XView Programming Manual

Volume Seven

XView Programming Manual

By Dan Heller

Updated for XView Version 3.2 by Thomas Van Raalte

O'Reilly & Associates, Inc.

XView Programming Manual
by Dan Heller
Updated for XView Version 3.2 by Thomas Van Raalte

X Series Editor: Tim O'Reilly

Editor: Dale Dougherty

Printing History:

January 1990:	First edition.
April 1990:	Minor corrections.
July 1990:	Second edition. Updated for XView Version 2.
October 1990:	Minor corrections.
September 1991:	Third edition. Updated for XView Version 3.
March 1992:	Minor corrections.
August 1992:	Minor corrections.
August 1993:	Minor additions. Updated for XView Version 3.2.

Volume 7: ISBN 0–937175–87-0 [11/93]

Table of Contents
Volume 7A: Programming Manual

Chapter 17 Resources 381

Chapter 18 Selections 393

Chapter 19 Drag and Drop 433

Chapter 20 The Notifier 459

Figures

Examples

Tables

This page intentionally left blank

to preserve original page counts.

Preface

XView (**X** Window-System-based **V**isual/**I**ntegrated **E**nvironment for **W**orkstations) is a user-interface toolkit to support interactive, graphics-based applications running under the X Window System. This toolkit, developed by Sun Microsystems, Inc., is derived from earlier toolkits for the SunView windowing system. With over 2000 SunView applications in the workstation market, there are many programmers already familiar with the SunView application programmer's interface (API).

XView has many advantages for programmers developing new applications in X, such as a mature and proven API based on the development experience of SunView programmers. It features an object-oriented style interface that is straightforward and simple to learn.

Like any X toolkit, XView provides a set of pre-built, user-interface objects such as canvases, scrollbars, menus, and control panels. The appearance and functionality of these objects follow the OPEN LOOK Graphical User Interface (GUI) specification. Jointly developed by Sun Microsystems and AT&T as the graphical user interface standard for System V Release 4, OPEN LOOK provides users with a simple, consistent, and efficient interface for performing tasks within an application.

XView is based upon Xlib, the lowest level of the X Window System available to the programmer. While developing XView user interfaces does not require Xlib programming experience, there are good reasons for learning more about Xlib, especially if your application renders graphics.

Please Read This Section!

This manual provides a basic introduction to developing applications using the XView Toolkit. You do not need any knowledge of SunView, and prior experience with the X Window System is helpful, but also not required. Nonetheless, like any complex system, a programmer needs to know a lot to program effectively in XView.

For each functional area in XView, there are chapters that present the basic concepts and suggest some common ways to implement and use a particular function. Also addressed are some snags to watch out for when implementing certain combinations of functions. Care was taken to keep the content of the chapters brief and to the point. Simple and straightforward functions are not discussed in depth.

The *XView Reference Manual* is a companion to this manual. It contains complete descriptions for all of the XView attributes and procedures, as well as additional reference material.

The XView Programming Manual has been updated to cover XView Version 3, and includes several appendices providing compatibility information for previous XView versions. XView Version 3 contains many features not available in previous versions, including: a notice package, a selection package and a drag and drop package. The new Version 3 packages are presented in the corresponding chapters in this manual.

If you can't figure out how to accomplish a task because it is not documented here, don't despair—that does not mean it cannot be done. Some features in XView are not addressed in this book—especially the more advanced ones. You are encouraged to experiment with the toolkit and discover new ways of using XView.

How to Use This Manual

The chapters in the book are designed to be read sequentially. However, that is not a strong requirement. Reading ahead probably won't affect your understanding of the material, although later chapters might reference earlier material.

The following paragraphs briefly describe the contents of this book:

Chapter 1, *XView and the X Window System*, provides a conceptual overview of the X Window System, the role of the XView Toolkit, and the OPEN LOOK graphical user interface. It provides a general introduction to basic X terminology, but it does not go into great detail about X.

Chapter 2, *The XView Programmer's Model*, provides an overview of XView as an object-oriented programming system. The programmer creates and modifies objects that implement the OPEN LOOK interface. This chapter also discusses windows as objects that receive events. It introduces callback functions as the method of registering application-specific event handlers.

Chapter 3, *Creating XView Applications*, begins from the application developer's point of view and explains the basic elements of an XView application. It describes what is involved in initializing XView and creating XView objects such as frames and subwindows.

Chapter 4, *Frames*, explains how to create window frames. There are two basic types of frames: base frames and command frames. Each application has at least one base frame that manages subwindows, panels, and other objects. It presents the routines used to create and manage frames.

Chapter 5, *Canvases and Openwin*, presents canvases as the most basic type of subwindow or window pane. It presents the canvas model, which permits a drawing surface larger than what is visible in the canvas subwindow.

Chapter 6, *Handling Input*, explains how events are handled by X, the Notifier, and XView objects.

Chapter 7,	*Panels*, explains a variety of OPEN LOOK controls that are implemented as items on a control panel. It demonstrates how to create and use buttons, check boxes, choices, lists, messages, toggles, text items , and sliders. A set of panel attributes controls the behavior in common with all panel items. There are also item-specific attributes.
Chapter 8,	*Text Subwindows*, describes how to create a text subwindow and how to use its text editing features.
Chapter 9,	*TTY Subwindows*, describes the tty subwindow that performs terminal emulation functions.
Chapter 10,	*Scrollbars*, covers the creation and use of scrollbars. A scrollbar is a window attached to another window, such as a canvas or text subwindow or a panel. The scrollbar package only manages the scrollbar; the application must gauge the impact of scrolling on its windows.
Chapter 11,	*Menus*, explains how to implement various sorts of pop-up menus.
Chapter 12,	*Notices*, explains how pop-up windows serve as notices or dialog boxes.
Chapter 13,	*Cursors*, shows various OPEN LOOK pointers and demonstrates their use.
Chapter 14,	*Icons*, describes the use of bitmap images as application icons. When an application is closed, or iconified, the application is represented on the screen as an icon.
Chapter 15,	*Nonvisual Objects*, describes objects that do not contain windows: Server, Screen, and Fullscreen.
Chapter 16,	*Fonts*, describes how to load and use fonts from the X server.
Chapter 17,	*Resources*, describes the implications of X resources for an XView application. Resources allow individual users to control and customize their environment.
Chapter 18,	*Selections*, discusses how XView applications communicate with other applications, including window managers and applications that are not OPEN LOOK-compliant. It shows how XView provides for selections according to the *Inter-Client Communication Conventions Manual*.
Chapter 19,	*Drag and Drop*, discusses how XView applications implement dragging and dropping, where data is transferred by selecting an item and moving it to another workspace location. This is one method XView provides for applications to communicate with other applications.
Chapter 20,	*The Notifier*, describes the Notifier and advanced event handling. It describes the relationship of the Notifier and X to the host operating system.
Chapter 21,	*Color*, discusses issues concerning color in windows and other XView objects.
Chapter 22,	*Internationalization*, discusses the internationalization features of XView.
Chapter 23,	*Help Facilities*, discusses the help mechanism available in XView packages.

Chapter 24,	*Error Recovery*, discusses error handling in XView packages.
Chapter 25,	*XView Internals*, discusses the internals to the XView packages and introduces the concepts involved in writing your own packages.
Appendix A,	*The Selection Service*, describes the selection service which provides compatibility with older versions of XView that did not have a `SELECTION` Package.
Appendix B,	*Notices*, describes the notice procedure `notice_prompt()` which provides compatibility with older versions of XView that did not have a `NOTICE` Package.
Appendix C,	*Mouseless Model Keyboard Mappings*, presents the mouseless model keyboard mappings.
Appendix D,	*XView Version 3.2 Additions*, describes the changes and additions for XView Version 3.2, including the File Chooser documentation.
Appendix E,	*OPEN LOOK User Interface Compliance*, discusses XView's compliance with the *OPEN LOOK GUI Functional Specification*.
Appendix F,	*Example Programs*, presents supplementary programs.

Assumptions

Readers should be proficient in the C programming language, although examples are provided for infrequently used features of the language that are necessary or useful when programming with X. In addition, general familiarity with the principles of raster graphics is helpful. Finally, if you do not understand how to use Xlib routines to render graphics, then writing useful programs might be difficult, although you should be able to build OPEN LOOK user interfaces easily.

Font Conventions Used in This Manual

Italic is used for:

- UNIX pathnames, filenames, program names, user command names, and options for user commands.
- New terms where they are introduced.

`Typewriter Font` is used for:

- Anything that would be typed verbatim into code, such as examples of source code and text on the screen.

- XView packages.*
- The contents of include files, such as structure types, structure members, symbols (defined constants and bit flags), and macros.
- XView and Xlib functions.
- Names of subroutines of the example programs.

`Italic Typewriter Font` is used for:

- Arguments to XView functions, since they could be typed in code as shown but are arbitrary.

Helvetica Italics are used for:

- Titles of examples, figures, and tables.

Boldface is used for:

- Chapter and section headings.

Related Documents

The C Programming Language by B. W. Kernighan and D. M. Ritchie

The following documents are included on the X11 source tape and are also available from Sun Microsystems, Inc. and Addison-Wesley Publishing Company, Inc.:

OPEN LOOK Graphical User Interface Functional Specification

OPEN LOOK Graphical User Interface Style Guide

The following books in the X Window System series from O'Reilly and Associates, Inc. are currently available:

Volume Zero — *X Protocol Reference Manual*
Volume One — *Xlib Programming Manual*
Volume Two — *Xlib Reference Manual*
Volume Three — *X Window System User's Guide*
Volume Four — *X Toolkit Intrinsics Programming Manual*
Volume Five — *X Toolkit Intrinsics Reference Manual*
Volume Six — *Motif Programming Manual*
Quick Reference — *The X Window System in a Nutshell*

*When referring to all members of a particular package, such as `CANVAS`, the notation `CANVAS_*` will be used. This should not be interpreted as a C-language pointer construct.

Requests for Comments

Please write to tell us about any flaws you find in this manual or how you think it could be improved, to help us provide you with the best documentation possible.

Our U.S. mail address, phone numbers, and e-mail addresses are as follows:

O'Reilly and Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472
in USA and Canada 1-800-998-9938,
international +1 707-829-0515

UUCP: uunet!ora!xview Internet: xview@ora.com

Obtaining the Example Programs

The example programs in this book are available electronically in a number of ways: by *ftp*, *ftpmail*, *bitftp*, and *uucp*. The cheapest, fastest, and easiest ways are listed first. If you read from the top down, the first one that works for you is probably the best. Use *ftp* if you are directly on the Internet. Use *ftpmail* if you are not on the Internet but can send and receive electronic mail to internet sites (this includes CompuServe users). Use BITFTP if you send electronic mail via BITNET. Use UUCP if none of the above works.

FTP

To use FTP, you need a machine with direct access to the Internet. A sample session is shown, with what you should type in boldface.

```
% ftp ftp.uu.net
Connected to ftp.uu.net.
220 FTP server (Version 6.21 Tue Mar 10 22:09:55 EST 1992) ready.
Name (ftp.uu.net:eileen): anonymous
331 Guest login ok, send domain style e-mail address as password.
Password: eileen@ora.com (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/xbook/xview
250 CWD command successful.
ftp> binary (Very important! You must specify binary transfer for compressed files.)
200 Type set to I.
ftp> get xview.ora.examples.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for xview.ora.examples.tar.Z.
226 Transfer complete.
ftp> quit
221 Goodbye.
%
```

If the file is a compressed tar archive, extract the files from the archive by typing:

```
% zcat xview.ora.examples.tar.Z | tar xvf –
```

System V systems require the following tar command instead:

```
% zcat xview.ora.examples.tar.Z | tar xovf –
```

If *zcat* is not available on your system, use separate uncompress and tar commands.

FTPMAIL

FTPMAIL is a mail server available to anyone who can send and receive electronic mail to and from Internet sites. This includes most workstations that have an email connection to the outside world, and CompuServe users. You do not need to be directly on the Internet. Here's how to do it.

You send mail to *ftpmail@decwrl.dec.com*. In the message body, give the name of the anonymous *ftp* host and the *ftp* commands you want to run. The server will run anonymous *ftp* for you and mail the files back to you. To get a complete help file, send a message with no subject and the single word "help" in the body. The following is an example mail session that should get you the examples. This command sends you a listing of the files in the selected directory, and the requested examples file. The listing is useful in case there's a later version of the examples you're interested in.

```
% mail ftpmail@decwrl.dec.com
Subject:
reply jerry@ora.com                (where you want files mailed)
connect ftp.uu.net
chdir /published/oreilly/xbook/xview
dir
binary
uuencode                           (or btoa if you have it)
get xview.ora.examples.tar.Z
quit
%
```

A signature at the end of the message is acceptable as long as it appears after "quit."

All retrieved files will be split into 60KB chunks and mailed to you. You then remove the mail headers and concatenate them into one file, and then *uudecode* or *btoa* it. Once you've got the desired file, follow the directions under FTP to extract the files from the archive.

VMS, DOS, and Mac versions of *uudecode*, *btoa*, *uncompress*, and *tar* are available. The VMS versions are on *gatekeeper.dec.com* in */archive/pub/VMS*.

BITFTP

BITFTP is a mail server for BITNET users. You send it electronic mail messages requesting files, and it sends you back the files by electronic mail. BITFTP currently serves only users who send it mail from nodes that are directly on BITNET, EARN, or NetNorth. BITFTP is a public service of Princeton University. Here's how it works.

To use BITFTP, send mail containing your *ftp* commands to *BITFTP@PUCC*. For a complete help file, send HELP as the message body.

The following is the message body you should send to BITFTP:

```
FTP  ftp.uu.net  NETDATA
USER  anonymous
PASS your Internet email address (not your bitnet address)
CD  /published/oreilly/xbook/xview
DIR
BINARY
GET  xview.ora.examples.tar.Z
QUIT
```

Once you've got the desired file, follow the directions under FTP to extract the files from the archive. Since you are probably not on a UNIX system, you may need to get versions of *uudecode*, *uncompress*, *btoa*, and *tar* for your system. VMS, DOS, and Mac versions are available. The VMS versions are on *gatekeeper.dec.com* in */archive/pub/VMS*.

Questions about BITFTP can be directed to Melinda Varian, *MAINT@PUCC* on BITNET.

UUCP

UUCP is standard on virtually all UNIX systems, and is available for IBM-compatible PCs and Apple Macintoshes. The examples are available by UUCP via modem from UUNET; UUNET's connect-time charges apply.

You can get the examples from UUNET whether you have an account or not. If you or your company has an account with UUNET, you will have a system with a direct UUCP connection to UUNET. Find that system, and type:

```
uucp uunet\!~published/oreilly/xbook/xview/xview.ora.examples.tar.Z yourhost\!~/yourname/
```

The backslashes can be omitted if you use the Bourne shell (*sh*) instead of *csh*. The file should appear some time later (up to a day or more) in the directory */usr/spool/uucppublic/**yourname***. If you don't have an account but would like one so that you can get electronic mail, then contact UUNET at 703-204-8000.

If you don't have a UUNET account, you can set up a UUCP connection to UUNET using the phone number 1-900-468-7727. As of this writing, the cost is 50 cents per minute. The charges will appear on your next telephone bill. The login name is "uucp" with no password. For example, an *L.sys/Systems* entry might look like:

```
uunet Any ACU 19200 1-900-468-7727 login:--login: uucp
```

Your entry may vary depending on your UUCP configuration. If you have a PEP-capable modem, make sure s50=255s111=30 is set before calling.

It's a good idea to get the file */published/oreilly/xbook/ls-lR.Z* as a short test file containing the filenames and sizes of all the files in the directory.

Once you've got the desired file, follow the directions under FTP to extract the files from the archive.

Acknowledgments

I always wanted to do this—but for my first *record album!* :-)

This book was influenced by an amalgamation of several sources: *The SunView Programmer's Manual* for design and structure of the chapters, the people on the XView development team at Sun Microsystems for technical detail and the latest up-to-the-minute changes, and my personal experience in programming for narrative content. Chapter 1, *XView and the X Window System*, is based on Chapter 1 of Volume Four, *X Toolkit Intrinsics Programming Manual*, by Adrian Nye.

This book was created using SoftQuad's *sqtroff*, a PostScript laser printer and a Sun 3/60 color workstation.

Special thanks to everyone at O'Reilly & Associates for their diligent efforts. In particular, Dale Dougherty, Daniel Gilly, Laurel Katz, Lenny Muellner, Chris Reilley, Ruth Terry, and Sue Willing. Many others pitched in for the final push to complete this book.

At Sun Microsystems, I'd like to thank Richard Probst who helped make this entire project possible, Tom Jacobs, for keeping everything in order and reading *all that e-mail*, Tony Hillman, and the rest of the reviewing squad.

Also, Bart Schaefer, for taking care of Mush while I've been too busy. Mike Ilnicki for continuing to play racquetball with me. Penguin's frozen yogurt for nutrition. David Letterman for being on at the perfect time: dinner.

. . . and most of all, I'd like to thank Tim O'Reilly—the only one who could talk me, a cast-in-stone programmer, into trying my hand at technical writing. Thanks for the confidence in me.

Acknowledgments for XView Version 3 Update

Thanks to John Stone for all his help in preparing this manual. John dealt with formatting issues and helped produce several review drafts. Special thanks also go to Darci Chapman for her assistance. Jon Lee edited several chapters of the draft.

Dale Dougherty kept the big picture in sight and managed this project. Thanks to Chris Reilley who did the illustrations. Lenny Muellner, Sue Willing, Michael Sierra, and Eileen Kramer put the final manual together.

The XView developers at Sun Microsystems spent many hours reviewing the new chapters as well as the draft of the complete, updated manual. In particular, Chris Kasso added extensively to the new Version 3 material and answered numerous questions. Thanks to Isa Hashim who made many additions and corrections. Mitch Jerome was very helpful and provided a large amount of new material. Darren Austin, Sri Atreya, Shirley Joe, and Shanmugh Natarajan reviewed several drafts and helped in numerous ways. Stan Raichlen helped to test the code and update the figures. Thanks also go to Tony Hillman, Bhaskar Prabhala, and Greg Kimura. Carole Coffland, Dave Borders, Jeremy Uejio, Dipti Ranganathan, Richard Goldstein, and Martha Venegas read the draft of this manual and made helpful suggestions. Despite all the reviews, any errors that remain are my own.

1
XView and the X Window System

The XView Toolkit allows a programmer to build the interface to an application without having to learn many of the details of the underlying windowing system. However, it is valuable to have some understanding of X before attempting to build applications under XView. This chapter introduces many of the most important concepts on which the X Window System is based and describes the computing environment for X applications. It also describes the role of the XView Toolkit in the X Window System.

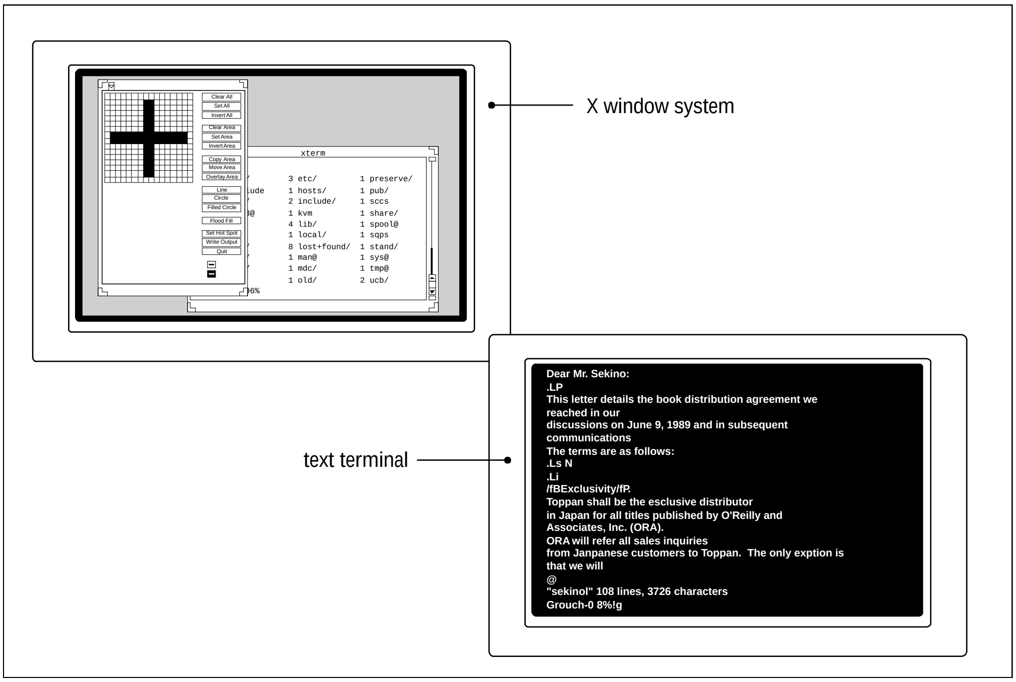
For the most part, this chapter assumes that you are new to programming the X Window System. This chapter describes the basics of the X Window System—further details will be described as necessary later in the manual. However, this book does not repeat the detailed description of Xlib programming found in Volume One, *Xlib Programming Manual*. If you already have some experience programming the X Window System, you might wish to begin at Chapter 2, *The XView Programmer's Model*.

1.1 The X Window System

X controls a bit-mapped display in which every pixel (dot on the screen) is individually controllable. This allows drawing of pictures in addition to text. Until recently, individual control of screen pixels was widely available only on personal computers (PCs) and high-priced technical workstations, while more general-purpose machines were limited to output on text-only terminals. X brings the same world of graphic output to both PCs and more powerful machines. Figure 1-1 shows an X application in comparison with a traditional text terminal.

Like other windowing systems, X divides the screen into multiple input and output areas called windows, each of which can act as an independent *virtual terminal*. Using a terminal emulator, windows can run ordinary text-based applications. However, windows can also run applications designed to take advantage of the graphic power of the bitmapped display.

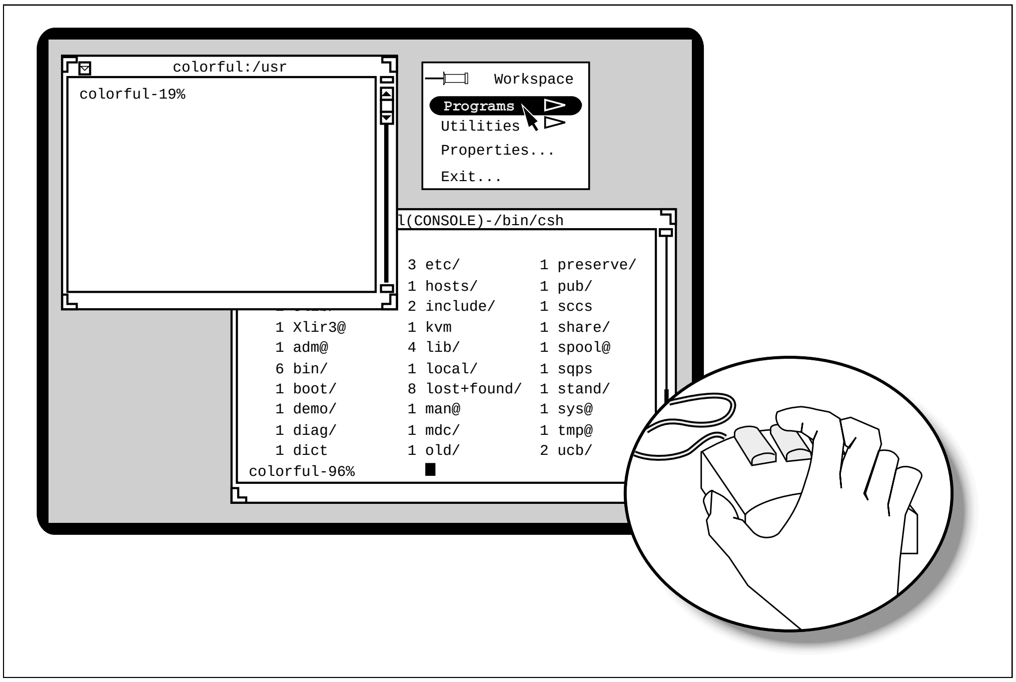
X takes user input from a *pointer*, which is usually a mouse but could just as well be a trackball or tablet. The pointer allows the user to point at certain graphics on the screen and use the buttons on the mouse to control a program without using the keyboard. This method of using programs is often easier to learn than traditional keyboard control, because it is more intuitive. Figure 1-2 shows a typical pointer being used to select a menu item.



```
/         3 etc/          1 preserve/
lude      1 hosts/        1 pub/
/         2 include/      1 sccs
3@        1 kvm           1 share/
          4 lib/          1 spool@
          1 local/        1 sqps
/         8 lost+found/   1 stand/
/         1 man@          1 sys@
/         1 mdc/          1 tmp@
          1 old/          2 ucb/
96%
```

```
Dear Mr. Sekino:
.LP
This letter details the book distribution agreement we
reached in our
discussions on June 9, 1989 and in subsequent
communications
The terms are as follows:
.Ls N
.Li
/fBExclusivity/fP.
Toppan shall be the esclusive distributor
in Japan for all titles published by O'Reilly and
Associates, Inc. (ORA).
ORA will refer all sales inquiries
from Janpanese customers to Toppan.  The only exption is
that we will
@
"sekinol" 108 lines, 3726 characters
Grouch-0 8%!g
```

Figure 1-1. An X application and a traditional text terminal



```
colorful-19%
```

```
                3 etc/          1 preserve/
                1 hosts/        1 pub/
                2 include/      1 sccs
   1 Xlir3@     1 kvm           1 share/
   1 adm@       4 lib/          1 spool@
   6 bin/       1 local/        1 sqps
   1 boot/      8 lost+found/   1 stand/
   1 demo/      1 man@          1 sys@
   1 diag/      1 mdc/          1 tmp@
   1 dict       1 old/          2 ucb/
colorful-96%
```

Figure 1-2. Selecting a menu item with the pointer

Of course, X also handles keyboard input. The pointer is used to direct the input focus of the keyboard (often called the *keyboard focus*) from window to window with only one application at a time able to receive keyboard input.

In X, as in many other window systems, each application need not (and in fact usually does not) consist of only a single window. Any part of an application can have its own separate window, simplifying the management of input and output within the application code. Such *child* windows are only visible within the confines of their parent window. Each window has its own coordinate system where the origin is the upper-left corner of the window inside its border. In basic X, windows are rectangular and oriented along the same axes as the edges of the display. The application or the user can change the dimensions of windows.

Many of the above characteristics are also true of several other window systems. What has made X a standard is that X is based on a network protocol—a predefined set of requests and replies—instead of system-specific procedure calls. The *X Protocol* can be implemented for different computer architectures and operating systems, making X device-independent. Another advantage of a network-based windowing system is that programs can run on one architecture while displaying on another. Because of this unique design, the X Window System can make a network of different computers cooperate. For example, a computationally-intensive application might run on a supercomputer but take input from and display output on a workstation across a network.

1.1.1 The Server and Client

To allow programs to be run on one machine and display on another, X was designed as a network protocol between two processes, one of which is an application program called a *client*, and the other, the *server*. The server is a resource server, controlling a user's resources (such as the display hardware, keyboard, and pointer) and making these resources available to user applications. In other words, the X server isolates the device-specific code from the application.

The server performs the following tasks:

- Allows access to the display by multiple clients. The server can deny access from clients running on certain machines.
- Interprets network messages from clients and acts on them. These messages are known as requests. Some requests command the server to do two-dimensional drawing or move windows, while others ask the server for information.
- Passes user input to clients sending network messages known as events, which represent key or button presses, pointer motion, and so forth. Events are generated asynchronously, and events from different devices might be intermingled. The server must de-multiplex the event stream and pass the appropriate events to each client.
- Maintains complex data structures, including windows and fonts, so that the server can perform its tasks efficiently. Clients refer to these abstractions by ID numbers. Server-maintained abstractions reduce the amount of data that has to be maintained by each client and the amount of data that has to be transferred over the network.

In X, the term *display* is often used as a synonym for server, as is the combined term *display server*. However, the terms *display* and *screen* are not synonymous. A screen is the actual piece of hardware on which the graphics are drawn. Both color and monochrome displays are supported. A server might control more than one screen. For example, one server might control a color screen and a monochrome screen for a user who wants to be able to debug an application on both types of screens without leaving his or her seat.

The communication path between a client and the server is called a *connection*. Several clients may be connected to a single server. Clients may run on the same machine as the server if that machine supports multitasking, or clients may run on other machines in the network. In either case, the X Protocol is used by the client to send requests to draw graphics or to query the server for information; it is used by the server to send user input or replies to requests back to the client. All communication from the client to the server and from the server to the client takes place using the X Protocol.*

It is common for a user to have programs running on several different hosts in the network, all invoked from and displaying their windows on a single screen (see Figure 1-3). Clients running remotely can be started from the remote machine or from the local machine using the network utilities *rlogin* or *rsh*.

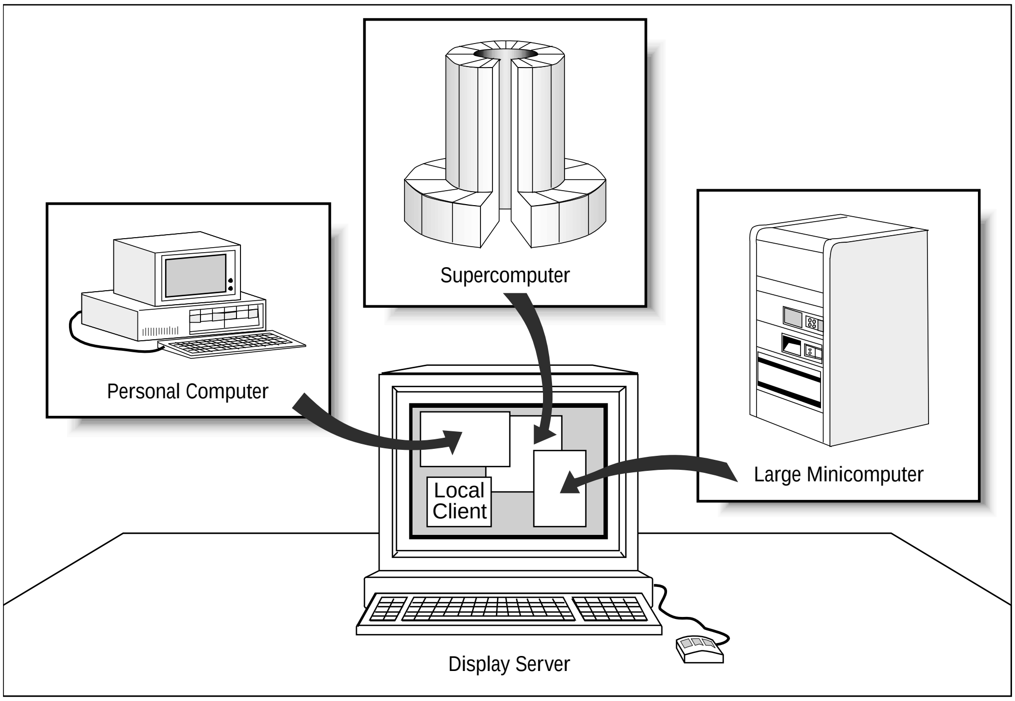


Figure 1-3. Applications can run on any system across the network

*The X Protocol is independent of the networking hardware and runs on top of any network that provides point-to-point packet communication. TCP/IP and DECnet are the only networks currently supported. For more information about the X Protocol, see Volume Zero, *The X Protocol Reference Manual*.

This use of the network is known as *distributed processing*. The most important application of this concept is to provide graphic output for powerful systems that cannot have built-in graphics capabilities. However, distributed processing can also help solve the problem of unbalanced system loads. When one host machine is overloaded, the users running clients on that machine can arrange for programs to run on other hosts.

1.2 The Software Hierarchy

There are many different ways to write X applications because X is not restricted to a single language, operating system, or user interface. The only requirement of an X application is that it generate and receive X Protocol messages.

Figure 1-4 shows the layering of software in an X application. Xlib is the lowest-level C language interface to X. The main task of Xlib is to translate C data structures and procedures into X Protocol events; it then sends them off and receives protocol packets in return that are unpacked into C data structures. Xlib provides full access to the capabilities of the X Protocol but does little to make programming easier. It handles the interface between an application and the network and includes some optimizations that encourage efficient network usage. The list of functions that Xlib performs is so extensive that if the programmer were responsible for handling all these pieces directly, application programs would be too large and prone to performance degradation and potential bugginess. For this reason, *toolkits* are used to modularize the more common functions that handle the user interface portion of an application.

XView is one of a half-dozen or so toolkits available for the X Window System. If you are familiar with other toolkits, you will recognize that the XView Toolkit is equivalent to the Xt Intrinsics and a widget set. Like the Intrinsics, XView is built upon Xlib. It is an object-oriented toolkit that provides reusable, configurable user interface components, equivalent to widgets.*

Toolkits handle many things for the programmer. They provide a framework for combining pre-built user interface components with application-specific code. For example, if the application needs to prompt the user for a filename, a toolkit should provide a component (a *command frame*) that is functionally capable of displaying the query to the user and providing the user's response to the application.

Any user interface component also needs to manage the interpretation of events delivered from the window system. When events are generated, the toolkit decides whether or not to propagate the event to the application or to use it for its own internal purposes. To continue the example, when the user types a filename in the command frame, events are generated in which the interface must decide whether that object should interpret the input or whether it should be sent to the application. A toolkit thus comprises a mechanism to dispatch events and a set of prebuilt interface objects that define the look and feel of an application.

*Widget sets are sometimes loosely referred to as toolkits. However, a toolkit comprises the functions of the Xt Intrinsics layer and one widget set (e.g., the Athena widget set). There are several different widget sets from various vendors that are designed to work with Xt. For more information on Xt Intrinsics-based toolkits, see Volume Four, *X Toolkit Intrinsics Programming Manual*.

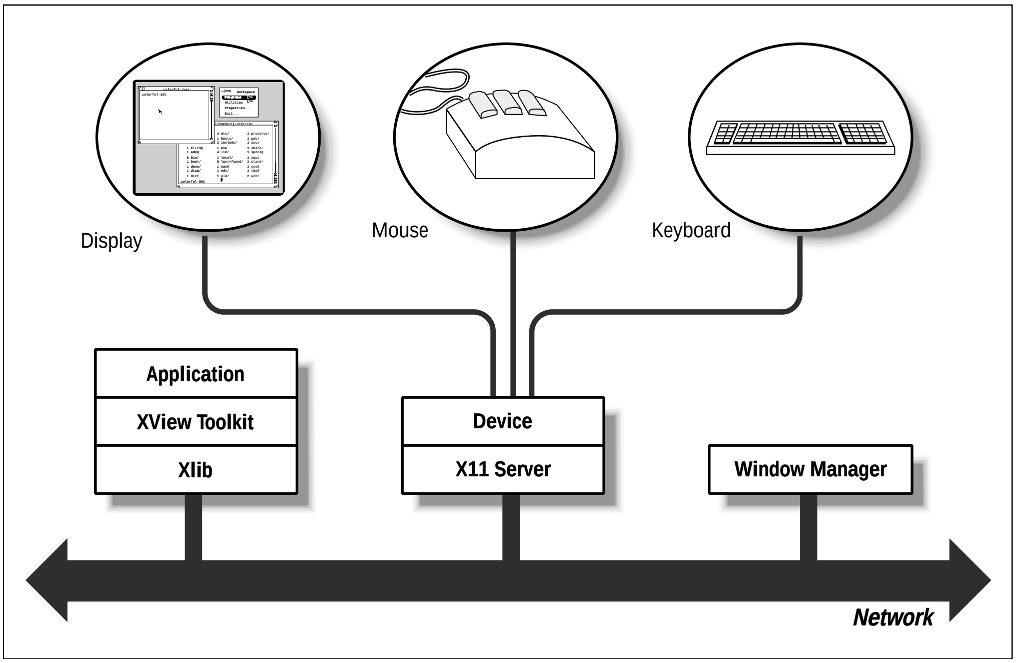


Figure 1-4. The software architecture of X applications

Note that using a toolkit does not preclude calling Xlib directly to accomplish certain tasks such as drawing. In XView, graphics rendering is done most efficiently by using Xlib drawing routines, for instance.

1.3 Extensions to X

Another thing to know about X is that it is *extensible*. The code includes a defined mechanism for incorporating extensions, so that vendors are not forced to modify the existing system in incompatible ways when adding features. An extension requires an additional piece of software on the server side and an additional library at the same level as Xlib on the client side. After an initial query to see whether the server portion of the extension software is installed, these extensions are used just like Xlib routines and perform at the same level.

Among the extensions currently being developed are support for 2D spline curves, for 3D graphics, and for Display PostScript™. These extensions can be used in toolkit applications just like Xlib can.

1.4 The Window Manager

Because multiple applications can be running simultaneously, rules must exist for arbitrating conflicting demands for input. For example, does keyboard input automatically go to whichever window the pointer is in, or must the user explicitly select a window for keyboard input?

Unlike most window systems, X itself makes no rules about this kind of thing. Instead, there is a special client called the *window manager* that manages the positions and sizes of the main windows of applications on a server's display. The window manager is just another client, but by convention, it is given special responsibility to mediate competing demands for the physical resources of a display including screen space, color resources, and the keyboard. The window manager allows the user to move windows around on the screen, resize them, and usually start new applications. The window manager also defines much of the visible behavior of the window system, such as whether windows are allowed to overlap or are tiled (side by side), and whether the keyboard focus simply follows the pointer from window to window or whether the user must click a pointer button in a window to change the keyboard focus (click-to-type).

Applications are required to give the window manager certain information that helps the window manager mediate competing demands for screen space or other resources. For example, an application specifies its preferred size and size increments. These are known as *window manager hints* because the window manager is not required to honor them. The XView Toolkit provides an easy way for applications to set the window manager hints.

The conventions for interaction with the window manager and with other clients have been standardized by the X Consortium as of July 1989 in a manual called the *Inter-Client Communications Conventions Manual* (ICCCM, for short). The ICCCM provides basic policy intentionally omitted from X itself, such as the rules for transferring data between applications (selections), transfer of keyboard focus, layout schemes, colormap installation, and so on. As long as applications and window managers follow the conventions set out in the ICCCM, applications created with different toolkits will be able to coexist and work together on the same server. Toolkit applications should be immune to the effects of the change from earlier conventions.

1.5 Handling Events

The window that X provides is the connection between the XView application and the X server. The reason X windows are important to XView is that these windows are the input targets for the user's focus. They are the actual objects that get events from the user and pass the events through to the XView world.

An X *event* is a data structure sent by the server that describes something that just happened that is of interest to the application. The sources of events are the user's input, the window system, the operating system, and the application programs. For example, the user's pressing a key on the keyboard or clicking a mouse button generates an event, and a window's being moved on the screen also generates events if it changes the visible portions of other applications. It is the server's job to distribute events to the various windows on the screen.

In XView, between the server and the application, there is an event dispatch mechanism called the Notifier, as shown in Figure 1-5.

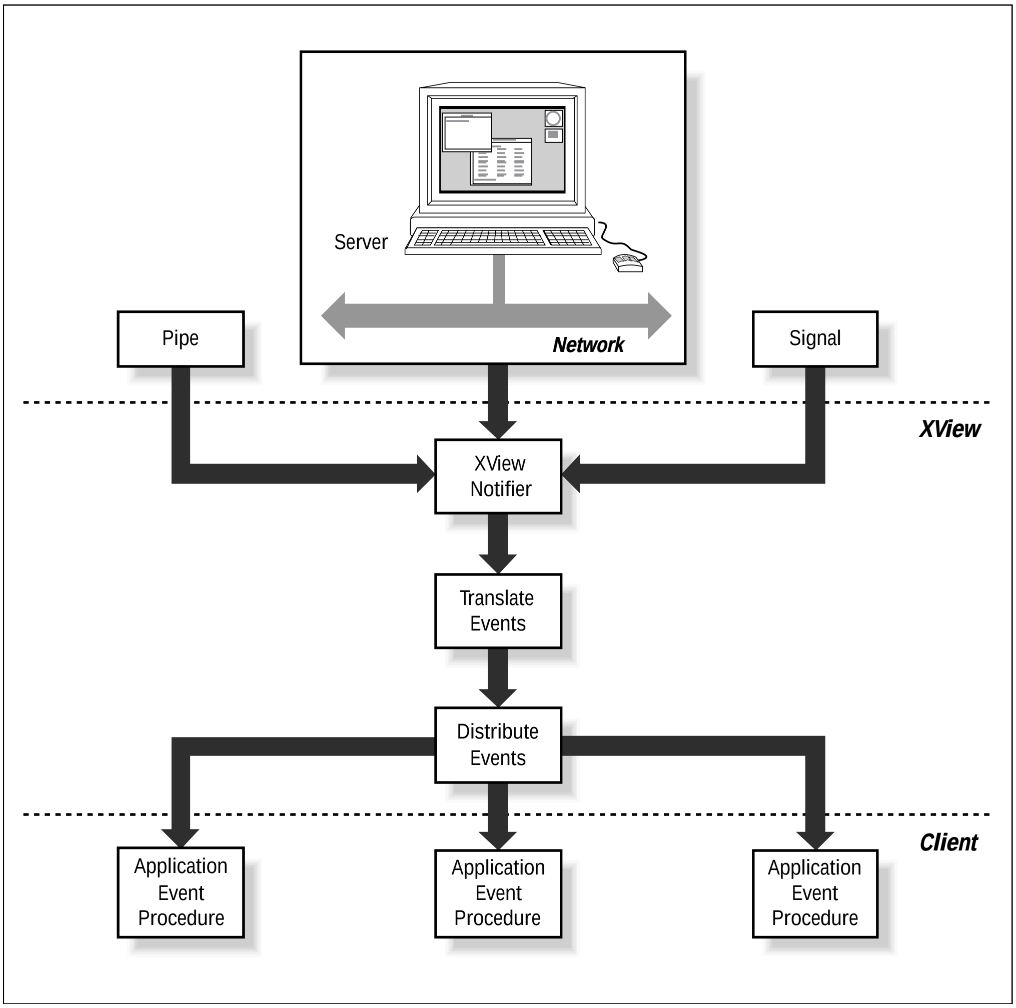

Server
Network
Pipe
Signal
XView
XView Notifier
Translate Events
Distribute Events
Client
Application Event Procedure
Application Event Procedure
Application Event Procedure

Figure 1-5. The Notifier exists between the server and the XView application

After the set-up phase of the application, where you create XView objects such as buttons or scrollbars and determine how they will interact with your application code, you have several choices for input distribution. The simplest method is to hand off control of your application to XView. From then on, the XView Notifier automatically distributes events to the objects created by the application. These objects process many events internally so that your application does not need to get involved.

The key point is that your application is only told about events for which it specifically requested to be notified. By responding to these events, the application can perform its tasks. For example, if the user types the letter *A* in a window, X will pass the event to XView, which in turn can pass it to the application. An application's event handler can interpret this event and display the letter typed in the window. Finally, control returns to the top level so the next event can be read. This is a typical cycle of events that happens for each event generated by the user.

If the application created a scrollbar, then it would track certain events, such as when the scroll button is pressed. XView actually sends a request to the X server to create the X window that will become part of the scrollbar object. In the application's request, it can ask to be notified about or to ignore certain user events in the X window created.

The window system dictates which window gets an event. If the window currently has the keyboard focus and that window does not want to process the event, it has the option of throwing away the event or dropping it to the window below it. This is usually the parent of the window. For instance, if a panel item gets a keyboard press (the user typed an *A*) and if the panel item does not want to deal with the event, then the panel item might be configured so that the event is passed to the item's parent: the panel itself.

The Notifier does not just do input distribution; it also allows selection of different input sources. In addition to handling window system events, your application can also handle a number of interrupts that might be generated by the operating system. Your application can respond to signals, input on a file descriptor and interval timers. You could also pass events between clients in the same process, interpose on a Notifier client to change its behavior, and receive notification on the death of a child process which you have spawned. The use of some of these input sources requires you to call the Notifier directly. The Notifier is covered again in the next two chapters as well as in Chapter 20, *The Notifier*.

1.6 Development of the XView Toolkit

Over the years, Sun Microsystems, Inc. has developed several toolkits, each more well-defined, more functional, and aesthetically superior than the last. SunView 1 was perhaps the first well-accepted user interface toolkit Sun provided. It had all the basic elements necessary to make a functional user interface—including a well-defined API (application programmer's interface). It introduced the attribute-value interface, which we'll examine in more detail in Chapter 2, *The XView Programmer's Model*. It has very few procedure calls. For instance, you use a single function to create all user interface objects. You have the option of using default values, in which case the object is created with only a few lines of code, or of setting the values of specific attributes as required. These attributes can be set at the time the object is created, or later on, by using a different function.

The SunView 1 Toolkit was based on SunWindows™, a kernel-based window system. It required that a single computer control both the application and the user's display and keyboard. The X Window System represents a new generation of window systems. It is server-based, which means that the client application can run on a different system than the server that controls the display.

Today there are several thousand SunView applications, and one of the aims of XView is to make it easy to bring those applications to the X Window System marketplace. In addition, Sun has made the source code to the XView Toolkit freely available. It will be shipped as part of the standard MIT X distribution as well as with UNIX System V Release 4.

XView provides a set of windows that include:

- *Canvases* on which programs can draw.
- *Text subwindows* with built-in editing capabilities.
- *Panels* containing items such as buttons, choice items, and sliders.
- *TTY subwindows* that emulate character-based terminals.

These windows are arranged as *subwindows* within *frames*, which are themselves windows. Frames can be transitory or permanent. Transient interactions with the user can also take place in *menus* which can pop up anywhere on the screen. We will look more at all XView objects when we cover the XView programming model in the next chapter.

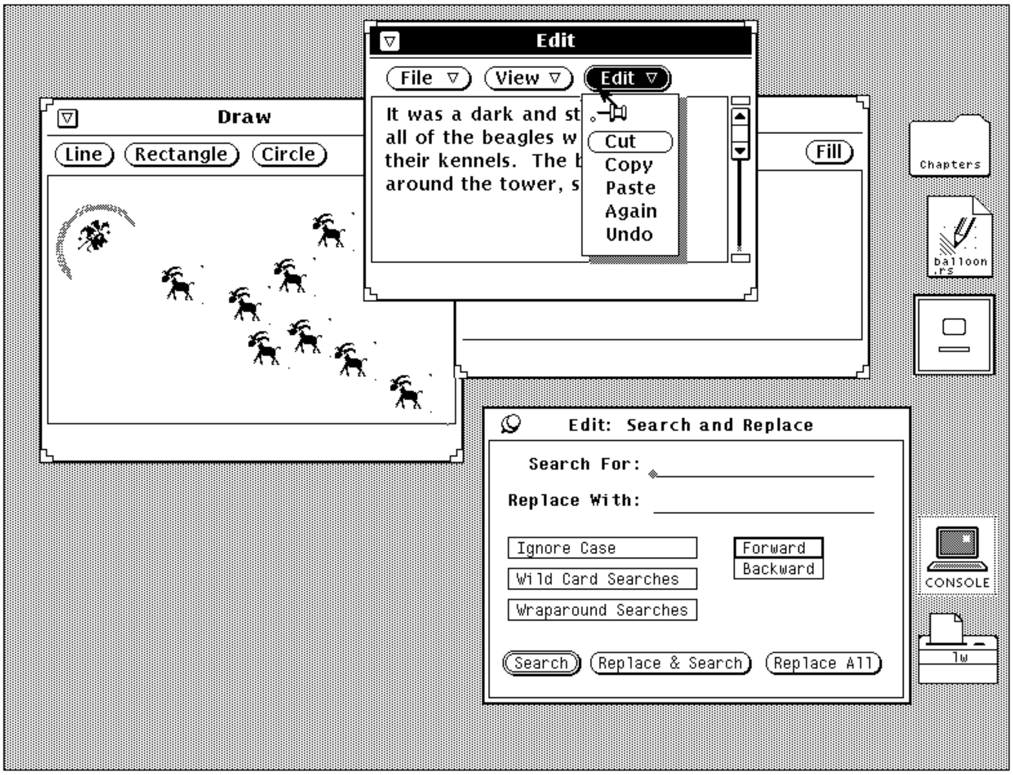
1.7 Versions of the XView Toolkit

Since XView was first released, many applications have been developed for XView and many others have been ported from SunView, as well as from other software platforms. XView development has continued at Sun; new packages have been written, extensions have been added, and the existing packages have been improved based on user's needs. This manual is written for the latest release of XView, Version 3, and it includes descriptions for all the important improvements available in this XView release.

1.8 OPEN LOOK Graphical User Interface

An important feature of the XView Toolkit is that it implements the OPEN LOOK Graphical User Interface (GUI). The OPEN LOOK GUI aims to provide users with a simple, consistent, and efficient interface. An example of an OPEN LOOK workspace is shown in Figure 1-6. OPEN LOOK is supported by Sun and AT&T as the graphical user interface standard for System V Release 4. Users and developers benefit from a standard because it ensures consistent behavior across a number of diverse applications. Programmers can concentrate on the design of the application without having to "invent" a user interface.

A well-defined user interface should be generalized enough so that it can be implemented on any operating system, windowing system, or graphics display. Because OPEN LOOK is not bound by any of these constraints, XView was built based entirely on specifications that could be mapped easily into the X Window System.



It was a dark and st
all of the beagles w
their kennels. The b
around the tower, s

Figure 1-6. A sample OPEN LOOK w

2
The XView Programmer's Model

XView is intended to simplify application development under the X Window System by providing the programmer with a set of predefined user interface components. These components implement the "look and feel" of the OPEN LOOK Graphical User Interface, developed by Sun Microsystems, Inc. and AT&T.

This chapter presents a model of XView for the programmer. It is important to understand this model before you begin writing XView applications. However, you might wish to skim the concepts presented in this chapter and proceed to Chapter 3, *Creating XView Applications*, to examine sample programs.

2.1 Object-oriented Programming

To the programmer, XView is an *object-oriented* toolkit. XView objects can be considered building blocks from which the user interface of the application is assembled. Each piece can be considered an *object* from a particular *package*. Each package provides a list of properties from which you can choose to configure the object. By selecting objects from the available packages, you can build the user interface for an application.

XView is based on several of the fundamental principles of object-oriented programming:

- Objects are represented in a class hierarchy.
- Objects are opaque data types.
- Objects have attributes which can be set via message passing functions.
- Objects may have callback procedures that are triggered by events.

We will look at how these concepts are implemented in XView in the sections that follow.

2.1.1 Object Class Hierarchy

XView defines classes of objects in a tree hierarchy. For example, *frame* is a subclass of the more general class *window*, which in turn is a subclass of *drawable*. Drawable, like user interface object classes, is a subclass of the *Generic Object* class. Figure 2-1 shows the XView class hierarchy and the relationships between the classes. Each class has identifying features that make it unique from other classes or packages. In XView, a class is often called a *package*, meaning a set of related functional elements. However, there are XView packages that are not members of the object class hierarchy, such as the Notifier package.

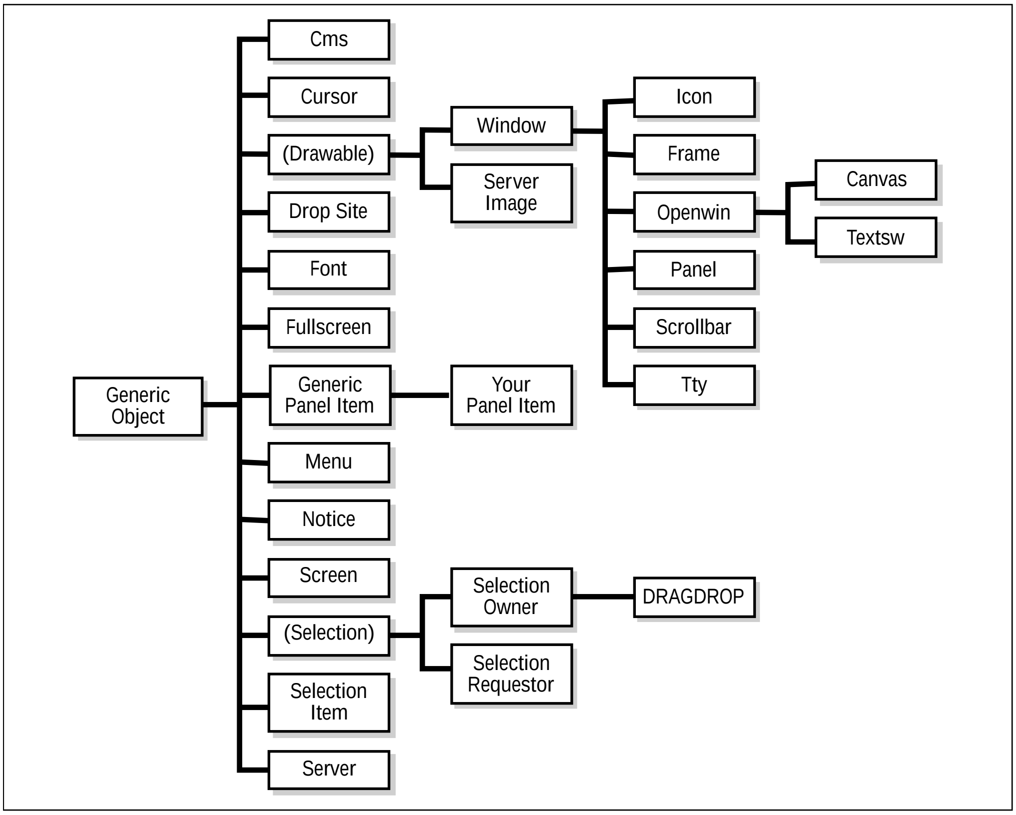


Figure 2-1. XView class hierarchy

Some objects are visual and others are not. Examples of visual objects include windows, scrollbars, frames, panels, and panel items. Nonvisual objects are objects which have no appearance, per se, but they have information which aids in the display of visual objects. Examples of nonvisual objects include the server, screen, and font objects. The screen, for example, provides information such as the type of color it can display or the default foreground and background colors that objects might inherit. The display can provide information about what fonts are available for objects that display text.

All objects, both visual and nonvisual, are a part of this object classing system. The system is extensible, so you can create new classes that might or might not be based on existing classes.

XView uses static subclassing and chained inheritance as part of its object-oriented model. All objects of a particular class inherit the properties of the parent class (also known as a superclass). The Generic Object `XV_OBJECT` contains certain basic properties that all objects share. For example, the same object can appear in many places on the screen to optimize storage. To keep a record of this, the Generic Object maintains a reference count of its instances. Since all objects have an owner, the parent of the object is stored in a field of the generic part of the object. As the needs of an object get more specific to a particular look or functionality, lower-level classes define properties to implement it.

Each class contains properties that are shared among all instances of that object. For example, *panels* are a part of the `PANEL` package, which has properties that describe, among other things, its layout (horizontal or vertical) or the spacing between items (buttons) in the panel. All panels share these properties, even though the state of the properties might differ for each instance of the object.

As mentioned earlier, XView uses subclassing so that each package can inherit the properties of its superclass. The `PANEL` package is subclassed from the `WINDOW` package, which has properties specific to all windows, such as window dimensions, location on the screen, border thickness, depth, visual, and colormap information. The `WINDOW` package is subclassed from the root object `XV_OBJECT`, as are all objects, and the panel can access generic information such as the size and position of itself.

2.1.2 Object Handles

When you create an object, the XView function returns a *handle* for the object. Later, when you wish to manipulate the object or inquire about its state, you pass its handle to the appropriate function. This reliance on object handles is a way of *information-hiding*. The handles are *opaque* in the sense that you cannot see through them to the actual data structure which represents the object.

Each object type has a corresponding type of handle. Since C does not have an `opaque` type, all the opaque data types mentioned above are `typedef`'d to the XView type `Xv_opaque` or `Xv_object`.

In addition to the opaque data types, there are several `typedef`s that refer not to pointers but to structures: `Event`, `Rect`, and `Rectlist`. Generally pointers to these structures are passed to XView functions, so they are declared as `Event *`, `Rect *`, etc. The reason that the asterisk (*) is not included in the `typedef` is that the structures are publicly available.

Table 2-1 lists each XView object, its owner, the package that defines it, and its data type.

Table 2-1. XView Objects, Owners, Packages, and Data Types

Name	Owner	Package	Data Type
canvas	frame	`CANVAS`	`Canvas`
canvas view	window or screen	`CANVAS_VIEW`	`Canvas_view`
cms	window	`CMS`	`Cms`
cursor	window or screen	`CURSOR`	`Xv_Cursor`
drag drop	window	`DRAGDROP`	`Dnd`
drop site item	window	`DROP_SITE_ITEM`	`Drop_site_item`
font	root window	`FONT`	`Xv_Font`
frame	frame or root window	`FRAME`	`Frame`
base frame	frame or root window	`FRAME_BASE`	
command frame	frame or root window	`FRAME_CMD`	
property frame	frame or root window	`FRAME_PROPS`	
fullscreen	root window	`FULLSCREEN`	`Fullscreen`
icon	window or screen	`ICON`	`Icon`
menu	server	`MENU`	`Menu`
command menu	null	`MENU_COMMAND_MENU`	`Menu`
choice menu	null	`MENU_CHOICE_MENU`	`Menu`
pullright menu	menu item	`MENU`	`Menu`
toggle menu	null	`MENU_TOGGLE_MENU`	`Menu`
menu item	menu	`MENUITEM`	`Menu_item`
openwin	frame	`OPENWIN`	`Openwin`
notice	window	`NOTICE`	`Xv_Notice`
panel	frame	`PANEL`	`Panel`
panel button	panel	`PANEL_BUTTON`	`Panel_button_item`
panel choice	panel	`PANEL_CHOICE`	`Panel_choice_item`
panel drop site	panel	`PANEL_DROP_SITE`	`Panel_drop_site_item`
panel item	panel	`PANEL_ITEM`	`Panel_item`
panel list	panel	`PANEL_LIST`	`Panel_list_item`
panel message	panel	`PANEL_MESSAGE`	`Panel_message_item`
panel multi-line text	panel	`PANEL_MULTILINE_TEXT`	`Panel_multiline_ text_item`
panel numeric text	panel	`PANEL_NUMERIC_TEXT`	`Panel_numeric_ text_item`
panel slider	panel	`PANEL_SLIDER`	`Panel_slider_item`
panel text	panel	`PANEL_TEXT`	`Panel_text_item`
screen	null	`SCREEN`	`Screen`
scrollbar	panel or canvas	`SCROLLBAR`	`Scrollbar`
selection	window	`SELECTION`	`Selection`
selection owner	window	`SELECTION`	`Selection_owner`
selection requestor	window	`SELECTION`	`Selection_requestor`
selection item	selection owner	`SELECTION_ITEM`	`Selection_item`
server	null	`SERVER`	`Server`
server image	screen	`SERVER_IMAGE`	`Server_image`

Table 2-1. XView Objects, Owners, Packages, and Data Types (continued)

Name	Owner	Package	Data Type
text subwindow	frame	`TEXTSW`	`Textsw`
tty	frame	`TTY`	`Tty`
window	frame	`WINDOW`	`Xv_Window`

2.2 Attribute-based Functions

A model such as that used by XView, which is based on complex and flexible objects, presents the problem of how the client is to manipulate the objects. The basic idea behind the XView interface is to provide a small number of functions, which take as arguments a large set of *attributes*. For a given call to create or modify an object, only a subset of all applicable attributes will be of interest.

2.2.1 Creating and Manipulating Objects

There is a common set of functions that allows the programmer to manipulate any object by referencing the object handle. The functions are listed in Table 2-2.

Table 2-2. Generic Functions

Function	Role
`xv_init()`	Establishes the connection to the server, initializes the Notifier and the Defaults/Resource-Manager database, loads the Server Resource Manager database, and parses any generic toolkit command line options.
`xv_create()`	Creates an object.
`xv_destroy()`	Destroys an object.
`xv_find()`	Finds an object that meets certain criteria; or if the object doesn't exist, creates it.
`xv_get()`	Gets the value of an attribute.
`xv_set()`	Sets the value of an attribute.

Using these six routines, objects can be created and manipulated from all packages available in XView. When the programmer wants to create an instance of an object from a certain package, the routine `xv_create()` is used. For example:

```
Panel panel;
panel = xv_create(panel_parent, PANEL, NULL);
```

Here, an instance of a panel has been created from the `PANEL` package. All its attributes are set to the panel's *default* properties because no object-specific attributes have been specified.

A handle to the new panel object is returned and stored in the variable `panel`. This handle is not a pointer and does not contain any useful information about the object itself.

The next section goes into detail about the use of `xv_set()` and `xv_get()`. Chapter 3, *Creating XView Applications*, discusses the use of `xv_init()`, `xv_destroy()`, and `xv_find()`.

2.2.2 Changing Object Attributes

The programmer uses the handle returned from the `xv_create()` function as a parameter to the functions `xv_get()` and `xv_set()` to get and set attributes of the object.

```
panel = xv_create(...)
xv_set(panel, PANEL_LAYOUT, PANEL_HORIZONTAL, NULL);
```

Here, the handle to the panel (`panel`) is used to change a `PANEL` package attribute, `PANEL_LAYOUT`, whose value is set to `PANEL_HORIZONTAL`. The attribute and value form an *attribute-value pair*. The functions `xv_create()`, `xv_destroy()`, `xv_find()`, `xv_set()`, and, to some extent, `xv_get()` use attribute-value pairs. The functions can have any number of pairs associated with the function call. These *variable argument lists* are always terminated by a `NULL` pointer as the last argument in the list. Note that `NULL`, not the constant 0 (zero), should be used as the terminating argument.

The effect of this function call is to change the layout of the panel from the previous value, whatever it might be, to horizontal.

2.2.3 Types of Attributes

Attributes can be divided into three categories. Those that apply to all XView objects are termed *generic* attributes. Attributes that are supported by many, but not all objects, are termed *common* attributes. Attributes that are associated with a particular package or class of objects are called *specific* attributes.

XView uses naming conventions to simplify the identification of the task of an attribute. Those attributes that apply to a specific package have their name prefixed by the package name. The attributes have prefixes that indicate the type of object they apply to, i.e., `CANVAS_*`, `CURSOR_*`, `FRAME_*`, `ICON_*`, `MENU_*`, `PANEL_*`, `SCROLLBAR_*`, `TEXTSW_*`, `TTY_*`, etc.

Common and generic attributes apply to several different object types and are prefixed by `XV_`. For example, the generic attribute `XV_HEIGHT` applies to all objects since all objects must have a height. In contrast, attributes that apply only to windows are prefixed by `WIN_`. Attributes such as `WIN_HEIGHT` and `WIN_WIDTH` apply to all windows regardless of whether they happen to be panels or canvases.

The value part of an attribute-value pair can differ from attribute to attribute. The reason for this is that the attribute may describe a wide range of values. If the attribute describes the height or width of an object, the value associated with the attribute will be an integer.

However, sometimes the attribute requires a variable-length list of values—this too must be NULL-terminated.

Look at the following code fragment that specifies an attribute-value list at the creation of a panel item:

```
Panel_item panel_item;
panel_item = xv_create(panel, PANEL_CHOICE_STACK,
    XV_WIDTH,             50,
    XV_HEIGHT,            25,
    PANEL_LABEL_X,        100,
    PANEL_LABEL_Y,        100,
    PANEL_LABEL_STRING,   "Open File"
    PANEL_CHOICE_STRINGS, "Append to file",
                          "Overwrite contents",
                          NULL,
    NULL);
```

All the attributes except PANEL_CHOICE_STRINGS take a single value. The PANEL_CHOICE_STRINGS attribute takes a list of strings, and that list is NULL-terminated. The last NULL terminates the list of attribute-value pairs passed to the xv_create() function.

Don't worry for now what each of these attributes does. Simply notice the mixture of generic attributes (XV_WIDTH and XV_HEIGHT) and class-specific attributes (all the PANEL_* attributes). Because all packages are subclasses of the XV_OBJECT package, the XV_* attributes can be used with all xv_create() calls.

2.3 Internal Attribute-Value Lists

For a discussion of the way that XView handles attribute-value lists internally, see Chapter 25, *XView Internals*. The subject is important for those who wish to write XView extensions or utilize the advanced features of the error package, but programmers interested in general XView programming usage can skip that chapter.

2.4 Types of Objects

The following section describes on a conceptual level the different types of objects that XView offers. In many cases, figures taken from the *OPEN LOOK GUI Specification Guide* are used to show the appearance of the object. Details about the objects themselves, how to create them, their properties, their default values, and so forth are discussed in later chapters that are specific to those object packages. A list of the objects that can be created include:

- Generic Objects
- Windows
- Frames
- Openwins

- Canvases
- Text Windows
- Menus
- Scrollbars

2.4.1 Generic Objects

The Generic Object is the root object of the class hierarchy. One never creates an instance of a Generic Object because, by itself, it has no function. Figure 2-2 shows the path taken when an object is created.

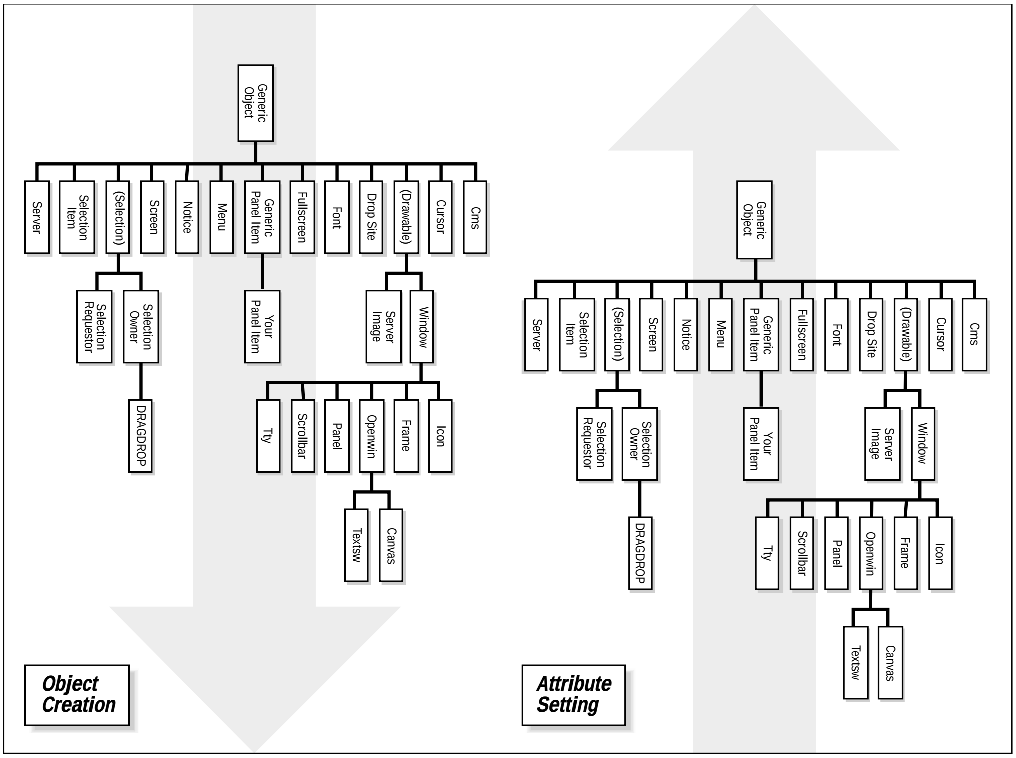


Figure 2-2. Object creation is top down; attribute setting is bottom up

First, the Generic Object is created; then the subclass of that object is created all the way down until the object class of the type of object desired is created. At that point, a complete instance of the object has been created with all the default properties of the classes set. If there were any attribute-value pairs specified in the `xv_create()` call, those attributes are set in reverse order—the attributes specific to the class of the instance of the object are set first, followed by its parent's class attributes and so on, until the generic attributes are set.

Consider the code below, which creates a panel:

```
extern Xv_Font font;
Panel panel;

panel = xv_create(frame, PANEL,
                  XV_Y,        5,
                  WIN_HEIGHT,  50,
                  PANEL_FONT,  font,
                  NULL);
```

When the panel is created, the first thing created is a generic object. Next a window instance is created, followed by a panel object. Each is created with the default properties of the object specific to each class.

Here, the reverse traversal takes place, and the attributes specified in the `xv_create()` call for each class are set to override the default properties inherited from the class. First, the panel package attributes are set. The panel's default font is controlled by the attribute `PANEL_FONT`; its assigned value, *`font`*, must be previously initialized. Then the window package attributes are set. The panel's window width is controlled by the attribute `WIN_WIDTH` which is not explicitly set, so its assigned value defaults to `WIN_EXTEND_TO_EDGE`. This value indicates that the width of the window should always be the width of its parent. The height of the window, however, is specified. So the window package sets the height to be 50 pixels.

Finally, the generic attributes are set. The panel's *x* and *y* location, indicating where it should be placed within its parent, is controlled by setting the `XV_X` and `XV_Y` attributes. The example sets the *y* position only; the *x* position is not set because the window package is told to extend the width of the panel to the edges of its parent. The parent in this case is the object `frame` (which is presumed to be from the `FRAME` package).

2.4.2 Window Objects

Many XView objects contain X windows in order to display themselves and receive events. Examples include frames, tty windows, scrollbars, and icons.

The XView window class, like the Generic Object class, is a hidden class: a *window* object is never explicitly created. Rather, an object that is a subclass of the window class is created. This includes most visual objects with the exception of panel items.

Nonvisual objects are so named because they do not contain, or are not a subclass of, windows. Fonts, for example, are displayed in windows, or in a memory image or somewhere that contains a bitmap, but fonts do not contain or require windows to be used.

Some attributes of windows include depth (`XV_DEPTH`), the border width around their perimeter (`WIN_BORDER_WIDTH`) as well as foreground and background colors.

2.4.3 Frames and Subframes

There are two kinds of frames:

- Base Frames
- Pop-up Frames

With one exception, all frames are free-floating windows that contain subwindows that are bound by the frame and tiled (they do not overlap one another). Base frames reside on the root window and are not constrained by any other window, though all frames can overlap one another. The base frame is also known as the application's frame. (More than one base frame may be associated with an application.) Subframes are frames whose owner is a frame; they are controlled by the base frames of the application. For example, extraneous dialog boxes (subframes) will go away if the main application's base frame is *iconified* (closed). Figure 2-3 shows an example of a fully-featured base frame from the *OPEN LOOK GUI Specification Guide*.

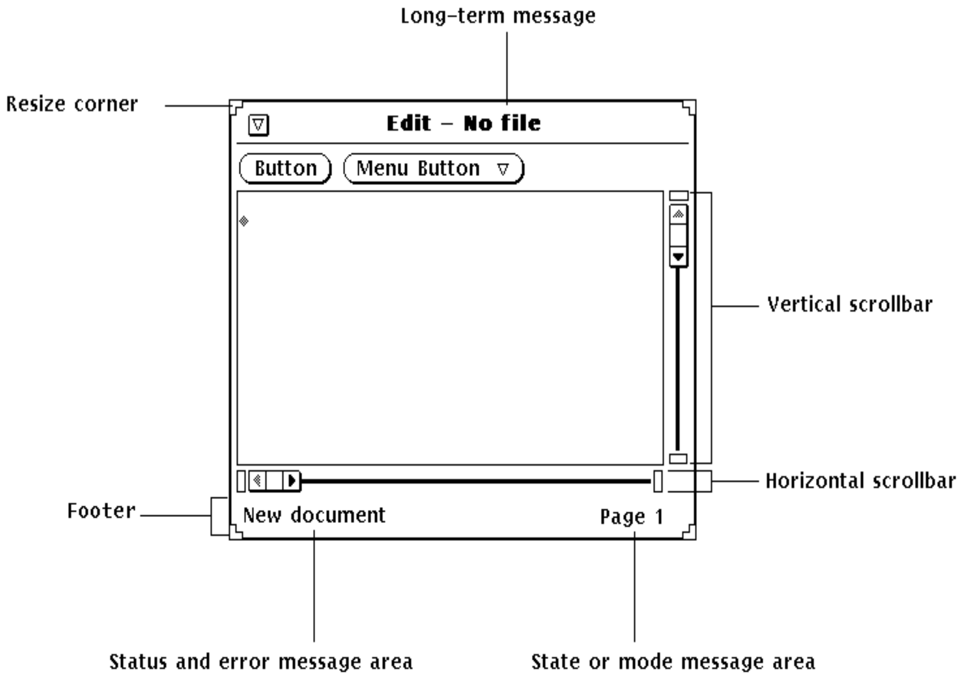


Figure 2-3. Fully-featured base frame (includes optional elements)

Chapter 4, *Frames*, goes into more detail about the elements of a frame and how to set attributes and override default values. It should be noted that many features of the frame are attributes of the window manager. Figure 2-3 assumes an OPEN LOOK-compliant window manager; if another window manager is used, base frames might not look the same. XView defines attributes that give hints to the window manager to provide such features as title bar information, resize corners, and so on. If a non-OPEN LOOK window manager is used, there is no guarantee that these attributes will have any effect.

Pop-up frames are typically used to perform one or more transient functions. They are not intended to stay up after the set of functions has been completed, although they might remain up if the user or the application so chooses. This functionality can be handled by a pushpin at the upper-left corner of the frame. There are different kinds of pop-up frames:

Command Frames	give operands and set parameters needed for a command. This is implemented as a subframe that contains a default panel.
Help Frames	display help text for the object under the pointer. This is implemented as a text subwindow within a subframe.
Notices	are special pop-up windows that are used to confirm requests, to display messages and conditions that must be brought to the user's attention and handled immediately. These require immediate attention and can suspend the application by disallowing the focus from leaving the Notice.

Figure 2-4 shows a sample unpinned command frame from the *OPEN LOOK GUI Specification Guide*.

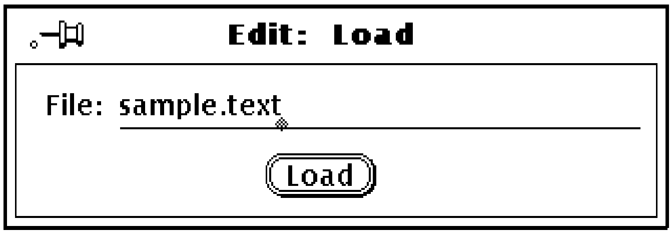
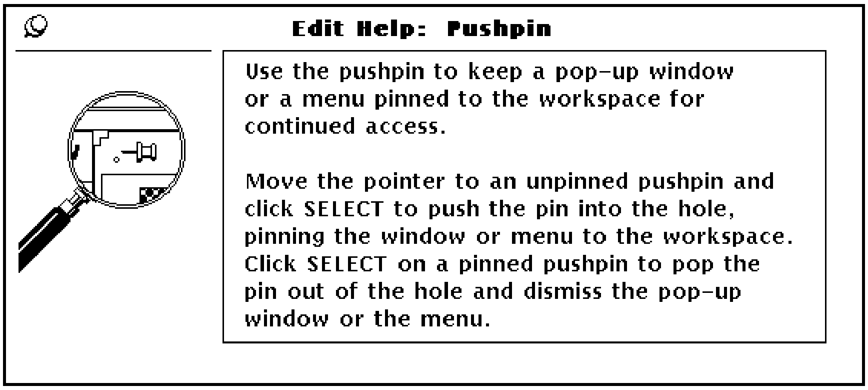


Figure 2-4. Sample unpinned command frame

The user may select the option, and the frame will be *dismissed* (be undisplayed). The pushpin at the upper-left corner is out of its hole. If the pushpin were in, the command frame would remain visible even after the user selects an action to take. Figure 2-5 shows a sample pinned help frame from the *OPEN LOOK GUI Specification Guide*.


Edit Help: Pushpin

Use the pushpin to keep a pop-up window or a menu pinned to the workspace for continued access.

Move the pointer to an unpinned pushpin and click SELECT to push the pin into the hole, pinning the window or menu to the workspace. Click SELECT on a pinned pushpin to pop the pin out of the hole and dismiss the pop-up window or the menu.

Figure 2-5. Sample help window

Figure 2-6 shows a sample notice from the *OPEN LOOK GUI Specification Guide*.

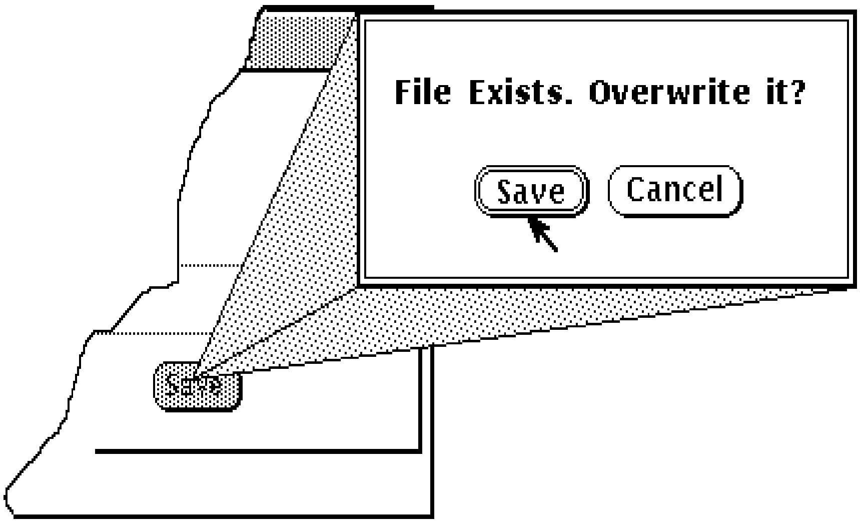

File Exists. Overwrite it?

Figure 2-6. Sample notice

The user can do nothing but choose either Save or Cancel. Choosing either one will cause the notice to be dismissed immediately.

2.4.4 Subwindows

Subwindows differ from frames in several basic ways. They never exist independently; they are always owned and maintained by a frame or another window, and they may not themselves own frames. While frames can be moved freely around the screen, subwindows are constrained to fit within the borders of the frame to which they belong. Also, in contrast to frames, subwindows are *tiled*—they may not overlap each other within their frame. Within these constraints (which are enforced by a run-time *boundary manager*), subwindows may be moved and resized by either a program or a user.

Canvas subwindows and text subwindows are subclassed from the `OPENWIN` package, a hidden class which implements the notion of *splittable views* described by OPEN LOOK. Figure 2-7 shows an example of one canvas object providing separate views into one graphic image. Each view into the object has scrollbars attached. The scrollbars provide the ability to scroll independently from all the other views attached to the subwindow and to split the views again.

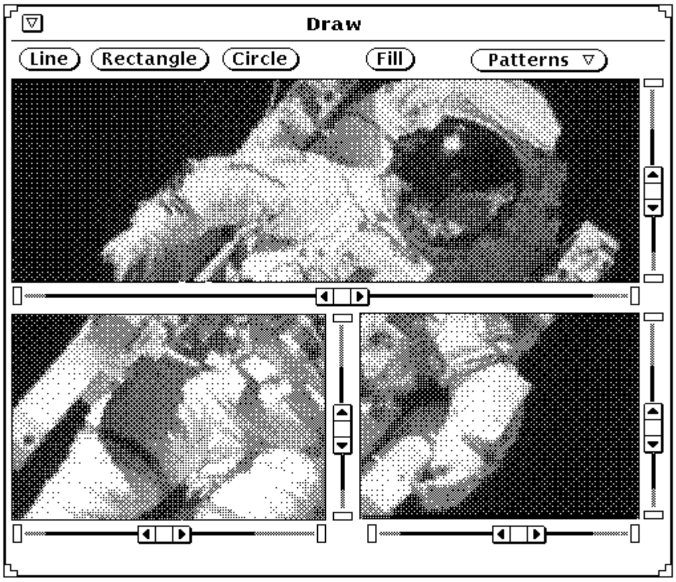


Figure 2-7. A window with multiple views

All the views, however, are still a part of the same `OPENWIN` object. Using the scrollbars, the user can split or join different views.

2.4.4.1 Canvas subwindows

The canvas is the most basic type of subwindow. It provides a drawing surface—a place in which the result of Xlib graphics calls can be displayed. A canvas object can be configured to permit the application to draw on an area larger than the size of the visible window. The entire region representing the drawing surface is a window called the *paint window*. The visible portion of the paint window is the *view window*. It is the view window that appears in the canvas subwindow. In the previous figure, the paint window contains a picture of an astronaut. Multiple view windows each show a particular region of the paint canvas. The view windows are independent of each other. See Chapter 5, *Canvases and Openwin*, for a full discussion and illustration of the Canvas model.

2.4.4.2 Text subwindows

Another basic window type is a text subwindow. It provides basic text editing capabilities using the OPEN LOOK text editing model.

2.4.4.3 Panels

A panel (or *control area*) is an unbordered region of a window where controls such as buttons and settings are displayed. The panel also controls the arrangement of its controls in a horizontal or vertical fashion. The panel shown in Figure 2-8 presents the typical positioning of the control area—the top of a base frame with a canvas subwindow under it.

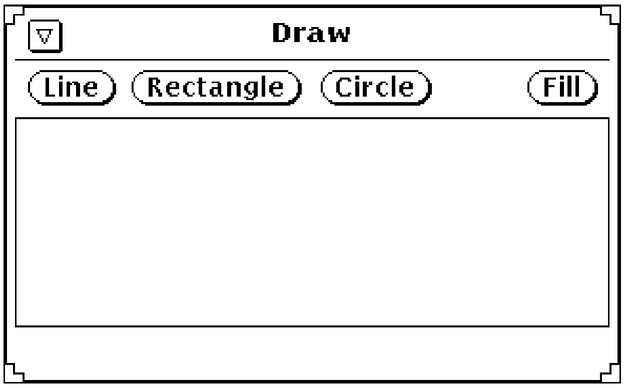


Figure 2-8. A control area above a subwindow

The panel shown in Figure 2-9 presents a control area that is to the right of a canvas subwindow.

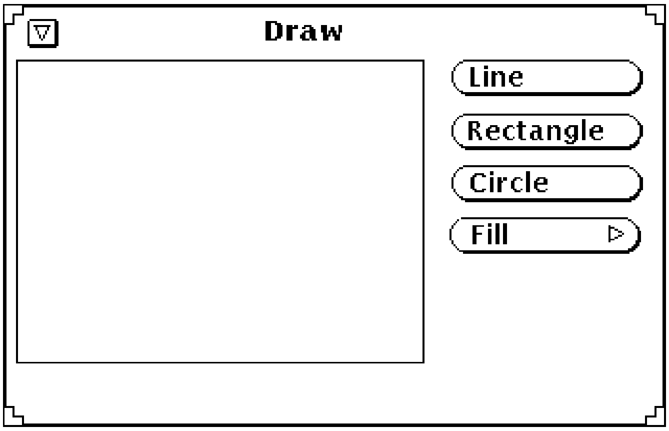


Figure 2-9. A control area to the right of a pane

Control areas within panes usually contain varied combinations of the following controls:

- Buttons
- Check boxes
- Drop Target Items
- Exclusive and nonexclusive choice lists
- Gauges
- Sliders
- Text and numeric fields

A command frame (subframe) contains only a panel and no other subwindows. Figure 2-10 shows a control area in a command frame. It contains text fields, choice lists, and buttons. See Chapter 7, *Panels*, for a discussion of panel items and the `PANEL` package.

2.4.4.4 Menus

Menus are subclassed from the Generic Object. A menu by itself is a windowless object. Only when the menu is activated by the user is it bound to a window. This implementation avoids creating multiple X11 windows (one for each menu) since not all the menus will be displayed at once. XView has three types of menus:

1. Pop-up menus that are displayed when the user presses the menu button in a window.
2. Pullright menus that are displayed as a menu to the right of a menu.
3. Pulldown menus that are displayed below a menu button on a panel.

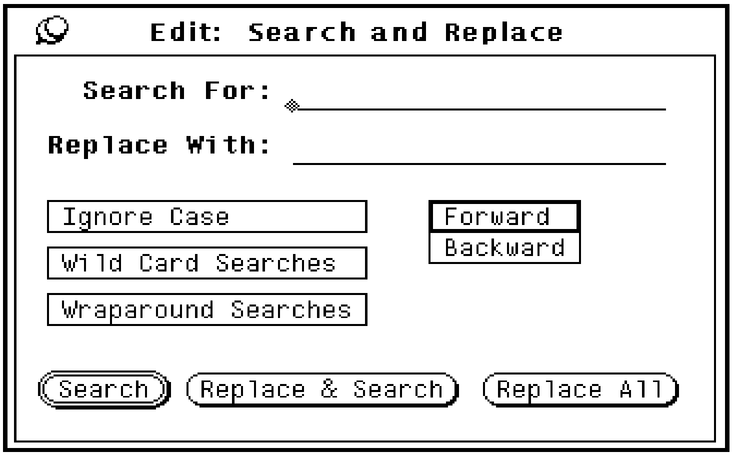


Figure 2-10. A control area in a command window

Figure 2-11 shows an example of a pop-up menu on the left; on the right, a pullright submenu is displayed.

Pushpins can be used in some menus, allowing them to be *pinned* so that the menu remains on the screen for repeated use.

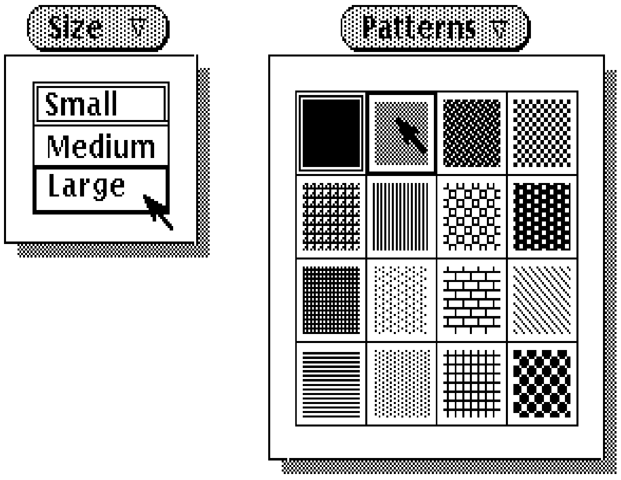


Figure 2-11. Example of a pop-up menu with a pullright submenu

2.4.4.5 Scrollbars

Scrollbars implement the OPEN LOOK metaphor of an elevator on a cable. These components are shown in Figure 2-12.

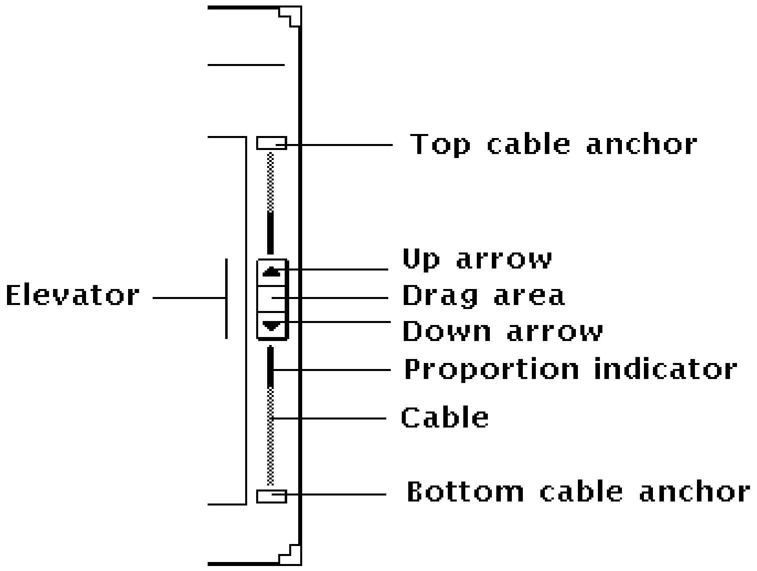


Figure 2-12. Vertical scrollbar components

A scrollbar is an object that can exist independently or attach itself to various types of subwindows. The scrollbar is subclassed from the `WINDOW` class since it is a visual object. However, because its functionality is very tightly bound to other objects, the scrollbar is sometimes considered to be a property of those objects. `OPENWIN` subclasses (canvas and text-based packages) require scrollbars to provide splittable views, and scrollbars can be created automatically by such objects. Typically, it is your responsibility to pass a hint to the object that it should create the scrollbar using the appropriate attribute-value pair. Nevertheless, scrollbars can be manually attached or detached to `OPENWIN` objects, or they can be created independently of these objects for other purposes entirely.

The `SCROLLBAR` package manages only the scrollbar window. It does not control the window to which it is attached. When a scrolling action results from the user clicking on a portion of the scrollbar, the window to which the scrollbar is attached must modify its data (a view in most cases). It is not the scrollbar's responsibility to notify the window it is attached to. The scrollbar informs the object interested in its scrolling by use of callback routines that the owner of the scrollbar must install.

Scrollbars can be oriented vertically or horizontally, but some packages might not allow a particular scrollbar orientation. Text subwindows, for example, contain vertical scrollbars by default but do not permit horizontal scrollbars.

2.4.4.6 Icons

An icon is a small image representing the application when the application's frame is in a closed, or iconified, state. The `ICON` package is very small. It is subclassed from the `WINDOW` package because it is a window that displays graphics and accepts input. The only attributes that you can set in the `ICON` package specify the image to display in the window and the geometry of the image. Other important attributes that an icon can have (such as width, height, label, and font) are attributes of the generic class.

2.4.5 Nonvisual Objects

There are several nonvisual objects that cannot be represented on the screen but are subclassed from the Generic Object:

CMS	Colormap segments (cms) are objects that are associated with windows which provide their color specifications. `Cms` objects may be shared by multiple windows.
DROP_SITE_ITEM	The drop site item is a rectangle that is an area used for dragging an object and dropping data associated with the object onto the drop site's application.
FONT	The font package allows the programmer to request fonts of varying attributes such as font *family* and *style*. Fonts can be accessed by name, size or scaling.
SCREEN	This object describes the visual and other characteristics of the physical screen. This object is separate from the Xlib SCREEN object.
SELECTION	This package allows clients to transfer data between applications.
SERVER	This package interacts with the X server. The window-server is the program that does the drawing to the screen and receives the user's input. The server also maintains font information and user-configurable resources, which can be set for specific applications.

These objects are closely tied with the X Window System, and they are manipulated by making requests to set or get attributes from X.

2.5 The Notifier Model

XView is a *notification-based* system. The Notifier acts as the controlling entity within a user process, reading input from the operating system and formatting it into higher-level *events*, which it distributes to the different XView objects.*

2.5.1 Callback Style of Programming

In the conventional style of interactive programming, the main control loop resides in the application. An editor, for example, will read a character, take some action based on the character, then read the next character, and so on. When a character is received that represents the user's request to quit, the program exits. Figure 2-13 illustrates this conventional approach.

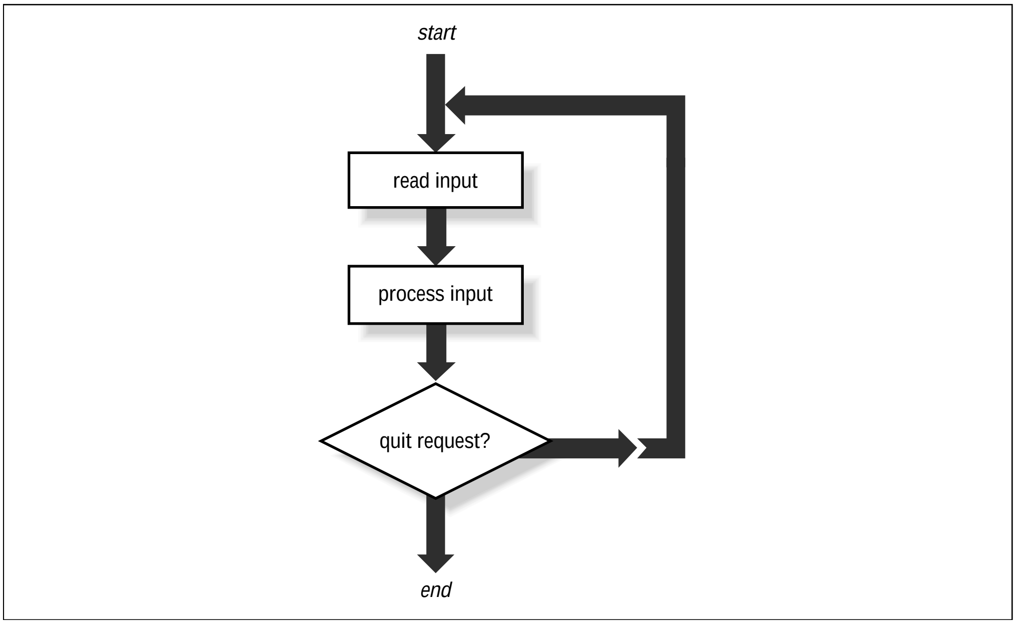


Figure 2-13. Flow of control in a conventional program

Notification-based systems invert this straight line control structure. The main control loop resides in the Notifier, not the application. The Notifier reads events and *notifies*, or *calls out* to, various procedures which the application has previously registered with the Notifier. These procedures are called *notify procs* or *callback procs*. This control structure is shown in Figure 2-14.

*XView events are in a form that you can easily use: an ASCII key has been pressed, a mouse button has been pressed or released, the mouse has moved, the mouse has entered or exited a window, etc. Events are described in detail in Chapter 6, *Handling Input*.

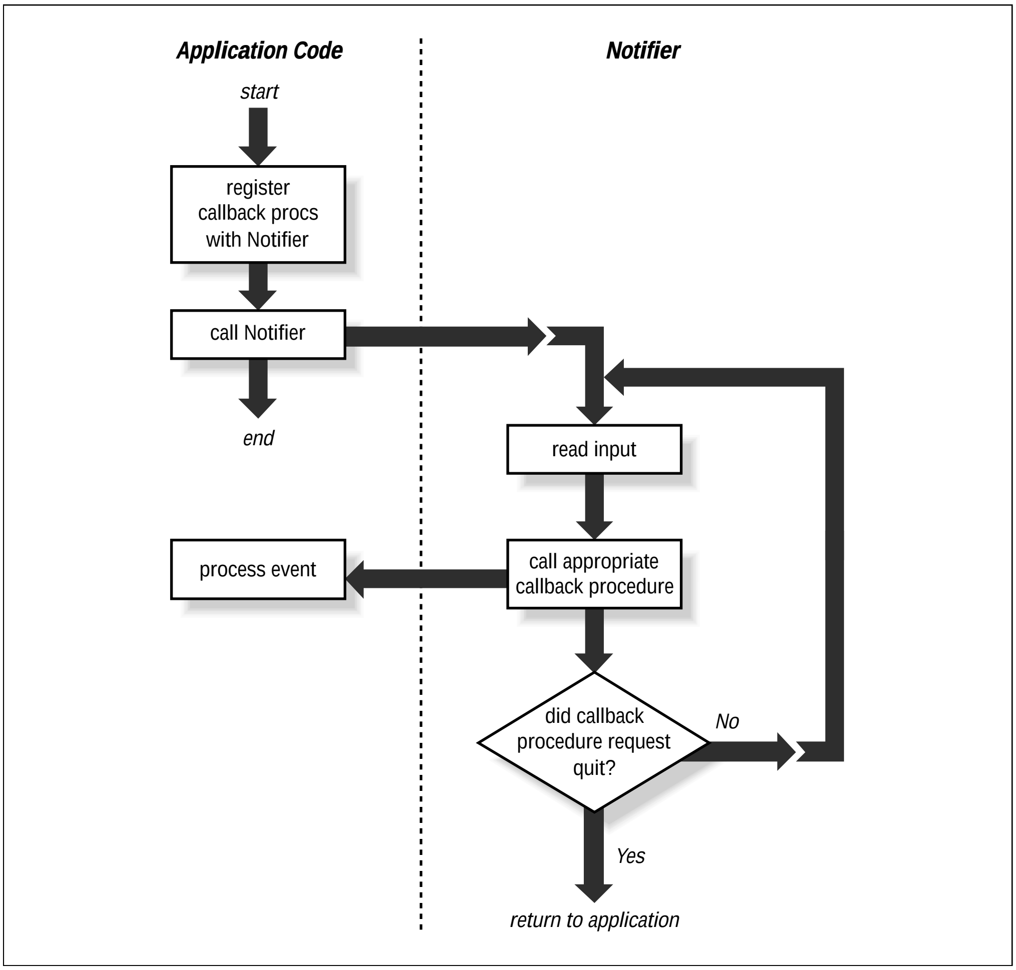

Application Code
Notifier
start
register callback procs with Notifier
call Notifier
end
read input
process event
call appropriate callback procedure
did callback procedure request quit?
No
Yes
return to application

Figure 2-14. Flow of control in a Notifier-based program

2.5.2 Why a Notification-based System?

If you are not used to it, this callback style of programming takes some getting used to. Its big advantage is that it takes over the burden of managing a complex, event-driven environment. In XView, an application typically has many objects. In the absence of a centralized Notifier, each application must be responsible for detecting and dispatching events to all the objects in the process. With a centralized Notifier, each component of an application receives only the events the user has directed towards it.

2.5.3 Relationship Among the Notifier, Objects, and the Application

It is not necessary for you to interact with the Notifier directly in your application. XView has a two-tiered scheme in which the packages that support the various objects—panels, canvases, scrollbars, etc.—interact with the Notifier directly, registering their own callback procedures. The application, in turn, registers its own callback procedures with the object.

Typically, when writing an XView application, you first create the various windows and other objects you need for your interface and register your callback procedures with the objects. Then you pass control to the Notifier. The work is done in the various callback procedures.

Let's illustrate the relationship of the Notifier. Figure 2-15 illustrates how the Notifier receives X events from the X server, as well as operating system "events" such as signals or input on file descriptors. Event procedures are supplied by XView packages as well as the application itself.

The main point of Figure 2-15 is to clarify the double-tiered callback scheme. How you register the callback procedures will be explained in Chapter 5, *Canvases and Openwin*, and Chapter 7, *Panels*.

One point worth mentioning is the distinction between the *event procedures* for the canvases and the *notify procedures* for the panel items. They are all callback procedures, but they have different purposes. The canvas's event procedure does not do much work—basically, it calls out to the application's event procedure each time an event is received. The application sees every event and is free to interpret the events however it likes.

The event procedure for panels, on the other hand, does quite a bit of processing. It determines which item should receive the event and places its own interpretation on events—the middle mouse button is ignored, and the left mouse button down over an item is interpreted as a tentative activation of the item, etc. It does not call back to the notify procedure for the item until it receives a left mouse button up over the item. So panel item notify procedures are not so much concerned with the event that caused them to be called as with the fact that the button was pushed, a new choice made, etc.

2.5.4 Calling the Notifier Directly

As mentioned previously, for many applications, you will not need to call or be called by the Notifier directly—the Notifier calls back to the subwindows, which in turn call back to your application.

However, if you need to use signals or be notified of the death of a child process which you have spawned, you do need to call the Notifier directly.

The Notifier also provides calls that allow you to insert your own routine in the event stream ahead of a window. This technique is known as *interposition*.

When and how to call the Notifier directly is covered in Chapter 20, *The Notifier*.

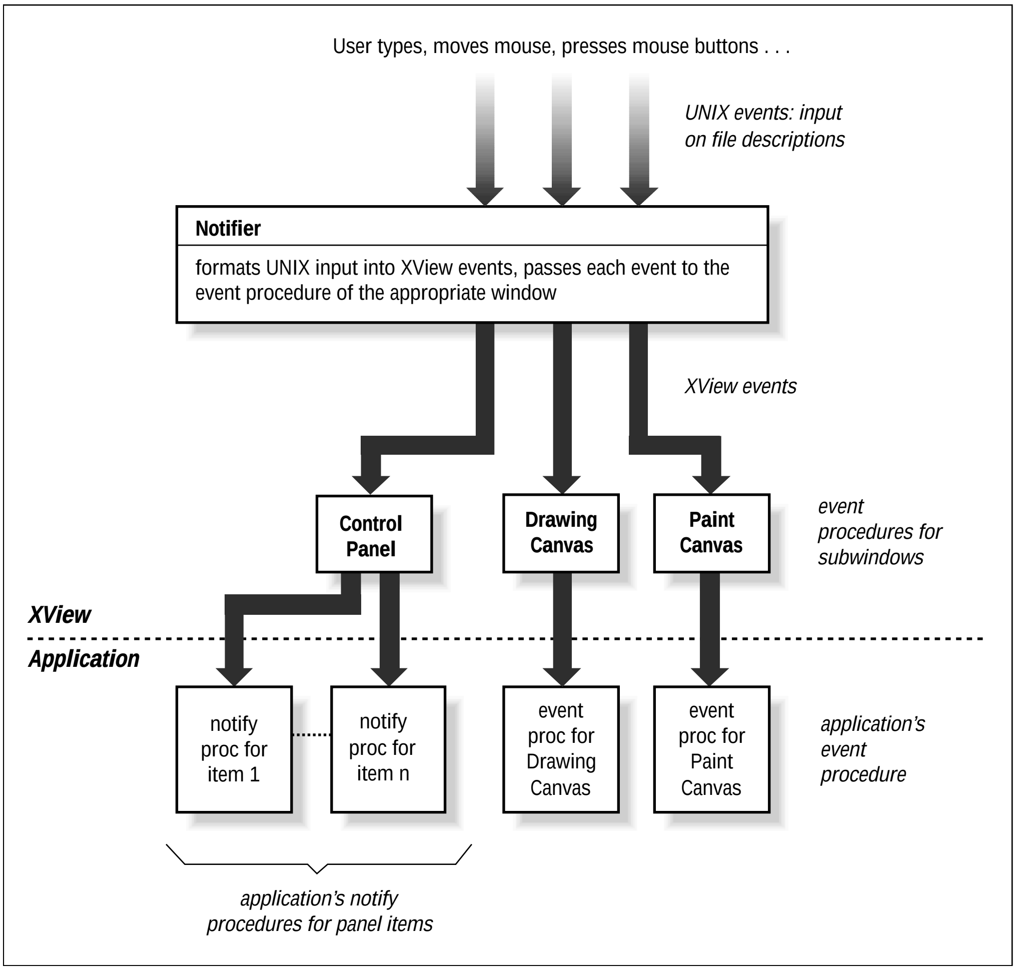

User types, moves mouse, presses mouse buttons . . .
UNIX events: input on file descriptions
Notifier
formats UNIX input into XView events, passes each event to the event procedure of the appropriate window
XView events
Control Panel
Drawing Canvas
Paint Canvas
event procedures for subwindows
XView
Application
notify proc for item 1
notify proc for item n
event proc for Drawing Canvas
event proc for Paint Canvas
application's event procedure
application's notify procedures for panel items

Figure 2-15. Flow of input events in an XView application

3
Creating XView Applications

This chapter covers the XView programming interface. It describes the basic XView distribution and how you use it to compile and link XView applications. It also presents the proper structure for XView applications. The structure can be summarized as:

- Initialize XView using `xv_init()`.
- Create a top-level window (`FRAME`) to manage subwindows.
- Add subwindows as children of the `FRAME`.
- Add objects to subwindows.
- Specify notification callbacks and select input events.
- Call `xv_main_loop()` to start the dispatching of events.

This chapter also discusses error recovery procedures.

3.1 Interface Overview

This section gives an overview of the XView programming interface. It covers reserved words and naming conventions in XView. It also includes a complete sample program, which we will look at more closely when describing the calling sequence for a program.

3.1.1 Compiling XView Programs

To compile an XView program, you must link with the XView library and the OPEN LOOK graphics library. These libraries comprise the entire XView Toolkit. XView is written for X11, of course, so you need to add the standard X library, which contains all the Xlib routines.

Thus, to compile a typical XView application whose source is *myprog.c*, you use the command:

```
% cc myprog.c –lxview –lolgx –lX11 –o myprog
```

3.1.2 XView Libraries

The XView library is made up of two other libraries: *libxvol.a* and *libxvin.a*. XView functions are found mostly in the library *libxvol.a*. These libraries include the code to create and manipulate high-level objects such as frames, panels, scrollbars and icons. These packages in turn call routines in *libxvin.a* to create and manipulate windows and interact with the Notifier. These libraries are both included in the library *libxview.a*. The XView libraries call routines in the Xlib library (*libX11.a*) that do the drawing on the screen.

The library specified by `-lolgx` is the OPEN LOOK graphics library. This library has routines that draw all the OPEN LOOK objects such as scrollbars and panel items. This library is not called from the client application; it is only called by the internals to XView.

Many of the images used by this library come from special fonts that must be installed on your X11 server. All servers newer than X11R4, as well as the X11/NeWS server, should have these fonts.

3.1.3 Header Files

The basic definitions needed by an XView application (windows, frames, menus, icons and cursors) are obtained by including the header file <*xview/xview.h*>. All XView applications should have the line:

```
#include <xview/xview.h>
```

This header file includes many other header files that set up standard types. It also declares external functions and includes some system-specific header files that are required by all the XView header files.* Once <*xview/xview.h*> has been included, other include files specific to the packages are included. Each object package has its own header file to declare object types, to provide definitions for frequently used macros and to make external definitions for routines that are specific to that object's package. Frequently these files include other files, which in turn may include other packages or system header files.

For instance, if your code uses the `FRAME`, `PANEL` and `FONT` packages, then these include files must be specified:

```
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/font.h>
```

However, investigation shows that some header files include other header files by default. For example, <*xview/xview.h*> includes <*xview/frame.h*>. There are "wrappers" inside the XView header files which prevent any one of them from being included more than once.

*XView also includes C++ bindings for all its public functions.

3.1.4 Naming Conventions

All the examples throughout this manual follow a consistent method in the naming of data types, package names, and even variable names. Because of the large number of packages and data types, you could easily confuse what a lexical string represents. Therefore, you are advised to follow certain criteria when naming variables and declaring data types that are not specific to XView. Whatever naming convention you choose, you should always try to be consistent.

3.1.4.1 Reserved names

XView reserves names beginning with the object types, as well as certain other prefixes, for its own use. The prefixes in Table 3-1 should not be used by applications in lowercase, uppercase, or mixed case.

Table 3-1. Reserved Prefixes

`attr_`	`icon_`	`server_image_`
`canvas_`	`menu_`	`string_`
`cms`	`notice_`	`termsw_`
`cursor_`	`notify_`	`text_`
`defaults_`	`panel_`	`textsw_`
`dnd_`	`pixrect_`	`tty_`
`dragdrop_`	`pr_`	`ttysw_`
`drop_`	`pw_`	`win_`
`ei_`	`r1_`	`window_`
`es_`	`rect_`	`wmgr_`
`ev_`	`screen_`	`xv_`
`event_`	`scroll_`	
`font_`	`scrollbar_`	
`frame_`	`selection_`	
`fullscreen_`	`seln_`	
`generic_`	`server_`	

To help you choose what *not* to use for data types and other lexical tokens in your application, review Table 2-1, "XView Objects, Owners, Packages, and Data Types."

3.1.5 Example of XView-style Programming

The flavor of the XView programming interface is illustrated by the code in Example 3-1. This program, *quit.c*, creates a frame containing a panel with one item: a button labeled Quit.

There are a few things to notice in the program. First, note the `NULL` that terminates the attribute lists in the `xv_create()` and `xv_set()` calls. The most common mistake in using attribute lists is to forget the final `NULL`. This will not be flagged by the compiler as an error.

The results are actually unpredictable, but the most common result is that XView will generate a run-time error message and the program will exit.

Second, the object returned by the `xv_create()` for the `PANEL_BUTTON` is not stored into a variable. This is primarily because it is not needed by any other portion of the code. One of the most common programming inefficiencies is the use of global variables when they are not needed. If you are not going to reference an object created via `xv_create()`, you should not retain its handle. If you need its handle, but only temporarily, then it should be a local variable, not a global or `static` one. For example, the `panel` variable (type `Panel`) is used as a local variable.

Example 3-1. The quit.c program

```
/*
 * quit.c -- simple program to display a panel button that says "Quit".
 * Selecting the panel button exits the program.
 */
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>

Frame frame;

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    void quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
        XV_HEIGHT,      100,
        NULL);

    panel = (Panel)xv_create (frame, PANEL, NULL);

    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);

    xv_main_loop (frame);
    exit(0);
}

void
quit()
{
    xv_destroy_safe(frame);
}
```

Figure 3-1 shows the output resulting from *quit.c*. In the sections that follow, we are going to look at how this program demonstrates the structure of XView programs.

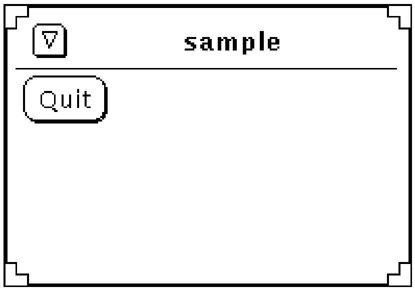


Figure 3-1. A frame containing a Quit button

3.2 Initializing XView

Initializing the XView system should be done as soon as possible in the application. The `xv_init()` function performs many tasks, including:

- Opening the connection to the server.
- Initializing the Notifier.
- Initializing the Resource Manager database.*

The form of `xv_init()` is:

```
Xv_Server
xv_init(attrs)
    <attribute-value list> attrs;
```

By default, `xv_init()` opens a connection to the server described by the `DISPLAY` environment variable. With the appropriate command-line options (discussed later), a different server may be specified. No matter which server is ultimately used, `xv_init()` returns a handle to that server object.

All subsequent XView objects that are created will use this server by default. This includes the physical screen(s) and resources. If you want your application to span multiple servers, you need to open a separate connection to those servers via the `SERVER` package. For further information on how to do this and other details of the `SERVER` package, see Chapter 15, *Non-visual Objects*, for details.

*See Chapter 17, *Resources*, for more information about the resource database.

3.2.1 Using xv_init()

Initialization should be done before the application attempts to parse its own command-line options. Since many programs tend to have command-line parameters, a program tends to report unknown parameters as illegal arguments. Because XView parameters can also be specified on the command line to the application, the program must be able to distinguish between the application's parameters and XView's parameters.

`xv_init()` accepts the attributes `XV_INIT_ARGS` and `XV_INIT_ARGC_PTR_ARGV` for purposes of parsing command-line arguments. These attributes both take two parameters as values: `argc` and `argv`. These are typically the same ones passed into `main()`. Using the `XV_INIT_ARGC_PTR_ARGV` attribute, the `xv_init()` function can be told to modify `argc` and `argv` by removing parameters that are XView-specific, like so:

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

When `xv_init` returns, `argv` contains only those parameters that are not specific to XView, and the application can now assume that all remaining arguments are specific to the application. So a hypothetical command line might look like this:

```
% program –display maui:0
```

The command line is first parsed by `xv_init()`, and in this case, all arguments are stripped from the `argv` variable, leaving just `argv[0]`, whose value is `program`. `argc` is modified to have the value 1 (it was originally 3). The example command-line parameters change the default server to be the X server running on the machine named *maui*.

The macro `XV_INIT_ARGS` is similar:

```
xv_init(XV_INIT_ARGS, argc, argv, NULL);
```

Here, `argc` and `argv` are not modified at all and are returned unchanged by `xv_init()`. Therefore, the *value*, not the address, of `argc` is used. This method is less advantageous for initializing XView because it leaves the application with the responsibility of parsing XView command-line parameters later.

NOTE

Once XView has been initialized, subsequent calls to `xv_init()` are ignored, as are all parameters consisting of `XV_INIT_ARGS` or `XV_INIT_ARGC_PTR_ARGV`.

A common error that users make is to enter bad command-line arguments. These arguments can be specific to XView or specific to the application, so XView handles the XView-specific argument and then expects the programmer to handle application-specific arguments.

Upon receiving a bad argument, XView prints an error message, indicating what XView-specific values are legal, and then calls `exit(1)`. The function that provides this message is specified by the attribute `XV_USAGE_PROC`. In most cases, you want to leave this alone because it is *not* the way you handle application-specific arguments.

The attribute XV_ERROR_PROC is used to install an error recovery routine. See Chapter 24, *Error Recovery*, for details about error handling.

3.3 Creating and Modifying Objects

After the system has been initialized, objects can be created and modified using xv_create(), xv_find(), xv_get(), and xv_set(). A closer look at xv_create() and xv_find() shows how these functions can be used to create new objects or find existing objects with particular attributes from various packages.

3.3.1 Using xv_create()

xv_create() is typically used as shown in Example 3-2.

Example 3-2. xv_create() creates XView objects

```
#include <xview/xview.h>

main(argc, argv)
char *argv[ ];
{
    Frame frame;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    frame = (Frame)xv_create(NULL, FRAME, NULL);
    xv_main_loop(frame);
}
```

In Example 3-2, a frame is created from the FRAME package and no additional attribute-value pairs are specified. Therefore, all the default properties from the frame class are set into the instance of this new frame when it is created.

The form of xv_create() is:

```
Xv_object
xv_create(owner, package, attrs)
    Xv_object        owner;
    Xv_pkg           package;
    <attribute-value list> attrs
```

In most cases, owner is another XView object. As shown in the example, when the frame is created, it has no owner, per se. This means that the owner should default to a pre-specified owner that XView has in mind. Defaulting is not always possible, but for this base frame, the default owner is the root window of the current server. As a child of the root window, the frame is under the constraints that the window manager might impose upon it (the colormap, for example).

Objects must have an owner for several reasons. One reason is that the X server may not be running on the same machine as the application (client) program. Therefore, because XView is running on the client side, objects that are created by the application have to contact the server. There could be more than one server to contact if the application supports multiple displays or runs on several machines simultaneously. To do this, XView needs to know what server or screen a particular object is associated with.

Another reason is that certain attributes are inherited from the owner, such as color and event masks. It is up to the individual package to determine what it inherits from its owner.

When `owner` is `NULL`, the owner of the object being created is either defaulted to a predetermined owner, or the object is said to have *delayed binding*. That is, the object is not associated with any other object until it is displayed on the screen. A scrollbar that is created with a `NULL` owner will not be displayed until it is attached to an object, and this object becomes its owner. Most objects are required to have owners at the time of their creation. Frames, windows and fonts must have a valid (default) owner because they need to access the screen's default colors, available fonts and so on.

Table 3-2 shows the default owner if `owner` is `NULL` in the call to `xv_create`.

Table 3-2. Default Ownership of Objects

Object's Package	Owner	If Owner is `NULL`
`CANVAS`	Frame	The window manager of `xv_default_screen`
`CMS`	Screen	`xv_default_screen`
`CURSOR`	Window, screen or anything that returns `XV_ROOT`	The window manager of `xv_default_screen`
`DRAGDROP`	Window	`NULL` owner not allowed
`DROP_SITE_ITEM`	Window	`NULL` owner not allowed
`FRAME`	Another frame or the root window	The window manager of `xv_default_screen`
`Panel Items`	Panel	`NULL` owner not allowed
`MENU`	Ignores its owner	Always use `NULL`
`NOTICE`	Window	`NULL` owner not allowed
`PANEL`	Frame	`xv_default_screen`
`Menu Items`	Menu	Allows delayed binding
`ICON`	Same as cursor	Same as cursor
`SCROLLBAR`	An openwin object	Allows delayed binding
`SCROLLABLE_PANEL`	Frame	`NULL` owner not allowed
`SCREEN`, `FULLSCREEN`	Server	`xv_default_server`
`SELECTION_ITEM`	Selection Owner	`NULL` owner not allowed
`SELECTION_OWNER`	Window	`NULL` owner not allowed
`SELECTION_REQUESTOR`	Window	`NULL` owner not allowed
`SERVER`	Ignores its owner	Always use `NULL`

Table 3-2. Default Ownership of Objects (continued)

Object's Package	Owner	If Owner is NULL
SERVER_IMAGE	Screen	xv_default_screen
TEXT	Frame	The window manager of xv_default_screen

The object that is returned from `xv_create()` is an opaque data type called `Xv_object`. The return value should be coerced into the type of the object being created.

Each time `xv_create()` is used, it creates a new and entirely different object. The type of object that is returned depends on the *package* specified.

```
Panel panel;
panel = (Panel)xv_create(frame, PANEL, NULL);
```

Here, a panel is created as a child of a frame. As in the previous example, there are no attribute-value pairs specified, so the panel is created with all the default values intrinsic to a generic panel object from the PANEL package. The panel is then installed inside the frame accordingly. Panel items can be installed inside the panel as:

```
Panel_item button;
button = (Panel_item)xv_create(panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,  "Quit",
    PANEL_NOTIFY_PROC,   quit,
    NULL);
```

Here, `xv_create()` is used to create a panel item of type PANEL_BUTTON. This is a special type of object that is created inside of panels only. That is why the owner of the item is the panel created in the previous example. In the attribute-value list provided in this example, the label is set by specifying the PANEL_LABEL_STRING attribute and a string as the value portion of the pair. Similarly, the callback routine specified is the routine called `quit()`. Because the intent of panel buttons is to select them with the pointer, the callback routine is the function to call if the user presses the mouse button in the panel button item.

3.3.2 Using xv_find()

In all the examples so far, the routine `xv_create()` has been used to create new objects of different types or classes. However, it might not always be possible to know whether or not a particular object has been created. The best way to handle such cases is to use `xv_find()`. If the object has been created, `xv_find()` returns the handle to the pre-existing object; if not, `xv_find()` creates it. The definition of the routine follows:

```
Xv_opaque
xv_find (owner, package, attrs)
    Xv_object           owner;
    Xv_pkg              package;
    <attribute-value list>    attrs;
```

As you can see, the form of `xv_find()` is the same as `xv_create()`. Fonts are objects that usually only need to be created once and are then used throughout the application wherever necessary. For example, say the application needs to use the font named *fixed* (because it is usually available on any X server and is almost guaranteed to be found). Several places in the application need to use the font, but only one instance of the font needs to be created. To avoid multiple instances of the object, the following function call is made:

```
Xv_Font my_font;
my_font = xv_find(frame, FONT,
    FONT_NAME, "fixed",
    NULL);
```

This code segment demonstrates how `xv_find()` tries to find an existing font named *fixed* that has already been created by the application. If the application has not yet created this font, then `xv_find()` acts just like `xv_create()`, and a new font is created. This function is not intended to replace `xv_create()` at all. It is intended to be used in the case where only one instance of an object is desired and that one instance is shared throughout the application. While you could use `xv_find()` rather than `xv_create()` in the other examples shown so far, the problem arises if you need two copies of a particular instance of an object. For example, if you were going to create another `PANEL` using all the default values of the `PANEL` package, then `xv_find()` would return the previously created panel. Any new objects attached to that panel would also be attached to the other panel because they are, in fact, one and the same.

As shown, fonts are frequent users of the `xv_find()` function.*

3.3.3 Using xv_destroy()

The correct way for an XView application to exit is to destroy all objects created and call `exit()` with an appropriate exit status. The function `xv_destroy()` destroys an instance of an XView object. The function `xv_destroy_safe()` does the same thing but ensures that it is *safe* to do so.† In general, it is better to be safe than sorry. The definition of these routines are as follows:

```
int
xv_destroy_safe(object)
    Xv_opaque object;

int
xv_destroy(object)
    Xv_opaque object;
```

The return value from the routines is either `XV_OK` or `XV_ERROR`. Example 3-2 in Section 3.1.5, "Example of XView-style Programming," shows a base frame containing a panel subwindow with one panel button created inside it. The callback routine for the panel item, `quit()`, is intended to exit the application. Rather than actually calling `exit()`, a more elegant way to exit would be to destroy all the objects that have been created. If a text

*Chapter 16, *Fonts*, describes how to create fonts using `xv_find()`.
†Chapter 20, *The Notifier*, discusses the difference between a safe and an *immediate* destruction of an object.

subwindow had its text modified since the last update, this would give the package an opportunity to prompt the user for an update. Or you might install a routine that interposes any request for destruction on a particular object (such as a frame).*

The `quit()` routine looks like this:

```
void
quit()
{
    if (xv_destroy_safe(frame) == XV_OK)
        exit(0);
}
```

Rather than calling `xv_destroy_safe()` on all objects, it is only called for the base frame. Because the base frame is the owner of all the other objects, `xv_destroy()` and `xv_destroy_safe()` descend into the objects' children and destroys all of them with the same call.

Use the function `xv_destroy_safe()` to destroy objects from within the object's callback procedures. `xv_destroy_safe()` will delay destroying the object until it is safe to do so (that is, not while in the object's callback). For example, when a frame is destroyed from within the frame's `FRAME_DONE_PROC`, you need to use `xv_destroy_safe()` to ensure that the frame object is removed. (See `FRAME_DONE_PROC` in the next chapter for a description of a frame's "done" callback.)

`xv_destroy()` may be called at any time without notice. It may result from actions the user takes with the window manager, from a separate process or from events sent by other applications.

