} xrWindowFunctInfo;
```

funct	contains a pointer to the function that is to be invoked when a particular event occurs. The function must be of the form:

```
(*funct)(instance, message, data)
INT32   instance;
INT32   message;
INT8  * data;
```

instance is the instance of the type of element *funct* is to operate upon and is defined as a pointer to a function returning an integer. *message* is the message that is to be sent to the function. *data* points to the event which occurred to invoke this function.

instance	contains the object the function is to operate upon.
message	contains the message to the function.
processFlag	is a boolean which is set to TRUE or FALSE by the application or manager adding the xrWindowFunctInfo structure. It is used to turn the processing of the event list on and off, and for automatically calling the function attached to the window.
eventCount	A window can be set up so that more than one event will invoke a function. This field contains a count of that number of events.
eventList	contains a pointer to the set of events which will cause the function for this window to be invoked. The structure is defined as follows.

```
typedef struct
{
   INT32  inputType;
   UINT32 inputModifier;
   UINT32 inputCode;
} xrWindowEvent;
```

inputType	contains a single X event type which can cause the function to be called.

inputModifier	contains a mask of any button modifiers such as *ControlMask* or *ShiftMask* that were pressed. This field is only used for the X *ButtonPress* and *ButtonRelease* events.
inputCode	contains an additional value used to further distinguish the conditions necessary to invoke the function. This field is only used for the X *KeyPress, KeyRelease, ButtonPress,* and *ButtonRelease* events. For the key events, the field should be set to the character code of the key that is to bring up the window. For button events, the field should be set to one of the X defines of **Button1-Button5.**

Each window can have as many xrWindowFunct structures attached to it as is needed. When an input occurs on the window during a 'HOTREAD' (see below), the event lists contained in the xrWindowFunct structures attached to the window are searched to see if a match can be found. If the event matches one of the events listed for the window, the function for the event is invoked. If no match is found, the event is returned as normal input.

- MSG_REMOVEWINDOWFUNCT

  This message removes an xrWindowFunct structure that was previously added to the window *windowId. data* is a pointer to the function that matches the function supplied in the xrWindowFunct structure contained within the window.

- MSG_SETPROCESSFLAG

  A manager of a window may sometimes want to turn on and off the event list checking and function calling routines provided through MSG_ADDWINDOWFUNCT. This message sets the *processFlag* field of the xrWindowFunct structure to **TRUE** which turns on the processing function for the window identified by the parameter *windowId. data* is a pointer to the function whose *processFlag* is to be set.

- MSG_CLEARPROCESSFLAG

  This message clears the *processFlag* field of the xrWindowFunct structure by setting it to **FALSE** which turns off the processing function for the window identified by the parameter *windowId. data* is a pointer to the function whose *processFlag* is to be cleared.

The following messages provide for the various forms of input-gathering from Xrlib:

Xrlib supports both blocked and non-blocked input. Blocked input will cause the input routine to wait for input on one of the windows or file descriptors of the application before returning, or to return (before input occurs) if the application sets a timeout. Non-blocked input will return immediately when an input occurs or when there is an indicator that no input has occurred.

Regular reads and hot reads are the two modes of input available in Xrlib. When input occurs during a hot read, the set of functions attached to the window will be scanned to see if the input matches any of the events in the function's event list. If a match is found, the function will be called. Thus, hot read is the method by which an application calls the editors in a window, posts the menu, or performs any other function which has been added to the called window. Regular reads perform no scanning of the function list.

The following messages cause inputs to be returned to the application:

- MSG_BLKREAD

  This message copies an input event into the XEvent structure pointed to by *data* if there is an input pending. If no input is pending, the call waits until input occurs and then returns that input.

- MSG_NONBLKREAD

This message copies an input event into the XEvent structure pointed to by *data* if there is an input pending. If no input is pending, *data* is left unchanged and NULL is returned.

- MSG_BLKHOTREAD

  This message copies an input event into the XEvent structure pointed to by *data* if there is an input pending. If no input is pending, the call waits until an input occurs and returns that input. If the input is a select input, the editors will be invoked; if the input is a menu input, the menuing system will be invoked.

- MSG_NONBLKHOTREAD

  This message copies an input event into the XEvent structure pointed to by *data* if there is an input pending. If no input is pending, *data* is left unchanged and NULL is returned. If the input is a select input, the editors will be invoked; if the input is a menu input, the menuing system will be invoked.

The following messages manipulate the X11R library input queue:

- MSG_PUSHEVENT

  This message adds an event structure pointed to by *data* to the front of the input queue. The next "read" returns this event.

- MSG_PEEKEVENT

  This messages fills out the event structure pointed to by *data* with a copy of the event at the front of the input queue. The input queue is unaffected by this call. If the input queue is empty, this call will fail.

- MSG_CLEAR

  This message clears the input queue of all pending input. *data* is unused for this call and should be set to NULL.

The following messages are used to set up and change the input sources from which XrInput() is to gather input.

- MSG_ADDINPUT

  This message adds a file descriptor from which XrInput() is to gather input. *data* points to an xrFDInput structure which is defined as follows.

```
typedef struct
{
   INT32 fd;
   INT8  type;
} xrFDInput;
```

fd	contains the file descriptor for the source from which XrInput() is to gather input.
type	tells XrInput() which forms of input should be gathered from the file descriptor. The field can be set to any combination of the defines **XrREAD, XrWRITE,** or **XrEXCEPTION.** These defines should be "OR-ed" together to define the value of this field.

When input is pending on the file descriptor, an event will be returned which contains the data about the input. The event structure will have the *type* field set to **XrXRAY,** the *source* field set to the file descriptor, the *inputType* field set to **XrFILE** and the *inputCode* field set to either **XrREAD**, **XrWRITE**, or **XrEXCEPTION**.

- MSG_REMOVEINPUT

  This message removes a previously added file descriptor from the input set. *data* is a pointer to an xrFDInput structure defined by MSG_ADDINPUT and contains the file descriptor and the conditions under which the file descriptor is to be removed.

- MSG_SETTIMEOUT

  This message sets a timeout value in seconds for XrInput() to wait for input to occur from the file descriptors it is selecting upon. *data* is a pointer to the following structure:

```
struct timeval
{
   unsigned long tv_sec;
   long          tv_usec;
};
```

tv_sec	defines the number of seconds before a timeout occurs.
tv_usec	defines the number of microseconds before a timeout occurs.

This structure is defined in <time.h>.

The timeout value affects how blocked reads work. Normally, the blocked read will wait indefinitely for an input. This is accomplished by setting the "timeval" structure to the define **XrMAX_ALARM**. If an application sets a timeout value, the blocked read will wait only the number of seconds plus the number of microseconds given by the value. Values of timeouts have no effect on non-blocked reads. When a blocked read times out, an event of *type*: **XrXRAY** with an *inputType*: **XrTIMEOUT** will be returned.

- MSG_GETTIMEOUT

  This message returns the current timeout value. *data* is a pointer to a timeval structure which will be filled out to the current values.

2.5 The Editor List Manager

When an application attempts to create an instance of a field editor, it will always specify the window in which the instance is to be attached and displayed. The field editor code is then responsible for creating the editor instance, and attaching it to the specified window.

Each window has a linked list associated with it, made up of the editor structures for each field editor attached to that window. This linked list plays a very important part when determining how input within a window is handled. When an input event occurs within a window, only those editors which reside in this list will be able to accept input from the window.

For each process using the Xrlib facilities, the intrinsics maintain an array of window Id's and pointers to the editor list associated with each of the windows. However, an application's windows are not automatically added to this list. An application must first 'register' a window with the Xrlib intrinsics, before it will be added to this list, and thus be capable of having field editors attached to it. This registering is accomplished through the XrInput() routine using MSG_ADDWINDOW.

When a field editor attempts to attach a new editor instance to a window, the editor structure for that instance will be added to the end of the linked list. The input-dispatching routine searches the entire list; so that if two editor instances overlap, the instance which was created first has priority. This may be confusing to a user, so *application writers are encouraged not to overlap field editor instances.*

When an application requests that a particular editor instance be destroyed, the editor called removes the associated editor instance from the editor list. Failure to do this will cause unpredictable results, and would most likely result in a memory fault error occurring at some time.

The following intrinsic is provided, to allow field editors and applications to perform some editor-specific functions:

```
XrEditor (windowId, message, data)
Window   windowId;
INT32    message;
INT8   * data;
```

windowId contains the identifier of the window to which the editor list is attached, the *message* specifies the type of action which is to be taken, and *data* contains the information needed to complete the function.

The structure used in the editor list is discussed in the next section. (The messages that XrEditor() supports are discussed in the "Editor List Messages" section below.)

2.5.1 The Editor List Structure

An array of window Ids which have been registered with XrInput() is contained in XrEditor(). The groups of editors belonging to these windows are also attached to this array. The editor structure itself is simple and straightforward, and contains some of the global information for the instance. The structure also contains a pointer which the field editor can use to point to instance-specific data. The format for the editor structure is as follows:

```
typedef struct _xrEditor {
   struct _xrEditor * (*editorFunct)();
   Window             editorWindowId;
   RECTANGLE          editorRect;
   INT8               editorState;
   INT8             * editorData;
   struct _xrEditor * nextEditor;
} xrEditor;
```

editorFunct	is a pointer to the field editor routine associated with the instance. For example, if this were a scrollbar editor instance, this would be set to the address of XrScrollBar().
editorWindowId	is the window Id associated with the window to which this instance is attached. The field editor can use this information whenever it needs to redraw the instance.
editorRect	is a copy of the rectangle definition which is specified by the application when it creates an editor.
editorState	is the current state of the flags associated with this editor instance. These flag states are originally obtained by copying the 'editorState' field from the 'Info' structure.

editorData	serves as a pointer to the editor-specific data. The data pointed to by this element is different for each editor type, and has no fixed format. If an editor requires no additional information, this pointer should be set to NULL.
nextEditor	is the one field which *must not* be touched by a field editor. This field links this instance into the window's field editor list.

When an editor routine is called to create a new editor instance, the editor allocates the space required for an xrEditor and initializes each field. The editor then calls to XrEditor to add the editor instance to the window's editor list.

2.5.2 Editor List Messages

The messages that XrEditor() understands are divided into several functional sets. These functional sets include messages which the application can use to call XrEditor(), messages which should only be used by a field editor; and messages which are used internally and hence, should *not* be accessible to any application or field editor routine. Each of these messages will be discussed below, along with an explanation of how the parameters *windowId* and *data* are interpreted by the message.

The following set of messages should only be called by field editors.

- MSG_ADD

  Whenever a new field editor instance is created, it must be attached to a window. This message attaches the instance to the end of the linked list associated with the window. The *windowId* parameter is a 32-bit integer Id indicating the window to which the field editor instance is to be attached. The *data* parameter points to an initialized instance of the xrEditor structure. This structure will be linked into the editor list.

  The following example shows how the scrollbar editor attaches an instance to a particular window:

```
{
   xrEditor * scrollBar;

   /* Allocate an xrEditor structure */
   if ((scrollBar = (*xrMalloc)(sizeof (xrEditor))) == NULL)
   {
      xrErrno = XrOUTOFMEM;
      return ((xrEditor *) NULL);

   /* Fill in the required fields */
   scrollBar->editorFunct = XrScrollBar;
   scrollBar->editorWindowId = editorWindowId;
   scrollBar->editorState = editorState;
   scrollBar->editorData = sbDataPtr;
   XrCopyRect (&editorRect, &scrollBar->editorRect);

   /* Attach the editor to the window */
   if (XrEditor (editorWindowId, MSG_ADD, scrollBar) == NULL)
   {
      /* We failed; window must not have been registered */
   }
}
```

- MSG_REMOVE

  When an editor instance is destroyed, it must also be removed from the linked list maintained by the window to which it was attached. When removing an editor instance from the linked list, the field editor must supply an indication of which editor instance is to be removed, and which window it should be removed from. The *windowId* parameter contains the window which contains the instance. The *data* parameter points to the editor instance structure associated with the instance to be removed.

The following example shows how the scrollbar editor removes one of its instances:

```
{
   xrEditor * scrollBar;

   (*xrFree)(scrollBar->editorData);
   XrEditor (scrollBar->editorWindowId, MSG_REMOVE, scrollBar);
   (*xrFree)(scrollBar);
}
```

The following messages can be used by application programs and by other functions within Xrlib.

- MSG_PROCESSKEY

  This messages takes an XEvent structure as input and calls the editor in which the event occurred with MSG_EDIT. The event used to activate an editor must match the default conditions set up in XrInit(). The event structure contains the location at which the event occurred and XrEditor() uses this location to see which editor rectangle the point falls in. Normally, applications will not use MSG_PROCESSKEY since it is called directly from XrInput() each time an XrSELECT event occurs and the application is using hot reads. However, if an application doesn't want to use hot reads, it can call XrEditor() directly. The parameters to be sent to the function are the id of the window contained in *windowId*, the message, and a pointer to the XEvent structure which is passed through the *data* parameter.

  An example, showing how an application might use this feature, is shown below:

```
XrInput (NULL, MSG_BLKREAD, &inputEvent);

if (XrMapButton (XrSELECT, &inputEvent))
   XrEditor (myWindow, MSG_PROCESSKEY, &inputEvent);
```

- MSG_REDRAW

  Sometimes, an application may wish to redraw those field editors which lie within a particular portion of its window. To indicate the portion of the window which is to be checked, the application must create an instance of the 'RECTANGLE' structure, and fill it in to describe the draw region. The *data* parameter must then be set to point to this structure and the *windowId* parameter must be set to indicate the window to be redrawn.

  This message is most useful when a window which was partially or totally hidden is uncovered. When the window is uncovered, the application need only take the information describing the exposed region and pass it along to this message; the editors which lie within this region will then be redrawn automatically.

- MSG_REDRAWALL

  This message is similar to MSG_REDRAW except that it redraws the entire set of editors attached to a window. The *data* parameter is unused for this call and should be set to NULL. The *windowId* parameter should be set to indicate the window to be redrawn.

- MSG_SETSTATE
  Sometimes, an application may find it necessary to modify the state flags associated with all the editor instances in a particular window. If the new state flag values are the same for all

instances, the application can use this message instead of calling each of the field editors individually.

This might be useful when the application's window becomes inactive. The application might want all field editors in its window to become insensitive and when the window is reactivated, the instances will be redrawn as sensitive.

When an application issues this request, *windowId* must specify the window to be affected and *data* must contain the new state flag values.

An example describing how to use this feature is shown below:

```
switch (stateFlag)
{
   case ACTIVE:
      XrEditor(windowId, MSG_SETSTATE, XrVISIBLE | XrSENSITIVE);
   break;

   case INACTIVE:
      XrEditor(windowId, MSG_SETSTATE, XrVISIBLE);
   break;
}
```

2.6 Editor Groups

Some applications and the higher-level functions within Xrlib (such as a the panel manager) need to handle editors in groups. Specifically, the panel manager sets up groups of editors to define sub-panels. The XrEditorGroup() function provides this capability by separating the set of editors attached to a window into application-specified group and allowing each group to be manipulated as a unit. The types of functions which can be applied to editor groups include getting and setting the groups' editors states and clearing the rectangular area which defines the group. The syntax for the XrEditorGroup() function is as follows.

```
xrEditorGroup *
XrEditorGroup (groupInstance, message, data)
xrEditorGroup * groupInstance;
INT32           message;
INT8          * data;
```

The *groupInstance* parameter contains the pointer to the editor group. *message* is the function that is to be applied to the editor group. *data* is a pointer to a structure containing the information needed by *message* or will contain return information requested by the calling function.

The following sections discuss the editor group structure and the messages that XrEditorGroup() defines.

2.6.1 The Editor Group Structure

The editor list-handling routines maintain an array containing the set of windows which have been registered with XrInput(). Included in this array is a pointer to the first editor group. When a window is registered, an editor group is automatically created for the window, so for applications which are only doing simple editor management, the XrEditorGroup() function can be ignored. The set of editor groups attached to a window is built and accessed as a linked list. The editor group structure is defined as follows:

```
typedef struct _xrEditorGroup
{
   Window      groupWindowId;
   RECTANGLE   groupRect;
   INT8        groupState;
   xrEditor  * editorList;
   xrGraphic * graphicList;
   struct _xrEditorGroup * nextGroup;
} xrEditorGroup;
```

groupWindowId	contains the id of the window in which the group of editors reside.
groupRect	contains the rectangle which encloses the entire set of editors attached to this group. The rectangle is empty when a group is created and is then increased in size whenever an editor is added to the group. (Refer to XrUnionRect in the "Rectangle Routines" section below for further details.) The application also has access to this rectangle and can set it to any size. XrEditorGroup() will ensure that all of the editors reside within the bounds of the rectangle.
groupState	contains the state of the set of editors attached to this group. The states settings are the same as those used for individual editors. This field is accessible to the application through messages to this function and is used to set the state of an entire set of editors for the group just as an individual editor state would be set.
editorList	points to the set of editors attached to the group.
graphicList	points to the graphics list attached to the group. **This function is not implemented.**
nextGroup	points to the next group attached to the window.

2.6.2 Editor Group Messages

- MSG_NEW

  creates a new editor group, initializing the group's elements, adding the group into the group list attached to the window, and making it the active group. The concept of "active group" means that any editors created for the window will be attached through this group. For this call, *groupInstance* is unused and should be set to NULL. *data* should contain the window id of the window which is to contain the group. The group instance of the new group will be returned by this message.

- MSG_FREE

  destroys the editor group and frees the editors attached to the group. For this call, *data* is unused and should be set to NULL.

- MSG_GETDEFAULTGROUP

  returns the pointer to the default editor group that is added to a window when the window is registered with XrInput(). For this message, *data* should contain the id of the registered window.

- MSG_GETSTATE

  returns the state of the editor group. *data* is a pointer to an INT8 variable and will be set to the state of the group.

- MSG_SETSTATE

sets the state of the editor group. *data* contains the new group state which can be any combination of the editor state defines **XrVISIBLE** and **XrSENSITIVE**. The effect of setting the group state will be to activate or deactivate the editor processing for the **XrSENSITIVE** flag, and to clear or draw the editors for the **XrVISIBLE** flag. The states of the individual editors in the group are not changed when the group state is set.

- MSG_ADDTOGROUP

  tells the editor list handling function to which group new editors are to be added. If the window has only the default group defined, editors will be added to this group automatically. When a new group is created through MSG_NEW, it becomes the "active group" (that is, the group in which editors will be added) so that if the application wants to add editors to an existing group, it needs to issue the ADDTOGROUP message. For this message, *groupInstance* should contain the instance pointer of the group which is to be made active. *data* is unused and can be set to NULL. As stated, each window gets a default group set up for it when it is registered. This creates a problem in that the function which created the window will not know the group id number for the window and therefore cannot access editor group functions for the default group. This problem is handled through MSG_GETDEFAULTGROUP which returns the group instance of the default group.

- MSG_GETGROUPRECT

  returns the group rectangle through *data* which contains a pointer to a rectangle structure. *groupInstance* should be set to the instance pointer of the group in which the group rectangle is to be extracted.

- MSG_SETGROUPRECT

  sets the group rectangle to the rectangle pointed to by *data*. *groupInstance* should be set to the instance pointer of the group in which the group rectangle is to be set. The rectangle supplied must be large enough to contain the set of editors attached to the window or the message will fail.

- MSG_ADJUSTGROUPRECT

  recalculates the group rectangle for the group point by *groupInstance* to the minimal rectangle needed to enclose the editors within the group. It is mainly used by editors when a MSG_RESIZE or MSG_MOVE is issued to them. For these cases, the editors need to make sure the editor rectangle does not extend beyond the bounds of the group rectangle or the editor cannot be activated.

- MSG_CLEARRECT

  is used to clear the area defined by the group rectangle to the windows background tile. *groupInstance* should be set to the instance pointer of the group which is to be cleared. *data* is unused and can be set to NULL.

2.7 Window/Editor Relationship

We have discussed how an editor attaches an instance to a window and how groups of editors are handled by the editor-grouping routine. This section deals with this topic in more detail. This discussion covers the following topics:

- Organization of the field-editor list.
- Organization of the elements within this list.
- Steps for updating the list.

• What the list is used for.

Every Xrlib window has a pointer to a linked list of editor groups, and each editor group has a linked list of editors. When a field editor instance is created, it is added to the linked list attached to the currently active editor group. The particular field editor adds the instance to the list when it receives a MSG_NEW message. Similarly, when an editor instance is destroyed, the field editor must make sure that the instance is removed from the window's editor list. Catastrophic problems will occur if this is not done! Removing an editor instance from the window's list involves issuing an XrEditor() request, with the MSG_REMOVE message.

Figure 2-1 shows how a windows-editor list might appear:

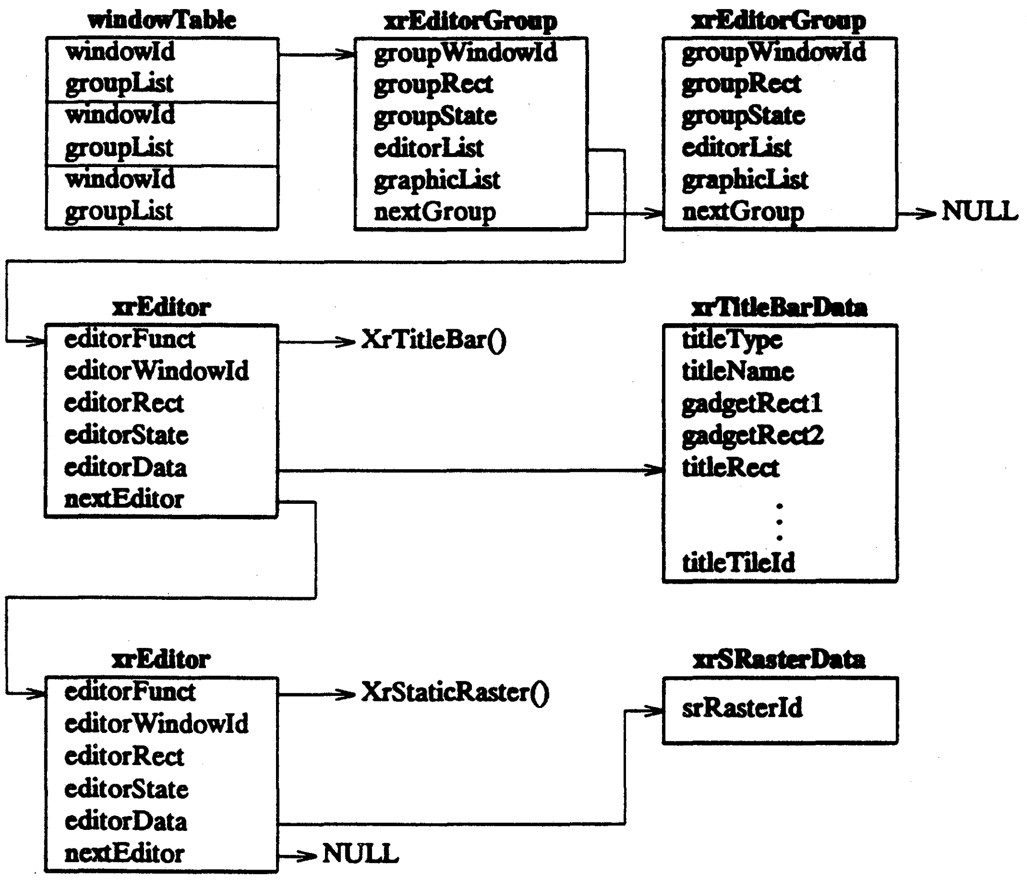


Figure 2-1. Sample Windows-Editor List

The xrEditor structure address is returned to the application after an editor instance has been created. This address is used to identify the instance for all future editor calls. Each time an application invokes the editor- handling routine, for example, XrScrollBar() for the scrollbar editor, it must pass in the pointer to the xrEditor structure associated with the instance to be modified. The field editor uses this pointer to access all of the information describing a particular instance.

Updating the editor list is the responsibility of the field editors. Whenever a field editor instance is created, the xrEditor structure for that instance must be attached to the window, using the XrEditor() routine. When an editor instance is destroyed, that instance must be removed from the window's editor list.

The primary purpose for maintaining the editor list is to facilitate distribution of input events. When input is received by XrInput(), it traverses the editor list to see if the event occurred in one of the window's editor instances. If it did, the event is passed to the field editor for processing. If an editor instance has been created, but not attached to a window's editor list, it will be unable to receive any input from the Xrlib input dispatcher. There are situations where this is actually desirable, but this topic will be covered in chapter 5 "Building a Field Editor."

2.8 Geometric Functions

The following sections contain a set of geometric functions used for manipulating point and rectangle structures.

2.8.1 Point Routines

The following list contains the set of functions which can be used for manipulating point structures.

```
XrSetPt (pt, x, y)
POINT * pt;
INT16   x;
INT16   y;
```

XrSetPt assigns the two coordinates, *x* and *y*, to the point structure *pt*.

```
XrOffsetPt (pt, x, y)
POINT * pt;
INT16   x;
INT16   y;
```

XrOffsetPt changes the values of the point structure *pt* by adding the *x* and *y* offsets to the members of *pt*.

```
XrCopyPt(srcPt, dstPt)
POINT * srcPt;
POINT * dstPt;
```

XrCopyPt copies the point structure *srcPt* into *dstPt*.

```
XrAddPt(srcPt, dstPt)
POINT * srcPt;
POINT * dstPt;
```

XrAddPt adds the point structures *srcPt* and *dstPt* together placing the results in *dstPt*.

```
XrSubPt (srcPt, dstPt)
POINT * srcPt;
POINT * dstPt;
```

XrSubPt subtracts the point structure *srcPt* from *dstPt* and places the results in *dstPt*.

```
XrEqualPt (ptA, ptB)
POINT * ptA;
POINT * ptB;
```

XrEqualPt() compares the point structures *ptA* and *ptB* and returns **TRUE** if they are equal. It returns **FALSE** otherwise.

2.8.2 Rectangle Routines

The following list contains the set of functions which can be used for manipulating rectangle structures.

```
XrSetRect (rect, x, y, width, height)
RECTANGLE * rect;
INT16       x;
INT16       y;
INT16       width;
INT16       height;
```

XrSetRect() assigns the values of the rectangle *rect* with the appropriate values in *x, y, width,* and *height.*

```
XrSetPtRect (rect, topLeft, botRight)
RECTANGLE * rect;
POINT     * topLeft;
POINT     * botRight;
```

XrSetPtRect() calculates and assigns the values of the rectangle *rect* using the two points *topLeft* and *botRight.*

```
XrCopyRect(srcRect, dstRect)
RECTANGLE * srcRect;
RECTANGLE * dstRect;
```

XrCopyRect() copies the rectangle *srcRect* into the rectangle *dstRect.*

```
XrOffsetRect(rect, dx, dy)
RECTANGLE * rect;
INT16       dx;
INT16       dy;
```

XrOffsetRect() moves the rectangle *rect* by adding the value *dx* to the x coordinate of the rectangle and adding the value *dy* to the y coordinate of the rectangle.

```
XrInsetRect(rect, dx, dy)
RECTANGLE * rect;
INT16       dx;
INT16       dy;
```

XrInsetRect() shrinks or expands the rectangle *rect*. When *dx* and *dy* are positive, the left and right sides are moved in by the value *dx* and the top and bottom sides are moved together by the value *dy*. If the values are negative, the rectangle expands.

```
XrSectRect(srcRectA, srcRectB, dstRect)
RECTANGLE * srcRectA;
RECTANGLE * srcRectB;
RECTANGLE * dstRect;
```

The rectangles *srcRectA* and *srcRectB* are intersected to form a new rectangle which is placed in *dstRect*.

```
XrUnionRect (srcRectA, srcRectB, dstRect)
RECTANGLE * srcRectA;
RECTANGLE * srcRectB;
RECTANGLE * dstRect;
```

The union of the rectangles *srcRectA* and *srcRectB* is calculated and placed in *dstRect*.

```
XrPtInRect (pt, rect)
POINT     * pt;
RECTANGLE * rect;
```

XrPtInRect() determines if the point *pt* is in the rectangle *rect*. TRUE is returned if this is the case, FALSE otherwise.

```
XrPt2Rect(ptA, ptB, dstRect)
POINT * ptA;
POINT * ptB;
RECTANGLE * dstRect;
```

XrPt2Rect() generates the smallest rectangle which will enclose the two points *ptA* and *ptB*. The calculated rectangle is placed in *dstRect*.

```
XrEqualRect(rectA, rectB)
RECTANGLE * rectA;
RECTANGLE * rectB;
```

XrEqualRect() compares the two rectangles, *rectA* and *rectB* and returns **TRUE** if they are equal or **FALSE** if they are not equal.

```
XrEmptyRect (rect)
RECTANGLE * rect;
```

XrEmptyRect() returns **TRUE** if the rectangle *rect* is empty or **FALSE** if it is not. A rectangle is empty if the bottom coordinate is less than the top, or the left coordinate is greater than the right.

2.9 Utility Routines

This set of functions provide an application programmer with some frequently used utilities.

2.9.1 StringWidth

The string width function is used to calculate the length, in pixels of a character string. It has the following syntax.

```
XrStringWidth (fontInfo, str, charWidth)
XFontStruct * fontInfo;
STRING      * str;
INT32         charWidth;
```

XrStringWidth() calculates the width in pixels of a character string using the font information contained in the structure pointed at by *fontInfo*. The pixel width will be calculated and returned as the value of the function. The *charWidth* parameter contains the length of the string in characters or is set to the define **XrNULLTERMINATED** if the string is null-terminated.

2.9.2 _XrSetUpGC

Xrlib creates a default global graphics context, called **_xrDefaultGC**, when first initialized. **_XrSetUpGC** allows an application to copy this default graphics context.

```
_XrSetUpGC(gcontext)
GC gcontext;
```

This function takes a target graphics context as a parameter and copies all the contents of **_xrDefaultGC** to it.

2.9.3 XrMapButton

Xrlib automatically sets up the **XrSELECT**, **XrSELECTUP**, **XrMENUPOST**, and **XrMENUITEMSELECT** conditions for the application when the application calls XrInit(). When a button event occurs, several functions within Xrlib and the application may need to determine whether the button event matches one of the above conditions. The following function provides this capability.

```
XrMapButton (eventCode, event)
INT8      eventCode;
XEvent * event;
```

XrMapButton() takes as a parameter an eventCode which contains one of the above defines and an XEvent. The event parameter will be compared against the conditions necessary for the above defines and if a match is found, **TRUE** will be returned. If no match is found, **FALSE** will be returned.

2.9.4 XrGetWindowEvent

Functions within Xrlib and the application may need to fill out an xrWindowFunct structure to add a processing function to a window. Filling out this structure for handling XEvents is straightforward, but filling out this structure to handle the **XrSELECT**, **XrSELECTUP**, **XrMENUPOST**, and **XrMENUITEMSELECT** conditions is more difficult. The following function provides an easy way to initialize this structure.

```
XrGetWindowEvent (eventCode, windowEvent)
INT8 eventCode;
xrWindowEvent * windowEvent;
```

This function takes one of the above defines as the "eventCode" and a pointer to an xrWindowEvent structure as parameters. It fills out the xrWindowEvent structure with the information which will match the conditions for the define.

2.9.5 XrVersion

An application can find out which version of Xrlib it is using by calling the XrVersion() function which has the following syntax.

```
INT8 * XrVersion ()
```

XrVersion() returns a pointer to a string which contains the version of Xrlib being used. The string will have for the format of **X-ray Version** *##* - *##.##* The *##* contains the version of the X11 library Xrlib was built with. The *##.##* can be ignored. The string returned is static and thus cannot be modified.

This page left blank intentionally.

3 Field Editors

"Field editors" is a collection of tools that aids applications writers by providing them with a common user interface across a wide variety of programs.

3.1 Introduction

Editors are most often used as building blocks in higher level user interface constructs, (panels and dialog managers) but they are not limited to such use. Applications writers should feel free to incorporate these tools into their programs in whatever manner they wish.

Each editor provides and manages a specific piece of the user interface. When an editor instance is created, it takes over a rectangular region of a window and fields all input events occurring within that region.

Many of the X11R library field editors allow applications to specify the font to be used when an instance is created. If a font is not specified, the editor will use the default system base font. (Refer to the manual page for *XrInit*() in appendix A for information about the system base font.)

3.1.1 Editor Calling Sequence

All editors provide the same calling sequence. The calling sequence format is as follows:

```
xrEditor *
XrEditorName (instance, message, data)

    xrEditor * instance;
    INT32      message;
    INT8     * data;
```

The *instance* parameter specifies which editor instance is being referenced. This value is returned when the instance is created, and must then be specified in all future references to that instance.

The *message* parameter specifies what action an editor should take regarding a specified editor instance. (Refer to the manual page(s) for the individual editors for an overview of how each editor handles messages.)

The *data* parameter is an arbitrary value (a pointer or a scalar) which is coerced to a specific data type by an editor. Refer to the manual page(s) in appendix A for the individual editors to see the type of data that each message expects.

3.1.2 Creating An Editor Instance

Before an application can use an editor, it must create an instance of that editor. This is normally a three-step process:

1. The application makes a call to the editor and obtains the coordinates of the 0-based rectangle into which the instance will fit. This normally involves supplying information describing the instance to be created.
2. The application will then offset this rectangle to position the instance in the desired location within a window.

3. The application now asks the editor to create an instance occupying the region of the window specified by the rectangle.

Upon completion of this three-step process, the editor will return a unique pointer to the instance structure associated with the newly-created editor instance; this is referred to as the *instance pointer.* This pointer should be saved by the application program because it will be needed whenever the application communicates with the editor regarding this particular instance.

3.1.3 Editor Messages

Most communications with the Xrlib editors are made using the base set of editor messages listed below. Information for any additional messages (which some editors may require in order to get their work done) is contained in the manual pages for those individual editors.

- MSG_NEW
- MSG_SIZE
- MSG_SETSTATE
- MSG_GETSTATE
- MSG_MOVE
- MSG_RESIZE
- MSG_REDRAW
- MSG_FREE
- MSG_EDIT

The calling sequence, and a brief overview is presented for each of the common editor messages:

MSG_NEW

This message is the means by which an application may request that an editor instance be created. Each editor has a structure which it will expect the application to have filled out with the information describing the instance to be created.

If the create request succeeds, the editor will return a pointer to the instance structure; this must be used for all future calls made in reference to this particular instance. If the create request fails, then a NULL value is returned.

The following block of code gives an example of the steps needed to create an instance of the scrollbar editor:

```
{
   xrEditor      * SBinstance;
   xrScrollBarInfo SBinfo;

   /* Fill out the structure defining the instance */

   /* Create the instance */
   SBinstance = XrScrollBar (NULL, MSG_NEW, &SBinfo);
}
```

MSG_SIZE

This message is used primarily by an application to obtain the minimal 0-based rectangle, which would completely contain the described editor instance. Once an application has obtained this information, it can offset the rectangle, to relocate it to the desired portion of the window.

This message will expect the application to pass in a pointer to the same type of structure used by the MSG_NEW message. However, for this message, the whole structure will not need to be filled out prior to calling the editor. Each editor will require a different amount of data to be specified, before this call is made. Refer to the individual manual pages for more details.

The bounds rectangle which is returned by this message, is important, because it describes the region of the window which is to be occupied by the particular editor instance. Any input events which occur within this region, will be passed along to the editor, to be processed relative to the particular editor instance.

In return for making a MSG_SIZE request, the editor will fill in the *editorRect* portion of the passed-in structure, with the coordinates of the 0-based rectangle.

The following block of code outlines how to issue this message:

```
{
   xrScrollBarInfo SBinfo;

   /* Fill out the structure defining the instance */

   /* Ask for the bounds rectangle */
   XrScrollBar (NULL, MSG_SIZE, &SBinfo);
}
```

MSG_SETSTATE

Each editor instance has a series of state flags associated with it which determine both its behavior and its appearance. These are initially specified when the instance is created, but may be modified at any time.

Although an editor may support many different state flags, each editor is required to support at least the following two flags:

- XrVISIBLE
- XrSENSITIVE

XrVISIBLE defines the appearance of a particular editor instance. When this state is set, the instance will be drawn in the specified window. When this state is cleared, the area occupied by the instance will be filled with the background tile for the window, thus making the instance invisible.

XrSENSITIVE defines the behavior of a particular editor instance; it only comes into play when an instance is visible. This state defines how an editor instance will behave when any pointer events occur within its rectangular region. When this state is set, the editor will process any input that occurs within its region. When cleared, all input events will be ignored for that instance. Whenever possible, an editor instance will use some form of visual feedback to indicate whether or not it is sensitive. Refer to the individual field editor manual pages for a description of this visual feedback.

The following block of code outlines the calling sequence for this message:

```
{
   INT8       stateFlags;
   xrEditor * SBinstance;

   /* Create the editor instance */

   /* Make the instance invisible */
   stateFlags = XrSENSITIVE;
```

```
      XrScrollBar (SBinstance, MSG_SETSTATE, stateFlags);
   }
```

MSG_GETSTATE

As was mentioned in the previous section, each editor instance has a series of state flags associated with it. Sometimes, it is useful for an application to inquire as to the current settings of these state flags. This message provides this capability. It will return the current settings of the state flags, to which an application may modify, and then reset for a particular editor instance.

The following block of code shows how this message may be used:

```
   {
      INT8        stateFlags;
      xrEditor * SBinstance;

      /* Create the editor instance */

      /* Get the current state flags */
      XrScrollBar (SBinstance, MSG_GETSTATE, &stateFlags);
   }
```

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the *editorRect* will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window; i.e., relative to the window's origin. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location. This occurs only if the instance is visible.

```
   {
      POINT   newOrigin;
      xrEditor * SBinstance;

      /* Create the editor instance */

      /* Move the scrollbar origin to (50,50) */

      XrSetPt(&newOrigin,50,50);
      XrScrollBar(SBinstance,MSG_MOVE,&newOrigin);
   }
```

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window and will then redraw the instance using the new size and location information. This occurs only if the instance is visible.

```
{
   RECTANGLE   newSize;
   xrEditor * SBinstance;

   /* Create the editor instance */

   /*
    * Resize the scrollbar instance by increasing its
    * length. Also move the origin of the instance
    * to (50,50).
   /*

   newSize.x=50;
   newSize.y=50;
   newSize.height=SBinstance->editorRect.height;
   newSize.width=SBinstance->editorRect.width + 40;
   XrScrollBar(SBinstance, MSG_RESIZE, & newSize);
}
```

MSG_REDRAW

A situation may arise where an application finds the need to redraw an editor instance. This can occur for several reasons:

- An instance was obscured by something, but is now no longer obscured.
- The application has changed some aspect of the instance, and the editor needs to redraw it, to bring the instance in line with the changes.

This message is provided precisely with this in mind.

Several types of redraw modes are available. However, not all editors support all of the modes. Fortunately, all editors support the *XrREDRAW_ALL* mode. When this mode is specified, the complete editor instance will be redrawn.

The following block of code outlines how this message may be used:

```
{
   xrEditor * SBinstance;
   /* Create the editor instance */

   /* Redraw the complete instance */
   XrScrollBar (SBinstance, MSG_REDRAW, XrREDRAW_ALL);
}
```

MSG_FREE

When an application no longer needs a particular editor instance, it should destroy it. When an instance is destroyed, it is automatically removed from the window. If the instance is visible, it will no longer process input events, and no further editor messages should be issued using that instance pointer.

The following block of code outlines how an instance is destroyed:

```
{
   xrEditor * SBinstance;

   /* Destroy the instance */
   XrScrollBar (SBinstance, MSG_FREE, NULL);
}
```

MSG_EDIT

This message provides the means by which input events may be processed in regard to a particular editor instance.

Normally, an application will not use **MSG_EDIT**. Xrlib will route input events to the appropriate editor via **XrInput()** if the applications uses the **MSG_NONBLKHOTREAD** or **MSG_BLKHOTREAD** messages. XrInput() will automatically catch all pointer events and redirect them to the field editor when a 'HOTREAD' message is used.

However, if the application uses either **MSG_BLKREAD** or **MSG_NONBLKREAD** when collecting its input from the **XrInput()** routine, then the application will be responsible for determining which editor instance should receive a pointer event. The application must determine what to do with the event and if that requires that a field editor be called then it must send a **MSG_EDIT** message to the appropriate editor. Refer to the manual page for **XrEditor()**, in appendix A for a discussion on how this is done.

For all editors, the MSG_EDIT message will expect a pointer to an *XEvent* structure to be passed in.

After the editor has processed the event, it will normally return some information via the input queue to the application indicating what was done.

Most field editor instances are activated when a select event occurs within the rectangular region occupied by the instance. Most, but not all, editors will wait for the user to release the select button before returning to the application. Thus the application should be prepared to receive and handle 'select up' events even though they can probably be ignored.

A sample call for this message would be:

```
{
   xrEditor  * SBinstance;
   XEvent      eventInfo;

   XrScrollBar (SBinstance, MSG_EDIT, &eventInfo);
}
```

3.1.4 Return Value

Upon successful completion of any of the above messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value is the pointer to the newly-created editor instance structure.

If any message request fails, a NULL value is returned.

3.2 Description

The following field editors are provided:

- A titlebar editor
- A scrollbar editor
- A radiobutton editor (*)
- A checkbox editor (*)
- A push button editor (*)
- A text edit editor
- A static text editor
- A raster select editor
- A raster editor
- A static raster editor (*)
- A page editor
- A scrollbar list editor
- A groupbox editor

Before discussing individual field editors, we will differentiate between the two types of field editors that are included in the X11R library toolbox. Editors generally fall into one of the following groupings:

- SINGLE-ENTITY INSTANCES
  All of the editors listed above, except those whose names are followed by an (*) symbol fall into this category. When an application creates an instance of one of these editors, the instance will contain only one entity. For example, when a scrollbar editor instance is created, it contains only one scrollbar, not two or three. The majority of Xrlib field editors follow this model.
- MULTIPLE-ENTITY INSTANCES
  The field editors whose names are followed by the (*) symbol fall into this group. These editors are special, because the instances they create may contain one or more editor entities. For example, when an application creates a checkbox editor instance, it may contain one checkbox or even ten checkboxes. Each checkbox is referred to as an entity, and since these entities were created by a single request, they will be treated as a group. One advantage to this method is that the editor automatically takes care of calculating the size and position of each of the entities; the application need only supply the rectangle dimensions to contain the instance, and the field and column counts. A second advantage is that the application need only make a single request to create a group of related buttons or checkboxes; applications rarely create just a single radio button or checkbox.

Multiple entity field editors differ from single entity field editors in two other ways:

- INDIVIDUAL STATE FLAGS
  For a multiple-entity instance, each of the individual entities may have a set of state flags associated with it, in addition to the state flags associated with the instance as a whole. This feature provides an application program with increased flexibility for controlling the behavior and appearance of the instance. Each editor instance has a set of state flags associated with it (XrSENSITIVE and XrVISIBLE), which are queried and modified using the MSG_GETSTATE and MSG_SETSTATE

messages. A multiple entity editor may also keep a second set of state flags for each entity, which are modified using the MSG_GETITEMSTATES and MSG_SETITEMSTATES messages. The state flags associated with the whole instance have precedence over the entity's state flags; this implies that if the XrVISIBLE flag is cleared for the whole instance, then none of the entities will be drawn, regardless of their individual state flag values. Conversely, if the XrVISIBLE flag is set for the whole instance, then the editor will use each entity's state flag setting to determine whether or not an entity should be drawn.

- ENHANCED SET OF FIELD EDITOR MESSAGES
  In addition to the standard set of field editor messages, multiple-entity editors may also choose to support several new messages. Two of these: (MSG_GETITEMSTATES and MSG_SETITEMSTATES) were covered in the preceding paragraph. The additional messages are provided to allow an application to obtain more information describing how the individual entities within an instance are organized. When an application creates an instance, it specifies the editor rectangle into which the instance is to reside, but since the field editor automatically calculates the size and position of the individual entities within the instance, the application has no idea where each entity will be displayed within the editor rectangle. This presents no problem for most applications, but for a panel controller which needs to know where each entity in the panel is located, (so that field traversal may be performed) this is vital information. This problem is solved by the addition of two field editor messages: MSG_GETITEMCOUNT and MSG_GETITEMRECTS. These two messages, when used together, provide an application with the tools needed to determine the size and position of each entity within an editor instance.

The following sections present brief discussions of each of the field editors provided in the X11R library. In addition to describing how each editor works, an example showing how the editor can be used and a picture of an instance of the editor is provided.

All requests issued by an application regarding a particular editor instance are implemented by of a set of predefined editor messages. These messages (or commands) provide the means by which an application creates, modifies, and destroys an editor instance. (For further information about messages, refer to the section entitled "Editor Messages" above.)

3.3 A Titlebar Editor

The titlebar editor allows an application to create and display a rectangular titlebar within a window. The titlebar size and location are arbitrary, and are specified by the application. However, a titlebar is normally placed along the top of the window, and runs the full width of the window.

When an application creates a titlebar, it is allowed some freedom in customizing the instance, to meet its particular needs. In addition to the size and location of the titlebar, (which were covered earlier) an application may also specify which components should be used when the titlebar is constructed, along with a description of how the instance should handle pointer events.

A titlebar instance can be composed of from 1 to 3 components. The component pieces of a titlebar are:

title string	The most common use for a titlebar is to display a string, which provides some indication to the user as to what a window is being used for. When specified, a title string will be displayed in the center of the titlebar, using the font specified by the application program. On either side of the title string will be displayed the title string border; this serves as a visual indicator for the state of the window.

gadget boxes — Sometimes an application will wish to provide a means for allowing a user to easily and quickly request some frequently-used action, so two gadget boxes are provided for this purpose. As part of a title bar, two boxes (one in the left corner, and one in the right corner) are available for application-specific use. The application may specify a single character to be displayed within the box, which serves as an icon to the user, to describe what action will occur when the box is selected. For example, if a program wanted to set one of the boxes up to produce help information when it is selected, then it might choose to define the gadget character as the '?' character. When a user selects one of the gadget boxes with the pointer, notification will be passed on to the application. The application can then perform whatever action is appropriate.

Other characteristics of a titlebar that may be controlled by a program include how the instance is displayed (if it is visible) and how the instance responds to pointer selects (if it is sensitive). These pieces of state information are specified when the instance is first created, and can be modified anytime thereafter.

An example showing how to create a titlebar instance is presented below:

```
{
   xrTitleBarInfo  tbInfo;
   xrEditor      * tbInstance;

   /*
    * Create a titlebar instance, which is located at the top of the window,
    * extends from the left to the right edge of the window, and is composed
    * of:
    *
    *      1) A title string
    *      2) Gadget box 1 (icon = 'X')
    *      3) Gadget box 2 (icon = '?')
    *
    */
   tbInfo.editorWindowId = applic_window;
   XrSetRect (&tbInfo.editorRect, 0, 0, 0, 0);
   tbInfo.editorState = (XrVISIBLE | XrSENSITIVE);
   tbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   tbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   tbInfo.editorFont = NULL;
   tbInfo.titleName = "DATACOMM";
   tbInfo.gadgetIcon1 = "X";
   tbInfo.gadgetIcon2 = "?";

   /* Call the titlebar editor and create the instance */
   tbInstance = XrTitleBar (NULL, MSG_NEW, &tbInfo);
}
```

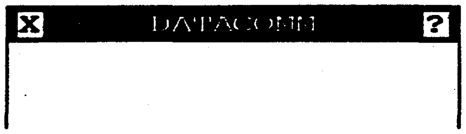


Figure 3-1. Sample Titlebar Instance

3.4 A Scrollbar Editor

The scrollbar editor allows an application to create a vertical or horizontal scrollbar instance within a specified window. The size and location of the scrollbar instance is controlled by the application, but the normal convention followed when displaying a scrollbar is to display a vertical scrollbar along the right edge of the window, and a horizontal scrollbar along the bottom edge.

When an application creates a scrollbar instance, the instance editor allows as much control as possible over how the instance will appear and behave by allowing the application to specify the instance configuration. This includes:

- The height and width of the scrollbar.
- The orientation (vertical or horizontal) of the scrollbar.
- The location of the scrollbar.
- The size of the slide box.
- The components displayed within the scrollbar.

When a scrollbar is created, it is drawn as a rectangular box, with a scroll arrow at either end. The region which occupies the area between the two scroll arrows is known as the scroll region. The scroll box resides within the scroll region and serves as a visual feedback; it indicates the current value associated with the scrollbar.

A user is provided with several means for manipulating a scrollbar, each of which is explained below:

- By selecting one of the scroll arrows, a user may cause an application to scroll the information within its window. The direction of the scroll is dependent upon which scroll arrow was selected, while the amount the information is scrolled is controlled by the application. When a scroll arrow is selected, the scrollbar editor does not modify the scrollbar instance; instead, it sends an input event notifying the application that the scroll arrow was selected. It is then up to the application to modify the information within its window, and reposition the slide box for the scrollbar.
- By Selecting the slide box, a user may interactively move the slide box. When the select key is released by the user, the interactive slide operation will complete. At that time, the editor notifies the application that the slide box has changed position and will pass along the new position. The application should then update its display to match the new value.
- When the user selects a point within the slide region, but not within the slide box, the editor passes an input event to the application notifying it of the position where the select occurred. The application can choose to ignore this, or it can update its display and then reposition the slide box for the scrollbar.

An example showing how to create a scrollbar instance is presented below:

```
{
   xrScrollBarInfo  sbInfo;
   xrEditor       * sbInstance;

   /*
    * Create a vertical scrollbar instance, located along
    * the right edge of the window, and extending from the top to the
    * bottom of the window.
    */
   sbInfo.editorWindowId = applic_window;
   sbInfo.editorState = (XrVISIBLE | XrSENSITIVE);
```

```
    sbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
    sbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
    sbInfo.orientation = XrVERTICAL;
    XrScrollBar (NULL, MSG_GETPARAMETERS, &sbInfo.configuration);

    /*
     * Obtain the coordinates of the rectangle needed to contain the
     * vertical scrollbar of width 11.
     */
    sbInfo.editorRect.width = 11;
    XrScrollBar (NULL, MSG_SIZE, &sbInfo);

    /*
     * Stretch and offset the editorRect so that the instance will
     * be located at the left edge of the window.
     * Assume that the application has saved the window dimensions
     * in a variable named window_size.
     */
    sbInfo.editorRect.height = window_size.height-2;
    XrOffsetRect (&sbInfo.editorRect, window_size.width -1 -11,1);

    /* We are now ready to create the scrollbar instance. */
    sbInstance = XrScrollBar (NULL, MSG_NEW, &sbInfo);
}
```

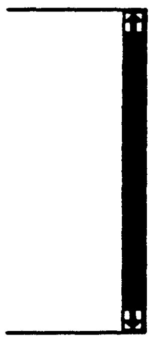


Figure 3-2. Sample Scrollbar Instance

3.5 A Radiobutton Editor

The radiobutton editor allows an application to create and display a collection of related buttons in a series of rows and columns. The size of the buttons is dependent upon how the buttons are defined, and upon the font used to display them.

When an application creates a radiobutton instance, it is allowed as much flexibility as possible for customizing how the instance will be displayed. Among the options which may be specified by the application are:

- The number of rows and columns into which the buttons are to be displayed.
- The font to be used when displaying the button labels; this also determines the size of the buttons.
- The button labels themselves; these are optional, and an application is free to supply labels for some, or for all of the buttons.

- The foreground and background pens to be used each time the instance is displayed.
- The action the editor should take each time a pointer select occurs within the button instance.

A radiobutton is drawn as a circle, with an optional label to its right. When a button has been selected, the circle will be drawn and filled with the instance's foreground color; all non-selected buttons are drawn as circles filled with the instance's background color. When the instance is drawn, it will be laid out so that all items in a row and column line up; this sometimes involves placing some extra padding between buttons, if their labels are of different lengths.

Radiobuttons operate in a fashion very similar to the channel-select buttons on your radio. Only a single button may be selected at any time. Each time a new button is selected, the previously active button is made inactive.

A user can use the pointer to select a button. When this occurs, the editor redraws the instance to reflect the new active button, and will then send some event information to the application, informing it of the new active button. An application is then free to perform any action which might be implied by the button.

Radiobuttons are most often used when an application wishes to supply a set of options to the user, only one of which may be selected. For example, if an application allows the user to change the pen color to one of four choices, radio buttons would be an ideal means for allowing the user to do this.

The example below demonstrates the steps required to create a radiobutton instance. For this demonstration, we will use the example of the pen color, mentioned above:

```
{
   xrRadioButtonInfo  rbInfo;
   xrEditor         * rbInstance;
   INT16              active_rb_button = 1;
   INT8             * rbLabels[] = {"BLACK", "WHITE", "BLUE", "GREEN"};

   /*
    * Create a radio button instance composed of four buttons and
    * laid out in two columns.  Each of the buttons will have a label
    * associated with it.
    */

   /*
    * Ask the editor to supply the editorRect which
    * is to contain the radio button instance.
    */
   rbInfo.editorFont = NULL;
   rbInfo.numFields = 4;
   rbInfo.numCols = 2;
   rbInfo.labels = rbLabels;
   XrRadioButton (NULL, MSG_SIZE, &rbInfo);

   /*
    * Offset the editorRect so that the instance will be located
    * in the middle of the window.  Assume that the application has
    * saved a copy of the window dimensions in a variable named window_size.
    */
   xOffset = window_size.width/2 - rbInfo.editorRect.width/2;
   yOffset = window_size.height/2 - rbInfo.editorRect.height/2;
   XrOffsetRect (&rbInfo.editorRect, xOffset, yOffset);

   /* Now, ask the editor to create the desired instance. */
   rbInfo.editorWindowId = applic_window;
   rbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
```

```
    rbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
    rbInfo.editorState = (XrVISIBLE | XrSENSITIVE);
    rbInfo.stateFlags = NULL;
    rbInfo.value = &active_rb_button;
    rbInstance = XrRadioButton (NULL, MSG_NEW, &rbInfo);
}
```

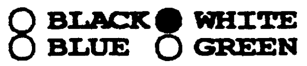


Figure 3-3. Sample Radiobutton Instance

3.6 A CheckBox Editor

The checkbox editor allows an application to create and display a collection of related checkbox items in a series of rows and columns. The size of the boxes are dependent upon how the boxes are defined, and upon the font used to display them.

When an application creates a checkbox instance, the editor allows it as much flexibility as possible for customizing how the instance will be displayed. The options which may be specified by the application are:

- The number of rows and columns into which the boxes are to be displayed.
- The font to be used when displaying the box labels.
- The box labels themselves; these are optional, and an application is free to supply labels for some, or for all of the boxes.
- The foreground and background pens used when drawing the instance.
- The action to be taken each time a pointer select occurs within a checkbox.

An individual checkbox is drawn as a square, with an optional label to its right. When a particular checkbox has been selected, the square will be drawn and filled with the foreground color; the state of the other checkboxes in the instance will not be altered. When the instance is drawn, it will be laid out so that all items in a row and column line up. This sometimes involves placing padding between checkboxes, if the labels are of different lengths.

Checkboxes differ from radio buttons in at least two ways:

- Checkboxes are square; radio buttons are round.
- Checkboxes place no restrictions upon how many items in an instance may be active at any given time; radio buttons allow only one item to be active at a time.

When a checkbox item is drawn, if it has been selected, it will be drawn as a filled square; if it is not currently selected, then it will be drawn as an unfilled square. Whenever a checkbox is selected, its value toggles. This means that when an active box is selected, it will become inactive, and vice-versa.

A user can select a checkbox using the pointer. When this occurs, the editor will redraw the instance to reflect the new checkbox value, and will then send an Xrlib input event to the application, informing it that the instance has changed. The Xrlib input event contains the index of the checkbox which changed. An application is then free to perform any action which might be implied by the checkbox.

Under normal circumstances, modifying the state of a checkbox does not cause any immediate action. What normally happens is that a user will modify several checkboxes, and when he is ready for some action to occur regarding the checkbox values, he will notify the application. This is sometimes accomplished by using the PushButton editor to create a button which says "I am done."

Checkboxes are most often used when an application wishes to supply a set of options to the user, of which any number may be selected. For example, let's assume we have an application which allows a user to draw polygons, and to specify if the polygons should be filled, surrounded with a border, or patterned. The user may choose one or more of these options.

The example below demonstrates the steps required to create a checkbox instance using the above polygon example.