3.3.4 Using xv_set() and xv_get()

As discussed in the previous chapter, attributes about objects can be set, reset and retrieved using the calls `xv_set()` and `xv_get()`. The definition of these routines are:

```
Xv_opaque
xv_set(object, attrs)
    Xv_object           object;
    <attribute-value list>   attrs;

Xv_opaque
xv_get(object, attr)
    Xv_object           object;
    Attr_attribute       attr;
```

`xv_set()` is just like `xv_create()` with respect to the attribute-value parameters. Use `xv_set()` to set or change the value of one or more attributes of an object that has already

*Chapter 20, *The Notifier*, describes how to install destroy interpose functions.

been created. The following code segment uses a single `xv_set()` call to change three attributes of a frame:

```
#include <xview/xview.h>

main()
{
    Frame frame;
    frame = xv_create(NULL, FRAME, NULL);
    ...
    xv_set(frame,
        FRAME_LABEL,      "XView Demo",
        FRAME_SHOW_LABEL, TRUE,
        FRAME_NO_CONFIRM, TRUE,
        NULL);
    ...
    xv_main_loop(frame);
}
```

`xv_get()` is different from `xv_set()` in that the `value` parameter is not passed to the function—instead, the value is returned from `xv_get()`:

```
Xv_Window root_win;
Frame frame;
Rect *rect;

/* create the base frame for the application */
frame = xv_create(NULL, FRAME, NULL);

/* get the root window of the base frame of the application */
root_win = (Xv_Window *) xv_get(frame, XV_ROOT);

/* get the dimensions (rectangle) of the root window */
rect = (Rect *) xv_get(root_win, XV_RECT);
```

Because `xv_get()` returns the value of the attribute specified, only one attribute of the object can be retrieved by an `xv_get()` call. The return value for the function is going to be an opaque data type, so it must be *typecast* into the type expected. However, note that the value XV_ERROR might be returned in the event that the object passed is not a valid object or if an attribute does not apply. In this case, the return value should be checked to see if it is XV_ERROR. One potential problem is that the value of XV_ERROR might happen to be the same as the expected return value. Fortunately, an error returned from `xv_get()` is unlikely in a properly written application.

In many packages, certain properties may be retrieved but not set. For example, you may use `xv_get()` for the property WIN_FRAME to get the window's frame but you may not use `xv_set()` to set the window's frame. In this case, `xv_set` returns XV_ERROR. In the more likely event that the call was successful in setting attributes using `xv_set()`, then the value XV_OK will be returned.

For some XView attributes that take strings as values in `xv_create()` or `xv_set()`, the string is copied, but for other attributes, the passed pointer is used directly. Since XView's internal memory allocation methods may change in future releases, you should not write code that depends in any way on it. The following example shows the type of code that should be avoided, since it depends on XView's internal memory allocation method:

```
/*
* Set panel label string on panel button
*/
xv_set(panel_button, PANEL_LABEL_STRING, array_ptr, NULL);


/*
* DO NOT attempt the following
*
* Check if the memory pointer for the panel button label
* is the same as array_ptr */

if (array_ptr == xv_get(panel_button, PANEL_LABEL_STRING))  {
        ...
}
```

3.3.5 Precedence of Resource Options

In the X Window System, the user can configure the interface according to options available in specific applications. The user accomplishes this through a *resource database* that resides in the X server. XView provides several ways for the programmer to set default values and to accommodate the user's specifications for properties such as frame colors, fonts and window geometry among others (for more information on resources, see Chapter 17, *Resources*). There are several ways that properties can be set, including: using `xv_set()` on an attribute corresponding to the property, using command-line options, using values from the `.Xdefaults` file or using the values specified when calling `xv_create()`. In addition, XView or the window manager may determine some default values for certain properties. Among these different ways of setting options, programatically, the following precedence from highest to lowest is maintained:

1. A call to `xv_set()`.
2. Any command-line options.
3. Values specified in the `.Xdefaults` file.
4. Values specified or inherited in a call to `xv_create()`.
5. Toolkit or window manager defaults.

NOTE

The precedence above does *not* apply for locale commmand line options. See Chapter 22, *Internationalization*, for more details on locale command-line options.

3.4 xv_main_loop() and the Notifier

Once all the objects have been created, you are ready to have all the windows displayed and have event processing begin. At this point, the program calls `xv_main_loop()`. The job of `xv_main_loop()` is to start the Notifier. Once the Notifier has started, the program will begin to receive and process events such as `Expose`, `MapNotify`, `ConfigureNotify`, `KeyPress`, and so on. The X server generates these events and sends them to the client. While it is up to the client to handle all events that the X server sends to it, the Notifier layer of XView handles much of this work automatically.

The Notifier's main job is to process these events and dispatch them to the client if it has registered a callback routine for that event type with the Notifier. Otherwise, the Notifier might ignore the event. Of course, XView attempts to provide reasonable default actions for all events that the application typically does not want to deal with. For example, a simple application that contains nothing but a command frame (which has nothing but a panel/control area) might not care to handle resize events if the user resizes the window. XView must handle this so it can resize the panel and/or reposition the panel items within it.

Those events that the application would be most interested in are things like `KeyPress` and `ButtonPress` events of various types. For events like these, the application should install callback routines for the Notifier to call if one of those events has taken place.

In the examples shown, the only callback routine installed is the one in the panel item, `quit()`. When selected, the Notifier notifies the application by calling the callback routine associated with the object in which the event took place. In this case, the Notifier calls the routine `quit()` and the application has control of the program again. As one might expect, the Notifier has relinquished control of the program while the application's callback routine is being called. The Notifier does no more event processing at all until the callback routine has returned. However, the programmer can query events from within the callback routine if necessary.

If the code within the callback routine creates new objects or destroys existing objects, nothing will happen on the display until the callback routine is finished and returns control to the Notifier.

Just because the Notifier handles the delivery of events to the application, that does not mean that the application will be notified of all events that might occur. The application is only notified of the events that it has registered with the Notifier. Events that the client can register with the Notifier include `CreateWindow`, `MapWindow`, `ConfigureWindow`, `QueryFont`, `GetInputFocus`, and so on. These are general X events, not XView-specific events. However, XView has a corresponding event definition for the purpose of registering events with the Notifier. Event registration is covered in detail in Chapter 20, *The Notifier*. Event types and specifications are discussed in Chapter 5, *Canvases and Openwin*, and Chapter 6, *Handling Input*.

When a frame is displayed on the screen, a `MapNotify` event is generated by the X server (since the frame is *mapped*, or displayed, to the screen). However, there has been no callback routine specified to handle the map event, so the Notifier passes it back to XView, which handles it internally. This default action, in fact, does nothing special; it simply allows the frame to be displayed. Further events are generated: expose events, visibility events (for the frames that are covered up by the new frame), enter and leave events when the user moves the mouse in and out of the frame, motion events, and so on. If none of these events have an application-defined callback routine associated with them, the Notifier handles them.

Note that when objects such as frames or canvases are created, only the objects themselves and the associated attributes of those objects are created. What is *not* created are the objects' windows. These are not created until after `xv_main_loop()` is called. This is due to the fact that one of the events that is generated is the *realize* event—this indicates that an object has been realized to the screen and a window has been (or needs to be) generated. The objects' packages internally handle the creation of windows at the appropriate time. Since that does not occur until after the call to `xv_main_loop()`, there should be no attempts to render graphics into objects' windows before then.

4 Frames

A frame is a container for other windows. It manages the geometry and placement of *subwindows* that do not overlap and are fixed within the boundary of the frame. The OPEN LOOK specification refers to subwindows, or *panes*, as *tiled* windows because they do not overlap one another. Subwindow types include canvases, text subwindows, panels, and scrollbars. These subwindows cannot exist without a parent frame to manage them. Figure 4-1 shows the class hierarchy for the `FRAME` package.

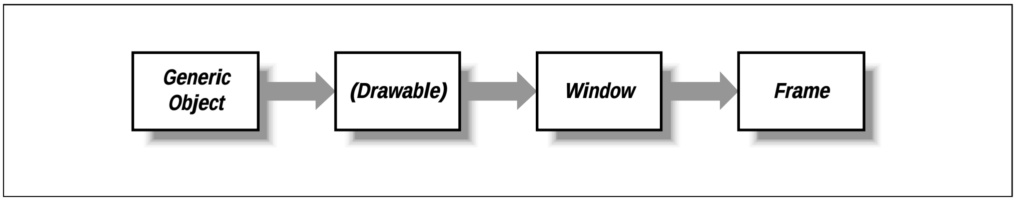
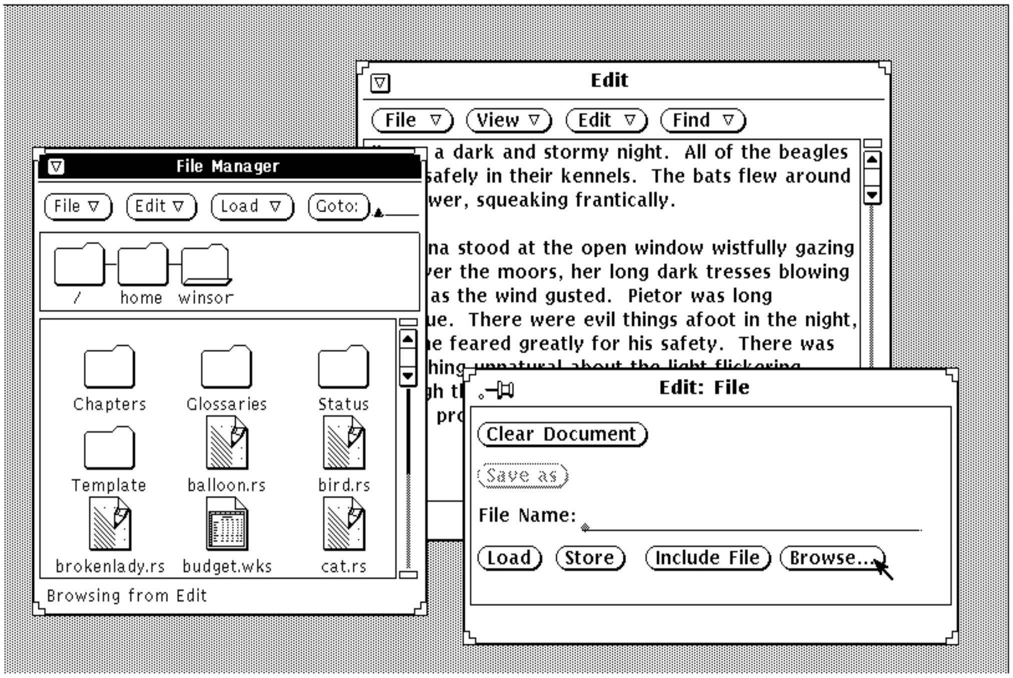


Figure 4-1. Frame package class hierarchy

Figure 4-2 shows an example of a screen that displays three frames, each one containing at least one subwindow. Note that frames do overlap. The File Manager frame has the key-board focus, as indicated by the title bar having its foreground and background colors reversed. The setting of the keyboard focus is handled by the window manager, not the `FRAME` package. In this case, an OPEN LOOK window manager is using click-to-type to set the keyboard focus. This is demonstrated by the cursor's location within an unselected frame (the *Edit: File* frame).

The `FRAME` package provides the following capabilities:

- A communication path between the application and the window manager.
- A mechanism to receive input for the application.
- A visual container for user interface objects.
- A method to group windows with related functionality.
- A mechanism to manage footers.



a dark and stormy night. All of the beagles
afely in their kennels. The bats flew around
wer, squeaking frantically.

na stood at the open window wistfully gazing
er the moors, her long dark tresses blowing
as the wind gusted. Pietor was long
ue. There were evil things afoot in the night,
e feared greatly for his safety. There was
hing

Figure 4-2. Three base frames

A frame depends upon the window manager for its *decorations* and many basic operations. The FRAME package does *not* manage headers (title bars), resize corners or the colors of those objects. These are all strictly functions of the window manager. The application gives hints to the window manager about some of these attributes through the FRAME package (including not to display decorations at all if so desired), but results vary depending on which window manager the user is running. The examples in this book assume the user is running an OPEN LOOK window manager.

Frames do not manage events; this task is left up to the windows that the frame manages. That is, frames do not get mouse and keyboard events and propagate them to child windows. While frames are subclassed from the window package, the frame's window rarely sees any events at all, and if they do, these are not intended to be processed by the application programmer.

4.1 Types of Frames

Basically, two types of frames are available in XView: base frames and command frames. The main frame of the application is called the *base frame*. The base frame resides on the root window; its handle is passed to `xv_main_loop()` to begin application processing.

A special kind of frame, called a *command frame*, is created with a panel subwindow by default. Command frames are useful as help frames, property frames and such defined by OPEN LOOK. Programmatically, a command frame is no different from a frame with one subwindow that is a panel.

A base frame's *parent* is the root window, whereas a subframe's parent is another frame (either a base frame or a subframe). When a frame goes away (quit or close), all of its child windows, including subframes, also go away. For example, assume you create a command subframe to display application-specific help. When this command subframe is activated, it might display explanatory text along with an OK button to dismiss the help. If you close the base frame, the help subframe also closes.

XView allows for multiple frames that are not children of the base frame. For instance, you could create a help frame that is independent of the application's base frame. The parent of this frame is the root window of the display and not the base frame. The help frame will remain visible even if the base frame goes away. The term subframe defines a relationship among frames at creation time and a slight difference in functionality.

4.1.1 The Role of the Window Manager

It is important to understand what effect the window manager has in determining the appearance and behavior of an XView frame. As mentioned earlier, many attributes defined in the `FRAME` package are really hints to the window manager. The window manager is responsible for frame and window decoration, as well as the size and placement of windows on the screen (screen geometry). That is, it is the window manager's job to provide such decorations as title bars and to set attributes such as the color of decorations. It also handles resizing windows, moving windows, closing windows (iconifying) and so on. The application can ask the window manager to do things in a certain way, but the window manager is not obligated to act on these requests.

For your application and the window manager to communicate properly, the window manager must comply with the specifications in the *Inter-Client Communication Conventions Manual* (ICCCM).* Since the window manager is a client of the X server just as the application is a client, the two clients must follow specified conventions to communicate with one another. The `FRAME` package assumes it is communicating with an OPEN LOOK-compliant window manager. If not, some of the frame attributes might not work as described here.

*The *Inter-Client Communication Conventions Manual* is reprinted as Appendix L of Volume Zero, *X Protocol Reference Manual.*

For example, using an OPEN LOOK window manager, a command subframe is a pop-up window that has a pushpin in the upper-left corner. The state of the pushpin (unpinned or pinned) determines whether or not the window remains on the screen after the command has been executed. The pin objects are provided by the OPEN LOOK window manager. If XView applications are run with another window manager, they might not necessarily be pinnable.

4.2 Base Frames

Let's first create a base frame using the default attribute settings. To create a frame, use `xv_create`, specifying the owner of the frame and identifying the `FRAME` package. The program in Example 4-1 shows how to create a simple base frame.

Example 4-1. The simple_frame.c program

```
#include <xview/xview.h>

main()
{
    Frame frame;
    frame = (Frame)xv_create(NULL, FRAME, NULL);
    xv_main_loop(frame);
}
```

Specifying `NULL` as the owner argument tells the `FRAME` package to use the default value, which specifies the root window on the current screen. The macro `FRAME` is defined to be `FRAME_BASE`, which is the base `FRAME` package in XView.

Note that the header file associated with the `FRAME` package, <*xview/frame.h*>, is included indirectly by <*xview/xview.h*>. (A separate inclusion has no harmful effect, however.)

The frame displayed by this program is shown in Figure 4-3.

4.2.1 XView Initialization and Base Frames

The first XView object to be created by an application is typically the base frame. However, `xv_init()` is always called first to get any command-line parameters, initialize the connection to the X server, set resources, and so on. The code segment in Example 4-2 shows how `argc` and `argv` can be used in conjunction with `xv_init()`.

Example 4-2. Creating a base frame after calling xv_init()

```
main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame;
    ...
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    frame = (Frame)xv_create(NULL, FRAME, NULL);
```

Example 4-2. Creating a base frame after calling xv_init() (continued)

```
    ...
    xv_main_loop(frame);
    exit(0);
}
```

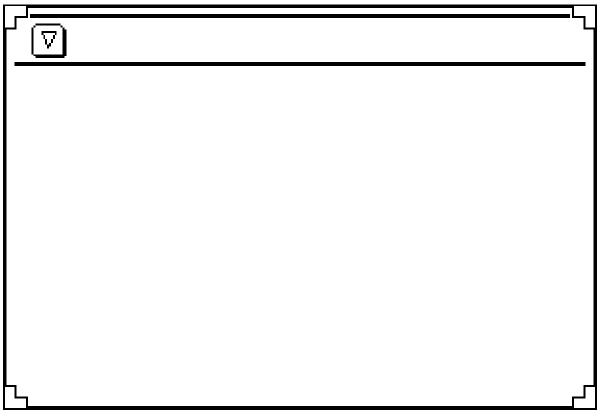


Figure 4-3. Simple base frame created without any FRAME attributes specified

In this sample code segment, the program expects no parameters that are specific to the program. Any parameters that the user supplies are expected to be specific to XView or X. When `xv_init()` is called with attribute `XV_INIT_ARGC_PTR_ARGV`, `&argc` and `argv` are passed as values; upon return, `argv` is stripped of the parameters that are X or XView-specific. The `argc` variable is also modified to reflect the number of parameters that remain after processing. If the user specified no other parameters, then `argc` should be 1 and `argv` should contain one element in its array of strings: the name of the program (since that is what `argv[0]` is). If there are any more parameters, then the application has the opportunity to look for application-specific parameters or to report unknown parameters as errors.

When the attribute `XV_INIT_ARGS` is used, the value of `argc` (not the address of `argc`) is passed, and neither variable is modified.

4.2.2 Headers and Footers

The OPEN LOOK window manager provides frames with headers that display text. The header typically displays information such as the application name. The text is centered and its font is not alterable by the application. The header also has a Close button at the upper-left corner; selecting the button causes the frame to iconify—that is, the frame turns into a graphical image, or *icon*, and is displayed (probably) elsewhere on the screen. The application's *state* is now closed. This usually indicates that the program is idle.

The footer of a base frame shows text such as error messages, a page number, the date or other miscellaneous information. The footer is split into two parts: the left footer where text is left justified, and the right footer where the text is right justified.

Unless otherwise specified by giving the appropriate attribute-value pairs, the header of a base frame is displayed, but the footer is not. The header does contain the abbreviated menu button, but there is no default header label. Thus, to create a header with a label, the attribute `FRAME_LABEL` must have a string value.* This may be a constant string or a variable pointing to a string. When the frame is displayed, the string will be centered in the header. The header label can be turned off by setting the `FRAME_SHOW_HEADER` attribute to `FALSE`. In this case, even if the header label is set, the header (including the Close button) will not be displayed at all. If `FRAME_SHOW_HEADER` is later set to `TRUE`, then the label will be displayed again (see Figure 4-4). Note, some window managers, (including **olwm**) will only honor requests to change certain aspects of the decor window when the window is coming out of withdrawn state (unmapped). Any requests made to a mapped window to change information about the decor will be held off until the window is withdrawn and mapped again. Thus the result of setting the header to `FALSE` will not be seen until the frame is unmapped and then mapped again.

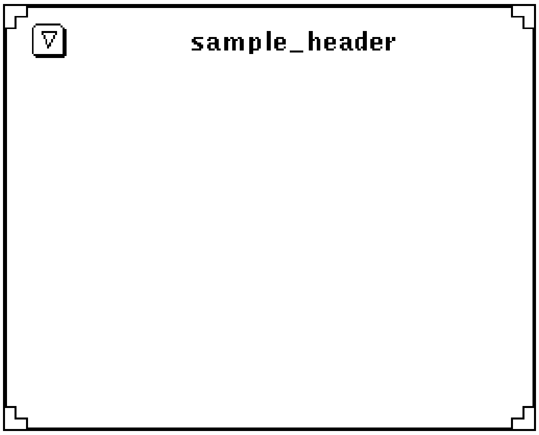


Figure 4-4. A sample header label display in a frame

The code segment in Example 4-3 sets `FRAME_SHOW_HEADER` to `FALSE` at creation but sets it to `TRUE` in a separate call. The header displays the name of the program.

*`FRAME_LABEL` is defined to be `XV_LABEL` in <*xview/frame.h*>.

Example 4-3. Setting separate values for a frame header

```
    ...
    Frame frame;
    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL, argv[0],
        FRAME_SHOW_HEADER, FALSE, NULL);
    ...
    xv_set(frame, FRAME_SHOW_HEADER, TRUE, NULL);
    ...
```

Footers in base frames are not displayed by default, so setting either the left or right footer messages also requires the boolean `FRAME_SHOW_FOOTER` to be set to `TRUE`. Note that setting the footer on and off resizes the total size of the base frame, and while it does not cause any subwindows to be resized, it is rather distracting to change the frame frequently (or at all). Therefore, you should decide ahead of time whether you are going to use footers and set them to be on or off at the time the frame is created. If the footer is no longer needed, set the left or the right footer string to the null string—*not* the constant `NULL`. That is, use `""`. Figure 4-5 shows what is displayed when the code in Example 4-4 is run.

Example 4-4. Creating a footer

```
    ...
    Frame frame;
    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,        "hdrs_n_footers",
        FRAME_SHOW_FOOTER,  TRUE,
        FRAME_LEFT_FOOTER,  "left side",
        FRAME_RIGHT_FOOTER, "right side",
        NULL);
    ...
```

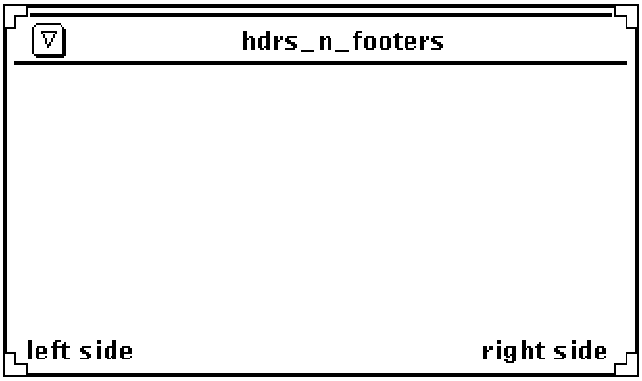


Figure 4-5. Headers and footers on base frame

4.2.3 Closed Base Frames

Base frames are distinct from other types of frames because they can be closed, or *iconified*. When the frame is closed, an icon replaces the entire base frame, including all subwindows and control areas.* If any subframes are associated with the base frame, then they are taken down for as long as the application is closed. Using the appropriate attribute-value pairs, it is possible to set the image and size of the icon. By default, no icon is associated with a base frame and the size of the area occupied by an icon is 64x64. See Chapter 14, *Icons*, for more information on creating icons used by frames. The bounding box (or `Rect`) of the icon is independent of the size of the icon, so the bounding box should be set explicitly if its value is anything other than the default.

The frame's dimensions when closed and the icon it uses may be set using `xv_create()` or by using `xv_set()` after the frame has been created. Figure 4-6 shows what any application or base frame looks like when it is closed, providing that the application uses the base frame with default values. The figure also shows a graphical icon used by an application.

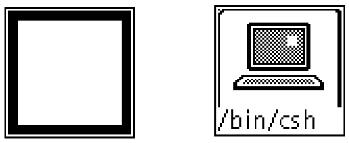


Figure 4-6. Default icon and application icon

To set the icon for the base frame, the icon must already have been created. However, assuming one is available, the icon can be set in the base frame using the `FRAME_ICON` attribute-value pair. `FRAME_ICON` will not have an immediate effect on mapped icon's. One must first unmap the icon, change the icon, and then remap. Because the icon might not be the default size (64x64), it is usually a good idea to set the size of the frame when it is in the closed state. To do this, set the attribute `FRAME_CLOSED_RECT` to be a pointer to a variable of type `Rect *` (pointer to a `Rect`). (`Rect` is an XView data type defined in <*xview/rect.h*>.)

The best way to handle it is to use `xv_set()` after the call to `xv_create()`:

```
 . . .
extern Icon icon;
Frame       frame;
Rect        rect;
 . . .
rect.r_width = (int)xv_get(icon, XV_WIDTH);
rect.r_height = (int)xv_get(icon, XV_HEIGHT);
frame = (Frame)xv_create(NULL, FRAME, NULL);
xv_set(frame,
    FRAME_ICON,         icon,
    FRAME_CLOSED_RECT,  &rect,
    NULL);
```

*Only base frames have icons associated with them.

This code segment sets the size of the icon area to be whatever size the icon is. Since FRAME_CLOSED_RECT takes a pointer to a variable of type Rect, the address of the variable is given as the value parameter to the call to xv_set(). Also note that the other fields of the rect variable, (r_top, r_left) are not used because FRAME_CLOSED_RECT only uses the width and height dimensions from the variable.

The following call can be made to determine whether the frame is currently closed from within the application:

```
is_closed = (Boolean) xv_get(frame, FRAME_CLOSED);
```

This call is useful for applications that are graphics-intensive. If a complex piece of code is about to be executed, the application could check to see if the frame is open to display the graphics.

Another way to change the icon, without worrying about whether the frame is mapped, is to change the server_image associated with the icon.

```
extern Icon icon;
Frame       frame;

icon = xv_get(frame, FRAME_ICON);
xv_set(icon, ICON_IMAGE, new_image, ICON_MASK_IMAGE, new_mask, NULL);
```

This method of changing the icon works regardless of whether the frame is in its iconic state, assuming the Server_image new_image is defined.

4.2.4 Quit Confirmation

OPEN LOOK specifies that a notice is generally not needed to confirm a “quit” action unless data will be lost. The base frame, which usually handles this type of action through the window manager, can be set to ask for confirmation. The attribute FRAME_NO_CONFIRM, which defaults to TRUE, can be set to FALSE to force confirmation:

```
frame = (Frame)xv_create(NULL, FRAME,
    FRAME_NO_CONFIRM, FALSE,
    NULL);
```

When the attribute FRAME_NO_CONFIRM is set to FALSE and the user initiates a “quit” action, a notice dialog box will appear to request confirmation.

4.3 Command Frames

Command frames are normally subframes in that they are most often created as children of base frames. Most of the time, they are pop-up frames that serve one function and then go away. Instead of having a Close button in the frame's header, the command frame has a pushpin. The pushpin governs whether the frame remains up after the user performs the functions that the pop-up frame provides. When a command frame is created, a default panel is also created automatically. The panel on a command frame can be used to hold the panel items, such as buttons or sliders, that the user interacts with. `xv_get()` can be used on command frames to get the default panel and to avoid creating a new one. A pop-up frame can have its own child pop-up frame, but this is not a good programming practice.

Example 4-5 shows a simple program that creates a pop-up frame as a child of the base frame. Figure 4-7 displays the output of this program.

Example 4-5. Creating a subframe

```
/*
 * subframe.c -- display a subframe from a base frame.
 */
#include <xview/xview.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame, subframe;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME,
        XV_WIDTH,       100,
        XV_HEIGHT,      100,
        FRAME_LABEL,    "Base Frame",
        NULL);

    subframe = (Frame)xv_create(frame, FRAME_CMD,
        XV_WIDTH,       100,
        XV_HEIGHT,      100,
        FRAME_LABEL,    "Popup",
        NULL);

    xv_set(subframe, XV_SHOW, TRUE, NULL);

    xv_main_loop(frame);
}
```

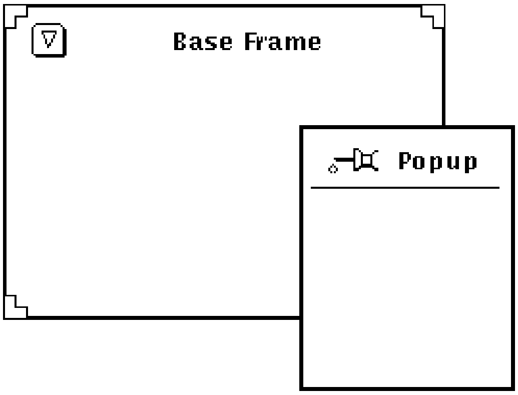


Figure 4-7. Base frame with a command frame

4.3.1 Manually Displaying Frames

The attribute XV_SHOW sets whether a window (or frame, in this case) will be displayed. If it is set to FALSE, the frame will not be seen. Base frames are always displayed because xv_main_loop sets the XV_SHOW attribute to TRUE. Pop-up frames are not displayed unless the XV_SHOW attribute is set by the application. An application might create many pop-up dialog boxes initially and then determine the appropriate time to actually display them. A callback routine invoked by some action (e.g., panel button selection) from the base frame might be used to display the pop-up frame. When a command frame is displayed, the new cursor is moved to the default panel item within the command frame.

When a frame is displayed, the server allocates space for it. If you create an application that uses many objects and requires a large amount of server memory to display those objects, then, on servers with limited memory, you may need to specially manage or limit the number of frames that are displayed.

4.3.2 The Pushpin

On a command frame, the pushpin at the upper-left corner is unpinned by default. The pushpin is controlled by using both the FRAME_CMD_DEFAULT_PIN_STATE and the FRAME_CMD_PIN_STATE attributes. FRAME_CMD_DEFAULT_PIN_STATE controls the initial state of the command frame's pin when the frame goes from unmapped (withdrawn) to mapped state. It is valid for both mapped and unmapped frames. However, if the frame is currently mapped, the change will be visible only on the next transition from unmapped to mapped state. Valid values for the state are defined in *<xview/frame.h>* and include: FRAME_CMD_PIN_IN and FRAME_CMD_PIN_OUT.

FRAME_CMD_PIN_STATE returns the current state of the pin. It is valid for both mapped and unmapped frames. For unmapped frames this always returns FRAME_CMD_PIN_OUT.

A simple example can demonstrate how some of the frame attributes interact. The program in Example 4-6 builds a base frame with a panel button that says, “Hello.” If selected, a command frame pops up. The new frame has a panel button that says, “Push Me,” and if pushed, “Hello World” is printed to *stdout*. If the command frame’s pushpin is out, once the Push Me button is selected, the frame is taken down. To get the frame up again, the user must select Hello again. However, if the pin is in, then the frame remains up for repeated use. Selecting Hello while the frame is already up causes the command frame to be raised to the top of the window tree. This is useful in case the command frame gets obscured by other windows.

Example 4-6. Using several frame attributes

```
/*
 * popup.c -- popup a frame and allow the user to interact with
 * the new popup frame.
 */
#include <xview/xview.h>
#include <xview/panel.h>

Frame frame;     /* top level application base-frame */
Frame subframe;  /* subframe (FRAME_CMD) is a child of frame */

main(argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    int show_cmd_frame(), pushed();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    /* Create base frame */
    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL, argv[0],
        NULL);

    /* Install a panel and a panel button */
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING, "Hello",
        PANEL_NOTIFY_PROC,  show_cmd_frame,
        NULL);

    /* Create the command frame -- not displayed until XV_SHOW is set */
    subframe = (Frame)xv_create(frame, FRAME_CMD,
        FRAME_LABEL,         "Popup",
        NULL);

    /* Command frames have panels already created by default -- get it */
    panel = (Panel)xv_get(subframe, FRAME_CMD_PANEL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,  "Push Me",
        PANEL_NOTIFY_PROC,   pushed,
        NULL);

    xv_main_loop(frame);
}
```

Example 4-6. Using several frame attributes (continued)

```
/* Called when base frame’s button is pushed -- show/raise subframe */
show_cmd_frame(item, event)
Frame item;
Event *event;
{
    xv_set(subframe, XV_SHOW, TRUE, NULL);
}

/* Called when command frame’s button is pushed */
pushed(item,event)
Panel_item  item;
Event  *event;
{
    printf("Hello world.\n");

    /* Check to see if the pushpin is in -- if not, close frame */
    /* Return value of FRAME_CMD_PIN_STATE is cast to int  */
    if ((int)xv_get(subframe,
                FRAME_CMD_PIN_STATE) == FRAME_CMD_PIN_IN)
        xv_set(subframe, XV_SHOW, FALSE, NULL);
}
```

Several things are noteworthy in this sample program. First, there is only one `Panel` variable (called `panel`). It is used to store the handle to the panel created by the base frame. It does not matter that this variable is also used to store the return value of the call to:

```
xv_create(frame, PANEL, NULL)
```

because its use is temporary. This demonstrates that the programmer need not maintain handles to objects that are never referenced. A common programming efficiency error is to declare many variables that reference objects created via `xv_create()` and then never to use them. The panel is needed, but only long enough to use it as the *owner* of the panel button that is created. Once ownership is established, the handle to that panel is no longer needed. Therefore, the variable is reused in the call that gets the *already created panel* from the subframe. This call is as follows:

```
panel = (Panel)xv_get(subframe, FRAME_CMD_PANEL)
```

The handles to the buttons are never needed, so the return values of those creation calls are ignored. On the other hand, it is always prudent to check for return values in case of error—had the panel creation returned a `NULL` handle, then the buttons should not be created. The sample programs do not demonstrate this type of error checking in order to keep the examples simple and readable.

4.3.3 The FRAME_DONE_PROC Procedure

If the pushpin is pushed in by the user, the application has no knowledge of this action. However, if the user pulls the pin out, then the toolkit calls the command frame’s `FRAME_DONE_PROC` routine. By default, if the parent of command frame is `NULL`, then the frame is unmapped. The programmer can override this behavior by installing another `FRAME_DONE_PROC` routine. This is only needed if you want to check that the frame can be dismissed. Suppose that the purpose of the subframe is to query for a filename. If the filename was not given, you might want to display a notice indicating that and allow the user to type in a filename. In this case, the FRAME_DONE_PROC is responsible for doing the necessary checking and deciding whether to display a notice or to take the frame down.

When you install your own FRAME_DONE_PROC routine, XView will *not* take down the frame regardless of what your code does. If you want the frame to be unmapped, by set XV_SHOW to FALSE.

The parameter passed to your routine is a handle to the subframe itself, as shown in Example 4-7.

Example 4-7. The subframe.c program

```
#include <xview/xview.h>
#include <xview/frame.h>

/*
 * subframe.c -- create a base frame that has an associated subframe.
 * Pull the pin out of the subframe and its FRAME_DONE_PROC procedure
 * gets called.
 */
main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame, subframe;
    int done_proc();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);

    subframe = (Frame)xv_create(frame, FRAME_CMD,
        FRAME_DONE_PROC, done_proc,
        XV_SHOW,         TRUE,
        NULL);
    xv_main_loop(frame);
}

/*
 * when the pushpin is pulled out, this routine is called
 */
done_proc(subframe)
Frame subframe;
{
    /* we have the choice of vetoing or granting the user’s
     * request to dismiss the frame -- if we choose to dismiss
     * the frame, we must do it manually.  Like so:
     */
    xv_set(subframe, XV_SHOW, FALSE, NULL);
    /* otherwise, we should push the pin back in */
}
```

The call to xv_set(XV_SHOW, FALSE) is safe because no action is taken at all if this was not the result of the pin being pulled out. The frame will not go down (i.e., the xv_set() is a no-op) if the pin is in or is already out. This gives the programmer the flexibility of using the same function (something like `done_proc()`) in other places throughout the application. For example, it could be used in a callback routine for a panel button.

If the subframe must be taken down regardless of whether the pin is in or out, the application should forcefully remove the pin using:

```
xv_set(subframe, FRAME_CMD_DEFAULT_PIN_STATE, FALSE, NULL);
```

Forcefully removing the pin in this fashion does not result in the frame's `done` procedure getting called. This routine is only called when the pin has been removed as a result of an action that the user has taken and when the application has no knowledge of this action.

4.3.4 Showing Resize Corners

It should also be noted that the command frame has no resize corners as the base frame does. This is the default behavior for command frames, but the attribute `FRAME_SHOW_RESIZE_CORNER` can be set to `TRUE` to force the resize corners to be shown. This allows the user to resize command frames the same as base frames. Base frames have resize corners by default, but they can be turned off at creation time or at any time *before the frame is mapped to the screen*. After the frame has been displayed, the resize corners may not be turned off.

4.3.5 Minimum and Maximum Frame Sizes

The attribute `FRAME_MIN_SIZE` allows you to specify a minimum a frame can be resized. It takes two integer parameters, specifying the minimum width and height of the frame. This information is passed onto the window manager as part of the `WM_NORMAL_HINTS` property. Note that the minimum size is only a hint to the window manager. Some window managers may choose to ignore certain application specified hints. Setting both the minimum width and height to 0 effectively removes any application controlled minimum restriction on size. Similarly, `FRAME_MAX_SIZE` allows you to specify a maximum size that a frame can be resized.

4.4 Miscellaneous Attributes

Some frame attributes discussed in the following sections only work with OPEN LOOK window managers. Attributes such as `FRAME_BUSY`, `FRAME_SHOW_RESIZE_CORNERS`, `FRAME_CMD_PINSTATE` communicate with the window manager. You will get unpredictable results if you are not running an OPEN LOOK window manager or if there is no window manager running at all.

4.5 Busy Frames

When running the Push Me application, you might notice a delay between the time that the Hello button is pressed and the time that the subframe is displayed. If a delay might confuse the user about what might be happening, you can provide visual feedback that the application is at work. You can set the FRAME_BUSY attribute for the frame that issues the request that might cause the delay. In Example 4-6 we set the XV_SHOW attribute in the base frame. Thus, show_cmd_frame() might have looked like the following code fragment:

```
show_cmd_frame(subframe,event)

Frame subframe;
Event *event;
{
    xv_set(baseframe, FRAME_BUSY, TRUE, NULL);
    xv_set(subframe, XV_SHOW, TRUE, NULL);
    xv_set(baseframe, FRAME_BUSY, FALSE, NULL);
}
```

The effect of this action is that the base frame's header will be grayed out and the cursor will change to a timeout cursor. When the subframe has been displayed, the base frame's appearance is resumed and the cursor restored. If excessively long delays are expected, then this method might not be adequate—all other buttons and events are suspended until the callback routine has returned control to the Notifier.

Note that FRAME_BUSY only grays the title bar and sets the busy cursor for the frame passed to xv_set(). If your application has many subframes and you wish each of them to become busy, you need to set this attribute for each frame.

4.6 Frame Sizes

The size of any type of frame can be set or queried using either of two convenience functions available from the FRAME package. They are frame_get_rect() and frame_set_rect(). Both use a Rect data type. The origin of the frame as well as its width and height can be set using frame_set_rect(). Of course, the frame must already be created in order to use this function. In the following code, the frame is set at 10,10 on the screen and the dimensions are set to 200 by 300:

```
Frame frame;
Rect rect;

rect.r_top = rect.r_left = 10;
rect.r_width = 200;
rect.r_height = 300;
 . . .
frame = (Frame)xv_create(NULL, FRAME, NULL);

frame_set_rect(frame, &rect);
```

Conversely, the dimensions as well as the position of the frame can be gotten:

```
extern Frame frame;
extern Rect rect;
...
frame_get_rect(frame, &rect);
printf("frame is at %d, %d and is %d by %d\n",
    rect.r_left, rect.r_top, rect.r_width, rect.r_height);
```

If the position of a subframe is queried with:

```
xv_get(subframe, XV_X)
```

or

```
xv_get(subframe, XV_Y)
```

the values returned will be relative to the parent frame. Note: a subframe here is any frame that is created with another frame as its owner, for example:

```
subframe = xv_create(frame, FRAME, ..., NULL);
```

4.7 Frame Colors

In general the frame's color is determined from the value associated with the `OpenWindows.WindowColor` resource. This color will be inherited by the frame and its subwindows. In order to maintain a consistent look across tools, applications should *not* override the colors chosen by the frame.

However, some applications may need to override the default color for the frame. This can be done by creating a CMS and setting it on the frame (see Chapter 21, *Color*, for information on CMS). Doing so will cause the frame to inherit the color at index 0 as the background color and the color at index *n-1* as the foreground color. If the application wants the frame to inherit colors different than those at indices 0 and *n-1*, it must explicitly set those colors with `WIN_FOREGROUND_COLOR` and `WIN_BACKGROUND_COLOR`. Note that this must happen in an `xv_set()` after the frame is created as the frame will override these values during create and use the X resource values.

This is shown in Example 4-8.

Example 4-8. Changing a frame's color

```
/*
 *    frame_color.c
 * This program demonstrates how to set the frame's foreground and
 * background color and make it propagate to the children of the frame.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/cms.h>

#define RED 0
#define BLUE 1
```

Example 4-8. Changing a frame's color (continued)

```
main(argc, argv)
    int      argc;
    char   **argv;
{
    Frame    frame;
    Panel    panel;
    Cms      cms;

    (void)xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);

    cms = xv_create(NULL, CMS,
        CMS_SIZE,           CMS_CONTROL_COLORS + 2,
        CMS_CONTROL_CMS,      True,
        CMS_NAMED_COLORS,      "red", "blue" , NULL,
        NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);

    xv_set(frame,
            WIN_CMS, cms,
            WIN_FOREGROUND_COLOR, CMS_CONTROL_COLORS + RED,
            WIN_BACKGROUND_COLOR, CMS_CONTROL_COLORS + BLUE,
            NULL);

    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,    "Push Me",
        NULL);

    xv_main_loop(frame);
}
```

4.8 Child Windows

The very purpose of frames is to manage subwindows such as panels and canvases. Basically, parenting responsibilities are handled transparently by XView and do not require intervention by the programmer. When creating a new object (such as a panel), you simply specify the frame as the panel's parent or owner.

There are several attributes that can obtain subwindows and subframes from the frame. FRAME_NTH_SUBWINDOW and FRAME_NTH_SUBFRAME are attributes that can be used with xv_get(). Assuming the application has created a subframe, the following code fragment will return the first subframe created:

```
Frame subframe;

subframe = (Frame)xv_get(frame, FRAME_NTH_SUBFRAME, 1);
```

Similarly, if a frame creates a panel and then a canvas, you can get the canvas (because it was the second one created) by using the call:

```
Canvas canvas;

canvas = (Canvas)xv_get(frame, FRAME_NTH_SUBWINDOW, 2);
```

If you attempt to get a subwindow or subframe index but it does not exist, `xv_get()` will return `NULL`.

Laying out subwindows in frames is somewhat automatic, but more explicit layouts can be accomplished by using the macro defined in *<xview/frame.h>* called `frame_fit_all()`. This macro loops on `xv_get()` to get each `FRAME_NTH_SUBWINDOW` and call `window_fit()` to make sure that all the subwindows fit in the frame (if possible). `window_fit()` also serves as a hint for the frame to give it permission to resize any of its subwindows whenever a resize event occurs. For example, say a frame contains a canvas subwindow, but that subwindow's dimensions are set via `XV_WIDTH` and `XV_HEIGHT`. If the user uses the window manager to resize the frame, the frame may or may not resize the canvas depending on whether or not it was given permission to do so via either of the calls to `window_fit()`, `window_fit_height()`, or `window_fit_width()`. See Chapter 5, *Canvases and Openwin*, for more information on `window_fit()`.

4.9 Window Loop

The procedure `xv_window_loop()` maps the frame passed and makes all of the application's other frames and windows "busy" in a way similar to using the attribute `FRAME_BUSY`. However, this procedure does not cause the cursor to change to a stopwatch and the frame header does not show the gray pattern. `xv_window_loop()` does not lock the screen. The form of this procedure is:

```
Xv_opaque
xv_window_loop(frame)
    Frame   frame;
```

`xv_window_loop()` does not return until a call to `xv_window_return()` is made. The form for `xv_window_return()` is:

```
void
xv_window_return(return_val)
     Xv_opaque     return_val;
```

The frame passed in to `xv_window_loop()` can have more than one subwindow of any type.

Making a frame busy in this way should normally be done in a callback associated with the frame. For example, a callback originating from a button on the frame. The return value passed in to `xv_window_return()` is the value returned by `xv_window_loop()`. Since the screen is *not* locked using `xv_window_loop()` the user might be able to dismiss the frame using the window manager. Doing this will cause the application to hang since `xv_window_return()` was not called. You can avoid this by attaching a destroy procedure to the frame; in the destroy procedure, issue a call to `xv_window_return()`.

4.10 Removing Decorations

If a frame does not wish to be controlled by the window manager, and thus have no window manager decorations such as resize corners or pins, the frame should set WIN_TOP_LEVEL_NO_DECOR to TRUE. This attribute is only valid when the frame is created. For more information, see the discussion of override_redirect in Volume One, *Xlib Programming Manual*.

4.11 Setting Properties and Saving Command-line Options

Several frame attributes support setting window properties according to the specifications of the ICCCM (see Volume Zero, *X Protocol Reference Manual*, for more information on ICCCM). These attributes support the WM_SAVE_YOURSELF and WM_COMMAND protocols that set an application's startup options. Also refer to Section 20.9.5, "Modifying A Frame's Destruction," for more information on saving command-line options.

FRAME_WM_COMMAND_ARGC_ARGV lets an application set the command-line options that can be used to (re)start it. The options passed, in addition to XView options, are stored on a property called WM_COMMAND on the frame window. The options passed are stored by XView and will be added to the XView options on the WM_COMMAND property on the frame window, upon receiving a WM_SAVE_YOURSELF request from the session/window manager. The program *xprop* can be used to display a window's properties. Only one base frame window of the application needs to have this property set. This property is read possibly by a session manager to restart clients. The first argument is the number of strings passed in the second argument. The second argument is a pointer to an array containing the command-line option strings. The strings passed are copied and cached on the frame. The following code shows an example using FRAME_WM_COMMAND_ARGC_ARGV.

```
Frame base_frame, second_frame;
char *argv[10];
int  argc = 0;

argv[argc++] = "-I"
argv[argc++] = "ls"
argv[argc++] = "-bold_font"
argv[argc++] = "courier-bold-14"

/*
 * This ensures that the above options are stored
 * on the base frame
 */
 xv_set(base_frame, FRAME_WM_COMMAND_ARGC_ARGV,
                           argc, argv, NULL);
```

Setting this attribute's arguments to NULL and -1 prevents any command-line option information from being saved on the frame. If there are two or more base frames in the application, the second and subsequent base frames should set their FRAME_WM_COMMAND_ARGC_ARGV attributes' arguments to NULL and -1 if they want to avoid multiple invocations of the same application by the session manager.

```
/*
 * This ensures that no command-line information will
 * be stored on this frame.
 */
 xv_set(second_frame,
           FRAME_WM_COMMAND_ARGC_ARGV, NULL, -1, NULL);
```

FRAME_WM_COMMAND_ARGC returns the number of command-line option strings stored on the frame. FRAME_WM_COMMAND_ARGV returns the array containing the command-line option strings stored on the frame. The strings in the array must not be modified by client programs. If the value returned is -1, this means that no command-line information is stored on the frame.

FRAME_WM_COMMAND_STRINGS works in a similar fashion to FRAME_WM_COMMAND_ARGC_ARGV, but it uses strings. It lets an application set the command-line options that can be used to (re)start it. The options passed, in addition to XView options are stored on a property called WM_COMMAND on the frame window. The following example uses FRAME_WM_COMMAND_STRINGS.

```
Frame base_frame, second_frame;

/*  Ensure that the given options are stored on
 *  on the base frame
 */
xv_set(base_frame, FRAME_WM_COMMAND_STRINGS,
        "-I",
        "ls",
        "-bold_font",
        "courier-bold-14",
        NULL,
   NULL);
```

Setting this attribute to -1 prevents any command-line option information from being saved on the frame. For example:

```
xv_set(second_frame, FRAME_WM_COMMAND_STRINGS,
                       -1, NULL, NULL);
```

4.12 Destroying Frames

When the application wants to exit, the user typically initiates the action via the frame menu. A call to `xv_destroy()` destroys the object as well as all objects descended from it. Therefore, all the objects created by an application can be destroyed simply by destroying the base frame, assuming that the base frame is the owner of all those objects. Subframes of the base frame are included, as are icons and panels and so forth. There are exceptions to this (such as server images or fonts), but those exceptions are covered later in chapters specific to those objects.

The following code segment demonstrates how to destroy a base frame. When the routine `quit()` is called, which calls `xv_destroy_safe()` on the base frame, it destroys all the objects in the frame's tree, including panels and panel items.

```
{
    ...
    xv_main_loop(frame);
    puts("The program is now done.");
    exit(0);
}

quit()
{
    xv_destroy_safe(frame);
}
```

What is significant about this segment is that there is code following the call to `xv_main_loop()` so that the routine will return when no more frames are left to display. The `FRAME` package keeps track of all the frames in the application. Each time a frame is created or destroyed, the `FRAME` package updates its internal count of the number of existing frames. This includes frames that are not displayed or frames that are iconified. When the last frame is destroyed, the Notifier stops and `xv_main_loop()` returns. (Note that the `FRAME` package has its own destruction procedures.) Most applications simply exit as shown in the code fragment above. However, if desired, more frames can be created and the Notifier can be restarted. For example, the following code shows how the same base frame may be created five times, assuming that the program does not exit in some manner:

```
Frame   frame;
int     i;

for (i = 0; i < 5; i++) {
    frame = (Frame)xv_create(NULL, FRAME, NULL);
    ...
    xv_main_loop(frame);
}
```

For this to work, there must be a call to `xv_destroy()` for each frame in the application. Granted, this example is rather silly, but consider an application driven by timer interrupts or by network traffic listening for a particular request. Here, there may be no frames displayed until the timer goes off or until the network protocol is initiated. Once this happens, the application that requires user input will create the base frames and enter `xv_main_loop()`. When the user is done and has destroyed all the frames, `xv_main_loop()` returns and the application can continue waiting for alarm timeouts or listening for network traffic.

4.13 Frame Resize and Repaint Events

This section contains some information on resize events for frames. For a detailed discussion of events, refer to Chapter 6, *Handling Input*.

When the size of a window is changed or the window is moved (either by the user or programmatically), a `WIN_RESIZE` event is generated to give the client a chance to adjust any relevant internal state to the new window size. You should *not* repaint the window when receiving a resize event. You will receive a separate `WIN_REPAINT` event when a portion of the window needs to be repainted.

Top level frames and any other top level windows, when moved, may get multiple resize events, from the server and from the window manager. ICCCM mandates that the window manager send these events when the top level window is moved or resized. You can detect this with the test:

```
event_xevent(event)->xconfigure.send_event
```

which returns `TRUE` on events generated by the window manager. Note that all synthetic events delivered will follow real events. For more information on the event actions mandated by ICCCM and of the coordinate space mapping, refer to Section L.4.1.15 of *Inter-Client Communications Manual* in Volume Zero, *X Protocol Reference Manual*.

4.14 Frame Package Summary

Table 4-1 lists the attributes in the `FRAME` package; the procedures are listed below. This information is described fully in the *XView Reference Manual*.

```
frame_get_rect()
frame_set_rect()
xv_window_loop()
```

Table 4-1. Frame Attributes

FRAME_ACCELERATOR	FRAME_NEXT_PANE
FRAME_BUSY	FRAME_NO_CONFIRM
FRAME_CLOSED	FRAME_NTH_SUBFRAME
FRAME_CLOSED_RECT	FRAME_NTH_SUBWINDOW
FRAME_CMD_DEFAULT_PIN_STATE	FRAME_PREVIOUS_ELEMENT
FRAME_CMD_PANEL	FRAME_WM_COMMAND_ARGC
FRAME_CMD_PIN_STATE	FRAME_WM_COMMAND_ARGC_ARGV
FRAME_DEFAULT_DONE_PROC	FRAME_WM_COMMAND_ARGV
FRAME_DONE_PROC	FRAME_WM_COMMAND_STRINGS
FRAME_FOCUS_DIRECTION	FRAME_X_ACCELERATOR
FRAME_FOCUS_WIN	FRAME_PREVIOUS_PANE
FRAME_ICON	FRAME_RIGHT_FOOTER
FRAME_INHERIT_COLORS	FRAME_SHOW_FOOTER

Table 4-1. Frame Attributes (continued)

`FRAME_LABEL`	`FRAME_SHOW_HEADER`
`FRAME_LEFT_FOOTER`	`FRAME_SHOW_LABEL`
`FRAME_MAX_SIZE`	`FRAME_SHOW_RESIZE_CORNER`
`FRAME_MIN_SIZE`	

5
Canvases and Openwin

Perhaps the most important object in the XView Toolkit is a *canvas*—the area in which an application displays graphics and handles input. The canvas object is similar to that of a painter's canvas. The artist's painting is drawn onto the canvas and the canvas is mounted in a frame. The canvas may be larger than the frame, but the person looking at the canvas only sees what is within the boundaries of the frame. If the canvas is larger than what is viewable, the painting can be moved around making different portions of the canvas visible.

An XView canvas object allows the user to view a graphic image that is too large for the window or even the display screen. The viewable portion of the graphic image is part of the *viewport* or *view window* of the image. Many different views of the image can use the same canvas object. While each view maintains its own idea of what it is displaying, the canvas object manages all the view windows as well as the graphic image that all views share. The ability for the canvas to maintain different views of the graphic image is a property that is inherited from the canvas's superclass, the `OPENWIN` package. These properties provide for *splitting* and *scrolling* views. You cannot create a canvas object with multiple views; views are split and joined generally by the user via the attached scrollbars. It is possible to programmatically split and scroll views, but OPEN LOOK's interface specification indicates that scrollbars provide the ability to split views. When a view is split, each new view may be further split into two more views, and so on. All the views are still a part of the same canvas object.

The `OPENWIN` package is an example of a *hidden* class. You cannot instantiate an openwin object independently from canvas or text subwindows. Figure 5-1 shows the openwin and canvas object hierarchy.

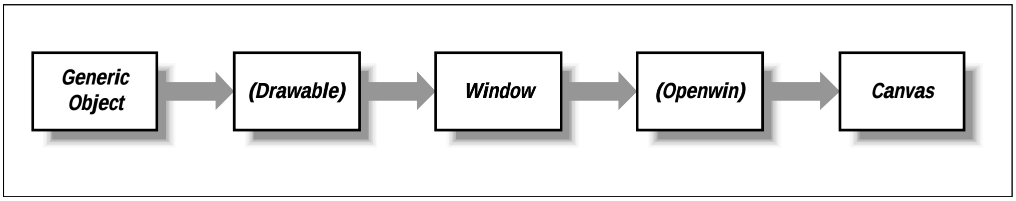


Figure 5-1. Canvas class hierarchy

Chapter 8, *Text Subwindows*, contains information about text subwindows. The canvas object is different from the text object in that it maintains an image that can be manipulated by the user. The openwin object, and thus the canvas object, are broken down into three parts: the *main subwindow*, the *view window*, and the *paint window*.

Each view displays a portion of a corresponding paint window. The paint window need not be the size of the corresponding view or the canvas subwindow. Figure 5-2 shows an example of one canvas object providing separate views into one graphic image. What each view displays is independent of what the other views display. A view may even display a portion of an image that is currently displayed in another view.

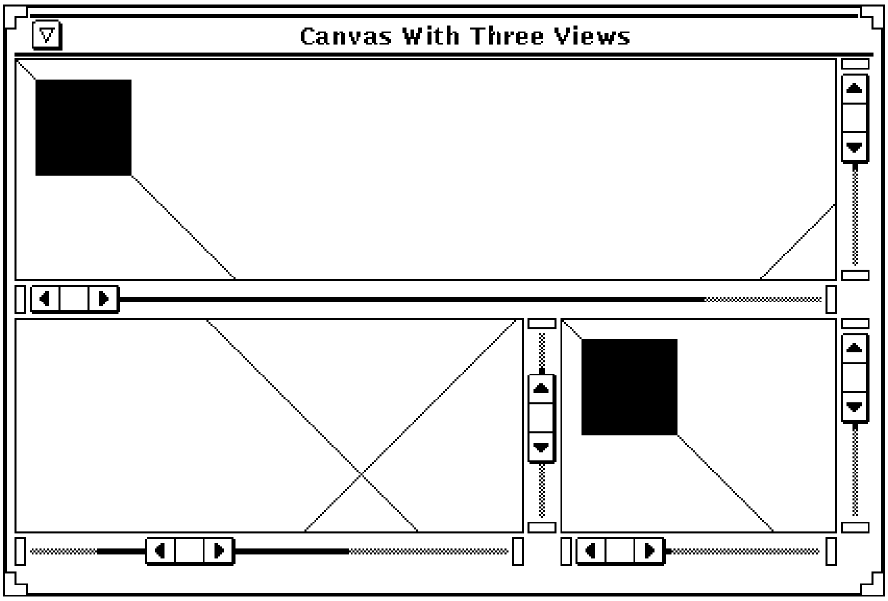


Figure 5-2. A canvas subwindow with multiple views

5.1 Canvas Model

The components of a canvas subwindow and their relationships can be seen in Figure 5-3. To summarize, three types of windows are involved with the canvas object:

Canvas Subwindow	Owned by a frame and manages one or more views. The canvas is subclassed from the `OPENWIN` package so all Openwin attributes must be set to the instance of the canvas object.
View Window	Represents the visible portion of the paint window—whenever the paint window associated with a view window changes, it is reflected in the view window. If there is more than one view window, the views are tiled. Vertical and/or horizontal scrollbars can be attached to the view subwindow to allow the user to modify which portion of

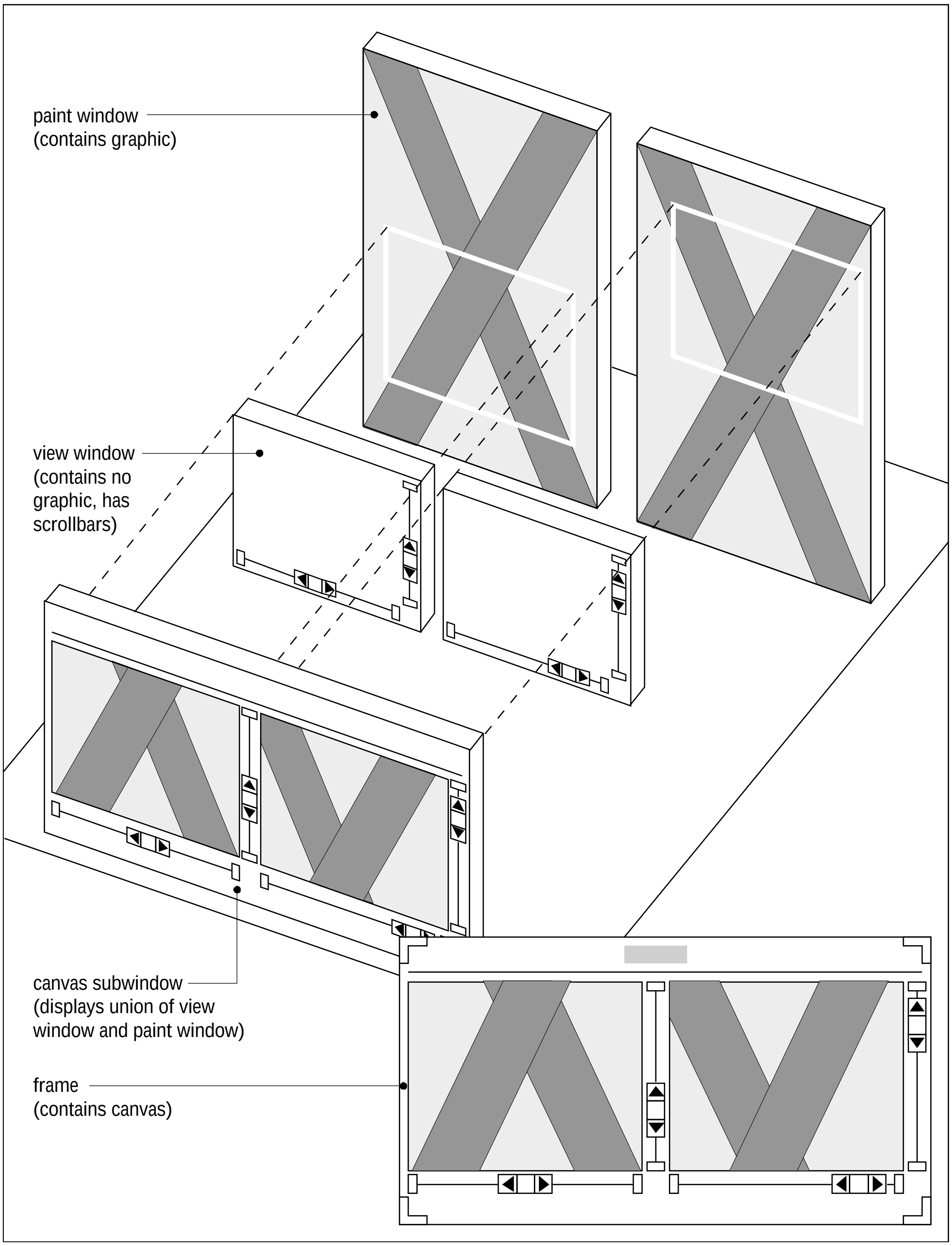

paint window
(contains graphic)
view window
(contains no graphic, has scrollbars)
canvas subwindow
(displays union of view window and paint window)
frame
(contains canvas)

Figure 5-3. Canvases, views, and paint windows

the paint window is displayed for that particular view. The size of the view window can vary among all the views. Only views can be split. No graphics or user events take place in this window.

Paint Window	Graphics and events (mouse/keyboard) take place in the paint window. There is one paint window per view window. All paint windows in the canvas are the same size regardless of the size of the canvas or of the corresponding view windows. When a view is split, the old view reduces in size and a new view is created. With the new view, a new paint window is created that is identical to the paint window from the old view. This includes the same visual, width, height, depth, and graphic image. However, callback functions and event masks are *not* inherited and *must* be manually installed in all new paint windows.

5.2 Creating a Canvas

The `CANVAS` package is defined in the header file <*xview/canvas.h*> so programs that use canvases must include this file. This header file includes the `OPENWIN` package automatically. Like all objects in XView, a canvas is created with `xv_create()`:

```
Canvas canvas;

canvas = (Canvas)xv_create(owner, CANVAS, attrs);
```

Here, `xv_create()` returns a handle to a new canvas subwindow. The `owner` of a canvas must be a `FRAME` object. All three subwindows of the canvas are created at this point: the canvas subwindow, the view window and the paint window. By using this syntax, the attributes of the canvas default to those set for the `CANVAS` package plus any attributes that may be inherited from the owner of the canvas.

The windows in the canvas object inherit many of their attributes from the canvas's parent, screen, or display (depending on the window property). However, some attributes are set or reset explicitly. For example, the attribute `CANVAS_RETAINED`, which controls whether the server should retain windows, is turned on.* Toggling this attribute causes the canvas to cycle through all of the paint windows and change their `WIN_RETAINED` attribute. Also, the window has its `BitGravity` set to `NorthWestGravity` by default. This value is set by the attribute `CANVAS_FIXED_IMAGE`. If `TRUE`, then the `BitGravity` property on the paint window is set to `NorthWestGravity`. If `FALSE`, `BitGravity` is set to `ForgetGravity`. This is discussed in more detail in Section 5.3, "The Repaint Procedure."

The width and height of the paint window and the view window default to the size of the canvas when it is realized. Unless otherwise specified, those sizes are governed by the object that is the owner of the canvas.

*`CANVAS_RETAINED` does not affect the view windows, which are not retained.

5.2.1 Drawing in a Canvas

The use of canvases implies that your application wants to display a graphic image that the user can either manipulate or generate. In conventional (not server-client based) windowing systems, when you request to create a window, you usually get a window back that you can draw into. This is not exactly true for X. Although you get a window back from the request, you are not guaranteed to be able to draw into it until it has been successfully mapped (displayed) on the screen. In the general case for XView, this does not happen until `xv_main_loop()` is called. Therefore, you should *not* do the following:

```
canvas = (Canvas)xv_create(frame, CANVAS, NULL);
win = (Window)xv_get(canvas_paint_window(canvas), XV_XID);

XDrawString(dpy, win, gc, x, y, "Hello World.", 11);
```

Instead, you should design your program such that your repaint routine knows exactly what the contents of the canvas should be so that it can reproduce the image. Here are two helpful hints for typical applications:

- *Use your repaint proc*. Never call any graphics routines before `xv_main_loop()` is called. Routines that draw anything into windows should be called directly or indirectly from your canvas repaint procedure or event handler.
- *Use internal data*. The repaint routine should be able to repaint a canvas window based on some sort of internal data. To maintain system performance, the data should be in core; it should *not* be on disk (in a file), from a network connection or from interaction with the user. If the data is received from those media, the data should already have been updated by the time the repaint routine is called.

To give you an idea of the issues involved here, we'll look at several common applications that typically use canvases.

5.2.1.1 Draw programs

Draw programs are usually applications that maintain a *display list* of geometric shapes such as lines, circles, rectangles and so on. The user generates these shapes using the mouse or keyboard. The canvas's `WIN_EVENT_PROC` procedure handles mouse and keyboard input from the user, and the user interface generally describes the shape that is currently being drawn.

When the user initiates mouse clicks, drags or keyboard actions, the event handler picks up these events and is fully expected to modify the canvas window accordingly (e.g., by "rubber banding" the object). Upon receipt of the appropriate event (button release, perhaps), the event handler adds the new geometric item to the display list.

In order to reconstruct what the canvas should be displaying, the repaint routine references the updated display list. Obviously, the display list should contain enough information in it to be able to tell the repaint routine what colors to use, line thickness, and so on.

5.2.1.2 Paint programs

Paint programs are *pixel based*, meaning that the image that the user manipulates is typically a bitmap (monochrome) or a color pixmap. In this case, the pixmap is used as the internal data. If the user uses a *brush* to modify the image, the modification to the canvas window is handled in the event procedure as above, but the image that the user is editing is also updated so that when the repaint routine is called, it can use a function such as `XCopyArea()` to copy the portion of the image onto the canvas window. This is also a case where the event handler may draw directly onto the window.

If the program starts up and already has an image to work with (e.g., the user requests to load a previously saved image that was stored into the file), *do not render the image onto the paint window before* `xv_main_loop()` *is called.* Eventually, the repaint procedure will be called and the image should be rendered at that time.

5.2.1.3 Text-based programs

If you want to display text, as in desktop publishing packages, or if you just want a simple program that displays a window with text, then the same thing applies as with the previous examples: render the text in the repaint routine based on internal data (e.g., a text string or set of strings). Again, the data must contain enough information to allow your repaint routine to repaint the text as it was intended (e.g., font type, style, size, color).

Event-handling routines that accept keyboard input may render the new text directly to the canvas window, but the text entered should also be saved internally for the benefit of the repaint routine.

5.2.1.4 Visualization programs

It is common to have an application that displays the time, network traffic, CPU usage, file system integrity, or the current Dow Jones Industrial Averages. Such applications get their data from a variety of sources such as the system clock, UNIX sockets, the file system or the output of another application that has been forked. In the past, such applications might have been written where the repaint routine accesses the information. This model does not apply in a networked windowing system because of its asynchronous nature; the window system and the application may not be in sync. A program should have a separate method for retrieving data apart from the repaint procedure. See the program *animate.c* in Chapter 20, *The Notifier*, for an example.

In many of these cases (with the possible exception of the *meters*), if the application is well designed, the repaint routines and the event handlers may be calling the same internal routines which render graphics. Therefore, when writing functions that draw into a canvas, you should consider the generic case where the function could be called from a repaint routine, event handler or anywhere else. The *canvas_event.c* shown in Example 5-3 calls the repaint routine directly from the event handler.

5.2.1.5 Rendering graphics

The preferred form of rendering graphics from an XView application is to use Xlib graphics calls. Volume One, *Xlib Programming Manual*, has a complete discussion of Xlib graphics programming. Throughout this book, you will find examples of drawing into canvases using Xlib graphics routines. Appendix F, *Example Programs*, has several longer programs that demonstrate Xlib graphics. The XView graphics model, which is available, is almost identical to the SunView model for graphics and is provided for backwards compatibility with SunView. Because XView graphics calls are wrappers to the underlying Xlib calls, these functions are not recommended for graphics-intensive applications or for use by programmers who are not already familiar with SunView.

5.3 The Repaint Procedure

It is always the responsibility of the application to repaint its canvas at any time. Even though there may be a retained canvas that the X server maintains, there is no guarantee that there will be enough memory for the server to maintain it. For this reason, all canvases should install a routine to handle repainting.

To install a repaint procedure, use the attribute `CANVAS_REPAINT_PROC` and specify a callback function as its value:

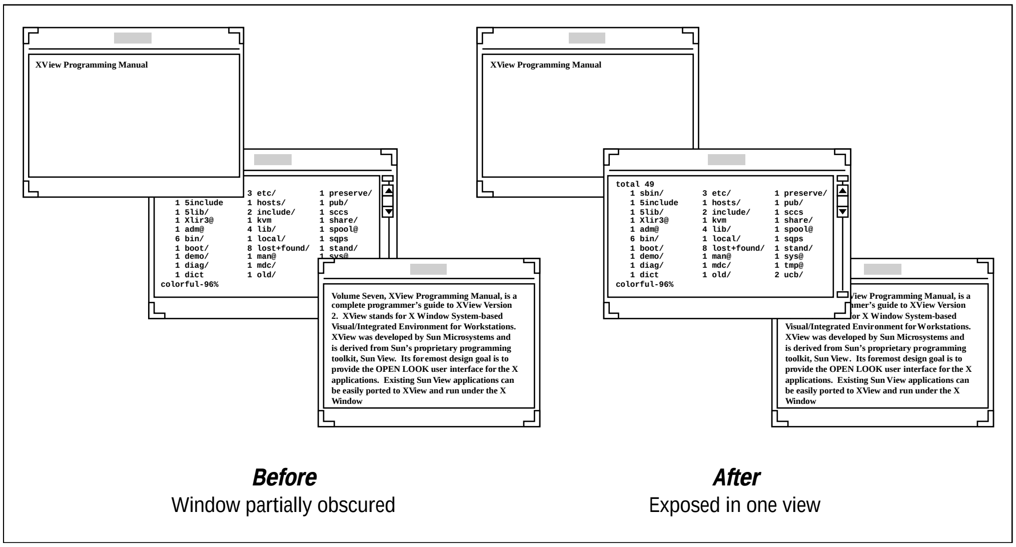
```
extern void my_repaint_proc();
 . . .
canvas = (Canvas)xv_create(frame, CANVAS,
    ...
    CANVAS_REPAINT_PROC,    my_repaint_proc,
    ...
    NULL);
```

The repaint routine installed is called any time all or a portion of the canvas needs to be re-displayed. This always happens when the canvas is mapped on the screen for the first time (causing an `Expose` event). If the canvas is *not* retained, the repaint procedure is called when:

- The canvas is resized.
- The canvas has been moved in front of obscuring windows.
- The user uses the scrollbar to render a different part of the paint window visible.

If the canvas *is* retained and has not changed size, the server refreshes the window without calling the repaint routine. This includes all exposures except for those that are the result of a resize of the window or the initial mapping of the canvas onto the screen. However, if the canvas is not retained, the repaint routine is called in all of these cases.

The repaint callback routine will be called once for each view the canvas is maintaining. If the canvas has been split several times, then the repaint routine will get called for each view that needs repainting. One of the parameters to the callback routine is a variable that describes the region that has been exposed or needs repainting. When a window is initially displayed on the screen, the exposed region is the entire canvas. However, this area may not be a contiguous area of the window. For example, as shown in Figure 5-4, if a window that is partially obscured by two windows is brought forward, two separate areas are exposed.


XView Programming Manual

```
                    3 etc/          1 preserve/
   1 5include       1 hosts/        1 pub/
   1 5lib/          2 include/      1 sccs
   1 Xlir3@         1 kvm           1 share/
   1 adm@           4 lib/          1 spool@
   6 bin/           1 local/        1 sqps
   1 boot/          8 lost+found/   1 stand/
   1 demo/          1 man@          1 sys@
   1 diag/          1 mdc/
   1 dict           1 old/
colorful-96%
```

Volume Seven, XView Programming Manual, is a complete programmer's guide to XView Version 2. XView stands for X Window System-based Visual/Integrated Environment for Workstations. XView was developed by Sun Microsystems and is derived from Sun's proprietary programming toolkit, Sun View. Its foremost design goal is to provide the OPEN LOOK user interface for the X applications. Existing Sun View applications can be easily ported to XView and run under the X Window

XView Programming Manual

```
total 49
   1 sbin/          3 etc/          1 preserve/
   1 5include       1 hosts/        1 pub/
   1 5lib/          2 include/      1 sccs
   1 Xlir3@         1 kvm           1 share/
   1 adm@           4 lib/          1 spool@
   6 bin/           1 local/        1 sqps
   1 boot/          8 lost+found/   1 stand/
   1 demo/          1 man@          1 sys@
   1 diag/          1 mdc/          1 tmp@
   1 dict           1 old/          2 ucb/
colorful-96%
```

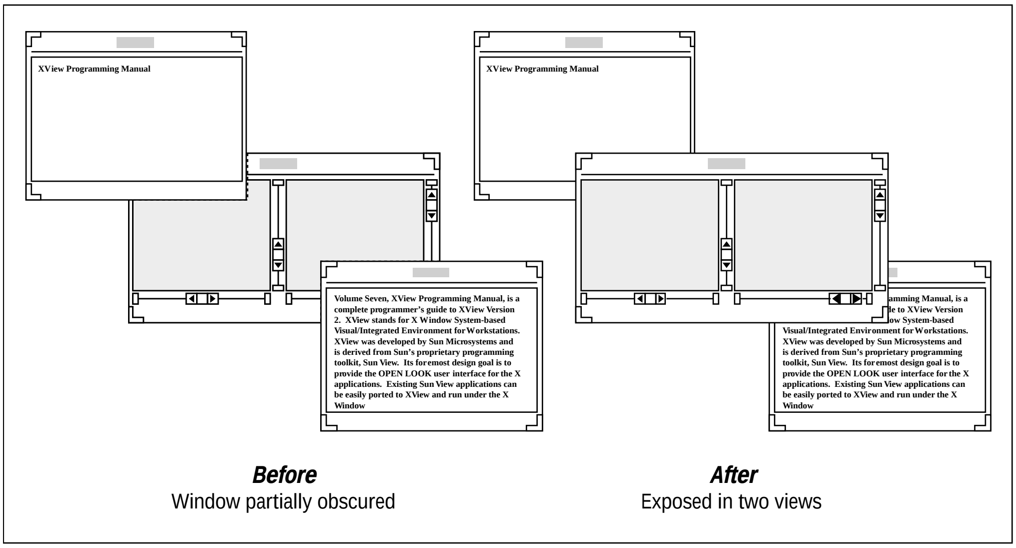
View Programming Manual, is a
mer's guide to XView Version
or X Window System-based
Visual/Integrated Environment for Workstations.
XView was developed by Sun Microsystems and
is derived from Sun's proprietary programming
toolkit, Sun View. Its foremost design goal is to
provide the OPEN LOOK user interface for the X
applications. Existing Sun View applications can
be easily ported to XView and run under the X
Window

Figure 5-4. Window before and after an Expose event

The attribute `WIN_COLLAPSE_EXPOSURES` governs how many times the repaint routine is called. By default, XView collapses `Expose` (and `GraphicsExpose`) X events destined for the same window; that is, XView waits for the exposure count member to reach zero. After all exposure X events have arrived (when `count == 0`), XView generates a `WIN_REPAINT` event and calls the repaint procedure. This `WIN_REPAINT` represents all the areas in the window that have been damaged (it can be a disjoint set of rectangles). The X event associated with the `WIN_REPAINT` event (accessed through the `event_xevent()` macro) represents a bounding rectangle of all the damaged areas in the window (basically the union of all the damaged areas in the window).

However, sometimes the application wants to monitor the incoming `Expose` (and `GraphicsExpose`) X events, monitoring piece by piece the count member in the `Expose` X event itself. This can be done by setting `WIN_COLLAPSE_EXPOSURES` to `FALSE` on the canvas's paint window. As exposures come into a window with this attribute set to `FALSE`, they will be immediately sent to the client's repaint procedure. The client will thus receive several `WIN_REPAINT` events, all for the same window. The area of exposure is set to each region as it is exposed.

By default, the repaint routine is called once per window exposed. However, there may be situations where there are more windows exposed in the same canvas object. For example, in Figure 5-5, two view windows have been exposed as a result of bringing the window to the top of the window stack. In this case, the repaint routine will be called twice—once for the paint window of each view exposed. Only one area of each view window is exposed, so the value of the `WIN_COLLAPSE_EXPOSURES` attribute does not apply.



Volume Seven, XView Programming Manual, is a complete programmer's guide to XView Version 2. XView stands for X Window System-based Visual/Integrated Environment for Workstations. XView was developed by Sun Microsystems and is derived from Sun's proprietary programming toolkit, Sun View. Its foremost design goal is to provide the OPEN LOOK user interface for the X applications. Existing Sun View applications can be easily ported to XView and run under the X Window

Figure 5-5. Window with two views before and after an Expose event

If at any time you need to get the entire viewable area of the canvas, or more specifically, of an arbitrary paint window within the canvas, you can use the attribute `CANVAS_VIEW-ABLE_RECT`:

```
Rect *rect;
Xv_Window pw = canvas_paint_window(canvas);

rect = (Rect *)xv_get(canvas, CANVAS_VIEWABLE_RECT, pw);
```

The `rect` pointer returned points to an internal data structure that describes the viewable area of the paint window specified. This structure changes for each call, so if the value is to be retained, it should be copied.

Before the repaint routine is called, the window can be cleared in one of two ways. If `CANVAS_AUTO_CLEAR` is set to `TRUE`, then the paint window being repainted is automatically cleared. `CANVAS_AUTO_CLEAR` is really defined as `OPENWIN_AUTO_CLEAR` since this is a property of the `OPENWIN` package. Automatically clearing the window happens any time the window needs repainting. This means that the exposed area represents the entire window. If this attribute is set to `FALSE` (the default), the repaint routine should prepare to clear all, or portions of, the window that needs to be repainted. The contents of the window in the exposed areas is undefined. If you are going to repaint those areas opaquely (that is, leave no transparent portions), then you do not need to clear the area. However, if any transparent portion of the area will be repainted (e.g., if your `gc.function` is set to `GXxor`), you should clear the window first by using `XClearArea()` or `XClearWindow()`.

Alternatively, the window may be cleared automatically by the Xlib internals if the window has actually changed size. The BitGravity attribute for the window controls whether the data in the window is cleared or just moved around to different locations of the window according to its new size. If BitGravity is set to ForgetGravity, then the data in the window is discarded, resulting in the window getting cleared and the canvas's repaint procedure getting called. As mentioned before, this value can be set by setting the attribute CANVAS_FIXED_IMAGE to FALSE. But to have more direct control over the BitGravity of the window, the window attribute WIN_BIT_GRAVITY may be set to any of the legal values provided by Xlib (ForgetGravity, NorthGravity, NorthWestGravity and so on). Section 4.3.3, "Bit Gravity," in Volume One, *Xlib Programming Manual*, discusses this in full detail. Note: this should not be confused with WIN_WINDOW_GRAVITY which controls the reposition of subwindows when a parent window is resized. This task is left to the FRAME package, since it controls subwindow layout.

If the attribute CANVAS_CMS_REPAINT is set to TRUE, the repaint procedure is called automatically whenever a new colormap segment is set on the canvas or the foreground and background colors of the canvas are changed using WIN_FOREGROUND_COLOR and WIN_BACKGROUND_COLOR.

The parameters to the repaint routine provide information about which window and which areas within the window need to be repainted.

The repaint procedure takes one of two different forms:

```
void
repaint_proc(canvas, paint_window, repaint_area)
    Canvas        canvas;
    Xv_Window     paint_window
    Rectlist     *repaint_area;
```

or:

```
void
repaint_proc(canvas, paint_window, dpy, xwin, area)
    Canvas        canvas;
    Xv_Window     paint_window
    Display      *dpy;
    Window        xwin;
    Xv_xrectlist *area;
```

The routine takes the first or second form depending on the value of the attribute CANVAS_X_PAINT_WINDOW. If FALSE, the first, simpler form of the repaint procedure is called. If TRUE, the repaint routine gets passed the parameters shown in the second form. The second method is more useful since it saves you from writing code for extracting the Display and the XID of the paint window.

In both forms, the first two parameters are rather obvious—the paint_window is the paint window associated with the canvas that needs repainting (not the view window). The paint_window contains an X window whose XID can be gotten by using xv_get():

```
Window xwin = (Window)xv_get(paint_window, XV_XID);
```

xwin is set to the actual X window referenced by the paint window. This is the way to obtain the XID from the paint window because the first form of the repaint procedure (when CANVAS_X_PAINT_WINDOW is FALSE) does not provide a handle for you.

In the first form of the repaint procedure, the only other parameter is the `repaint_area`. This is a linked list of rectangular areas in the paint window that have been exposed and need to be repainted. The `Rectlist` type is a linked list of `Rect`'s as shown in *<xview/rect.h>*. The declaration for the `Rect` type is:

```
typedef struct rect {
    coord r_left, r_top;
    short r_width, r_height;
} Rect;
```

The type `coord` is `#define`'d as type `short`. The `Rectlist` is declared in *<xview/rectlist.h>* as:

```
typedef struct  rectnode {
   struct rectnode *rn_next;  /* Pointer to next rectnode */
   struct rect      rn_rect;
} Rectnode;

typedef struct rectlist {
   coord           rl_x, rl_y; /* Offset to apply to each rect
                                * in list including bound */
   struct rectnode *rl_head;    /* Pointer to first rectnode */
   struct rectnode *rl_tail;    /* Pointer to last rectnode */
   struct rect      rl_bound;   /* Describes bounding rect of
                                * all rects in list */
} Rectlist;
```

The `repaint_area` parameter is of type `Rectlist`, so the application has several ways it can approach repainting the window. It could just ignore the parameter and repaint the entire window, or it could repaint the entire area described by the `rl_bound` field of the `rectlist` structure or it could loop through all the `Rectnode`'s and repaint those areas individually. Deciding which method to choose should be based on how complicated the redrawing is.

When `CANVAS_X_PAINT_WINDOW` is set to `FALSE`, the internal clipping for the paint window is set to be the same region or regions described by the repaint parameter of the repaint routine. In most cases, this means that `OPENWIN_AUTO_CLEAR` will only clear those areas—not the entire window.* The default behavior may be overridden by setting `WIN_NO_CLIPPING` to `TRUE` for the *canvas*. Setting this attribute cycles through all the paint windows in the canvas and sets the window property `WIN_NO_CLIPPING`. Having the clipping on the window turned off means that `OPENWIN_AUTO_CLEAR` will cause the entire window to be cleared.†

By setting the attribute `CANVAS_X_PAINT_WINDOW` to `TRUE`, the repaint routine gets passed the extra X-specific parameters as shown in the second form of the repaint procedure. These extra parameters include handles to the `Display` and the X window of the `paint_window`.

*For those using the SunView-compatible drawing routines such as `pw_vector()`, rendering is clipped to the area or areas described by `repaint_area`.
†This also affects the SunView-compatible routines—`pw_vector()` will not be clipped.

Example 5-1 shows how a repaint procedure might be used.

Example 5-1. The line.c program

```
/*
 * line.c -- demonstrates installing a repaint routine in a canvas.
 * The routine is called whenever the canvas needs to be repainted.
 * This usually occurs when the canvas is exposed or resized.
 */
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame;
    void  canvas_repaint_proc();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);

    (void) xv_create(frame, CANVAS,
        CANVAS_REPAINT_PROC,    canvas_repaint_proc,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        NULL);

    xv_main_loop(frame);
}

/*
 * repaint routine draws a line from the top left to the bottom right
 * corners of the window
 */
void
canvas_repaint_proc(canvas, paint_window, dpy, xwin, xrects)
Canvas        canvas;         /* unused */
Xv_Window     paint_window;   /* unused */
Display      *dpy;
Window        xwin;
Xv_xrectlist *xrects;         /* unused */
{
    GC gc;
    int width, height;

    gc = DefaultGC(dpy, DefaultScreen(dpy));
    width = (int)xv_get(paint_window, XV_WIDTH);
    height = (int)xv_get(paint_window, XV_HEIGHT);

    XDrawLine(dpy, xwin, gc, 0, 0, width, height);
}
```

Because the program uses Xlib calls (and thus, the repaint routine is being passed different parameters), the header files *<X11/Xlib.h>* and *<xview/xv_xrect.h>* must be added at the top of the program. The parameters `dpy` and `xwin` are handles to the X `Display` and the X `Window`, respectively. The `GC` (graphics context) is taken from the default `GC` of the screen.

The `xrects` parameter represents the exposed, or "damaged," region of the paint window. The `Xv_xrectlist` is declared in *<xview/xv_xrect.h>* as:

```
     #define XV_MAX_XRECTS 32
     typedef struct {
        XRectangle   rect_array[MAX_XRECTS];
        int          count;
     } Xv_xrectlist;
```

When the canvas is first displayed, this region will be the entire paint window (or the portion that is viewable by the view window). But in cases where another window that has partially obscured it moves away, perhaps only a portion of the paint window will need repainting. Therefore, the `xrects` variable can aid in setting the clip rectangles in the `GC`, as shown in Example 5-2.

Example 5-2. Repainting objects within a damaged region

```
/* canvas_repaint_proc()
 *
 * Draws onto the canvas using Xlib drawing functions.
 *
 * Uses the current clipping rectangle to:
 *  1. Restrict graphics output by setting the
 *     clip_mask in the graphics context.
 *  2. Do "smart repainting" by only painting the objects
 *     that lie within the damaged region (not being done
 *     in this example).
 */
void
repaint_proc(canvas, paint_window, display, xid, xrects)
    Canvas canvas;
    Xv_Window paint_window;
    Display *display;
    Window xwin;
    Xv_xrectlist *xrects;
{
    extern GC gc;
    int width, height;

    width = (int)xv_get(paint_window, XV_WIDTH);
    height = (int)xv_get(paint_window, XV_HEIGHT);

    /*
     * Set clip rects, if any
     */
    if (xrects)
        XSetClipRectangles(display, gc, 0, 0, xrects->rect_array,
            xrects->count, Unsorted);
    else {
        XGCValues gc_val;

        gc_val.clip_mask = None;
```

Example 5-2. Repainting objects within a damaged region (continued)

```
        XChangeGC(display, gc, GCClipMask, &gc_val);
    }

    XDrawLine(display, xwin, gc, 0, 0, width, height);
}
```

Because this routine sets the clip mask of the `GC`, we want to be sure that we *do not use the default* `GC` of the screen as we did in Example 5-2 or it will interfere with other programs (such as the window manager). The `GC` shown here is declared as `extern`, assuming that the application has created it somewhere else using `XCreateGC()`.

5.4 Controlling Canvas Sizes

The size of the canvas subwindow is usually determined by the frame window. Thus, the canvas changes as the user resizes the frame. Applications largely concern themselves with the size of the paint window.* The paint window does not affect the size of the viewable canvas, but the viewable portion of the paint window is important.

Although the width and height of the canvas subwindow can be set explicitly, unless done so, the default size of the subwindow and the paint window is determined by the parent frame. If the frame resizes, the canvas object resizes proportionally according to how the frame chooses to resize the canvas. If several other windows (canvases, panels, whatever) are in the frame, the frame might choose to lay out and size those subwindows differently (according to available and required space from other windows). The canvas window itself, as well as all window objects, can be sized using `XV_WIDTH` and `XV_HEIGHT`.

5.4.1 Automatic Canvas Sizing

The paint window's size may fluctuate with that of the canvas subwindow's size. The attributes `CANVAS_AUTO_EXPAND` and `CANVAS_AUTO_SHRINK` maintain the relation of the canvas subwindow and paint window in the event of any kind of window resizing. Both of these attributes default to `TRUE`, allowing the paint window to always correspond to the size of the canvas subwindow. If the canvas subwindow becomes larger, the paint window size changes to that size. If the frame changes size, the canvas subwindow changes size and so does the paint window. This happens regardless of how many view windows there are. The size of view windows does not affect the size of the paint window.

Specifically, if `CANVAS_AUTO_EXPAND` is `TRUE`, then the width and height of the paint window cannot be less than that of the canvas subwindow. Setting the attribute `CANVAS_AUTO_EXPAND` allows the paint window to grow bigger as the user stretches the window. If a resize of the subwindow occurs such that the size of the paint window is less than the size of the canvas subwindow, the paint window is expanded to be at least that size.

*There may be more than one paint window to a canvas; but all paint windows in a canvas are the same size, so it's a moot point.

Conversely, if the canvas subwindow's size shrinks, then the paint canvas size does not change because its size is already greater than or equal to the size of the canvas subwindow—no expansion is necessary.

If CANVAS_AUTO_SHRINK is TRUE, the canvas object checks that width and height of the paint window are not greater than that of the canvas subwindow. Setting CANVAS_AUTO_SHRINK forces the paint window to grow smaller as the size of the canvas subwindow gets smaller. If the user resizes the frame such that the canvas subwindow is smaller than the size of the paint window, then the paint window is reduced to the size of the new subwindow.

You can also set a minimum width and height for the canvas using the attributes CANVAS_MIN_PAINT_WIDTH and CANVAS_MIN_PAINT_HEIGHT. Regardless of whether or not CANVAS_AUTO_SHRINK is set to TRUE or FALSE, the attributes CANVAS_MIN_PAINT_WIDTH and CANVAS_MIN_PAINT_HEIGHT impose the minimum CANVAS_WIDTH and CANVAS_HEIGHT respectively.

5.4.2 Explicit Canvas Sizing

The attributes CANVAS_WIDTH and CANVAS_HEIGHT can be set to establish the size of the *paint window*. Automatic sizing should be turned off. Otherwise, as soon as the canvas window is realized, the paint window may be automatically resized to the new dimensions. This all depends on whether either or both of the auto-expand or auto-shrink attributes are set. The following code fragment shows that one can be set and the other unset for specific needs:

```
Canvas canvas;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_AUTO_EXPAND,  TRUE,
    CANVAS_WIDTH,        100,
    CANVAS_HEIGHT,       200,
    NULL);
```

With these settings, the paint window will initially be set to 100 by 200. If the subwindow is realized at a larger size, the canvas will be expanded to the new dimensions. That is, if the frame in which the canvas resides is larger, it may affect the initial size of the paint window. However, if the canvas is realized or resized at smaller dimensions, the canvas will retain its original size. In short, these settings will force the paint window to grow to the maximum size that the window will ever be—it will never shrink. In typical usage, you would set the auto-expand and auto-shrink attributes to FALSE and explicitly set CANVAS_WIDTH and CANVAS_HEIGHT. Alternatively, you would not initialize the width and height and set both CANVAS_AUTO_EXPAND and CANVAS_AUTO_SHRINK to TRUE. A draw program might allow the paint window to be sized automatically, since the display list of geometric objects is the underlying feature of the program. However, a paint program would set explicit width and height attributes of the graphic, disallowing any resizing of that graphic.

The following code fragment creates a canvas with a fixed-size paint window that is not affected by resizing:

```
Canvas canvas;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_AUTO_EXPAND,  FALSE,
    CANVAS_WIDTH,        1000,
    CANVAS_HEIGHT,       1000,
    NULL);
```

This call sets the initial size of the paint window to 1000 by 1000 pixels. The origin of the paint window's coordinate system is the upper-left corner (0,0) and the lower-right corner (`CANVAS_WIDTH-1, CANVAS_HEIGHT-1`). Note that we did not set the size of the canvas subwindow. Instead, we allowed it to be determined by the frame size. The size of the paint window remains constant regardless of how the frame and canvas subwindow is resized. If the frame or the canvas subwindow resizes, the subwindow merely changes its view of the underlying paint window, which remains constant.

In the following code fragment, we set the size of the canvas subwindow, using generic attributes:

```
Canvas canvas;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_AUTO_EXPAND,  FALSE,
    CANVAS_WIDTH,        1000,
    CANVAS_HEIGHT,       1000,
    XV_WIDTH,            200,
    XV_HEIGHT,           100,
    NULL);
```

Here, a canvas subwindow is created that is 200 pixels wide and 100 pixels high. All other attributes about this canvas object are the same as the previous example: the paint window is going to be 1000x1000 in width and height. The problem with this canvas is that the user has no way to view different parts of the paint window. To handle that, scrollbars should be attached to the canvas to provide scrolling. See Section 5.5, "Scrolling Canvases."

5.4.3 Tracking Changes in the Canvas Size

In the event that the canvas paint window has been resized, the program has the opportunity to track this event by installing a callback routine. This routine is installed using `CANVAS_RESIZE_PROC`. The client's resize procedure is called only when the width or height of the canvas's paint window changes. Its form is:

```
void
sample_resize_proc(canvas, width, height)
    Canvas canvas;
    int    width;
    int    height;
```

The parameters to the resize procedure are the canvas, and the width and height of the canvas.

If you need to handle resize events for the canvas or the view windows, provide an event handler for those windows, using the attributes WIN_EVENT_PROC and WIN_CONSUME_EVENTS (see Section 5.7, "Handling Input in the Canvas Package," for more information).

5.5 Scrolling Canvases

Many applications need to view and manipulate a large object through a smaller viewing window. To facilitate this, packages that are subclassed from the openwin class may have scrollbars attached to their subwindows.

The following code fragment creates a canvas that can be scrolled in both directions:

```
Canvas     canvas;
Scrollbar  h_scrollbar, v_scrollbar;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_EXPAND,  FALSE,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_WIDTH,        1000,
    CANVAS_HEIGHT,       1000,
    NULL);

h_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION, SCROLLBAR_HORIZONTAL,
    NULL);

v_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION, SCROLLBAR_VERTICAL,
    NULL);
```

Because the SCROLLBAR package is being used here, the header file <*xview/scrollbar.h*> must be included. Chapter 10, *Scrollbars*, discusses scrollbars more completely and also gives further examples of how to scroll canvases.

The owner of the scrollbars is the canvas so that the scrollbars are automatically attached to the canvas's view. If the user scrolls the canvas, your canvas's repaint procedure will be called provided that the canvas's WIN_RETAINED attribute is set to FALSE. This is important because setting WIN_RETAINED to TRUE assumes that you are not interested in handling repainting for scrolling. In other words, as long as the user does not do anything that changes the contents of the image, you do not need to be informed when the user scrolls the image. If you want to be informed of scrolling, set WIN_RETAINED to FALSE and your repaint routine will be called with the *exposed area* parameter describing the new area that just scrolled into view.* If there are many views in the canvas, the paint window associated with the view that scrolled is in the second parameter to the repaint function: the paint_window.

*The exposed area passed to the repaint procedure is of type Xv_xrectlist if CANVAS_X_PAINT_WINDOW is set to TRUE or Rectlist if CANVAS_X_PAINT_WINDOW is set to FALSE.

5.6 Splitting Canvas Views

There are two methods by which the application may split the views of a canvas (or any openwin-classed object). The first method is for the user to use the scrollbars to split views. This method is more common and complies with the OPEN LOOK specification. The alternate method is for the application to make calls to `xv_set()` with attribute-value pairs that tell where and how a view should be split.

Whenever views are split, the following attributes are propagated from the split paint window to the new paint window:

- `WIN_BACKGROUND_COLOR`
- `WIN_FOREGROUND_COLOR`
- `WIN_CMS`
- `WIN_COLUMN_GAP`
- `WIN_COLUMN_WIDTH`
- `WIN_CURSOR`
- `WIN_EVENT_PROC`
- `WIN_ROW_GAP`
- `WIN_ROW_HEIGHT`
- `WIN_X_EVENT_MASK`

5.6.1 Splitting Views Using Scrollbars

To set up the canvas so that the user can split it using scrollbars, the canvas should have scrollbars attached as shown in the previous example with the additional attribute `SCROLLBAR_SPLITTABLE` set to `TRUE`:

```
h_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION,    SCROLLBAR_HORIZONTAL,
    SCROLLBAR_SPLITTABLE,   TRUE,
    NULL);

v_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION,    SCROLLBAR_VERTICAL,
    SCROLLBAR_SPLITTABLE,   TRUE,
    NULL);
```

With this attribute set, the scrollbars have the ability to split view windows in two. The user splits the view by selecting the *cable anchors* at the endpoints of the scrollbars and dragging them towards the center of the scrollbar. A pop-up menu provided with the scrollbars also provides this functionality. The entire view will be split at the point the mouse button is released, provided there is enough room for a new view at that point. Each view can scroll its own underlying paint window independently of other views.

5.6.2 Splitting Views Using xv_set()

Splitting a view by setting attribute-value pairs in the view is a less common method since the scrollbar already provides this functionality. However, there is a programmatic interface for splitting views whether or not those views have scrollbars attached to them.

The attribute OPENWIN_SPLIT is followed by a list of attribute-value pairs that indicate specifically how a view is to be split. Only attributes that are prefixed with OPENWIN_SPLIT may be used in this NULL-terminated list. Other attributes are ignored. The following demonstrates how an arbitrary view window can be split into two parts:

```
Xv_Window view;
view = (Xv_Window)xv_get(canvas, OPENWIN_NTH_VIEW, 0);

xv_set(canvas,
    OPENWIN_SPLIT,
        OPENWIN_SPLIT_VIEW,       view,
        OPENWIN_SPLIT_DIRECTION,  OPENWIN_SPLIT_HORIZONTAL,
        NULL,
    NULL);
```

This very simple example shows that the first *view window* in the canvas will be split horizontally. The place in which the split takes place is, by default, the position of the scrollbar in the view. Assuming the code fragment above, the window is split so that the new view is the same width as the original view but the height is split at the position of the scrollbar. The original view is the remaining height and is on top of the new view.

5.6.3 Getting View Windows

If a canvas has been split several times, resulting in multiple view and paint windows, it is possible to get a handle to a particular view or paint window. This can be done either at the time the view was split or by using xv_get().

5.6.3.1 Getting the newest view

If you want to be notified when the user splits or joins views, you can specify the attribute OPENWIN_SPLIT_INIT_PROC for when the user *splits* a view, and OPENWIN_SPLIT_DESTROY_PROC when the user *joins* a view. These attributes are set in the canvas (or any openwin object). Set these functions by using xv_create() or xv_set() in the following manner:

```
extern void init_split(), join_view();

xv_create(frame, CANVAS,
    ...
    OPENWIN_SPLIT,
        OPENWIN_SPLIT_INIT_PROC,    init_split,
        OPENWIN_SPLIT_DESTROY_PROC, join_view,
    NULL,
    ...
NULL);
```

Write the split and join functions, which take the following parameters:

```
void
init_split(origview, newview, pos)
    Xv_Window origview, newview;
    int pos;

void
join_view(view)
    Xv_Window view;
```

The `pos` parameter above represents the split position, in pixels, of the view. The `origview` and the `newview` parameters represent the view that was originally split and the new resulting view, respectively. These are *not* the paint windows; they are the views themselves. To get a handle to the associated paint window from these views, you can use:

```
Xv_Window paint_window;

paint_window = (Xv_Window)xv_get(view, CANVAS_VIEW_PAINT_WINDOW);
```

Example 6-1 in Chapter 6, *Handling Input*, shows how to handle input in different views.

5.6.3.2 Getting arbitrary views

For each view in an `OPENWIN` object, you can get either the view window or the paint window by choosing either the `CANVAS_NTH_PAINT_WINDOW` or the `OPENWIN_NTH_VIEW` attribute and an integer value for the view window. The first window is 0 and the last window is *n*-1, where *n* is the number of view windows. For instance, to get the second paint window in the canvas, you can use:

```
xv_get(canvas, CANVAS_NTH_PAINT_WINDOW, 1, NULL);
```

You can get the number of available views by calling:

```
int nviews = (int)xv_get(canvas, OPENWIN_NVIEWS);
```

Remember that the number of views corresponds directly to the number of paint windows. Each paint window can be accessed in order, using a simple loop like the following:

```
Xv_Window window;
Canvas    canvas;
int       i = 0;

while (window = (Xv_Window)xv_get(canvas, CANVAS_NTH_PAINT_WINDOW, i)) {
    draw_into_window(window);
    i++;
}
```

The call to `xv_get()` returns `NULL` if you try to get a window number that does not exist (XView does the error checking). Thus, the loop terminates when `xv_get()` returns `NULL`. The value of `i` represents the number of views in the canvas subwindow.

XView provides a pair of macros that facilitate looping through a set of views in a canvas: `CANVAS_EACH_PAINT_WINDOW` and `CANVAS_END_EACH`. The previous loop could be written as:

```
Xv_Window window;
Canvas    canvas;

CANVAS_EACH_PAINT_WINDOW(canvas, window)
    draw_into_window(window);
CANVAS_END_EACH
```

Because the paint windows are different from the view windows, a slightly different method is used for getting view windows:

```
Xv_Window view;
Canvas    canvas;
int       i = 0;

while (window = (Xv_Window)xv_get(canvas, OPENWIN_NTH_VIEW, i)) {
    /* process window */
    i++;
}
```

There is also a macro that loops through all the views in the canvas:

```
Xv_Window view;
Canvas    canvas;

OPENWIN_EACH_VIEW(canvas, view)
    ...
OPENWIN_END_EACH
```

You can get the paint window associated with a view by using the attribute `CANVAS_VIEW_PAINT_WINDOW`:

```
Xv_Window view;
Xv_Window paint_window;

paint_window = (Xv_Window)xv_get(view, CANVAS_VIEW_PAINT_WINDOW);
```

This is useful in situations where you are given the view window and need to get the paint window associated with it. For example, the routines called when views are *split* or *joined* are passed handles to view windows. When a view is split, you will need to get the paint window associated with the new view to install event or repaint callbacks.

5.7 Handling Input in the Canvas Package

This section discusses, to a limited degree, the method for handling and specifying events in a canvas. For a detailed discussion of the types of events used and the proper method for handling them, see Chapter 6, *Handling Input*.

5.7.1 Default Events

The default canvas_paint_window event mask is composed of: KBD_USE, KBD_DONE, WIN_MOUSE_BUTTONS, ACTION_HELP, and WIN_ASCII_EVENTS. Other events may be added to the window event mask by using xv_set() and passing the appropriate parameters. The following shows how to enable notification once the Meta key has gone down by enabling META events:

```
xv_set(canvas_paint_window(canvas),
    WIN_CONSUME_EVENT, WIN_META_EVENTS,
    NULL);
```

An application that does not need to know about release events can ignore all release events from mouse buttons and keyboard keys, by enabling WIN_UP_EVENTS by calling:

```
xv_set(canvas_paint_window(canvas),
    WIN_CONSUME_EVENT, WIN_UP_EVENTS,
    NULL);
```

5.7.2 Notification of Events

In addition to specifying which events the application needs to know about, the program should also install an event callback routine that is called when one of the specified events takes place. The callback routine for event handling is installed using WIN_EVENT_PROC. Included are samples that demonstrate how to handle events appropriately using a combination of repaint and event callback routines. However, for a complete discussion of events, you should consult Chapter 6, *Handling Input*, and Chapter 20, *The Notifier*.

The sample program, *canvas_event.c*, in Example 5-3 first creates a base frame. Then it creates a canvas with the attribute CANVAS_X_PAINT_WINDOW set to TRUE because its repaint procedure (repaint_proc) uses Xlib routines to clear the window and draw text strings in the canvas.

Next we specify the events that the application should handle when they occur on the paint_window. We are going to listen for keyboard events, pointer motion events and pointer button events. We have not assigned any responses to these events yet; we have just registered them for this window with XView so that the application will be called back if they occur.

We then set the paint window's event handling procedure to be event_proc. This is the routine that will decide what to do when the events occur. XView is then started up by calling xv_main_loop(), in which event processing starts.

The event_proc is called by the Notifier whenever a registered event takes place in any view that has an event handling procedure set. The event_proc looks at the type of event it has received and determines the appropriate message to display in the paint window. There is a different message for each type of event that we have registered. There are three message buffers, one each for keyboard events, pointer motion events and pointer button events. After the message buffers are updated, the repaint procedure is called to display them. Note that we are reusing the `repaint_proc`, instead of writing more code just to display the messages. See Section 5.2.1, "Drawing in a Canvas."

If the event we have received is of no interest to us, then we return. It is important to do this because the events `WIN_REPAINT` and `WIN_RESIZE` are delivered regardless of the events we have registered with the Notifier. These two events will eventually result in the Notifier calling the repaint procedure anyway, so it is not necessary to call it redundantly from here. See Chapter 6, *Handling Input*, for details about this.

In *canvas_event.c*, the `repaint_proc` simply clears the paint window and then displays the three messages in it, at a fixed position and using the default font. In the case of a pure repaint callback (from the Notifier, not the `event_proc`), the messages will just repeat the last event's messages.

Example 5-3. The canvas_event.c program

```
/*
 *  canvas_event.c
 *  Demonstrates how to get keyboard and mouse events in an canvas
 *  window.  Looks for keyboards, pointer movement and button
 *  events and displays the info in the canvas.
 */
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>

void    event_proc(), repaint_proc();
char    kbd_msg[128], ptr_msg[128], but_msg[128];

/*
 * main()
 *      Create a canvas specifying a repaint procedure.
 *      Get the paint window for the canvas and set the input
 *      mask and the event procedure.
 */
main(argc, argv)
int argc;
char *argv[];
{
    Frame       frame;
    Canvas      canvas;

    /* Initialize XView */
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    /* Create windows -- base frame and canvas. */
    frame = (Frame)xv_create(NULL, FRAME, NULL);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               300,
        XV_HEIGHT,              110,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    repaint_proc,
        NULL);
    window_fit(frame);

    /* Set input mask */
    xv_set(canvas_paint_window(canvas),
```

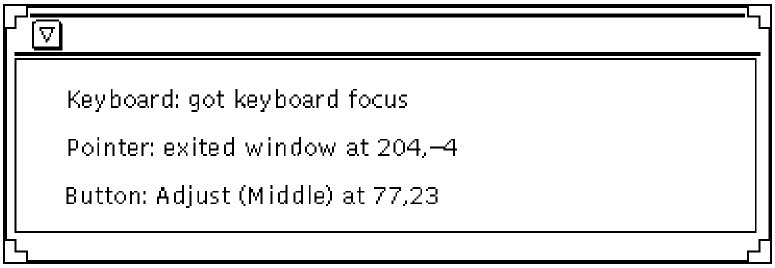
Example 5-3. The canvas_event.c program (continued)

```
        WIN_EVENT_PROC,         event_proc,
        WIN_CONSUME_EVENTS,
            KBD_DONE, KBD_USE, LOC_DRAG, LOC_MOVE, LOC_WINENTER,
            LOC_WINEXIT, WIN_ASCII_EVENTS, WIN_MOUSE_BUTTONS,
            NULL,
        NULL);

    /* Initial messages */
    strcpy(kbd_msg, "Keyboard: key press events");
    strcpy(ptr_msg, "Pointer: pointer movement events");
    strcpy(but_msg, "Button: button press events");

    /* Start event loop */
    xv_main_loop(frame);
}

/*
 * event_proc()
 *      Called when an event is received in the canvas window.
 *      Updates the keyboard, pointer and button message strings
 *      and then calls repaint_proc() to paint them to the window.
 */
void
event_proc(window, event)
Xv_Window window;
Event    *event;
{
    if (event_is_ascii(event))
        sprintf(kbd_msg, "Keyboard: key ’%c’ %d pressed at %d,%d",
                event_action(event), event_action(event),
                event_x(event), event_y(event));
    else
        switch (event_action(event)) {
            case KBD_USE:
                sprintf(kbd_msg, "Keyboard: got keyboard focus");
                break;
            case KBD_DONE:
                sprintf(kbd_msg, "Keyboard: lost keyboard focus");
                break;
            case LOC_MOVE:
                sprintf(ptr_msg, "Pointer: moved to %d,%d",
                    event_x(event), event_y(event));
                break;
            case LOC_DRAG:
                sprintf(ptr_msg, "Pointer: dragged to %d,%d",
                    event_x(event), event_y(event));
                break;
            case LOC_WINENTER:
                sprintf(ptr_msg, "Pointer: entered window at %d,%d",
                    event_x(event), event_y(event));
                break;
            case LOC_WINEXIT:
                sprintf(ptr_msg, "Pointer: exited window at %d,%d",
                    event_x(event), event_y(event));
                break;
            case ACTION_SELECT:
            case MS_LEFT:
```

Example 5-3. The canvas_event.c program (continued)

```
                sprintf(but_msg, "Button: Select (Left) at %d,%d",
                    event_x(event), event_y(event));
                break;
            case ACTION_ADJUST:
            case MS_MIDDLE:
                sprintf(but_msg, "Button: Adjust (Middle) at %d,%d",
                    event_x(event), event_y(event));
                break;
            case ACTION_MENU:
            case MS_RIGHT:
                sprintf(but_msg, "Button: Menu (Right) at %d,%d",
                    event_x(event), event_y(event));
                break;
            default:
                return;
        }

    /* call repaint proc directly to update messages */
    repaint_proc((Canvas)NULL, window,
        (Display *)xv_get(window, XV_DISPLAY),
        xv_get(window, XV_XID), (Xv_xrectlist *) NULL);
}

/*
 * repaint_proc()
 *      Called to repaint the canvas in response to damage events
 *      and the initial painting of the canvas window.
 *      Displays the keyboard, pointer and button message strings
 *      after erasing the previous messages.
 */
void
repaint_proc(canvas, paint_window, dpy, xwin, xrects)
Canvas        canvas;           /* Ignored */
Xv_Window     paint_window;     /* Ignored */
Display      *dpy;
Window        xwin;
Xv_xrectlist *xrects;           /* Ignored */
{
    GC gc = DefaultGC(dpy, DefaultScreen(dpy));

    XClearWindow(dpy, xwin);
    XDrawString(dpy, xwin, gc, 25, 25, kbd_msg, strlen(kbd_msg));
    XDrawString(dpy, xwin, gc, 25, 50, ptr_msg, strlen(ptr_msg));
    XDrawString(dpy, xwin, gc, 25, 75, but_msg, strlen(but_msg));
}
```

The result produced by Example 5-3 is shown in Figure 5-6.



```
Keyboard: got keyboard focus
Pointer: exited window at 204,-4
Button: Adjust (Middle) at 77,23
```

Figure 5-6. A window created with canvas_event.c

5.8 Canvas and Openwin Package Summaries

Table 5-1 shows the attributes for the `CANVAS` package and Table 5-2 shows the `OPENWIN` attributes. The macros for these packages are shown below. This information is described fully in the *XView Reference Manual*.

```
CANVAS_EACH_PAINT_WINDOW()
CANVAS_END_EACH
OPENWIN_EACH_VIEW()
OPENWIN_END_EACH()
```

Table 5-1. Canvas Attributes

CANVAS_AUTO_EXPAND	CANVAS_PAINTWINDOW_ATTRS
CANVAS_AUTO_SHRINK	CANVAS_REPAINT_PROC
CANVAS_FIXED_IMAGE	CANVAS_RESIZE_PROC
CANVAS_HEIGHT	CANVAS_RETAINED
CANVAS_MIN_PAINT_HEIGHT	CANVAS_VIEW_CANVAS_WINDOW
CANVAS_MIN_PAINT_WIDTH	CANVAS_VIEW_PAINT_WINDOW
CANVAS_NO_CLIPPING	CANVAS_VIEWABLE_RECT
CANVAS_NTH_PAINT_WINDOW	CANVAS_WIDTH
CANVAS_PAINT_CANVAS_WINDOW	CANVAS_X_PAINT_WINDOW
OPENWIN_VIEWCLASS	

Table 5-2. Openwin Attributes

OPENWIN_ADJUST_FOR_HORIZONTAL_SCROLLBAR	OPENWIN_SPLIT_DESTROY_PROC
OPENWIN_ADJUST_FOR_VERTICAL_SCROLLBAR	OPENWIN_SPLIT_DIRECTION
OPENWIN_AUTO_CLEAR	OPENWIN_SPLIT_INIT_PROC
OPENWIN_HORIZONTAL_SCROLLBAR	OPENWIN_SPLIT_POSITION
OPENWIN_NO_MARGIN	OPENWIN_SPLIT_VIEW
OPENWIN_NTH_VIEW	OPENWIN_SPLIT_VIEW_START
OPENWIN_NVIEWS	OPENWIN_VERTICAL_SCROLLBAR
OPENWIN_SHOW_BORDERS	OPENWIN_VIEW_ATTRS
OPENWIN_SPLIT	
WIN_COLUMNS	WIN_ROWS

6
Handling Input

In discussing canvases in the previous chapter, we showed how to do some basic event handling in the canvas's paint window. This chapter goes into detail about the content of such events, the breakdown of the `Event` data structure that describes the event, the different types of input events that can be handled, and the specifics about what gets passed to the program's event handling callback routine. Chapter 20, *The Notifier*, should be reviewed for an in-depth discussion of how to handle events on a somewhat more advanced level. It addresses the event types themselves, how to register which events you want to be notified of, and how to interpret the events you receive.

This chapter also describes the interface to the special keyboard and mouse features including soft function keys, virtual keyboards, mouseless model keyboard mappings, and accelerator keys. These features support OPEN LOOK Level 2 functions that permit you to use multiple "virtual keyboards," and software-supported function keys. You also have the option of using the keyboard as a locator device along with or in place of a mouse.

This chapter discusses the following:

- The design of event handling.
- The breakdown of the `Event` data structure.
- Registering an event handler and the events you are interested in.
- Interpreting the events your event handler received.
- Sending messages to other windows or clients.
- Explicit reading of events from the server.
- Using the soft function keys.
- Virtual keyboards.
- The mouseless model.
- Using accelerators.

6.1 Introduction to Events in XView

Events are generated from several sources, including standard devices such as the keyboard and mouse, special input devices such as graphics tablets, and the window system itself. XView does not directly receive events from the hardware devices; the X server is responsible for managing these events and communicating them to the XView application. The Notifier receives all X events on behalf of the client application and dispatches them to the appropriate callback procedures registered by XView objects.

Because the Notifier multiplexes the input stream between windows, each individual window operates under the illusion that it has the user's full attention. That is, it sees precisely those input events that the user has directed to it. Each window indicates which events it is prepared to handle using *input masks*. An event callback procedure processes the events when they occur. Many windows have default event handlers that are installed internally by certain XView packages. Panels, for example, have a default event notification procedure that is used to identify which panel item received the input. Text subwindows capture events to allow typing and monitor text selection generated by using the mouse. However, some events may result in separate callback procedures being notified as well—repaint and resize events, for example, may have separate callback routines installed.

Applications can send messages to separate windows within the same application or to other applications running under separate processes. The X function `XSendEvent()` sends client-specified X events to other windows. `XClientMessageEvent` is one such event that can be sent. XView provides the function `xv_send_message()` as an interface for sending client messages in this manner.

The X Window System has a full set of events that can be sent to client applications by the server. It is highly recommended that you review Chapter 8, *Events*, in Volume One, *Xlib Programming Manual*, for specifics on the nature of these events.

6.2 Classes of Events

Events are grouped into the following classes:

- Semantic Events
- ASCII Events
- Locator Button Events
- Locator Motion Events
- Function Key Events
- Repaint and Resize Events
- Client Messages
- Selection Events

Each of these event types are discussed in Section 6.3, "Registering Events," and Section 6.7, "Interpreting Client Messages." There are separate issues to consider in both cases. However, *semantic* event codes are common to both. We will first address event IDs and semantic events and then move directly on to event registration. After we discuss how to register the events you are interested in, we will discuss how your event handling routine can interpret the events it receives.

6.2.0.1 Event IDs

Event IDs are integer values that have been assigned somewhat, but not completely, arbitrarily to events that may be generated in X.* Some event IDs represent keyboard events, function key events, window resize and repaint events, and so on. Although both X and XView understand the same events, there is no correlation between the event ID and Xlib's event codes. Thus, keyboard event IDs are *not* key symbols as defined by X's `KeySym` or `XKeyEvent.keycode` fields.

Event IDs are helpful when debugging applications; you can stop in event-handling routines and print the value of a particular event ID to identify the specific event type. One helpful hint is that the ASCII event IDs correspond to the ASCII codes generated. That is, if the user presses the *x* key, then the event ID is *x*.

6.2.0.2 Semantic events

Also called *action* events, semantic events are used to describe the *meaning* of an event using semantic phrases, so to speak, rather than using the literal event code represented by an event *ID*. That is, the definition of the *macro* describes the value of the event as well as its meaning. When someone presses a key marked HELP on the keyboard, the event ID may represent a large, cryptic numeral such as 65034. However, the semantic code associated with the key event would be `ACTION_HELP`.

There are also cases where the keyboard has been remapped by some applications for other purposes. For example, in OPEN LOOK, the SELECT button defaults to the leftmost physical mouse button, also known as button **one**. Normally this button is activated using the index finger for a right-handed person. A left-handed person who places the mouse on the left may remap the semantics of the leftmost and rightmost buttons (assuming a multibutton mouse). This allows the left-handed user to also use his index finger to activate the SELECT button.

We recommend that you use semantic actions when referring to events for consistency with other applications as well as consistency with different computers. Section 5, in the *XView Reference Manual*, provides a list of the semantic events. These events are defined in *<xview/win_input.h>*.

*This concept is carried over from SunView and is foreign to X.

6.3 Registering Events

Typically, when you specify which events a particular window is interested in, you also specify an event handler for that window. You register an event handler with an XView *window*, not the object with which the window is associated. Sometimes the object and the window are one and the same (i.e., panels), but in other cases, they are not. Specifically, to register an event handler for canvases you use the *paint window* as in:

```
Xv_Window window;
Canvas    canvas;
int       win;

window = (Xv_Window)xv_get(canvas, CANVAS_NTH_PAINT_WINDOW, win);
```

For text subwindows, you use the *view window*:

```
Xv_Window window;
Textsw    textsw;
int       win_no;

window = (Xv_Window)xv_get(textsw, OPENWIN_NTH_VIEW, win_no);
```

Once you have obtained the window, you can install the event handler using the `WIN_EVENT_PROC` attribute:

```
xv_set(window,
    WIN_EVENT_PROC,     sample_event_proc,
    WIN_CONSUME_EVENTS, WIN_ASCII_EVENTS, WIN_MOUSE_BUTTONS, NULL,
    NULL);
```

The window is of type `Xv_Window`. The event handler is a function that is called whenever any of the registered events occur in the specified windows. Details about this routine and how to interpret the events delivered to it are discussed later in this chapter, starting with Section 6.4, "The Event Handler."

This section discusses specifically how to register and unregister the events in which you are interested for a particular window. There are several ways to describe events you wish to register. You can specify action (semantic) events, literal events or event *classes* defined by XView (as shown in the example), or you can use X event masks (familiar to Xlib programmers). Which method you choose for registering events does not affect the function of the event handler; it maintains its function of receiving and understanding events.

6.3.1 Specifying X Event Masks

The simplest and most direct method for people familiar with Xlib programming is to use X event masks.* To register or unregister events using X masks, you can use:

```
WIN_CONSUME_X_EVENT_MASK
WIN_IGNORE_X_EVENT_MASK
```

*See Volume One, *Xlib Programming Manual*, for a complete discussion of event masks and their implications.

The value for these attributes is a mask made up of any of the event masks defined in *<X11/X.h>*. Do not confuse event masks with actual event types.

WIN_CONSUME_X_EVENT_MASK appends the specified event mask to the existing event mask for the object. However, to set the event mask explicitly to the specified mask, use WIN_EVENT_MASK. To clear the event mask completely, use:

```
xv_set(window, WIN_X_EVENT_MASK, NoEventMask, NULL);
```

The following code segment demonstrates how a canvas would register interest in the keyboard and mouse buttons:

```
xv_set(window,
    WIN_CONSUME_X_EVENT_MASK, ButtonPressMask | KeyPressMask,
    NULL);
```

Notice that we did not specify ButtonReleaseMask or KeyReleaseMask. Thus, the event handler is going to receive the down-events only.

When you specify event masks with the attribute WIN_IGNORE_X_EVENT_MASK, then those events are not delivered to your event handler. You cannot use this attribute to unregister all events and expect that to unregister your event handler. You must set the WIN_EVENT_PROC to NULL to do that. It is perfectly legal to have no events registered with a window.

6.3.2 Specifying XView Events

XView events (or event *types*) are simply alternate ways to specify events when you register them or interpret them. When using these event types to register events, you are not adding any more functionality than using the Xlib event registration scheme in the previous section. However, XView event types may make it easier to identify more precisely which events you are interested in being notified of.

The header files *<xview/win_input.h>* and *<xview/win_event.h>* have several data types, and most XView event definitions allow you to register events with windows or the Notifier. There are two methods available to do this: using the Inputmask data structure or specifying XView event codes directly.

Although the Inputmask method tends to be less elegant than using XView event types, it is necessary in order to use xv_input_readevent() and other advanced-usage functions. See Section 6.8, "Reading Input Directly," for details about how the Inputmask structure is used.