```
{
   xrCheckBoxInfo   cbInfo;
   xrEditor       * cbInstance;
   INT8             cb_values[3] = {TRUE, FALSE, TRUE};
   INT8           * cbLabels[] = {"BORDERED", "FILLED", "PATTERNED"};

   /*
    * Create a checkbox instance composed of three boxes, and laid
    * out in a single column.
    */

   /*
    * Ask the editor to supply the editorRect which
    * is to contain the checkbox instance.
    */
   cbInfo.editorFont = NULL;
   cbInfo.numFields = 3;
   cbInfo.numCols = 1;
   cbInfo.labels = cbLabels;
   XrCheckBox (NULL, MSG_SIZE, &cbInfo);

   /*
    * Offset the editorRect, so that the instance will be located
    * in the middle of the window.  Assume that the application has
    * saved a copy of the window dimensions in a variable named window_size.
    */
   xOffset = window_size.width/2 - cbInfo.editorRect.width/2;
   yOffset = window_size.height/2 - cbInfo.editorRect.height/2;
   XrOffsetRect (&cbInfo.editorRect, xOffset, yOffset);

   /* Now, ask the editor to create the desired instance. */
   cbInfo.editorWindowId = applic_window;
   cbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   cbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   cbInfo.editorState = (XrVISIBLE | XrSENSITIVE);
   cbInfo.stateFlags = NULL;
   cbInfo.values = cb_values;
   cbInstance = XrCheckBox (NULL, MSG_NEW, &cbInfo);
}
```

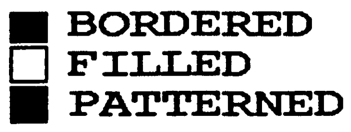

■ BORDERED
□ FILLED
■ PATTERNED

Figure 3-4. Sample Checkbox Instance

3.7 A Pushbutton Editor

The pushbutton editor allows an application to create and display a collection of related buttons in a series of rows and columns. The size of the buttons are dependent upon how the buttons are defined, and upon the font used to display them.

When an application creates a push-button instance, the editor allows the application as much flexibility as possible for customizing how the instance will be displayed. The options which may be specified by the application are:

- The number of rows and columns into which the buttons are to be displayed.
- The font to be used when displaying the button labels; this also determines the size of the buttons.
- The button labels themselves; these are optional, and an application is free to supply labels for some, or for all of the buttons.
- The foreground and background pens to be used each time the instance is displayed.

A pushbutton is drawn as an oval, with an optional label placed in the center. The label may be empty, be a single line of text, or contain multiple lines of text. When a select occurs within a pushbutton, the pushbutton will be redrawn as active (the oval will be filled with the foreground color). When the user releases the select button, the pushbutton will be redrawn as inactive (the oval will be filled with the background color). After the user has released the select button, an input event will be returned to the application informing it which button was selected. When the instance is drawn, it will be laid out so that all items in a row and column line up; this sometimes involves placing padding between buttons if the button labels are of different lengths.

Pushbuttons are most often used to bring about an immediate action. For instance, a panel usually has at least two pushbuttons defined, giving a user the option of exiting and saving the panel information, or of exiting and discarding the information. When a user selects one of the pushbuttons, the panel is immediately exited, and the current operation is ended.

Pushbuttons are also useful when an application wishes to provide a means for a user to signal that an action is complete, or that it is safe to continue with an operation. For example, let's assume that we have an application which can encounter error conditions to which the user must supply some indication of how they should be handled. The user's choices are to abort the operation, to ignore the error and continue the operation, or to retry the operation which failed. As soon as the user has selected a button, an action will occur and the program will remove the button instance.

The following example shows the steps for creating a pushbutton instance using the example above, and also using multi-lined labels:

```
{
   xrPushButtonInfo  pbInfo;
   xrEditor        * pbInstance;
   INT8            * pbLabels[]={"ABORT\enOPERATION",
                                 "CONTINUE\enOPERATION",
                                 "RETRY\enOPERATION"};

   /* Create push button instance, with three buttons in a single column*/

   /* Ask the editor for the rectangle which contains the instance */
   pbInfo.editorFont = NULL;
   pbInfo.numFields = 3;
   pbInfo.numCols = 1;
   pbInfo.defaultButton = -1;
   pbInfo.borderWidth = 2;
   pbInfo.labels = pbLabels;
   XrPushButton (NULL, MSG_SIZE, &pbInfo);

   /*
    * Offset the editorRect, so that the instance will be located
    * in the center of the window.  Assume that the application has saved
    * a copy of the window dimensions in a variable named window_size.
    */
   xOffset = window_size.width/2 - pbInfo.editorRect.width/2;
   yOffset = window_size.height/2 - pbInfo.editorRect.height/2;
   XrOffsetRect (&pbInfo.editorRect, xOffset, yOffset);

   /* Now, ask the editor to create the desired instance. */
   pbInfo.editorWindowId = applic_window;
   pbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   pbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   pbInfo.editorState = (XrVISIBLE | XrSENSITIVE);
   pbInfo.stateFlags = NULL;
   pbInstance = XrPushButton (NULL, MSG_NEW, &pbInfo);
}
```

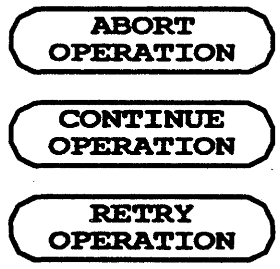


Figure 3-5. Sample Pushbutton Instance

3.8 A Text-Edit Editor

The text-edit editor provides application programs with a powerful line-editing facility. It allows applications to create and display single lines of editable text within a rectangular region of a window.

When an application creates a text-editor instance, it can define many of the instance's characteristics. This provides flexibility over how the instance will be displayed. The features which an application may control are:

- The location of the editor instance.
- The font to be used when displaying the editing string.
- The colors to be used when drawing the instance.
- The optional label to display next to the editing string.
- The position of the optional label.
- The character cell width.
- The type of character insertion mode to use.

When a text edit instance is displayed, the editing string is displayed with a rectangular border drawn around it. If an optional field label was specified, it will be displayed either to the left or to the right of the border, depending upon what was specified by the application. When the instance is available for editing, a text cursor will be displayed within the editing string. The text cursor indicates where the string is currently being edited. The text cursor will be displayed as either a vertical bar or as or a filled rectangle, depending upon the insertion mode the instance is currently configured for.

This editor supports most of the standard keyboard editing functions:

Text Insertion

> The text edit editor is capable of operating in both the normal and the insert-character modes. When operating in normal mode, the text cursor is displayed as a filled rectangle; typing a character will cause that character to be displayed at the cursor position. If a character is already displayed at the cursor position, it will be overwritten by the new character. When operating in insert-character mode, the text cursor is displayed as a vertical bar located between two character positions. Typing a character causes all characters to the right of the insertion bar to be shifted one character position to the right, and the new character to be inserted. The editor toggles between normal and insert-character mode each time the user presses the "insert character" key.

Backspace

> Causes the character to the left of the text cursor to be deleted and all characters to its right to be shifted one character position to the left; the text cursor also moves one position to the left.

Delete Character

> This causes the character at the text cursor position to be deleted, and all characters to the right of the cursor to be shifted one character position to the left. The position of the text cursor does not change.

Clear Line

> causes the character at the text cursor position and all characters to the right of the text cursor to be deleted; the position of the text cursor does not change.

Delete Line

> causes all characters in the editing string to be deleted and the text cursor to be moved

back to the start of the editing string area.

Cursor Left
: moves the text cursor one character position to the left.

Cursor Right
: moves the text cursor one character position to the right.

The text edit editor does not notify the application every time the user modifies a text string; it notifies the program only after a major event has occurred. These are referred to as break conditions and improve performance by not forcing the editor to 'handshake' with the application every time the user modifies the string. When an application does receive notification, it should perform whatever action is required under the circumstances of the notification. The following break conditions are supported by this editor:

- The input field is empty. This may occur when the last character in the editing string is deleted, or when the user deletes or clears the line.
- The first character was typed. This will occur when the user has typed a character into a previously empty editing string.
- An unknown XEvent is received, such as an exposure event, or an input timeout.
- The field is exited. This occurs when the user specifies that he has finished editing the string. This can be done by pressing the TAB, BACKTAB, or RETURN keys.

Text edit instances are most often used when an application needs to obtain a name (such as a filename) from a user. Applications rarely use text edit instances directly; an application will normally create a panel containing a text edit instance and some pushbuttons. The pushbuttons are provided to allow the user to indicate that he is done.

The following example demonstrates the steps required for an application to create its own text editing instance, without the benefit of a panel:

```
{
   xrTextEditInfo  teInfo;
   xrEditor      * teInstance;
   INT8            filename[21];
   INT8          * label = "File Name";

   /*
    * Create a text edit instance which allows a user to enter a filename.
    */

   /*
    * Obtain the coordinates of the rectangle needed to contain
    * the text editor instance.
    */
   teInfo.editorFont = NULL;
   teInfo.label = label;
   teInfo.maxChars = 20;
   teInfo.cellWidth = XrAVGWIDTH;
   XrTextEdit (NULL, MSG_SIZE, &teInfo);

   /*
    * Offset the editorRect so that the instance will be located
    * in the middle of the window.  Assume that the application has
    * saved a copy of the window dimensions in a variable named window_size.
    */
   xOffset = window_size.width/2 - teInfo.editorRect.width/2;
   yOffset = window_size.height/2 - teInfo.editorRect.height/2;
```

```
    XrOffsetRect (&teInfo.editorRect, xOffset, yOffset);


    /* Now, ask the editor to create the desired instance. */
    teInfo.editorWindowId = applic_window;
    teInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
    teInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
    teInfo.editorState = (XrVISIBLE | XrSENSITIVE);
    teInfo.insertPos = 0;
    teInfo.insertMode = XrINTERACTIVE;
    teInfo.labelPosition = XrLEFT_ALIGNED;
    teInfo.string = filename;
    filename[0]='\e0';
    teInstance = XrTextEdit (NULL, MSG_NEW, &teInfo);
}
```

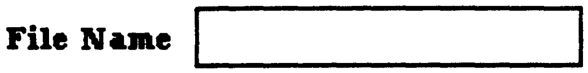


Figure 3-6. Sample Text Edit Instance

3.9 A Static Text Editor

The static text editor provides an application with a simple means for displaying a block of static strings within a window. No editing features are provided for the strings, and once the instance has been created, its contents cannot be changed.

As with all of the previous editors, the static text editor provides ample flexibility for an application when defining an instance. The instance characteristics which an application may control are as follows:

- The font to be used when displaying the string.
- The alignment to use when displaying each line.
- The pen colors to use when drawing the instance.
- The width of the region in which the text is to be displayed.

When an application creates a static text instance, it provides the definition of a rectangle into which the text will be drawn. The editor will parse each word of the text string and will generate lines which fit within the rectangle. If the bottom of the rectangle is reached before all of the text has been displayed, that portion of the text which remains will not be displayed. The editor does not draw a border around the text; this is left for the application to perform.

As each line of text is parsed, it will be aligned and then displayed. The editor allows text to be center, left, or right aligned.

An application most often uses static text for displaying feedback to the user. For instance, when an application creates a message box, it normally includes a message telling the user why the message box was displayed; static text would be used here.

In the following example, the application displays a block of static text in the center of its window.

```
{
   xrStaticTextInfo   stInfo;
   xrEditor         * stInstance;

   /*
    * Create a static text instance which displays several lines of text in
    * the center of the window and which left justifies each line of text.
    */

   /*
    * Ask the editor to supply an editorRect which will contain the
    * static text instance.  Specify 155 pixels as the width of the region
    * in which the text will be displayed.
    */
   stInfo.editorRect.width = 155;
   stInfo.string = "Unable to read the disk; Select new action";
   stInfo.editorFont = NULL;
   stInfo.alignment = XrLEFT_ALIGNED;
   XrStaticText (NULL, MSG_SIZE, &stInfo);

   /*
    * Offset the editorRect so that the instance will be located
    * in the center of the window.  Assume that the application has
    * saved a copy of the window dimensions in a variable named window_size.
    */
   xOffset = window_size.width/2 - stInfo.editorRect.width/2;
   yOffset = window_size.height/2 - stInfo.editorRect.height/2;
   XrOffsetRect (&stInfo.editorRect, xOffset, yOffset);

   /* Now, ask the editor to create the desired instance. */
   stInfo.editorWindowId = applic_window;
   stInfo.editorState = (XrVISIBLE | XrSENSITIVE);
   stInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   stInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   stInstance = XrStaticText (NULL, MSG_NEW, &stInfo);
}
```

Unable to read the disk;
Select new action

Figure 3-7. Sample Static Text Instance

3.10 A Raster-Select Editor

The raster-select editor provides an application with the means for displaying a group of *n* x *m* pixel raster images.

When an application creates a raster-select instance, it has some control over how the instance will be laid out. It specifies the number of raster boxes to display, the number of rows and columns into which the boxes are to be divided, and the dimensions of the raster images.

At any given time, only one raster box may be selected as the active choice. The active choice is indicated by a solid border surrounding the box. The user is able to interactively change the active box by using the pointer to select a new one. Whenever a new active box is selected, the patterned indicator will be moved to the new box and the application will be notified of the change.

Each raster box is specified by a pixmap Id which is obtained by registering the raster data with the appropriate "X" server. Once an instance has been created, an application is free to modify any of the patterns in the raster boxes. To do this, the application need only pass in the new pixmap Id for a particular raster box.

More often than not, it will be a graphics program which uses this type of editor, because this editor provides an ideal mechanism for allowing a user to select such things as an area-fill pattern or a line-drawing pattern.

The following example displays a series of four raster boxes. The boxes represent area-fill patterns which could be selected by a user. For this example, assume that the pixmap Id's have been obtained elsewhere, and are stored in a variable named "rasterIds."

```
{
   xrRasterSelectInfo    rsInfo;
   xrEditor            * rsInstance;
   INT16                 activeRaster = 1;
   extern Pixmap         rasterIds[];

   /* Create a raster select instance composed of 4 selectable boxes */

   /* Ask the editor for the editorRect containing the instance */
   rsInfo.rasterCount = 4;
   rsInfo.colCount = 2;
   rsInfo.rasterHeight = 16;
   rsInfo.rasterWidth = 16;
   XrRasterSelect (NULL, MSG_SIZE, &rsInfo);

   /*
    * Now, move the editorRect box so that it is located
    * in the center of the window. Assume that the application has
    * saved a copy of the window dimensions in a variable named window_size.
    */
   xOffset = window_size.width/2 - rsInfo.editorRect.width/2;
```

```
        yOffset = window_size.height/2 - rsInfo.editorRect.height/2;
        XrOffsetRect (&rsInfo.editorRect, xOffset, yOffset);

        /* Ask the editor to create the desired instance. */
        rsInfo.editorWindowId = applic_window;
        rsInfo.editorState = (XrVISIBLE | XrSENSITIVE);
        rsInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
        rsInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
        rsInfo.rasterIds = &rasterIds[0];
        rsInfo.activeRaster = &activeRaster;
        rsInstance = XrRasterSelect (NULL, MSG_NEW, &rsInfo);
    }
```

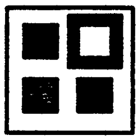


Figure 3-8. Sample Raster-Select Instance

3.11 A Raster Editor

The raster editor provides an application with the means for displaying a raster image, and then allowing a user to interactively modify (or edit) the image. This editor supports the concept of color within the raster image, and will allow the image to assume a pixel depth of one bit, or of one, two, three, or four bytes.

A raster editor is a useful tool for helping a user to design or modify custom raster images and icons. It can also be used by a graphics program to allow a user to define his own area fill pattern, or a new cursor shape.

The editor is fairly flexible concerning the size of the raster images it can handle in that it allows an application to specify an image as small as 1 by 1 pixels. The image need not be square.

When a raster editor instance is created, the application can specify how much of the raster image is to be displayed; this is referred to as the "view region." The view region can be from 1 x 1 pixels up to the size of the raster image. Only that portion of the raster image which lies within the view region will be visible and available for editing. If the image is larger than the view region, the user needs a mechanism for making the hidden portions of the image visible. When this situation arises, the editor will display two scrollbars; one to the right of the image for vertical scrolling, and one below the image for horizontal scrolling. These scrollbars provide the user with an easy mechanism for viewing and modifying the complete raster image.

The raster editor also allows an application to specify a pixel size value which will be used to expand each of the visible pixels within the image. This allows an application to shrink or enlarge the image, depending upon the display it is running on. For example, if an application requests a pixel size of 8, then each of the visible pixels will be drawn as an 8 by 8 square of pixels. This makes it easy for a user to modify individual pixels.

The editor provides a command, which allows the application to specify the pixel pattern to be used whenever the user selects a pixel within the raster image. It is up to the application to provide a user with the means for selecting the pixel pattern to be used.

The user can modify a raster edit instance in two ways:

- The user may select a pixel by moving the cursor over it and then pressing and releasing the "select" button. This causes the editor to change the selected pixel to whatever pixel pattern the application has set up. After the pixel has been modified, the editor will pass an input event to the application, informing it that the raster image has changed.
- The user may also modify a whole region of the image at once. This is accomplished by moving the cursor over a pixel, pressing the select button, and then moving the cursor. Each pixel that the cursor passes over will be changed to the pixel pattern specified by the application. When the user releases the select button, or upon receipt of any other XEvent, the operation completes and the editor passes an input event to the application, informing it that the raster image has changed.

The following example shows how an application could implement a simple icon editor. The raster image to be displayed will be 16 by 16 pixels with a pixel depth of one bit. The view region will also be defined as 16 by 16 pixels. This example also includes a set of radio buttons displayed above the raster editor instance. These radio buttons allow the user to select whether the edited bits in the raster image change to black, or to white. Assume that the raster data has been defined by a variable named rasterImage.

```
{
   xrRasterEditInfo  reInfo;
   xrRadioButtonInfo rbInfo;
   xrEditor        * reInstance;
   xrEditor        * rbInstance;

   /* Variables describing the raster editor instance */
   INT16         active_rb_button = 0;
   INT8         *rbLabels[] = {"BLACK", "WHITE"};
   extern UINT16 rasterImage[];

   /*
    * Create a raster-select instance in the center of the window.
    * Also display a two-button radio button instance which will
    * allow the user to select the editing color.
    */

   /*
    * Ask the radio button editor for the editorRect required to
    * contain the desired radio button instance.
    */
   rbInfo.editorFont = NULL;
   rbInfo.numFields = 2;
   rbInfo.numCols = 2;
   rbInfo.labels = rbLabels;
   XrRadioButton (NULL, MSG_SIZE, &rbInfo);

   /*
    * Ask the raster editor for the editorRect required to contain the
    * desired instance.
    */
   reInfo.imageData = XCreateImage(_xrCurrentDisplay,
                                   DefaultVisual(_xrCurrentDisplay,
                                      DefaultScreen(_xrCurrentDisplay)),
                                   XrBIT1, XYBitmap, 0, rasterImage,
                                   16, 16, 16, 0);
   reInfo.imageData->byteorder = LSBFirst;
   reInfo.imageData->bitmap_bit_order = LSBFirst;
   reInfo.pixelSize = 8;
   reInfo.viewRegion.width = 16;
```

```
    reInfo.viewRegion.height = 16;
    XrRasterEdit (NULL, MSG_SIZE, &reInfo);

    /*
     * Offset the two editorRects so that the radio buttons are stacked
     * above the raster editor instance in the center of the window.
     * Be sure to add some padding between the two instances.
     * (we'll use 8 pixels).
     */
    xOffset1 = window_size.width/2 - rbInfo.editorRect.width/2;
    xOffset2 = window_size.width/2 - reInfo.editorRect.width/2;
    yOffset1 = window_size.height/2 -
               (rbInfo.editorRect.height + reInfo.editorRect.height + 8) / 2;
    yOffset2 = yOffset1 + rbInfo.editorRect.height + 8;
    XrOffsetRect (&rbInfo.editorRect, xOffset1, yOffset1);
    XrOffsetRect (&reInfo.editorRect, xOffset2, yOffset2);

    /* Now, ask the editors to create the desired instances. */
    rbInfo.editorWindowId = applic_window;
    rbInfo.editorState = (XrVISIBLE | XrSENSITIVE);
    rbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
    rbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
    rbInfo.labels = rbLabels;
    rbInfo.value = &active_rb_button;
    rbInstance = XrRadioButton (NULL, MSG_NEW, &rbInfo);

    reInfo.editorWindowId = applic_window;
    reInfo.editorState = (XrVISIBLE | XrSENSITIVE);
    reInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
    reInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
    reInfo.pixelColor = 0;
    reInstance = XrRasterEdit  (NULL, MSG_NEW, &reInfo);
}
```

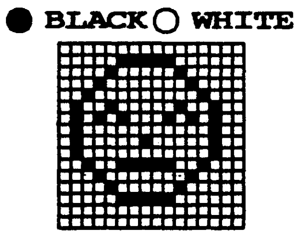


Figure 3-9. Sample Raster-Editor Instance

3.12 A Static-Raster Editor

The static-raster editor allows applications to display an uneditable raster image within a window. Once the image has been displayed, neither the application nor the user can change it.

Static-raster images are useful because they allow programs to display icons which help a user to understand what is happening. One common use for a static-raster image is in a message box. A message box is normally constructed from three parts:

- A series of buttons.
- A static-text string informing the user of some condition.
- A static-raster image indicating to the user why the message box is present. Sometimes the static-raster image is a STOP sign or a YIELD sign.

As with the raster-select editor, the static-raster editor does not expect an application to supply raster data. Instead, the application passes in a pixmap Id obtained by registering the raster data with the appropriate 'X' server.

For our example, we will display a simple 16 x 16 pixel icon, in the center of our window. We will assume that the raster image data is already registered with the server, and that the pixmap Id has been saved in a variable named rasterId.

```
{
   xrStaticRasterInfo   srInfo;
   xrEditor           * srInstance;
   extern Pixmap        rasterId;

   /* Create a 16 by 16 static raster image */

   /*
    * Ask the editor for the editorRect to contain our static raster image
    */
   srInfo.rasterHeight = 16;
   srInfo.rasterWidth = 16;
   XrStaticRaster (NULL, MSG_SIZE, &srInfo);

   /*
    * Offset the editorRect so that it is located in the center
    * of our window.  Assume that the application has saved a
    * copy of the window dimensions in a variable named window_size.
```

```
     */
    xOffset = window_size.width/2 - srInfo.editorRect.width/2;
    yOffset = window_size.height/2 - srInfo.editorRect.height/2;
    XrOffsetRect (&srInfo.editorRect, xOffset, yOffset);

    /*
     * Now, ask the editor to create the desired instance.
     */
    srInfo.editorWindowId = applic_window;
    srInfo.editorState = (XrVISIBLE | XrSENSITIVE);
    srInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
    srInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
    srInfo.rasterId = rasterId;
    srInstance = XrStaticRaster (NULL, MSG_NEW, &srInfo);
}
```

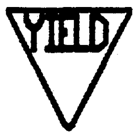


Figure 3-10. Sample Static-raster Instance

3.13 A Page Editor

The page editor provides an application with the means for allowing a user to enter and modify multiple lines of text data. A user can easily create both small and large documents using the cursor keys and the standard keyboard editing facilities.

The page editor operates in insert mode. This means that whenever a user presses a key, that key's character will be entered into the editing buffer at the position indicated by the cursor, and all characters to its right in the buffer will be shifted one character position to the right.

The application is responsible for specifying the size of the visible editing region which is itself specified in terms of lines of text and of character columns. When a user is entering text, the cursor automatically wraps to the next line when a line becomes full. If this line is the last visible line, the text will be scrolled up one line.

The page editor works with fixed-space fonts only; *proportional fonts are not supported.*

When a page editor instance is created, the application passes a pointer to the editing buffer and an indication of the buffer's size. If the buffer becomes full while the editing session is taking place, the editor will attempt to expand the buffer unless the application has disabled this feature.

When a page editor instance is activated by issuing a select within the editing region or by issuing a MSG_ACTIVATE, the editor retains control of the input channel until of the following conditions arise:

- A select occurs outside of the editing region.
- An unknown XEvent is received, such as an input timeout.
- The buffer becomes full and buffer expansion is disabled.

- The buffer becomes full and the buffer expansion fails.
- The buffer becomes full and has then been expanded.

The first of the above conditions is called a break condition. When this condition occurs, the page editor instance is deactivated and control is returned to the application. The editor will not expect to be invoked again until the next time it becomes active.

The other four conditions are referred to as status events wherein the editor temporarily returns control to the application. The application is expected to process the event and then to either re-invoke the editor using MSG_ACTIVATE or to make the page editor instance inactive using MSG_DEACTIVATE.

The following example shows how a page editor instance can be created:

```
{
   xrPageEditInfo    peInfo;
   INT8 * buffer;
   xrEditor * peInstance;

   /*
    *Create an instance of the page editor using a
    * 1K buffer and disabling buffer expansion.
    * The editing region is 24 lines by 80 columns.
    * Get the editor rectangle definitions */
   /*

   peInfo.rowCount = 24;
   peInfo.colCount = 80;
   peInfo.editorFont = XLoadQueryFont(_xrCurrentDisplay, "fixed");
   XrPageEdit (NULL, MSG_SIZE, & peInfo);

   /* Ask the editor to create the instance */
   peInfo.editorWindowId = applic_window;
   peInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   peInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   peInfo.editorState = XrVISIBLE | XrSENSITIVE;
   peInfo.buffer = (char*) malloc(1024);
   peInfo.bufferCount = 0;
   peInfo.bufferSize = 1024;
   peInfo.maxCharCount = 1024;
   peInfo.tabWidth = 5;
   peInstance = XrPageEdit (NULL, MSG_NEW, & peInfo);

   /* Activate the instance */
   XrPageEdit (peInstance, MSG_ACTIVATE,NULL);
}
```

3.14 A Scrollable List Editor

The list editor provides the application with the means for displaying a set of static strings. These strings are displayed in a single column, with an optional titlebar displayed above the strings, and a scrollbar displayed to the right of the strings.

If more strings are defined than can be displayed in the viewable portion of the list editor instance, then the user may use the scrollbar to scroll through the complete list contents.

Using the pointer, a user may select any of the list items, by moving the cursor over the desired item, and then pressing and releasing the pointer button configured as the 'select' button. Alternatively, the user may press the 'select' button, and then move the cursor; the list editor will track the cursor, highlighting the item over which the cursor is currently placed - when the button is released, the highlighted item will be selected.

The list editor also provides the facilities for adding additional items to an existing list, deleting items from an existing list, and replacing items in a list.

The following example demonstrates how a list editor instance would be created:

```
{
   xrEditor * leInstance;
   xrListEditInfo leInfo;
   INT8 * items[] = {"Elroy Taft",
                     "Rosey McKay",
                     "Astro Lee",
                     "George Duncombe",
                     "Spacely Hall",
                     "Cogsley Houser",
                     "Jane Miller"
   };
   INT8 selections[] = {TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE };


   /* Ask the editor for the rectangle enclosing this instance */
   leInfo.editorFont = NULL;
   leInfo.titleString = "Jetsons";
   leInfo.titleFont = NULL;
   leInfo.stringCount = 7;
   leInfo.strings = items;
   leInfo.maxStringWidth = 0;
   leInfo.viewCount = 5;
   XrListEdit (NULL, MSG_SIZE, &leInfo);

   /* Set the location for the list editor */
   leInfo.editorRect.x = 40;
   leInfo.editorRect.y = 40;

   /* Create the instance */
   leInfo.editorWindowId = applic_window;
   leInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   leInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   leInfo.editorState = (XrVISIBLE | XrSENSITIVE);
   leInfo.selectionStates = selections;
   leInfo.cellWidth = 0;
   leInfo.scrollPosition = 0;
   leInstance = XrListEdit (NULL, MSG_NEW, &leInfo);
}
```

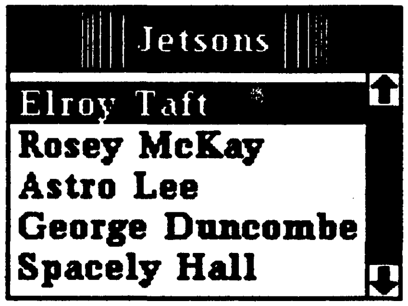


Figure 3-11. Sample Scrollable List Instance

3.15 A Groupbox Editor

The groupbox editor provides the application with the means for displaying a box with a label in the upper left corner. This is most often used to enclose a group of related editor instances, thus indicating that the editors are closely related to one another.

A groupbox editor instance is currently unique within the Xrlib field editor domain: it is the only field editor which does not have a selectable region (i.e. a hot spot) associated with it. This allows the groupbox to overlap the editor rectangles associated with the editor instances it contains, without producing any unfriendly side effects.

When creating a groupbox editor instance, the application will normally determine the region which is to be enclosed (if enclosing a group of editor instances, this may be done by taking the union of all of the editor rectangles), then adding some padding and space for the box border. The following formula may prove helpful when doing this, since they provide a general mechanism for taking any padding and the border width into consideration:

1) Determine the rectangle which describes the region to be enclosed.
2) Modify the rectangle, using the following formula, to take into account the border and any desired padding:

```
x = x - (borderWidth + horizontalPadding)
y = y - (MAX (borderWidth, fontHeight) + verticalPadding)
width = width + (2 * (borderWidth + horizontalPadding))
height = height + (2 * verticalPadding) + borderWidth +
      MAX(borderWidth, fontHeight)
```

NOTE: MAX() is a simple macro, which returns the larger of the two values passed in. It is defined as follows:

```
#define MAX(a,b)  ((a > b) ? a : b)
```

The following example outlines the steps for creating a groupbox editor instance, which will surround two text editor instances.

```
#define BORDER   7      /* Use a 7 pixel wide border */
#define VPADDING 2      /* Use 2 pixels of vertical padding */
#define HPADDING 5      /* Use 5 pixels of vertical padding */
#define MAX(a,b) ((a > b> ? a : b)

{
   xrEditor * gbInstance;
   xrGroupBoxInfo gbInfo;


   /*
    * For this example, we will assume that the application has already
    * created the two text editor instances elsewhere, and that their
    * instance pointers reside in 'teInstance1' and 'teInstance2'.
    */


   /* Create the groupbox instance */
   gbInfo.editorWindowId = applic_window;
   gbInfo.editorState = XrVISIBLE;
   gbInfo.editorFGColor = BlackPixel(_xrCurrentDisplay,_xrCurrentScreen);
   gbInfo.editorBGColor = WhitePixel(_xrCurrentDisplay,_xrCurrentScreen);
   gbInfo.editorFont = NULL;
   gbInfo.label = "Text Information";
   gbInfo.borderWidth = BORDER;
   XrUnionRect (&teInstance1->editorRect, &teInstance2->editorRect,
                &gbInfo.editorRect);

   /*
    * Modify the rectangle definition, to take into account the border
    * and the padding.
    */
   gbInfo.editorRect.x -= (BORDER + HPADDING);
   gbInfo.editorRect.width += 2 * (BORDER + HPADDING);
   gbInfo.editorRect.y -= MAX (BORDER, xrBaseFontInfo->height) + VPADDING;
   gbInfo.editorRect.height += (2 * VPADDING) + BORDER +
                               MAX (BORDER, xrBaseFontInfo->height);

   gbInstance = XrGroupBox (NULL, MSG_NEW, &gbInfo);
}
```

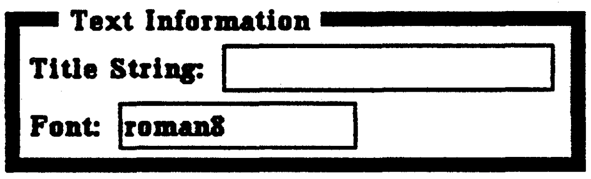


Figure 3-12. Sample Groupbox Instance

4 Dialogs

This section describes the dialog level of Xrlib. It explains how to create and use panels, menus, and message boxes – the major tools used to communicate with users.

This chapter assumes that you're familiar with:

- The X11 library.
- X11R intrinsics.
- X11R field editors.

Dialogs provide the highest level of communication with the user. Xrlib provides three types of dialogs:

- **Menu.** For presenting and selecting one user choice out of many that may be available.
- **Message box.** For presenting a message or simple choice to the user.
- **Panel.** For presenting several types of information and collecting appropriate user input.

Each type of dialog presents to the user a separate window that's dedicated to a particular type of communication with the user.

Many times a program must warn the user or ask a simple multiple choice question. Message boxes provide this capability in an easy-to-use format. When a message window appears on the display, the user must respond to the question or statement displayed by selecting one of the choices.

A message box is made up of an icon (a symbol), some descriptive text, and buttons. Icon placement, text formatting, and button placement is performed automatically by the message box manager. If you need to perform a more complex interaction, you should use a panel.

Panels are generally used to display the state of an application and allow the user to change that state. A panel is a collection of *field editors*, which can manipulate the state of an application, contained in a single window.

4.1 Menus

Menus allow users to view the choices available to them at any one time without having to remember command words or special keys or even to select an option. In *Xrlib*, menus are lists of items (text strings, lines, etc.) displayed in a pop-up window. A menu is displayed with selectable items in bold and unselectable items in gray.

To select a menu item, move the pointer to the item and press a select button or key. (The selection process depends upon the human interface model.) This process and others are configurable.

4.1.1 About Menus

Selecting a menu item involves pointing at an item, which highlights it, and then selecting it. The selection information is returned to the application through XrInput().

An Xrlib menu can be thought of as a tree. The leaves of the tree are the menu items, which represent commands or options. The branches of the tree are paths to *submenus*. Submenus are menus that have been linked to higher level menus.

The user's view and interactions with the Xrlib menuing system are configurable. Configuring menus is described under "Initializing Xrlib" in chapter 3.

Using Xrlib Menus. A user brings up a menu by performing a *MenuPost* action (described under "Initializing Xrlib," in chapter 3). The default method for performing this action is pressing the right button on a pointing device. When this is done, the menu associated with the window under the cursor is posted. The menu is posted as near the cursor as possible while keeping the entire menu on the display. When the menu is posted, moving the cursor within the menu highlights the menu item the cursor is over. The user may select one of the items in the posted menu by pressing the *MenuSelect* button on the pointing device. (MenuSelect is also configurable.)

If the posted menu contains submenus, they will appear when the user moves the cursor into the right quarter of the item the submenu is associated with.

The menu disappears when the user does one of two things:

- Selects (using the MenuSelect button) an item.
- Selects outside the menu or over an unselectable area.

4.1.2 Menu Messages and Structures

To communicate with the menu manager, the application uses menu messages and structures. This section describes the calling sequences, messages, and structures used with menus.

The menu manager is called with the following syntax:

```
xrMenu * XrMenu(menuInstance, message, data)
   xrMenu   * menuInstance;
   INT32      message;
   INT8     * data;
```

menuInstance is a pointer to the menu of interest that is returned when the menu is created. *message* is the command to be performed by the menu manager. *data* is a pointer to a structure or scalar value containing the information needed by the message given. The type of structure pointed to by *data* varies depending on the message sent.

Menu Messages. The application controls menus using messages, which are menu manager commands. The messages XrMenu() understands are described in the following list.

- MSG_NEW

  MSG_NEW creates the xrMenu structure described earlier. The *menuInstance* parameter should be set to NULL, because it is not used. The *data* parameter is a pointer to an *xrMenuInfo* structure. This structure is set totally by the programmer. Its contents will determine the appearance of the menu to be created. This structure is:

```
typedef struct
{
    INT8              * menuTitle;
    INT8             ** menuItems;
    INT32               numItems;

    xrPanelContext    * menuContext;
    INT16               menuId;
    INT32               menuStyle;
} xrMenuInfo;
```

menuTitle	*menuTitle* is a NULL-terminated string that will be used as the title of the menu in an XrTitleBar editor. If this value is NULL, no title will be displayed. If an empty string is given, no title will be displayed.
menuItems	*menuItems* is an array of NULL-terminated strings that specify the strings displayed in a menu. These strings may also contain commands to the menu manager. (These commands are described under "Menu Item Language," below.)
numItems	*numItems* is the number of strings in the menuItems array.
menuContext	*menuContext* points to a panel context structure described in the panel chapter. Setting this pointer to NULL gives defaults for all its values. The winBackground, winForeground, borderWidth, foregroundColor, backgroundColor, fontInfo, and cursor fields of the menuContext structure are used by the menu manager.

Note

If colors other than the defaults are to be used, the background and foreground pixmaps will normally need to be created to match these colors. Failure to do this will produce strange effects for some color combinations.

menuId	*menuId* is returned in the xrEvent structure when a menu item is selected along with the menu instance pointer and item selected. This value gives the programmer a scalar value to identify the menu.
menuStyle	*menuStyle* is not currently used but is reserved for future use.

(Filling out an *xrMenuInfo* structure is described under "Using Menus," below.)

MSG_NEW returns a pointer to the menu created if successful, or a NULL on failure.

- MSG_FREE

MSG_FREE destroys a menu and frees all resources allocated to the menu by the menu manager. *menuInstance* is a pointer to the menu to be freed. *data* should be set to NULL. Care should be taken *not* to refer to a menu after it has been freed. Xrlib doesn't check for a valid menu pointer. MSG_FREE returns TRUE on success, NULL on failure.

- MSG_EDIT

MSG_EDIT forces the display of any currently active menu for a given window. This message is usually made transparent to the application by *XrInput*(), but may be called directly if desired. *data* should point to an *XEvent* structure which has its' *type* field set to *ButtonRelease*. For the duration of the call to *XrMenu*, the global variable XrMENUITEMSELECT should be changed from its' default value (XrRIGHTBUTTONUP), to that of XrRIGHTBUTTONDOWN. When *XrMenu* has received a menu selection event, it places it at the head of the input queue where it can be accessed by a subsequent call to *XNextEvent*. Following this sequence, it is important to remember to restore the XrMENUITEMSELECT variable to its' original state.

(The values returned in an *XrMenu* event are described under "Using Menus," below.)

- MSG_ACTIVATEMENU

MSG_ACTIVATEMENU makes the specified menu the current menu for the given window. A menu may be active for several windows at the same time. When the user requests a menu be displayed for a window, the active menu will be displayed. *data* is an X window id.

Note

The window to which a menu is attached MUST have been registered with Xrlib with the MSG_ADDWINDOW command BEFORE adding the menu.

- MSG_DEACTIVATEMENU

  MSG_DEACTIVATEMENU deactivates the given menu for the given window. If a menu is active on a window, you *must* deactivate it before activating a new one. This is true even if re-activating the same menu on a window. *data* is an X window id.

- MSG_ADDSUBMENU

  MSG_ADDSUBMENU allows you to add one menu to another as a submenu. The submenu is created in the same way as the parent menu – this message simply allows you to link menus into a tree structure. *menuInstance* is a pointer to the parent menu. *data* points to an *xrMenuIndex* structure that specifies the location in the menu tree.

```
typedef struct
{
   struct _xrMenu     *  menuInstance;
   INT32                 itemIndex;
   INT32                 itemData;
} xrMenuIndex;
```

This structure is used when setting the attributes of menu items. The value passed in the itemData parameter will vary depending on the message.

Set the *menuInstance* pointer to the submenu instance pointer. Set the *itemIndex* portion of the structure to the item in question, and the *itemData* member to NULL.

Note

A menu should not appear more than once in a menu path. This will cause unexpected results in screen repainting. The details of using submenus is given under "Using Menus" below.

- MSG_REMOVESUBMENU

  MSG_REMOVESUBMENU removes a link between two menus. Neither of the menus are destroyed, there is simply not a path between them any longer. *menuInstance* is a pointer to the parent menu. *data* points to an *xrMenuIndex* structure that specifies the location in the menu tree. Set the *menuInstance* pointer to the submenu instance pointer. Set the *itemIndex* portion of the structure to the item in question, and the *itemData* member to NULL.

- MSG_ACTIVATEITEM

  MSG_ACTIVATEITEM allows you to make an item in a menu selectable (default). Unselectable text items are shown in grey and are not selectable by the user. *data* is an integer indicating the item to be made selectable.

- MSG_DEACTIVATEITEM

  MSG_DEACTIVATEITEM allows you to make a text item in a menu unselectable. Unselectable text items are shown grey and are not selectable by the user. *data* is an integer indicating the item to be made unselectable.

- MSG_SETITEMFUNCT

  MSG_SETITEMFUNCT sets a function to be executed when a menu item is selected by the user. *data* points to an *xrMenuIndex* structure. The *itemIndex* member determines which item is associated with the function, and the *itemData* member contains the pointer to the function. The function should be type INT32 and expect no parameters. Parameters may be passed by using an item function in conjunction with an item event (described below.)

Note

If your function reads an event from the event queue, or calls functions that do (i.e. *XrPanel*(), *XrMessageBox*()), it *must* push an event back on the queue before exiting.

- MSG_SETITEMEVENT

  MSG_SETITEMEVENT sets an event to be returned when a menu item is selected by the user. The *data* variable points to an *xrMenuIndex* structure. The *itemIndex* member determines which item is associated with the event, and the *itemData* member contains the pointer to the event. The event supplied must be of XEvent or xrEvent type. This is because XEvent and xrEvent are the only event types that are guaranteed to be large enough to contain any other event type. When an item event is defined in a menu, only that event is pushed on the input queue by the menu manager. The normal menu event is not pushed. This feature allows the programmer to receive events of their own creation from the menu manager as this event will be returned instead of a menu event when the item is selected. It is also useful to use in conjunction with item functions to pass them parameters.

Menu Item Language. The MenuItems field is a pointer to an array of text strings that contain both text to be displayed and escaped commands. The commands are preceeded with by '\\'. The Xrlib toolbox also provides messages that are passed in with menu item strings to inform the menu manager that a particular item is special. Commands used in conjunction with text must precede the text in the string. The following table describes the messages allowed.

Command	Action
\\DA	Item is disabled. (unselectable).
\\KE	Item has keyboard equivalent c.
\\-	Single line (unselectable).
\\=	Double line (unselectable).

- \\DA

  The item in this string is disabled when the menu is created. It is displayed in a half-tone and can not be selected by the user.

- \\KEc

  The item has keyboard equivalent *c*, where c is any capital letter between A and Z that is always mapped into a control character. If control *c* is typed from the keyboard by the user, you will receive an *xrEvent* of type *XrMENU*, just as if the user had selected the item normally. This will occur even if the menu is not currently visible. If the menu is active, control keys described in the menu items will generate XrMENU events. This mechanism is provided so experienced users may bypass selecting menu items with the mouse by pressing control keys.

- \\-

  A dash causes an unselectable line to be drawn in the menu. This is useful for dividing groups of menu items. This command should appear by itself as a menu item with no other text.

- \\=

  An equals sign will cause an unselectable double line to be drawn in the menu. This command should appear by itself as a menu item with no other text.

4.1.3 Using Menus

This section concentrates on relatively simple use of the Xrlib menuing system. This section describes how to create and maintain menus and submenus that display text. (More advanced features are described under "Advanced Use of Menus," below.)

Creating a Menu. Creating a menu consists of the following steps:

1. Filling out an *xrMenuInfo* structure by creating an array of menu items and a menu title.
2. Sending MSG_NEW to the menu manager with the info structure created in step 1.
3. Adding any submenus to your menu.
4. Sending MSG_ACTIVATEMENU to the menu manager with the window id of the menu the window is associated with.

Note

The window to which a menu is attached must be registered with Xrlib using the MSG_ADDWINDOW command before adding the menu.

Each of these steps will now be discussed in detail.

The first step in creating a menu is to fill out an *xrMenuInfo* structure. This is a simple matter of setting a title string and an array of item strings to appropriate values. A typical example might be:

```
xrMenu      * myMenu;
xrMenuInfo    myMenuInfo;
INT8        * myMenuTitle = "myMenu";
INT8        * myMenuItems[7] = ( "Item1",
                                 "\e\eDAItem2",
                                 "Item3",
                                 "Item4",       /* Step 1 */
                                 "submenuname",
                                 "\e\e=",
                                 "Item7"
                               );
```

With these values set at initialization, the code to fill out the info structure and create the menu is:

```
myMenuInfo.title = myMenuTitle;
myMenuInfo.menuItems = myMenuItems;
myMenuInfo.numItems = 7;
myMenuInfo.menuId = 1;

myMenu = XrMenu(NULL, MSG_NEW, &myMenuInfo); /* Step 2 */
```

The fifth item in this menu is a link to a submenu. This submenu is created in the same fashion as above. Assume you have filled out:

```
xrMenu      * mySubMenu;
xrMenuInfo    mySubMenuInfo;
xrMenuIndex   subMenuPosition;
```

Create the submenu as above with a MSG_NEW. Set the menuId value to a different number than the other menu. This will give you an easy way to distinguish between the menus when receiving input. Then add the submenu with the following code:

```
subMenuPosition.menuInstance = mySubMenu;
subMenuPosition.itemIndex = 5;              /* Step 3 */
XrMenu(myMenu, MSG_ADDSUBMENU, &subMenuPosition);
```

The menu is ready to add to a window. Assuming you created a window *myWindow*. The menu is added to myWindow as follows:

```
XrMenu(myMenu, MSG_ACTIVATEMENU, myWindow);   /* Step 4  */
```

The menu will be posted when the application is doing an *XrInput*() with the message MSG_BLKHOTREAD or MSG_NONBLKHOTREAD and the event which corresponds to XrMENUPOST occurs.

Note

If you wish to add and remove submenus dynamically from a menu tree, you must deactivate and then reactivate the main menu after changing the submenus.

Getting Input From a Menu. Getting input from a menu is a simple matter of responding to xrEvents of inputType *XrMENU*. For example, an event of this type will place the following values in the event variable *xrInput*.

```
xrInput.type       => XrXRAY
xrInput.serial     => 0
xrInput.send_event=> 0
xrInput.display    => Pointer to current display
xrInput.source     => Window id (window menu was posted for)
xrInput.inputType  => XrMENU
xrInput.value2     => myMenu.menuId
xrInput.value3     => selected item index
xrInput.valuePtr   => menu instance pointer
```

The following code fragment shows a typical application receiving input from the menu manager.

```
        xrEvent xrInput;

        XrInput(NULL, MSG_BLKNOTREAD, &xrInput);

        xrEvent xrInput = (xrEvent *) &myinput;/* Cast XEvent to xrEvent */

        switch(xrInput.inputType)  { /*  what kind of event?  */
                .
                .
                .
        case XrMENU:
           switch (xrInput.value2) { /*  Which menu?        */

           case 0:        /* Main Menu */
             switch(xrInput.value3) {

             case 0:     /* ITEM 1 */
                .
                .                /* process first item of main menu  */
                .
                break;

             case 1:     /* ITEM 2 */
                .
                .                /* process second item of main menu */
                .
                break;
             case 2:    /* ITEM 3 */
                .
                .                /* process nth item of main menu    */
                .
                break;
             }
           case 1:            /*  Selected sub menu    */
             switch(xrInput.value3) {
                .
                .                 /* process submenu items */
                .
             }
           }
        }
```

Removing a Menu. Removing a menu is a simple matter of sending this message to the menu manager:

```
        XrMenu(myMenu, MSG_FREE, NULL);
```

where *myMenu* is your menu. This action frees all resources associated with the given menu that were allocated by the menu manager. Be careful *not* to reference a menu after it has been freed. (If the menu being freed is a submenu, be sure to remove it from the menu tree before it is freed.)

Note

XrMenu sets the save_under window attribute to **true**. If the X server you are using has the save_under feature implemented, then any window obscured by a menu will be restored when the menu is destroyed. See *'Xlib - C Language Interface, Protocol Version 11'* for more information.

4.1.4 Advanced Use of Menus

The previous section concentrated on the basic use of menus. This section discusses advanced topics, such as keyboard equivalents, item function execution, and item events.

Menu Keyboard Equivalents. If you are developing a program and expect some of the users to executed commands from the keyboard, you should use *keyboard equivalents.* The following declaration shows a menu using keyboard equivalents.

```
INT8      * myMenuItems[3] = {  "\e\eKEXCut",
                                "\e\eKEPPaste",
                                "\e\eKECCopy",
                              };
```

A user typing control C (case independent), will cause you to receive an xrEvent of type XrMENU, with the proper indices to show you the Copy command was selected. The keyboard equivalent will be shown in the menu to the right of the menu item.

Keyboard equivalents are updated only when the main menu for a window is made active or is deactivated. To add and remove submenus dynamically, deactivate then reactivate the main menu to update the keyboard equivalents.

Menu Item Functions. The Xrlib menu manager allows you to specify a function to be executed directly when a user selects a menu item. The function should be of type INT32 and accept no parameters. If you specify a menu item function for a menu item, that function is called before the XrMENU event is pushed onto the input queue. (If you need to pass parameters to the function, use the function in combination with a menu item event and pass the parameters in the event, as described below under "Menu Item Events.")

The following example demonstrates how to define a menu item function (without parameters). It assumes you have a function called "Beeper" that turns a beeper on and off.

```
xrMenu        * myMenu;
xrMenuInfo      myMenuInfo;
xrMenuIndex     functionIndex;
INT8          * myMenuTitle = "myMenu";
INT8          * myMenuItems[7] = {  "BEEPON",
                                    "\e\eDABEEPOFF",
                                    "Item3",
                                    "Item4",
                                    "submenuname",
                                    "\e\e=",
                                    "Item7"
                                  };
```

```
        myMenuInfo.title = myMenuTitle;
        myMenuInfo.menuItems = myMenuItems;
        myMenuInfo.numItems = 7;
        myMenuInfo.menuId = 1;

        myMenu = XrMenu(NULL, MSG_NEW, &myMenuInfo);

        functionIndex.itemIndex = 0;
        functionIndex.data = (INT32) &Beeper;

        XrMenu(myMenu, MSG_SETITEMFUNCT, &functionIndex);

        functionIndex.itemIndex = 1;

        XrMenu(myMenu, MSG_SETITEMFUNCT, &functionIndex);
```

This menu allows a user to turn beeping on and off in the program.

When the user selects BeepOff, for example, Beeper is called directly by the menu manager.

Note

If your function reads an event from the event queue, or calls functions that do (i.e. *XrPanel*(), *XrMessageBox*()), it MUST push an event back on the queue before exiting.

Menu Item Events. The Xrlib menu manager allows you to specify an XEvent or xrEvent to be returned by the menu manager when an item is selected. The event specified **must** be of XEvent or xrEvent type. This is because XEvent and xrEvent are the only event types that are guaranteed to be large enough to contain any other event type. Each item in a menu could have its own XEvent or xrEvent to return when selected. The values in the event could change along with the program state as the event is owned by the programmer. When an item event is defined in a menu, only that event is pushed on the input queue by the menu manager. The normal XrMENU event is not pushed.

In the following example, assume you have a function called Beeper that turns a beeper on and off. Since the same function was used before, assume it has been set up the same way in this example.

The function now needs parameters for the duration and tone of the beep. These values have been saved in an event called *beepEvent*. The following code fragment shows how to attach this event to the proper menu items.