The attribute WIN_CONSUME_EVENTS is used to register events via XView event types. The value for this attribute is a NULL-terminated list of the XView events defined in the header files mentioned above. Let's re-examine the example used in the previous section:

```
xv_set(window,
    WIN_EVENT_PROC,     sample_event_proc,
    WIN_CONSUME_EVENTS, WIN_ASCII_EVENTS, WIN_MOUSE_BUTTONS, NULL,
    NULL);
```

The values WIN_ASCII_EVENTS and WIN_MOUSE_BUTTONS encompass all the ASCII codes from 0 to 127, inclusive, and the mouse button events. The events specified are added to the existing input mask for the window. This does not override what the window had previously; the event mask specified is *appended* to the input mask for that window. To set an explicit event mask, the value `WIN_NO_EVENTS` should be specified first in the list.

```
    xv_set(window,
        WIN_EVENT_PROC,     sample_event_proc,
        WIN_CONSUME_EVENTS,
            WIN_NO_EVENTS, WIN_ASCII_EVENTS, WIN_MOUSE_BUTTONS, NULL,
        NULL);
```

All events are cleared at the point in which `WIN_NO_EVENTS` is given in the list. If it is given in the middle of the list, then the events specified previous to that point are forgotten. If it is the only attribute in the list, then the event mask for the window is cleared. Note that this does not mean that your event handler will receive no events. There are certain events that are sent to your window, and thus to your event handler, whether you want them or not. This is addressed later in this chapter.

You can specify which events to ignore in the same way:

```
    xv_set(window,
        WIN_IGNORE_EVENTS,
            WIN_UP_ASCII_EVENTS, LOC_WINENTER, LOC_WINEXIT,
            NULL,
        NULL);
```

Here we are telling the window to ignore the events caused by *releasing* the ASCII keys on the keyboard as well as window-enter and window-exit events. As with `WIN_X_EVENT_MASK`, you cannot use `WIN_IGNORE_EVENTS` to unregister all the events and expect the event handler to be unregistered. It is perfectly legal to have a window that has no events registered with it.

While these attributes take a `NULL`-terminated list, you can use `WIN_CONSUME_EVENT` or `WIN_IGNORE_EVENT` to consume or ignore *one* event. Using these attributes, it is not necessary to specify a list of events.

6.3.2.1 Mouse events

The mouse (or *locator*) resides at an *x,y* coordinate position in pixels; this position is transformed by XView to the coordinate space of the window receiving an event. You can request mouse motion events by specifying `LOC_MOVE` or `LOC_DRAG`. A `LOC_MOVE` event is reported when the mouse moves, regardless of the state of the locator buttons. If you only want to know about locator motion when a button is down, then enable `LOC_DRAG` instead of `LOC_MOVE`. This will greatly reduce the number of motion events that your application has to process. If you have both specified, you will only receive one event or the other; you will never receive one followed by the other.

Even if you do not request move or drag events, you may still monitor when the mouse moves in and out of windows by specifying `LOC_WINENTER` and `LOC_WINEXIT`. In the case of `LOC_WINENTER`, the window installs its colormap into the server, and, for `LOC_WINEXIT`, the window's colormap is uninstalled. If you have registered the colormap notify event (`WIN_COLORMAP_NOTIFY`), then you will receive the appropriate events when you enter and leave a window.

Each button that is associated with the mouse is assigned an event code; the *i-th* button is assigned the code BUT(i). Thus, the event codes MS_LEFT, MS_MIDDLE, and MS_RIGHT correspond to BUT(1), BUT(2) and BUT(3). These are actual key codes, not semantic codes. You can specify setting the buttons explicitly (as shown here) or as a group (WIN_MOUSE_BUTTONS) or as semantic events using ACTION_SELECT, ACTION_ADJUST, and ACTION_MENU.*

If you want your applications to work seamlessly with the mouseless model, you should specify semantic events. Refer to Section 6.13, *The Mouseless Model*, for details on using the keyboard as a locator device.

6.3.2.2 Keyboard events

In order to be notified of keyboard events, you must specify any one of several XView event types depending on which event you want. Unlike the X event mask KeyPressMask, which generates events for all keys on the keyboard, including function keys, XView allows you to be more specific about which keyboard events you are interested in while not having to specify explicit keys.

The following list contains input event descriptors that may be used:

WIN_ASCII_EVENTS
: Enable ASCII keycodes—these are events that fall between 0 and 127 in the ASCII character set. Using this attribute specifies up-events as well as down-events.

WIN_UP_ASCII_EVENTS
: This is used mostly by WIN_IGNORE_EVENTS to turn off receiving the release event that usually follows a key press.

WIN_UP_EVENTS
: This is a general facility for ignoring all release events from mouse buttons and keyboard keys.

The XView event types KBD_USE and KBD_DONE can be specified to notify you when your window obtains the keyboard focus. OPEN LOOK specifies the click-to-type method for keyboard focus, so you may not get keyboard focus just because the mouse entered a window. The user can specify how keyboard focus should work without letting the application know about it. When a window gets keyboard focus and the window's event mask has KBD_USE set, then the window's event procedure is called.

*If you are writing applications and you use ACTION_SELECT, ACTION_ADJUST, and ACTION_MENU, users can easily adjust for a left handed mouse with the command xmodmap.

6.3.2.3 Resize and repaint events

When the size of a window is changed or the window is moved, (either by the user or programmatically), a WIN_RESIZE event is generated to give the client a chance to adjust any relevant internal state to the new window size. You should *not* repaint the window when receiving a resize event. You will receive a separate WIN_REPAINT event when a portion of the window needs to be repainted.

Top level frames and any other top level windows, when moved, may get multiple resize events, from the server and from the window manager. ICCCM mandates that the window manager send these events when the top level window is moved or resized. You can detect this with the test:

```
event_xevent(event)->xconfigure.send_event
```

which returns TRUE on events generated by the window manager. Note that all synthetic events delivered will follow real events. For more information on the event actions mandated by ICCCM and of the coordinate space mapping, refer to Section L.4.1.5 of *Inter-Client Communications Manual* in Volume 0, *X Protocol Reference Manual*.

If you are using a canvas subwindow, you will not need to track resize and repaint events directly. The CANVAS package receives these events, computes the new window dimensions or the precise area requiring repainting, and calls your resize or repaint procedures directly. See Chapter 5, *Canvases and Openwin*, for more details.

NOTE

> You will always get WIN_RESIZE events sent to your event handler routine. Currently, you cannot prevent these events from being delivered to your event handler.

As pointed out in Chapter 5, *Canvases and Openwin*, there may be a WIN_REPAINT event generated for each region of a window that gets exposed. However, you can set whether or not all those exposure events are collapsed into one expose event specifying a region that covers the entire exposed area. The attribute WIN_COLLAPSE_EXPOSURES can be set to TRUE (the default) or FALSE on the paint window to prevent multiple expose events.

You get graphics exposure events (WIN_GRAPHICS_EXPOSE) whenever you draw into a window using a GC whose graphics_exposures field is set to True. The same is true for WIN_NO_EVENTS. These events are not selected via WIN_CONSUME_EVENTS and cannot be ignored using this method. The only way to avoid receiving these events is by setting the graphics_exposures field in the GC to False. Choose the best way to deal with it for your application.

6.3.2.4 Client messages

Client messages are events that cannot be ignored by your event handler. Typically, these are messages that are used to implement a predefined protocol between your application and other applications that are familiar with the protocol. Because client messages cannot be ignored, we do not address the issue of client messages in this section. Read Section 6.7, “Interpreting Client Messages,” for more information on how to interpret these events.

6.3.2.5 Miscellaneous events

Consuming any one of the following events causes all of them to be consumed, as specified by the X Protocol:

- `WIN_CIRCULATE_NOTIFY`
- `WIN_DESTROY_NOTIFY`
- `WIN_GRAVITY_NOTIFY`
- `WIN_MAP_NOTIFY`
- `WIN_REPARENT_NOTIFY`
- `WIN_RESIZE`
- `WIN_UNMAP_NOTIFY`

Alternatively, you could just select `WIN_STRUCTURE_NOTIFY`, which selects all of the above events.

If you select `WIN_CREATE_NOTIFY` on a parent object, you will receive this event whenever a child window of the parent object is created. You will also receive any of the above listed events on the subwindows whenever `WIN_CREATE_NOTIFY`, or, alternatively, `WIN_SUBSTRUCTURE_NOTIFY`, is consumed on the parent.

Consuming any one of the following events causes all of them to be consumed, as specified in the X Protocol:

- `WIN_CIRCULATE_REQUEST`
- `WIN_CONFIGURE_REQUEST`
- `WIN_MAP_REQUEST`

Alternatively, you could just select `WIN_SUBSTRUCTURE_REDIRECT`, which selects all of the above events. A window manager is really the only client that should ever be interested in any of these.

6.4 The Event Handler

When one of the events in which you have expressed interest occurs, your event handler is called. The form of this routine is:

```
void
sample_event_proc(window, event, arg)
    Xv_Window   window;
    Event       *event;
    Notify_arg  arg;
```

The arguments to the routine are the window the event occurred in, a pointer to a data structure describing information about the event itself, and an optional argument supplied by the XView package responsible for the function being called.*

The attribute `WIN_EVENT_PROC` allows you to specify an event procedure for an application's window. Events are dispatched so event procedures that you register are the last in line to receive events. First, events are sent to the *base event handler* for the package, and then to the event handler you specify by setting `WIN_EVENT_PROC` or other callbacks for the individual packages. XView has a two-tiered scheme in which the packages —panels, canvases, scrollbars, etc.—interact with the Notifier directly, registering their own callbacks. Your application, in turn, registers its own callback procedures with the package. You can also interpose an event handling routine so that your application can intercept events before they reach the base event handler. See Chapter 20, *The Notifier*, for a description of interposition.

6.5 The Event Structure

Events that are generated are passed to event handling procedures by the Notifier as `Event` pointers (type `Event *`). This structure is declared in <*xview/win_input.h*>:

```
typedef struct  inputevent {
    short           ie_code;          /* input code */
    short           ie_flags;
    short           ie_shiftmask;     /* input code shift state */
    short           ie_locx, ie_locy; /* mouse position */
    struct timeval  ie_time;          /* time of event */
    short           action;           /* keymapped ie_code */
    Xv_object       ie_win;           /* window receiving event */
    char           *ie_string;        /* keycode binding string */
    XEvent         *ie_xevent;        /* actual XEvent struct */
} Event;
```

The `Event` data structure contains all the information about the event. The fields are broken down as shown in Table 6-1.

*This parameter is currently unused. It is available for new XView packages, extensions to them or for advanced Notifier usage. See Section 20.6.2, "Posting with an Argument," in Chapter 20, *The Notifier*.

Table 6-1. Event Structure Fields

Field	Contents
`ie_code`	Actual XView event ID, as defined in <*xview/win_event.h*> and <*xview/win_input.h*>.* Event codes can take on any value in the range 0 through 65535. The values are useful when debugging.
`ie_flags`	Indicates whether the event was an up- or down-event, if applicable. A down-event occurs when a mouse button or keyboard key goes down. There must be a corresponding up-event, although the client may choose to ignore up-events.
`ie_shiftmask`	If a Shift key, Control key and/or mouse button was down when the event occurred, this mask will have the appropriate bits set.
`ie_locx, ie_locy`	*x,y* coordinates of the position of the locator (mouse) relative to the window in which the event occurred.
`ie_time`	The time of the event.
`action`	Semantic code representing predefined actions specific to the window manager or OPEN LOOK.
`ie_win`	Window in which the event took place.
`ie_string`	String in which a keycode (found in `ie_code`) is bound using `XRebindKeysym()`.
`ie_xevent`	The actual event structure generated by X. This event structure arrives untouched by XView for events generated by the server.

6.6 Determining the Event

In the `Event` structure, there is a pointer to the `XEvent` structure that was delivered by the X server as a direct result of the event that it describes. This section discusses how to interpret the event based on information in the `Event` structure only; it does not address the `XEvent` structure.

The header files <*xview/win_input.h*> and <*xview/win_event.h*> contain many macros that should be used rather than referencing fields in the `Event` data structure. If the structure is modified in the future, then the macros will be modified to support the changes and your code will not have to change. For example, to get the window in which the event took place, you should not use the following (assume *event* is of type `Event *`):

```
window = event->ie_win;
```

Instead you should use:

```
window = event_window(event);
```

*<*xview/panel.h*> also has a few event definitions, but they are not widely used. See Chapter 7, *Panels*, for details.

To determine the actual event ID, you could use:

```
event_id(event)
```

This macro returns the actual event ID that took place, such as `MS_LEFT` to indicate the left mouse button. However, as discussed earlier, we recommend that you use the semantic action events provided by the macro:

```
event_action(event)
```

In the case where the user selected the left mouse button, `event_action()` would return `ACTION_SELECT`. The two values do not map to the same thing; consider the case where left-handed users have re-mapped the mouse settings. On the other hand, if there is no action associated with an event, `event_action()` is set to `event_id()`. For example, consider what happens when the letter *a* is pressed or when the `Expose` event is generated.

6.6.0.1 Event states

When a mouse button or keyboard event occurs, the event may be the result of a button or key being released or pressed. The way to determine this state from a particular event is to use one of these two macros:

```
event_is_up(event)
event_is_down(event)
```

6.6.0.2 Modifier keys

Modifier keys include the left and right Shift keys, the Control key, and the Meta key. The locations of these keys on the user's keyboard are dependent on the make and model of your keyboard. The functions of modifier keys are to modify particular keyboard or mouse states. For example, the Shift key, when modifying the *a* key, results in the *A* key.

Unless you have explicitly requested to be notified of *modifier* key events, you will not be informed when their state changes (e.g., when a user presses or releases one of the keys). You probably do not need to know this anyway. Instead, you only need to know the state of the key at the time you are evaluating another event. In this case, you can use any of the following macros:

```
event_shift_is_down(event)
event_ctrl_is_down(event)
event_meta_is_down(event)
```

6.6.1 Keyboard Events

When XView translates keyboard events into `Event` codes, it does translation of the key depending on the state of the modifier keys. For example, when the user types Shift-A, intending to type an uppercase *A*, then the event ID (that is, `event->ie_code`) is *A*. You do not need to use `event_shift_is_down` to translate the key to the uppercase.*

*This is in contrast to the value of `XKeyEvent.keycode` used by Xlib.

The following macro is used to determine if a key event is within the ISO character set:

```
event_is_iso(event)
    Event *event;
```

You can use the following macro to determine if an event is an ASCII key:*

```
event_is_ascii(event)
    Event *event;
```

This result does not tell you if you have a *printable* character. Depending on the font you are using, you may not be able to print anything with this event code. However, you can use any of the macros in <*ctype.h*> to determine whether the character is printable, a control character, a digit, a punctuation mark, and so on. Remember, this works because you are using the *already-translated* version of the event code.

The macro `event_string()` can be used to determine the string value associated with the event ID. This value is the result of a call to `XLookupString()`. The macro `event_string()` only returns a value if the string returned from XLookupString() is greater than one character long. Thus, for normal ASCII events, `event_string()` will be `NULL`. Only if the application programmer has rebound the key (using `XRebindKeysym()`) to a string greater than one character will `event_string()` report that string:

```
Display *dpy = (Display *)xv_get(frame, XV_DISPLAY);
char *newstring = "Nine";

XRebindKeysym(dpy, XK_9, 0, 0, newstring, strlen(newstring));
```

Here, the *9* key is rebound to generate the string "Nine" whenever it is pressed. The following macro determines whether or not a string is associated with the event:

```
event_is_string(event)
```

In the event callback, the following code could be used:

```
 . . .
if (event_is_string(event))
    printf("string = %s\n", event_string(event));
 . . .
```

Function keys differ from keyboard to keyboard, and the default key mappings for your server are configurable (at the time you build your server). However, XView has provisions for keyboards that are sectioned off into four sets of fifteen function keys: left, top, right, and

*In Version 3, `event_is_ascii(event)`, `event_is_iso(event)` and `event_is_meta(event)` use the XView semantic action to determine whether the event is ASCII, ISO or META, respectively. If the event is a modified ASCII or ISO key that maps to a semantic action (e.g., Meta-c mapping to `ACTION_COPY`), then the Version 3 macros will return `FALSE`. In Version 2, the event code was used, which would result in Meta-c returning `TRUE` regardless of the semantic action. Applications that want modified ASCII or ISO events (e.g., a terminal emulator) should examine the event code directly by using `event_id(event)`.

bottom keys. To determine which set of keys a particular event is associated with, you can use the following macros:

```
event_is_key_left(event)
event_is_key_right(event)
event_is_key_top(event)
event_is_key_bottom(event)
```

To determine which function set a particular function key belongs to, use:

```
KEY_TOP(key)
KEY_LEFT(key)
KEY_RIGHT(key)
KEY_BOTTOM(key)
```

Here, you do not pass the event, you pass the event *ID*. Thus, to test to see if a particular event were the fifth function key in the top row of keys, you would use:

```
if (event_is_key_top(event) && event_id(event) == KEY_TOP(5))
    /* process the fifth top-function key. */
```

Notice that we are using `event_id()` rather than `event_action()`. The reason for this is that some function keys are mapped to particular semantic actions, and we want to be sure the user hits the *fifth* function key on the top row. Had we used `event_action()` instead, the equality test may have failed.

Individual XView applications can define labels for the function keys. Refer to Section 6.12, "Soft Function Keys and Virtual Keyboards," for more information on labeling the function keys.

6.6.1.1 Mouse events

XView supports a locator device (typically a mouse) with up to ten buttons on it. XView also supports a mouseless locator, which operates through the keyboard and is described in Section 6.13, "The Mouseless Model." A mouse locator device may generate various types of events, such as:

Motion events	These events are generated whenever the mouse moves.
Drag events	These events are generated when any of the mouse buttons are down and the mouse moves.
Button events	These events are generated whenever the state of a mouse button changes (e.g., when a button goes up or down).

The following macros determine the state of particular buttons for a three-button mouse (the most common type).

```
event_is_button(event)
event_left_is_down(event)
event_middle_is_down(event)
event_right_is_down(event)
event_button_is_down(event)
```

Notice that none of these macros indicates an exclusive button state; that is, if `event_right_is_down` returns `TRUE`, that does not exclude the left button from being down, too.

You can determine which mouse button is changing state in the same way you determine a function key position, by passing the *ID* of the event to:

```
BUT(i)
```

To determine whether the second mouse button was down, you could use:

```
if (event_is_down(event) && event_id(event) == BUT(2))
    /* process button-2 event handling */
```

Again, we recommend that rather than determining the state of a particular mouse button, you use the semantic codes:

```
if (event_is_down(event) && event_action(event) == ACTION_ADJUST)
    /* process "adjust" event handling */
```

6.6.1.2 Keyboard focus

One way for the application to explicitly set keyboard focus to a particular window is to use `win_set_kbd_focus()`. The calling parameters are the window and the `XID` of the window that is supposed to get the focus. For example, if a window gets a `LOC_WINENTER` event, the keyboard focus can be obtained:

```
my_event_proc(window, event)
Xv_Window      window;
Event          *event;
{
   switch (event_id(event))  {
        ...
        case LOC_WINENTER:
            win_set_kbd_focus(window, xv_get(window, XV_XID));
            break;
        ...
   }
}
```

The function `win_set_kbd_focus()` generates a `KBD_USE` event for the window that is getting the focus and a `KBD_DONE` event for the window that lost focus. Windows that need to detect `KBD_DONE` and `KBD_USE` events must specify `KBD_DONE`/`KBD_USE` in their event masks.

The attribute `WIN_SET_FOCUS` behaves in the same way as `win_set_kbd_focus()` except that it checks to see if the window in question has `KBD_USE`/`KBD_DONE` selected. If it does not, the focus is not set on the window. Example usage of `WIN_SET_FOCUS` (it does not take any argument):

```
    xv_set(window, WIN_SET_FOCUS, NULL);
```

NOTE

> The X protocol restricts an unmapped window from holding the input focus.

Another way to grab keyboard focus for a window is to grab all the input for that window. This can be accomplished in various ways, including the use of the `FULLSCREEN` package described in Chapter 15, *Nonvisual Objects*. However, a much more convenient method is to use the `WIN_GRAB_ALL_INPUT` attribute. Setting this attribute to `TRUE` causes a grab that forces all input to be directed to that window. Setting it to `FALSE` releases the grab.

This is useful when you want to display a dialog box having panel items and confirmation or cancel buttons that the user must respond to before interacting with any other portion of the application. In this case, you should set the grab on the panel. Be sure to reset this attribute once it is no longer needed. Also, be careful when you code the segment of the program that uses the grab, or you might generate a grab you cannot get out of.

6.6.1.3 Selection events

Selection events may be delivered to a window's event procedure if the window owns a selection or is making selection requests. Generally, these events should be ignored if the application is using XView's `SELECTION` package. If the application is doing its own selections by calling Xlib functions directly, then these events will be of interest.

6.7 Interpreting Client Messages

Client messages may be delivered to your event handler for a variety of reasons. For instance, the event may have been sent by another source using `XSendEvent()` or `xv_send_message()`, which would imply that the sender of the message had loaded some arbitrary information into client-message format and sent it to you directly. In this case, the sender assumes you know how to interpret the information in the message.

6.7.1 Sending and Reading Client Messages

`xv_send_message()` is used to send client messages to other windows. The form for this function is:

```
Xv_private int
xv_send_message(window, addressee, msg_type, format, data, len)
    Xv_object       window;
    Xv_opaque       addressee;
    char           *msg_type;
    int             format;
    Xv_opaque      *data;
    int             len;
```

This function sends the message encoded in `data` to the `addressee` window. If the `addressee` parameter is an X window, then the message is sent to that window. Otherwise, the `addressee` may be either `PointerWindow` or `InputFocus` to correspond to the window under which the pointer happens to be lying or the window which happens to have the current focus. This depends on whether the user has click-to-type or focus-follows-mouse mode in the window manager.

The `window` parameter is an XView window/object from which the event is being sent. This is only used to extract the `Display *`. The `format` may be 8, 16, or 32. The value 8 is typically used to represent string values. `len` is the number of elements in the data. The size of one element is defined by the value of `format`.

The actual `XEvent` that is generated is `XClientMessageEvent`. When the Notifier detects the event, before passing it on to your event handler, it checks to see if the message content represents a drag and drop operation. If so, then the event action is set to the appropriate action. The `XEvent` structure's `ie_xevent` field remains unchanged.

If the client message is *not* a drag and drop operation, then you are responsible for deciphering the message. Clearly, this is something you have to be expecting, or there is no way to tell what to do with the information. Thus, you can create your own protocol between clients. You can determine the content of the message using `xv_get()` and the attributes:

```
WIN_MESSAGE_DATA
WIN_MESSAGE_TYPE
WIN_MESSAGE_FORMAT
```

Alternatively, you can obtain this information by accessing the data directly from the `XEvent` portion of the Event. These attributes map directly to the `XClientMessageEvent` data structure in *<X11/Xlib.h>*. Therefore, you could reference the appropriate fields in the `ie_xevent` pointer from the `Event` structure passed to the callback function. Using the type, format and data of the client message, you can read the message content. See Volume One, *Xlib Programming Manual*, for more information about unwrapping a client message.

6.8 Reading Input Directly

You can read input *immediately* using `xv_input_readevent()`. This function, which returns the window associated with the event read, takes the form:

```
Xv_object
xv_input_readevent(window, event, block, type, im)
    Xv_object        window;
    Event           *event;
    int              block, type;
    Inputmask       *im;
```

The `window` parameter identifies the window you want to read the events from. If `NULL`, `XNextEvent()` returns the window that received the next event. In this case, you should probably have a server grab for the window from which you are reading the event. Otherwise, you will have to propagate the received event to the appropriate window later.

The event parameter is a pointer to an Event type that is filled in when the function returns. The block parameter indicates whether or not the function should wait if there are no events pending to be read. If block is FALSE and there are no events, the function returns immediately without having read an event.

The type parameter tells whether to use the input mask already set in the window or whether to use the input mask specified by the im parameter. The Inputmask is declared in *<xview/win_input.h>*:

```
typedef   struct inputmask {
    short     im_flags;
    char      im_keycode[IM_MASKSIZE];
} Inputmask;
```

The structure consists of an input code array and flags that indicate which user actions belong in the input queue. To initialize the input mask im call the function bzero():

```
bzero((char *)&im, sizeof(im));
```

The following macros are used to manipulate XView event codes in an Inputmask:

```
win_setinputcodebit(im, code)
win_unsetinputcodebit(im, code)
```

Here, code is an XView code as described earlier. The flags field may be set to any of the following bits:

IM_NEGEVENT	Send input negative events (release or "up" events), too. This includes all keyboard keys and mouse buttons.
IM_ASCII	Enable ASCII codes 0 through 127—equivalent to WIN_ASCII_EVENTS.
IM_META	Enable the META codes 128-255—equivalent to WIN_META_EVENTS.
IM_NEGASCII	Enable release or "up" ASCII codes 0 through 127—this is more specific than IM_NEGEVENT above. It is primarily used to unset the code bits once ASCII bits have been set.
IM_NEGMETA	Enable release, or "up" META codes 128 through 255—used to unset these code bits.
IM_TOP	Enable TOP function keys.
IM_NEGTOP	Enable release events for TOP function keys.

With these macros, we set the Inputmask that is passed to xv_read_inputevent(). As it turns out, this method can also be used to set the input mask for regular windows. This is not the recommended method for providing the input mask, but it can be done by specifying the attribute WIN_INPUT_MASK:

```
Inputmask im;

win_setinputcodebit(im, WIN_MOUSE_BUTTONS);
win_setinputcodebit(im, WIN_ASCII_EVENTS);
im.im_flags &~ IN_NEGEVENTS;

xv_set(window, WIN_INPUT_MASK, &im, NULL);
```

Similarly, you can get the input mask in the same way:

```
Inputmask *im;

im = (Inputmask *)xv_get(window, WIN_INPUT_MASK);
```

6.9 Sample Program

This section provides a sample program that demonstrates most of what has been discussed in this chapter. In Example 6-1, the canvas window where the events occur may be split into several views. Each new view handles its own events and therefore handles its own graphic rendering into its paint window.

The intent is for the user to split the views several times and move the mouse between the views. Each view prints the events it receives in its own window at the upper-right corner. New views created from a split view may not be positioned correctly to see the text describing the events. It is not possible to scroll individual views programmatically to arbitrary locations, so the user must do so manually.

Pay careful attention to which window receives events so as to get a feeling for how the keyboard focus is handled. In some cases, the keyboard focus does not follow the mouse—a particular view may continue to receive keyboard focus even though the mouse is no longer in that subwindow. Usually, selecting the SELECT mouse button forces the focus to be directed to that view window.

Example 6-1. The canvas_input.c program

```
/*
 * canvas_input.c --
 * Display a canvas whose views may be split repeatedly.  The event
 * handler is installed for each view, so events are displayed in
 * each paint window.
 */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>
#include <xview/xv_xrect.h>

Canvas  canvas;
Frame   frame;
char    msg[128];
void    init_split(), my_event_proc(), my_repaint_proc();

main(argc,argv)
int     argc;
char    *argv[];
{
    /*
     * Initialize, create base frame (with footers) and create canvas.
     */
```

Example 6-1. The canvas_input.c program (continued)

```
    xv_init(XV_INIT_ARGS, argc, argv, NULL);
    frame = (Frame)xv_create(NULL,FRAME,
        FRAME_LABEL,            "Split View Windows.",
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);
    canvas = (Canvas)xv_create(frame,CANVAS,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        OPENWIN_SPLIT,
            OPENWIN_SPLIT_INIT_PROC,    init_split,
            NULL,
        CANVAS_REPAINT_PROC,    my_repaint_proc,
        NULL);

    (void) xv_create(canvas, SCROLLBAR,
        SCROLLBAR_SPLITTABLE,   TRUE,
        SCROLLBAR_DIRECTION,    SCROLLBAR_VERTICAL,
        NULL);
    (void) xv_create(canvas, SCROLLBAR,
        SCROLLBAR_SPLITTABLE,   TRUE,
        SCROLLBAR_DIRECTION,    SCROLLBAR_HORIZONTAL,
        NULL);

    /*
     *  Set input mask
     */
    xv_set(canvas_paint_window(canvas),
        WIN_CONSUME_EVENTS,
            WIN_NO_EVENTS,
            WIN_ASCII_EVENTS, KBD_USE, KBD_DONE,
            LOC_DRAG, LOC_WINENTER, LOC_WINEXIT, WIN_MOUSE_BUTTONS,
            NULL,
        WIN_EVENT_PROC, my_event_proc,
        NULL);

    xv_main_loop(frame);
}

/*
 * when a viewport is split, this routine is called.
 */
void
init_split(splitview, newview, pos)
Xv_Window splitview, newview;
int pos;
{
    Xv_Window   view;
    int         i = 0;

    /*
     * Determine view # from the new view and set its scrollbar to 0,0
     */
    OPENWIN_EACH_VIEW(canvas, view)
        if (view == splitview) {
            /* identify the view # of the view the user just split. */
            sprintf(msg, "Split view #%d", i+1);
            xv_set(frame, FRAME_LEFT_FOOTER, msg, NULL);
```

Example 6-1. The canvas_input.c program (continued)

```
        } else if (view == newview) {
            xv_set(xv_get(canvas, OPENWIN_VERTICAL_SCROLLBAR, view),
                SCROLLBAR_VIEW_START, 0,
                NULL);
            xv_set(xv_get(canvas, OPENWIN_HORIZONTAL_SCROLLBAR, view),
                SCROLLBAR_VIEW_START, 0,
                NULL);
        }
        i++;
    OPENWIN_END_EACH
    sprintf(msg, "Total views: %d", i);
    xv_set(frame, FRAME_RIGHT_FOOTER, msg, NULL);
}

/*
 * Called when an event is received in an arbitrary paint window.
 */
void
my_event_proc(window, event, arg)
Xv_Window       window;
Event           *event;
Notify_arg      arg;
{
    register char *p = msg;

    *p = 0;

    /* test to see if a function key has been hit */
    if (event_is_key_left(event))
        sprintf(p, "(L%d) ", event_id(event) - KEY_LEFTFIRST + 1);
    else if (event_is_key_top(event))
        sprintf(p, "(T%d) ", event_id(event) - KEY_TOPFIRST + 1);
    else if (event_is_key_right(event))
        sprintf(p, "(R%d) ", event_id(event) - KEY_RIGHTFIRST + 1);
    else if (event_id(event) == KEY_BOTTOMLEFT)
        strcpy(p, "bottom left ");
    else if (event_id(event) == KEY_BOTTOMRIGHT)
        strcpy(p, "bottom right ");
    p += strlen(p);


    if (event_is_ascii(event)) {
        /*
         * note that shift modifier is reflected in the event code by
         * virtue of the char printed is upper/lower case.
         */
        sprintf(p, "Keyboard: key '%c' (%d) %s at %d,%d",
            event_action(event), event_action(event),
            event_is_down(event)? "pressed" : "released",
            event_x(event), event_y(event));
    } else switch (event_action(event)) {
        case ACTION_CLOSE :
            xv_set(frame, FRAME_CLOSED, TRUE, NULL);
            break;
        case ACTION_OPEN :
            strcpy(p, "frame opened up");
```

Example 6-1. The canvas_input.c program (continued)

```
            break;
        case ACTION_HELP :
            strcpy(p, "Help (action ignored)");
            break;
        case ACTION_SELECT :
            sprintf(p, "Button: Select (Left) %s at %d,%d",
                event_is_down(event)? "pressed" : "released",
                event_x(event), event_y(event));
            break;
        case ACTION_ADJUST :
            sprintf(p, "Button: Adjust (Middle) %s at %d,%d",
                event_is_down(event)? "pressed" : "released",
                event_x(event), event_y(event));
            break;
        case ACTION_MENU :
            sprintf(p, "Button: Menu (Right) %s at %d,%d",
                event_is_down(event)? "pressed" : "released",
                event_x(event), event_y(event));
            break;
        case SHIFT_RIGHT :
            sprintf(p, "Keyboard: right shift %s",
                event_is_down(event)? "pressed" : "released");
            break;
        case SHIFT_LEFT :
            sprintf(p, "Keyboard: left shift %s",
                event_is_down(event)? "pressed" : "released");
            break;
        case SHIFT_LEFTCTRL : case SHIFT_RIGHTCTRL :
            sprintf(p, "Keyboard: control key %s",
                event_is_down(event)? "pressed" : "released");
            break;
        case SHIFT_META :
            sprintf(p, "Keyboard: meta key %s",
                event_is_down(event)? "pressed" : "released");
            break;
        case SHIFT_ALT :
            sprintf(p, "Keyboard: alt key %s",
                event_is_down(event)? "pressed" : "released");
            break;
        case KBD_USE:
            sprintf(p, "Keyboard: got keyboard focus");
            break;
        case KBD_DONE:
            sprintf(p, "Keyboard: lost keyboard focus");
            break;
        case LOC_MOVE:
            sprintf(p, "Pointer: moved to %d,%d",
                    event_x(event),event_y(event));
            break;
        case LOC_DRAG:
            sprintf(p, "Pointer: dragged to %d,%d",
                    event_x(event), event_y(event));
            break;
        case LOC_WINENTER:
            win_set_kbd_focus(window, xv_get(window, XV_XID));
            sprintf(p, "Pointer: entered window at %d,%d",
```

Example 6-1. The canvas_input.c program (continued)

```
                    event_x(event), event_y(event));
            break;
        case LOC_WINEXIT:
            sprintf(p, "Pointer: exited window at %d,%d",
                    event_x(event), event_y(event));
            break;
        case WIN_RESIZE :
        case WIN_REPAINT :
            return;
        default :
            /* There are too many ACTION events to trap -- ignore the
             * ones we're not interested in.
             */
            return;
    }

    my_repaint_proc(canvas, window,
        xv_get(canvas, XV_DISPLAY), xv_get(window, XV_XID), NULL);
}

/*
 * my_repaint_proc()
 *      Called to repaint the canvas in response to damage events
 *      and the initial painting of the canvas window.
 *      Displays the keyboard, pointer and button message strings
 *      after erasing the previous messages.
 */
void
my_repaint_proc(canvas, pw, dpy, xwin, xrects)
Canvas          canvas;
Xv_Window       pw;
Display         *dpy;
Window          xwin;
Xv_xrectlist    *xrects;
{
    char        win_num[16];
    Xv_Window   w;
    int         i = 0;
    GC          gc = DefaultGC(dpy, DefaultScreen(dpy));

    /*
     * Determine which paint window we're writing in.
     */
    CANVAS_EACH_PAINT_WINDOW(canvas, w)
        if (w == pw)
            break;
        i++;
    CANVAS_END_EACH
    sprintf(win_num, "(Window #%d) ", i+1);

    XClearWindow(dpy, xwin);
    XDrawString(dpy, xwin, gc, 25, 25, win_num, strlen(win_num));
    XDrawString(dpy, xwin, gc, 25, 45, msg, strlen(msg));
}
```

This sample program initializes XView and creates a frame. It then creates a canvas with two scrollbars attached to it. Chapter 5, *Canvases and Openwin*, addresses how to attach scrollbars to a canvas. The canvas installs a callback routine that will be called when its views are split. This routine installs the existing event masks and callback routine in the new view's paint window. Remember, that is necessary because new windows need to be initialized by the application.

The callback routine `my_event_proc()` handles all events for all the windows in the program. It determines which event has taken place and constructs a descriptive message identifying the event. It then calls the repaint routine to display the message in the window in which the event occurred. Note that the repaint callback routine may be called by the application—it is not a function reserved for the window system to call exclusively. In this case, the graphics is limited to calling `XDrawString()` to display text.

6.10 Extensions for Events

In X11, it is possible to create extensions to the server that may generate their own set of events, depending on the way your X11 server has been configured.* For example, in X11 Release 4, the *Shape* extension was added, allowing you to display windows of arbitrary shape in addition to the usual rectangular windows. Not all X11 servers support all known extensions, and there is a further limitation: all window managers may not be able to handle a given extension like the Shape extension. Therefore, this chapter is only intended for those who are well aware of how server extensions work and are using them in their applications.

The only thing that XView cares about with respect to extensions is the delivery of events that have an event type outside of the normal range defined by the X Protocol. In other words, events that are defined by the server extension. Thus, the attribute SERVER_EXTENSION_PROC is used to specify a function to be called when such an extension event occurs. Unlike other event handlers, you do not register an extension procedure with a window, you register it with the server object itself:

```
extern void proc();
xv_set(XV_SERVER_FROM_WINDOW(frame),
    SERVER_EXTENSION_PROC, proc,
    NULL);
```

Note the use of XV_SERVER_FROM_WINDOW. This macro returns the `Xv_Server` object associated with an XView object. The object can be any one that contains a window (so most panel items are excluded). In this case, it happens to be a frame. Be aware that if your application is using multiple servers, you should use an object associated with the server that contains the extension.

*Server extensions should not be confused with XView extensions discussed in Chapter 25, *XView Internals*.

The proc function is called whenever there is an event associated with the server extension. The form of the procedure is:

```
void
ext_event_proc(dpy, event, object)
    Display *dpy;
    XEvent  *event;
    Xv_object object;
```

The display and event types are strictly X11 types defined in *<X11/Xlib.h>*. The object is an XView object that is associated with the event, if available. If it is impossible to determine the object, the value is NULL.

6.11 Selecting Events on Other Clients

To select for and receive X events destined for windows that are not owned by the application, use the attributes SERVER_EXTERNAL_XEVENT_MASK and SERVER_EXTERNAL_XEVENT_PROC. The most common use for these attributes is for an application to select for PropertyNotify events on the root window in order to receive notification when a new RESOURCE_MANAGER property has been written by the user.

For each window, the client can specify an X mask representing the events it wants to receive. Additionally, an XView object handle can be provided as an argument which is later returned as a parameter in the event callback. The following example code demonstrates how to select for PropertyNotify and ButtonPress events on the root window:

```
xv_set (server,
          SERVER_EXTERNAL_XEVENT_MASK, RootWindow(dpy, 0),
                     ButtonPressMask | PropertyChangeMask,
                frame,
          SERVER_EXTERNAL_XEVENT_PROC, root_event_proc, frame,
          NULL);
```

The callback is defined as follows:

```
void
root_event_proc(server, display, xevent, xv_object)
     Xv_server    server;
     Display     *display;
     XEvent      *xevent;
     Xv_opaque    xv_object;
```

On xv_create() and xv_set(), SERVER_EXTERNAL_XEVENT_MASK takes three arguments: an XID of the window the client wants to select for events on, an X event mask, and an XView object handle. For xv_get(), SERVER_EXTERNAL_XEVENT_MASK should be passed two arguments: an XID of a window and an XView object handle. It will return the X event mask set on that XID.

On xv_create() and xv_set(), two arguements should be passed to SERVER_EXTERNAL_XEVENT_PROC: a ptr to a function to be used as the callback, and an XView object handle to associate the callback to. A client can register a separate callback for different

XView object handles. For `xv_get()`, only the XView object handle should be passed as an argument. It will return a ptr to the callback function.

`SERVER_EXTERNAL_XEVENT_PROC` and `SERVER_EXTERNAL_XEVENT_MASK` can be set in any order. Each attribute can be temporarily disabled then re-enabled later without having to set the other attribute. Setting the first argument of `SERVER_EXTERNAL_XEVENT_PROC` to `NULL` will disable the callback for the object specified as the second argument.

The callback set using `SERVER_EXTERNAL_XEVENT_PROC` may be called back for an X event that was not specified using `SERVER_EXTERNAL_XEVENT_MASK`. The reason being is that other objects within the toolkit or application may select for X events on the same window. When these events are delivered, it is difficult to map the X event back to X masks, in turn making it difficult to determine who selected them. Thus the toolkit lets the application determine if it wants to use the event or not.

6.12 Soft Function Keys and Virtual Keyboards

This section describes the *soft function keys* and *virtual keyboards*. OPEN LOOK encourages applications to use the function keys for tasks specific to the individual application. The soft function key labels are configurable for each application and provide application-specific labels for the function keys. Soft function keys allow “function keys” to be selected from an on-screen panel using the mouse. Thus, the number of function keys an application uses is not tied to a particular keyboard, since the on-screen window can display all the function keys. A pop-up window shows the function keys with labels configured specifically for the window that has the keyboard focus.

Virtual keyboards allow users to configure the keyboard to match any of the supported international keyboards. For detailed descriptions of the soft function keys and Virtual keyboards features, refer to Chapter 14 in the *OPEN LOOK GUI Functional Specification*.

6.12.1 Soft Function Keys

Provided an application does *not* use hard-wired function key bindings, the soft function keys feature provides a portable method for labeling, displaying and selecting an application’s function keys. To display the function keys, a user selects “Function Keys” from the Workspace menu under *Utilities*. The function keys window appears at the bottom of the screen as shown in Figure 6-1.

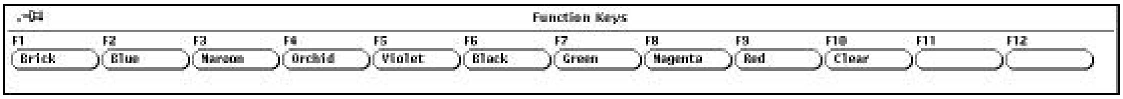


Figure 6-1. A sample function keys window

When the input focus changes to a window that uses the function keys, the function keys window is updated to reflect the new set of functions (only if the application is using the soft function keys). If an application uses more functions than a particular keyboard provides, it is the application's responsibility to provide a "More" label on one of the function keys. When the "More" key is selected, the application should show the additional labels for the remaining functions keys, as is shown in Figure 6-2.

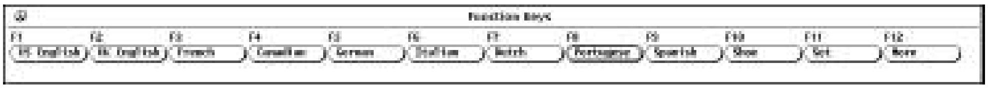


Figure 6-2. Sample function keys window with a MORE key

Labels for the soft function keys are specified using the `WIN_SOFT_FNKEY_LABELS` attribute. The following example shows how to set soft function keys for a canvas application:

```
canvas = (Canvas) xv_create (frame,CANVAS,
                    CANVAS_X_PAINT_WINDOW,TRUE,
                    NULL);

xv_set(canvas_paint_window(canvas),
        WIN_FNSOFT_KEY_LABELS,"Red\nGreen\nBlue\nMaroon\nOrchid\n
                    Violet\nMagenta\nCoral\nTurquoise\n
                    Yellow\nBrick\nBlack\n",
        WIN_EVENT_PROC, my_Event_proc,
        NULL);
```

The value for the `WIN_SOFT_FNKEY_LABELS` is a string of 12 labels, with each label separated by "\n." In this example, whenever the canvas gets the input focus, the soft function key labels are updated to "Red, Green, Blue . . . ."

Individual applications maintain the banks of labels for their soft function keys. The "More" key indicates to the application that it is necessary to update the `WIN_SOFT_FNKEY_LABELS` value. XView provides the method for labeling the soft function keys on the screen. The application needs to provide the functionality for each of the function keys whose labels it defines.

An application that uses more than 12 function keys should label the 12th function key "More." When the application receives notification that the 12th function key or the 12th function key button on the screen has been selected, then `WIN_SOFT_FNKEY_LABELS` should be reset to change the labels for the additional function keys. The new bank should also have a "More" key to toggle back to the first bank, or to the next bank of function keys if one is available. The application also needs to adjust its response to reflect the "new" function keys, corresponding to the new labels.

6.12.2 Virtual Keyboards

Virtual keyboards are an XView feature that allows users to reconfigure their keyboards. Keyboards can be logically configured to any of the international keyboards supported by OpenWindows.

6.12.2.1 Multiple language support

Pressing the Language key, R2 by default, presents the user with a choice of all the supported international keybindings (the key binding for the language key is defined by the `OpenWindows.KeyboardCommands.Translate` resource.) A set of soft function keys allows a user to select a Virtual keyboard. When a keyboard is selected from the function keys, the keybindings are displayed on-screen as is shown in Figure 6-3.

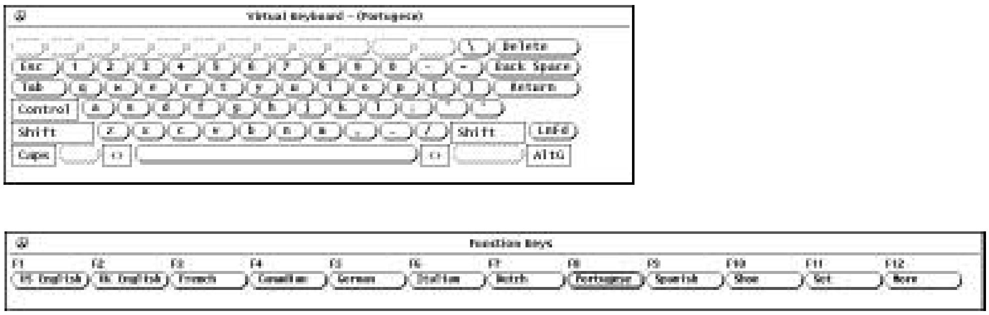


Figure 6-3. Sample virtual keyboard binding

Selecting a character set and the Set key from the soft function keys binds a character set to the physical keyboard. Holding down the Language key and typing temporarily binds the currently selected character set to the physical keyboard. When the Language key is released, the currently set character set is restored.

6.13 The Mouseless Model

The mouseless input model allows users to run applications on an OPEN LOOK desktop without using a mouse. Thus, XView gives two options for the locator device: first, the standard mouse or other locator device, and alternatively, the mouseless model, which lets the user navigate between and select objects on the desktop using keyboard commands.

This section covers the mouseless model implementation, including:

- Mouseless model semantic actions.
- Keyboard command resource binding.

The resource `OpenWindows.KeyboardCommands` controls the level of mouseless operation. This resource may have one of three values:

`SunView1`	Defines only those keyboard commands that were present using SunView1. This is the default setting and defines actions for basic operations such as CUT, COPY, and PASTE.
`Basic`	Enables the SunView1 commands, plus the mouseless model *Local Navigation* and *Text Editing* commands. Using this setting, only objects that normally take the input focus will accept input focus.
`Full`	Enables all mouseless model keyboard commands and depending on the application or window manager enables display of the special mouseless model *Location Cursor* which indicates the current pointer position. Using this setting, all objects that can normally be manipulated with the mouse can accept input focus. Thus, actions that are normally performed with the mouse and its buttons may be performed using the keyboard instead.

When the mouseless model is in use (`OpenWindows.KeyboardCommands` set to Full), applications are responsible for displaying the Location Cursor. The Location Cursor is described in detail in Section 6.13.4.

6.13.1 Keyboard Command Mapping

Each mouseless keyboard command is assigned an XView semantic action, each of which has an `ACTION_` prefix. Appendix C, *Mouseless Model Keyboard Mappings*, lists all of the mouseless semantic actions and the corresponding event ID for each semantic action. Depending on the value of the resource `OpenWindows.KeyboardCommands`, which determines the level of mouseless operation, some or all of these mouseless semantic events are handled internally by XView packages, by the window manager, or by individual application programs.

Most of the mouseless keyboard commands are mapped to a combination of a modifier key and a standard key. As described earlier in this chapter, the modifier keys include the left and right Shift keys, the Control key, and the Meta key. The Alt key is also a modifier key for the mouseless model (on some X servers, Alt is mapped to Meta). Sun Type-4 keyboard mapping, several mouseless keyboard commands map directly to unmodified ASCII characters. These unmodified commands are defined with the semantic action `ACTION_NULL_EVENT`. Table 6-2 shows the mouseless keyboard commands that are mapped to `ACTION_NULL_EVENT`. The contents of the specified XView variable contain the value of the specified resource. To process one of these commands, check to see if the `event_id` of the `Event` equals the value of one of the XView variables. If so, then the event will be translated to the corresponding `ACTION_` command.

Table 6-2. Mouseless Keyboard Commands with Action ACTION_NULL_EVENT

Command	Resource	Default Value	XView Variable
`CANCEL`	`keyboard.cancel`	`Escape`	`xv_iso_cancel`
`DEFAULT_ACTION`	`keyboard.defaultAction`	`Return`	`xv_iso_default_action`
`INPUT_FOCUS_HELP`	`keyboard.inputFocusHelp`	`?`	`xv_iso_input_focus_help`
`NEXT_ELEMENT`	`keyboard.nextElement`	`Tab`	`xv_iso_next_element`
`SELECT`	`keyboard.select`	`Space`	`xv_iso_select`

XView mouseless semantic action names are also mapped to provide SunView1 keyboard functions under XView. Most of the SunView1 navigation and editing commands do not conflict with the mouseless model. However, there are a few commands that *do* conflict. Table 6-3 shows the SunView1 commands that conflict with the mouseless model.

Table 6-3. SunView1 Commands That Conflict with the Mouseless Model

Key Combination	SunView1 Command	Mouseless Command	Resolution
`Ctrl-Tab`	`SELECT_FIELD_FORWARD`	`NEXT_ELEMENT`	`Moved`
`Shift-Ctrl-Tab`	`SELECT_FIELD_BACKWARD`	`PREVIOUS_ELEMENT`	`Moved`
`Shift-Ctrl-/`	`GO_WORD_BACKWARD`	`INPUT_FOCUS_HELP`	`Moved`
`Home`	`GO_DOCUMENT_START`	`LINE_START`	`Moved`
`End`	`GO_DOCUMENT_END`	`LINE_END`	`Moved`
`PgUp (R9)`	`GO_PAGE_BACKWARD`	`PANE_START`	`Dropped`
`PgDn (R15)`	`GO_PAGE_FORWARD`	`PANE_END`	`Dropped`
`Shift-Up`	`UP`	`SELECT_UP`	`Moved`
`Shift-Down`	`DOWN`	`SELECT_DOWN`	`Moved`
`Shift-Left`	`LEFT`	`SELECT_LEFT`	`Moved`
`Shift-Right`	`RIGHT`	`SELECT_RIGHT`	`Moved`

The values in the “Resolution” field have the following meaning:

Dropped	means that the specified SunView1 functionality is not available in XView.
Moved	means that the SunView1 functionality is available in XView, but a different key combination is used than in SunView1.

6.13.2 Mouseless Model Resources

Each Mouseless semantic action mapping and its corresponding key binding is determined by the value of a resource. The default key bindings are based on the Sun Type-4 keyboard. The bindings occur when applications are initialized and `xv_init()` reads the resource names. Appendix C, *Mouseless Model Keyboard Mappings*, lists all of the Mouseless resources, grouped according to the values loaded for the three different mouseless model modes: SunView1, Basic, and Full. Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual*, lists all XView resources alphabetically (all the mouseless resources begin with `OpenWindows.KeyboardCommand`.) Mouseless model resources use the following naming conventions:

`OpenWindows.KeyboardCommand.` *XViewSemanticAction*

XViewSemanticAction is the name of the XView semantic action without its `ACTION_` prefix (note that the capitalization for the action is also changed and the underscore characters "_" are deleted). For example:

```
OpenWindows.KeyboardCommand.JumpRight: period+Ctrl
```

corresponds to binding the semantic action `ACTION_JUMP_RIGHT` to period modified by the Control character.

Each resource may define several mappings for an individual action. Multiple mappings are separated by "," and have the following general format:

```
mapping[,mapping...]
```

The following resource definition is an example:

```
OpenWindows.KeyboardCommand.FindBackward: F+Meta,L9+Shift
```

Each Mouseless mapping is of the form:

KeysymName[+*Modifier* . . .]

In other words, each mapping is separated by a comma, and if the keysym is modified, then each modifier is separated by a plus sign. A modifier is either "Shift," "Ctrl," "Alt," or "Meta."

When an alphabetic character is the keysym, the case of the KeysymName is important. For uppercase characters, use the uppercase alphabetic keysyms, for example L, instead of the lowercase with a "Shift" modifier. When an alphabetic character is not modified by shift, then use the lowercase alphabetic keysym (for example, l+Meta). Do not list unmodified ASCII keyboard commands should not be listed.

6.13.3 Using the Mouseless Model

If the keyboard is used to control the locator, then the window manager, XView packages, and your XView applications handle the keyboard's mouseless semantic events. This section describes the roles of the window manager and the individual XView applications.

6.13.3.1 The role of the window manager

Using the mouseless model, the window manager intercepts the following actions:

- Actions that change the input focus between windows or to the workspace.
- Actions that accelerate window menu operations.
- Actions that toggle the pushpin.

These three categories include the actions shown in Table 6-4.

Table 6-4. Mouseless Actions Handled by the Window Manager

Back	Front
Open	Close
Dismiss	Window Properties
Toggle Pushpin	Go To Workspace
Refresh	Quit
Move	Resize
Full Size	Restore size
Next Window	Previous Window
Last Window	Next Application
Previous Application	Bring Up Window Menu
Jump Input Focus To Mouse	Jump To Workspace Background
Bring Up Workspace Menu?	

6.13.3.2 Application responsibilities

Applications are responsible for displaying the location cursor and for handling navigation in canvases (for example, non-text data panes). Interaction with the location cursor is described in the following section. The semantic action, `ACTION_PANE_BACKGROUND` (`JumpToPane-Background`), also needs to be handled by XView applications.

Applications can use the unmodified ASCII keyboard commands that are intercepted by the mouseless model by accessing the appropriate XView variables as shown in Table 6-2.

6.13.4 The Location Cursor

The *Location Cursor* is a borderless frame subwindow that indicates the locator position when the mouseless model is set to Basic or Full. When the location cursor is moved to a new pane and it is assigned the same colormap segment and background color as that pane. The location cursor is created when the frame is created.

Frame subwindows access the handle of the Location Cursor window with the following call:

```
xv_get(frame, FRAME_FOCUS_WIN)
```

Initially, the Location Cursor is created unmapped (XV_SHOW is FALSE). When a frame subwindow receives a KBD_USE, it calls the function frame_kbd_use(). frame_kbd_use() sets the FRAME_FOCUS_WIN's colormap segment (WIN_CMS) and background color (WIN_BACKGROUND_COLOR) to be the same as the subwindow. If the subwindow then needs to display the Location Cursor, it sets the WIN_PARENT for the FRAME_FOCUS_WIN to its paint window. It also needs to position the XV_X and XV_Y to the appropriate position and then set XV_SHOW to TRUE.

Note that text fields and panes have a caret to indicate input focus and position, so they would not display the Location Cursor window. The XV_X and XV_Y coordinates are relative to the subwindow's paint window. When a subwindow receives KBD_DONE, it sets the XV_SHOW to FALSE for the FRAME_FOCUS_WIN.

The Location Cursor either points up or to the right. This is controlled by setting FRAME_FOCUS_DIRECTION on the frame to either of the values FRAME_FOCUS_UP or FRAME_FOCUS_RIGHT.

On pop-up menus, it is the application's responsibility to pass a valid *x,y* coordinate in the event structure when the event is the keyboard MENU command, as in the following:

```
(event_action(event) == ACTION_MENU && !event_is_button(event))
```

This action is necessary because the ACTION_MENU could be the result of a key press, in which case the cursor could be anywhere on the screen.

6.13.5 Events

Since keyboard events have a dual role when using the mouseless model, applications must use semantic actions instead of event ID's to control their application. This is required since an ASCII event ID could also be a mouseless semantic action.

6.14 Using Accelerators

Accelerators permit applications to define notify procedures that are called for specific events. For example, a particular key combination might be used to quit the application. For Version 3.2, XView supports Menu accelerators. Refer to Appendix D of *Version 3.2 and the File Chooser* for details on using menu accelerators.

To implement window-level accelerators, use the attributes FRAME_ACCELERATOR and FRAME_X_ACCELERATOR. These attributes are set on the frame containing the window where the accelerator is defined. FRAME_ACCELERATOR associates an accelerator with an event code. FRAME_X_ACCELERATOR associates an accelerator with an X keysym.

The first value for the attribute FRAME_ACCELERATOR is the event code for an unmodified key (not shifted) that when combined with the Meta modifier, forms the key sequence for the accelerator. Similarly, the first value for the attribute FRAME_X_ACCELERATOR is an X keysym instead of an event code.

The second value for both FRAME_ACCELERATOR and FRAME_X_ACCELERATOR is the accelerator notify routine to call when the accelerator is used. This routine is a user-defined callback with the following form:

```
void
accelerator_notify_proc(value, event)
   Xv_opaque   value;
   Event       *event;
```

The parameter *value* is the value passed as the third argument to FRAME_ACCELERATOR or FRAME_X_ACCELERATOR. In this example, the value is an Xv_opaque that passed to the accelerator notify procedure. Usually this is an XView object handle. For example, if you want to accelerate a PANEL_BUTTON, if you set *value* to the button item handle, then you could set the accelerator notify procedure to be the same as the button's notify procedure as in the following example:

```
xv_set(frame,
        FRAME_ACCELERATOR, 'b', file_btn_notify_proc, file_button,
        NULL);
```

For accelerators to work, you need to set OpenWindows.KeyboardCommands to Full. For more details on setting this resource, refer to Section 6.13, "The Mouseless Model."

NOTE

Certain key combinations are reserved for semantic actions within XView and should not be used for accelerators. These key combinations are listed in Appendix C, *Mouseless Model Keyboard Mappings*.

This page intentionally left blank

to preserve original page counts.

7
Panels

The XView PANEL package implements the OPEN LOOK *control area*. Panels are used in many different contexts—property sheets, notices, and menus all use panels in their implementation. The main function of a panel is to manage a variety of *panel items*. Figure 7-3 shows examples of panel items from the *OPEN LOOK GUI*. Because some panel items may not contain windows that handle their own events, the PANEL package is responsible for propagating events to the appropriate panel item. This chapter addresses issues specific to panels, the management of panel items and the distribution of events to those items. We look at basic issues common to all panel items before introducing each of the eight different panel item packages. Finally, we look at several advanced topics regarding panel usage.

The PANEL package is subclassed from the WINDOW package. In typical usage, you create a panel and set certain panel-specific attributes. Figure 7-1 shows the class hierarchy for panels and Figure 7-2 shows the class hierarchy for a panel item.

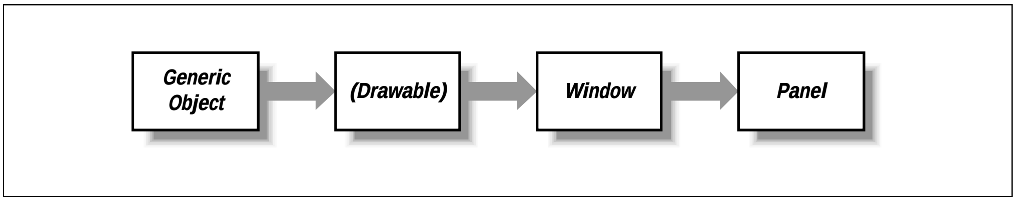


Figure 7-1. Panel package class hierarchy

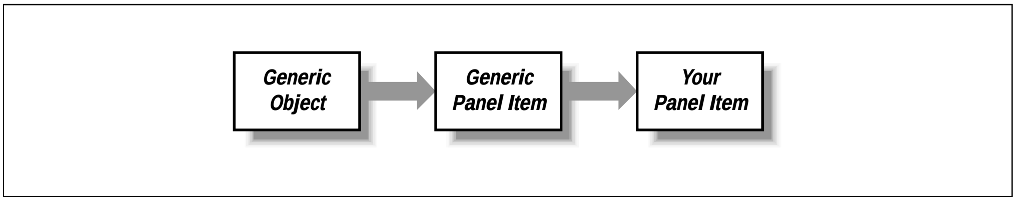


Figure 7-2. Panel item class hierarchy

Panels set up and manage event-handling masks and routines for themselves and their panel items. The application does not set event masks or install an event callback routine unless it needs to track events above and beyond what the PANEL package does by default (typical applications will not need to do this). Even so, this is probably better accomplished via interposing functions as discussed in Chapter 20, *The Notifier*.

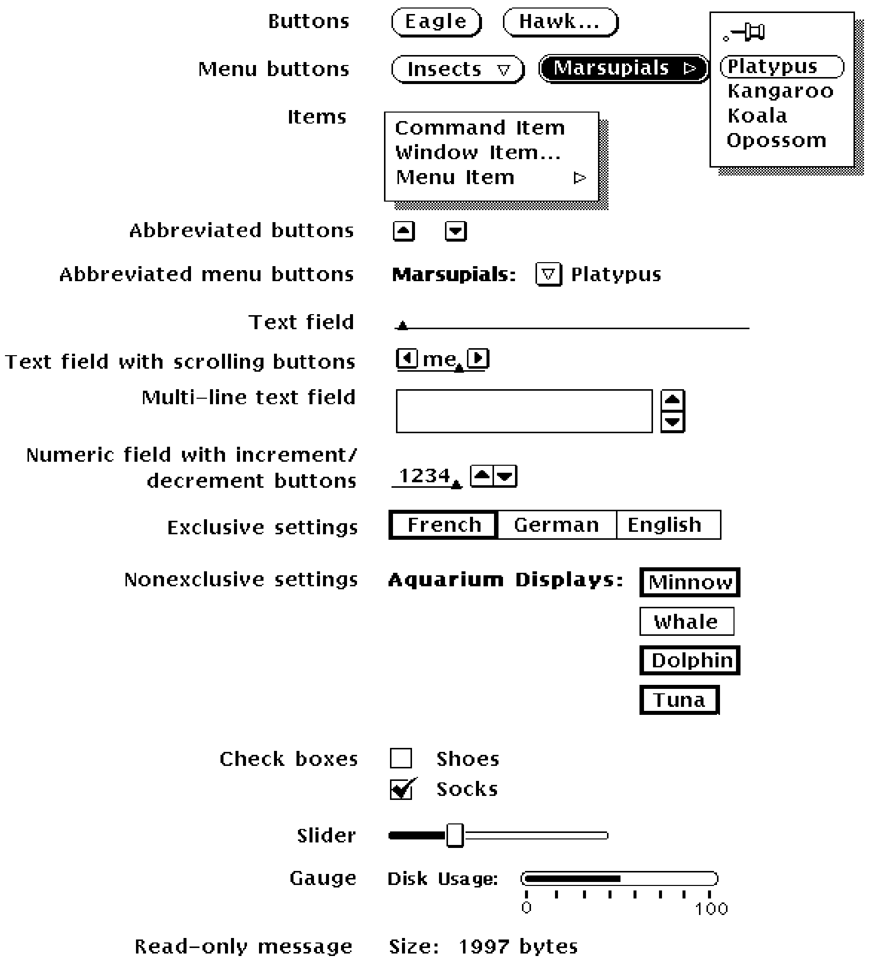


*Figure 7-3. Controls in an OPEN LOOK GUI implementation**

* XView supports the scrolling list item but it is not shown here.

The PANEL package handles all the repainting and resizing events automatically. Panels are not used to display graphics, so there is no need to capture repaint events. Rather than deal with other events specifically callback routines are not installed on panels, but set for each panel item. Because of the varying types of panel items, each item's callback function may be invoked by a different action from the user. While clicking on a panel button is all that is necessary to activate the button's callback routine, a text panel item might be configured to call its notification callback routine when the user presses the RETURN key.

Since panel items express interest in different events, it is the responsibility of the PANEL package to track all events within the panel's window and dispatch events to the proper panel item depending on its type. In some cases, if an event happens over a certain panel item and that item is not interested in that event, the event may be sent to another panel item. For example, what happens if a key is pressed over a panel button? Because the panel button has no interest in the event, the panel will send the event to a text panel item, if one exists elsewhere in the panel.

Section 7.19, "Advanced Panel Usage," describes panel event handling and repainting.

7.1 Creating a Panel

You create a panel by calling `xv_create()` and specifying the PANEL package and a NULL-terminated list of attribute-value pairs. A panel must be created as a child of the frame. All programs that use panels or panel items must include *<xview/panel.h>*. Because panels are uninteresting without panel items, Example 7-1 shows how to create a simple frame, a panel and a panel button. Selecting the panel button causes the program to exit. This is the same *quit.c* program as shown in Chapter 3, *Creating XView Applications*.

Example 7-1. The quit.c program

```
/*
 * quit.c -- simple program to display a panel button that says "Quit".
 * Selecting the panel button exits the program.
 */
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>

Frame frame;

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    void quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
```

Example 7-1. The quit.c program (continued)

```
        XV_HEIGHT,      100,
        NULL);

    panel = (Panel)xv_create (frame, PANEL, NULL);

    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);

    xv_main_loop (frame);
    exit(0);
}

void
quit()
{
    xv_destroy_safe(frame);
}
```

7.1.0.1 Fonts and panels

OPEN LOOK is somewhat restrictive about the use of fonts within panels. Many panel items cannot have their fonts changed at all. Of those that can, none can have their fonts set individually.* However, if a font is set in the panel itself, it is then inherited by the panel items whose fonts can be changed. This guarantees the consistency of fonts in panels. In addition, a panel's font may only be specified when the panel is created.

7.1.1 Scrollable Panels

Scrollable panels are not OPEN LOOK-compliant, but are provided for historical reasons. They are basically just like panels, except that typically not all panel items are in view. A vertical scrollbar attached to the panel allows the user to navigate to the panel items desired. Again, because this type of interface is not OPEN LOOK-compliant, you are discouraged from using this package.

In order to deal with the complications involved with attaching a scrollbar to a panel, the scrollable panel package is subclassed from the `CANVAS` package and thus, the `OPENWIN` package. Scrollable panels are created the same way panels are, but the package name to use is `SCROLLABLE_PANEL`. The scrollable panel package does not create the scrollbars, however. You must create them separately:

* The exact list of panel items varies and may change. Currently, it includes the text used in the value of panel text items.

```
Scrollable_panel sp;
Scrollbar sb;

sp = xv_create(frame, SCROLLABLE_PANEL, NULL);
sb = xv_create(sp, SCROLLBAR, NULL);
```

The principle difference between canvases and scrollable panels is the management of events and the existence of panel items. In canvases, the programmer installs callback routines for events and for repaint and resize routines, and an input mask is set for notification of certain events that the application is interested in. However, like normal panels, the scrollable panel does this automatically. Other than this, scrollable panels may take the same attributes as normal panels.

7.2 Creating Panel Items

Like other XView object, panel items are created using `xv_create()`:

```
Panel_item
xv_create(panel, item_type, attrs)
    Panel panel;
    <item type>             item_type;
    <attribute-value list> attrs;
```

The value of `item_type` must be a panel item from one of the panel item packages:

- `PANEL_ABBREV_MENU_BUTTON`
- `PANEL_BUTTON`
- `PANEL_CHECK_BOX`
- `PANEL_CHOICE`
- `PANEL_CHOICE_STACK`
- `PANEL_DROP_TARGET`
- `PANEL_GAUGE`
- `PANEL_LIST`
- `PANEL_MESSAGE`
- `PANEL_MULTILINE_TEXT`
- `PANEL_NUMERIC_TEXT`
- `PANEL_SLIDER`
- `PANEL_TEXT`
- `PANEL_TOGGLE`

The items in this list represent the items found in Figure 7-4. Each item's type can be retrieved by calling:

```
Panel_item_type type;
type = (Panel_item_type)xv_get(panel_item, PANEL_ITEM_CLASS);
```

`Panel_item_type` is an enumerated type found in *<xview/panel.h>* and contains the following types:

- `PANEL_ABBREV_MENU_BUTTON_ITEM`
- `PANEL_BUTTON_ITEM`
- `PANEL_CHOICE_ITEM`
- `PANEL_DROP_TARGET_ITEM`
- `PANEL_GAUGE_ITEM`
- `PANEL_LIST_ITEM`
- `PANEL_MESSAGE_ITEM`
- `PANEL_MULTILINE_TEXT_ITEM`
- `PANEL_NUMERIC_TEXT_ITEM`
- `PANEL_SLIDER_ITEM`
- `PANEL_TEXT_ITEM`
- `PANEL_TOGGLE_ITEM`
- `PANEL_EXTENSION_ITEM`

Most panel items have no windows associated with them (internally, the package uses Xlib calls to draw the items directly onto the panel window). There are no windows associated with panel items, so panel items should be careful not to overlap one another because one item will not *clip* the one under it. For event processing, when an event occurs in a panel, the package scans through items in the panel in the order that they were created. The event is dispatched to the first item whose item rectangle includes the x,y coordinate of the event. If there is another item that shares the same space, then it will not see the event. For this reason, it is important that panel items are *tiled* and do not intersect partially or completely.

There are cases where the panel item displayed in a particular location is dependent on the state of the application. In such cases, only one panel item would be visible and the other(s) would be hidden. If you need to use more than one panel item at the same location, make one panel item visible by setting `XV_SHOW` to `TRUE`, and hide the other panel item by setting `XV_SHOW` to `FALSE`. Depending on the state of the application, you can toggle the `XV_SHOW` values in each of the panel items.

7.3 Layout of Panels and Panel Items

This section covers the layout of panels and panel items. The layout for panels describes their orientation and the spacing between panel items. The section describing the layout of panel items shows the components of a panel item. Normally, applications do *not* need to alter the default layout mechanism for panels or for panel items.

7.3.1 Panel Layout

A panel lays out panel items in rows and columns. The width and height of the rows and columns may be set by the PANEL package automatically, or by setting window attributes, as well as by setting the sizes of the panel items themselves. The default layout mechanism is usually all that is necessary. You do not need to give explicit layout methods for the general case. Using this approach, the width and height of panels may change dynamically and you will not need to adjust the positions for panel items. Because absolute positioning methods do not allow for dynamic scaling, they are *not* generally recommended.

Whenever panel items are created, they are added to the adjacent panel item in either a row-first or a column-first order. As each new item is added, it is placed at the next position depending on the value of the panel's PANEL_LAYOUT attribute. The default value, PANEL_HORIZONTAL, lays out panel items horizontally until the items have reached the edge of the panel. A new row is then started and the next item is placed at the left edge of the panel. Setting the value of PANEL_LAYOUT to PANEL_VERTICAL causes items to be laid out by column first; when the height of the panel is reached, the next column is started.

If you plan to fit many panel items in a single row or column and want to use default positioning and window_fit() (see Section 7.5, "Sizing Panels") it may be necessary to create the panel with a size greater than the default width or height. Otherwise, items could be placed in a new row or column before expected.

The gap between items as they are laid out vertically and horizontally can be set by the attributes PANEL_ITEM_X_GAP and PANEL_ITEM_Y_GAP. The default gap is 10 pixels in the horizontal direction and 13 pixels in the vertical direction. The panel gap can be set on an entire panel or on individual panel items within a panel.

When you want the next item to start a new row and the panel layout is PANEL_HORIZONTAL, use the attribute PANEL_NEXT_ROW. This specifies that the item is to start a new row and is to be offset by the number of pixels specified. Setting PANEL_NEXT_ROW to the value –1 uses the value of PANEL_ITEM_Y_GAP for the row gap.

Similarly, when you want the next item to start a new column, and the panel layout is PANEL_VERTICAL, use the attribute PANEL_NEXT_COL. It specifies that the item is to start a new column is to be offset by the number of pixels specified. Setting PANEL_NEXT_COL to value –1 uses the value of PANEL_ITEM_X_GAP column gap (see Section 7.4, "Explicit Panel Item Positioning").

Since different panel items have different sizes, the grid is not rigidly adhered to and the position of items may fluctuate within a row. However, it is possible to force items to line up to specific rows and columns, as will be shown later.

Figure 7-4 shows how panels items are laid out as they are created.

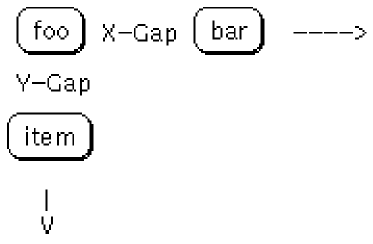


Figure 7-4. Layout of panel items

7.3.2 Panel Item Layout

A panel item is made up of two parts: a label and a value. The area for a panel item's label is called the *label rectangle*. The area for a panel item's value is called the *value rectangle*. The entire area where a panel item may be placed is the panel item's *item rectangle*. The components of a panel item are shown in Figure 7-5. Some panel items, like `PANEL_MESSAGE` and `PANEL_BUTTON`, only have a label. Their value rectangle has a width and height of 0. Other panel items may not have a label; in these cases, the label rectangle has a width and height of 0 (e.g., `PANEL_TEXT`; however, if a `PANEL_TEXT` item does have a label, the label rectangle width and height will be greater than 0).

Except where mandated by OPEN LOOK (e.g., `PANEL_SLIDER`), the panel item attribute `PANEL_LAYOUT` determines the orientation between the label and value. If the layout is horizontal, the value rectangle is placed to the right of the label rectangle. If the layout is vertical, the value rectangle is placed below the label rectangle. Note that `PANEL_LAYOUT` is both a panel and a panel item attribute. However, these represent two different values.

You can embed newlines into strings supplied to the `PANEL_LABEL_STRING` attribute. The newlines tell XView to generate multi-line labels for panel items. Note the sub-strings are right-justified. By default, using multi-line labels, the value rectangle is placed after the last line of the label rectangle.

The attribute `PANEL_BORDER` adds a border around the panel. In a 3D implementation, this border is two pixels wide and presents a "chiseled" appearance. In 2D, the border is one pixel wide. Since the border is rendered directly on the Panel, it is the job of the application to make sure that no Panel_item's overlap the border.

7.4 Explicit Panel Item Positioning

In some cases, you may need to alter the the default panel item positioning. To position panel items explicitly, you can use one of two general methods: relative panel item positioning, or absolute panel item positioning. Using relative panel item positioning, you can adjust your panel items and still accommodate various font sizes. This section describes both of these methods for positioning panel items.

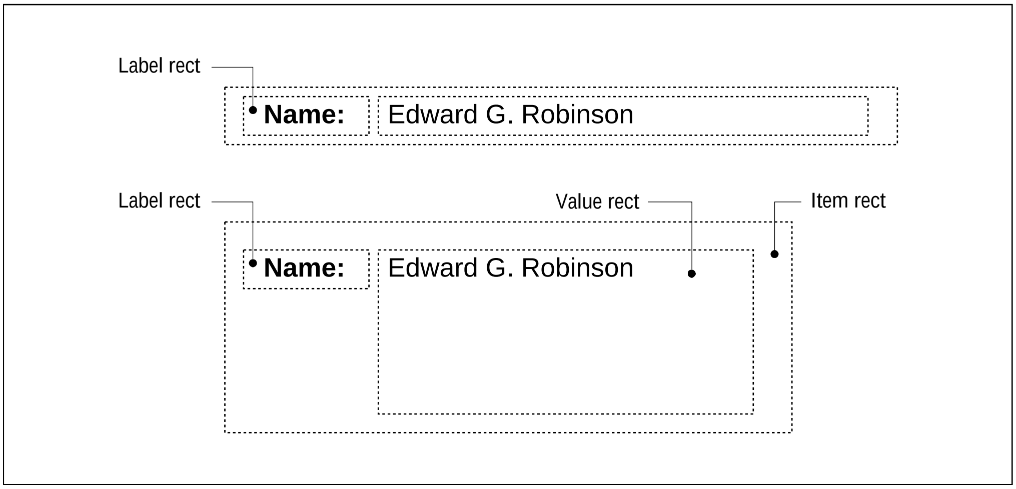


Figure 7-5. Panel item value rectangle and label rectangle

7.4.1 Relative Panel Item Positioning

You can use either of two general methods to accomplish relative panel item positioning. The first method uses the attributes `PANEL_ITEM_X_GAP` and `PANEL_ITEM_Y_GAP` to change the size of the row and column gap between panel items. In the following example, these attributes are used to change the gap between columns to 20 pixels and the gap between rows to 10 pixels:

```
xv_set(panel,
       PANEL_ITEM_X_GAP, 20,
       PANEL_ITEM_Y_GAP, 10,
       NULL);
/* Create all the panel items */
xv_create(panel, PANEL_BUTTON,
       PANEL_LABEL_STRING, "First",
       PANEL_NOTIFY_PROC, first_notify_proc,
       NULL);
xv_create(panel, PANEL_BUTTON,
       PANEL_LABEL_STRING, "Second",
       PANEL_NOTIFY_PROC, second_notify_proc,
       NULL);
```

In the second method for relative item positioning, each panel item you create is laid out relative to the previously created item. After an item is created, you get its XV_RECT and add individual gap values. You then use XV_X and XV_Y to position the next panel item. The following example demonstrates this method:

```
first = xv_create(panel, PANEL_BUTTON,
       PANEL_LABEL_STRING, "First",
       PANEL_NOTIFY_PROC, first_notify_proc,
       NULL);
rect = (Rect *) xv_get(first, XV_RECT);
     /* Position the next item 20 pixels to the right*/
     /* right of the previous item                   */
second = xv_create(panel, PANEL_BUTTON,
       PANEL_LABEL_STRING, "Second",
       PANEL_NOTIFY_PROC, second_notify_proc,
       XV_X, rect_right(rect) + 20,
       NULL);
rect = (Rect *) xv_get(second, XV_RECT);
     /* Position the next item 30 pixels to the right */
     /* of the previous item                          */
third = xv_create(panel, PANEL_BUTTON,
       PANEL_LABEL_STRING, "Third",
       PANEL_NOTIFY_PROC, third_notify_proc,
       XV_X, rect_right(rect) + 30,
       NULL);
```

7.4.2 Absolute Panel Item Positioning

Using absolute panel item positioning, the attributes XV_X and XV_Y specify the position of panel items explicitly. These attributes specify absolute positioning relative to the panel. The attributes PANEL_ITEM_X and PANEL_ITEM_Y are specific to panels. These values reflect the coordinates of the last item created. Therefore, they are both get-only attributes. The following code fragment shows how both are used:

```
panel = (Panel)xv_create(frame, PANEL, NULL);

xv_create(panel, PANEL_BUTTON,
    XV_X,               50,
    XV_Y,               75,
    PANEL_LABEL_STRING, "Quit",
    NULL);

printf("The last item in the panel was at %d %d.0,
    xv_get(panel, PANEL_ITEM_X), xv_get(panel, PANEL_ITEM_Y));
```

This code segment would print:

```
The last item in the panel was at 50 75.
```

When positioning panel items, if the position of the new panel item extends beyond the edges of the panel in the positive direction, the size of the panel increases to include the item. This happens regardless of whether the panel's size was set explicitly at the time it was created.*

*Setting an item to negative coordinates does not increase the size of the panel—only if the item is wide enough or high enough to stretch into the panel window will any of it be visible.

7.4.2.1 General positioning of items

Two functions that are available for general positioning of panel items within windows are `xv_row()` and `xv_col()`. These functions use the values of `WIN_ROW_GAP` and `WIN_COLUMN_GAP` of the panel. While these attributes control the spacing between panel items, the distance between items and the edge of the panel is set to a constant 4 pixels.

Consider the following code fragment that positions items within regimented rows and columns:

```
int         rows, cols;
extern char *names[3][5];

frame = (Frame)xv_create(NULL, FRAME, NULL);
panel = (Panel)xv_create(frame, PANEL,
    WIN_ROW_GAP,          70,
    WIN_COLUMN_GAP,       20,
    NULL);

for (rows = 0; rows < 3; rows++)
    for (cols = 0; cols < 5; cols++)
        (void) xv_create(panel, PANEL_BUTTON,
            XV_X,               xv_col(panel, cols),
            XV_Y,               xv_row(panel, rows),
            PANEL_LABEL_STRING, names[rows][cols],
            PANEL_NOTIFY_PROC,  (rows+cols==0)? quit : selected,
            PANEL_CLIENT_DATA,  frame,
            NULL);
```

This code displays a panel with a 70-pixel distance between the upper-left corners of the panel items. Thus, each panel item must be less than 70 pixels wide or they will overlap one another. This contrasts with `PANEL_ITEM_X_GAP` and `PANEL_ITEM_Y_GAP`, which specify the gap between the right and left sides of adjacent panel items. Note that this does not affect the distance between the edge of the panel and the items along the perimeter; that distance remains constant at 4 pixels.

The `xv_col()` and `xv_row()` method of explicit panel item placement does not take into account the sizes of the panel items. If not enough space is given between the rows and columns (`WIN_COLUMN_GAP`, `WIN_ROW_GAP`), then items will lie on top of one another. Neither does this method take into account dynamic scaling of items and fonts. As a result, this and all absolute positioning methods are *not* recommended.

7.4.3 Layout of Panel Items with Values

For panel items with values, `PANEL_VALUE_X` and `PANEL_VALUE_Y` can be used instead of the `XV_X` and `XV_Y` attributes mentioned above to align the value rectangles of panel items. The label portion of the panel item is then positioned automatically according to the panel layout characteristics currently in effect. This is useful, for example, when you need to align a series of `PANEL_TEXT` items. It is important *not* to use `XV_X` and `XV_Y` when using the label and value positioning attributes.

7.5 Sizing Panels

The size of a panel, by default, extends to the bottom and right edges of the frame in which it is placed (assuming there are no other subwindows in the frame). Alternatively, the panel's dimensions can be set explicitly using `XV_WIDTH` and `XV_HEIGHT`. If it is important to maintain the layout of the panel items in a panel, then the dimensions should be set explicitly.

More often than not, you want the panel to be just the minimum height and width required to encompass all of its items. You can set the minimum height and width using the macros `window_fit_height()` or `window_fit_width()`, respectively. You can set both in a single call to `window_fit()`. These macros are called after all the items have been created.

The attributes `PANEL_EXTRA_PAINT_WIDTH` and `PANEL_EXTRA_PAINT_HEIGHT` specify the increment by which a panel will grow in the *x* and *y* directions, respectively.

7.6 Panel Item Values

Many panel items are associated with a specific value. A text item has a *string* value, a numeric text item has an integer value, a choice item has a current-choice value, and so on.*
To set a value, use:

```
xv_set(item, PANEL_VALUE, value, NULL)
```

Of course, the type of `value` is dependent on the type of panel item whose value is being set. Consider the following examples:

```
/* Set the text field in the text item to print "Hello World" */
xv_set(text_item, PANEL_VALUE, "Hello world.", NULL);

/* Set the numeric value of the numeric text item to be 10 */
xv_set(text_num_item, PANEL_VALUE, 10, NULL);

/* Set the current choice in choice_item to be the fifth choice */
xv_set(choice_item, PANEL_VALUE, 4, NULL);
```

NOTE

The values for string-valued attributes are dynamically allocated when they are created or set. The value you specify is *copied* into the newly allocated space. If a previous value was present, the panel item frees its old data first.†

Panel item values are retrieved in a similar way:

```
xv_get(item, PANEL_VALUE);
```

*For details about the value type of a panel item, see the corresponding sections on specific panel items (Sections 7.9 through 7.18).

†This contrasts with menu items. See Chapter 11, *Menus*.

Since the `xv_get()` routines are used to retrieve attributes of all types, you should cast the value returned into the type appropriate to the attribute being retrieved:

```
int val;
val = (int)xv_get(num_text_item, PANEL_VALUE);
printf("The int–value in the num_text_item is: ’%d’0, val);
```

NOTE

`xv_get()` does not dynamically allocate storage for the values it returns. If the value returned is a pointer, it points directly into the panel’s private data. It should be considered *read-only*—do not change the contents of the pointer; it is your responsibility to copy the information pointed to.

7.7 Iterating Over a Panel’s Items

You can iterate over each item in a panel with the two attributes `PANEL_FIRST_ITEM` and `PANEL_NEXT_ITEM`. A pair of macros, `PANEL_EACH_ITEM()` and `PANEL_END_EACH` are also provided for this purpose. For example, to destroy each item in a panel:

```
Panel_item item;

PANEL_EACH_ITEM(browser, item)
    xv_destroy(item);
PANEL_END_EACH
```

Note that a semicolon is not required after `PANEL_END_EACH`.

7.8 Panel Item Classes

Nine types of panel items are presented here:

- Panel Buttons, Menu Buttons, and Abbreviated Menu Buttons
- Checkboxes
- Exclusive and Nonexclusive Choices
- Abbreviated Choices
- Scrolling Lists
- Message Items
- Sliders
- Text Items (including numeric and multiline text items)
- Drop Target Items

Items are made up of one or more displayable components. One component shared by all item types is the *label*. An item label is either a string or a graphic image.

The user interacts with items through various methods ranging from mouse button selection to keyboard input. This interaction typically results in a *callback* function being called for the panel item. The callback functions also vary on a per-item basis. Each item type is described in the following sections.

7.9 Button Items

A button item allows the user to invoke a command or bring up a menu. Examples of various buttons are listed in Figure 7-6. The button's label identifies the name of the command or menu. A button label that ends in three dots (. . .) indicates that a pop-up menu will be displayed when the button is selected.

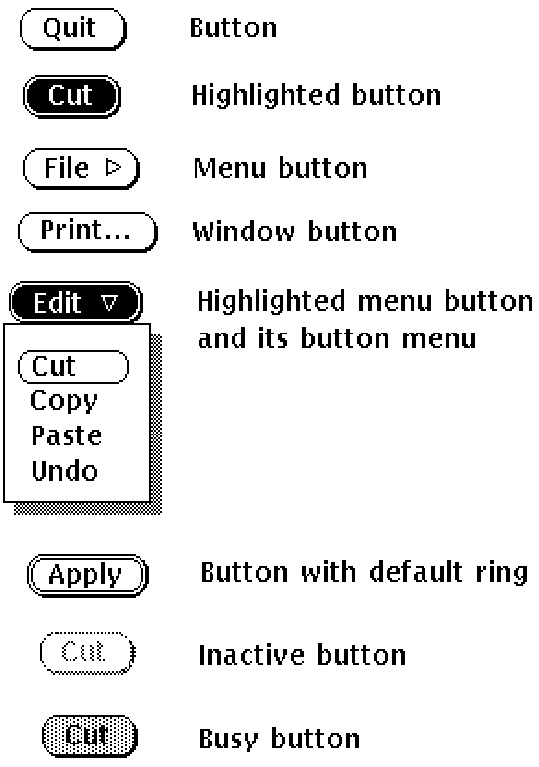


Figure 7-6. Visual feedback for button controls

A button requires a label specified by the attribute `PANEL_LABEL_STRING` or `PANEL_LABEL_IMAGE`. Buttons do not have a `PANEL_VALUE` associated with them.

7.9.1 Button Selection

The user invokes a panel button by clicking the SELECT mouse button on it. When this happens, the button's notify procedure is called. The notify procedure is installed by specifying PANEL_NOTIFY_PROC, as in the following example:

```
xv_create(panel, PANEL_BUTTON,
    PANEL_NOTIFY_PROC,   quit_proc,
    PANEL_LABEL_STRING,  "Quit",
    NULL);
```

When the button is selected, the notify procedure is called. The form of the notify procedure for a button is:

```
void
button_notify_proc(item, event)
    Panel_item  item;
    Event      *event
```

The function does not return a value, but if the action that the button had intended to take fails, then you should set the item's PANEL_NOTIFY_STATUS to XV_ERROR (e.g., if the button was labeled "save" but the actual save operation failed). It is set to XV_OK by default. If the button is part of an unpinned Command Frame, setting PANEL_NOTIFY_STATUS to XV_ERROR will prevent the Command Frame from being dismissed. Use MENU_NOTIFY_STATUS for a menu. In a callback, setting this to XV_ERROR on the menu that was notified prevents the frame from being dismissed.

When a button's notify procedure is called, the button's *busy* state is set. When a button's busy state is set to TRUE, the button does not accept further input (clicking the SELECT mouse button will do nothing to the button item). The busy state is cleared when the notify procedure exits. The busy state can be maintained after exiting the notify procedure by setting PANEL_BUSY to TRUE from within the notify procedure.

7.9.1.1 Making a button inactive

The attributes PANEL_INACTIVE is used to make a panel button inactive. If TRUE, the button item cannot be selected. Inactive items are displayed with gray-out pattern as shown in Figure 7-6.

7.9.2 Menu Buttons

It is often useful to attach a menu to a button. The menu may provide alternate values or functions for the button to invoke. Since the menu is a separate entity (in other words, it is not created when the button is created—you have to create it on your own), the menu may have callback routines associated with it and its menu items.*

* Read Chapter 11, *Menus*, for details of menus and menu items.

When a menu button receives an `ACTION_MENU` down event, the button's notify procedure is called before the menu is displayed. This gives you the chance to modify the menu beforehand. When the menu button is selected by clicking the SELECT button, the button's notify procedure is called before the menu's notify procedure.

Menu buttons contain a triangle pointing in the direction in which the menu will be displayed.

The *btn_menu.c* program in Example 7-2 demonstrates how a menu can be attached to a panel button.

Example 7-2. The btn_menu.c program

```
/*
 * btn_menu.c -- display a panel that has an OPEN LOOK menu button.
 * The choices displayed are Yes, No and Quit.  If Quit is selected
 * in the menu, the program exits.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/openmenu.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Menu        menu;
    int         selected();
    void        menu_proc();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);

    /* Create the menu _before_ the panel button */
    menu = (Menu)xv_create(NULL, MENU,
        MENU_NOTIFY_PROC,       menu_proc,
        MENU_STRINGS,           "Yes", "No", "Quit", NULL,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Y/N/Q",
        PANEL_NOTIFY_PROC,      selected,
        PANEL_ITEM_MENU,        menu, /* attach menu to button */
        NULL);
    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

int
selected(item, event)
Panel_item item;
Event *event;
{
```

Example 7-2. The btn_menu.c program (continued)

```
    printf("%s selected...\n", xv_get(item, PANEL_LABEL_STRING));
    return XV_OK;
}

void
menu_proc(menu, menu_item)
Menu menu;
Menu_item menu_item;
{
    printf("Menu Item: %s\n", xv_get(menu_item, MENU_STRING));
    if (!strcmp((char *)xv_get(menu_item, MENU_STRING), "Quit"))
        exit(0);
}
```

The output produced by this program is shown in Figure 7-7.

7.9.2.1 Destroying menu buttons

XView automatically destroys the menu associated with the button unless there are other references to the menu. For each button or any other XView object that references the menu in question, the menu's `XV_REF_COUNT` is incremented. Thus, if five buttons reference the same menu, then destroying one button will not destroy the menu, but it will decrement the reference count by one. If the destruction of the panel item would decrement the menu's reference count to `0`, then the menu itself is destroyed. Therefore, if you wish to prevent the menu from being destroyed, you can forcefully increment the menu's reference count by at least one more than what it is. That way, no matter how many times the menu is used, its reference count will always be set to a value greater than zero, and it will never be destroyed by destroying panel items that use it. Of course, you can always call `xv_destroy()` on the menu explicitly when you want to destroy the menu once and for all.

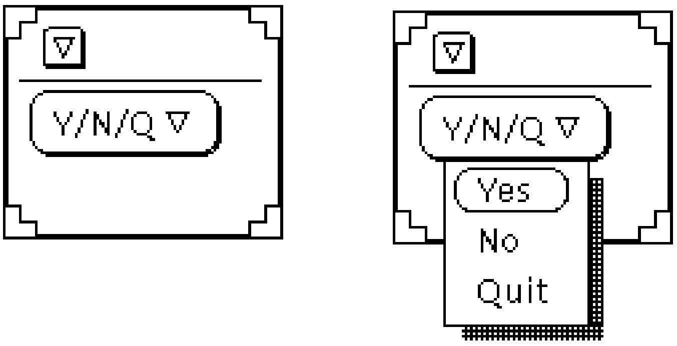


Figure 7-7. Sample menu button (unselected and selected)

You can use the attributes `XV_INCREMENT_REF_COUNT` and `XV_DECREMENT_REF_COUNT` on any item.

```
    xv_set(menu, XV_INCREMENT_REF_COUNT, NULL);
```

This is not recommended except when you need to circumvent the normal functionality of XView.

7.9.3 Panel Button Width

The width of a panel button's string or image is available and can be set with the attribute `PANEL_LABEL_WIDTH`. This attribute does not include the width of a panel button's end caps or menu marks. `PANEL_LABEL_WIDTH` has no effect on a `PANEL_BUTTON` until the panel button's `PANEL_LABEL_STRING` or `PANEL_LABEL_IMAGE` is set.

To make all the panel buttons the same width for a group of buttons that are not all menu buttons or all non-menu buttons, requires three steps:

1. Set the `PANEL_LABEL_WIDTH` for each button.
2. Find the maximum `XV_WIDTH` of all the buttons in the group. This value includes the width of any end caps or menu marks.
3. Finally, add the difference between each button's `XV_WIDTH` and the maximum `XV_WIDTH` of all the buttons to each button's `PANEL_LABEL_WIDTH`:

```
xv_set(item, PANEL_LABEL_WIDTH,
       (int)xv_get(item, PANEL_LABEL_WIDTH) +
       max_XV_WIDTH - (int)xv_get(item,XV_WIDTH),
       NULL);
```

7.9.4 Abbreviated Menu Buttons

Abbreviated menu buttons are just like menu buttons. However, they do not display the label inside the button, but to the left, as shown in Figure 7-8.

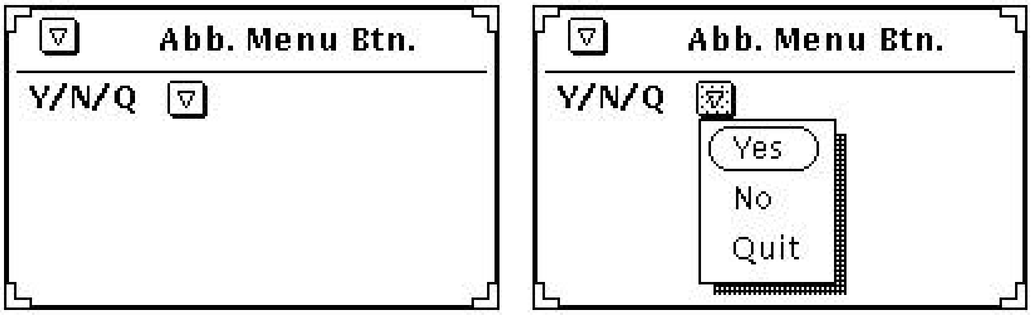


Figure 7-8. Sample abbreviated menu button

Abbreviated menu items are created using the PANEL_ABBREV_MENU_BUTTON package.

```
Panel_item item;
extern Menu menu; /* created separately */

item = xv_create(panel, PANEL_ABBREV_MENU_BUTTON,
    PANEL_ITEM_MENU,   menu,
    NULL);
```

Notification is the same as in full-size menu buttons.

7.10 Choice Items

Choice items provide a list of different choices to the user in which one or more choices may be selected. There are variations of choice items which implement different OPEN LOOK objects such as:

- Exclusive and Nonexclusive Choices (or Settings)
- Abbreviated Choice Items
- Checkboxes

Behind the flexibility of presentation lies a uniform structure consisting of a label, a list of choices and an indication of which choice or set of choices is currently selected. The choices can be displayed as either text strings (PANEL_CHOICE_STRINGS) or server images (PANEL_CHOICE_IMAGES). The number of choices in a choice list is returned by the PANEL_NCHOICES attribute. This is true for toggle items as well.

7.10.1 Display and Layout of Item Choices

The attribute PANEL_DISPLAY_LEVEL determines which of an item's choices are actually displayed. The display level may be set to:

PANEL_ALL	The default. All choices are shown.
PANEL_CURRENT	Only the current choice is shown.
PANEL_NONE	No choices are shown.

The choices are laid out either horizontally or vertically next to the label depending on the value of the item's value for PANEL_LAYOUT. By default, this value is PANEL_HORIZONTAL.

Sometimes the number of choices in a choice list can get long and the menu for the item (if any) may look aesthetically bad or not fit on the screen. You can specify that choices appear in row and column layout by specifying either the number of rows or the number of columns with the attributes PANEL_CHOICE_NROWS and PANEL_CHOICE_NCOLS. If both are specified, the last one specified takes precedence.

7.10.2 Exclusive and Nonexclusive Choices

When a default choice item is created, its type is an exclusive choice item allowing the user to select only one choice from the list. The value of the panel item is the currently selected choice. The index of the first choice is 0. In the following example, we create several choice items as exclusive settings:

```
xv_create(panel, PANEL_CHOICE,
    PANEL_LABEL_STRING,      "Choice - Exclusive",
    PANEL_CHOICE_STRINGS,    "One", "Two", "Three", "Four", NULL,
    PANEL_NOTIFY_PROC,       selected,
    PANEL_VALUE,             3,
    NULL);
```

This code fragment produces the panel item shown in Figure 7-9.

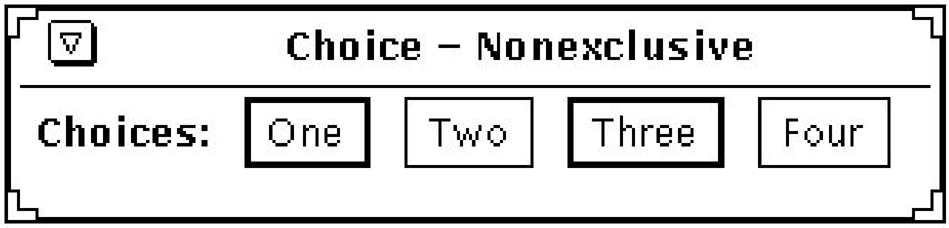


Figure 7-9. Sample panel with exclusive choices

Figure 7-9 represents a panel item that has four choices, the fourth of which is *set*. If the user makes another choice, the value of the item, and therefore the value of PANEL_VALUE, changes to the ordinal number of the choice.

The choice item can be made nonexclusive, allowing more than one of the choices to be selected, when the attribute PANEL_CHOOSE_ONE is set to FALSE. The macro PANEL_TOGGLE has been defined as:

```
PANEL_CHOICE, PANEL_CHOOSE_ONE, FALSE
```

This macro affects the panel item in two ways. More than one choice may be set in the visual feedback, and the value of the item is set as a mask indicating the choices selected. In the following example, the choice items in the previous example are shown as nonexclusive settings:

```
xv_create(panel, PANEL_TOGGLE,
    PANEL_LABEL_STRING,      "Choice - Nonexclusive",
    PANEL_CHOICE_STRINGS,    "One", "Two", "Three", "Four", NULL,
    PANEL_NOTIFY_PROC,       selected,
    PANEL_VALUE,             5,
    NULL);
```

This code fragment would produce the panel item shown in Figure 7-10.

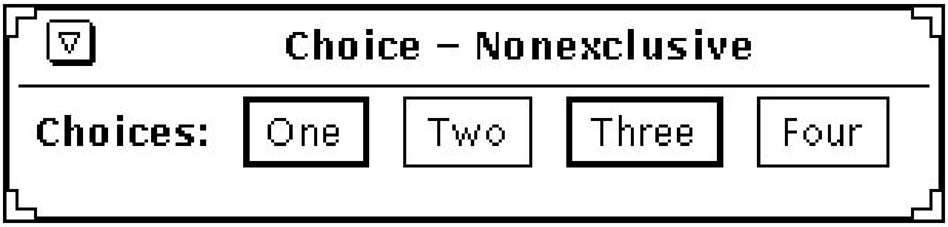


Figure 7-10. Sample panel with nonexclusive choices

Figure 7-10 represents a panel item that has two choices set, the first and the third. The reason for this is that the panel's value is set to 5, which is a mask that represents the first and third bits. For example, 5 in binary is . . . 00101.* In the binary representation, the first and third bits from the right are 1's. This means that the first and third choices are selected. This is how the value is interpreted on calls to `xv_set()` or `xv_create()`, and how it is returned for calls to `xv_get()`.

To get the image for a choice item's current choice (assuming the choice item is a `Server_image` choice):

```
Server_image image;
image = (Server_image)xv_get(item, PANEL_CHOICE_IMAGE,
                             xv_get(item, PANEL_VALUE));
```

For choice items whose `PANEL_CHOOSE_NONE` value is `TRUE`, a `PANEL_VALUE` of -1 may be set or returned, indicating that no choices are set for that item.

Setting `PANEL_CHOOSE_NONE` allows choice items to have no currently selected item. This attribute is not applicable if `PANEL_CHOOSE_ONE` is `FALSE`.

7.10.3 Abbreviated Choices

Abbreviated choices are exclusive choices that either display no value or only the current value. A menu is used to display all the choices. To implement abbreviated choice items, the macro `PANEL_CHOICE_STACK` is used. This macro creates an abbreviated choice item that displays only the current value. It is defined as:

```
PANEL_CHOICE, PANEL_DISPLAY_LEVEL, PANEL_CURRENT
```

To create an abbreviated choice item that does *not* display the current value, use the `PANEL_ABBREV_MENU_BUTTON` package and set `PANEL_DISPLAY_LEVEL` to `PANEL_NONE` (refer to Section 7.9.4, "Abbreviated Menu Buttons," for details). The following example demonstrates creating an abbreviated choice item that displays the current value:

```
xv_create(panel, PANEL_CHOICE_STACK,
    PANEL_LAYOUT,           PANEL_VERTICAL,
```

*The value for nonexclusive choice items is stored as an `unsigned int` and the maximum number of nonexclusive choice items is 32.

```
        PANEL_LABEL_STRING,     "Abbreviated Choice",
        PANEL_CHOICE_STRINGS,   "One", "Two", "Three", "Four", NULL,
        PANEL_NOTIFY_PROC,      selected,
        PANEL_VALUE,            1,
        NULL);
```

The panel item created by this code is shown in Figure 7-11.

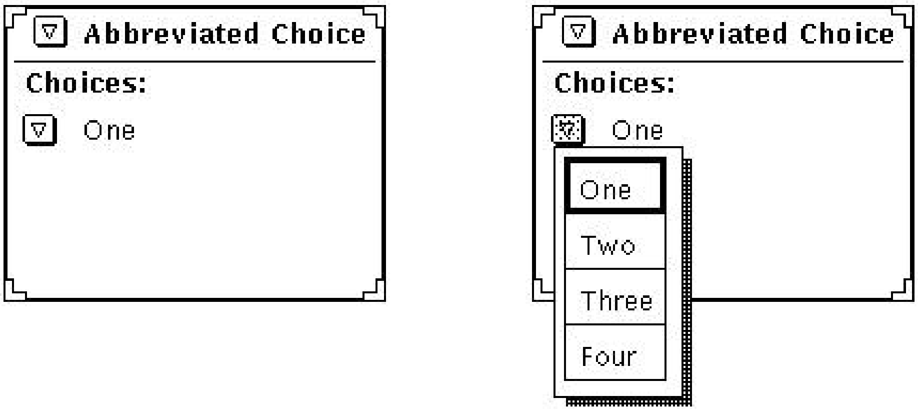


Figure 7-11. Sample panel with abbreviated choice (unselected and selected)

Here, since only the current selection is visible, the only way to make other choices in the item is to bring up a menu. The value of the panel item is the same as for an exclusive PAN-EL_CHOICE item.

7.10.4 Checkbox Choices

Checkboxes are nonexclusive choices that use checkmarks to indicate the selected choices. Unselected choices have empty checkboxes. The following example demonstrates check-boxes:

```
    xv_create(panel, PANEL_CHECK_BOX,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_LABEL_STRING,     "Choices",
        PANEL_CHOICE_STRINGS,   "One", "Two", "Three", "Four", NULL,
        PANEL_NOTIFY_PROC,      selected,
        PANEL_VALUE,            5,
        NULL);
```

The panel item created by this code is shown in Figure 7-12.

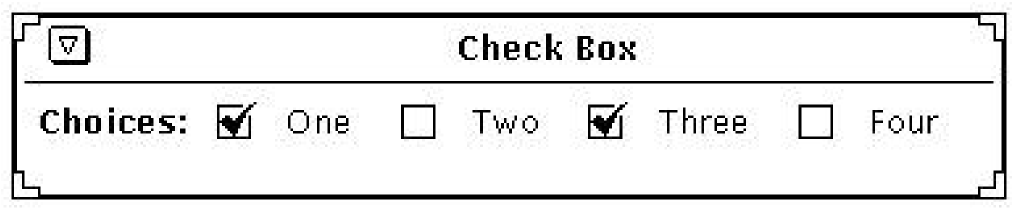


Figure 7-12. Sample panel with checkbox

All of the choices can be selected in the same way as a `PANEL_TOGGLE`.

7.10.5 Choice Selection and Notification

The user can make a selection from a choice item by selecting the desired choice directly with the SELECT mouse button.

The procedure specified via the attribute `PANEL_NOTIFY_PROC` will be called when any of its choices are selected. If a choice item's current selection or value changes as a result of a call to `xv_set()` from somewhere else, then the notify procedure is not called. The choice notify procedure is passed the item, the current value of the item, and the event that caused notification:

```
void
choice_notify_proc(item, value, event)
    Panel_item  item;
    int         value;
    Event      *event;
```

Like the button's notify procedure, the choice notify procedure is also a void function. If the function fails to perform its task, you should set the `item`'s `PANEL_NOTIFY_STATUS` to `XV_ERROR`.

For exclusive choices, the value passed to the notify procedure is the ordinal number corresponding to the current choice (the choice that the user has just selected). The first choice has ordinal number zero. For nonexclusive choices, the value is a mask of the currently selected choices in the list (see Section 7.11.3, "List Selection").

The `event` is the event that caused the notify procedure to be called. For these types of choices, the event action will probably be `ACTION_SELECT`.

7.10.6 Foreground Color in Choice Items

Colors for panel choice items may be set with the `PANEL_CHOICE_COLOR` attribute. This attribute sets the foreground color index for specified choices. The following example demonstrates a choice using the foreground color attribute.

```
xv_create(panel, PANEL_TOGGLE,
    PANEL_LABEL_STRING,      "Choices",
    PANEL_CHOICE_STRINGS,    "One", "Two", "Three", "Four", NULL,
    PANEL_NOTIFY_PROC,       selected,
    PANEL_VALUE,             5,
    PANEL_CHOICE_COLOR, 0, RED, NULL,
    PANEL_CHOICE_COLOR, 1, BLUE, NULL,
    PANEL_CHOICE_COLOR, 2, RED, NULL,
    PANEL_CHOICE_COLOR, 3, BLUE, NULL,
    NULL);
```

7.10.7 Parallel Lists

Parallel lists are lists of values for particular attributes that correspond to each choice in the panel item. An example of a parallel list is PANEL_CHOICE_XS and PANEL_CHOICE_YS. These two attributes take as values a NULL-terminated list of coordinates to specify explicit placement of the choices when they are displayed (assuming PANEL_ALL is the display format).

WARNING

> The attributes PANEL_CHOICE_XS, PANEL_CHOICE_YS, PANEL_CHOICE_X and PANEL_CHOICE_Y are provided for SunView1 Compatibility. They are mentioned here for explanatory purposes only. Their use allows you to create applications that may *not* be OPEN LOOK-compliant.

These attributes are used to display choices in adjacent rows and columns, as in the following example:

```
xv_create(panel, PANEL_CHOICE,
    PANEL_CHOICE_STRINGS,     "One", "Two", "Three", NULL,
    PANEL_CHOICE_XS,          10, 70, 130, NULL,
    PANEL_CHOICE_YS,          90, NULL,
    PANEL_VALUE,              2,
    PANEL_NOTIFY_PROC,        notify_proc,
    NULL);
```

The choice item has three choices: the strings “One”, “Two,” and “Three.” We have specified explicit positioning of the choice items using the attributes PANEL_CHOICE_XS and PANEL_CHOICE_YS. These attributes take precedence over PANEL_LAYOUT, so that layout is ignored if specified. Note that the list PANEL_CHOICE_YS has only one element. When any of the parallel lists are abbreviated in this way, the last element given will be used for the remainder of the choices. So, in the example above:

```
90, NULL,
```

serves as shorthand for:

```
90, 90, 90, NULL,
```

All the choices will appear at *y* coordinate 90, while the *x* coordinates for the choices will be 10, 70, and 130, respectively.

You cannot specify that a choice appear at *x = 0* or *y = 0* by using the attributes PANEL_CHOICE_XS or PANEL_CHOICE_YS. Since these attributes take NULL-terminated lists as values, the zero would be interpreted as the terminator for the list. You may achieve the desired effect by setting the positions individually. The attributes PANEL_CHOICE_X or PANEL_CHOICE_Y take as values the number of the choice followed by the desired position. The following example demonstrates setting the position of choice items:

```
Panel_item   choice;
int          i;
extern char *strings[ ];

choice = (Panel_item)xv_create(panel, PANEL_CHOICE,
    PANEL_CHOOSE_ONE,        FALSE,
    NULL);
for (i = 0; i < sizeof(strings) / sizeof(char *); i++)
    xv_set(choice,
        PANEL_CHOICE_STRING, i, strings[i],
        PANEL_CHOICE_X,      i, i*20,
        PANEL_CHOICE_Y,      i, i*20,
        NULL);
```

After the choice item is created, the *x* and *y* positions of the choices are set individually in a loop.

Once a set of choice items is created, the rectangle that encloses a specified choice may be returned using the attribute PANEL_CHOICE_RECT. It takes an integer argument representing the index of the choice whose rect pointer is returned.

7.11 Scrolling Lists

OPEN LOOK's specification for *scrolling lists* is implemented by the PANEL_LIST panel item. List items allow the user to make selections from a scrolling list of choices larger than can be displayed on the panel at one time. The selections can be exclusive or nonexclusive, like the choice items outlined in the previous section. The list is made up of strings or images and a scrollbar that functions like any scrollbar in XView, except that it cannot be split.* List items are laid out in rows only—one list entry per row. Below is a code fragment for creating a simple list:

```
xv_create(panel, PANEL_LIST,
    PANEL_LIST_STRINGS,     "One", "Two", "Three", "Four", NULL,
    NULL);
```

The list items produced by this code are shown in Figure 7-13.

* See Chapter 10, *Scrollbars*, for a further description of how scrollbars work.

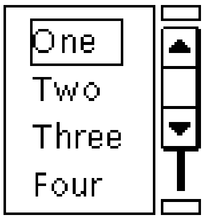


Figure 7-13. Sample panel with scrolling list

7.11.1 Displaying List Items

You can use either text strings or server images to display the choice to the user; you can even intermix them. You specify the choices either one at a time or in a group. To set only one choice, use `PANEL_LIST_STRING` or `PANEL_LIST_GLYPH`. When creating a new string or glyph entry, if the index into the list specified is larger than the total number of entries, then the new item is added to the end of the list. Use `PANEL_LIST_STRINGS` or `PANEL_LIST_GLYPHS` to set all the choices in a group. If no items exist in the list, the appropriate number of rows are created to fit all of the items. If the list already contains items, then the first *n* rows of items are replaced by the newly specified strings or glyphs (where *n* is the number of strings or glyphs specified).

The width of the list item can be set to explicit pixel values using `PANEL_LIST_WIDTH`. The minimum value for this attribute is 25. This reserves enough space for the list's borders and margins. Setting `PANEL_LIST_WIDTH` to -1 extends the width of the scrolling list box to the edge of the panel. Setting `PANEL_LIST_WIDTH` to 0 sets the width to that of the widest row in the scrolling list. Alternatively, the number of rows that are displayed in the list item can be controlled through the value of the `PANEL_LIST_DISPLAY_ROWS` attribute. This value governs the height, in rows, of the list item.

The default panel font for a scrolling list is the default font for the panel. To specify a particular font for a scrolling list row, use `PANEL_LIST_FONT`, which takes two arguments, a row number and a font. To set the fonts for multiple rows, use `PANEL_LIST_FONTS`. Note that the font specification using either `PANEL_LIST_FONT` or `PANEL_LIST_FONTS` should follow the creation of the rows (for example, by `PANEL_LIST_STRINGS`).

By default, a scrolling list does not have a title. To add a title to a scrolling list, use `PANEL_LIST_TITLE` as in the following code segment:

```
    xv_set(panel_list_item, PANEL_LIST_TITLE, "Patterns", NULL);
```

The title appears above the list items. The package makes a copy of the string passed to the `PANEL_LIST_TITLE` attribute. The package also will free the string when the title string is no longer needed.

The height of each row in the list may be set using `PANEL_LIST_ROW_HEIGHT`. All rows have the same height. If the items in the list are glyphs, then the height of each row must be specified by at least the height of the tallest glyph in the list. This should be determined before the list of glyphs is set in the list item. Entries in the list can be either glyphs or strings;

an entry containing both a string and a glyph will display both. The glyph will be on the left and the string will be on the right. Consider the program in Example 7-3.

Example 7-3. The list_glyphs.c program

```
/*
 * list.c -- show a scrolling list with three items in it.
 * Each item is an icon (a pattern) and a string.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/svrimage.h>

#define gray1_width 16
#define gray1_height 16
static char gray1_bits[ ] = {
    0x55, 0x55, 0xaa, 0xaa, 0x55, 0x55, 0xaa, 0xaa, 0x55, 0x55,
    0xaa, 0xaa, 0x55, 0x55, 0xaa, 0xaa, 0x55, 0x55, 0xaa, 0xaa,
    0x55, 0x55, 0xaa, 0xaa, 0x55, 0x55, 0xaa, 0xaa, 0x55, 0x55,
    0xaa, 0xaa
};

#define gray2_width 16
#define gray2_height 16
static char gray2_bits[ ] = {
    0x11, 0x11, 0x00, 0x00, 0x44, 0x44, 0x00, 0x00, 0x11, 0x11,
    0x00, 0x00, 0x44, 0x44, 0x00, 0x00, 0x11, 0x11, 0x00, 0x00,
    0x44, 0x44, 0x00, 0x00, 0x11, 0x11, 0x00, 0x00, 0x44, 0x44,
    0x00, 0x00
};

#define gray3_width 16
#define gray3_height 16
static char gray3_bits[ ] = {
    0x22, 0x22, 0xee, 0xee, 0x33, 0x33, 0xee, 0xee, 0x22, 0x22,
    0xee, 0xee, 0x33, 0x33, 0xee, 0xee, 0x22, 0x22, 0xee, 0xee,
    0x33, 0x33, 0xee, 0xee, 0x22, 0x22, 0xee, 0xee, 0x33, 0x33,
    0xee, 0xee
};

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Server_image gray1, gray2, gray3;
    extern void exit(), which_glyph();

    xv_init(XV_INIT_ARGS, argc, argv, NULL);

    gray1 = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               gray1_width,
        XV_HEIGHT,              gray1_height,
        SERVER_IMAGE_BITS,      gray1_bits,
        NULL);
    gray2 = (Server_image)xv_create(NULL, SERVER_IMAGE,
```

Example 7-3. The list_glyphs.c program (continued)

```
        XV_WIDTH,               gray2_width,
        XV_HEIGHT,              gray2_height,
        SERVER_IMAGE_BITS,      gray2_bits,
        NULL);
    gray3 = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               gray3_width,
        XV_HEIGHT,              gray3_height,
        SERVER_IMAGE_BITS,      gray3_bits,
        NULL);
    frame = (Frame)xv_create(NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);

    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);

    (void) xv_create(panel, PANEL_LIST,
        PANEL_LIST_ROW_HEIGHT,   16,
        PANEL_LIST_GLYPHS,       gray1, gray2, gray3, NULL,
        PANEL_LIST_STRINGS,      "Pattern1", "Pattern2", "Pattern3", NULL,
        PANEL_LIST_CLIENT_DATAS, 1, 2, 3, NULL,
        PANEL_NOTIFY_PROC,       which_glyph,
        NULL);

    window_fit(panel);
    window_fit(frame);

    xv_main_loop(frame);
}

void
which_glyph(item, string, client_data, op, event, row)
Panel_item     item; /* panel list item */
char          *string;
caddr_t        client_data;
Panel_list_op  op;
Event         *event;
int            row;
{
    printf("item = %s (#%d), op = %d0, string, client_data, op);
}
```

The output produced by *list_glyphs.c* is shown in Figure 7-14.

The height of each row in the list is determined by the height of the scrolling list font. If glyphs are used, the programmer is responsible for row height. In the example, all the glyphs are the same height (16 pixels), so the calculation is easy: `PANEL_LIST_ROW_HEIGHT` is set to 16. If a glyph exceeds the row height, then a warning is printed and the glyph is ignored.

The use of the notify procedure is discussed in Section 7.11.4, "List Notification."

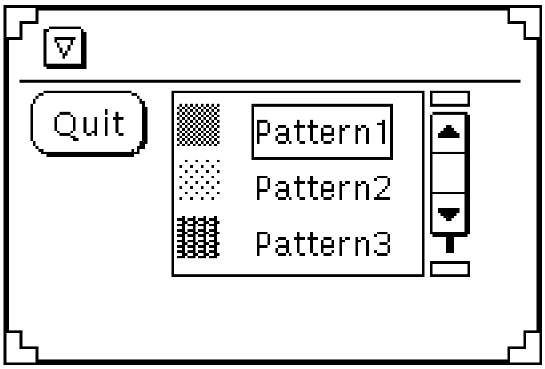


Figure 7-14. Output of program list_glyphs.c

7.11.2 Adding and Deleting List Entries

List entries are denoted by row number. The first entry is row 0. Entries are added and deleted from the list at run time. Several attributes enable you to add and delete entries in a list item. The attributes that let you delete list entries are: `PANEL_LIST_DELETE`, `PANEL_LIST_DELETE_ROWS`, and `PANEL_LIST_DELETE_SELECTED_ROWS`. Using `PANEL_LIST_DELETE`, the attribute value specifies a single list item to delete. The string and/or image resources are deallocated and the list is updated appropriately.*

The attribute `PANEL_LIST_DELETE_ROWS` deletes multiple list item rows. This attribute takes two integer arguments. The first argument is the starting row number, the second argument is the number of rows to delete. In the following example, rows 6 through 8 in `panel_list_item` are removed. Row 0 is the first row.

```
xv_set(panel_list_item,
    PANEL_LIST_DELETE_ROWS, 6, 2,
    NULL);
```

If you use `PANEL_LIST_DELETE` to delete multiple list items, you need to delete list items in descending order. When a row is deleted, the row numbers are adjusted to a sequential order. For example, to delete rows `1` through `5`:

```
int row;
for (row = 5; row >=1; row--)
    xv_set(panel_list_item, PANEL_LIST_DELETE, row, NULL);
```

To add to a scrolling list, starting at a particular row, you can use `PANEL_LIST_INSERT_STRINGS` or `PANEL_LIST_INSERT_GLYPHS`. `PANEL_LIST_INSERT_STRINGS` inserts strings into a specified scrolling list before a specified row. `PANEL_LIST_INSERT_GLYPHS` inserts glyphs into a specified scrolling list before a specified row.

*See Appendix D for a description of an improved list insertion method that is available in XView Version 3.2. and newer releases.

To add a new row, PANEL_LIST_INSERT is used in the same way as PANEL_LIST_DELETE. When adding a new row in this manner, all the succeeding row numbers are incremented and the list size grows by one. Space for a new item is created, and a new string or glyph may be added. These are all done at the time the attribute is evaluated, so they may be combined into one xv_set() call. You can move a row by deleting it from its old location and reassigning it to a new location. Look at the following code:

```
char *buf[128];

strcpy(buf, xv_get(list_item, PANEL_LIST_STRING, 4));

xv_set(list_item,
    PANEL_LIST_DELETE,    4,
    PANEL_LIST_INSERT,    8,
    PANEL_LIST_STRING,    8, buf,
    NULL);
```

The value for the string *must* be copied because as soon as the list item is deleted, the data is freed.

If a panel list may *not* contain duplicate entires, PANEL_LIST_INSERT_DUPLICATE needs to be set to FALSE. The default value for this attribute is TRUE, which allows duplicate strings to be inserted.

7.11.3 List Selection

Items in the list are selected by using the SELECT mouse button while pointing at an item or by dragging the pointer over the list items, or with the attribute PANEL_LIST_SELECT. When PANEL_LIST_SELECT is used to select an item that is currently visible, then the list may be scrolled when PANEL_LIST_SELECT is set. To disable the scrolling, set XV_SHOW to FALSE for the scrolling list before the specified row is selected with PANEL_LIST_SELECT.

Selected choice(s) can be set at list creation or later by using PANEL_LIST_SELECT. For example:

```
PANEL_LIST_SELECT, 3, TRUE,
```

will select row three. If the list item is nonexclusive, you can set more than one choice at one time. For example:

```
PANEL_LIST_SELECT, 3,  TRUE,
PANEL_LIST_SELECT, 13, TRUE,
PANEL_LIST_SELECT, 14, TRUE,
```

will select rows 3, 13, and 14.

To determine if a row is selected, use:

```
xv_get(list_item, PANEL_LIST_SELECTED, i);
```

This call to `xv_get()` returns `TRUE` if row *i* is selected. To return the first selected row, use `PANEL_LIST_FIRST_SELECTED` as follows:

```
int first_selected;
first_selected = (int)xv_get(list_item, PANEL_LIST_FIRST_SELECTED);
```

`PANEL_LIST_NEXT_SELECTED` returns the next selected row after a specified row. This attribute takes a single integer argument representing the row to start from. If no row is selected following the specified row, `PANEL_LIST_NEXT_SELECTED` returns -1.

7.11.4 List Notification

The procedure specified via the attribute `PANEL_NOTIFY_PROC` is called when a row is selected, de-selected, added, or deleted. List notify procedures are passed the following: the list item, the string indicating the label of the item being acted upon, any client data associated with the list entry, a parameter indicating the action being taken, the event which caused notification, and the row number of the row being operated on. The form of the procedure is:

```
int
list_notify_proc(item, string, client_data, op, event, row)
    Panel_item          item; /* panel list item */
    char                *string;
    Xv_opaque           client_data;
    Panel_list_op       op;
    Event               *event;
    int                 row; /* row number */
```

`item` is the panel list item that contains the row that was acted upon. The `string` parameter is the label of the row. If there is no string associated with the row, the parameter is `NULL`. If the row contains both a string and an image, then the string is passed as the parameter.