```
        xrMenu      * myMenu;
        XEvent        beepEvent;
        xrMenuInfo    myMenuInfo;
        xrMenuIndex   eventIndex;
        INT8        * myMenuTitle = "myMenu";
        INT8        * myMenuItems[7] = { "BEEPON",
                                         "\e\eDABEEPOFF",
                                         "Item3",
                                         "Item4",
                                         "submenuname",
                                         "\e\e=",
                                         "Item7"
                                       };
        myMenuInfo.title = myMenuTitle;
```

```
myMenuInfo.menuItems = myMenuItems;
myMenuInfo.numItems = 7;
myMenuInfo.menuId = 1;

myMenu = XrMenu(NULL, MSG_NEW, &myMainMenu);

eventIndex.itemIndex = 0;
eventIndex.data = (INT32) &beepEvent;

XrMenu(myMainMenu, MSG_SETITEMEVENT, &eventIndex);

eventIndex.itemIndex = 1;

XrMenu(myMainMenu, MSG_SETITEMFUNCT, &eventIndex);
```

Be sure that the beeper function pushes an event back on the input queue after looking at the beep parameters.

Menu Structure. The menu manager keeps all the information it needs for its operations on a particular menu in a menu structure. The programmer doesn't need to access this; however, when designing a program, it is helpful to know what is in this structure. These structures are shown below:

```
typedef struct
{
   xrEditor          * menuEditor;
   Window              menuWindow;
   Cursor              menuCursor;
   INT32               totalWidth;
   INT32               totalHeight;
   INT16               menuId;
   INT32               menuStyle;
   POINT               menuOrigin;
   INT32               borderWidth;
   Window            * currentWindows;
   INT32               numWindows;
   xrMenuIndex       * currentPath;
   INT32               pathLength;
   Pixmap              menuPixmap;
   INT32               stickyMenu;
} xrMenu;
```

The xrMenu structure is used internally by the menu manager. Its structures are accessed through the the messages described earlier. You will not normally deal with the members of this structure directly, the XrMenu structure is created for you by the menu manager when you create a menu.

menuEditor	*menuEditor* is the menu editor instance for this menu. You will not need to deal with this editor directly, but rather through the menu item language and menu messages.
menuWindow	*menuWindow* is the window identifier for the given menus' window.
menuCursor	*menuCursor* is the cursor identifier for the cursor to be used with the menu. This value is given by the programmer when the menu is created.
totalWidth,	*totalWidth and totalHeight* are the size of the given menu including its window borders.
menuId	*menuId* is set by the programmer when creating a menu. This value is returned in the xrEvent structure when a menu item is selected along with the menu instance pointer and item selected. This value gives the programmer a

	scalar value to identify the menu.
menuStyle	*menuStyle* is not used at this time but is reserved for future use.
menuOrigin	*menuOrigin* is set to the origin of the menu window while the menu is displayed.
currentWindows	*currentWindows* is an array of windows to which this menu belongs. Each window may have its own menu, and a menu may be shared among windows. The MSG_ACTIVATEMENU message adds an existing menu to a window.
numWindows	*numWindows* gives the length of the previous array.
currentPath	*currentPath* is a pointer to an array of menu indices indicating the last path taken through this menu to select an item. This value is useful if the path taken to select an item is needed by the application. This value is set dynamically by the menu manager as the menu is used.
pathLength	*pathLength* gives the length of the previous array.
menuPixmap	*menuPixmap* is used internally by the menu manager while a menu is displayed.
stickyMenu	*stickyMenu* is currently not implemented. [This is an internal structure, shown for reference only.] It may be ignored.

4.2 Message Boxes

Many times an application needs to warn a user or ask a simple multiple choice question. Message boxes provide this capability and are easy to program.

A message box is made up of any combination of an icon, some descriptive text, and push buttons. Icon placement, text formatting, and button placement is handled by the message box manager automatically.

Message boxes are inherently transitory–they exist only while their window is showing on the display. A message box is removed from the display when a user presses one of the push buttons, a keyboard key (when the cursor is over the window), or the message box times out. More complex interactions are handled by panels.

Message boxes also show a way to program panels easily. If you are designing a panel needed by many people, consider making it a dialog manager for ease of programming.

4.2.1 Message Box Structures and Messages

Message Box structures and messages allow a programmer to communicate with the Message Box manager. This section describes the structures, calling sequences, and messages used with message boxes.

Communication with the message box manager takes the following form:

```
XrMessageBox(msgInfo, message, data)
   xrMsgBoxInfo  * msgInfo;
   INT32           message;
   INT8          * data;
```

- *msgInfo* contains the data used to create the message box.
- *message* is the message sent to the message box manager. (Currently only MSG_EDIT and MSG_SIZE.) and
- *data* is a pointer to an xrEvent structure in the case of MSG_EDIT, to a RECTANGLE structure in the case of MSG_SIZE. The result of the dialog with the user will be returned in the event structure with MSG_EDIT. The size of the message box when it is displayed will be returned in the rectangle passed with MSG_SIZE.

Message Box Structure. The message box manager acquires all the information it needs about a message box in a message box information structure. This structure is shown below:

```
typedef struct
{
   POINT              messageOrigin;
   Window             relativeTo;
   xrPanelContext   * messageContext;

   INT32              rasterHeight;
   INT32              rasterWidth;
   Pixmap             rasterId;

   INT8             * messageText;

   INT8            ** messageButtons;
   INT32              numButtons;
} xrMsgBoxInfo;
```

messageOrigin	*messageOrigin* is the origin of the window that contains the message box.
relativeTo	*relativeTo* is a window the message box should be created relative to. If this field is NULL, the message box will be created relative to the root window.
messageContext	*messageContext* is a structure of parameters for the panel used to create the message box. This structure is described in the section on panels.
rasterHeight	*rasterHeight* is the height of the pixmap to be used in the message box.
rasterWidth	*rasterWidth* is the width of the pixmap to be used in the message box.
rasterId	*rasterId* is an X Pixmap id that identifies the raster image to be placed in the message box.
messageText	*messageText* is an unformatted, NULL-terminated, text string. This string is shown (formatted by the static text editor), in the middle of a message box window. No text will be shown if messageText is NULL. Refer to *XrStaticText*() for the rules on formatting text.
messageButtons	*messageButtons* is an array of strings used as labels in the buttons of a message box. The buttons are formatted by the *XrPushButton* editor described under "*XrPushButton*()." The creation of a message box will not fail if this parameter is NULL, but if there are no buttons, the user will need to press a keyboard key, or the timer (in *messageContext*) should be set to remove the dialog after a specified time. (Using the timer is described under "Panels," below.)
numButtons	*numButtons* indicates the number of strings held within *buttonStrings*.

Message Box Messages. Message Box currently has only two messages: MSG_EDIT, and MSG_SIZE. MSG_SIZE returns the size a message box would be if it were created with the given parameters. MSG_EDIT together with an xrMsgBoxInfo structure displays a message box window and makes it active. The output from a message box is returned through the event passed to the message box manager with MSG_EDIT. MSG_EDIT returns TRUE if the message box was created and used successfully, FALSE if not.

4.2.2 Using Message Boxes

Use of message boxes is very simple, involving these steps:

1. Creating any combination of an icon, text string, and an array of button labels.
2. Filling out a message box info structure.
3. Sending a MSG_EDIT (or MSG_SIZE) to the message box manager.
4. Inspecting the event passed to the message box manager to determine the results of the dialog with the user.

The values returned in a *XrMESSAGEBOX* event are shown below:

```
type      => XrXRAY
inputType => XrMESSAGEBOX
inputCode => XrPUSHBUTTON or XrPANELTIMER
value1    => Button index (-1 if no button selected)
```

If a user presses a keyboard key, the index of the first button in the panel (index 0) will be returned in value1.

The following code segment demonstrates the steps to create and use a message box.

```
Window          myWindow
xrEvent         messageEvent;
xrMsgBoxInfo    myMessageBox;    /* message box declarations */
Pixmap          myPixmap = /* pixmap */
INT8          * myMessage = " Way to go, Ed, Fred, Rick, Tom!";
INT8          * myButtons[] = {"East", "West"};
                .
                .
                .
XrCopyPt(myMessagePt, myMessageBox.messageOrigin);
myMessageBox.relativeTo = myWindow;
myMessageBox.messageContext = NULL;  /* uses default message context */



myMessageBox.rasterId = myPixmap;
myMessageBox.rasterHeight = height;
myMessageBox.rasterWidth = width;
myMessageBox.messageText = myMessage;
myMessageBox.messageButtons = myButtons;
myMessageBox.numButtons = 2;       /* (step 2)     */
                .
                .
                .
                /* Message Box call (3) */
   XrMessageBox(&myMessageBox, MSG_EDIT, &messageEvent))

                /* get results (4) */
```

```
        /* Check value1 to see which button was selected */
```

NOTE. When using the MessageBox manager, it is possible for incorrect state information to occur in other windows of the application. This incorrect information will become visible if the user selects that particular window while the message box is up. This problem is caused by the message box manager consuming all events generated for the application. The symptoms include: an application not being redrawn properly, checkboxes showing state information inconsistent with the application, and others.

To avoid this problem, use the following method:

1. Set up a window function using the call: *XrInput(Window, MSG_ADDWINDOWFUNCT, info)* for exposure events on all your registered windows, as described in sample program #4, section 1.5.4, in the function *DrawWindow*(). This function will be called automatically when an exposure event occurs on any of your registered windows. The function should take care of redrawing your windows.
2. Save the event mask for all of your windows using the call: *XGetWindowAttributes*().
3. Set the event mask for all of your windows to: *ExposureMask*.
4. Call the message box manager.
5. Set the event mask back to its original setting.

The first step should be performed at the beginning of your program. The other steps should be performed each time a message box is used.

The following code segment demonstrates the workaround:

```
    XWindowAttributes wattr;
    XrInput(myWindow, MSG_ADDWINDOWFUNCT, &info);
                    .
                    .
                    .
    XGetWindowAttributes(_xrCurrentDisplay, myWindow, &wattr);
    XSelectInput(_xrCurrentDisplay, myWindow, ExposureMask);
    XrMessageBox(&myMessageBox, MSG_EDIT, &messageEvent);
    XSelectInput(_xrCurrentDisplay, myWindow, wattr.your_event_mask);
```

4.3 Panels

4.3.1 About Panels

Panels are one of the most powerful tools provided by Xrlib. They are generally used to display the state of an application and allow the user to change that state. For example, the message box manager uses a panel.

A panel is a collection of field editors displayed in a window and controlled by the *panel manager*. The panel manager is in charge of the following functions:

- Panel display.

- Passing control to a field editor selected by a user.
- Routing editor activity to the application.
- Activating multiple panels per process.
- Managing subpanels.

When the panel manager is asked by the application to create a panel, it sends messages to the field editors in the panel instructing them to create instances of themselves, display themselves, make themselves active, and so on.

Optionally, the panel may be sized and positioned on screen by the user via the window manager. Otherwise, the panel will be sized and positioned directly by the application. Refer to the MSG_SETOVERRIDEMODE option.

When the user makes any change to a panel, such as entering text or checking a check box, the panel manager passes control to the appropriate editor. The editor displays any changes and then returns control to the panel manager. The panel manager then routes the output from the editor back to the application. The changes made are noted by the application, which makes appropriate changes to its own data structures.

Notice especially that the Xrlib concept of what can be in a panel is not limited to the usual letters and numbers of forms on alpha terminals. Because panels use editors to manage their fields, a field can contain almost any kind of information. This aids the programmer in that many more difficult-to-program user interactions are easily expressed by a panel.

4.3.2 Panel Messages and Structures

The panel structures and messages allow a programmer to communicate with the panel manager. This section describes the structures, calling sequences, and messages used with panels.

All communication with panels are achieved with the following syntax:

```
xrPanel * XrPanel(panelInstance, message, data)
   xrPanel  * panelInstance;
   INT32      message;
   INT8     * data;
```

panelInstance is a pointer to the panel of interest, and is returned when a panel is created. *message* is the message sent to the panel manager, and *data* is a pointer to a structure or a scalar, containing information needed by the panel manager and associated with the message. The type of structure *data* points to varies depending on the message sent.

Panel Messages. Panels are controlled with a set of messages. The messages XrPanel() understands are described in the following list. Remember, all messages to the panel manager are given with the following syntax.

```
xrPanel * XrPanel(panelInstance, message, data)
   xrPanel  * panelInstance;
   INT32      message;
   INT8     * data;
```

- MSG_NEW

  MSG_NEW creates a panel. The *panelInstance* parameter may be set to NULL, since it is not used. The *data* parameter should be set to a structure of type *xrPanelInfo* (shown below). The panel information structure is shown below. The structure must be filled out by your program before sending a MSG_NEW to the panel manager.

MSG_NEW returns a pointer to a panel, failure is indicated by a return value of NULL. (MSG_NEW is described under "Using Panels" below.)

```
typedef struct
{
   POINT            panelOrigin;
   SIZE             panelSize;
   Window           relativeTo;
   Window           childOf;
   xrPanelContext   * panelContext;

   xrPanelField     * fieldList;
   INT32            numFields;
   INT32            panelId;
} xrPanelInfo;
```

panelOrigin	*panelOrigin* is set by the programmer to define the origin of a panels window.
panelSize	*panelSize* is a size structure that determines the size of the window the panel resides in. Setting the height and width of the panelSize structure to 0 causes the panel manager to determine the windows size.
relativeTo	*relativeTo* is a window id. The programmer should set this value if the panel is to be created relative to a particular window, otherwise the value should be set to NULL. If it is set to NULL, the panel will be created relative to the root window.
childOf	*childOf* will create the panels window as a child of the specified window. If NULL is given, the panels window will be a child of the root window. If a childOf window is specified, it will normally be most convenient to specify that same window as the relativeTo window.
panelContext	*panelContext* is the structure described earlier that determines the physical and semantic aspects of a panel. If this member is NULL, default values will be used. There is a message (MSG_GETPANELCONTEXT) to fill out one of these structures if you wish to modify a few of the default values. The xrPanelContext structure is described below.
fieldList	*fieldList* is a pointer to an array of editor information structures. The editors in this array will make up the panel seen by the user. The *xrPanelField* structure is shown below:

```
typedef struct
{
   xrEditor        * (* editorFunct)();
   INT8            * editorData;
   xrEditor        * editorInstance;
} xrPanelField;
```

(The use of this structure and its members are described in under "Using Panels," below.)

numFields	*numFields* is the number of fields specified in the fieldList member. This number must be greater than 0.
panelId	*panelId* is an integer set by the programmer when creating a panel or adding a subpanel. This value is returned by the panel manager when a panel has been edited to help identify the panel.

The *xrPanelContext* structure mentioned above determines the physical and semantic aspects of a given panel.

```
typedef struct
{
   Pixmap         winBackground;
   Pixmap         winForeground;
   INT16          borderWidth;
   INT32          foregroundColor;
   INT32          backgroundColor;
   XFontStruct  * fontInfo;
   Cursor         cursor;

   INT32          showFlag;
   xrEditor     * currentEditor;
   INT32          timer;
   INT32          (* initFunct)();
} xrPanelContext;
```

winBackground	*winBackground* is the Pixmap that will be used for the background of the given panel. This value defaults to the pixmap id returned by the XrLib global variable *WhitePixmap*, or the value set in the .Xdefaults file for the foreground pixmap.
winForeground	*winForeground* is the Pixmap used to draw the panel's window border. This value defaults to the pixmap id returned by the Xrlib global variable *BlackPixmap*, or the value set in the .Xdefaults file for the background pixmap.
borderWidth	*borderWidth* is the width of the X window border for the given panel. This value defaults to 1, or as set by the user in the
foregroundColor	*foregroundColor* is the color used to draw in the foreground of the panel. This value defaults to the color returned by the X macro *WhitePixel(_xrCurrentDisplay, _xrCurrentScreen)*, or the value set in the .Xdefaults file for the foreground color. (This value is not used by the panel manager currently, but is used by other dialog managers that make use of the panel context structure.)
backgroundColor	*backgroundColor* is the color used to draw in the background of the panel. This value defaults to the color returned by the X macro *BlackPixel(_xrCurrentDisplay, _xrCurrentScreen)*, or the value set in the .Xdefaults file for the background color. (This value is not used by the panel manager currently, but is used by other dialog managers that make use of the panel context structure.)
fontInfo	*fontInfo* is the font used for drawing text in the panel. It defaults to the value specified in the user's .Xdefaults file. (This value is not used by the panel manager currently, but is used by other dialog managers that make use of the panel context structure.)

cursor	*cursor* is the cursor to be used with the panel. It defaults to xrDefaultCursor.
showFlag	*showFlag* determines whether the panel is shown when it is sent a MSG_NEW. This value defaults to TRUE.
currentEditor	*currentEditor* indicates that a field editor should be activated upon receipt of a MSG_EDIT by the panel manager. This value defaults to NULL indicating no editor is current.
timer	*timer* is a value in seconds. This feature of the panel manager enables you to receive an event with an inputCode of XrPANELTIMER if the number of seconds you set elapses before the user edits the panel in question. (The timer starts after you send the panel a MSG_EDIT). The panel is not removed after the timer elapses, and the value of *timer* is not changed. It is up to your program to determine what should happen when the timer goes off. This value defaults to -1. (Not set)
initFunct	*initFunct* is called each time a panel is displayed. The application supplies this function. This value defaults to NULL.

- MSG_SIZE

MSG_SIZE returns the size of a panel given a *xrpanelInfo* structure. *data* should point to a *xrpanelInfo* structure. The instance parameter is not used and should be set to NULL. The size is returned in the *panelSize* member of the panel information structure.

- MSG_FREE

MSG_FREE destroys a panel and frees the memory allocated by the panel manager. MSG_FREE also reclaims any space used by subpanels associated with the panel being freed. *data* should be set to NULL in most cases, but if you wish to retain the window and editors set it to 1. This destroys the panel manager's knowledge of the panel, but allow you to continue using it. This is useful if you want to control a window you have set up to be a panel without wasting any space in the panel manager. A main application window could be built in this manner.

Care should be taken not to refer to a panel after it has been freed.

- MSG_EDIT

MSG_EDIT causes the panel manager to begin looking for input to a particular panel. MSG_EDIT should be sent to the panel manager when an application expects input from a panel. *data* should point to an *XEvent or xrEvent* structure. If necessary, the event returned may be coerced into an xrEvent structure and will have the following values when MSG_EDIT succeeds.

```
type       = XrXRAY
serial     = 0
send_event = 0
display    = current display pointer
source     = Window id
inputType  = XrPANEL
inputCode  = XrPANELEDITOR (if editor),
             XrPANELTIMER (if timer went off)
             XrPANELINPUT (on other events)
value2     = panelId (Set by you when creating the panel
             or adding a subpanel.)
value3     = editor index
valuePtr   = editor group instance
```

If the inputCode is XrPANELEDITOR, the next XrEvent() read from the input queue will give the output of that editor. If the inputCode is XrPANELINPUT, the next XrInput() read from the input queue will give some other type of input (i.e. various window events, keyboard events...). The important field to key from when looking at a panel event is the inputCode field. This field tells you what happened in the panel. Unless you push the event on the event queue, the type and inputType fields will always be the same for panels. (The use of MSG_EDIT is described under "Using Panels" below.)

- MSG_CURRENTEDITOR

  MSG_CURRENTEDITOR instructs the panel manager to invoke a specific editor when it receives MSG_EDIT. This allows a panel to have an active field (such as a TextEdit field) without an input from the user. It is often used to restart a TextEdit field when it exited because of a break condition. *data* is a pointer to the editor instance to be activated when MSG_EDIT is received. This editor remains current until the value is changed. A NULL in the data parameter indicates no editor is current.

- MSG_MOVE

  MSG_MOVE allows a panel to be moved about the display. *data* is a pointer to a point structure. The panel is moved relative to the window specified in the *relativeTo* member of the panel structure.

- MSG_RESIZE

  MSG_RESIZE allows a panels window to be resized. *data* is a pointer to a rectangle structure. The panel is moved to the x,y location of the rectangle, and the panels window size is changed to the height and width values of the rectangle.

- MSG_SHOW

  MSG_SHOW shows a panel and allows editing of that panel. A panel MUST be showing in order to honor a MSG_EDIT. *data* is not used and may be set to NULL. A panel will be shown automatically upon creation if the *showFlag* field of the panelInfo structure is set to TRUE.

- MSG_HIDE

  MSG_HIDE hides a panel. *data* is not used and may be set to NULL.

- MSG_REDRAW

  MSG_REDRAW causes the editors of a panel to be redrawn. *data* is not used and may be set to NULL.

- MSG_GETPANELCONTEXT

  MSG_GETPANELCONTEXT fills out an xrPanelContext structure with the panel defaults. This structure may then be modified by the programmer before a panel is created. *data* is a pointer to an xrPanelContext structure.

- MSG_NEWSUBPANEL

  MSG_NEWSUBPANEL adds a group of editors to a previously created panel. *panelInstance* should be the pointer to the parent panel, and *data* is a pointer to an *xrPanelInfo* structure. The *editorInstance, numFields, and panelId,* members of the xrPanelInfo structure are the only ones that need be filled out to generate the subpanel.

  The SUBPANEL messages are an interface to the editor group facilities provided by the Xrlib intrinsics. It is suggested you read the section titled *Editor Groups* if you intend to use subpanels. A full discussion of subpanels is given in the section *advanced use of panels.*

- MSG_FREESUBPANEL

MSG_FREESUBPANEL frees all resources allocated by the panel manager connected with the subpanel in question. *data* is a subpanel instance pointer.

- MSG_SHOWSUBPANEL

  MSG_SHOWSUBPANEL causes the editors of a subpanel to be shown by the panel manager. *data* is the subpanel instance pointer.

- MSG_HIDESUBPANEL

  MSG_HIDESUBPANEL deactivates the editor group in question. *data* is the subpanel instance pointer.

- MSG_SETOVERRIDEMODE

  MSG_SETOVERRIDEMODE provides the application with a mechanism to allow window manager interaction in the placing and sizing of panels. By default, the application places, sizes and maps panels directly without any interaction with the window manager.

  For instance, transitory pop-up panels normally appear at specific locations set in the program. However, in an application with a top-level panel that remains on screen constantly, it makes sense to allow the user to set the location of the window via the window manager.

  The Xrlib library maintains an internal 'override' flag, which indicates if any new panel created should bypass the window manager (TRUE) or if the window manager should be given a chance to position the panel (FALSE). The default setting of this flag is TRUE.

  To modify the 'override' flag for all panels, the MSG_SETOVERRIDE message should be issued with the instance parameter set to NULL and the data parameter set to TRUE or FALSE. This will affect all subsequent panel creations.

  To modify the 'override' flag for a single existing panel, issue the MSG_SETOVERRIDE message with the instance parameter set to the target panel's instance pointer and the data parameter set to TRUE or FALSE.

Note

Most window managers only intercept the first mapping request made for any given window. Since a panel is registered to a single window, if that window exists and has been previously mapped or is currently mapped, changing this flag will have no effect if the window is remapped.

4.3.3 Using Panels

Working with panels may be simple or complex depending on the sophistication of the panel needed. The first portion of this section will concentrate on relatively simple, yet powerful means of using panels. More complex interactions will be discussed in the next section, "Advanced use of Panels."

The topics covered in this section are:

- Creating a panel.
- Manipulating a Panels Window.
- Getting input from the fields of a panel.
- Removing a panel when finished.

Creating a Panel. Creating a panel involves the following steps:

1. Create info structures for all editors in the panel.
2. Fill out an xrPanelField array with the locations of the structures and the editors associated with them.
3. Fill out an *xrPanelInfo* structure with the information gathered above, and other information such as location and size. If the height or width of panelSize is 0, the panel manager will determine a default size for the panel giving some white space padding on the bottom and right of the panel. The *panelContext* member of the *panelInfo* structure can be set to NULL in most cases, causing all default values in the context to be used.
4. Send a MSG_NEW message to the panel manager with the xrPanelInfo structure created above.

Each of these steps will now be discussed in detail.

Creating a Panel: Step 1. Creating editor information structures is covered in the chapter on Field Editors.

Creating a Panel: Step 2. The next step in creating a panel is to fill out a xrPanelField array structure (shown below):

```
typedef struct
{
   xrEditor         * (* editorFunct)();
   INT8             * editorData;
   xrEditor         * editorInstance;        /*  return value   */
} xrPanelField;
```

For example, to define a panel with ten editors, declare:

```
xrPanelField  myPanelFields[10];
```

If in step 1 you filled out a button editor information structure called: *buttonEditInfo*, then you need to fill out *myPanelFields*[*0*] like so:

```
myPanelFields[0].editorFunct = XrPushButton;
myPanelFields[0].editorData = (INT8 *) &buttonEditInfo;
```

The *editorFunct* field is a pointer to the editor function in question. The *editorData* field is a pointer to the editor information structure filled out in step 1. The *editorInstance* field will be filled out by the panel manager when the panel is created.

The process outlined above needs to be done for all ten fields in the panel.

Creating a Panel: Steps 3 and 4. All the information you will typically need for a panel has been assembled. The task now at hand is to fill out a panel information structure with this data. You will normally declare a xrPanelInfo and a panel structure at the beginning of your program.

```
xrPanelInfo   myPanelInfo;
xrPanel      * myPanel;
```

For example, *myPanelInfo* might look something like this:

```
myPanelInfo.fieldList = myPanelFields;
myPanelInfo.numFields = 10;
myPanelInfo.relativeTo = myWindow;
myPanelInfo.childOf = NULL;
XrCopyPt(&myPanelOrigin, &myPanelInfo.panelOrigin);
myPanelInfo.panelSize.height = 0;  /* use default size */

/* Use default panelContext  */
myPanelInfo.panelContext = NULL;

/* Create the panel    */
myPanel = XrPanel(NULL, MSG_NEW, &myPanelInfo)
```

This code creates *myPanel.* In the remaining sections, myPanel will be manipulated.

Manipulating a Panels Window. The panel manager provides several commands for dealing with a panels window.

The MOVE message takes a point and relocates the panel relative to the origin of the *relativeTo* window in the panel structure. If you set the *relativeTo* member of the panel structure to NULL, the panel will be relocated relative to the origin of the screen.

The MSG_RESIZE message moves the panel to an absolute position on the display, and changes the size of the panels window.

The MSG_SHOW and MSG_HIDE messages simply display and remove the panel from the screen respectively. A panel will not receive input if it is hidden, hence if you send a MSG_EDIT to a hidden panel, XrPanel() returns NULL.

The MSG_REDRAW will cause all components to redraw themselves. So far, we have discussed editors as components of a panel.

The following list gives examples of using these messages:

- MSG_MOVE

```
XrPanel(myPanel, MSG_MOVE, &newPoint);
```

- MSG_RESIZE

```
XrPanel(myPanel, MSG_RESIZE, &newRectangle);
```

- MSG_SHOW

```
XrPanel(myPanel, MSG_SHOW, NULL);
```

- MSG_HIDE

```
          XrPanel(myPanel, MSG_HIDE, NULL);
```

- MSG_REDRAW

```
          XrPanel(myPanel, MSG_REDRAW, NULL);
```

Getting Input From a Panel. The goal of creating a panel is to get input from the user. Receiving this input is achieved through the use of MSG_EDIT and the *XrInput*() function described in the Xrlib intrinsics chapter. When calling the panel manager with a MSG_EDIT, an XEvent or xrEvent structure is passed in to be filled out. The event returned will always be a XrXRAY type of event which should be coerced into an xrEvent type, as needed. The xrEvent structure will be filled out according to the following rules:

```
type       = XrXRAY
serial     = 0
send_event = 0
display    = current Display pointer
source     = Window id
inputType  = XrPANEL
inputCode  = XrPANELEDITOR (if editor),
             XrPANELTIMER (if timer went off),
             XrPANELINPUT (on some other event).
value2     = panelId (Set by you when creating the panel
                    or adding a subpanel.)
value3     = editor index
valuePtr   = editor group instance
```

If the inputCode is XrPANELEDITOR, the next event read from the input queue will give the output of that editor. If the inputCode is XrPANELINPUT, the next event read from the input queue will give some other type of input (such as window events and keyboard events). The important field to key from when looking at a panel event is the inputCode field. This field tells you what happened in the panel. Unless you push the event, the type and inputType fields will always be the same for panels.

The following code fragment shows a common way to obtain input from a panel.

```
xrEvent panelEvent;
xrEvent myInput;

myPanel = XrPanel(NULL, MSG_NEW, &myPanelInfo);

/* showFlag defaults to true upon creation, so we don't need MSG_SHOW */

if (XrPanel(myPanel, MSG_EDIT, &panelEvent) != NULL)

     /* Read the editor event.           */
     XrInput(NULL, MSG_NONBLKREAD, &myInput);

else
   /* panel call failed, check xrErrno and proceed */
/* panelEvent.value3 is set to a field index, this indicates */
/* which field in the panel was edited.                      */

processPanel = 1;
while (processPanel) {

switch(panelEvent.inputCode)
{
 case XrPANELEDITOR:

   switch (panelEvent.value3) {

   case 0:
      /* Get data from myInput for field 0.
         process that data.              */
      break;

   case 1:
      /*  Get data from myInput for field 1.
          process that data.             */
      break;
        .
        .
   case 9:
      /* End condition.
         Send MSG_FREE to panel if you are done with it.  */
      processPanel = 0;
      break;
}
  case XrPANELINPUT:
        .
        .
   }

   If (XrPanel(myPanel, MSG_EDIT, &panelEvent) != NULL)
      XrInput(NULL, MSG_NONBLKREAD, &myInput);
   else
      /* MSG_EDIT failed check xrErrno and proceed    */

   }
}
```

The MSG_EDIT does not have to come immediately after a MSG_NEW. You may display a panel and wait until the user accesses its window to send it a MSG_EDIT. This feature enables you to display multiple panels and keep them active for the duration of an application.

One condition that requires an example is a break condition associated with XrTextEdit. It's not obvious how to do this. The following code fragment is an example of how to use Xrlib to test for a break condition.

```
xrEvent editorEvent,
        event;

XrPanel (panelInstance, MSG_EDIT, event);
if (event.inputCode == XrPANELEDITOR)
{
   if ((editorEvent.inputType == XrEDITOR) &&    /* Are we in a text edit  */
       (editorEvent.inputCode == XrTEXTEDIT))    /* field?
       {
          if (editorEvent.value1 == XrTEDIT_BREAK)  /* Check for Break */
```

```
            {
               XrPanel (panelInstance, MSG_CURRENTEDITOR, NULL);
            }
            else
            {
               XrPanel (panelInstance, MSG_CURRENTEDITOR, editorEvent.valuePtr);
            } /*
                * Else force text editor to be recalled when a break occurs.
         .      * This is called for status event breaks, such as
         .      *   moving the cursor to an empty field or typing the first
         .      *   character in the field.
         }      */
    }
```

Removing a Panel. Removing a panel is a simple matter of sending this message to the panel manager:

```
    XrPanel(myPanel, MSG_FREE, NULL);
```

This action frees all memory associated with a panel that was allocated by the panel manager.

4.3.4 Advanced Use of Panels

The previous section concentrated on basic use of panels. This section discusses advanced topics, such as dealing with panel editors directly, and working with subpanels.

Communicating Directly With the Fields of a Panel. There are times when you will need to communicate directly with the editors of a panel as the result of some action by a user. For example, the input of one editor is the output of another, or a change in an editor value cause other editors to change status.

Direct communication with editors is through the *xrPanelField* structure you filled out to create the panel. This structure is shown again below:

```
    typedef struct
    {
       xrEditor         * (* editorFunct)();
       INT8             * editorInfo
       xrEditor         * editorInstance;
    } xrPanelField;
```

In previous examples, when you created the panel, you filled out the editorFunct and editorInfo members for each editor in the panel, but did not fill out the editorInstance member. This member is filled out by the panel manager when the panel is created. This pointer gives you direct access to each editor in the panel. The following code fragment exemplifies the procedure:

```
    /*  Editor 1, (a check box) changed status    */
    /*  As a result, editor 6 (a TextEdit field)  */
    /*  needs to be made sensitive.               */

    XrTextEdit(myPnlFields[6].edtrInstance, MSG_SETSTATE, XrSENSITIVE);
```

Only inquiry and state setting messages should be sent directly to an editor in a panel. Unexpected results will occur if you delete field editors from a panel directly, or send XrTextEdit editors MSG_EDIT directly.

Subpanels. Many times a group of panels are very similar, with perhaps only one or two sets of editors changing between panels. It might be more convenient in this situation for an application to deal with just one panel and change the set of editors according to the state of the panel or application. This task is accomplished with *subpanels*. The SUBPANEL messages are an interface to the editor group facilities provided by the Xrlib intrinsics. (For more information about using subpanels, refer to "Editor Groups" in chapter 3.)

Subpanels have many of the same characteristics as a panel, but have a special set of messages. There are a few special rules governing subpanels.

- A panel may contain an unlimited number of subpanels, any or all of which may be active, but caution should be taken not to overlap the editors of panels and subpanels.
- subpanels share the window of their parent. Changes made to the parent panels window will also affect the subpanel.
- Messages are not sent to subpanels, they are sent directly to the parent of the subpanel.

As mentioned earlier, panels have a special set of messages that pertain specifically to subpanels:

- MSG_NEWSUBPANEL
- MSG_FREESUBPANEL
- MSG_SHOWSUBPANEL
- MSG_HIDESUBPANEL

These messages allow you to add a subpanel to a panel, remove a subpanel, make a subpanel active, and make a subpanel inactive respectively. Activating a subpanel adds its editors to the window of the parent panel. Deactivating a subpanel removes its editors and clears the rectangle the subpanel covered.

Using Subpanels. If you have already created a panel, adding a subpanel is a simple matter of filling out a new panelInfo structure with the contents of the subpanel and sending this information with a MSG_NEWSUBPANEL to the panel. The following code fragment shows the process.

```
XrPanelInfo       myPanelInfo     /* parent panel details    */
XrPanel         * myPanel         /* parent panel handle     */
XrPanelInfo       subPanel1Info   /* 1st subpanel details   */
XrPanel         * subPanel1       /* 1st panel handle        */
XrPanelInfo       subPanel2Info   /* 2nd subpanel details   */
XrPanel           subPanel2       /* 2nd subpanel handle    */
          .
          .
          .
/*  Fill out the structures as shown in the          */
/*  section on using panels.  After creating the     */
/*  parent panel, the subpanels may be added as shown.  */

subPanel1Info.panelId = 1;
subPanel2Info.panelId = 2;
subPanel1 = XrPanel(myPanel, MSG_NEWSUBPANEL, &subPanel1Info);
subPanel2 = XrPanel(myPanel, MSG_NEWSUBPANEL, &subPanel2Info);

XrPanel(myPanel, MSG_SHOWSUBPANEL, subPanel1);

/* This message activates subpanel #1.  Activation     */
/* causes the subpanels editors to be added to myPanels */
/* window.                                              */
```

Obtaining input from a panel containing subpanels is virtually the same process described under "Using Panels" above. The major difference is in using either the panelId that you set or the editorGroup instance pointer to determine which subpanel the input came from.

Panel Structures. The panel manager keeps all the information it needs for its operations on a particular panel in a panel structure. The main panel structure is described below. Other structures are used with messages to communicate with the panel manager. They are described with their respective messages.

```
typedef struct _xrPanel {
        RECTANGLE               panelRect;
        RECTANGLE               panelZeroRect;
        Window                  relativeTo;
        Window                  childOf;
        xrPanelContext        * panelContext;

        xrEditor              * editorInstance;
        INT32                   numFields;
        INT32                   panelId;

        Window                  panelWindow;
        xrEditorGroup         * editorGroup;
        struct _xrPanel      ** subPanel;
        INT32                   numSubPanels;
} xrPanel;
```

panelRect	*panelRect* contains the location and size of the panel.
panelZeroRect	*panelZeroRect* contains the location and size of the panel in a zero-based rectangle.
relativeTo	*relativeTo* is a window the panel should be created and moved relative to.
childOf	*childOf* is the parent window of the panel.
panelContext	*panelContext* is a structure (described above) that provides the physical and semantic parameters to be used with a particular panel.
editorInstance	*editorInstance* is a pointer to an array of the editor instances that make up the panel.
numFields	*numFields* is the number of instances contained in the preceding array.
panelId	*panelId* is set by your program when creating the panel. It helps identify the panel when receiving input from a panel that contains subpanels.
panelWindow	*panelWindow* is set to the panels X window id when the panel is showing on the display.
editorGroup	*editorGroup* is the editor group pointer for this panel. This is used when subpanels are added to a panel.
subPanel	*subPanel* is an array of pointers to subpanels. The use of subpanels is described under "Advanced Use of Panels" above.
numSubPanels	*numSubPanels* is the number of subPanels associated with this panel.

A Panel Example. For an example of the usage of panels in an application, please see the program RB/filePanel.c provided in this distribution as part of the Raster Builder field editor source code.

5 Building a Field Editor

The following sections contain the guidelines a programmer should follow when developing a new field editor. Some of the details describing how an editor works are also included.

Under normal circumstances, the underlying structure of field editors is fairly constant, regardless of what specific actions the editor may perform. Knowing this, it can be an easy task to take an existing editor, change the editor specific portion of it, and thus easily create a new field editor. Adhering to the guidelines presented in this chapter will help to make this possible in the future also.

5.1 Naming Conventions

Each editor must supply a single entry point, through which all action and status requests are received. A unique name needs to be assigned to this entry point; preferably one which describes the function of the editor. By convention, the name must be at least 2 words, which will be compressed together, with the first letter of each word capitalized, and the prefix "Xr" added to the front. For example:

- The raster editor is called **XrRasterEdit()**
- The raster select editor is called **XrRasterSelect()**
- The titlebar editor is called **XrTitleBar()**

Throughout this document, the notation "XrEditorName" will be used; it refers to the name assigned to this primary editor entry point.

Besides a convention regarding the naming of the editor entry point, there is also a convention regarding how the editor entry point is defined. All editors *must* adhere to this convention, otherwise they will not work correctly! The proper syntax for defining an editor's entry point is:

xrEditor * XrEditorName (editorName, message, data)

where *editorName* is a pointer to an editor structure; the name of this structure should be the same as the editor's name, except that the "Xr" prefix is dropped, and the first letter is in the lower case.

message is an integer value, which specifies the action to be performed by the editor.

data is a pointer to something; the editor can coerce this pointer to point to whatever it likes, depending upon the requested action.

Some examples of editor entry point declarations are:

```
xrEditor * XrRasterEdit (rasterEdit, message, data)

xrEditor * XrScrollBar (scrollBar, message, data)

      xrEditor * rasterEdit, *scrollBar;
      INT32      message;
      INT8     * data;
```

5.1.1 Data Structure Naming

Almost every editor has a need to define several data structures; some are strictly internal, and thus used only by the editor, while others define structures which are used by application programs to pass data to the editor. There are two structures in particular, for which a naming convention has been defined; one of these also has a portion of its structural organization defined.

The first structure of interest is the structure by which an editor is first initialized. This structure will normally contain all of the information needed by the editor, to create a single instance of itself. To indicate that this structure contains initialization data for a particular editor, the name of the structure will be constructed by taking the editor name [XrEditorName], changing the first character to lower case, and then appending the word "Info" to the end of it. For example, the initialization structure used by the **XrScrollBar()** editor is called "xrScrollBarInfo". The actual organization of this structure is partially fixed, to make it easier for code to access information in these structures, without forcing them to know the complete layout of the structure. For instance, a non-editor related code module may want to know what window the requested editor instance is to be tied to, without knowing what the actual type of editor it is. For this reason, the first five fields of an "Info" structure must be organized in the following fashion, and must use the field names shown:

```
typedef struct {
     Window    editorWindowId;
     RECTANGLE editorRect;
     INT8      editorState;
     INT32     editorFGColor;
     INT32     editorBGColor;
       .
       .
} xrEditorNameInfo;
```

After the first five fields, the editor writer is free to add any fields he desires, which might be required to initialize the editor.

The second structure of interest is the internal structure used by the editor to keep track of data and state information for a given editor instance. Although the information in this structure is directly available to an application program, they should be discouraged from accessing it directly; instead, they should use the mechanism provided by the editor, to access this data. The convention for naming this structure involves taking the name of the editor [XrEditorName], changing the first character to lower case, and then appending the word "Data" to the end of it. The actual organization of this structure is left entirely up to the editor writer, since the editor should be the only one interested in accessing this data directly. A sample declaration would be:

```
typedef struct {
       .
       .
       .
} xrEditorNameData;
```

Any other data structures needed by an editor should, if possible, follow the naming convention of appending a descriptive word to the end of the editor name.

5.1.2 Define Naming

Since the X11R library toolbox facilities will be used by a large selection of programs, it is a good idea to make all possible attempts to avoid naming collisions between toolbox defines and the defines declared by the program using toolbox. To aid in meeting this goal, all defines declared by an Xrlib components, whether it be an intrinsic or a field editor, must have the following format:

```
#define XrDEFINENAME    <value>
```

Following this guideline should help avoid naming conflicts.

5.1.3 Variable Naming

For the most part, editor writers are free to use their own judgement when naming local variables within their editor modules. However, as all good software developers know, using a name which in one fashion or another describes how the variable is used, greatly increases the readability of a program. There will be several local variable naming conventions, which will be used throughout the field editors. These include the following rules:

- When a pointer to an editor's "Info" structure [xrEditorNameInfo] is used, its name should be constructed by taking the lowercase initials of the editor name, and appending the word "InfoPtr" to it. For example, when the **XrScrollBar()** editor needs a pointer to an "Info" structure, it will define a local variable as follows:

```
xrScrollBarInfo *sbInfoPtr
```

- When a pointer to an editor's "Data" structure [xrEditorNameData] is used, its name should be constructed by taking the lowercase initials of the editor's name, and appending the work "DataPtr" to it. For example, when the **XrScrollBar()** editor needs a pointer to its data structure, it will define a local variable as follows:

```
xrScrollBarData *sbDataPtr
```

5.2 Coding Conventions

When defining a structure needed by an editor, the convention is to "typedef" the structure, thus making it easier to create an instance of that structure. As an example, the "Info" structure needed by an editor will have the following declaration:

```
typedef struct {
    Window      editorWindowId;
    RECTANGLE   editorRect;
    INT8        editorState;
    INT32       editorFGColor;
    INT32       editorBGColor;
     .
     .
} xrEditorNameInfo;
```

Then, to create an instance of this structure, the following line of code will suffice:

```
xrEditorNameInfo editorNameInfo;
```

5.3 Memory Management

The X11R library toolbox intrinsics provide an application with the means for supplying its own memory management facilities. If an application does not choose to supply these routines, then the standard ones (malloc(), realloc(), calloc() and free()) will be used.

If an application does choose to supply its own memory management facilities, then it must provide four new entry points, which perform the identical functions as malloc(), realloc(), calloc() and free().

The redefining of these functions is allowed only once; this is at the time the application initializes the Xrlib toolbox code, by means of the **XrInit()** intrinsic.

What all of this means to a field editor is that they ***must never*** invoke malloc(), realloc(), calloc() or free() directly! Instead, they ***must*** use the four global function pointers supplied by the Xrlib intrinsics. By following this rule, the field editor should continue to function correctly, regardless of which memory management scheme is being used.

The four global function pointers are outlined below:

```
char * (*xrMalloc)();
int    (*xrFree)();
char * (*xrRealloc)();
char * (*xrCalloc)();
```

As an example, to use the **xrMalloc()** function pointer, the following code will work:

```
{
   char * buffer;

   buffer = (*xrMalloc)(1024);
}
```

5.4 Drawing and Buffer Flushing

Field editors should refrain from issuing XFlush() requests, if possible. This call can slow down the performance of an application, if issued too often. We have taken the strategy that when an editor draws an instance, it will not automatically flush the output buffer afterwards. Rather, it will be up to the application to issue the XFlush() request, when it wants; this allows the application to create several editors, before flushing the output buffer. As a side, when an application calls the **XrInput()** routine, to obtain an input event, the output buffer is automatically flushed.

5.5 Field Editor Messages

As was mentioned in an earlier section of this document, each editor provides only a single entry point, by which all work requests must be received. Since an editor must be capable of processing a relatively large number of requests, a parameter is provided by each editor's entry point, which allows a program to specify the action it would like performed.

There are several messages which are required to be understood by all editors. All editors *must* accept these messages and respond in the described fashion.

The sub-sections that follow give a detailed description of each of these required messages. As a quick reference, the required messages are:

- MSG_NEW
- MSG_SIZE
- MSG_MOVE
- MSG_GETSTATE
- MSG_SETSTATE
- MSG_REDRAW
- MSG_EDIT
- MSG_FREE

MSG_SIZE can be called at any time, even before an editor instance has been created. However, before any of the other messages can be invoked, you must first create the editor instance, using the MSG_NEW message. This is because the MSG_NEW message returns the pointer to the editor structure, which is required by all of the other messages, except for MSG_SIZE. All editors must enforce this rule, by verifying that the pointer to an editor structure, which is passed in as the first parameter, is not NULL.

The discussion which follows will also include one non-required message: MSG_RESIZE.

The entry point into the editor is defined as follows:

```
xrEditor * XrEditorName (editorName, message, data)

    xrEditor * editorName;
    INT32     message;
    INT8     * data;
```

Each of the messages is responsible for converting the *data* parameter into a pointer to the desired structure. Unless noted, all messages return a non-NULL pointer, if the message was successfully handled. If the message failed, then a NULL pointer is returned.

5.5.1 MSG_NEW

The MSG_NEW message is the mechanism by which an application program can request an instance of an editor be created. Internally, the steps involved in creating an instance of an editor are as follows:

- Allocate a block of memory to hold the data required by the editor to process the instance; this memory will stay around for the duration of the instance's lifetime.
- Determine the size of the rectangle needed to contain the instance, and fill the data structure with the information.

- Attach the editor instance to the specified editor window; this is the call which creates the editor structure, whose pointer is returned to the application program when a successful status is returned.
- Draw the editor instance in the specified window.

Before this message is issued, the application must first fill out an "Info" structure, which will be used by the editor to construct the new instance. Each editor is free to determine what data it requires with this call; however, the first five fields of the "Info" structure are fixed for all editors, and are laid out as follows:

```
typedef struct {
     Window      editorWindowId;
     RECTANGLE   editorRect;
     INT8        editorState;
     INT32       editorFGColor;
     INT32       editorBGColor;
            .
            .
} xrEditorNameInfo;
```

editorWindowId:	This is the window which is used for displaying the editor instance.
editorRect:	This defines the rectangle into which the editor instance will be drawn. Any time an input event occurs within this region, the editor associated with this rectangle will be notified.
editorState:	Allows an application program to define the initial state of an editor instance. At the present time, only two state characteristics are defined. They are XrVISIBLE and XrSENSITIVE. XrVISIBLE signals to the editor that the instance should be drawn within the window; when cleared, the instance will not be drawn. XrSENSITIVE, when set along with XrVISIBLE, indicates that the editor should be notified anytime an input event occurs within the instance's rectangle; if either XrSENSITIVE or XrVISIBLE is cleared, the editor will not be notified of input events.
editorFGColor	Allows an application program to specify the foreground color to be used when drawing the editor instance. If the application has set this value to -1, the default foreground color should be used. This can be referenced using the Xrlib global xrForegroundColor.
editorBGColor	Allows an application program to specify the background color to be used when drawing the editor instance. If the application has set this value to -1, the default background color should be used. This can be referenced using the Xrlib global xrBackgroundColor.

Upon successful completion of this message, a pointer to the newly created editor structure associated with this instance will be returned. This value should be saved by the calling program, since this pointer is a required parameter for all future calls to the editor, when in regard to this instance.

A sample of the proper calling sequence is outlined below:

```
xrEditor *enEditorPtr;
xrEditorNameInfo enInfo;

enEditorPtr = XrEditorName (NULL, MSG_NEW, &enInfo);
```

5.5.2 MSG_FREE

When an editor instance is no longer needed by an application program, the MSG_FREE message is used to notify the editor. The "data" parameter is not used by this message, so the calling program should set this to NULL. The important parameter is the pointer to the editor structure [editorName], which was returned when the instance was first created. This structure is used by the editor to determine what instance is no longer needed. Internally, the following steps are involved:

- If the instance is visible, then the area occupied by the instance should be filled with the window's background tile, thus making the instance invisible.
- The block of memory which was allocated at create time, and used to hold the data describing this instance, is freed up.
- The editor structure is removed from the list of editors attached to the specified window, and the editor structure is destroyed.

Note

After issuing a MSG_FREE message, the editor pointer is no longer valid, and should not be used; it is up to the application program, not the editor however, to enforce this.

5.5.3 MSG_SIZE

All editors must allow this message to be issued at anytime, even if an instance has not been created. This message is used primarily by programs to determine the size of the rectangular region needed to enclose the specified editor instance. The program will normally fill out a copy of the editor's "Info" structure with information describing a hypothetical editor instance. The editor will use this information to calculate the size of the rectangle, and will return the rectangle size (in 0 based format) in the "editorRect" portion of the "Info" structure. This allows an application program to then offset the rectangle to whatever location is desired, and to then issues a MSG_NEW message to create the real editor instance.

When this message is issued, an editor should not allocate any permanent data structures, nor should it return a pointer to an editor structure. It's only job is to return the size of the rectangle!

The amount and type of information required by an editor to determine the size of the rectangle is very editor specific. This information will be covered in later sections, when each editor is discussed.

For this request, the first parameter (the instance pointer) is not used, and should be set to NULL.

5.5.4 MSG_REDRAW

Most editors will allow an application to modify the current value of the editor. When an application does modify the value of an editor instance, the editor will need to be notified, so that the instance can be redrawn to match the new value. Just what is performed when a redraw request is received, is very editor specific.

All editors will require an additional piece of information, describing what type of redraw to perform; this will be interpreted as a 32 bit integer value. Again, this value is editor specific. It may include such options as redrawing the whole instance, or just redrawing a particular portion of an instance.

At the present time, the following redraw modes are defined:

XrREDRAW_ALL	This will tell the editor to completely redraw the specified editor instance. The action performed should be very similar to what was done

	when the editor was first displayed.
XrREDRAW_ACTIVE	This will tell the editor that a new active selection has been specified for the instance. The editor must then redraw only those portions which are necessary, to reflect this.
XrREDRAW_SLIDE	For those editors which have some form of a slide mechanism, this message will tell the editor that a new slide position has been specified. The editor must then redraw only those portions which are necessary, to show the new slide position.

A sample of the proper calling sequence is outlined below:

```
xrEditor   *enEditorPtr;

XrEditorName (enEditorPtr, MSG_REDRAW, XrREDRAW_ALL);
```

5.5.5 MSG_GETSTATE

Each editor maintains a copy of the current state for each of its instances; this refers to the setting of the XrVISIBLE and XrSENSITIVE characteristics. Upon request, this information will be returned to an application program. A pointer to an 8 bit integer should be passed as a parameter when making this call; the current state flags, which are saved in a field within the editor structure, will be returned in the pointed to 8 bit value. The editor structure pointer, passed in as the first parameter, is used to determine which instance to return state information about.

A sample of the proper calling sequence is outlined below:

```
INT8      state_flags;
xrEditor * enEditorPtr;

XrEditorName (enEditorPtr, MSG_GETSTATE, &state_flags);
```

5.5.6 MSG_SETSTATE

This message allows an application to modify the state flags (XrSENSITIVE and XrVISIBLE) for an editor instance. The first parameter, the pointer to the editor structure, will indicate which instance to modify. The third parameter should contain the new state flags for the instance. Under normal circumstances, when an editor receives one of these messages, it will save the new state flags in the instance's editor structure, and will then redraw the editor using the new state information.

A sample of the proper calling sequence is outlined below:

```
INT8      state_flags = (XrSENSITIVE | XrVISIBLE);
xrEditor * enEditorPtr;

XrEditorName (enEditorPtr, MSG_SETSTATE, state_flags);
```

5.5.7 MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the editor rectangle associated with the instance is not changed. To relocate an editor instance, a new origin point for the editor instance's rectangle must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window; i.e. relative to the window's origin.

When this message is issued, the "instance" parameter must point to the editor structure associated with the instance which is to be moved, while the "data" parameter must point to a "POINT" structure containing the new editor rectangle origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible. At the same time, the editor will automatically force the editor group's rectangle to be recalculated. If an application plans to relocate several editor instances all at once, its best bet would be to first make all of them invisible, then relocate all of them, and then make them all visible again.

5.5.8 MSG_RESIZE

This message provides an application with a means for both changing the size of the editor rectangle associated with a particular editor instance, and changing the location of the new editor rectangle. All restrictions regarding the editor rectangle size, which applied when the instance was first created using MSG_NEW, still apply. If an invalid editor rectangle is specified, then the resize request will fail.

When this message is issued, the "instance" parameter must point to the editor structure associated with the instance which is to be resized, while the "data" parameter must point to a "rectangle" structure containing the new size and origin for the editor rectangle.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible. At the same time, the editor will automatically force the editor group's rectangle to be recalculated.

5.5.9 MSG_EDIT

This message is a signal to the editor that an input event occurred within one of its editor instances. The particular instance is specified by the passed in editor structure pointer. A pointer to the input event is also passed in; the pointer refers to a structure of type "XEvent" or "xrEvent". The editor should verify that the event occurred within one of its editor rectangles, and also that the event is one which is supported by the editor; if it is not, then the editor should return with a "failed" status. Since most field editors are triggered by an XrSELECT event, the editor can use the utility routine **XrMapButton()** to determine if the event maps into an XrSELECT event. If the event is one which the editor knows how to handle, then it should perform whatever action is implied by the event, and should then give some notification to the application program that the editor instance has been modified; this is accomplished by generating another input event and pushing it back onto the front of the application's input queue. The information contained within the returned "xrEvent" structure will include an indication of which editor did the work, the instance which was modified, and any other editor specific information.

If the event caused the editor to modify the value associated with the instance, then it should redraw the instance to reflect the new value.

After processing the select event, a field editor should monitor the input queue and swallow the corresponding select up event; if any other X event is received before the select up, then it should be pushed back onto the input queue, followed by the xrEvent event, and the editor should return.

5.6 Returning Field Editor Status

Each time an action request is received by an editor, it needs to have a mechanism for returning some status information, indicating whether or not the message was successfully handled. This is accomplished by returning a pointer to the appropriate editor structure. If the NULL pointer is returned, then the message request failed; any other value implies that the message completed successfully.

An example would be:

```
xrEditor *
XrEditorName (editorName, message, data)

   xrEditor * editorName;
   INT32      message;
   INT8     * data;

{
      .
      .
   if (failed)
      return ((xrEditor *)NULL);
   else
      return (editorName);
}
```

An example of how a program would use this value would be:

```
{
   xrEditor *enEditorPtr;

   if (XrEditorName (enEditorPtr, MSG_REDRAW, XrREDRAW_ALL) == NULL)
        /* Request failed */
   else
        /* Request succeeded */
}
```

5.7 Returning Editor Information

Many times, after an editor has been modified by some input event, it would like to notify the application program to which it is attached that some change has taken place. Fortunately, a mechanism to accomplish this has been provided. The general rule for returning notification information is through the application's input queue. To do this, take an "xrEvent" structure and fill it in with the information you would like returned to the application program. Once the structure has been filled, you make a call to **XrInput()**, using the MSG_PUSHEVENT message, which will add the input event to the front of the application's input queue. The next time the application issues a read request, your information will be passed to it.

The following fields in the "xrEvent" structure are available for use by an editor:

type	Must be set to "XrXRAY".
serial	Must be set to zero.
send_event	Must be set to zero.
display	Must be set to the Xrlib Global variable _xrCurrentDisplay.
source	Must be set to the window id associated with the field editor.
inputType	When an editor returns an event, this field should be set to "XrEDITOR".
inputCode	This field is used to uniquely identify the editor which generated the event. For example, the scrollbar editor will set this to "XrSCROLLBAR". When assigning your editor one of these values, it must not be in the range 0-0x3FFF, since this range is reserved by Hewlett-Packard.
valuePtr	This field is used by an editor to uniquely identify the editor instance which generated the event. The pointer to the instance's editor structure must be returned here.
valuePt	This field is a POINT structure used by an editor to return the cursor position at which an event occurred.
value1	This is an optional field, which allows an editor to return some sort of indication as to what action the editor last took. For example, when the "up arrow" portion of a scrollbar is selected, the scrollbar editor will set this field to "XrSCROLL_UP".
value2	Sometimes an editor will want to return a value as part of the event information. That is the purpose of this field. If an editor has no value to return, then this field does not need to be set. Each editor is free to use this field in whatever fashion is reasonable, to return the editor's value.

Here is an example of creating an xrEvent structure and filling it in.