`op` is one of the following:

```
PANEL_LIST_OP_SELECT
PANEL_LIST_OP_DESELECT
PANEL_LIST_OP_VALIDATE
PANEL_LIST_OP_DELETE
```

If the user selects a row that is not currently selected, the notify procedure is called twice. The first time it is called with the previously selected row, the `op` is `PANEL_LIST_OP_DESELECT`. The next time the function is called, the `op` is `PANEL_LIST_OP_SELECT`. If the user selects an item that is already selected, the function is called once, passing the row selected (`op` is `PANEL_LIST_OP_SELECT`). Validate is called when the user inserts a new row (*op* is set to `PANEL_LIST_OP_VALIDATE`). The notify procedure should return `XV_OK` to accept the row, or `XV_ERROR` to reject the row. Delete is called when the user deletes a row (*op* is set to `PANEL_LIST_OP_DELETE`).

7.11.4.1 List item client data

The client_data parameter is set to whatever client data is associated with the row. *list_glyphs.c* uses the attribute PANEL_LIST_CLIENT_DATAS to assign a set of values to the list items. Each value could have been assigned to the rows one by one using the attribute PANEL_LIST_CLIENT_DATA. This attribute takes two values: the first is the number of the row to assign the data to, and the second is the data itself.

```
xv_create(panel, PANEL_LIST,
    ...
    PANEL_LIST_CLIENT_DATA, 0, "one",
    PANEL_LIST_CLIENT_DATA, 1, "two",
    PANEL_LIST_CLIENT_DATA, 2, "three",
    ...
    NULL);
```

You can still assign client data to the panel list item itself using XV_KEY_DATA as with any other XView object. However, this data can only be retrieved using xv_get() from the panel list item itself, the first parameter in the callback function.

7.11.5 The Scrolling List Menu

PANEL_ITEM_MENU sets or gets the scrolling list's read menu if the Scrolling List is in read mode or the edit menu if the Scrolling List is in edit mode. The mode of the scrolling list is set with the attribute PANEL_LIST_MODE. PANEL_LIST_MODE takes one of two values: PANEL_LIST_READ and PANEL_LIST_EDIT. The attribute PANEL_VALUE_STORED_LENGTH controls the amount of text that may be edited when in edit mode.

7.12 Message Items

Message items display a text or image message within a panel. The only visible component of a message item is the label itself. Message items are useful for annotations of all kinds, including titles, comments, descriptions, pictures and dynamic status messages. The message is often used to identify elements on the panel. A message has no value.

You may set or change the label for a message item via PANEL_LABEL_STRING or PANEL_LABEL_IMAGE. Message items can have notify procedures which are called when SELECT_UP occurs over the message item. Panel message items are the only panel items whose font can be set to boldface. The boldness of message items is controlled using the attribute PANEL_LABEL_BOLD.

Since messages cannot be selected, their primary use is for display purposes only. In Example 7-4, two message items display normal pipeline pressure, and a warning for high pipeline pressure.

Example 7-4. The message_item.c program

```
#include <xview/xview.h>
#include <xview/panel.h>

Frame         frame;
Panel_item    message;
Panel         panel;
Panel_item    slider;


void
slider_notify_proc(item, value, event)
    Panel_item     item;
    int            value;
    Event    *event;
{
    xv_set(message,
     PANEL_LABEL_STRING,
         value > 500 ? "** HIGH PIPELINE PRESSURE **" : "Okay",
     0);
}


main(argc, argv)
    int       argc;
    char  **argv;
{
    xv_init(XV_INIT_ARGS, argc, argv, 0);

    frame = xv_create(NULL, FRAME,
                FRAME_LABEL,    "Message Item",
                0);

    panel = xv_create(frame, PANEL,
     PANEL_LAYOUT, PANEL_VERTICAL,
     0);

    slider = xv_create(panel, PANEL_SLIDER,
     PANEL_LABEL_STRING, "Pipeline pressure (psi):",
     PANEL_MIN_VALUE, 0,
     PANEL_MAX_VALUE, 1000,
     PANEL_NOTIFY_PROC, slider_notify_proc,
     PANEL_SHOW_RANGE, TRUE,
     PANEL_VALUE, 100,
     0);
    message = xv_create(panel, PANEL_MESSAGE,
     PANEL_LABEL_STRING, "Okay",
     0);

    window_fit(panel);
    window_fit(frame);

    xv_main_loop(frame);
    exit(0);
}
```

Messages produced by this program are shown in Figure 7-15 and Figure 7-16.

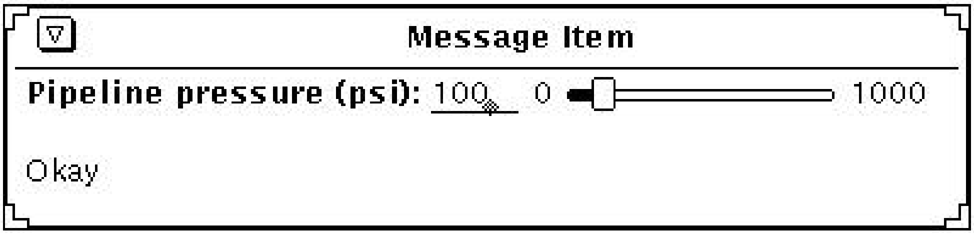


Figure 7-15. Sample panel with message item

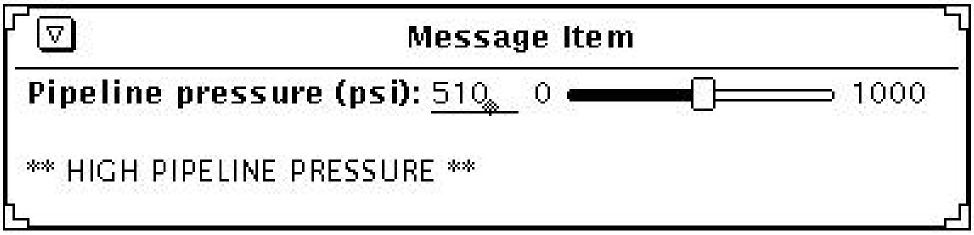


Figure 7-16. Sample panel with message item–High Pressure

7.13 Slider Items

Slider items allow the graphical representation and selection of a value within a range. Sliders are appropriate for situations where it is desired to make fine adjustments over a continuous range of values. The user selects the slider bar and drags it to the value that he wishes. A slider has the following displayable components: the label, the current value, the slider bar and the minimum and maximum allowable integral values (the range), end boxes, tick marks, tick mark minimum and maximum tick strings, as well as minimum and maximum value text strings.

Sliders may be horizontal or vertical depending on the value of `PANEL_DIRECTION`. This attribute defaults to `PANEL_HORIZONTAL`, but may be set to a vertical orientation by using the value `PANEL_VERTICAL`. The attribute `PANEL_SLIDER_END_BOXES` sets whether the boxes at the endpoints of the slider are visible. This attribute defaults to `FALSE`. The `PANEL_TICKS` attribute takes a numeric value that indicates how many evenly spaced "tick-marks" are drawn next to the item. When `PANEL_SHOW_VALUE` is `TRUE`, the current value is shown after the label in an editable text field. The minimum and maximum allowable values are set with `PANEL_MIN_VALUE` and `PANEL_MAX_VALUE`. The width of the slider bar can be adjusted using the `PANEL_SLIDER_WIDTH` attribute. When `PANEL_SHOW_RANGE` is `TRUE`, the minimum value of the slider `PANEL_MIN_VALUE` is shown to the left of the slider bar and the maximum value `PANEL_MAX_VALUE` is shown to the right of the slider bar.* The attributes `PANEL_MIN_TICK_STRING` and `PANEL_MAX_TICK_STRING` specify text labels for the minimum and maximum tick values. On horizontal sliders, these strings appear underneath the maximum and minimum tick marks. If the attribute `PANEL_SHOW_RANGE` does not adequately describe the slider values, `PANEL_MIN_VALUE_STRING` and

*The top and bottom of the slider is used when the orientation is vertical.

PANEL_MAX_VALUE_STRING can specify string value labels for the minimum and maximum slider values. On horizontal sliders, these strings appear to the left/right of the minimum/maximum end boxes.

7.13.1 Slider Selection

Only the slider bar of a slider may be selected. When the SELECT button is pressed within the slider drag box, the black area of the bar will advance or retreat with the position of the cursor. The slider value can also be changed via the numeric text field by clicking in the slider bar to the left (horizontal sliders) or below (vertical sliders) the drag box to decrement the value, or by clicking in the slider bar to the right (horizontal sliders) or above (vertical sliders) to increment the value.

7.13.2 Slider Notification

Slider notify procedures are passed the item, the item's value at time of notification, and the event which caused notification:

```
void
slider_notify_proc(item, value, event)
    Panel_item  item;
    int         value;
    Event      *event;
```

The notification behavior of a slider is controlled by the value of PANEL_NOTIFY_LEVEL. It can be set to one of two values:

PANEL_DONE	The default. The notify procedure is called only when the SELECT button is released within the panel or when the user types in a new value for the slider's numeric text field.
PANEL_ALL	The notify procedure is called whenever the value of the slider is changed; this includes when the user selects, drags or releases the SELECT button in a slider. For each movement of the mouse while dragging the slider drag box, the slider's notify procedure is called.

7.13.3 Slider Value

The value of a slider is an integer in the range PANEL_MIN_VALUE to PANEL_MAX_VALUE. You can retrieve or set a slider's value with the attribute PANEL_VALUE and the functions xv_set() or xv_get().

The following code fragment produces a slider with a label:

```
    xv_create(panel, PANEL_SLIDER,
        PANEL_LABEL_STRING,    "Brightness: ",
        PANEL_VALUE,           75,
        PANEL_MIN_VALUE,       0,
        PANEL_MAX_VALUE,       100,
        PANEL_SLIDER_WIDTH,    300,
        PANEL_TICKS,           5,
        PANEL_NOTIFY_PROC,     brightness_proc,
        NULL);
```

The output is shown in Figure 7-17.

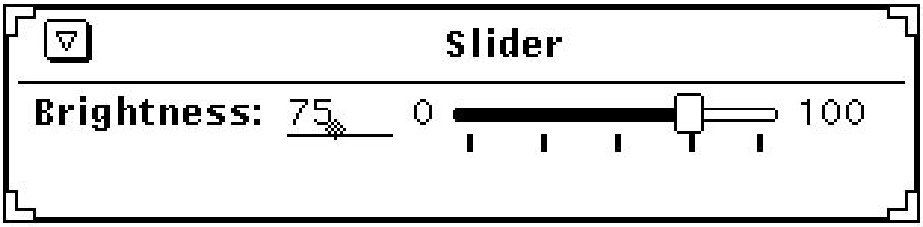


Figure 7-17. Sample panel with slider item

7.14 Gauges

Gauges are just like sliders, but they are “output only” items. That is, you set the value of the item and the display of the gauge changes just as it would for sliders. Also, there is no optional type-in field and there is no drag box for the user to interactively change the value of the gauge. The gauge is intended to be used only as a feedback item.

To create a gauge, use the `PANEL_GAUGE` package. To set a gauge’s width or height, use `PANEL_GAUGE_WIDTH`. This attribute sets the length of the object, whether it is vertically or horizontally oriented. As with the slider, the orientation is set by the attribute `PANEL_DIRECTION`.

7.15 Text Items

A panel text item contains as its value a `NULL`-terminated string. It contains only printable characters with no newlines. When a panel receives keyboard input (regardless of where the pointer is as long as it is within the boundaries of the panel), the keyboard event is passed to the item with the keyboard focus. A caret is used to indicate the insertion point where new text is added. You can type in more text than fits on the text field. If this happens, a right arrow pointing to the left will appear on the left on the field, indicating that some text to the left of the displayed text is no longer visible. Similarly, if text is inserted causing text on the right to move out of the visible portion of the text item, then an arrow pointing to the right will appear to the right of the text.

Text items use the attribute `PANEL_LABEL_STRING` as do most other panel items, to label the text item. The value of a text item is also a string, an is set by the attribute `PANEL_VALUE`, as shown by the following code:

```
xv_create(panel, PANEL_TEXT,
    PANEL_LABEL_STRING,  "Name:",
    PANEL_VALUE,         "Edward G. Robinson",
    NULL);
```

The output from this code fragment is shown in Figure 7-18.

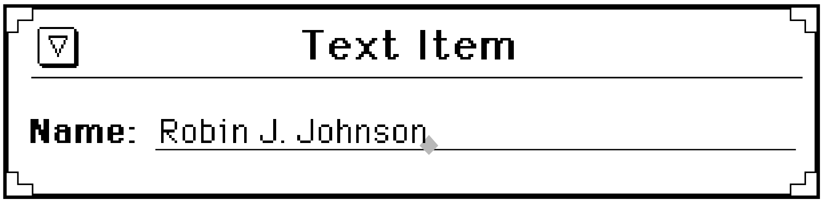


Figure 7-18. Sample panel with text item

If the item's `PANEL_LAYOUT` is set to `PANEL_VERTICAL`, the value is placed below the label. The default is `PANEL_HORIZONTAL`.

The number of characters of the text item's value that are displayed is set via `PANEL_VALUE_DISPLAY_LENGTH`. Note that the length of the value display specified by `PANEL_VALUE_DISPLAY_LENGTH` may not be less than the combined width of the left and right "more text" buttons. In 12-point font, this is four characters.

`PANEL_VALUE_DISPLAY_LENGTH` is useful for fixed width characters. `PANEL_VALUE_DISPLAY_LENGTH` converts characters to pixels by multiplying the number of characters by the default character width for the font being used. To set the number of characters in the text item's value that are displayed in a variable width font, use `PANEL_VALUE_DISPLAY_WIDTH`. The argument for `PANEL_VALUE_DISPLAY_WIDTH` is expressed in pixels instead of characters. The maximum number of characters that can be typed into a text item (independently of how many are displayable) is set via the attribute `PANEL_VALUE_STORED_LENGTH`. When characters are entered beyond the display length, and when `PANEL_VALUE_STORED_LENGTH` is greater than the display length, the value string is scrolled one character to the left so that the most recently entered character is always visible. As the string scrolls to the left, the leftmost characters move out of the visible display area. The presence of these temporarily hidden characters is indicated by a small left-pointing triangle.

It is sometimes desirable to have a protected field where the user can enter confidential information. The attribute `PANEL_MASK_CHAR` is provided for this purpose. When the user enters a character, the character specified as the value of `PANEL_MASK_CHAR` will be displayed in place of the character the user has typed. Setting `PANEL_MASK_CHAR` to an asterisk (*) would produce a string of asterisks instead of the characters typed. The value of the text is still the string the user types.

If you want to disable character echo entirely so that the caret does not advance and it is impossible to tell how many characters have been entered, use the space character as the mask. You can remove the mask and display the actual value string at any time by setting the mask to `NULL`.

7.15.1 The Current Keyboard Focus

A panel may have several keyboard focus items that can accept keyboard input. Only one of these items may be *current* at any given time. The current keyboard focus item is the one to which keyboard input is directed and is indicated by a caret at the item's value. Selection of a keyboard focus item (i.e., pressing and releasing the SELECT mouse button anywhere within the item's bounding box) causes that item to become the current keyboard focus item.

NOTE

> When the resource `OpenWindows.KeyboardCommands` is set to SunView1 or Basic, only text items will receive the keyboard focus within a panel.

You can find out which keyboard focus item has the caret or give the caret to a specified keyboard focus item by means of the panel attribute `PANEL_CARET_ITEM`:

```
current_item = (Panel_item)xv_get(panel, PANEL_CARET_ITEM);
xv_set(panel, PANEL_CARET_ITEM, another_item, NULL);
```

You can set the current item to the next or previous keyboard focus item in the panel by using the following two routines:

```
Panel_item
panel_advance_caret(panel)
    Panel panel;

Panel_item
panel_backup_caret(panel)
    Panel panel;
```

They return the new panel item that has received the keyboard focus. Advancing past the last keyboard focus item places the keyboard focus at the first keyboard focus item, while backing up past the first keyboard focus item places the keyboard focus at the last keyboard focus item.

7.15.2 Text Selection

You can use the attribute PANEL_TEXT_SELECT_LINE to select and highlight the entire contents of the text field.

7.15.3 Text Notification

The notification behavior of text items is more complex than that of other item types. You can control whether your notify procedure is called on each input character or only on selected characters.

When your notify procedure will be called is determined by the value of PANEL_NOTIFY_LEVEL. Possible values are given in Table 7-1.

Table 7-1. Text Item Notification Level

Notification Level	Level Causes Notify Procedure to be Called . . .
PANEL_NONE	Never
PANEL_NON_PRINTABLE	On each non-printable input character.
PANEL_SPECIFIED	If the input char is found in the string given for the attribute PANEL_NOTIFY_STRING.
PANEL_ALL	On each input character.

The default for PANEL_NOTIFY_LEVEL is PANEL_SPECIFIED, and the default for PANEL_NOTIFY_STRING is \n\r\t (i.e., notification on line-feed, carriage-return and tab). The value PANEL_SPECIFIED only works for characters that do *not* map to semantic actions. To provide notification for characters that map to semantic actions, such as 177 (delete normally maps to ACTION_ERASE_CHAR_BACKWARD), use either PANEL_EVENT_PROC, notify_interpose_event_func(), or set PANEL_NOTIFY_LEVEL to PANEL_ALL.

The user's editing characters are treated specially (for example the backspace character). They are *not* appended to the value string. If you have asked for a character by including it in PANEL_NOTIFY_STRING, the PANEL package calls your notify procedure. After the notify procedure returns, the appropriate editing operation will be applied to the value string. (Note that the editing characters are never appended to the value string, regardless of the return value of the notify procedure.)

Characters other than the special characters described above are treated as follows. If the character typed by the user does *not* result in your notify procedure getting called, then the character, if printable, is appended to the value string. If it is not printable, it is ignored. If your notify procedure *is* called, what happens to the value string and whether the keyboard focus moves to another item is determined by the notify procedure's return value. Table 7-2 shows the possible return values.

Table 7-2. Return Values for Text Item Notify Procedures

Value Returned	Action Caused
`PANEL_INSERT`	Character is appended to item's value.
`PANEL_NEXT`	Keyboard focus moves to next text item.
`PANEL_PREVIOUS`	Keyboard focus moves to previous text item.
`PANEL_NONE`	Ignore the input character.

If you do not specify your own notify procedure, the default procedure, `panel_text_notify()`, is called at the appropriate time as determined by the setting of `PANEL_NOTIFY_LEVEL`.

7.15.4 Writing Your Own Text Notify Procedure

By writing your own notify procedure, you can tailor the notification behavior of a given keyboard focus item to support the needs of any application. At one extreme, you may want to process each character as the user types it in. For a different application, you may not care about the characters as they are typed in and may only want to look at the value of the string in response to some other button. A typical example is getting the value of a filename field when the user presses a **Load File** panel button.

The form of the text notification procedure is:

```
Panel_setting
panel_text_notify(item, event)
    Panel_item  item;
    Event      *event;
```

This procedure returns a panel setting enumeration that has the following effects:

`PANEL_NONE`	Do not advance the keyboard focus to the next keyboard focus item. The current keyboard focus item and insertion point remain unchanged.
`PANEL_NEXT`	The keyboard focus moves to the *next* keyboard focus item (defined by the keyboard focus item that was created *after* the current keyboard focus item). If there is no next text item, the first keyboard focus item in the panel is used.
`PANEL_PREVIOUS`	The keyboard focus moves to the *previous* keyboard focus item in the panel (defined by the keyboard focus item that was created *before* the current keyboard focus item). If there is no previous keyboard focus item, the last keyboard focus item in the panel is used.
`PANEL_INSERT`	The character which caused the notification procedure to be called is inserted into the text item's value at the location of the caret (insertion point).

7.15.5 Text Value

You can set or get the value of a keyboard focus item at any time via PANEL_VALUE. The following call retrieves the value of name_item into name:

```
Panel_item name_item;
char name[NAME_ITEM_MAX_LENGTH];
        ...
strcpy(name, (char *)xv_get(name_item, PANEL_VALUE));
```

Note that name_item should have been created with a PANEL_VALUE_STORED_LENGTH not greater than NAME_ITEM_MAX_LENGTH so the buffer name will not overflow.

7.16 Numeric Text Items

Panel numeric text items are virtually the same as panel text items except that the value displayed is of type int. Also, convenience features (such as increment and decrement buttons) ease the manipulation of the text string's numeric value. There is little programmatic difference between the text item and the numeric text item. You create a numeric text item using the PANEL_NUMERIC_TEXT package. You can also set the minimum and maximum range for the numeric text field by using PANEL_MIN_VALUE and PANEL_MAX_VALUE, respectively.

7.17 Multiline Text Items

Multiline text items are a special type of panel text item that allow a text field to display multiple lines. You create a multiline text item using the PANEL_MULTILINE_TEXT package. Multiline text items use the attribute PANEL_DISPLAY_ROWS to specify the number of rows of text to display. PANEL_VALUE_DISPLAY_LENGTH specifies the length in characters of a row in a multiline text field. PANEL_VALUE_DISPLAY_WIDTH specifies the length in pixels of each row. A multiline text item will have scrollbars if the stored length is greater than the displayed length (rows $\times$ columns). The maximum stored length for a multiline text item is specified using PANEL_VALUE_STORED_LENGTH.* If PANEL_LINE_BREAK_ACTION is PANEL_WRAP_AT_CHAR, the lines wrap as soon as the number of characters on a line exceeds the length of the line. If this attribute is PANEL_WRAP_AT_WORD, the lines in the multiline text item wrap only at word breaks.

* XView Version 3 *only* supports multiline text items with scrollbars.

NOTE

> Multiline text items display multiple rows, or “lines,” but do not contain embedded returns or line feeds.

Example 7-5 shows how to create a multiline text item with scrollbars.

Example 7-5. The multiline.c program

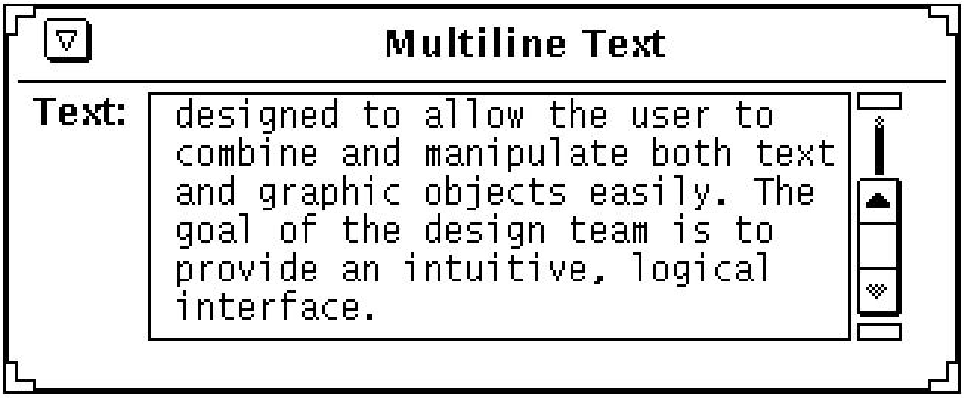
```
/*
 * multiline.c -- simple panel multiline text item example.
 */
#include <xview/xview.h>
#include <xview/panel.h>

Frame frame;
Panel panel;

main(argc, argv)
    int argc;
    char **argv;
{

    xv_init(XV_INIT_ARGS, argc, argv, NULL);

    frame = xv_create(NULL, FRAME, NULL);
    panel = xv_create(frame, PANEL, NULL);
    xv_create(panel, PANEL_MULTILINE_TEXT,
        PANEL_LABEL_STRING, "Product Description:",
        PANEL_DISPLAY_ROWS, 6,
        PANEL_VALUE_DISPLAY_LENGTH, 32,
        PANEL_VALUE, "This wonderful product is \
designed to allow the user to combine and manipulate both \
text and graphic objects easily. The goal of the design team \
is to provide an intuitive, logical interface.",
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}
```

The output of this code fragment is shown in Figure 7-19.

7.18 Drop Target Items

A panel *drop target item* is a bordered image in a panel area that is used to transfer data between applications. Before you use a panel drop target item you need to be familiar with the SELECTION and DRAGDROP packages; these are described in Chapter, 18, *Selections*, and Chapter 19, *Drag and Drop*.

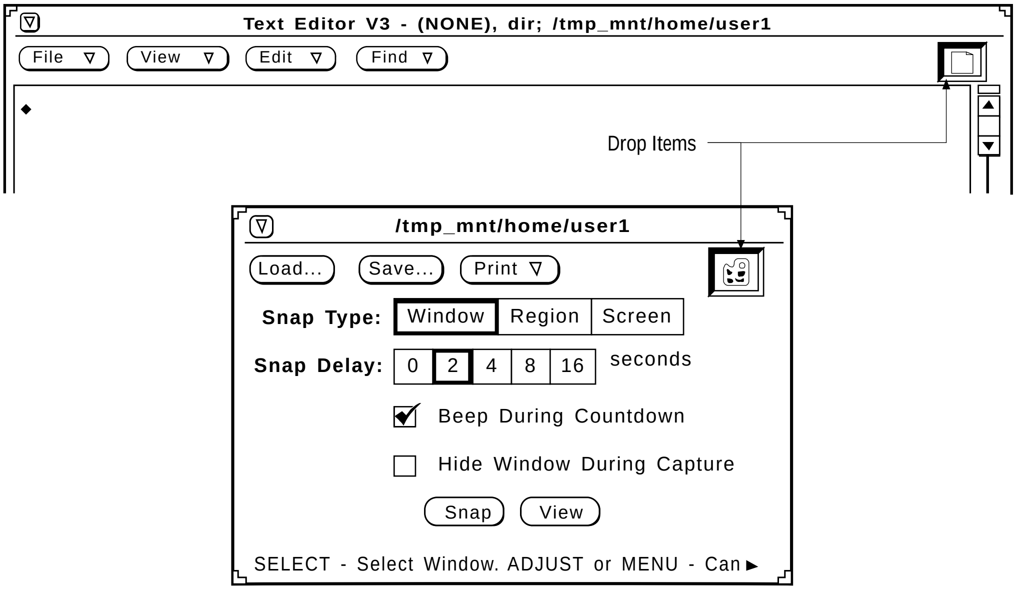
A panel drop target item is an object in the class `Panel_drop_target_item` which is equivalent to a `Panel_item`. A drop target item's owner is a `Panel`. Examples of several drop target items are shown in Figure 7-20.



designed to allow the user to combine and manipulate both text and graphic objects easily. The goal of the design team is to provide an intuitive, logical interface.

Figure 7-19. Panel multiline text item

To use the PANEL_DROP_TARGET package in an application, you need to include both the *<xview/dragdrop.h>* and *<xview/svrimage.h>* header files. The program *panel_dnd.c*, showing a panel drag and drop example, is presented in Appendix F, *Example Programs.*



SELECT - Select Window. ADJUST or MENU - Can

Figure 7-20. Sample panel with drop target items

7.18.1 Programming a Panel Drop Target Item

To use a panel drop target item follow these programming steps:

- Create the drop target item.
- Specify the glyphs.
- Create drag and drop object.
- Define the drop target item's requestor.
- Control the glyphs.
- Drop on the target item.
- Drag from the drop target item.

7.18.1.1 Create the drop target item

You create a panel drop target using `xv_create()` with the `PANEL_DROP_TARGET` package. The following code fragment shows a how to create a drop target item called `drop_target`:

```
Panel_drop_target_item       drop_target;
    drop_target = xv_create(panel, PANEL_DROP_TARGET, NULL);
```

7.18.1.2 Specify the glyphs

The attribute `PANEL_DROP_GLYPH` specifies the glyph for a "normal" drop target. The normal glyph is shown when the drop target item is inactive (no data transfer is occurring). `PANEL_DROP_BUSY_GLYPH` specifies the glyph for the "busy" drop target. The busy drop target glyph is displayed when the drop target item is receiving a drop, or when data is being sent to another application. You can define the bits for the glyphs as follows (assuming the files *normal.icon* and *busy.icon* contain appropriate data):

```
static unsigned short normal_bitmap[ ] = {
#include "normal.icon"
};

static unsigned short busy_bitmap[ ] = {
#include "busy.icon"
};
```

Once the bits are defined, you create server images to represent them as follows:

```
normal_glyph = xv_create(NULL, SERVER_IMAGE,
    XV_HEIGHT, 64,
    XV_WIDTH, 64,
    SERVER_IMAGE_DEPTH, 1,
    SERVER_IMAGE_BITS, normal_bitmap,
    NULL);
```

```
busy_glyph = xv_create(NULL, SERVER_IMAGE,
    XV_HEIGHT, 64,
    XV_WIDTH, 64,
    SERVER_IMAGE_DEPTH, 1,
    SERVER_IMAGE_BITS, busy_bitmap,
    NULL),
```

Set the panel drop target attributes to use the server images defined above:

```
xv_set(drop_target,
    PANEL_DROP_GLYPH, normal_glyph,
    PANEL_DROP_BUSY_GLYPH, busy_glyph,
    NULL);
```

7.18.1.3 Create the drag and drop object

If the drop target item will support drags, create a `Drag_drop` object and set the attribute `PANEL_DROP_DND`. This attribute is the `DRAGDROP` object associated with the panel drop target item. The `Drag_drop` object is used to initiate a drag and drop operation. If `PANEL_DROP_DND` does not exist, then the panel drop target item will not support drags and is called an *empty* drop target. In this case, `PANEL_DROP_FULL` will be `FALSE` (the default).

```
Drag_drop dnd;
    dnd = xv_create(panel, DRAGDROP, NULL);
```

If the drop target item will support drags, then set the appropriate attributes on the `Dnd` object (rank, cursor, etc.).

Create a selection item associated with the `Dnd` object. This defines the data and the conversion(s) supported for the source of the drag. For example:

```
xv_create(dnd, SELECTION_ITEM,
        SEL_DATA, "dnd selection data",
        NULL);
```

After data is defined, you need to set `PANEL_DROP_FULL` to `TRUE`. When set to `TRUE`, `PANEL_DROP_FULL` indicates that valid, “draggable” data is set on the `PANEL_DROP_DND` object’s selection items. For example:

```
xv_set(drop_target,
    PANEL_DROP_DND,  dnd,
    PANEL_DROP_FULL, TRUE,
    NULL);
```

7.18.1.4 Define the drop target item’s requestor

The panel package creates a selection requestor, from the `SELECTION_REQUESTOR` package that is associated with each drop target item. This selection requestor’s attributes need to be set. `PANEL_DROP_SEL_REQ` returns the `SELECTION_REQUESTOR` associated with the panel drop target item. The following code fragment gets the selection requestor associated with an item:

```
Selection_requestor sel_req;
```

```
        sel_req = xv_get(item, PANEL_DROP_SEL_REQ);
```

7.18.1.5 Controlling the glyphs

The drop target item package handles this step. When something is dragged into the drop target box, the busy glyph is displayed. This action is initiated when an ACTION_DRAG_PREVIEW semantic action and LOC_WINENTER event id combination is received on the drop target box. When the cursor is dragged out of the drop target box, the glyph changes back to the normal state. An ACTION_DRAG_PREVIEW semantic action and LOC_WINEXIT event id combination on the drop target box initiates this.

7.18.1.6 Dropping on the drop target

When something is dropped on the drop target item, the panel package calls dnd_decode_drop(). This action is initiated by an ACTION_DRAG_COPY or ACTION_DRAG_MOVE. When dnd_decode_drop() returns, the panel drop target item's notify procedure is called. For details, see Section 7.18.2, "Drop Target Notification."

7.18.1.7 Dragging from the drop target item

When SELECT is pressed while over the drop target item, if PANEL_DROP_FULL is TRUE, dnd_send_drop() is called. At this time the glyph is changed to its busy state. When dnd_send_drop() returns, the panel drop target item's notify procedure is called.

7.18.2 Drop Target Notification

When the user drops an item on the panel drop target item, the item and the event that initiated the drop are passed to the notify procedure. The form of a panel-drop-target notify procedure is:

```
int
drop_target_notify_proc(item, value, event)
   Panel_drop_target_item  item;
   int                     value
   Event                  *event;
```

The *item* is the panel drop target that was dropped on. The argument *value* contains the return value from dnd_decode_drop(). The *event* is the event which initiates the drop (i.e., ACTION_DRAG_COPY or ACTION_DRAG_MOVE).

When the notify procedure returns, the glyph is returned to its normal state. The notify procedure returns one of the following: XV_OK or XV_ERROR. When the notify procedure returns, one of the following actions occurs:

1. If the return value is XV_OK, then the function dnd_done() is called.
2. If the return value is XV_ERROR, then the function dnd_done() is *not* called.

7.19 Advanced Panel Usage

The following sections address some advanced topics dealing with panels. They cover attaching data to panel items, repainting panels and panel items, and handling events in panels and in panel items. Handling panel repainting and panel events are features which are available but are not generally used by most applications. Attaching data to panel items should be a practice closely followed for more efficient programs.

7.19.1 Attaching Data to Panel Items

Callback routines are called separately and independently from the application's main routine. If the callback routine needs data, there are two ways to make it available. One way is to store the data in global variables or data structures so the callback routine can reference the data. Another way is to attach the data directly to panel items (whose handle has already been retrieved by the notification procedure).

In the spirit of good programming practice, it is wise to avoid creating global variables. The preferred method for making data available to callback routines is to *attach* the data to the panel items. A handle to the panel item is already made available to the callback function as the first parameter to the function. Two attributes can be used to attach data to panel items: `XV_KEY_DATA` or `PANEL_CLIENT_DATA`.*

Using `XV_KEY_DATA` requires a *key*, which must be some arbitrary, but unique, integer. An example of usage follows:

```
static int My_item_key = 0;
 ...
if (!My_item_key) My_item_key =  xv_unique_key();
xv_create(panel, PANEL_BUTTON,
    PANEL_BUTTON_LABEL,     "Push Me",
    XV_KEY_DATA,            My_item_key, "text",
    PANEL_NOTIFY_PROC,      my_notify_proc,
    ...
    NULL);
```

If this method is used, `my_notify_proc()` retrieves the data using `xv_get()`:

```
char *data = (char *)xv_get(item, XV_KEY_DATA, My_item_key);
```

The common element in this case is the use of `My_item_key`. A `static int` need not be used, but it is better than creating a global `int` variable to store the key. If that were the case, you might as well make the *data* portion that you wanted to attach to the panel item a global variable.

One advantage to using `XV_KEY_DATA` is that you can specify any number of keys and attach as many pieces of data to objects as you like.

*`XV_KEY_DATA` can be used with any XView object. Client data can as well, but the name to use varies with the object package, e.g., `WIN_CLIENT_DATA` is for windows, `MENU_CLIENT_DATA` is for menus, etc.

An alternate method for attaching data to panel items is to use PANEL_CLIENT_DATA. This attribute is similar to XV_KEY_DATA in that the data is attached to the item, but in this case, you can only attach one piece of data to the panel item. You can, however, use *both* XV_KEY_DATA and PANEL_CLIENT_DATA on the same panel item. The advantage to using PANEL_CLIENT_DATA is that you do not have to keep track of a key. Since you can only attach one piece of data to a panel item, xv_get() returns only that data.

For a real example of this situation, let's modify the *quit.c* program (from the beginning of this chapter), which displays a frame, panel and a panel item labeled Quit. The modified program, *client_data.c*, is shown in Example 7-6. Selecting the panel button exits the program gracefully. The call to xv_destroy_safe(frame) causes the frame to be destroyed and thus, xv_main_loop() returns and the program exits.

Example 7-6. The client_data.c program

```
/*
 * client_data.c -- demonstrate the use of PANEL_CLIENT_DATA attached
 * to panel items.  Attach the base frame to the "Quit" panel item so
 * that the notify procedure can call xv_destroy_safe() on the frame.
 */
#include <xview/xview.h>
#include <xview/panel.h>

main(argc, argv)
int argc;
char *argv[];
{
    Frame  frame;
    Panel  panel;
    int    quit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      quit,
        PANEL_CLIENT_DATA,      frame,
        NULL);

    xv_main_loop(frame);
    puts("The program is now done.");
    exit(0);
}

quit(item)
Panel_item item;
{
    Frame frame = (Frame)xv_get(item, PANEL_CLIENT_DATA);
    xv_destroy_safe(frame);
}
```

In this program, the frame object is not a global variable; it is a local, or *automatic*, variable. Since there is a close association between the panel button and the frame (meaning that the panel button is going to access the handle to the frame in the callback routine), we attach the frame to the panel button as *client data*. PANEL_CLIENT_DATA takes a generic address of type caddr_t. In the callback routine for the panel button, the first parameter to the callback function is the panel item that called the notification. From that handle, the frame is retrieved via xv_get().

This worked because the frame was created via xv_create(). That is, the object was *allocated*. You cannot attach data that has not been allocated. Thus, the following code segment is not advised:

```
dummy_function()       /* Wrong way */
{
    char *home = (char *)getenv("HOME");
    xv_set(panel_item, XV_KEY_DATA, HOME_KEY, home, NULL);
}
```

The reason for this is that getenv() returns a pointer to static data that is *overwritten on each call*. The next call that the application makes to getenv() will change the value for the panel item XV_KEY_DATA. Likewise, the following should not be used:

```
dummy_function()     /* Also wrong */
{
    char home[MAXPATHLEN], *ptr;
    if ((ptr = (char *)getenv("HOME")) != NULL) {
        (void) strcpy(home, ptr);
        xv_set(panel_item, XV_KEY_DATA, HOME_KEY, home, NULL);
    }
}
```

This does not work because as soon as dummy_function() returns, the value of home is lost because it is an automatic variable. The correct way to handle this is to make home either a static variable or a pointer whose storage is allocated via malloc().

The problem with home being static is that if dummy_function() is called more than once, the value of home will be overwritten on each call. So, the *best* way to handle this case is to allocate the data:

```
dummy_function()     /* Best way */
{
    extern char *malloc(), *getenv();
    char *home, *ptr;
    if ((ptr = getenv("HOME")) != NULL &&
        (home = malloc(strlen(ptr)+1))) {
        (void) strcpy(home, ptr);
        xv_set(panel_item, XV_KEY_DATA, HOME_KEY, home, NULL);
    }
}
```

Having allocated data for XV_KEY_DATA, we now assume the responsibility of freeing that data when the object is destroyed. Otherwise, the data is left free with no references to it. This is also known as creating a *memory leak*. Because you don't always know when an object is being destroyed (destroying a panel may or may not cause a panel item to be destroyed), you can specify a function that explicitly frees the data pointed to by XV_KEY_DATA. To do this, use the attribute XV_KEY_DATA_REMOVE_PROC:

```
dummy_function()
{
    extern void free_data();
```

```
        extern char *malloc(), *getenv();
        char *home, *ptr;
        if ((ptr = getenv("HOME")) != NULL &&
            (home = malloc(strlen(ptr)+1))) {
            (void) strcpy(home, ptr);
            xv_set(panel_item,
                XV_KEY_DATA,             HOME_KEY, home,
                XV_KEY_DATA_REMOVE_PROC, HOME_KEY, free_data,
                NULL);
        }
    }

    void
    free_data(object, key, data)
    Xv_object object;
    int       key;
    caddr_t   data;
    {
        free(data);
    }
```

Whenever an object is freed, all the "key data" objects are scanned. If "remove procedures" are associated with them, they are called with the key data as the parameter. In this case, `free_data()` is called, which frees the data associated with that particular key. The "remove procedure" is called only after the object has been completely destroyed; therefore, there should be no attempt to access the destroyed object in the procedure.

If you need to assign a new key to the *same* key data of an object, the old key data is automatically freed by XView by calling the remove procedure (if it exists). If you wish to delete a key without having to assign a new key, then you can call:

```
    xv_set(object, XV_KEY_DATA_REMOVE, key, NULL);
```

7.19.2 Using PANEL_REPAINT_PROC

The PANEL package provides an property for installing a repaint routine. Warning: use of the repaint routine allows you to generate a **Non-OPEN LOOK** user interface.

Example 7-7 demonstrates how a repaint routine can be installed on a panel. This repaint routine draws a gray background on the panel behind any existing panel items.

Example 7-7. The panel_repaint.c program

```
/*
 * panel_repaint.c -- repaint a panel background without disturbing
 * the repainting of panel items.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/svrimage.h>
#include <X11/Xlib.h>
#include <X11/X.h>
#include <X11/bitmaps/gray1>
```

Example 7-7. The panel_repaint.c program (continued)

```
#define PANEL_GC_KEY    101  /* any arbitrary number */

main(argc, argv)
int argc;
char *argv[ ];
{
    Display     *display;
    Frame        frame;
    Panel        panel;
    int          quit();
    void         panel_repaint();
    XGCValues    gcvalues;
    Server_image grey;

    Mask        gcmask = 0L;
    GC          gc;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_REPAINT_PROC,     panel_repaint,
        NULL);

    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      quit,
        PANEL_CLIENT_DATA,      frame,
        NULL);

    window_fit(frame);

    grey = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               gray1_width,
        XV_HEIGHT,              gray1_height,
        SERVER_IMAGE_DEPTH,     1, /* clarify for completeness*/
        SERVER_IMAGE_BITS,      gray1_bits,
        NULL);

    display = (Display *)xv_get(panel, XV_DISPLAY);
    gcvalues.stipple = (Pixmap) xv_get(grey, XV_XID);
    gcvalues.fill_style = FillOpaqueStippled;
    gcvalues.plane_mask = 1L;
    gcvalues.graphics_exposures = False;
    gcvalues.foreground = BlackPixel(display, DefaultScreen(display));
    gcvalues.background = WhitePixel(display, DefaultScreen(display));
    gcmask = GCStipple | GCFillStyle | GCPlaneMask |
        GCGraphicsExposures | GCForeground | GCBackground;
    gc = XCreateGC(display, xv_get(panel, XV_XID), gcmask, &gcvalues);

    /* attach the GC to the panel for use by the repaint proc above */
    xv_set(panel, XV_KEY_DATA, PANEL_GC_KEY, gc, NULL);

    xv_main_loop(frame);
    exit(0);
}
```

Example 7-7. The panel_repaint.c program (continued)

```
/*
 * repaint procedure for the panel paints a gray pattern over the
 * entire panel.  Use the GC attached to the panel via XV_KEY_DATA.
 */
void
panel_repaint(panel, pw, p_area)
Panel      panel;
Xv_Window  pw;
Rectlist   p_area;
{
    /* get the GC attached to the panel in main() */
    GC gc = (GC)xv_get(panel, XV_KEY_DATA, PANEL_GC_KEY);

    /* call XFillRectangle on the entire size of the panel window */
    XFillRectangle(xv_get(panel, XV_DISPLAY), xv_get(pw, XV_XID), gc,
        0, 0, xv_get(pw, XV_WIDTH), xv_get(pw, XV_HEIGHT));

    /* Note this repaints the entire panel.  It is best to  */
    /* repaint just the rectangles passed in p_area         */

}

quit(item)
Panel_item item;
{
    Frame frame = (Frame)xv_get(item, PANEL_CLIENT_DATA);
    xv_destroy_safe(frame);
}
```

The output produced by this program is shown in Figure 7-21.

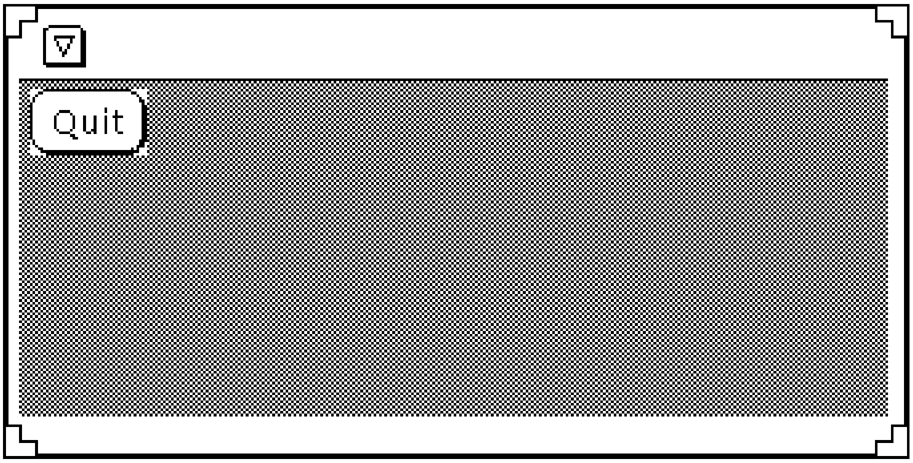


Figure 7-21. Panel with gray background

The PANEL package does not retain its paint windows by default, so the repaint routine may be called more frequently than one might expect. Therefore, when the panel is created, the attribute WIN_RETAINED may be set to TRUE; otherwise, the routine should try to be as computationally cheap as possible to maintain good performance. It is not recommended that you retain the panel's window unless you have provided a repaint routine that might utilize graphics expensively. Typically, you will set the background to a solid color, render a pattern, or display an image.

If a panel item is added, deleted or moved, then the repaint routine is called regardless of whether or not the panel's window is retained.

7.19.3 Painting Panel Items

To repaint either an individual item or an entire panel, use:

```
panel_paint(panel_object, paint_behavior)
    Panel_item    panel_object;
    Panel_setting paint_behavior;
```

The panel_object may be a panel item or a panel itself. If it is a panel, the items within the panel are repainted one by one. The argument paint_behavior is either PANEL_CLEAR, which causes the rectangle occupied by the panel or item to be cleared prior to repainting, or PANEL_NO_CLEAR, which causes repainting to be done without any prior clearing. This setting will override the default paint behavior set in the panel item's PANEL_PAINT attribute.

7.19.4 Panel Event Handling

This section describes how the PANEL package handles events. If you require a behavior not provided by default, you can write your own event handling procedure for either an individual item or the panel as a whole. The default behavior for handling panel events conforms to OPEN LOOK and should be sufficient for most users. This section is intended *only* for expert users.

WARNING

Changing the default PANEL package event handling behavior allows you to create applications that are *not* OPEN LOOK-compliant.

The default event handling mechanism for panels processes events for all the panel items in a uniform way. A single routine reads the events, updates an internal state machine, and maps the event to an *action* to be taken by the item. Actions fall into two categories: *previewing* and *accepting*. The previewing action gives the user visual feedback indicating what will happen when the mouse button is released. The accepting action causes the item's value to be changed and/or its notify procedure to be called, with the event passed as an argument.

The default event-to-action mapping is given in Table 7-3.

Table 7-3. Default Event to Action Mapping

Event	Action
SELECT button down or drag with SELECT button down.	Begin previewing.
Drag with SELECT button down.	Update previewing.
Drag out of item rectangle with SELECT button down.	Cancel preview.
SELECT button up	Accept.
MENU button down	Display menu & accept user's selection.
Keystroke	Accept keystroke if text item.

What actually happens when an item is told to perform one of the above actions depends on the type of item. For example, when asked to begin previewing, a button item inverts its label, a message item does nothing, a slider item redraws the shaded area of its slider bar, etc.

ASCII events and some `action` events (described in Chapter 6, *Handling Input*) are automatically redirected towards the item with keyboard focus. Since only one item at a time may receive keyboard events, if there is more than one item in the panel that can receive keyboard input, the one that is receiving the keyboard events has a solid location cursor (a small or large solid triangle, also referred to as a caret). You may use `PANEL_CARET_ITEM` with `xv_set()` or `xv_get()` to set or get the item that currently has the keyboard focus.

Handling events by the application in panels is a task best avoided since the panel does this automatically. But there are certainly situations where the application might like to supervise or handle events itself. In such situations, there are several methods available for event handling. You can use:

- `notify_interpose_event_func()`
- `PANEL_BACKGROUND_PROC`
- `PANEL_EVENT_PROC`

For normal panels, each of these methods should be used on the panel itself. However, for the `SCROLLABLE_PANEL`, the event handler must be set on the panel's paint window(s) exactly as is done for canvases (using the attribute `WIN_EVENT_PROC`).

Each of these methods works somewhat differently from one another, but they all have one thing in common: they are notified when events happen in panels.

7.19.5 Using an Interpose Function

The Notifier's interpose functions may be installed on panels just as they are for any other window-based package. This is the recommended method for special event handling for panels and for panel items, since it allows you to interfere with the normal event processing for the destination panel (or panel item). However, events can continue to be dispatched to the panel (panel item) through the use of the `notify_next_event_func()`. Refer to Chapter 20, *The Notifier*, for details on interposition.

7.19.6 Using PANEL_BACKGROUND_PROC

The `PANEL_BACKGROUND_PROC` is similar to the `WIN_EVENT_PROC` except that the notification routine is only notified of events that do not happen in, or are redirected to, any panel items. The application would want to know about events that are not sent to panel items:

```
extern void my_event_proc();

panel = (Panel)xv_create(frame, PANEL,
    PANEL_BACKGROUND_PROC,   my_event_proc,
    NULL);
```

The parameters to the routine for `PANEL_BACKGROUND_PROC` are:

```
void
my_event_proc(panel, event)
    Panel  panel;
    Event *event;
```

The `PANEL_BACKGROUND_PROC` does not, by default, get keyboard events passed to it. Therefore, rather than trying to set this mask explicitly in the panel's window, the attribute `PANEL_ACCEPT_KEYSTROKE` can be set to `TRUE`. With this attribute set, ASCII events and function-key events are passed to the routine, provided there are no panel items that accept keyboard input. If there are such items, those panel items will continue to get keyboard events regardless of the attribute `PANEL_ACCEPT_KEYSTROKE`. If you wish to get keyboard events instead of the panel items that consume those events, you should use an event interposing function discussed in Chapter 20.

7.19.7 Using PANEL_EVENT_PROC

Just as `PANEL_BACKGROUND_PROC` specifies a routine to handle events that happen outside of panel items, you can also get events that happen only *within* panel items using `PANEL_EVENT_PROC`.

```
xv_set(panel, PANEL_EVENT_PROC, my_event_proc, NULL);
```

Using this routine causes the default event handling for that item to be ignored in favor of the new event procedure. In other words, this routine *does* interfere with the normal event processing for panel items, and the panel item's callback routine is no longer automatically called by the `PANEL` package.

Applications can get the event handler routine for a panel item with the following call:

```
Panel_item item;
event_proc = (void (*)())xv_get(item, PANEL_EVENT_PROC);
```

You may not assume that the default event handler for any panel item is `panel_default_handle_event()`, as in previous XView versions. You must `xv_get()` the item's `PANEL_EVENT_PROC`. The event handler has the following parameters:

```
void
panel_item_event_proc(item, event)
    Panel_item   item;
    Event       *event;
```

7.19.8 Event Handling Example

The program *item_move.c* (a longer program listed in Appendix F, *Example Programs*) demonstrates how events can be handled in panels. The program allows you to create, destroy, and move around three different types of panel items. Two panels are displayed (see Figure 7-22) The control panel contains two panel items: a text item to type in panel item names and a choice item providing the different item types that may be created. The destination panel is where newly created panel items are placed. It is boxed. After items are created, you may move them around the destination panel using the MENU mouse button. Moving the item off the destination panel deletes the item.

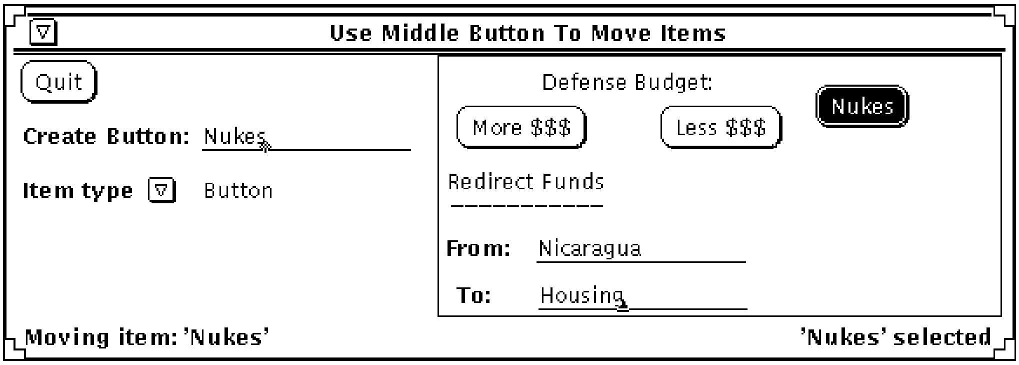


Figure 7-22. Output of item_move.c in use

The code for *item_move.c* uses the interpose function, `notify_interpose_event_func()`, to handle events within the destination panel. This interferes with the normal event processing for the destination panel; however, events continue to be dispatched to the panel items through the use of the `notify_next_event_func()`. Since the event function is only interested in MENU button events that occur on the destination panel's panel items, it might seem appropriate to use `PANEL_EVENT_PROC` since it is designed to notify the routine only when panel items receive events. However, this method would not work for this application because when the mouse button is dragged around the frame to move the item, the dragging events might move outside of the button and when that occurred the event callback would not be called.

7.20 Panel Package Summary

Table 7-4 lists the procedures and macros for the PANEL package. Table 7-5 lists the attributes for the PANEL package. This information is fully described in the *XView Reference Manual*.

Table 7-4. Panel Procedures and Macros

Procedures and Macros	
panel_advance_caret()	PANEL_CHECK_BOX
panel_backup_caret()	PANEL_CHOICE_STACK
panel_paint()	PANEL_EACH_ITEM()
panel_text_notify()	PANEL_END_EACH()
	PANEL_TOGGLE

Table 7-5. Panel Package Attributes

Panel Attributes	
PANEL_ACCEPT_KEYSTROKE	PANEL_LAYOUT
PANEL_BACKGROUND_PROC	PANEL_LINE_BREAK_ACTION
PANEL_BUSY	PANEL_LIST_CLIENT_DATA
PANEL_CARET_ITEM	PANEL_LIST_CLIENT_DATAS
PANEL_CHILD_CARET_ITEM	PANEL_LIST_DELETE
PANEL_CHOICE_COLOR	PANEL_LIST_DELETE_ROWS
PANEL_CHOICE_IMAGE	PANEL_LIST_DELETE_SELECTED_ROWS
PANEL_CHOICE_IMAGES	PANEL_LIST_DISPLAY_ROWS
PANEL_CHOICE_NCOLS	PANEL_LIST_FIRST_SELECTED
PANEL_CHOICE_NROWS	PANEL_LIST_FONT
PANEL_CHOICE_RECT	PANEL_LIST_FONTS
PANEL_CHOICE_STRING	PANEL_LIST_GLYPH
PANEL_CHOICE_STRINGS	PANEL_LIST_GLYPHS
PANEL_CHOOSE_NONE	PANEL_LIST_INSERT
PANEL_CHOOSE_ONE	PANEL_LIST_INSERT_DUPLICATE
PANEL_CLIENT_DATA	PANEL_LIST_INSERT_GLYPHS
PANEL_CURRENT_ITEM	PANEL_LIST_INSERT_STRINGS
PANEL_DEFAULT_ITEM	PANEL_LIST_MODE
PANEL_DEFAULT_VALUE	PANEL_LIST_NEXT_SELECTED
PANEL_DIRECTION	PANEL_LIST_NROWS
PANEL_DISPLAY_LEVEL	PANEL_LIST_ROW_HEIGHT
PANEL_DISPLAY_ROWS	PANEL_LIST_SCROLLBAR
PANEL_DROP_BUSY_GLYPH	PANEL_LIST_SELECT

Table 7-5. Panel Package Attributes (continued)

Panel Attributes	
PANEL_DROP_DND	PANEL_LIST_SELECTED
PANEL_DROP_FULL	PANEL_LIST_SORT
PANEL_DROP_GLYPH	PANEL_LIST_STRING
PANEL_DROP_HEIGHT	PANEL_LIST_STRINGS
PANEL_DROP_SEL_REQ	PANEL_LIST_TITLE
PANEL_DROP_SITE_DEFAULT	PANEL_LIST_WIDTH
PANEL_DROP_WIDTH	PANEL_MASK_CHAR
PANEL_EVENT_PROC	PANEL_MAX_TICK_STRING
PANEL_EXTRA_PAINT_HEIGHT	PANEL_MAX_VALUE
PANEL_EXTRA_PAINT_WIDTH	PANEL_MAX_VALUE_STRING
PANEL_FEEDBACK	PANEL_MIN_TICK_STRING
PANEL_FIRST_ITEM	PANEL_MIN_VALUE
PANEL_FIRST_PAINT_WINDOW	PANEL_MIN_VALUE_STRING
PANEL_FOCUS_PW	PANEL_NCHOICES
PANEL_GAUGE_WIDTH	PANEL_NEXT_COL
PANEL_GINFO	PANEL_NEXT_ITEM
PANEL_INACTIVE	PANEL_NEXT_ROW
PANEL_ITEM_CLASS	PANEL_NO_REDISPLAY_ITEM
PANEL_ITEM_COLOR	PANEL_NOTIFY_LEVEL
PANEL_ITEM_CREATED	PANEL_NOTIFY_PROC
PANEL_ITEM_DEAF	PANEL_NOTIFY_STATUS
PANEL_ITEM_LABEL_RECT	PANEL_NOTIFY_STRING
PANEL_ITEM_MENU	PANEL_OPS_VECTOR
PANEL_ITEM_NTH_WINDOW	PANEL_PAINT
PANEL_ITEM_NWINDOWS	PANEL_PRIMARY_FOCUS_ITEM
PANEL_ITEM_RECT	PANEL_READ_ONLY
PANEL_ITEM_VALUE_RECT	PANEL_REPAINT_PROC
PANEL_ITEM_WANTS_ADJUST	PANEL_SHOW_RANGE
PANEL_ITEM_WANTS_ISO	PANEL_SHOW_VALUE
PANEL_ITEM_X	PANEL_SLIDER_END_BOXES
PANEL_ITEM_X_GAP	PANEL_SLIDER_WIDTH
PANEL_ITEM_X_POSITION	PANEL_STATUS
PANEL_ITEM_Y	PANEL_TEXT_SELECT_LINE
PANEL_ITEM_Y_GAP	PANEL_TICKS
PANEL_ITEM_Y_POSITION	PANEL_TOGGLE_VALUE
PANEL_JUMP_DELTA	PANEL_VALUE
PANEL_LABEL_BOLD	PANEL_VALUE_DISPLAY_LENGTH
PANEL_LABEL_FONT	PANEL_VALUE_DISPLAY_WIDTH
PANEL_LABEL_IMAGE	PANEL_VALUE_FONT
PANEL_LABEL_STRING	PANEL_VALUE_STORED_LENGTH
PANEL_LABEL_WIDTH	PANEL_VALUE_UNDERLINED
PANEL_LABEL_X	PANEL_VALUE_X
PANEL_LABEL_Y	PANEL_VALUE_Y

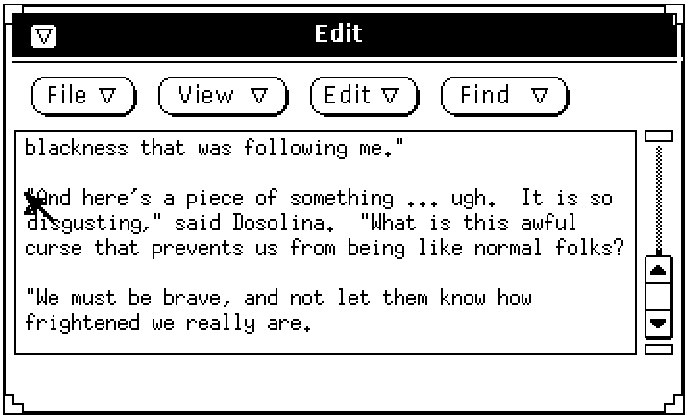
Table 7-6. New and Changed Panel Package Attributes (Version 3.2)

PANEL_LIST_INACTIVE	PANEL_LIST_MASK_GLYPHS
PANEL_LIST_DELETE_INACTIVE_ROWS	PANEL_LIST_ROW_VALUES
PANEL_LIST_DO_DBL_CLICK	PANEL_LIST_EXTENSION_DATA
PANEL_LIST_MASK_GLYPH	PANEL_LIST_EXTENSION_DATAS

8
Text Subwindows

This chapter describes the `TEXTSW` package, which allows a user or client to display and edit a sequence of ASCII characters. Figure 8-1 shows an example of a text subwindow. The text contains a vertical scrollbar but may not contain a horizontal scrollbar. The vertical scrollbar can be used to split views into several views (see Chapter 5, *Canvases and Openwin*). The font used by the text can be specified using the `TEXTSW_FONT` attribute, but only one font per text subwindow can be used, regardless of how many views there may be.

The contents of a text subwindow are stored in a file or in memory on the client side, not on the X server. Whether the *source* of the text is stored on disk or in memory is transparent to the user. When the user types characters in the text subwindow, the source might be changed immediately or synchronized later depending on how the text subwindow is configured. The `TEXTSW` package provides basic text editing features such as inserting arbitrary text into a file. It also provides complex operations such as searching for and replacing a string of text.



```
blackness that was following me."

"And here's a piece of something ... ugh.  It is so
disgusting," said Dosolina.  "What is this awful
curse that prevents us from being like normal folks?

"We must be brave, and not let them know how
frightened we really are.
```

Figure 8-1. A sample text subwindow

8.1 Creating Text Subwindows

Applications need to include the file <*xview/textsw.h*> to use text subwindows. You create a text subwindow the same way you create any XView object, by calling `xv_create()` with the appropriate type parameters:

```
Textsw textsw;
textsw = (Textsw)xv_create(base_frame, TEXTSW, NULL);
```

The font used by the text can be specified using `TEXTSW_FONT`; only one font per text subwindow can be used. Figure 8-2 shows the class hierarchy for the text subwindows.

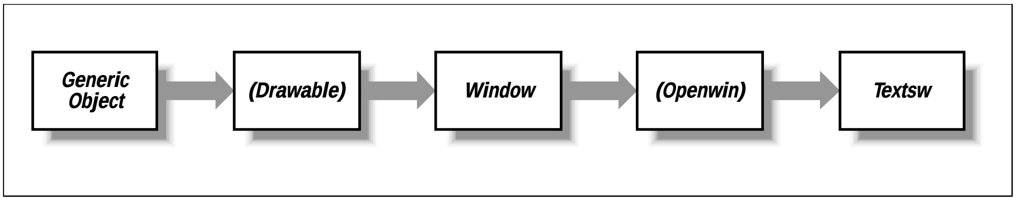


Figure 8-2. Textsw class hierarchy

8.2 Setting Text Subwindow Attributes

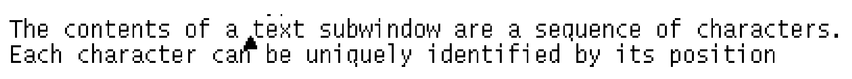
As for all XView objects, you can set attribute-value pairs to configure the text subwindow accordingly. Like the `CANVAS` package, the text subwindow object is subclassed from the `OPENWIN` package and can therefore be split into separate views. The package handles all of its own events and redisplaying of text, so none of these things is handled by the application.

Most text subwindow attributes are orthogonal; that is, attribute order does not effect the object. In a few cases, however, the attributes in a list might interact, so you must specify them in a particular order. Such cases are noted in the sections that follow. For example, you must pass `TEXTSW_STATUS` first in any call to `xv_create()`, if you want to find the status after setting some other attribute in the same call.

8.3 Text Subwindow Contents

The contents of a text subwindow are a sequence of characters. Each character can be uniquely identified by its position in the sequence (type `Textsw_index`). Editing operations, such as inserting and deleting text, can cause the index of successive characters to change. The valid indices are 0 through *length* –1 inclusive, where *length* is the number of characters currently in the text subwindow, returned by the `TEXTSW_LENGTH` attribute.

The text subwindow has a notion of the current index after which the next character will be inserted. This is called the *insertion point* and is indicated by a caret, as shown in Figure 8-3.



```
The contents of a text subwindow are a sequence of characters.
Each character can be uniquely identified by its position
```

Figure 8-3. A caret marks the insertion point

8.4 Editing a Text Subwindow

A text subwindow can be edited by the user or by a client program. When you create a text subwindow, the user is normally allowed to edit it. By using the special attributes discussed in this section, the client program can edit the subwindow. These edits are then stored in */tmp/text**Process-id.Counter***.

The next five sections explain the functions and attributes that you will use to load, read, write, edit, and finally save a text file.

8.4.1 Loading a File

You can load a file into a text subwindow by using `TEXTSW_FILE`, as in:

```
xv_set(textsw, TEXTSW_FILE, file_name, NULL);
```

Keep in mind that if the existing text has been edited, then these edits will be lost. To avoid such loss, first check whether there are any outstanding edits by calling:

```
int modified = (int)xv_get(textsw, TEXTSW_MODIFIED)
```

If there have been updates, you may choose to synchronize with the *source* if necessary. That is, if the existing text is part of a file, you can overwrite the existing changes before loading in a new file.

The above call to `xv_set()`, which loads the new file, positions the new text so that the first character displayed has the same index as the first character that was displayed in the previous file. This is probably not what you want. The code segment below shows how to load the file at a set position:

```
xv_set(textsw,
    TEXTSW_FILE,  file_name,
    TEXTSW_FIRST, position,
    NULL);
```

The first character displayed has its index set by `position`. The order of these attributes matters. Because attributes are evaluated in the order given, reversing the order would first reposition the existing file, then load the new file. This would cause an unnecessary repaint. It would also mis-position the old file if it was shorter than `position`.

8.4.2 Checking the Status of the Text Subwindow

Both of the calls in the previous example blindly trust that the load of the new file was successful. This is, in general, a bad idea. The following code segment shows how to find out whether the load succeeded, and if not, why it failed:

```
Textsw textsw;
Textsw_status status;

textsw = (Textsw)xv_create(base_frame,
    TEXTSW,
    TEXTSW_STATUS,    &status,
    TEXTSW_FILE,      file_name,
    TEXTSW_FIRST,     position,
    NULL);
```

NOTE

The TEXTSW_STATUS attribute and handle must appear in the attribute list *before* the operation whose status you want to determine.

The TEXTSW_STATUS attribute is only valid for xv_create().

The range of values for such a variable are enumerated in Table 8-1. Note that in the first column, each value begins with the prefix TEXTSW_STATUS_, which has been omitted from the table to improve readability.

Table 8-1. Range of Values for Status Variables

Value (TEXTSW_STATUS_ ...)	Description
OKAY	The operation encountered no problems.
BAD_ATTR	The attribute list contained an illegal or unrecognized attribute.
BAD_ATTR_VALUE	The attribute list contained an illegal value for an attribute, usually an out-of-range value for an enumeration.
CANNOT_ALLOCATE	A call to calloc(2) or malloc(2) failed.
CANNOT_OPEN_INPUT	The specified input file does not exist or cannot be accessed.
CANNOT_INSERT_FROM_FILE	The operation encountered a problem when trying to insert from file.
OUT_OF_MEMORY	The operation ran out of memory while editing in memory.
OTHER_ERROR	The operation encountered a problem not covered by any of the other error indications.

8.4.3 Writing to a Text Subwindow

To insert text into a text subwindow at the current insertion point, call:

```
Textsw_index
textsw_insert(textsw, buf, buf_len)
    Textsw  textsw;
    char   *buf;
    int     buf_len;
```

The return value is the number of characters actually inserted into the text subwindow. This number will equal `buf_len` unless either the text subwindow has had a memory allocation failure or the portion of text containing the insertion point is read only. The insertion point is moved forward by the number of characters inserted.

This routine does not do terminal-style interpretation of the input characters. Thus, editing characters (such as CTRL-H or DEL for character erase) are simply inserted into the text subwindow rather than performing edits to the existing contents of the text subwindow. To emulate a terminal, scan the characters to be inserted and invoke `textsw_edit()` where appropriate, as described in the next section.

8.4.3.1 Setting the insertion point

The attribute `TEXTSW_INSERTION_POINT` is used to interrogate and set the insertion point. For instance, the following call determines where the insertion point is:

```
Textsw_index point;

point = (Textsw_index)xv_get(textsw, TEXTSW_INSERTION_POINT);
```

Whereas the following call sets the insertion point to be just before the third character of the text:

```
xv_set(textsw, TEXTSW_INSERTION_POINT, 2, NULL);
```

To set the insertion point at the end of the text, set `TEXTSW_INSERTION_POINT` to the special index `TEXTSW_INFINITY`. This call does not ensure that the new insertion point will be visible in the text subwindow, even if `TEXTSW_INSERT_MAKES_VISIBLE` is `TRUE`. To guarantee that the caret will be visible afterwards, call `textsw_possibly_normalize()`, a procedure that is described later in this chapter.

8.4.4 Reading from a Text Subwindow

Many applications that incorporate text subwindows never need to read the contents of the text directly from the text subwindow. For instance, the text subwindow might display text for the user to view but not to edit.

Even when the user is allowed to edit text, some applications simply wait for the user to perform some action that indicates that all of the edits have been made. The application can then use either `textsw_save()` or `textsw_store_file()` to place the text in the file.

The text can then be read via the usual file input utilities, or the file itself can be passed off to another routine or program.

It is, however, useful to be able to directly examine the text in the text subwindow. You can do this using the TEXTSW_CONTENTS attribute. The code fragment below illustrates how to use TEXTSW_CONTENTS to get a span of characters from the text subwindow. It gets 1000 characters beginning at position 500 out of the text subwindow and places them into a NULL-terminated string.

```
#define TO_READ 1000

char         buf[TO_READ+1];
Textsw_index next_pos;

next_pos = (Textsw_index) xv_get(textsw, TEXTSW_CONTENTS, 500,
                                 buf, TO_READ);

if (next_pos != 500+TO_READ) {
    /* handle error case */
} else
    buf[TO_READ] = '\0';
```

8.4.5 Deleting Text

You can delete a contiguous span of characters from a text subwindow by calling:

```
Textsw_index
textsw_delete(textsw, first, last_plus_one)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
```

first specifies the first character of the span that will be deleted; last_plus_one specifies the first character *after* the span that will *not* be deleted. first should be less than or equal to last_plus_one. To delete to the end of the text, pass the special value TEXTSW_INFINITY for last_plus_one.

The return value is the number of characters deleted or:

```
last_plus_one - first
```

unless the specified span is read-only. If the insertion point is in the span being deleted, it will be left at first.

A side effect of calling textsw_delete() is that the deleted characters become the contents of the global Clipboard. To remove the characters from the text subwindow without affecting the Clipboard, call:

```
Textsw_index
textsw_erase(textsw, first, last_plus_one)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
```

Again, the return value is the number of characters removed, and last_plus_one can be TEXTSW_INFINITY.

Both of these procedures will return 0 if the operation fails.

8.4.6 Emulating an Editing Character

You can emulate the behavior of an editing character, such as CTRL-H, with `textsw_edit()`:

```
Textsw_index
textsw_edit(textsw, unit, count, direction)
    Textsw    textsw;
    unsigned  unit, count, direction;
```

Depending on the value of `unit`, this routine will erase either a character, a word, or a line. Set `unit` to:

- `TEXTSW_UNIT_IS_CHAR` to erase individual characters.
- `TEXTSW_UNIT_IS_WORD` to erase the span of characters that make up a word (including any intervening white space or other nonword characters).
- `TEXTSW_UNIT_IS_LINE` to erase all characters in the line on one side of the insertion point.

If the `direction` parameter is 0, the operation will affect characters after the insertion point; otherwise, it will affect characters before the insertion point.

The `count` parameter determines the number of times the operation will be applied. Set it to 1 to do the edit once or to a value greater than 1 to do multiple edits in a single call. `textsw_edit()` returns the number of characters actually removed.

For example, suppose you want to interpret the function key `F7` as meaning *delete word forward*. On receiving the event code for the `F7` key, you would make the call:

```
textsw_edit(textsw, TEXTSW_UNIT_IS_WORD, 1, NULL);
```

8.4.7 Replacing Characters

While a span of characters can be replaced by calling `textsw_erase()` followed by `textsw_insert()`, character replacement is done most efficiently by calling:

```
Textsw_index
textsw_replace_bytes(textsw, first, last_plus_one, buf, buf_len)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
    char         *buf;
    int           buf_len;
```

The span of characters to be replaced is specified by `first` and `last_plus_one`, just as in the call to `textsw_erase()`. The new characters are specified by `buf` and `buf_len`, just as in the call to `textsw_insert()`. Once again, if `last_plus_one` is `TEXTSW_INFINITY`, the replace operation affects all characters from `first` to the end of the text. If the insertion point is in the span being replaced, it will be left at:

```
first + buf_len
```

The return value is the net number of bytes inserted. The number is negative if the original string is longer than the one that replaces it. If a problem occurs when an attempt is made to replace a span, it will return an error code of 0.

`textsw_replace_bytes()`, like `textsw_erase()`, does *not* put the characters it removes on the global Clipboard.

8.4.8 The Editing Log

All text subwindows allow the user to undo editing actions. The `TEXTSW` package keeps a running log of all the edits. If a file is associated with the text subwindow, this log is kept in a file in the */tmp* directory. This file can grow until the file system in which this directory resides runs out of space. To limit the size of the edit log and to avoid filling up all of */tmp*, the user can set the text wrap-around size with `TEXTSW_WRAPAROUND_SIZE`. If there is no associated file, the edit log is kept in memory, and the maximum size of the log is controlled by the attribute `TEXTSW_MEMORY_MAXIMUM`, which defaults to 20,000 bytes.

Unfortunately, once a memory-resident edit log has reached its maximum size, no more characters can be inserted into or removed from the text subwindow. In particular, since deletions as well as insertions are logged, space cannot be recovered by deleting characters.

It is important to understand how the edit log works, since you might want to use a text subwindow with no associated file to implement a temporary scratch area or error message log. If such a text subwindow is used for a long time, the default limit of 20,000 bytes might well be reached, and either the user or your code will be unable to insert any more characters, even though only a few characters might be visible in the text subwindow. Therefore, it is recommended to set `TEXTSW_MEMORY_MAXIMUM` to a much higher value, say 200,000.

8.4.9 Which File is Being Edited?

To find out the name of the file in the text subwindow, call:

```
int
textsw_append_file_name(textsw, name)
    Textsw  textsw;
    char    *name;
```

If the text subwindow is editing memory, then this routine will return a nonzero value. Otherwise, it will return 0 and append the name of the file to the end of `name`. The following code gets the name of the current file:

```
char name[BUFSIZ];

name[0] = '\0';
if (textsw_append_file_name(textsw, name) == 0)
    printf("File name is: %s\n", name);
```

8.4.9.1 Interactions with the file system

Suppose the current file is called *myfile*. If the user chooses `textsw_save()`, the following sequence of file operations occurs:

- *myfile* is copied to *myfile%*.
- The contents of *myfile%* are combined with information from the edit log file (*/tmp/Text**Process-id.Counter***) and written over *myfile*, thereby preserving all its permissions, etc.
- The edit log file is removed from */tmp*.

If *myfile* is a symbolic link to *../some_dir/otherfile*, then the backup file is created as *../some_dir/otherfile%*.

Keep in mind that the user can change the current directory by selecting “Load File” or “Set Directory” from the text subwindow menu. If *myfile* is a relative path name, then both the copy to *myfile%* and the save take place in the current directory.

8.5 Saving Edits in a Subwindow

To save any edits made to a file currently loaded into a text subwindow call:

```
unsigned
textsw_save(textsw, locx, locy)
    Textsw  textsw;
    int     locx, locy;
```

`locx` and `locy` are relative to the upper-left corner of the text subwindow and are used to position the upper-left corner of the alert should the save fail for some reason—usually they should be 0. The return value is 0 if and only if the save succeeded.

8.5.1 Storing Edits

The text subwindow might not contain a file, or the client might wish to place the edited version of the text (whether or not the original text came from a file) in some specific file. To store the contents of a text subwindow to a file, call:

```
unsigned
textsw_store_file(textsw, filename, locx, locy)
    Textsw  textsw;
    char   *filename;
    int     locx, locy;
```

Again, `locx` and `locy` are used to position the upper-left corner of the message box. The return value is 0 if and only if the store succeeded.

By default, this call changes the file that the text subwindow is editing, so that subsequent saves will save the edits to the new file. To override this policy, set the attribute `TEXTSW_STORE_CHANGES_FILE` to `FALSE`.

8.5.2 Discarding Edits

To discard the edits performed on the contents of a text subwindow, call:

```
void
textsw_reset(textsw, locx, locy)
    Textsw  textsw;
    int     locx, locy;
```

`locx` and `locy` are as above. Note that if the text subwindow contains a file that has not been edited, the effect of `textsw_reset` is to unload the file and replace it by memory provided by the `TEXTSW` package; thus, the user will see an absolutely empty text subwindow. Alternatively, if the text subwindow was already editing memory, then another, untouched, piece of primary memory will be provided and the edited piece will be deallocated.

8.6 Setting the Contents of a Text Subwindow

The rest of this chapter describes the other functions that are available for text subwindows. These features include setting the contents of a subwindow, setting the primary selection, and dealing with multiple or split views.

You might want to set the initial contents of a text subwindow that your application uses. To set the initial contents of a text subwindow, use one of three attributes: `TEXTSW_INSERT_FROM_FILE`, `TEXTSW_FILE_CONTENTS`, and `TEXTSW_CONTENTS`. Each attribute is illustrated in code fragments given below.

8.6.1 TEXTSW_FILE_CONTENTS

The attribute `TEXTSW_FILE_CONTENTS` allows a client to initialize the text subwindow contents from a file yet still edit the contents in memory. The user can return a text subwindow to its initial state after an editing session by choosing "Undo All Edits" in the text menu.

The code fragment below shows how you would use this attribute:

```
extern char *filename;

xv_set(textsw,
   TEXTSW_FILE_CONTENTS, filename,
   TEXTSW_FIRST,         0,
   NULL);
```

When the client calls the undo routine and `filename` is not a null string, the memory used by the text subwindow is reinitialized with the contents of the file specified by `filename`.

When the client calls the undo routine and the `filename` is a null string, the memory used by the text subwindow is reinitialized with the previous contents of the text subwindow.

8.6.2 TEXTSW_CONTENTS

TEXTSW_CONTENTS lets you insert a text string from memory, instead of a file, into the text subwindow. The default for this attribute is NULL.

Using xv_create() with this attribute specifies the initial contents for a nonfile text subwindow.

Using xv_set() with this attribute sets the contents of a window, as in:

```
xv_set(textsw, TEXTSW_CONTENTS, "text", NULL);
```

If you use xv_get() with this attribute, you will need to provide additional parameters, as in:

```
xv_get(textsw, TEXTSW_CONTENTS, pos, buf, buf_len);
```

The return value is the next position to be read. The buffer array:

```
buf[0 ... buf_len-1]
```

is filled with the characters from textsw beginning at the index pos and is NULL-terminated only if there were too few characters to fill the buffer.

8.6.3 TEXTSW_INSERT_FROM_FILE

TEXTSW_INSERT_FROM_FILE allows a client to insert the contents of a file into a text subwindow at the current insertion point. It is the programming equivalent of a user choosing “Include File” from the text menu.

The code below demonstrates this attribute:

```
Textsw          textsw;
Textsw_status   status;

xv_set(textsw,
    TEXTSW_STATUS,           &status,
    TEXTSW_INSERT_FROM_FILE, filename,
    NULL);
```

Three status values can be returned for this attribute when the argument TEXTSW_STATUS is passed in the same call to xv_create() or xv_set():

```
TEXTSW_STATUS_OKAY
TEXTSW_STATUS_CANNOT_INSERT_FROM_FILE
TEXTSW_STATUS_OUT_OF_MEMORY
```

8.7 Positioning the Text Displayed in a Text Subwindow

Usually, more text is managed by the text subwindow than can be displayed all at once. As a result, it is often necessary to determine the indices of the characters that are being displayed and to control exactly which portion of the text is visible.

8.7.1 Screen Lines and File Lines

When there are long lines in the text, it is necessary to distinguish between two definitions of “line of text.”

A *screen line* reflects what is actually displayed on the screen. A line begins with the left-most character in the subwindow and continues across until either a newline character or the right edge of the subwindow is encountered. A *file line*, on the other hand, can only be terminated by the newline character. It is defined as the span of characters starting after a newline character (or the beginning of the file) running through the next newline character (or the end of the file).

Whenever the right edge of the subwindow is encountered before the newline, if the following attribute-value pair were specified:

```
TEXTSW_LINE_BREAK_ACTION, TEXTSW_WRAP_AT_CHAR
```

then the next character and its successors would be displayed on the next lower screen line. In this case, there would be two screen lines, but only one file line. From the perspective of the display there are two lines; from the perspective of the file, only one. On the other hand, if the following attribute-value pair were specified:

```
TEXTSW_LINE_BREAK_ACTION, TEXTSW_WRAP_AT_WORD
```

then the entire word would be displayed on the next line.

Unless otherwise specified, all text subwindow attributes and procedures use the *file line* definition. Line indices have a zero-origin, like the character indices; that is, the first line has index 0, not 1.

8.7.2 Absolute Positioning

Two attributes are provided to allow you to specify which portion of the text is displayed in the text subwindow.

Setting the attribute TEXTSW_FIRST to a given index causes the first character of the line containing the index to become the first character displayed in the text subwindow. Thus, the following call causes the text to be positioned so that the first displayed character is the first character of the line that contains index 1000:

```
xv_set(textsw, TEXTSW_FIRST, 1000, NULL);
```

Since the text subwindow is subclassed from the OPENWIN package and can be split into several views, the previous code fragment would only cause the positioning of one view. To position all of the views in a text subwindow, use the attribute TEXTSW_FOR_ALL_VIEWS, as in the following call:

```
xv_set(textsw,
    TEXTSW_FOR_ALL_VIEWS, TRUE,
    TEXTSW_FIRST,         1000,
    NULL);
```

Conversely, the following call retrieves the index of the first displayed character:

```
index = (Textsw_index)xv_get(textsw, TEXTSW_FIRST);
```

A related attribute, useful in similar situations, is TEXTSW_FIRST_LINE. When used in a call on xv_set() or xv_get(), the value is a file line index within the text.

You can determine the character index that corresponds to a given line index (both zero-origin) within the text by calling:

```
Textsw_index
textsw_index_for_file_line(textsw, line)
    Textsw  textsw;
    int     line;
```

The return value is the character index for the first character in the line, so character index 0 always corresponds to line index 0.

8.7.3 Relative Positioning

To move the text in a text subwindow up or down by a small number of lines, call the routine:

```
void
textsw_scroll_lines(textsw, count)
Textsw  textsw;
int     count;
```

A positive value for count causes the text to scroll up, while a negative value causes the text to scroll down.

When calling textsw_scroll_lines(), you might want to know how many screen lines are in the text subwindow. You can find this out by calling:

```
int
textsw_screen_line_count(textsw)
    Textsw  textsw;
```

8.7.4 Which File Lines are Visible?

Exactly which file lines are visible on the screen is determined by calling:

```
void
textsw_file_lines_visible(textsw, top, bottom)
    Textsw  textsw;
    int    *top, *bottom;
```

This routine fills in the addressed integers with the file line indices of the first and last file lines being displayed in the specified text subwindow.

8.7.4.1 Guaranteeing what is visible

To ensure that a particular line or character is visible, call:

```
void
textsw_possibly_normalize(textsw, position)
    Textsw       textsw;
    Textsw_index position;
```

The text subwindow must be displayed on the screen before this function will work.

If the character at the specified `position` is already visible, then this routine does nothing. If it is not visible, then it repositions the text so that it is visible and at the top of the subwindow.

If a particular character should always be at the top of the subwindow, then calling the following routine is more appropriate:

```
void
textsw_normalize_view(textsw, position)
    Textsw       textsw;
    Textsw_index position;
```

8.7.4.2 Ensuring that the insertion point is visible

Most of the programmatic editing actions do not update the text subwindow to display the caret, even if `TEXTSW_INSERT_MAKES_VISIBLE` is set. If you want to ensure that the insertion point is visible, use:

```
textsw_possibly_normalize(textsw,
    (Textsw_index) xv_get(textsw, TEXTSW_INSERTION_POINT));
```

8.8 Finding and Matching a Pattern

A common operation performed on text is to find a span of characters that match some specification. The text subwindow provides several rudimentary pattern matching facilities. This section describes two functions that you can call in order to perform similar operations.

8.8.1 Matching a Span of Characters

To find the nearest span of characters that match a pattern, call:

```
int
textsw_find_bytes(textsw, first, last_plus_one, buf,
                  buf_len, flags)
```

```
    Textsw       textsw;
    Textsw_index *first, *last_plus_one;
    char         *buf;
    unsigned      buf_len;
    unsigned      flags;
```

The pattern to match is specified by `buf` and `buf_len`. The matching operation looks for an exact and literal match—it is sensitive to case and does not recognize any kind of metacharacter in the pattern. `first` specifies the position at which to start the search. If `flags` is 0, the search proceeds forward through the text; if `flags` is 1, the search proceeds backwards. The return value is –1 if the pattern cannot be found; otherwise it is some non-negative value, in which case the indices addressed by `first` and `last_plus_one` will have been updated to indicate the span of characters that match the pattern.

8.8.2 Matching a Specific Pattern

Another useful operation is to find delimited text. For example, you might want to find the starting and ending brace in a piece of code. To find a matching pattern, call:

```
    int
    textsw_match_bytes(textsw, first, last_plus_one,
                       start_sym, start_sym_len,
                       end_sym, end_sym_len, field_flag)
    Textsw       textsw;
    Textsw_index *first, *last_plus_one;
    char         *start_sym, *end_sym;
    int           start_sym_len, end_sym_len;
    unsigned      field_flag;
```

`first` stores the starting position of the pattern that you want to search for. `last_plus_one` stores the cursor position of the end pattern. Its value is one position past the text. `start_sym` and `end_sym` store the beginning position and ending position of the pattern, respectively. `start_sym_len` and `end_sym_len` store the starting and ending pattern's length, respectively.

Use one of the following three field flag values to search for matches:

TEXTSW_DELIMITER_FORWARD
: Begins from `first` and searches forward until it finds `start_sym` and matches it forward with `end_sym`.

TEXTSW_DELIMITER_BACKWARD
: Begins from `first` and searches backward for `end_sym` and matches it backward with `start_sym`.

TEXTSW_DELIMITER_ENCLOSE
: Begins from `first` and expands both directions to match `start_sym` and `end_sym` of the next level.

If no match is found, then `textsw_match_bytes()` will return a value of –1. If a match is found, then it will return the index of the first match.

The code fragment below can be used to find delimited text. Notice that the `field_flag` value is `TEXTSW_DELIMITER_FORWARD`.

```
Textsw_index     first, last_plus_one, pos;

first = (Textsw_index) xv_get(textsw, TEXTSW_INSERTION_POINT);
pos = textsw_match_bytes(textsw, &first, &last_plus_one,
                  "/*", 2,
                  "*/", 2, TEXTSW_DELIMITER_FORWARD);
if (pos > 0) {
     textsw_set_selection(textsw, first, last_plus_one, 1);
     xv_set(textsw, TEXTSW_INSERTION_POINT, last_plus_one, NULL);
} else
    (void) window_bell(textsw);
```

This code searches forward from `first` until it finds the starting /* and matches it forward with the next */. If no match is found, a bell will ring in the text subwindow.

8.9 Marking Positions

Often a client wants to keep track of a particular character or group of characters that are in the text subwindow. Given that arbitrary editing can occur in a text subwindow and that it is very tedious to intercept and track all of the editing operations applied to a text subwindow, it is often easier to simply place one or more marks at various positions in the text subwindow. These marks are automatically updated by the text subwindow to account for user and client edits. There is no limit to the number of marks you can add.

A new mark is created by calling:

```
Textsw_mark
textsw_add_mark(textsw, position, flags)
    Textsw        textsw;
    Textsw_index  position;
    unsigned      flags;
```

The `flags` argument is either `TEXTSW_MARK_DEFAULTS` or `TEXTSW_MARK_MOVE_AT_INSERT`. The latter causes an insertion at the marked position to move the mark to the end of the inserted text, whereas the former causes the mark to not move when text is inserted at the mark's current position. As an example, suppose that the text managed by the text subwindow consists of the two lines:

```
this is the first line
not this, which is the second
```

Assume a mark is set at position 5 (just before the *i* in *is* on the first line) with `flags` of `TEXTSW_MARK_MOVE_AT_INSERT`.

If the user makes a selection just before the *is* (thereby placing the insertion point before the *i*, at position 5) and types an *h*, making the text read:

```
this his the first line
not this, which is the second
```

the mark moves with the insertion point and they both end up at position 6.

However, if the `flags` had been `TEXTSW_MARK_DEFAULTS`, then the mark would remain at position 5 after the user typed the *h*, although the insertion point moved on to position 6.

Now, suppose instead that the user made a selection before the *this* on the first line, and typed *Kep*, making the text read:

```
Kepthis is the first line
not this, which is the second
```

In this case, no matter what `flags` the mark had been created with, it would end up at position 8, still just before the *i* in *is*.

If a mark is in the middle of a span of characters that is subsequently deleted, the mark moves to the beginning of the span. Going back to the original scenario, with the original text and the mark set at position 5, assume that the user deletes from the *h* in *this* through the *e* in *the* on the first line, resulting in the text:

```
te first line
not this, which is the second
```

When the user is done, the mark will be at position 1, just before the *e* in *te*.

The current position of a mark is determined by calling:

```
Textsw_index
textsw_find_mark(textsw, mark)
    Textsw       textsw;
    Textsw_mark  mark;
```

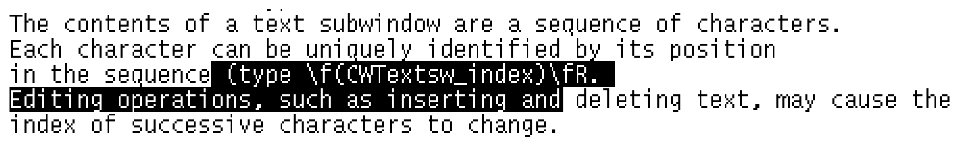
An existing mark is removed by calling:

```
void
textsw_remove_mark(textsw, mark)
    Textsw       textsw;
    Textsw_mark  mark;
```

Note that marks are dynamically allocated, and it is the client's responsibility to keep track of them and to remove them when they are no longer needed.

8.9.1 Getting a Text Selection

A user selects a portion of the contents of the text subwindow using a pointer. A text selection is indicated on the screen with reverse-video highlighting. An application needs to know which window has the current selection and what the contents of a text selection are. The `TEXTSW` package does not provide procedures to get this information. Instead, these functions are carried out by the Selection Service. For an example of how this is done, see Chapter 18, *Selections*. Figure 8-4 shows a text selection.



```
The contents of a text subwindow are a sequence of characters.
Each character can be uniquely identified by its position
in the sequence (type \f(CWTextsw_index)\fR.
Editing operations, such as inserting and deleting text, may cause the
index of successive characters to change.
```

Figure 8-4. A text selection

8.9.2 Setting the Text Selection

Primary and secondary selections are maintained. The primary or secondary selection can be set by calling the following:

```
void
textsw_set_selection(textsw, first, last_plus_one, type)
    Textsw       textsw;
    Textsw_index  first, last_plus_one;
    unsigned     type;
```

A value of 1 for type means *primary selection*, while a value of 2 means *secondary selection* and a value of 17 is *pending delete*. Note that there is no requirement that all or part of the selection be visible; use `textsw_possibly_normalize()` to guarantee visibility (see Section 8.7.4, "Which File Lines are Visible?").

8.10 Dealing with Multiple Views

By splitting a text view, the user can create multiple views of the text being managed by the text subwindow. Although these additional views are usually transparent to the client code controlling the text subwindow, it might occasionally be necessary for a client to deal directly with all of the views. This is accomplished by using the following routines, with the knowledge that split views are simply extra text subwindows that happen to share the text of the original text subwindow.

```
Textsw
textsw_first(textsw)
    Textsw textsw;
```

Given an arbitrary view out of a set of multiple views, `textsw_first()` returns the first view (currently, this is the original text subwindow that the client created). To move through the other views of the set, call:

```
Textsw
textsw_next(textsw)
    Textsw textsw;
```

Given any view of the set, `textsw_next()` returns some other member of the set or NULL if there are none left to enumerate. The loop coded below is guaranteed to process all of the views in the set:

```
for (textsw = textsw_first(any_split); textsw;
               textsw = textsw_next(textsw)) {
    /* processing involving textsw */
}
```

When you create a text subwindow, take into account that the user might split the window. If you try to do something like enlarge the window, you might run into problems.

8.11 Text Subwindow Destroy Confirmation

A confirmation notice is displayed when a text subwindow is about to be destroyed. A text subwindow is destroyed when the text subwindow or its enclosing frame is the object of a `xv_destroy()` call (this may occur when application is quit from the window manager's menu). Supplying the text subwindow confirmation notice is referred to as *vetoing the destroy*. A confirmation notice is provided when the text subwindow's *ignore limit* has been reached. The ignore limit specifies the number of edits permitted before the confirmation notice is displayed and is set with `TEXTSW_IGNORE_LIMIT`. Valid values for `TEXTSW_IGNORE_LIMIT` are 0, meaning destroy will be vetoed if any edits have been done, and `TEXTSW_INFINITY`, meaning the destroy will never be vetoed.

8.12 Notifications from a Text Subwindow

The text subwindow notifies its client about interesting changes in the subwindow's or text's state by calling a notification procedure. It also calls this procedure in response to user actions. If the client does not provide an explicit notification procedure by using the attribute `TEXTSW_NOTIFY_PROC`, then the text subwindow provides a default procedure. The declaration for this procedure looks like:

```
void
notify_proc(textsw, avlist)
    Textsw       textsw;
    Attr_avlist  avlist;
```

`avlist` contains attributes that are the members of the `Textsw_action` enumeration.

Your notification procedure must be careful either to process all of the possible attributes or to pass through the attributes that it does not process to the standard notification procedure. This is important because among the attributes that can be in the *avlist* are those that cause the standard notification procedure to implement the possible *Front*, *Back*, *Open*, *Close*, and *Quit* accelerators of the user interface.

Example 8-1 presents a client notify procedure for a text subwindow.

Example 8-1. Client notify procedure for a text subwindow

```
void (*textsw_default_notify)();

void
client_notify_proc(textsw, attributes)
```

Example 8-1. Client notify procedure for a text subwindow (continued)

```
Textsw          textsw;
Attr_avlist     attributes;
{
    int  pass_on   = FALSE;
    Attr_avlist    attrs;

    for (attrs = attributes; *attrs; attrs = attr_next(attrs)) {
        switch ((Textsw_action)(*attrs)) {
            case TEXTSW_ACTION_CAPS_LOCK:
                /* Swallow this attribute */
                ATTR_CONSUME(*attrs);
                break;
            case TEXTSW_ACTION_CHANGED_DIRECTORY:
                /* Monitor the attribute, don't swallow it */
                strcpy(current_directory, (char *)attrs[1]);
                pass_on = TRUE;
                break;
            default:
                pass_on = TRUE;
                break;
        }
    }
    if (pass_on)
        textsw_default_notify(textsw, attributes);
}

textsw_default_notify =
    (void (*)())xv_get(textsw, TEXTSW_NOTIFY_PROC);
xv_set(textsw, TEXTSW_NOTIFY_PROC, client_notify_proc, NULL);
```

The `Textsw_action` attributes that can be passed to your notify procedure are listed in Table 8-2. Note that in the first column, each attribute begins with the prefix `TEXTSW_ACTION_`, which has been omitted from the table to improve readability. Remember that the attributes constitute a special class that are passed to your text subwindow notification procedure. They are not attributes of the text subwindow in the usual sense and cannot be retrieved or modified using `xv_get()` or `xv_set()`.

Table 8-2. Textsw_action Attributes

Attribute (`TEXTSW_ACTION_ ...`)	Type	Description
`CAPS_LOCK`	`Boolean`	The user pressed the Caps Lock key to change the setting of the Caps Lock (it is initially 0, meaning off).
`CHANGED_DIRECTORY`	`char *`	The current working directory for the process has been changed to the directory named by the provided string value.
`EDITED_FILE`	`char *`	The file named by the provided string value has been edited. Appears once per session of edits (see below).

Table 8-2. Textsw_action Attributes (continued)

Attribute (TEXTSW_ACTION_ ...)	Type	Description
EDITED_MEMORY	(no value)	Monitors whether an empty text subwindow has been edited.
FILE_IS_READONLY	char *	The file named by the provided string value does not have write permission.
LOADED_FILE	char *	The text subwindow is being used to view the file named by the provided string value.
TOOL_CLOSE	(no value)	The frame containing the text subwindow should become iconic.
TOOL_DESTROY	Event *	The tool containing the text subwindow should exit, without checking for a veto from other subwindows. The value is the user action that caused the destroy.
TOOL_QUIT	Event *	The tool containing the text subwindow should exit normally. The value is the user action that caused the exit.
TOOL_MGR	Event *	The tool containing the text subwindow should do the window manager operation associated with the provided event value.
USING_MEMORY	(no value)	The text subwindow is being used to edit a string stored in primary memory, not a file.

The attribute TEXTSW_ACTION_EDITED_FILE is a slight misnomer, as it is given to the notify procedure *after* the first edit to *any* text, whether or not it came from a file. This notification happens only once per session of edits, whereas, on the other hand, notification of TEXTSW_ACTION_LOADED_FILE is considered to terminate the old session and start a new one.

NOTE

The attribute TEXTSW_ACTION_LOADED_FILE must be treated very carefully because the notify procedure gets called with this attribute in several situations: after a file is initially loaded, after any successful "Save Current File" menu operation, after an "Undo All Edits" menu operation, and during successful calls to textsw_reset(), textsw_save(), and textsw_store().

The appropriate response by the procedure is to interpret these notifications as being equivalent to:

> *The text subwindow is displaying the file named by the provided string value; no edits have been performed on the file yet. In addition, any previously displayed or edited file has been either reset, saved, or stored under another name.*

8.12.1 Text Subwindow Interposition

If you need to interpose on a text subwindow, get the text subwindow's *view window* and interpose on it.

```
Xv_Window window;
Textsw    textsw;
int       win_no;

window = (Xv_Window)xv_get(textsw, OPENWIN_NTH_VIEW, win_no);
```

For more information on registering events for text subwindows, see Section 6.3, "Registering Events;" for more information on interposition, refer to Section 20.9, "Interposition."

8.13 Text Subwindow Package Summary

Table 8-3 lists the procedures and macros for the `TEXTSW`. Table 8-4 lists the attributes for the `TEXTSW` package. This information is described fully in the *XView Reference Manual*.

Table 8-3. Text Subwindow Procedures and Macros

Procedures and Macros	
`textsw_add_mark()`	`textsw_next()`
`textsw_append_file_name()`	`textsw_normalize_view()`
`textsw_delete()`	`textsw_notify_proc()`
`textsw_edit()`	`textsw_possibly_normalize()`
`textsw_erase()`	`textsw_remove_mark()`
`textsw_file_lines_visible()`	`textsw_replace_bytes()`
`textsw_find_bytes()`	`textsw_reset()`
`textsw_find_mark()`	`textsw_save()`
`textsw_first()`	`textsw_screen_line_count()`
`textsw_index_for_file_line()`	`textsw_scroll_lines()`
`textsw_insert()`	`textsw_set_selection()`
`textsw_match_bytes()`	`textsw_store_file()`

Table 8-4. *Text Subwindow Attributes*

TEXTSW_AGAIN_RECORDING	TEXTSW_IGNORE_LIMIT
TEXTSW_AUTO_INDENT	TEXTSW_INSERT_FROM_FILE
TEXTSW_AUTO_SCROLL_BY	TEXTSW_INSERT_MAKES_VISIBLE
TEXTSW_BLINK_CARET	TEXTSW_INSERTION_POINT
TEXTSW_BROWSING	TEXTSW_LENGTH
TEXTSW_CHECKPOINT_FREQUENCY	TEXTSW_LINE_BREAK_ACTION
TEXTSW_CLIENT_DATA	TEXTSW_LOWER_CONTEXT
TEXTSW_CONFIRM_OVERWRITE	TEXTSW_MEMORY_MAXIMUM
TEXTSW_CONTENTS	TEXTSW_MULTI_CLICK_SPACE
TEXTSW_CONTROL_CHARS_USE_FONT	TEXTSW_MULTI_CLICK_TIMEOUT
TEXTSW_DESTROY_VIEW	TEXTSW_NOTIFY_PROC
TEXTSW_DISABLE_CD	TEXTSW_READ_ONLY
TEXTSW_DISABLE_LOAD	TEXTSW_STATUS
TEXTSW_EDIT_COUNT	TEXTSW_STORE_CHANGES_FILE
TEXTSW_EXTRAS_CMD_MENU	TEXTSW_SUBMENU_EDIT
TEXTSW_FILE	TEXTSW_MODIFIED
TEXTSW_FILE_CONTENTS	TEXTSW_SUBMENU_FILE
TEXTSW_FIRST	TEXTSW_SUBMENU_FIND
TEXTSW_FIRST_LINE	TEXTSW_SUBMENU_VIEW
TEXTSW_FONT	TEXTSW_UPPER_CONTEXT
TEXTSW_HISTORY_LIMIT	
XV_LEFT_MARGIN	XV_RIGHT_MARGIN

9
TTY Subwindows

The TTY (or *terminal emulator*) subwindow emulates a standard terminal, the principal difference being that the row and column dimensions of a TTY subwindow can vary from that of a standard terminal. In a TTY subwindow, you can run arbitrary programs, including a complete interactive shell. Or you can emulate terminal interface applications that use the *curses*(3X) terminal screen optimization package without actually running a separate process. The TTY subwindow accepts the standard ANSI escape sequences for doing ASCII screen manipulation, so you can use *termcap* or *termio* screen-handling routines. This chapter discusses the TTYSW package. Figure 9-1 shows the class hierarchy for the TTYSW package.

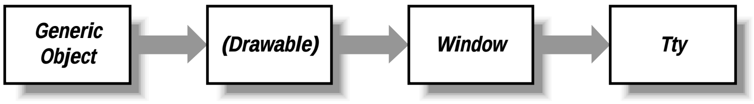


Figure 9-1. TTY package class hierarchy

9.1 Creating a TTY Subwindow

Programs using TTY subwindows must include the file <*xview/tty.h*>. Like all XView windows, you create a TTY subwindow by calling `xv_create()` with the appropriate type parameter, as in:

```
Tty tty;
tty = xv_create(frame, TTY, NULL);
```

The default TTY subwindow will fork a shell process and the user can use it interactively to enter commands. This program does not interact with the processing of the application in which the TTY subwindow resides; it is an entirely separate process. For example, if you want to start the TTY subwindow with another program, say *man*, you can do so by specifying the name of the program to run via the TTY_ARGV attribute, as shown in Example 9-1.

Example 9-1. The sample_tty.c program

```
/*
 * sample_tty.c -- create a base frame with a tty subwindow.
 * This subwindow runs a UNIX command specified in an argument
 * vector as shown below.  The example does a "man cat".
 */
#include <xview/xview.h>
#include <xview/tty.h>

char *my_argv[ ] = { "man", "cat",  NULL };

main(argc, argv)
char *argv[ ];
{
    Tty tty;
    Frame frame;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    frame = (Frame)xv_create(NULL, FRAME, NULL);
    tty = (Tty)xv_create(frame, TTY,
        WIN_ROWS,       24,
        WIN_COLUMNS,    80,
        TTY_ARGV,       my_argv,
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

The output of Example 9-1 is shown in Figure 9-2. Note that you can have only one TTY subwindow per process.

9.2 Driving a TTY Subwindow

You can drive the terminal emulator programmatically. There are procedures both to send input to the terminal emulator (as if the user had typed it in the TTY subwindow) and to send output (as if a program running in the TTY subwindow had output it). You can send input to a TTY subwindow programmatically with the function:

```
int
ttysw_input(tty, buf, len)
    Tty   tty;
    char *buf;
    int   len;
```

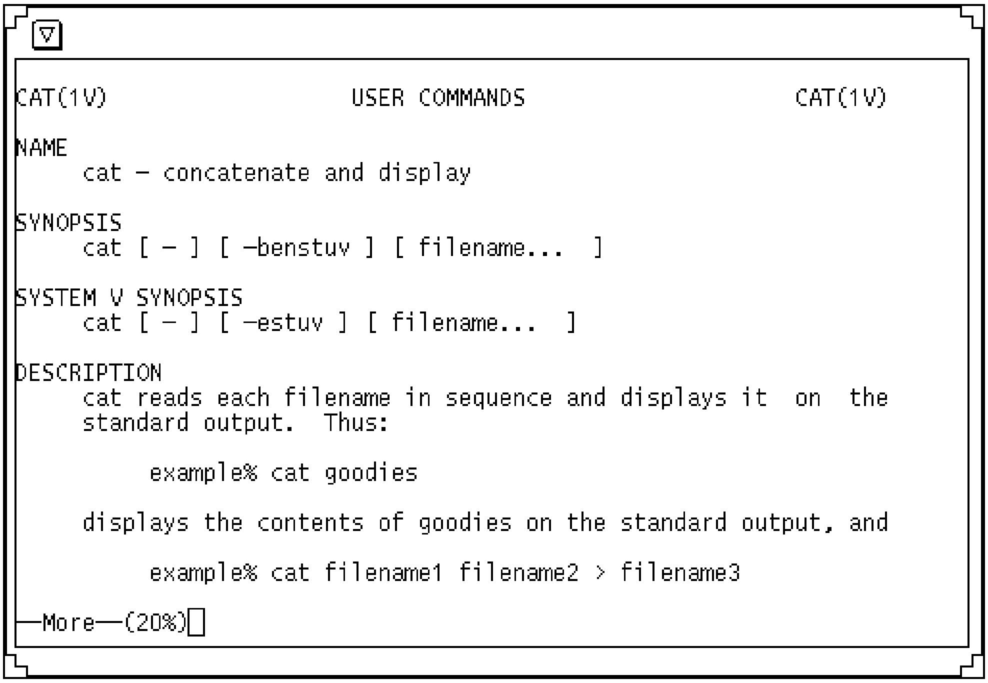
`ttysw_input()` appends the character sequence in `buf` that is `len` characters long onto `tty`'s input queue. It returns the number of characters accepted. The characters are treated as if they were typed from the keyboard in the TTY subwindow. `ttysw_input()` provides a simple way for a window program to send input to a program running in its TTY subwindow. You can send output to a TTY subwindow programmatically with the function:

```
int
ttysw_output(tty, buf, len)
    Tty   tty;
    char *buf;
    int   len;
```

`ttysw_output()` runs the character sequence in `buf` that is `len` characters long through the terminal emulator of `tty`. It returns the number of characters accepted. The effect is similar to executing this:

```
echo character_sequence > /dev/ttyN
```

where tty*N* is the pseudo-TTY associated with the TTY subwindow. `ttysw_output()` can be used to send ANSI escape sequences to the TTY subwindow.



```
CAT(1V)                   USER COMMANDS                    CAT(1V)

NAME
     cat - concatenate and display

SYNOPSIS
     cat [ - ] [ -benstuv ] [ filename...  ]

SYSTEM V SYNOPSIS
     cat [ - ] [ -estuv ] [ filename...  ]

DESCRIPTION
     cat reads each filename in sequence and displays it  on  the
     standard output.  Thus:

          example% cat goodies

     displays the contents of goodies on the standard output, and

          example% cat filename1 filename2 > filename3

--More--(20%)
```

Figure 9-2. Output of sample_tty.c

Note the differences between the input and output TTY routines. If an application is running in the TTY subwindow, then the characters sent to the TTY subwindow using `ttysw_input()` are sent to that program as its `stdin`. Characters sent to the TTY subwindow using `ttysw_ouput()` are sent to the TTY subwindow itself and have nothing to do with the application that might be running in the window.

The program in Example 9-2 creates a text subwindow in which the user can type input. There is a panel button called "Text to TTY" which, if selected, reads the data in the text subwindow and sends it to the TTY subwindow using `ttysw_input()`.

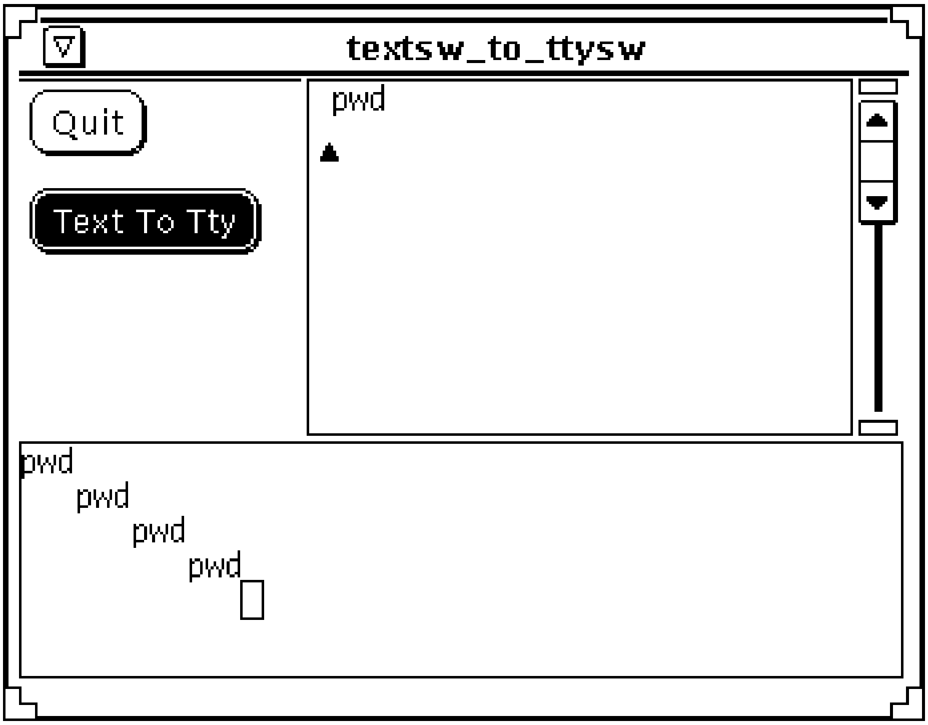
Example 9-2. The textsw_to_ttysw.c program

```
/*
 * textsw_to_ttysw.c -- send text from a text subwindow to a
 * tty subwindow using ttysw_output()
 */
#include <stdio.h>
#include <xview/panel.h>
#include <xview/xview.h>
#include <xview/textsw.h>
#include <xview/tty.h>
Textsw  textsw;
Tty     ttysw;

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    void        text_to_tty(), exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL, argv[0],
        NULL);
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT, PANEL_VERTICAL,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Text To Tty",
        PANEL_NOTIFY_PROC,      text_to_tty,
        NULL);
    window_fit(panel);

    textsw = (Textsw)xv_create(frame, TEXTSW,
        WIN_ROWS,       10,
        WIN_COLUMNS,    80,
        NULL);
    ttysw = (Tty)xv_create(frame, TTY,
        WIN_BELOW,      textsw,
        WIN_X,          0,
        TTY_ARGV,       TTY_ARGV_DO_NOT_FORK,
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

Example 9-2. The textsw_to_ttysw.c program (continued)

```
/*
 * callback routine for the panel button -- read text from textsw
 * and send it to the ttysw using ttysw_output()
 */
void
text_to_tty(item, event)
Panel_item item;
Event *event;
{
    char buf[BUFSIZ];

    (void) xv_get(textsw, TEXTSW_CONTENTS, 0, buf, sizeof buf);
    ttysw_output(ttysw, buf, strlen(buf));
}
```

Figure 9-3 shows the output of Example 9-2.



```
pwd
```

```
pwd
    pwd
        pwd
            pwd
```

Figure 9-3. Output of textsw_to_ttysw.c

Using `ttysw_output()` shows that the text is simply output to the dummy terminal emulator described by `ttysw`. `ttysw_input()` is useful for sending data as input to a program running in the TTY subwindow. For example, a window-based front end for a text editor could be written where all the common functions such as Save and Next Page can be programmed into panel buttons. Selecting one of those panel buttons would cause a constant string to be sent to the application to be processed as input. The *write filename* function in *vi* could have a button that uses `ttysw_input()` to send the string (w!\n) to the TTY subwindow containing the program.

9.3 Monitoring the Program in the TTY Subwindow

When you use the `TTY_ARGV` attribute to pass the name of a program to run to the TTY subwindow, the program runs as a forked child process. If the attribute:

```
TTY_QUIT_ON_CHILD_DEATH
```

is set to `TRUE`, then the application exits when the forked program exits. But, by default, this attribute is set to `FALSE`. You can use `TTY_PID` to monitor the state of the child process running in the TTY window via the Notifier using `notify_set_wait3_func()`. The client's `wait3()` function gets called when the state of the process in the TTY subwindow changes:*

```
#include <sys/wait.h>
static Notify_value     my_wait3();

    ...
    ttysw = xv_create(base_frame, TTY,
        TTY_ARGV,                   my_argv,
        NULL);
    child_pid = (int)xv_get(ttysw, TTY_PID);
    notify_set_wait3_func(ttysw, my_wait3, child_pid);
    ...
```

The `wait3()` function can then do something useful, such as destroying the TTY window or starting up another process. The code fragment below detects when any of the TTY subwindow's child processes has died.

```
static Notify_value
my_wait3(ttysw, pid, status, rusage)
Tty              ttysw;
int              pid;
union wait      *status;
struct rusage   *rusage;
{
    int    child_pid;

    notify_next_wait3_func(ttysw, pid, status, rusage);
    if (!(WIFSTOPPED(*status))) {
     /* rerun the program */
        xv_set(ttysw, TTY_ARGV, my_argv, NULL);
```

*This includes when the program *stops* in addition to when it exits.

```
            child_pid = (int)xv_get(ttysw, TTY_PID);
            notify_set_wait3_func(ttysw, my_wait3, child_pid);
        }
        return NOTIFY_DONE;
    }
```

You can set TTY_PID as well as get it, but if you set it, you are responsible for setting:

```
notify_set_wait3_func()
```

to catch the child's death. You are also responsible for directing the standard input and standard output of the child to the pseudo-TTY.

9.4 Talking Directly to the TTY Subwindow

Setting TTY_ARGV to TTY_ARGV_DO_NOT_FORK tells the system not to fork a child in the TTY subwindow. In combination with TTY_TTY_FD, this allows the tool to use standard I/O routines to read and write to the TTY subwindow by getting the file descriptor of the pseudo-TTY associated with the TTY subwindow. You can then use this to read and write to the pseudo-TTY using standard UNIX I/O routines.

Example 9-3 uses a TTY subwindow to create a pseudo *terminal* in which *curses* routines can be used. Five panel items are displayed. Along with the usual Quit button to exit the program, a Print button displays the text in the text panel item at the coordinates input in the X and Y numeric text items.

Example 9-3. The ttycurses.c program

```
/*
 * ttycurses.c -- An application that uses a tty subwindow that
 * emulates a tty so well, you can use curses(3x) routines in it.
 * This program does not handle resizes -- resizing the base frame
 * produces unpredictable results.  To handle resizing properly,
 * the application should install a resize event handler and
 * call endwin() followed by initscr() to reinitialize curses
 * to reflect the size of the window.
 *
 * cc ttycurses.c -lxview -lcurses -ltermlib
 */
#include <curses.h>
#undef WINDOW /* defined by curses.h -- needs to be undefined */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/tty.h>

/* panel items contain the x,y info for outputting text to the ttysw */
Panel_item  x, y, text;

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
```

Example 9-3. The ttycurses.c program (continued)

```
    Panel       panel;
    Tty         ttysw;
    char        buf[16];
    void        output(), exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);

    panel = (Frame)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,             "Quit",
        PANEL_NOTIFY_PROC,              exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,             "Print",
        PANEL_NOTIFY_PROC,              output,
        NULL);
    x = (Panel_item)xv_create(panel, PANEL_NUMERIC_TEXT,
        PANEL_LABEL_STRING,             "X:",
        PANEL_VALUE_DISPLAY_LENGTH,     3,
        NULL);
    y = (Panel_item)xv_create(panel, PANEL_NUMERIC_TEXT,
        PANEL_LABEL_STRING,             "Y:",
        PANEL_VALUE_DISPLAY_LENGTH,     3,
        NULL);
    text = (Panel_item)xv_create(panel, PANEL_TEXT,
        PANEL_LABEL_STRING,             "Text:",
        PANEL_VALUE_DISPLAY_LENGTH,     10,
        PANEL_VALUE,                    "X",
        NULL);
    window_fit(panel);

    ttysw = (Tty)xv_create(frame, TTY,
        WIN_BELOW,      panel,
        WIN_X,          0,
        TTY_ARGV,       TTY_ARGV_DO_NOT_FORK,
        NULL);
    window_fit(frame);

    dup2((int)xv_get(ttysw, TTY_TTY_FD), 0); /* dup2 closes 0 first */
    dup2((int)xv_get(ttysw, TTY_TTY_FD), 1); /* dup2 closes 1 first */

    /* initscr() initializes the curses package and determines
     * characteristics about the window as if it were a terminal.
     * The curses specific variables, LINES and COLS are now set
     * to the row and column sizes of the window.
     */
    initscr();

    xv_set(x, PANEL_MAX_VALUE, COLS-1, NULL);
    xv_set(y, PANEL_MAX_VALUE, LINES-1, NULL);
    sprintf(buf, "LINES: %d", LINES–1);
```

Example 9-3. The ttycurses.c program (continued)

```
    xv_set(frame, FRAME_LEFT_FOOTER, buf, NULL);
    sprintf(buf, "COLS: %d", COLS−1);
    xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);

    xv_main_loop(frame);
}
/*
 * callback routine for the <print> panel button.  Get the coordinates
 * and the text to print on the tty subwindow and use curses library
 * routines to render the text.
 */
void
output()
{
    int X = (int)xv_get(x, PANEL_VALUE);
    int Y = (int)xv_get(y, PANEL_VALUE);
    char *Text = (char *)xv_get(text, PANEL_VALUE);
    mvaddstr(Y, X, Text);
    refresh();
}
```

9.5 TTY Subwindow Function Key Escape Sequences

XView provides a default *.ttyswrc* file for the `shelltool` which provides the SunView escape key sequences for the following keys: L3, F1-F12, R1-R7, R9, R11, R13, and R15 (for more details see *$OPENWINHOME/lib/.ttyswrc*). To override the default file, place any `.ttyswrc` file in your *$HOME* directory. To avoid using any *.ttyswrc* file, place the following line in your *.Xdefaults* file:

```
term.useAlternateTtyswrc:  False
```

In addition, you will need to remove any *.ttyswrc* files from your *$HOME* directory. To specify a different alternate file than the one in *$OPENWINHOME/lib/.ttyswrc*, place the following into your *.Xdefaults* file:

```
term.alternateTtyswrc:  filename
```

where *filename* is the path and name of the file to use. For example:

```
term.alternateTtyswrc:  /usr/lib/ttyswrc
```

When using a Sun Type 4 keyboard with an X11 server other than OpenWindows 3.0, escape key sequences cannot be produced for keys F11 and F12 due to a limitation in the standard X11 keysym definition file. The base keysym file distributed by the X Consortium limits the number of unique function keys to a maximum of 35 (for more details see *$OPENWINHOME/include/X11/keysym.h*). The Sun Type 4 keyboard has 37 keysyms. For OpenWindows, there are two extra keysyms Sun_F36 and Sun_F37 which allow XView and other X-based programs to perform unique actions on the F11 and F12 keys (for more details see *$OPENWINHOME/include/X11/Sunkeysym.h*).

9.6 TTY Package Summary

The TTYSW procedures are shown in Table 9-1. Table 9-2 lists the TTYSW attributes. This information is described fully in the *XView Reference Manual*.

Table 9-1. TTY Subwindow Procedures

ttysw_input()
ttysw_output()

Table 9-2. TTY Subwindow Attributes

TTY_ARGV
TTY_ARGV_DO_NOT_FORK
TTY_CONSOLE
TTY_PAGE_MODE
TTY_PID
TTY_QUIT_ON_CHILD_DEATH
TTY_TTY_FD
WIN_FONT
WIN_SET_FOCUS

10
Scrollbars

Scrollbars are used to change what you view in a subwindow. For instance, in a text subwindow, scrollbars are used to scroll through a document. In a canvas subwindow, scrollbars can be used to see another portion of the paint window (which can be larger than the canvas subwindow). This chapter addresses specific functions of scrollbars themselves. These functions are applicable to any XView package that has scrollbars attached. If you are interested in how to utilize scrollbars for a particular package, you should consult the chapter that discusses that package.

OPEN LOOK describes scrollbars using the visual metaphor of an elevator riding on a cable, which is attached at both ends to anchors. Figure 10-1 shows a scrollbar from the *OPEN LOOK GUI Specification Guide*.

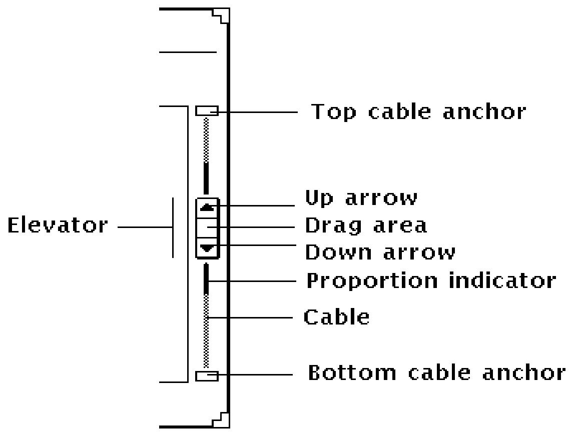


Figure 10-1. An OPEN LOOK scrollbar

The elevator contains directional arrows and a drag box. A subwindow can have vertical or horizontal scrollbars. Horizontal scrollbars are placed to the right of the subwindow while vertical scrollbars are placed at the bottom. OPEN LOOK defines precisely how scrollbars look and behave—the programmer or user cannot change it. All the programmer can control is the scrollbar's color, length, and various other common and generic attributes.

One of the functions of the scrollbar is the ability to *split a view*. The `OPENWIN` package provides objects such as text subwindows and canvases that may be split into several views; the scrollbar provides the functional interface. Certain XView packages, such as text subwindows, automatically create their own scrollbars. Canvases, on the other hand, require the programmer to create and attach scrollbars.

The scrollbar's look and feel is related to the size of the object it scrolls. Attributes are associated with each of the following terms:

Orientation	The orientation of a scrollbar indicates whether it is horizontal or vertical.
Object Length	The length of the object is registered with the scrollbar. The *proportional indicator* (the darkened part of the elevator cable) uses this value. For example, the object length for a text subwindow is the number of lines in the editing buffer.
Page Length	When the object length is larger than what the view window can contain, the overall area is broken up into *pages*. When the user selects the elevator cable, the scrollbar scrolls in page segments in the direction of the cursor (e.g., left, right, up, or down) relative to the elevator.
Unit Length	When the user clicks on the elevator arrows, the scrollbar scrolls one *unit*. Units are measured in pixels, so arbitrary or abstract objects that are to be scrolled should be measured in terms of pixels so that scrolling seems consistent with the object. For example, a text subwindow sets its scrollbar's unit length to the size of the characters in the font. Unit scrolling results in the window moving line by line up or down.
View Length	The view length is the same size as the height or width of the subwindow the scrollbar is associated with depending on the scrollbar's orientation.

Figure 10-2 illustrates the terminology used above.

10.1 Creating Scrollbars

The definitions necessary to use scrollbars are found in the header file <*xview/scrollbar.h*>. The basic scrollbar is created using the following code fragment:

```
Scrollbar scrollbar;

scrollbar = (Scrollbar)xv_create(owner, SCROLLBAR, NULL);
```

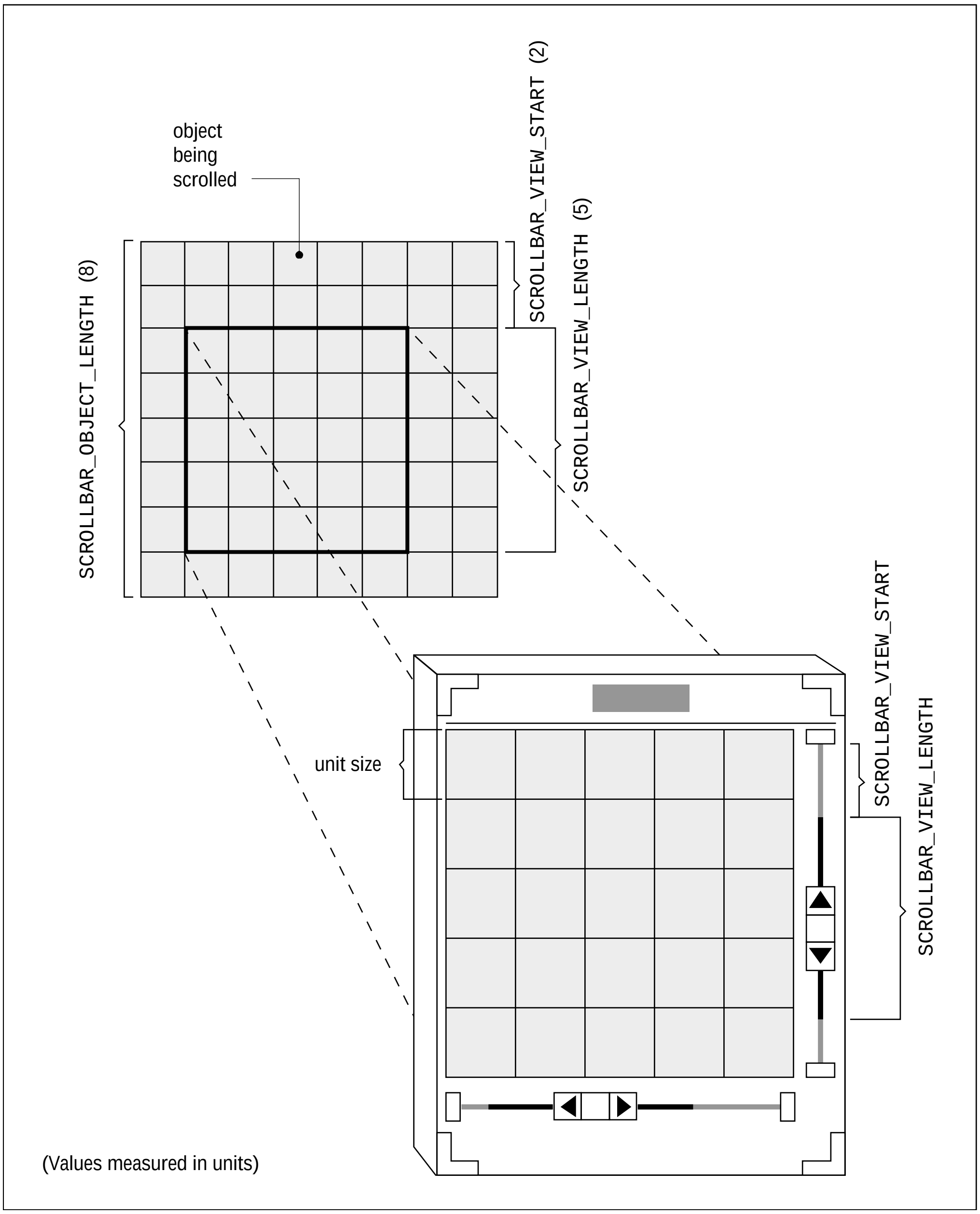


Figure 10-2. Relationship between a scrollbar and the object it scrolls

The `owner` must be an object subclassed from the `OPENWIN` package or the `FRAME` package. Figure 10-3 shows the class hierarchy for the `SCROLLBAR` package.

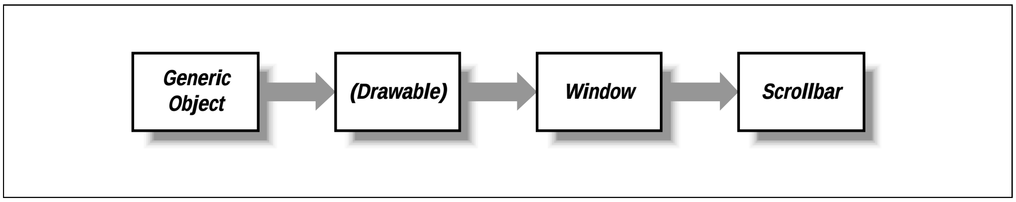


Figure 10-3. Scrollbar class hierarchy

The scrollbar inherits certain attributes from the parent while other attributes are initialized automatically. For example, if the owner of the scrollbar is a canvas, the scrollbar's color is inherited from the canvas, while the scrollbar's object length is set by the canvas explicitly; that is, you are not required to set it. This is usually desirable when creating objects that are used together.

10.2 Relationship Between Scrollbars and Objects

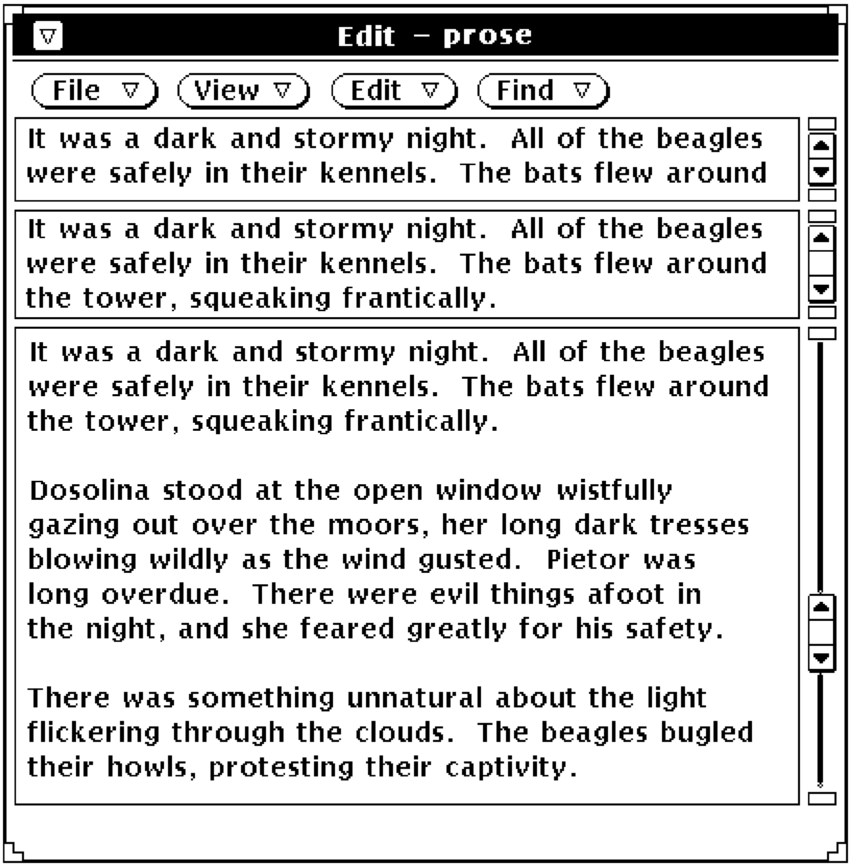
Most scrollbar attributes describe the relationship between the scrollbar and the object such as a canvas or text subwindow that is affected by scrolling. The foremost is `SCROLLBAR_PIXELS_PER_UNIT`, which describes the number of pixels in a scrolling *unit*. For text subwindows, the unit is the text width and height. For canvases, it is one pixel (by default). If you were to build a canvas subwindow intended to browse a set of 64x64 bitmaps, then you would set this to 64. Scrolling actions occur in scrollbar units, so this would mean that the clicking on one of the elevator arrows causes a scrolling movement of 64 pixels at a time. Most scrollbar attribute values are based on the *unit* value.

The size of the object itself (a graphic image, text stream or whatever) is stored as the `SCROLLBAR_OBJECT_LENGTH` while the size of the viewable window is represented as the scrollbar's `SCROLLBAR_VIEW_LENGTH`. After having been scrolled, the scrollbar's current offset into the object is reflected in `SCROLLBAR_VIEW_START`. When *paging* is done (selecting the *cable* portion of the elevator), the amount scrolled is set by `SCROLLBAR_PAGE_LENGTH`. These values are in object units, so to get their values in pixels, multiply by the value of `SCROLLBAR_PIXELS_PER_UNIT`.

The scrollbar manages its own events, resizes and repaints automatically. It is not necessary to interpose event handlers for the scrollbar. By default, the event handling mechanism determines the type of scrolling that has been done and changes the appropriate attributes. All `OPENWIN` objects that support scrollbars also redisplay the window to show the results of scrolling.

Even though you do not need to know when the scrollbar is scrolled to manage the scrolling, you might be interested in knowing when the scrolling action occurs. XView objects such as text subwindows that manage their own data (text, in this case) handle this automatically. See Chapter 5, *Canvases and Openwin*, for discussion on scrolling canvases.

If a window that has a scrollbar is resized, the scrollbar is resized accordingly. If the window is sized too small for all of the parts of the scrollbar to be visible or usable, then those parts cannot be available. At the very least, the scrolling *arrows* must be visible. Figure 10-4 shows a text subwindow that has been split twice. Notice the scrollbars to the right of the text subwindows. The uppermost window cannot be split again because the minimum size of the scrollbar has been reached.



It was a dark and stormy night. All of the beagles
were safely in their kennels. The bats flew around

It was a dark and stormy night. All of the beagles
were safely in their kennels. The bats flew around
the tower, squeaking frantically.

It was a dark and stormy night. All of the beagles
were safely in their kennels. The bats flew around
the tower, squeaking frantically.

Dosolina stood at the open window wistfully
gazing out over the moors, her long dark tresses
blowing wildly as the wind gusted. Pietor was
long overdue. There were evil things afoot in
the night, and she feared greatly for his safety.

There was something unnatural about the light
flickering through the clouds. The beagles bugled
their howls, protesting their captivity.

Figure 10-4. Splitting a text subwindow twice

10.3 An Example

Let's suppose that you want to display a list of icons that have dimensions of 64x64. You wish to display the icons in rows and columns in a canvas. Because there may be more icons than the canvas can display at once, you attach scrollbars to the canvas. When the user uses the scrollbars to view the icons, each scrolling action should scroll an entire icon or set of icons into view. *Paging* should scroll the next *page* of icons into view.

For demonstration purposes, rather than display actual icons, we present a grid where each *cell* in the grid represents an icon (see Figure 10-5).

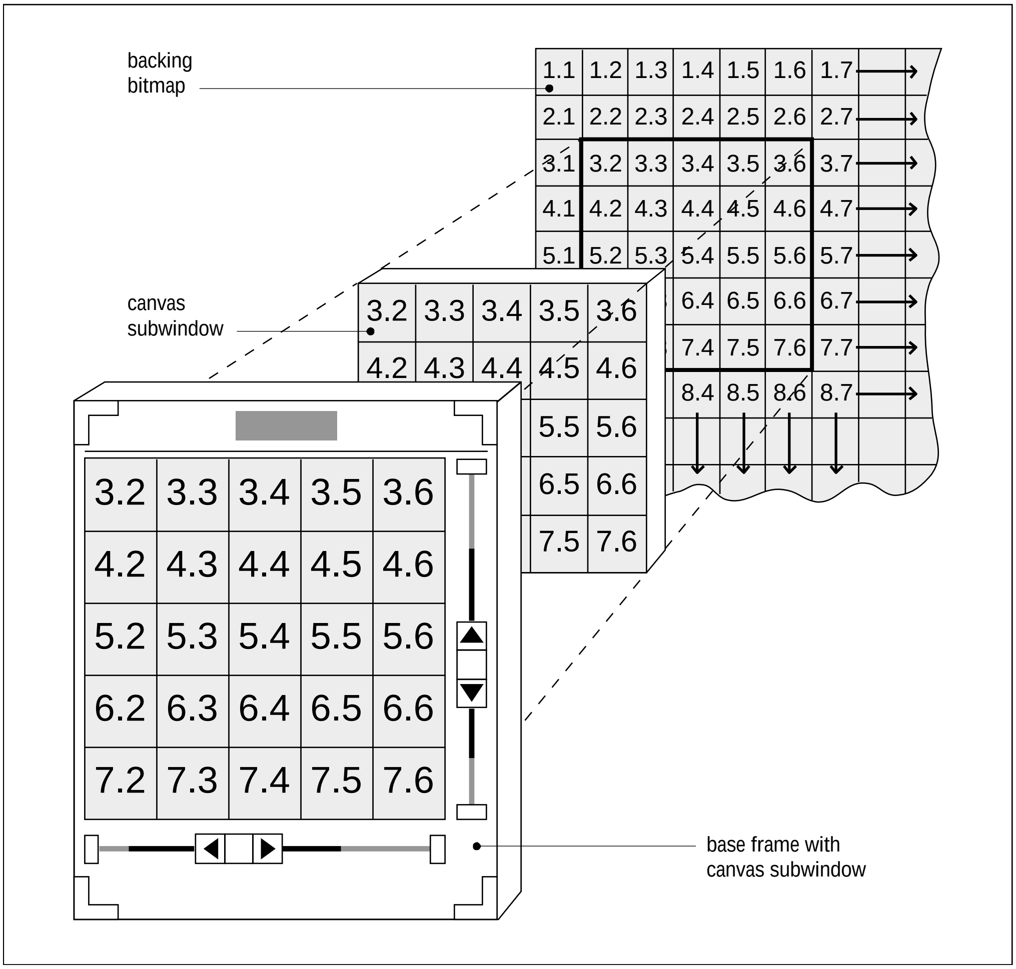


Figure 10-5. Model for scroll_cells.c

Each cell is considered a *unit* to the scrollbars, so the number of pixels per scrollbar-unit must be set to the size of the cell. Thus, the attribute SCROLLBAR_PIXELS_PER_UNIT is set to 64 for each scrollbar (the width and height are the same). With this attribute set, when the user selects an arrow on the scrollbar, an entire cell is scrolled into view (depending on which arrow is selected).

We set SCROLLBAR_PAGE_LENGTH to be the same as SCROLLBAR_VIEW_LENGTH to specify the paging size. When the user selects any part of the scrollbar cable, the view is paged and a new set of icons is scrolled into view (depending on which side of the elevator is selected). The page length could be set to one unit less than the view length, so that paging causes the last cell in the old block to be the first cell in the new block. Remember, the “lengths” mentioned here are given in *units*.

The setting of the scrollbar unit size also assures that the upper-left corner of a cell maps to the upper-left corner of the window so as not to display a portion of the cell. This guarantee cannot be made for the lower and right-hand edges of the window because we cannot control the resizing of the frame by the user.

In the program in Example 10-1, the variable cell_map is a Pixmap of depth 1. But, the depth is arbitrary—we use 1 because we know that the icons we are displaying are of depth 1. The canvas, on the other hand, may be any depth at all; color canvases have a depth greater than 1. Copying drawables of different depths onto one another is an X Protocol error, so we use XCopyPlane() to guarantee that the pixmap is rendered into the canvas correctly.

Example 10-1. The scroll_cells.c program

```
/*
 * scroll_cells.c -- scroll a bitmap of cells around in a canvas.
 * The cells are rectangular areas labeled with numbers which may
 * represent arbitrary data such as icon images.  The cell sizes are
 * defined to be 64 by 64 aligned in rows and columns.  This example
 * is used to demonstrate how to configure scrollbars to accommodate
 * arbitrary data within a window.
 */
#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>   /* Using Xlib graphics */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>
#include <xview/font.h>
#include <xview/xv_xrect.h>

#define CELL_WIDTH              64
#define CELL_HEIGHT             64
#define CELLS_PER_HOR_PAGE      5 /* when paging w/scrollbar */
#define CELLS_PER_VER_PAGE      5 /* when paging w/scrollbar */
#define CELLS_PER_ROW           8
#define CELLS_PER_COL           16

Pixmap          cell_map;       /* pixmap copied onto canvas window */
Scrollbar       horiz_scrollbar;
Scrollbar       vert_scrollbar;
```

Example 10-1. The scroll_cells.c program (continued)

```
GC              gc;             /* General usage GC */

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame       frame;
    Canvas      canvas;
    void        repaint_proc();

    /* Initialize, create frame and canvas... */
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);

    canvas = (Canvas)xv_create(frame, CANVAS,
        /* make subwindow the size of a "page" */
        XV_WIDTH,               CELL_WIDTH * CELLS_PER_HOR_PAGE,
        XV_HEIGHT,              CELL_HEIGHT * CELLS_PER_VER_PAGE,
        /* canvas is much larger than the window */
        CANVAS_WIDTH,           CELL_WIDTH * CELLS_PER_ROW + 1,
        CANVAS_HEIGHT,          CELL_HEIGHT * CELLS_PER_COL + 1,
        CANVAS_AUTO_EXPAND,     FALSE,
        CANVAS_AUTO_SHRINK,     FALSE,
        /* don't retain window -- we'll need
         * to repaint it all the time */
        CANVAS_RETAINED,        FALSE,
        /* we're using Xlib graphics calls in repaint_proc() */
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    repaint_proc,
        /* we'll be repainting over exposed areas,
         * so don't bother clearing */
        OPENWIN_AUTO_CLEAR,     FALSE,
        NULL);

    /*
     * Create scrollbars attached to the canvas.  When user clicks
     * on cable, page by the page size (PAGE_LENGTH).  Scrolling
     * should move cell by cell, not by one pixel (PIXELS_PER_UNIT).
     */
    vert_scrollbar = xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,            SCROLLBAR_VERTICAL,
        SCROLLBAR_PIXELS_PER_UNIT,      CELL_HEIGHT,
        SCROLLBAR_OBJECT_LENGTH,        CELLS_PER_COL,
        SCROLLBAR_PAGE_LENGTH,          CELLS_PER_VER_PAGE,
        SCROLLBAR_VIEW_LENGTH,          CELLS_PER_VER_PAGE,
        NULL);
    horiz_scrollbar = xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,            SCROLLBAR_HORIZONTAL,
        SCROLLBAR_PIXELS_PER_UNIT,      CELL_WIDTH,
        SCROLLBAR_OBJECT_LENGTH,        CELLS_PER_ROW,
        SCROLLBAR_PAGE_LENGTH,          CELLS_PER_HOR_PAGE,
        SCROLLBAR_VIEW_LENGTH,          CELLS_PER_HOR_PAGE,
```

Example 10-1. The scroll_cells.c program (continued)

```
        NULL);

    /*
     * create pixmap and draw cells into it ... this is the abstraction.
     * The cell_map is copied into the window via XCopyPlane in the
     * repaint procedure.
     */
    {
        short           x, y, pt = 0;
        Xv_Font         font;
        XPoint          points[256]; /* keep Xlib calls to a minimum */
        XGCValues       gcvalues;
        Display *dpy = (Display *)xv_get(canvas, XV_DISPLAY);

        font = (Xv_Font)xv_find(frame, FONT,
            FONT_NAME,          "icon",
            NULL);
        cell_map = XCreatePixmap(dpy, DefaultRootWindow(dpy),
            CELLS_PER_ROW * CELL_WIDTH + 1,
            CELLS_PER_COL * CELL_HEIGHT + 1,
            1); /* We only need a 1-bit deep pixmap */

        /* Create the gc for the cell_map -- since it is 1-bit deep,
         * use 0 and 1 for fg/bg values.  Also, limit number of
         * events generated by setting graphics exposures to False.
         */
        gcvalues.graphics_exposures = False;
        gcvalues.background = 0;
        gcvalues.foreground = 1;
        if (font)
            gcvalues.font = (Font)xv_get(font, XV_XID);
        gc = XCreateGC(dpy, cell_map,
            GCFont|GCForeground|GCBackground|GCGraphicsExposures,
            &gcvalues);

        if (!font) {
            /* dot every other pixel */
            for (x = 0; x <= CELL_WIDTH * CELLS_PER_ROW; x += 2)
                for (y = 0; y <= CELL_HEIGHT * CELLS_PER_COL; y += 2) {
                    if (x % CELL_WIDTH != 0 && y % CELL_HEIGHT != 0)
                        continue;
                    points[pt].x = x, points[pt].y = y;
                    if (++pt == sizeof points / sizeof points[0]) {
                        XDrawPoints(dpy, cell_map, gc, points, pt,
                            CoordModeOrigin);
                        pt = 0;
                    }
                }
            if (pt != sizeof points) /* flush remaining points */
                XDrawPoints(dpy, cell_map, gc,
                            points, pt, CoordModeOrigin);
        }
        /* Icon font not available.  Instead, label each cell
         * with a string describing the cell's coordinates.
         */
        for (x = 0; x < CELLS_PER_ROW; x++)
```

Example 10-1. The scroll_cells.c program (continued)

```
            for (y = 0; y < CELLS_PER_COL; y++) {
                char buf[8];
                if (!font) {
                    sprintf(buf, "%d,%d", x+1, y+1);
                    XDrawString(dpy, cell_map, gc,
                        x * CELL_WIDTH + 5, y * CELL_HEIGHT + 25,
                        buf, strlen(buf));
                } else {
                    buf[0] = x + y * CELLS_PER_COL;
                    XDrawString(dpy, cell_map, gc,
                        x * CELL_WIDTH, y * CELL_HEIGHT, buf, 1);
                }
            }
        /* we’re now done with the cell_map, so free gc and create
         * a new one based on the window that will use it.  Otherwise,
         * the GC may not work because of different depths.
         */
        if (font)
            xv_destroy(gc);
        XFreeGC(dpy, gc);
        gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
        gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
        gcvalues.plane_mask = 1L;
        gc = XCreateGC(dpy, DefaultRootWindow(dpy),
            GCForeground|GCBackground|GCGraphicsExposures, &gcvalues);
    }

    /* shrink frame to minimal size and start notifier */
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * The repaint procedure is called whenever repainting is needed in
 * a paint window.  Since the canvas is not retained, this routine
 * is going to be called any time the user scrolls the canvas.  The
 * canvas will handle repainting the portion of the canvas that
 * was in view and has scrolled onto another viewable portion of
 * the window.  The xrects parameter will cover the new areas that
 * were not in view before and have just scrolled into view.  If
 * the window resizes or if the window is exposed by other windows
 * disappearing or cycling through the window tree, then the number
 * of xrects will be more than one and we’ll have to copy the new
 * areas one by one.  Clipping isn’t necessary since the areas to
 * be rendered are set by the xrects value.
 */
void
repaint_proc(canvas, paint_window, dpy, win, xrects)
Canvas          canvas;
Xv_Window       paint_window;
Display         *dpy;
Window          win;
Xv_xrectlist    *xrects;
{
    int x, y;
```

Example 10-1. The scroll_cells.c program (continued)

```
    x = (int)xv_get(horiz_scrollbar, SCROLLBAR_VIEW_START);
    y = (int)xv_get(vert_scrollbar, SCROLLBAR_VIEW_START);

    for (xrects->count--; xrects->count >= 0; xrects->count--) {
        printf("top-left cell = %d, %d -- %d,%d %d,%d0, x+1, y+1,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y,
            xrects->rect_array[xrects->count].width,
            xrects->rect_array[xrects->count].height);

        XCopyPlane(dpy, cell_map, win, gc,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y,
            xrects->rect_array[xrects->count].width,
            xrects->rect_array[xrects->count].height,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y, 1L);
    }
}
```

10.4 Managing Your Own Scrollbar

A scrollbar may have *delayed binding*—that is, it may be created without an owner and attached to objects that were created separately.

In most cases, you would probably never need to create a scrollbar that was not part of a text subwindow or a canvas. These two packages handle all of the dirty work involved in managing and maintaining the types of attributes mentioned above. If you are using the `CANVAS` or `TEXTSW` packages, you do not need to worry about any of this. If you do try to create your own scrollbars and have them manage your own windows, you will probably find that you will have reinvented the wheel in the form of the `CANVAS` package.

If you are going to attempt this type of activity, you will need to follow these guidelines:

- Maintain the relationship between the object to be scrolled and the scrollbar itself. This includes using all the scrollbar attributes mentioned in Section 10.1, "Creating Scrollbars."
- Manage geometry (size, position, and orientation) of the scrollbar. You must place the scrollbars in the appropriate places around the object you intend to scroll. Typically, the scrollbars should match the width and height of the object being scrolled.
- Install appropriate `SCROLLBAR_NORMALIZE_PROC` and `SCROLLBAR_COMPUTE_SCROLL_PROC` procedures to change the display of the scrolling object.

10.4.1 Monitoring When Scrollbar Events Occur

When events take place in the scrollbar, the scrollbar normally interprets these events as *scrolling events* and adjusts itself appropriately. Since scrollbars are attached to objects such as canvases and text subwindows, those objects are also notified of the scrolling event so they can control the display of the data within their associated windows. For example, a canvas may get a `SCROLLBAR_REQUEST` event indicating that the user has initiated a scrolling action and that the object associated with the scrollbar needs to change its display by the requested amount.

The object to which the scrollbar is attached is set using the scrollbar attribute, `SCROLLBAR_NOTIFY_CLIENT`.* The internals to the scrollbar attempt to get information from this client, such as its size. For canvases, the *view* window is used. Since you normally query for user events on the canvas's *paint* window, this doesn't interfere with the scrollbar processing and also explains why your event handlers never see this event. For text subwindows, programmers normally do not concern themselves with events, so again, scrollbar processing is not affected.

If you are interested in managing the scrolling mechanisms of a scrollbar, or if all you need is to be notified of *when* the user invokes scrolling actions, you can install an event-interposing function on the scrollbar itself. This involves using the routine `notify_interpose_event_func()` discussed in Chapter 20, *The Notifier*. You can set one up in the following way:

```
Canvas      canvas;
Scrollbar   sb;
Notify_func monitor_scroll();
 . . .
canvas = xv_create(frame, CANVAS, NULL);

sb = xv_create(canvas, SCROLLBAR, NULL);

notify_interpose_event_func( xv_get(sb, SCROLLBAR_NOTIFY_CLIENT),
                             monitor_scroll, NOTIFY_SAFE);
 . . .
```

When the user invokes any scrolling events in the scrollbar, the function `monitor_scroll` is called with the event type set to `SCROLLBAR_REQUEST`.

Example 10-2 demonstrates how this is done in an application. By default, a canvas is set up of size `1000x1000` and a scrollbar attached. When the user scrolls the canvas, the function `monotir_scroll()` is called, which prints information about how much the canvas scrolled.

Example 10-2. The scrollto.c program

```
/* scroll_to.c -- demonstrate how to monitor the scrolling
 * requests invoked by the user.  Requests can be monitored,
 * ignored or changed programmatically.  This program creates
```

*While this is a settable attribute, it is not recommended that you change the notify client for scrollbars for the current release.

Example 10-2. The scrollto.c program (continued)

```
 * a canvas window by default or a textsw with the -textsw
 * command line option.  Both contain a scrollbar.
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/textsw.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame           frame;
    Textsw          textsw;
    Canvas          canvas;
    Scrollbar       sbar;
    Notify_value    monitor_scroll();

    (void) xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create(NULL, FRAME, NULL);

    if (argc > 1 && !strcmp(argv[1], "-textsw")) {
        textsw = xv_create(frame, TEXTSW,
            TEXTSW_FILE_CONTENTS, "/etc/termcap",
            NULL);
        sbar = xv_get(textsw, TEXTSW_SCROLLBAR);
    } else {
        canvas = xv_create(frame, CANVAS,
            CANVAS_WIDTH, 1000,
            CANVAS_HEIGHT, 1000,
            CANVAS_AUTO_SHRINK, FALSE,
            CANVAS_AUTO_EXPAND, FALSE,
            NULL);
        sbar = xv_create(canvas, SCROLLBAR,
            SCROLLBAR_DIRECTION, SCROLLBAR_VERTICAL,
            SCROLLBAR_PIXELS_PER_UNIT, 10,
            NULL);
    }
    notify_interpose_event_func(xv_get(sbar, SCROLLBAR_NOTIFY_CLIENT),
        monitor_scroll, NOTIFY_SAFE);

    xv_main_loop(frame);
}

/*
 * To change the behavior of the scrolling of the canvas, do not pass
 * on the event via notify_next_event_func() when the event type is
 * SCROLLBAR_REQUEST.
 */
Notify_value
monitor_scroll(client, event, sbar, type)
Notify_client     client;
Event            *event;
Scrollbar         sbar;
```

Example 10-2. The scrollto.c program (continued)

```
Notify_event_type type;
{
    int     view_start, last_view_start, pixels_per, is_neg = 0, total;

    if (event_id(event) == SCROLLBAR_REQUEST) {
        view_start = (int)xv_get(sbar, SCROLLBAR_VIEW_START);
        last_view_start = (int)xv_get(sbar, SCROLLBAR_LAST_VIEW_START);
        pixels_per = (int)xv_get(sbar, SCROLLBAR_PIXELS_PER_UNIT);
        if ((total = view_start - last_view_start) < 0)
            total = -total, is_neg = 1;
        printf("scrolled from %d to %d: %d pixels (%d units) %s\n",
            last_view_start, view_start, pixels_per * total, total,
            is_neg? "up" : "down");
    }
    return notify_next_event_func(client, event, sbar, type);
}
```

If the command-line option `-textsw` is given, a text subwindow is used instead of a canvas.

The application can change the scrolling behavior by not calling the function `notify_next_event_func()`. It can choose to set the scrollbar to any position it desires via `xv_set()` and the appropriate attributes, or it can ignore the scroll request entirely. In any event, the function should return either `NOTIFY_DONE` or the return value of `notify_next_event_func()`.

The parameters to `monitor_scroll()` include the `client` (the object set by `SCROLLBAR_NOTIFY_CLIENT`), as well as the `event` (which is probably `SCROLLBAR_REQUEST`), the scrollbar itself, and an unused `type` parameter indicating whether this was called via `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`.

10.4.2 Providing a Scrollbar Compute Procedure

Normally, the starting position of the scrollbar's current view is computed by its package's default scroll procedure, `scrollbar_default_compute_scroll_proc()`. You can install you own scroll procedure using the `SCROLLBAR_COMPUTE_SCROLL_PROC` attribute. This procedure converts the physical scrollbar information into client object information; that is, it returns the offset and the object length of the object to scroll. The form of the compute scroll routine is:

```
void
scrollbar_compute_scroll_proc(sb, pos, length, motion,
                                  &offset, &object_length)
    Scrollbar       sb;
    int             pos;
    int             length;
    Scroll_motion   motion;
    unsigned long  offset;
    unsigned long  object_length;
```

This procedure computes the offset, and the scrollbar package will scroll to this offset into the object when the compute procedure returns. This function should return the offset and object_length where pos is the position in the cable. The `default_compute_scroll_proc` can be called to perform the normal scroll. If a `normalize_proc` is not set then the offset becomes the viewstart, after bounds checking, and the scrollbar package will scroll to this offset into the object. For example:

Example 10-3. Scrollbar compute scroll procedure example

```
void
compute_scroll1(scrollpub,pos,avail_cable,motion,offset,object_len)
        Scrollbar scrollpub;
        int pos;
        int avail_cable;
        Scroll_motion motion;
        unsigned long *offset;
        unsigned long *object_len;
        {
        int     new_start = TEXTSW_CANNOT_SET;
        int     lines = 0;

        *obj_length = es_get_length(folio->views->esh);

        switch (motion) {
        case SCROLLBAR_ABSOLUTE:
                if (length == 0)
                new_start =  pos;
                else
                new_start = *obj_length *  pos / length;
                break;
        case SCROLLBAR_POINT_TO_MIN:
        case SCROLLBAR_MIN_TO_POINT:{
                if (lines == 0)
                        lines++;        /* Always make some progress */
                if (motion == SCROLLBAR_MIN_TO_POINT)
                        lines = -lines;
                }
                break;
        case SCROLLBAR_PAGE_FORWARD:
                lines = line_table.last_plus_one - 2;
                break;
        case SCROLLBAR_PAGE_BACKWARD:
                lines = last_plus_one + 2;
                break;
        case SCROLLBAR_LINE_FORWARD:
                lines = 1;
                break;
        case SCROLLBAR_LINE_BACKWARD:
                lines = -1;
                break;
        case SCROLLBAR_TO_START:
                new_start = 0;
                break;
        case SCROLLBAR_TO_END:
                new_start = *obj_length;
                break;
        default:
                break;
        }
```

Example 10-3. Scrollbar compute scroll procedure example (continued)

```
        xv_set(sb, SCROLLBAR_VIEW_LENGTH, last_plus_one - first,
            0);
        *offset = first;
        return (XV_OK);
}
```

In this example, the `textsw` package keeps track of the object size based on the number of characters in the view. When the user scrolls, the object length will most probably change, so the `textsw` package uses its own compute scroll procecure to calculate a new object length and offset each time there is a scroll request. This is needed for cases when the user scrolls backwards and data is still coming into the `textsw`, so the object length grows (the proportional indicator shrinks).

10.4.2.1 Indicating scrollbar motion

The attribute `SCROLLBAR_MOTION` provides the scrolling motion that resulted during a scrollbar_request event. This attribute is get only. Possible valid motions returned are:

```
ABSOLUTE
POINT_TO_MIN (from here_to_top on menu)
PAGE_FORWARD
LINE_FORWARD
MIN_TO_POINT (from top_to_here on menu)
PAGE_BACKWARD
LINE_BACKWARD
TO_END
TO_START
PAGE_ALIGNED
```

10.4.3 Providing a Scrollbar Normalize Procedure

The scrollbar package provides for a special offset routine that may be used to adjust the new scroll position before the scrollbar package scrolls to the starting location computed by the scrollbar compute procedure. This special offset routine, called the normalize procedure, allows you to perform a scroll adjustment when, for example, the new scroll position would split an object in the view. By default, no normalize procedure is specified and the scrollbar package scrolls to the starting location computed by the compute procedure, as shown in the previous section. Use `SCROLLBAR_NORMALIZE_PROC` to name a normalize procedure used to adjust the offset.

The function set with `SCROLLBAR_NORMALIZE_PROC` should return vstart. The function takes the offset given by the `compute_proc` and adjusts it. The scrollbar package will then scroll to this offset into the object. The form of the normalize scroll routine is:

```
void
my_scrollbar_normalize_proc(sb, voffset, motion, vstart)
   Scrollbar       sb;
   long unsigned   offset;
```

```
        Scroll_motion  motion;
        long unsigned  *vstart; /* new offset, this is the new view start*/
```

See the following example:

Example 10-4. Scrollbar normalize procedure example

```
panel_normalize_scroll(sb, offset, motion, vs)
                Scrollbar       sb;
                long unsigned   offset;
                Scroll_motion   motion;
                long unsigned  *vs; /* new offset  == new viewstart */
        {
                line_ht = (int) xv_get(sb, SCROLLBAR_PIXELS_PER_UNIT);

                /* If everything in the panel is in view, then don't scroll. */
                if ((int) xv_get(sb, SCROLLBAR_OBJECT_LENGTH) <=
                        (int) xv_get(sb, SCROLLBAR_VIEW_LENGTH))
                        return (*vs = offset);

                switch (motion) {
                case SCROLLBAR_ABSOLUTE:
                case SCROLLBAR_LINE_FORWARD:
                case SCROLLBAR_TO_START:
                        align_to_max = TRUE;
                        scrolling_up = TRUE;
                        break;

                case SCROLLBAR_PAGE_FORWARD:
                case SCROLLBAR_TO_END:
                        align_to_max = TRUE;
                        scrolling_up = TRUE;

                        break;

                case SCROLLBAR_POINT_TO_MIN:
                        align_to_max = TRUE;
                        scrolling_up = TRUE;
                        break;

                case SCROLLBAR_MIN_TO_POINT:
                        align_to_max = TRUE;
                        scrolling_up = FALSE;
                        break;

                case SCROLLBAR_PAGE_BACKWARD:
                case SCROLLBAR_LINE_BACKWARD:
                        align_to_max = FALSE;
                        scrolling_up = FALSE;
                        break;
                }

         *vs = offset;
         return (XV_OK);
        }
```

The panel package uses this to ensure panel items are aligned properly and partial items are not visible. The scrollbar package calls `scrollbar_compute_scroll_proc` and `scrollbar_normalize_proc` in that order whenever any scrolling is done.

10.5 Scrollbar Package Summary

The procedures and macros in the `SCROLLBAR` package are listed in the next two tables. Table 10-1 lists the procedure for the `SCROLLBAR` package. Table 10-2 lists the attributes in the `SCROLLBAR` package. This information is described fully in the *XView Reference Manual*.

Table 10-1. Scrollbar Procedures

`scrollbar_paint()`
`scrollbar_default_compute_scroll_proc()`

Table 10-2. Scrollbar Attributes

SCROLLBAR_DIRECTION	SCROLLBAR_PAGE_LENGTH
SCROLLBAR_LAST_VIEW_START	SCROLLBAR_PIXELS_PER_UNIT
SCROLLBAR_MENU	SCROLLBAR_SPLITTABLE
SCROLLBAR_NOTIFY_CLIENT	SCROLLBAR_VIEW_LENGTH
SCROLLBAR_OBJECT_LENGTH	SCROLLBAR_VIEW_START
SCROLLBAR_MOTION	

This page intentionally left blank

to preserve original page counts.

11 Menus

Menus play an important role in an application’s user interface. An OPEN LOOK menu may display text or graphics. Menus may be attached to most XView objects such as *menu buttons*, *scrollbars* or *text subwindows*, or they may exist independently from objects and be displayed on demand.

The user may cause a menu to be pinned up by selecting an optional *pushpin* in the pop-up menu. When this happens, the menu is taken down and a corresponding command frame is put up at the same location. Panel items in the pinup window correspond to the menu items in the menu. Once a menu has been pinned up, the user continues to interact with it just as if the menu were popped up each time. Menus that are used frequently are good candidates for having pushpins so the user does not have to repeat the sequence of redisplaying the menu to make selections.

OPEN LOOK requires that menus have titles. Menus or submenus that originate from *menu buttons* or *pullright* items do not need to have titles, since the name of the menu button or menu item acts as the title.

Fonts may not be specified in either menu items or menu titles; menu items follow the same constraints outlined for *panel buttons*. However, if text is not used, then menu items may contain graphic images, in which case, the font is of no concern. That is, you could specify a `Server_image` that has a string rendered in a particular font.

11.1 Menu Types

There are three different types of menus: *pop-up*, *pulldown*, and *pullright* menus. The general term *pop-up menu* may describes all three types in certain contexts since menus are *popped up*. However, pulldown and pullright menus have distinct characteristics that make them unique.

11.1.1 Pop-up Menus

Pop-up menus are displayed when the user selects the MENU mouse button over XView objects such as *scrollbars* or *text subwindows*. An OPEN LOOK window manager also utilizes pop-up menus in the root window and from base frame title bars. XView objects handle the display of menus automatically. Applications may wish to track `ACTION_MENU` events in objects such as canvases and display their own pop-up menus. Figure 11-1, from the *OPEN LOOK GUI Specification Guide*, shows a *Window* menu-generated from the title bar of an OPEN LOOK base frame.

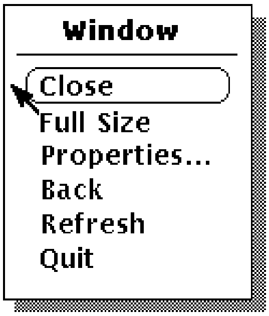


Figure 11-1. The Window menu

11.1.2 Pulldown Menus

Pulldown menus are attached to *menu buttons*. Menu buttons have a set of choices associated with them that the user can access only via the pulldown menu. When the user presses the MENU mouse button over a menu button, the choices are displayed in the form of a pulldown menu. If the menu button is selected using the SELECT button, the default menu item is selected. See Chapter 7, *Panels*, for details on creating menu buttons. Figure 11-2, from the *OPEN LOOK GUI Specification Guide*, shows sample pulldown menus activated from a menu button.

11.1.3 Pullright Menus

OPEN LOOK provides for items in the menu to have *pullright menus* associated with them. Also called *cascading menus*, these menus are activated from the user dragging the MENU mouse button to the right of a menu item that has an arrow pointing to the right. The cascading menu that results is a pop-up menu that can also have menu items with pullrights attached. Figure 11-3, from the *OPEN LOOK GUI Specification Guide*, shows a pullright menu originating from a menu item in a pulldown menu.

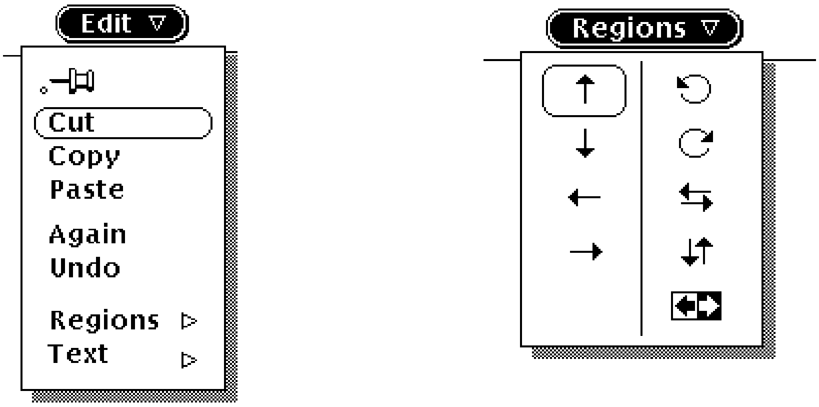


Figure 11-2. Menu buttons each with a pulldown menu

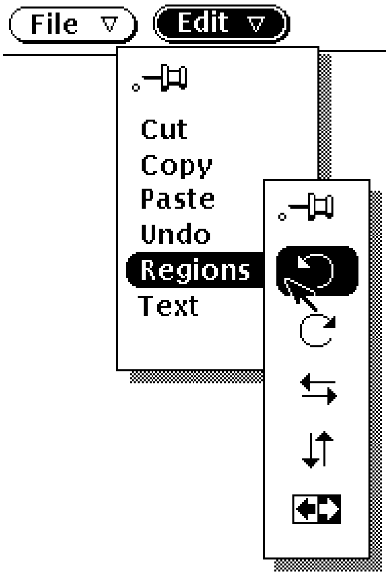


Figure 11-3. Pushpins in a menu and a submenu

11.2 Menu Items

In addition to the menu types, there are different types of menu items: *choice*, *exclusive*, and *nonexclusive*. The different menu item types may be associated with each type of menu.

Each menu has a *default selection* associated with it. This item is displayed uniquely from other menu items and designates a default action to take if the user wants to select the menu without displaying it (see *pulldown menus* below). Typically, the 0th item in the menu is the default, but that may be changed either by the application or by the user.

11.2.1 Choice Items

The *choice* item is the default menu item type used when a menu is created. The default selection in a menu has a ring around it. When a pop-up menu is displayed, it is positioned so that the mouse is pointing at the default item. Choice menu items may have pullright menus associated with them, in which case there is a pullright arrow at the right side of the item. If the selection of a menu item brings up a dialog box (command frame), then the label for the menu item typically ends in ellipses (. . .).

11.2.2 Exclusive Items

When a choice item is selected, an action is taken and the menu forgets about it. Exclusive menu items retain the fact that they are selected even after the menu has popped down. If the user selects a new item, the new item is remembered. Because this is an exclusive menu, only one choice may be selected at a time. The *default* item is indicated by a double-lined box around the item. Figure 11-4 is from the *OPEN LOOK GUI Specification Guide*.

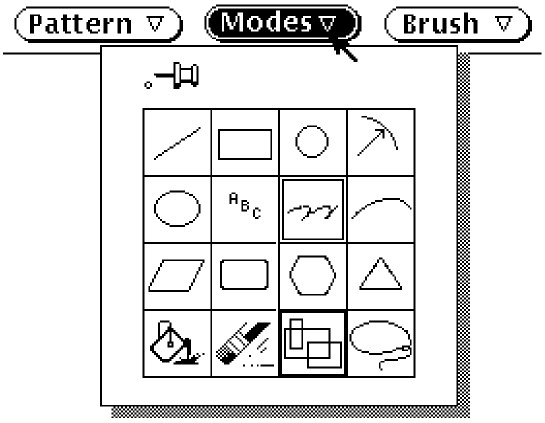


Figure 11-4. Exclusive settings on a menu

When exclusive settings are used on menus, the current choice has a bold border when the pointer is not on a menu choice. When the user drags the pointer onto other settings, the bold border follows the pointer. Exclusive choice menus may not have items with pullright menus.

11.2.3 Nonexclusive Items

Also called *toggle items*, menus that have toggle items support multiple choices from the menu to be selected at the same time. That is, the user may *toggle* whether a particular choice is selected. This action has no affect on the other menu items. Figure 11-5 shows an example of a menu that has items and a submenu that has nonexclusive settings.

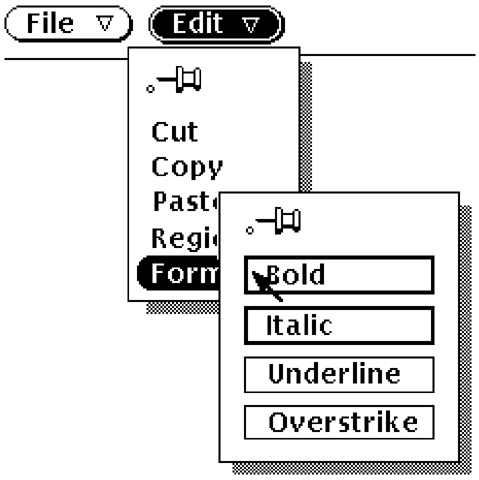


Figure 11-5. Nonexclusive settings on a submenu

In this figure, the chosen settings on the submenu are Bold and Italic. The choices not selected are Underline and Overstrike.

11.3 Creating Menus

The header file for the `MENU` package is <*xview/openmenu.h*>, but the file is already included by <*xview/xview.h*>. Another name for the `MENU` package is `MENU_COMMAND_MENU`. The basic menu is created using `xv_create()`:

```
Menu menu;

menu = (Menu)xv_create(server, MENU, NULL);
```

Menus (and sometimes menu items) are discrete objects that may have *delayed binding*. That is, they may be created independently from any XView object and attached later using attributes specific to that object, such as the way that PANEL_ITEM_MENU can be used to attach an existing menu to a *menu button*. However, the association between menus and the items they are attached to does not imply "ownership."

The owner of a menu is a server object. By default, (if NULL is specified as the owner) the default server is used. Menus may be used only on the server specified; they may not be shared across different servers. Thus, the menu owner is only a concern for applications that spread across multiple servers. See Chapter 15, *Nonvisual Objects*, for details on opening a connection to different servers.

The *parent* of a menu, however, may be a pullright item from another menu. See Section 11.8, "Pullright Menus," later in this chapter.

Figure 11-6 shows the class hierarchy for the a menu object.

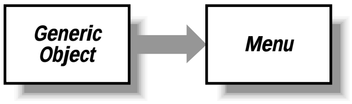


Figure 11-6. Menu class hierarchy

Exclusive menus are created using the MENU_CHOICE_MENU package, as in the following example:

```
Menu menu;

menu = (Menu)xv_create(NULL, MENU_CHOICE_MENU,
    MENU_STRINGS,  "choice1", "choice2", "choice3", NULL,
    NULL);
```

Nonexclusive menus are created using the MENU_TOGGLE_MENU package, as in the following example:

```
Menu menu;
Server_image image1, image2, image3;

menu = (Menu)xv_create(NULL, MENU_TOGGLE_MENU,
    MENU_IMAGES,  image1, image2, image3, NULL,
    NULL);
```

11.4 Displaying Menus

Menus are displayed (popped up) using the function `menu_show()`.* It displays the specified menu and immediately returns. The function takes the form:

```
void
menu_show(menu, window, event, NULL);
    Menu          menu;
    Xv_Window     window;
    Event        *event;
```

The `menu` is a menu created from `xv_create()` or a menu extracted from an existing XView object (such as a button menu). The X window associated with the menu calls `XGrabPointer()` to grab the server's mouse events. The pointer grab stays in effect until the user releases the MENU mouse button (e.g., the `ACTION_MENU` action with `event_is_up()` being `TRUE`). This is independent of the event that caused the menu to be displayed. Releasing the MENU button results in the user having either made a selection, *not* made a selection or pinned up the menu (provided that the menu has a pushpin).

The `window` attribute defines the window where the menu appears. The `event` parameter contains the event that caused the decision to display the menu. The most common use for it is to extract the *x,y* coordinate pair so as to remember the location of the pointer at the time the menu was displayed. This event structure can be retrieved later by calling:

```
Event *event = (Event *)xv_get(menu, MENU_FIRST_EVENT);
```

Similarly, when the user releases the MENU button, this event can be retrieved using the `MENU_LAST_EVENT` attribute.

The last parameter to `menu_show()` must be `NULL`. It actually represents a list of attribute-value pairs but is used internally by other XView packages that utilize menus.

Menus can also be created at run time by procedures that are called whenever a menu is needed. This is covered in Section 11.9, "Menu-generating Procedures."

The routine `MENU_DONE_PROC` is called whenever a pop-up menu has been taken down (after a menu item has been selected), pinned up, or simply dismissed without a selection being made. This overrides the default action of setting `XV_SHOW` to `FALSE`, so this responsibility lies with the `MENU_DONE_PROC` routine.

*This function is called internally by other XView objects such as scrollbars, menu buttons, and text subwindows to display menus associated with them.

11.5 A Simple Program

Given the information provided so far, we can demonstrate how to pop up a menu. Example 11-1 shows how the canvas object tracks pointer events and calls `menu_show()` when the `ACTION_MENU` event occurs.

Example 11-1. The simple_menu.c program

```
/*
 * simple_menu.c -
 * Demonstrate the use of an XView menu in a canvas subwindow.
 * A Menu is brought up with the MENU mouse button.  The choices
 * in the menu toggle the display of the scrollbar next to the canvas.
 */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>

#define SCROLLBAR_KEY   100
#define MENU_KEY        200

main(argc,argv)
int     argc;
char    *argv[];
{
    Frame       frame;
    Canvas      canvas;
    Scrollbar   scrollbar;
    Menu        menu;
    void        menu_notify_proc(), pw_event_proc();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);


    /*
     *  Create a frame, canvas and menu.
     *  A canvas receives input in its canvas_paint_window().
     */
    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,    argv[0],
        NULL);
    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,       300,
        XV_HEIGHT,      200,
        NULL);
    scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,    SCROLLBAR_VERTICAL,
        NULL);

    menu = (Menu)xv_create(NULL, MENU,
        MENU_TITLE_ITEM,        "Scrollbar",
        MENU_STRINGS,           "On", "Off", NULL,
        MENU_NOTIFY_PROC,       menu_notify_proc,
        XV_KEY_DATA,            SCROLLBAR_KEY, scrollbar,
        NULL);
```

Example 11-1. The simple_menu.c program (continued)

```
    xv_set(canvas_paint_window(canvas),
        WIN_EVENT_PROC,         pw_event_proc,
        XV_KEY_DATA,            MENU_KEY, menu,
        NULL);

    window_fit(frame);
    window_main_loop(frame);
}

/*
 * menu_notify_proc - toggle the display of the scrollbar
 * based on which menu item was chosen.
 */
void
menu_notify_proc(menu, menu_item)
Menu menu;
Menu_item menu_item;
{
    char *menu_choice = (char *)xv_get(menu_item, MENU_STRING);
    int show_it = !strcmp(menu_choice, "On");

    xv_set(xv_get(menu, XV_KEY_DATA, SCROLLBAR_KEY),
        XV_SHOW,        show_it,
        NULL);
}

/*
 * Call menu_show() to display menu.
 */
void
pw_event_proc(canvas_pw, event)
Xv_Window       canvas_pw;
Event *event;
{
    if (event_action(event) == ACTION_MENU && event_is_down(event)) {


        Menu menu = (Menu)xv_get(canvas_pw, XV_KEY_DATA, MENU_KEY);
        menu_show(menu, canvas_pw, event, NULL);
    }
}
```

In Example 11-1 above, *simple_menu.c* shows the simplest and most common method for creating and using pop-up menus. The menu, menu items, and callback routine are all created at the same time using the call:

```
menu = (Menu)xv_create(NULL, MENU,
    MENU_TITLE_ITEM,        "Scrollbar",
    MENU_STRINGS,           "On", "Off", NULL,
    MENU_NOTIFY_PROC,       menu_notify_proc,
    NULL);
```

Figure 11-7 shows the result of running *simple_menu.c* and selecting the menu.

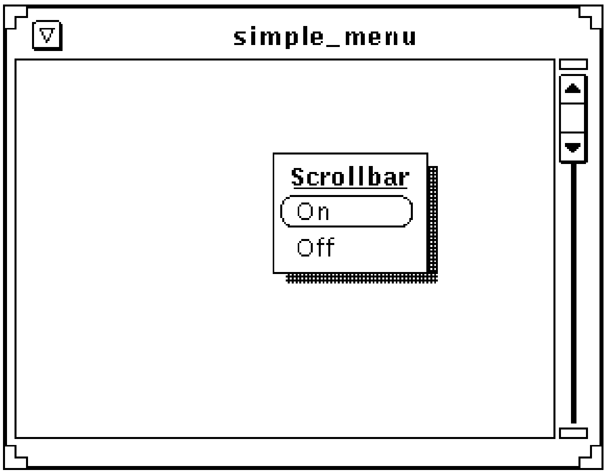


Figure 11-7. Output of simple_menu.c when the menu is popped up

Since this menu is not attached to a menu button panel item and is not a pullright menu of another menu item, a title bar is added using `MENU_TITLE_ITEM`. When a menu has a title, the 0th menu item is the menu title. In the example above, the 0th menu item is the title item labeled "Scrollbar." The first menu item is the first item created after the title. Above, the first menu item is the item labeled "On" and the second menu item is the item labeled "Off." When there is no title, the first menu item is placed in position zero. Add a title bar using `MENU_TITLE_ITEM` *only* if the menu is a pop-up menu. Using titles on menus originating from menu buttons is *not* OPEN LOOK-compliant.

The attribute `MENU_STRINGS` takes a list of strings and creates a menu item for each string.

NOTE

> The `MENU` package, in contrast to the `PANEL` package, does not save strings which you pass in as the menu item's label (string). You should either pass a constant string, as in the example above, or static storage that you have dynamically allocated (e.g., `malloc()`).

`menu_notify_proc()` is a routine that is called whenever any of the menu items are selected. The routine is passed a handle to the menu and the menu item selected. The canvas serves no other purpose than to capture events—the event callback routine for the canvas determines if the user generated the `ACTION_MENU` event and, if so, calls `menu_show()`.

The use of `XV_KEY_DATA` is used to associate one object with another. In this case, we associate the scrollbar with the menu so when the menu callback routine is called, the scrollbar can be retrieved easily. The menu, on the other hand, is associated with the canvas's paint window since that window is going to get the event that pops up the menu.

The use of XV_KEY_DATA removes the need for the menu and the scrollbar to be global variables. It also clarifies the association between the objects that would otherwise not be as clear. Other examples of XV_KEY_DATA, including a complete discussion on usage, can be found in Chapter 7, *Panels*, and Chapter 3, *Creating XView Applications*.

11.6 Creating Menu Items

As noted, the use of MENU_STRINGS results in the creation of menu items *in-line*; that is, they are created automatically by the MENU package during the creation of the menu object. Other methods for creating menu items in-line include using the attributes, MENU_ITEM and MENU_ACTION_ITEM.*

MENU_ACTION_ITEM is used as a shortcut for separately specifying a label and a callback routine:

```
menu = (Menu)xv_create(NULL, MENU,
    MENU_ACTION_ITEM,  "item1",  callback1,
    MENU_ACTION_ITEM,  "item2",  callback2,
    ...
    NULL);
```

Two other methods for creating menu items are to:

- Use a separate call to xv_create() with the MENUITEM package.
- Provide a menu-generation routine.

The following subsections discuss the use of MENU_ITEM to create menu items in-line, MENUITEM to create separate menu items, and MENU_GEN_PROC to specify a routine that creates menus.

11.6.1 Using MENU_ITEM

Using the attribute MENU_ITEM indicates that a new menu item is to be created and appended to the existing menu. Use of this attribute means that the menu item is created in-line so a separate call to xv_create() is not necessary. The form of the value portion of the attribute is a NULL-terminated list of menu-item, attribute-value pairs:

```
extern void on_notify_proc(), off_notify_proc();
menu = (Menu)xv_create(NULL, MENU,
    MENU_TITLE_ITEM,      "Scrollbar",
    MENU_ITEM,
        MENU_STRING,      "On",
        MENU_NOTIFY_PROC, on_notify_proc,
        NULL,
    MENU_ITEM,
        MENU_STRING,      "Off",
        MENU_NOTIFY_PROC, off_notify_proc,
        NULL,
```

*The attributes MENU_IMAGES and MENU_ACTION_IMAGE are just like MENU_STRINGS and MENU_ACTION_ITEM except that Server_images are used as labels rather than text.

```
        NULL);
```

The code fragment shown above creates a menu with the same two menu items as shown in the previous example, except that the menu items are created more directly by the use of MENU_ITEM. Here we can specify item-specific attributes rather than accept all the defaults for the menu. In this case, we set a different notification routine for each menu item.*

If you have a Server_image to display rather than a string, you can replace MENU_STRING above with MENU_IMAGE and specify a Server_image rather than a string.

11.6.2 Using MENU_ACTION_ITEM

Rather than specifying menu item creation using a separate attribute-value list, the attribute MENU_ACTION_ITEM can be used as a shortcut, as shown in the example below:

```
menu = (Menu)xv_create(NULL, MENU,
    MENU_TITLE_ITEM,      "Scrollbar",
    MENU_ACTION_ITEM,     "On",    on_notify_proc,
    MENU_ACTION_ITEM,     "Off",   off_notify_proc,
    NULL);
```

The attribute MENU_ACTION_IMAGE, with a Server_image as its value, may be used interchangeably with MENU_ACTION_ITEM and its string value.

11.6.3 Using MENUITEM

The MENUITEM package allows you to create separate menu items using separate calls to xv_create(). The attributes used are menu item-specific attributes—the same as those that are used in the MENU_ITEM attribute above.

```
Menu_item   on, off;

on = (Menu_item)xv_create(NULL, MENUITEM,
    MENU_STRING,        "On",
    MENU_NOTIFY_PROC,   on_notify_proc,
    NULL);

off = (Menu_item)xv_create(NULL, MENUITEM,
    MENU_STRING,        "Off",
    MENU_NOTIFY_PROC,   off_notify_proc,
    NULL);

xv_set(menu,
    MENU_APPEND_ITEM,   on,
    MENU_APPEND_ITEM,   off,
    NULL);
```

These menu items are not created *in-line*; they are created independently using separate calls to xv_create(). They must therefore be added to the menu independently. In this case,

*Notification (callback) routines are discussed in Section 11.13, "Notification Procedures."

they are added using MENU_APPEND_ITEM (see the next section for more information).

11.7 Adding Menu Items

There are several methods for adding separately created menu items to menus. For a list of menu item attributes, see the package summary at the end of this chapter. These attributes can be used when you are using xv_set() on a menu.

The code fragment below demonstrates the use of MENU_APPEND_ITEM. The menu items are created independently of the menu itself. They are added to the menu as they are created by using the attribute MENU_APPEND_ITEM:

```
char         *names[ ] = { "One", "Two", "Three", "Four", "Five" };
Menu         menu;
Menu_item    mi;
int          i;
void         my_notify_proc();

menu = (Menu)xv_create(NULL, MENU, NULL);

for (i = 0; i < 5; i++) {
    mi = (Menu_item)xv_create(NULL, MENUITEM,
        MENU_STRING,        names[i],
        MENU_NOTIFY_PROC,   my_notify_proc,
        MENU_RELEASE,
        NULL);
    xv_set(menu, MENU_APPEND_ITEM, mi, NULL);
}
```

This use of MENU_RELEASE is to indicate that the menu item is intended to be freed when the item's parent menu is destroyed. This attribute takes *no value*; specifying it is equivalent to specifying a TRUE value. Not specifying it implies FALSE. In-line menu items have this attribute set by default, but menu items that are not created in-line must set this attribute explicitly if you want them to be freed automatically. You do not want to set this attribute if you intend to use this menu item in more than one menu or if you want to reuse it later. See Section 11.17, "Destroying Menus."

11.8 Pullright Menus

A pullright menu is simply another menu that is attached to a menu item. Note that for a menu item to contain a pullright menu, the pullright menu must already have been created. This means that any menu group should be created from the bottom up. The attributes MENU_PULLRIGHT, MENU_PULLRIGHT_ITEM, and MENU_PULLRIGHT_IMAGE all allow a pullright menu to be attached to a menu item.

In the first case, MENU_PULLRIGHT can be assigned to a menu item to attach an existing menu to it, as shown below:

```
extern Server_image image1, image2, image3;
```

```
Menu image_menu, menu;
void image_notify_proc();

image_menu = (Menu)xv_create(NULL, MENU,
    MENU_IMAGES,        image1, image2, image3, NULL,
    MENU_NOTIFY_PROC,  image_notify_proc,
    NULL);

menu = (Menu)xv_create(NULL, MENU,
    MENU_ITEM,
        MENU_STRING,      "images",
        MENU_PULLRIGHT,  image_menu,
        NULL,
    NULL);
```

In the previous example, a menu of server images is created and is initialized to contain three images. Another menu is created that is initialized for one menu item, but that menu item has a pullright menu that is set to the `menu_images` menu.

The menu item created may also be initialized using the `MENU_PULLRIGHT_ITEM` attribute. This attribute takes two parameters as its *value*: a string and a menu. Therefore, the above code fragment could have been written:

```
menu = (Menu)xv_create(NULL, MENU,
    MENU_PULLRIGHT_ITEM, "images", image_menu,
    NULL);
```

Had the label for the pullright menu item been a `Server_image` rather than a string, the call would look like:

```
extern Server_image label_image;

menu = (Menu)xv_create(NULL, MENU,
    MENU_PULLRIGHT_IMAGE, label_image, image_menu,
    NULL);
```

In the code fragment below, we use another piece of code to demonstrate the same principle. This example demonstrates how a menu that represents font sizes may be set as the pullright menu for a list of fonts:

```
Menu       font_menu, size_menu;
Menu_item  mi;
int        i;
void       notify_font();
char       buf[4], *p;
...
size_menu = (Menu)xv_create(NULL, MENU,
    MENU_NOTIFY_PROC,   notify_size,
    NULL);
for (i = 8; i <= 20; i += 2) {
    sprintf(buf, "%d", i);
    p = strcpy(malloc(strlen(buf)+1), buf);
    mi = (Menu_item)xv_create(NULL, MENUITEM,
        MENU_STRING,      p,
        MENU_RELEASE,
        MENU_RELEASE_IMAGE,
        MENU_NOTIFY_PROC, notify_font,
        NULL);
    xv_set(size_menu, MENU_APPEND_ITEM, mi, NULL);
}

font_menu = (Menu)xv_create(NULL, MENU,
    MENU_TITLE_ITEM,     "Fonts",
    MENU_PULLRIGHT_ITEM, "courier",        size_menu,
    MENU_PULLRIGHT_ITEM, "boston",         size_menu,
    MENU_PULLRIGHT_ITEM, "times-roman",    size_menu,
    MENU_PULLRIGHT_ITEM, "lucidasans",     size_menu,
    MENU_PULLRIGHT_ITEM, "palatino-roman", size_menu,
    NULL);
```

Each item in the main menu (font_menu) has a pullright menu (size_menu) associated with it. In the for loop where the string for the menu item is assigned, the data is allocated using malloc() and buf is copied into the allocated data using strcpy(). We cannot use buf directly, because unlike panel items, the menu item string is not copied by the MENU package—we must pass in allocated data. Because of this, we also specify the attribute MENU_RELEASE_IMAGE so that when the item is destroyed, the allocated data will be freed. Also note that because we used xv_create() to create the menu item, we specify MENU_RELEASE to indicate that the menu item should be freed when the parent menu is destroyed.

11.9 Menu-generating Procedures

In certain situations, the menu items for a particular menu cannot be known ahead of time. For example, a mail application allows users to write mail messages to files in a designated *folder directory*. If a menu is going to display the current folders in that directory, then the menu items should be updated any time a folder is created or deleted from that directory. But rather than updating the folder at the time the directory contents change, it would be better to scan the directory and use each filename in the directory as a menu item.

For such situations, it is necessary to defer the creation of the folder menu until it needs to be displayed. Therefore, you still create the pullright menu item so the user can select the item. But rather than specifying a pullright menu associated with the item, specify a routine that will generate the menu. When the menu needs to be displayed, the routine is called which returns a menu.

To do this, you specify the attribute MENU_GEN_PULLRIGHT when creating the menu item, as shown below:

```
    Menu menu, gen_folder_menu();
    void change_to_folder();
```

```
menu = (Menu)xv_create(NULL, MENU,
    MENU_TITLE_ITEM,        "Mail Folders",
    MENU_NOTIFY_PROC,       change_to_folder,
    MENU_STRINGS,           "/usr/spool/mail", "~/mbox", NULL,
    MENU_ITEM,
        MENU_STRING,        "~/Mail"
        MENU_GEN_PULLRIGHT, gen_folder_menu,
        NULL,
    NULL);
```

There is a shortcut attribute that allows you to specify both the menu item's string and the MENU_GEN_PULLRIGHT procedure in the same call. The attribute is MENU_GEN_PULLRIGHT_ITEM. It is used as follows:

```
MENU_GEN_PULLRIGHT_ITEM, "~/Mail", gen_folder_menu,
```

The menu-generating routine may do whatever is necessary to build a new menu, but you should be careful that the routine does not take too much processing time, since the user is waiting with the MENU button pressed for the menu to be displayed. Also remember that a pointer grab is going on, so the routine should avoid any interaction with the user (such as error dialog boxes).

The form of the menu-generating procedure is:

```
Menu
menu_gen_proc(menu_item, op)
    Menu_item menu_item;
    Menu_generate op;
```

This routine may be called *each time* the menu is needed. If the menu only needs to be created once, you can return to the same menu each time you need the menu.

The op parameter is one of the following enumerated types:

```
typedef enum {
    MENU_DISPLAY,
    MENU_DISPLAY_DONE,
    MENU_NOTIFY,
    MENU_NOTIFY_DONE
} Menu_generate;
```

op indicates the condition in which your routine has been called. The MENU_DISPLAY value indicates that the menu is going to be displayed while MENU_DISPLAY_DONE indicates that the menu has been displayed and dismissed. If the user makes a selection in the menu, the routine is called with MENU_NOTIFY before the menu's callback routine is called, then again with MENU_NOTIFY_DONE after the routine is called. If the user makes no selection, the latter two cases are not called. If they are called (the user did make a selection), then the latter two cases are called *after* MENU_DISPLAY_DONE is called.

Because you create your menus, you would think that you should destroy them as well. However, because of the unpredictable sequence of actions taken by the user, there is no way to determine when to free the menu. Therefore, you should never destroy menus at all. If the menu-generating procedure is called multiple times for the same menu, just reconstruct the menu from the same menu handle that you have.

Furthermore, the menu-generating routine must always return the same menu that it passed to you. You cannot return other menus to display. If the new menu is going to contain a completely different set of menu items, you should destroy all the menu items before creating the new list. As it is the same with PANEL_LIST items, menu items are destroyed in *reverse* order.

The special case for this problem is: what if there is no menu to redisplay again? In this case, you are allowed to build a new menu and return a handle to it. The following code shows an example, testing to see if there already is a menu associated with a particular pullright menu item.

```
Menu
menu_gen_proc(menu_item, op)
Menu_item menu_item;
Menu_generate op;
{
    int i;
    Menu menu;

    ...
    switch (op) {
        ...
        case MENU_DISPLAY :
            if (menu = (Menu)xv_get(menu_item, MENU_PULLRIGHT)) {
                /* first destroy old menu items */
                for (i = (int)xv_get(menu, MENU_NITEMS); i > 0; i--) {
                    xv_set(menu, MENU_REMOVE, i, NULL);
                    xv_destroy(xv_get(menu, MENU_NTH_ITEM, i));
                }
            else
                menu = (Menu)xv_create(NULL, menu, NULL);
            /* now rebuild the menu items */
        ...
    }
    ...
}
```

In the above code fragment, we are removing the menu items sequentially in reverse order by using the MENU_REMOVE attribute. We start with the last item and move to item 1. The first item, remember, is the title item, if it exists. If you want to retain this item, stop at menu item 2.

The sample program *menu_dir2.c* in Appendix F, *Example Programs*, demonstrates how a menu-generating routine is used.

A debugging hint: If your menu-generating routine generates a run-time error, be careful when trying to debug the program under a debugger. The problem is that when you run the program in a debugger and the program generates a run-time error, the debugger will stop execution and wait for input. In the meantime, the server has a pointer grab so keyboard focus is directed to the menu's window which is not able to receive input.

At this point, there is no way to interact with any program on the console—you will have to go to another server, computer or terminal connected to your workstation and kill the debugger (this will terminate the program and release the pointer grab). You may think of more clever ways to handle this situation depending on your workstation configuration, but the point is that you should be aware of the extremely inconvenient side effects whenever you play with server grabs.

11.9.1 Parent Menus

Recall that the menu notification routines take two parameters: the menu that was popped up and the menu item that was selected. However, if the user chose an item from a long cascade of pullright menus, it may be necessary to determine the initial (root) menu of the cascade. To support this, the attribute MENU_PARENT is used to get the owner of a menu or menu item.

This attribute can only be used with `xv_get()`. When MENU_PARENT is used with a menu item, `xv_get()` returns the handle of the enclosing menu.

```
Menu menu;

menu = xv_get(item, MENU_PARENT);
```

On the other hand, if `xv_get()` is passed a menu, the menu item returned is the menu item from which the submenu was pulled-right.

```
Menu_item item;

item = xv_get(menu, MENU_PARENT);
```

If the item returned is NULL, the menu is the root menu.

MENU_PARENT is only valid while the menu is active. Since menus can be shared, saying that a menu's parent is the one who uses it as a MENU_PULLRIGHT is not valid, since many menus could have that one menu as a MENU_PULLRIGHT.

The following code fragment shows how the entire menu cascade is traversed, starting from the leaf of the menu tree (the item the user selected).

```
Menu menu, item;

while (item = (Xv_opaque)xv_get(menu, MENU_PARENT))
    if ((Xv_pkg *)xv_get(item, XV_TYPE) != MENUITEM)
        break;
    else
        menu = xv_get(item, MENU_PARENT);
```

The above loop starts by getting the parent of an arbitrary menu. This menu could be the `menu` parameter in a menu item's callback routine. If the parent menu returned is NULL, then the menu is already the top level menu. Otherwise, get the type of the object returned. If the menu is a pullright menu, then the parent of the menu should be a MENUITEM (since its pullright is a menu). If it is not, then it could be a server object. Whatever it is, we have reached the top level of the menu cascade and should break out of the loop.

11.9.2 Using MENU_GEN_PROC

MENU_GEN_PROC specifies a function that is used to modify, add, or delete menu items from the menu whose handle is passed to the procedure. The *op* argument tells the state of the menu when the function is called. The argument *op* is one of the values: MENU_DISPLAY, MENU_DISPLAY_DONE, MENU_NOTIFY, or MENU_NOTIFY_DONE as defined by **Menu_generate** in *openmenu.h*. You do not destroy the menu itself. If you do not know what the item will show as text or as an image at the time the menu is created or if there is other unknown information, you can defer the creation of the menu item until the item is actually needed by specifying the item creation routine.

11.10 Using Toggle Menus

Toggle menus are menus with nonexclusive settings. The user can toggle menu items, turning them *on* or *off*. More than one menu item may be selected at a time. The only difference for creating these menu items is that MENU_TOGGLE_MENU is used as the *package* parameter to xv_create() and that menu items may not have pullright menus associated with them. Therefore, these are typically simple menus.

In the code below, we build a toggle menu that has three items in it. If the menu has been displayed and the user makes a selection, on or off, the notification routine is called no differently from any other menu notification procedure:

```
void toggle_bold(), toggle_size(), toggle_italic();
Menu menu;

menu = (Menu)xv_create(NULL, MENU_TOGGLE_MENU,
    MENU_TITLE_ITEM,    "Text Rendering",
    MENU_ACTION_ITEM,   "Bold Style",     toggle_bold,
    MENU_ACTION_ITEM,   "Large Font",     toggle_size,
    MENU_ACTION_ITEM,   "Italics",        toggle_italic,
    NULL);
```

In this case, we are specifying three different notify procedures for each menu item. Since each performs a completely separate function, the menu items need not call the same routine.

To determine exactly which menu items are selected, you must loop through all the items in the menu:

```
toggle_notify(menu, item)
Menu menu
Menu_item item;
{
    int i;

    for (i = (int)xv_get(font_menu, MENU_NITEMS); i > 0; i--)
        if (xv_get(xv_get(font_menu, MENU_NTH_ITEM, i),
                    MENU_SELECTED)) {
            printf("item %d selected\n", i);
            /* do whatever other processing may need to be done */
        }
}
```

This loop starts at the last item and works towards the first. The first item starts at 1, not 0; the *0th* item is the menu's title item and cannot be retrieved.

11.11 Menu Layout

By default, pop-up menus place their items vertically. If there are too many items, a new column may be started in order to display the entire menu on the screen. You can specify the number of rows and columns for the menu by using the attributes `MENU_NROWS` and `MENU_NCOLS`.

Although specifying menu item layout is certainly legal and acceptable to OPEN LOOK, explicit menu item layout should be avoided for anything other than static menus. Dynamic menus will have problems maintaining menu item order, and if you use a pin-up menu, the command frame will almost certainly not match the appearance of the menu. To guarantee that your pin-up menu looks the same as the menu, specify your own pin-up procedure. (See the following section for more information.)

11.12 Making Pin-up Menus

As the programmer, you may give the user the option of pinning up a menu by providing the pushpin in the pop-up menu. To accomplish this, XView provides the attribute `MENU_GEN_PIN_WINDOW`. If specified, XView generates the pin window frame automatically by creating a command frame, a panel and a series of panel items that correspond to the menu items. These pin window components are actually created the first time the user pulls down the menu and pushes the pin in. You cannot use `xv_get` to retrieve the command frame for the menu until after the menu is pinned. This new frame is dynamic, so any changes to the menu are reflected in the pinup frame provided it is not currently being displayed. If the pinup frame is currently being displayed and the menu contents change, the pinned menu will not reflect the new changes. The changes will appear the next time the menu is pinned.

Since menu items are translated into panel items in a pinned menu, programmers should not allow more than 32 unique values for a pinned menu (32 is the size of an **unsigned int** on most machines).

MENU_GEN_PIN_WINDOW takes two values as parameters. One is a base frame; the other is a string that acts as a title for the frame. The menu's pin window is sized according to the width of the widest menu item, not according to the title. You should choose a title that will fit within the size of the pin window. If the title specified in MENU_GEN_PIN_WINDOW is too long, then it will be truncated. Example 11-2 shows how to create a menu with a pushpin.

Example 11-2. How to create a menu containing a pushpin

```
Frame  frame;
Menu   menu;
void   func1(), func2();
 ...
/* Create base frame for the application */
frame = (Frame)xv_create(NULL, FRAME, NULL);
 ...
menu = (Menu)xv_create(NULL, MENU,
    /* the pinup menu subframe is a child of the base frame */
    MENU_GEN_PIN_WINDOW,    frame,  "title",
    MENU_ITEM,
        MENU_STRING,       "item1",
        MENU_NOTIFY_PROC,  func1,
        NULL,
    MENU_ITEM,
        MENU_STRING,       "item2",
        MENU_NOTIFY_PROC,  func2,
        NULL,
    NULL);
 ...
```

The new command frame is created as a subframe of the `frame` value. The title label for the menu and the frame is the *title* value. Note that you should not use MENU_TITLE_ITEM if you are using MENU_GEN_PIN_WINDOW since a menu item is automatically inserted at the top of the menu. If this item is removed, the pin will also be removed.

When the menu is displayed as a result of a call to `menu_show()`, a pushpin in the upper-left corner of the menu is displayed allowing the user to pin up the menu, that causes the menu to go away and the subframe to be displayed. You can get a handle to this subframe if you need it by calling:

```
Frame subframe = (Frame)xv_get(menu, MENU_PIN_WINDOW);
```

You can get a handle to the panel associated with that frame by calling:

```
Panel panel = (Panel)xv_get(frame, FRAME_CMD_PANEL);
```

If you choose to write your own pin window-generating procedures, there are several attributes that you might find helpful in implementing your routines:

MENU_PIN	This Boolean attribute indicates that the menu has a pushpin. If an application removes the first menu item, which is the title for the pinned menu, then the menu's pin will also be removed.
MENU_PIN_WINDOW	This attribute assigns the command frame you created to the menu's pin window. Once the pushpin is pushed in, the MENU package automatically sets the command frame's XV_SHOW attribute to TRUE, allowing the frame to be displayed.

MENU_PIN_PROC	This attribute provides the menu with a procedure that is called when the user pushes the pin in. You may override the default procedure that shows the pin_window of the menu by providing your own routine using this attribute. This routine overrides the default behavior of setting XV_SHOW on the command frame, so this responsibility lies in this routine.
MENU_DONE_PROC	This routine is called whenever a pop-up menu has been taken down after a menu item has been selected, pinned up or simply dismissed without a selection being made. Again, this overrides the default action of setting XV_SHOW to FALSE, so this responsibility lies with the MENU_DONE_PROC routine.

11.13 Notification Procedures

When a menu item is selected, a notification procedure is called to notify the host application that the user has made a selection. If the menu item does not have a notify procedure, the parent menu's notification procedure is called instead. If the menu does not have a notification procedure, no action is taken. Be sure that each menu item or its parent menu has a MENU_NOTIFY_PROC routine associated with it. Otherwise, choosing a menu item has no effect. If you wish to make a menu item *inactive*, you should set the attribute MENU_INACTIVE to TRUE.

The primary difference between the MENU package and the PANEL package with respect to the notification mechanism is that if a menu item has no notification procedure associated with it, the notify procedure of the parent menu is used. The PANEL package has no such feature. To differentiate between the notify procedure of a *menu* and the notify procedure of a *menu item*, the term *action procedure* is sometimes used to refer to the menu item's notify procedure. Thus, you may come across the term *action procedure* or the attribute MENU_ACTION_PROC outside of this manual.

However, because there is functionally no difference between the MENU_ACTION_PROC and the MENU_NOTIFY_PROC and since all other XView objects use notify procedures to register their callbacks, we are going to attempt to maintain consistency and avoid potential confusion with the terms by referring to both notify procedures commonly using MENU_NOTIFY_PROC.

The form of the callback routine is:

```
void
menu_notify_proc(menu, menu_item)
    Menu menu;
    Menu_item menu_item;
```

The program in Example 11-3 demonstrates several of the concepts introduced in the chapter so far. *xv_menu.c* creates a simple frame, a canvas, and a pop-up menu. The menu is a *static* menu because all the menu items are created in-line with the menu-generating procedure. The menu itself has a notify procedure which is called if any of the menu items specified by MENU_STRINGS are selected. The result is to display the text of the selected item in the header of the frame. An additional menu item is specified that has a pullright menu that can be pinned up.

Example 11-3. The xv_menu.c program

```
/*
 * xv_menu.c -
 *      Demonstrate the use of an XView menu in a canvas subwindow.
 *      Menu is brought up with right mouse button and the selected
 *      choice is displayed in the canvas.  Allows menu to be pinned.
 */
#include <xview/xview.h>
#include <xview/canvas.h>

Frame   frame;

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Canvas      canvas;
    Menu        menu;
    void        my_notify_proc(), my_event_proc();
    extern void exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,    argv[0],
        NULL);
    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,       300,
        XV_HEIGHT,      200,
        NULL);
    menu = (Menu)xv_create(NULL, MENU,
        MENU_TITLE_ITEM,        "Junk",
        MENU_STRINGS,           "Yes", "No", "Maybe", NULL,
        MENU_NOTIFY_PROC,       my_notify_proc,
        MENU_ITEM,
            MENU_STRING,        "Save",
            MENU_NOTIFY_PROC,   my_notify_proc,
            MENU_PULLRIGHT,
                xv_create(canvas, MENU,
                    MENU_GEN_PIN_WINDOW,        frame, "Save",
                    MENU_ITEM,
                        MENU_STRING,            "Update Changes",
                        MENU_NOTIFY_PROC,       my_notify_proc,
                        NULL,
                    NULL),
            NULL,
        MENU_ITEM,
            MENU_STRING,        "Quit",
            MENU_NOTIFY_PROC,   exit,
            NULL,
        NULL);

    xv_set(canvas_paint_window(canvas),
```

Example 11-3. The xv_menu.c program (continued)

```
        WIN_CONSUME_EVENTS,     WIN_MOUSE_BUTTONS, NULL,
        WIN_EVENT_PROC,         my_event_proc,
        /* associate the menu to the canvas win for easy retrieval */
        WIN_CLIENT_DATA,        menu,
        NULL);

    window_fit(frame);
    window_main_loop(frame);
}

/*
 * my_notify_proc - Display menu selection in frame header.
 */
void
my_notify_proc(menu, menu_item)
Menu menu;
Menu_item menu_item;
{
    xv_set(frame,
        FRAME_LABEL,    xv_get(menu_item, MENU_STRING),
        NULL);
}

/*
 * Call menu_show() to display menu on right mouse button push.
 */
void
my_event_proc(window, event)
Xv_Window window;
Event *event;
{
    if (event_action(event) == ACTION_MENU && event_is_down(event)) {
        Menu menu = (Menu)xv_get(window, WIN_CLIENT_DATA);
        menu_show(menu, window, event, NULL);
    }
}
```

The output of Example 11-3 is shown in Figure 11-8.

11.14 Finding Menu Items

You can use `xv_find()` to locate menu items that match certain attribute-value pairs. The form of using `xv_find()` for menus is:

```
Menu_item item;

item = (Menu_item)xv_find(menu, MENUITEM,
    <attribute-value list>,
    NULL);
```

`xv_find()` returns menu items, not menus. By default, when you search for items, each item in a menu is searched before descending into a menu item's pullright menu, should it

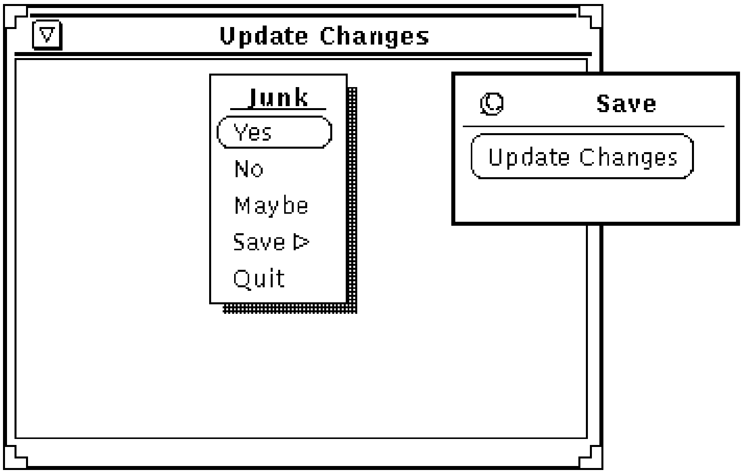


Figure 11-8. Output of xv_menu.c

exist. However, you can override this default behavior by specifying the attribute MENU_DESCEND_FIRST. During a menu search, if an item with a pullright is found, then this attribute indicates whether the search should continue through the pullright or to the next item in the current menu.

If a menu item or a menu item's pullright is a *generate* procedure, the generate procedure is called despite the fact that the menu or menu item will not be displayed. No matter how many attributes are given, xv_find() will return the first item found that matches all given attributes even though the item may have more attributes associated with it.

When specifying attribute-value pairs, you specify attributes in the same way as when you use xv_create(). For example, if you want to find a menu item with the string value of "fonts" and the callback routine of my_notify_proc, you would use:

```
menu_item = (Menu_item)xv_find(menu, MENUITEM,
    MENU_STRING,        "fonts",
    MENU_NOTIFY_PROC,  my_notify_proc,
    NULL);
```

Unless the attribute XV_AUTO_CREATE is set to FALSE, if xv_find() does not find the menu item that you are looking for, a new menu item will be created.

11.15 Initial and Default Menu Selections

Two special menu items are the *default item* (`MENU_DEFAULT_ITEM`) and the *selected item* (`MENU_SELECTED_ITEM`). The default item defaults to the *first* item in the menu, and the *selected item* is the selected item (or item**s** for `MENU_TOGGLE_MENU` menus).

Although the default menu item may be set by using `xv_set()`, the user may interactively change the default menu item by holding down the CONTROL key while also selecting a menu item with the MENU button. Therefore, if the user selects a menu item that has a pullright menu, *but the pullright menu is not activated*,* when your notify procedure is called, you may choose to descend into the pullright menu and find the default menu item and call that item's callback routine.

11.16 Unpinned Command Frame Dismissal

XView normally handles unpinned command frame dismissal for you when a user action within the command frame completes successfully. If a menu is brought up from a Menu Button in the command frame, and the user makes a selection from the menu, the command frame is dismissed if the pushpin (if visible) is out. By default, the attribute `MENU_NOTIFY_STATUS` is set to `XV_OK`, which indicates that the command frame should be dismissed if the pushpin is out, and the callback returns successfully. However, if the user-specified action does not complete successfully, you may not want the command frame to be dismissed. In this case, within the menu's notify procedure or within the menu item's notify procedure, you should set the value of `MENU_NOTIFY_STATUS` to `XV_ERROR`. This indicates that the user selection was invalid or failed, and prevents the command frame from being dismissed.

11.17 Destroying Menus

Destruction of menus is an important task because menus are frequently used and, if their resources are not freed adequately, you could find the size of your application growing rapidly until your system runs out of available memory. Therefore, proper cleanup of menu destruction is imperative. Menus are destroyed using `xv_destroy()`. In the case of *static menus*, nothing more is required than calling `xv_destroy()`. This is because the internals of XView automatically set attributes discussed in this section.

Be aware of several situations, such as when you:

- Allocate your own strings or server images as menu item labels.
- Create your own menu items using `xv_create(NULL, MENUITEM, ...)`.
- Generate your own pullright menus.

*This might happen if the user did not drag the mouse far enough to the right.

The destruction phase walks down each menu item in the menu and tests each menu item to see if it has the MENU_RELEASE attribute set. This is not a Boolean attribute—it has no value associated with it at all. If you specify the attribute, the attribute is set. If you do not specify it, then the attribute is not set. As noted, menu items that have been created in-line have MENU_RELEASE set already.

Menu items that you create yourself do not have MENU_RELEASE set by default. You also may or may not *want* it set. If you plan to reuse menu items—a need that is common—then you do not want to set this attribute. However, you must maintain a handle to the menu item or it is lost. If the attribute is set, then the menu item is freed, but no other data associated with the menu item is destroyed. Only the item itself is. If you have any allocated data associated with the menu item, then you either need to free it yourself or give a hint to XView to free it for you.

The following subsections discuss other data allocated for menu items. Remember that freeing menus and menu items is not done automatically; this only happens as a result of your calling xv_destroy(). So, if you decide to free menus or menu items, you should be sure to free pullright menus and/or client data yourself beforehand.

There are cases when xv_destroy() will not remove a menu. In order to free the memory associated with a menu using xv_destroy(), you need to be certain that no objects reference the menu. For example, if you attach a menu to a panel button item using the attribute PANEL_ITEM_MENU, you need to be sure to clear the PANEL_ITEM_MENU attribute *before* you try to destroy the menu. In this example, the following calls would be required to clear the panel button item's attached menu, and to destroy the menu. For more information on this topic, refer to the description of XV_REF_COUNT in Chapter 7, *Panels*.

```
xv_set(panel_item, PANEL_ITEM_MENU, NULL, NULL);
xv_destroy(menu);
```

11.17.1 Freeing Allocated Strings

If you create a menu item with allocated data, you should *not* use them in a MENU_STRINGS list. Instead, you should create the menu items individually, as shown in Example 11-4.

Example 11-4. Creating individual menu items

```
char *str1;

if (str1 = malloc(strlen(buf)+1))
    strcpy(str1, buf);

menu = xv_create(NULL, MENU,
    MENU_ITEM,
        MENU_STRING,   str1,
        MENU_RELEASE_IMAGE,
        NULL,
    NULL);
```

The code in Example 11-4 shows a menu item that is created in-line because it is created using the MENU_ITEM attribute. However, because the string used as the menu item's label is allocated, we need to provide XView with a hint to release this data.

Similarly, if we used `xv_create()` to create a `Server_image` as the menu item's label, the `MENU_RELEASE_IMAGE` attribute suffices to free that data as well.

11.17.2 Freeing Pullright Menus

Even though a menu item has `MENU_RELEASE` set, if a pullright menu is associated with it, the menu will not be freed. In many cases, this is acceptable because many menu items may share the same pullright menu. If you are sure you do not need the menu anymore, then you should free it. Note that freeing the menu will attempt to free the menu items within it.

This is most commonly done in menu-generating routines installed as the `MENU_GEN_PULLRIGHT` attribute.

11.17.3 Menu Client Data

If a menu item is freed, you should be sure to free any client data that is associated with it. Client data may have been attached to the menu item using `XV_KEY_DATA` or `MENU_CLIENT_DATA`.

If you created menus for panel buttons, and you destroy the MENU button (or the panel associated with that button), then you are responsible for destroying the menu you created. The panel does not handle this for you. Destroying the menu attached to menu buttons is done the same way as it is for menus.

11.18 Example Program

The following brief descriptions are introductory notes about the programs *menu_dir.c* (listed in Example 11-5) and *menu_dir2.c* (listed in Appendix F, *Example Programs*). The comments in the programs as well as the code itself should be read for full details.

menu_dir.c demonstrates many of the features of the `MENU` package presented in this chapter. It displays a menu that contains all the files from the current directory. If a pathname is given on the command line, that directory is used. The entire menu hierarchy is built initially at start-up time, so directories that do not have extremely long paths should be specified.*

For each directory found, a new menu is created and the directory is descended building items for the new menu. *menu_dir2.c* also builds cascading menus for directories, but instead of descending into the directory tree, a menu-generating routine is called only if the user tries to go into a pullright.

Example 11-5. The menu_dir.c program

```
/*
```

*Don't even think of specifying /.

Example 11-5. The menu_dir.c program (continued)

```
 * menu_dir.c -
 * Demonstrate the use of an XView menu in a canvas subwindow.
 * A menu is brought up with the MENU mouse button and displays
 * menu choices representing the files in the directory.  If a
 * directory entry is found, a new pullright item is created with
 * that subdir as the pullright menu's contents.  This implementation
 * creates the entire directory tree initially.  Do not attempt to
 * build a tree from /.  You will most likely run out of resources.
 *
 * argv[1] indicates which directory to start from.
 */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <sys/stat.h>
#include <sys/dir.h>
#include <X11/Xos.h>
#ifndef MAXPATHLEN
#include <sys/param.h> /* probably sun/BSD specific */
#endif /* MAXPATHLEN */

Frame   frame;

/*
 * main -
 *      Create a frame, canvas and menu.
 *      A canvas receives input in its canvas_paint_window().
 *      Its callback procedure calls menu_show().
 */
main(argc,argv)
int     argc;
char    *argv[];
{
    Canvas      canvas;
    extern void exit();
    void        my_event_proc();
    Menu        menu;
    Menu_item   mi, add_path_to_menu();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,            argv[1]? argv[1] : "cwd",
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);
    canvas = (Canvas)xv_create(frame, CANVAS,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       400,
        XV_HEIGHT,      100,
        NULL);

    mi = add_path_to_menu(argc > 1? argv[1] : ".");
    menu = (Menu)xv_get(mi, MENU_PULLRIGHT);

    /* associate the menu to the canvas win for easy retrieval */
    xv_set(canvas_paint_window(canvas),
        WIN_CONSUME_EVENTS,     WIN_MOUSE_BUTTONS, NULL,
```

Example 11-5. The menu_dir.c program (continued)

```
        WIN_EVENT_PROC,         my_event_proc,
        WIN_CLIENT_DATA,        menu,
        NULL);

    window_fit(frame);
    window_main_loop(frame);
}

/*
 * my_action_proc - display the selected item in the frame footer.
 */
void
my_action_proc(menu, menu_item)
Menu    menu;
Menu_item       menu_item;
{
    xv_set(frame,
        FRAME_LEFT_FOOTER,      xv_get(menu_item, MENU_STRING),
        NULL);
}

/*
 * Call menu_show() to display menu on right mouse button push.
 */
void
my_event_proc(canvas, event)
Canvas  canvas;
Event *event;
{
    if ((event_id(event) == MS_RIGHT) && event_is_down(event)) {
        Menu menu = (Menu)xv_get(canvas, WIN_CLIENT_DATA);
        menu_show(menu, canvas, event, NULL);
    }
}

/*
 * return an allocated char * that points to the last item in a path.
 */
char *
getfilename(path)
char *path;
{
    char *p;

    if (p = rindex(path, '/'))
        p++;
    else
        p = path;
    return strcpy(malloc(strlen(p)+1), p);
}

/*
 * The path passed in is scanned via readdir().  For each file in the
 * path, a menu item is created and inserted into a new menu.  That
 * new menu is made the PULLRIGHT_MENU of a newly created panel item
 * for the path item originally passed it.  Since this routine is
```

Example 11-5. The menu_dir.c program (continued)

```
 * recursive, a new menu is created for each subdirectory under the
 * original path.
 */
Menu_item
add_path_to_menu(path)
char *path;
{
    DIR                 *dirp;
    struct direct       *dp;
    struct stat         s_buf;
    Menu_item           mi;
    Menu                next_menu;
    char                buf[MAXPATHLEN];

    /* don't add a folder to the list if user can't read it */
    if (stat(path, &s_buf) == -1 || !(s_buf.st_mode & S_IREAD))
        return NULL;
    if (s_buf.st_mode & S_IFDIR) {
        int cnt = 0;
        if (!(dirp = opendir(path)))
            /* don't bother adding to list if we can't scan it */
            return NULL;
        next_menu = (Menu)xv_create(XV_NULL, MENU, NULL);
        while (dp = readdir(dirp))
            if (strcmp(dp->d_name, ".") && strcmp(dp->d_name, "..")) {
                (void) sprintf(buf, "%s/%s", path, dp->d_name);
                if (!(mi = add_path_to_menu(buf)))
                    /* unreadable file or dir - deactivate item */
                    mi = xv_create(XV_NULL, MENUITEM,
                        MENU_STRING,         getfilename(dp->d_name),
                        MENU_RELEASE,
                        MENU_RELEASE_IMAGE,
                        MENU_INACTIVE,       TRUE,
                        NULL);
                xv_set(next_menu, MENU_APPEND_ITEM, mi, NULL);
                cnt++;
            }
        closedir(dirp);
        mi = xv_create(XV_NULL, MENUITEM,
            MENU_STRING,        getfilename(path),
            MENU_RELEASE,
            MENU_RELEASE_IMAGE,
            MENU_NOTIFY_PROC,   my_action_proc,
            NULL);
        if (!cnt) {
            xv_destroy(next_menu);
            /* An empty or unsearchable directory - deactivate item */
            xv_set(mi, MENU_INACTIVE, TRUE, NULL);
        } else {
            xv_set(next_menu, MENU_TITLE_ITEM, getfilename(path), NULL);
            xv_set(mi, MENU_PULLRIGHT, next_menu, NULL);
        }
        return mi;
    }
    return (Menu_item)xv_create(NULL, MENUITEM,
        MENU_STRING,            getfilename(path),
```

Example 11-5. The menu_dir.c program (continued)

```
        MENU_RELEASE,
        MENU_RELEASE_IMAGE,
        MENU_NOTIFY_PROC,       my_action_proc,
        NULL);
}
```

11.19 Menu Package Summary

Table 11-1 lists the procedures and macros in the MENU package. Table 11-2 lists the attributes in the MENU package. This information is described fully in the *XView Reference Manual*.

Table 11-1. Menu Procedures and Macros

MENUITEM_SPACE()
menu_return_item()
menu_return_value()
menu_show

Table 11-2. Menu Attributes

MENU_APPEND_ITEM	MENU_NOTIFY_STATUS
MENU_CLASS	MENU_NROWS
MENU_COLOR	MENU_NTH_ITEM
MENU_CLIENT_DATA	MENU_PARENT
MENU_COL_MAJOR	MENU_PIN
MENU_DEFAULT	MENU_PIN_PROC
MENU_DEFAULT_ITEM	MENU_PIN_WINDOW
MENU_DESCEND_FIRST	MENU_PULLRIGHT
MENU_DONE_PROC	MENU_RELEASE
MENU_FIRST_EVENT	MENU_RELEASE_IMAGE
MENU_GEN_PIN_WINDOW	MENU_REMOVE
MENU_GEN_PROC	MENU_REMOVE_ITEM
MENU_GEN_PULLRIGHT	MENU_REPLACE
MENU_IMAGE	MENU_REPLACE_ITEM
MENU_IMAGES	MENU_SELECTED
MENU_INACTIVE	MENU_SELECTED_ITEM
MENU_INSERT	MENU_STRING
MENU_INSERT_ITEM	MENU_STRINGS
MENU_ITEM	MENU_TITLE
MENU_LAST_EVENT	MENU_TITLE_ITEM

Table 11-2. Menu Attributes (continued)

MENU_NCOLS	MENU_TYPE
MENU_NITEMS	MENU_VALID_RESULT
MENU_NOTIFY_PROC	MENU_VALUE
XV_DEPTH	XV_VISUAL
XV_VISUAL_CLASS	

This page intentionally left blank

to preserve original page counts.

12
Notices

A notice is a pop-up window that notifies the user of a problem or asks a question that requires a response. Generally, notices report serious warnings or errors. OPEN LOOK notices do not have headers or footers and cannot be moved. The XView notice object is subclassed from the XView generic object. As with any XView object, you can configure a notice using attributes and you can use `xv_create()`, `xv_get()`, and `xv_set()`. Figure 12-1 shows the notice object class hierarchy.

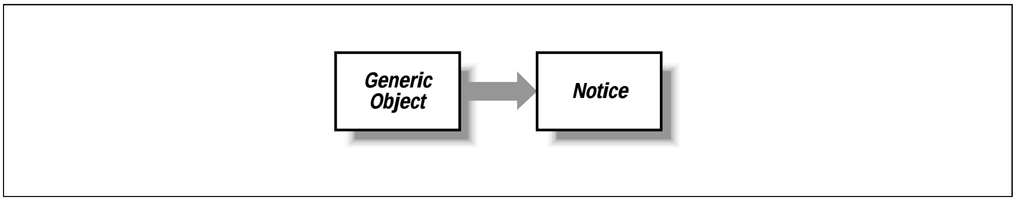


Figure 12-1. Notice class hierarchy

XView defines two types of notices, *Standard notices* and *screen-locking* notices:

- *Standard notices* do not lock the screen and are placed centered in the "owner" frame. This type of notice may either block the application's thread of execution, or not block.
- *Screen-locking notices* lock the screen and block input to all applications (the screen is locked with X grabs). These notices appear with a shadow that emanates from the location where an action in an application initiates the notice. This may be a panel button, such as "Quit," or some other XView object.

New applications that are created with XView Version 3 should use the `NOTICE` package described in this chapter. Older versions of XView only supported notices with a nonobject-oriented interface using the `notice_prompt()` function. For compatibility, `notice_prompt()` is still supported. However, for new applications its use is not recommended. Furthermore, the `NOTICE` package is implemented so that updating applications to use a notice object is an easy task. Notice objects are only created when the notice package is used (they are not created when `notice_prompt()` is used). For more information on `notice_prompt()`, refer to Appendix B, *Notices*.

Figure 12-2 shows an example of a notice window from the *OPEN LOOK GUI Specification*.

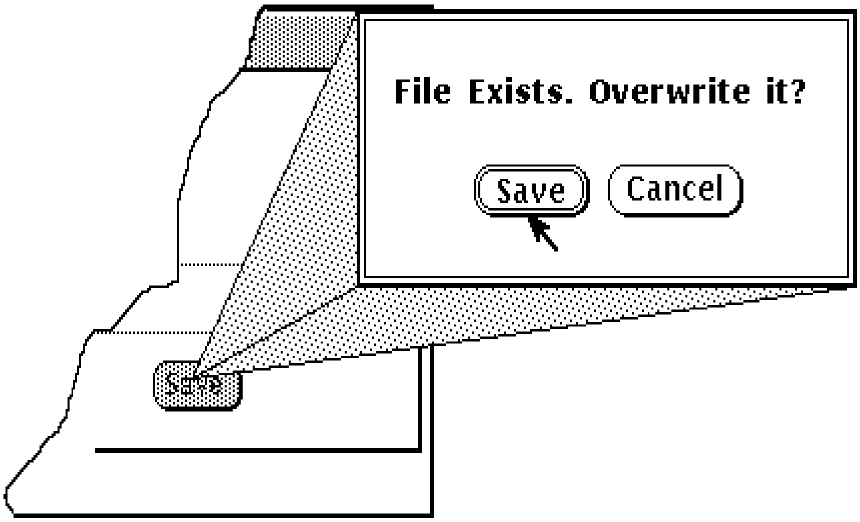


Figure 12-2. A sample notice window

12.1 Creating and Displaying Notices

To use the NOTICE package, include the header file <*xview/notice.h*>. It provides the necessary types and definitions for using the package. A notice object's type is Xv_Notice. In general, create a notice like any other XView object:

```
Xv_Notice   notice;
Frame       owner;
notice = xv_create(owner, NOTICE,
            NOTICE_MESSAGE_STRING,
                "Please confirm your action.",
                NULL,
            NULL);
```

Make a notice visible by setting XV_SHOW to TRUE:

```
xv_set(notice, XV_SHOW, TRUE, NULL);
```

Clicking on any button in a notice pops-down the notice. It is *not* necessary to set XV_SHOW to FALSE on a notice (the notice package handles this internally).

A notice must have an owner that is a subtype of WINDOW, such as a frame or a panel. If NULL is used for the owner, an error results and the notice is not created. Typically the window of the application that causes the notice to be created is the notice's owner. For example, if the user tries to type in a read-only text subwindow, a notice might appear from that window informing the user of the error.

Your application has control over the type of the notice (standard or screen-locking), the messages that are displayed in the notice window, and the choices available to the user as responses. The notice package creates the notice window, and depending on the type of the notice, either blocks input to the originating application and does *not* lock the screen while waiting for the user to make a selection on one of the available button choices, or, locks the screen and waits for one of the button choices to be selected (in this case, the screen is frozen and the application is blocked.) For both types of notices, after the user enters a response in a notice button, the notice window is unmapped.

The attributes `NOTICE_MESSAGE_STRING`, `NOTICE_MESSAGE_STRINGS`, and `NOTICE_MESSAGE_STRINGS_ARRAY_PTR` can be used to determine what strings are to be displayed in a notice. `NOTICE_MESSAGE_STRING` takes as its value one `NULL`-terminated string. This string, however, can contain the character “\n” to serve as a line break. The attribute `NOTICE_MESSAGE_STRINGS` can take a list of the above strings as its value. This list is `NULL`-terminated. Each string in the list will start on a new line. Lastly, `NOTICE_MESSAGE_STRINGS_ARRAY_PTR` takes a pointer to a `NULL`-terminated array of strings.

All the strings mentioned above are centered horizontally in the notice.

Below is an example using `NOTICE_MESSAGE_STRING`:

```
Xv_Notice   notice;
Frame       owner;
notice = xv_create(owner, NOTICE,
          NOTICE_MESSAGE_STRING,
            "Hello!\nPlease confirm your action.\nPress Continue",
          NULL);
```

The following example demonstrates `NOTICE_MESSAGE_STRINGS`:

```
Xv_Notice   notice;
Frame       owner;
notice = xv_create(owner, NOTICE,
            NOTICE_MESSAGE_STRINGS,
                 "Hello!",
                 "Please confirm your action.\nPress Continue",
            NULL,
            NULL);
```

`NOTICE_MESSAGE_STRINGS_ARRAY_PTR` takes an array of strings as in the following example:

```
char        array[5];

array[0]    = "Hello!";
array[1]    = "This is a sample notice.";
array[2]    = "Press Continue.";
array[3]    = NULL;

Xv_Notice   notice;
Frame       owner;

notice = xv_create(owner, NOTICE,
            NOTICE_MESSAGE_STRINGS_ARRAY_PTR,
                 array,
            NULL);
```

A sample standard notice is demonstrated in Example 12-1.

Example 12-1. The simple_notice.c program

```
/*
 * simple_notice.c -- Demonstrate the use of notices.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

Panel      panel;

main(argc,argv)
  int     argc;
  char    *argv[];
{
    Frame       frame;
    Xv_opaque   my_notify_proc();

    /*
     * Initialize XView, create a frame, a panel and one panel button.
     */
    xv_init(XV_INIT_ARGS, argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        NULL);

    /* make sure everything looks good */
    window_fit(panel);
    window_fit(frame);

    /* start window event processing */
    xv_main_loop(frame);
}

/*
 * my_notify_proc() -- called when the user selects the QUIT button.
 *      Here the user must choose YES or NO on the notice to confirm
 *      or deny quitting.
 */
Xv_opaque
my_notify_proc(item, event)
  Panel_item  item;
  Event      *event;
{
    Xv_notice   notice;
    int         notice_stat;

    notice = xv_create(panel, NOTICE,
        NOTICE_MESSAGE_STRINGS, "Do you really want to quit?", NULL,
        NOTICE_BUTTON_YES,      "Yes",
        NOTICE_BUTTON_NO,       "No",
        NOTICE_STATUS,          &notice_stat,
```

Example 12-1. The simple_notice.c program (continued)

```
        XV_SHOW, TRUE,
        NULL);

    switch (notice_stat)  {
        case NOTICE_YES:
     /* Quit */
            exit(0);
        break;

        case NOTICE_NO:
     /* Don't quit */
        break;

    }

    xv_destroy_safe(notice);
}
```

The program *simple_notice.c* contains a panel with a Quit button. When the user selects the Quit button, a notice pops up to prompt the user for confirmation. What the user sees is shown in Figure 12-3. If the user presses "Yes," the program exits.

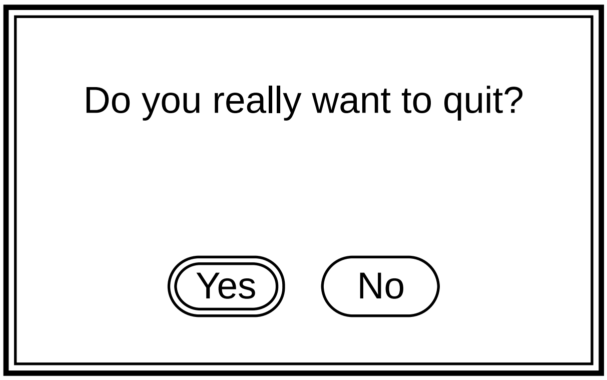

Do you really want to quit?

Figure 12-3. Output of simple_notice.c while the notice is up

12.1.1 Notice Values and Status

Two responses are normally available whenever a notice appears: "Yes" and "No." These are defined for convenience in <*xview/notice.h*>:

```
#define NOTICE_YES  1
#define NOTICE_NO   0
```

These values correspond to the notice's status. The notice status contains the value of the button that was selected on the notice. The location where the status is stored can be set with the attribute NOTICE_STATUS. If NOTICE_STATUS is not set, the notice status can still be obtained by using xv_get() on NOTICE_STATUS. If the attributes NOTICE_BUTTON_YES and NOTICE_BUTTON_NO are used, the notice status is either set to NOTICE_YES or NOTICE_NO, depending upon the user's notice button selection, which causes the notice to pop-down. As shown in *simple_notice.c*, the strings associated with the "Yes" and "No" buttons are set with the attributes NOTICE_BUTTON_YES and NOTICE_BUTTON_NO. The notice button created with NOTICE_BUTTON_YES is the default button and will have the OPEN LOOK default ring around it.

The notice choices listed above also respond to accelerator keys. In other words, in addition to the RETURN key, whatever key that is mapped to the semantic action ACTION_DEFAULT_ACTION can also be used to select the NOTICE_BUTTON_YES button. The key that is mapped to the semantic action ACTION_STOP can be used to select the NOTICE_BUTTON_NO button.

When the notice window is mapped, the cursor is immediately warped (moved) to the default button since it is the default response of the notice.

It is quite common for the application to have more than one appropriate response to some kind of notice prompt. Suppose that your application is an editor of some kind. If the user selects the Quit button and there have been changes to the file that have not been accounted for, you might wish to inform the user and allow more than one response: quit, updating changes; quit, ignoring changes; or cancel the quit all together. To implement more than two choices, use the NOTICE_BUTTON attribute to define the choices available:

```
xv_create(panel, NOTICE,
    NOTICE_MESSAGE_STRINGS,
        "There have been modifications since your last update",
        "Would you like to quit or continue editing?",
        NULL,
    NOTICE_BUTTON,    "Quit, Update changes",    101,
    NOTICE_BUTTON,    "Quit, Ignore changes",    102,
    NOTICE_BUTTON,    "Continue Editing",        103,
    NULL);
```

The NOTICE_BUTTON attribute takes two parameters: the button label* and the value for the button selection. The application should make its decision on how to proceed based on the button value specified. In this case, the notice would need to handle cases for NOTICE_STATUS having values of 101, 102, and 103 (in addition to possible errors).

Because the NOTICE_BUTTON attribute is used, there is no button that is bound to the default NOTICE_YES choice. But since OPEN LOOK requires a default button for every notice, the default button will be the first button. In this case, the "Quit, Update changes" button will have the default ring around it.

If no buttons are specified for a notice, a default button labeled "Confirm" with a value set to NOTICE_YES is provided.

* The button can display text only; no graphic images can be displayed.

The Mouseless Model allows keyboard actions for selecting and moving between notice buttons (refer to Section 6.13, “The Mouseless Model,” in Chapter 6, *Handling Input*).

12.2 Types of Notices

The `simple_notice.c` program shows a default, standard notice. The default notice does not lock the screen but does block the thread of execution. This section describes the two types of notices:

- Standard notices that do not lock the screen. Standard notices have two varieties: those that block the input to the application, and those that do not.
- Screen-locking notices that lock the screen and blocks input to the application.

The `Boolean` attribute `NOTICE_LOCK_SCREEN` determines the type of the notice. When `NOTICE_LOCK_SCREEN` is `FALSE`, the attribute `NOTICE_BLOCK_THREAD` determines whether the notice blocks the input to the application.

After creating a notice and popping it up or down, you can change its type. For example, if `notice` is created with `NOTICE_LOCK_SCREEN` set to `TRUE` (a screen-locking notice), the following call is valid:

```
xv_set(notice, NOTICE_LOCK_SCREEN, FALSE,
     NULL);
```

After doing this your notice is a standard notice and you can, if necessary, use the type-specific attributes that apply to the standard notice (such as `NOTICE_BLOCK_THREAD`).

12.2.1 Standard Notices

When the attribute `NOTICE_LOCK_SCREEN` is set to `FALSE`, the notice is a *standard notice* and it does not lock the screen (this is the default value for `NOTICE_LOCK_SCREEN`). Whether it blocks the thread of execution depends upon the value of the attribute `NOTICE_BLOCK_THREAD` (its default value is `TRUE`). The attribute `NOTICE_STATUS` is used to determine which button was pressed.

When the notice is displayed, the state of the application’s other windows is as follows:

- If `NOTICE_BLOCK_THREAD` is set to `TRUE`, then no windows of the application, except the notice window, will detect mouse and keyboard input (also see `xv_window_loop()`).
- If `NOTICE_BLOCK_THREAD` is set to `FALSE`, only the frame that owns the notice will ignore mouse and keyboard input. Additional frames that need to be put in this state can be added with `NOTICE_BUSY_FRAMES`. All such frames will have their headers grayed out (also see `FRAME_BUSY`).

Table 12-1 lists additional notice attributes that apply only to standard notices (when NOTICE_LOCK_SCREEN is FALSE).

Table 12-1. Notice Attributes (used with NOTICE_LOCK_SCREEN = FALSE)

Attribute	Procedures
NOTICE_BLOCK_THREAD	create, set
NOTICE_BUSY_FRAMES	create, set
NOTICE_EVENT_PROC	create, set

Standard notices are centered in the owner frame and are always on top of the frame that owns them. If the frame is in the iconified state when the notice is mapped, the notice will be placed centered at the location of the pointer.

NOTICE_BLOCK_THREAD is relevant only for standard notices, those that have NOTICE_LOCK_SCREEN = FALSE. Example 12-2 shows a standard, thread-blocking notice.

Example 12-2. Creating a standard notice

```
/*
 * Display a notice that does not lock the screen.
 * This doesn’t return until a button on the notice is pressed.
 * This is a standard blocking notice.
 */
frame_notice = xv_create(frame, NOTICE,
            NOTICE_LOCK_SCREEN, FALSE, /* default */
            NOTICE_BLOCK_THREAD, TRUE, /* default */
            NOTICE_MESSAGE_STRINGS,
                "Are you sure you want to Quit?",
            NULL,
            NOTICE_BUTTON_YES, "Confirm",
            NOTICE_BUTTON_NO, "Cancel",
            NOTICE_NO_BEEPING, TRUE,
            NOTICE_STATUS, &result,
            XV_SHOW, TRUE,
            NULL);

switch (result)  {
case NOTICE_YES:
     /*confirm */
      exit(0);
      break;
case NOTICE_NO:
   /* Cancel */
      break;
default:
      break;
}
```

12.2.1.1 Using a notice callback

The previous notice examples did not use a callback for notice events. A callback may be defined to handle the notice events for a standard notice. The attribute NOTICE_EVENT_PROC specifies an application-defined callback procedure that is called when any of the buttons on the notice are selected. This procedure has the following format:

```
void
notice_event_proc(notice, value, event)
    Xv_Notice  notice;    /* public handle to notice      */
    int        value;     /* value associated with button */
    Event      *event;    /* Pointer to struct with event info.*/
```

This procedure is called before the notice pops down. If a procedure is not specified for NOTICE_EVENT_PROC, the notice still pops down when a button is pressed. Example 12-3 demonstrates a notice using NOTICE_EVENT_PROC. In this example, the callback procedure my_notice_event_proc() is defined to handle all the notice button selections.

Example 12-3. A notice using a callback

```
xv_create(parent, NOTICE,
        NOTICE_LOCK_SCREEN, FALSE,
        NOTICE_BUTTON, "Save Changes", 100,
        NOTICE_BUTTON, "Cancel", 101,
        NOTICE_BUTTON, "Quit", 102,
        NOTICE_MESSAGE_STRINGS,
            "Press Save Changes to save changes to file and quit",
            "Press Cancel to continue",
            "Press Quit to quit",
        NULL,
        NOTICE_EVENT_PROC, my_notice_event_proc,
        NULL);


/* my_notice_event_proc() Procedure */

my_notice_event_proc(notice, value, event)
Xv_Notice  notice;
int        value;
Event      *event;
{
    switch(value)  {

    case 100:
        /* code for save changes */
    break;

    case 101:
        /* code for cancel */
    break;

    case 102:
        /* code for quit */
    break;

    default:
        printf("Bad button value!!\n");
```

Example 12-3. A notice using a callback (continued)

```
    break;
    }

}
```

12.2.1.2 Selecting the busy frames

You can use NOTICE_BUSY_FRAMES to specify the frames or sub-windows that should be set to "busy" during notice pop-up. NOTICE_BUSY_FRAMES only takes frames for its values. The following code shows how to use NOTICE_BUSY_FRAMES.

```
void   my_notice_event_proc();

xv_create(parent, NOTICE,
        NOTICE_LOCK_SCREEN, FALSE,
        NOTICE_BUTTON, "Save Changes", 100,
        NOTICE_BUTTON, "Cancel", 101,
        NOTICE_BUTTON, "Quit", 102,
        NOTICE_MESSAGE_STRINGS,
            "Press Save Changes to save changes to file and quit",
            "Press Cancel to continue",
            "Press Quit to quit",
        NULL,
        NOTICE_EVENT_PROC, my_notice_event_proc,
          NOTICE_BUSY_FRAMES,
               sub_frame1,  /* frames to make busy  */
               sub_frame2,  /* during pop-up.       */
               NULL,
        NULL);
```

12.2.2 Notices That Lock the Screen

To create a screen-locking notice, set NOTICE_LOCK_SCREEN to TRUE. Screen-locking notices lock the screen *and* block the thread of execution for the application. You create a screen-locking notice as follows:

```
notice = xv_create(frame, NOTICE,
            NOTICE_LOCK_SCREEN, TRUE,
            NOTICE_MESSAGE_STRINGS,
                "Are you sure you want to Quit?",
                NULL,
            NOTICE_BUTTON_YES, "Confirm",
            NOTICE_BUTTON_NO, "Cancel",
            XV_SHOW, TRUE,
            NULL);
```

Similarly, the calls below set the type of the notice, `error_notice`, to make it a screen-locking notice.

```
xv_set(error_notice, XV_SHOW, TRUE,
     NOTICE_LOCK_SCREEN, TRUE,
     NULL);
```

All screen-locking notices block the thread of execution. Additional attributes apply when `NOTICE_LOCK_SCREEN` is `TRUE`. Table 12-2 lists the attributes that apply when `NOTICE_LOCK_SCREEN` is `TRUE`.

Table 12-2. Screen-Locking Notice Attributes (for NOTICE_LOCK_SCREEN = TRUE)

Attribute	Procedures
NOTICE_FOCUS_XY	create, set
NOTICE_TRIGGER	create, set
NOTICE_TRIGGER_EVENT	create, set

The position from which the notice shadow emanates is described by the attribute `NOTICE_FOCUS_XY`. This value defaults to the current mouse position when the application maps the notice. Example 12-4 shows code for a screen-locking notice.