```
xrEvent xrevent;

        /* Fill out the xrEvent Structure */

        xrevent.type = XrXRAY;
        xrevent.serial = 0;
        xrevent.send_event = 0;
        xrevent.display = _xrCurrentDisplay;
        .
        .  /* Fill out the remaining values */
        .
```

Some field editors, such as the scrollbar and raster edit editors, have an interactive mode of operation. This mode is entered when a user issues a select within a certain region of the editor, and normally continues until the user releases the select button. When the select up occurs, the editor will push a xrEvent event onto the front of the application's input queue and return. However, there is a second way in which this interactive mode may be terminated. Upon receipt of any X event other than the select up, the field editor should terminate and push two events. The first event pushed should be the X event which caused the operation to terminate, while the second event should be the xrEvent generated by the editor. These must be pushed in this order!

5.8 Field Editor Layout and Template 1

Now that most of the conventions for building an editor have been discussed, lets take a look at how an editor should be organized. Even though a set of editors may perform radically different functions, their underlying structure should be the same. The example given below is very general so it should be very easy to understand. You may use it as a template when developing a new editor.

Since the underlying structure of most field editors is identical, it makes sense to extract as much of the common code as possible into utility routines that can then be shared by all field editors. This reduces code size and makes it easier to fix problems when they occur. The template presented below does not take advantage of the common utility routines because we want to demonstrate what is involved in handling each of the messages understood by the sample field editor. The section following the sample will discuss the common utilities, will present a second template, and show what the sample field editor would look like if the common utilities were used.

```
xrEditor *
XrEditorName (editorName, message, data)

   xrEditor * editorName;
   INT32      message;
   INT8     * data;

{
   /* Determine the action being requested */
   switch (message)
   {
      case MSG_NEW:
      {
          /* Create a new instance of this editor */
          xrEditorNameInfo * enInfoPtr = (xrEditorNameInfo *)data;
          xrEditorNameData * enDataPtr;

          /*
           * Get a pointer to the information describing the editor
           * instance and make sure the pointer is valid.
           */
          if (enInfoPtr == NULL)
          {
             xrErrno = XrINVALIDPTR;
             return ((xrEditor *) NULL);
          }
          else if (enInfoPtr->editorWindowId == 0)
          {
             xrErrno = XrINVALIDID;
             return ((xrEditor *) NULL);
          }

          /* Allocate some data area for this instance */
          if ((enDataPtr = (xrEditorNameData *)
              (*xrMalloc)(sizeof (xrEditorNameData))) == NULL)
          {
             xrErrno = XrOUTOFMEM;
             return ((xrEditor *)NULL);
          }

          /* Call the routine responsible for creating the instance */
          if (createEditorName (enDataPtr, enInfoPtr, MSG_NEW) == FALSE)
```

```
     {
        /* Create failed; xrErrno was set by createEditorName() */
        (*xrFree)(enDataPtr);
        return ((xrEditor *)NULL);
     }

     /* Allocate and initialize the editor structure for the instance */
     if (editorName = (xrEditor *)(*xrMalloc)(sizeof(xrEditor)) == NULL)
     {
        (*xrFree)(enDataPtr->extraData);
        (*xrFree)(enDataPtr);
        xrErrno = XrOUTOFMEM;
        return ((xrEditor *)NULL);
     }
     _XrInitEditorStruct (editorName, enInfoPtr, enDataPtr,
                          XrEditorName);

     /* Attempt to attach the editor to the specified window */
     if (XrEditor (enInfoPtr->editorWindowId, MSG_ADD, editorName) == FALSE)
     {
        /* The attach request failed; xrErrno set by XrEditor() */
        (*xrFree)(enDataPtr->extraData);
        (*xrFree) (enDataPtr);
        (*xrFree) (editorName);
        return ((xrEditor *) NULL);
     }

     /* Lastly, draw the editor instance */
     if (editorName->editorState & XrVISIBLE)
        drawEditorName(editorName, NULL);

     /* Create request was successful */
     return (editorName);
}



case MSG_FREE:
{
     /* Destroy the specified editor instance */
     xrEditorNameData * enDataPtr;

     if (editorName == NULL)
     {
        xrErrno = XrINVALIDID;
        return ((xrEditor *)NULL);
     }

     /*
      * Remove the visible instance from the window,
      * free up the memory used to hold the data describing
      * this instance, and disconnect the instance from
      * the window to which it was attached.
      */
     if (editorName->editorState & XrVISIBLE)
     {
        _XrMakeInvisible (editorName->editorWindowId,
                          &editorName->editorRect, TRUE);
     }
     enDataPtr = (xrEditorNameData *) editorName->editorData;
     (*xrFree)(enDataPtr->extraData);
```

```
        (*xrFree)(enDataPtr);
        XrEditor (editorName->editorWindowId, MSG_REMOVE, editorName);
        (*xrFree)(editorName);
        return (editorName);
    }



    case MSG_GETSTATE:
    {
        /* Return the current state flags for an editor instance */
        if (editorName == NULL)
        {
           xrErrno = XrINVALIDID;
           return ((xrEditor *)NULL);
        }
        else if (data == NULL)
        {
           xrErrno = XrINVALIDPTR;
           return ((xrEditor *)NULL);
        }

        *data = editorName->editorState;
        return (editorName);
    }



    case MSG_SIZE:
    {
        /*
         * Return the size of the rectangle needed to enclose
         * an instance of this editor, using the specifications
         * passed in by the application program.
         */

        xrEditorNameInfo *enInfoPtr;

        enInfoPtr = (xrEditorNameInfo *)data;

        if (enInfoPtr == NULL)
        {
           xrErrno = XrINVALIDPTR;
           return ((xrEditor *) NULL);
        }
        if (sizeEditorName (enInfoPtr, &enInfoPtr->editorRect) == FALSE)
        {
           /* Size request failed; xrErrno set by sizeEditorName() */
           return ((xrEditor *)NULL);
        }

        return (editorName);
    }
```

```
case MSG_SETSTATE:
{
     /* Change the state flags for a particular editor instance */
     INT8 oldState;
     INT8 newState;

     if (editorName == NULL)
     {
        xrErrno = XrINVALIDID;
        return ((xrEditor *)NULL);
     }

     oldState = editorName->editorState;
     newState = (INT8) data;
     editorName->editorState = (INT8) data;

     /* Redraw the editor, using the new state flags */
     if (((oldState & XrVISIBLE) != (newState & XrVISIBLE)) ||
        ((newState & XrVISIBLE) &&
        ((oldState & XrSENSITIVE) != (newState & XrSENSITIVE))))
     {
        drawEditorName (editorName, NULL);
     }
     return (editorName);
}



case MSG_MOVE:
{
   /* Move the origin for the specified editorRect */
   POINT      * ptPtr (POINT *)data;
   RECTANGLE    oldRect;

   if (editorName == NULL)
   {
      xrErrno = XrINVALIDID;
      return ((xrEditor *) NULL);
   }
   else if (ptPtr == NULL)
   {
      xrErrno = XrINVALIDPTR;
      return ((xrEditor *) NULL);
   }

   /* Reset the origin of the editor rectangle */
   XrCopyRect (&editorName->editorRect, &oldRect);
   editorName->editorRect.x = ptPtr->x;
   editorName->editorRect.y = ptPtr->y;

   if (editorName->editorState & XrVISIBLE)
   {
      /* Make the instance invisible, if necessary */
      _XrMakeInvisible (editorName->editorWindowId, &oldRect, TRUE);

      /* Redraw the relocated instance */
      drawEditorName (editorName, NULL);
   }

   /* Force recalculation of editor group rectangle */
   XrEditorGroup (NULL, MSG_ADJUSTGROUPRECT, editorName);
```

```
      return (editorName);
   }



   case MSG_REDRAW:
   {
      /* Redraw the specified editor instance */
      INT32 redrawMode = (INT32)data;

      if (editorName == NULL)
      {
         xrErrno = XrINVALIDID;
         return ((xrEditor *) NULL)
      }
      else if (redrawMode != XrREDRAW_ALL)
      {
         xrErrno = XrINVALIDOPTION;
         return ((xrEditor *) NULL)
      }

      if (editorName->editorState & XrVISIBLE)
         drawEditorName (editorName, NULL);
      return (editorName);
   }



   case MSG_EDIT:
   {
      /*
       * Process the incoming event, and generate a return
       * event, to indicate to the application program how
       * the editor instance was modified.
       */

      XButtonEvent *eventPtr = (XButtonEvent *) data;
      xrEvent       returnEvent;
      XEvent        exitEvent;

      if ((!_XrCatchableKey (editorName, eventPtr) ||
          (XrMapButton (XrSELECT,eventPtr) != XrSELECT))
      {
         /*
          * Don't set xrErrno, since receiving a event we don't
          * know how to handle is not really an error.
          */
         return ((xrEditor *)NULL);
      }

      /* Initially fill out the return event */
      returnEvent.type = XrXRAY;
      returnEvent.serial = 0;
      returnEvent.send_event = 0;
      returnEvent.display = _xrCurrentDisplay;
      returnEvent.source = eventPtr->window;
      returnEvent.inputType = XrEDITOR;
      returnEvent.inputCode = XrEDITORNAME;
      returnEvent.valuePtr =  (INT32) editorName;
      returnEvent.valuePt.x = eventPtr->x;
```

```
            returnEvent.valuePt.y = eventPtr->y;

            /* Process the input event */
            processEditorName (editorName, eventPtr, &returnEvent);

            /* Swallow the select up event */
            while (XrInput (NULL, MSG_BLKREAD, &exitEvent) == FALSE);
            if (XrMapButton (XrSELECTUP, &exitEvent) != TRUE)
            {
               /* Not a select up, so push back onto the input queue */
               XrInput (NULL, MSG_PUSHEVENT, &exitEvent);
            }

            /* Push the editor event onto the input queue */
            XrInput (NULL,MSG_PUSHEVENT, &returnEvent);

            return (editorName);
         }




         default:
            /* All other messages are invalid */
            xrErrno = XrINVALIDMSG;
            return ((xrEditor *)NULL);

      } /* end of switch */
   } /* end of XrEditorName() */
```

5.9 Field Editor Utility Functions

The Xrlib library contains a set of utility functions that provide facilities that are commonly needed by field editors. Among these utilities are functions which display 8 bit text strings, obtain internal font information, and initialize an editor structure.

The field editor writer is not *required* to use the facilities discussed in the following sections. The routines are programming aids provided so that editor writers need not waste their time duplicating functionality which already exists. Since the routines are in the Xrlib library, they will be included only once in a given application. This will prevent the duplication of code and lead to smaller application programs.

5.9.1 Common Field Editor Message Handlers

The underlying structure of all field editors is basically the same. Only the routines for creating, drawing and processing events tend to differ among field editors. So it is logical to extract as much common code as possible, place it in some shared utility routines, and make it accessible to all field editors. This reduces the size of each field editor, and reduces the chance of a stray error finding its way into the code of individual field editors.

The following is a list of each of the common message handling utility routines, along with a brief explanation of how they get their work done.

```
xrEditor *
_MsgNew (instance, infoPtr, dataStructSize, createFunct, drawFunct,
            freeFunct, editorHandler, drawOption)

   xrEditor         * instance;
   xrEditorTemplate * infoPtr;
   INT32              dataStructSize;
   INT32              (*createFunct)();
   INT32              (*drawFunct)();
   INT32              (*freeFunct)();
   INT32              (*editorHandler)();
   INT32              drawOption;
```

_MsgNew() will take care of everything needed to create a new editor instance. If successful, it will return the instance pointer for the new editor instance. On failure, it will return NULL.

The routine validates the pointer contained in "infoPtr" and the window id. Then it will allocate a block of memory to hold the data describing this particular instance. This data structure is different for each field editor and the "dataStructSize" parameter specifies how large the structure needs to be. The routine will then invoke the specified "createFunct()" to allow the field editor to initialize the data structure, allocate some memory for the editor structure, initialize it, and attach it to the specified window. The last step is to tell the field editor to draw the instance, using the specified "drawFunct". The "drawOption" is passed as the single parameter to "drawFunct".

The "editorHandler" parameter must be a pointer to the handler function for the field editor [**XrScrollBar()** for the scrollbar editor]. If the create process fails for some reason, it will invoke the function specified by the "freeFunct" parameter before returning. This is done so that if a field editor has allocated additional memory in its create function, it will be freed up. This feature can be disabled by setting the "freeFunct" parameter to NULL. The "freeFunct" routine is passed a single parameter, a pointer to the editor data structure.

```
xrEditor *
_MsgFree (instance, freeFunct)

   xrEditor * instance;
   INT32      (*freeFunct)();
```

_MsgFree() will free up all memory used by a particular field editor instance and remove its image from the window. It will fail and return NULL if the "instance" parameter is set to NULL. Otherwise, it will return the instance pointer. Deleting an editor instance involves removing its image from the window, freeing all memory used by the instance, and disconnecting the instance from the window to which it was attached. In addition, if the field editor had allocated some memory itself, it can specify a function to release this memory. This is done by means of the "freeFunct" parameter. If "freeFunct" is set to NULL, then this feature is disabled. The "freeFunct" routine is invoked with a single parameter, a pointer to the editor data structure.

```
xrEditor *
_MsgGetState (instance, stateFlags)

   xrEditor * instance;
   INT8     * stateFlags;
```

_MsgGetState() will take a copy of the current state flags associated with the specified editor instance, and return them in the variable pointed to by the "stateFlags" parameter. Upon successful completion, the editor instance pointer will be returned. This routine will fail and return NULL, if either of the parameters has been set to NULL.

```
xrEditor *
_MsgSetState (instance, stateFlags, drawFunct, drawOption)

   xrEditor * instance;
   INT8       stateFlags;
   INT32      (*drawFunct)();
   INT32      drawOption;
```

_MsgSetState() will assign new state flag values, specified by the "stateFlags" parameter, to the specified editor instance. Then it will redraw the instance. The editor's drawing routine, specified by the "drawFunct" parameter, will be invoked and passed the editor instance pointer and the "drawOption" value. Upon successful completion, the instance pointer will be returned. This routine will fail and return a NULL value, if the "instance" parameter is set to NULL.

```
xrEditor *
_MsgRedraw (instance, redrawMode, drawFunct, drawOption)

   xrEditor * instance;
   INT32      redrawMode;
   INT32      (*drawFunct)();
   INT32      drawOption;
```

_MsgRedraw() will handle a redraw request only if the XrREDRAW_ALL option is specified as the redrawMode. If your field editor needs to handle other redraw options, you will have to write your own MSG_REDRAW handler. After verifying the "instance" and "redrawMode" parameter, the editor's drawing function, specified by the "drawFunct" parameter, will be invoked and passed the editor instance pointer and the "drawOption" value. This routine will fail and return NULL, if the "instance" parameter is set to NULL, or the "redrawMode" is set to something other than XrREDRAW_ALL.

```
xrEditor *
_MsgEdit (instance, event, processFunct, inputCode)

   xrEditor     * instance;
   XEvent       * event;
   INT32          (*processFunct)();
   INT16          inputCode;
```

_MsgEdit() serves as a MSG_EDIT handler for those editors which only accept XrSELECT events. This routine will check to see if the instance is capable of handling an event (by checking the state flags, to see if the instance is both visible and sensitive), will verify that the select occurred within the bounds of the editor instance, and it will verify that the event is a select request. If these conditions are met, the event processing routine for the field editor specified by the "processFunct" parameter will be invoked. It will be passed three parameters: the editor instance pointer, a pointer to the XEvent event structure, and a pointer to an xrEvent structure (this is the event which will be pushed onto the input queue later). Before calling the "processFunct", this routine will first fill out the type, serial, send_event, source, inputType, inputCode, valuePtr and valuePt fields in the xrEvent structure. The processing routine *must* fill in the "value1" field and should feel free to place values into any of the other fields it wants to.

Upon completion, this routine will strip out the select up event matching the select event which caused the editor to be invoked. Which means the processing routine should not attempt to strip out the select up event itself! If a select up event is received, it is removed from the input queue. If any other event is received, it will be pushed back onto the front of the input queue and **_MsgEdit()** will no longer wait for the select up event. The return event will then be placed on the front of the application's input queue, to indicate what action was taken by the editor. The event will be filled out as follows:

> type = XrXRAY
> serial = 0
> send_event = 0
> display = _xrCurrentDisplay
> source = window id
> inputType = XrEDITOR
> inputCode = "inputCode" parameter
> value1 = Editor specific value
> value2 = Editor specific value
> value3 = Editor specific value
> valuePtr = "instance" parameter
> valuePt = POINT structure containing the location of the event

The processing routine can set any other fields it might need, since a pointer to the return event is passed to it.

5.9.2 The Graphics Context Structure

The drawing routines provided as part of these utilities, use a structure called the "graphics context" to obtain all drawing characteristics needed when drawing a particular shape. A graphics context can be thought of as a drawing environment. Current field editors provided in the Xrlib toolbox use up to 8 distinct drawing environments. These are globally available to applications as **xrEditorGC1** through **xrEditorGC8**. In addition, the default graphics context **xrDefaultGC** is provided. The "graphics context" structure is a facility provided as part of X Windows. See the chapter on "Graphics Resource Functions" in the 'Xlib - C Language Interface, Protocol Version 11' document for more information on creating and using this facility.

5.9.3 Graphics Context Functions

Almost every field editor needs to perform some drawing function using its foregraound and background colors. Since most of the drawing functions use only the foreground color field within a graphics context, it is necessary to set up two graphics contexts. One which has the desired foreground color set as the foreground color and a second which has the desired backround color as the foreground color. This will allow us to draw using both foreground and background colors. The **_XrInitEditorGCs()** function does provides this capability and in addition, sets up the font field in both structures.

```
_XrInitEditorGCs (foreColor, backColor, fontId)

INT32    foreColor;
INT32    backColor;
Font     fontId;
```

Upon completion of this call, the graphics context referred to by **xrEditorGC1** will have the foreground color field set to the value specified by the *foreColor* parameter, the background color field set to the value of the *backColor* parameter, and the *fontId* set to the specified font. Also the **xrEditorGC2** graphics context will have the foreground color set to *backColor*, the backrground color set to *foreColor* and the fontId set to the specified font.

You may choose to not specify a font by setting the *fontId* parameter to 0.

5.9.4 Shape-Drawing Functions

The utility routines include a series of functions which allow a field editor to draw solid/tiled filled shapes, with an optional border around the object. The types of objects provided here are rectangles, ovals, ellipses and polygons. Each of the drawing functions will use one of the graphics context structures, which were explained earlier. Each of the utility functions is explained below:

```
_XrLine (windowId, gc, x1, y1, x2, y2)

   Window windowId;
   GC     gc;
   INT16  x1;
   INT16  y1;
   INT16  x2;
   INT16  y2;
```

_XrLine() will draw a line between the two points designated as (x1,y1) and (x2,y2), using the drawing environment specified within the graphics context "gc". This is similar to the *XDrawLine*() function described in the "Xlib - C Language Interface, Protocol Version 11" document.

```
_XrRectangle (windowId, gc, drawRect)

   Window      windowId;
   GC          gc;
   RECTANGLE * drawRect;
```

_XrRectangle() will draw a border around the specified rectangle using information within the graphics context. The interior of the rectangle is not affected. This is similar to the *XDrawRectangle*() function described in the "Xlib - C Language Interface, Protocol Version 11" document.

```
_XrFillRectangle (windowId, gc, drawRect)

   Window      windowId;
   GC          gc;
   RECTANGLE * drawRect;
```

_XrFillRectangle() will fill the region defined by the passed-in rectangle definition using information specified within the graphics context. This is similar to the *XFillRectangle*() function described in the "Xlib - C Language Interface, Protocol Version 11" document.

```
_XrBorderFillRectangle (windowId, borderGC, fillGC, drawRect)

   Window      windowId;
   GC          borderGC;
   GC          fillGC;
   RECTANGLE * drawRect;
```

_XrBorderFillRectangle() will first draw a border around the specified rectangle, using the information specified within the graphics context designated by the "borderGC" parameter. Then, then this routine will fill the interior of the region defined by the passed-in rectangle definition using the information within the 'fillGC' graphics context.

5.9.5 Additional Text Facilities

These procedures allow text to be placed anywhere within the bounds of a window and allow a field editor to obtain further information about a font.

```
_XrImageText8 (windowId, gc, len, x, y, string)

   Window   windowId;
   GC       gc;
   INT32    len;
   INT16    x;
   INT16    y;
   STRING8  string;
```

_XrImageText8() will display the specified text string , using the information specified within the graphics context. The length of the string *must* be specified. If the string is NULL terminated, then the "len" parameter should be set to the define "XrNULLTERMINATED". Otherwise, the length must be provided in the len parameter. This routine is similar to the *XDrawImageString()* function described in the "Xlib - C Language Interface, Protocol Version 11" document.

```
_XrTextInfo (fontInfo, textInfo)

   XFontStruct  * fontInfo;
   xrTextInfo   * textInfo;
```

_XrTextInfo() allows a program to obtain some additional information describing a particular font. This additional information includes the ascent, descent, maximum character width, average character width, and the leading for the font. The information is returned in the structure pointed to by the "textInfo" parameter.

5.9.6 Common Editor Functions

This section will outline some general purpose field editor routines that do not fall under any of the previously discussed sections.

```
_XrInitEditorStruct (editorPtr, infoPtr, dataPtr, function)

   xrEditor         * editorPtr;
   xrEditorTemplate * infoPtr;
   char             * dataPtr;
   xrEditor         * (*function)();
```

Whenever a field editor creates a new instance, it must allocate an editor structure, which will then be attached to the specified window. The editor structure must be initialized to contain the following information:

- The window Id.
- The editor function.
- The editor state.
- The editor rectangle.
- A pointer to the instance specific data.

Using the passed parameters, **_XrInitEditorStruct()** will fill each of these fields within the editor structure pointed to by the "editorPtr" parameter.

```
_XrCatchableKey (editorPtr, eventPtr)

   xrEditor        * editorPtr;
   xrEvent         * eventPtr;
```

Whenever a field editor receives a MSG_EDIT request, it must first check to see if it is an event which it should handle. This involves making the following set of checks:

- Verify that the instance pointer is not NULL.
- Verify that the event pointer is not NULL.
- Verify that the instance is both SENSITIVE and VISIBLE.
- Verify that the event occurred within the instance's editor rectangle.

_**XrCatchableKey**() will take care of making all of these checks. If all of the above conditions are met, then a value of TRUE will be returned. Otherwise, a value of FALSE is returned.

```
_XrMakeInvisible (windowId, rectPtr, makeGCFlag)

   Window      windowId;
   RECTANGLE * rectPtr;
   INT8        makeGCFlag;
```

All field editors need to be able to make a particular editor instance become invisible; such as when an application clears the VISIBLE state flag. Normally, this involves filling the region defined by the instance's editor rectangle with the background tile for the window to which the instance is attached. This is exactly what _**XrMakeInvisible**() will do. Currently, the "makeGCFlag" is unused and should be set to NULL.

5.10 Field Editor Template 2

Now that we have discussed each of the utility routines available to field editor writers, we will take our original field editor template, and rewrite it using these utilities. If you are still unclear as to the exact functions performed by these common utilities, look back at the first editor template presented earlier that did not use the common routines, and instead have all their functionally inline.

```
xrEditor *
XrEditorName (editorName, message, data)

   xrEditor * editorName;
   INT32      message;
   INT8     * data;

{
   /* Determine the action being requested */
   switch (message)
   {
```

```
case MSG_NEW:
{
   /* Create a new instance of this editor */
   return ((xrEditor *)_MsgNew(editorName, data, sizeof(xrEditorNameData),
                              createEditorName, drawEditorName,
                              enFreeMemory, XrEditorName, NULL));
}



case MSG_FREE:
{
   /* Destroy the specified editor instance */
   return ((xrEditor *) _MsgFree (editorName, enFreeMemory));
}



case MSG_GETSTATE:
{
   /* Return the current state flags for an editor instance */
   return ((xrEditor *) _MsgGetState (editorName, data));
}



case MSG_SETSTATE:
{
   /* Change the state flags for a particular editor instance */
   return ((xrEditor *) _MsgSetState (editorName, data,
                                      drawEditorName, NULL));
}



case MSG_REDRAW:
{
   /* Redraw the specified editor instance */
   return ((xrEditor *) _MsgRedraw (editorName, data,
                                    drawEditorName, NULL));
}



case MSG_SIZE:
{
     /*
      * Return the size of the rectangle needed to enclose
      * an instance of this editor, using the specifications
      * passed in by the application program.
      */
     xrEditorNameInfo *enInfoPtr;

     enInfoPtr = (xrEditorNameInfo *)data;

     if (enInfoPtr == NULL)
```

```
      {
         xrErrno = XrINVALIDPTR;
         return ((xrEditor *) NULL);
      }
      if (sizeEditorName (enInfoPtr, &enInfoPtr->editorRect) == FALSE)
         /* Size request failed; xrErrno set by sizeEditorName() */
         return ((xrEditor *)NULL);

      return (editorName);
}
```

```
case MSG_MOVE:
{
   /* Move the origin for the specified editorRect */
   POINT     * ptPtr (POINT *)data;
   RECTANGLE   oldRect;

   if (editorName == NULL)
   {
      xrErrno = XrINVALIDID;
      return ((xrEditor *) NULL);
   }
   else if (ptPtr == NULL)
   {
      xrErrno = XrINVALIDPTR;
      return ((xrEditor *) NULL);
   }

   /* Reset the origin of the editor rectangle */
   XrCopyRect (&editorName->editorRect, &oldRect);
   editorName->editorRect.x = ptPtr->x;
   editorName->editorRect.y = ptPtr->y;

   if (editorName->editorState & XrVISIBLE)
   {
      /* Make the instance invisible, if necessary */
      _XrMakeInvisible (editorName->editorWindowId, &oldRect, TRUE);

      /* Redraw the relocated instance */
      drawEditorName (editorName, NULL);
   }

   /* Force recalculation of editor group rectangle */
   XrEditorGroup (NULL, MSG_ADJUSTGROUPRECT, editorName);
   return (editorName);
}
```

```
case MSG_EDIT:
{
   /*
    * Process the incoming event, and generate a return
    * event, to indicate to the application program how
    * the editor instance was modified.
    */
   return ((xrEditor *) _MsgEdit (editorName, data,
                              processEditorName, XrEDITORNAME));
```

```
      }


      default:
      {
         /* All other messages are invalid */
         xrErrno = XrINVALIDMSG;
         return ((xrEditor *) NULL);
      }
   }  /* end of switch */
}  /* end of XrEditorName() */



/*
 * Free up the block of memory which our create routine
 * allocated at the time we first created the instance.
 */

INT32
enFreeMemory (enDataPtr)

   xrEditorNameData * enDataPtr;

{
   (*xrFree) (enDataPtr->extraData);
}
```

5.11 Sample Field Editors

This section will attempt to provide some insight into the organization of an Xrlib field editor. You should refer to the actual source code for the indicated field editors to see how they perform certain tasks. Each of the field editors presented here follows the template which was presented in the previous section.

In particular, there are three concepts which we will attempt to cover in this section:

- Provide an example of a simple field editor (XrStaticRaster), to demonstrate the general organization of a typical field editor.
- Provide an example of a multiple entity field editor (XrCheckBox), to demonstrate a sample algorithm to lay out one type of multiple entity field editor. And demonstrate how input handling is performed for this type of editor.
- Provide an example of how an editor is constructed by combining an existing field editor together with some new software, to form a new editor. This concept is useful when an editor needs to be created which contains a scrollbar, along with some raster data (XrRasterEdit).

Each of the sub-sections which follow will explain one of the above concepts.

5.11.1 Simple Field Editor

The static raster editor is an example of a short and simple field editor. When a new instance is created, there is little work which needs to be done, other than displaying the raster data itself. It also demonstrates some of the steps taken to process incoming input events. (Refer to the source listing for the Static Raster field editor.)

5.11.2 Multiple Entity Field Editors

Field editors, such as the checkbox and pushbutton editors find it convenient to allow an application to create multiple entities of the editor with a single request. Usually, these types of editors are created as a group of related items. Rarely are a single checkbox or a single pushbutton created.

Designing an editor to handle multiple entities can be a real challenge. In particular, the algorithm used to determine the placement of the individual items is not always easy to derive. The example which follows shows one algorithm which may be used to do this placement. Note that this is not the only algorithm available and depending upon how the items are to be arranged, this algorithm may not fit your needs. But this example may be enough to help you customize your own layout routines.

The algorithm used by the checkbox field editor to calculate the location of each of the individual checkboxes makes the following assumptions:

- Each checkbox is composed of a square box and an optional label located to the right, separated by some white space.
- The checkboxes will be laid out in rows and columns, and must fit within the specified editorRect rectangle structure.
- Each checkbox in a given column will line up.
- If the editorRect is larger than needed to contain the checkboxes, the extra space will be equally distributed as white space between the rows and columns.

Before covering the algorithm to picture how a sample checkbox editor instance might look. The outline for the editorRect is provided to show the relationship between the checkbox location and the borders of the editorRect.

```
----...................----...........
|  |   button 1        |  |   button 2.
----                   ----           .
.                                     .
.                                     .
----                   ----           .
|  |   button 3        |  |   button 4.
----                   ----           .
.                                     .
.                                     .
----                                  .
|  |   button 5                       .
----..................................
```

The algorithm used by the checkbox editor works in the following manner:

- Determine the number of rows in the instance, and the width (in pixels) of each column of checkboxes. Use this information to determine if the editorRect is larger than needed. If it is, this extra space can be used as padding between the rows and columns. The padding values will then be calculated and saved.

- Set the initial checkbox location to the upper left hand corner of the editorRect. This is where the first checkbox will be placed. This location will be updated later to indicate the location of the next checkbox.
- Do the layout calculations. This is done one column at a time, starting with column 1 (the leftmost one). As each element in this column is laid out, a check is made to see if it is the widest element in the column. This information is used later to offset the next column. After calculating the checkbox location, the location for the optional label will be calculated.
- After a column has been laid out, the algorithm will calculate the position for the first checkbox in the next column, and then repeat the process for the each column.
- Now all of the entities have been laid out in equally spaced rows and columns.

5.11.3 Creating From Existing Editors

Sometimes a field editor may find it advantageous to incorporate an instance of another editor, rather than attempting to write new code to re-create the desired piece. As an example of this, the raster editor has the capability of displaying a scrollbar as part of one of its instances. It doesn't attempt to duplicate the scrollbar code. Instead, it simply creates a scrollbar instance, and attaches it to its editor instance.

The major advantages of allowing an editor to do this are:

- The field editor writer does not waste time trying to re-create portions of an existing field editor.
- The software which controls the incorporated editor is already debugged and running.
- The incorporated component(s) look and behave the same as any other instances of those field editors.

Before discussing the how to do this, there are two terms we need to define. The term "main editor" refers to the new field editor being designed, and the term "sub-editor" refers to those existing editors being incorporated into the main editor.

The mechanism for incorporating other field editors is simple to use. But, there are several rules which must be followed if you expect your new field editor to function:

- All sub-editors which are included within an instance of the main editor, must reside *COMPLETELY* within the editorRect associated with the main editor instance. Since a field editor knows what components to include when created, it can make sure the application supplied editorRect is large enough to hold all components. The editor's MSG_SIZE routine must also take these sub-editors into consideration. This normally involves having the main editor do a MSG_SIZE for each sub-editor it use, and combining this information to help determine its total size.

  The main editor *must* completely contain the sub-editors, because sub-editors do not reside in the editor list maintained by the window. Therefore, they are unable to receive input events via normal **XrInput()** dispatching. This is resolved by having the main editor check input events to see if they were really meant for a sub-editor. Remember - the main editor created the sub-editor instances, so it knows exactly where their editorRects are located. The limitation is that the main editor only receives events which occur within its editorRect. So if a sub-editor does not lie completely within the main editor's rectangle, then that portion of the sub-editor outside the rectangle will never receive input events.

- When a main editor instance is created, it will normally create and display its components, before calling the sub-editors to create their components. This helps to guarantee that all the components overlap in the proper fashion. As the sub-editors are created, they need to be removed from the window's editor list. The main editor may behave in an unfriendly fashion if this is not done, because it will not have absolute control over its sub-editors. Remove sub-editors from the editor list is by using the **XrEditor()** intrinsic, with the MSG_REMOVE message. Pass the instance pointer for the editor instance to be removed, along with a pointer to the window to which it is attached.

- Normally, when an application creates a field editor, it returns a pointer to the editor structure associated with that new editor instance. This serves to act as a unique identifier for all future references to that instance. This is also true when a main editor creates sub-editors; each sub-editor will return its own editor instance pointer. The main editor *must* define space within its own data structures to hold these values for the sub-editors. Without these values the main editor can not communicate with the sub-editors. A separate xrEditor variable for each sub-editor must be included in the main editor. For instance:

```
xrEditor * subEditor1,
         * subEditor2;
```

- Since the sub editors do not reside in the editor list, it is the responsibility of the main editor to keep them up-to-date. Because the editors have been grouped into a single entity, they must act as a single entity. This implies that anytime the main editor receives a MSG_REDRAW, MSG_FREE, MSG_MOVE or MSG_SETSTATE message, these commands should also be performed for the sub editors.
- Remember, it is the responsibility of the main editor to route input events to the sub-editors, as needed. Each time the main editor receives an input event, it must check to see if it belonged to one of the sub-editors. This is done by seeing if the event location fell within the bounds of one of the sub-editor's editorRect rectangles. If it does not belong to a sub-editor, then the main editor can process the event. Otherwise, the main editor must invoke the sub-editor's input handling routine by issuing a MSG_EDIT message to the sub-editor, and passing the event information with it. When the sub-editor is done processing the event, it will in turn push event information onto the front of the input queue, and return. The main editor can then remove the event from the front of the queue, and perform the desired action. But, before the main editor can return it needs to first push a fake "select up" event on the front of the input queue, followed by a second event containing its own result information. The reason for this is discussed in a paragraph which follows. This event handling mechanism allows the information generated by the sub-editors to be kept and processed by the main editor.
- Finally, when the main editor instance is freed up, it *must* issue a MSG_FREE to each of its sub-editors. Otherwise the space occupied by the sub-editor data structures will become "lost" and irrecoverable until the application terminates.

All field editors should swallow select up events which occur after the select event which caused their processing routine to be called. This is done to prevent an application from being inundated with select up events. If an editor uses the **_MsgEdit()** edit routine to invoke its processing routine, this will be handled. **_MsgEdit()** automatically waits for the select up event and swallows it. Unfortunately, this can create a problem, when one editor's processing routine issues a MSG_EDIT to another editor. For example the raster editor calls the scrollbar editor. When the raster editor receives a select event, the following occurs:

- **XrInput()** will pass the event to **XrRasterEdit()** for processing.
- **XrRasterEdit()** invokes **_MsgEdit()**, passing the event, and a pointer to the raster edit event processing routine [processREdit()].
- **_MsgEdit()** verifies the event, and then passes it to **processREdit()**.

- **processREdit()** determines that the event was in one of the scrollbars, and issues a MSG_EDIT to **XrScrollBar()**, passing it the event.
- **XrScrollBar()** invokes **_MsgEdit()**, passing the event, and a pointer to the scrollbar event processing routine [processScrollBar()].
- **_MsgEdit()** verifies the event, and then passes it to **processScrollBar()**.
- **processScrollBar()** processes the event, and returns to **_MsgEdit()**.
- **_MsgEdit()** waits for the select up to occur, swallows it, pushes the return event generated by **processScrollBar()**, and then returns to **XrScrollBar()**.
- **XrScrollBar()** returns to **processREdit()**.
- **processREdit()** pops the scrollbar event off the input queue, processes it, and then returns to **_MsgEdit()**.
- **_MsgEdit()** waits for the select up to occur, so that it can swallow it. Unfortunately, the first call to **_MsgEdit()**, by the scrollbar editor, already swallowed the select up event.

Thus, **_MsgEdit()** proceeds to "hang", until some X event occurs. This is undesirable! The solution is for **processREdit()** to push a fake select up event onto the front of the input queue, after processing the event returned by **XrScrollBar()**. If this is done, then everything works fine. The following code sample shows how this can be done, using a utility routine provided by the Xrlib intrinsics:

```
XEvent          event;
xrWindowEvent   windowEvent;
XButtonReleasedEvent *selectUp = (XButtonReleasedEvent *) &event

XrGetWindowEvent (XrSELECTUP, &windowEvent);
selectUp->type = windowEvent.type;
selectUp->serial = 0;
selectUp->send_event = 0;
selectUp->display = _xrCurrentDisplay;
selectUp->state = windowEvent.modifier;
selectUp->button = windowEvent.code;

XrInput (NULL, MSG_PUSHEVENT, selectUp);
```

A Reference Information (3X)

This appendix contains manual pages for the functions and library provided with the Xrlib toolbox. The following list shows the page reference for each topic. When using the *man* utility, use the MAN reference with the indicated uppercase and lowercase characters.

Topic	MAN Page
XrAddPt	XrPoint(3X)
XrCheckBox	XrCheckBox(3X)
XrCopyPt	XrPoint(3X)
XrCopyRect	XrRectangle(3X)
XrEditor	XrEditor(3X)
XrEditorGroup	XrEditorGroup(3X)
XrEmptyRect	XrRectangle(3X)
XrEqualPt	XrPoint(3X)
XrEqualRect	XrRectangle(3X)
xrErrno	xrErrno(3X)
XrGetWindowEvent	XrUtilities(3X)
XrGlobals	XrGlobals(3X)
XrGroupBox	XrGroupBox(3X)
XrInit	XrInit(3X)
XrInput	XrInput(3X)
XrInputConf	XrInputConf(3X)
XrInputInit	XrInputInit(3X)
XrInputMap	XrInputMap(3X)
XrInputMute	XrInputMute(3X)
XrInsetRect	XrRectangle(3X)
XrListEdit	XrListEdit(3X)
XrMapButton	XrUtilities(3X)
XrMenu	XrMenu(3X)
XrMessageBox	XrMessageBox(3X)
XrOffsetPt	XrPoint(3X)
XrOffsetRect	XrRectangle(3X)
XrPageEdit	XrPageEdit(3X)
XrPanel	XrPanel(3X)
XrPoint	XrPoint(3X)
XrPtInRect	XrRectangle(3X)
XrPt2Rect	XrRectangle(3X)
XrPushButton	XrPushButton(3X)
XrRadioButton	XrRadioButton(3X)
XrRasterEdit	XrRasterEdit(3X)
XrRasterSelect	XrRasterSelect(3X)
XrRectangle	XrRectangle(3X)
XrResource	XrResource(3X)
XrScrollBar	XrScrollBar(3X)

Topic	MAN Page
XrSectRect	XrRectangle(3X)
XrSetPt	XrPoint(3X)
XrSetPtRect	XrRectangle(3X)
XrSetRect	XrRectangle(3X)
_XrSetUpGC	XrUtilities(3X)
XrStaticRaster	XrStaticRaster(3X)
XrStaticText	XrStaticText(3X)
XrStringWidth	XrUtilities(3X)
XrSubPt	XrPoint(3X)
XrTextEdit	XrTextEdit(3X)
XrTitleBar	XrTitleBar(3X)
XrUnionRect	XrRectangle(3X)
XrUtilities	XrUtilities(3X)
XrVersion	XrUtilities(3X)

NAME

Xlib - C Language X Window System Interface Library

SYNOPSIS

#include <X11/Xlib.h>

DESCRIPTION

This library is the low level interface for C to the X protocol. The list of subroutines is too long to list here, comprising more than 250 subroutines.

This library gives complete access to all capability provided by the X window system, and is intended to be the basis for other higher level libraries for use with X

More subroutines will be added to this library with time.

FILES

/usr/include/X11/Xlib.h, /usr/lib/X11lib.a

ORIGIN

MIT Distribution

SEE ALSO

'Xlib - C Language Interface, Protocol Version 11'
X(1)

NAME

XrCheckBox - an editor for check boxes

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

XrEditor*
XrCheckBox (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The checkbox editor is used to create and process a group of related checkboxes within a given window. The number of checkboxes, and how they are displayed, is completely controlled by the application. Each checkbox may also include an optional text label, which will be displayed to the right of the checkbox.

The size of the checkboxes are proportional to the size of the font being used to display the text labels. By allowing an application to specify this font, the size of the checkboxes can be adjusted for individual displays.

This field editor supports two distinct sets of state flags: those associated with the whole instance (which are modified using MSG_GETSTATE and MSG_SETSTATE), and those which are assigned to the individual checkboxes (which are modified using MSG_GETITEMSTATES and MSG_SETITEMSTATES). The state flags which are associated with the instance as a whole have precedence over the individual state flags. For example, if the XrVISIBLE flag is cleared for the instance, then none of the checkboxes will be displayed, regardless of their individual state flag settings. But, if the XrVISIBLE flag is set for the instance, then the field editor will look at the state of the XrVISIBLE flag associated with each checkbox, to determine if the entity should be displayed.

Using the cursor and the mouse, a user can interactively select a checkbox, by moving the cursor over one of the boxes, and 'clicking' the mouse button. This will cause the state of the chosen box to toggle. If the box had previously been active (filled), then it will now become inactive (not filled). Likewise, if the box was previously inactive (not filled), then it will now become active (filled). An application will then be notified of the change, and may handle it accordingly.

The checkbox editor will not restrict the number of boxes which may be active at any given moment, within a given instance. Instead, each checkbox in an instance is treated like an independent entity; its value does not rely on nor affect the value of any other checkbox in the instance.

A checkbox editor instance is composed of 3 components:

- The editor instance's background area.
- The individual checkboxes.
- The checkbox labels.

Of these 'selectable' regions, only select events which occur in one of the individual checkboxes will affect the instance. When one of these is selected, the editor will toggle the state of the checkbox, and then the application program will be notified that the value of one of the checkboxes has changed, along with the index of the checkbox which was modified. If a select event occurs in any of the other components, the application will be notified that a select occurred, but the editor instance was not modified.

When an editor instance is created, the initial value and the label strings for each checkbox will be specified by the application program. For the values, a pointer to an array of 8 bit integer values must be passed in. For the label strings, a pointer to an array of character strings may be passed in. The checkbox editor will not save copies of these pieces of data. Instead, it will save a copy of the pointers only. This will allow an application program to have immediate access to the checkbox values, since the array resides in the application's domain.

The editor allows an application to modify the data values associated with the checkboxes, however, the checkbox labels must not be modified, once the instance has been created. This editor will rely on the fact that when an application modifies any of the checkbox data values, the application will tell the editor to redraw; this allows the editor to remain in sync with the current checkbox data. Modifying the data, without doing a redraw, will cause the editor to behave in an unfriendly fashion.

This editor allows an application program to obtain information about an instance in two ways:

> By means of an input event returned when a checkbox
> is modified.
>
> By means of the shared data values mentioned above.
> This refers to the array of data describing the
> value of each checkbox.

When the editor draws the specified checkbox instance, it will attempt to space out the checkboxes, to take full advantage of the specified *editorRect*. If the rectangle is larger than needed, then the extra space will be divided equally as blank space between the rows and columns. If the rectangle is too small, then the request will fail.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a checkbox editor instance in a port. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   XFontStruct   * editorFont;
   INT16         numFields;
   INT16         numCols;
   INT8          ** labels;
   INT8          * values;
   INT8          * stateFlags;
} xrCheckBoxInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the region into which the checkbox instance is to be located. If the rectangle is larger than is needed to hold the specified instance, then the extra space will be used as padding between the checkboxes. If the rectangle is too small, then the request will fail.

editorState
This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
This field specifies the foreground color to be used when drawing the editor instance. It will be used to display the checkbox labels, and the checkbox borders. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
This field specifies the background color to be used when drawing the editor instance. It will be used to fill the interior of an inactive checkbox. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont
This is a pointer to a structure which describes the font to be used when creating this editor instance. If the pointer has been set to **NULL**, then the editor will use the default system base font. The font is important not only because it describes how the labels will look, but it will also be used to determine how large the checkboxes should be.

numFields
This value indicates the total number of checkboxes which are included in this instance. Once specified, this value cannot be changed.

numCols
This value indicates the number of columns into which the checkboxes are to be displayed. Using this value, and *numFields*, the editor will automatically calculate the number of rows. If this value is greater than *numFields*, then the create request will fail.

labels
This is a pointer to an array of **NULL** terminated character strings, representing the labels associated with each checkbox. If this pointer is **NULL**, then none of the checkboxes will have labels. The first entry in the array will correspond to checkbox 0, the second to checkbox 1, etc. If an entry in this array is set to **NULL**, then the corresponding checkbox will have no label. The editor will not keep a copy of each of these strings; instead, it will only keep a copy of the pointer to the string array. It will rely on the application to *NOT* modify these strings, once the instance has been created.

values
This is a pointer to an array of 8 bit integer values, representing the initial state of each of the checkboxes. If this pointer is **NULL**, then the create request will fail. The first entry in this array corresponds to the value of checkbox 0, etc. A value of **TRUE** signifies that the checkbox should be drawn as 'active' (filled), while a value of **FALSE** signifies that a checkbox should be drawn as 'inactive' (not filled). The editor will not keep a copy of this array. Instead, it will only keep a copy of the pointer to this array. Any time the value of a checkbox is modified, the editor will update the appropriate location in this array, and then notify the application. The application has immediate access to the checkbox values, since the array resides in its data area.

stateFlags
This is a pointer to an array of values, each containing the initial state flag settings to be associated with each checkbox in the instance. If this pointer has been set to **NULL**, then the state flags for each entity will be set to **XrSENSITIVE** and **XrVISIBLE**. The first entry in this array corresponds to checkbox 0, etc.

The editor will then draw the checkbox editor instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application This value must be used thereafter, whenever the application wishes to refer to this particular editor instance.

MSG_FREE
This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, which was returned by the **MSG_NEW** message; this parameter specifies which instance is to be destroyed.

When a checkbox instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE
This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified checkbox editor instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE
This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given checkbox editor instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If an instance is visible, but not sensitive, then each checkbox will be drawn and filled with a 50% pattern.

MSG_GETITEMSTATES
This message allows an application to obtain a copy of the individual state flags associated with each of the checkboxes contained within the specified editor instance. These state flags differ from the state flags obtained using **MSG_GETSTATE**. The *instance* parameter must specify the editor instance to be queried, while the *data* parameter should point to an array of 8 bit integer values. The state flags will be returned by means of this array. The flags associated with checkbox 0 will be returned in the first slot in the array, etc.

MSG_SETITEMSTATES
This message allows an application to modify the individual state flags associated with each of the checkboxes contained within the specified editor instance. Presently, only the **XrVISIBLE** and the **XrSENSITIVE** flags are supported. The *instance* parameter must specify the editor instance which is to be modified, while the *data* parameter should point to an array of 8 bit integer values. The new state flags which are to be assigned to each entity within the instance will be obtained from this array. The value in the first slot of the array will be assigned to

checkbox 0, etc. After the new state flags have been saved, only those checkboxes whose state flags changed will be redrawn.

MSG_GETITEMCOUNT

This message allows an application to obtain a count, which indicates the number of individual checkboxes in the specified instance. The editor will assume that the *instance* parameter specifies the editor instance to be queried, and the *data* parameter points to a 32 bit integer value; the item count value will be returned by means of this integer.

This message is useful when used in conjunction with the **MSG_GETITEMRECTS** message. It allows an application to obtain the number of items in the instance, so that the application can then allocate enough memory to hold the rectangle information returned by **MSG_GETITEMRECTS.**

MSG_GETITEMRECTS

This message returns the coordinates for the rectangle which describes each of the individual checkboxes; these rectangles describe only the individual checkboxes - it does not include the labels. The message will expect the *instance* parameter to indicate the editor instance to be queried, and the *data* parameter to point to a structure of the following format:

```
        RECTANGLE  itemRects[x];
```

This array will then be filled with the rectangle information, and returned to the application.

Before an application can make this call, it needs to know the number of items in the specified instance, so that it can allocate a structure large enough to hold all of the rectangle information. The application should use the **MSG_GETITEMCOUNT** message to obtain this information; the application can then allocate an array large enough to hold all of the rectangle entries.

The order of the rectangle items returned in the array directly correlates to the order the items were originally created in. The first element in the array will describe checkbox 0, the second checkbox 1, etc.

This message is useful to those applications which have a need of knowing where the individual items in a checkbox instance are located. The most common use would be by a forms controller, which would use the information to place an 'active field' indicator by a given item.

MSG_REDRAW

This editor provides an application with the means for requesting that an instance of checkboxes be redrawn. The current values, labels and font information will be used.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The checkbox editor supports the following redraw mode:

- **XrREDRAW_ALL**

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a checkbox editor instance. The checkbox editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   XFontStruct * editorFont;
   INT16         numFields;
   INT16         numCols;
   INT8       ** labels;
   INT8        * values;
   INT8        * stateFlags;
} xrCheckBoxInfo;
```

The only fields which must be filled out by the application program BEFORE issuing this call, are the *numFields*, *numCols*, *labels*, and *editorFont* fields; all other fields are ignored.

Using the supplied information, the editor will determine the size of the rectangle needed to contain this instance. In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain this instance; an application program can then offset this rectangle, to position it where ever it likes, within its window.

If *numCols* > *numFields*, then the request will fail.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location. This occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a checkbox instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the first parameter, *instance*, indicating which checkbox instance to process, and the third parameter, *data*, pointing to an *XEvent* structure.

The checkbox editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within a checkbox instance, the first thing done is to determine if one of the checkboxes was selected.

If one of the checkboxes was selected, then the value of the selected checkbox will be toggled, and the particular box redrawn. Afterwards, an input event will be added to the front of the application's input queue, informing it that one of the checkboxes was selected. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	the display pointer
source	=	the window id
inputType	=	XrEDITOR
inputCode	=	XrCHECKBOX
value1	=	XrSELECT
value2	=	index of the modified checkbox
valuePtr	=	pointer to instance's editor structure

The editor will update the value of the selected checkbox in the array shared with the application program. This value is then directly available to the application program.

If a select occurs within a checkbox instance, but not within the boundary of a checkbox, then the editor will do nothing but push an input event onto the front of the application's input queue. The event will notify the application that the editor was selected, but no action took place. In addition, the cursor position at the time the select event occurred will be returned. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	the display pointer
source	=	the window id
inputType	=	XrEDITOR
inputCode	=	XrCHECKBOX
value1	=	NULL
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

ERROR CONDITIONS

Messages to the checkbox editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

numCols > *numFields* [*XrINVALIDPARM*].

values is set to NULL [XrINVALIDPTR].

An invalid editor rectangle is specified [XrINVALIDRECT].

A call to 'X' fails { XCreatePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** was specified [XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

numCols > *numFields* [*XrINVALIDPARM*].

MSG_RESIZE

An invalid editor rectangle is specified [XrINVALIDRECT].

MSG_GETSTATE, MSG_MOVE, MSG_RESIZE, MSG_GETITEMRECTS,

MSG_GETITEMSTATES, MSG_SETITEMSTATES and MSG_GETITEMCOUNT

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [INVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrEditor - Facilities providing some field editor manipulations

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
```

```
XrEditor (windowId, message, data)
Window windowId;
INT32 message;
INT8 * data;
```

DESCRIPTION

XrEditor() provides a group of functions used to manipulate the list of editors attached to a window. The messages to the function can be used by both the Xrlib editors and the application. However, there are restrictions on the use of some of the messages. These restrictions will be noted along with the message descriptions.

The messages use the xrEditor structure for passing the editor information. The editor structure is defined as follows.

```
typedef struct _xrEditor
{
  struct _xrEditor * (*editorFunct)();
  Window editorWindowId;
  RECTANGLE editorRect;
  INT8 editorState;
  INT8 * editorData;
  struct _xrEditor * nextEditor;
} xrEditor;
```

The following set of messages should only be issued by field editors.

MSG_ADD

This message allows a field editor to attach a particular editor instance to the active editor group within a specific window. Once an editor instance has been attached to a window, it is then able to receive input. The *windowId* parameter specifies the window to which the instance is to be attached. The *data* parameter points to the xrEditor structure associated with the editor instance. The XrEditor() function maintains a linked list of editor instances attached to each window and the new editor instance will be linked to the end of this list.

MSG_REMOVE

This message allows a field editor to remove a particular editor instance from the list of editors attached to a window. Once an editor instance becomes unattached from a window, it is no longer able to receive any user input. Therefore, an editor instance should only be unattached when it is no longer needed. The *windowId* parameter specifies the window to which the instance is currently attached. The *data* parameter points to the xrEditor structure associated with the editor instance.

The following set of messages can be issued by an application.

MSG_PROCESSKEY

This message will take a passed-in XEvent structure, and see if the event occurred within any of the field editor instances associated with the specified window. If the event did not occur within a rectangle of an editor group whose state is visible and sensitive, the message will fail. Also, if the event did not occur within one of the window's field editors, then the message will fail. A value of **FALSE** will be returned for both cases. If, however, the event did occur within one of the window's field editor instances, and the editor group in which the editor resides is visible and sensitive, then the handler routine for that editor instance will be called and given the opportunity to process the event. The return value generated by the field editor will be returned when processing is completed.

At invocation time, the *windowId* parameter must specify the window in which the input event occurred and the *data* parameter must point to the XEvent structure containing the input information.

MSG_REDRAW

This message allows an application to request that all field editor instances which lie within a specified rectangular portion of a window be completely redrawn. This is especially useful when a portion of a window which was hidden by another window is uncovered.

To use this facility, the *windowId* parameter must be set to specify the window to be partially redrawn. The *data* parameter must point to a RECTANGLE structure which defines the rectangular region to be redrawn.

MSG_REDRAWALL

This message provides an application with the means of easily redrawing all of the editors attached to a window. For this message, *windowId* should contain the window in which the editors are to be redrawn. *data* is unused and should be set to NULL.

MSG_SETSTATE

This function allows an application to quickly modify the state flags associated with every field editor instance attached to a particular window. The state flags for each instance will be set to the same value. The only flags that should be used with this message are **XrVISIBLE** and **XrSENSITIVE**, because they are the only flags supported by all field editors. Do not use any other state flags such as **XrTRANSPARENT**, **XrFIXEDORIGIN**, **XrAUTOREDRAW_OFF** or **XrMULTIPLE_SELECTION**. To use this function, the application must set the *windowId* parameter to specify the window whose field editors are to be modified, and the *data* parameter must be set to the state value to be used for the set of editors.

RETURN VALUE

windowId is returned if the call to XrEditor() succeeds. **FALSE** is returned otherwise.

ERROR CONDITIONS

If XrEditor() returns **FALSE**, the *xrErrno* global will be set to one of the following values.

xrErrno will be set to **XrINVALIDID** for all of the messages if the window identified by the *windowId* parameter has not been registered with XrInput().

xrErrno will be set to **XrINVALIDMSG** for any messages besides the messages listed above.

xrErrno will be set to **XrINVALIDPARM** whenever the *data* parameter contains invalid information.

xrErrno will be set to **XrINVALIDEVENT** if **MSG_PROCESSEVENT** fails because the event does not fall within one of the editors attached to the window or the event does not fall within a visible and sensitive editor group.

ORIGIN

HP

SEE ALSO

XrInput(3X), XrEditorGroup(3X), xrErrno(3X)

NAME

XrEditorGroup - Facilities for handling field editor groups.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditorGroup *
XrEditorGroup (groupInstance, message, data)
xrEditorGroup * groupInstance;
INT32 message;
INT8 * data;
```

DESCRIPTION

XrEditorGroup() provides the messages necessary to setup and access groupings of editors. This provides for the capability of separating the set of editors attached to a window into specified groups and manipulate the editors as a unit. The types of functions which can be applied to editor groups include getting and setting the groups editor state, clearing the rectangular area which defines the group, etc... XrEditorGroup() is provided for advanced editor usage and can be ignored for normal accessing and handling of editors.