Example 12-4. Creating a screen-locking notice

```
int   return_val;
/* Create notice
 * Pop up notice with shadow from (100, 200) relative to "parent"
 * - this blocks
 */
notice = xv_create(parent, NOTICE,
     NOTICE_LOCK_SCREEN, TRUE,
     NOTICE_BUTTON, "Save Changes", 100,
     NOTICE_BUTTON, "Cancel", 101,
     NOTICE_BUTTON, "Quit", 102,
     NOTICE_MESSAGE_STRINGS,
        "Press Save Changes to save changes to file and quit",
        "Press Cancel to continue",
        "Press Quit to quit",
     NULL,
     NOTICE_FOCUS_XY, 100, 200,
     NOTICE_STATUS, &notice_stat,
     XV_SHOW, TRUE,
     NULL);

/*
* Notice pops down when a button is pressed.
* notice_stat contains the value of the button that was clicked on.
*/
switch (notice_stat)  {
         case 100:
                /* save changes */
              break;
```

Example 12-4. Creating a screen-locking notice (continued)

```
        case 101:
             /* cancel */
            break;

        case 102:
             /* quit */
            break;
}
```

12.2.2.1 Notice triggers

If you want to assign accelerators to screen-locking notice buttons, or if you find it necessary to give the user the choice of using mouse buttons or keyboard events to respond to a notice, you can identify *triggers* that pop down the notice. The value of `NOTICE_STATUS` in this case is `NOTICE_TRIGGERED`, and the event that caused the trigger will be in the `Event` specified by `NOTICE_TRIGGER_EVENT`. When triggers are not used, the `Event` pointer can be `NULL`. Example 12-5 shows how to use `NOTICE_TRIGGER` to catch a particular event in a notice.

Example 12-5. The trigger_notice.c program

```
/*
 * trigger_notice.c -- Demonstrate the use of triggers in notices.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Xv_opaque   my_notify_proc();
    extern void exit();

    /*
     * Initialize XView, create a frame, a panel and one panel button.
     */
    xv_init(XV_INIT_ARGS, argc, argv, NULL);
    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Move",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        NULL);

    /* make sure everything looks good */
```

Example 12-5. The trigger_notice.c program (continued)

```
    window_fit(panel);
    window_fit(frame);

    /* start window event processing */
    xv_main_loop(frame);
}

/*
 * my_notify_proc() -- called when the user selects the "Move"
 * panel button.  Put up a notice to get new coordinates
 * to move the main window.
 */
Xv_opaque
my_notify_proc(item, event)
Panel_item  item;
Event       *event;
{
    int         result, x, y;
    Panel       panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);
    Frame       frame = (Frame)xv_get(panel, XV_OWNER);
    Xv_notice   notice;

    x = event_x(event), y = event_y(event);
    printf("original click relative to panel: %d, %d0, x, y);
    notice = xv_create(panel, NOTICE,
     NOTICE_LOCK_SCREEN, TRUE,
     NOTICE_TRIGGER_EVENT, event,
     NOTICE_STATUS, &result,
     XV_SHOW, TRUE,
        NOTICE_FOCUS_XY,        x, y,
        NOTICE_MESSAGE_STRINGS,
            "You may move the window to a new location specified by",
            "clicking the Left Mouse Button somewhere on the screen",
            "or cancel this operation by selecting
            NULL,
        NOTICE_BUTTON_YES,      "cancel",
        NOTICE_TRIGGER,         MS_LEFT,
        NOTICE_NO_BEEPING,      TRUE,
        NULL);

    if (result == NOTICE_TRIGGERED) {
        x = event_x(event) + (int)xv_get(frame, XV_X);
        y = event_y(event) + (int)xv_get(frame, XV_Y);
        printf("screen x,y: %d, %d0, x, y);
        xv_set(frame, XV_X, x, XV_Y, y, NULL);
    }

    xv_destroy_safe(notice);
}
```

When this program is run and the user selects the Move panel button, a notice is displayed instructing the user to select a new position for the application window. When the user selects a new location, the window frame moves to that position.

When the notice pops down, the `Event` structure that `NOTICE_TRIGGER_EVENT` points to contains the event that triggered the notice (popped it down). The *x* and *y* coordinates in the `Event` structure are relative to the origin of the notice-owner window.

To translate these coordinates to screen-specific coordinates, save the original event location and add to that the (*x, y*) coordinates returned in `NOTICE_TRIGGER_EVENT` when the notice pops down, as well as the current coordinates of the frame (main application).

Before leaving *trigger_notice.c*, we should mention the attribute `NOTICE_NO_BEEPING` that is used to prevent the notice from beeping when it is displayed. Beeping the screen is usually done when there is an error condition you wish to alert the user about. In this example, there is no error condition—it is a simple dialog with the user.

12.3 Destroying a Notice

Notices can be destroyed with `xv_destroy(notice)`. If a notice is destroyed when it is visible, it will be taken down. Use `xv_destroy_safe()` if the destruction is done in a `NOTICE_EVENT_PROC`.

12.4 Another Example

In the previous example, we used many of the attributes covered in this section in addition to using some generic and common attributes for the panel items. Example 12-6 goes a little further to demonstrate how the `NOTICE` package works in conjunction with the rest of XView. It creates a frame, a panel with two panel buttons, and a message item. Initially, only the Quit button and the Commit button are displayed. When the user selects either button, a notice pops up asking the user to confirm or cancel the proposed action. If the user confirms quitting the program, the program quits. Otherwise, the result, either Confirmed or Canceled, is displayed as the text of the message item. In previous examples, the notice is destroyed immediately after it is unmapped and the status is obtained. In this example, it is not destroyed, but is reused over and over again.

Example 12-6. The notice.c program

```
/*
 * notice.c --
 * This application creates a frame, a panel, and 3 panel buttons.
 * A message button, a Quit button (to exit the program) and a
 * dummy "commit" button.  Extra data is attached to the panel
 * items by the use of XV_KEY_DATA.  The callback routine for the
 * quit and Commit buttons is generalized enough that it can apply
 * to either button (or any arbitrary button) because it extracts
 * the expected "data" (via XV_KEY_DATA) from whatever panel
```

Example 12-6. The notice.c program (continued)

```
 * button might have called it.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

/*
 * assign "data" to panel items using XV_KEY_DATA ... attach the
 * message panel item, a prompt string specific for the panel
 * item’s notice prompt, and a callback function if the user
 * chooses "yes".
 */
#define MSG_ITEM        10 /* any arbitrary integer */
#define NOTICE_PROMPT   11
#define CALLBACK_FUNC   12

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Panel_item  msg_item;
    Xv_opaque   my_notify_proc();
    extern int  exit();

    /*
     * Initialize XView, and create frame, panel and buttons.
     */
    xv_init(XV_INIT_ARGS, argc, argv, NULL);
    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    msg_item = (Panel_item)xv_create(panel, PANEL_MESSAGE, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        XV_KEY_DATA,            MSG_ITEM,       msg_item,
        /*
         * attach a prompt specific for this button used by
         * the notice.
         */
        XV_KEY_DATA,            NOTICE_PROMPT,  "Really Quit?",
        /*
         * a callback function to call if the user answers "yes"
         * to prompt
         */
        XV_KEY_DATA,            CALLBACK_FUNC,  exit,
        NULL);
    /*
     * now that the Quit button is under the message item,
     * layout horizontally
     */
```

Example 12-6. The notice.c program (continued)

```
    xv_set(panel, PANEL_LAYOUT, PANEL_HORIZONTAL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Commit...",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        XV_KEY_DATA,            MSG_ITEM,       msg_item,
        /*
         * attach a prompt specific for this button used by
         * notices
         */
        XV_KEY_DATA,            NOTICE_PROMPT,  "Update all changes?",
        /*
         * Note there is no callback func here, but one could be
         * written
         */
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * my_notify_proc()
 * The "key data" associated with the panel item is extracted via
 * xv_get().  The resulting choice is displayed in the panel
 * message item.
 */
Xv_opaque
my_notify_proc(item, event)
Panel_item  item;
Event      *event;
{
    int          result;
    int        (*func)();
    char        *prompt;
    Panel_item   msg_item;
    Panel        panel;
    static Xv_notice    notice = NULL;

    func = (int(*)())xv_get(item, XV_KEY_DATA, CALLBACK_FUNC);
    prompt = (char *)xv_get(item, XV_KEY_DATA, NOTICE_PROMPT);
    msg_item = (Panel_item)xv_get(item, XV_KEY_DATA, MSG_ITEM);
    panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);
    /*
     *  Create the notice and get a response.
     */
    if (!notice)  {
        notice = xv_create(panel, NOTICE,
          NOTICE_LOCK_SCREEN, TRUE,
          NOTICE_STATUS, &result,
          XV_SHOW, TRUE,
            NOTICE_MESSAGE_STRINGS,
                    prompt,
                    "Press YES to confirm",
                    "Press NO to cancel",
                    NULL,
```

Example 12-6. The notice.c program (continued)

```
            NOTICE_BUTTON_YES,      "YES",
            NOTICE_BUTTON_NO,       "NO",
            NULL);
    }
    else  {
      /*
       * If the notice has been created, just set its
       * message strings and show it.
       */
      xv_set(notice,
          XV_SHOW, TRUE,
            NOTICE_MESSAGE_STRINGS,
                    prompt,
                    "Press YES to confirm",
                    "Press NO to cancel",
                    NULL,
          NULL);
    }

    switch(result) {
        case NOTICE_YES:
            xv_set(msg_item, PANEL_LABEL_STRING, "Confirmed", NULL);
            if (func)
                (*func)();
            break;
        case NOTICE_NO:
            xv_set(msg_item, PANEL_LABEL_STRING, "Cancelled", NULL);
            break;
        case NOTICE_FAILED:
            xv_set(msg_item, PANEL_LABEL_STRING, "unable to pop-up",
              NULL);
            break;
        default:
            xv_set(msg_item, PANEL_LABEL_STRING, "unknown choice",
              NULL);
    }
}
```

12.5 Notice Package Summary

Table 12-3 lists the attributes for the `NOTICE` package. These attributes are described fully in the *XView Reference Manual*.

Table 12-3. Notice Attributes

NOTICE_BLOCK_THREAD	NOTICE_LOCK_SCREEN
NOTICE_BUSY_FRAMES	NOTICE_MESSAGE_STRING
NOTICE_BUTTON	NOTICE_MESSAGE_STRINGS
NOTICE_BUTTON_NO	NOTICE_MESSAGE_STRINGS_ARRAY_PTR
NOTICE_BUTTON_YES	NOTICE_NO_BEEPING
NOTICE_EVENT_PROC	NOTICE_STATUS
NOTICE_FOCUS_XY	NOTICE_TRIGGER
NOTICE_FONT	NOTICE_TRIGGER_EVENT
XV_KEY_DATA	XV_SHOW

13
Cursors

A *cursor* is an image that tracks the mouse on the display. Each window has its own cursor which you can change. There are some cursors defined by OPEN LOOK that correspond to specific window manager operations such as resizing or dragging windows. For these cases, you cannot redefine a cursor. However, for windows in your application, you can assign any cursor image you like.

13.1 Creating Cursors

To use the `CURSOR` package, include the header file *<xview/cursor.h>*. It provides the necessary types and definitions for using the package. The cursor object's type is `Xv_Cursor`. Figure 13-1 shows the class hierarchy for a cursor object.

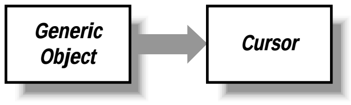


Figure 13-1. Cursor class hierarchy

In general, to create a cursor, create an image and a cursor using that image as the `CURSOR_IMAGE` data:

```
Server_image  svr_image;
Xv_Cursor     cursor;

cursor = (Xv_Cursor)xv_create(owner, CURSOR,
    CURSOR_IMAGE,  svr_image,
    NULL);
```

The `owner` of the cursor may be any XView object. The root window associated with the XView object is used internally by the `CURSOR` package. If `NULL`, then the root window of the default screen is used.

The cursor is then assigned to a window associated with an XView object such as a frame, canvas, or panel:

```
xv_set(window, WIN_CURSOR, cursor, NULL);
```

You must supply the handle of an XView window in the parent parameter when getting `WIN_CURSOR`. Getting `WIN_CURSOR` on the root window will return `NULL`. It is illegal to assign a cursor to a window if the screens do not match. This is normally not a problem unless you are using multiple displays in your application. In this case, you should be sure to use an XView object that has a common display as the owner for the cursor. In the code line above, *`window`* should be the visible window to the application. For canvases and panels, this should be the *paint window*, not the canvas or panel object itself.* If you assign your own cursor to an openwin object (such as a canvas or panel) and the object has been split (either by the user *splitting views* or by the application), then the application is responsible for assigning the cursor to each new paint window.

13.1.1 simple_cursor.c

To introduce how to use the `CURSOR` package, we'll start with a short program that shows how to set the cursor for a canvas.

Example 13-1. The simple_cursor.c program

```
/*
 * simple_cursor.c -- create a cursor (looks like an hourglass) and
 * assign it to a canvas window.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/cursor.h>
#include <xview/svrimage.h>

/* data that describes the cursor's image -- see SERVER_IMAGE below */
short cursor_bits[ ] = {
/* Width=16, Height=16, Depth=1, */
    0x7FFE,0x4002,0x200C,0x1A38,0x0FF0,0x07E0,0x03C0,0x0180,
    0x0180,0x0240,0x0520,0x0810,0x1108,0x23C4,0x47E2,0x7FFE
};

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame        frame;
    Canvas       canvas;
    Xv_Cursor    cursor;
    Server_image svr_image;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

*See Chapter 5, *Canvases and Openwin*, for more information about the paint window.

Example 13-1. The simple_cursor.c program (continued)

```
    /*
     * create a server image to use as the cursor's image.
     */
    svr_image = (Server_image)xv_create(XV_NULL, SERVER_IMAGE,
        XV_WIDTH,               16,
        XV_HEIGHT,              16,
        SERVER_IMAGE_BITS,      cursor_bits,
        NULL);
    /*
     * create a cursor based on the image just created
     */
    cursor = (Xv_Cursor)xv_create(XV_NULL, CURSOR,
        CURSOR_IMAGE,           svr_image,
        NULL);

    /*
     * Create a base frame and a canvas
     */
    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               100,
        XV_HEIGHT,              100,
        NULL);
    /*
     * set the cursor to the paint window for the canvas
     * Do not set it for the canvas itself.
     */
    xv_set(xv_get(canvas, CANVAS_NTH_PAINT_WINDOW, 0),
        WIN_CURSOR,             cursor,
        NULL);

    window_fit(frame);
    window_main_loop(frame);
}
```

Beware that if a canvas (or any openwin object) is split, the new view (which has a corresponding paint window) does not inherit the cursor from the old view window.* Note that the server images used in cursors must be one-bit deep. Cursors can have two colors associated with them by specifying foreground and background colors; you cannot specify server images whose depths are greater than 1. See Section 13.4, "Color Cursors."

*Chapter 5, *Canvases and Openwin*, discusses splitting views.

13.2 Predefined Cursors

A number of predefined cursors are available in the CURSOR package for use as OPEN LOOK cursors. To use these cursors, you may specify the CURSOR_SRC_CHAR and CURSOR_MASK_CHAR attributes with certain predefined constants as values for these attributes. In <*xview/cursor.h*>, there are some OPEN LOOK cursor defines prefixed by OLC_. When using these attributes, you should not use the CURSOR_IMAGE attribute since you cannot use both simultaneously. Using the previous example, we can remove the SERVER_IMAGE references and modify the call to create the cursor:

```
cursor = xv_create(NULL, CURSOR,
    CURSOR_SRC_CHAR, OLC_BUSY_PTR,
    NULL);
```

Predefined cursors are really images from a pre-built font. The *value* in the attribute-value pair is the character to use from that font—or rather, it is the index into the array of glyphs that the font contains. The glyph from the font is extracted and used as the image. You can use the attribute CURSOR_MASK_CHAR similarly. This image is used as the mask for the source image. If no mask is given, the same image used as the source is used as the mask.*

13.3 The Hotspot and Cursor Location

The *hotspot* on a cursor is the location in which the cursor is located if the user generates an event like pressing a mouse button or typing at the keyboard, or if you were to query its position. For example, if a cursor is shaped like an arrow, the hotspot should be at the tip of the arrow. If the hotspot for a cursor were set to (0, 0), then the hotspot would be the upper-left corner of the image used. A cursor shaped like a bull's eye (16x16) might have its hotspot at (7, 7) to indicate that the focus for the cursor is in the middle.† You set a cursor's hotspot with the attributes CURSOR_XHOT and CURSOR_YHOT. CURSOR_XHOT specifies the *x* coordinate of the hotspot. CURSOR_YHOT specifies the *y* coordinate of the hotspot. You can find out what the current position of the cursor is by using the attribute WIN_MOUSE_XY, as in:

```
r = (Rect *)xv_get(window, WIN_MOUSE_XY);
```

The return value from xv_get() is a pointer to a Rect structure. The r_width and r_height fields of this structure are unused (0, 0), but the r_top and r_left fields indicate the position of the hotspot for the cursor with respect to the window, window. The program in Example 13-2 demonstrates how this is used, and it shows how to create your own pixmap for a cursor image.

*See XCreateGlyphCursor and XCreatePixmapCursor in Volume Two, *Xlib Reference Manual*.
†The value 7, 7 is used because the origin is at 0, 0—not 1, 1.

Example 13-2. The hot_spot.c program

```
/*
 * hot_spot.c -- create a cursor and query its position on the
 * screen and in the panel's window.
 * Our own function, create_cursor(), attaches a new cursor to the
 * window parameter passed into the function.
 */
#include <X11/X.h>
#include <X11/Xlib.h>           /* for the xlib graphics */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/cursor.h>
#include <xview/svrimage.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame       frame;
    Panel       panel;
    void        do_it();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    /*
     * Create a base frame, a panel, and a panel button.
     */
    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    create_cursor(xv_get(panel, CANVAS_NTH_PAINT_WINDOW, 0));
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Push Me",
        PANEL_NOTIFY_PROC,      do_it,
        NULL);

    window_fit(panel);
    window_fit(frame);
    window_main_loop(frame);
}

/*
 * When user selects the panel button, the current mouse location is
 * printed relative to the panel's window and to the screen.
 * This location is governed by the hot spot on the cursor.
 */
void
do_it(item, event)
{
    Rect *r;
    Panel panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);

    r = (Rect *)xv_get(xv_get(panel, XV_ROOT), WIN_MOUSE_XY);
    printf("Root window: %d %d\n", r->r_left, r->r_top);
    r = (Rect *)xv_get(xv_get(panel,
                    CANVAS_NTH_PAINT_WINDOW, 0), WIN_MOUSE_XY);
    printf("Panel window: %d %d\n", r->r_left, r->r_top);
}
```

Example 13-2. The hot_spot.c program (continued)

```
/*
 * create_cursor() creates a bull’s eye cursor and assigns it
 * to the window (parameter).
 */
create_cursor(window)
Xv_Window window;
{
    Xv_Cursor      cursor;
    Server_image   image;
    Pixmap         pixmap;
    Display        *dpy = (Display *)xv_get(window, XV_DISPLAY);
    GC             gc;
    XGCValues      gcvalues;

    image = (Server_image)xv_create(XV_NULL, SERVER_IMAGE,
        XV_WIDTH,       16,
        XV_HEIGHT,      16,
        NULL);
    pixmap = (Pixmap)xv_get(image, XV_XID);
    /* Create GC with reversed foreground and background colors to
     * clear pixmap first.  Use 1 and 0 because pixmap is 1-bit deep.
     */
    gcvalues.foreground = 0;
    gcvalues.background = 1;
    gc = XCreateGC(dpy, pixmap, GCForeground|GCBackground, &gcvalues);
    XFillRectangle(dpy, pixmap, gc, 0, 0, 16, 16);
    /*
     * Reset foreground and background values for XDrawArc() routines.
     */
    gcvalues.foreground = 1;
    gcvalues.background = 0;
    XChangeGC(dpy, gc, GCForeground | GCBackground, &gcvalues);
    XDrawArc(dpy, pixmap, gc, 2, 2, 12, 12, 0, 360 * 64);
    XDrawArc(dpy, pixmap, gc, 6, 6, 4, 4, 0, 360 * 64);

    /* Create cursor and assign it to the window (parameter) */
    cursor = xv_create(XV_NULL, CURSOR,
        CURSOR_IMAGE,   image,
        CURSOR_XHOT,    7,
        CURSOR_YHOT,    7,
        NULL);
    xv_set(window, WIN_CURSOR, cursor, NULL);

    /* free the GC -- the cursor and the image must not be freed. */
    XFreeGC(dpy, gc);
}
```

When the program is running, each time the panel button is pushed, it prints the cursor’s position relative to the panel’s window and relative to the root window (absolute screen coordinates). You can move the base frame around on the screen to see how the root window coordinates change.

The routine `create_cursor()` creates a bull's eye cursor for the window passed as the parameter to the routine. The cursor image must be a `Server_image`, so we first create a server image, then get the `Pixmap` associated with it using the `XV_XID`, and lastly use Xlib graphics to draw two circles in the pixmap.

We need a `GC`, so we create one based on the pixmap obtained from the server image. The pixmap is one-bit deep, so the foreground and background colors are set to 0, 1 (to clear the pixmap first), then to 1, 0 so as to draw the two circles. We then create the cursor using the server image and setting the hotspots accordingly. We free the `gc`, but the `Server_image` (which contains the pixmap) and the `cursor` must not be freed so the cursor can be maintained by the window.

If you would rather set the cursor for a window using raw Xlib calls such as `XCreatePixmapCursor`, `XCreateGlyphCursor`, or `XCreateFontCursor`, use the X window associated with the `Xv_Window` parameter. To get it, use:

```
xv_get(window, XV_XID)
```

and assign an X `Cursor` object to that window.*

13.4 Color Cursors

You can define the foreground and background colors of a cursor independently of the window the cursor is assigned to. You may not have more than two colors per cursor because X does not support color images as cursor glyphs. Thus, to create or modify an existing cursor to have color, you need to specify foreground and background colors. Because of the use of color, the header file *<xview/cms.h>* must be included. The colors are of type `Xv_singlecolor` and should be initialized before use:

```
#include <xview/cms.h>
...
Xv_singlecolor   fg, bg;

bg.red = 250, bg.green = 230, bg.blue = 30;
fg.red = 180, fg.green = 100, fg.blue = 20;

cursor = xv_create(NULL, CURSOR,
    CURSOR_IMAGE,             image,
    CURSOR_FOREGROUND_COLOR,  &fg,
    CURSOR_BACKGROUND_COLOR,  &bg,
    NULL);
```

Note, by default, a cursor is created with a mask equal to the image; therefore, there is no background color. To use a background color, set the cursor's `CURSOR_BACKGROUND_COLOR` and the `CURSOR_IMAGE` attributes, and then set the `CURSOR_OP` as follows:

```
xv_set(cursor,CURSOR_OP,PIX_SRC,NULL);
```

*See Volume One, *Xlib Programming Manual*.

13.5 Support for Text Drag and Drop

The Cursor package supports drag and drop cursors for text. There are attributes that change the cursor to indicate that text is being dragged. When the currently selected item is over an acceptable drop site, a preview cursor is displayed. There is also an attribute to support a reject cursor as well, but it is an unsupported attribute.

The attribute `CURSOR_DRAG_STATE` indicates whether the cursor is over a neutral zone (`CURSOR_NEUTRAL`), a valid drop zone (`CURSOR_ACCEPT`), or an invalid drop zone (`CURSOR_REJECT`). The shape of the cursor varies depending on the state.*

`CURSOR_DRAG_TYPE` changes the cursor to indicate whether a move (`CURSOR_MOVE`) or copy (`CURSOR_DUPLICATE`) operation is being performed. The duplicate version has a shadow. When combined with `CURSOR_STRING`, you get either a text move or a text duplicate cursor.

`CURSOR_STRING` creates a drag and drop Cursor for text. The value of the attribute is the string which is to be displayed inside the *flying punch card*. If the string exceeds 3 characters, only the first 3 characters are displayed, and an arrow is shown within the cursor. `CURSOR_STRING` is mutually exclusive of `CURSOR_IMAGE`, `CURSOR_SRC_CHAR`, and `CURSOR_MASK_CHAR`. The string is not copied. Once the Drag and Drop operation is complete, the objects used in the operation must be destroyed.

Example 13-3 shows how to use a drag and drop text cursor with a drag and drop object.

Example 13-3. Using drag and drop text cursors

```
if (drag and drop started) {
  type = event_ctrl_is_down(event) ? CURSOR_DUPLICATE : CURSOR_MOVE;
  neutral_drag_cursor = xv_create(window, CURSOR,
          CURSOR_STRING, selected_string,
          CURSOR_DRAG_TYPE, type,
          CURSOR_DRAG_STATE, CURSOR_NEUTRAL,
          NULL);
  accept_drag_cursor = xv_create(window, CURSOR,
          CURSOR_STRING, selected_string,
          CURSOR_DRAG_TYPE, type,
          CURSOR_DRAG_STATE, CURSOR_ACCEPT,
          NULL);
  xv_set(dnd_object,
          DND_CURSOR, neutral_drag_cursor,
          DND_ACCEPT_CURSOR, accept_cursor,
          NULL);
}
```

*Note: The current drag and drop protocol does not support a reject cursor.

13.6 Cursor Package Summary

Table 13-1 shows the CURSOR package procedure. Table 13-2 lists the attributes for the CURSOR package. This information is described fully in the *XView Reference Manual*.

Table 13-1. Cursor Procedure

cursor_copy()

Table 13-2. Cursor Attributes

CURSOR_BACKGROUND_COLOR	CURSOR_OP
CURSOR_DRAG_STATE	CURSOR_SRC_CHAR
CURSOR_DRAG_TYPE	CURSOR_STRING
CURSOR_FOREGROUND_COLOR	CURSOR_XHOT
CURSOR_IMAGE	CURSOR_YHOT
CURSOR_MASK_CHAR	
XV_SHOW	XV_XID

14
Icons

A user may *close* an application to save space on the display. The program is still running and it may even be active, but it is not receiving input from the user. In order to represent the application in its closed state, an *icon* is used. An icon is a small picture that represents the application, as shown in Figure 14-1 from the *OPEN LOOK GUI Specification Guide*.

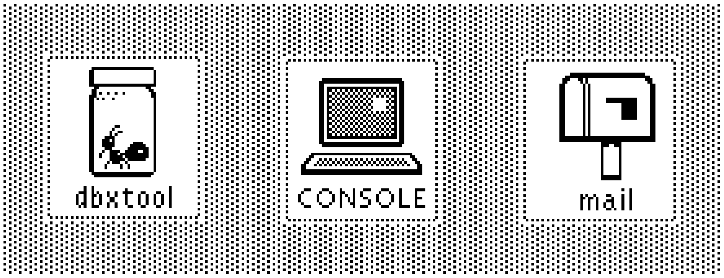


Figure 14-1. Three bordered default icons

The graphic image that icons use may be used for other purposes and, therefore, may be shared among other objects in the application. But the icon image should be designed to easily identify the application while in a closed state. Icons may also have text associated with them. Space is limited, so the text is usually the name of the application.

14.1 Creating and Destroying Icons

To use the `ICON` package, include the header file *<xview/icon.h>*. Figure 14-2 shows the class hierarchy for the `ICON` package.

The form for creating an icon is:

```
Icon icon;

icon = (Icon)xv_create(owner, ICON, attributes, NULL);
```

The `owner` of an icon is a base frame, but it may be created with a `NULL` owner. Once an icon is assigned to a frame, the owner of the icon is changed to that frame. This is another example of *delayed binding*.

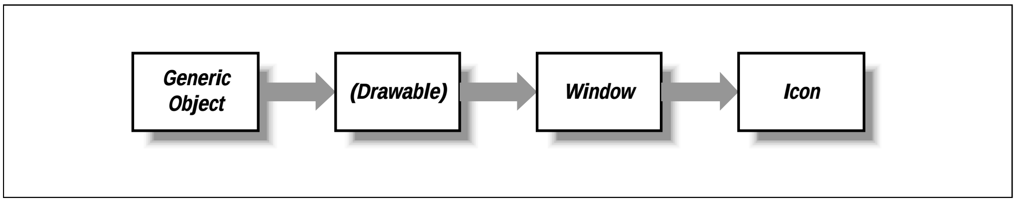


Figure 14-2. Icon package class hierarchy

When destroying an icon, the server image associated with the icon is not destroyed—it is your responsibility to free the server image and the pixmap associated with the icon if needed.

14.2 The Icon's Image

The most important thing about the icon is its graphic representation, so you will also need to be familiar with the SERVER_IMAGE package described in Chapter 15, *Nonvisual Objects*. Once an image is created, you can create an icon and assign it to a frame. This chapter does not discuss the creation of server images for icons, whether they originate from filenames or from the actual data.

The program in Example 14-1 creates two server images—it uses *open.icon* as the image for the panel button and *closed.icon* for the application's icon.* Pressing the panel button causes the application to *close* to its iconic state. You must use a window manager function to open the application back up again.

Example 14-1. The icon_demo.c program

```
/*
 * icon_demo.c -- demonstrate how an icon is used.  Create a server
 * image and create an icon object with the image as the ICON_IMAGE.
 * Use the icon as the frame's icon.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/svrimage.h>
#include <xview/icon.h>

short open_bits[ ] =  {
#include "open.icon"
};

short closed_bits[ ] =  {
#include "closed.icon"
};

main(argc, argv)
```

*The files *open.icon* and *closed.icon* are not included in this book due to their length and complexity. They are bitmap files represented in ASCII *hex* notation. The files are included with the XView distribution.

Example 14-1. The icon_demo.c program (continued)

```
int     argc;
char    *argv[ ];
{
    Frame               frame;
    Panel               panel;
    Server_image        open_image, closed_image;
    Icon                icon;
    void                close_frame();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);

    open_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               64,
        XV_HEIGHT,              64,
        SERVER_IMAGE_BITS,      open_bits,
        NULL);

    closed_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               64,
        XV_HEIGHT,              64,
        SERVER_IMAGE_BITS,      closed_bits,
        NULL);

    (void) xv_create(panel, PANEL_MESSAGE,
        PANEL_LABEL_IMAGE,      open_image,
        PANEL_NOTIFY_PROC,      close_frame,
        NULL);

    icon = (Icon)xv_create(frame, ICON,
        ICON_IMAGE,             closed_image,
        XV_X,                   100,
        XV_Y,                   100,
        NULL);
    xv_set(frame, FRAME_ICON, icon, NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

void
close_frame(item, event)
Panel_item item;
Event *event;
{
    Frame frame = (Frame)xv_get(xv_get(item,
        PANEL_PARENT_PANEL), XV_OWNER);
    xv_set(frame, FRAME_CLOSED, TRUE, NULL);
}
```

The callback routine for the panel button, `close_frame()`, makes a call to an XView routine which sends window manager requests from the client to the window manager. In this case, we set the frame's `FRAME_CLOSED` attribute to `TRUE` to request that the window manager iconify the application associated with the `frame` parameter.*

The position of the image with respect to the icon is set using the attribute `ICON_IMAGE_RECT`, which takes as its value a pointer to a `Rect` structure. The `r_top` and `r_left` fields of the structure indicate the offset from the upperleft corner of the icon where the image is placed. The `r_width` and `r_height` fields describe the size of the image. If the icon is going to be a different size from the size of the icon's image, or if there is going to be text used with this icon, then the `ICON_IMAGE_RECT` attribute should be used. Section 14.2.1, "The Icon Text," has an example.

14.2.0.1 Color icons

You can make color icons in several ways. You can create a color server image and use that as the `ICON_IMAGE`. You can set the foreground and background colors for a monochrome image (1-bit deep image) and change the colormap of the icon. For example:

```
Icon          icon;
unsigned long foreground_index, background_index;
Server_image  image;  /* assume 1-bit deep monochrome image */
Cms           cms;

icon = (Icon)xv_create(frame, ICON,
    ICON_IMAGE,           image,
    WIN_CMS,              cms,
    WIN_FOREGROUND_COLOR, foreground_index,
    WIN_BACKGROUND_COLOR, background_index,
    NULL);
```

The icon created assumes that the colormap object, cms, has been created. The foreground and background colors are indices into the colormap (see Chapter 21, *Color*, for more information on colormap segments). Also see the program *x_draw.c* in Appendix F, *Example Programs*.

Example 14-2. Color cursors

```
#include <xview/xview.h>
#include <xview/svrimage.h>
#include <xview/cms.h>

/* Icon data */
static unsigned short icon_bits[ ] = {
#include "cardback.icon"
};

main(argc,argv)
int     argc;
char    *argv[ ];
{
```

*Window manager functions are discussed in Chapter 4, *Frames*.

Example 14-2. Color cursors (continued)

```
    Frame         frame;

    Icon          icon;
    unsigned long foreground_index, background_index;
    Server_image  image;  /* assume 1-bit deep monochrome image */
    Cms           cms;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create(NULL,FRAME,NULL);

    image = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,             64,
        XV_HEIGHT,            64,
        SERVER_IMAGE_DEPTH,   1,
        SERVER_IMAGE_BITS,    icon_bits,
        NULL);

    foreground_index = 2;
    background_index = 0;

    cms = (Cms)xv_create(NULL,CMS,
        CMS_SIZE,             3,
        CMS_NAMED_COLORS,     "green","red","blue",NULL,
        NULL);

    icon = (Icon)xv_create(frame, ICON,
        ICON_IMAGE,           image,
        WIN_CMS,              cms,
        WIN_FOREGROUND_COLOR, foreground_index,
        WIN_BACKGROUND_COLOR, background_index,
        NULL);

    xv_set(frame, FRAME_ICON, icon, NULL);

    xv_main_loop(frame);
}
```

14.2.0.2 ICON_TRANSPARENT

If the `ICON_TRANSPARENT` attribute is set to `TRUE` (it's `FALSE` by default), the icon's background color is set to the background color of the root window. This may give the effect that the icon is transparent in the case where the root window is a solid color. However, be careful when the root window has a backing bitmap pattern or a different colormap from the icon.

14.2.0.3 ICON_MASK_IMAGE

The attribute ICON_MASK_IMAGE may be used to clip all drawing into the pixmap to the bits set in the Pixmap or Server_image specified here. Therefore, this image should be a 1-bit deep bitmap. The image used as the icon mask is usually a "shadow" of the icon's normal image. That is, it is the same image "filled in," resulting in a totally black 1-bit deep icon that has the same shape as the icon's image.

Example 14-2 can be modified to use this attribute by adding the following code:

```
image_mask = (Server_image)xv_create(NULL, SERVER_IMAGE,
    XV_WIDTH,            64,
    XV_HEIGHT,           64,
    XV_DEPTH,            1,
    SERVER_IMAGE_BITS,    closed_image_mask_bits,
    NULL);

icon = (Icon)xv_create(frame, ICON,
    ICON_IMAGE,      closed_image,
    ICON_MASK_IMAGE, image_mask,
    XV_X,            100,
    XV_Y,            100,
    NULL);
```

When used in conjunction with the ICON_TRANSPARENT attribute, it may be possible to create an icon that appears to have a shape other than a square.

14.2.1 The Icon Text

Each icon can have text associated with it. This text is not part of the icon's image; it is rendered on top of the image after the image is rendered. To specify the text displayed in the icon, use the generic attribute ICON_LABEL.* By default, the text is displayed at the bottom of the icon area. This may overlap the icon's image. You can change the position in which the text is rendered by using the attribute ICON_LABEL_RECT. The value of the attribute describes a rectangular region in which the text will overwrite anything underneath it. If the text does not fit, it is clipped by this region. To keep the entire image on the icon and display the text without writing over the image, define your icon to be large enough to include both the image and the extents of the text without these regions overlapping. This might make your icon a nonstandard size, however. (The size of an icon is typically 64x64 pixels.)

The code fragment in Example 14-3 implements added sizes to compensate for text while preserving enough area to display the entire icon.

Example 14-3. Redefining an icon's size to include its label

```
...
Rect         image_rect, label_rect;
Server_image image;
Icon         icon;
...
```

*ICON_LABEL is defined to be XV_LABEL in <*xview/icon.h*>.

Example 14-3. Redefining an icon's size to include its label (continued)

```
rect_construct(&image_rect, 0, 20, 64, 64);
rect_construct(&label_rect, 0, 0, 64, 20);
icon = xv_create(frame, ICON,
    XV_WIDTH,               64,
    XV_HEIGHT,              64 + 20,
    XV_LABEL,               "Sample",
    ICON_LABEL_RECT,        &label_rect,
    ICON_IMAGE,             image,
    ICON_IMAGE_RECT,        &image_rect,
    NULL);
xv_set(frame, FRAME_ICON, icon, NULL);
```

The first thing we do is construct the image and label area by using the `rect_construct()` macro found in *<xview/rect.h>*. The image is positioned at `0, 20` and is a size of 64x64. The text is positioned at the upperleft corner (`0, 0`), extends to the width of the icon and is 20 pixels high. The size of the icon is 64, and the height of the text is 20, so when we create the icon, we set the height of the icon object using `XV_HEIGHT` at `64+20`.

14.2.2 ICON_TRANSPARENT_LABEL

The `ICON_TRANSPARENT_LABEL` attribute specifies a string that is drawn into the icon using the foreground color only. Pixels other than those in the font set are not affected.

Creating, setting, and getting `ICON_TRANSPARENT_LABEL` is equivalent to creating, setting, and getting `ICON_LABEL`, except that the string is drawn in the foreground color only.

14.3 Icon Package Summary

There are no procedures or macros in the `ICON` package. Table 14-1 shows the attributes for the `ICON` package. This information is described fully in the *XView Reference Manual*.

Table 14-1. Icon Attributes

ICON_FONT	ICON_LABEL_RECT
ICON_HEIGHT	ICON_MASK_IMAGE
ICON_IMAGE	ICON_TRANSPARENT
ICON_IMAGE_RECT	ICON_TRANSPARENT_LABEL
ICON_LABEL	ICON_WIDTH
XV_LABEL	

15
Nonvisual Objects

This chapter addresses nonvisual objects—objects that are not elements of the user interface. Nonvisual objects include the screen, the display, the X11 server, and server images. The `FULLSCREEN` package is used to grab the X server, and an instance of it is considered a non-visual object. Nonvisual objects are not viewed on the screen, but they have a place in the XView object hierarchy. Like all XView objects, they share many of the generic and common properties and can be manipulated using `xv_create()`, `xv_set()`, `xv_get()`, or `xv_find()`.

Nonvisual objects are typically used internally by XView and are seldom used directly in an application. Therefore, this chapter contains advanced material that may not be essential to all programmers. Figure 15-1 shows the class hierarchy for nonvisual objects.

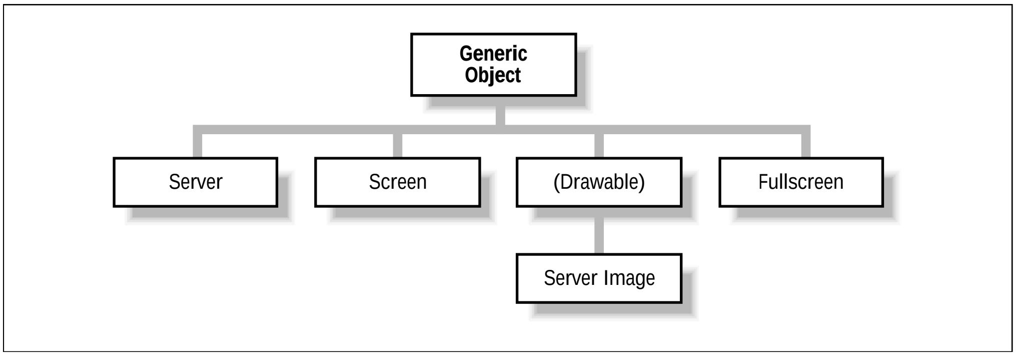


Figure 15-1. Nonvisual objects class hierarchy

15.1 The Display

There is no XView *Display* object. If you need the `Display` data structure as defined by X, you can get the value of the attribute `XV_DISPLAY`. This can be used on virtually any visible XView object except for panel items. For example, to get the display associated with the `Frame`, use:

```
Display *dpy;

dpy = (Display *)xv_get(frame, XV_DISPLAY);
```

The object does not have to be displayed or visible—just created. To do this, the header file <*X11/Xlib.h*> must be included for the declaration of the `Display` data structure. This structure contains a great deal of information that describes attributes of the workstation, the server being used, and more. See Volume One, *Xlib Programming Manual*, and Volume Two, *Xlib Reference Manual*, for more information.

15.2 The Screen Object

An `Xv_Screen` is associated with virtually all XView objects. To use the `Xv_Screen` object, you must include the file <*xview/screen.h*>. To get a handle on the current screen, use `xv_get()` on an object:

```
Xv_Screen xv_screen;

xv_screen = (Xv_Screen)xv_get(frame, XV_SCREEN);
```

The `Xv_Screen` object carries useful information such as the screen number of the root window, all the visuals, the colormap, the server, and so on, that are associated with that screen.

The `Xv_Screen` object differs from the `Screen` data structure defined by Xlib and, in fact, has nothing to do with the X11 `Screen` data type (defined in <*X11/Xlib.h*>). That is, you cannot use the following call to `xv_get()` to get a corresponding `Screen` pointer.

```
xv_get(xv_screen, XV_XID)        /* Doesn't work */
```

There is no associated `XID` for the `SCREEN` package.

Because the X `Screen` type provides information not available from the `Xv_Screen` object, it may be useful to get the X `Screen`. Once the XView screen is obtained, the X `Screen` type can be gotten by using the `SCREEN_NUMBER` of the screen and the `Display` pointer associated with an arbitrary visual object. Example 15-1 demonstrates how to do this for a frame.

Example 15-1. Getting a pointer for a particular frame object (screen.c)

```
/*
 * screen.c -- get some simple info about the current screen:
 * width, height, depth.
 */
#include <xview/xview.h>
#include <xview/screen.h>

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Frame               frame;
    Xv_Screen           screen;
    Display             *dpy;
    int                 screen_no;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);

    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    printf("Server display = ’%s’0, dpy->vendor);
    screen = (Xv_Screen)xv_get(frame, XV_SCREEN);

    screen_no = (int)xv_get(screen, SCREEN_NUMBER);
    printf("Screen #%d: width: %d, height: %d, depth: %d0,
        screen_no,
        DisplayWidth(dpy, screen_no),
        DisplayHeight(dpy, screen_no),
        DefaultDepth(dpy, screen_no));
}
```

As shown in Example 15-1, you can use any of the macros defined in <*X11/Xlib.h*> to get information about the default screen such as the width, height, depth, and so on. From this information, you can get information about the physical frame buffer. These are Xlib-related issues not covered in this book.

15.2.1 Multiple Screens

Each X11 server supports multiple screens. Each screen can have different attributes such as colormaps, depth, size and so on. The screens can actually be different physical devices, although they are connected to the same physical computer. Each screen has its own root window as well, and since the root window is the parent for all base frames, XView can allow windows to exist on any screen.

The way we take advantage of this capability is to first establish a connection to the X11 server and then to get the root window of each screen. With a handle to the root window, we can use it as the parent to any frame we create.

Example 15-2 demonstrates how two frames can be created on two different screens attached to a single server. Note that this code relies on the fact that the server supports more than one screen.

Example 15-2. Display a base frame on two screens

```
/*
 * multiscreen.c -- display a base frame on two different screens
 * attached to the same X11 server.  In order for this program to
 * work, you must have two screens.
 */
#include <xview/xview.h>

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Xv_Server  server;
    Xv_Screen  screen_0, screen_1;
    Xv_Window  root_0, root_1;
    Frame      frame_0, frame_1;

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);

    screen_0 = (Xv_Screen) xv_get(server, SERVER_NTH_SCREEN, 0);
    root_0 = (Xv_Window) xv_get(screen_0, XV_ROOT);

    screen_1 = (Xv_Screen) xv_get(server, SERVER_NTH_SCREEN, 1);
    root_1 = (Xv_Window) xv_get(screen_1, XV_ROOT);

    frame_0 = (Frame) xv_create(root_0, FRAME,
        FRAME_LABEL,    "SCREEN 0",
        NULL);

    frame_1 = (Frame) xv_create(root_1, FRAME,
        FRAME_LABEL,     "SCREEN 1",
        NULL);

    xv_set(frame_1, XV_SHOW, TRUE, NULL);
    xv_main_loop(frame_0);
}
```

The program implements the design discussed above: `xv_init()` opens a connection to the server and returns a handle to the `Xv_Server` object (see the following section for details). It also retrieves the `Xv_Screen` object as well as the root window for each screen. Next, a base frame is created for each root window. However, since we are going to call `xv_main_loop()` on `frame_0`, we need to insert `frame_1` into the window tree. Otherwise, it will never be mapped to its screen because `xv_main_loop()` only installs and maps the window of the object passed to it.

15.3 The SERVER Package

The SERVER package may be used to initialize the connection with the X server running on any workstation on the network. Once the connection has been made, the package allows you to query the server for information. xv_init(), the routine that initializes the XView Toolkit, opens a connection to the server and returns a handle to an Xv_Server object. While more than one server can be created, xv_init() only establishes a connection to *one* server. The server object returned by xv_init() is also the server pointed to by the external global variable, xv_default_server. Programs that do not save the Xv_Server object returned by xv_init() can reference this global variable instead.

Subsequent connections to other X11 servers must be made using separate calls to xv_create(). Note that using separate screens is not the same as establishing a connection to other servers—the same server can support multiple screens. See the previous section for ways to access multiple screens in a server.

15.3.1 Creating a Server (Establishing a Connection)

When making any reference to Xv_Server objects, applications should include <*xview/server.h*>. You can open a connection to any server by using xv_create():

```
Xv_Server server;
extern char *server_name;

server = (Xv_Server)xv_create(NULL, SERVER,
    XV_NAME, server_name,
    NULL);
```

Because there is no owner for a server, the owner parameter is ignored and you may pass NULL. The server described by server_name is assumed to have been initialized already. It should be set to the standard format:

hostname:*display.screen*

For example:

```
zipcode:0.1
```

connects the second screen on the first display to the host named *zipcode*. If the connection fails, NULL is returned.

Remember that the user can specify which display is the default by using the *-display* option:

```
% program_name –display zipcode:0
```

Remember that this command-line switch is parsed internally by XView when you call xv_init() in the following way:

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

If xv_init() has not been called by the time the first call to xv_create() is called, the call to xv_create() calls xv_init() internally. This means that if the program gets around to calling xv_init() after it has made any calls to xv_create(), it is a no-op.

Likewise, `xv_init()` creates a server instance, so you cannot establish the *initial* server after calling `xv_init()`.

15.3.2 Connecting to Multiple Servers

You can establish connections to other servers as well as the server opened by `xv_init()` by using `xv_create()` in the way shown above. The standard way for a user to specify a connection to a server is the **`-display`** switch; to allow the user to specify a connection to another server, you should provide an additional command-line option that you parse yourself.

The following code segment allows the user to specify an additional server by using the command-line switch **`-display2`**:

```
Xv_Server server1, server2 = NULL;

server1 = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

/* XView has parsed all the args it knows -- now look for ours */
while (*++argv) {
    if (!strcmp(*argv, "-display2")) {
        if (!*++argv) {
            fputs("Missing server name.\n", stderr);
            exit(1);
        }
        server2 = xv_create(NULL, SERVER, XV_NAME, *argv, NULL);
    }
}

if (server2 == NULL) {
    fputs("Must specify second server.\n", stderr);
    exit(1);
}
```

If you do this, a connection will be established for both servers.

Applications that have connected to multiple servers will typically exit when one of the X servers goes down, causing the windows displayed on the other server to go away. To avoid having the application exit, application programmers should consider forking separate processes for each server connection. Thus losing a server connection can be handled in a reasonable manner.

15.3.3 Getting the Server

One way to get the server to which the application is connected is from the `Xv_Screen` object described in the previous section:

```
server = (Server)xv_get(xv_get(frame, XV_SCREEN), SCREEN_SERVER);
```

With the server object, you can tell the server to synchronize with your application by calling:

```
Xv_Server server;
 . . .
xv_set(server, SERVER_SYNC, TRUE, NULL);
```

The SERVER_SYNC attribute flushes the request buffer and waits for all events and errors to be processed by the server. When the argument is TRUE, as above, SERVER_SYNC discards all events on the input queue.

SERVER_SYNC_AND_PROCESS_EVENTS has the same behavior as SERVER_SYNC, but this processes any events that arrive as a result of the XSync().

```
Xv_Server server;

xv_set(server, SERVER_SYNC_AND_PROCESS_EVENTS, NULL);
```

Note that this attribute takes no value—you specify it and no other attributes. This attribute makes sense only in xv_set().

The attributes SERVER_ATOM and SERVER_ATOM_NAME help manage atoms. SERVER_ATOM is equivalent to XInternAtom() (with the only_if_exists flag set to false). It caches the results on the server object so that subsequent requests for the same atom will not require a round-trip to the X server. For example:

```
Atom  atom;
atom = (Atom) xv_get(server_object, SERVER_ATOM, "TIMESTAMP");
```

SERVER_ATOM_NAME is equivalent to XGetAtomName(). It also caches the results on the server object. The returned string is maintained by XView and should not be modified or freed. XView will free up all strings associated with atoms on that server when the server object is destroyed. For example:

```
char *atom_name;
atom_name (char *)xv_get(server_object, SERVER_ATOM_NAME, atom);
```

15.4 Server Images

A server image is a graphic image stored on the X server. Images on the client side can be stored as XImages or as memory *pixrects*.* The XView Server_image object is not equivalent to X Pixmaps, although pixmaps are part of the Server_image object. Even though pixmaps are stored on the server, the XView object is a client-side object. Because it is an XView object, you can query the dimensions of a Server_image by using XV_WIDTH or XV_HEIGHT, which is something you cannot do with X11 Pixmaps. The server image structure is defined in <*xview/svrimage.h*> as follows:

```
typedef struct {
        Xv_drawable_struct parent_data;
```

*The term *pixrect* is a data type brought over from SunView.

```
            Xv_opaque           private_data;
            Xv_embedding        embedding_data;
            Pixrect             pixrect;
    }   Xv_server_image;
```

15.4.1 Creating Server Images

Applications that wish to use the SERVER_IMAGE package should include <*xview/svrimage.h*>. Server_image objects contain graphic data that is used in icons, cursors, panel buttons—in fact, just about everything in XView that contains graphics. The Server_image object is created using xv_create() in the following manner:

```
#include <xview/svrimage.h>
...
Server_image image;

image = (Server_image)xv_create(owner, SERVER_IMAGE,
    attrs,
    NULL);
```

The owner in the call to xv_create() is an Xv_Screen object. The server that owns this screen owns the newly created image. If the owner is NULL, then the default screen is used. The dimensions of Server_image objects are 16 by 16 by 1, unless the attributes XV_WIDTH, XV_HEIGHT or SERVER_IMAGE_DEPTH are specified. The bitmap data for the server image may be set using either SERVER_IMAGE_BITS or SERVER_IMAGE_X_BITS depending on the format of the data. The data format choices are arrays of short or char types. X11 bitmap data is represented as array of chars, while Sun's pixrect library represents the data as an array of shorts.* The following code segment uses SERVER_IMAGE_BITS to produce a one-bit deep image that looks like a trash can:

```
short image_bits[ ] = {
#include <images/trash.icon>
};

Server_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
    XV_WIDTH,           32,
    XV_HEIGHT,          30,
    SERVER_IMAGE_BITS,  image_bits,
    NULL);
```

Here, the trash can icon was created with its bits stored in an array of shorts. A call to xv_get() with the attribute SERVER_IMAGE_BITS returns the bits for the server image. Manipulating this data does not change the appearance of the server image. A call to xv_set() using SERVER_IMAGE_BITS to specify a new bitmap is required to change a server image.

*Many of Sun's existing applications should use SERVER_IMAGE_BITS when porting to XView. This attribute must be used in order to load bitmap data created by *iconedit*.

To load an image stored as an array of chars (the format used by X11), use SERVER_IMAGE_ X_BITS:

```
#include <X11/bitmaps/xlogo32>

xlogo_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
    XV_HEIGHT,             xlogo32_width,
    XV_WIDTH,              xlogo32_height,
    SERVER_IMAGE_X_BITS,   xlogo32_bits,
    NULL);
```

In both of these cases, the file specified on the #include line must be accessible at the time the program is compiled. Once compiled, the data for the image is stored in the program and the file is no longer needed (e.g., the file may be deleted and the program still displays the image).

Rather than including the file containing the image's bitmap data, you could specify the actual file:

```
char *file = "/usr/include/X11/bitmaps/xlogo32";

server_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
    SERVER_IMAGE_BITMAP_FILE,   file,
    NULL);
```

Be aware that this file must exist and be accessible by any person who runs this program at run time. If for some reason the file is not accessible, then an error is generated. And because the program may be run from any directory, a full pathname should be specified. As shown, the file points to a static string, but file could have a value that is changed by selecting from a list of bitmap filenames. In this case, the code fragment could use xv_set() to set the filename and thus, the Server_image's data. Whenever xv_set() is used to change the data of the image like this, the values of XV_WIDTH and XV_HEIGHT are automatically updated.

Many XView objects require Server_images as values (such as MENU_IMAGE_STRINGS in the MENU package). If you have already created a pixmap and wish to attach it to a server image, you can use:

```
image = (Server_image)xv_set(NULL, SERVER_IMAGE,
    SERVER_IMAGE_PIXMAP,  pixmap,
    NULL);
```

The attribute SERVER_IMAGE_PIXMAP can also be used in xv_get() to return the XID of the pixmap associated with the Server_image.

Normally, a Server_image destroys its pixmap when a new pixmap is created using SERVER_IMAGE_BITS, SERVER_IMAGE_X_BITS, or SERVER_IMAGE_PIXMAP. This default behavior can be turned off by setting the SERVER_IMAGE_SAVE_PIXMAP attribute to TRUE. Be sure to maintain a handle to the pixmap if you specify this attribute and destroy the Server_image.

If the depth of an image is unspecified, it defaults to 1. To create a color image, use SERVER_IMAGE_DEPTH to specify an alternate depth that can support color. You can also specify a colormap to use with this image by specifying SERVER_IMAGE_CMS. This is used for multiplane color images and must be specified before the image bits are set. The colormap specified is assumed to have been created already using the CMS package.

Example 15-3 demonstrates how to use a server image by creating a frame with a panel. On the panel is a button that uses a server image as the `PANEL_LABEL_IMAGE`. The bits used are the same as the *trash.icon* used above.

Example 15-3. The svrimage.c program

```
/*
 * svrimage.c -- demonstrate how a server image can be created and
 * used.  The "bits" used to create the image are taken arbitrarily
 * from <images/trash.icon>
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/svrimage.h>
#include <X11/Xlib.h>

short image_bits[ ] =  {
    0x0000,0x0000, 0x0000,0x0000, 0x0000,0x0000, 0x0000,0x0000,
    0x0007,0xE000, 0x0004,0x2000, 0x03FF,0xFFC0, 0x0200,0x0040,
    0x02FF,0xFF40, 0x0080,0x0100, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x0091,0x1300,
    0x00C0,0x0200, 0x003F,0xFC00
};

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Frame               frame;
    Server_image        image;
    Panel               panel;
    void                exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    image = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               32,
        XV_HEIGHT,              30,
        SERVER_IMAGE_BITS,      image_bits,
        NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_MESSAGE,
        PANEL_LABEL_IMAGE,      image,
        PANEL_NOTIFY_PROC,      exit,
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}
```

15.5 The FULLSCREEN Package

The FULLSCREEN package allows XView clients to grab the server for keyboard and/or pointer use either exclusively or nonexclusively with other applications. This package is used primarily to prompt the user for immediate feedback on a question or to notify the user of an error that needs attention. Typically, the user responds with a button press or a keyboard event. The NOTICE package uses the FULLSCREEN package extensively to implement its functionality. In most cases, you need nothing more than the NOTICE package and should rarely need to use the FULLSCREEN package. The need for this package arises if you choose to implement your own notice or perhaps a user interface item that is not OPEN LOOK-compliant. In either case, this is advanced usage and is beyond the scope of this book. Using the FULLSCREEN package can be very dangerous because it uses the X server's grabbing functions in Xlib. It is possible to get into a state from which you cannot get out except by killing the server remotely or rebooting your workstation. When using a debugger, be extremely careful that you do not set breakpoints within code when the server is in the middle of a grab of some kind. Whatever you do, *do not step through code that creates a* FULLSCREEN *instance.* If this is unavoidable, you should prepare for it by making sure that you have remote access to your workstation or by attaching a terminal to it so you can kill the debugger to free the server.

The flow of control for client code using the FULLSCREEN package is to create a fullscreen instance (grabbing the server), scan for a particular event and destroy the fullscreen instance (freeing the server).

Creating a fullscreen object (grabbing the server) involves `xv_create()` as usual:

```
Fullscreen fs;

fs = xv_create(owner, FULLSCREEN, NULL);
```

The `owner` in this case may be a visible XView object that has a window associated with it* and is currently displayed on the screen (e.g., XV_SHOW is TRUE). If `owner` is NULL, then the root window of the default screen is used as the owner.

The attribute WIN_CURSOR can be used with a fullscreen object to set the mouse cursor displayed while the fullscreen object is active. The default value for WIN_CURSOR is inherited by the fullscreen object from its owner.

Example 15-4 uses the FULLSCREEN package. A simple panel with two panel buttons is created. The Quit button quits the program, and the Fullscreen button calls the `grab()` routine that grabs the server using the FULLSCREEN package and waits for a button to be pressed. Once this happens, the routine frees the fullscreen object, thus releasing the server. Event masks can be specified when creating a FULLSCREEN object to set what kind of events the client window will accept or detect. The event masks are specified using the regular WIN_* attributes. A similar event mask is also needed for `xv_input_readevent()`.†

*This does not include panel items.
†See Chapter 6, *Handling Input*, for more information about `xv_input_readevent()`.

Example 15-4. The fullscreen.c program

```
/*
 * fullscreen.c
 * Demonstrate the fullscreen package.  Create a panel button that
 * creates a fullscreen instance, thus grabbing the X server.  User
 * presses a mouse button to release the server.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/fullscreen.h>

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
    Panel       panel;
    void        exit(), grab();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Fullscreen",
        PANEL_NOTIFY_PROC,      grab,
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * Notify procedure for when the "Fullscreen" button is pushed.
 * Create a fullscreen instance, scan for a button event, then
 * destroy it.
 */
void
grab(item, event)
Panel_item item;
Event *event;
{
    Panel       panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);
    Frame       frame = (Frame)xv_get(panel, XV_OWNER);
    Fullscreen  fs;
    Inputmask   im;

    /* set up an input mask for the call to xv_input_readevent(). */
    win_setinputcodebit(&im, MS_LEFT);
    win_setinputcodebit(&im, MS_MIDDLE);
    win_setinputcodebit(&im, MS_RIGHT);
    win_setinputcodebit(&im, LOC_MOVE);
```

Example 15-4. The fullscreen.c program (continued)

```
    /*
     * Create a fullscreen object (initialize X server grab).
     * Specify which events should be allowed to pass through.
     * These events should match the input mask coded above.
     */
    fs = xv_create(panel, FULLSCREEN,
        WIN_CONSUME_EVENTS,
            WIN_MOUSE_BUTTONS, LOC_MOVE, NULL,
        NULL);

    /* Loop till user generates a button event */
    while (xv_input_readevent(panel, event, TRUE, TRUE, &im) != -1)
        if (event_is_button(event))
            break;

    /* Destroy the fullscreen (release the X server grab) */
    xv_destroy(fs);

    /* Report which button was pushed. */
    printf("event was button %d (%d, %d)0,
        event_id(event) - BUT_FIRST+1,
        event_x(event) + (int)xv_get(frame, XV_X),
        event_y(event) + (int)xv_get(frame, XV_Y));
}
```

When this program is run and the user selects the `Fullscreen` panel item, the X server is grabbed and the user must select one of the mouse buttons to release it. To users, it may appear as though they can select another panel button. Although the `panel` window is the owner of the fullscreen object, events that occur while the server is grabbed by the fullscreen object are *not* propagated to XView objects under the pointer. In Example 15-4, if the user presses the mouse button when the pointer is on top of any panel button a panel button while in fullscreen, the button-*down* event will trigger the call to `xv_input_readevent()` and break the loop. The corresponding button-up event is not read yet and will get read by normal event processing after the call to `grab()` returns. If the button-up event happened over a panel button, then the panel button's notify routine will be called.

The event masks set by the `FULLSCREEN` package and by the `Inputmask` do not interfere with the event masks in any XView window.

15.5.0.1 Debugging and the FULLSCREEN package

There are four global variables in the `FULLSCREEN` package that can be used to help debug XView programs that grab the server, keyboard or pointer. Note that these variables can only be used via the `FULLSCREEN` package. Here are the variables with their default values:

```
int fullscreendebug = 0;
int fullscreengrabserver = 1;
int fullscreengrabpointer = 1;
int fullscreengrabkbd = 1;
```

When `fullscreengrabserver` is set to 0 (in source code or in debugger), the X server will *not* be grabbed despite requests to grab it.

When `fullscreengrabpointer` is set to 0, the pointer will *not* be grabbed despite requests to grab it.

When `fullscreengrabkbd` is set to 0, the keyboard will *not* be grabbed despite requests to grab it.

When `fullscreendebug` is set to 1, no grabs of any kind are performed.

15.6 Nonvisual Package Summary

There are procedures or macros in the nonvisual packages. Table 15-1 lists the attributes in the `SCREEN` package. Table 15-2 lists the attributes in the `SERVER` and `SERVERIMAGE` packages and Table 15-3 lists the attributes in the `FULLSCREEN` package. This information is described fully in the *XView Reference Manual*.

Table 15-1. Screen Attributes

SCREEN_NUMBER
SCREEN_SERVER
XV_ROOT

Table 15-2. Server and Server Image Attributes

SERVER_ATOM	SERVER_EXTERNAL_XEVENT_PROC
SERVER_ATOM_NAME	SERVER_NTH_SCREEN
SERVER_EXTENSION_PROC	SERVER_SYNC
SERVER_EXTERNAL_XEVENT_MASK	SERVER_SYNC_AND_PROCESS_EVENTS
XV_DISPLAY	XV_NAME

Table 15-3. Fullscreen Attributes

FULLSCREEN_ALLOW_EVENTS	FULLSCREEN_KEYBOARD_GRAB_KBD_MODE
FULLSCREEN_ALLOW_SYNC_EVENT	FULLSCREEN_KEYBOARD_GRAB_PTR_MODE
FULLSCREEN_COLORMAP_WINDOW	FULLSCREEN_OWNER_EVENTS
FULLSCREEN_CURSOR_WINDOW	FULLSCREEN_PAINT_WINDOW
FULLSCREEN_GRAB_KEYBOARD	FULLSCREEN_POINTER_GRAB_KBD_MODE
FULLSCREEN_GRAB_POINTER	FULLSCREEN_POINTER_GRAB_PTR_MODE
FULLSCREEN_GRAB_SERVER	FULLSCREEN_RECT
FULLSCREEN_INPUT_WINDOW	FULLSCREEN_SYNC

Table 15-3. Fullscreen Attributes (continued)

`WIN_CONSUME_EVENT`	`WIN_IGNORE_EVENT`
`WIN_CONSUME_EVENTS`	`WIN_IGNORE_EVENTS`
`WIN_CURSOR`	`WIN_INPUT_MASK`

This page intentionally left blank

to preserve original page counts.

16
Fonts

In X, a large number of fonts are provided on the server. Deciding which font to use and then trying to specify fonts by name can be difficult since there are many different styles and sizes of fonts. Most fonts are used to render text strings. The images, or glyphs, represent a character set defined mostly by the language used. However, a font may be built to support glyphs that have nothing to do with a language. Fonts are stored on the server and are associated with the display of your workstation. The *font ID* is stored in the graphics context (`GC`), which is used by Xlib functions like `XDrawString()`. Using fonts to render text is perhaps the most common application. For example, the `Courier` font family displays the classic typewriter or constant-width character set. This text is set in Times-Roman, a proportionally spaced font. Often within a font family, there are different styles, such as **bold** or *italic*, and different point sizes.* For example, `lucidasans-bold-14` refers to the lucidasans font family, the style is bold, and the point size is 14.

Not all server fonts have a variety of styles and sizes. These special-purpose fonts are generally specified by name only—there are no corresponding styles or families for these fonts.

When accessing fonts, you typically want to specify a font either by *name* or by the *family*, *style*, and *size* or *scale* of the font. In addition, XView provides an interface for determining the dimensions (in pixels) of characters and strings rendered in a specified font.

OPEN LOOK uses predefined fonts for certain items such as panel buttons and other user interface elements. These items cannot be changed, but you can assign text fonts to panel choices, text subwindows and other types of windows. We will address these issues later in this chapter.

*Note that point sizes on workstations are based on pixels, whereas point sizes for typesetters and printers are based on inches.

16.1 Creating Fonts

Applications that use the FONT package must include the header file, <*xview/font.h*>. In XView, when a font object is created, it loads the font from the X server. When we say, “create a font,” we really mean, “load a font from the server and create an XView font object associated with that font.” Figure 16-1 shows the class hierarchy for the font package.

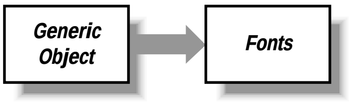


Figure 16-1. Font package class hierarchy

While fonts can be created using `xv_create()`, it may not be necessary to create a new instance of a font. Fonts are typically cached on the server, and XView may already have a handle to a particular font. Therefore, you should obtain a handle to the font if it already exists, rather than create another instance of the same font. `xv_find()` can be used to return the handle of an existing font. If the handle does not exist, `xv_find()` can create a new instance of the font.

Both `xv_find()` and `xv_create()` will return an object of the type `Xv_Font` when using the FONT package. The form of the call is:

```
Xv_Font font;
font = (Xv_Font) xv_create(owner, FONT, attrs, NULL);
```

or:

```
Xv_Font font;
font = (Xv_Font) xv_find(owner, FONT, attrs, NULL);
```

The *owner* of the font is usually the window in which the font is going to be used. The actual X font is loaded from the server associated with the owner object. If the owner is NULL, the default server is used. Fonts may be used on any window, memory pixmaps, or `Server_image`, but these objects must have the same `display` associated with them as the font, or you will get an X Protocol error. What this means is that a font can only be used on the server on which it was created. This is only any issue if your XView application is running on multiple servers at the same time. If the parent is NULL, the default server is used. Otherwise, the server as determined from the parent object is used. This is only an issue if your XView application is running on several servers at the same time.

Once a font object is created, it can be used to render text by assigning the font’s XV_XID to the `font` field of a graphics context (GC) and then using any of the Xlib routines that use fonts such as `XDrawString()`. Example 16-1 lists *simple_font.c*, a program that builds a simple frame and canvas. The repaint routine for the canvas displays the string “Hello World” at the upper-left corner of the window.

Example 16-1. The simple_font.c program

```
/*
 * simple_font.c -- very simple program showing how to render text
 * using a font gotten from xv_find().  Hello World is printed in
 * the upper-left corner of a canvas window.
 */
#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>   /* X.h and Xlib.h used for Xlib graphics */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/font.h>
#include <xview/xv_xrect.h>

#define GC_KEY  10 /* any arbitrary number -- used for XV_KEY_DATA */

main(argc, argv)
int     argc;
char    *argv[];
{
    Frame       frame;
    Canvas      canvas;
    XGCValues   gcvalues;
    Xv_Font     font;
    void        my_repaint_proc();
    Display     *dpy;
    GC          gc;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               400,
        XV_HEIGHT,              200,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    my_repaint_proc,
        NULL);
    window_fit(frame);

    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    font = (Xv_Font)xv_find(frame, FONT,
                            FONT_NAME, "lucidasans-12", NULL);
    if (!font) {
        fprintf(stderr, "%s: \
                cannot use font: lucidasans-12\n", argv[0]);
        font = (Xv_Font)xv_get(frame, XV_FONT);
    }

    /* Create a GC to use with Xlib graphics -- set the fg/bg colors
     * and set the Font, which is the XV_XID of the XView font object.
     */
    gcvalues.font = (Font)xv_get(font, XV_XID);
    gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
    gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
    gcvalues.graphics_exposures = False;
    gc = XCreateGC(dpy, RootWindow(dpy, DefaultScreen(dpy)),
```

Example 16-1. The simple_font.c program (continued)

```
        GCForeground | GCBackground | GCFont | GCGraphicsExposures,
        &gcvalues);

    /* Assign the gc to the canvas object so we can use the same
     * gc each time we draw into the canvas.  Also avoids a global
     * variable to store the GC.
     */
    xv_set(canvas, XV_KEY_DATA, GC_KEY, gc, NULL);
    xv_main_loop(frame);
}

/*
 * Called every time the window needs repainting.
 */
void
my_repaint_proc(canvas, pw, dpy, xwin, xrects)
Canvas          canvas;
Xv_Window       pw;
Display         *dpy;
Window          xwin;
Xv_xrectlist    *xrects;
{
    GC gc = (GC)xv_get(canvas, XV_KEY_DATA, GC_KEY);

    XDrawString(dpy, xwin, gc, 10, 20,
        "Hello World", 11); /* 11 = strlen("Hello World") */
}
```

The program attempts to create the font named "lucidasans-12." If the font is not found, the frame's font is used as a backup. This font must exist, so there is no need to check for a failed return value.

Since the text is rendered using Xlib graphics, we need to use a GC that has the right attributes set: the foreground and background colors and a font. Because this GC is specifically used for the canvas window, we are going to *attach* the GC to the canvas by using the generic attribute XV_KEY_DATA. Using a unique key, GC_KEY (which can be any integer since no other keys have been assigned to the object yet), the GC is attached with the call:

```
xv_set(canvas, XV_KEY_DATA, GC_KEY, gc, NULL);
```

Later, in my_repaint_proc(), the GC is retrieved:

```
GC gc = (GC)xv_get(canvas, XV_KEY_DATA, GC_KEY);
```

This method of storing the GC by using XV_KEY_DATA avoids the need for a global variable.

When creating some fonts, it may take quite some time for the font to be found and completely loaded—especially large fonts, since they may be created at runtime. Loading the font may result in the user having to wait longer than expected. It is recommended that the application provide visual feedback if the user must wait for some time. Do this by setting the FRAME_BUSY attribute to TRUE for the parent frame:

```
xv_set(frame, FRAME_BUSY, TRUE, NULL);
font = (Xv_Font) xv_find(frame, FONT,
    FONT_NAME,  "lucidasans-24",
    NULL);
xv_set(frame, FRAME_BUSY, FALSE, NULL);
```

This code fragment attempts to create a 24-point size font. Note that not all servers can do so, either because of memory limitations or because the server cannot scale fonts at will. In this case, the font returned may be `NULL`.

16.1.1 Font Families and Styles

One way to create fonts is to specify a font family, style, and size. The family of a font describes its basic characteristics. Figure 16-2 shows the Courier family in different styles and a range of point sizes.

Courier plain
10 12 14 16 18

Courier Bold
10 12 14 16 18

Courier italic
10 12 14 16 18

Figure 16-2. The Courier font in different styles and sizes

Some font families and styles known to XView are predefined in *<xview/font.h>*. To use a font other than the ones listed, you may specify any font family and style known by your X server. You may also specify fonts by *name* (see Section 16.1.4, “Fonts by Name”). The list of XView font families include:

- `FONT_FAMILY_DEFAULT`
- `FONT_FAMILY_DEFAULT_FIXEDWIDTH`
- `FONT_FAMILY_LUCIDA`
- `FONT_FAMILY_LUCIDA_FIXEDWIDTH`
- `FONT_FAMILY_ROMAN`
- `FONT_FAMILY_SERIF`
- `FONT_FAMILY_CMR`
- `FONT_FAMILY_GALLENT`
- `FONT_FAMILY_HELVETICA`
- `FONT_FAMILY_OLGLYPH`*
- `FONT_FAMILY_OLCURSOR`*

*The families `FONT_FAMILY_OLGLYPH` and `FONT_FAMILY_OLCURSOR` are used internally by XView packages. They are not for general/public use.

The family FONT_FAMILY_DEFAULT is the default font for XView. The FONT_FAMILY_DEFAULT_ FIXEDWIDTH font is the default fixed-width font.

All the characters in fixed-width fonts occupy the same amount of space. Other fonts are proportionally spaced; that is, each character may occupy a different amount of space.

The available styles are:

- FONT_STYLE_DEFAULT
- FONT_STYLE_NORMAL
- FONT_STYLE_BOLD
- FONT_STYLE_ITALIC
- FONT_STYLE_OBLIQUE
- FONT_STYLE_BOLD_ITALIC
- FONT_STYLE_BOLD_OBLIQUE

The default style indicates the default font's type for XView.

The call to xv_find() in *simple_font.c* could have been written to specify the family and style of the font rather than the name of the font:

```
font = (Xv_Font)xv_find(frame, FONT,
    FONT_FAMILY,    FONT_FAMILY_LUCIDA,
    FONT_STYLE,     FONT_STYLE_NORMAL,
    NULL);
```

Since normal is the default style of the font, this example renders the same font as in the earlier example. However, we could specify a different style:

```
font = (Xv_Font)xv_find(frame, FONT,
    FONT_FAMILY,    FONT_FAMILY_LUCIDA,
    FONT_STYLE,     FONT_STYLE_BOLD,
    NULL);
```

This call returns a bold style of the lucidasans font. For most font families, you can specify a font family with any style, although some families may not support an italic or bold style of the font. Therefore, you should be prepared to handle a NULL return from the call to xv_create() or xv_find():

```
if (!(font = (Xv_Font)xv_find(canvas, FONT,
    FONT_FAMILY,   FONT_FAMILY_COUR,
    FONT_STYLE,    FONT_STYLE_ITALIC,
    NULL))) {
    /* Handle the case where the font fails. */
    font = (Xv_Font)xv_get(canvas, XV_FONT);
}
```

16.1.2 Font Sizes

Fonts can be specified in any size from one point on up, as long as the bitmapped font of that size exists on the server. If the server supports scalable fonts, it can scale the font to the specified size, depending on the amount of available memory on the server. When a size is specified, provided the font is not already loaded, a new font is created at run time according to the family and style in the size specified, as below:

```
Xv_Font font;

font = (Xv_Font)xv_find(canvas, FONT,
    FONT_FAMILY,    FONT_FAMILY_ROMAN,
    FONT_STYLE,     FONT_STYLE_BOLD,
    FONT_SIZE,      36,
    NULL);
```

This code fragment attempts to find or create a font from the Times-Roman font family in bold and in 36-point. The font may already exist in the server, or the font's family may exist but not the size. In the latter case, the server must attempt to scale the font to the specified size. If the font cannot be created because of memory limitations, the call to `xv_find()` will return `NULL`. Because a 36-point font takes a lot of memory, it might take a while to load this font.

16.1.3 Scaling Fonts

Bitmapped fonts are not scalable, but they are provided in so many sizes that they appear to be scalable to any size. On the other hand, some fonts are scalable because they are not stored as static bitmaps. However, in order to scale even these fonts, the server must support font scaling. Sun's *xnews* server is an example of such a server.

You can request a relative scale of a font with respect to other fonts within the same family. The relative scales are *small*, *medium*, *large* and *extra large*. These scales are represented by the attributes `WIN_SCALE_SMALL`, `WIN_SCALE_MEDIUM`, `WIN_SCALE_LARGE`, and `WIN_SCALE_EXTRALARGE`.

These attributes are members of the enumerated type `Window_rescale_state`. They are `WINDOW` attributes because fonts can be scaled in proportion to their windows. For example, if your application is resized to a larger or smaller size, you may wish to reset the fonts for some windows according to different scaling sizes. By default, the sizes of the fonts corresponding to the scale as shown in Table 16-1.

Table 16-1. Default Font Sizes

Attribute	Font Size
WIN_SCALE_SMALL	10
WIN_SCALE_MEDIUM	12
WIN_SCALE_LARGE	14
WIN_SCALE_EXTRALARGE	19

If FONT_SIZE is not specified when a font is requested, the size will correspond to the *medium* scale size. Specifying the font size overrides the request for a scaling factor. The code fragment below requests a font from the Lucida family with an italic style in large scale:

```
Xv_Font   font;

font = (Xv_Font)xv_find(canvas, FONT,
    FONT_FAMILY,    FONT_FAMILY_LUCIDA,
    FONT_STYLE,     FONT_STYLE_ITALIC,
    FONT_SCALE,     WIN_SCALE_LARGE,
    NULL);
```

If you have already created a font and wish to get it in a different scale, the attribute FONT_RESCALE_OF is specified with with two arguments: a font and a Window_rescale_state.

```
Xv_Font  font, small_font;

font = (Xv_Font)xv_find(canvas, FONT,
    FONT_FAMILY,    FONT_FAMILY_LUCIDA,
    FONT_STYLE,     FONT_STYLE_BOLD,
    NULL);

. . .

small_font = (Xv_Font)xv_find(canvas, FONT,
    FONT_RESCALE_OF,    font, WIN_SCALE_SMALL,
    NULL);
```

You can reset the sizes of a fonts' scale factors by using the attribute FONT_SIZES_FOR_SCALE. This attribute takes four *values* that correspond to each scaling factor:

```
font = (Xv_Font)xv_find(canvas, FONT,
    FONT_FAMILY,           FONT_LUCIDA,
    FONT_STYLE,            FONT_STYLE_NORMAL,
    FONT_SIZES_FOR_SCALE,  12, 14, 16, 22,
    NULL);
```

In this example, the Lucida font is created so that the small, medium, large, and extra-large scaling sizes are 12, 14, 16, and 22, respectively. The font size returned is 14-point.

16.1.4 Fonts by Name

When specifying a font by name, the entire name of the font is given as it appears in the output of a program such as *xlsfonts* or by the Xlib call `XListFonts()`.* These programs provide a complete list of all the fonts available on the server.

In Example 16-1 above, the program *simple_font.c* showed how to load a font using a given name:

```
font = (Xv_Font) xv_find(canvas,
                         FONT, FONT_NAME, "lucidasans-12", NULL);
```

When specifying fonts by name, other attributes normally used to specify family, style, and size or scale are ignored in favor of the font name. However, it is possible to determine the family, style, and scale of a font by looking at its name. For example, with names such as those shown below, this is easy.

```
lucidasans-12
lucidasans-bold-12
```

The X Logical Font Description (XLFD) Conventions, which defines a standard way of naming a font, is a X Consortium standard. The font names are constructed using this convention are unique, and descriptive (the name contains most of the information about the font.) For details on these conventions, refer to Volume Zero, *X Protocol Reference Manual*. Below are some sample XLFD font names:

```
-adobe-courier-bold-o-normal--10-100-75-75-m-60-iso8859-1
-adobe-courier-medium-r-normal--12-120-75-75-m-70-iso8859-1
```

When attributes other than `FONT_NAME` are used, XView will try to construct a font name using the XLFD Conventions based on the information passed (family, style, size) and dependent on the fonts available on the server. The constructed name will be returned when `xv_get()` is used with `FONT_NAME`.

Likewise, if `FONT_NAME` is used, XView will load that font, and will try to decrypt the information as stored in the name to fill in information for `FONT_FAMILY`, `FONT_STYE`, and so on.

16.2 Font Dimensions

Once a font is created, it can be used in applications using Xlib routines, as demonstrated in *simple_font.c* earlier in this chapter. In the program, text is rendered using the Xlib routine `XDrawString()`. This routine uses the font ID from the `GC` parameter. This ID is extracted from the font using `xv_get()` and `XV_XID`. You can also get a pointer to the font's `XFontStruct` structure by specifying the attribute `FONT_INFO` to `xv_get()`. This data structure is what describes the characteristics of the font such as width and height for each character. See Volume One, *Xlib Programming Manual*, for more information.

*See Volume Two, *Xlib Reference Manual*, for a description of `XListFonts()`.

You can get information about the sizes of individual characters or the dimensions of entire strings of characters in a particular font using several methods. You can use the information in the XFontStruct data structure obtained from the font, or you can use Xlib routines such as XTextWidth() or XTextExtents(), or you can use xv_get() along with XView attributes such as FONT_CHAR_WIDTH, FONT_CHAR_HEIGHT, FONT_DEFAULT_CHAR_HEIGHT, FONT_DEFAULT_CHAR_WIDTH, and FONT_STRING_DIMS.

The usage is as follows:

```
Xv_Font  font;
int      width, height;
```

The following code shows how to get the dimensions of a particular character in a font:

```
width  = (int)xv_get(font, FONT_CHAR_WIDTH, 'm');
height = (int)xv_get(font, FONT_CHAR_HEIGHT, 'm');
```

The calls to xv_get() return the width and height of the characters in pixels for that particular font. If you are using a fixed-width font, then each character will be the same width, so you can specify the default character width and height of the font. The following code shows how to get the dimensions of characters for a fixed-width font:

```
Xv_Font font;
int width, height;

width  = (int)xv_get(font, FONT_DEFAULT_CHAR_WIDTH);
height = (int)xv_get(font, FONT_DEFAULT_CHAR_HEIGHT);
```

If you use FONT_DEFAULT_CHAR_WIDTH (or height) on a variable-width font, you will get the *average* width of a character in that font.

To get the width and height dimensions of a complete string of text in a given font, use the FONT_STRING_DIMS attribute:

```
extern Xv_Font    font;
Font_string_dims  dims;

(void) xv_get(font, FONT_STRING_DIMS, "Hello World", &dims);
```

In this case, the call to xv_get() returns a pointer to the dims structure passed as the last argument. The return value may be ignored since the dims parameter will have the value filled in upon return of xv_get(). The Font_string_dims data structure is as follows:

```
typedef struct {
    int  width;
    int  height;
} Font_string_dims;
```

Thus, xv_get() returns the dimensions of the string for the font specified. This would be equivalent to calling XTextExtents() in the manner below:

```
Font_string_dims dims;
extern char   *str;
int            len = strlen(str);
extern Xv_Font font;
XFontStruct   *font_info = (XFontStruct *)xv_get(font, FONT_INFO);
int            direction, ascent, descent;
XCharStruct    overall_return;

(void) XTextExtents(font_info, str, len,
            &direction, &ascent, &descent, &overall_return);
dims.width = overall_return.width;
dims.height = ascent + descent;
```

16.3 Font Package Summary

There are no procedures or macros in the FONT package. The font attributes are shown in Table 16-2. They are described fully in the *XView Reference Manual*.

Table 16-2. Font Attributes

FONT_CHAR_HEIGHT	FONT_RESCALE_OF
FONT_CHAR_WIDTH	FONT_SCALE
FONT_DEFAULT_CHAR_HEIGHT	FONT_SIZE
FONT_DEFAULT_CHAR_WIDTH	FONT_SIZES_FOR_SCALE
FONT_INFO	FONT_STRING_DIMS
FONT_FAMILY	FONT_STYLE
FONT_NAME	
XV_XID	

17
Resources

In the X Window System, the user can configure the interface according to options available in specific applications. The user accomplishes this through a *resource database* that resides in the X server. The X Protocol provides many ways to access the resource database, as well as many functions to aid in this task. You should consult Chapter 11, *Managing User Preferences*, in Volume One, *Xlib Programming Manual*, for a complete, in-depth discussion of X resource specification and management. Related programs include *xrdb*, and related functions can be found in Volume Two, *Xlib Reference Manual*.

XView provides many functions that allow the programmer to interact with the server to get or set resources specified by the user. Robust applications should account for user-definable defaults. That is, your programs should always consider the user's wishes for changing attribute values for things like fonts, colors and maybe even window sizes, as long as the values do not interfere with the normal running of the program. You should also provide the user with a list of resources that can be set, and you should test for them in your application.

17.1 Predefined Defaults

All of the packages in XView look for predefined defaults that the user can set in his/her resource environment. Table 17-1 outlines these defaults, their types and legal values. Refer to Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual* for a description of each of the XView resources. Note that the term *maxint* refers to the maximum value for an integer on your particular machine. These values tend to be rather large with respect to the intended values used by these resources. It is up to the user to use "reasonable" values.

NOTE

Table 17-1 does not include the mouseless model resources. Refer to Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual* for a list of the mouseless resources.

Table 17-1. *Resources and Default Values Understood by XView*

Name	Type	Default	Legal Values
alarm.audible	**boolean**	True	True, False
alarm.visible	**boolean**	True	True, False
cmdtool.checkpointFrequency	**integer**	0	0, <*maxint*>
cmdtool.maxLogFileSize	**integer**	<*maxint*>	0, <*maxint*>
font.name	**string**	NULL	*fontname-size*
icon.font.name	**string**	NULL	*fontname-size*
icon.pixmap	**string**	NULL	*pixmap filename*
icon.footer	**string**	NULL	*footer string*
icon.x	**integer**	0	—
icon.y	**integer**	0	—
keyboard.arrowKeys	**string**	Yes	Yes, No
keyboard.leftHanded	**string**	No	Yes, No
mouse.multiclick.space	**integer**		
notice.PopupJumpCursor	**boolean**	True	True, False
notice.beepCount	**integer**	1	—
openWindows.DragRightDistance	**integer**	100	0, <maxint>
openWindows.3DLook.color	**boolean**	True	True, False
openWindows.3DLook.monochrome	**boolean**	True	True, False
openWindows.multiClickTimeout	**integer**	4	1 through 10 (in sec/10)
openWindows.dragRightDistance	**integer**	5	—
openWindows.scrollbarPlacement	**string**	right	Left, Right
OpenWindows.MonospaceFont	**string**	NULL	fontname-size
OpenWindows.RegularFont	**string**	NULL	fontname-size
OpenWindows.BoldFont	**string**	NULL	fontname-size
OpenWindows.Scale	**string**	NULL	Small, Medium, Large, Extra-Large
scrollbar.repeatDelay	**integer**	100	0, 999 (msec delay interval)
scrollbar.pageInterval	**integer**	100	0, 999 (msec delay interval)
scrollbar.lineInterval	**integer**	1	0, 999 (msec delay interval)
server.name	**string**	getenv(DISPLAY)	*hostnamedisplay*
term.boldStyle	**string**	Invert	xview.ICCCMCompli-ant@boolean@True@True, False None, OFFSET_X, OFFSET_Y, OFFSET_X_AND_Y, OFFSET_X_AND_XY, OFFSET_Y_AND_XY, OFFSET_X_AND_Y_AND_XY, OFFSET_XY, INVERT
term.enableEdit	**boolean**	True	True, False
term.inverseStyle	**string**	Enable	ENABLED, DISABLED, SAME_AS_BOLD
term.underlineStyle	**string**	Enable	ENABLED, DISABLED, SAME_AS_BOLD
text.againLimit	**integer**	1	0, 500
text.autoIndent	**boolean**	False	True, False
text.autoScrollBy	**integer**	1	0, 100

Table 17-1. Resources and Default Values Understood by XView (continued)

Name	Type	Default	Legal Values
text.blinkcaret	**boolean**	True	True, False
text.checkpointFrequency	**integer**	0	0, <*maxint*>
text.confirmOverwrite	**boolean**	True	True, False
text.displayControlChars	**boolean**	False	True, False
text.enableScrollbar	**boolean**	True	True, False
text.extrasMenuFilename	**string**	*.text_extra_menu* (in */usr/lib*)	*filename*
text.insertMakesCaretVisible	**string**	If_auto_scroll	If_auto_scroll, Always
text.lineBreak	**string**	Wrap_word	Clip, Wrap_char, Wrap_word
text.margin.bottom	**integer**	0	–1, 50
text.margin.left	**integer**	8	0, 2000
text.margin.right	**integer**	0	0, 2000
text.margin.top	**integer**	2	–1, 50
text.maxDocumentSize	**integer**	20000	0, 0x80000000
text.retained	**boolean**	False	True, False
text.storeChangesFile	**boolean**	True	True, False
text.tabWidth	**integer**	8	0, 50
text.undoLimit	**integer**	50	0, 500
window.color.foreground	**string**	"0 0 0"	3 RGB values, 0-255
window.color.background	**string**	"255 255 255"	3 RGB values, 0-255
window.columns	**integer**	80	—
window.height	**integer**	34 rows	Greater than 0
window.header	**string**	NULL	*header string*
window.iconic	**boolean**	False	True, False
window.inheritcolor	**integer**	1	
window.mono.disableRetained	**boolean**	False	True, False
window.rows	**integer**	34	—
window.scale	**string**	Medium	Small, Medium, Large, Extra_Large
window.width	**integer**	80 columns	Greater than 0
window.x	**integer**	0	—
window.y	**integer**	0	—

Note: In XView 3.2 and later, the resource `Font.Name` will still be read by any XView application, however, if `OpenWindows.MonospaceFont`, `OpenWindows.RegularFont` or `OpenWindows.BoldFont` exist in the X Resource Database, they will take precedence.

XView 3.2 and later uses the new resource `OpenWindows.Scale`. In earlier releases of XView, "scale" was determined by reading the resource "Window.Scale." The new resource is synonymous with the old resource name and XView will continue to look at the old resource "Window.Scale" if the new resource, `OpenWindows.Scale`, does not exist. XView provides many functions to get resource values from the database. Basically, values come in several types: `int`, `character` (strings), `boolean` (which can be specified using a number of string values; see below), and `enumerated` values.

Example 17-1 lists a simple application that introduces the use of one of the functions provided by the `defaults` package. The program *default_text.c* creates a frame with a text subwindow whose font is specified by the user's resource `textsw.font`.

Example 17-1. The default_text.c program

```
/*
 * default_text.c -- use the defaults package to get a font name from
 * the resource database to set the textsw's font.
 */
#include <xview/xview.h>
#include <xview/font.h>
#include <xview/defaults.h>
#include <xview/textsw.h>

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
    Xv_Font     font;
    char        *name;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);

    name = defaults_get_string("textsw.font","Textsw.Font", "fixed"),
    font = xv_find(frame, FONT,
        FONT_NAME,      name,
        NULL);

    xv_create(frame, TEXTSW,
        XV_FONT,        font,
        WIN_COLUMNS,    80,
        WIN_ROWS,       10,
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

default_text.c shows how an XView program queries the resource database for a string value associated with a name-class pair. `defaults_get_string()` gets the string value associated with `Textsw.Font`. If the resource database has this attribute, then the value is returned. If not, the default (`fixed`) is returned.

17.2 XView Resource Database Functions

The functions that XView provides are defined in <*xview/defaults.h*>. The following functions are provided by XView for setting and getting resource values to and from the database. Note that when setting resources *to* the database, the database on the *server* is updated—not the user's *defaults* database. Resources not updated to the user's defaults database are not retained for the next time the X server is started.

```
void
defaults_init_db()
```

This function is called automatically by `xv_init()`, so it need not be called by your application. `defaults_init_db()` calls `XrmInitialize()`.

```
void
defaults_load_db(filename)
    char *filename;
```

`defaults_load_db()` loads the database residing in the specified filename or the server database if filename is `NULL`. The database found in `filename` is loaded via `XrmGetFileDatabase()` and is merged into the existing resource database via `XrmMergeDatabases()`.

```
void
defaults_store_db(filename)
    char *filename;
```

This function writes the database to the specified file via `XrmPutFileDatabase()`. This must be done in order to ensure that the database is accessible the next time the server is started.

```
Bool
defaults_exists(name, class)
    char *name;
    char *class;
```

This function returns `TRUE` if the resource exists in the database via `XrmGetResource()`.

For more information, refer to Volume One, Chapter 11, *Managing User Preferences.*

17.2.1 Boolean Resources

```
Bool
defaults_get_boolean(name, class, default_value)
    char *name, *class;
    int   default_value;
```

```
void
defaults_set_boolean(resource, value)
    char *resource;
    Bool  value;
```

`defaults_get_boolean()` looks up the name-class pair in the resource database and returns `TRUE` if the value is one of the following:

- `True`
- `Yes`
- `On`
- `Enabled`
- `Set`
- `Activated`
- `1`

It returns `FALSE` if the value is one of the following:

- `False`
- `No`
- `Off`
- `Disabled`
- `Reset`
- `Cleared`
- `Deactivated`
- `0`

If the value is none of the above, a warning message will be displayed and the default value will be returned. If the resource is not found, no error message is printed but the default value is still returned.

`defaults_set_boolean()` sets the resource to the value specified.

17.2.2 Integer Resources

```
int
defaults_get_integer(name, class, default_value)
    char *name;
    char *class;
    int   default_value;

int
defaults_get_integer_check(name, class, default_value,
                           minimum, maximum)
    char *name;
    char *class;
    int   default_value;
    int   minimum;
    int   maximum;
```

```
void
defaults_set_integer(resource, value)
    char *resource;
    int   value;
```

`defaults_get_integer()` looks up the name-class pair in the resource database and returns the resulting integer value. If the database does not contain the resource, the default value is returned.

`defaults_get_integer_check()` looks up the name-class pair in the resource database and returns the resulting integer value. If the value in the database is not between the values `minimum` and `maximum` (inclusive), an error message is printed and the default value is returned. If the resource is not found, no error message is printed but the default value is returned.

`defaults_set_integer()` sets the resource to the value specified.

17.2.3 Character Resources

```
char
defaults_get_character(name, class, default_char)
    char *name;
    char *class;
    char  default_char;

void
defaults_set_character(resource, character)
    char *resource;
    char  character;
```

`defaults_get_character()` looks up the name-class pair in the resource database and returns the resulting character value. If the resource is not found, then the default character value is returned.

`defaults_set_character()` sets the resource to the character value.

17.2.4 String Resources

```
char *
defaults_get_string(name, class, default_str)
    char *name;
    char *class;
    char *default_str;

void
defaults_set_string(resource, string)
    char *resource;
    char *string;
```

`defaults_get_string()` returns the string value associated with the specified name-class pair in the resource database. If the resource is not found, the default string value is returned. The procedure `defauts_get_string()` returns a pointer into a static buffer maintained by the defaults package. The application should not attempt to free this pointer.

This buffer will be overwritten by the next call to the defaults package, so the application should maintain a copy if necessary.

`defaults_set_string()` sets the resource to the specified string.

17.2.5 Enumerated Resources

Enumerated resources are those whose values are string values, but the legal values for the resource are restricted to a predefined list. For example, say you want to allow the user to specify the font scale for the font `lucidasans`. The legal `scale` values are: small, medium, large, and extra large. You could use `defaults_get_string()` and determine, using `strcmp()`, whether the value returned is one of the legal scale values. Or you could use `defaults_get_enum()` and pass in a table describing the legal values. The *table* is an array of elements of the type:

```
typedef struct _default_pairs {
    char   *name;  /* Name of pair */
    int     value; /* Value of pair */
} Defaults_pairs;
```

The *name* is a string, and the *value* is the returned value associated with the name. This value may be your own value and need not be sequential, but it must be an `int`.

```
int
defaults_get_enum(name, class, pairs)
    char           *name;
    char           *class;
    Defaults_pairs *pairs;
```

`defaults_get_enum()` looks up the value associated with name and class and scans the `pairs` table and returns the associated value. If no match is found, an error is generated and the value associated with the last entry is returned. `defaults_get_enum()` calls `defaults_get_string()` and determines the value returned by calling `defaults_lookup()` (below), passing the returned string as the `name` parameter.

```
int
defaults_lookup(name, pairs)
    char           *name;
    Defaults_pairs *pairs;
```

`defaults_lookup()` linearly scans the `pairs` array looking for `name`. The value associated with `name` is returned. The `pairs` array *must* contain a last element with a `NULL` `name` and a legal `value` associated with it. This value is returned if `name` does not match the `name` field of any of the elements in the `pairs` parameter.

Example 17-2 shows a program that implements the idea discussed earlier for allowing the user to specify a scale for a font.

Example 17-2. The default_size.c program

```
/*
 * default_scale.c -- demonstrate the use of defaults_get_enum().
 * Specify a table of font scales and query the resource database
```

Example 17-2. The default_size.c program (continued)

```
 * for legal values.  For example, you may have the following in
 * your .Xdefaults (which must be loaded into the resource database):
 *      font.scale: large
 */
#include <xview/xview.h>
#include <xview/font.h>
#include <xview/defaults.h>
#include <xview/textsw.h>

Defaults_pairs size_pairs[ ] = {
    "small",            WIN_SCALE_SMALL,
    "medium",           WIN_SCALE_MEDIUM,
    "large",            WIN_SCALE_LARGE,
    "extralarge",       WIN_SCALE_EXTRALARGE,
    /* the NULL entry is the default if Resource not found */
    NULL,               WIN_SCALE_MEDIUM,
};

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
    Xv_Font     font;
    int         scale;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);

    scale = defaults_get_enum("font.scale", "Font.Scale", size_pairs);
    /* get the default font for the frame, scaled to resource */
    font = xv_find(frame, FONT,
        FONT_RESCALE_OF,        xv_find(frame, FONT, NULL), scale,
        NULL);

    xv_create(frame, TEXTSW,
        XV_FONT,                font,
        WIN_COLUMNS,            80,
        WIN_ROWS,               10,
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

In *default_size.c*, the `pairs` table describes an association between string constants and scaling factors. The resource database may specify the resource `font.scale` that describes a scaling factor. If the value in the resource is not one of the legal values included in the table, an error message is printed to inform the user that s/he has specified the resource incorrectly. In this case, or in the case where the resource is not specified, the default resource (associated with the `NULL` entry) is returned, namely, `WIN_SCALE_MEDIUM`.

NOTE

The `value` field in the `Defaults_pair` data structure is an `int`, so do not attempt to assign pointers, functions or attributes to this field. You can assign enumerated types since they are interpreted as `int`.

17.3 Creating Resource Instances

The attribute `XV_INSTANCE_NAME` is used to associate an instance name with an XView object. The instance name is used to construct the resource name used by the Resource Manager to perform lookups. The resource name is constructed by concatenating the instance names of all objects in the current object's lineage, starting with the name of the application or whatever was passed in with the *–name* command-line option, ending with the XView attribute name. The XView attribute name remains in lowercase. `XV_INSTANCE_NAME` is normally used with `XV_USE_DB`. Assume the name of an application is `app`:

```
Frame     frame;
Panel     panel;

frame = (Frame)xv_create(NULL, FRAME,
                        XV_INSTANCE_NAME, "base_frame",
                        NULL);

panel = (Panel)xv_create(frame, PANEL,
                        XV_INSTANCE_NAME, "panel",
                        XV_USE_DB,
                            XV_WIDTH, 100,
                            XV_HEIGHT, 200,
                            NULL,
                        NULL);
```

For this code fragment, assuming the name of the application is "app," the resource names constructed for lookup of the width and height of the panel are:

```
app.base_frame.panel.xv_width
app.base_frame.panel.xv_height
```

Entries in the Resource Manager could look like:

```
app.base_frame.panel.xv_width:400
app.base_frame.panel.xv_height:500
```

If these entries were not present in the Resource Manager, the width and height of the panel would take the default values of 100 and 200 respectively.

The attribute `XV_USE_DB` specifies a set of attributes that are to be searched in the X11 Resource Manager database. `XV_USE_DB` takes a `NULL`-terminated list of attribute value pairs as its values. During program execution, each attribute in this `NULL`-terminated list of attributes is looked up in the X11 Resource manager database. If the attribute is not found in the database, then the value specified in the attribute-value pair is used as the default value.

The list of customizable attributes can be found in the *XView Reference Manual*.

18 Selections

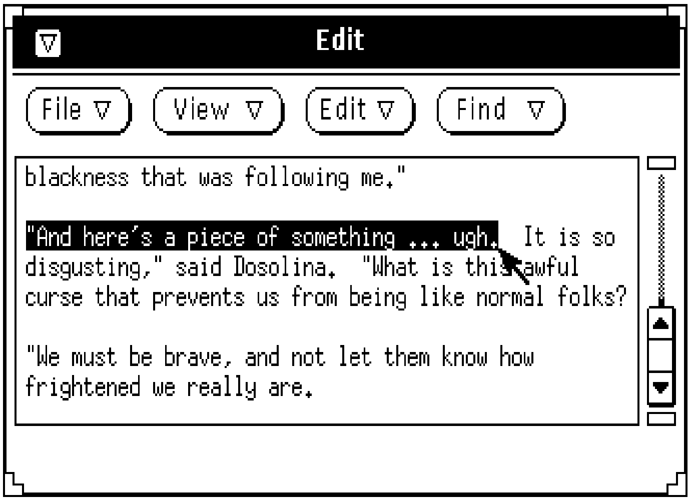
The X Window System provides several methods for separate applications to exchange information with one another. One of these methods is the use of selections. A *selection* transfers arbitrary information between two clients. An in-depth discussion of the selection mechanism that X provides is available in the *Xlib Programming Manual*. This chapter describes XView's new selection package. Previous revisions of XView provided a selection mechanism that did not use objects.

The selection mechanism for previous versions of XView used special functions and structures to implement selections. The older selection mechanism is still supported and it is described in Appendix A, *The Selection Service*. The new package implements selections as objects. The application programmer interface for the new selection package conforms to the standard XView model, using `xv_create()`, `xv_set()`, and `xv_get()`.

XView selections are used to exchange data between different applications or for communications within a single application. Many, but not all selections use text; a section of text is selected, then the selected text is cut or copied, and pasted to another area. The selection mechanism is not limited to text. For example, selections can be used to transfer filenames, sound data, a file's timestamp, or other information. Selections require only that the sender and the recipient have knowledge of the format of the data being transferred. Therefore, selections can be used to transfer graphics between applications that can understand a common format for communicating graphics. XView selections conform to the conventions in the ICCCM (Interclient Communications Conventions Manual). For a description of the ICCCM conventions, refer to Volume Zero, *X Protocol Reference Manual*.

In OPEN LOOK, you select objects in basically the same way that you select windows or icons —using the SELECT and ADJUST mouse buttons. Figure 18-1, from the *OPEN LOOK GUI Specification Guide*, shows one way to select text. OPEN LOOK describes three functions that operate on selected objects: CUT, COPY, and PASTE. These are *core functions* that are accessed from the keyboard.* Table 18-1 summarizes text selection for the OPEN LOOK GUI.

*The function keys that are bound to these functions vary from keyboard to keyboard depending on the make and model of the computer.



blackness that was following me."

"And here's a piece of something ... ugh. It is so disgusting," said Dosolina. "What is this awful curse that prevents us from being like normal folks?

"We must be brave, and not let them know how frightened we really are.

Figure 18-1. Dragging the pointer to select text

Table 18-1. Selecting Text

Action	Off Selection	On Selection
Click SELECT	Insert point is set at the pointer location.	When SELECT is released, insert point is set at pointer location and selection is cleared.
Drag SELECT	Text is highlighted as pointer is dragged (wipe-through selection).	Text move pointer is displayed. When you release SELECT, text is moved to pointer location if that location is outside the highlighted area.
Click ADJUST	Extends the highlighting to the pointer location extending either the beginning or the end of the current selection.	Moves the end of the highlighting to the pointer location. Beginning of selection is preserved.
Drag ADJUST	Adjusts an existing selection as the pointer is dragged (wipe-through). Beginning of selection is anchored at insert point.	Adjusts an existing selection as pointer is dragged (wipe-through). Beginning of selection is anchored at insert point.

If an application uses objects such as `TEXTSW` or `PANEL_TEXT_ITEM`, the CUT, COPY, and PASTE command keys are internally implemented. These selection functions are built-in for these objects.

Note that although OPEN LOOK specifies the selection of graphic objects, no XView objects currently support selection of graphical objects. A possible implementation is to have a canvas object, which has graphic objects displayed in it, set selections based on the event sequences outlined in Table 18-1. A *draw* application might consider a drawn geometric shape as a graphic object, whereas a *paint* application might consider the pixels in an arbitrary area as the graphical object.

18.1 The XView Selection Model

The XView selection model is based upon the requestor/owner model of peer-to-peer communications. Selections communicate data between an *owner client* and a *requestor client*. An owner client has the selection data and the requestor client wants that data. XView uses an object instantiated from the `SELECTION_OWNER` package to handle the selection owner, and an object instantiated from the `SELECTION_REQUESTOR` package to handle the selection requestor. Besides the owner and requestor objects, there is an XView hidden class, `SELECTION`, that handles information common to both a requestor and an owner. XView also provides an *optional* selection-item object, instantiated from the `SELECTION_ITEM` package, that can simplify the owner side of selections (to simplify this discussion, we do not cover the selection item until later in this chapter).

For XView to implement selections, XView tracks all X events that are generated from selections. These events are: `SEL_CLEAR`, `SEL_NOTIFY`, and `SEL_REQUEST` (X selection events are mapped to these XView events). Your application should ignore these events if you are using the selection package. If you want to implement selections with Xlib routines, then you can use these events; however, mixing is not allowed. That is, do not try to use both these events *and* use the XView the selection package.

The selection package allows you to select the *rank* of your selection. The rank is an atom representing a name on the server. The default rank is called the primary rank (`XA_PRIMARY`). For selections involving text, the *primary* rank is normally indicated on the screen by inverting (*highlighting*) its contents. Selections made while a function key is held down are considered *secondary* selections* (for text selections this is usually indicated with an underscore under the selection).

Other ranks are also available; OPEN LOOK uses the CLIPBOARD rank to hold data that has been cut or copied, using the cut or copy operations (with the CUT or COPY keys). This selection is inserted into an application with the paste operation (using the PASTE key). Atoms take up server resources, therefore the primary, secondary, and clipboard ranks are commonly used for selections; they are used over and over unless more than three selections need to be created. Other selections are used when the primary selection must be left undisturbed.

When a user interface element, such as a text subwindow wants to allow the user to make a selection, internally, it creates a *selection-owner* object. Through this selection-owner object, the text subwindow can *acquire* a selection rank (primary, secondary, etc) and provide it with data. An application can have many such selection owners, but there is typically one selection owner per XView object. Normally a selection rank is associated with an owner—a

*Which function key depends on your particular computer. By default the `L6` function key should work for Sun Workstations or the `F6` key for other computers.

separate owner need not be created just to utilize other selection ranks. However, only one owner can acquire a particular selection rank at any one time. If one application acquires a selection, and then another application acquires the same selection, the first application loses the selection ownership. The application that becomes the new selection owner may change the data associated with the selection.

Since the selection package conforms to the conventions of the ICCCM, selections between non-XView applications should be seamless.

18.2 How Selection Works (Without a Selection Item)

Here is a brief overview of the steps that take place during a single transfer of data from one XView application to another. We'll assume we have two applications, application *A* and application *B*, either of which can operate as the owner or the requestor.

Initially, both applications are in exactly the same state, neither has acquired any selections, and neither is the owner or the requestor. We'll also assume that selections are implemented using actions tied to mouse buttons or the CUT and PASTE function keys.

1. The user selects an item in application *A* and the application highlights the item. At this time, application *A* would acquire the primary selection.
2. In the XView Selection Owner package, nothing further happens until an `ACTION_COPY` event is delivered. At this time, application *A* acquires the selection and the primary selection is copied to the clipboard.
3. The user pastes the selected data into application *B*, causing an `ACTION_PASTE` to be sent to application *B*.
4. The `ACTION_PASTE` event may cause the selection requestor to request to receive the selected data into application *B*.
5. The selection data request invokes an application-defined conversion procedure which is associated with the Selection Owner in application *A*. It is the responsibility of the Selection Owner to convert the data into a format specified by application *B* or to reject the request.
6. Once the data conversion is finished, successfully or unsuccessfully, an application-defined reply procedure is invoked in application *B*. This procedure is associated with the selection requestor and either displays the data in application *B*, if the data conversion was successful, or indicates that the kind of data selected in application *A* cannot be pasted in application *B*, or that the kind of data requested by *B* cannot be supplied by *A*.
7. Once the selection data transfer is completed, a application-defined done procedure may be called by the selection owner. The done procedure may be used to free the memory associated with the selection, or perform other cleanup that is required.
8. If another application acquires the selection, application *A*'s "lose" procedure is called. This procedure is used to tell the selection owner (application *A*) that it has lost

ownership of the selection. For example, the lose procedure might unhighlight text that was previously selected and highlighted.

18.2.1 Highlighting the Selection (Selection Owner)

The first task for the application that is to become the selection owner is to mark the selection. For example, in a selection transfer involving text, when the pointer is placed over the text, an `ACTION_SELECT` should highlight the selection. While `LOC_DRAG` occurs, the highlight would be extended over the selection. If you are creating your own selections and you want your application to be OPEN LOOK compliant, you need to handle events and highlighting as specified in the *OPEN LOOK GUI Functional Specification*. Some XView packages, such as `TEXTSW` and `PANEL_TEXT_ITEM`, provide this functionality internally.

18.2.2 Making the Selection (Selection Owner)

After the user selects an item and it is highlighted, the application waits to receive an event of interest. For example, in a text subwindow the act of highlighting an item causes the primary selection to be acquired. At this time the XView application creates a selection owner object, if one was not previously created. For the text subwindow example, an `ACTION_COPY` event causes the CLIPBOARD selection to be acquired (note that in this case two selections are acquired since the COPY operation in OPEN LOOK uses the CLIPBOARD).

Applications perform three steps to become the selection owner:

1. A `Selection_owner` object is created using `xv_create()` (assuming a previously created selection owner is *not* being reused).
2. A mechanism for replying to the selection request should be set. This may be accomplished either by setting a conversion procedure or by creating a selection item (see Section 18.3.1, "The Selection Item"). The conversion procedure, or the selection item, allows the selection data to be associated with the selection owner and converted to the type a requestor expects.
3. The selection must be acquired.

A selection-owner object is created using `xv_create()`:

```
Selection_owner   sel_owner;
sel_owner = xv_create(window, SELECTION_OWNER,
                      NULL);
```

A selection-owner object's owner is a window or a window based object. The `SELECTION_OWNER` package is defined in the header file *<xview/sel_pkg.h>*. Programs that use a selection owner must include this file. Figure 18-2 shows the class hierarchy for the selection-owner object.

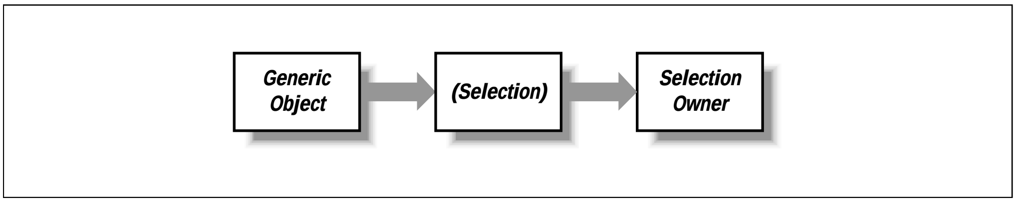


Figure 18-2. Selection owner class hierarchy

If a selection uses a conversion procedure to convert the selection, then the attribute `SEL_CONVERT_PROC` specifies the name of the conversion procedure. Section 18.2.4, "Converting the Selection," describes the conversion procedure. The selection is acquired by setting `SEL_OWN` to `TRUE`.

```
xv_set(sel_owner, SEL_CONVERT_PROC, convert_proc,
                  SEL_OWN, TRUE,
                  NULL);
```

By default, the selection is associated with the primary rank (`XA_PRIMARY`). If your selection needs to use another rank, use either `SEL_RANK` or `SEL_RANK_NAME` to specify the rank.

```
xv_set(sel_owner, SEL_CONVERT_PROC, convert_proc,
                  SEL_RANK_NAME, "SECONDARY",
                  SEL_OWN, TRUE,
                  NULL);
```

`SEL_RANK` takes an atom as an argument. `SEL_RANK_NAME` takes a string as an argument and creates an atom from the string; if the atom already exists, it is re-used. In X terms, this is called interning an atom. (See the description for `XInternAtom` in the *Xlib Reference Manual*.)

After an application makes a request and a selection owner responds to that request, the selection owner waits for notification that the requestor received the data. The maximum time that the selection owner will wait for an acknowledgement from the selection requestor is set with `SEL_TIMEOUT_VALUE`. This value is specified in seconds. If this value is exceeded, the selection is invalid. This value does *not* limit how long a selection owner may hold a selection. A selection owner may hold a selection for any length of time, unless another owner acquires the same selection, in which case the owner loses the selection ownership.

Note that the attributes `SEL_RANK`, `SEL_RANK_NAME`, and `SEL_TIMEOUT_VALUE` are `SELECTION` attributes. They apply to both the selection owner and to the selection requestor.

When and if a selection owner receives a selection request, the selection data needs to be converted. To convert a selection an application can create a selection-item object, or it can define a selection conversion procedure. We cover these possibilities in later sections.

18.2.3 Requesting the Selection (Selection Requestor)

When the user pastes the selection into an application, the ACTION_PASTE event should be used to trigger the application to become the selection requestor. A selection requestor requests to receive the selected data. The selection requestor allows an application to obtain selection data in a specified form. XView permits the selection requestor to receive a selection in one of two ways:

- A blocking request. The application is blocked until the transfer completes, either successfully or unsuccessfully.
- A non-blocking request. The application is not blocked during the selection data transfer.

This section covers both blocking and non-blocking requests.

An application requests a selection from a selection owner by performing two steps: first, the application creates a selection requestor object and sets its attributes. Second, the request is "posted." Posting the request sends a request to the selection owner.

Create a selection requestor from the SELECTION_REQUESTOR package:

```
Selection_requestor  sel_requestor;
sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                             NULL);
```

The SELECTION_REQUESTOR package is defined in the header file *<xview/sel_pkg.h>*. The owner of a selection requestor object is a window-based object. The class hierarchy for a selection requestor is shown in Figure 18-3.

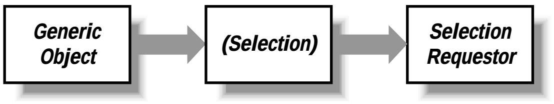


Figure 18-3. Selection requestor class hierarchy

The primary rank is the default rank used in a selection request. To use a rank other than the primary, use either SEL_RANK or SEL_RANK_NAME to specify the desired rank.

The time of the event that triggered selection request, the ACTION_PASTE event, should be set using the attribute SEL_TIME. If you do not set this attribute, the selection package does its best to set the SEL_TIME internally; however, this requires additional processing that is not required if the application sets SEL_TIME.*

*SEL_TIME fixes certain race conditions that would occur if the time of the events associated with selection the were not known.

18.2.3.1 Specifying the target type (selection requestor)

By default, a selection requestor uses the primary rank with the target set to string, which asks the selection owner to convert string data. When the requestor wants the owner to send data, it informs the owner of the type of information it expects using a selection *target*. The target may represent the type of the selection data; however, the selection transfer may not actually involve the selection data. A selection requestor may simply request a timestamp from the selection owner, or some other characteristic of the selection data, such as its length. The selection requestor can also request that the selection owner send a list of the types it can convert. When the requestor specifies the target named `TARGETS`, the owner should return a list of the types that it is able to convert to.

The attribute `SEL_TYPE` specifies the atom name for the requested target. The default value is `XA_STRING`. This may be set to `XA_INTEGER`, or to another atom. `SEL_TYPE_NAME` uses a string argument for the type, such as "`INTEGER`," internally, the package converts this string to an atom.

A single selection request may be used to request more than one target. For example, the selection requestor may desire a list of the possible types the owner can convert as well as selection data. This type of a request, where more than one target is requested at a time, is a *MULTIPLE* request. To initiate a multiple request, use the attribute `SEL_TYPES` with a `NULL`-terminated list of atoms to specify the targets requested, or use `SEL_TYPE_NAMES` with a `NULL`-terminated list of strings.

Two additional attributes allow the requestor to make a multiple request. `SEL_APPEND_TYPES` appends a `NULL`-terminated list of target types on to the existing list. `SEL_APPEND_TYPE_NAMES` appends a `NULL`-terminated list of names on to the current list. By default, a selection requestor selects a string, so the following request adds `TARGETS` to the request list, thus creating a multiple request:

```
xv_set(sel, SEL_APPEND_TYPE_NAMES, "TARGETS", NULL, NULL);
```

18.2.3.2 SEL_REPLY_PROC (selection requestor)

The selection requestor uses `SEL_REPLY_PROC` to specify the reply procedure. This attribute takes a function pointer as an argument. The reply procedure is used as a communication mechanism between the selection owner and the selection requestor. The reply procedure is invoked by the selection package after the conversion, when the response from the selection owner arrives.

If the request is a multiple request, the reply procedure is called once for each target requested:

```
Selection_requestor  sel_requestor;

sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                                  SEL_REPLY_PROC, ReplyProc,
                                  NULL);
```

The reply procedure is described in Section 18.2.5, "Handling the Response."

18.2.3.3 Timeout for a selection response

The attribute SEL_TIMEOUT_VALUE specifies the maximum time that the selection requestor expects the particular request conversion to take. This should be set to the time selection owner will take to convert the selection and return data to the selection requestor.

18.2.3.4 Requesting the CLIPBOARD selection—blocking

To make a blocking request, the selection requestor uses the SEL_DATA attribute to initiate the request. Note that a blocking request puts the entire application on hold during the selection transfer. This type of selection transfer is not recommended for large selections, since the application may be put on hold for a significant amount of time. SEL_DATA takes two arguments, a pointer to an `unsigned long` which is set to the number of elements returned in the selection buffer, and a pointer to an integer which is set to the data format. Using `xv_get()` with the selection requestor and SEL_DATA, with the specified arguments returns the selection data. Example 18-1 shows how the selection requestor requests the clipboard selection using a blocking request.

Example 18-1. Requesting the CLIPBOARD selection--blocking

```
Selection_requestor sel_requestor;

main()
{
          .
          .
          .
     sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                                SEL_RANK_NAME "CLIPBOARD", NULL);
          .
          .
          .
}

window_event_handler(..., event, ...)

Event          *event;

{
    char           *data;
    int            format; /* size of data element: 8,16, 32 bits.*/
    unsigned long  length; /* number of data elements */

    switch (event_action(event) {
        case ACTION_PASTE:
            /* Initiate a blocking request, and get the data */
            data = (char *) xv_get(sel_requestor,
                                    SEL_DATA, &length, &format);
            /* returns with length set to SEL_ERROR          */
            /* if the if request fails.                      */
            break;
    }
}
```

Above, the attribute SEL_DATA with xv_get() sends the selection request to the selection owner. The returned data is placed in *data*. If the conversion was successful, the application needs to free the returned data, data when it is finished using it.

18.2.3.5 Requesting the CLIPBOARD selection—non-blocking

If a selection requestor wants to make a non-blocking request, it calls the procedure sel_post_req() which is defined by the selection package. This procedure sends a non-blocking request to the selection owner.

The procedure sel_post_req() has the following format:

```
     int
     sel_post_req( sel_req )
          Selection_requestor    sel_req;
```

The procedure sel_post_req() returns XV_OK or XV_ERROR. If no reply procedure is defined, XV_ERROR is returned. Example 18-2 shows how the selection requestor requests the clipboard selection using a non-blocking request.

Example 18-2. Non-blocking selection request

```
long                *file_buffer;     /* contents of selected file */
long                 file_buffer_size; /* # of longs in file_buffer */
Selection_requestor  sel_requestor;

main()
{
          .
          .
          .
     sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                                  SEL_REPLY_PROC, ReplyProc,
                                  SEL_RANK_NAME, "CLIPBOARD",
                                  NULL);

}
window_event_handler(..., event, ...)

Event          *event;

{

     switch (event_action(event) {
     case ACTION_PASTE:
              /* initiate non-blocking request */
              sel_post_req(sel_requestor);
              break;
          }
}
```

18.2.4 Converting the Selection (Selection Owner)

When the selection requestor posts a request, the request is sent via the X server to the client which currently owns the selection.* If the request is sent to an XView-based application, the toolkit will determine the appropriate selection-owner object to forward the request to. If the selection-owner object has a conversion procedure registered (see `SEL_CONVERT_PROC`), it will be called with information about the request. If the selection-owner object does not have a conversion procedure defined, but does have selection items registered on behalf of the selection owner, the toolkit will determine if the request matches any of the selection items. If a match is made, the toolkit will respond to the request with the selection item's data. If a match is not made, the toolkit will reject the request.

If a selection-owner object uses selection items for all targets for which it is willing to respond to, then there is no need to register a conversion procedure. If the selection-owner object uses selection items for some responses and converts other responses in a conversion procedure, then the default selection conversion procedure (`sel_convert_proc()`) must be called from within the selection-owner object's conversion procedure. This is where the selection-owner object's selection items are handled.

It is the responsibility of the selection owner to either convert the data into the type specified by the requestor or reject the request. The conversion of data typically happens in the selection-owner object's conversion procedure (see `SEL_CONVERT_PROC`). The conversion procedure should convert the selection it holds to the requested type. The conversion procedure's return value is set to `TRUE` if the owner successfully converts the selection to the target type and to `FALSE` if it rejects the selection request.

The form of the conversion procedure is:

```
int
convert_proc( sel, replyType, replyBuff, length, format )
       Selection_owner  sel;
       Atom             *replyType;
       Xv_opaque        *replyBuff;
       unsigned long    *length;
       int              *format;
```

The argument *sel* specifies the selection owner. When the conversion routine is called, *replyType* is set to the target requested. On return, it should indicate the type the selection was converted to. For example, the selection requestor may be requesting the target `LENGTH` so *replyType* would have the value `LENGTH` when the conversion routine is called. Before the conversion routine returns, *replyType* would be set to `INTEGER`, representing the type of the value returned. As another example, if the requestor is requesting the target `FILE_NAME`, the conversion procedure needs to explicitly set *replyType* to the atom which describes the converted type, in this case, `TEXT`.

The *replyBuff* is a pointer to a buffer address which contains the converted data. Do *not* return automatic storage in *replyBuff*. If your selection data uses more data than a `long`

*For performance reasons, local selection transfers using `sel_post_req()` do not go through the server, and the data is transfered within the application itself.

int, be sure to malloc the storage for the converted data. This storage can later be freed in a SEL_DONE_PROC.

The conversion procedure is called with *length* set to the server's maximum allowed buffer size. Before the conversion procedure returns, *length* should be assigned the number of elements in *replyBuff*.

format specifies a pointer to the element size of the data. *format* should be set to the element size used in replyBuff. Valid values are 8, 16, or 32 for 8-bit, 16-bit, or 32-bit, quantities, respectively.

All applications supporting selections are required by the ICCCM to convert the TARGETS type. This type returns a list of types the conversion procedure understands and supports. The conversion procedure should include the code necessary to convert this target (see ICCCM for more details).

For selections that are larger than the maximum size allowed by the server, the selection package sends the data in increments. The selection package handles this "large" selection case automatically. That is, if the conversion procedure returns a *replyBuff* larger than the maximum size allowed by the server, the selection package will divide the buffer into reasonable sizes and send the data in increments using the INCR target described by the ICCCM. If you want to specify that a selection should be transferred incrementally, you may specify an incremental transfer. An incremental transfer is specified by constructing an increment message. The increment message and incremental transfer are covered in Section 18.4, "How to Send Data Incrementally."

18.2.4.1 The default conversion procedure

The selection owner has a default conversion procedure called sel_convert_proc(). This conversion procedure is predefined to convert TARGETS for several targets whose conversion is supported by the selection package. This procedure also handles conversions where selection items are used (see Section 18.3.1, "The Selection Item").

Normally, an application-defined procedure should call sel_convert_proc() before returning. This assures that the selection-owner object will handle conversions involving a selection-item object. If you are sure that your conversion procedure handles all the targets that your application will support, then you do not need to call sel_convert_proc() from your conversion procedure.

18.2.4.2 Sample selection owner with conversion procedure

Example 18-3 shows a selection-owner object and defines the conversion procedure, convert_proc(). When an ACTION_COPY event is detected, sel_owner acquires the selection by setting the attribute SEL_OWN to TRUE. At this time you should also set SEL_TIME to the time on to the time of the ACTION_COPY event. If you do not set the selection time using SEL_TIME, the selection owner package sets the time, but this requires the package to perform additional processing.

The sample procedure `convert_proc()` is invoked when a selection requestor posts a request (not shown here). In this routine, the selection rank is stored in *selection* and the response data, "Primary Selection content . . . " is is stored in *str*. Lastly, *format*, *length*, *type*, and *replyBuff* are assigned the appropriate values to convert the data (*str*) to the requested type (`XA_INTEGER` or `XA_STRING`).

Example 18-3. Selection owner program

```
static int convert_proc();
Selection_owner     sel_owner;

main()
{
     :
     sel_owner = xv_create(window, SELECTION_OWNER,
                SEL_CONVERT_PROC, convert_proc,
                NULL);
     :
}

window_event_handler(..., event, ...)
      :
    Event          *event;
      :
{
      :
     switch (event_action(event) {
     case ACTION_COPY:
          xv_set(sel_owner, SEL_OWN, TRUE, NULL);
          break;
         }
      :
}

int
convert_proc( sel_owner, type, replyBuff, length, format )
    Selection_owner    sel_owner;
    Atom              *type;
    Xv_opaque         *replyBuff;
    unsigned long     *length;
    int               *format;
{
    int    len;
    Atom   selection;
    char   str[50];

    selection = (Atom) xv_get( sel_owner, SEL_RANK );
    strcpy( str,"Primary selection content...\n");

    if ( *type == TARGETS )  {
       /* Support target conversion here */
       return( TRUE );
    }

    if ((selection == XA_PRIMARY ) && ( *type == XA_STRING )) {
        *format = 8;
        *length = strlen( *str ) + 1 ;
```

Example 18-3. Selection owner program (continued)

```
        *type = type;
        *replyBuff = (Xv_opaque) strcpy(str);
        return(TRUE);
     }

    if ((selection == XA_PRIMARY ) && ( *type == XA_INTEGER )) {
        len = strlen(str);
        *format = 32;
        *length = 1;
        *replyBuff = (Xv_opaque)&len;
        return(TRUE);
    }
    /*   call sel_convert_proc(sel_owner, type,
                              replyBuff, length, format ); */
    /* return, or call the default conversion procedure */

}
```

18.2.5 Handling the Response (Selection Requestor)

Once the selection owner has responded to the request, the selection-requestor object is notified. If the selection-requestor object used a blocking request, then the blocked xv_get() will return with the response. If the requestor had made a non-blocking request then the response would be sent to the requestor's callback procedure. This callback procedure is called a reply procedure and can be registered on the requestor object using the SEL_REPLY_PROC attribute.

Information passed into the reply procedure will indicate whether the selection request was completed successfully. If it was successful, the requestor is free to use the response as appropriate. Typically, it will display the data in a window. The format for the application-defined reply procedure is:

```
void
reply_proc( sel_req, target, type, replyValue, length, format )
     Selection_requestor  sel_req;
     Atom                 target;
     Atom                 type;
     Xv_opaque            replyValue;
     unsigned long        length;
     int                  format;
```

The argument *sel_req* is the selection requestor. The name of the request/response is supplied in *target*. For example, `TARGETS`, `STRING`, or `LENGTH`. The argument *type* specifies the type of the target returned, values corresponding to the three mentioned, would be `TARGETS`, `STRING` and `INTEGER`. *replyValue* specifies the data content returned. *length* specifies the length of the data. *format* specifies the format of the data. Example 18-4 shows a sample reply procedure.

The data for *reply_Value* is malloced. The application is responsible for freeing this data once it is finished using it.

If the response was returned in increments, then the reply procedure will be called back more than once with individual buffers representing part of the response. The reply procedure must be able to handle incremental responses. See Section 18.4, "How to Send Data Incrementally," for more information on how to do this.

In the case where multiple targets are requested, the selection's reply procedure is called more than once.

Example 18-4. Sample reply procedure – SelectionReplyProc

```
void
SelectionReplyProc(sel, target, type, value, length, format)
    Selection_requestor   sel;
    Atom                target;
    Atom                type;
    Xv_opaque           value;
    unsigned long       length;
    int                 format;
{
    if (length == SEL_ERROR) {
        SelectionError(sel, target, *(int *)value);
        return;
    }
    if (target == ATOM(server, "TARGETS")) {
     textsw_insert(textsw,
              "Holder will convert the following targets:\n", 43);
               /* 43 is the size of the string */
      do {
        Atom *targets = (Atom *)value;
        char *target_name;
        if (targets[--length]) {
            target_name = (char *)xv_get(server, SERVER_ATOM_NAME,
                                targets[length]);
            textsw_insert(textsw, "\t", 1);
            textsw_insert(textsw, target_name, strlen(target_name));
            textsw_insert(textsw, "\n", 1);
        }
      } while(length);
    }
    else
    if (target == ATOM(server, "TIMESTAMP")) {
        char buf[10];
        textsw_insert(textsw, "TIMESTAMP of acquisition: ", 26);
        sprintf(buf, "%U\n", *(unsigned long *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
    if (target == ATOM(server, "LENGTH")) {
        char buf[10];
        textsw_insert(textsw, "Length of selection: ", 21);
        sprintf(buf, "%d\n", *(int *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
```

Example 18-4. Sample reply procedure – SelectionReplyProc (continued)

```
    if (target == ATOM(server, "STRING")) {
        static int   incr = False;

        if (type == ATOM(server, "INCR")) {
            textsw_insert(textsw, "Contents of the selection:\n", 27);
            incr = True;
        }
        else
        if (length) {
            if (!incr)
                textsw_insert(textsw, "Contents of the selection:\n", 27);
            textsw_insert(textsw, (char *)value, length);
            textsw_insert(textsw, "\n", 1);
        }
         else
            incr = False;
    }
    else
    if (target == ATOM(server, "DELETE")) {
        textsw_insert(textsw, "The Selection has been deleted\n", 31);
    }

    textsw_insert(textsw, LINE, strlen(LINE));
}
```

18.2.5.1 Handling selection reply procedure errors

If the selection conversion fails, and you are using a non-blocking request, the reply procedure is called with *replyValue* set to an error code and *length* set to `SEL_ERROR`. The reply procedure error codes are defined in *sel_pkg.h* and shown in Table 18-2.

If the selection conversion fails, and you are using a blocking request without a reply procedure defined, the `xv_get()` returns with the *length* argument set to `SEL_ERROR`, *format* set to 0, and a `NULL` value is returned. If a reply procedure is defined, its *replyValue* argument is set to one of the error codes shown in Table 18-2.

Table 18-2. Error Codes

Code	Description
`SEL_BAD_CONVERSION`	If the conversion is refused by the selection holder or there is no holder of the selection, this value is returned. This may mean that there is no owner for the selection, that the owner does not support the conversion implied by target, or that the server did not have sufficient space.
`SEL_BAD_TIME`	The SelectionNotify time does not match the package time value.
`SEL_BAD_WIN_ID`	The SelectionNotify requestor ID does not match the package requestor ID.

Table 18-2. Error Codes (continued)

Code	Description
SEL_BAD_PROPERTY	Obsolete.
SEL_BAD_PROPERTY_EVENT	Obsolete.
SEL_PROPERTY_DELETED	Obsolete.
SEL_TIMEDOUT	Selection timed out.

A sample application-defined errror routine, for a non-blocking error handler, is shown in Example 18-5. Code similar to this should be invoked to handle errors in your selection application.

Example 18-5. Sample selection reply error handler

```
SelectionError(sel, target, errorCode)
    Selection_requestor    sel;
    Atom        target;
    int               errorCode;
{
    Atom     rank;
    char    *rank_string;
    char    *target_string = (char *)xv_get(server,
                                        SERVER_ATOM_NAME, target);
    char     msg[100];

    rank = (Atom)xv_get(sel, SEL_RANK);
    rank_string = (char *)xv_get(server, SERVER_ATOM_NAME, rank);

    sprintf(msg, "Selection failed for rank ‘‘%s’’ on target ‘‘%s’’: ",
                                     rank_string, target_string);
    textsw_insert(textsw, msg, strlen(msg));

    switch(errorCode) {
      case SEL_BAD_CONVERSION :
        textsw_insert(textsw, "Conversion Rejected",
        strlen("Conversion Rejected"));
        break;

      case SEL_BAD_TIME:
        textsw_insert(textsw, "Bad Time Match",
                 strlen("Bad Time Match"));
        break;

      case SEL_BAD_WIN_ID:
        textsw_insert(textsw, "Bad Window Match",
                 strlen("Bad Window Match"));
        break;

      case SEL_TIMEDOUT:
        textsw_insert(textsw, "Timeout", strlen("Timeout"));
        break;
    }
}
```

.8.2.6 If the Selection is Lost (Selection Owner)

If the owner loses the selection, because someone else acquired the same rank selection, the lose procedure is called. The selection-owner object attribute `SEL_LOSE_PROC` specifies the selection owner's lose procedure. This procedure is called by the toolkit, to inform the selection owner that it has lost the ownership of the given selection. The form of the lose procedure is shown below:

```
void
lose_proc( sel )
    Selection_owner  sel;
```

sel specifies the selection-owner object.

One example of `lose_proc()` usage is for unhighlighting the highlighted text in a `textsw`. The code fragment below shows a selection lose procedure that simply informs the user that the selection was lost.

```
void
SelectionLoseProc(sel)
    Selection_owner   sel;
{
    xv_set(frame, FRAME_LEFT_FOOTER, "Lost Selection...", NULL);
}
```

18.2.7 Cleanup – When the Selection Completes (Selection Owner)

The attribute `SEL_DONE_PROC` specifies the application-defined *done* procedure that is called by the toolkit when a selection request has successfully completed. It is called once following each successful transfer of data to the requestor. Thus, if the selection request is a multiple, or results in an incremental reply, the done procedure is called more than once. This procedure can be used to deallocate any selection replies allocated in the conversion procedure.

```
void
done_proc( sel, replyBuff, target )
     Selection_owner  sel;
     Xv_opaque        replyBuff;
     Atom             target;
```

The argument *sel* specifies the selection-owner object. *replyBuff* specifies the address which contains the converted data. *target* specifies the target type returned by the conversion procedure.

Example 18-6 shows a sample done procedure.

Example 18-6. Sample done procedure – SelectionDoneProc

```
void
SelectionDoneProc(sel, data, target)
    Selection_owner   sel;
    Xv_opaque         data;
    Atom              target;
{
    if (target == ATOM(server, "STRING"))
       free((char *)data);
}
```

18.3 How Selection Works (With a Selection Item)

This section describes using a selection item that can eliminate the selection owner’s need for a conversion procedure. This may simplify the programmer’s job of implementing a selection transfer.

Selection items are good to use for conversions that involve static data—data that will not change over the lifetime of the selection. Do not use a selection item for selections where the data associated with the selection is not known until the selection request is received (using a selection conversion procedure is best for this case). Also, do not use a selection item for any selection conversion that has side effects (for example, a delete request).

Below is a brief overview of the steps that take place during a single transfer of data from one XView application to another using a selection item. We’ll assume we have two applications, application *A* and application *B*, either of which can operate as the owner or the requestor.

1. (Owner side.) The user makes a selection in application *A* and the application highlights the selection.
2. (Owner side.) The user may pre-register a conversion for a particular target by creating a selection-item object and setting its attributes.
3. (Owner side.) In the XView Selection package, nothing further happens until an event of interest is detected, for example, an `ACTION_COPY` event. At this time, a selection item may claim ownership of the current selection in application *A*. Also at this time, selection data may be attached to the selection-item object.
4. (Requestor side.) The user pastes the selected data into application *B*, causing an `ACTION_PASTE` event to occur.
5. (Requestor side.) The `ACTION_PASTE` event may cause the Selection Requestor to make a non-blocking request to receive the selected data into application *B*.

6. (Owner side.) When there is a match between a selection-item object's target type, and a requested target, the selection package converts and sends the selection data to the selection requestor. XView internally handles the data conversion by calling the package's default conversion procedure `sel_convert_proc()`.

7. (Requestor side.) Once the data conversion has completed (either successfully or unsuccessfully), the application-defined reply procedure is invoked. The data may displayed in application *B* (if the conversion routine was successful) or the user may simply be given some indication that data was received. Other possibilities include: the kind of data selected in application *A* cannot be pasted in application *B*, or that the kind of data requested by *B* cannot be supplied by *A*.

8. (Owner side.) Once the selection transfer is complete, an application-defined "done" procedure may be called by the selection owner. The done procedure may be used to free the memory associated with the selection, or to perform other cleanup that is required.

9. (Owner side.) If another application acquires the selection, application *A*'s "lose" procedure is called. This procedure is used to handle notification of loosing selection ownership. It tells the selection owner (application *A*) that it has lost ownership of the selection. For example, the lose procedure might unhighlight text that was previously selected and highlighted.

18.3.1 The Selection Item

The first task for the application that is to become the selection owner, using a selection-item object, is to pre-register a conversion or several conversions. Below, we describe this step in detail. Also, the selection needs to be marked. For example, when the pointer is placed over text in a canvas, an `ACTION_SELECT` should highlight the selection. While `LOC_DRAG` occurs, the highlight would be extended over the selection.

After the user makes a selection and it is marked, the application waits to receive an event of interest. For example, in a text subwindow, the act of selecting an item causes the primary rank to be acquired. For the text subwindow example, an `ACTION_COPY` event causes the CLIPBOARD selection to be acquired (note that in this case two selections are acquired since the COPY operation in OPEN LOOK uses the clipboard).

With a selection item, applications perform three steps to become the selection owner:

1. A `Selection_owner` object must be created using `xv_create()` (assuming a previously created selection-owner object is *not* being re-used.) A selection-item object is created and its attributes are set.

2. The selection must be acquired.

3. The selection data is associated with the selection item.

A selection-owner object is created using `xv_create()`:

```
Selection_owner   sel_owner;
sel_owner = xv_create(window, SELECTION_OWNER,
                      NULL);
```

To pre-register a selection item for a string conversion, use the following code:

```
Selection_item   sel_item;
sel_item = xv_create(sel_owner,
                 SELECTION_ITEM,
                 SEL_TYPE_NAME,    "STRING",
                 SEL_FORMAT,       8,  /* bits per unit (char)*/
                 NULL);
```

Pre-register a conversion by setting several selection-item object attributes. `SEL_TYPE_NAME` specifies the type, using a string for the atom type of the selection. `SEL_FORMAT` specifies the format for the data. Valid values are 8, 16, or 32 for 8-bit, 16-bit, or 32-bit quantities, respectively. `SEL_TYPE` specifies the type of the conversion that the item supports. To support more than one type, you need to use additional selection items, one for each type. Since a string conversion is the default for a selection item, the following code registers the same conversion at that shown above:

```
Selection_item     sel_item;
sel_item = xv_create(sel_owner, SELECTION_ITEM,
                    NULL);
```

The owner of a selection-item object is an object of type `Selection_owner`. The owner of a selection item defines the rank to which the item belongs. Figure 18-4 shows the class hierarchy for a selection-item object.

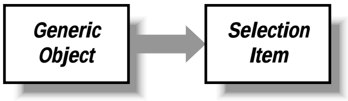


Figure 18-4. Selection item class hierarchy

A selection is acquired by setting `SEL_OWN` to `TRUE` on the selection-owner. By default, the selection is associated with the primary rank (`XA_PRIMARY`). If your selection needs to use another rank, use either `SEL_RANK` or `SEL_RANK_NAME` to specify the rank as shown below:

```
xv_set(sel_owner, SEL_OWN, TRUE,
                 SEL_RANK_NAME "CLIPBOARD",
                 NULL);
```

After the selection-owner object and the selection-item object are created, and the selection is acquired, data needs to be associated with the selection item (if the selection item is defined, its data can be set at any time). If the selection is a string transfer using the primary selection, this step is easy:

```
char  *sel_string;

sel_string="Sample selection data.";

xv_set(sel_item,
    SEL_DATA, sel_string, /*pointer to highlighted data*/
    SEL_LENGTH, strlen(sel_string),
    NULL);
```

The attribute SEL_DATA associates the selection data with the selection-item object. SEL_LENGTH specifies the number of data elements in the selection item. Another selection item attribute, SEL_COPY indicates whether the selection item package should copy and maintain the selection-item object's data. If this is set to FALSE, it is up to the application to maintain the data. To pre-register a selection item for the targets type that returns a list of five atoms, use the following code:

```
Selection_item  sel_item;
Atom            targets[5];
/* initialize the targets array */
targets[0] = (Atom)xv_get(server, SERVER_ATOM, "TARGETS");
targets[1] = (Atom)xv_get(server, SERVER_ATOM, "TIMESTAMP");
targets[2] = (Atom)xv_get(server, SERVER_ATOM, "LENGTH");
targets[3] = (Atom)xv_get(server, SERVER_ATOM, "STRING");
targets[4] = (Atom)xv_get(server, SERVER_ATOM, "DELETE");

sel_item = xv_create(sel, SELECTION_ITEM,
                          SEL_TYPE_NAME,    "TARGETS",
                          SEL_FORMAT,       32,
                          SEL_LENGTH,       5,
                          SEL_DATA,        targets,
                          NULL);
```

Once a selection-item object is created, and the selection data is associated with the selection item, the selection item's owner waits to receive a request. When and if the item's owner receives a selection request, the selection data is converted, according to the specifications set using the selection item attributes. Data is supplied through the SEL_DATA attribute. Note that the default conversion procedure sel_convert_proc() needs to be called in order to convert a selection using a selection item.

The owner of a selection-item object, is a selection-owner object. A selection-owner object can access its selection items using the attributes SEL_FIRST_ITEM and SEL_NEXT_ITEM.

18.4 How to Send Data Incrementally (Selection Owner)

Selections whose data is larger than the server's maximum request size require special handling. The selection package checks these limits and handles these large cases internally, using incremental selections. The selection owner's conversion procedure may also specify, for any reason, that the data is to be sent in increments.

An incremental transfer is specified by constructing an increment message as follows:

1. Set *replyType* to an atom named `INCR`.
2. Set *replyBuff* to an integer representing a lower bound on the number of bytes of data in the selection.
3. Set *length* to 1.
4. Set *format* to 32.

The conversion routine should then return `TRUE`. The selection package sends the `INCR` message to the requestor and then calls the conversion routine repeatedly, for each buffer making up the response, until *length* is set to zero by the application. This indicates the end of the data transfer.

A selection owner can determine if a selection is larger than the maximum allowed by the server by comparing the value of the length variable. The conversion routine is called with *length* set to the server's maximum request size. If the owner has chosen to send the selection data in increments, the size of each increment should be less than or equal to *length*. If the size is larger, the selection package sends the data as a series of incremental transfers. Example 18-7 shows a conversion procedure that sends data incrementally.

Example 18-7. An incremental conversion procedure

```
static int
ConvertProc( selOwner, type, data, length, format )
Selection_owner    sel_owner;
    Atom          *type;
    Xv_opaque     *data;
    long          *length;
    int           *format;
{
    static char  *tmp=(char *) NULL;
    static int  firstTime=1;
    static int  numBytes=0;

    fileSize = TERM_CAP_SIZE;

    if (  *type == XA_STRING )   {
          /*
           * Send INCR message to the requestor.
          */
          if ( firstTime ) {
              static long   fSize;

              fSize = TERM_CAP_SIZE;
              *type = xv_get( server, SERVER_ATOM, "INCR");
```

Example 18-7. An incremental conversion procedure (continued)

```
            *data = (Xv_opaque) &fSize;
            *format = 32;
            *length = 1;
            firstTime = 0;
            return TRUE;
        }

        if ( tmp != NULL )
             free( tmp );

        tmp = (char *) malloc( BUFSIZE  );

        if( (numBytes = read( fd, tmp, BUFFSIZE )) == -1 ) {
            fprintf( stderr,"errno = %d \n", errno );
            numBytes=0;
        }
        if( (numBytes == 0 ) {
            free(tmp)
            tmp=null
        }
        *format = 8;
        *length = numBytes;
        *type = XA_STRING;
        *data = (Xv_opaque) tmp;
        return TRUE;
    }
    return FALSE;
}
```

18.4.1 How to Handle Incremental Replies (Selection Requestor)

The selection package starts an incremental reply by sending the `INCR` type to the reply procedure. *replyValue* is set to a lower bound on the number of bytes of data in the selection. The reply procedure needs to be able to build up the data as it is called repeatedly until all the selection data has been transferred.

The conversion procedure informs the reply procedure when the incremental transfer is complete by sending a message. The conversion procedure sets *length* to zero and *replyValue* to `NULL`, indicating the end of incremental data transfer. Clients need to free the *reply Value*. Example 18-8 shows how the reply procedure handles incremental replies.

Example 18-8. Incremental reply – IncrReply.c

```
static void
ReplyProc( selReq, target, type, replyBuf, len, format )
     Selection_requestor  selReq;
     Atom                 target;
     Atom                 type;
     Xv_opaque            replyBuf;
     unsigned long        len;
     int                  format;
{
```

Example 18-8. Incremental reply – IncrReply.c (continued)

```
    if ( len == SEL_ERROR ) {
        int   errCode;

        bcopy( (char *) replyBuf, (char *) &errCode, sizeof( int ) );
        switch( errCode ) {
            case SEL_BAD_CONVERSION :
                printf("ReplyProc: Conversion failed!\n");
                break;
            case SEL_BAD_TIME:
                printf("ReplyProc: Bad time!\n");
                break;
            case SEL_BAD_WIN_ID:
                printf("ReplyProc: Bad window id!\n");
                break;
            case SEL_TIMEDOUT:
                printf("ReplyProc: Timed out!\n");
                break;
        }
    }

    if ( len == 0 ) {
       printf("End of incremental data transfer.\n");
       return;
    }

    if ( type == xv_get( server, SERVER_ATOM, "INCR") ) {
       long  size;

       bcopy( (char *) replyBuf, (char *) &size, sizeof( long ) );
       printf("Get ready for INCR of size %d\n", size );
    }

    if ( ( type == XA_STRING ) && len )
       printf("%.*s\n", len, (char *) replyBuf );
}
```

18.5 Requesting and Converting Multiple Targets

A selection request may involve more than one target. This type of request is called a MULTIPLE request. Normally on both the selection owner side, and on the selection requestor side, a MULTIPLE request is handled by the packages, without any interaction by the application programmer. This section outlines how the selection-requestor package and the selection-owner package handle MULTIPLE requests.

The selection requestor package detects that more than one target has been requested when the following attributes are used to request more than a single target:

```
SEL_APPEND_TYPE
SEL_APPEND_TYPES
SEL_TYPES
SEL_TYPE_NAMES
```

The selection requestor then requests that the selection owner convert a list of targets.

On the selection owner side, a `MULTIPLE` request is treated as a stream of single requests. The package calls the conversion procedure as many times as required to convert all of the targets requested.

In the special case where a multiple request also involves an incremental target (`INCR`), the selection owner calls the application-defined conversion procedure with *format* set `SEL_MULTIPLE` to indicate that the selection request is a incremental request that is part of a multiple request.

18.6 Additional Transfer Mechanisms (Selection Requestor)

A selection requestor may associate data with the window property being used in the selection transaction. This property may be used by the selection owner for certain types of selection transfers. The selection-requestor object attributes: `SEL_PROP_DATA`, and `SEL_TYPE_INDEX` support bi-directional data transfer for a property associated with a selection. In addition, the attributes `SEL_PROP_FORMAT`, `SEL_PROP_LENGTH`, `SEL_PROP_TYPE`, and `SEL_PROP_TYPE_NAME`, also support this type of selection transfer. A selection requestor may associate a property with the selection that the requestor is requesting. For example, this associated property may be used to support an `INSERT_SELECTION` or a `INSERT_PROPERTY` target. Or properties may be used by the owner for other reasons. A selection-requestor object may associate data with the property that the selection is requesting. This information may be of interest to a selection owner for certain types of selection transfers.

18.7 Additional Transfer Mechanisms (Selection Owner)

If the selection requestor is sending data to the selection owner, using additional properties, the attribute `SEL_PROP_INFO` allows the selection owner to access this property data.

18.8 Sample Selection Owner Program with a Selection Item

Example 18-9 shows a simple panel with two text items and two buttons inside a frame. This example uses a selection item to return the TARGETS type and a conversion procedure to return the remaining supported types. The “Selection:” field allows the user to enter the desired selection rank. The “Contents:” field allows the user to enter a string, the selection data to be operated on. The “Own Selection” button causes the application to own the selection. The “Lose Selection” button causes the selection to be lost.

Example 18-9. Sample program – sel_hold.c

```
/*
 * sel_hold.c: Example of how to acquire and hold a selection.
 *
 */
#include <stdio.h>
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/font.h>
#include <xview/sel_pkg.h>

Frame             frame;
Xv_Server         server;
Panel             panel;
Panel_item        p_selection,
                  p_contents,
                  p_own,
                  p_lose;

Selection_owner   sel;
Selection_item    sel_targets;

#define ATOM(server, name)  (Atom)xv_get(server, SERVER_ATOM, name)

main(argc, argv)
    int    argc;
    char **argv;
{
    Panel_setting   NotifyProc();
    int             SelectionConvertProc();
    void            SelectionDoneProc(),
                    SelectionLoseProc();
    Xv_Font         font;
    Atom            targets[5];

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create((Window)NULL, FRAME,
              XV_X,      520,
              XV_Y,      655,
              XV_LABEL,  "Selection Holder Example",
              FRAME_SHOW_FOOTER,  True,
              NULL);

    panel =  xv_create(frame, PANEL, NULL);


    p_selection = xv_create(panel, PANEL_TEXT,
                 PANEL_LABEL_STRING,        "Selection:",
                 PANEL_VALUE_DISPLAY_LENGTH, 40,
                 PANEL_NOTIFY_PROC,          NotifyProc,
                 PANEL_ITEM_X,               xv_col(panel,0),
                 PANEL_ITEM_Y,               xv_row(panel,0),
                 NULL);
```

Example 18-9. Sample program – sel_hold.c (continued)

```
    p_contents = xv_create(panel, PANEL_TEXT,
                  PANEL_LABEL_STRING,        "Contents:",
                  PANEL_VALUE_DISPLAY_LENGTH, 40,
                  PANEL_ITEM_X,               xv_col(panel,0),
                  PANEL_ITEM_Y,               xv_row(panel,1),
                  NULL);

    p_own = xv_create(panel, PANEL_BUTTON,
                  PANEL_LABEL_STRING, "Own Selection",
                  PANEL_NOTIFY_PROC,   NotifyProc,
                  PANEL_ITEM_X,        xv_col(panel,5),
                  PANEL_ITEM_Y,        xv_row(panel,2),
                  NULL);

    p_lose = xv_create(panel, PANEL_BUTTON,
                  PANEL_LABEL_STRING, "Lose Selection",
                  PANEL_NOTIFY_PROC,   NotifyProc,
                  PANEL_ITEM_X,        xv_col(panel,30),
                  PANEL_ITEM_Y,        xv_row(panel,2),
                  NULL);

     /* Create a selection owner object */
    sel = xv_create(panel, SELECTION_OWNER,
                  SEL_CONVERT_PROC,    SelectionConvertProc,
                  SEL_DONE_PROC,       SelectionDoneProc,
                  SEL_LOSE_PROC,       SelectionLoseProc,
                  NULL);

    targets[0] = (Atom)xv_get(server, SERVER_ATOM, "TARGETS");
    targets[1] = (Atom)xv_get(server, SERVER_ATOM, "TIMESTAMP");
    targets[2] = (Atom)xv_get(server, SERVER_ATOM, "LENGTH");
    targets[3] = (Atom)xv_get(server, SERVER_ATOM, "STRING");
    targets[4] = (Atom)xv_get(server, SERVER_ATOM, "DELETE");

     /* Create a selection item, owned by the selection owner we just
      * created.  This pre-registers a conversion, in this case a
      * conversion for ‘‘TARGETS’’.
      */
    sel_targets = xv_create(sel, SELECTION_ITEM,
                   SEL_TYPE_NAME,    "TARGETS",
                   SEL_FORMAT,       32,
                   SEL_LENGTH,       5,
                   SEL_DATA,        (Xv_opaque)targets,
                   NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
    exit(0);
}
```

The selection-owner object `sel` specifies the conversion procedure, the done procedure, and the lose procedure. Next, the targets array is filled with supported atoms. These are the names for the supported conversions. The selection-item object’s conversion is specified using the attributes: SEL_TYPE_NAME, SEL_FORMAT, and SEL_LENGTH. The attribute SEL_DATA associates the selection data with the selection item.

18.8.0.1 The notify procedure

Example 18-10 shows how the NotifyProc() routine is used to handle the selection rank entry, owning the selection, and losing the selection. If a selection rank has been entered, it is read from the panel text item and then assigned to sel, the selection-owner object. If the "own" button is pressed, the current selection is acquired by setting SEL_OWN to TRUE and SEL_TIME to the time of the event. If the "lose" button is pressed, the current selection is released by setting SEL_OWN to FALSE and SEL_TIME to the time of the event.

Example 18-10. The notify procedure – NotifyProc()

```
Panel_setting
NotifyProc(item, event)
    Panel_item       item;
    Event           *event;
{
    if (item == p_selection) {
       char     *rank;

         /* Get the rank of the selection     */
         /* the user would like to use.       */
         rank = (char *)xv_get(item, PANEL_VALUE);

         /* Set the rank to our selection owner object. */
         xv_set(sel, SEL_RANK_NAME, rank, NULL);

        return(PANEL_NEXT);

    } else if (item == p_own) {

         /* The user pressed the ‘‘own’’ button, so we */
         /* acquire the selection.                     */
         xv_set(sel, SEL_OWN, True,
                     SEL_TIME, event_time(event),
                     NULL);

         xv_set(frame,
                FRAME_LEFT_FOOTER, "Acquired Selection...", NULL);

    } else if (item == p_lose) {

         /* The user pressed the ‘‘lose’’ button, so we lose ownership
          * of the selection.
          */
         xv_set(sel, SEL_OWN, False,
                     SEL_TIME, event_time(event),
                     NULL);

         xv_set(frame, FRAME_LEFT_FOOTER, "Lost Selection...", NULL);
    }

    return(PANEL_DONE);
}
```

18.8.0.2 The conversion procedure

The conversion procedure is specified by the SEL_CONVERT_PROC attribute. This conversion procedure, SelectionConvertProc, is called whenever an application makes a request to the selection owned by this application. This procedure is shown in Example 18-11. The selection owner responds to the requesting application by converting the acquired selection to the target type requested. A well behaved routine provides for the case of a TARGETS request by returning a list of valid target types to the requestor. In this case, the selection owner returns the length of the string selected for LENGTH, a copy of the contents of the string selected for STRING, and sets the string to NULL and returns NULL for the DELETE target.

Example 18-11. Sample conversion procedure – SelectionConvertProc

```
int
SelectionConvertProc(sel, target, data, length, format)
    Selection_owner    sel;
    Atom              *target;  /* Input/Output */
    Xv_opaque         *data;    /* Output */
    unsigned long     *length;  /* Output */
    int               *format;  /* Output */
{

    /* Request for the length of the selection. */
    if (*target == ATOM(server, "LENGTH")) {
          static unsigned long  len;
          char                 *contents;

          contents = (char *)xv_get(p_contents, PANEL_VALUE);
          len = strlen(contents);

          *target = ATOM(server, "INTEGER");
          *format = 32;
          *length = 1;
          *data = (Xv_opaque)&len;
          return(True);
    }
    /* Request for the string contents of the selection. */
    else if (*target == ATOM(server, "STRING")) {
          char                  *contents;

          contents = (char *)xv_get(p_contents, PANEL_VALUE);

          *target = ATOM(server, "STRING");
          *format = 8;
          *length = strlen(contents);
          *data = (Xv_opaque)strdup(contents);
          return(True);
    }
    /* Request to delete the selection. */
    else if (*target == ATOM(server, "DELETE")) {
          xv_set(p_contents, PANEL_VALUE, "", NULL);
          *target = ATOM(server, "NULL");
          *format = 32;
          *length = 0;
          *data = (Xv_opaque)NULL;
          return(True);
```

Example 18-11. Sample conversion procedure – SelectionConvertProc (continued)

```
    } else
    /* Call the default selection conversion procedure.
     * It handles requests for any pre-registered
     * conversions, including TARGETS.
     */
      return(sel_convert_proc(sel, target, data, length, format));
}
```

18.8.1 The Done Procedure

The selection done procedure is called after each conversion has happened. This gives the application a chance to free up memory.

```
void
SelectionDoneProc(sel, data, target)
    Selection_owner  sel;
    Xv_opaque        data;
    Atom             target;
{
    if (target == ATOM(server, "STRING"))
      free((char *)data);
}
```

Note, in this example, we only alloc data for string requests so we only free it for string data.

18.8.2 The Lose Procedure

The lose procedure lets the selection package lose ownership a selection gracefully. For example, the user can can be informed that the selection was lost, as in the following example.

```
void
SelectionLoseProc(sel)
    Selection_owner     sel;
{
    xv_set(frame, FRAME_LEFT_FOOTER, "Lost Selection...", NULL);
}
```

18.9 Sample Selection Requestor Program

Example 18-12 shows three panel items that are created to handle the selection rank, the target(s) request, and initiating the request. Below them is a `textsw` that displays the response to the request.

Example 18-12. Sample selection requestor program – sel_req.c

```
/*
 * sel_req.c: Example of how to make requests to a selection owner for
 *            the selection contents.
 */
#include <stdio.h>
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/font.h>
#include <xview/sel_pkg.h>

Frame          frame;
Textsw         textsw;
Xv_Server      server;
Panel          panel;
Panel_item     p_selection,
               p_target,
               p_request;

Selection_requestor  sel;

#define TARGETS         1<<0
#define TIMESTAMP       1<<1
#define LENGTH          1<<2
#define STRING          1<<3
#define DELETE          1<<4
#define LINE      "----------------------------------------------"

#define ATOM(server, name)   (Atom)xv_get(server, SERVER_ATOM, name)

main(argc, argv)
    int     argc;
    char  **argv;
{
    void       MakeRequest(),
               SelectionReplyProc(),
               RequestChoice();
    Xv_Font    font;

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create((Window)NULL, FRAME,
             XV_X,            10,
             XV_Y,            10,
             XV_LABEL,        "Selection Requestor Example",
             NULL);
    panel = xv_create(frame, PANEL,
                PANEL_LAYOUT,       PANEL_VERTICAL,
                NULL);

    p_selection = xv_create(panel, PANEL_TEXT,
                PANEL_LABEL_STRING,            "Selection:",
                PANEL_VALUE_DISPLAY_LENGTH,     40,
                PANEL_NOTIFY_PROC,              MakeRequest,
                NULL);

    p_target = xv_create(panel, PANEL_TOGGLE,
                PANEL_LABEL_STRING,       "Request:",
                PANEL_NOTIFY_PROC,         RequestChoice,
                PANEL_CHOICE_STRINGS,     "TARGETS",
                                          "TIMESTAMP",
                                          "LENGTH",
                                          "STRING",
                                          "DELETE",
                                           NULL,
                NULL);
    p_request = xv_create(panel, PANEL_BUTTON,
                PANEL_LABEL_STRING,      "Make Request",
                PANEL_NOTIFY_PROC,        MakeRequest,
                PANEL_ITEM_X,             xv_cols(panel, 20),
                NULL);

    window_fit(panel);

    textsw = xv_create(frame, TEXTSW,
                XV_X,          0,
                XV_FONT,       font,
                WIN_BELOW,     panel,
                NULL);

       /* Create a selection requestor object. */
    sel = xv_create(panel, SELECTION_REQUESTOR,
                SEL_REPLY_PROC,       SelectionReplyProc,
                NULL);

    window_fit(frame);
    xv_main_loop(frame);
    exit(0);
}
```

In the requestor application, a `PANEL_TEXT` item is created where the selection rank can be entered by the user. Next, a set of nonexclusive toggle buttons are created to allow the user to choose the target type for the conversion by the selection owner. A “Make Request” button is created that will send a request to the selection owner every time it is pressed. A `TEXTSW` is created to display the output of the request and lastly, a selection-requestor object, `sel`, is created.

Example 18-13 shows the notify procedure associated for the selection requestor example *sel_req.c*. It sets and/or appends selection types to the selection requestor based on the choices selected by the user.

Example 18-13. Sample requestor notify procedure

```
void
RequestChoice(item, value, event)
    Panel_item       item;
    unsigned int     value;
    Event           *event;
{
    int   set = False;

    /* Build the request based on the toggle items the user has selected. */

    if (value & TARGETS) {
        if (set)
              xv_set(sel, SEL_APPEND_TYPE_NAMES, "TARGETS", NULL, NULL);
           else
              xv_set(sel, SEL_TYPE_NAME, "TARGETS", NULL);
              set = True;
    }

    if (value & TIMESTAMP) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "TIMESTAMP", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "TIMESTAMP", NULL);
            set = True;
    }

    if (value & LENGTH) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "LENGTH", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "LENGTH", NULL);
        set = True;
    }

    if (value & STRING) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "STRING", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "STRING", NULL);
        set = True;
    }

    if (value & DELETE) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "DELETE", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "DELETE", NULL);
        set = True;
    }
}
```

The notify procedure, shown in Example 18-14, is associated with the “Make Request” button. It initiates the request by doing three things:

- Getting the selection rank.
- Setting the selection rank.
- Posting a selection request.

Example 18-14. Sample make request notify procedure

```
void
MakeRequest(item, event)
    Panel_item  item;
    Event       *event;
{
    if (item == p_selection) {
        char *rank = NULL;

        /* Set the rank of the selection we are */
        /* going to make requests to.          */
        rank = (char *)xv_get(item, PANEL_VALUE);
        xv_set(sel, SEL_RANK_NAME,        rank,
                         NULL);
    }
    else {
        /* Post a non-blocking request to the selection owner. */
        sel_post_req(sel);
        textsw_erase(textsw, 0, TEXTSW_INFINITY);
    }
}
```

18.9.0.1 Sample reply procedure

Example 18-15 shows the reply procedure for *sel_req.c*.

Example 18-15. Selection reply procedure

```
void
SelectionReplyProc(sel, target, type, value, length, format)
    Selection_requestor   sel;
    Atom                 target;
    Atom                 type;
    Xv_opaque            value;
    unsigned long        length;
    int                  format;
{
    if (length == SEL_ERROR) {
        SelectionError(sel, target, *(int *)value);
        return;
    }
    if (target == ATOM(server, "TARGETS")) {
     textsw_insert(textsw,
              "Holder will convert the following targets:\n", 43);
               /* 43 is the size of the string */
```

Example 18-15. Selection reply procedure (continued)

```
      do {
        Atom *targets = (Atom *)value;
        char *target_name;

        if (targets[--length]) {
            target_name = (char *)xv_get(server, SERVER_ATOM_NAME,
                                   targets[length]);

            textsw_insert(textsw, "\t", 1);
            textsw_insert(textsw, target_name, strlen(target_name));
            textsw_insert(textsw, "\n", 1);
        }
      } while(length);

    }
    else
    if (target == ATOM(server, "TIMESTAMP")) {
        char buf[10];

        textsw_insert(textsw, "TIMESTAMP of acquisition: ", 26);
        sprintf(buf, "%U\n", *(unsigned long *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
    if (target == ATOM(server, "LENGTH")) {
        char buf[10];

        textsw_insert(textsw, "Length of selection: ", 21);
        sprintf(buf, "%d\n", *(int *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
    if (target == ATOM(server, "STRING")) {
        static int   incr = False;

        if (type == ATOM(server, "INCR")) {
            textsw_insert(textsw, "Contents of the selection:\n", 27);
            incr = True;
        }
        else
        if (length) {
            if (!incr)
                textsw_insert(textsw, "Contents of the selection:\n", 27);
            textsw_insert(textsw, (char *)value, length);
            textsw_insert(textsw, "\n", 1);
        }
         else
            incr = False;
    }
    else
    if (target == ATOM(server, "DELETE")) {
        textsw_insert(textsw, "The Selection has been deleted\n", 31);
    }

    textsw_insert(textsw, LINE, strlen(LINE));
}
```

18.9.0.2 Sample error procedure

The error procedure for this example is shown in Example 18-16.

Example 18-16. Sample error procedure – SelectionError.c

```
SelectionError(sel, target, errorCode)
    Selection_requestor    sel;
    Atom        target;
    int              errorCode;
{
    Atom     rank;
    char    *rank_string;
    char    *target_string = (char *)xv_get(server,
                                       SERVER_ATOM_NAME, target);
    char     msg[100];

    rank = (Atom)xv_get(sel, SEL_RANK);
    rank_string = (char *)xv_get(server, SERVER_ATOM_NAME, rank);

    sprintf(msg, "Selection failed for rank ‘‘%s’’ on target ‘‘%s’’: ",
                                    rank_string, target_string);
    textsw_insert(textsw, msg, strlen(msg));

    switch(errorCode) {
      case SEL_BAD_CONVERSION :
        textsw_insert(textsw, "Conversion Rejected",
        strlen("Conversion Rejected"));
        break;

      case SEL_BAD_TIME:
        textsw_insert(textsw, "Bad Time Match",
                 strlen("Bad Time Match"));
        break;

      case SEL_BAD_WIN_ID:
        textsw_insert(textsw, "Bad Window Match",
                 strlen("Bad Window Match"));
        break;

      case SEL_TIMEDOUT:
        textsw_insert(textsw, "Timeout", strlen("Timeout"));
        break;
    }
}
```

18.10 Selection Package Summary

Table 18-3 lists the selection procedures. Table 18-4 lists the selection attributes, including all the attributes for the `SELECTION` package, as well as the `SELECTION_OWNER`, `SELECTION_REQUESTOR`, and the `SELECTION_ITEM` attributes. These attributes and procedures are described fully in the *XView Reference Manual*.

Table 18-3. Selection Procedures

Procedure	Package
`sel_convert_proc()`	Selection Owner
`sel_post_req()`	Selection Requestor

Table 18-4. Selection Attributes

SEL_APPEND_TYPE_NAMES	SEL_PROP_FORMAT
SEL_APPEND_TYPES	SEL_PROP_INFO
SEL_COPY	SEL_PROP_LENGTH
SEL_CONVERT_PROC	SEL_RANK
SEL_DATA	SEL_RANK_NAME
SEL_DONE_PROC	SEL_REPLY_PROC
SEL_FIRST_ITEM	SEL_TIME
SEL_FORMAT	SEL_TIMEOUT_VALUE
SEL_LENGTH	SEL_TYPE
SEL_LOSE_PROC	SEL_TYPE_INDEX
SEL_NEXT_ITEM	SEL_TYPE_NAME
SEL_OWN	SEL_TYPE_NAMES
SEL_PROP_DATA	SEL_TYPES
XV_XID	

19
Drag and Drop

This chapter describes the XView packages that support drag and drop. *Drag and drop* allows data to be moved within an application or between applications. Using drag and drop, objects are selected, then moved or copied to a new location. The selected object is the drag and drop *source*. Selecting the source is referred to as *sourcing the drag*. An area within the source application or within another application *receives the drop* in a pre-registered *drop-site*.

This chapter describes packages that are new for XView Version 3. The function `xv_decode_drop()` that supports dragging and dropping is now obsolete. Note that `xv_decode_drop()` will *not* work correctly once an application is compiled for V3. The function is still provided for backwards compatibility for XView Version 2 programs.

Many OPEN LOOK applications use drag and drop operations. For example, an application may allow text in a text subwindow to be sourced and copied or moved to a new location within the text subwindow. Similarly, a File Manager's file icon can be selected and dragged to the Print Tool icon for printing. OPEN LOOK defines many uses for drag and drop, including loading files into an application. Figure 19-1 shows a file being dragged into an editor and Figure 19-2 shows the application loading the file (receiving the drop).

Drag and drop supports *drop previewing* where the drop-site image changes to show that it is a valid drop-site. It also supports *drag feedback* where the pointer image changes to indicate that an item is being dragged. The data transfer for an XView drag and drop operation is normally implemented using selections. You need to be familiar with the selection package to use drag and drop. The selection package is described in Chapter 18, *Selections*. Note that the operation of drag and drop using OPEN LOOK differs from selections in the following ways:

- Drag and drop is a single gesture rather than a series of separate actions.
- Drag and drop operates on complete pre-registered data objects. Using drag and drop, the application decides which areas may be selected, and then the user may select and drag the area.

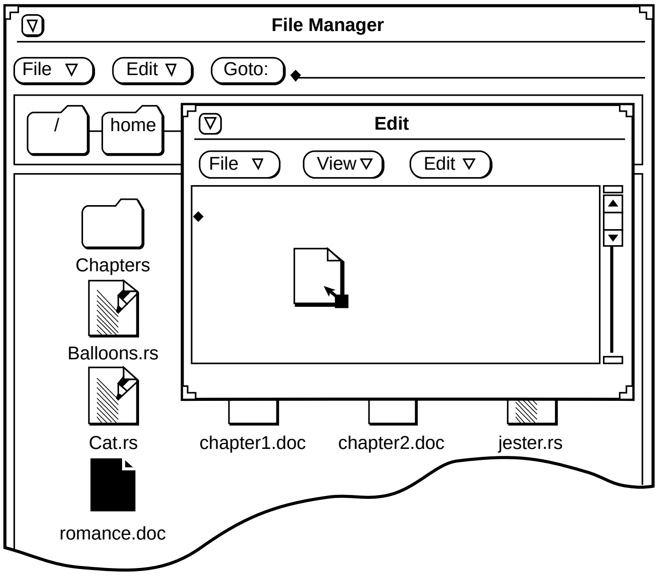
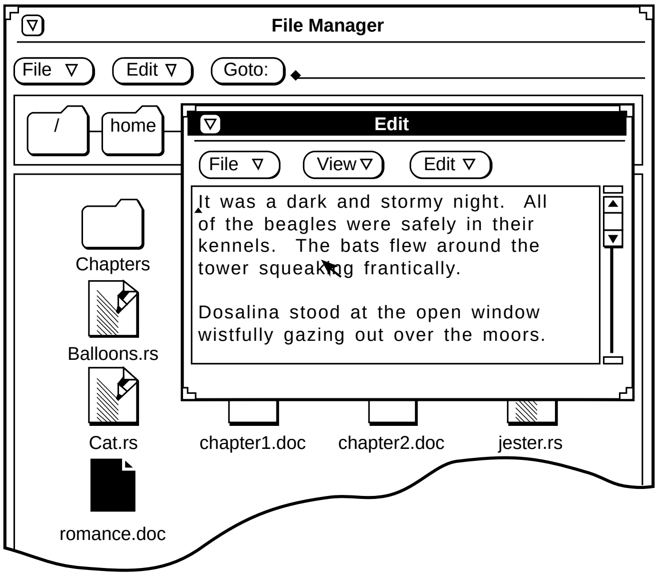


Figure 19-1. Dragging a file onto an application



It was a dark and stormy night. All
of the beagles were safely in their
kennels. The bats flew around the
tower squeaking frantically.

Dosalina stood at the open window
wistfully gazing out over the moors.

Figure 19-2. Loading a file by dropping

19.1 Drag and Drop Objects

Drag and drop is facilitated using two objects: `DRAGDROP` and `DROP_SITE_ITEM`; a `DRAGDROP` object represents the source of the drag. A `DROP_SITE_ITEM` object indicates a destination that is a valid drop-site. All applications need to include the file <*xview/dragdrop.h*> to use these packages. All `DRAGDROP` attributes are prefixed by `DND` and all `DROP_SITE_ITEM` attributes are prefixed by `DROP_SITE`. Both Figure 19-3 and Figure 19-4 show the class hierarchy for these objects.

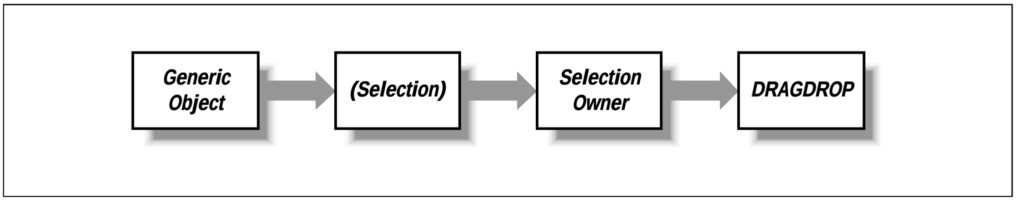


Figure 19-3. DRAGDROP class hierarchy

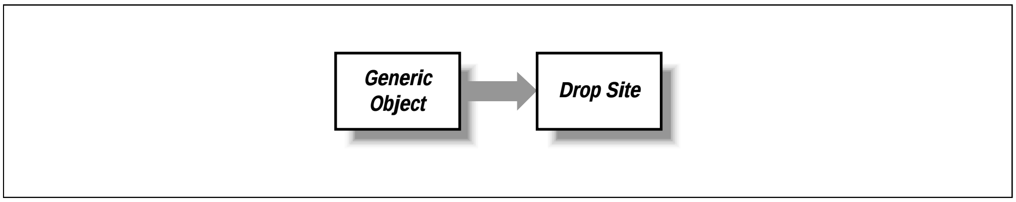


Figure 19-4. DROP_SITE_ITEM class hierarchy

Drag and drop operations involve three steps which the following sections describe:

- The application that is to receive the drop must register a region or several regions as valid drop-sites. This lets the application inform other applications that it is interested in receiving drops.
- The application containing the source for the drag and drop determines the point at which to initiate the drag operation. For example, a drag operation could be initiated when text or a file icon is selected and dragged. A `DRAGDROP` object is created by the source application. The drag begins when the application calls the procedure `dnd_send_drop()`.
- An application that has registered a drop-site must be prepared to receive a drop.

19.2 Registering Drop-sites

An application that is to receive a drop must register a region or several regions as valid drop-sites. This allows drag sources to determine the validity of potential drop-sites and enables the application to receive the events for previewing and dropping. For example, a graphic icon or any other window may be registered as a drop-site. The drop-sites may consist of one region or of several regions described by one or more rectangles within the application. If an application does not register an area as a drop-site, it cannot receive or preview drops.

An object instantiated from the `DROP_SITE_ITEM` package describes a drop-site. A drop-site item is subclassed from the `GENERIC` package and is not a visible object. The following example demonstrates how to make the rectangle representing an entire window a drop-site:

```
drop_site = (Xv_drop_site)xv_create(window, DROP_SITE_ITEM,
              DROP_SITE_ID,          1234,
              DROP_SITE_REGION,      xv_get(window, WIN_RECT),
              DROP_SITE_EVENT_MASK,  DND_ENTERLEAVE | DND_MOTION,
              DROP_SITE_DEFAULT,     TRUE,
              NULL);
```

Instances of `DROP_SITE_ITEM` are owned by window-based objects. The attribute `DROP_SITE_ID` is an uninterpreted ID that may be used by the application to distinguish one drop-site from the next. This attribute is useful when more than one drop-site is set on an object (see Section 19.2.2, "Handling Events," for details on using this attribute). Section 19.2.1, "Adding and Deleting Regions," describes regions and the attribute `DROP_SITE_REGION`. The attribute `DROP_SITE_EVENT_MASK` is a mask set on the drop-site item. It is used to specify if the region(s) within the drop-site should receive preview events (see Section 19.2.2, "Handling Events," for more details).

`DROP_SITE_DEFAULT` establishes a default drop-site for drops forwarded from the window manager. Such drops include drops on icons and window manager decorations. Only one drop-site default should be specified per base frame; specifying more than one will have unpredictable results. Setting this attribute provides only a hint to the window manager. The drop-site default is used most often as a way to have drops forwarded from iconified applications.

19.2.1 Adding and Deleting Regions

Individual drop-site regions should correspond to those objects within the window that may receive drops. For example, if there is a 64x64 icon at (10,10) within a canvas that is an acceptable drop-site, then a drop-site item should be created. The drop-site item's region should be set to a rectangle with coordinates at (10,10) and a size of 64x64. If the position of the objects within the window change or the objects are clipped by other objects, *it is the responsibility of the application* to update the drop-site item to correspond to the new position of these items. For example, if an object is deleted, its region should be removed. If an object is being clipped by its containing window border, or a sibling window border, and is no longer viewable, its region should be removed or updated to reflect the viewable portion of the drop-site. If an object moves, relative to the base frame, the drop-site item region needs to be updated by deleting the old region and adding the new, updated region. Drop-sites obscured by other application windows need not be updated. When the window that owns a drop-site is unmapped, the drop-sites it owns are also unmapped (made “invisible”) to drop sources. They are returned when the window is mapped again.

If an application does not keep the region information up-to-date, users may be given false indication that *old* drop-site areas can receive drops.

If a drop-site item’s owner is destroyed with `xv_destroy()`, any drop-site regions attached to it are also destroyed.

The attribute `DROP_SITE_REGION` associates a region to a drop-site item. `DROP_SITE_REGION` *appends* regions to any existing regions within the drop-site item. The coordinates in the `rect` should be relative to the drop-site item’s owner’s window. An `xv_get()` of a region returns a pointer to an allocated `Rect *` structure. This should be freed using `xv_free()` once the application has finished using it.

The attribute `DROP_SITE_REGION_PTR` is similar to `DROP_SITE_REGION` except that it accepts a `NULL`-terminated array of regions. It appends to any existing regions within the drop-site item. A `NULL` `rect` is defined to be one with width or height equal to 0.

The `DROP_SITE_DELETE_REGION` attribute removes a region from the drop-site item. When a `NULL` is passed as a value for this attribute, all regions in the drop-site are removed.

The `DROP_SITE_DELETE_REGION_PTR` attribute removes a list of regions from the drop-site item. Passing a `NULL` as an argument removes all regions in the drop-site. Typically, when a region needs to be updated, `xv_set()` should be called with two attributes: `DROP_SITE_DELETE_REGION_PTR` with a `NULL` value, followed by `DROP_SITE_REGION_PTR`. Note that the order of these two attributes is important.

19.2.2 Handling Events

The drop-site item’s owner is a window. The event procedure for the owner window receives the drag and drop action events and handles them accordingly. Drag and drop semantic events are shown in Table 19-1.

Table 19-1. Drag and Drop Semantic Events

Event	Purpose
`ACTION_DRAG_PREVIEW`	Preview event.
`ACTION_DRAG_MOVE`	Move the source data to the drop-site window.
`ACTION_DRAG_COPY`	Copy the source data to the drop-site window.

19.2.2.1 Preview events

The attribute DROP_SITE_EVENT_MASK sets a mask on the drop-site item to specify if the region(s) within the drop-site should receive preview events. Preview events indicate the state of the mouse relative to the drop-site. Preview events are indicated when event_action() returns ACTION_DRAG_PREVIEW (in this case, event_id() returns either LOC_WINENTER or LOC_WINEXIT). These events cue the drop-site when the mouse enters or leaves a drop-site region. Drop-sites can also select for LOC_DRAG events that are sent as the mouse moves across a region. These events are delivered to the *event procedure of the drop-site item's owner*, (the pointer coordinates are contained within the event). The coordinates are in the drop-site owner's coordinate space.

DROP_SITE_EVENT_MASK is only a hint since there is no guarantee the source of the drag will send previewing events. When preview events arrive, an application may invert the image of the drop-site. This or some other change that indicates the area is an acceptable drop-site.

An uninterpreted ID specified with the attribute DROP_SITE_ID may be used by the application program to distinguish one drop-site from the next. This attribute is useful when more than one drop-site has been set on an object. This ID is sent along with the ACTION_DRAG_PREVIEW, ACTION_DRAG_MOVE, and ACTION_DRAG_COPY events. If no DROP_SITE_ID is set, the package creates a unique ID for each drop-site region.

19.2.2.2 Event forwarding

The procedure dnd_is_forwarded() lets an application determine if a drop or preview event is forwarded from some other site. This may happen when the user drops on the window manager's decoration window, or on a icon when the application is iconified (only if DROP_SITE_DEFAULT is set). The corresponding drop or preview event will have its DND_FORWARDED flag set.

In general, if an application handles previewing, it should check to see if the preview event was forwarded. If the event was forwarded, it should *not* invert or highlight the drop-site.

19.2.2.3 Handling drop and preview events

A preview event is sent to the drop-site owner when the user is dragging over a registered drop-site. A drop event is sent to the drop-site owner when the user is dragging over a registered drop-site and releases the mouse buttons. There are two ways that the drop-site owner can handle such events in order to preview or to receive the drop:

- The window can set a WIN_EVENT_PROC and define a callback procedure to handle the drop.
- The window can use a notification interposing routine.

If your application uses an interposing routine, two types of interposition procedures must be registered for receiving both drop and previewing events. Both a safe, NOTIFY_SAFE, and an immediate, NOTIFY_IMMEDIATE, interposer needs to be registered. If both safe and immediate interposers are not registered, there are cases when events may be delayed in arriving at the drop-site and may not arrive “on time.”

19.3 Sourcing the Drag

The application containing the source for the drag and drop determines the point to initiate the drag operation. For example, a drag operation could be initiated when text or a file icon is selected and dragged. What events determine the initiation point are entirely up to the application programmer.

Sometime before, or when an application initiates a drag and drop operation it needs to create a drag and drop object and set its attributes. An instance of a `DRAGDROP` object is created using `xv_create()`:

```
dnd_object = xv_create(owner, DRAGDROP,
          <attribute-value list>,
          NULL)
```

A `DRAGDROP` object’s owner is an `Xv_window` and the *<attribute-value list>* may contain drag and drop attributes (`DND` prefix) or since the `DND` package is subclassed from selection owner package, any of the `SELECTION_OWNER` attributes. For example, the drag and drop object `dnd` below uses a selection attribute:

```
dnd = (Xv_drag_drop)xv_create(window, DRAGDROP,
          DND_TYPE,           DND_COPY
          DND_X_CURSOR,       arrow_cursor,
          DND_ACCEPT_CURSOR,  drop_here_cursor,
          SEL_CONVERT_PROC,   SelectionConvert,
          NULL);
```

The owner of the drag and drop object should be the window in which the drag operation is initiated from. Since the `DRAGDROP` object is used as part of the selection transaction that implements the drag and drop operation, the selection transaction must also be completed before the `DRAGDROP` object can be reused.

NOTE

A drag and drop object can only be used in one drag and drop operation at a time.

The attribute `DND_TYPE` defines whether the drag and drop operation will be a copy or a move. This is just a hint to the destination. If the type is a `DND_MOVE` operation and if the destination honors the hint, the destination asks the source to convert the `DELETE` target. The `DND_ACCEPT_CURSOR` attribute, and other cursor related attributes are described in the Section 19.3.3, “Defining the Drag/Accept Cursor.”

Once the `DRAGDROP` object is created, it can be used to initiate a drag and drop operation by calling `dnd_send_drop()`.

19.3.1 Initiating the Drop Operation

To initiate a drag operation, after the drag and drop object is created, an application calls `dnd_send_drop()`. This procedure is defined by XView and is responsible for changing the root cursor, sending previewing events to appropriate drop-sites and sending a *trigger* message, an event of type `ACTION_DRAG_COPY` or `ACTION_DRAG_MOVE`, to the drop-site that is to receive the drop. In sequence, `dnd_send_drop()` does the following:

- Creates a unique selection if this is required.
- Grabs the pointer and changes the cursor.
- Sends preview events to possible drop-sites.
- Sends the *kicker* message to the drop-site to initiate the transfer.
- Insures that the recipient acknowledges the drop.
- Returns a status.

For example, an application such as `filemgr` calls this routine when the user drags the mouse for a number of pixels over a selected file icon.

The form for `dnd_send_drop()` is:

```
int
dnd_send_drop(object)
   Dnd  object;
```

Dnd is a drag and drop object: `dnd_send_drop()` does not return until the user releases all of the mouse buttons, drops the object, or hits the STOP key. The procedure `dnd_send_drop()` returns one of the values shown in Table 19-2.

Table 19-2. dnd_send_drop() Return Values

Return Value	Description
`DND_ILLEGAL_TARGET`	The user dropped on an object that has not registered interest in drag and drop.
`DND_ROOT`	The user dropped on the root window.
`DND_SELECTION`	A unique selection could not be obtained.
`DND_TIMEOUT`	The destination did not respond to the kicker message (the drop).
`DND_ABORTED`	The user aborted the drag operation by hitting the STOP key.
`XV_OK`	The drag and drop operation has begun.
`XV_ERROR`	An unexpected error occurred.

19.3.2 Interaction with the Selection Package

Using drag and drop, data is transferred through selections. Since the DRAGDROP package is subclassed from the SELECTION_OWNER package, it fully supports selections. A selection is associated with the drag and drop object which allows the drop-site item's owner to obtain the data through selection transactions. The selection is normally a transient selection that persists only for the duration of the drag and drop operation. A transient selection should be used even if the dragged object is already associated with a selection such as the primary selection (PRIMARY). If the primary selection were used, and another application acquired the primary selection during the drag and drop transfer, the drag and drop operation would fail.

The application can use the SEL_RANK attribute to associate a selection with the drag and drop object. If the application does not own a selection (SEL_OWN) at the time dnd_send_drop() is called, XView creates a transient selection and associates it with the drag and drop object. This selection can be obtained by using xv_get() on the drag and drop object with the SEL_RANK attribute. *It is the responsibility of the application to disown a selection associated with the drag and drop object when it has completed the operation*. This is regardless of whether the application or the toolkit initially established ownership of the selection.

If XView created a transient selection and dnd_send_drop() failed (did not return XV_OK), the toolkit will disown the selection.

Since the source will be required to reply to requests from the destination for data conversion, the source should have either a conversion procedure or selection items set on the drag and drop object *before* it calls dnd_send_drop().

19.3.3 Defining the Drag/Accept Cursor

The cursor changes when an object is being dragged. The cursor used to indicate dragging an object is set with DND_CURSOR. This attribute changes the pointer used during the drag portion of the drag and drop operation. The default drag cursor is the predefined OPEN LOOK drag cursor. DND_X_CURSOR is an alternative to DND_CURSOR; it accepts an XID of a cursor instead of an Xv_object.

The attribute DND_ACCEPT_CURSOR defines the special "accept" cursor that is used when the mouse is over an acceptable drop-site. The default value for this attribute is a predefined OPEN LOOK drag and drop cursor. DND_ACCEPT_X_CURSOR is an alternative way to set the accept cursor. This attribute accepts an XID of a cursor instead of an Xv_cursor.*

The cursor package also provides special support for drag and drop cursors for text. Refer to Chapter 13, *Cursors*, for information on these special drag and drop cursors.

*Future implementations of this package will also support a DND_REJECT_CURSOR that may be used when drop-sites are currently unreceptive to drops.

19.3.4 Timeout Value

The attribute DND_TIMEOUT_VALUE defines the amount of time to wait for an acknowledgment from the drop destination after the kicker message has been sent to the source (ACTION_DRAG_COPY or ACTION_DRAG_MOVE).

19.4 Receiving a Drop

An application that has registered a drop-site may receive a drop. A window or object that owns the drop-site item may receive one of two events when a valid drop-site is dropped on. An ACTION_DRAG_COPY indicates a copy and an ACTION_DRAG_MOVE indicates a move. These events are the *trigger* message that indicate that a drop has happened.

Upon receiving a drop event, the application should call dnd_decode_drop() and pass in as a parameter the ACTION_DRAG_MOVE or ACTION_DRAG_COPY event that it received along with an instantiated SELECTION_REQUESTOR object. The SELECTION_REQUESTOR object is used since drag and drop data transfers use the selection package to facilitate the transfer.

The dnd_decode_drop() function initiates a data transfer using the selection mechanism. It decodes the drop event as follows:

- It associates the selection rank defined within the drop event with the selection object that was passed into dnd_decode_drop().
- It sends an acknowledgment to the source of the drag and drop, informing it that the transaction has begun.
- It returns the drop-site item that was dropped on if dnd_decode_drop() could begin the drag and drop transaction or returns XV_ERROR if it fails to initiate the drag and drop transaction. This may happen when the selection defined within the drop event does not exist.

The form of the procedure dnd_decode_drop() is:

```
Xv_drop_site
dnd_decode_drop(sel_object, drop_event)
    Selection_requestor    sel_object;
    Event                 *drop_event;
```

If dnd_decode_drop() returns a drop-site item, it is the responsibility of the application to continue the drag and drop transaction. This occurs by making selection requests on the selection object passed into dnd_decode_drop(). See Chapter 18, *Selections*, for details on selection transfers.

The data transfer mechanism for sending and receiving drop data is not limited to using selections. Any mechanism could be used for the transfer, including: the file system, sockets, or one of the other *Alternate Transport Mechanisms.* Typically, such an alternate transfer mechanism would be initiated after the initial selection request.

The macro `dnd_is_local(drop_event)` returns `TRUE` if the source and destination of the drag and drop are the same application. This macro is available so that local (occurring within a single client) drag and drops can be optimized.

19.4.0.1 The move operation

If the application receives an `ACTION_DRAG_MOVE`, it should simulate a move by performing a copy followed by a delete. The delete is initiated by the requestor asking the holder to convert the `DELETE` target. For example, a blocking selection transfer uses the following code:

```
xv_set(sel_object, SEL_TYPE_NAME, "DELETE", NULL);
(void)xv_get(sel_object, SEL_DATA, &length, &format);
```

A non-blocking selection transfer would be initiated as follows:

```
xv_set(sel_object, SEL_TYPE_NAME, "DELETE", NULL);
sel_post_request(sel_object);
```

The application should only ask the owner to delete the selection if the drop event indicates a move operation *and* the application determines it is appropriate to do so. This will typically happen after the application has *successfully* transferred the data. To avoid data loss, it is very important to assure that the data was successfully transferred to the destination application before the data is deleted from the application that is the source of the drag.

19.4.0.2 The done procedure

At the end of the drag and drop operation, after the data has been transferred, the application received the drop must inform XView that the drag and drop operation has been completed. It does so by calling `dnd_done()`:

```
dnd_done(sel_object);
   Selection_requestor sel_object;
```

19.5 Sample Program—Sourcing a Drag

Example 19-1 shows a program that sources a drag. This program and Example 19-2 show how to implement a drop-site item and receive a drop.

Example 19-1. Sourcing a drag

```
/*
 * source1.c - Example of how to source a drag and drop operation.
 *
 */

#include <stdio.h>
#include <sys/types.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/canvas.h>
#include <xview/cursor.h>
#include <xview/dragdrop.h>
#include <xview/sel_pkg.h>
#include <xview/xv_xrect.h>
#include <xview/svrimage.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>

short drop_icon[ ] = {
#include "./drop.icon"
};

#define  POINT_IN_RECT(px, py, rx, ry, rw, rh) \
              ((px) >= rx && (py) >= ry && \
              (px) < rx+rw && (py) < ry+rh)

#define  STRING_MSG  "chromosome: DNA-containing body of the nucleus."

#define  HOST   0
#define  STRING 1
#define  LENGTH 2

Frame          frame;
Canvas     canvas;
Dnd        dnd;
Cursor         arrow_cursor;
Server_image   arrow_image;
Server_image   arrow_image_mask;
Server_image   box_image;
Server_image   drop_here_image;
Cursor         drop_here_cursor;
Selection_owner sel;
Selection_item  selItem[5];
Atom       selAtom[5];

int  SelectionConvert();
extern int sel_convert_proc();

static XColor   fg = {0L, 65535, 65535, 65535};
```

Example 19-1. Sourcing a drag (continued)

```
static XColor   bg = {0L, 0, 0, 0};

typedef struct _DragObject {
    Server_image  image;
    int           x, y;
    unsigned int  w, h;
    int           inverted;
} DragObject;

DragObject  dO;

main(argc, argv)
    int    argc;
    char **argv;
{
    void        EventProc(),
          SelectionLose(),
          PaintCanvas();
    Xv_Server   server;
    Cursor arror_cursor;

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);

    frame = xv_create((Window)NULL, FRAME,
                XV_LABEL,       "Drag & Drop Source",
                XV_X,                10,
                XV_Y,                10,
                FRAME_SHOW_FOOTER,   True,
                0);

    canvas = xv_create(frame, CANVAS,
                        XV_HEIGHT,        100,
                        XV_WIDTH,         300,
                        CANVAS_REPAINT_PROC,    PaintCanvas,
                        CANVAS_X_PAINT_WINDOW,  TRUE,
                        0);

    xv_set(canvas_paint_window(canvas),
                        WIN_BIT_GRAVITY,  ForgetGravity,
                WIN_CONSUME_EVENTS,
                     WIN_MOUSE_BUTTONS,
                     LOC_DRAG,
                               WIN_RESIZE,
                          0,
                        WIN_EVENT_PROC,   EventProc,
                        0);
                /* Create the drag cursor images */
    arrow_image = xv_create(NULL, SERVER_IMAGE,
                SERVER_IMAGE_BITMAP_FILE, "arrow.bm", NULL);
    arrow_image_mask = xv_create(NULL, SERVER_IMAGE,
                SERVER_IMAGE_BITMAP_FILE, "arrow_mask.bm", NULL);
                /* Create a cursor to use in dnd ops. */
    arrow_cursor = XCreatePixmapCursor(XV_DISPLAY_FROM_WINDOW(canvas),
                (XID)xv_get(arrow_image, XV_XID),
                (XID)xv_get(arrow_image_mask, XV_XID),
                &fg, &bg, 61, 3);
    drop_here_image = xv_create(NULL, SERVER_IMAGE,
                XV_WIDTH,       64,
                XV_HEIGHT,      64,
                SERVER_IMAGE_BITS,   drop_icon,
                0);

    drop_here_cursor = XCreatePixmapCursor(
                               XV_DISPLAY_FROM_WINDOW(canvas),
                (XID)xv_get(drop_here_image, XV_XID),
                (XID)xv_get(drop_here_image, XV_XID),
                &fg, &bg, 32, 32);

    dO.image = xv_create(NULL, SERVER_IMAGE,
                SERVER_IMAGE_BITMAP_FILE, "arrowb.bm", NULL);
    dO.w = xv_get(dO.image, XV_WIDTH);
    dO.h = xv_get(dO.image, XV_HEIGHT);
    dO.inverted = False;

    CreateSelection(server, canvas_paint_window(canvas));

    window_fit(frame);

    xv_main_loop(frame);
    exit(0);

}
CreateSelection(server, window)
    Xv_Server   server;
{
    char   name[15];
    int       len;

            /* Primary selection, acquired whenever the arrow
             * bitmap is selected by the user.
             * This is not a requirement for dnd to work.
             */
    sel = xv_create(window, SELECTION_OWNER,
                SEL_RANK,       XA_PRIMARY,
                SEL_LOSE_PROC,       SelectionLose,
                0);

                /* Create the drag and drop object.  */
    dnd = xv_create(window, DRAGDROP,
                DND_TYPE,       DND_COPY,
                DND_X_CURSOR,        arrow_cursor,
                DND_ACCEPT_X_CURSOR, drop_here_cursor,
                SEL_CONVERT_PROC,    SelectionConvert,
                0);

                /* Associate some selection items with the dnd object.*/
    (void) gethostname(name, 15);
    selAtom[HOST] = (Atom)xv_get(server, SERVER_ATOM, "HOST_NAME");
    selItem[HOST] = xv_create(dnd, SELECTION_ITEM,
                SEL_TYPE,       selAtom[HOST],
                SEL_DATA,       (Xv_opaque)name,
```

Example 19-1. Sourcing a drag (continued)

```
                0);

    selAtom[STRING] = (Atom)XA_STRING;
    selItem[STRING] = xv_create(dnd, SELECTION_ITEM,
                SEL_TYPE,       selAtom[STRING],
                SEL_DATA,       (Xv_opaque)STRING_MSG,
                0);

    len = strlen(STRING_MSG);
    selAtom[LENGTH] = (Atom)xv_get(server, SERVER_ATOM, "LENGTH");
    selItem[LENGTH] = xv_create(dnd, SELECTION_ITEM,
                SEL_TYPE,       selAtom[LENGTH],
                SEL_FORMAT,          sizeof(int)*NBBY,
                SEL_LENGTH,          1,
                SEL_DATA,       (Xv_opaque)&len,
                0);
}

void
EventProc(window, event)
Xv_Window       window;
Event           *event;
{
    static int drag_pixels = 0;
    static int dragging = False;

    switch (event_action(event)) {
      case ACTION_SELECT:
     if (event_is_down(event)) {
         dragging = False;
                /* If the user selected our dnd object, highlight
                 * the box and acquire the primary selection.
                 */
         if (POINT_IN_RECT(event_x(event), event_y(event),
                          dO.x, dO.y, dO.w, dO.h)) {
          xv_set(sel, SEL_OWN, True, 0);
          dO.inverted = True;
          PaintObject(dO, xv_get(window, XV_XID),
                                 XV_DISPLAY_FROM_WINDOW(window));
         } else
          /* If the user selected outside of the dnd object,
           * de-highlight the object. And release the primary
           * selection.
           */
          xv_set(sel, SEL_OWN, False, 0);
     } else
         drag_pixels = 0;
        break;
      case LOC_DRAG:
                /* If the user dragged at least five pixel over our
                 * dnd object, begin the dnd operation.
                 */
        if (event_left_is_down(event)) {
         if (POINT_IN_RECT(event_x(event),
                      event_y(event),dO.x,dO.y,dO.w,dO.h))
          dragging = True;
```

Example 19-1. Sourcing a drag (continued)

```
            if (dragging && drag_pixels++ == 5) {
             xv_set(frame, FRAME_LEFT_FOOTER, "Drag and Drop:", 0);
             switch (dnd_send_drop(dnd)) {
            case XV_OK:
                xv_set(frame, FRAME_LEFT_FOOTER,
                          "Drag and Drop: Began", 0);
                break;
               case DND_TIMEOUT:
                xv_set(frame, FRAME_LEFT_FOOTER,
                          "Drag and Drop: Timed Out",0);
                break;
               case DND_ILLEGAL_TARGET:
                xv_set(frame, FRAME_LEFT_FOOTER,
                          "Drag and Drop: Illegal Target",0);
                break;
               case DND_SELECTION:
                xv_set(frame, FRAME_LEFT_FOOTER,
                          "Drag and Drop: Bad Selection",0);
                break;
               case DND_ROOT:
                xv_set(frame, FRAME_LEFT_FOOTER,
                          "Drag and Drop: Root Window",0);
                break;
               case XV_ERROR:
                xv_set(frame, FRAME_LEFT_FOOTER,
                          "Drag and Drop: Failed",0);
                break;
             }
            drag_pixels = 0;
            }
      }
        break;
    }
}

PaintObject(object, win, dpy)
    DragObject   object;
    Window  win;
    Display     *dpy;
{
    static GC   gc;
    static int  gcCreated = False;

    if (!gcCreated) {
        XGCValues gcv;
        gcv.stipple = (Pixmap) xv_get(object.image, XV_XID);
        gcv.fill_style = FillStippled;
        gc = XCreateGC(dpy, win, GCStipple|GCForeground|GCBackground|
                                   GCFillStyle, &gcv);
        XSetForeground(dpy, gc, BlackPixel(dpy, XDefaultScreen(dpy)));
        XSetBackground(dpy, gc, WhitePixel(dpy, XDefaultScreen(dpy)));
    }

    if (object.inverted) {
     XSetFillStyle(dpy, gc, FillSolid);
```

Example 19-1. Sourcing a drag (continued)

```
      XDrawRectangle(dpy, win, gc, object.x-1, object.y-1, 66, 66);
      XSetFillStyle(dpy, gc, FillStippled);
    } else
      XClearWindow(dpy, win);

    XSetTSOrigin(dpy, gc, object.x, object.y);
    XFillRectangle(dpy, win, gc, object.x, object.y, 65, 65);
}

void
PaintCanvas(canvas, paint_window, dpy, xwin, xrects)
    Canvas        canvas;         /* unused */
    Xv_Window     paint_window;   /* unused */
    Display      *dpy;
    Window        xwin;
    Xv_xrectlist *xrects;         /* unused */
{
    unsigned    width, height;
    int         x, y;

    width = xv_get(paint_window, XV_WIDTH);
    height = xv_get(paint_window, XV_HEIGHT);

    x = (width/2)-(dO.w/2);
    y = (height/2)-(dO.h/2);

    dO.x = x;
    dO.y = y;

    PaintObject(dO, xwin, dpy);
}
/* The convert proc is called whenever someone makes a request
 * to the dnd selection.  Two cases we handle within the convert
 * proc: DELETE and _SUN_SELECTION_END.  Everything else we pass
 * on to the default convert proc which knows about our selection
 * items.
 */
int
SelectionConvert(seln, type, data, length, format)
    Selection_owner   seln;
    Atom       *type;
    Xv_opaque        *data;
    long       *length;
    int              *format;
{
    Xv_Server         server = XV_SERVER_FROM_WINDOW(xv_get(seln,
                                    XV_OWNER));

    if (*type == (Atom)xv_get(server,
                       SERVER_ATOM, "_SUN_SELECTION_END")) {
          /* Destination has told us it has completed the drag
          * and drop transaction.  We should respond with a
          * zero-length NULL reply.
          */
      xv_set(dnd, SEL_OWN, False, 0);
      xv_set(frame, FRAME_LEFT_FOOTER, "Drag and Drop: Completed",0);
```

Example 19-1. Sourcing a drag (continued)

```
     *format = 32;
     *length = 0;
     *data = NULL;
     *type = (Atom)xv_get(server, SERVER_ATOM, "NULL");
     return(True);
    } else if (*type == (Atom)xv_get(server,
                                SERVER_ATOM, "DELETE")) {
                /* Destination asked us to delete the selection.
                 * If it is appropriate to do so, we should.
                 */
     *format = 32;
     *length = 0;
     *data = NULL;
     *type = (Atom)xv_get(server, SERVER_ATOM, "NULL");
     return(True);
    } else
                /* Let the default convert procedure deal with the
                 * request.
                 */
     return(sel_convert_proc(seln, type, data, length, format));
}

/* When we lose the primary selection, this procedure is called.
 * We dehigh-light our selection.
 */
void
SelectionLose(seln)
    Selection_owner seln;
{
    Xv_Window   owner = xv_get(seln, XV_OWNER);

    if (xv_get(seln, SEL_RANK) == XA_PRIMARY) {
        dO.inverted = False;
        PaintObject(dO, xv_get(owner,
                          XV_XID), XV_DISPLAY_FROM_WINDOW(owner));
    }
}
```

19.6 Sample Program—Drop Site Item and Destination

Example 19-2 provides a example of a program that creates a drop-site item and sets up to receive a drop.

Example 19-2. A drop-site item example

```
/*
 * dest.c - Example of how to register interest in receiving
 *          drag and drop events and how to complete a drag
 *          and drop operation.
 *
 */
#include <stdio.h>
```

Example 19-2. A drop-site item example (continued)

```
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/canvas.h>
#include <xview/panel.h>
#include <xview/font.h>
#include <xview/dragdrop.h>
#include <xview/xv_xrect.h>
#include <X11/Xlib.h>

#define  DROP_WIDTH      65
#define  DROP_HEIGHT     65

#define  BULLSEYE_SITE      1

Frame         frame;
Canvas        canvas;
Panel         panel;
Xv_drop_site  drop_site;
Server_image  drop_image;
Server_image  drop_image_inv;
Panel_item    p_string,
              p_length,
              p_host;

Selection_requestor  sel;

int      inverted;

main(argc, argv)
    int argc;
    char **argv;
{
    void      EventProc(),
            PaintCanvas(),
            ResizeCanvas();
    Xv_Font      font;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create((Window)NULL, FRAME,
                        XV_X,       330,
                        XV_Y,       10,
                        XV_WIDTH,   10,
                        XV_LABEL,   "Drag & Drop Destination",
                        NULL);


    panel = xv_create(frame, PANEL,
                        PANEL_LAYOUT,  PANEL_VERTICAL,
                        NULL);

    p_string = xv_create(panel, PANEL_TEXT,
                        PANEL_LABEL_STRING,       "Dropped Text:",
                        PANEL_VALUE_DISPLAY_LENGTH, 50,
                        NULL);
    p_host = xv_create(panel, PANEL_TEXT,
```

Example 19-2. A drop-site item example (continued)

```
                    PANEL_LABEL_STRING,       "From host:",
                    PANEL_VALUE_DISPLAY_LENGTH, 15,
                    NULL);
p_length = xv_create(panel, PANEL_TEXT,
                    PANEL_LABEL_STRING,       "Length:",
                    PANEL_VALUE_DISPLAY_LENGTH, 6,
                    NULL);

window_fit(panel);

canvas = xv_create(frame, CANVAS,
                    XV_HEIGHT,      100,
                    XV_WIDTH,       WIN_EXTEND_TO_EDGE,
                    XV_X,           0,
                        WIN_BELOW,  panel,
                        CANVAS_REPAINT_PROC,    PaintCanvas,
                        CANVAS_RESIZE_PROC,     ResizeCanvas,
                        CANVAS_X_PAINT_WINDOW,  TRUE,
                        NULL );

xv_set(canvas_paint_window(canvas),
                        WIN_BIT_GRAVITY,        ForgetGravity,
                        WIN_CONSUME_EVENTS,
                                      WIN_RESIZE,
                                      NULL,
                        WIN_EVENT_PROC,  EventProc,
                        NULL);

drop_image = xv_create(NULL, SERVER_IMAGE,
                        SERVER_IMAGE_BITMAP_FILE, "./bullseye.bm",
                        NULL);

drop_image_inv = xv_create(NULL, SERVER_IMAGE,
                        SERVER_IMAGE_BITMAP_FILE, "./bullseyeI.bm",
                        NULL);

       /* Selection requestor object that will be
       * passed into dnd_decode_drop() and later used
       * to make requests to the source of the
       * drop.
       */
sel = xv_create(canvas, SELECTION_REQUESTOR, NULL);

       /* This application has one drop site with
       * site id BULLSEYE_SITE and whose shape will
       * be described by a rectangle.  If
       * animation is supported, it would like to
       * receive LOC_DRAG, LOC_WINENTER and
       * LOC_WINEXIT events.
       */
drop_site = xv_create(canvas_paint_window(canvas), DROP_SITE_ITEM,
                        DROP_SITE_ID,          BULLSEYE_SITE,
                        DROP_SITE_EVENT_MASK,   DND_ENTERLEAVE,
                        NULL);

inverted = False;
```

Example 19-2. A drop-site item example (continued)

```
    window_fit(frame);
    xv_main_loop(frame);
    exit(0);
}

void
EventProc(window, event)
    Xv_Window       window;
    Event           *event;
{
    switch (event_action(event)) {
           /* When drop previewing is available, if
           * the drop site has selected for previewing
           * events (DROP_SITE_EVENT_MASK) then it will
           * receive ACTION_DRAG_PREVIEW events from
           * the source as requested.
           */
      case ACTION_DRAG_PREVIEW:
       switch(event_id(event)) {
         case LOC_WINENTER:
           inverted = True;
           break;
         case LOC_WINEXIT:
           inverted = False;
           break;
         case LOC_DRAG:
           break;
       }
       PaintCanvas(NULL, window, XV_DISPLAY_FROM_WINDOW(window),
                  xv_get(window, XV_XID), NULL);
       break;
      case ACTION_DRAG_COPY:
      case ACTION_DRAG_MOVE: {
       Xv_drop_site  ds;
       Xv_Server  server = XV_SERVER_FROM_WINDOW(event_window(event));

           /* If the user dropped over an acceptable
           * drop site, the owner of the drop site will
           * be sent an ACTION_DROP_{COPY, MOVE} event.
           */
           /* To acknowledge the drop and to associate the
           * rank of the source’s selection to our
           * requestor selection object, we call
           * dnd_decode_drop().
           */
       if ((ds = dnd_decode_drop(sel, event)) != XV_ERROR) {

           if (xv_get(ds, DROP_SITE_ID) == BULLSEYE_SITE)
              UpdatePanel(server, sel);

            /* If this is a move operation, we must ask
            * the source to delete the selection object.
            * We should only do this if the transfer of
            * data was successful.
            */
           if (event_action(event) == ACTION_DRAG_MOVE) {
              int length, format;

                xv_set(sel, SEL_TYPE_NAME, "DELETE", NULL);
                (void)xv_get(sel, SEL_DATA, &length, &format);
           }

            /* To complete the drag and drop operation,
            * we tell the source that we are all done.
            */
           dnd_done(sel);
           inverted = False;
           PaintCanvas(NULL, window, XV_DISPLAY_FROM_WINDOW(window),
                      xv_get(window, XV_XID), NULL);

       } else
           printf ("drop error\n");
       break;
      }
      default:
        break;
    }
}


UpdatePanel(server, sel)
    Xv_Server              server;
    Selection_requestor       sel;
{
    int     length,
            format,
           *string_length;
    char    buf[7],
           *string,
           *hostname;

    xv_set(sel, SEL_TYPE, XA_STRING, NULL);
    string = (char *)xv_get(sel, SEL_DATA, &length, &format);
    if (length != SEL_ERROR) {
           xv_set(p_string, PANEL_VALUE, string, NULL);
           free (string);
    }

    xv_set(sel,
            SEL_TYPE, xv_get(server, SERVER_ATOM, "LENGTH"), NULL);
    string_length = (int *)xv_get(sel, SEL_DATA, &length, &format);
    if (length != SEL_ERROR) {
       sprintf(buf, "%d", *string_length);
        xv_set(p_length, PANEL_VALUE, buf, NULL);
        free ((char *)string_length);
    }

    xv_set(sel, SEL_TYPE, xv_get(server, SERVER_ATOM, "HOST_NAME"), NULL);
    hostname = (char *)xv_get(sel, SEL_DATA, &length, &format);
    if (length != SEL_ERROR) {
           xv_set(p_host, PANEL_VALUE, hostname, NULL);
```

Example 19-2. A drop-site item example (continued)

```
            free (hostname);
    }
    xv_set(sel, SEL_TYPE_NAME, "_SUN_SELECTION_END", NULL);
    (void)xv_get(sel, SEL_DATA, &length, &format);
}

void
PaintCanvas(canvas, paint_window, dpy, xwin, xrects)
    Canvas        canvas;         /* unused */
    Xv_Window     paint_window;   /* unused */
    Display      *dpy;
    Window        xwin;
    Xv_xrectlist *xrects;         /* unused */
{
    static GC   gc;
    static int  gcCreated = False;
    static int  lastMode = False;
    int         width, height;
    int         x, y;
    Rect       *r;

    if (!gcCreated) {
        XGCValues gcv;
        gcv.stipple = (Pixmap) xv_get(drop_image, XV_XID);
        gcv.foreground = BlackPixel(dpy, XDefaultScreen(dpy));
        gcv.background = WhitePixel(dpy, XDefaultScreen(dpy));
        gcv.fill_style = FillStippled;
        gc = XCreateGC(dpy, xwin, GCStipple|GCForeground|GCBackground|
                                  GCFillStyle, &gcv);
    }

    if (lastMode != inverted) {
       if (!inverted)
           XSetStipple(dpy, gc, (Pixmap) xv_get(drop_image, XV_XID));
       else
           XSetStipple(dpy, gc, (Pixmap) xv_get(drop_image_inv,
                                                 XV_XID));
       lastMode = inverted;
    }

    width = xv_get(paint_window, XV_WIDTH);
    height = xv_get(paint_window, XV_HEIGHT);

    x = (width/2)-(DROP_WIDTH/2);
    y = (height/2)-(DROP_HEIGHT/2);

    XClearArea(dpy, xwin, x, y, DROP_WIDTH, DROP_HEIGHT, False);
    XSetTSOrigin(dpy, gc, x, y);
    XFillRectangle(dpy, xwin, gc, x, y, DROP_WIDTH, DROP_HEIGHT);
}

void
ResizeCanvas(canvas, width, height)
    Canvas        canvas;
    int         width;
    int         height;
```

Example 19-2. A drop-site item example (continued)

```
{
    int         x, y;
    Rect        rect;

    x = (width/2)-(DROP_WIDTH/2);
    y = (height/2)-(DROP_HEIGHT/2);

    rect.r_left = x;
    rect.r_top = y;
    rect.r_width = DROP_WIDTH;
    rect.r_height = DROP_HEIGHT;

                     /* Update the drop site information. */
    xv_set(drop_site, DROP_SITE_DELETE_REGION_PTR, NULL,
                    DROP_SITE_REGION, &rect,
                    NULL);
}
```

19.7 Drag and Drop Package Summary

Table 19-3 lists the procedures and macros used for dragging and dropping. Table 19-4 lists the `DROP_SITE_ITEM` and `DRAGDROP` package attributes. These attributes, procedures, and macros are described fully in the *XView Reference Manual*.

Table 19-3. DROP_SITE_ITEM and DRAGDROP Procedures and Macros

`dnd_decode_drop()`
`dnd_done()`
`dnd_is_forwarded()`
`dnd_is_local()`
`dnd_send_drop()`

Table 19-4. DROP_SITE_ITEM and DRAGDROP Attributes

DROP_SITE_ITEM Attributes	DRAGDROP Attributes
`DROP_SITE_DEFAULT`	`DND_ACCEPT_CURSOR`
`DROP_SITE_DELETE_REGION`	`DND_ACCEPT_X_CURSOR`
`DROP_SITE_DELETE_REGION_PTR`	`DND_CURSOR`
`DROP_SITE_EVENT_MASK`	`DND_TIMEOUT_VALUE`
`DROP_SITE_ID`	`DND_TYPE`
`DROP_SITE_REGION`	`DND_X_CURSOR`
`DROP_SITE_REGION_PTR`	

20 The Notifier

In this chapter, we look at the Notifier in greater detail, discussing its role in processing events for an XView application. This chapter serves as an introduction to the Notifier and covers its use for most applications. You should be familiar with the topics covered in Chapter 6, *Handling Input*, before you read this chapter.

20.1 Basic Concepts

The Notifier maintains the flow of control in an application. To understand the basic concepts of the Notifier, we must distinguish between two different styles of input handling, *mainline* and *event-driven* input, and consider how they affect where the flow of control resides within a program.

20.1.1 Mainline Input Handling

The traditional type of input handling of most text-based applications is mainline-based and input-driven. The flow of control resides in the main routine and the program *blocks* when it expects input. That is to say, no other portion of the program may be executed while the program is waiting for input. For example, in a mainline-driven application, a C programmer will use `fgets()` or `getchar()` to wait for characters that the user types. Based on the user's input, the program chooses an action to take. Sometimes, that action requires more input, so the application calls `getchar()` again. The program does not return to the main routine until the processing for the current input is done.

The tight control represented by this form of input handling is the easiest to program since you have control at all times over what to expect from the user and you can control the direction that the application takes. There is only one source of input—the keyboard—and the user can only respond to one interface element at a time. A user's responses are predictable in the sense that you know that the user is going to type *something*, even if you do not know what it is.

20.1.2 Event-driven Input Handling

Windowing systems are designed such that many sources of input are available to the user at any given time. In addition to the keyboard, there are other input devices, such as the mouse. Each keystroke and mouse movement causes an *event* that the application might consider. These keystroke and mouse events are generated from the window system. Further, there are other types of events that are generated from the window system itself and from other processes. Another aspect of event-driven input handling is that you are not guaranteed to have any predictable sequence of events from the user. That is, a user can position the mouse on an object that receives text as input. Before the user is done typing, the user can move the mouse to another window and select a panel button of some sort. The application cannot (and should not) expect the user to type in window *A* first, then move to window *B* and select the button. A well-written program should expect input from any window to happen at any time.

20.2 Functions of the Notifier

The Notifier can do any of the following:

- Handle *software interrupts*—specifically, UNIX signals such as `SIGINT` or `SIGCONT`.
- Notice state changes in processes that your process has spawned (e.g., a child process that has died).
- Read and write through file descriptors (e.g., files, pipes, and sockets).
- Receive notification of the expiration of timers so that you can regularly flash a caret or display animation.
- Extend, modify, or monitor XView Notifier clients (e.g., noticing when a frame is opened, closed, or about to be destroyed.)
- Use a non-notification-based control structure while running under XView (e.g., porting programs to XView).

The Notifier also has provisions, to a limited degree, to allow programs to run in the Notifier environment without inverting their control structure.

20.3 How the Notifier Works

Up until now, we have been saying that you should register an event handler for objects when they want to be notified of certain events such as mouse motion or selection or keyboard input. What you may not have been aware of is that you are indirectly registering these event handlers with the Notifier. When you specify *callbacks* or *notify procedures*, the XView object specified is said to be the *client* of the Notifier. Look at the following code:

```
extern void my_event_handler();

xv_set(canvas,
    CANVAS_PAINTWINDOW_ATTRS,
        WIN_CONSUME_X_EVENT_MASK,  ButtonPressMask | KeyPressMask,
        WIN_EVENT_PROC,            my_event_handler,
        NULL,
    NULL);
```

In the above code, each paint window of the canvas becomes a client of the Notifier.*

Generally stated, the Notifier detects events in which its clients have expressed an interest and dispatches these events to the proper clients in a predictable order. In the X Window System, events are delivered to the application by the X server. In XView, it is the Notifier that receives the events from the server and dispatches them to its clients. After the client's notify procedure processes the event, control is returned to the Notifier.

20.3.1 Restrictions

The Notifier imposes some restrictions on its clients. Designers should be aware of these restrictions when developing software to work in the Notifier environment. These restrictions exist so that the application and the Notifier do not interfere with each other. More precisely, since the Notifier is multiplexing access to user process resources, the application needs to respect this effort so as not to violate the sharing mechanism.

For example, a client should not call `signal` (3). The Notifier is catching signals on behalf of its clients. If a client sets up its own signal handler, then the Notifier will never notice the signal. The program should call `notify_set_signal_func()` instead of `signal` (3) (see Section 20.5, "Signal Handling").

*`CANVAS_PAINTWINDOW_ATTRS` tells the `CANVAS` package to register the input mask and callback routine for each of its paint windows.

20.3.1.1 System calls to avoid

Assuming an environment with multiple clients and an unknown Notifier usage pattern, you should not use any of the following system calls or C library routines:

`signal` (3)	The Notifier is catching signals on behalf of its clients. If you set up your own signal handler over the one that the Notifier has set up, then the Notifier will never notice the signal.
`sigvec` (2)	The same applies for `sigvec` (2) as for `signal` (3) above.
`sigaction` (2)	The same applies for `sigaction` (2) as for `signal` (3) above.
`setitimer` (2)	The Notifier is managing two of the process's interval timers on behalf of its many clients. If you access an interval timer directly, the Notifier could miss a timeout. Use `notify_set_itimer_func()` instead of `setitimer` (2).
`alarm` (3)	Because `alarm` (3) sets the process's interval timer directly, the same applies here as for `setitimer` (2) above.
`getitimer` (2)	When using a Notifier-managed interval timer, you should call `notify_itimer_value()` to get its current status. Otherwise, you can get inaccurate results.
`wait3` (2)	The Notifier notices child process state changes on behalf of its clients. If you do your own `wait3` (2), then the Notifier may never notice the change in a child process or you may get a change of state for a child process in which you have no interest. Use `notify_set_wait3_func()` instead of `wait3` (2).
`wait` (2)	The same applies for `wait` (2) as does for `wait3` (2) above.
`ioctl` (2) `( . . . , FIONBIO, . . . )`	This call sets the blocking status of a file descriptor. The Notifier needs to know the blocking status of a file descriptor in order to determine if there is activity on it. `fcntl` (2) has an analogous request that should be used instead of `ioctl` (2).
`ioctl` (2) `( . . . , FIOASYNC, . . . )`	This call controls a file descriptor's asynchronous IO (input/output) mode setting. The Notifier needs to know this mode in order to determine if there is activity on it. `fcntl` (2) has an analogous request that should be used instead of `ioctl` (2).
`popen and pclose` (2)	In the SunOS, these functions call `wait` (2). Hence, you should avoid using these for the reasons mentioned above.
`system` (3)	In the SunOS, this function calls `signal` (3) and `wait` (2). Hence, you should avoid using this for the reasons mentioned above.

20.4 What is a Notifier Client?

A *client* of the Notifier is anything that has registered a callback routine with it. In XView, a client is an object such as a canvas or panel that you have created using `xv_create()`. Typically, most of the event registration happens at the time such clients are created. However, to the Notifier, a client is nothing more than an ID that distinguishes it from all other Notifier clients. Thus, you could identify a client using a number such as "43" or the address of an object as in `&foo`. XView objects are commonly used as Notifier clients because `xv_create()` returns a unique handle to an object that has been allocated dynamically.

The client ID is of type `Notify_client` as declared in <*xview/notify.h*>.

Certain `notify_*` functions create Notifier clients. For example:

```
Notify_client  client = (Notify_client)10101; /* arbitrary */
Notify_func     destroy_func();

notify_set_destroy_func(client, destroy_func);
```

In this code fragment, up until the call to `notify_set_destroy_func()`, the `client` may not have been registered as a client to the Notifier. When you call `notify_set_destroy_func()`, internally, the Notifier package looks up in its table of clients whether there is any other client using that identifier (the client handle is the identifier, in this case this is an arbitrary value). If there is a client using the identifier, then the destroy function specified is set for that client's destroy handling. Otherwise, a new Notifier client is allocated and the client handle is also used as a handle to the new Notifier client.

Notifier clients are *not* XView objects for you to create or manipulate using `xv_create()` or `xv_set()`. In fact, the Notifier is completely independent of XView and can be used in applications that do not even use XView objects.

20.4.1 Types of Interaction

Client interaction with the Notifier falls into two general categories:

- Event Handling – A client may receive events and respond to them using event handlers. As you have seen, event handlers (callbacks and notify procedures) do much of the work in the Notifier environment.
- Interposition – A client may request that the Notifier install a special type of event handler, supplied by the client, to be *interposed* ahead of the current event handler for a given type of event and client. This allows clients to screen incoming events and redirect them and to monitor and change the status of other clients. A thorough discussion of interposition is presented later in this chapter.

A client establishes an interest in a certain type of event by registering an event handler to respond to it. The following sections cover registration procedures for special signal and UNIX-related event handlers, for application-defined client event handlers, and for interposing event handlers. All event handlers, including interposers, return a value of `NOTIFY_DONE`, `NOTIFY_IGNORED`, or `NOTIFY_UNEXPECTED`, depending on how the event was handled. These three return values are sometimes used to convey important information about the status of the event in the Notifier (usually the function that called `notify_next_*_func()` looks at the return value). These return values and their significance are covered in Section 20.9.4, "Invoking the Next Function."

20.5 Signal Handling

Signals are UNIX software interrupts. The Notifier multiplexes access to the UNIX signal mechanism. A Notifier client may ask to be notified that a UNIX signal occurred either when it is received (asynchronously) and/or later during normal processing (synchronously).

Clients may define and register a signal event handler to respond to any UNIX signal desired. However, some of the signals that you might catch in a traditional UNIX program should be caught instead by the Notifier.

CAUTION

> Clients of the Notifier should not directly catch any UNIX signals using `signal` (3), `sigvec` (2), or `sigaction` (2). There are critical stages of event reading and dispatching that, if interrupted, could cause the program to jump to another location and interrupt the communication protocol between the X server and the application.

Exceptions to this are noted later in this section.

20.5.1 Signals to Avoid

Clients should not have to catch any of the following signals (even via `notify_set_signal_func()` described below). If they are, you are probably utilizing the Notifier inappropriately. The Notifier catches these signals itself under a variety of circumstances and handles them appropriately.

`SIGALRM`	Caught by the Notifier's interval timer manager. Use `notify_set_itimer_func()` instead.
`SIGVTALRM`	The same applies for `SIGVTALRM` as for `SIGALRM` above.
`SIGTERM`	Caught by the Notifier so that it can tell its clients that the process is going away. Use `notify_set_destroy_func()` if that is why you are catching `SIGTERM`.
`SIGCHLD`	Caught by the Notifier so that it can do child process management. Use `notify_set_wait3_func()` instead.

SIGIO	Caught by the Notifier so that it can manage its file descriptors that are running in asynchronous I/O mode. Use `notify_set_input_func()` or `notify_set_output_func()` if you want to know when there is activity on your file descriptor.
SIGURG	Caught by the Notifier so that it can dispatch exception activity on a file descriptor to its clients. Use `notify_set_exception_func()` if you are looking for out-of-band communications when using a socket.

The last two signals in the list are considered advanced topics and are not covered in this manual.

20.5.2 A Replacement for signal()

Instead of using `signal()` to catch signals delivered by UNIX, you should register a signal event handler by calling `notify_set_signal_func()`. This function allows you to call another function when a specified signal is generated. Its form is as follows:

```
Notify_func
notify_set_signal_func(client, signal_func, sig, when)
    Notify_client      client;
    Notify_func        signal_func;
    int                sig;
    Notify_signal_mode when;
```

`signal_func` is the function to call when the signal described by `sig` occurs.*

The `when` parameter is either `NOTIFY_SYNC` or `NOTIFY_ASYNC`. `NOTIFY_SYNC` causes notification during normal processing. In other words, the delivery of the signal is delayed to avoid interrupting Xlib Protocol communication between the X server and the application. When it is safe to do so, your `signal_func` function is called and you can display a notice, exit, or jump to another place in the program. Typically, there is a very short time between signal delivery and notification to your callback routine. It is only a little slower than when you use `signal()`.

`NOTIFY_ASYNC` causes notification as soon as the signal is received. This mode mimics the UNIX `signal` (3) semantics.

CAUTION

When using asynchronous signals, your routine may set a variable, a condition, or a flag indicating that the signal was received, or it may change any internal state to your program. Do *not* make any XView, Xlib, or Notifier calls or call any function that might manipulate any XView data structures. Also, do not call `longjmp()` or `setjmp()`; they can cause a condition that can interfere with the X11 Protocol.

*See <*signal.h*> for a list of signals and definitions.

`notify_set_signal_func()` returns a pointer to the function that was installed before you set the new function. You should use this to reset the function if you want to unregister your installed function.

When the specified signal occurs, your `signal_func` is called:

```
Notify_value
signal_func(client, sig, when)
    Notify_client      client;
    int                sig;
    Notify_signal_mode when;
```

The parameters to the signal handler are not the same as the ones given to a signal handler in the call to `signal`(3). However, it is not advisable to use either one, except in unusual circumstances. As a general rule, you should use `notify_set_signal_func()` for all signal handling. If you want the signal *code* or *context* from the signal that was generated (two parameters that are passed to a normal UNIX signal handler), you can use:

```
int
notify_get_signal_code()

struct sigcontext *
notify_get_signal_context()
```

These two functions take no parameters—they return the signal code and context (respectively) of the last signal generated. If you wish to save these values, you can copy them.

Using the Notifier, you can catch any signal except `SIGKILL` and `SIGSTOP`, which cannot be caught by a UNIX application. Attempting to do so generates an error.

An example of common signal handling is shown below:

```
#include <xview/notify.h>

 ...
extern Notify_value sigint_func();

notify_set_signal_func(frame, sigint_func, SIGINT, NOTIFY_SYNC);
 ...

Notify_value
sigint_func(client, sig, when)
Notify_client client;
int sig;
{
    puts("received interrupt -- exiting");
    xv_destroy_safe(frame);
    return NOTIFY_DONE;
}
```

The return value of your signal handler tells whether you handled the signal or ignored it.*

*Currently, the return value is ignored.

20.5.3 Timers

One specific type of signal is a *timer*. Timers can be set up to call a routine after the passage of a specified amount of time.* Such a routine may cause a caret to flash at regular intervals or allow a clock application to change the time display. The timer is handled differently from other signals in that multiple timers can be installed for various clients (say a flashing caret and a clock in the same application). There can only be one timer function (`timer_func`) per client.

Because the Notifier handles timers, you should not make calls to such routines as `sleep()` or `setitimer()` (see Section 20.11, "Emulating a Sleep Call," for information on how to "sleep" in XView). As pointed out above, you should not attempt to use `signal()` to catch `SIGALRM` or `SIGVTALRM` to trap timer signals. To set timers and be notified when they expire, use `notify_set_itimer_func()`. The form of the call is:

```
Notify_func
notify_set_itimer_func(client, timer_func, which, value, ovalue)
    Notify_client   client;
    Notify_func     timer_func;
    int             which;
    struct itimerval *value, *ovalue;
```

The parameter `which` indicates which type of timer you want to use. Its value is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

The `value` parameter is a pointer to an `itimerval` which indicates the initial timeout and an interval timeout. The interval timeout is what is used after the initial timeout times out. If the initial timeout is 0, then the timer is not called, regardless of the value of the interval timer. The granularity of the timer is dependent on your hardware architecture and operating system. It is perhaps unwise to assume that you will be notified as frequently as 30 milliseconds or less. Some cases may have higher minimum limits—your mileage may vary.

The `ovalue` is also a pointer to an `itimerval` structure. If there was a previous timeout that has yet to expire, `itimerval` will have that time filled in. You may pass a `NULL` as `ovalue` if you are not interested in this value.

The `timer_func` parameter is the function to call when the timer expires. `notify_set_itimer_func()` returns the function that was previously set for this client (see `notify_set_signal_func` above). The form of the `timer_func` is:

```
Notify_value
timer_func(client, which)
    Notify_client client;
    int           which;
```

Example 20-1 demonstrates one possible use of `notify_set_itimer_func()`. This program displays an animation of several icons stored in the *icon* font. When animating, the next icon in a sequence is displayed using `XDrawString()` (since the icon is actually part of the font). The slider controls the rate at which the next icon is drawn. If the slider is set to 0, the animation stops. The slider is a panel item whose callback routine makes calls to

*Event processing still takes place during this time segment.

`notify_set_itimer_func()` to set the timer to the new value or to turn it off by setting the function to `NOTIFY_FUNC_NULL`.

Example 20-1. The animate.c program

```
/*
 * animate.c -- use glyphs from the "icon" font distributed with XView
 * to do frame-by-frame animation.
 */
#include <stdio.h>
#include <ctype.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xos.h>            /* for <sys/time.h> */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/font.h>
#include <xview/notify.h>

Frame            frame;
Display          *dpy;
GC               gc;
Window           canvas_win;
Notify_value     animate();
struct itimerval timer;

#define ArraySize(x)  (sizeof(x)/sizeof(x[0]))
char *horses[] = { "N", "O", "P", "Q", "R" };
char *boys[] = { "\007", "\005", "\007", "\010" };
char *men[] = { "\\", "]", "Y", "Z", "[" };
char *eyes[] = {
    "2", "5", "4", "3", "4", "5",
    "2", "1", "0", "/", "0", "1"
};

int max_images = ArraySize(horses);
char **images = horses;
int cnt;

main(argc, argv)
int     argc;
char    *argv[];
{
    Panel       panel;
    Canvas      canvas;
    XGCValues   gcvalues;
    Xv_Font     _font;
    XFontStruct *font;
    void        adjust_speed(), change_glyph();
    extern void exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);
```

Example 20-1. The animate.c program (continued)

```
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    xv_create(panel, PANEL_SLIDER,
        PANEL_LABEL_STRING,     "Millisecs Between Frames",
        PANEL_VALUE,            0,
        PANEL_MAX_VALUE,        120,
        PANEL_NOTIFY_PROC,      adjust_speed,
        NULL);
    xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING,     "Glyphs",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_DISPLAY_LEVEL,    PANEL_ALL,
        PANEL_CHOICE_STRINGS,   "Horse", "Man", "Boy", "Eye", NULL,
        PANEL_NOTIFY_PROC,      change_glyph,
        NULL);
    window_fit(panel);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               64,
        XV_HEIGHT,              64,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        NULL);
    canvas_win = (Window)xv_get(canvas_paint_window(canvas), XV_XID);

    window_fit(frame);

    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    _font = (Xv_Font)xv_find(frame, FONT,
        FONT_NAME,      "icon",
        NULL);
    font = (XFontStruct *)xv_get(_font, FONT_INFO);

    gcvalues.font = font->fid;
    gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
    gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
    gcvalues.graphics_exposures = False;
    gc = XCreateGC(dpy, RootWindow(dpy, DefaultScreen(dpy)),
        GCForeground | GCBackground | GCFont | GCGraphicsExposures,
        &gcvalues);

    xv_main_loop(frame);
}

void
change_glyph(item, value)
Panel_item item;
int value;
{
    cnt = 0;
    if (value == 0) {
        max_images = ArraySize(horses);
```

Example 20-1. The animate.c program (continued)

```
        images = horses;
    } else if (value == 1) {
        max_images = ArraySize(men);
        images = men;
    } else if (value == 2) {
        max_images = ArraySize(boys);
        images = boys;
    } else if (value == 3) {
        max_images = ArraySize(eyes);
        images = eyes;
    }
    XClearWindow(dpy, canvas_win);
}

/*ARGSUSED*/
Notify_value
animate(client, which)
Notify_client  client;
int            which;
{
    XDrawImageString(dpy, canvas_win, gc, 5, 40, images[cnt], 1);
    cnt = (cnt + 1) % max_images;

    return NOTIFY_DONE;
}

void
adjust_speed(item, value)
Panel_item item;
int value;
{
    if (value > 0) {
        timer.it_value.tv_usec = (value + 20) * 1000;
        timer.it_interval.tv_usec = (value + 20) * 1000;
        notify_set_itimer_func(frame, animate,
            ITIMER_REAL, &timer, NULL);
    } else
        /* turn it off */
        notify_set_itimer_func(frame, NOTIFY_FUNC_NULL,
            ITIMER_REAL, NULL, NULL);
}
```

Figure 20-1 shows one frame of an animated horse sequence produced by *animate.c*.

20.5.4 Handling SIGTERM

The `SIGTERM` signal is a software *terminate* signal. If you receive a `SIGTERM` signal, another process is telling your application to terminate. Rather than handling this signal with `notify_set_signal_func()`, you could use `notify_set_destroy_func()`. It takes the form:

```
Notify_func
notify_set_destroy_func(client, destroy_func)
```

```
        Notify_client  client;
        Notify_func    destroy_func;
```

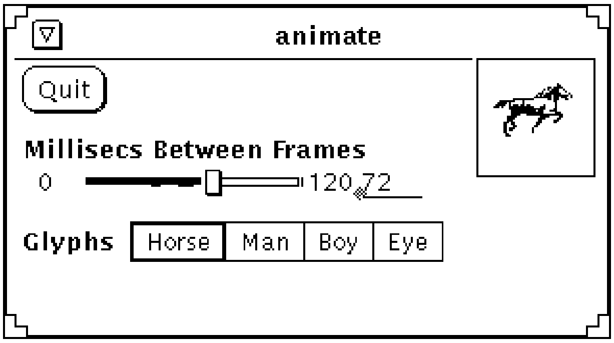


Figure 20-1. Output of animate.c

Like the other Notifier functions, this one also returns the previously set function that was handling this signal. Note that it only interprets `SIGTERM`—it does not get called if the application chooses to kill itself by calling either `exit()` or `xv_destroy()` on the base frame or if the user selects the *quit* option from the title bar (provided by the OPEN LOOK window manager).

20.5.5 Handling SIGCHLD

Let's say that you want to fork a process to run another program. UNIX requires that you perform some housekeeping on that process. The minimum housekeeping required is to notice when that process dies and to reap it. Normally, the system call `wait()` is used to do this. By default, `wait()` will block until a spawned process has terminated. When `wait()` returns, it has information about the process that died. Rather than have the application sit and wait for a spawned (child) process to die, it should continue processing events and be notified automatically when the process dies.

To handle this, you can register a `wait3` event handler* that the Notifier will call whenever a child process changes state (dies) by calling the following:

```
    Notify_func
    notify_set_wait3_func(client, wait3_func, pid)
        static Notify_client client;
        Notify_func          wait3_func;
        int                  pid;
```

In the above call, the `pid` identifies the particular child process that the client wants to wait for. The `wait3_func` is the function to call when that child has died. Another reason that an application should use `notify_set_wait3_func()` is the semantics of the `wait3`

*The name *wait3 event* originates from the `wait3` (2) system call. There are other forms of *wait* including `wait()`, `wait2()`, and `wait4()`. Since `wait4()` is not generally available, `wait3()` is the most efficient form of the *wait* functionality.

(2) system call. `wait3` (2) will return with status about *any* process that has changed state. If two clients are managing different child processes, they should hear only about their own process. The Notifier keeps straight which client is managing which process.

20.5.5.1 Reaping dead processes

Many clients using child process control simply need to perform the required reaping after a child process dies. These clients can use the predefined `notify_default_wait3()` as their `wait3` event handler. This default handler does nothing but return—the fact that it handled the event is good enough for UNIX. The Notifier automatically removes a dead process's `wait3` event handler.

The code segment in Example 20-2 demonstrates how a *wait3* handler can be set up.

Example 20-2. Demonstrating a wait3 handler

```
#include <xview/notify.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

/* canvas created in another part of the program */
extern Canvas canvas;

/* declare our own wait3 handler */
Notify_value my_wait3_handler();

fork_it(argv)
char *argv[ ];
{
    int pid;

    /* here's the fork -- two processes are going to execute code
     * from this point down.
     */
    switch (pid = fork()) {
        case -1:
            perror("fork");
            return;
        case 0: /* this is the child of the fork -- the new process */
            /* execute the specified command */
            execvp(*argv, argv); /* execvp doesn't return unless failed */
            perror("execvp");
            _exit(0); /* don't call exit() -- man exec() for info */
        default: /* this is the parent -- the original process */
            /* Register a wait3 event handler */
            (void) notify_set_wait3_func(canvas, my_wait3_handler, pid);
    }
    /* parent returns -- child is happily processing away */
}

static Notify_value
my_wait3_handler(me, pid, status, rusage)
    Notify_client  me;
    int            pid;
```

Example 20-2. Demonstrating a wait3 handler (continued)

```
    union wait    *status;
    struct rusage *rusage;
{
    if (WIFEXITED(*status)) {
        /* Child process exited with return code */
        printf("child exited with status %d\n", status->w_retcode);
        /* Tell the Notifier that you handled this event */
        return (NOTIFY_DONE);
    }
    /* Tell the Notifier that you ignored this event */
    return (NOTIFY_IGNORED);
}
```

Example 20-2 is a simple example for demonstration only. There are other things to consider for a complete and proper method for forking new processes. See Section 20.8, "Reading and Writing through File Descriptors," for a full example program that uses `notify_set_wait3_func()`.

20.6 Interaction with RPC

XView provides a function that allows XView and RPC to easily work together. RPC provides for a client and an RPC server. The client makes a remote procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

The RPC server is similar to the XView notifier; it waits for requests and dispatches them to a procedure. The RPC server typically does some initialization (using the function `svc_register`...) and then calls `svc_run()`, which is similar to `xv_main_loop()`. XView works with RPC by incorporating `svc_run()`'s functionality into the notifier.

If you use an RPC server that also needs to work with XView, you should place the XView function `notify_enable_rpc_svc()` in your `main()` program, and do not call `svc_run()`. This function takes an **int** that tells the notifier whether it should handle RPC requests.