The structure used for each editor group is defined below.

```
typedef struct _xrEditorGroup
{
  RECTANGLE          groupRect;
  INT8               groupState;
  xrEditor          * editorList;
  xrGraphic         * graphicList;
  struct _xrEditorGroup * nextGroup;
} xrEditorGroup;
```

The following set of messages are used to create, access, and destroy editor groups.

MSG_NEW

This messages creates a new editor group, initializing the elements, adding it into the group list attached to the window, and making it the active group. The concept of "active group" means that any editors created for the window will be attached through this group. For this call, *groupInstance* and *data* are both unused and should be set to NULL. The group instance of the new group will be returned by this message.

MSG_FREE

This message destroys the editor group and frees the editors attached to the group. For this call, *data* is unused and should be set to NULL.

MSG_GETDEFAULTGROUP

This message returns the point to the default editor group that is added to a window when the window is registered with XrInput(). For this message, *data* should contain the id of the registered window.

MSG_GETSTATE

This message returns the state of the editor group. *data* is a pointer to an INT8 variable and will be set to the state of the group.

MSG_SETSTATE

This message sets the state of the editor group. *data* contains the new group state which can be any combination of the editor state defines **XrVISIBLE** and **XrSENSITIVE.**

MSG_ADDTOGROUP
: This message tells the editor list handling function which group new editors are to be added. If the window has only the default group defined, editors will be added to this group automatically. When a new group is created through MSG_NEW, it becomes the "active group" (i.e. the group in which editors will be added) so that if the application wants to add editors to an existing group it needs to issue the MSG_ADDTOGROUP message. For this message, *groupInstance* should contain the instance pointer of the group which is to be made active. The *data* value is unused and should be set to NULL As stated, each window gets a default group set up for it when it is registered. Applications which need to make this group the "active group" must get the group instance pointer through message MSG_GETDEFAULTGROUP.

MSG_GETGROUPRECT
: This message returns the group rectangle through *data* which contains a pointer to a rectangle structure. *groupInstance* should be set to the instance pointer of the group in which the group rectangle is to be extracted.

MSG_SETGROUPRECT
: This message sets the group rectangle to the rectangle pointed at by *data*. The *groupInstance* value should be set to the instance pointer of the group in which the group rectangle is to be set. The rectangle supplied must be large enough to contain the set of editors attached to the window or the message will fail.

MSG_ADJUSTGROUPRECT
: This message causes the group rectangle to be recalculated and set to the minimum rectangle needed to enclose the set of editors attached to the window. For this message, *data* is unused and can be set to NULL.

MSG_CLEARRECT
: This message is used to clear the area defined by the group rectangle. This is done by redrawing the window's current background tile in the defined area. *groupInstance* should be set to the instance pointer of the group which is to be cleared. *data* is unused and can be set to NULL.

RETURN VALUE

groupInstance is returned if the call to XrEditorGroup() succeeds. NULL is returned otherwise.

ERROR CONDITIONS

If XrEditorGroup() returns NULL, the *xrErrno* global will be set to one of the following values.

xrErrno will be set to **XrINVALIDID** for all of the messages if the group identified by the *groupInstance* parameter is invalid.

xrErrno will be set to **XrINVALIDMSG** for any messages besides the messages listed above.

xrErrno will be set to **XrINVALIDPARM** whenever the *data* parameter contains invalid information.

xrErrno will be set to **XrOUTOFMEM** if MSG_NEW fails because it cannot allocate memory for the new group.

xrErrno will be set to **XrINVALIDRECT** if the rectangle specified in the *data* parameter for the MSG_SETGROUPRECT is either to small to contain the set of editors in the group or extends beyond the bounds of the window in which the group resides.

ORIGIN

HP

SEE ALSO

XrInput(3X), XrEditor(3X), xrErrno(3X)

NAME

xrErrno - error indicator for Xrlib toolbox calls

SYNOPSIS

```
#include <Xr11/Xrlib.h>
extern INT32 xrErrno
```

DESCRIPTION

xrErrno is an external variable whose value is set whenever an error occurs within an Xrlib toolbox procedure. This value can be used to obtain a more detailed description as to why a toolbox request failed. Most Xrlib toolbox procedures will return either a **NULL** value, or a value of zero, when they fail. The individual descriptions for each of the toolbox facilities specify the details. *xrErrno* is not cleared on successful toolbox calls, so its value should be checked only when an error has been indicated.

Each toolbox procedure's manual page attempts to list all possible error numbers which could occur as a result of issuing that request. The following is a complete list of the error names. The numeric values can be found in **<Xr11/defs.h>** but should not be used.

XrOUTOFMEM Out of memory
: This error is normally returned as a result of the called function being unable to allocate enough memory to perform the specified task.

XrINVALIDID Invalid Id
: This error implies that an invalid Id (window Id, font Id, editor instance pointer, etc) was specified.

XrINVALIDOPTION Invalid option
: This error will occur if a parameter is set to a value which is not among a fixed set of supported values for that particular parameter.

XrINVALIDRECT Invalid rectangle specification
: This error will normally occur when a function is passed a rectangle definition, which for one reason or another is invalid.

XrINVALIDPTR Invalid (NULL) pointer specified
: This error will occur when a required pointer parameter has been set to NULL, instead of to a valid pointer value.

XrPARMOUTOFRANGE Parameter is out of range
: This error will normally occur when a parameter has been set to value which does not fall within the range of values allowed for that parameter.

XrINVALIDDEPTH Invalid raster depth
: This error will occur when an invalid raster depth is specified.

XrINVALIDMSG Invalid message
: This error will occur when an unknown message is passed to one of the Xrlib toolbox functions.

XrXCALLFAILED A call to the 'X' server failed
: This error may occur any time one of the Xrlib functions issues a call to the 'X' server, which subsequently fails for some reason.

XrINVALIDEVENT Invalid event
: This error will occur when an event be used by one of the Xrlib toolbox functions is invalid or not the correct type.

XrFILEERROR File error
: This error will occur when an Xrlib function fails when it attempts to access a disc file.

XrINVALIDPARM Invalid parameter
: This error serves as a catch all for all other error condition.

XrINVALIDTYPE Invalid type
: This error is used by the resource manager when an invalid resource type is referenced.

XrINVALIDSTATE Invalid state
: This error is used by the resource manager to indicate that a state setting has prevented a message from succeeding.

ORIGIN

Hewlett-Packard

NAME

XrGlobals - the globally accessible data structure descriptions.

SYNOPSIS

#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

DESCRIPTION

Xrlib initializes the following set of globals variables which are used by Xrlib functions and can also be used by the application. None of these variables should be reset by the application or grave errors could result.

BlackPixmap	-	A pixmap composed of all black pixels.
WhitePixamp	-	A pixmap composed of all white pixels.
xrBaseFontInfo	-	Xrlib base font.
xrCalloc	-	The pointer to the calloc() function.
xrErrno	-	Error code set when a function fails.
xrFree	-	The pointer to the free() function.
xrMalloc	-	The pointer to the malloc() function.
xrRealloc	-	The pointer to the realloc() function.
xrZeroPt	-	A POINT structure set to {0, 0}.
xrZeroRect	-	A RECTANGLE structure set to {0, 0, 0, 0}.
xrWindowForeground	-	Xrlib default window foreground pixmap.
xrWindowBackground	-	Xrlib default window background pixmap.
xrBorderWidth	-	Xrlib default window border width.
xrForegroundColor	-	Xrlib default foreground color.
xrBackgroundColor	-	Xrlib default background color.
xrDefaultCursor	-	Xrlib default cursor.
_xrCurrentDisplay	-	A pointer to the current display.
_xrCurrentScreen	-	The index of the current screen.

Xrlib provides one default and eight global graphics context structures. **xrDefaultGC** is the default graphics context used by all Xrlib field editors, and is available for examination by applications. This structure is filled out by **XrInit()** and must *never* be written over! Global graphics contexts **xrEditorGC1** through **xrEditorGC8** are used by the Xrlib field editors and may also be used by applications. For more information on graphics context structures, see the 'Xlib C Language Interface - Protocol Version 11' document.

Xrlib also defines an error variable, *xrErrno* which is set to an error code by Xrlib functions when the functions fail.

ORIGIN

HP

SEE ALSO

XrInit(3X), xrErrno(3X)

NAME

XrGroupBox - an editor used to draw a containment box

SYNOPSIS

```
#include <Xr11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrGroupBox (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The groupbox editor provides the application with the means for displaying a box with an optional label in the upper left corner. This is most often used to enclose a group of related editor instances, thus indicating that the editors are closely related to one another.

A groupbox editor instance is currently unique within the X-ray field editor domain: it is the only field editor which does not have a selectable region (i.e. a hot spot) associated with it. This allows the groupbox to overlap the editor rectangles associated with the editor instances it contains, without producing any unfriendly side effects.

When creating a groupbox editor instance, the application will normally determine the region which is to be enclosed (if enclosing a group of editor instances, then this may be done by taking the union of all of the editor rectangles), then adding some padding and space for the box border. The following formulas may prove helpful when doing this, since they provide a general mechanism for taking any padding and the border width into consideration:

```
x = x - (borderwidth + horizontalPadding)
y = y - (MAX (borderWidth, fontHeight) + verticalPadding)
width = width + (2 * (borderWidth + horizontalPadding))
height = height + (2 * verticalPadding) + borderWidth +
        MAX (borderWidth, fontHeight)
```

NOTE: MAX() is a simple macro, which returns the larger of the two values passed in.
It is defined as follows:

```
#define MAX(a,b)   ((a > b) ? a : b)
```

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a group box editor instance in a window. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window       editorWindowId;
   RECTANGLE    editorRect;
   INT8         editorState;
   INT32        editorFGColor;
   INT32        editorBGColor;
```

```
    XFontStruct  * editorFont;
    INT8         * label;
    INT32        borderWidth;
  } xrGroupBoxInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This field describes the location and size of the region in which the groupbox editor is to be located. It does not describe the region which is to be enclosed by the groupbox. This value must take into account the border width and any padding. If the formulas discussed in the introduction are used, then this is automatically taken care of. If the groupbox is not large enough to display the complete label string, then the string will be clipped until it fits. A rectangle with a height or width of 0 is invalid, and if the rectangle is not large enough to contain an empty box of the specified border width, the create request will fail.

editorState

This field contains the initial value of the state flags for this editor instance. Because the groupbox editor has no hot spot the only state flag supported is **XrVISIBLE.** The **XrSENSITIVE** flag is ignored.

editorFGColor

This field specifies the foreground color to be used when drawing the editor instance. This is uses to draw the group box border, and the box label. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor

This field specifies the background color to be used when drawing the editor instance. This is used to fill the background area for the box label. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont

This is a pointer to a structure which describes the font to be used when creating this editor instance. If the pointer has been set to NULL, then the editor will use the default system base font.

label

This is a pointer to the text label (8-bit characters) which will be displayed in the upper left corner of the groupbox. If this pointer is set to NULL, or points to an empty string, then a label will not be displayed.

borderWidth

This value indicates the line width, in pixels, which should be used when drawing the groupbox border. If a border width less than or equal to zero is specified, then no borders are drawn.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure returned by MSG_NEW; this parameter specifies which instance is to be destroyed.

When a group box instance is destroyed, it will be internally disconnected from the window to which it was attached, and will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE** flag for the specified groupbox editor instance. The *instance* parameter specifies which instance to query. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrVISIBLE** flag, for a given groupbox editor instance. The *instance* parameter specifies which instance to modify. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flags. After saving the new state flags, the editor instance will be redrawn to reflect the new state. If an instance is not visible, then the groupbox outline will be drawn using the background tile for the window, making it invisible.

MSG_REDRAW

This editor provides an application with a means to redraw a groupbox editor instance. The *instance* parameter specifies which instance to redraw. The *data* parameter must be a 32 bit integer value, which specifies the type of redraw to be performed.

The only redraw mode supported by this editor is:

- **XrREDRAW_ALL**

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a groupbox editor instance, based upon the label to be displayed and the width of the borders. It returns the definition for the smallest rectangle needed to describe the specified instance; the application may modify this definition later, if desired, but if the rectangle size is decreased, then the label will be clipped when the instance is created.

The groupbox editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   XFontStruct * editorFont;
   INT8        * label;
   INT32       borderWidth;
} xrGroupBoxInfo;
```

The fields which must be filled out by the application program BEFORE issuing this call, are the *label*, *editorFont* and *borderWidth* fields; all other fields are ignored.

Using the supplied values, the editor will determine the size of the smallest rectangle needed to contain this instance. In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance; an application program can then offset or enlarge this rectangle, to position and size it however it likes, within it's window.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a RECTANGLE structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information. This occurs only if the instance is visible.

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the groupbox editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is an invalid size [XrINVALIDRECT].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW
The *data* parameter has been set to an unknown redraw mode [XrINVALIDOPTION].

MSG_SIZE
data is set to NULL [XrINVALIDPTR].

MSG_RESIZE
data is set to NULL [XrINVALIDPTR].

editorRect is an invalid size [XrINVALIDRECT].

MSG_GETSTATE and MSG_MOVE
data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE
The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN
HP

SEE ALSO
XrInput(3X), XrInit(3X)

NAME

XrInit - the Xrlib initialization function.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
```

XrInit(displayPtr, screen, allocFuncts)

```
Display       * displayPtr;
INT32           screen;
xrAllocFuncts * allocFuncts;
```

DESCRIPTION

The XrInit() routine sets up Xrlib for use by an application by initializing internal data and adding a set of resources that the Xrlib routines need. It must be called before any Xrlib functionality is used and can only be called once.

The *displayPtr* parameter is a display pointer as returned by the XOpenDisplay() function. *screen* parameter is the default screen. This can be obtained by using the DefaultScreen() macro with the *displayPtr* value as an argument. The *xrAllocFuncts* parameter is a pointer to the following structure.

```
typedef struct
{
  char * (*newMalloc)();
  int   (*newFree)();
  char * (*newRealloc)();
  char * (*newCalloc)();
} xrAllocFuncts;
```

This structure contains the set of allocation functions that Xrlib routines should use for storage allocation. This set of function pointers only needs to be specified by application which want to provide their own storage management. All of the pointers must be supplied if any of the allocation functions are to be changed. The parameter should be set to NULL for applications which only want to use the default system allocation functions.

The set of function pointer contained within this structure are copied into a set of Xrlib globals variables. They are called *xrMalloc*, *xrFree*, *xrRealloc*, and *xrCalloc*. These variables are used by Xrlib functions which need storage allocation and can also be used by the application.

XrInit() also initializes the resource list with a set of resources. This includes resource types of **XrTYPE_BITMAP**, and **XrTYPE_BITMAPID**, Refer to the programmer's manual for specifics about the set of resources set up by XrInit().

RETURN VALUE

XrInit() returns **TRUE** when it is successful, **FALSE** otherwise.

ERROR CONDITIONS

If XrInit() returns **FALSE** the *xrErrno* global will be set to one of the following values.

xrErrno to be set to **XrINVALIDPARM** under the following conditions.

XrInit() is called more than once.

ORIGIN

HP

SEE ALSO

XrGlobals(3X), XrResource(3X)

NAME

XrInput - the input handling function.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

XrInput (windowId, message, data)
Window windowId;
INT32 message;
INT8 * data;
```

DESCRIPTION

XrInput() handles the requests for input by an application from a window or windows and the setting of window and input attributes necessary to handle editors and panel managers.

Most of the messages will use the xrEvent structure which is defined as follows.

```
typedef struct
{
  UINT32        type;
  unsigned long serial
  Bool          send_event
  Display       * display;
  INT32         source;
  INT16         inputCode;
  INT8          inputType;
  INT8          value1;
  INT16         value2;
  INT16         value3;
  POINT         valuePt;
  INT32         valuePtr;
} xrEvent;
```

When input occurs from a window, the *type* field of the structure will be set to one of X's input types or to the define **XrXRAY** for Xrlib input types. Refer to the X documentation for a complete description of X input. An application can generate Xrlib input and use editors, panels, and menus, if the window is registered with XrInput. The is accomplished through the MSG_ADDWINDOW message listed below. If the *type* field of the xrEvent structure is set to **XrXRAY**, the *serial* field will contain the serial number of the last request processed by the server. *send_event* field will contain a non-zero (TRUE) value if the event is the result of a SendEvent request, the *display* field will contain a pointer to the current display, the *source* field will contain the id of the window in which the input occurred, and the *inputType* field will be set to one of the defines listed in the table below.

Type	Meaning
XrEDITOR	Input from an editor
XrPANEL	Input from a panel
XrMENU	Input from a menu
XrMESSAGEBOX	Input from a message box
XrFILE	Input from a file descriptor
XrTIMEOUT	Input timed out

An application can also gather input from any file descriptor that has been opened. Examples of this would be input from a pty or socket. If this type of input is pending, the *source* field in the xrEvent structure will be set to the file descriptor and the *inputType* field will be set to **XrFILE.** It is then up to the application to get the input from the file descriptor.

The messages which XrInput() provides are broken into several functional groupings.

MSG_ADDWINDOW

If an application wants to gather Xrlib type input or place editors within a window it has created, Xrlib needs to be told about the window. Issuing this message causes a window to be registered with XrInput(), which then sets up the other routines which need to know about the window. To register a window, *windowId* should be set to the id of the window to be registered and *data* needs to point to an xrWindowData structure which is defined as follows:

```
typedef struct
{
  RECTANGLE windowRect;
  Pixmap    foreTile;
  Pixmap    backTile;
} xrWindowData;
```

MSG_REMOVEWINDOW

When an application destroys a window, it should call XrInput() so that the window will be removed from XrInput()'s tables. For this message, *windowId* should be set to the id of the window to be removed and *data* is unused and can be set to **NULL.**

MSG_SETWINDOWDATA

If an application changes the size or tiles of a window, XrInput() needs to be given the data to modify its tables. For this message, *windowId* should be set to the id of the window in which the data is to be set and *data* should be set to point to an xrWindowData structure which contains the needed information.

MSG_GETWINDOWDATA

This messages returns the data for a window. For this call, *windowId* should be set to the id of the window in which the data is contained and *data* is a pointer to an xrWindowData structure. The members of this structure will be set to the data values for the window.

MSG_ADDWINDOWFUNCT

Xrlib high-level manager functions have the ability to be called automatically upon a particular input or inputs. An example of this is the menu manager displaying a menu upon a menu event input. To accomplish this function, XrInput() maintains a set of information for each registered window. This message is the means by which a particular window gets the capabilities described above. For this message, *data* is a pointer to the following structure:

```
typedef struct
{
  INT8  processFlag;
  xrPFI (*funct)();
  INT32 instance;
  INT32 message;
  INT32 eventCount;
  xrWindowEvent * eventList;
} xrWindowFunct;
```

The *funct* field contains a pointer to the function that is to be invoked when a particular event occurs. The function must be of the form:

```
(*funct)(instance, message, data)
INT32 instance;
int message;
INT8 * data;
```

instance is the instance of the type of element *funct* is to operate upon. *message* is the message that is to be sent to the function. *data* will be a pointer to the event which occurred to invoke this function.

The rest of the parameters contained within the xrWindowFunct structure are defined as follows. *processFlag* is a boolean which is set to **TRUE** or **FALSE** by the application or manager adding the xrWindowFunct structure. It is used to turn on and off the processing of the event list and the automatic calling of the function attached to the window. *eventCount* contains a count of that number of events. *eventList* contains a pointer to an array of window event structures which are defined as follows:

```
typedef struct
{
  INT32  type;
  INT32  modifier;
  INT32  code;
} xrWindowEvent;
```

The *type* field of this structure can be set to any one of the X event types. The *modifier* and *code* fields are used for button or key events to further distinguish the function. Each window can have as many xrWindowFunct structures attached to it as is needed. When input occurs on the window the event lists contained within the xrWindowFunct structures that are attached to the window will be searched to see if a match can be found. If the event matches one of the events listed for the window the function for the event is invoked. If no match is found, the event is returned as normal input.

MSG_REMOVEWINDOWFUNCT
: This message removes an xrWindowFunct structure that was previously added to the window *windowId*. *data* is a pointer to the function that matches the function supplied in the xrWindowFunct structure contained within the window.

MSG_SETPROCESSFLAG
: A manager of a window may sometimes want to turn on and off the event list checking and function calling provided through MSG_ADDWINDOWFUNCT. This message sets the *processFlag* field of the xrWindowFunct structure to **TRUE** which turns on the processing function for the window identified by the parameter *windowId*. The *data* field is a pointer to the function whose *processFlag* is to be set.

MSG_CLEARPROCESSFLAG
: This message is used to turn off a processing function for a window. Issuing this message causes *processFlag* to be set to **FALSE** which turns off the processing function for the window identified by the parameter *windowId*. The *data* value is a pointer to the function whose *processFlag* is to be cleared.

The following set of messages provide for the different forms of gathering input from Xrlib.

Xrlib supports both blocked and non-blocked input. Blocked input will cause the input routine to wait for input to occur on one of the windows or the file descriptors before returning. Blocked input will also return before input occurs is the application has supplied a timeout value. Non-blocked input will return immediately with either input or an indicator that no input has occurred.

Two modes of input are available within Xrlib. They are regular reads and hot reads. The difference between the two types of read involve the activation of the event list processing of the events attached to the window. For hot reads, when input occurs, the set of events attached to the window is scanned. If an event is found which matches the input, the corresponding function is called. This is the means by which all of the editors and the menu system is invoked.

MSG_BLKREAD
: This message copies an input event into the XEvent structure pointed at by *data* if there is input pending. If there is no input, the call will wait until input occurs and return the input.

MSG_NONBLKREAD
: This message copies an input event into the XEvent structure pointed at by *data* if there is input pending. If there is no input, *data* is left unchanged and **FALSE** is returned.

MSG_BLKHOTREAD
: This message copies an input event into the XEvent structure pointed at by *data* if there is input pending. If there is no input, the call will wait until input occurs and return the input. If the input is select or menu input, the editors or menuing system will be invoked if appropriate.

MSG_NONBLKHOTREAD
: This message copies an input event into the XEvent structure pointed at by *data* if there is input pending. If there is no input, *data* is left unchanged and **FALSE** is returned. If the input is select or menu input, the editors or menuing system will be invoked if appropriate.

Xrlib provides a set of messages for manipulating the input queue. The following list contains this set of messages.

MSG_PUSHEVENT
: This message places an event structure pointed to by *data* onto the front of the input queue. The next read will return this event.

MSG_PEEKEVENT
: This messages fills out the event structure pointed to by *data* from the event on the front of the input queue. The input queue is left unaffected by this call. If the input queue is empty, this call will fail.

MSG_CLEAR
: This message clears the input queue of all pending input. *data* is unused for this call and can be set to NULL.

The following list contains the set of messages to setup and change which sources XrInput() is to gather input.

MSG_ADDINPUT
: This message adds a file descriptor for which XrInput() is to gather input. *data* is a pointer to an xrFDInput structure which is defined as follows.

```
typedef struct
{
  INT32 fd;
  INT8 type;
} xrFDInput;
```

The *fd* field contains the file descriptor for the source of the input in which XrInput() is to gather input. The *type* field tells XrInput() which forms of input should be gathered from the file descriptor. It can be set to any combination of the defines **XrREAD**, **XrWRITE**, or **XrEXCEPTION.** These defines should be "Or'ed" together to define the value of this field.

MSG_REMOVEINPUT
: This message removes a previously added file descriptor from the input set. *data* must be set to point to an xrFDInput structure discussed in MSG_ADDINPUT. The structure should contain the file descriptor and the conditions which are to be removed.

MSG_SETTIMEOUT
: This message sets a timeout value, in seconds, for XrInput() to wait for input to occur from the file descriptors it is selecting upon. *data* is a pointer to the following structure.

```
struct timeval
{
  unsigned long tv_sec;
  long         tv_usec;
};
```

The *tv_sec* field defines the number of seconds to wait before a timeout is to occur. The *tv_usec* field defines the number of micro seconds to wait before a timeout is to occur. This structure is defined in <time.h>.

The timeout value will affect how blocked reads work. Normally, the blocked read will wait indefinitely for input. If an application sets a timeout value, the blocked read will wait only the number of seconds + the number of micro seconds given by the value. The value of the timeout will have no effect on non-blocked reads. When a blocked read times out, an event of *type*: **XrXRAY** with an *inputType*: **XrTIMEOUT** will be returned.

MSG_GETTIMEOUT
: This message returns the current timeout value. *data* is a pointer a timeval structure which will be filled out to the current values.

RETURNED VALUE

For all of XrInput()'s messages, **TRUE** is returned if the function succeeds and **FALSE** is returned if it fails.

ERROR CONDITIONS

If XrInput() fails, the *xrErrno* global will be set to one of the following values:

XrINVALIDID for all messages if the passed in window id is invalid.

XrINVALIDMSG for any messages besides the messages listed above.

XrINVALIDPARM whenever the data parameter contains invalid information.

XrOUTOFMEM if MSG_ADDWINDOW cannot allocate the needed space for a window.

For non-blocking messages, a return value of **FALSE** indicates that their was no input.

ORIGIN

HP

NAME

XrInputConf - configure the input mapping

SYNOPSIS

```
#include <Xr11/keycode.h>

int
XrInputConf (param, value)
int param;
int value;
```

DESCRIPTION

This routine changes the configuration of the input mapping provided by **XrInputMap(3X)**. **param** indicates which configuration parameter is to be changed or queried, while **value** contains the value to which the parameter is to be set to, or, if set to -1, indicates that the value is being queried.

param can be set to one of the following values:

K_TYPE
: If **value** is -1, the current keyboard type is returned. If **value** is K_HP or K_DEFAULT, the keyboard type will be changed.

K_LANGUAGE
: If **value** is -1, the current keyboard language is returned. If **value** is a legal keyboard language (K_HP_USASCII to K_HP_JAPANESE), the keyboard language is changed. If **value** is not a legal language, the language will be reset to the power-on language as determined by **XrInputInit(3X)**.

K_ASCII8
: If **value** is -1, the current state is returned. If **value** is 1, 8 bit mode is in effect. If **value** is 0, ISO 7 bit mode is in effect.

K_MUTE
: If **value** is -1, the current state is returned. If **value** is 1, muting is in effect. If **value** is 0, muting is in turned off. When muting is enabled, the diacriticals are collapsed with the characters they are modifying and the new code is returned. For example, if R_TILTA (tilde accent) is received, it is saved and no code is returned. Then an N is received. The code R_N_TILTA is returned. Muting will not be done if K_ASCII8 is not set regardless of the value of K_MUTE.

K_16BITIO_ENABLE
: If **value** is -1, the current state is returned. If **value** is 1, the right extend char key will cause K_16BITIO to be set and the left extend char key will cause K_16BITIO to be cleared. If **value** is 2, the left extend char key will cause K_16BITIO to be set and the right extend char key will cause K_16BITIO to be cleared. If **value** is 3, the right extend char key will cause K_16BITIO to be toggled. If **value** is 4, the left extend char key will cause K_16BITIO to be toggled. If K_16BITIO is set, 16-bit input is enabled. If **value** is 0, 16-bit input is disabled and K_16BITIO will not be set. The default value is 4 for K_HP_JAPANESE. The default value is 0 for any other current language.

K_ALT_ENABLE
: If **value** is -1, the current state is returned. If **value** is 1, the right extend char key will cause K_ALT to be set and the left extend char key will cause K_ALT to be cleared. If **value** is 2, the left extend char key will cause K_ALT to be set and the right extend char key will cause K_ALT to be cleared. If **value** is 3, the right extend char key will cause K_ALT to be toggled. If **value** is 4, the left extend char key will cause K_ALT to be toggled. If K_ALT is set, an alternate language-dependent key mapping will be used. If **value** is 0, the extend char keys will

behave like shift keys (i.e. the alternate map will be used when the extend char key is down at the same time as the key being mapped. The meta modifier is set when the extend char keys are pressed). The default value for K_HP_KATAKANA is 1. The default value for K_HP_JAPANESE is 3. The default value for any other current language is 0.

K_ALT

K_16BITIO

These flags are read-only. See K_ALT_ENABLE and K_16BITIO_ENABLE for descriptions. Alternate keyboard maps will not be used if K_ASCII8 is not set regardless of the value of K_ALT or the meta modifier.

ORIGIN

HP

SEE ALSO

XrInputInit(3X),XrInputMap(3X),XrInputMute(3X)

NAME

XrInputInit - initialize the input mapping

SYNOPSIS

#include <Xr11/keycode.h>

unsigned char *
XrInputInit()

DESCRIPTION

This routine is called by **XrInit(3X)**, and *must* never called directly by an application. **XrInit**.

This call initializes input structures which support the input mapping routines **XrInputConf()**, **XrInput-Map()** and **XrInputMute()**. These routines provide more general mapping services than **XLookup-String()** and take into account the differences between DEC, SUN, and HP keyboards, including the various international versions of the HP keyboard. **XrInputMap()** performs the mapping. In special cases, **XrInputConf** and **XrInputMute** are used to control the mapping mode , or to access ISO 7-bit mapping directly on a character by character basis.

Memory is allocated to hold the input structures. The type and language of the keyboard is determined by the display type field for the current display and the tables are set up for that language. If the key-board is a HP keyboard, the keyboard type is set to K_HP and the language is determined. If it is not an HP keyboard, the keyboard type is set to K_DEFAULT and the language is set to K_HP_USASCII. A pointer to the ISO 7 bit to Roman 8 bit conversion table for that language is returned. This table should be used to map your output if K_ASCII8 is not set (in 7 bit mode). K_ASCII8 is set (1) on initialization. K_MUTE, K_ALT_ENABLE and K_16BITIO_ENABLE are set as appropriate for the language. K_ALT and K_16BITIO are read-only flags which are initially clear.

ORIGIN

HP

SEE ALSO

XrInputConf(3X), XrInputMap(3X), XrInputMute(3X)

NAME

XrInputMap - map the input

SYNOPSIS

```
#include <Xr11/keycode.h>

unsigned char *
XrInputMap(kptr, nbytes)
XKeyEvent *kptr;
int * nbytes;
```

DESCRIPTION

This call maps the keycode returned from X into 8 bit character codes, 16 bit character codes and function codes, taking into account the language of the keyboard if the keyboard type is K_HP (a HP keyboard). If the keyboard type is K_DEFAULT or if the user has a .Xkeymap in his home directory, **XLookupString(3X)** will first be used. If **XLookupString()** returns one or more characters, that string will be returned. If **XLookupString()** returns either no characters, it will be ignored and the key will be mapped as follows. If the key is a character key, the appropriate table is used based on the modifiers and the language of the keyboard. Muting will be done if appropriate for the language. Conversion from 8 bit to ISO 7 bit will be done if the K_ASCII8 is not set. If the key is a function key, two bytes will be returned. The first byte will be 0x80 and the second byte will be the function code. If the key has a different function when shifted, a different code will be returned. **Xr11/keycode.h** contains the defines for the function codes. The following table lists codes that will be generated for the various function keys.

The **kptr** parameter is a pointer to an XKeyEvent structure returned by issuing a read request to either the Xlib or Xrlib input routines.

The **nbytes** parameter is a pointer to an integer. It contains the number of bytes in the string which is returned. A pointer to a static counted character string which must not be touched by a client is returned by **XrInputMap()**. If **nbytes** contains a 1, the string consists of one 8 bit character. If **nbytes** contains a 2 and the first byte is 0x80, then the second byte is a function code. If **nbytes** contains a 2 and the first byte is not 0x80 or ***nbytes** is larger than 2, a string of characters is returned.

LK201 Code	Return Code	Shifted Code	HP key label	DEC key label
0126	F1_KEY	F9_KEY	f1	F1
0127	F2_KEY	F10_KEY	f2	F2
0130	F3_KEY	F11_KEY	f3	F3
0131	F4_KEY	F12_KEY	f4	F4
0132	F5_KEY	F13_KEY	f5	F5
0133	BREAK_KEY	RESET_KEY	Reset/Break	***
0134	STOP_KEY	SH_STOP_KEY	Stop	***
0135	MENU_KEY	MENU_KEY	Menu	**
0136	SYS_KEY	USER_KEY	User/System	**
0137	ENTER_KEY	PRINT_KEY	Print/Enter	***
0140	CLR_LINE_KEY	CLR_ALL_LINE_KEY	Clear line	**
0141	DEL_LINE_KEY	DEL_LINE_KEY	Delete line	**
0142	INSERT_LINE_KEY	INSERT_LINE_KEY	Insert line	**
0143	CLR_DISP_KEY	CLR_ALL_DISP_KEY	Clear display	**
0144	F6_KEY	F14_KEY	f6	F6
0145	F7_KEY	F15_KEY	f7	F7

0146	F8_KEY	F16_KEY	f8	F8
0147	MENU_KEY	MENU_KEY	**	F9
0150	SYS_KEY	USER_KEY	**	F10
0161	'\033'	'\033'	***	F11
0162	BS_KEY	BS_KEY	**	F12
0163	'\n'	'\n'	***	F13
0164	HOME_UP_KEY	HOME_DN_KEY	**	F14
0174	HELP_KEY	HELP_KEY	***	Help
0175	DO_KEY	DO_KEY	***	Do
0200	CLR_LINE_KEY	CLR_ALL_LINE_KEY	**	F17
0201	CLR_DISP_KEY	CLR_ALL_DISP_KEY	**	F18
0202	INSERT_LINE_KEY	INSERT_LINE_KEY	**	F19
0203	DEL_LINE_KEY	DEL_LINE_KEY	**	F20
0204	BS_KEY	BS_KEY	Back space	**
0210	TAB_KEY	BACKTAB_KEY	Tab (keypad)	**
0211	'\033'	'\177'	DEL/ESC	***
0212	FIND_KEY	FIND_KEY	***	Find
0213	INSERT_CHAR_KEY	INSERT_CHAR_KEY	Insert char	Insert Here
0214	DEL_CHAR_KEY	DEL_CHAR_KEY	Delete char	Remove
0215	SELECT_KEY	SH_SELECT_KEY	Select	Select
0216	PREV_KEY	PREV_KEY	Prev	Prev Screen
0217	NEXT_KEY	NEXT_KEY	Next	Next Screen
0225	ENTER_KEY	ENTER_KEY	Enter	Enter
0241	BLANK_1_KEY	BLANK_5_KEY	(Blank 1)	PF1
0242	BLANK_2_KEY	BLANK_6_KEY	(Blank 2)	PF2
0243	BLANK_3_KEY	BLANK_7_KEY	(Blank 3)	PF3
0244	BLANK_4_KEY	BLANK_8_KEY	(Blank 4)	PF4
0247	CURS_LF_KEY	ROLL_LT_KEY	(left arrow)	(left arrow)
0250	CURS_RT_KEY	ROLL_RT_KEY	(right arrow)	(right arrow)
0251	CURS_DN_KEY	ROLL_DN_KEY	(down arrow)	(down arrow)
0252	CURS_UP_KEY	ROLL_UP_KEY	(up arrow)	(up arrow)
0253	HOME_UP_KEY	HOME_DN_KEY	(up-left arrow)	**
0260	CAPS_KEY	CAPS_KEY	Caps	Lock
*0261	TO_ROMAN_KEY	TO_ROMAN_KEY	Ext char(left)	Compose Char
*0262	TO_ALT_KEY	TO_ALT_KEY	Ext char(right)	***
*0263	KANJI_LF_KEY	KANJI_LF_KEY	(left Kanji)	***
*0264	KANJI_RT_KEY	KANJI_RT_KEY	(right Kanji)	***
0274	'\177'	'\177'	***	(backspace)
0275	RETURN_KEY('\r')	RETURN_KEY('\r')	Return	Return
0276	TAB_KEY('\t')	BACKTAB_KEY('\t')	Tab	Tab

The Return and Tab keys will return the characters in the parentheses from non-HP keyboards since **XLookupString()** will return those values.

* The Extend char keys generate these codes only if K_ALT_ENABLE is 1. See **XrInputConf()** for alternate setting of K_ALT_ENABLE. When K_ALT is set, TO_ALT_KEY is generated and when it is cleared, TO_ROMAN_KEY is generated. If K_ALT_ENABLE is 0, no code will be generated and **nbytes** will be set to zero.

** This function is generated by different keys (LK201 codes) on the HP keyboard and the DEC keyboard.

*** This function exists only on one of the keyboards, either HP only or DEC only. Avoid using these functions if you wish your application to run on both machines.

ORIGIN

HP

SEE ALSO

XrInputConf(3X), XrInputInit(3X), XrInputMute(3X)

NAME

XrInputMute - map the input

SYNOPSIS

```
#include <Xr11/keycode.h>

unsigned char
XrInputMute(c)
unsigned char c;
```

DESCRIPTION

Normally, **XrInputMap()** would be used to map the keys. It calls **XrInputMute()** as part of its mapping process. However, if you have a character for which you would like the 8 bit to ISO 7 bit conversion done or the muting done, this routine can be used. The c parameter represents the character which is to be converted.

ORIGIN

HP

SEE ALSO

XrInputConf(3X), XrInputInit(3X), XrInputMap(3X)

NAME

XrListEdit - an editor used to display a scrollable list of strings

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrListEdit (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The list editor provides the application with the means for displaying a set of static strings. These strings are displayed in a single column, with an optional titlebar displayed above the strings, and an optional scrollbar displayed to the right of the strings.

When a list editor instance is created, the application can specify the number of strings which are to be initially contained within the list, along with a count specifying how many of the list items are to be displayed at any given time.

Once a list editor instance has been created, the application is provided with the means for adding new items to the list, replacing existing items or deleting items. In addition, the application may also query the state of an item, modify the state of an item, or obtain a copy of a list item string.

Using the cursor and the mouse, a user will be able to select a single string within the list in one of several ways:

> By 'clicking' the mouse over a list item, the item over which the cursor is located will be highlighted with some visible feedback to indicate the state has changed to 'selected'; any previously highlighted item will be automatically unhighlighted.
>
> By pressing the mouse button, and then dragging the cursor, the list editor will track the cursor, highlighting the item over which the cursor is currently placed; releasing the mouse button will cause the tracking to stop, as will the receipt of any other X event.

At the time the list editor instance is created, the application may specify whether a scrollbar should be included as part of the instance. If the application creates a list editor instance without a scrollbar, it should be careful not to supply more strings than can be displayed, because the user will have no way of scrolling the list. An application would most often use this option when it wants to display a small number of list items, and it does not want the editor rectangle to include the extra space required by the scrollbar.

If specified at creation time, the list editor will display a vertical scrollbar to the right of the list of strings. If the list contains fewer items than can be displayed within the list, then the scrollbar will be displayed as insensitive, and the list will not scroll. If the list contains more items than can be displayed within the list, then the scrollbar will be displayed as sensitive, and the user may move through the list. The list editor will take sole responsibility for processing the scrollbar; however, any time the list is scrolled, an event will be returned to the application program, informing it of the new scroll position.

This scrollbar provides the means by which a user can scroll through the complete set of list items. There are currently 4 ways in which a scroll operation may be requested:

- The 'UP' arrow on the scrollbar is selected.

- The 'DOWN' arrow on the scrollbar is selected.
- A position within the scroll area is selected.
- The slide box is interactively moved.

Depending upon which of the above methods is used, a different amount of scrolling may take place:

When either of the scroll arrows is selected, the list items will move 1 item in the requested direction.

When a select occurs within the scroll area, the requested portion of the list will then be displayed.

When the slide box is selected and interactively moved by the user, the list contents will scroll, in realtime, thus providing the user with an immediate view of the portion of the list that will be visible.

For each of the above events, the list editor will return an input event, informing the application that the list has been scrolled, along with the new scroll position.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a list editor instance in a window. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
    Window        editorWindowId;
    RECTANGLE     editorRect;
    INT8          editorState;
    INT32         editorFGColor;
    INT32         editorBGColor;
    XFontStruct * editorFont;
    INT8        * titleString;
    XFontStruct * titleFont;
    INT16         stringCount;
    INT8       ** items;
    INT8        * selectionStates;
    INT32         maxStringWidth;
    INT8          cellWidth;
    INT16         viewCount;
    INT16         scrollPosition;
    INT8          scrollableFlag;
} xrListEditInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the region into which the list editor is to be located. This must include space for the string items, the scrollbar (if wanted), and the titlebar (if wanted). The application writer is encouraged to use the MSG_SIZE command to obtain this rectangle, since the create request will fail if the rectangle is not the exact size required to contain the specified instance.

editorState

This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor

This field specifies the foreground color to be used when drawing the editor instance. The foreground color is used to draw the border around the instance, and to draw each of the list items. This color is also used to draw the scrollbar and the optional titlebar. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor

This field specifies the background color to be used when drawing the editor instance. The background color is used to fill the interior of the list area. This color is also used to draw the scrollbar and the optional titlebar. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont

This points to the structure which contains all of the information necessary to describe the font to be used when displaying the list items. If this pointer is NULL, then the default system base font will be used.

titleString

This contains a pointer to the optional title string. If this pointer is set to NULL, then a titlebar will not be displayed at the top of the list. The title string will be displayed using the specified title font.

titleFont

This points to the structure which contains all of the information necessary to describe the font to be used when displaying the title string. If this pointer is NULL, then the default system base font will be used.

stringCount

This field contains a count indicating the number of items which are to be initially contained within the list editor instance. It describes the number of items which are present in the *items* array, and also the number of entries in the *selectionStates* array.

items

This is a pointer to an array of string pointers. Each of the string pointers points to a single NULL terminated list item. The use of NULL pointers within this array is discouraged. If the list initially contains no string items, then this pointer may be set to NULL.

selectionStates

This points to an array of 8 bit values, each one representing the selection state of one of the initial list items. A list item may be displayed as selected if its value is set to TRUE, and will be displayed as unselected if its value is set to FALSE. Since the list editor only allows a single item to be selected at any time, the first item with a value of TRUE will be highlighted, and all subsequent items will be displayed as unselected, regardless of what their selection state value is. If this pointer is set to NULL, then all of the items will be initially set to the unselected mode. If this pointer is not NULL, then the array MUST contain a value for each of the initial items.

maxStringWidth

This value represents the maximum number of bytes. It is one of the parameters needed to calculate the maximum width (in pixels) of the list area. If *maxStringWidth* is greater than zero, the list area width will be calculated by multiplying the *maxStringWidth* value (the number of bytes) by the *cellWidth* value described below. By setting this value to zero, the editor will search through the list of items, and use the length of the longest string; if the string array pointer has been set to **NULL**, then the create request will fail. Internally, the list area width is stored as a pixel value, so when the longest string is used, it is the pixel width of the longest string which is saved, not the character width.

cellWidth

This value allows the application to specify the other parameter needed to calculate the list area width. If *maxStringWidth* was set to zero, then the *cellWidth* value is not used. Otherwise, the *cellwidth* value specifies the pixel width which is to be multiplied by the maximum character count, thus producing the maximum width of the list area, in pixels. To specify the maximum cell width for the font being used, use the define **XrMAXWIDTH**. For a cell width which is the average size for the font, use the define **XrAVGWIDTH**. If any other value is specified, then it will be interpreted as the cellwidth, in pixels, which is to be used.

viewCount

This value specifies the maximum number of items which will be displayed at any given time; this value must be greater than zero. It is valid for this value to be larger than the *stringCount* value. If *viewCount* is less than *stringCount*, then the scrollbar (if the list had one) will become operational.

scrollPosition

This value indicates the index of the item which should be initially displayed at the top of the list area; zero indicates the first item.

scrollableFlag

This boolean value allows the application to specify whether a scrollbar should be created for the list editor instance. If set to **TRUE**, then a scrollbar will be created.

The editor will then draw the list editor instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure returned by **MSG_NEW**; this parameter specifies which instance is to be destroyed.

When a list editor instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the list editor instance indicated by the *instance* parameter. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given list editor instance. The *instance* parameter specifies which editor instance is to be affected. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible.

MSG_REDRAW

This message provides an application with the means for requesting that a list editor instance be redrawn. When this message is invoked, the *instance* parameter must point to the editor structure for the instance to be redrawn, and the *data* parameter must be a 32 bit integer which specifies the type of redraw which is to occur.

The list editor supports only a single redraw mode:

- **XrREDRAW_ALL.**

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a particular list editor instance. Depending upon the description supplied by the application program, this rectangle may hold only a list editor instance and a scrollbar editor instance, or it may additionally include a titlebar editor instance. The list editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   XFontStruct * editorFont;
   INT8        * titleString;
   XFontStruct * titleFont;
   INT16       stringCount;
   INT8      ** items;
   INT8        * selectionStates;
   INT32       maxStringWidth;
   INT8        cellWidth;
   INT16       viewCount;
   INT16       scrollPosition;
   INT8        scrollableFlag;
} xrListEditInfo;
```

The only fields which must be filled out by the application program BEFORE issuing this call, are the *editorFont*, *titleString*, *titleFont*, *maxStringWidth*, *cellWidth*, *viewCount* and *scrollableFlag* fields in the list editor data structure; all other fields are ignored. There is an exception to this: if *maxStringWidth* is set to zero, then the application must supply the initial set of items in the *items* field.

Using the supplied information, the editor will determine the size of the rectangle needed to contain this instance. In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance; an application program can then offset this rectangle, to position it where ever it likes, within its window.

MSG_POSITION

This message provides an application with the means for positioning the slide box within the scrollbar. In effect, this positions a particular string at the top of the list area. The *instance* parameter must point to the editor structure associated with the instance to be modified, and the *data* parameter must contain an INT16 value, specifying the new slide box position; this value is interpreted as the index of the string which should appear at the top of the list area.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated so that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, and the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location. This occurs only if the instance is visible.

MSG_GETITEMSELECTION

This message provides an application with a means for obtaining the selection state (**TRUE** if item is highlighted, **FALSE** otherwise) of a single list item. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being queried, and the *data* parameter must point to an instance of the following structure:

```
typedef struct {
     INT16 itemNumber;
     INT8  selectionState;
} xrSelectionState;
```

The index of the item being queried should be placed in the *itemNumber* field before the call is made; in return, the *selectionState* field will be set to either **TRUE** or **FALSE**.

If the application is interested in finding out which (if any) of the items is currently selected, then it may set the *itemNumber* field to -1, and the editor will automatically set the *itemNumber* field to the index of the selected item, and will also fill in the *selectionState* field; if there is not an selected item within the list, then the editor will leave the *itemNumber* field set to -1.

MSG_SETITEMSELECTION

This message provides an application with a means for modifying the selection state (**TRUE** if item is to be highlighted, **FALSE** otherwise) of a single list item. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being modified, and the *data* parameter must point to an instance of the following structure:

```
typedef struct {
     INT16 itemNumber;
     INT8  selectionState;
} xrSelectionState;
```

The index of the item being modified must be placed in the *itemNumber* field before the call is made, while the new selection state for the item must be placed in the *selectionState* field; it should be set to either **TRUE** if the item is to be highlighted, or **FALSE** if the item is not to be highlighted. If the index is out of range, then all items are unhighlighted. This provides the application with a quick means for deactivating all items in the list, without having to know the index of the selected item.

Since the list editor only allows a single item to be highlighted at any given time, if the specified item is having its selection state being set to **TRUE**, then any previously highlighted item will be automatically unhighlighted.

MSG_GETITEMCOUNT

This message allows the application to obtain the number of items currently in a list editor instance. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being queried, and the *data* parameter must point to a 32 bit integer value, into which the item count will be placed.

MSG_DELETEITEM

This message allows the application to delete a single item from a list editor instance. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being modified, and the *data* parameter must contain a 16 bit integer value, representing the index of the item being deleted. If the index is out of range, the request will fail.

MSG_REPLACEITEM

This message allows the application to replace a single item in a list editor instance. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being modified, and the *data* parameter must point to an instance of the following structure:

```
typedef struct {
     INT16  itemNumber;
     INT8   selected;
     INT8 * item;
} xrListItemData;
```

The *itemNumber* field must be set to the index of the item to be replaced; if this value is greater than the current item count, then the item will be added to the end of the list. The *selected* field should be set to **TRUE** if the item is to be highlighted, otherwise it should be set to **FALSE**. The *item* field should point to the **NULL** terminated item string.

When the new item is displayed, it will be clipped to fit within the list area, if necessary.

MSG_ADDITEM

This message allows the application to add a single item into a list editor instance; this does not replace an existing list item. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being modified, and the *data* parameter must point to an instance of the following structure:

```
typedef struct {
     INT16  itemNumber;
     INT8   selected;
     INT8 * item;
} xrListItemData;
```

The *itemNumber* field must be set to the index of where the item is to be placed; if this value is greater than the current item count, then the item will be added to the end of the list - if this value is less than zero, then the item will be placed at the beginning of the list. The *selected* field should be set to **TRUE** if the item is to be highlighted, otherwise it should be set to **FALSE.** The *item* field should point to the **NULL** terminated item string.

When the new item is displayed, it will be clipped to fit within the list area, if necessary.

MSG_QUERYITEM

This message provides the application with the means for querying all the information for a particular list item; this includes the selection state and the list item string. When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is being modified, and the *data* parameter must point to an instance of the following structure:

```
typedef struct {
     INT16  itemNumber;
     INT8   selected;
     INT8 * item;
} xrListItemData;
```

The application must set the *itemNumber* field to the index of the item being queried; the editor will fill in the *selected* field with **TRUE** if the item is currently highlighted, otherwise it will be set to **FALSE.** The editor will also allocate some memory to hold the string associated with the list item, and will return a pointer to this string in the *item* field; it is the applications duty to free up the memory, using *xrFree,* used to hold the string. If an application is not interested in the string information, then it would be wiser to use **MSG_GETITEMSELECTION,** since that message does not have the overhead of having to allocate a buffer to hold the string.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a list editor instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which list editor to process, and the *data* parameter pointing to an *XEvent* structure.

The list editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within the list editor, the first thing done is to determine which of the following regions was selected:

- The list region.
- The scrollbar.
- The titlebar.
- The background area.

Depending upon the region selected, a different action is taken:

Scrollbar

The list will be scrolled vertically. Upon completion of the scroll operation, the following event will be pushed onto the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrLISTEDIT
value1	=	XrSCROLLBAR
valuePt	=	new scroll position
valuePtr	=	pointer to instance's editor structure

Titlebar

The following event will be pushed onto the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrLISTEDIT
value1	=	XrTITLEBAR
valuePtr	=	pointer to instance's editor structure

List region

The selected list item will be highlighted, and an input event will be added to the front of the application's input queue, informing it that the list has been modified. If no list item was modified, then the *value2* field will be set to -1. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrLISTEDIT
value1	=	XrSELECTION_CHANGED
value2	=	index of selected item

valuePtr	=	pointer to instance's editor structure

Any other region
This implies that the select occurred within the
instance's editor rectangle,
but not within any of the previously mentioned components.
The following input event will be returned:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrLISTEDIT
value1	=	NULL
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the list editor will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is an invalid size [XrINVALIDRECT].

maxStringWidth < 0 [XrPARMOUTOFRANGE].

maxStringWidth > 0 and *cellWidth* contains an invalid value [XrPARMOUTOFRANGE].

maxStringWidth = 0 and *items* is set to NULL.

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw option other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

maxStringWidth < 0 [XrPARMOUTOFRANGE].

maxStringWidth > 0 and *cellWidth* contains an invalid value [XrPARMOUTOFRANGE].

maxStringWidth = 0 and *items* is set to NULL.

MSG_RESIZE
: *data* is set to NULL [XrINVALIDPTR].
: *editorRect* is an invalid size [XrINVALIDRECT].

MSG_POSITION
: An invalid scroll position is specified [XrINVALIDPARM].

MSG_ADDITEM, MSG_REPLACEITEM
: Not enough memory available [XrOUTOFMEM].

MSG_DELETEITEM
: The item index is out of range [XrPARMOUTOFRANGE].
: The list is currently empty [XrINVALIDPARM].

MSG_REPLACEITEM and MSG_ADDITEM
: *data* is set to NULL [XrINVALIDPTR].
: The string pointer is **NULL [XrINVALIDPTR].**

MSG_GETSTATE, MSG_GETITEMSELECTION, MSG_SETITEMSELECTION

MSG_GETITEMCOUNT, MSG_QUERYITEM and MSG_MOVE
: *data* is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE
: The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrMenu - the Xrlib Menu Manager.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrMenu *
XrMenu (menuInstance, message, data)
xrMenu   * menuInstance;
INT32    message;
INT8     * data;
```

DESCRIPTION

XrMenu is the Xrlib menu manager. This man page gives a brief summary of the menu manager structures, messages, and item language.

STRUCTURES

The following structures are used to communicate with the menu manager. The menu structures shown below are described in the "Programming With The Xrlib User Interface Toolbox" manual.

```
typedef struct
{
  INT8          * menuTitle;
  INT8         ** menuItems;
  INT32           numItems;
  xrPanelContext * menuContext;
  INT16           menuId;
  INT32           menuStyle;
} xrMenuInfo;

typedef struct _xrMenuIndex
{
  struct _xrMenu   * menuInstance;
  INT32             itemIndex;
  INT32             itemData;
} xrMenuIndex;
```

MESSAGES

The following messages allow the programmer to communicate with the Xrlib menu manager.

MSG_NEW

MSG_NEW creates a menu structure. The *menuInstance* parameter should be set to NULL, as it is not used. The *data* parameter is a pointer to a *menuInfo* structure.

MSG_FREE

MSG_FREE destroys a menu, its associated menu editor, and frees all resources allocated by the menu manager. *menuInstance* is a pointer to the menu to be freed. *data* should be set to NULL. Care should be taken not to refer to a menu after it has been freed.

MSG_EDIT

MSG_EDIT posts the currently active menu for a given window. This command will be made transparently to the application by the input model most of the time, but may be called by you if necessary.

MSG_ACTIVATEMENU
: MSG_ACTIVATEMENU makes the given menu the current menu for the given window. A menu may be current for several windows at the same time. When the input model gets a *post menu* input from the user, it will post the current menu for the window the input came from. *data* is a window id.

MSG_DEACTIVATEMENU
: MSG_DEACTIVATEMENU makes the given menu inactive for the given window. *data* is a window id.

MSG_ADDSUBMENU
: MSG_ADDSUBMENU allows you to add one menu to another as a sub-menu. The sub-menu is created in the same way as the parent menu, this message simply allows you to link menus into a tree structure. *menuInstance* is a pointer to the sub-menu. *data* points to an *xrMenuIndex* structure that specifies the location in the menu tree. Set the *menuInstance* pointer to the parent menu instance pointer. Set the *itemIndex* portion of the structure to the item in question, and the *itemData* member to NULL.

MSG_REMOVESUBMENU
: MSG_REMOVESUBMENU removes a link between two menus. Neither of the menus are destroyed, there is simply not a path between them any longer. *menuInstance* is a pointer to the sub-menu. *data* points to an *xrMenuIndex* structure that specifies the location in the menu tree. Set the *menuInstance* pointer to the parent menu instance pointer. Set the *itemIndex* portion of the structure to the item in question, and the *itemData* member to NULL.

MSG_ACTIVATEITEM
: MSG_ACTIVATEITEM allows you to make an item in a menu selectable. Unselectable items are shown greyed and are not selectable by the user. *data* is an integer index to the item to be made active. *menuInstance* is the menu instance in question.

MSG_DEACTIVATEITEM
: MSG_DEACTIVATEITEM allows you to make an item in a menu unselectable. Unselectable items are shown greyed and are not selectable by the user. *data* is an integer index to the item to be made active. *menuInstance* is the menu instance in question.

MSG_SETITEMFUNCTION
: MSG_SETITEMFUNCTION sets a function to be executed when an item is selected by the user. *data* points to an *xrMenuIndex* structure. The *menuInstance* member of this structure points to the menu in question. The *itemIndex* member determines which item is associated with the function, and the *itemData* member contains the pointer to the function.

MSG_SETITEMEVENT
: MSG_SETITEMEVENT sets an event to be returned when a menu item is selected by the user. *data* points to an *xrMenuIndex* structure. The *menuInstance* member of this structure points to the menu in question. The *itemIndex* member determines which item is associated with the event, and the *itemData* member contains the pointer to the event created by the programmer.

ITEM LANGUAGE

The XRMENU item language allows the programmer to send special commands on a per-item basis to the Xrlib menu manager. These commands are described in the following table.

Command	Action
\DA	Item is disabled (unselectable)
\KEc	Item has keyboard equivalent c
\-	Single line (unselectable)
\=	Double line (unselectable)
NOTE: c is mapped into a control character.	

RETURN VALUES

Upon successful completion of any of the messages, a non-NULL value is returned. For the MSG_NEW message, this non-NULL value is the pointer to the newly created menu or panel instance.

If the message request fails, NULL is returned.

ERROR CONDITIONS

Messages to XrMenu will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_FREE

A call to 'X' failed [XrXCALLFAILED].

MSG_EDIT

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_ACTIVATEMENU

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_DEACTIVATEMENU

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_ADDSUBMENU

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REMOVESUBMENU

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_ACTIVATEITEM

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_DEACTIVATEITEM

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_SETITEMFUNCTION

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_SETITEMEVENT

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

ORIGIN

HP

NAME

XrMessageBox - the Xrlib Message Box Manager.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
```

XrMessageBox (msgInfo, message, data)
```
xrMsgBoxInfo msgInfo;
INT32 message;
INT8 *  data;
```

DESCRIPTION

XrMessageBox() is the Xrlib Message Box manager. Many times an applications programmer needs to warn a user or ask a simple multiple choice question. Message boxes provide this capability, and are easy to program.

A message box is made up of an icon, some descriptive text, and buttons. Icon placement, text formatting, and button placement is handled by the message box manager automatically.

STRUCTURES

The message box manager acquires all the information it needs about a message box in a message box information structure. This structure is shown below:

```
typedef struct
{
  POINT          messageOrigin;
  Window         relativeTo;
  xrPanelContext  * messageContext;

  INT32          rasterHeight;
  INT32          rasterWidth;
  Pixmap         rasterId;

  INT8          * messageText;

  INT8         ** messageButtons;
  INT32          numButtons;
} xrMsgBoxInfo;
```

MESSAGES

XrMessageBox() has only two messages: MSG_EDIT, and MSG_SIZE. MSG_SIZE returns the size a message box would be if it were created with the given parameters. MSG_EDIT display a message box window and makes it active.

RETURN VALUE

XrMessageBox() returns **TRUE** if successful, and **NULL** if on failure.

ERROR CONDITIONS

Messages to the message box manager will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_EDIT

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

Memory cannot be allocated[XrOUTOFMEM].

Unknown message [XrINVALIDMSG].

NOTES

There is a bug in the message box manager that can potentially cause incorrect state information to be presented to the user. This is caused by the message box manager consuming all events generated for the application. The symptoms of this bug include: an application not being redrawn properly, checkboxes showing state information inconsistent with the application, and others.

The workaround for this problem is as follows:

1) Set up a window function using the call: *XrInput(Window, MSG_ADDWINDOWFUNCT, info)* for exposure events on all your registered windows. This function will be called automatically when an exposure event occurs on any of your registered windows. The function should take care of redrawing your windows.

2) Save the event mask for all of your windows using the call: *XGetWindowAttributes*().

3) Set the event mask for all of your windows to: *ExposureMask*.

4) Call the message box manager.

5) Set the event mask back to its original setting.

The following code segment demonstrates the workaround:

```
XWindowAttributes wattr;

XrInput(myWindow, MSG_ADDWINDOWFUNCT, &info);
        .
        .
        .
XGetWindowAttributes(display, myWindow, &wattr);
XSelectInput(display, myWindow, ExposureMask);
XrMessageBox(&myMessageBox, MSG_EDIT, &messageEvent);
XSelectInput(display, myWindow, wattr.your_event_mask);
```

ORIGIN

HP

NAME

XrPageEdit - an editor for entering and editing multiple lines of text.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrPageEdit (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The page edit editor provides an application with the means for placing a multi-line text editing instance anywhere within the bounds of a window. The editor always operates in insert mode, and currently only supports the use of fixed space fonts. It is capable of handling both printable and non-printable (control) characters.

When creating a page editor instance, the application must pass in a definition for the rectangular region into which the editor instance will be drawn. If the rectangle is not the correct size to hold the instance, then the editor will refuse to create the instance. To prevent this from happening, an application should use the **MSG_SIZE** message to obtain this rectangle.

ACTIVE FIELD

The page edit editor supports the concept of an active field. When a given page editor instance has control of the input stream, it is considered to be active, and it will indicate this by visibly changing its format. The means by which it indicates this fact is twofold:

It will display a cursor at the current character position within the instance.

It will redraw the border for the editing region (which is usually 1 pixel wide) as a 3 pixel wide border.

There are two means by which a page edit instance can be told to become active:

By issuing a MSG_ACTIVATE request to the editor.

By issuing a MSG_EDIT request to the editor, and passing it any input event understood by the page editor.

COMPONENTS

The two most important pieces of a page editor instance are the cursor and the editing buffer. The purpose of the cursor, is to indicate to the user where the next character typed will be placed, and also what portion of the editing buffer will be modified when one of the editing keys is pressed. As the user types keys, those characters to the right of the cursor will be shifted one position further to the right, and the information will be added at the location of the cursor. After the new character has been placed, the cursor will be moved to the right of the newly added character. Both printable ASCII characters and control character can be displayed; control characters are displayed as "^char". The cursor is always displayed as a block cursor.

When the instance is first created, the application program will pass in a pointer to a buffer, which will be used by the editor to store the current editing session. In addition, the application must supply values indicating the size of the buffer, the number of characters currently residing within the buffer, and the maximum size the buffer should be allowed to reach.

While an editing session is in progress, the editor will check to see if the buffer is full after each character is added. When the buffer does become full, the editor will attempt to automatically expand the buffer, unless the buffer has reached the maximum size, as specified by the application; this upper limit check can be disabled, if the application wishes to allow the buffer to grow indefinitely. In order for buffer expansion to be possible, the application must use the **(*xrMalloc)**() call to initially obtain its editing buffer; statically defined buffers will not work correctly.

EDITING MODES

The page editor supports only a single mode of character entering: insert mode.

When a new character is entered, all characters to the right of the cursor are shifted one position to the right, and the character is then placed at the cursor position. Afterwards, the cursor is moved one position to the right; if the cursor is at the right edge of the editing region, then the cursor will be placed at the beginning of the next line, scrolling, if necessary.

EDITING

The page editor supports most of the standard editing commands which a user may generate from the keyboard. Refer to **XrInputMap(3X)** for a description on how these editing features may be accessed from a non-HP keyboard. Among the editing commands which will be supported are the following:

Cursor Left
: This will cause the cursor to be moved one character position to the left. If the cursor is at the beginning of a line, then it will wrap to the end of the preceding line, scrolling, if necessary.

Cursor Right
: This will cause the cursor to be moved one character position to the right. If the cursor is at the end of a line, then it will wrap to the beginning of the next line, scrolling, if necessary.

Cursor Up
: This will cause the cursor to be moved one line up. If the cursor is already at the first display line, then it will scroll the display down one line, and then move the cursor.

Cursor Down
: This will cause the cursor to be moved one line down. If the cursor is already at the last display line, it will scroll the display up one line and then move the cursor.

Roll Up
: This will cause the screen contents to be rolled up one line. This causes a new line to be displayed at the bottom of the editing region.

Roll Down
: This will cause the screen contents to be rolled down one line. This causes a new line to be displayed at the top of the editing region.

Next Page
This will cause the next page of the editing buffer to be displayed. The last line of the current page will be the top line of the new page.

Previous Page
This will cause the previous page of the editing buffer to be displayed. The top line of the current page will be the bottom line of the new page.

Home Up
This will cause the cursor to be moved to the beginning of the editing buffer. The display will be updated to match as needed.

Home Down
This will cause the cursor to be moved to the end of the editing buffer. The display will be updated to match as needed.

Backspace
This will cause the character to the left of the cursor to be deleted and all characters to its right to be shifted one position to the left. The cursor will also move one position to the left. If the cursor was at the start of a line, then the Newline character at the end of the preceding line will be deleted. This has the effect of joining the two lines together.

Delete Char
This will cause the character underneath the cursor to be deleted, and all characters to its right will be shifted one position to the left. The cursor will remain in its same position. If the cursor is currently located at the end of a line, then the Newline character terminating that line will be deleted. This has the effect of joining the current line with the line following it.

Delete Line
This will delete the entire line where the cursor is currently located.

Clear Line
This will cause all characters under and to the right of the cursor to be deleted. The location of the cursor will be unchanged.

Insert Line
This will cause a new line to be started above the line in which the cursor is currently located. The cursor will then be moved to the start of the new line.

MOUSE POINTER

As was mentioned above, the cursor's position is automatically updated after each character placement operation or editing request. However, there is another method for specifying a new cursor position. By using the mouse pointer, you can move the cursor to anywhere within the current editing region, and then 'click' the mouse pointer button. This will cause the cursor to be placed at, or as near as possible to, the selected character position. If a position past the end of the buffer contents is selected, then the cursor will automatically be placed after the last character in the buffer.

DEACTIVATION

Once a page edit instance has been made 'active', there are several ways for it to become 'inactive'. When a page edit instance is deactivated, it will turn off its cursor, and it will redraw the editing region border as a 1 pixel wide box. The instance will no longer accept input, until it has been reactivated.

The first method for deactivating a page edit instance, is to use the mouse pointer to select a region outside the editor's rectangular region. When the page editor determines that a select event occurred outside its domain, it will deactivate the instance, and then place two input events onto the front of the application's input queue:

Input event 1
: This first input event placed on the input queue will be a copy of the select event which caused the instance to be deactivated. This event will notify the application that a select occurred somewhere else within the display.

Input event 2
: This will be the event information generated by the page edit editor. It will inform the application that the instance has been exited, and that editing is now complete.

A second method of deactivating a page edit instance can take place when the page editor has temporarily returned control to the application, because one of the status events was generated; the status events are composed of "unknown event", "buffer full", "buffer expanded" and "buffer expansion failed". When an application receives one of these events, it will normally process it, and then return control to the editor, by means of the **MSG_ACTIVATE** message. However, if the application does not want control to return to the editor, but instead, wishes to deactivate the instance, then it can issue a **MSG_DEACTIVATE** request.

RETURNED INFORMATION

The page editor will not return an input event to the calling application each time the page edit instance is modified. Instead, it will only return to the application when a major event occurs. These are also referred to as break conditions. There are three categories of break conditions. One is termed a permanent break event, and the other two are termed temporary status events:

-A select event occurred outside the page editor instance.

- The buffer is full, the buffer was expanded or the buffer expansion failed.

-An X event other than a KeyPress, KeyRelease, XrSELECT or XrSELECTUP event is received.

For the first type, the editor will return, and not expect to be called again, until the instance has again been activated. For the second type, the editor is merely returning some status information to the application. As soon as the application has processed the status information, it should immediately re-invoke the editor, by issuing a MSG_ACTIVATE. Failing to follow this rule may produce unpredicatable results. The third type is similar to the second type, in that it too is only returning some status information. It will return a status event telling the application that an unknown X event was received, along with the unknown X event. The application should then process the X event which is on the input queue AFTER the status event, and then re-invoke the editor in the fashion listed above.

EDITOR MESSAGES

MSG_NEW
: This message will be the means by which an application program can create an instance of the page editor within a window. It will expect the *instance* parameter to be set to NULL, and the

data parameter to point to a filled out instance of the following structure:

```
typedef struct {
    Window          editorWindowId;
    RECTANGLE       editorRect;
    INT8            editorState;
    INT32           editorFGColor;
    INT32           editorBGColor;
    XFontStruct   * editorFont;
    INT8          * buffer;
    INT32           bufferCount;
    INT32           bufferSize;
    INT32           maxCharCount;
    INT32           rowCount;
    INT32           colCount;
    INT32           tabWidth;
} xrPageEditInfo;
```

editorWindowId
: This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect
: This describes the location and size of the region into which the page edit instance is to be displayed. If a rectangle too small to hold the instance is specified, the editor will refuse to handle the request and will fail.

editorState
: This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
: This field specifies the foreground color to be used when drawing the editor instance. The borders and all text are drawn using the foreground color. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
: This field specifies the background color to be used when drawing the editor instance. The interior of the editing region is drawn using the background color. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont
: This points to the structure which contains all of the information necessary to describe the font to be used when displaying the page editor instance. If this pointer is NULL, then the default system base font will be used. This must refer to a fixed spaced font, since the page editor does not currently support proportionally spaced fonts.

buffer
: This is a pointer to the editing buffer. If this pointer is NULL, then the MSG_NEW command will fail.

bufferCount
This field indicates the number of bytes currently residing in the buffer. This may contain any value greater than or equal to zero, and less than or equal to the size of the buffer.

bufferSize
This value indicates the current size of the editing buffer. When the byte count reaches this limit, the buffer will be expanded, if possible.

maxCharCount
This value represents the maximum number of bytes which will be allowed in the page editor instance. This information will be used by the editor to prevent the editing buffer from being expanded past this limit. If an application wishes to allow the buffer to be expanded whenever necessary, without imposing any upper limit, then this value should be set to -1.

rowCount
This value specifies the number of lines which are to be displayed within the page editor instance; an instance cannot be created with fewer than 1 line visible.

colCount
This value specifies the number of character columns which are to be displayed within the page editor instance. An instance cannot be created with less than 1 column visible. The total editing region will be *rowCount* lines by *colCount* columns in size.

tabWidth
This value specifies how often a tab stop should be placed within the editing region. These come into play when the TAB key is entered by the user. When the TAB key is entered, the cursor will be moved to the next tab stop in the current line. If there are no more tab stops in the current line, the cursor moves to the start of the next line.

The editor will then draw the page edit instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE
This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, returned by **MSG_NEW**; this parameter specifies which instance is to be destroyed.

When a page editor instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse pointer selects or keyboard input, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE
This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified page edit instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given page edit instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If an instance is visible, but not sensitive, then the editing region will be drawn and filled with a 50% pattern. The current contents of the editing region will also be displayed.

MSG_ACTIVATE

This message allows an application to force a page edit instance active, thus causing it to turn on the cursor, widen the editing region border, and start monitoring all incoming keyboard events. This message also serves as the only means for re-entering an active page edit instance, after an application has received a status event from the instance. When a **MSG_ACTIVATE** message is issued, the editor instance must be both visible and sensitive. If this is not the case, the message will fail.

When this message is issued, the *instance* parameter must specify the page edit instance which is to be activated. The *data* parameter is unused, and should be set to **NULL**.

MSG_DEACTIVATE

This message allows an application to force a page edit instance inactive. Normally, when a page edit instance returns a status event to an application (such as the buffer is now full, or an unknown X event was received), the application processes the status event and then reinvokes the page editor. However, it is possible for an application to decide that it does not want to reinvoke the editor instance, but rather, it wants the instance to be deactivated. The instance will be drawn as inactive, with the cursor turned off, and the border redrawn as a single pixel wide line.

When this message is issued, the *instance* parameter must specify the page edit instance which is to be deactivated. The *data* parameter is unused. If the page edit instance is already inactive, then this request will be ignored.

MSG_GETBUFINFO

This message allows an application to obtain several pieces of information which describe the current state of the editing buffer. This includes the address of the buffer, the number of characters currently in the buffer, the current buffer size, and the maximum size the buffer is allowed to be. This information is particularly important to an application after the editing buffer has been expanded. When the application is notified that the buffer has been expanded, it MUST issue this request. Since the editor uses *xrRealloc*() (refer to **XrInit(3X)**) to increase the size of the buffer, it is possible that the location of the buffer may change as it is resized. This message allows the application to get the new buffer information, and prevent a memory fault from occurring, because the application referenced the old buffer area.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be queried. The *data* parameter must point to an instance of the following structure, where the buffer information will be placed:

```
typedef struct {
   INT8  * buffer;
   INT32  bufferCount;
   INT32  bufferSize;
```

```
    INT32  maxCharCount;
  } xrPageEditBufInfo;
```

MSG_SETBUFINFO

This message allows an application to change the address of the editing buffer. When this message is issued, the page editor will display the new buffer and will place the cursor at the first character position within the buffer.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be modified. The *data* parameter must point to an instance of the following structure, which must contain the new buffer information:

```
typedef struct {
   INT8  * buffer;
   INT32  bufferCount;
   INT32  bufferSize;
   INT32  maxCharCount;
} xrPageEditBufInfo;
```

MSG_REDRAW

This editor provides an application with the means for requesting that a page edit instance be redrawn.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The page editor supports the following redraw modes:

- **XrREDRAW_ALL**

- **XrREDRAW_TEXT**

XrREDRAW_ALL will cause the editor to redraw the complete editor instance, while **XrREDRAW_TEXT** will cause only the editing buffer contents to be redrawn.

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a given page edit instance.

The page editor expects the *instance* parameter to be set to **NULL**, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
    Window        editorWindowId;
    RECTANGLE     editorRect;
    INT8          editorState;
    INT32         editorFGColor;
    INT32         editorBGColor;
    XFontStruct  * editorFont;
    INT8         * buffer;
    INT32          bufferCount;
```

```
      INT32       bufferSize;
      INT32       maxCharCount;
      INT32       rowCount;
      INT32       colCount;
      INT32       tabWidth;
   } xrPageEditInfo;
```

The fields which *must* be filled out by the application program BEFORE issuing this call, are the *rowCount, colCount* and *editorFont* fields.

If *editorFont* has been set to **NULL**, then the default system base font will be used when calculating the rectangle size. If a particular font is requested it will be used. This must refer to a fixed spaced font, since the page editor does not currently support proportionally spaced fonts.

The rectangle calculated will be the 0 based rectangle, into which the specified page edit instance will fit, using the specified font and row/column sizes.

In return, *editorRect* will be set to describe the rectangle into which the described instance will fit.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified. The top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the page editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location. This occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routine, when an input event occurs while a page edit instance is active.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which page edit instance to process, and the *data* parameter pointing to an *XEvent* structure.

The **MSG_EDIT** handler for the page edit editor is much more complex than most editors, since it understands a much broader range of input events. Making up the list of valid input types for this editor are:

All X KeyPress and KeyRelease events.

All X button events which map into either an XrSELECT or XrSELECTUP event (see XrMapButton(3X) and XrInit(3X)).

All other X events will be treated by the editor as an unknown event, and will be returned to the application for local processing. This is discussed further in a later section dealing with unknown events.

The sections below will attempt to describe how the editor handles each of the above input types.

X KeyPress and KeyRelease events (keys that map to regular ASCII characters only)

If the page editing buffer is not already filled to its maximum allowable size, then the following will occur:

All characters in the buffer, which are at or to the right of the cursor, will be shifted one place to the right within the editing buffer.

The new character will be inserted into the buffer, at the cursor position.

The cursor will be moved one character position to the right. If the end of the line is surpassed, or if the character was a Newline character, then the cursor will be moved to the start of the next line. If the character was a TAB, it will be expanded, and the cursor will move to the next TAB stop.

The instance will be forced active (if not already).

Afterwards, a check will be made to see if the editing buffer is now full. If the buffer is full, then one of three things will occur:

If the buffer has reached its maximum allowable size, as specified by the application, then the following status event will be appended to the front of the application's input queue, and the editor will return (the editor will expect the application to re-invoke the editor, using either MSG_ACTIVATE or MSG_DEACTIVATE, once it has finished handling this buffer event):

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrPAGEEDIT
value1	=	XrBUFFER_FULL
valuePtr	=	pointer to instance's editor structure

If the buffer is not at its maximum allowable size, but the attempt to expand the buffer fails, then the following status event will be appended to the front of the application's input queue, and the editor will return (the editor will expect the application to re-invoke the editor, using either MSG_ACTIVATE or MSG_DEACTIVATE, once it has finished handling this buffer event):

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display

source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrPAGEEDIT
value1	=	XrBUFEXPANDFAILED
valuePtr	=	pointer to instance's editor structure

If the buffer expansion succeeds, then the following status event will be appended to the front of the application's input queue, and the editor will return (the editor will expect the application to re-invoke the editor, using either MSG_ACTIVATE or MSG_DEACTIVATE, once it has finished handling this buffer event). When an application receives this event, it MUST issue a MSG_GETBUFINFO request, to obtain the new editing buffer address, since expanding the buffer may cause its address to change.

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrPAGEEDIT
value1	=	XrBUFEXPANDED
valuePtr	=	pointer to instance's editor structure

If none of the above events occurred, then the editor will block, waiting for the user to generate the next input event.

When a new character is displayed, if it falls within the range of printable Ascii characters, then the character will be displayed in its normal fashion. If, however, the character is a control character, which are normally considered unprintable, then even though the character only takes up a single spot within the editing buffer, it will be displayed as a two character sequence. For example, if the character 'control X' were entered, then it would be displayed as '^X'.

X KeyPress and KeyRelease events (Editing keys)

Cursor Right
If the cursor is not already at the end of the editing buffer, it will be shifted one character position to the right. If the cursor is at the end of a line, then the cursor will wrap to the beginning of the next line, scrolling if necessary.

Cursor Left
If the cursor is not already at the beginning of the editing buffer, then it will be shifted one character position to the left. If the cursor is at the beginning of a line, it will wrap to the last character in the preceding line, scrolling if necessary.

Cursor Up
If the cursor is not already at the first line of the editing buffer, then it will be moved up one line. If the cursor is in the first display line, then the display will be scrolled down one line.

Cursor Down
If the cursor is not already at the last line of the editing buffer, then it will be moved down one line. If the cursor is in the last display line, then the display will be scrolled up one line.

Roll Up
This will cause the screen contents to roll one line up, and display a new line at the bottom of the editing region.

Roll Down
This will cause the screen contents to roll down one line, and display a new line at the top of the editing region.

Next Page
This will cause the next page of the editing buffer to be displayed. The last line of the current page will be the top line of the new page.

Previous Page
This will cause the previous page of the editing buffer to be displayed. The top line of the current page will be the bottom line of the new page.

Home Up
This will cause the cursor to be moved to the beginning of the editing buffer. The display will be updated to match as needed.

Home Down
This will cause the cursor to be moved to the end of the editing buffer. The display will be updated to match as needed.

Backspace
If the cursor is not already at the beginning of the buffer, the character to the left of the cursor will be deleted, and all characters to its right will be shifted one position to the left. The cursor will also move one position to the left. If the cursor is located at the beginning of a line, the backspace will cause the Newline character at the end of the preceding line to be deleted. This will join the two lines together.

Delete Char
If the cursor is not at the end of the buffer, then the character underneath the cursor will be deleted, and all characters to its right will be shifted one position to the left. If the character being deleted is the Newline at the end of a line, then that line will be joined with the line which follows it.

Delete Line
This causes the line where the cursor is located to be deleted. The line being deleted is considered to start after the last Newline character preceding the cursor, and going up to and including the next Newline character after the cursor. Afterwards, the cursor is moved to the start of the line which followed the line deleted.

Clear Line

This action will delete those characters under and to the right of the cursor to be deleted on the current line. The end of the line is considered to be where the next Newline is encountered.

Insert Line

This will cause a new line to be opened within the editing buffer immediately before the line in which the cursor is located. The cursor will then be positioned at the start of the new line.

X ButtonPress or ButtonRelease Events

The only other type of X ButtonEvent understood by this editor is the one which maps into either a **XrSELECT** or **XrSELECTUP** event. All other button events are treated as an unknown event. When a select occurs, the editor will first check to see if it occurred within the instance's rectangle.

If it did not occur within the rectangle, then the instance will be redrawn as 'not active', and two input events will be added to the front of the application's input queue:

1) A copy of the select input event which we just received.

```
2) type       = XrXRAY
   serial     = 0
   send_event = 0
   display    = pointer to the current display
   source     = the window Id
   inputType  = XrEDITOR
   inputCode  = XrPAGEEDIT
   value1     = XrPEDIT_BREAK
   value2     = XrSELECT
   valuePtr   = pointer to instance's editor structure.
```

NOTE: the second input event will end up at the front of the queue.

The editor will then return, and not expect to be called again, until the next time it should be made active.

If the select occurred within the editing region, and the instance was active, then the cursor will be moved to the character position nearest to where the select occurred. The editor will then block, waiting for the user to generate the next event. If the instance was not active, then the event will cause the instance to be activated.

Unknown Events

Any event which is received by the page editor, which does not fall under any of the groupings discussed above, will be treated as an unknown event, and will be passed back to the application for processing. When the application receives notification of an unknown event from the page editor, it should request the next event from its input queue (the unknown event) and then process it. Since this is only a temporary break condition, the editor will expect the application to re-invoke the editor, once it has finished processing the unknown event.

The following two events will be pushed onto the front of the application's input queue:

1) A copy of the unknown X event which we just received.

```
2) type       = XrXRAY
   serial     = 0
   send_event = 0
   display    = pointer to the current display
   source     = the window Id
   inputType  = XrEDITOR
   inputCode  = XrPAGEEDIT
   value1     = XrUNKNOWN_EVENT
   valuePtr   = pointer to instance's editor structure.
```

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the page editor will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is too small to hold the specified instance [XrINVALIDRECT].

The specified font is not a fixed width font [XrINVALIDPARM].

buffer is set to NULL [XrINVALIDPTR].

maxCharCount is set to zero [XrINVALIDPARM].

bufferSize is less than or equal to zero [XrINVALIDPARM].

bufferCount is greater than *maxCharCount* [*XrPARMOUTOFRANGE*].

bufferCount is greater than *bufferSize* [*XrPARMOUTOFRANGE*].

bufferCount is less than zero [XrINVALIDPARM].

rowCount is less than or equal to zero [XrPARMOUTOFRANGE].

colCount is less than or equal to zero [XrPARMOUTOFRANGE].

A call to 'X' failed { XCreatePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** or **XrREDRAW_TEXT** is specified

[XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

The specified font is not fixed width [XrINVALIDPARM].

rowCount is less than or equal to zero [XrPARMOUTOFRANGE].

colCount is less than or equal to zero [XrPARMOUTOFRANGE].

MSG_ACTIVATE

The instance is not sensitive and visible [XrINVALIDPARM].

MSG_SETBUFINFO

data is set to NULL [XrINVALIDPTR].

buffer is set to NULL [XrINVALIDPTR].

maxCharCount is set to zero [XrINVALIDPARM].

bufferSize is less than or equal to zero [XrINVALIDPARM].

bufferCount is greater than *maxCharCount* [*XrPARMOUTOFRANGE*].

bufferCount is greater than *bufferSize* [*XrPARMOUTOFRANGE*].

bufferCount is less than zero [XrINVALIDPARM].

MSG_GETSTATE, MSG_MOVE and MSG_GETBUFINFO

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X), XrMapButton(3X)

NAME

XrPanel - The Xrlib Panel Manager.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrPanel *
XrPanel (panelInstance, message, data)
xrPanel  * panelInstance;
INT32     message;
INT8    * data;
```

DESCRIPTION

XrPanel is the Xrlib panel manager. This man page gives a brief summary of the panel manager structures, and messages. For a more complete description of the panel manager, refer to the "Dialogs" chapter in "Programming With The Xrlib User Interface Toolbox."

STRUCTURES

The following structures are used to communicate with the panel manager.

The xrPanelInfo Structure

This structure is used to create a panel in conjunction with MSG_NEW.

```
typedef struct {
     POINT           panelOrigin;
     SIZE            panelSize;
     Window          relativeTo;
     Window          childOf;
     xrPanelContext  * panelContext;

     xrPanelField    * fieldList;
     INT32           numFields;
     INT32           panelId;
} xrPanelInfo;

typedef struct {
     xrEditor     * (* editorFunct) ();
     INT8         * editorData;
     xrEditor     * editorInstance;
}xrPanelField;
```

The panel context structure

This structure contains the information used by the panel manager to deal with a particular panel.

```
typedef struct {
     Pixmap        winBackground;
     Pixmap        winForeground;
     INT16         borderWidth;
     INT32         foregroundColor;
     INT32         backgroundColor;
     XFontStruct   * fontInfo;
```

```
        Cursor      cursor;

        INT32       showFlag;
        xrEditor    * currentEditor;
        INT32       timer;
        INT32       (* initFunct)();
    }xrPanelContext;
```

A description of each member of these structures is given in the "Dialogs" chapter of "Programming With The Xrlib User Interface Toolbox."

MESSAGES

The following messages allow the programmer to give the panel manager directives.

MSG_NEW
: MSG_NEW creates the panel structure discussed earlier. The *panelInstance* parameter may be set to NULL, as it is not used. The *data* parameter should be set to a structure of type *xrPanelInfo*. MSG_NEW returns a pointer to a panel, failure is indicated by a return value of NULL.

MSG_FREE
: MSG_FREE destroys a panel and frees the associated memory. *data* may be set to NULL for this command.

MSG_SIZE
: MSG_SIZE returns the size of a panel given a *panelInfo* structure. *data* should point to a panelInfo structure. The size is returned in the *panelSize* member of the panelInfo structure.

MSG_EDIT
: MSG_EDIT causes the panel manager to begin looking for input to a particular panel. MSG_EDIT should be sent to the panel manager when an application expects input from a panel. *data* should point to an xrEvent structure.

MSG_CURRENTEDITOR
: MSG_CURRENTEDITOR instructs the panel manager to invoke a specific editor when MSG_EDIT is received. This allows a panel to have an active field (such as a TextEdit field) without a input from the user. *data* points to an *xrEditor* instance.

MSG_MOVE
: MSG_MOVE allows a panel to be moved about the display. *data* is a pointer to a point structure.

MSG_RESIZE
: MSG_RESIZE allows a panel to be resized by the application. *data* is a pointer to a rectangle structure. The panel will be moved to an absolute point on the display and its window will change to the given size.

MSG_HIDE
: MSG_HIDE hides a panel. *data* is not used and may be set to NULL.

MSG_SHOW
: MSG_SHOW shows a panel. *data* is not used and may be set to NULL. A panel will be shown automatically upon creation if the *showFlag* field of the panelInfo structure is set to TRUE.

MSG_REDRAW
: MSG_REDRAW causes the editors of a panel to be redrawn. *data* is not used and may be set to NULL.

MSG_GETPANELCONTEXT

MSG_GETPANELCONTEXT fills out an xrPanelContext structure with the panel defaults. *data* should point to an xrPanelContext structure.

MSG_NEWSUBPANEL

MSG_NEWSUBPANEL creates a sub-panel. *panelInstance* should be the pointer to the parent panel, and *data* is a *xrPanelInfo* structure.

MSG_FREESUBPANEL

MSG_FREESUBPANEL destroys a sub-panel and its associated memory.

MSG_SHOWSUBPANEL

MSG_SHOWSUBPANEL causes the editors of a sub-panel to be added to its main panel. *data* should point to an xrEditorGroup structure indicating which sub-panel is to be activated.

MSG_HIDESUBPANEL

MSG_HIDESUBPANEL removes the editors from its associated main panel. *data* should point to an xrEditorGroup structure indicating which sub-panel is to be deactivated.

MSG_SETOVERRIDE

MSG_SETOVERRIDE provides an application with a mechanism to allow window manager interaction in the placing and sizing of panels. By default, the application places, sizes and maps panels directly without any window manager interaction. .

The X-ray library maintains an internal 'override' flag, which indicates if any new panel should bypass the window manager (TRUE) or if the window manager should be given a chance to position the panel (FALSE). The default setting of this flag is TRUE.

MSG_SETOVERRIDE permits the alteration of the 'override' flag. If MSG_SETOVERRIDE is issued with the instance parameter set to NULL and the *data* parameter set to TRUE or FALSE, all subsequent panel creations will be affected. If a panel instance pointer is provided as the instance parameter, only that specific panel will be affected.

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value is returned. For the MSG_NEW message, this non-NULL value is the pointer to the newly created menu or panel instance.

If the message request fails, NULL is returned.

ERROR CONDITIONS

Messages to the panel manager will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_FREE

panelInstance is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_EDIT

data is set to NULL [XrINVALIDPTR].

Panel is not showing [XrINVALIDEVENT]

A call to 'X' failed [XrXCALLFAILED].

MSG_CURRENTEDITOR

data is set to NULL [XrINVALIDPTR].

MSG_MOVE

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_RESIZE

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_HIDE

A call to 'X' failed [XrXCALLFAILED].

MSG_SHOW

A call to 'X' failed [XrXCALLFAILED].

MSG_REDRAW

A call to 'X' failed [XrXCALLFAILED].

MSG_GETPANELCONTEXT

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_NEWSUBPANEL

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM]

MSG_FREESUBPANEL

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_SHOWSUBPANEL

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

MSG_HIDESUBPANEL

data is set to NULL [XrINVALIDPTR].

A call to 'X' failed [XrXCALLFAILED].

ORIGIN

HP

NAME

XrPoint: (XrAddPt, XrSubPt, XrSetPt, XrOffsetPt, XrCopyPt, XrEqualPt) - calculations with points.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

XrAddPt (srcPt, dstPt)
POINT * srcPt;
POINT * dstPt;

XrSubPt (srcPt, dstPt)
POINT * srcPt;
POINT * dstPt;

XrSetPt (pt, x, y)
POINT * pt;
INT16 x;
INT16 y;

XrOffsetPt (pt, x, y)
POINT * pt;
INT16 x;
INT16 y;

XrCopyPt (srcPt, dstPt)
POINT * srcPt;
POINT * dstPt;

XrEqualPt (ptA, ptB)
POINT * ptA;
POINT * ptB;
```

DESCRIPTION

XrAddPt

Add the coordinates of *srcPt* to the coordinates of *dstPt*, and return the result in *dstPt*.

XrSubPt

Subtract the coordinates of *srcPt* from the coordinates of *dstPt*, and return the result in *dstPt*.

XrSetPt

Assign the two coordinates to a variable of type POINT.

XrOffsetPt

Change the values of the point structure *pt* by adding the *x* and *y* offsets to the members of *pt*.

XrCopyPt

Copy *srcPt* into *dstPt*.

XrEqualPt

Compare the two points and return **TRUE** if they are equal or **FALSE** if not.

ORIGIN

HP

NAME

XrPushButton - an editor that controls a set of pushbuttons.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrPushButton (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The push button editor is used to create and process a group of related push buttons within a given window. The number of push buttons, and how they are displayed, is completely controlled by the application. Each push button may include an optional text label, which will be displayed in the center of the oval representing the button. This label may be a single line of text, or it may be multiple lines of text.

The size of the push buttons are proportional to the size of the font being used to display the text labels. By allowing an application to specify this font, the size of the push buttons can be adjusted for individual displays.

An application may specify one of the push buttons to be treated as a default selection. This implies that that button will be drawn with a double wide border around it. This feature may be disabled, if not needed.

This field editor supports two distinct sets of state flags: those associated with the whole instance (which are modified using MSG_GETSTATE and MSG_SETSTATE), and those which are assigned to the individual pushbuttons (which are modified using MSG_GETITEMSTATES and MSG_SETITEMSTATES). The state flags which are associated with the instance as a whole have precedence over the individual state flags. For example, if the XrVISIBLE flags is cleared for the instance, then none of the pushbuttons will be displayed, regardless of their individual state flag settings. But, if the XrVISIBLE flag is set for the instance, then the field editor will look at the state of the XrVISIBLE flag associated with each pushbutton, to determine if the entity should be displayed.

Using the cursor and the mouse, a user is able to interactively select a push button, by moving the cursor over one of the buttons, and 'clicking' the mouse button. When the mouse button is depressed, the selected push button will be drawn as active (solid filled); when the user releases the mouse button, the selected push button will be redrawn as inactive (not filled), and an event will be returned to the application, notifying it that a push button was selected.

When the editor draws the specified push button instance, it will attempt to space out the individual buttons, to take full advantage of the specified *editorRect.* If the rectangle is larger than needed, then the extra space will be divided equally as blank space between the rows and columns. If the rectangle is too small, then the request will fail.

When displaying a push button editor instance, the button sizes may vary between the various rows and columns. However, to prevent the instance from looking disorganized, the following steps will be taken:

Each button within a particular horizontal row, will be the same height.

Each button within a particular vertical column, will be the same width.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a push button editor instance in a window. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window       editorWindowId;
   RECTANGLE    editorRect;
   INT8         editorState;
   INT32        editorFGColor;
   INT32        editorBGColor;
   XFontStruct  * editorFont;
   INT16        numFields;
   INT16        numCols;
   INT16        defaultButton;
   INT16        borderWidth;
   INT8         ** labels;
   INT8         * stateFlags;
} xrPushButtonInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the region into which the push button instance is to be located. If the rectangle is larger than is needed to hold the specified instance, then the extra space will be used as padding between the individual push buttons. If the rectangle is too small, then the request will fail.

editorState

This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor

This field specifies the foreground color to be used when drawing the editor instance. This is used when displaying the button labels, and for filling an active button. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor

This field specifies the background color to be used when drawing the editor instance. This is used for filling the interior of an inactive button. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont
This is a pointer to a structure which describes the font to be used when creating this editor instance. If the pointer has been set to NULL, then the editor will use the default system base font. The font is important not only because it determines how the labels will look, but it is also used to determine how large the buttons should be.

numFields
This value indicates the total number of push buttons which are included in this instance. Once specified, this value cannot be changed.

numCols
This value indicates the number of columns into which the push buttons are to be displayed. Using this value, and *numFields*, the editor will automatically calculate the number of rows. If this value is greater than *numFields*, then the create request will fail.

defaultButton
This value indicates which push button should be displayed as the default selection; this involves drawing the button with a double wide border. This value must be in the range *0* - (*numFields* - *1*); any value outside this range will disable this feature. Setting this field to -1 is the standard way to disable this feature.

borderWidth
This value describes the width of the border surrounding each of the push buttons. This value must be greater than zero.

labels
This is a pointer to an array of NULL terminated character strings, representing the labels associated with each push button. If this pointer is set to NULL, then none of the buttons will have labels. If any of the pointers within the array are NULL, or if a label has a length of zero, then those buttons will not have labels. The first entry in the array will correspond to push button 0, the second to push button 1, etc. The editor will not keep a copy of each of these strings; instead, it will only keep a copy of the pointer to the string array. It will rely on the application to NOT modify these strings, once the instance has been created.

As was stated earlier, a label may be a single line of text, or multiple lines of text. To create multiple lines, insert a NEWLINE character (octal 012) at the end of each line of the label.

stateFlags
This is a pointer to an array of 8 bit integer values, each containing the initial state flag settings to be associated with each pushbutton in the instance. If this pointer has been set to NULL, then the state flags for each entity will be set to XrSENSITIVE and XrVISIBLE. The first entry in the array corresponds to pushbutton 0, etc.

The editor will then draw the push button editor instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, returned by the **MSG_NEW** message. This parameter specifies which instance is to be destroyed.

When a push button instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified push button instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed. REMEMBER: these are the state flags for the whole instance; not the state flags for the individual buttons.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given push button editor instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If an instance is visible, but not sensitive, then each push button will be drawn and filled with a 50% pattern.

MSG_GETITEMSTATES

This message allows an application to obtain a copy of the individual state flags associated with each of the pushbuttons contained within the specified editor instance. These state flags differ from the state flags obtained using **MSG_GETSTATE.** The *instance* parameter must specify the editor instance to be queried, while the *data* parameter should point to an array of 8 bit integer values. The state flags will be returned by means of this array. The flags associated with pushbutton 0 will be returned in the first slot in the array, etc.

MSG_SETITEMSTATES

This message allows an application to modify the individual state flags associated with each of the pushbuttons contained within the specified editor instance. Presently, only the **XrVISIBLE** and **XrSENSITIVE** flags are supported. The *instance* parameter must specify the editor instance which is to be modified, while the *data* parameter should point to an array of 8 bit integer values. The new state flags which are to be assigned to each entity within the instance will be obtained from this array. The value in the first slot of the array will be assigned to pushbutton 0, etc. After the new state flags have been saved, only those pushbuttons whose state flags changed will be redrawn.

MSG_GETITEMCOUNT

This message allows an application to obtain a count, which indicates the number of individual push buttons in the specified instance. The editor will assume that the *instance* parameter specifies the editor instance to query, and the *data* parameter points to a 32 bit integer value.

The item count value will be returned by means of this integer.

This message is useful when used in conjunction with the **MSG_GETITEMRECTS** message. It allows an application to obtain the number of items in the instance, so that the application can then allocate enough memory to hold the rectangle information returned by **MSG_GETITEMRECTS.**

MSG_GETITEMRECTS

This message returns the coordinates for the rectangle which describes each of the individual push buttons. The message will expect the *instance* parameter to specify the editor instance to query, and the *data* parameter to point to a structure of the following format:

```
        RECTANGLE   itemRects[x];
```

This array will then be filled with the rectangle information, and returned to the application.

Before an application can make this call, it needs to know the number of items in the specified instance, so that it can allocate a structure large enough to hold all of the rectangle information. The application should use the **MSG_GETITEMCOUNT** message to obtain this information. The application can then allocate an array large enough to hold all of the rectangle entries.

The order of the rectangle items returned in the array directly correlates to the order the items were originally created in. The first element in the array will describe button 0, the second button 1, etc.

This message is useful to those applications which have a need of knowing where the individual items in a push button instance are located. The most common use would be by a forms controller, which would use the information to place an 'active field' indicator by a given item.

MSG_REDRAW

This editor provides an application with the means for requesting that an instance of push buttons be redrawn. The current values, labels and font information will be used.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

This message is useful for when an application wishes to modify the value or the state flags for a particular button. The application need only modify the appropriate values in the shared arrays, and then issue a **MSG_REDRAW** request. The editor instance will automatically be drawn to match the new data.

The push button editor supports the following redraw mode:

- **XrREDRAW_ALL**

When the **XrREDRAW_ALL** option is specified, the complete editor instance will be redrawn.

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a push button editor instance. The push button editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   XFontStruct   * editorFont;
   INT16         numFields;
   INT16         numCols;
   INT16         defaultButton;
   INT16         borderWidth;
   INT8          ** labels;
   INT8          * stateFlags;
} xrPushButtonInfo;
```

The only fields which *must* be filled out by the application program BEFORE issuing this call, are the *numFields, numCols, labels, borderWidth, defaultButton* and *editorFont* fields. All other fields are ignored.

In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance. An application can offset and stretch this rectangle, to position it where ever it likes, within its window.

If *numCols* > *numFields,* then the request will fail.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified. The top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location. This occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect.* All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect.*

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information. This occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a push button instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the first parameter, *instance*, indicating which push button instance to process, and the third parameter, *data*, pointing to an *XEvent* structure.

The push button editor only handles an event if event maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within a push button instance, the first thing done is to determine if one of the individual push buttons was selected.

If one of the push buttons was selected, then that button will be drawn as active (filled), and the editor will then wait for the user to release the select button. When the user releases the select button, the push button will again be drawn as inactive (not filled). Afterwards, an input event will be added to the front of the application's input queue, informing it that one of the push buttons has been selected. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrPUSHBUTTON
value1	=	XrSELECT
value2	=	index of button which was selected
valuePtr	=	pointer to instance's editor structure

If a select occurs within a push button instance, but not within the boundary of a button, then the editor will do nothing but push an input event onto the front of the application's input queue. The event will notify the application that the editor was selected, but no action took place. It will also include the cursor position at the time the select event occurred. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrPUSHBUTTON
value1	=	NULL
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the push button editor will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

numCols > *numFields* [*XrINVALIDPARM*].

The width specified in *borderWidth* is zero [XrINVALIDPARM].

An invalid editor rectangle is specified [XrINVALIDRECT].

A call to 'X' failed { XCreatePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

numCols > *numFields* [*XrINVALIDPARM*].

The width specified in *borderWidth* is zero [XrINVALIDPARM].

MSG_RESIZE

An invalid editor rectangle is specified [XrINVALIDRECT].

MSG_GETSTATE, MSG_MOVE, MSG_RESIZE, MSG_GETITEMRECTS,

MSG_GETITEMSTATES, MSG_SETITEMSTATES and MSG_GETITEMCOUNT

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrRadioButton - an editor for controlling radio buttons

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrRadioButton (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The radio button editor is used to create and process a group of related radio buttons within a given window. The number of radio buttons, and how they are displayed, is completely controlled by the application. Each radio button may also include an optional text label, which will be displayed to the right of the button.

The size of the radio buttons are proportional to the size of the font being used to display the text labels. By allowing an application to specify this font, the size of the radio buttons can be adjusted for individual displays.

Radio buttons are very similar to checkboxes. But, there are at least two very distinct differences, which is why two different editors are provided:

Checkboxes are square, while radio buttons are round.

Checkboxes allow more than 1 item to be active at a time, while radio buttons allow only a single item to be active at any time.

When a new radio button is made active, the previously active button is made inactive. The editor will not allow the situation to occur where no button is active.

This field editor supports two distinct sets of state flags: those associated with the whole instance (which are modified using MSG_GETSTATE and MSG_SETSTATE), and those which are assigned to the individual radio buttons (which are modified using MSG_GETITEMSTATES and MSG_SETITEMSTATES). The state flags which are associated with the instance as a whole have precedence over the individual state flags. For example, if the **XrVISIBLE** flag is cleared for the instance, then none of the radio buttons will be displayed, regardless of their individual state flag settings. But, if the **XrVISIBLE** flag is set for the instance, then the field editor will look at the state of the **XrVISIBLE** flag associated with each radio button, to determine if the entity should be displayed.

Using the cursor and the mouse, a user will be able to interactively select a radio button, by moving the cursor over one of the buttons, and 'clicking' the mouse button. This will cause the chosen button to become 'active' (filled); the previously active button will now become 'inactive' (not filled). An application will then be notified of the change, and may handle it accordingly.

A radio button editor instance is composed of 3 components:

- The editor instance's background area.
- The individual buttons.
- The button labels.

Of these 'selectable' regions, only select events which occur in one of the individual radio buttons will affect the instance. When a radio button is selected, the editor will set the state of the button to 'active', and then the application program will be notified that the state of the radio button instance has changed. If a select event occurs in any of the other components, the application will be notified that a select occurred, but the editor instance was not modified.

When an editor instance is created, the index of the initially active radio button, and the label strings for each radio button will be specified by the application program. For the active index, a pointer to a 16 bit value must be passed in. For the label strings, a pointer to an array of NULL terminated character strings may be passed in. The radio button editor will not save copies of these pieces of data. Instead, it will save a copy of the pointers only. This will allow an application program to have immediate access to the index of the active radio button, since the value resides in the application's domain.

This editor will rely on the fact that when an application modifies this piece of shared data, the application will tell the editor to redraw; this allows the editor to remain in sync with the current radio button data. Modifying the data, without doing a redraw, will cause the editor to behave in an unfriendly fashion.

This editor allows an application program to obtain information about an instance in two ways:

By means of an input event returned when the active radio button is changed.

By means of the shared variable mentioned above. This refers to the value containing the index of the active radio button.

When the editor draws the specified radio button instance, it will attempt to space out the individual buttons, to take full advantage of the specified *editorRect*. If the rectangle is larger than needed, then the extra space will be divided equally as blank space between the rows and columns of buttons. If the rectangle is too small, then the request will fail.

EDITOR MESSAGES

MSG_NEW

This message is the means by which an application program can create a radio button editor instance in a window. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window       editorWindowId;
   RECTANGLE    editorRect;
   INT8         editorState;
   INT32        editorFGColor;
   INT32        editorBGColor;
   XFontStruct  * editorFont;
   INT16        numFields;
   INT16        numCols;
   INT8         ** labels;
   INT16        * value;
   INT8         * stateFlags;
} xrRadioButtonInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect
: This describes the location and size of the region into which the radio button instance is to be located. If the rectangle is larger than is needed to hold the specified instance, then the extra space will be used as padding between the individual radio buttons. If the rectangle is too small, then the request will fail.

editorState
: This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
: This field specifies the foreground color to be used when drawing the editor instance. This is used to display the button label, and to fill the interior of an active button. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
: This field specifies the background color to be used when drawing the editor instance. This is used to fill the interior of an inactive button. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont
: This is a pointer to a structure which describes the font to be used when creating this editor instance. If the pointer has been set to **NULL**, then the editor will use the default system base font. The font is important not only because it describes how the labels will look, but it will also be used to determine how large the buttons should be.

numFields
: This value indicates the total number of radio buttons which are included in this instance. Once specified, this value cannot be changed.

numCols
: This value indicates the number of columns into which the radio buttons are to be displayed. Using this value, and *numFields,* the editor will automatically calculate the number of rows. If this value is greater than *numFields,* then the create request will fail.

labels
: This is a pointer to an array of **NULL** terminated character strings, representing the labels associated with each radio button. If this pointer is **NULL**, then none of the radio buttons will have labels. The first entry in the array will correspond to radio button 0, the second to radio button 1, etc. If an entry in this array is **NULL**, then the corresponding radio button will have no label. The editor will not keep a copy of each of these strings; instead, it will only keep a copy of the pointer to the string array. It will rely on the application to NOT modify these strings, once the instance has been created.

value
: This is a pointer to a 16 bit integer value, representing the index of the active radio button. If this pointer is **NULL**, or if the value it points to is out of range, then the create request will fail. The valid range of values for this field is 0 to (numFields - 1). The editor will not keep a copy of this value. Instead, it will only keep a copy of the pointer to this value. Any time the active radio button is changed, the editor will update this value , and then notify the application. The application will then have immediate access to the index of the

active radio button, since the value resides in its data area.

stateFlags

This is a pointer to an array of values, each containing the initial state flag values to be associated with each radio button in the instance. If this pointer has been set to NULL, then the state flags for each entity will be set to **XrSENSITIVE** and **XrVISIBLE**. The first entry in this array corresponds to radio button 0, etc.

The editor will then draw the radio button editor instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure which was returned by the **MSG_NEW** message. This parameter specifies which instance is to be destroyed.

When a radio button instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified radio button editor instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given radio button editor instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If an instance is visible, but not sensitive, then each radio button will be drawn and filled with a 50% pattern.

MSG_GETITEMSTATES

This message allows an application to obtain a copy of the individual state flags associated with each of the radio buttons contained within the specified editor instance. These state flags differ from the state flags obtained using **MSG_GETSTATE.** The *instance* parameter must specify the editor instance to be queried, while the *data* parameter should point to an array of 8 bit integer values. The state flags will be returned by means of this array. The flags associated with radio button 0 will be returned in the first slot in the array, etc.

MSG_SETITEMSTATES

This message allows an application to modify the individual state flags associated with each of the radio buttons contained within the specified editor instance. Presently, only the **XrVISIBLE** and **XrSENSITIVE** flags are supported. The *instance* parameter must specify the editor instance which is to be modified, while the *data* parameter should point to an array of 8 bit integer values. The new state flags which are to be assigned to each entity within the instance will be obtained from this array. The value in the first slot of the array will be assigned to radio button 0, etc. After the new state flags have been saved, only those radio buttons whose state flags changed will be redrawn.

MSG_GETITEMCOUNT

This message allows an application to obtain a count, which indicates the number of individual radio buttons in the specified instance. The editor will assume that the *instance* parameter specifies the editor instance to query, and the *data* parameter points to a 32 bit integer value. The item count value will be returned by means of this integer.

This message is useful when used in conjunction with the **MSG_GETITEMRECTS** message. It allows an application to obtain the number of items in the instance, so that the application can then allocate enough memory to hold the rectangle information returned by **MSG_GETITEMRECTS.**

MSG_GETITEMRECTS

This message returns the coordinates for the rectangle which describes each of the individual radio buttons. these rectangles describe only the individual radio buttons - it does not include the labels. The message will expect the *instance* parameter to specify the editor instance to query, and the *data* parameter to point to a structure of the following format:

```
RECTANGLE  itemRects[x];
```

This array will then be filled with the rectangle information, and returned to the application.

Before an application can make this call, it needs to know the number of items in the specified instance, so that it can allocate a structure large enough to hold all of the rectangle information. The application should use the **MSG_GETITEMCOUNT** message to obtain this information. The application can then allocate an array large enough to hold all of the rectangle entries.

The order of the rectangle items returned in the array directly correlates to the order the items were originally created in. The first element in the array will describe button 0, the second button 1, etc.

This message is useful to those applications which have a need of knowing where the individual items in a radio button instance are located. The most common use would be by a forms controller, which would use the information to place an 'active field' indicator by a given item.

MSG_REDRAW

This editor provides an application with the means for requesting that an instance of radio buttons be redrawn. It will also allow an application program to set up a new active radio button index, and then request that only the new active button and the previous active button be redrawn. The current value, labels and font information will be used.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The radio button editor supports the following redraw modes:

- **XrREDRAW_ALL**

- **XrREDRAW_ACTIVE**

When the **XrREDRAW_ALL** option is specified, the complete editor instance will be redrawn. When the **XrREDRAW_ACTIVE** option is specified, only the previous and current active radio buttons will be redrawn.

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a radio button editor instance. The radio button editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window       editorWindowId;
   RECTANGLE    editorRect;
   INT8         editorState;
   INT32        editorFGColor;
   INT32        editorBGColor;
   XFontStruct  * editorFont;
   INT16        numFields;
   INT16        numCols;
   INT8         ** labels;
   INT16        * value;
   INT8         * stateFlags;
} xrRadioButtonInfo;
```

The only fields which *must* be filled out by the application program BEFORE issuing this call, are the *numFields*, *numCols*, *labels* and *editorFont* fields. All other fields are ignored.

Using the supplied information, the editor will determine the size of the rectangle needed to contain this instance. In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance. An application program can then offset this rectangle, to position it where ever it likes, within its window.

If *numCols* > *numFields*, then the request will fail.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified. The top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a radio button instance,

When such an event occurs, a MSG_EDIT message will be issued to the editor, with the *instance* parameter indicating which radio button instance to process, and the *data* parameter pointing to an *XEvent* structure.

The radio button editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within a radio button instance, the first thing done is to determine if one of the individual radio buttons was selected.

If one of the radio buttons was selected, then that button will become the 'active' choice, and the previous active choice will be made inactive; both buttons will be redrawn. Afterwards, an input event will be added to the front of the application's input queue, informing it that a new active radio button was selected. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRADIOBUTTON
value1	=	XrSELECT
value2	=	index of new active radio button
value3	=	index of previously active radio button
valuePtr	=	pointer to instance's editor structure

The editor will update the value of the active radio
button in the location shared with the application program.
This value is then directly available to the application program.

If a select occurs within a radio button instance, but not within
the boundary of a button, then the editor will do

nothing but push an input event onto the front of the application's
input queue.
The event will notify the application that the editor was selected,
but no action took place.
It will also contain the cursor position at the time the
select event occurred.
The returned
xrEvent
structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRADIOBUTTON
value1	=	NULL
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the radio button editor will fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

numCols > numFields [XrINVALIDPARM].

value is set to NULL [XrINVALIDPTR].

The radio button index, pointed to by the *value* field is out of range [XrPARMOUTOFRANGE].

An invalid editor rectangle is specified [XrINVALIDRECT].

A call to 'X' failed { XCreatePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

The radio button index, pointed to by the *value* field is out of range [XrPARMOUTOFRANGE].

A redraw mode other than **XrREDRAW_ALL** or **XrREDRAW_ACTIVE** is specified [XrINVALIDOPTION].

MSG_SIZE
data is set to NULL [XrINVALIDPTR].

numCols > *numFields* [*XrINVALIDPARM*].

MSG_RESIZE
An invalid editor rectangle is specified [XrINVALIDRECT].

MSG_GETSTATE, MSG_MOVE, MSG_RESIZE, MSG_GETITEMRECTS,

MSG_GETITEMSTATES, MSG_SETITEMSTATES and MSG_GETITEMCOUNT
data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE
The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN
HP

SEE ALSO
XrInput(3X), XrInit(3X)

NAME

XrRasterEdit - an editor that creates and controls a raster editor.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrRasterEdit (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The raster editor is used to create and process an instance of a raster editor within a given window. It will allow an application program to specify a raster image from as small as 1x1 pixels. The specified raster image need not be square.

When a raster editor instance is created, the application can specify how much of the raster image is to be displayed at any given time. This is referred to as the 'view region'. The view region can be from as small as 1 x 1, up to the size of the raster image. Each pixel within this view region will be expanded and displayed as a 'pixelSize'x'pixelSize' bit area, thus allowing a user to easily select and modify a pixel within the region. Each pixel will be surrounded by a one bit wide border.

The 'pixelSize' value is specified at the time the instance is created. It allows an application to size the raster editor instance, so that it will be usable in any sized display. Any value greater than or equal to 4 may be specified as the pixel expansion factor.

Using the cursor and the mouse, a user will be able to select a pixel to be changed in one of several ways:

By 'clicking' the **XrSELECT** mouse button, the pixel over which the cursor is located will be changed to whatever color the application has specified. By pressing the **XrSELECT** mouse button, and then dragging the cursor, any pixels over which the cursor passes will be changed to whatever color the application has specified. Releasing the mouse button will cause the action to stop, as will the receipt of any other X event.

If the raster image is larger than the view region, in either direction, then the editor will display and manage either/both a vertical and/or a horizontal scrollbar. The horizontal scrollbar will be located beneath the raster edit display, while the vertical scrollbar will be located to the right of the raster edit display. The raster editor will take sole responsibility for processing the scrollbars. However, any time the raster image is scrolled, an event will be returned to an application program, informing it of the new view origin.

These scrollbars provide the means by which a user can scroll through the complete raster area. There are currently 4 ways in which a scroll operation may be requested:

- The 'UP' (LEFT) arrow is selected.
- The 'DOWN' (RIGHT) arrow is selected.
- A position within the scroll area is selected.
- The slide box is interactively moved.

Depending upon which of the above methods is used, a different amount of scrolling may take place:

When either of the scroll arrows is selected, the view region will pan 1 pixel in the requested direction.

When a select occurs within the scroll area, the requested portion of the raster image will then be displayed.

When the slide box is selected, the raster image will not be updated, until the user has finished moving the slide box. When slide box movement has completed, the editor will redraw the instance, and display the requested portion of the raster area.

For each of the above events, the raster editor will return an input event, informing the application that the raster image has been scrolled, along with the new view origin.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a raster editor instance in a window. It will expect the *instance* parameter to be set to **NULL**, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window     editorWindowId;
   RECTANGLE  editorRect;
   INT8       editorState;
   INT32      editorFGColor;
   INT32      editorBGColor;
   INT32      pixelColor;
   INT32      pixelSize;
   SIZE       viewRegion;
   XImage     * imageData;
} xrRasterEditInfo;
```

editorWindowId
This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect
This describes the location and size of the region into which into which the raster editor is to be located. This must include space for the view region, and scrollbars, if needed. When the instance is laid out, the view region will be positioned starting at the upper left of the *editorRect*, the horizontal scrollbar will be positioned at the lower left, and the vertical scrollbar will be positioned at the upper right of the *editorRect*.

editorState
This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
This field specifies the foreground color to be used when drawing the editor instance. The foreground color is used to draw the border around each pixel in the raster image, and by the scrollbars. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
This field specifies the background color to be used when drawing the editor instance. The background color is used only by the scrollbars. If this is set to -1, the default

background color (see **XrInit(3X)**) will be used.

pixelColor
This field specifies the color to which a pixel should be set, when it is 'selected' by the user.

pixelSize
This describes the height and width into which each displayed pixel is to be expanded. This must be a value greater than or equal to 4. Any other value will cause the create request to fail.

viewRegion
This describes the height and width of the view region for the instance. The dimensions for these values may be in the range from 1 up to the corresponding dimension of the raster image. If a view region is specified which is larger than the raster image, then the request will fail.

imageData
This field is an XImage ID that points to an XImage structure. The XImage structure contains all of the information describing the raster image which is to be edited. This includes the size of the image (height and width), the depth of each pixel within the image, and a pointer to the actual image data.

The editor will then draw the raster editor instance in the specified window. The initial location of the view region is the upper right corner of the raster area.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE
This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure returned by MSG_NEW; this parameter specifies which instance is to be destroyed.

When a raster editor instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE
This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the raster editor instance indicated by the *instance* parameter. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed. If any scrollbars are being used by this instance, then their state will be identical to that of the raster editor instance.

MSG_SETSTATE
This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given raster editor instance, along with any scrollbars which may be in

use by this instance. The *instance* parameter specifies which editor instance is to be affected. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. The scrollbar editor will also be invoked, if necessary, to cause the scrollbars associated with this instance to be redrawn using the new state information. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible.

MSG_SETCOLOR

This message allows an application to specify the color to which a pixel should be set, when it is 'selected' by the user. The *data* parameter will be interpreted as a 32 bit integer value, containing the color information. The *instance* parameter specifies the editor instance to be affected.

MSG_REDRAW

Since the buffer containing the data describing the raster area resides in the application's domain, a means is provided to allow the application to change the data, and then tell the editor about it; the editor will then update the visible portion of the instance, to match the new data.

Using the **MSG_REDRAW** message, an application can force the editor to redraw a raster editor instance. The editor will use the data in the array pointed to by the *imageData*, structure, as the new raster information. The complete editor instance will be redrawn.

When this message is invoked, the *instance* parameter must point to the editor structure for the instance to be redrawn, and the *data* parameter must be a 32 bit integer which specifies the type of redraw which is to occur.

The raster editor supports only a single redraw mode:

- **XrREDRAW_ALL.**

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a raster editor instance. Depending upon the description supplied by the application program, this rectangle may hold only a raster editor instance, or it may also include up to 2 scrollbars. The raster editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   INT32       pixelColor;
   INT32       pixelSize;
   SIZE        viewRegion;
   XImage      * imageData;
} xrRasterEditInfo;
```

The only fields which *must* be filled out by the application program BEFORE issuing this call, are the *imageData*, *pixelSize*, and *viewRegion* fields. All other fields are ignored.

Using the supplied raster image height, width sizes, the view region height and width sizes, and the pixel size values, the editor will determine the size of the rectangle needed to contain this instance. If either the height or width of the raster image is greater than the corresponding view region dimension, then this rectangle will also take into account the scrollbars. In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance. An application program can then offset this rectangle, to position it anywhere within it's window.

Specifying a raster height or width or a view region height or width less than 1 will cause this request to fail.

MSG_POSITION

This message provides an application with the means for positioning the view region within the raster image. The *instance* parameter must point to the editor structure associated with the instance to be modified, while the *data* parameter must point to a POINT structure, containing the new origin of the view region; the top left corner of the view region will be positioned at the specified (x,y) position within the raster image.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified. The top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a RECTANGLE structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information. This occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a raster editor instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which raster editor to process, and the *data* parameter pointing to an *XEvent* structure.

The raster editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within the raster editor, the first thing done is to determine which of the following regions was selected:

- The raster editor region.
- The vertical scrollbar.
- The horizontal scrollbar.
- The background area.

Depending upon the region selected, a different action is taken:

Vertical scrollbar

The view box will be scrolled vertically within the raster area. Upon completion of the scroll operation, the following event will be pushed onto the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRASTEREDIT
value1	=	XrSCROLLBAR
valuePt	=	new view origin
valuePtr	=	pointer to instance's editor structure

Horizontal scrollbar

The view box will be scrolled horizontally within the raster area. Upon completion of the scroll operation, the following event will be pushed onto the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRASTEREDIT
value1	=	XrSCROLLBAR
valuePt	=	new view origin
valuePtr	=	pointer to instance's editor structure

Raster edit region

The selected pixel(s) will be modified, and an input event will be added to the front of the application's input queue, informing it that the raster area has been modified. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRASTEREDIT
value1	=	XrSELECT
valuePtr	=	pointer to instance's editor structure

If the raster was modified by the user dragging the
cursor, and the draw operation was terminated by the
receipt of an X event other than a select up event,
then the X event which caused the drag operation
to terminate will also be pushed on the front of the input
queue, BEFORE the xrEvent discussed above is pushed.
If the terminating event was a select up event, then the
field editor will swallow it.

Since the actual raster image data resides in the application's
domain, it can have immediate access to the data.

Background region

This implies that the select occurred within the
editorRect,
but not within any of the previously mentioned components.
An input event will be returned that includes the cursor position
at the time the select event occurred:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRASTEREDIT
value1	=	NULL
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the raster editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is an invalid size [XrINVALIDRECT].

A dimension of the raster image size or the view region size is less than 1 [XrINVALIDPARM].

A dimension of the view region is greater than the corresponding dimension of the raster image [XrINVALIDPARM].

pixelSize is an invalid value [XrPARMOUTOFRANGE].

The pointer to the image data has been set to NULL [XrINVALIDPTR].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw option other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

A dimension of the raster image size or the view region size is less than 1 [XrINVALIDPARM].

A dimension of the view region is greater than the corresponding dimension of the raster image [XrINVALIDPARM].

pixelSize is an invalid value [XrPARMOUTOFRANGE].

MSG_RESIZE

data is set to NULL [XrINVALIDPTR].

editorRect is an invalid size [XrINVALIDRECT].

MSG_POSITION

An invalid view region origin is specified [XrINVALIDPARM].

MSG_GETSTATE and MSG_MOVE

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrRasterSelect - an editor that displays and allows selection of

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrRasterSelect (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8      * data;
```

DESCRIPTION

The raster select editor is used to create and process an instance of a raster selector within a given window. It will allow an application program to specify an arbitrary number of 'n' X 'm' pixel rasters, of which one will be highlighted as the active selection.

The actual raster data is not supplied to the editor. Instead, the application must pass in the pixmap Id's associated with each of the raster images. It is up to the application to guarantee that it does not pass in an invalid pixmap Id.

Using the cursor and the mouse, a user will be able to interactively select a new active raster box, by moving the cursor over one of the raster boxes, and 'clicking' the mouse button. This will cause the 'active selection' highlight to be moved to the selected raster box. An application will then be notified of the change, and may handle it accordingly.

A raster select editor instance is composed of 4 components:

- The editor instance's border box.
- The editor instance's background area.
- The 'active selection' highlight.
- The individual raster boxes.

When one of the individual raster boxes is selected, the application program will be notified as to which raster box is now the active one. If a select event occurs in any of the other components, the application will be notified that a select occurred, but the editor instance was not modified.

The index of the active raster box is not saved locally by the editor. Instead, it will save a pointer to the 16 bit integer value which was specified by the application program at the time the instance was created. This allows the application to have immediate access to the index of the active raster box, since the information also resides in it's own data area. This editor will rely on the fact that when an application modifies this piece of shared data, it will tell the editor to redraw; this allows the editor to remain in sync with the current raster data. Modifying the data, without doing a redraw, will cause the editor to behave in an unfriendly fashion.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a raster select editor instance in a window. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   Pixmap      * rasterIds;
   INT32       rasterHeight;
   INT32       rasterWidth;
   INT16       rasterCount;
   INT16       colCount;
   INT16       * activeRaster;
} xrRasterSelectInfo;
```

editorWindowId
This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect
This field describes the location and size of the region into which the raster select editor is to be located. If the region is not large enough to hold the described instance, then the create request will fail. If the rectangle is larger than is needed, then the extra space will be used as padding between the boxes.

editorState
This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
This field specifies the foreground color to be used when drawing the editor instance. This is uses to draw all borders, and the active indicator. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
This field specifies the background color to be used when drawing the editor instance. This is uses to fill the background area within the If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

rasterIds
This is a pointer to an array of pixmap Ids, which identify each of the rasters to be displayed in this instance.

rasterHeight
This value describes the height of each of the raster images, in pixels.

rasterWidth
This value describes the width of each of the raster images, in pixels.

rasterCount
This indicates to the editor the total number of raster boxes to display, as part of this editor instance. There is no limit imposed on this value by the editor.

colCount
This specifies the number of columns to be used when displaying the raster editor instance. The row count will be calculated by the editor, using the *rasterCount* and *colCount* values.

activeRaster
This is a pointer to a 16 bit integer value, which indicates to the editor which of the raster boxes is to be highlighted as the active raster. The range of valid values for this field are:

[0 : rasterCount - 1]

where 0 indicates the first raster box, and (rasterCount - 1) indicates the last raster box. Specifying a value outside this range will cause the **MSG_NEW** request to fail.

The editor will save this pointer, and will update the value it points to any time a change occurs within the editor instance.

A check is made to make sure the *editorRect* specified is large enough to contain the specified instance. If the *editorRect* is not large enough, then the request will fail. To prevent this message from failing for this reason, the **MSG_SIZE** message should be used to create the *editorRect*. The application can then increase the height or width of the rectangle, if desired.

The editor will then draw the raster select editor instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE
This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure returned by MSG_NEW; this parameter specifies which instance is to be destroyed.

When a raster select instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE
This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified raster select editor instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE
This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given raster select editor instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flags. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus

making it invisible. If an instance is not sensitive, then the active indicator will not be displayed.

MSG_GETITEMCOUNT

This message allows an application to obtain a count, which indicates the number of raster select boxes in the specified instance. The editor will assume that the *data* parameter points to a 32 bit integer value; the item count value will be returned by means of this integer value. The *instance* parameter indicates the editor instance to query.

This message is useful when used in conjunction with the **MSG_GETITEMRECTS** message. It allows an application to obtain the number of items in the instance, so that the application can then allocate enough memory to hold the rectangle information returned by **MSG_GETITEMRECTS.**

MSG_GETITEMRECTS

This message returns the coordinates for the rectangle which describes each of the raster select boxes. The *instance* parameter will indicate the editor instance to be queried. The message will expect the *data* parameter to point to a structure of the following format:

```
RECTANGLE  itemRects[x];
```

This array will then be filled with the rectangle information, and returned to the application.

Before an application can make this call, it needs to know the number of items in the specified instance, so that it can allocate a structure large enough to hold all of the rectangle information. The application should use the **MSG_GETITEMCOUNT** message to obtain this information; the application can then allocate an array large enough to hold all of the rectangle entries.

The order of the rectangle items returned in the array directly correlates to the order the items were originally created in. The first element in the array will describe raster box 0, the second raster box 1, etc.

This message is useful to those applications which have a need of knowing where the individual items in a raster select instance are located. The most common use would be by a panel controller, which would use the information to place an 'active field' indicator by a given item.

MSG_REDRAW

This editor provides an application with a means to change the active raster box.

To select a different raster box as the active one, an application can simply place a new value in the integer value it set up to hold the active index, and then tell the editor to redraw, hilighting the new active box. This will force the editor to redraw the original active box, with the hi-light no longer around it, and then redraw the new active raster box, with the hi-light around it. If the new index is out of range, then the redraw will not take place, and the request will fail; the editor will restore the active index value to it's previous valid setting.

If an application would like the complete editor instance redrawn, then it could issue the redraw request, and specify that all of the raster boxes should be redrawn.

The *data* parameter must be a 32 bit integer value, which specifies the type of redraw to be performed.

The 2 redraw modes supported by this editor are:

- **XrREDRAW_ALL**
- **XrREDRAW_ACTIVE**

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a raster select editor instance. The raster select editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   Pixmap      * rasterIds;
   INT32       rasterHeight;
   INT32       rasterWidth;
   INT16       rasterCount;
   INT16       colCount;
   INT16       * activeRaster;
} xrRasterSelectInfo;
```

The fields which must be filled out by the application program BEFORE issuing this call, are the *rasterHeight*, *rasterWidth*, *rasterCount* and *colCount* fields; all other fields are ignored.

Using the supplied values, the editor will determine the size of the smallest rectangle needed to contain this instance. In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance; an application program can then offset or enlarge this rectangle, to position and size it however it likes, within it's window.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a RECTANGLE structure containing the new size and origin for the *editorRect.*

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_CHANGERASTER

This message provides the means for an application to change the raster image displayed in one of the raster select boxes. To replace the contents of an existing raster box, the application must specify both the index of the box to be modified, and a new pixmap Id for the image. It will be up to the application to free the old pixmap id on the server. using if desired.

This message will expect the *instance* parameter to specify which raster select instance is to be modified, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
  INT16   rasterIndex;
  Pixmap  pixmapId;
} newRaster;
```

The *rasterIndex* field should be set to the index of the raster box which is to be changed. The *pixmapId* field should be set to the pixmap Id associated with the new raster image.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a raster select instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which raster select editor to process, and the *data* parameter pointing to an *XEvent* structure.

The raster select editor only handles an event if it maps to an **XrSELECT** event, as described in **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within the raster select editor, the first thing done is to determine if one of the raster boxes was selected.

If one of the raster boxes was selected, then the visible 'active highlight' will be erased, and moved to the selected raster box. Afterwards, an input event will be added to the front of the application's input queue, informing it that one of the raster boxes was selected. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRASTERSELECT
value1	=	XrSELECT
value2	=	index of active raster box

value3	=	index of previously active raster box
valuePtr	=	pointer to instance's editor structure

The editor will update the active index memory location to contain the index of the new active raster box. This value is then directly available to the application program.

If a select occurs within a raster editor instance, but not within the boundary of a raster select box, then the editor will do nothing but push an input event onto the front of the application's input queue. The event will notify the application that the editor was selected, but no action took place. It will also include the cursor position at the time the select occurred. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrRASTERSELECT
value1	=	NULL
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the raster select editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is an invalid size [XrINVALIDRECT].

rasterIds is set to NULL [XrINVALIDPTR].

rasterHeight or *rasterWidth* is less than or equal to zero [XrINVALIDPARM].

rasterCount is set to zero [XrINVALIDPARM].

colCount > *rasterCount [XrINVALIDPARM]*.

activeRaster is set to NULL [XrINVALIDPTR].

activeRaster indicates an out of range index value [XrPARMOUTOFRANGE].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

The *data* parameter has been set to an unknown redraw mode [XrINVALIDOPTION].

An invalid active raster box index is specified [XrPARMOUTOFRANGE].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

rasterCount is set to zero [XrINVALIDPARM].

colCount > *rasterCount* [*XrINVALIDPARM*].

rasterHeight or *rasterWidth* is less than or equal to zero [XrINVALIDPARM].

MSG_RESIZE

data is set to NULL [XrINVALIDPTR].

editorRect is an invalid size [XrINVALIDRECT].

MSG_CHANGERASTER

rasterIndex is out of range [XrPARMOUTOFRANGE].

MSG_GETSTATE and MSG_MOVE

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrRectangle: (XrSetRect, XrSetPtRect, XrCopyRect, XrOffsetRect, XrInsetRect, XrSectRect, XrUnionRect, XrPtInRect, XrPt2Rect, XrEqualRect, XrEmptyRect) - calculations with rectangles.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>
```

XrSetRect (rect, x, y, width, height)
```
RECTANGLE * rect;
INT16 x;
INT16 y;
INT16 width;
INT16 height;
```

XrSetPtRect (rect, topLeft, botRight)
```
RECTANGLE * rect;
POINT * topLeft;
POINT * botRight;
```

XrCopyRect (srcRect, dstRect)
```
RECTANGLE * srcRect;
RECTANGLE * dstRect;
```

XrOffsetRect (rect, dx, dy)
```
RECTANGLE * rect;
INT16 dx;
INT16 dy;
```

XrInsetRect (rect, dx, dy)
```
RECTANGLE * rect;
INT16 dx;
INT16 dy;
```

XrSectRect (srcRectA, srcRectB, dstRect)
```
RECTANGLE * srcRectA;
RECTANGLE * srcRectB;
RECTANGLE * dstRect;
```

XrUnionRect (srcRectA, srcRectB, dstRect)
```
RECTANGLE * srcRectA;
RECTANGLE * srcRectB;
RECTANGLE * dstRect;
```

XrPtInRect (pt, rect)
```
POINT * pt;
RECTANGLE * rect;
```

XrPt2Rect (ptA, ptB, dstRect)
```
POINT * ptA;
POINT * ptB;
RECTANGLE * dstRect;
```

XrEqualRect (rectA, rectB)
RECTANGLE * rectA;
RECTANGLE * rectB;

XrEmptyRect (rect);
RECTANGLE * rect;

DESCRIPTION

XrSetRect
: Assign the four members of a rectangle. The result is a rectangle with values of *x, y, width, height.*

XrSetPtRect
: Assign the two boundary points to the rectangle. The result is a rectangle with the *x* and *y* fields set to the fields of the *topLeft,* parameter and the *width* and *height* fields set to the difference of the *topLeft* and *botRight* point parameters.

XrCopyRect
: Assign the coordinates of the source rectangle *srcRect* into the destination rectangle *dstRect.*

XrOffsetRect
: Move the rectangle *rect* by adding *dx* to each horizontal coordinate and *dy* to each vertical coordinate.

XrInsetRect
: Shrink or expand the rectangle *rect.* The left and right sides are moved in by the amount specified by *dx;* the top and bottom are moved towards the center by the amount specified by *dy.* If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (0, 0, 0, 0).

XrSectRect
: Calculate the rectangle that is the intersection of the two source rectangles *srcRectA* and *srcRectB,* place the result into the rectangle *dstRect* and return **TRUE** if they intersect or **FALSE** if they do not.

XrUnionRect
: Calculate the smallest rectangle which encloses both input rectangles and return the resulting rectangle in *dstRect.*

XrPtInRect
: Determine whether the point is in the rectangle. Return **TRUE** if it is and **FALSE** if it isn't.

XrPt2Rect
: Return the smallest rectangle which encloses the two input points.

XrEqualRect
: Compare the two rectangles and return **TRUE** if they are equal or **FALSE** if not.

XrEmptyRect
: Return **TRUE** if the given rectangle is an empty rectangle or **FALSE** if not. A rectangle is considered empty if the width or height fields of the rectangle are less than or equal to 0.

ORIGIN

HP

NAME

XrResource - the resource manager

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

XrResource (message, data)
INT32 message;
INT8 * data;
```

DESCRIPTION

The resource manager is used to add, find, or remove resource objects from an internally managed list. A resource object is identified by its functional type and a user-supplied resource number within that type.

Several of messages use the following structure for passing or returning data.

```
typedef struct
{
  UINT16   resourceType;
  INT32    resourceId;
  INT8     resourceState;
  INT32    resourceFile;
  INT8     * resourceObject;
} xrResourceInfo;
```

MSG_ADD

This message adds to the resource manager the resource identified by the *data* parameter, which points to an xrResourceInfo structure. If the *resourceObject* field contains a valid pointer, the *resourceFile* field will be set to **XrMEMORY** to indicate the resource is from application memory.

If the resource type and id fields in *data* match a resource already within the resource list, the resource in the list will be replaced with the new resource. If the location of the old resource is from application memory the *resourceObject* field within *data* will be set to point at the object being replaced. This allows an application to maintain control of an object being replaced in the resource list.

MSG_REMOVE

This message removes from the resource list the object identified by the *data* parameter, which points to a filled out instance of an xrResourceInfo structure. Only the *resourceType* and *resourceId* fields need to be set by the application. When the resource is located in the resource managers lists, the *resourceFile* field is checked and if it is set to XrMEMORY, the *resourceObject* field in *data* will be set to point to the object. If the object is not found in the resource list this message will fail.

MSG_FIND

This message locates the requested resource within the resource list. *data* is a pointer to an xrResourceInfo structure. The *resourceType* and *resourceId* fields of the structure identify the resource to be found. When the resource is located, the rest of the fields of the structure will be filled out.

MSG_SETSTATE

This messages is used to set the state of a resource contained within the resource manager. For this message, *data* is a pointer to an xrResourceInfo structure which has the *resourceType*, *resourceId*, and *resourceState* fields filled out. This message will search for the resource, and when it is found, set its *resourceState* to the new state.

The following two messages provide for the creation or destruction of application defined resource types.

MSG_NEWTYPE

This message allows an application to create a new resource type. *data* is a pointer to the following structure.

```
typedef struct
{
  UINT16 resourceType;
  INT32 (*resourceHandler)();
} xrResourceTypeInfo;
```

MSG_FREETYPE

This message destroys and removes a resource type from the resource type list. *data* contains the resource type identifier. This message will fail if the identifier is not in the range of 1 - 63999 or if the resource type is not defined or if the there are any resources still attached to the resource type.

RETURN VALUE

XrResource() returns **TRUE** when successful, **FALSE** otherwise.

ERROR CONDITIONS

If XrError() returns **FALSE** the *xrErrno* global will be set to one of the following values.

xrErrno will be set to **XrINVALIDMSG** for any messages besides the messages listed above.

xrErrno will be set to **XrINVALIDID** any time a resource id is invalid or cannot be found.

xrErrno will be set to **XrINVALIDTYPE** any time a resource type is invalid.

xrErrno will be set to **XrINVALIDSTATE** any time a resource is to be modified and a state flag setting prevents it.

xrErrno will be set to **XrINVALIDPARM** whenever the data parameter contains invalid information.

xrErrno will be set to **XrOUTOFMEM** if MSG_ADD cannot allocate the needed space for a resource.

ORIGIN

HP

NAME

XrScrollBar - an editor that creates and controls a scroll bar.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrScrollBar (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The scrollbar editor is used to create and process an instance of a scrollbar within a given window. Both vertical and horizontal scrollbars are supported by the editor. Scrollbars can be manipulated by selecting one of the direction arrows, interactively moving the slide box, or by selecting a location within the scroll region.

The components making up a scrollbar instance are configurable, and can be modified at any time after the instance has been created. These components include the following:

- Scroll arrows
- A variable sized slide box
- The upper and lower limits of the scroll region (or slide area)

The scrollbar editor provides an application with two modes of interactive slide operations:

Normal and Transparent

In *Transparent* mode, when the user wants to interactively move the slide box, only an outline of the slide box is moved; the real slide box is not moved.

In *Normal* mode, when the user wants to interactively move the slide box, the slide box itself is moved.

The editor also allows the application program to specify a handler routine, which will be automatically invoked during an interactive slide operation, each time the slide box position is changed by the specified number of scroll region units. This allows an application to update its window contents as the slide box is moved.

The scroll region units are defined by the application; this allows an application to supply the minimum and maximum value for the scroll region. For a horizontal scrollbar, the minimum value is the left edge of the region, and the maximum value is the right edge of the region; for a vertical scrollbar, the minimum value is the upper edge of the region, and the maximum value is the lower edge of the region. Any time the slide box position is referenced, it will be specified in terms of these maximum and minimum values. All integer values in this range are referred to as scroll region units.

Both the height and the width of a scrollbar are definable by an application program. There are no real upper limits imposed upon a scrollbar instance, but there are lower limits. The limits relate to the ratio between the height and width of a scrollbar. For a vertical scrollbar, the minimum height of a scrollbar is calculated using the width of the scrollbar rectangle; for a horizontal scrollbar, the minimum width of a scrollbar is calculated using the height of the scrollbar rectangle. If the specified *editorRect* is not large enough to contain the requested scrollbar instance, then the create request will fail. If MSG_SIZE is used to obtain the minimum rectangle, then a create request should never fail for the above mentioned reasons.

The existing scrollbar behaves more like a valuator than a real scrollbar: it allows both the user and the application to place the slide box anywhere in the range of the upper and lower limits of the scroll region.

With the addition of a new state flag, **XrFIXEDORIGIN**, an application now has access to a new type of scrollbar; a fixed origin scrollbar. Fixed origin scrollbars differ from the "valuator" type scrollbars in the following manner: when the slide box is told to be positioned at value 'x', the origin of the slide box (upper edge for a vertical scrollbar, and left edge for a horizontal scrollbar) will be placed at that location. This type of scrollbar is particularly useful for applications displaying text, since the value of the slide box tells the application which line should be the first line displayed, and the size of the slide box indicates how many lines are visible. Another difference between the two types of scrollbars is that a fixed origin scrollbar only allows the slide box to be assigned a value in the range *min to* (*max - slide box size + 1*).

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a scrollbar instance in a window. It will expect the *instance* parameter to be set to **NULL**, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   INT8          orientation;
   xrSBParameters configuration;
} xrScrollBarInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the scrollbar instance. The rectangle must be at least large enough to hold a minimally sized scrollbar; if it is not, then the create request will fail. To prevent this from happening, an application should use the **MSG_SIZE** message to obtain the minimal rectangle, and it can then stretch and offset the rectangle, to obtain the desired size scrollbar.

editorState

This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE**, **XrVISIBLE**, **XrFIXEDORIGIN**, and **XrTRANSPARENT** flags.

editorFGColor

This field specifies the foreground color to be used when drawing the editor instance. This is used for all borders, and for filling the slide box. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor

This field specifies the background color to be used when drawing the editor instance.

This is used for the interior of the scrollbar region. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

orientation
This describes the orientation of the scrollbar instance, and can assume one of the following values: **XrHORIZONTAL** or **XrVERTICAL.**

configuration
This field specifies all of the configuration parameters needed to initially display the scrollbar instance. This field is an instance of the following structure:

```
typedef struct {
     INT8  components;
     INT16 min;
     INT16 max;
     INT16 slidePosition;
     INT16 slideSize;
     INT32 (*handler)();
     INT16 granularity;
} xrSBParameters;
```

The *components* field specifies the component parts which are to be initially displayed as part of the scrollbar instance. It is constructed by OR'ing together a combination of the **XrSLIDEBOX** and **XrSCROLLARROWS** flags.

The *min* and *max* values describe the units which are to be assigned to the top (left) and bottom (right) of the scroll region.

The *slidePosition* value specifies the initial position of the slide box within the scroll region. The position is expressed in scroll region units, and must be in the range *min to max* if not using a fixed origin scrollbar, or in the range *min to* (*max - slide box size* + *1*) if using a fixed origin scrollbar. If an invalid position is specified, then the create request will fail. If the **XrSLIDEBOX** flag is not set in the *components* field, then this field is ignored.

The *slideSize* value specifies the initial size of the slide box. It is expressed in scroll region units, and must be in the range *0 to* (*min - max*); if an invalid size is specified, then the create request will fail. If the **XrSLIDEBOX** flag is not set in the *components* field, then this field is ignored.

The *handler* field allows an application to specify its own handler routine, which will be automatically invoked by the scrollbar editor during interactive slide operations. If the *handler* field is set to **NULL**, then the scrollbar editor will disable this feature.

When the *handler* routine is invoked, it will be called in the following fashion:

```
INT32
handler (windowId, editorInstance, slidePosition)

  Window    windowId;
  xrEditor * editorInstance;
```

```
        INT16      slidePosition;
```

> The *granularity* value represents a value in the range *0 to* (*min* - *max*), where *min* and *max* are the limits of the scroll region.

The editor will then draw the scrollbar instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, which was returned by the MSG_NEW message; this parameter specifies which instance is to be destroyed.

When a scrollbar instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE, XrSENSITIVE, XrFIXEDORIGIN** and **XrTRANSPARENT** flags for the specified scrollbar instance. The *instance* parameter specifies which instance to inquire about. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE, XrVISIBLE, XrFIXEDORIGIN** and **XrTRANSPARENT** flags, for a given scrollbar instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, if either the **XrSENSITIVE** or **XrVISIBLE** flags changed. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If an instance is visible, but not sensitive, then only the outline of the scrollbar,the slide box, and the scroll arrows will be drawn. The **XrTRANSPARENT** flag controls the behavior of the slide box during interactive slide box move operations. The **XrFIXEDORIGIN** flag controls the operational behavior, as described earlier.

MSG_REDRAW

This editor provides an application with the means for requesting that a particular scrollbar instance be redrawn.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The scrollbar editor supports the following redraw mode:

- **XrREDRAW_ALL**

When the **XrREDRAW_ALL** option is specified, the complete scrollbar instance will be redrawn; similar to when the instance was first displayed.

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a minimally sized scrollbar of a specified width (if vertical) or height (if horizontal). Once an application program has obtained this information, it can stretch and offset the rectangle, thus customizing it to it's own needs. It expects the *instance* parameter to be set to **NULL**, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   INT8          orientation;
   xrSBParameters configuration;
} xrScrollBarInfo;
```

One field which must be filled out by the application program BEFORE issuing this call, is the *orientation* field; it must be set to either **XrVERTICAL** or **XrHORIZONTAL**.

If a vertical scrollbar was specified, then the width of the desired scrollbar must be specified. This is done by setting the following value in the structure:

editorRect.width

If the width of a vertical scrollbar is less than 11, the request will fail.

If a horizontal scrollbar was specified, then the height of the desired scrollbar must be specified. This is done by setting the following value in the structure:

editorRect.height

If the height of a horizontal scrollbar is less than 11, the request will fail.

In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain an instance of the smallest sized scrollbar with the specified orientation and height or width.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect.* All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect.*

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_GETPARAMETERS

This message provides an application with the means for obtaining the current configuration parameters associated with a particular scrollbar instance, or the means for initializing a parameter structure with the default configuration values. If an application wishes to obtain the current configuration parameters for an existing scrollbar instance, then it should set the *instance* parameter such that it specifies the instance to be queried; if an application wishes to only initialize a parameter structure with the default values, then it should set the *instance* parameter to NULL. The *data* parameter must to point to an instance of the following structure:

```
typedef struct {
     INT8  components;
     INT16 min;
     INT16 max;
     INT16 slidePosition;
     INT16 slideSize;
     INT32 (*handler)();
     INT16 granularity;
} xrSBParameters;
```

Each field in this structure will be filled in with the current configuration information for the specified instance, or the default value.

The default value associated with each field is shown below:

components = XrSCROLLARROWS | XrSLIDEBOX
min = 0
max = 100
slidePosition = 0
slideSize = 10
handler = NULL
granularity = 10

MSG_SETPARAMETERS

This message provides an application with the means for modifying the configuration parameters associated with a particular scrollbar instance. It will expect the *instance* parameter to specify the instance to be queried, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
     INT8  components;
     INT16 min;
     INT16 max;
     INT16 slidePosition;
     INT16 slideSize;
     INT32 (*handler)();
     INT16 granularity;
} xrSBParameters;
```

If the **XrSLIDEBOX** flag in the *components* field is not set, then the values in the *slidePosition, slideSize, handler* and *granularity* fields will be ignored.

After the values have been saved, the affected components within the instance will be redrawn. The whole instance is not redrawn.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a scrollbar instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which scrollbar to process, and the *data*, parameter pointing to an *XEvent* structure.

The scrollbar editor only handles a keystroke if it maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored. When a select event occurs within the scrollbar, the first thing done is to determine which of the following regions was selected:

- A scroll arrow.
- The slide box.
- The scroll region.

Depending upon the region selected, a different action is taken:

A scroll arrow box

If the scroll arrows are displayed as part of the instance, then an input event will be added to the front of the application's input queue, informing it that the particular scroll arrow was selected. The slide box position is not modified. The returned *xrEvent* structure is set to the following values:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR

inputCode	=	XrSCROLLBAR
value1	=	(see values listed below)
value2	=	current slide box position
valuePtr	=	pointer to instance's editor structure

The *value1* field will return an indication of which scrollbar arrow was selected. It will be set to one of the following values:

XrSCROLL_LEFT
XrSCROLL_RIGHT
XrSCROLL_UP
XrSCROLL_DOWN

If an application then wants the slide box position modified, it should calculate the new slide box position, and then use the **MSG_SETPARAMETERS** message to reposition the slide box.

An application could easily implement smooth, continuous scrolling, by polling for a select up event after it has received one of the above scroll arrow events. When the select up event is received, the application would know to stop the continuous scrolling.

Slide box

When the slide box is selected, the scrollbar editor will keep control, until any X event is received. Normally, it is a select up event which terminates the slide operation.

If the scrollbar is currently configured to operate in *Normal* mode, then as the user moves the cursor, the slide box will follow. If the slide operation terminates because a select up event is received, then the *xrEvent* event described below will be pushed onto the front of the application's input queue. If the slide operation terminates because any other X event is received, then the X event will be re-pushed onto the front of the input queue, and then the *xrEvent* event described below will be pushed. The *xrEvent* event, will be set to contain the following information:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrSCROLLBAR
value1	=	XrSCROLL_SLIDE
value2	=	new slide box position
valuePtr	=	pointer to instance's editor structure

If the scrollbar is currently configured to operate in *Transparent* mode, then as the user moves the cursor, an outline of the slidebox will follow. If the slide operation terminates because a select up event is received, then the *xrEvent* event described below will be pushed onto the front of the application's input queue. If the slide operation terminates because any other X event is received, then the X event will be re-pushed onto the front of the input queue, and then the *xrEvent* event described below will be pushed. The position of the slide box will not be changed. The position of the slide box outline will be returned;

it is up to the application to then instruct the scrollbar editor to move the slide box to that position, if desired. The *xrEvent* structure will contain the following information:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrSCROLLBAR
value1	=	XrTSCROLL_SLIDE
value2	=	current slide box position
value3	=	slide box outline position
valuePtr	=	pointer to instance's editor structure

For both of the above modes, as the slide box is moved, each time its position changes by the specified granularity, the specified interactive slide handler routine will be invoked.

Scroll region

When a select occurs within the scroll region portion of a scrollbar, the editor will return an input event to the application, informing it that a select occurred within the instance's scroll region. The editor will not automatically reposition the slide box. It is up to the application, once it has received notification, to update the slide box position. Returned as part of the input event, is the location of the cursor at the time of the select, expressed as in scroll region units.

The *xrEvent* structure will contain the following information:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrSCROLLBAR
value1	=	XrSCROLL_LESS or XrSCROLL_MORE
value2	=	current slide box position
value3	=	current location (in slide region units)
valuePtr	=	pointer to instance's editor structure

NOTE: if the slide box component is not currently being displayed, then the *value1* field will always be set to **XrSCROLL_MORE.**

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the scrollbar that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is not large enough to contain the scrollbar [XrINVALIDRECT].

The width for a vertical scrollbar is less than 11 [XrINVALIDPARM].

The height for a horizontal scrollbar is less than 11 [XrINVALIDPARM].

orientation is set to an unknown value [XrINVALIDOPTION].

granularity is outside the scrollbar unit range [XrPARMOUTOFRANGE].

max < = min [XrINVALIDPARM].

slidePosition is outside the scrollbar unit range [XrPARMOUTOFRANGE].

slideSize is outside the scrollbar unit range [XrPARMOUTOFRANGE].

A call to 'X' failed { XMakePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_RESIZE

data is set to NULL [XrINVALIDPTR].

editorRect is not large enough to contain the scrollbar [XrINVALIDRECT].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

The width for a vertical scrollbar is less than 11 [XrINVALIDPARM].

The height for a horizontal scrollbar is less than 11 [XrINVALIDPARM].

orientation is set set to an unknown value [XrINVALIDOPTION].

MSG_SETPARAMETERS

data is set to NULL [XrINVALIDPTR].

max < = min [XrINVALIDPARM].

slidePosition is outside the scrollbar unit range [XrPARMOUTOFRANGE].

slideSize is outside the scrollbar unit range [XrPARMOUTOFRANGE].

granularity is outside the scrollbar unit range [XrPARMOUTOFRANGE].

MSG_GETSTATE, MSG_MOVE and MSG_GETPARAMETERS

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW, MSG_GETPARMETERS and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrStaticRaster - an editor which displays a static raster image

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrStaticRaster (instance, message, data);
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The static raster editor provides an application with the means for placing uneditable raster images anywhere within the bounds of a window.

The application must specify a rectangular region, into which the raster image will be drawn. If the raster image will not fit exactly within the rectangle, then the create request will fail.

The actual raster data is not supplied to the editor. Instead, the application must pass in the pixmap Id associated with the raster image. The pixmap Id is obtained by registering the raster data with the appropriate server. Refer to the "X Library" chapter in the "Xlib - C Language Interface, Protocol Version 11" document for a further discussion on this subject. It is up to the application to guarantee that it does not pass in an invalid pixmap Id.

Once a static raster image has been displayed, the only attributes which may be modified are the **XrVISIBLE** and **XrSENSITIVE** state flags. The actual raster data cannot be modified.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a static raster image within a window. It will expect the *instance* parameter to be set to **NULL**, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   INT32       rasterHeight;
   INT32       rasterWidth;
   Pixmap      rasterId;
} xrStaticRasterInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the region into which the static raster image is to be displayed. If the raster image will not fit exactly within this rectangle, then the create request will fail.

editorState
This field contains the initial value of the state flags for this editor instance. If can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
This field specifies the foreground color to be used when drawing the editor instance. This value is unused by this field editor. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
This field specifies the background color to be used when drawing the editor instance. This value is unused by this field editor. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

rasterHeight
This specifies the height of the raster image, in pixels.

rasterWidth
This specifies the width of the raster image, in pixels.

rasterId
This contains the pixmap Id associated with the static raster data. The pixmap Id is obtained by registering the raster data with appropriate server.

The editor will then draw the static raster image in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE
This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure returned by MSG_NEW; this parameter specifies which instance is to be destroyed.

When a static raster instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE
This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the static raster instance indicated by the *instance* parameter. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE
This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for the static raster instance indicated by the *instance* parameter. The *data*

parameter is interpreted as an 8 bit integer value, containing the new state flag value. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible.

MSG_REDRAW

This message will provide an application with the means for requesting that a static raster image be redrawn.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The static raster editor supports the following redraw mode:

- **XrREDRAW_ALL**

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the height and width of the rectangle needed to contain a given static raster image.

The static raster editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   INT32       rasterHeight;
   INT32       rasterWidth;
   Pixmap      rasterId;
} xrStaticRasterInfo;
```

The fields which must be filled out by the application program BEFORE issuing this call, are the *rasterHeight* and *rasterWidth* fields.

In return, the *editorRect* field will be filled in with the coordinates for the 0 based rectangle needed to contain the instance; an application program can then offset this rectangle, to position it where ever it likes, within it's window.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routines, when an input event occurs within a static raster instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which static raster image to process, and the *data* parameter pointing to an *XEvent* structure.

The static raster editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton(3X)** and **XrInit(3X)**; all others are ignored.

When a select event occurs within a static raster image, the editor will do nothing but add an input event to the front of the application's input queue, informing it that a static raster image was selected. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrSTATICRASTER
value1	=	XrSELECT
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the static raster editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

rasterId is an invalid Id [XrINVALIDID].

editorRect is an invalid size [XrINVALIDRECT].

rasterHeight or *rasterWidth* is less than or equal to zero [XrINVALIDPARM].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW
A redraw mode other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_SIZE
data is set to NULL [XrINVALIDPTR].

rasterHeight or *rasterWidth* is less than or equal to zero [XrINVALIDPARM].

MSG_GETSTATE and MSG_MOVE
data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE
The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrStaticText - an editor which displays a block of static text

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrStaticText (instance, message, data);
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The static text editor provides an application with the means for placing uneditable blocks of text anywhere within the bounds of a window. In addition, the font to be used when displaying the text, and the type of alignment to use, can also be specified. The application must specify a rectangular region into which the text will be drawn. If the text will not fit completely within the rectangle, then only that portion which fits will be displayed.

Once a block of static text has been displayed, the only attributes which may be modified are the **XrVISIBLE** and **XrSENSITIVE** state flags. The actual textual data cannot be modified.

Four forms of text alignment will be provided by the editor:

- None
- Centered
- Left
- Right

Alignment will be done in relation to the rectangle passed in at the time the text request is made. The rules governing alignment are:

Centered
: The text will be displayed such that the center of each line of text is positioned at the center of the specified rectangle. Before the text is displayed, all leading and trailing spaces in the line will be stripped.

Left
: Each line of text will be displayed such that the first character of the line is positioned at the left most edge of the specified rectangle. Before the text is displayed, all leading and trailing spaces in the line will be stripped.

Right
: Each line of text will be displayed such that the last character in each line ends at the right most edge of the specified rectangle. Before the text is displayed, all leading and trailing spaces in the line will be stripped.

None
: Each line of text will be displayed such that the first character of the line is positioned at the left most edge of the specified rectangle. Leading and trailing spaces in a line of text are not stripped.

When the editor displays a block of static text, the initial pen position will be in the upper left hand corner of the *editorRect.* The bounds used when displaying the text are the right hand edge of the rectangle, and the bottom edge.

Each time the pen position goes beyond the right edge of the *editorRect,* or each time one of the following characters is encountered in the string, a new line will be started:

```
Newline        (octal 012)
Vertical tab   (octal 013)
Form feed      (octal 014)
Carriage return (octal 015)
```

If, at some point, the pen's y position goes below the bottom edge of the *editorRect,* then no more lines of text will be displayed.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a block of static text within a window. It will expect the *instance* parameter to be set to **NULL,** and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
    Window        editorWindowId;
    RECTANGLE     editorRect;
    INT8          editorState;
    INT32         editorFGColor;
    INT32         editorBGColor;
    XFontStruct   * editorFont;
    INT8          * string;
    INT32         alignment;
} xrStaticTextInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the region into which the block of static text is to be formatted and displayed. If the text will not fit completely within this rectangle, then the text will be clipped, until it does fit. However, if a rectangle with a width of 0 is specified, then the editor will refuse to handle the request, and it will fail.

editorState

This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor

This field specifies the foreground color to be used when drawing the editor instance. The foreground color is used to display the text. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor

This field specifies the background color to be used when drawing the editor instance. The background color is used to paint the area upon which the text is displayed. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont
This points to the structure which contains all of the information necessary to describe the font to be used when displaying the block of static text. If this pointer is NULL, then the default system base font will be used.

string
This should be a pointer to the NULL terminated block of text. If the pointer is NULL, or if the string has a length of 0, then the create request will fail; a string must be specified! A block of text can be composed of just about any characters; however, the editor will expect words to be separated by spaces, in order for it to properly parse the block. Unpredictable results will occur if words are not separated by spaces. Forced newline characters (Carriage return, Newline, Vertical tab and Form feed) can be included in the block of text; these will force the pen to the next line, even though the right edge of the *editorRect* was not reached.

alignment
This value will specify to the editor what form of text alignment should be used when displaying each line of text. The following values will be accepted:

- XrNO_ALIGNMENT
- XrCENTER_ALIGNED
- XrLEFT_ALIGNED
- XrRIGHT_ALIGNED

Any value other than one of those outlined above, will cause the create request to fail. Each line of static text will use this alignment.

The editor will then draw the block of static text in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE
This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, which was returned by MSG_NEW; this parameter specifies which instance is to be destroyed.

When a static text instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handler mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE
This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified block of static text . The *instance* parameter specifies which instance to use. The

data parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given block of static text. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible.

MSG_REDRAW

This editor provides an application with the means for requesting that a block of static text be redrawn. The text string data, font information, and the alignment which were specified when the instance was created, will be used.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The static text editor supports the following redraw mode:

- **XrREDRAW_ALL**

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the height of the rectangle needed to contain a given block of static text. The width of the rectangle MUST have been specified by the application program, via the *width* field within *editorRect*, BEFORE it issued this message. If the rectangle is specified with a width of 0, or the length of the string is 0, or the string pointer is NULL, then the request will fail.

The static text editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   XFontStruct  * editorFont;
   INT8         * string;
   INT32         alignment;
 } xrStaticTextInfo;
```

The fields which must be filled out by the application program BEFORE issuing this call, are the *string*, *alignment*, and *editorFont* fields. In addition, the width of the rectangle into which the block of static text is to be fit, must also be filled out. This involves setting the following field of the *editorRect:*

editorRect.width

The editor will set the origin (top and left) of the rectangle to (0,0).

If *editorFont* has been set to NULL, then the default system base font will be used when calculating the rectangle size; if a particular font is requested, then it will be used.

The rectangle calculated will be the smallest 0 based rectangle of the specified width, into which the specified block of static text will fit, using the specified font and text alignment mode. Only the height of the rectangle needs to be calculated, since the width was supplied by the application, and the origin (top and left) of the rectangle is forced to (0,0).

Since the width of the rectangle is fixed by the calling application, it is possible for a single word to be too long to fit on a line. If this occurs, then the editor will return with an error; the editor will not break up a word.

An application can then take this rectangle, and offset it, so that the origin of the block of text can be set.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the toolbox input routines, when an input event occurs within a static text instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which block of static text to process, and the *data* parameter pointing to an *XEvent* structure.

The static text editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton**(3X) and **XrInit**(3X); all others are ignored.

When a select event occurs within a block of static text, the editor will do nothing but add an input event to the front of the application's input queue, informing it that a block of static text was selected. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrSTATICTEXT
value1	=	XrSELECT
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the static text editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is specified with a height or width of zero [XrINVALIDRECT].

The *editorRect* width is too small [XrINVALIDRECT].

string is set to NULL [XrINVALIDPTR].

string points to an empty string [XrINVALIDPARM].

alignment is set to an unknown value [XrINVALIDOPTION].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_RESIZE

data is set to NULL [XrINVALIDPTR].

editorRect is specified with a height or width of zero [XrINVALIDRECT].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

The width of the rectangle has been set to zero [XrINVALIDRECT].

The width of the rectangle is too small [XrINVALIDRECT].

string is set to NULL [XrINVALIDPTR].

string points to an empty string [XrINVALIDPARM].

MSG_GETSTATE and MSG_MOVE

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

XrTextEdit - an editor for entering and editing a line of text.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrTextEdit (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The text edit editor provides an application with the means for placing a single line of editable text anywhere within the bounds of a window. In addition to allowing the text string to be specified, an optional field label and font may also be specified. The application must pass in a definition for the rectangular region into which the editor instance will be drawn. If the rectangle is not the correct size to hold the instance, then the editor will refuse to create the instance. To prevent this from happening, an application should use the **MSG_SIZE** message to obtain this rectangle.

ACTIVE FIELD

The text edit editor supports the concept of an active field. When a given text editor instance has control of the input stream, it is considered to be active, and it will indicate this by visibly changing its format. The means by which it indicates this fact is twofold:

> It will display a text cursor at the current character position within the instance.
>
> It will redraw the border for the text string (which is usually 1 pixel wide) as a 3 pixel wide border.

There are two means by which a text edit field is told to become active:

> By issuing a MSG_ACTIVATE request to the editor.
>
> By issuing a MSG_EDIT request to the editor, and passing it any input event understood by the text editor.

COMPONENTS

The two most important pieces of a text editor instance are the text cursor and the text string being edited. The purpose of the text cursor, is to indicate to the user where the next character typed will be placed, and also what portion of the text string will be modified when one of the editing keys is pressed. As the user types keys, the information will be added at the location of the text cursor; if insert mode is enabled, then those characters to the right of the text cursor will be shifted one position further to the right - otherwise they are left unchanged. After the new character has been placed, the text cursor will be moved to the right of the newly added character. Only printable ASCII characters will be displayed as part of a text string. Once the number of characters in the string has reached the maximum allowable number (as specified when the instance was created), no more characters will be added to the string.

When the instance is first created, the application program will pass in a pointer to a buffer, which will be used by the editor to store the current value of the string typed by the user; the buffer must be large enough to hold the whole string, plus a terminating NULL. When the buffer is passed in, it must contain the initial value for the field; it must also be NULL terminated. If the field is to be initially empty, then the first byte of the buffer must be the NULL character. If the specified string is too long to fit within the field, then it will be truncated, until it does fit.

TEXT INSERTION AREA

The text insertion area is the portion of a text edit instance in which the editable string is displayed. The width of this area is dependent upon two factors: the maximum number of characters which may be entered, and the character cell width; both of these are specified by the application when an instance is created.

Working with proportional fonts can be tricky, since the character cell width differs from character to character. What this means is that if a text edit instance needs to hold 20 characters, it's impossible to determine the width of the editing region, without leaving a lot of blank space.

This editor tackles this problem by allowing the application to specify the character cell width to use. If an application wants to guarantee that all characters typed will be visible, then it should set the cell width to XrMAXWIDTH, when the instance is created. If an application wants all characters to be visible most of the time, then it can set the cell width to XrAVGWIDTH. A third option is that the application can specify a particular cell width to use (specified in pixels).

If a user types into a text edit field, and his information will not fit within the text insertion area, the cursor will turn off; the editor will continue to collect the input, it just will not be displayed. The cursor will remain invisible, until it is repositioned back in the visible portion of the editing region.

EDITING MODES

The text editor supports two modes of character entering: normal and insert mode.

When the instance is operating in 'normal' mode, the text cursor is displayed as an inverse video rectangle, and will appear on top of the character at the current cursor position. When a new character is entered, it is placed at the cursor position, and will overwrite any previous character at that position; those characters to the right of the text cursor are left unchanged. Afterwards, if the text cursor is not already at the end of the text edit field, then the text cursor will be moved one position to the right.

When the instance is operating in 'insert character' mode, the text cursor is displayed as a vertical bar, and will appear between two character positions. When a new character is entered, all characters to the right of the insertion bar are shifted one position to the right, and the character is then placed at the insertion bar position. Afterwards, if the text edit field is not already full, then the insertion bar is moved one position to the right.

Insert mode is entered and exited by pressing the 'insert mode' key on your keyboard; this mode acts as a toggle.

EDITING

The text editor supports most of the standard line editing commands which a user may generate from the keyboard. Among the editing commands which will be supported are the following:

Cursor Left
: This will cause the text cursor to be moved one character position to the left, within the text area.

Cursor Right
This will cause the text cursor to be moved one character position to the right, within the text string displayed in the text area.

Backspace
This will cause the character to the left of the text cursor to be deleted, and all characters to its right to be shifted one position to the left; the text cursor will also move one position to the left.

Delete Char
This will cause the character underneath the text cursor to be deleted, and all characters to its right will be shifted one position to the left; the text cursor will remain in its same position.

Delete Line
This will clear all characters from the text region, and will move the text cursor back to the beginning of the text area.

Clear Line
This will cause all characters under and to the right of the text cursor to be deleted. The location of the text cursor will be unchanged.

MOUSE

As was mentioned above, the text cursor's position is automatically updated after each character placement operation or editing request. There is, however, another method for specifying a new text cursor position. By using the mouse, you can move the cursor to anywhere within the current text string, and then 'click' the mouse button. This will cause the text cursor to be placed at the selected character position. If a position past the end of the text string is selected, then the text cursor will automatically be placed after the last character in the string.

DEACTIVATION

Once a text edit field has been made 'active', there will be several ways for it to become 'inactive'. When a text edit field is deactivated, it will turn off its text cursor, and it will redraw the text border as a 1 pixel wide box. The instance will no longer accept input, until it has been reactivated.

The first method for deactivating a text field, is to use the mouse to select a region outside the editor's rectangular region; this region includes the text region and the optional label. When the text editor code determines that a select event occurred outside its domain, it will deactivate the field, and then place two input events onto the front of the application's input queue:

Input event 1
This first input event placed on the input queue will be a copy of the select event which caused the field to be deactivated. This event will notify the application that a select occurred somewhere else within the display.

Input event 2
This will be the event information generated by the text edit editor. It will inform the application that the field has been exited, and that editing is now complete.

A second method of deactivating a text field, is by pressing one of the special keys, which are collectively known as 'traversal' keys. Traversal keys are frequently used by a forms controller, to provide a means for a user to exit a field in a form, and then advance to the next field. The following traversal keys are defined:

Return
When the 'return' key is pressed, the text editor will deactivate the active field, and return an input event. The input event will be used to notify the application program that a text field was exited. If the application is a forms controller, then it can use this information to force the next field in the form to become active.

Tab
When the 'tab' key is pressed, the text editor will deactivate the active field, and return an input event. The input event will be used to notify the application program that a text field was exited. If the application was a forms controller, then it can use this information to force the next field in the form to become active.

Backtab
When the 'backtab' key is pressed, the text editor will deactivate the active field, and return an input event. The input event will be used to notify the application program (and the forms controller) that a text field was exited. If the application is a forms controller, then it can use this information to force the previous field in the form to become active.

When a traversal key causes a field to be exited, only a single input event is added to the application's input queue.

RETURNED INFORMATION

The text editor will not return an input event to the calling application each time the text field is modified. Instead, it will only return to the application when a major event occurs; these are also referred to as break conditions. There are four categories of break conditions:

- The field has been exited [by a traversal key].

- The field has now become empty, or the first character was typed.

- An X event other than a KeyPress, KeyRelease, XrSELECT or XrSELECTUP event is received.

- A control character is received.

For the first type, the editor will return, and not expect to be called again, until the field has again been activated. However, for the second type, the editor is merely returning some status information to the application. As soon as the application has processed the status information, it should immediately re-invoke the editor, either directly (by issuing a MSG_ACTIVATE), or by means of the forms controller. Failing to follow this rule may produce unpredicatable results. The third and fourth types are similar to the second type, in that they too are only returning some status information. They will return a status event telling the application that an unknown X event was received, along with the unknown X event. The application should then process the X event, which is on the input queue AFTER the status event, and then re-invoke the editor in one of the two fashions listed above.

INSTANCE LAYOUT

The layout for a text editor instance depends upon how it was defined. When an instance is drawn, it will be drawn within the bounds of the supplied 'editorRect'. If the 'editorRect' is not large enough to hold the specified instance, then the create request will fail.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create an editable text field within a window. It will expect the *instance* parameter to be set to **NULL**, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
   Window       editorWindowId;
   RECTANGLE    editorRect;
   INT8         editorState;
   INT32        editorFGColor;
   INT32        editorBGColor;
   XFontStruct  * editorFont;
   INT8         * label;
   INT8         * string;
   INT32        maxChars;
   INT32        insertPos;
   INT8         insertMode;
   INT8         labelPosition;
   INT8         cellWidth;
} xrTextEditInfo;
```

editorWindowId

This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect

This describes the location and size of the region into which the text edit field is to be displayed. If a rectangle too small to hold the instance is specified, then the editor will refuse to handle the request, and it will fail.

editorState

This field contains the initial value of the state flags for this editor instance. It can be composed of any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor

This field specifies the foreground color to be used when drawing the editor instance. The borders and all text are drawn using the foreground color. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor

This field specifies the background color to be used when drawing the editor instance. The interior of the text region is drawn using the background color. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont

This points to the structure which contains all of the information necessary to describe the font to be used when displaying the text edit field. If this pointer is **NULL**, then the default system base font will be used.

label

This is a pointer to the optional **NULL** terminated text label which will be displayed to the left of the text edit region. It will be displayed using the same font used within the text edit region. If this pointer is **NULL**, or if the string has a length of 0, then no label will be displayed.

string

This is a pointer to the buffer which will be used by the editor to store the working copy of the data in the text region. The editor will expect the buffer to be large enough to hold the maximum size string for this field, along with a terminating **NULL**. The buffer must contain the **NULL** terminated string representing the initial value for the text region. It will be by means of this buffer that an application program will have access to the current value of the text editor instance.

maxChars

This value represents the maximum number of bytes which will be allowed in the text editor instance. This information will be used by the editor to prevent a user from entering too much data into a field. It is also used by the editor, when calculating the size of the text region and the frame rectangle. This value should not include space taken up by the **NULL** character used to terminate the string.

insertPos

This value allows an application program to specify the initial position of the text cursor. The specified value indicates the character position within the buffer, for where the next character typed should be placed; e.g. 0 means place the cursor at the first character position, and the next character typed with be saved in buffer[0]. If a value is specified which is greater than the current length of the text string, then the text cursor will be placed at the last character position in the string.

insertMode

This value allows an application to specify the type of insertion mode the field editor should operate under. If the field is set to **XrALWAYS_ON** then insert mode will be forced on for the duration of the editing session; the insert character key will be ignored. If the field is set to **XrALWAYS_OFF** then insert mode will be forced off for the duration of the editing session, and the editor will operate in 'normal' mode; the insert character key will be ignored. If the field is set to **XrINTERACTIVE** then the editing mode will be controlled by the user pressing the insert character key; the editor will toggle in and out of insert mode each time the insert character key is pressed.

labelPosition

This value allows an application to specify whether the optional label should be displayed to the left or right of the text editing region. If a label is not specified for an instance, then this field will be ignored. The two valid settings for this field are **XrLEFT_ALIGNED** and **XrRIGHT_ALIGNED**.

cellWidth

This parameter specifies the character cell width which should be used when calculating the size of the text editing region. For the maximum cell width for a given font, use the define **XrMAXWIDTH.** For a cell width which is the average size for a given font, use the define **XrAVGWIDTH.** If any other value is desired, then the cell width (in pixels) may be specified directly; this information can be obtained by using the **XrStringWidth**() utility routine.

The editor will then draw the text edit field in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, returned by **MSG_NEW** ; this parameter specifies which instance is to be destroyed.

When a text editor instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects or keyboard input, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified text edit instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given text edit instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the editor instance will be redrawn, to reflect the new state. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If a field is visible, but not sensitive, then the insertion area will be drawn and filled with a 50% pattern; the current field value will also be displayed.

MSG_GETITEMCOUNT

This message allows an application to obtain a count, which indicates the number of insertion areas in the specified instance. The editor will assume that the *instance* parameter specifies the editor instance to query, and the *data* parameter points to a 32 bit integer value; the item count value will be returned by means of this integer.

This message is useful when used in conjunction with the **MSG_GETITEMRECTS** message. It allows an application to obtain the number of items in the instance, so that the application can then allocate enough memory to hold the rectangle information returned by **MSG_GETITEMRECTS.**

For this editor, a value of 1 will always be returned, since a text edit instance can presently only have one insertion area.

MSG_GETITEMRECTS

This message returns the coordinates for the rectangle which describes the insertion area; this rectangle describes only the insertion area - it does not include the label. The message will expect the *instance* parameter to specify the editor instance to query, and the *data* parameter to point to a structure of the following format:

```
rectangle   itemRects;
```

This structure will then be filled with the rectangle information, and returned to the application.

Before an application can make this call, it needs to know the number of insertion areas in the specified instance, so that it can allocate a structure large enough to hold all of the rectangle information. The application should use the **MSG_GETITEMCOUNT** message to obtain this information; the application can then allocate an array large enough to hold all of the rectangle entries.

This message is useful to those applications which have a need of knowing where the insertion area in a text edit instance is located. The most common use would be by a forms controller, which would use the information to place an 'active field' indicator by a given item.

MSG_ACTIVATE

This message allows an application to force a text edit instance active, thus causing it to turn on the text cursor, widen the editing region border, and start monitoring all incoming keyboard events. This message also serves as the only means for re-entering an active text edit field, after an application has received a status event from the instance. When a **MSG_ACTIVATE** message is issued, the editor instance must be both visible and sensitive; if this is not the case, then the message will fail.

When this message is issued, the *instance* parameter must specify the text edit instance which is to be activated; the *data* parameter is unused, and should be set to **NULL**.

MSG_DEACTIVATE

This message allows an application to force a text edit instance inactive. Normally, when a text edit instance returns a status event to an application (such as the field is now empty, or an unknown X event was received), the application processes the status event and then reinvokes the text editor. However, it is possible for an application to decide that it does not want to reinvoke the editor instance, but rather, it wants the instance to be deactivated. The instance will be drawn as inactive in the same fashion as when a traversal key is received.

When this message is issued, the *instance* parameter must specify the text edit instance which is to be deactivated; the *data* parameter is unused. If the text edit field is already inactive, then this request will be ignored.

MSG_INSERTMODE

This message allows an application to modify the insert mode under which the field editor will operate.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be modified. The *data* parameter must be a 8 bit integer value, set to either **XrALWAYS_ON, XrALWAYS_OFF** or **XrINTERACTIVE.** If **XrALWAYS_ON** is selected, then the editor will always operate with insert mode enabled; the insert character key will be ignored. If **XrALWAYS_OFF** is selected, then the editor will always operate with insert mode disabled (in normal mode); the insert character key will be ignored. If **XrINTERACTIVE** is selected, then the editor will allow the user to toggle into and out of insert mode, by pressing the insert character key on the keyboard.

If any other mode is selected, then the request will fail.

MSG_REDRAW

This editor provides an application with the means for requesting that a text edit instance be redrawn.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The text editor supports the following redraw modes:

- **XrREDRAW_ALL**

- **XrREDRAW_ACTIVE**

XrREDRAW_ALL will cause the editor to redraw the complete editor instance, while **XrREDRAW_ACTIVE** will cause only the text string portion of the instance to be redrawn.

If any other redraw mode is specified, then the request will fail.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a given text edit instance.

The text editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window        editorWindowId;
   RECTANGLE     editorRect;
   INT8          editorState;
   INT32         editorFGColor;
   INT32         editorBGColor;
   XFontStruct * editorFont;
   INT8        * label;
   INT8        * string;
   INT32         maxChars;
   INT32         insertPos;
   INT8          insertMode;
   INT8          labelPosition;
   INT8          cellWidth;
} xrTextEditInfo;
```

The fields which must be filled out by the application program BEFORE issuing this call, are the *label, maxChars, cellWidth* and *editorFont* fields.

If *editorFont* has been set to NULL, then the default system base font will be used when calculating the rectangle size; if a particular font is requested, then it will be used.

The rectangle calculated will be the smallest 0 based rectangle, into which the specified text edit instance will fit, using the specified font and character cell width.

In return, *editorRect* will be set to describe the optimal rectangle into which the described instance will fit. An application should feel free to lengthen this rectangle, but it should refrain from modifying the height.

It should be noted that if a cell size smaller than the maximum cell size for a given font is specified, there is no guarantee that the complete string typed by a user will be visible at one time.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect*. All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect*.

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routine, when an input event occurs while a text edit field is active.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which text edit instance to process, and the *data* parameter pointing to an *XEvent* structure.

The **MSG_EDIT** handler for the text edit editor is much more complex than most editors, since it understands a much broader range of input events. Making up the list of valid input types for this editor are:

Most X KeyPress and KeyRelease events.

All X button events which map into either an XrSELECT or XrSELECTUP event (see XrMapButton(3X) and XrInit(3X)).

All other X events will be treated by the editor as an unknown event, and will be returned to the application for local processing. This is discussed further in a later section dealing with unknown events.

The sections below will attempt to describe how the editor handles each of the above input types.

X KeyPress and KeyRelease events (Printable ASCII keys only)

If the text field is already full, or if this is a KeyRelease event, then the editor will ignore the event. Otherwise, the following will occur:

If the editor is currently operating with insert mode enabled, then all characters in the buffer, which are at or to the right of the text cursor, will be shifted one place to the right within the work buffer.

The new keycode will be inserted at buffer[insertion point].

The insertion point (text cursor) will be moved one character position to the right.

The text cursor will be forced on (if not already).

The instance will be redrawn.

Afterwards, a check will be made to see if the string was previously empty and this was the first character in the string. If it was, then the following status event will be appended to the front of the application's input queue, and the editor will return:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrTEDIT_FIRST
valuePtr	=	pointer to instance's editor structure

If the above event did not occur, then the editor will block, waiting for the user to generate the next input event.

X KeyPress and KeyRelease events (Editing keys)

Cursor Right

If the text cursor is not already at the end of the string, then it will be shifted one character position to the right. The field will then be redrawn as active, if it was not already.

Cursor Left

If the text cursor is not already at the beginning of the string, then it will be shifted one character position to the left. The field will then be redrawn as active, if it was not already.

Backspace

If the text cursor is not already at the beginning of the string, then the character to the left of the text cursor will be deleted, and all character to its right will be shifted one position to the left. The text cursor will also move one position to the left. The field will then be redrawn, and forced active (if it was not already). If the text field is now empty, then the following status event will be added to the front of the application's input queue, and the editor will return:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrTEDIT_EMPTY
valuePtr	=	pointer to instance's editor structure

Delete Char

If the text cursor is not at the end of the string, then the character underneath the text cursor will be deleted, and all characters to its right will be shifted one position to the left. The field will then be redrawn, and forced active (if it was not already). If the field is now empty, then the following status event will be appended to the front of the application's input queue, and the editor will return:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrTEDIT_EMPTY
valuePtr	=	pointer to instance's editor structure

Delete Line

This will move the text cursor to character position 0, and then delete all character to its right. The field will be redrawn, and forced active (if it was not already). The following status event will be appended to the front of the application's input queue, and the editor will return:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrTEDIT_EMPTY
valuePtr	=	pointer to instance's editor structure

Clear Line

This will operate the same as 'delete line', except that the text cursor is not moved and only those characters under and to the right of the text cursor are deleted. If the field is now empty, then the following status event will be appended to the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrTEDIT_EMPTY
valuePtr	=	pointer to instance's editor structure

X KeyPress and KeyRelease events (*Traversal keys*)

There are a special set of keys, which are referred to as traversal keys. These serve as the mechanism for exiting from a field, and deactivating it. The keys making up this list are:

- Return
- Tab
- Backtab

When a traversal key is pressed by the user, the editor will redraw the field as inactive, and then place the following input event onto the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrTEDIT_BREAK
value2	=	traversal direction indicator
value3	=	traversal key keycode
valuePtr	=	pointer to instance's editor structure

The traversal direction indicator will assume a value of either **XrNEXT** or **XrPREVIOUS.**

The traversal key keycode will be either (RETURN_KEY | K_s), (TAB_KEY | K_s) or (BACKTAB_KEY | K_s), as defined in <Xr11/keycode.h>. The *value3* field within the event structure is defined as an INT16 quantity. However, the traversal keycode is a UINT16 value; therefore, an application should coerce the signed *value3* value into an unsigned value, before comparing it to any of the defined keycode values listed above.

X ButtonPress or ButtonRelease events

The only type of X button event understood by this editor is the one which maps into either a **XrSELECT** or **XrSELECTUP** event. All other button events are treated as an unknown event. When a select occurs, the editor will first check to see if it occurred within the instance's rectangle.

If it did not occur within the rectangle, then the field will be redrawn as 'not active', and two input events will be added to the front of the application's input queue:

1) A copy of the select input event which we just received.

2)

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XTEXTEDIT
value1	=	XrTEDIT_BREAK
value2	=	XrSELECT
valuePtr	=	pointer to instance's editor structure

NOTE: the second input event will end up at the front of the queue.

The editor will then return, and not expect to be called again, until the next time it should be made active.

If the select occurred within the text region, then the field will be redrawn as active (if it already was not), and the text cursor will be moved to the character position nearest to where the select occurred. The editor will then block, waiting for the user to generate the next event.

Unknown Events and ASCII Control Characters

Any event which is received by the text editor, which does not fall under any of the groupings discussed above, will be treated as an unknown event, and will be passed back to the application for processing. When the application receives notification of an unknown event from the text editor, it should request the next event from its input queue (the unknown event) and then process it. Since this is only a temporary break condition, the editor will expect the application to re-invoke the editor, once it has finished processing the unknown event.

The following two events will be pushed onto the front of the application's input queue:

1) A copy of the unknown X event which we just received.

2)

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTEXTEDIT
value1	=	XrUNKNOWN_EVENT
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the text editor that fail, set the *xrErrno* global and return a NULL value, under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is too small to hold the specified instance [XrINVALIDRECT].

string is set to NULL [XrINVALIDPTR].

maxChars is set to zero [XrINVALIDPARM].

insertMode is set to an unknown value [XrINVALIDOPTION].

labelPosition is set to an unknown value [XrINVALIDOPTION].

cellWidth is set to an invalid value [XrPARMOUTOFRANGE].

A call to 'X' failed { XMakePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** or **XrREDRAW_ACTIVE** is specified [XrINVALIDOPTION].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

maxChars is set to zero [XrINVALIDPARM].

cellWidth is set to an invalid value [XrPARMOUTOFRANGE].

MSG_RESIZE
data is set to NULL [XrINVALIDPTR].

editorRect is too small to hold the specified instance [XrINVALIDRECT].

MSG_ACTIVATE
The instance is not sensitive and visible [XrINVALIDPARM].

MSG_INSERTMODE
An unknown insertion mode value is specified [XrINVALIDOPTION].

MSG_GETSTATE, MSG_MOVE, MSG_GETITEMCOUNT and MSG_GETITEMRECTS
data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE
The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN
HP

SEE ALSO
XrInput(3X), XrInit(3X), XrMapButton(3X)

NAME

XrTitleBar - an editor that creates and controls a title bar.

SYNOPSIS

```
#include <X11/Xlib.h>
#include <Xr11/Xrlib.h>

xrEditor *
XrTitleBar (instance, message, data)
        xrEditor * instance;
        INT32      message;
        INT8     * data;
```

DESCRIPTION

The titlebar editor is used to create and process an instance of a titlebar within a given window. It will allow an application program to specify a titlebar of arbitrary width, composed of any combination of title strings and two custom gadget boxes. The default system base font will be used when displaying the title string, unless a different font is specified when the instance is created.

A titlebar can be composed of anywhere from 1 to 4 regions. These regions are:

- The background
- The title string
- Gadget box 1
- Gadget box 2

Each of these regions is selectable by a user, and will result in an application being notified of the event and the region selected.

The custom gadget boxes are displayed at either end of the titlebar, with box 1 being on the left, and box 2 on the right. When an application requests that a gadget box be displayed, it must also specify an alphanumeric character, which will be displayed within the gadget box; to specify the character, an ASCII code must be supplied. This allows an application to use these gadget boxes as icons for frequently used actions.

Whenever possible, the complete title string will be displayed within the titlebar. However, the situation can occur where the titlebar is not long enough to hold the complete title string. Rather than refusing to draw the titlebar at all, the editor will display as much of the title string as possible.

There is, however, a lower size limit which will be imposed upon a titlebar instance. A titlebar must be large enough to hold the gadget boxes (if defined), the title string border, and at least the first character of the title string. An attempt to create a titlebar in a rectangle smaller than this lower limit, will fail. An application can easily obtain the rectangle describing this lower size limit, by using the **MSG_MINSIZE** message.

When an application wants to create a titlebar instance, it should use the **MSG_SIZE** message to create the rectangle to contain the instance. This rectangle will be the smallest 0 based rectangle needed to contain the complete instance, using the specified font; the rectangle can then be stretched, to make the titlebar the desired length. The height of the titlebar will be proportional to the font specified, and must not be modified by the application. If the height is not what the editor would expect, when using the specified font, then the editor will refuse to create the titlebar instance.

EDITOR MESSAGES

MSG_NEW

This message will be the means by which an application program can create a titlebar editor instance in a window. It will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to a filled out instance of the following structure:

```
typedef struct {
    Window        editorWindowId;
    RECTANGLE     editorRect;
    INT8          editorState;
    INT32         editorFGColor;
    INT32         editorBGColor;
    XFontStruct * editorFont;
    INT8        * titleName;
    INT8        * gadgetIcon1;
    INT8        * gadgetIcon2;
} xrTitleBarInfo;
```

editorWindowId
This field indicates the window to which the editor instance is to be attached. Any time the instance is redrawn, it will be redrawn in this window.

editorRect
This describes the location and size of the region into which the titlebar instance is to be drawn. It must be at least large enough to hold the minimal components of the titlebar, as outlined earlier. If it is too small, then the create request will fail. It will be possible for an application to request that a titlebar be created along the entire top of the window, by setting *editorRect* to (0,0,0,0); the editor will automatically calculate the rectangle needed. If an application attempts to create a titlebar with a height less than 11, then the create request will fail.

editorState
This field contains the initial value of the state flags for this editor instance. It can be composed on any combination of the **XrSENSITIVE** and **XrVISIBLE** flags.

editorFGColor
This field specifies the foreground color to be used when drawing the editor instance. If this is set to -1, the default foreground color (see **XrInit(3X)**) will be used.

editorBGColor
This field specifies the background color to be used when drawing the editor instance. If this is set to -1, the default background color (see **XrInit(3X)**) will be used.

editorFont
This is a pointer to a structure which describes the font to be used when creating this editor instance. If the pointer has been set to NULL, then the editor will use the default system base font. The font is important not only because it determines how the title string will look, but it is also used to determine the height of the titlebar.

titleName
This points to the NULL terminated string, which is to be displayed as the title. If *titleName* is either NULL, or points to an empty string, then a title string will not be displayed.

gadgetIcon1
: This points to a single character which should be displayed within gadget box 1. If this pointer is set to **NULL**, then gadget box 1 will not be displayed. This character uses the same font as the title string.

gadgetIcon2
: This points to a single character which should be displayed within gadget box 2. If this pointer is set to **NULL**, then gadget box 2 will not be displayed. This character uses the same font as the title string.

The editor will then draw the titlebar editor instance in the specified window.

Upon successful completion, a pointer to the newly created editor structure will be returned to the application. This value must be used there after, whenever the application wishes to refer to this particular editor instance.

MSG_FREE

This message is the mechanism by which an editor instance can be destroyed. The only parameter of importance is the *instance* parameter, which is a pointer to the editor structure, returned by MSG_NEW; this parameter specifies which instance is to be destroyed.

When a titlebar instance is destroyed, it will be internally disconnected from the window to which it was attached, it will no longer handle mouse selects, and it will be removed from the window, if the instance is visible.

After an editor instance has been destroyed, no further messages should be issued in regard to that instance.

MSG_GETSTATE

This message returns the current state of the **XrVISIBLE** and **XrSENSITIVE** flags for the specified titlebar editor instance. The *instance* parameter specifies which instance to use. The *data* parameter should be a pointer to an 8 bit integer value, into which the current state flags will be placed.

MSG_SETSTATE

This message allows an application program to modify the setting of the **XrSENSITIVE** and **XrVISIBLE** flags, for a given titlebar editor instance. The *data* parameter is interpreted as an 8 bit integer value, containing the new state flag values. After saving the new state flags, the titlebar will be redrawn. If an instance is not visible, then the rectangle which it occupies will be drawn using the background tile for the window, thus making it invisible. If an instance is visible, but not sensitive, then the titlebar will be drawn as follows:

The border for the titlebar instance will be drawn.

The title string will be displayed.

The title borders will not be displayed.

The gadget boxes will not be displayed.

MSG_REDRAW

This editor provides an application with the means for requesting that a particular titlebar instance be redrawn. The originally specified parameters will again be used.

When this message is issued, the *instance* parameter must be a pointer to the editor structure associated with the instance to be redrawn. The *data* parameter must be a 32 bit integer which specifies the type of redraw to perform.

The titlebar editor supports the following redraw mode:

- **XrREDRAW_ALL**

If any other redraw mode is specified, then the request will fail.

MSG_MINSIZE

As was mentioned earlier, it is possible for an application to create a titlebar which is too small to hold the complete title string. However, the titlebar must be long enough to display at least the gadget boxes (if defined), the title string border and the first character of the title string. This message will return the information describing the rectangle needed to contain these minimal components.

Some applications, such as a window controller, may like to know the minimum size a particular titlebar can shrink to, so that it can prevent a user from shrinking a window any smaller than that size. This message will supply this information.

This message will expect the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
   Window      editorWindowId;
   RECTANGLE   editorRect;
   INT8        editorState;
   INT32       editorFGColor;
   INT32       editorBGColor;
   XFontStruct  * editorFont;
   INT8       * titleName;
   INT8       * gadgetIcon1;
   INT8       * gadgetIcon2;
} xrTitleBarInfo;
```

The only fields which must be filled out by the application BEFORE issuing this call, are the *gadgetIcon1, gadgetIcon2, titleName* and *editorFont* fields; if the *editorFont* field is NULL, then the default system base font will be used when calculating the rectangle.

Using the supplied information, the editor will set the *editorRect* field to the coordinates for the 0 based minimum sized rectangle.

If *titleName, gadgetIcon1* and *gadgetIcon2* have all be set to NULL, or point to empty strings, then this request will fail. An empty titlebar is not allowed.

MSG_SIZE

This message allows an application to obtain the rectangle needed to contain a minimally sized titlebar instance. This differs from the MSG_MINSIZE message, in that the returned rectangle will be large enough to hold the gadget boxes (if defined), the title string border, and the complete title string.

The titlebar editor expects the *instance* parameter to be set to NULL, and the *data* parameter to point to an instance of the following structure:

```
typedef struct {
    Window      editorWindowId;
    RECTANGLE   editorRect;
    INT8        editorState;
    INT32       editorFGColor;
    INT32       editorBGColor;
    XFontStruct * editorFont;
    INT8        * titleName;
    INT8        * gadgetIcon1;
    INT8        * gadgetIcon2;
} xrTitleBarInfo;
```

The only fields which must be filled out by the application program BEFORE issuing this call, are the *editorFont, gadgetIcon1, gadgetIcon2* and *titleName* fields; all other fields are ignored.

If the application does not specify a specific font to be used (*editorFont* = = NULL), then the default system base font will be used when calculating the size of the rectangle.

In return, the *editorRect* field will be filled in with the coordinates for the smallest 0 based rectangle needed to contain the instance; an application program can then lengthen and offset this rectangle, to position it as it likes, within it's window.

If *titleName, gadgetIcon1* and *gadgetIcon2* have all been set to NULL, or point to empty strings, then this request will fail. An empty titlebar is not allowed.

MSG_MOVE

This message provides an application with a means for quickly relocating a particular editor instance within a window. The size of the *editorRect* associated with the instance is not changed. To relocate an editor instance, a new origin point for the instance's *editorRect* must be specified; the top left corner of the editor rectangle will then be translated such that it now coincides with the new origin. The origin point is interpreted as an absolute position within the window.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be moved, while the *data* parameter must point to a *POINT* structure, containing the new *editorRect* origin.

When an editor instance is relocated, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance at its new location; this occurs only if the instance is visible.

MSG_RESIZE

This message provides an application with a means for both changing the size of the *editorRect* associated with a particular editor instance, and also the location of the new *editorRect.* All restrictions regarding the *editorRect* size which applied when the instance was first created using MSG_NEW, still apply. If an invalid *editorRect* is specified, then the resize request will fail.

When this message is issued, the *instance* parameter must point to the editor structure associated with the instance which is to be resized, while the *data* parameter must point to a *RECTANGLE* structure containing the new size and origin for the *editorRect.*

When an editor instance is resized, the field editor will automatically remove the visual image of the instance from the window, and will then redraw the instance using the new size and location information; this occurs only if the instance is visible.

MSG_EDIT

Normally, an application will not issue this message; it is usually issued by the Xrlib input routine, when an input event occurs within a titlebar instance.

When such an event occurs, a **MSG_EDIT** message will be issued to the editor, with the *instance* parameter indicating which titlebar instance to process, and the *data* pointing to an *XEvent* structure.

The titlebar editor only handles an event if it maps to an **XrSELECT** event, as described by **XrMapButton**(3X) and **XrInit**(3X); all others are ignored. When a select event occurs within a titlebar instance, the first thing done is to determine which region the select occurred in.

All titlebar selects will fall within one of the following four regions:

- Gadget box 1
- Gadget box 2
- The title box (title string & title border)
- The background area

If gadget box 1 was selected, then an input event will be added to the front of the application's input queue, informing it of the event. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTITLEBAR
value1	=	XrGADGET_BOX1
valuePtr	=	pointer to instance's editor structure

If gadget box 2 was selected, then an input event will be added to the front of the application's input queue, informing it of the event. The returned *xrEvent* structure is set to the following value:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTITLEBAR
value1	=	XrGADGET_BOX2
valuePtr	=	pointer to instance's editor structure

If the title string region was selected, then the following input event will be added to the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTITLEBAR
value1	=	XrTITLE_REGION
valuePtr	=	pointer to instance's editor structure

If the select occurs outside any of the above mentioned regions, then the following input event will be added to the front of the application's input queue:

type	=	XrXRAY
serial	=	0
send_event	=	0
display	=	pointer to the current display
source	=	the window Id
inputType	=	XrEDITOR
inputCode	=	XrTITLEBAR
value1	=	NULL
valuePt	=	cursor position
valuePtr	=	pointer to instance's editor structure

RETURN VALUE

Upon successful completion of any of the messages, a non-NULL value will be returned. In the case of MSG_NEW, this non-NULL value will be the pointer to the newly created editor instance structure.

If a message request fails, then a NULL value is returned.

ERROR CONDITIONS

Messages to the titlebar editor will fail, set the *xrErrno* global, and return a NULL value under the following conditions:

MSG_NEW

data is set to NULL [XrINVALIDPTR].

editorWindowId is an invalid Id [XrINVALIDID].

editorRect is not large enough [XrINVALIDRECT].

titleName, gadgetIcon1 and *gadgetIcon2* are all set to NULL [XrINVALIDPARM].

titleName, gadgetIcon1 and *gadgetIcon2* all point to empty strings [XrINVALIDPARM].

A call to 'X' failed { XMakePixmap() } [XrXCALLFAILED].

Memory cannot be allocated [XrOUTOFMEM].

MSG_REDRAW

A redraw mode other than **XrREDRAW_ALL** is specified [XrINVALIDOPTION].

MSG_MINSIZE

data is set to NULL [XrINVALIDPTR].

titleName, gadgetIcon1 and *gadgetIcon2* are all set to NULL [XrINVALIDPARM].

titleName, gadgetIcon1 and *gadgetIcon2* all point to empty strings [XrINVALIDPARM].

MSG_SIZE

data is set to NULL [XrINVALIDPTR].

titleName, gadgetIcon1 and *gadgetIcon2* are all set to NULL [XrINVALIDPARM].

titleName, gadgetIcon1 and *gadgetIcon2* all point to empty strings [XrINVALIDPARM].

MSG_RESIZE

data is set to NULL [XrINVALIDPTR].

editorRect is not large enough [XrINVALIDRECT].

MSG_GETSTATE and MSG_MOVE

data is set to NULL [XrINVALIDPTR].

All messages, except MSG_NEW and MSG_SIZE

The *instance* parameter is set to NULL [XrINVALIDID].

ORIGIN

HP

SEE ALSO

XrInput(3X), XrInit(3X)

NAME

Utilities - Utilities for performing some general functions

SYNOPSIS

```
#include <X11/Xib.h>
#include <Xr11/Xrlib.h>
```

XrStringWidth (fontInfo, str, charWidth)
XFontStruct * fontInfo;
STRING8 str;
INT32 charWidth;

XrMapButton (eventCode, event)
INT8 eventCode;
XEvent * event;

XrGetWindowEvent (eventCode, windowEvent)
INT8 eventCode;
xrWindowEvent * windowEvent;

INT8 *
_XrSetUpGC (gc)
GC gc;

INT8 *
XrVersion ()

DESCRIPTION

XrStringWidth

XrStringWidth() calculates the width in pixels of a character string using the font information contained in the structure pointed at by *fontInfo.* The string is represented by the rest of the parameters. *str* points to the string to be used in the calculation. *charWidth* contains the length of the string in characters or is set to the define **XrNULLTERMINATED** if the string is null-terminated.

The pixel width of the string is returned as the value of the function.

XrMapButton

XrMapButton() takes as a parameter an eventCode which contains one of the following defines: **XrSELECT, XrSELECTUP, XrMENUPOST, XrMENUITEMSELECT.** The event parameter will be compared against the conditions necessary for the above defines and if a match is found **TRUE** will be returned. If no match is found, **FALSE** will be returned.

XrGetWindowEvent

XrGetWindowEvent() takes as input one of the defines **XrSELECT, XrSELECTUP, XrMENUPOST, XrMENUITEMSELECT** and fills out the xrWindowEvent structure with the values necessary for the define. The structure can then be used to add a function into a window that will be invoked under one of the defines conditions.

_XrSetUpGC

_XrSetUpGC() copies the default Xrlib graphics context (**xrDefaultGC**) to the graphics context supplied as a parameter. This initializes each field of the specified graphics context to the default colors, tiling, stipling, etc values used by Xrlib. the

XrVersion

XrVersion() returns a pointer to a string which contains the version of Xrlib being used. The string will have the format of **X-ray Version** *## - ##.##* The *##* contains the X library version that the Xr library was built with. The *##.##* can be ignored. The string returned is static and thus cannot be modified.

ORIGIN

HP

B Porting X10 Xrlib Applications to X11R

This appendix provides information on what has to be changed in order to 'port' X version 10 applications created with Xrlib to X version 11. X11 is quite different from X10 in many ways and depending on how Xrlib applications are coded, they may very easy or quite difficult to port.

The rule of thumb is: If you used Xrlib with very few Xlib calls, the port will be easy. If you used lots of Xlib and few Xrlib calls the port will be tough.

This appendix does not attempt to describe all the changes in X11 that will affect existing X10 applications, it only describes those changes that directly effect Xrlib. Remember that many subtle changes have occurred between X versions. For instance, "KeyPressed" events have become "KeyPress" events. Please see the 'Xlib - C Language Interface, Protocol Version 11' document for more information on porting X10 Xlib level calls to their X11R Xlib equivalents.

B.1 X11R Initialization

The calling sequence for initializing Xrlib used to be:

```
XrInit (allocFuncts)

   xrAllocFuncts * allocFuncts;
```

it is now the following:

```
XrInit (displayPtr, screen, allocFuncts)

   Display       * displayPtr;
   INT32           screen;
   xrAllocFuncts * allocFuncts;
```

XrInit() can only be invoked once, thus an application is limited to running on a single display. The new display and screen information is necessary to allow X11R to allocate the default resources (pixmaps, images, fonts, cursors, etc) supplied for Xrlib components and the application.

During initialization of X11R, system globals are initialized, and made available to both the application and other toolbox components:

_xrCurrentDisplay	This will be set to contain the display pointer specified as a parameter to **XrInit()**. This value is used by the Xrlib internals whenever a request is issued to the X11 library that requires a display pointer. All X resources (window Ids, font Ids, pixmap Ids, etc) which are passed to Xrlib must belong to the display specified at initialization time; otherwise, X errors will occur.
_xrCurrentScreen	This will be set to contain the index of the screen upon which Xrlib will be run. As with the display pointer, this information will be used by Xrlib whenever an X library call is issued which requires information about the screen being used.

The application is limited to running upon a single screen on a single display.

WhitePixmap	
BlackPixmap	The X10 library supplied the application with these two default pixmaps, which were frequently used when creating windows. The X11 library no longer supplies these resources. So, X11R now exports these two global pixmaps, which may be used in the identical fashion as their X10 predecessors. These pixmaps are created *AFTER* Xr11 has been initialized; if the globals are referenced before **XrInit()** is invoked, an X error will occur, since the pixmaps have not yet been created. Applications which want to use these will need to make sure their call to **XrInit()** occurs immediately following the request to open the display.

B.2 Fonts

The default base font for the X10 Xrlib was the *timrom12b* font. With the move to X11 a new font strategy was adopted so none of the X10 fonts are supported. Currently, X11 only officially supports two fonts: *variable* and *fixed*. The new default base toolbox font will be the *variable* font. Because of the differences between X10 and X11 fonts, any text strings displayed by an X10 application will have a different size when run on X11. This may require the porter to increase the size of windows, or re-position editors within a panel or window.

B.3 Font Structures

All X10 Xrlib structures which contained a pointer to an instance of the *FontInfo* structure, now expect a pointer to an instance of the *XFontStruct* structure. The affected structures are:

Structure
xrTextInfo
xrListEditInfo
xrGroupBoxInfo
xrPageEditInfo
xrCheckBoxInfo
xrPushButtonInfo
xrRadioButtonInfo
xrStaticTextInfo
xrTextEditInfo
xrTitleBarInfo
xrPanelContext
xrMenuEditor

In addition, the following X11R font associated variables have been changed.

xrBaseFontInfo	For the same reasons outlined above, the X11R global *xrBaseFontInfo* now points to an instance of the *XFontStruct* structure, instead of an instance of the *FontInfo* structure.
XrTYPE_FONTINFO	The data associated with the resource type **XrTYPE_FONTINFO** has changed. The pointer returned when an X10 font was opened, pointed to an instance of the **FontInfo** structure. X11 no longer supports this structure. Instead, when

an X11 font is opened, a pointer to an instance of the **XFontStruct** structure is returned. To match the change imposed by X, the data structure pointer associated with the **XrTYPE_FONTINFO** resource type is now a pointer to an *XFontStruct* structure.

B.4 Bitmaps and Pixmaps

During X10 Xrlib initialization, a group of default bitmap resources were created for use by the application; these resources were saved in the resource manager as type

XrTYPE_BITMAPID X11 no longer makes a distinction between pixmaps and bitmaps. Instead, bitmaps are simply pixmaps with a depth of 1. So, X11R creates a set of pixmaps of depth 1, identical to the bitmaps created by X10 Xrlib, and again stores them in the resource manager as type **XrTYPE_BITMAPID**

In addition, the data structure associated with the resource type **XrTYPE_BITMAPID** has changed slightly, and now has the following format:

```
typedef struct {
   INT16  width;
   INT16  height;
   Pixmap bitmapId;/* bitmapId is now a Pixmap */
} xrBitmapId;
```

XrTYPE_IMAGEID During X11R initialization, a group of default image resources will be created for use by the application or other toolbox components. These resources are saved in the resource manager as type **XrTYPE_IMAGEID.** In conjunction with the previously mentioned change, a new resource type, **XrTYPE_IMAGEID,** is now supported by Xrlib resource manager. This allows an application to store X image information within the resource tree. The structure associated with this resource type is shown below:

```
typedef struct {
   INT16    width;
   INT16    height;
   XImage * imageId;
} xrImageId;
```

The resource Id's associated with the image resources are the same as the Id's used for the bitmap resources such as: XrWHITE, XrBLACK, XrPERCENT25, etc.

B.5 Graphics Contexts

After X11R initialization, 8 graphics context structures are available for use by the application or any toolkit components. These are accessible through the names **xrEditorGC1 - xrEditorGC8.** Also, a single graphics context structure **xrDefaultGC** is supplied. The default graphics context is set to contain the X11R default foreground color, background color, and default font Id. This graphics context must never be modified by an application or toolkit component. It is supplied so that an application can copy the default values into one of the other graphics context structure, thereby forcing it to a known state.

As an alternate method for initializing one of the graphics context structures to a known state a utility routine is supplied by X11R:

```
_XrSetUpGC (gc)

GC  gc;
```

This routine initializes each field of the specified graphics context to the default X11R values (default colors, tile, stipple, etc) and is the encouraged technique for initializing the GCs.

Two of the graphics context support routines provided in the X10 version of Xrlib are no longer supported. These routines were provided to look like their X11 equivalents and are no longer needed. This includes the following functions: **_XrCopyGC()** and **_XrChangeGC()**. Existing applications which use these functions should now use the facilities provided by X11; these are **XCreateGC()**, **XCopyGC()** and **XChangeGC()**.

The third routine, **_XrInitEditorGCs()** has been altered slightly under X11. The change concerns the *fontId* parameter. In X10, to initialize the **xrEditorGC1** and **xrEditorGC2** graphics contexts with foreground and background colors, but no font, you would pass a -1 as the *fontId* parameter. This will no longer work. To do the same thing under X11 you must use a 0 as the *FontId* parameter. This is because the variable type for *Font* is now unsigned.

```
_XrInitEditorGCs (foreColor, backColor, fontId)

INT32  foreColor;
INT32  backColor;
Font   fontId;
```

B.6 Event Structures

The X11R event structure, *xrEvent*, has three additional fields added to it, as required by X11.

serial	contains the serial number of the last processed server request. Xrlib ignores this field. When forging events it should be set to zero.
send_event	is a boolean field. If the current event was generated by a SendEvent request then this field will be set to a non-zero (TRUE) value. Xrlib ignores this field. When forging events it should be set to zero.
display	

display — The last field is a display pointer to the display from which the event was received. Xrlib requires this field. When forging events it must be set to the _xrCurrentDisplay global value.

If an X10 application was statically initializing a copy of the *xrEvent* structure, that code will need to be changed to take into account the new fields. The new *xrEvent* structure is defined as follows:

```
typedef struct {
   INT32            type;
   unsigned long    serial;
   Bool             send_event;
   Display        * display;
   INT32            source;
   INT16            inputCode;
   INT8             inputType;
   INT8             value1;
   INT16            value2;
   INT16            value3;
   POINT            valuePt;
   INT32            valuePtr;
} xrEvent;
```

B.7 Event Handling

In X10, when an application retrieved an X event, it determined the type of the event, by *AND'ing* the *type* field of the event with one of the event mask values defined by X. For instance, to check if an event was an Xrlib generated event, the application would do the following:

```
if (event->type & XrXRAY)
{
    .
    .
    .
}
```

With X11, this has changed. Within the *type* field of an event, a mask value is no longer specified. Instead, a scalar value (1, 2, 3, ...) is specified. When an application wishes to determine the type of an event, it must do a comparison (= =) instead of a bitwise AND. For instance, to check if an event was an X11R generated event, the application would do the following:

```
if (event->type == XrXRAY)
{
    .
    .
    .
}
```

B.7.1 Event Structure Size

With X10, all event structures (XKeyEvent, XButtonEvent, xrEvent, ...) were the same size. This made it possible for an application to obtain any X event, using such calls as *XrInput*() or *XNextEvent*(), by passing in a pointer to any valid X event structure; after the event was obtained, its *type* field could be checked, and the event coerced to the appropriate structure. For example, the following was valid:

```
xrEvent event;          /* NOTE: this line contains the difference */
XButtonEvent * button;

XrInput (NULL, MSG_BLKHOTREAD, &event);
if (event.type & XButtonPressed)
{
   button = (XButtonEvent *) &event;
}
```

With X11, the restriction that all X event structures must be the same size has been removed. Instead, event structure sizes can vary. X maintains a single structure, called *XEvent*, which is the union of all known X event structures. Because *XEvent* is a union, it is guaranteed to be as large as the largest X event structure. Since the various event structure sizes can vary, it is no longer safe to read an event into a structure other than an *XEvent* structure. Attempting to do so may cause your stack to be overwritten, or other variables to be 'magically' modified. The only safe way to read in an X event is to do the following:

```
XEvent         event;   /* NOTE: this line contains the difference */
XButtonEvent * button;

XrInput (NULL, MSG_BLKHOTREAD, &event);
if (event.type == ButtonPress)
{
   button = (XButtonEvent *) &event;
}
```

Note

The *xrEvent* structure provided in X11R is padded to be as large as the *XEvent* structure. The padding was done so that X10 applications that used the *xrEvent* structure as a general event structure would not break when moved to X11.

B.7.2 Key and Button Event Values

In X10, when a button or key event was received, the information describing the type of event and the particular key/button and modifier keys were made available by means of the *type* and *detail* fields; the *type* field indicated the type of event, and the *detail* field contained both the key/button information, and the modifier key information. In X11R V10, when specifying the set of input which would cause a window function to be invoked, a copy of the following structure was supplied for each input:

```
typedef struct {
   UINT32 inputType;    /* Event type */
   INT16  inputCode;    /* Key/Button data and modifiers */
} xrWindowEvent;
```

In X11, an event is still identified by checking its *type* field. However, if it is a key or button event, the key/button data and the modifier data are no longer grouped together; the application must now check three fields. To reflect this change in X11R, the structure used to specify the events which cause a window function to be invoked has changed to the following:

```
typedef struct {
   INT32  type;         /* Event type */
   UINT32 modifier;     /* Button modifiers, for button event */
   UINT32 code;         /* Key code or Button identifier */
} xrWindowEvent;
```

Any application using window functions, will need to modify their code to use this new structure. Since the 'type' field is now a scalar value, instead of an event mask, multiple events can no longer be specified within a single *xrWindowEvent* structure. If multiple window events are capable of invoking a window function, then each event should be placed in it's own separate *xrWindowEvent* structure in an array. This array should be passed to X11R.

Note

When specifying the event type in an *xrWindowEvent* structure, you must specify the scalar event type (i.e. KeyPress) and not the event mask value (i.e. KeyPressMask).

The above mentioned change also affect the X11R utility routine *XrGetWindowEvent*(), since this routine takes an instance of the *xrWindowEvent* structure as a parameter. The information returned by this procedure will now be in the new format described above.

The menu manager supports a message which allows the application to attach an event to a menu item; the message is MSG_SETITEMEVENT. In X10, since all event structures were the same size, the application could pass in a pointer to any event structure. However, since in X11 only the *XEvent* and *xrEvent* structures are guaranteed to be large enough to hold any event, the X11R version of this message will expect the event pointer to be *XEvent* or *xrEvent* structure types.

B.8 Window Information Management

Under X10, when a window was created, the application was required to supply a foreground and background pixmap for the window. Also, X10 did not provide a means for clearing only a portion of a window to its background pixmap. The only facility available was a command which cleared the whole window. To complicate matters even more, X10 did not provide a means for inquiring what the background tile was for a window. This made it difficult for any toolbox to make an editor instance invisible. To remedy this in Xrlib V10, when an application registered a window it supplied additional pieces of information, such as the foreground and background tiles, and the window size and location.

X11 has gone a long way to remedy these shortcomings. It now provides facilities for clearing a portion of a window, and allows an application to query the background tile for a window. It is also now possible for an application to create a window while supplying a foreground and background pixel, instead of a foreground and background pixmap. This causes a problem because it is possible for an application to create a window using a foreground and background pixel and then register that window with X11R using different information. This will not break X11R because it no longer relies upon this information. However, the application must also be careful to not rely upon this information. If the application needs to obtain the foreground or background pixmap associated with a window, it should obtain them from X11. The *XrInput*() message **MSG_GETWINDOWDATA** should no longer be used.

B.9 Mouse Pointer Control

In the X10 model, operation was centered around a 3 button mouse model: left, middle and right buttons. Xrlib V10 allowed the user to specify, via the *Xdefaults* file, which of these buttons would generate a *select* event, a *menu post* event, and a *menu select* event. This was accomplished by assigning one of the following strings to each of the Xrlib action identifiers:

LeftButtonDown	LeftButtonUp
MiddleButtonDown	MiddleButtonUp
RightButtonDown	RightButtonUp

X11 uses a 5 button mouse model. The buttons are referred to as **button1** through **button5.** To accommodate this alteration, X11R supports both the 3 and the 5 button mouse model. When specifying the X11R action bindings, the user may use the set of action identifiers presented in the previous paragraph, or he may choose from the set shown below:

Button1Down	Button1Up	(Equivalent to LeftButton)
Button2Down	Button2Up	(Equivalent to MiddleButton)
Button3Down	Button3Up	(Equivalent to RightButton)
Button4Down	Button4Up	
Button5Down	Button5Up	

B.9.1 Key and Button Modifiers

The means by which the user may attached modifier keys to the X11R actions has been enhanced. In X11R V10, the user was able to attach modifiers (Control, Meta, Shift or None) to the Left, Middle and Right Buttons, by simply placing in the *Xdefaults* file one of the following strings followed by the desired modifier:

AllButtonModifier
LeftButtonDownModifier
LeftButtonUpModifier
MiddleButtonDownModifier
MiddleButtonUpModifier
RightButtonDownModifier
RightButtonUpModifier

X11R supports the above mentioned modifier strings and the set shown below:

Button1DownModifier	(Equivalent to LeftButtonDownModifier)
Button1UpModifier	(Equivalent to LeftButtonUpModifier)
Button2DownModifier	(Equivalent to MiddleButtonDownModifier)
Button2UpModifier	(Equivalent to MiddleButtonUpModifier)
Button3DownModifier	(Equivalent to RightButtonDownModifier)
Button3UpModifier	(Equivalent to RightButtonUpModifier)
Button4DownModifier	
Button4UpModifier	
Button5DownModifier	
Button5UpModifier	

The set of button modifiers supported by X10 were **Control, Shift and Meta.** X11 now supports an enhanced set of button modifiers, comprised of **Control, Shift, Modifier1** (Equivalent to Meta), **Modifier2, Modifier3, Modifier4** and **Modifier5.** X11R will support both of these sets of button modifiers within the *.Xdefaults* file.

B.10 XrMenu() Save Unders

In X10, the XrMenu field editor would attempt to save any raster image that it obscured. This was done via an extension supplied with some servers that allowed raster images to be retained and stored off-screen. In X11, the XrMenu field editor will attempt to do the same, by setting the **save_under** window attribute to **True.** However, while save_under is part of the X11 standard, few vendors have implemented it to date. So applications should expect that menus will overwrite whatever is on the screen and cause exposure events on any obscured windows when destroyed. For more information on **save_under** and other window attributes see the *'Xlib - C Language Interface, Protocol Version 11'* document.

B.11 Miscellaneous Changes

The X11R utility routine *XrStringWidth*(), used to calculate the width of a string in pixels, has not changed its calling sequence, however, it no longer uses the *charPad* and *spacePad* parameters.

The following drawing functions, supplied by Xrlib V10 as stepping stones to X11, are no longer supported. Applications should be altered to use the real X11 drawing facilities:

Function
_XrPoly()
_XrFillPoly()
_XrOval()
_XrFillOval()
_XrEllipse()
_XrFillEllipse()

This page left blank intentionally.

Index

All references to appendix A, "Reference Information," are indicated as page A-1.