```
void
notify_enable_rpc_svc(bool);
    int bool;
```

Using this approach, `xv_main_loop()` handles incoming RPC requests; dispatching them just as if `svc_run()` had been called.*

If `notify_enable_rpc_svc()` is enabled, performance will be affected by the additional call to `svc_getreqset()`.

*Internally XView uses `svc_getreqset(&ibits)` which acts the same way as `notify_dispatch()`. It checks `svc_fdset` and calls `svc_getreqset()` if a `svc_fdset` descriptor is readable, meaning a request is coming in.

20.7 Client Events

Client events are used by an application to communicate with clients of the Notifier. With the client event mechanism, you can have any portion of your application send an event. The Notifier dispatches the event to any of its clients that have expressed interest in that particular client event; essentially the event is sent, via the notifier, to another portion of the application, where it is handled. Events are *posted* to specific clients of the Notifier.

Client events are important and are frequently used internally by XView; they can also be used in an application. For example, if you want an application to notify an interested Notifier client that input has been received from a pipe, once the data has been read, the application can *post* the data to the Notifier as a client event (see Section 20.8, "Reading and Writing through File Descriptors").

From the Notifier's point of view, client events are defined and generated by the application. Client events are not interpreted by the Notifier in any way; the Notifier does not *detect* client events. An X event is also a client event. Internally, the window package reads the X event from the server and posts it to a client via `notify_post_event()`. When client events are *posted* to the Notifier, it dispatches these events to a receiving client's event handler. The receiving client then interprets the client events.

The process for the delivery of client events is similar to that of signals. However, because the entire process happens in the application, more control can be maintained by the Notifier. This additional level of control uses optional parameters that may be passed to client event handlers. These optional parameters include values for *safe* and *immediate* handling. With safe handling the Notifier sends event handlers notification when it is safe to do so. This may involve some delay between when an event is posted and when it is delivered. Alternatively, for immediate handling, an event handler may ask to be immediately notified when a client event is posted. A client event handler may have *both* a safe *and* an immediate event handler. Safe and immediate notification are covered in more detail later on in this section.

The remainder of this section describes how to register client event handlers and how to post client events.

NOTE

> XView internally registers event handlers for its internally defined client events when window-based objects are created. You will almost never need to register your own client event handler. It is more common that you will want to interpose in front of one of XView's client event handlers. Refer to Section 20.9, "Interposition," for details on interposition.

20.7.1 Receiving Client Events

To register a client event handler, use:

```
Notify_func
notify_set_event_func(client, event_func, when)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  when;

typedef enum notify_event_type {
    NOTIFY_SAFE         = 0,
    NOTIFY_IMMEDIATE    = 1,
} Notify_event_type;
```

The `when` parameter indicates whether the event handler will accept notifications only when it is safe (`NOTIFY_SAFE`) or for immediate notification and when it is safe (`NOTIFY_IMMEDIATE`). You may register two client event handlers to handle these cases individually, or just one client event handler to cover both.

The calling sequence of a client event handler is:

```
Notify_value
event_func(client, event, arg, when)
    Notify_client      client;
    Notify_event       event;
    Notify_arg         arg;
    Notify_event_type  when;
```

The `client` is the one that was passed to `notify_set_event_func()`. The `event` is passed through from `notify_post_event()`, which is described in the following section. The `arg` and `event` parameters are completely defined by the client. The types `Notify_arg` and `Notify_event` are of type `caddr_t`, a generic pointer type. The `when` parameter is the actual situation in which `event` is being delivered. It can have the value `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`, and can be different from `when_hint` of `notify_post_event()`.

The return value `Notify_value` may be one of `NOTIFY_DONE`, `NOTIFY_IGNORED`, or `NOTIFY_UNEXPECTED`. `NOTIFY_DONE` indicates that the event was acted on in some way. This implies that no further action is required by the client. If the safe event handler returns `NOTIFY_IGNORED`, this indicates that the event failed to provoke any action. If the immediate client event handler returns `NOTIFY_IGNORED`, then the same notification will be delivered to the safe client event handler when it is safe. A value of `NOTIFY_UNEXPECTED` indicates the event was not handled and not recognized. This return value may indicate an error condition.

20.7.2 Posting Client Events

A client event may be posted to the Notifier at any time. The poster of a client event may suggest to the Notifier when to deliver the event, but this is only a hint. The Notifier will see to it that it is delivered at an appropriate time (more on this below). The call to post a client event is:

```
typedef char * Notify_event;

Notify_error
notify_post_event(client, event, when_hint)
    Notify_client       client;
    Notify_event        event;
    Notify_event_type   when_hint;
```

The `client` handle from `notify_set_event_func()` is passed to `notify_post_event()`. `event` is defined and interpreted solely by the `client`. A return code of `NOTIFY_OK` indicates that the notification has been posted. Other values indicate an error condition. `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is unknown to the Notifier. `NOTIFY_NO_CONDITION` indicates that `client` has no client event handler registered with the Notifier.

Usually it is during the call to `notify_post_event()` that the client event handler is called. Sometimes, however, the notification is queued up for later delivery. The Notifier chooses between these two possibilities by noting which kinds of client event handlers `client` has registered, whether it is safe, and what the value of `when_hint` is. Here are the cases broken down by the kind of client event handlers `client` has registered:

Immediate only
: If `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`, the event is delivered immediately.

Safe only
: If `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`, the event is delivered when it is safe.

Both safe and immediate
: A client may have both an immediate client event handler as well as a safe client event handler. If `when_hint` is `NOTIFY_SAFE`, then the notification is delivered to the safe client event handler when it is safe. If `when_hint` is `NOTIFY_IMMEDIATE`, then the notification is delivered to the immediate client event handler right away. If the immediate client event handler returns `NOTIFY_IGNORED`, then the same notification will be delivered to the safe client event handler when it is safe.

20.7.2.1 Actual delivery time

For client events, other than knowing which event handler to call, the main function of the Notifier is to know when to make the call. The Notifier defines when it is safe to make a client notification. If it is not safe, then the event is queued up for later delivery. Here are the conventions:

- A client that has registered an immediate client event handler is sent a notification as soon as it is received. The client has complete responsibility for handling the event safely. It is rarely safe to do much of anything when an event is received asynchronously. Usually, just setting a flag that indicates that the event has been received is about the safest thing that can be done.
- A client that has registered a safe client event handler will have a notification queued up for later delivery when the notification was posted during an asynchronous signal notification. Immediate delivery is not safe because your process, just before receiving the signal, may have been executing code at any arbitrary place.
- A client that has registered a safe client event handler will have a notification queued up for later delivery if the client's safe client event handler hasn't returned from processing a previous event. This convention is mainly to prevent the cycle: Notifier notifies *A*, who notifies *B*, who notifies *A*. *A* could have had its data structures torn up when it notified *B* and was not in a state to be re-entered.

These conventions imply that a safe client event handler is called immediately from other signal handlers. The following sequence shows how the notification might progress.

- A client's input pending event handler is called by the Notifier.
- Two characters are read by the client's input pending event handler.
- The first character is given to the Notifier to deliver to the client's save event handler (the delivery is accomplished with `notify_post_event()`).
- Returning back to the input pending event handler, the second character is sent. This character is also delivered immediately.

20.7.3 Posting with an Argument

XView posts a fixed-field structure with each event. Sometimes additional data must be passed with an event. For instance, when the scrollbar posts an event to its owner to do a scroll, the scrollbar's handle is passed as an argument along with the event. The function `notify_post_event_and_arg()` provides this argument-passing mechanism (see below).

When posting a client event, there is the possibility of delivery being delayed. In the case of XView, the event being posted is a pointer to a structure. The Notifier avoids an invalid (dangling) pointer reference by copying the event if delivery is delayed. It calls routines the client supplies to copy the event information and later to free up the storage the copy uses.

`notify_post_event_and_arg()` provides this storage management mechanism.

```
Notify_error
notify_post_event_and_arg(client, event, when_hint, arg,
                          copy_func, release_func)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
    Notify_arg         arg;
    Notify_copy        copy_func;
    Notify_release     release_func;

typedef caddr_t Notify_arg;

typedef Notify_arg (*Notify_copy)();
#define NOTIFY_COPY_NULL   ((Notify_copy)0)

typedef void (*Notify_release)();
#define NOTIFY_RELEASE_NULL  ((Notify_release)0)
```

`copy_func()` is called to copy `arg` (and, optionally, `event)` when `event` and `arg` need to be queued for later delivery. `release_func()` is called to release the storage allocated during the copy call when `event` and `arg` were no longer needed by the Notifier.

Any of `arg`, `copy_func()`, or `release_func()` may be `NULL`. If `copy_func` is not `NOTIFY_COPY_NULL` and `arg` is `NULL`, then `copy_func()` is called anyway. This allows `event` the opportunity to be copied because `copy_func()` takes a pointer to `event`. The event pointed to may be replaced as a side effect of the copy call. The same applies to a `NOTIFY_RELEASE_NULL` release function with a `NULL arg` argument.

The `copy()` and `release()` routines are client-dependent, so you must write them yourself. Their calling sequences are the following:

```
Notify_arg
copy_func(client, arg, event_ptr)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event  *event_ptr;

void
release_func(client, arg, event)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event   event;
```

20.7.4 Posting Destroy Events

When a destroy notification is set, the Notifier also sets up a synchronous signal condition for `SIGTERM` that will generate a `DESTROY_PROCESS_DEATH` destroy notification. Otherwise, a destroy function will not be called automatically by the Notifier. One or two (depending on whether the client can veto your notification) explicit calls to `notify_post_destroy()` need to be made.

```
Notify_error
notify_post_destroy(client, status, when)
    Notify_client      client;
    Destroy_status     status;
    Notify_event_type  when;
```

NOTIFY_INVAL is returned if status or when is not defined. After notifying a client to destroy itself, all references to client are purged from the Notifier.

20.7.5 Delivery Time of Destroy Events

Unlike a client event notification, the Notifier does not try to detect when it is safe to post a destroy notification. Thus, a destroy notification can come at any time. It is up to the good judgement of a caller of notify_post_destroy() or notify_die() (described in Section 20.10, “Notifier Control”) to make the call when a client is not likely to be in the middle of accessing its data structures.

If status is DESTROY_CHECKING and the argument when is NOTIFY_IMMEDIATE, then notify_post_destroy() may return NOTIFY_DESTROY_VETOED, if the client does not want to go away. See Section 20.9.5, “Modifying an Objects’ Destruction,” for details on these values.

Often you want to tell a client to go away at a safe time. This implies that delivery of the destroy event will be delayed, in which case the return value of notify_post_destroy() cannot be NOTIFY_DESTROY_VETOED because the client has not been asked yet. To get around this problem, the Notifier will flush the destroy event of a checking/destroy pair of events if the checking phase is vetoed. Thus, a common idiom is:

```
(void) notify_post_destroy(client, DESTROY_CHECKING, NOTIFY_SAFE);
(void) notify_post_destroy(client, DESTROY_CLEANUP, NOTIFY_SAFE);
```

20.8 Reading and Writing Through File Descriptors

The Notifier is set up for testing whether or not there is input pending on file descriptors and whether file descriptors are available to accept data for writing. The file descriptors can represent files, pipes, and sockets. System calls such as read() or write() can be used on any of these file descriptors.

In an event-driven system, the application cannot wait for data to be read. Instead it is better for the system to notify the application when data can be read. System calls such as read() will *block* if there is no input to be read. That is, read() waits until there is something to read before it returns. If your application is blocking on a read, then it cannot process events that the user may be generating, such as selecting a panel button. Rather than blocking on a call to read(), the Notifier can inform you when there is input ready on that file descriptor so that when you finally call read(), it returns immediately.

To handle this, the Notifier provides functions such as `notify_set_input_func()` and `notify_set_output_func()` to test whether a file descriptor has data to be read or is ready for writing. These functions inform you of the status of file descriptors, whether they are files, pipes, or sockets.*

```
Notify_func
notify_set_input_func(client, input_func, fd)
    Notify_client   client;
    Notify_func     input_func;
    int             fd;
Notify_func
notify_set_output_func(client, output_func, fd)
    Notify_client   client;
    Notify_func     output_func;
    int             fd;
```

`input_func()` is called whenever the file descriptor (`fd`) has data to be read.† `output_func()` is called whenever the file descriptor (`fd`) is ready to receive data.

Generally, file descriptors open for writing are always ready to receive data, so this function may not be as widely used as the input function case. However, it is important if you are writing to a pipe where another process is probably on the other side of the pipe. Pipes have buffers which, when full, will not accept any more data on them until the process on the other side of the pipe reads the data written so far (thus emptying the buffer). If the process on the other side of the pipe is slow in reading the data you have written to it, then you may need to use `notify_set_output_func()` so that you can be notified when the pipe is ready to have more data written to it.

`input_func()` and `output_func()` take the following form:

```
Notify_value
func(client, fd)
    Notify_client  client;
    int            fd;
```

20.8.1 Reading Files

Our first example demonstrates how `notify_set_input_func()` can be used to read data from a file. The program is simple; it does not bother using any XView objects such as frames or canvases. Because of this, it makes use of `notify_start()`, which is covered in Section 20.10, "Notifier Control."

*`notify_set_exception_func()` is also available for determining if out-of-band data is available on a socket. Its format is described in the *XView Reference Manual*.
†Exceptions to this are discussed in Section 20.8.1, "Reading Files."

Example 20-3. The notify_input.c program

```
/*
 * notify_input.c -- use notify_set_input_func to monitor the state of
 * a file.  The Notifier is running and checking the file descriptors
 * of the opened files associated with the command line args.  The
 * routine installed by notify_set_input_func() is called whenever
 * there is data to be read.  When there is no more data to be read
 * for that file, the input function is unregistered.  When all files
 * have been read, notify_start() returns and the program exits.
 */
#include <stdio.h>
#include <sys/ioctl.h>
#include <xview/notify.h>

main(argc, argv)
char *argv[ ];
{
    Notify_value   read_it();
    Notify_client  client = (Notify_client)10101; /* arbitrary */
    FILE           *fp;

    while (*++argv)
        if (!(fp = fopen(*argv, "r")))
            perror(*argv);
        else {
            (void) notify_set_input_func(client, read_it, fileno(fp));
            client++; /* next client is new/unique */
        }

    /* loops continuously */
    notify_start();
}

/*
 * read_it() is called whenever there is input to be read.  Actually,
 * it is called continuously, so check to see if there is input to be
 * read first.
 */
Notify_value
read_it(client, fd)
Notify_client   client;
int fd;
{
    char buf[BUFSIZ];
    int bytes, i;

    if (ioctl(fd, FIONREAD, &bytes) == -1 || bytes == 0)
        (void) notify_set_input_func(client, NOTIFY_FUNC_NULL, fd);
    else
        do
            if ((i = read(fd, buf, sizeof buf)) > 0)
                (void) write(1, buf, i);
        while (i > 0 && (bytes -= i) > 0);
    return NOTIFY_DONE;
}
```

The comments in this sample program describe what is going on up front. However, behind the scenes, there are interesting things happening.

The first thing to notice is that the clients used in the Notifier start at `10101` and, for each file on the command line, the client is incremented by one. Remember, the client is an arbitrary identifier and can be any unique value. Since there are no other clients of the Notifier, we know that `10101` is going to be unique.

The function `read_it()` is installed to read data from the files given on the command line. The first thing that `read_it()` does is check if there is data to be read. The `ioctl()` call returns the number of bytes to read in the `bytes` variable. If there is data to be read, then `read()` is used to read buffers (of size `BUFSIZ`) until it has read all the bytes pending. This continues until the program has read the entire file. Once this happens, the `ioctl()` call will return that there are no bytes to read. Normally, it would return -1 indicating that the end of file has been reached. However, the Notifier has modified the state of the file descriptor for internal purposes (to set nonblocking mode). When we have reached the end of the file, we unregister the client by calling `notify_set_input_func()` with a `notify_func` of `NOTIFY_FUNC_NULL`.

notify_input.c is not terribly interesting because you do not need to be notified of data to be read on a *file*; you can just open the file, read it till `EOF`, and then close it. But if you know that the file is going to have data continuously added to it, you might want to be informed when there is new data to be read and then print it out.

This can be accomplished by not unregistering the input function with the Notifier when there is no data to be read. Unfortunately, your function is going to be called continuously whether there is new data to be read or not. This is true *only for file descriptors that represent files* and is not the case for pipes.

20.8.2 Reading and Writing on Pipes

A more interesting and likely problem occurs when an application has to execute another program, send data to it, and read output from it all at the same time. The best way to handle such a situation is as follows:

- Set up a pipe for each stream (such as `stdin` and `stdout`). This is done using the `pipe()` system call.
- `fork()` a new process. The child will execute the program (using `execvp()`) and have its input and output redirected to the appropriate ends of the pipes (using `dup2()`).
- The parent registers *input* and *output* functions to read and write from the "other" ends of the pipes (using `notify_set_input_func()`). Also, the parent calls `notify_set_wait3_func()`, as discussed earlier.

The example program that demonstrates this capability is a little longer, but it is still simple. It follows the steps outlined above and is the minimum needed to spawn a new process. Again, to keep the code short and simple, no XView objects are created. The existence of XView objects would not affect the program in any way.

The way to run this program is to specify a command on the command line of the program. For example:

```
% ntfy_pipe cat
```

Remember, there are *two* processes involved here: the parent process (ntfy_pipe) and the child process (cat). Both programs read from their respective stdin and write to their own stdout. However, because the child process (cat) was spawned off from the parent, we need to handle the IO (input/output) of the new process. After the call to fork(), its stdin and stdout must be redirected to the pipe that was set up before the fork. The parent, however, retains its IO—its stdin and stdout remain directed towards the window in which you typed the command.

Example 20-4. The ntfy_pipe.c program

```
/*
 * ntfy_pipe.c -- fork and set up a pipe to read the IO from the
 * forked process.  The program to run is specified on the command
 * line.  The functions notify_set_input_func() and
 * notify_set_output_func() are used to install functions which read
 * and write to the process' stdin and stdout.
 * The program does not use any XView code -- just the Notifier.
 */
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <sys/ioctl.h>
#include <xview/notify.h>

Notify_client client1 = (Notify_client)10;
Notify_client client2 = (Notify_client)11;

int pipe_io[2][2]; /* see diagram */
/*
 *                 [0]                           [1]
 *    child reads:  |========= pipe_io[0] ========| <- parent writes
 *   pipe_io[0][0]                                     pipe_io[0][1]
 *
 *    parent reads: |========= pipe_io[1] ========| <- child writes
 *   pipe_io[1][0]                                     pipe_io[1][1]
 *
 * The parent process reads the output of the child process by reading
 * pipe_io[1][0] because the child is writing to pipe_io[1][1].
 * The child process gets its input from pipe_io[0][0] because the
 * parent writes to pipe_io[0][1].  Thus, one process is reading from
 * one end of the pipe while the other is writing at the other end.
 */
main(argc, argv)
char *argv[];
{
    Notify_value        read_it(), write_it(), sigchldcatcher();
    int                 i, pid;
    FILE                *fp;
```

Example 20-4. The ntfy_pipe.c program (continued)

```
    if (!*++argv)
        puts("specify a program [w/args]"), exit(1);

    pipe(pipe_io[0]); /* set up input pipe */
    pipe(pipe_io[1]); /* set up output pipe */
    switch (pid = fork()) {
        case -1:
            close(pipe_io[0][0]);
            close(pipe_io[0][1]);
            close(pipe_io[1][0]);
            close(pipe_io[1][1]);
            perror("fork failed");
            exit(1);
        case  0: /* child */
            /* redirect child's stdin (0), stdout (1) and stderr(2) */
            dup2(pipe_io[0][0], 0);
            dup2(pipe_io[1][1], 1);
            dup2(pipe_io[1][1], 2);
            for (i = getdtablesize(); i > 2; i--)
                (void) close(i);
            for (i = 0; i < NSIG; i++)
                (void) signal(i, SIG_DFL);
            execvp(*argv, argv);
            if (errno == ENOENT)
                printf("%s: command not found.0, *argv);
            else
                perror(*argv);
            perror("execvp");
            _exit(-1);
        default: /* parent */
            close(pipe_io[0][0]); /* close unused portions of pipes */
            close(pipe_io[1][1]);
        }

    /* when the process outputs data, read it */
    notify_set_input_func(client1, read_it, pipe_io[1][0]);
    notify_set_wait3_func(client1, sigchldcatcher, pid);

    /* wait for user input -- then write data to pipe */
    notify_set_input_func(client2, write_it, 0);
    notify_set_wait3_func(client2, sigchldcatcher, pid);

    notify_start();
}

/*
 * callback routine for when there is data on the parent's stdin to
 * read.  Read it and then write the data to the child process via
 * the pipe.
 */
Notify_value
write_it(client, fd)
Notify_client   client;
int fd;
{
```

Example 20-4. The ntfy_pipe.c program (continued)

```
    char buf[BUFSIZ];
    int bytes, i;

    /* only write to pipe (child's stdin) if user typed anything */
    if (ioctl(fd, FIONREAD, &bytes) == -1 || bytes == 0) {
        notify_set_input_func(client, NOTIFY_FUNC_NULL, pipe_io[0][1]);
        close(pipe_io[0][1]);
    } else
        while (bytes > 0) {
            if ((i = read(fd, buf, sizeof buf)) > 0) {
                printf("[Sending %d bytes to pipe (fd=%d)]0,
                    i, pipe_io[0][1]);
                write(pipe_io[0][1], buf, i);
            } else if (i == -1)
                break;
            bytes -= i;
        }
    return NOTIFY_DONE;
}

/*
 * callback routine for when there is data on the child's stdout to
 * read.  Read, then write the data to stdout (owned by the parent).
 */
Notify_value
read_it(client, fd)
Notify_client   client;
register int fd;
{
    char buf[BUFSIZ];
    int bytes, i;

    if (ioctl(fd, FIONREAD, &bytes) == 0)
        while (bytes > 0) {
            if ((i = read(fd, buf, sizeof buf)) > 0) {
                printf("[Reading %d bytes from pipe (fd=%d)]0,
                    i, fd);
                (void) write(1, buf, i);
                bytes -= i;
            }
        }
    return NOTIFY_DONE;
}

/*
 * handle the death of the child.  If the process dies, the child
 * dies and generates a SIGCHLD signal.  Capture it and disable the
 * functions that talk to the pipes.
 */
Notify_value
sigchldcatcher(client, pid, status, rusage)
Notify_client client; /* the client noted in main() */
int pid; /* the pid that died */
union wait *status; /* the status of the process (unused here) */
struct rusage *rusage; /* resources used by this process (unused) */
{
```

Example 20-4. The ntfy_pipe.c program (continued)

```
    if (WIFEXITED(*status)) {
        printf("Process termined with status %d0, status->w_retcode);
        /* unregister input func with appropriate file descriptor */
        notify_set_input_func(client, NOTIFY_FUNC_NULL,
            (client == client1)? pipe_io[1][0] : 0);
        return NOTIFY_DONE;
    }
    puts("SIGCHLD not handled");
    return NOTIFY_IGNORED;
}
```

Assuming that the program was run with `cat` (1) as described earlier, input to the child process comes from the function `write_it()`. That is, `write_it()` is the function that *writes* to the other side of the pipe that the child is reading from. This function was set up as the *input_func* for the *parent's* `stdin`. The function gets its input from whatever you type in the window in which you ran the program. It could have gotten its input from a text panel item or a selection using the selection service. It reads whatever data is sent through the pipe and ends up as the `stdin` for the child process.

Similarly, when the child process writes anything to its `stdout`, the data is redirected through the pipe. The parent reads from the other end of the pipe in `read_it()`. Whatever the parent reads is written to the parent's `stdout`, which is the user's tty window. It does this by calling `write(1, buf, i)`. This line of the program can easily be replaced by:

```
    textsw_insert(textsw, buf, i);
```

This shows how you can redirect the output (including `stderr` output) from a child process to a text subwindow.

The last thing to note about the program is its treatment of extraneous file descriptors and of signals handled by the child process after forking. When using `fork()`, the child process inherits all the file descriptors and signal handling set up by the parent.* This could seriously affect your program if the child process gets certain signals. In this case the best thing for the child to do is loop through all the signals that are dealt with by the parent and reset them to `SIG_DFL` using `signal` (3). Then it can close all file descriptors except for the child's `stdin` (0), `stdout` (1), and `stderr` (2) descriptors.

We use the `signal()` system call here because it is assumed that the child is not going to execute any of the XView code in the parent's program. Therefore, the child does not need to use the `notify_set_signal_func()` routine to unregister the signals. Releasing these signals in this way completely disassociates the program from the parent.

This very general program is used as an example, but it is useful in all applications that need to run external processes. The only modifications it needs are alternate sources for the input and output of the parent. `main()` could be replaced by a general function that simply gets an `argv` parameter containing the program to execute, including arguments.

*Unless the file descriptors were set to *close on exec* using `ioctl(fd, FIOCLEX, 0)`.

The program could be modified to be used as a replacement for the `system` (3) call. Even if you do not expect to read the child's `stdout` or send data to its `stdin`, the child process still needs to be handled differently from `system` (3) because that function calls `wait()` and attempts to do its own signal handling.

20.8.3 Exception Occurred Events

Exception occurred notifications are similar to input pending notifications. The only known devices that generate exceptions are stream-based socket connections when an out-of-band byte is available. Thus a `SIGURG` signal catcher is set up by the Notifier, much like a `SIGIO` for asynchronous input.

```
Notify_func
notify_set_exception_func(nclient, func, fd)
    Notify_client   nclient;
    Notify_func     func;
    int             fd;
```

20.8.4 Getting an Event Handler

This section contains a list of the `notify_get_*_func()` routines. These functions allow you to retrieve the value of a client's event handler. Once you have the value of a client's event handler, you could modify or replace the event handler. It is recommended that you use the Notifier's interposition mechanism instead of using these functions. The arguments for these functions parallel the associated `notify_set_*_func()` functions described previously, except for the absence of the event handler function pointer. Refer to the *XView Reference Manual* for a description of the arguments.

For all of the `notify_get_*_func()` functions in the following list, a return value of `NOTIFY_FUNC_NULL` indicates an error. If the client is unknown, then `notify_errno` is set to `NOTIFY_UKNOWN_CLIENT`. If no event handler is registered for the specified event, then `notify_errno` is set to `NOTIFY_NO_CONDITION`. Other values on `notify_errno` are possible, depending on the event; for example, `NOTIFY_BAD_FD` if an invalid file descriptor is specified.

Here is a list of the event handler retrieval routines:

- `notify_get_input_func(client, fd)`
- `notify_get_event_func(client, when)`
- `notify_get_output_func(client, fd)`
- `notify_get_exception_func(client, fd)`
- `notify_get_signal_func(client, which)`
- `notify_get_itimer_func(client, signal, mode)`

- `notify_get_wait3_func(client, pid)`
- `notify_get_destroy_func(client)`

20.9 Interposition

The Notifier provides a mechanism called *interposition,* with which you can intercept control of the internal communications within XView. Interposition is a powerful way to both monitor and modify window behavior in ways that extend the functionality of a window object.

Interposition allows a client to intercept an event before it reaches the *base event handler*. The base event handler is the one set originally by a client. The client can call the base event handler before and/or after its own handling of the event or not at all. This allows the application to override the handling provided by XView's base event handler, or to add special event handling. The notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client.

There may be more than one interposer for a Notifier client. As each interposer is added, it is inserted ahead of the last interposer installed. Interposes may also be removed from the interposer list. When an event arrives, the Notifier calls the function at the top of the interposer list for that client.

Keeping track of which function to call when running down the list of interposers is done by maintaining an *interposition stack*. The number of interpositions allowed on the stack is limited to a maximum size (about six levels deep). Since the base event handler is placed on the interposition stack, there are five usable levels, although few applications should require more than two levels of interposition. Figure 20-2 illustrates the flow of control with interposition. Note that the interposer could have stopped the flow of control to the next event handler.

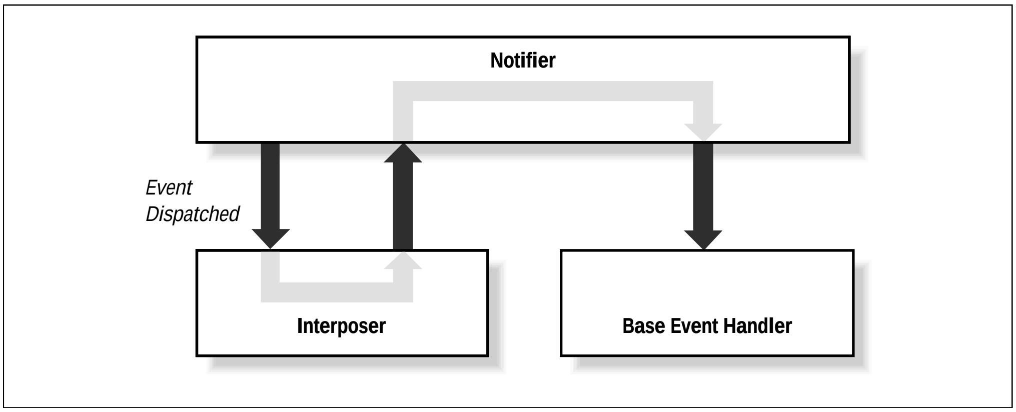


Figure 20-2. Flow of control in interposition

20.9.1 Uses of Interposition

Typically, it is application-level code that uses interposition. But, in general, any client's creator may want to use interposition. There are many reasons why an application might want to interpose a function in the call path to a client's event handler.

- An application may want to use the fact that a client has received a particular event as a trigger for some application-specific processing.
- An application may want to filter the events to a client, thus modifying the client's behavior.
- An application may want to extend the functionality of a client by handling events that the client is not programmed to handle.

XView window objects utilize the Notifier for much of their communication and cooperation. Thus, if an application wanted to monitor the user actions directed to a particular window, the application would use interposition to get into the flow of control.

20.9.2 Interface to Interposition

The Notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client. Here is a typical example of interposition:

- An application creates a client. The client has set up its own client event handler using `notify_set_event_func()`. XView does this internally, using a default event handler, when you create a window based object.
- The application tells the Notifier that it wants to interpose a function in front of the client's event handler by calling `notify_interpose_event_func()`, which uses the same calling sequence as `notify_set_event_func()`.
- When the application's interposed function is called, it tells the Notifier to call the next function, i.e., the client's function, via a call to `notify_next_event_func()`, which uses the same calling sequence as that passed to the interposer function.

Note that you can only interpose if a base event handler has been set with `notify_set_*_func` (one of the `notify_set` functions). If no function is set, `notify_interpose_*_func()` will return an error.

20.9.3 Registering an Interposer

This section lists the routines that allow you to interpose your own function in front of an event handler. The arguments to each `notify_interpose_*_func()` function like the associated `notify_set_*_func()` function described in the previous sections on the Notifier. Refer to the associated `notify_set_*_func()` description, or to the *XView Reference Manual* for details on the various arguments.

- notify_interpose_destroy_func()
- notify_interpose_exception_func()
- notify_interpose_event_func()
- notify_interpose_input_func()
- notify_interpose_itimer_func()
- notify_interpose_output_func()
- notify_interpose_signal_func()
- notify_interpose_wait3_func()

The return values from these functions may be NOTIFY_OK, NOTIFY_UNKNOWN_CLIENT, NOTIFY_NO_CONDITION, NOTIFY_UNEXPECTED, or NOTIFY_FUNC_LIMIT. For NOTIFY_OK the interposition was successful. For NOTIFY_UNKNOWN_CLIENT the client is not known to the notifier. For NOTIFY_NO_CONDITION there is no event handler of the type specified. For NOTIFY_FUNC_LIMIT the level of interposition was exceeded. NOTIFY_FUNC_LIMIT means you are trying to use more than the maximum allowed levels of interposition.

If the return value is something other than NOTIFY_OK, then notify_errno contains the error code.

20.9.4 Invoking the Next Function

This section lists the routines that you call from your interposed function in order to invoke the next function in the interposition sequence (if you want to override the normal sequence, do not use these routines). The arguments to each notify_next_*_func() function are the same as the arguments passed to the interposer function. Refer to the *XView Reference Manual*, for details on the arguments.

- notify_next_destroy_func()
- notify_next_exception_func()
- notify_next_event_func()
- notify_next_input_func()
- notify_next_itimer_func()
- notify_next_output_func()
- notify_next_signal_func()
- notify_next_wait3_func()

The return value for these functions may be one of the following: NOTIFY_DONE, NOTIFY_IGNORED, or NOTIFY_UNEXPECTED. NOTIFY_DONE indicates that the event was acted on in some way. This implies that no further action is required by the client. If the safe event handler returns NOTIFY_IGNORED, this indicates that the event failed to provoke any action. If the immediate client event handler returns NOTIFY_IGNORED, then the same notification will be delivered to the safe client event handler when it is safe. A value of NOTIFY_UNEXPECTED indicates the event was not handled and not recognized. This return value may indicate an error condition.

20.9.5 Removing an Interposed Function

This section presents a list of routines that allow you to remove the interposer function that you installed using a notify_interpose_*_func() call. The arguments to each notify_remove_*_func() function are the same as the arguments passed to the associated notify_set_*_func() function described in previous sections. Refer to the previous section, or to the *XView Reference Manual*, for details on these arguments. Note the one exception to this rule is that the arguments to notify_remove_itimer_func() are a subset of the arguments to notify_set_itimer_func().

- notify_remove_destroy_func()
- notify_remove_exception_func()
- notify_remove_event_func()
- notify_remove_input_func()
- notify_remove_itimer_func()
- notify_remove_output_func()
- notify_remove_signal_func()
- notify_remove_wait3_func()

If the function returns successfully, the return value will be NOTIFY_OK. Otherwise, the error codes are the same as those associated with the notify_interpose_*_func() calls.

20.9.6 An Interposition Example

You can notice when a frame opens or closes by interposing in front of the frame's *client event handler*. The client event handler that you want to interpose in front of is the default client event handler supplied by XView. To register an interposer, the following routine is used:

```
Notify_error
notify_interpose_event_func(client, event_func, type)
    Notify_client     client;
    Notify_func       event_func;
    Notify_event_type type;
```

The client must be the handle of the Notifier client in front of which you are interposing. In XView, this is the handle returned from xv_create(), for a Tty subwindow or a

`Textsw`, the handle is the `OPENWIN_NTH_VIEW` of the window. For a `Canvas,` the handle returned from is `CANVAS_NTH_PAINT_WINDOW`.

Let’s say that the application is displaying some animation and wants to do the necessary computation only when the frame is open. It can use interposition to notice when the frame opens or closes.

In Example 20-5, note the call to `notify_next_event_func()`. This function transfers control to the frame’s client event handler through the Notifier. The routine `notify_next_event_func()` takes the same arguments as the interposer.

Example 20-5. Transferring control through the Notifier

```
#include <xview/xview.h>

main()
{
    Frame frame;
    Notify_value my_frame_interposer();

    /* Create the frame */
    frame = xv_create(NULL, FRAME, NULL);

    /* Interpose in front of the frame’s event handler */
    (void) notify_interpose_event_func(frame,
           my_frame_interposer, NOTIFY_SAFE);
    ...

    /* Show frame and start dispatching events */
    xv_main_loop(frame);
}

Notify_value
my_frame_interposer(frame, event, arg, type)
Frame              frame;
Event             *event;
Notify_arg         arg;
Notify_event_type  type;
{
    int closed_initial, closed_current;
    Notify_value value;

    /* Determine initial state of frame */
    closed_initial = (int) xv_get(frame, FRAME_CLOSED);

    /* Let frame operate on the event */
    value = notify_next_event_func(frame, (Notify_event)event, arg, type);

    /* Determine current state of frame */
    closed_current = (int) xv_get(frame, FRAME_CLOSED);

    /* Change animation if states differ */
    if (closed_initial != closed_current) {
        if (closed_current) {
            /* Turn off animation because closed */
            (void) notify_set_itimer_func(me, my_animation,
                ITIMER_REAL, ITIMER_NULL, ITIMER_NULL);
```

Example 20-5. Transferring control through the Notifier (continued)

```
        } else {
            /* Turn on animation because opened */
            (void) notify_set_itimer_func(me, my_animation,
                ITIMER_REAL, &NOTIFY_POLLING_ITIMER, ITIMER_NULL);
        }
    }
    return (value);
}
```

In Example 20-5, the base event handler is intended to handle the event (so that the frame gets closed/opened). If the interposed function replaces the base event handler and you do not want the base event handler to be called at all, your interposed procedure should not call `notify_next_event_func()`.

20.9.7 Interposing on Resize Events

Another use of interposition is to give your application more control over the layout of its subwindows. To do this, you set up an interpose event handler which checks if the event type is `WIN_RESIZE`. If so, rather than calling `notify_next_event_func()` to dispatch the event to the normal handler for resizing, you call your own resize routine:

```
Notify_value
my_frame_interposer(frame, event, arg, type)
Frame              frame;
Event             *event;
Notify_arg         arg;
Notify_event_type  type;
{
    Notify_value value;

    if (event_action(event) == WIN_RESIZE)
        value = resize(frame);
    else
        value = notify_next_event_func(frame, (Notify_event)event, arg, type);

    return(value);
}
```

20.9.8 Modifying an Object's Destruction

Suppose an application must detect when the user selects the “Quit” menu item in order to perform some application-specific confirmation. To accomplish this, the application should interpose a new function in front of the object's *client-destroy event* handler using the following routine:

```
Notify_error
notify_interpose_destroy_func(client, destroy_func)
    Notify_client client;
    Notify_func   destroy_func;
```

For an XView object, destruction may originate from several sources:

- The application may be terminated by a software interrupt from the calling process.
- The connection to the server has been lost and the Notifier is informing the application that it is dying.
- `xv_destroy(`*`object)`* was called from within the application.
- The user may select the "Quit" menu item from the window manager's pulldown menu.

Each of these scenarios results in a different destruction method. When the object is going to be destroyed, the process happens in two phases. First, the object's destroy-interposer is called, informing it of the impending destruction. At this point, the interposer can *veto* the destruction or it can allow it to take place—at which time the Notifier proceeds to phase two, the actual object destruction.

Destroy event handlers use a status parameter to determine which phase of destruction the Notifier is in—whether the Notifier is requesting if it is feasible for the client to be terminated at present (phase one, DESTROY_CHECKING) or if it is making a request to terminate (phase two, DESTROY_CLEANUP or DESTROY_PROCESS_DEATH).

The destroy-interpose function takes the following form:

```
Notify_value
destroy_func(client, status)
    Notify_client client;
    Destroy_status status;

typedef enum destroy_status {
    DESTROY_PROCESS_DEATH,
    DESTROY_CHECKING,
    DESTROY_CLEANUP,
    DESTROY_SAVE_YOURSELF,
} Destroy_status;
```

If the `status` argument is DESTROY_CHECKING and the client cannot terminate at present, the destroy event handler should call `notify_veto_destroy()`, indicating that termination would not be advisable at this time, and return normally. If the client can terminate at present, then the destroy handler should do nothing; a subsequent call will tell the client to actually destroy itself. This veto option is used, for example, to give a text subwindow the chance to ask the user to confirm the saving of any editing changes when quitting a tool.

If `status` is DESTROY_PROCESS_DEATH, then the client can count on the entire process dying and should do whatever it needs to do to clean up its outside entanglements, such as updating a file used by a text subwindow. Since the process is about to die, it need not free allocated memory which is implicitly freed by the process's termination.

Since the entire process is dying, you cannot display a notice or do any sort of prompting with the user to try to veto this request.

However, if `status` is `DESTROY_CLEANUP` then the client is asked to destroy itself and to be very tidy about cleaning up all the process internal resources that it is using, as well as its outside entanglements. This may be called on frames which are not the sole frames used by the application. If a frame is dismissed but the application is still running (e.g., dismissing a pinned up menu), then the frame should clean up—including freeing allocated memory.

If the `status` is set to `DESTROY_SAVE_YOURSELF` when the window manager has sent a `WM_SAVE_YOURSELF` message to all the clients on the desktop, this means the user may have selected the "Save Workspace" option from an OPEN LOOK window manager's menu. Basically, this means that the application should save its current state in such a way that it could be resumed in the same state at a later time. For example, a text editing program would update the file being edited, a graphics program would output its image display to a file, or whatever.

When the user selects "Save Workspace" from an OPEN LOOK window manager, XView sets its status to `DESTROY_SAVE_YOURSELF` and then checks to see if the attribute `FRAME_WM_COMMAND_ARGC_ARGV` is set. XView informs the window manager of the option and the window manager writes the application's default command-line options to `~/.openwin-init`. If `FRAME_WM_COMMAND_ARGC_ARGV` is set, the window manager appends any user-specified command-line options to the default values written to `~/.openwin-init`.

Normally `FRAME_WM_COMMAND_ARGC_ARGV` is set when an application is initialized with `xv_init()`. In the special case when there are two base frames for an application, `FRAME_WM_COMMAND_ARGC_ARGV` should be set to -1 for one of the base frames (and for additional subsequent base frames for the application). This prevents the application from appending two sets of command-line options to `~/.openwin-init`.

20.9.8.1 Interposing a client destroy handler

We present an example of interposing in front of the frame's client-destroy event handler. The following program displays a frame, a panel, and a panel button labeled "Quit." When the user chooses the panel button or the frame's "Quit" menu selection, the interposer is called and indicates the fact that the frame is about to go away.

Example 20-6. The interpose.c program

```
/*
 * interpose.c -- shows how to use an interpose destroy function
 */
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/notice.h>

Frame frame;

Notify_value
destroy_func(client, status)
```

Example 20-6. The interpose.c program (continued)

```
Notify_client client;
Destroy_status status;
{
    if (status == DESTROY_CHECKING) {
        int answer = notice_prompt(client, NULL,
            NOTICE_MESSAGE_STRINGS, "Really Quit?", NULL,
            NOTICE_BUTTON_YES,  "No",
            NOTICE_BUTTON_NO,   "Yes",
            NULL);
        if (answer == NOTICE_YES)
            notify_veto_destroy(client);
    } else if (status == DESTROY_CLEANUP) {
        puts("cleaning up");
        /* allow frame to be destroyed */
        return notify_next_destroy_func(client, status);
    } else if (status == DESTROY_SAVE_YOURSELF)
        puts("save yourself?");
    else
        puts("process death");
    return NOTIFY_DONE;
}

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    int   quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
        XV_HEIGHT,      100,
        NULL);
    notify_interpose_destroy_func(frame, destroy_func);

    panel = (Panel)xv_create (frame, PANEL, NULL);
    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);
    xv_main_loop(frame);
}

int
quit()
{
    xv_destroy_safe(frame);
    return XV_OK;
}
```

The first time the interposer is called, the status is DESTROY_CHECKING. Here, we display a notice prompting the user to confirm the quit. We want the *default* action to be "No," so we make the NOTICE_YES button have the label "No" and the NOTICE_NO button have the label "Yes." This is because the notice's *default* button is the NOTICE_YES button.

If the user selects the default "No" choice, then notify_veto_client() is called, vetoing the destruction request. Otherwise, the routine returns and the destruction sequence continues as usual. The routine is then called again, notifying that it is being destroyed this time. It calls notify_next_destroy_func() to allow the process to continue. This is where any process cleaning up should take place.

20.9.8.2 Enabling panel item interposition

The attribute PANEL_POST_EVENTS saves the application from the burden of installing a base event handler on a panel item. This replaces the default event handler, which for performance reasons, does not use the notifier. Setting attribute PANEL_POST_EVENTS to TRUE replaces the default panel item event handler with another one that calls notify_post_event(). Note that setting this attribute alone makes no functional difference between the panel item and any other panel items (besides the difference in performance due to events being dealt through the notifier rather than being sent directly).

If PANEL_POST_EVENTS is TRUE, a client may then proceed to call notify_interpose_event_func() on a panel item. All interposers on panel items set up this way should be of type NOTIFY_IMMEDIATE. The interposer reaches the next or default event handler via the notify_next_event_func() call.

This attribute has no bearing whatsoever on the PANEL_EVENT_PROC mechanism. The client may still replace the default event handler for a particular event by using PANEL_EVENT_PROC.

The following example illustrates interposing on a panel item:

```
ptxt = xv_create(panel, PANEL_TEXT,
              PANEL_LABEL_STRING,  "Text:",
              PANEL_POST_EVENTS,   TRUE,
              NULL );

/* make textfield all upper case */
notify_interpose_event_func(ptxt, all_upper, NOTIFY_IMMEDIATE);

     [...............]

Notify_value
all_upper ( client, event, arg, type )
   Notify_client client;
   Notify_event event;
   Notify_arg arg;
   Notify_event_type type;
{
   int id = event_action((Event *) event);

   if ( isascii(id) && islower(id) )
     event_set_action((Event *) event, toupper(id));
```

```
        else if ( id == ACTION_PASTE )
            /* whatever, ..... */
    /*
    * panel_default_handle_event, text_handle_event,
    * next interposer, .....
    */
    return notify_next_event_func(client, event, arg, type);
    }
```

20.10 Notifier Control

The Notifier is started automatically by calling `xv_main_loop()`. The function `xv_main_loop()` takes a window object (typically the base frame of the application) and sets `XV_SHOW` to `TRUE`. Then it calls `notify_start()`, and the Notifier is running. The function `notify_start()` loops through the Notifier's processing loop, waiting for events in which its clients have expressed interest. This continues until the application calls `notify_stop()` or there are no more clients registered. At this time, `notify_start()` returns and the application continues or finishes. Essentially, the Notifer enters a loop, blocks until there is input (which it dispatches), and then blocks to wait for more input.

When `notify_start()` is called, the Notifier takes over and none of the application code is executed unless it is directly or indirectly called by *callback* procedure. If the callback routine calls `notify_stop()`, when that callback function returns, `notify_start()` returns and, as a result, `xv_main_loop()` returns. The Notifier can be stopped by using this method despite that there are still clients registered with the Notifier. At this point, the application may call `notify_start()` again, or it may attempt to do either *implicit* or *explicit* dispatching of events.

Explicit dispatching is done by calling `notify_dispatch()`. Here, the Notifier steps through one iteration of its normal control loop. This allows you to monitor events during the execution of a time consuming or computationally complex portion of the program. Say your application generates a complex fractal image. The process is initiated from the selection of a panel button. Because the generation of fractals is very time consuming, you may wish to check every once in a while to see if the user has generated any events that need to be processed—like selecting a panel button labeled `Stop`. In this case, the callback procedure for the panel button should call `notify_stop()` and return. The Notifier returns to the top-level, `notify_start()` returns (and thus, `xv_main_loop()` returns), and your application begins to generate the fractal image. During each iteration of the loop (or more often, if necessary), `notify_dispatch()` is called to ensure processing of any pending events that the user might have generated.

Implicit dispatching indicates that you are going to make calls to `read()` or `select()`, system calls that *block*. If `notify_do_dispatch()` is called, then events the user generates (such as moving the mouse or selecting a panel button) will continue to be processed even though the `read()` has not yet returned.

Implicit and explicit dispatching may be used before or after the call to `xv_main_loop()` or `notify_start()`, but it is not permitted to call these functions while the Notifier is looping on its own (from within a call to `notify_start()`). Therefore, you should never attempt to do direct dispatching from within a callback routine or any function that has been called indirectly by the Notifier.

These two methods of dispatching make porting programs that do not use the Notifier much easier. Thus, building an XView interface on top of a typical mainline-based, input-driven program is also much easier. Programs written from scratch should try to follow the event-driven input style of program design and try to avoid using implicit or explicit dispatching whenever possible.

20.10.1 Mass Destruction

The following routine causes all the client destruction routines to be called immediately with a destroy status set to `status`:

```
Notify_error
notify_die(status)
   Destroy_status   status;
```

This routine causes all the client destruction functions to be called immediately with `status` as the reason. The return values are `NOTIFY_OK` or `NOTIFY_DESTROY_VETOED`; the latter indicates that someone called `notify_veto_destroy()` and status was `DESTROY_CHECKING`. It is then the responsibility of the caller of `notify_die()` to exit the process, if so desired. Refer to the discussion on `notify_post_destroy` for more information.

20.10.2 Implicit Dispatching

Implicit dispatching is used whenever you wish to loop on a call that might block, such as `read` (2). Before calling `read()`, you should first call the function: `notify_do_dispatch();` This tells the Notifier that you are going to do implicit dispatching and that it should use its own version of `read()` rather than using the standard system call as the function. The two are equivalent with one exception:`read()` will return 0 on `EOF` rather than -1, as you might expect.

After `notify_do_dispatch()` has been called, you can call `notify_dispatch()` directly (to process events you know have already been delivered), call `read()`, or both.

The following example program demonstrates how this can be done. The program creates our usual frame, panel, and “Quit” panel button, but instead of calling `xv_main_loop()`, we create a small loop which reads `stdin` waiting for typed input.

Example 20-7. The ntfy_do_dis.c program

```
/*
 * ntfy_do_dis.c -- show an example of implicit notifier dispatching
 * by calling notify_do_dispatch().  Create a frame, panel and "Quit"
 * button, and then loop on calls to read() from stdin.  Event
 * processing is still maintained because the Notifier uses its own
 * non-blocking read().
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>

Frame frame;

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    char  buf[BUFSIZ];
    int   n, quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
        XV_HEIGHT,      100,
        XV_SHOW,        TRUE,
        NULL);

    panel = (Panel)xv_create (frame, PANEL, NULL);

    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);

    /* Force the frame to be displayed by flushing the server */
    XFlush(xv_get(frame, XV_DISPLAY));

    /* tell the Notifier that it should use its own read() so that it
     * can also detect and dispatch events.  This allows us to loop
     * in this code segment and still process events.
     */
    notify_do_dispatch();

    puts("Frame being displayed -- type away.");
    while ((n = read(0, buf, sizeof buf)) >= 0)
        printf("read %d bytes0, n);

    printf("read() returned %d0, n);
}

int
quit()
```

Example 20-7. The ntfy_do_dis.c program (continued)

```
{
    xv_destroy_safe(frame);
    return XV_OK;
}
```

There are several things to note here. First, because `xv_main_loop()` is not called, we need to explicitly set the attribute `XV_SHOW` to `TRUE` for the base frame. Otherwise, it will never be displayed. Also, because of the nature of event dispatching, we must *flush* the connection between the X server and the application to make sure that the frame is displayed by the time the first `read()` call returns. If this is not done, the frame is not displayed until after the first `read()` returns.

20.10.3 Explicit Dispatching

Frequently, the programmer is plagued with the following problem: A great deal of processing has to be done to process something that the user has initiated. As mentioned before, the programmer might want to generate a fractal image or compute the value of *pi* when the user selects a panel button. Any time-consuming process that requires this type of functionality should utilize explicit dispatching. It seems like an easy solution to `fork()` and let the background process perform the operation, but this is frequently an expensive operation and should be avoided if explicit dispatching is sufficient. If forking is required, then refer to Section 20.5.5, "Handling SIGCHLD," and Section 20.8.2, "Reading and Writing on Pipes."

The call `notify_dispatch()` does explicit event dispatching immediately to service an event that you know is waiting to be read and dispatched. Typically, this is called at particular locations within a control loop that is processing time consuming tasks. It is assumed that this control loop does *not* call `read()` or `select()`. Rather, explicit dispatching is used within loops that might do heavy computation such as graphics processing or number crunching. This is necessary because while the computations are busy computing, the user might be attempting to interact with the application by selecting a panel button.

If `notify_dispatch()` is called frequently enough, then performance of the user interface may still be perceived as acceptable even though the program is very busy with its computations. Architecturally, such a program is designed in a similar way with *ntfy_do_dis.c* in Section 20.10.1, "Implicit Dispatching." That is, there is a central processing loop which is doing the main work of the program. In this case, `notify_do_dispatch()` is not called.

```
#include <xview/xview.h>
 . . .
int finished;

main(argc, argv)
char *argv[ ];
{
    Display *dpy;
    Frame frame;

      ...

      xv_create(NULL, FRAME, NULL);
```

```
        ...

        dpy = (Display *)xv_get(frame, XV_DISPLAY);
        /* flush everything before starting loop */
        XFlush(dpy);
        while (!finished) {
            notify_dispatch();
            XFlush(dpy);
            /* compute PI to the next place */
            process_pi();
        }
    }
```

Notice that after `notify_dispatch()`, `XFlush()` is called. It is imperative that this happen; otherwise, any Xlib calls (which result from many `xv_create()` or `xv_set()` calls) may not be displayed. You only need to call `XFlush()` once after one or more calls to a sequence of calls to `notify_dispatch()`. The rule of thumb is to flush the server whenever you want to see the latest display.

20.11 Emulating a sleep() Call

XView allows you to emulate the system call `sleep()`. You cannot simply make the call since this would cause problems for the Notifier. Example 20-8 demonstrates how XView applications can implement a “sleep.”

Example 20-8. Emulating a sleep

```
#include <xview/xview.h>
#include <xview/panel.h>
#include <sys/time.h>

Frame   frame;
Panel   panel;

main(argc,argv)
int     argc;
char    *argv[ ];
{
        int     sleep_for_awhile();

        xv_init(XV_INIT_ARGS,   argc,argv, 0);

        frame = xv_create(0,FRAME, 0);

        panel = xv_create(frame,PANEL, 0);

        xv_create(panel,PANEL_BUTTON,
                PANEL_NOTIFY_PROC,      sleep_for_awhile,
                PANEL_LABEL_IMAGE,
                  panel_button_image(panel,"Sleep For Awhile",0,0),
                NULL);

        window_fit(panel);
        window_fit(frame);
```

Example 20-8. Emulating a sleep (continued)

```
        xv_main_loop(frame);
}

sleep_for_awhile()
{
        sleep(5);
}

sleep(sec)
int     sec;
{
        our_sleep(sec,0);
}

usleep(usec)
int     usec;
{
        our_sleep(0,usec);
}

our_sleep(sec,usec)
int     sec,usec;
{
        int     oldmask,mask;
        struct  timeval tv;

        tv.tv_sec = sec;
        tv.tv_usec = usec;

        mask = sigmask(SIGIO);
        mask |= sigmask(SIGALRM); /* change = to |= JLM */
        oldmask = sigblock(mask);

        if ((select(0,0,0,0,&tv)) == -1) {
                perror("select");
        }

        sigsetmask(oldmask);
}
```

20.12 Advanced Notifier Usage

This section covers Notifier *prioritization* and *scheduling*. These topics should be considered advanced topics, since their use changes the way XView schedules and prioritizes event handling in the Notifier. If you incorrectly modify these areas of the Notifier, you may experience severe problems with your application.

NOTE

These facilities should rarely be used by clients; a client should normally rely on XView's default scheduling and prioritizing scheme.

20.12.1 Prioritization

The order in which a particular client's conditions are notified may be controlled by providing a `prioritizer` operation. Assuming asynchronous or immediate notifications have already been sent, the default prioritizer makes its notifications in the following order:

- Interval timer notifications (`ITIMER_REAL` and then `ITIMER_VIRTUAL`).
- Child process control notifications.
- Synchronous signal notifications by ascending signal numbers.
- Exception file descriptor activity notifications by ascending `fd` numbers.
- Handle client events by order in which received.
- Output file descriptor activity notifications by ascending `fd` numbers.
- Input file descriptor activity notifications by ascending `fd` numbers.

20.12.1.1 Providing a prioritizer

This section describes how a client can provide its own prioritizer.

```
Notify_func
notify_set_prioritizer_func(client, prioritizer_func)
    Notify_client  client;
    Notify_func    prioritizer_func;
```

The function `notify_set_prioritizer_func()` takes an opaque client handle and the function to call before any notifications are sent to `client`. The previous function that would have been called is returned. If this function was never defined, then the default prioritization function is returned. If the `prioritizer_func()` argument supplied is `NOTIFY_FUNC_NULL`, then no client prioritization is done for `client` and the default prioritizer is used.

The calling sequence of a prioritizer function is shown below:

```
Notify_value
prioritizer_func(client, nfd, ibits_ptr, obits_ptr
                 ebits_ptr, nsig, sigbits_ptr, auto_sigbits_ptr,
                 event_count_ptr, events, args)
Notify_client       client;
fd_set              *ibits_ptr, *obits_ptr, *ebits_ptr;
int                 nfd, nsig, *sigbits_ptr, *auto_sigbits_ptr;
                    *event_count_ptr;
Notify_event        *events;
```

```
Notify_arg          *args;
#define SIGBIT(sig)        (1 << ((sig) -1))
```

In this function, `client` from the associated `notify_set_prioritizer_func()` is passed to `prioritizer_func()`. In addition, all the notifications that the Notifier is planning on sending to `client` are described in the other parameters. This data reflects only data that `client` has expressed interest in by asking for notification of these conditions. The remaining arguments and the return values are described in the *XView Reference Manual*.

20.12.1.2 Dispatching events

From within a prioritization routine, the following functions are called to cause the specified notifications to be sent:

- `notify_event()`
- `notify_input()`
- `notify_output()`
- `notify_exception()`
- `notify_itimer()`
- `notify_signal()`
- `notify_wait3()`

The notifier won't send any notifications it wasn't planning on sending anyway, so one can't use these calls to drive clients programmatically. A return value of `NOTIFY_OK` indicates that `client` was sent the notification. The return value for an unknown client, `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is not recognized by the Notifier and no notification was sent. A return value of `NOTIFY_NO_CONDITION` indicates that `client` does not have the requested notification pending and no notification was sent.

A client may choose to replace the default prioritizer. Alternatively, a client's prioritizer may call the default prioritizer after sending only a few notifications. Any notifications not explicitly sent by a client prioritizer will be sent by the default prioritizer (when called), in their normal turn. Once notified, a client will not receive a duplicate notification for the same event.

Signals indicated by bits in `sigbits_ptr` should call `notify_signal()`. Signals in `auto_sigbits_ptr` need special treatment:

- `SIGALRM` means that `notify_itimer()` should be called with a `which` of `ITIMER_REAL`.
- `SIGVTRM` means that `notify_itimer()` should be called with a `which` of `ITIMER_VIRTUAL`.
- `SIGCHLD` means that `notify_wait3()` should be called.
- `SIGTSTP` means `notify_destroy()` should be called with `status`

DESTROY_CHECKING.

- SIGTERM means notify_destroy() should be called with status DESTROY_CLEANUP.
- SIGKILL means notify_destroy() should be called with status DESTROY_PROCESS_DEATH.

Asynchronous signal notifications, destroy notifications, and client event notifications that were delivered right when they were posted do not pass through the prioritizer.

20.12.1.3 Getting the prioritizer

The routine notify_get_prioritizer_func() returns the current prioritizer of a client:

```
Notify_func
notify_get_prioritizer_func(client)
   Notify_client client;
```

This function takes an opaque client handle. The function that will be called before any notifications are sent to client is returned. If this function was never defined for client, then a default function is returned. A return value of NOTIFY_FUNC_NULL indicates an error. If client is unknown, then notify_errno is set to NOTIFY_UNKNOWN_CLIENT.

20.12.2 Scheduling the Notifier

Scheduling the notifier allows you to control the order in which clients are notified, which is done by a particular client's prioritizer function (refer to the previous section, "Prioritization"). The scheduler has the following calling sequence:

```
Notify_func
notify_set_scheduler_func(scheduler_func)
   Notify_func  scheduler_func;
```

The notify_set_scheduler_func() function allows you to arrange the order in which clients are called. The argument scheduler_func is the function to call to do the scheduling of clients. The previous function that would have been called is returned. This returned function will, almost always, be important to store and call later because it is most likely the default scheduler.

Replacement of the default scheduler is most often done by a client that needs to make sure that other clients don't take too much time servicing all of their notifications. For example, if doing "real-time" cursor tracking in a user process, the tracking client wants to schedule itself ahead of other clients, whenever there is input pending on the mouse.

The calling sequence of a scheduler function is:

```
Notify_value
scheduler_func(n, clients)
   int            n;
   Notify_client  *clients;
```

The argument `n` is passed into `scheduler_func()`. This is a list of clients, all of which are slated to receive some notification this time around. The scheduler scans `clients` and makes calls to `notify_client()` (refer to the next section, "Dispatching Clients"). Clients so notified should have their slots in `clients` set to `NOTIFY_CLIENT_NULL`. The return value from `scheduler_func()` is one of the following:

- `NOTIFY_DONE` – All of the clients had a chance to send notifications. The implies that no further clients should be scheduled this time around the notification loop. Unsent notifications are preserved for consideration the next time around the notification loop.
- `NOTIFY_IGNORED` – One or more `clients` were scheduled; that is, some `clients` may have been scheduled, but not all. This implies that another scheduler should try to schedule any clients in `clients` that are not `NOTIFY_CLIENT_NULL`.

20.12.2.1 Dispatching clients

The following routine is called from scheduler routines to cause all pending notifications for `client` to be sent:

```
Notify_error
notify_client(client)
   Notify_client  client;
```

The return value is `NOTIFY_OK`, `NOTIFY_NO_CONDITION`, or `NOTIFY_UNKNOWN_CLIENT`. The return value `NOTIFY_OK` indicates the client was notified. `NOTIFY_NO_CONDITION` indicates no conditions for client. This might mean `notify_client()` was already called with this client handle. `NOTIFY_UNKNOWN_CLIENT` indicates an unknown client.

20.12.2.2 Getting the scheduler

The following routine returns the function that will be called to do client scheduling:

```
Notify_func
notify_get_scheduler_func()
```

This function is always defined to be the default scheduler.

20.13 Error Codes

This section describes the basic error handling scheme used by the Notifier and lists the meaning of each of the possible error codes. Every call to the Notifier returns a value that indicates success or failure. On an error condition, `notify_errno` describes the failure. `notify_errno` is set by the Notifier as `errno` is set by UNIX system calls. (i.e., `notify_errno` is set only when an error is detected during a call to the Notifier. It is not reset to `NOTIFY_OK` on a successful call to the Notifier.)

```
enum notify_error {
    ... /* Listed below */
};
```

```
typedef enum notify_error Notify_error;

extern Notify_error notify_errno;
```

Table 20-1 contains a complete list of error codes.

Table 20-1. Notifier Error Codes

Error Code	Description
`NOTIFY_OK`	The call was completed successfully.
`NOTIFY_UNKNOWN_CLIENT`	The `client` argument is not known by the Notifier. A `notify_set_*_func` call needs to be made in order for the Notifier to recognize it.
`NOTIFY_NO_CONDITION`	A call was made to access the state of a condition, but the condition was not set with the Notifier for the client in question. This situation can occur when a `notify_get_*_func()` type call is made before the equivalent `notify_set_*_func()`. Also, the Notifier automatically clears some conditions after they have occurred, e.g., when an interval timer expires.
`NOTIFY_BAD_ITIMER`	The `which` argument to an interval timer routine was not valid.
`NOTIFY_BAD_SIGNAL`	The `signal` argument to a signal routine was out of range.
`NOTIFY_NOT_STARTED`	A call to `notify_stop()` was made, but the Notifier was never started.
`NOTIFY_DESTROY_VETOED`	A client refused to be destroyed during a call to `notify_die()` or `notify_post_destroy()` when `status` was `DESTROY_CHECKING`.
`NOTIFY_INTERNAL_ERROR`	Some internal inconsistency in the Notifier itself has been detected.
`NOTIFY_SRCH`	The `pid` argument to a child process control routine was not valid.
`NOTIFY_BADF`	The `fd` argument to an input or output routine was not valid.
`NOTIFY_NOMEM`	The Notifier dynamically allocates memory from the heap. This error code is generated if the allocator could not get any more memory.
`NOTIFY_INVAL`	Some argument to a call to the Notifier contained an invalid argument.
`NOTIFY_FUNC_LIMIT`	An attempt to set an interposer function has encountered the limit of the number of interposers allowed for a single condition.

The routine `notify_perror()` acts like library call `perror (3)`.

```
notify_perror(str)
    char *str;
```

`notify_perror()` prints the string `str`, followed by a colon, and followed by a string that describes `notify_errno` to `stderr`.

20.14 Issues

Here are some additional issues surrounding the Notifier:

- The layer over the UNIX signal mechanism is not complete. Signal blocking (`sigblock` (2)) can still be done safely in the flow of control of a client to protect critical portions of code as long as the previous signal mask is restored before returning to the Notifier. Signal pausing (`sigpause` (2)) is essentially done by the Notifier. Signal masking (`sigmask` (2)) can be accomplished via multiple `notify_set_ signal_func()` calls. Setting up a process signal stack (`sigstack` (2)) can still be done. Setting the signal catcher mask and on-signal-stack flag (`sigvec` (2)) could be done by reaching around the Notifier, but this is not supported.
- Not all process resources are multiplexed (e.g., `rlimit (2)`, `setjmp (2)`, `umask (2)`, `setquota (2)`, and `setpriority (2))`, only ones that have to do with flow of control multiplexing. Thus, some level of cooperation and understanding needs to exist among packages in the single process.
- One might intercept `close` (2) and `dup` (2) calls so that the Notifier is not waiting on invalid or incorrect file descriptors if a client forgets to remove its conditions from the Notifier before making these calls.
- One might intercept `signal` (3) and `sigvec` (2) calls so that the Notifier does not get confused by programs that fail to use the Notifier to manage its signals.
- One might intercept `setitimer` (2) calls so that the Notifier does not get confused by programs that fail to use the Notifier to manage interval timers.
- One might intercept `ioctl`(2) calls so that the Notifier does not get fouled up by programs that use `FIONBIO` and `FIOASYNC` instead of the equivalent `fcntl` (2) calls.
- One might intercept `readv` (2) and `write` (2) just like `read` (2) and `select` (2) so that a program does not tie up the process.
- The Notifier is not a lightweight process mechanism that maintains a stack per thread of control. However, if such a mechanism becomes available, then the Notifier will still be valuable for its support of notification-based clients.
- Client events are disjointed from UNIX events. This is done to give complete freedom to clients as to how events are defined. One could imagine certain clients wanting to unify client and UNIX events. This could be done with a layer of software on top of the Notifier. A client could define events as pointers to structures that contain event codes and event specific arguments. The event codes would include the equivalents of UNIX event notifications. The event specific arguments would contain, for example, the file descriptor of an input-pending notification. When an input-pending notification from the Notifier was sent to a client, the client would turn around and post the equivalent client event notification.

- One could imagine extending the Notifier to provide a record and replay mechanism that would drive an application. However, this is not supported by the current interface.

21
Color

The X Window System has various ways of allocating, specifying, and using colors. While all of these methods are available to applications without XView intervening, XView provides its own model for color specification that may be used as an alternative. It does not provide anything more than what is already available, but it may provide a simpler interface to request and specify colors. This model is especially useful when specifying colors for XView objects, such as panel buttons and scrollbars.

This chapter does not directly discuss how to use colormaps and related Xlib color-specific functions. For a discussion of them, see Volume One, *Xlib Programming Manual*, Chapter 7, *Color*. This chapter discusses only the XView color model.

Obviously, the user cannot view colors in an application without having a color display. But you cannot tell at the time your application is written whether the user's display is going to be able to support color. You can use the `DisplayDepth()` macro to determine whether the user's display can handle color:

```
Display *dpy = (Display *)xv_get(frame, XV_DISPLAY);
extern use_color;

if (DefaultDepth(dpy, DefaultScreen(dpy)) < 2)
    use_color = False;
```

21.1 XView Color Model

XView applications deal with color by using *colormap segments*. Use the `CMS` package to create a colormap segment. Figure 21-1 shows the class hierarchy for the `CMS` package.

As a simple introduction, the following code fragment creates a colormap segment with the specified colors and returns a handle to it:

```
cms = (Cms)xv_create(NULL, CMS,
    CMS_SIZE,         4,
    CMS_NAMED_COLORS, "white", "red", "green", "blue", NULL,
    NULL);
```

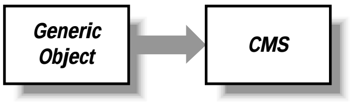


Figure 21-1. CMS package class hierarchy

Window-based objects (canvases, panels, textsw, etc.) use colormap segments to get their colors. These objects get a default colormap segment when they are created, but you can assign a new one using the `WIN_CMS` attribute:

```
canvas = (Canvas)xv_create(frame, CANVAS,
    WIN_CMS, cms,
    NULL);
```

Colormap segments must be applied to windows to assure that the window can access the color you are attempting to draw into. However, there is much to understand about colors, colormaps, colormap segments, visuals, servers, and X11 to be able to use colors in an efficient and robust way. Unless done correctly, you may produce code that only works on specific machines or in specific environments.

21.1.0.1 What is a colormap segment?

A colormap segment (cms), is a subset of the available cells in a colormap on the X server. These are XView entities (i.e., not Xlib) that provide a veneer over the Xlib color mechanism. Colormap segments can be created as either *static* or *dynamic* and are derived from an underlying colormap of the same type.

Any object subclassed from the window object may allocate and use colormap segments. You can use Xlib routines to do all your color and colormap manipulation within canvas windows, pixmaps, and other X-related objects, but you must use the colormap segment API for XView objects.

More than one XView object may reference the same colormap segment. However, a colormap segment does not keep track of the objects that are using it. The application is required to keep track of changes in colors and update its objects accordingly.

The internals to XView attempt to create colormap segments that are as small as possible so that numerous segments can share the same underlying colormap. If a colormap segment requires more colors than the current colormap has space for, a new colormap must be created.

21.1.1 Colormap Segment Types

You can create static or dynamic colormap segments. The type is set with the CMS_TYPE attribute. Its value can be either XV_STATIC_CMS or XV_DYNAMIC_CMS. A colormap segment's type cannot change, so the CMS_TYPE attribute is for xv_create() only.

The X11 Protocol specifies that the Visual type of a window must be declared at the time it is created. Therefore, XView objects that allocate colormap segments are required, at creation time, to specify the type of visual they propose to use. The default visual is determined by the default visual of the screen.

21.1.1.1 Static colormap segments

Applications must always use static colormap segments unless they require read-write colors. Colors allocated from a static colormap segment are shared among *all* applications. In static colormap segments, when a new color is asked for, the XView library will try to return the closest (or exact) matching color from the server (using XAllocColor()) if the default colormap on the server is StaticColor.

Whenever possible, a colormap segment is derived from the default colormap obtained from the screen in which the window resides (e.g., DefaultColormap()). Only when the colors on that colormap have been exhausted is a new colormap allocated. If XView needs to allocate a new colormap for a new cms, it does this internally. It is impossible for the application to specify a colormap for a colormap segment.

The cells in a static cms are initialized once and are read-only from then on. Static colormap segments, by sharing color cells across applications, use the shared hardware colormap resources more efficiently and reduce *flashing*. Flashing is a blinking effect you sometimes get when moving the cursor in and out of various windows on the screen. This is caused by the server popping different colormaps in and out as you move from one window to the next.

21.1.1.2 Dynamic colormap segments

When you ask for a dynamic cms, XView sends a request to the server to allocate read-write colors. When colors are requested from this cms, the color returned is the exact color; the closest match is not returned as it is with static colors.

21.2 Creating Colormap Segments

Applications that use color must include the file *<xview/cms.h>*. A cms can be created using the standard call to `xv_create()` with the package name `CMS`.

```
Cms cms;

cms = (Cms)xv_create(parent, CMS, attrs, NULL);
```

The *parent* of a colormap segment is the XView screen object with which the colormap is associated. If a parent is not specified, the default screen of the default server is used as the parent.

21.2.0.1 Cms size

A cms may contain as many colors as you like as long as they fit within the largest colormap you can create. Having more than one colormap segment reference the same color value is perfectly legal and reasonable. Data is frequently shared among segments for optimal efficiency.

When creating a colormap segment, you must specify its size (i.e., the number of colors it has) using the `CMS_SIZE` attribute. If you don't set the size, it defaults to the macro, `XV_DEFAULT_CMS_SIZE`, which is 2. `CMS_SIZE` is a create-only and get-only attribute; once a colormap is created, its size cannot be changed, although you can query a colormap's size using `xv_get()`.

If all the colors in a colormap segment are not initialized, the uninitialized colors are undefined and should not be used. You can change colors within a dynamic cms at any time. Uninitialized colors of a static cms can be initialized using `xv_set()`, but once initialized, they may not be changed.

You might want to create a segment of a larger size than the number of colors you assign it because you may not have all the colors you know you'll need right away. For example, you create a colormap segment of size *n*, but initialize it with only *n-4* colors. When the rest of the colors are ready to be loaded into the segment, you do so at the location of the uninitialized colors, namely, at index *n-4*. The `CMS_INDEX` attribute is used to specify this location.

CAUTION

All the colors in a static cms must be specified at the time of creation to avoid race conditions in the associated X11 colormap. Because static colormaps are shared with other applications, if you request *n* colors but do not initialize all of them, another application could request and initialize enough colors to fill up the colormap before you get a chance to set the rest of your colors.

When creating a cms or setting new colors, you may specify `CMS_COLOR_COUNT` to indicate the number of colors to load. Again, if you want to load these colors at a position other than the beginning of the segment, use `CMS_INDEX`. This is typically used only when you are creating a colormap segment and not initializing each color right away. Or, if you have already done this, you are adding more colors to a prebuilt colormap segment that hasn't had all of its colors initialized. Therefore, this is a create-only and set-only attribute.

21.2.1 Specifying Colors

You can specify actual colors by name or by RGB values. When using RGB values, you can use XView or Xlib data structures. In each case, we are going to create a colormap segment with the same four colors: white, red, green, and blue.

21.2.1.1 Specifying colors by name

The attribute `CMS_NAMED_COLORS` takes as its value a `NULL`-terminated list of strings representing color names:

```
cms = (Cms)xv_create(parent, CMS,
    CMS_SIZE,         4,
    CMS_NAMED_COLORS, "white", "red", "green", "blue", NULL,
    NULL);
```

The colors specified by the names are converted into actual values using `XParseColor()`* and allocated into the colormap segment using `XAllocColor()` (for static colormaps) or `XStoreColor()` (for dynamic colormaps). The example shown probably works because the colors used are common colors found on most X servers' color databases. However, you should be careful when requesting named colors in this fashion because the database may not contain the color name you specify. If any of the colors requested fails, then no change to the cms is affected and `xv_set()` returns `XV_ERROR`. If a cms is being created via `xv_create()` and an error occurs, no cms is created and `xv_create()` returns `NULL`.

`CMS_NAMED_COLORS` cannot be used by `xv_get()`.

21.2.1.2 Specifying colors by RGB values

You can request colors more directly by specifying the actual red, green, and blue (RGB) values using one of two attributes. `CMS_COLORS` takes as a value an array of `Xv_singlecolor` objects. This XView-defined type is declared as:

```
typedef struct xv_singlecolor {
    unsigned char red, green, blue;
} Xv_singlecolor;
```

*`XParseColor()` is an Xlib call that maps `char *` color names into RGB values.

We can use the following to produce a cms with the same colors:

```
static Xv_singlecolor colors[ ] = {
    { 255, 255, 255, }, /* white */
    { 255, 0, 0 },      /* red */
    { 0, 255 0, },      /* green */
    { 0, 0, 255 },      /* blue */
};
cms = xv_create(NULL, CMS,
    CMS_SIZE,    4,
    CMS_COLORS,  colors,
    NULL);
```

Alternatively, you can use the attribute `CMS_X_COLORS` to specify an array of `XColor` structures, defined in <*X11/Xlib.h*> as:

```
typedef struct {
    unsigned       long pixel;
    unsigned short red, green, blue;
    char           flags;  /* do_red, do_green, do_blue */
    char           pad;
} XColor;
```

Here is a way to produce a colormap segment with the same colors but using an array of `XColors`:

```
static XColor colors[ ] = {
/* white */ { 0, 255<<8, 255<<8, 255<<8, DoRed|DoGreen|DoBlue, 0 },
/* red */   { 0, 255<<8,      0,      0, DoRed|DoGreen|DoBlue, 0 },
/* green */ { 0,    0,   255<<8,      0, DoRed|DoGreen|DoBlue, 0 },
/* blue */  { 0,    0,        0, 255<<8, DoRed|DoGreen|DoBlue, 0 },
};
cms = xv_create(NULL, CMS,
    CMS_SIZE,      4,
    CMS_X_COLORS,  colors,
    NULL);
```

Note that the color values in the red, green, and blue fields of the `XColor` data structure are left-shifted by 8. For more details on specifying colors with Xlib, see Volume One, *Xlib Programming Manual*.

When storing colors, if the colormap segment type is static, XView uses `XAllocColor()`. If the colormap segment type is dynamic, XView uses `XAllocColorCells()` and `XStoreColors()` to allocate a dynamic colormap and store the requested colors in it.

After setting colors in a cms, the pixel values for the colors can be retrieved. These pixel values are indices into the colormap itself, not the colormap *segment*. See Section 21.3, "Color and Pixel Values."

The XView attributes to set colors can be used to get the colors from the cms as well. For example, to get the colors into an array of `Xv_singlecolor`, use:

```
Xv_singlecolor colors[SIZE];

xv_get(cms, CMS_COLORS, colors);
```

The size of the `colors` array *must be the same size as the color segment* because `xv_get()` gets the entire colormap segment and stores the colors at the base of the array.

You cannot get a partial list of colors from the cms. The same is true for getting the colors into an array of `XColor`:

```
XColor colors[SIZE];

xv_get(cms, CMS_X_COLORS, &colors);
```

In each case, `xv_get()` returns a pointer to the base of the array of data structures passed as the third argument. We choose to ignore it here because the array itself is sufficient.

21.2.2 Cms Name

Colormap segments can be named using `CMS_NAME`.* Windows can change between colormaps by setting their `WIN_CMS_NAME` to the names of allocated colormap segments. However, this old-style method of specifying colormap segments for windows is made obsolete by the attribute `WIN_CMS`, the recommended method for assigning colormap segments to windows.

If unnamed, a cms will get a unique name assigned to it. You can retrieve that name using `xv_get()` and `CMS_NAME`. `CMS_NAME` is also the only attribute for which you can use `xv_find()` for the `CMS` package.

21.3 Color and Pixel Values

By now, you know that a *color* is defined to be a set of red, green, and blue (RGB) intensity values. For example, red is represented by a full intensity for the red value and no intensity for the green and blue values. Other shades are achieved by raising or lowering the intensities of one or more of the RGB values. A colormap is an array of RGB values; a *pixel* is an index into that array.

21.3.0.1 Logical versus real indices

If a colormap segment of size *n* is created, its logical indices range from 0 to *n*-1. XView attributes that take color values (e.g., `WIN_FOREGROUND_COLOR`, `PANEL_ITEM_COLOR`, etc.) always deal with logical index values.

The *real* indices of a colormap segment are the actual indices into the hardware colormap. Each colormap segment maintains an internal table to translate from logical to real indices. Real index values (*pixels*) are used for setting the foreground and background colors in `GC`s used in Xlib calls.

*`CMS_NAME` is defined to be `XV_NAME`.

The index table from a colormap segment can be obtained using the attribute CMS_INDEX_TABLE.

```
unsigned long *colors;

colors = (unsigned long *)xv_get(cms, CMS_INDEX_TABLE);
```

Similarly, the pixel values from a *window-based object* can be obtained using the window attribute, WIN_X_COLOR_INDICES:

```
unsigned long *colors;

colors = (unsigned long *)xv_get(canvas, WIN_X_COLOR_INDICES);
```

Note the object passed to xv_get().

In both cases, the returned value is a pointer to an array of unsigned long types. For a colormap segment of size *n*, colors[0], colors[1], . . . colors[*n*-1] contain the actual pixel values corresponding to the colors in the underlying X11 colormap. These pixel values can be used in calls to XSetForeground() or XSetBackground() to change GC values.

To get the real pixel value corresponding to a logical index value, you can use the CMS_PIXEL attribute:

```
Cms cms;
unsigned long red, blue;

cms = (Cms)xv_create(NULL, CMS,
    CMS_SIZE,          2,
    CMS_NAMED_COLORS,  "red", "blue", NULL,
    NULL);

red = (unsigned long)xv_get(cms, CMS_PIXEL, 0);
blue = (unsigned long)xv_get(cms, CMS_PIXEL, 1);
```

CMS_PIXEL can only be used to get the value of pixels; you cannot use it to set cms color values.

As discussed earlier, the CMS_X_COLORS attribute can be used to get an array of XColor elements. This data structure has a pixel field that can be referenced to get the pixel values associated with colors.

21.3.1 Foreground and Background Colors

Foreground and background colors correspond to the last and first colors in a colormap segment. That is, the segment's *background* color corresponds to the logical index 0 in the cms, whereas the *foreground* color corresponds to the logical index *n*-1 (where *n* is the size of the cms).

Identifying the foreground and background colors on certain XView objects may not be as straightforward as it appears. For example, the background color of a canvas is the color the entire canvas is painted with when XClearArea() is called. The foreground color is a thin border color around the inside perimeter of each canvas view window. This is not necessarily the color in which graphics are rendered into the canvas using `XCopyArea()` or `XDrawString()`. The color of the graphical images you see in a canvas is dependent on the foreground color set in the `GC` used by the Xlib routines.

The real pixel values for the foreground and background colors of a cms may be obtained directly using the attributes `CMS_FOREGROUND_PIXEL` and `CMS_BACKGROUND_PIXEL`.

```
Cms            cms;
unsigned long  fg, bg;

bg = (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
fg = (unsigned long)xv_get(cms, CMS_FOREGROUND_PIXEL);
```

21.3.1.1 Colors of control objects

A *control object* in XView refers to panels, scrollbars, notices, and menus. These packages cannot have their foreground and background modified programmatically in the same way as other window-based objects. This restriction applies to the 3D interface; the 2D interface may allow the foreground and background colors to be modified. However, OPEN LOOK states that the background of all control objects appear consistent; thus, a single *control color* is used. On panels, it is possible, but not recommended, to set the `WIN_FOREGROUND_COLOR` and `WIN_BACKGROUND_COLOR` attributes. Panel items, may have their colors set by setting `PANEL_ITEM_COLOR` to a logical index into the panel's cms. XView allows the user, not the programmer, to set the background color for control objects via the resource database. This is discussed in Section 21.5, "The Control Colormap Segment."

21.4 The color_logo.c Program

Using the basic principles discussed so far, we present Example 21-1 to demonstrate the creation and initialization of a colormap segment for an XView canvas. Pixel values are extracted from the colormap segment and set into a GC's foreground. Xlib calls are then used to render various items in the same four basic colors we've been using.

Example 21-1. The color_logo.c program

```
/* color_logo.c --
 *  This program demonstrates the combined use of the XView color
 *  model/API and Xlib graphics calls. The program uses XView to
 *  create and manage its colormap segment while doing its actual
 *  drawing using Xlib routines.
 *  The program draws the X logo in red, green and blue in a canvas.
 */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/cms.h>
#include <xview/xv_xrect.h>
#include <X11/bitmaps/xlogo64>

/* Color indices */
```

Example 21-1. The color_logo.c program (continued)

```
#define WHITE           0
#define RED             1
#define GREEN           2
#define BLUE            3
#define NUM_COLORS      4

GC gc;                      /* used for rendering logos */
unsigned long *pixel_table; /* pixel values for colors */
Pixmap        xlogo;        /* the xlogo */

/* Create a frame, canvas, and a colormap segment and assign the
 * cms to the canvas.  CMS_INDEX_TABLE returns the actual colormap
 * indices and are used to set the gc’s foreground for XCopyPlane
 * calls.
 */
main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame         frame;
    XGCValues     gc_val;
    XGCValues     gcvalues;
    void          canvas_repaint_proc();
    Cms           cms;
    static Xv_singlecolor colors[ ] = {
        { 255, 255, 255 }, /* white */
        { 255,   0,   0 }, /* red */
        { 0,   255,   0 }, /* green */
        { 0,     0, 255 }, /* blue */
    };

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    cms = (Cms) xv_create(NULL, CMS,
        CMS_SIZE, 4,
        CMS_COLORS, colors,
        NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       448,
        XV_HEIGHT,      192,
        NULL);

    (void) xv_create(frame, CANVAS,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    canvas_repaint_proc,
        WIN_CMS,                cms,
        NULL);

    /* Get the actual indices into the colormap */
    pixel_table = (unsigned long *)xv_get(cms, CMS_INDEX_TABLE);

    /* create the xlogo -- get display/window from the frame obj */
    xlogo = XCreateBitmapFromData(
        xv_get(frame, XV_DISPLAY), xv_get(frame, XV_XID),
```

Example 21-1. The color_logo.c program (continued)

```
        xlogo64_bits, xlogo64_width, xlogo64_height);

    /* setup gc for rendering logos to screen */
    gcvalues.graphics_exposures = False;
    gcvalues.background = pixel_table[WHITE];
    gc = XCreateGC(xv_get(frame, XV_DISPLAY), xv_get(frame, XV_XID),
        GCBackground | GCGraphicsExposures, &gcvalues);

    xv_main_loop(frame);
}

/* Draws onto the canvas using Xlib drawing functions.
 * Draw the X logo into the window in three colors. In each case,
 * change the GC's foreground color to the pixel value specified.
 */
void
canvas_repaint_proc(canvas, pw, display, win, xrects)
Canvas      canvas;   /* unused */
Xv_Window   pw;       /* unused */
Display     *display;
Window      win;
Xv_xrectlist *xrects; /* unused */
{
    /* Use XCopyPlane because the logo is a 1-bit deep pixmap. */
    XSetForeground(display, gc, pixel_table[RED]);
    XCopyPlane(display, xlogo, win, gc, 0, 0,
        xlogo64_width, xlogo64_height, 64, 64, 1);

    XSetForeground(display, gc, pixel_table[GREEN]);
    XCopyPlane(display, xlogo, win, gc, 0, 0,
        xlogo64_width, xlogo64_height, 192, 64, 1);

    XSetForeground(display, gc, pixel_table[BLUE]);
    XCopyPlane(display, xlogo, win, gc, 0, 0,
        xlogo64_width, xlogo64_height, 320, 64, 1);
}
```

Example 21-1 uses Xlib routines to draw into the canvas's paint window. Therefore, the `GC`'s foreground color is set to an index from the colormap being used by that paint window. In order to get the correct color from the colormap, we need to get the color table for the window using the attribute `CMS_COLOR_INDEX`. The repaint routine draws the X logo in the specified colors.

21.5 The Control Colormap Segment

The management of colormap segments is a little different for control objects (panels, notices, menus, etc.) than it is for other XView objects. In order for XView to provide the same control colors for control objects, these objects must use a *control colormap segment*. This is just like a normal cms except that parts of it are reserved for the predefined colors.

Since OPEN LOOK suggests that the background of control objects in an application appear in a consistent color, XView sets that color to be that specified by the resource `OpenWindows.WindowColor` from the user's environment. This color is used as the background color, and along with a few others, they are set aside in the first few indices of the control colormap segment.

These *control* colors are used to provide a 3D look for the control objects. Thus, the control colormap segment must be used for all control objects.

Aside from the added colors, there is little difference between a control cms and a normal cms. The cms may still request many colors and there is no limit to the choice of colors used in the new cms. However, these extra colors cannot be used as the background for control objects.

A control colormap segment is created by setting the boolean attribute `CMS_CONTROL_CMS` to `TRUE` in the call to `xv_create()`. The macro `CMS_CONTROL_COLORS` (defined in <*xview/cms.h*>) indicates how many predefined control colors there are, so the first `CMS_CONTROL_COLORS` indices in the cms are initialized by the XView library. If the application requires *n* other colors in this cms, it must explicitly ask that the segment be created with a size of:

```
n + CMS_CONTROL_COLORS
```

In such a case, the application must refer to its own *n* colors using the index range:

```
CMS_CONTROL_COLORS to CMS_CONTROL_COLORS + n-1
```

An application-defined colormap segment set on a control object (such as a panel) must be a control colormap segment so that the object can be pointed with the 3D look.

```
#define WHITE       0
#define RED         1
#define GREEN       2
#define BLUE        3
#define NUM_COLORS  4

control_cms = xv_create(NULL, CMS,
    CMS_SIZE,          CMS_CONTROL_COLORS + NUM_COLORS,
    CMS_CONTROL_CMS,   TRUE,
    CMS_NAMED_COLORS,  "white", "red", "green", "blue", NULL,
    NULL);
```

We set the boolean attribute CMS_CONTROL_CMS to TRUE to indicate that we are creating a control cms. Notice that the size of the colormap segment is the number of colors *we* specified plus the number of control colors. This colormap segment contains both control colors and our colors; XView automatically allocates our colors after the control colors.

When we create the panel, we specify the new colormap segment as the panel's cms using the common window attribute, WIN_CMS:

```
    panel = (Panel)xv_create(frame, PANEL,
        WIN_CMS, control_cms,
        NULL);
```

When we reference our own specified colors, we must offset those color values by CMS_CONTROL_COLORS:

```
    /* assume a 1-bit deep 16x16 square pixmap */
    extern Server_image chip;

    xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING, "Colors",
        PANEL_CHOICE_IMAGES, chip, chip, chip, chip, NULL,
        PANEL_CHOICE_COLOR, 0, WHITE + CMS_CONTROL_COLORS,
        PANEL_CHOICE_COLOR, 1, RED + CMS_CONTROL_COLORS,
        PANEL_CHOICE_COLOR, 2, GREEN + CMS_CONTROL_COLORS,
        PANEL_CHOICE_COLOR, 3, BLUE + CMS_CONTROL_COLORS,
        NULL);
```

Here, we create a choice item whose choices are colored "chips" in solid colors corresponding to the color names offset by the control colormap segment.

21.5.1 Coloring Panel Items

You can specify colors for panel items using PANEL_ITEM_COLOR. This attribute takes as a value an index into a colormap segment. The value -1 is reserved for the panel's foreground color, whatever that may be. It is also the default color of panel items unless you have specified otherwise with PANEL_ITEM_COLOR. The scrollbar on a PANEL_LIST will always take on the window color that is specified in the Workspace Properties sheet. PANEL_ITEM_COLOR on a PANEL_LIST will only affect the scrolling list's label, title, and rows. Remember, if you're going to be using the 3D interface, then you must create the cms as a *control* cms. Example 21-2 briefly demonstrates how to create colored panel items.

Example 21-2. The color_panel.c program

```
/* color_panel.c --
 * This program demonstrates how to set panel items to different
 * colors using the XView API for color.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/cms.h>

/* Color indices */
#define WHITE           0
#define RED             1
```

Example 21-2. The color_panel.c program (continued)

```
#define GREEN           2
#define BLUE            3
#define NUM_COLORS      4

/* Create a frame, panel, and a colormap segment and assign the
 * cms to the panel.
 */
main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Cms         cms;
    extern void exit(), pressed();
    static Xv_singlecolor colors[ ] = {
        { 255, 255, 255 }, /* white */
        { 255,   0,   0 }, /* red */
        { 0,   255,   0 }, /* green */
        { 0,     0, 255 }, /* blue */
    };

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    cms = (Cms) xv_create(NULL, CMS,
        CMS_CONTROL_CMS,        TRUE,
        CMS_SIZE,               CMS_CONTROL_COLORS + 4,
        CMS_COLORS,             colors,
        NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);

    panel = xv_create(frame, PANEL,
        WIN_CMS,        cms,
        NULL);

    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Red",
        PANEL_ITEM_COLOR,       CMS_CONTROL_COLORS + RED,
        PANEL_NOTIFY_PROC,      pressed,
        NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Green",
        PANEL_ITEM_COLOR,       CMS_CONTROL_COLORS + GREEN,
        PANEL_NOTIFY_PROC,      pressed,
        NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Blue",
        PANEL_ITEM_COLOR,       CMS_CONTROL_COLORS + BLUE,
        PANEL_NOTIFY_PROC,      pressed,
        NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
```

Example 21-2. The color_panel.c program (continued)

```
        PANEL_ITEM_COLOR,       CMS_CONTROL_COLORS + WHITE,
        PANEL_NOTIFY_PROC,      exit,
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

void
pressed(item, event)
Panel_item item;
Event *event;
{
    char *name = (char *)xv_get(item, PANEL_LABEL_STRING);
    Frame frame = xv_get(xv_get(item, PANEL_PARENT_PANEL), XV_OWNER);

    xv_set(frame, FRAME_LEFT_FOOTER, name, NULL);
}
```

Notice how the color index for each panel item is offset by the number of colors in the control color item. If all the references to the `CMS_CONTROL_COLORS` and `CMS_CONTROL_CMS` attributes were removed, the result would be a 2D panel whose panel items are the same colors as their names.

21.6 Using xv_find() with Colormap Segments

`xv_find()` can be used to find a previously created colormap segment. Currently, the attribute `CMS_NAME` is the only attribute used by `xv_find()` to find a match.

```
    cms = (Cms)xv_find(screen, CMS,
        CMS_NAME, "palette",
        XV_AUTO_CREATE, FALSE,
        NULL);
```

This example returns a handle to a colormap segment whose name is "palette." `XV_AUTO_CREATE` is set to `FALSE` to prevent a new colormap segment from being created. If the colormap segment with that name is not found, then `NULL` is returned.

21.7 Canvases and Colormaps

When the colormap segment associated with a canvas is changed, the contents of the canvas must be repainted to reflect the new colors. If the boolean attribute CANVAS_CMS_REPAINT is set to TRUE, the library automatically calls the canvas's repaint procedure each time a new colormap segment is set on the canvas. The damage list passed to the routine contains the dimensions of the entire paint window.

Note that the application itself must track any changes in the contents of a colormap segment. CANVAS_CMS_REPAINT enables the library to generate a synthetic repaint event only when the actual colormap segment is switched.

For dynamic colormap segments, when a color changes, the pixel value remains the same but the *color* represented by the index into the colormap segment changes. Therefore, the repaint routine is not called and the window's appearance changes automatically. This method of colormap manipulation is commonly used to implement color animation.

21.8 Multi-visual Support

XView allows you to create windows and colormap segments using arbitrary visuals (See Chapter 7 of Volume One, *Xlib Programming Manual* for a description of visuals in X11). You can use this functionality to access any visual that the X11 server supports. The attributes described in this section allow you to indicate visuals you want associated with your windows and colormap segments.

The attribute XV_VISUAL specifies the exact visual that will be used in the creation of a window or colormap segment. The value is a pointer a Visual structure (available from XMatchVisualInfo or XGetVisualInfo). This attribute applies only to WINDOW and CMS objects.

The attribute XV_VISUAL_CLASS specifies the class of the visual that will be used in the creation of the window or colormap segment. The value should be one of the following:

```
StaticGray
GrayScale
StaticColor
PseudoColor
TrueColor
DirectColor
```

If the server that the window is being created for does not support the visual class specified, XView uses the default visual. XV_VISUAL_CLASS also only applies to a WINDOW or to a CMS object.

The attribute WIN_DEPTH specifies the depth of the window that you are creating. If the value specified by this attribute is not supported by the server, the library uses the default depth instead. This attribute, like all WIN_* attributes applies only to WINDOW objects.

21.8.1 Using the Visual Attributes

Both XV_VISUAL and XV_VISUAL_CLASS do essentially the same thing. XV_VISUAL is more specific. Use XV_VISUAL_CLASS, optionally in conjunction with WIN_DEPTH, if you have no preference about the specific visual, but just require a certain visual class. However, if you want to make sure that you are using a specific visual that the server supports, use XGetVisualInfo or XMatchVisualInfo to find the visual and then use the visual as an argument to XV_VISUAL.

If you are creating your own colormap segment(s), make sure that they are created with the same visual as the window you will be using with the CMS. For example, if you have a window that requires a cms, do the following:

```
cms = (Cms)xv_create(screen, CMS,
                XV_VISUAL, (Visual *)xv_get(window, XV_VISUAL),
                ...
                NULL);
```

Note that you should never assume that creating a window with XV_VISUAL_CLASS or WIN_DEPTH will use the class and depth you specify. If the server does not have a visual of the given class and depth, you will not be supplied with the visual you specify. If your application depends on using a specific visual, be sure to check the visual class and depth of the window, using xv_get(), to see if you did indeed get the required visual. Alternatively, you may query the server before the creating the window, using XGetVisualInfo or XMatchVisualInfo, to check the visuals that the server supports.

21.9 Another Example

This final example demonstrates just about all of the features discussed in this chapter. It includes creating a colormap segment, initializing color, using foreground and background colors, and setting colors on specific XView objects, including panel items.

In Example 21-3, the user selects objects (from the "Objects" item) to be colored by selecting one of the color tiles from the "Colors" choice item. The callback function for this panel item calls color_notify(), which sets the currently selected colors on the foreground or background of the selected objects. Whether to use foreground or background colors is dependent on the value of the "Fg/Bg" panel item; the items whose colors are affected are retrieved by getting the value of the "Objects" panel item. Since more than one object can be selected from this item, the value is a mask of the selected items. We loop through each bit in the mask identifying which objects should have their colors set.

Example 21-3. The color_objs.c program

```
/*
 * color_objs.c --
 *    This program demonstrates the use of color in XView. It allows
 *    the user to choose the foreground and background colors of the
 *    various objects in an interactive manner.
 */
#include <xview/xview.h>
```

Example 21-3. The color_objs.c program (continued)

```
#include <xview/svrimage.h>
#include <xview/textsw.h>
#include <xview/panel.h>
#include <xview/cms.h>
#include <xview/notice.h>

#define SELECT_TEXTSW           0
#define SELECT_TEXTSW_VIEW      1
#define SELECT_PANEL            2
#define SELECT_ICON             3

#define NUM_COLORS              8

/* Icon data */
static short icon_bits[ ] = {
#include "cardback.icon"
};

/* solid black square */
static short black_bits[ ] = {
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF
};

Panel_item      objects;
Textsw          textsw;
Panel           panel;
Icon            icon;

/*
 * main()
 *    Create a panel and panel items. The application uses panel items
 *    to choose a particular object and change its foreground and
 *    background colors in an interactive manner. Create a textsw.
 *    Create an icon. All the objects share the same colormap segment.
 */
main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame         frame;
    Panel_item    color_choices, panel_fg_bg;
    Cms           cms;
    int           i;
    Server_image  chip, icon_image;
    void          color_notify();
    extern void   exit();
    static Xv_singlecolor cms_colors[ ] = {
        { 255, 255, 255 }, /* white  */
        { 255, 0, 0 },     /* red    */
        { 0, 255, 0 },     /* green  */
        { 0, 0, 255 },     /* blue   */
        { 255, 255, 0 },   /* yellow */
        { 188, 143, 143 }, /* brown  */
        { 220, 220, 220 }, /* gray   */
        { 0, 0, 0 },       /* black  */
```

Example 21-3. The color_objs.c program (continued)

```
    };

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,    argv[0],
        NULL);

    cms = (Cms)xv_create(NULL, CMS,
        CMS_NAME,           "palette",
        CMS_CONTROL_CMS,    TRUE,
        CMS_TYPE,           XV_STATIC_CMS,
        CMS_SIZE,           CMS_CONTROL_COLORS + NUM_COLORS,
        CMS_COLORS,         cms_colors,
        NULL);

    /* Create panel and set the colormap segment on the panel */
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,       PANEL_VERTICAL,
        WIN_CMS,            cms,
        NULL);

    /* Create panel items */
    objects = (Panel_item)xv_create(panel, PANEL_TOGGLE,
        PANEL_LABEL_STRING,     "Objects",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_CHOICE_STRINGS,   "Textsw", "Textsw View",
                                "Panel", "Icon", NULL,
        NULL);

    panel_fg_bg = (Panel_item)xv_create(panel, PANEL_CHECK_BOX,
        PANEL_LABEL_STRING,     "Fg/Bg",
        PANEL_CHOOSE_ONE,       TRUE,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_CHOICE_STRINGS,   "Background", "Foreground", NULL,
        NULL);

    chip = (Server_image)xv_create(XV_NULL, SERVER_IMAGE,
        XV_WIDTH,           16,
        XV_HEIGHT,          16,
        SERVER_IMAGE_DEPTH, 1,
        SERVER_IMAGE_BITS,  black_bits,
        NULL);
    color_choices = (Panel_item)xv_create(panel, PANEL_CHOICE,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_LABEL_STRING,     "Colors",
        PANEL_CLIENT_DATA,      panel_fg_bg,
        XV_X,                   (int)xv_get(panel_fg_bg, XV_X),
        PANEL_NEXT_ROW,         15,
        PANEL_CHOICE_IMAGES,
            chip, chip, chip, chip, chip, chip, chip, chip, NULL,
        PANEL_CHOICE_COLOR,     0, CMS_CONTROL_COLORS + 0,
        PANEL_CHOICE_COLOR,     1, CMS_CONTROL_COLORS + 1,
        PANEL_CHOICE_COLOR,     2, CMS_CONTROL_COLORS + 2,
        PANEL_CHOICE_COLOR,     3, CMS_CONTROL_COLORS + 3,
        PANEL_CHOICE_COLOR,     4, CMS_CONTROL_COLORS + 4,
        PANEL_CHOICE_COLOR,     5, CMS_CONTROL_COLORS + 5,
        PANEL_CHOICE_COLOR,     6, CMS_CONTROL_COLORS + 6,
        PANEL_CHOICE_COLOR,     7, CMS_CONTROL_COLORS + 7,
        PANEL_NOTIFY_PROC,      color_notify,
        NULL);

    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void)window_fit_height(panel);

    /* create textsw and set the colormap segment for it */
    textsw = (Textsw)xv_create(frame, TEXTSW,
        WIN_CMS,              cms,
        WIN_BELOW,            panel,
        WIN_ROWS,             15,
        WIN_COLUMNS,          80,
        TEXTSW_FILE_CONTENTS, "/etc/motd",
        WIN_BACKGROUND_COLOR, CMS_CONTROL_COLORS + 0,
        NULL);

    /* adjust panel dimensions */
    (void)xv_set(panel, WIN_WIDTH, xv_get(textsw, WIN_WIDTH), NULL);

    icon_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,             64,
        XV_HEIGHT,            64,
        SERVER_IMAGE_DEPTH,   1,
        SERVER_IMAGE_BITS,    icon_bits,
        NULL);
    /* associate icon with the base frame */
    icon = (Icon)xv_create(XV_NULL, ICON,
        ICON_IMAGE,           icon_image,
        WIN_CMS,              cms,
        WIN_BACKGROUND_COLOR, CMS_CONTROL_COLORS + 0,
        NULL);
    xv_set(frame, FRAME_ICON, icon, NULL);

    window_fit(frame);

    xv_main_loop(frame);
}

/*
 * This routine gets called when a color selection is made.
 * Set the foreground or background on the currently selected object.
 * WIN_FOREGROUND_COLOR & WIN_BACKGROUND_COLOR allow the application
 * to specify indices into the colormap segment as the foreground
 * and background values.
 */
void
color_notify(panel_item, choice, event)
Panel_item      panel_item;
int             choice;
Event           *event;
```

Example 21-3. The color_objs.c program (continued)

```
{
    int cnt;
    Xv_opaque object, get_object();
    unsigned objs = (unsigned)xv_get(objects, PANEL_VALUE);
    int fg = (int)xv_get(xv_get(panel_item, PANEL_CLIENT_DATA),
                     PANEL_VALUE);

    /* the value of the objects panel item is a bit mask ... "on" bits
     * mean that the choice is selected.  Get the object associated
     * with the choice and set it's color.  "&" tests bits in a mask.
     */
    for (cnt = 0; objs; cnt++, objs >>= 1)
        if (objs & 1)
            if ((object = get_object(cnt)) != panel)
                xv_set(object,
                    fg? WIN_FOREGROUND_COLOR : WIN_BACKGROUND_COLOR,
                    CMS_CONTROL_COLORS + choice, NULL);
            else if (fg)
                PANEL_EACH_ITEM(panel, panel_item)
                    xv_set(panel_item,
                        PANEL_ITEM_COLOR, CMS_CONTROL_COLORS + choice,
                        NULL);
                PANEL_END_EACH
            else
                notice_prompt(panel, NULL,
                    NOTICE_FOCUS_XY, event_x(event), event_y(event),
                    NOTICE_MESSAGE_STRINGS,
                        "You can't set the color of a panel.", NULL,
                    NOTICE_BUTTON_YES, "Ok",
                    NULL);
}

/*
 *    Return the XView handle to nth object.
 */
Xv_opaque
get_object(n)
int n;
{
    switch (n) {
        case SELECT_TEXTSW:
            return textsw;
        case SELECT_TEXTSW_VIEW:
            return xv_get(textsw, OPENWIN_NTH_VIEW, 0);
        case SELECT_PANEL:
            return panel;
        case SELECT_ICON:
            return icon;
        default:
            return textsw;
    }
}
```

21.10 Cms Package Summary

Table 21-1 lists the attributes for the `CMS` package. This information is described fully in the *XView Reference Manual*.

Table 21-1. Cms Attributes

CMS_BACKGROUND_PIXEL	CMS_NAME
CMS_COLOR_COUNT	CMS_NAMED_COLOR
CMS_COLORS	CMS_PIXEL
CMS_CONTROL_CMS	CMS_SCREEN
CMS_FOREGROUND_PIXEL	CMS_SIZE
CMS_INDEX	CMS_TYPE
CMS_INDEX_TABLE	CMS_X_COLORS
XV_DEPTH	XV_VISUAL
XV_VISUAL_CLASS	

22 Internationalization

In this chapter, I discuss the features in XView that allow applications to be internationalized. For additional information on internationalization for XView Version 3.2, refer to the *XView 3.2 Developer's Notes*, Part II "Internationalization," distributed by Sun Microsystems, Inc. It describes the internationalization modifications by package as well as XView's wide character support. Internationalized applications do not have to be redesigned or recompiled to run in a different language or *locale*. The goal is to make a single object version of an application support any number of languages. The attribute `XV_USE_LOCALE` enables the internationalization features. All internationalized applications must set this attribute to `TRUE`.

The basic idea in internationalization is to separate language-specific data from the rest of the application code. Adapting the application to any supported language, or *localization*, then requires modification of only the language-specific data. The task of internationalization includes translation of application-specific strings and modification of layout files. All other aspects of the user interface are handled by XView.

The internationalization features discussed in this chapter are:

Locale Setting
: Before running an internationalized application, a user must choose what language to run in. This is done by the locale setting mechanism of XView.

Localized Text Handling
: Developers need to be able to write application strings (error messages, menu labels, etc.) in their native language and have those strings retrieved in the language specified by the locale. This process is described in Section 22.2, "Localized Text Handling."

Object Layout and Customization
: When an application is run in a non-native language, the dimensions of its strings may change (among other things). This may cause the dimensions of objects with text strings in them, such as buttons and panels, to change. Object Layout is the mechanism by which the screen location of objects is modified to accommodate these kinds of changes. Section 22.3, "Object Layout and Customization," describes how an application can be customized to accommodate various object layouts.

22.1 Locale Setting

This section describes the following internationalization topics:

- Locale definition.
- How to enable internationalization features in XView.
- How to switch the locale from the OpenWindows Localization Properties Sheet.
- How the locale is set in XView using locale attributes, command-line options, XView locale resources, and the ANSI-C/POSIX `setlocale()` function.
- Locale implementation limits and restrictions.

22.1.1 Locale Definition

In an internationalized program, the *locale* specifies the language and cultural conventions used in a program. Locale setting affects the display and manipulation of language-dependent features.

XView defines a program's locale and the categories within each locale. These categories refer to the language-dependent features defined by ANSI-C and OPEN LOOK. These features include the display characteristics of monetary, numeric, and time fields.

XView offers several methods of setting locale. In order of priority, locale is set as follows:

1. XView locale attributes.
2. Locale command-line options.
3. XView locale resources (using the OpenWindows Workspace Property Sheet or an entry in the X resource database).

22.1.2 Enabling Internationalization

The attribute `XV_USE_LOCALE` enables the internationalization features in XView. For internationalized applications to work properly, this attribute must be set to `TRUE`. This attribute is only valid for `xv_init()`. The following is an example:

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
            XV_USE_LOCALE, TRUE,
        NULL);
```

22.1.2.1 Setting path of locale-specific files

The attribute `XV_LOCALE_DIR` specifies the location of an application's locale-specific files. The directory structure referenced by `XV_LOCALE_DIR` is:

```
<XV_LOCALE_DIR>/locale/LC_MESSAGES
<XV_LOCALE_DIR>/locale/app-defaults
<XV_LOCALE_DIR>/locale/app-defaults/<application name>
```

where *locale* is expanded differently depending on which internationalized feature is using `XV_LOCALE_DIR`. For a discussion of the specific use of `XV_LOCALE_DIR`, refer to Section 22.2, "Localized Text Handling," and to Section 22.3, "Object Layout and Customization."

22.1.3 OpenWindows Localization Properties Sheet

The OpenWindows Localization Properties Sheet, shown in Figure 22-1, provides the OPEN LOOK user interface for setting locale. To set locale, bring up the OpenWindows Workspace Property sheet and choose "Localization" from the menu. The system then allows you to specify the following categories within the Localization Menu:

Basic Setting
: Specifies the country of the user interface. Once the Basic Setting is specified, the user can set the specific settings. The specific settings define properties for each locale.

Display Language
: Specifies the language in which labels, messages, menu items, help text, and so on, are shown.

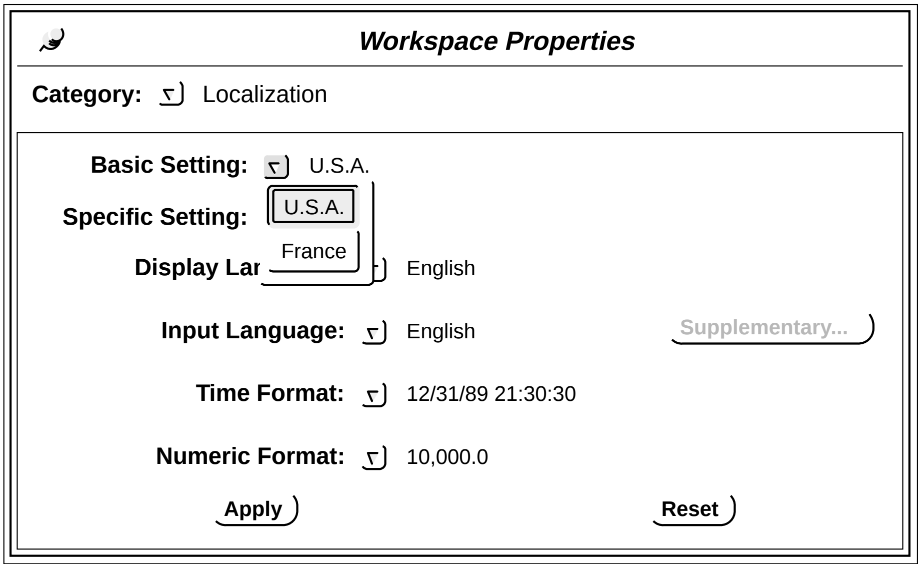
Input Language
: Specifies the language used for input.

Time Format
: Specifies the format for time and date displays.

Numeric Format
: Defines the numeric format.

Supplementary Settings
: Provides a selection of additional settings for any of the Specific Settings. That is, if additional selections are required to specify Display Language, Input Language, Time or Numeric Format, pressing the Supplementary Settings button will display them.

The Localization Property Sheet sets locale via XView locale resources in the X resource manager database. This process is described later in this chapter. Once a setting is made in the Localization Properties Sheet, subsequent applications will adopt the new environmental settings.


Workspace Properties

Category: Localization

Basic Setting: U.S.A.

Specific Setting:

Display Lar English

Input Language: English

Time Format: 12/31/89 21:30:30

Numeric Format: 10,000.0

Figure 22-1. OPEN LOOK localization menu

22.1.4 XView Locale Attributes

Several attributes let you specify the locale from within an application. Each of these locale attributes corresponds to the locale parameters described for the OpenWindows property sheet. The attributes are shown in Table 22-1.

Table 22-1. XView Locale Attributes

Locale Parameter	XView Attribute
Basic Setting	`XV_LC_BASIC_LOCALE`
Display Language	`XV_LC_DISPLAY_LANG`
Input Language	`XV_LC_INPUT_LANG`
Numeric Format	`XV_LC_NUMERIC`
Time Format	`XV_LC_TIME_FORMAT`

These attributes specify the different OPEN LOOK locale categories. They can only be set at creation time through `xv_init()`, but can be queried at any time with `xv_get()` on any object that is a subclass of `WINDOW` or `SERVER`. The value that is set using the above attributes must be a valid locale name in the system.*

Locale attributes are used in situations when an application developer needs to force a program to operate in a specific locale. For example, if a program only works in the French locale, then the locale attributes can be used to hardwire the locale to French, preventing

*The valid locale names are dependent upon the operating system used.

anyone from switching the locale using X resources or command-line options. Each of these attributes takes a `char *` argument. Once again, these attributes are only valid for `xv_init()` and `xv_get()`. For example, set the basic locale to French with the following code:

```
xv_init(  XV_USE_LOCALE,  TRUE,
     XV_LC_BASIC_LOCALE,  "fr",
     XV_INIT_ARGC_PTR_ARGV,    &argc, argv,
     NULL);
```

22.1.5 Command-line Options for Specifying Locale

The locale command-line options allow the locale to be set when an application is started. These options may be used to override the system environment for a specific program. The command-line options are shown in Table 22-2, and in Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual*.

Table 22-2. Locale Command-line Options

Locale Parameter	Command-line Option
Basic Setting	`–lc_basiclocale`
Display Language	`–lc_displaylang`
Input Language	`–lc_inputlang`
Numeric Format	`–lc_numeric`
Time Format	`–lc_timeformat`

An example of these command-line options follows:

```
% textedit –lc_basiclocale fr –lc_displaylang C –lc_inputlang fr
```

This command line brings up a text editor with messages and panel labels in English. The user's input would be in French.

22.1.5.1 XView locale resources

Locale can also be set through resources written in the X11 Resource Manager database (*~/.Xdefaults*). The workspace property sheet uses the XView Locale Resources to set locale parameters. Resources are shown in Table 22-3.

Table 22-3. XView Locale Resources

Locale Parameter	Resource Name
Basic Setting	`basicLocale`
Display Language	`displayLang`
Input Language	`inputLang`
Numeric Format	`numeric`
Time Format	`timeFormat`

The example below shows locale resource entries in the X resource database. This example shows the resources set to the value of "C." This is the default locale on most operating systems. As far as character input/output is concerned, "C" is synonymous with 7-bit ASCII.

```
*basicLocale:     C
*displayLang:     C
*inputLang:       C
*numeric:         C
*timeFormat:      C
```

This example shows the resource set to the value "C". This value specifies the English language (American).

22.1.5.2 ANSI-C/POSIX

Internally, XView uses several standard ANSI-C/POSIX functions. The `setlocale()` is an internal function that should **not** be used in application programs. If locale attributes, command-line options, or resources are not specified, XView uses environment variables to set locale information. This method follows the ANSI-C/POSIX specification for locale announcement. The environmental variables are the same as the POSIX locale categories. POSIX categories are shown in Table 22-4.

Table 22-4. POSIX Categories

OPEN LOOK Category	POSIX Category
Basic Setting	`LC_CTYPE/LANG*`
Display Language	`LC_MESSAGES`
Input Language	`n/a`
Numeric Format	`LC_NUMERIC`
Time Format	`LC_TIME`
n/a	`LC_MONETARY`
n/a	`LC_COLLATE`

*`LANG` (Basic Setting above) is not one of the POSIX categories, but is a convenience environment variable used to set the default for other locales. Basic Locale sets `LC_CTYPE`, `LC_MONETARY`, and `LC_COLLATE`. Input Language will be the same as `LC_CTYPE`.

22.1.6 Limits and Restrictions

The Basic Locale setting determines the character set used by XView. The other locale categories can differ from the basic setting, but they cannot require a different character set from the Basic Locale. For example, setting the Basic Locale to “fr” (French) and the Display Language to “ja” (Japanese) will not work because the French and Japanese locales require different character sets.

The following restrictions thus apply:

1. If basic locale setting is the “C” locale, then all other locale categories must be in the “C” locale.
2. If the Basic Locale is set to a locale other than the “C” locale, then all other locale categories must be set either to a locale that uses the same character set as the basic setting, or to the “C” locale.

22.2 Localized Text Handling

When creating an internationalized application, developers need to write text strings in their native language (error messages, menu labels, button labels) and have those strings appear in foreign languages based on the locale established in the environment. Note that the term “foreign language” refers to non-native languages. XView allows you to define any language as the native language and any other language as the foreign language.

Localized text handling allows a developer to write text strings in the native language and have those strings retrieved in a foreign language. To implement localized text handling in an XView program, the developer must perform the following tasks:

1. For all displayable text strings, use the string as an argument in the `gettext()` or `dgettext()` functions. These functions accept native language strings as arguments, and return the equivalent foreign language strings.
2. Extract the native language text strings used with `gettext()` and `dgettext()` from the source and store them in a portable message file. This extraction can be done by hand, or through the source filter `xgettext`.
3. Provide the foreign language equivalent for each text string, and put it in the portable message file. The portable message file can be edited using any plain text editor which supports both the native and foreign language.
4. Run the `msgfmt` program on the portable message file to create a *text domain*. Here is an explanation on domains. The complete list of messages that exists in a system can be divided in two different ways: Those that belong to a particular language can be grouped together in a *locale*.

   Those that logically belong to a “function” (for lack of a better word), “package,” “library,” etc., can be grouped together in a “text domain.” Messages are grouped in domains for convenience and maintenance. A domain contains messages from its “group” in all the valid languages of the system. Likewise, a locale contains all the messages in a specific language (this covers all domains of the system). Example of domains are, “panel_labels,” “library_error_messages,” “menu_labels,” and so on.

So, to obtain the correct translation of a message string, one needs to know what language it is to be displayed in (locale - French, Japanese, etc), and which logical group it belongs to (domain - error messages, button labels, etc.) The text domain is in binary file format and is not interchangeable between CPU architectures. This file cannot be edited using an editor; it must be created using `msgfmt`.

XView follows the UniForum Messaging Proposal (1003.1b) specification of *gettext, dgettext, text domain,* and *portable source file format for messages* (PSFFM).

22.2.1 Localized Text Handling – Application Programmer Interface

Several functions support localized text handling. They are described in this section.

22.2.1.1 gettext()

The procedure gettext( ) retrieves the foreign language string for the passed string. The format of gettext( ) is:

```
char *
gettext(msg_id)
    char   *msg_id;
```

where “msg_id” is the native language string for which the foreign language equivalent is required. The locale used for retrieval is the current setting of the Display Language category (`LC_MESSAGES`). The domain used is the “current domain,” or the domain set by the last call to “textdomain( ).” If a call to textdomain( ) has not been made, the default domain is used.

If gettext( ) cannot find the foreign language equivalent for “msg_id,” then “msg_id” itself is used.

The example below searches the current text domain for the foreign language string that is identified by the key “wrongbutton.” The foreign language string is then assigned to “message:”

```
message = gettext("wrongbutton");
```

22.2.1.2 dgettext()

Another function, `dgettext()`, allows you to specify the text domain from which to retrieve the foreign language string without changing the current text domain. Its format is:

```
char *
dgettext (domain_name, string)
   char *domain_name;
   char *string;
```

Where *domain_name* is the text domain containing the desired foreign language equivalent, and *string* is the native language string for which the foreign language equivalent is desired.

The following example searches for the foreign language equivalent `wrongbutton` in the text domain `library_error_strings`, then assigns the result to `message`. The current text domain, which was active before this call, is not changed.

```
message = dgettext("library_error_strings",
      "wrongbutton");
```

22.2.1.3 textdomain()

Use `textdomain()` to specify the current text domain. It has the following format:

```
char *
textdomain(domain_name)
   char *domain_name;
```

Where *domain_name* is the desired current text domain.

The following example sets the current text domain to `library_error_strings`. All subsequent `gettext()` calls automatically search `library_error_strings` for the foreign language equivalent.

```
textdomain("library_error_strings");
```

The value of the current text domain can be queried as follows:

```
current_domain = textdomain(NULL);
```

The domain may be set to the implementation-defined default domain by calling `textdomain()` with a domain name set to an empty string. Such a call to `textdomain()` will also return the implementation-defined default domain. (Refer to the man page on `textdomain` for specific information on the default domain for your operating system.) For example:

```
default_domain = textdomain("");
```

Multiple text domains can be used within the same application. This allows different types of strings to be grouped into separate text domains. For example, one text domain might contain error messages from the XView library, a second text domain might contain XView menu and button text, and a third might contain strings from the application.

22.2.1.4 bindtextdomain()

The current Uniforum messaging proposal does not specify a method for identifying the location of text domains in the file system. The function `bindtextdomain()` performs this function. The function defines the location of the text domain files to the library. The format of the `bindtextdomain()` call is:

```
char *
bindtextdomain(domain_name, binding)
   char *domain_name;
   char *binding;
```

The following example sets the binding for the text domain `library_error_strings` to */home/myapp/lib/locale*:

```
bindtextdomain("library_error_strings", "/home/myapp/lib/locale");
```

All subsequent gettext() and dgettext() calls for the text domain library_error_strings will look in:

```
/home/myapp/lib/locale/<Display Language Setting>/LC_MESSAGES
```

for the message file library_error_strings.mo.

The `bindtextdomain()` function also has a default domain binding. The default domain binding is applied to any text domains that haven't been bound explicitly by a `bindtextdomain()` call. Unlike the other bindings, the default binding is actually a list of directories that are searched in order. There is always at least one directory in the list: */usr/lib/locale*.

Additional directories can be added to the list as follows:

```
bindtextdomain("", "/home/myapp/lib/locale");
```

Directories are searched in the order in which they are added to the list. In the above example, */usr/lib/locale* would be searched first followed by */home/myapp/lib/locale*.

An opaque snapshot of the state of the default binding list can be taken with the following:

```
(char *) state = bindtextdomain("", NULL);
```

Note that this is similar in principle to using `setlocale()`:

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

The state can be restored at a later time using the following:

```
bindtextdomain("", state);
```

22.2.1.5 Examples

The following three examples retrieve the same strings, but have different effects on the text domain. The first example does not change the current text domain. The second and third examples change the current text domain to `library_error_strings`, then retrieve the foreign language string of `wrongbutton`.

```
message = dgettext("library_error_strings",
          "wrongbutton");

textdomain("library_error_strings");
message = gettext("wrongbutton");

textdomain("library_error_strings");
message = dgettext("", "wrongbutton"); \\* "" = current domain *\\
```

22.2.1.6 XV_LOCALE_DIR

If `XV_LOCALE_DIR` is set in `xv_init()`, XView calls `bindtextdomain()` on the application's behalf to prepend the pathname specified by `XV_LOCALE_DIR` to the default domain path list. All unbound text domains would be located in the directory with the general format:

```
<XV_LOCALE_DIR>/<Display Language Setting>/LC_MESSAGES/domain
```

"domain" is the name of the text domains. The default value of `XV_LOCALE_DIR` is *$OPENWINHOME/lib/locale*.
An example, using `XV_LOCALE_DIR` follows:

```
xv_init(XV_INIT_ARGS, argc, argv,
        XV_LOCALE_DIR, "/app_home/lib/locale",
        NULL);
```

22.2.2 Creating a Text Domain

Once the program is written using `gettext()` and `dgettext()`, text domains are created using the following procedures:

1. For example, you could run `xgettext` on the following source file fragment (see the man page for `xgettext` for proper usage):

```
message1 = dgettext("library_error_string", "Save");
message2 = dgettext("library_error_string", "File");
message3 = dgettext("library_error_string", "Edit");
```

This filter extracts the strings from the source files, and places them in a portable message file. The format of the portable message file produced is shown below:

```
domain "library_error_strings
msgid "File"
msgstr
msgid "Save"
msgstr
msgid "Edit"
msgstr
```

The `xgettext` filter currently does not expand `#define` commands. If you have `#define` strings which need to be extracted, then you need to run your application through the C preprocessor as follows:

```
cc -P file.c
```

This runs the source file through the C preprocessor only, and outputs to a file with a suffix, `.i`. You should then run this `.i` file through `xgettext` to extract all strings:

```
xgettext xgettext options file.i
```

This will produce a file called *library_error_string.po.*

2. Add the foreign language equivalents for all the native language strings in the portable message file between quotes next to `msgstr`. Use a plain text editor that supports the character set/encoding for the language.

   ```
   domain "library_error_strings
   msgid "File"
   msgstr "<french for File>"
   msgid "Save"
   msgstr "<french for Save>"
   msgid "Edit"
   msgstr "<french for Edit>"
   ```

   If you wish to create multiple text domains, you may separate the related text strings into separate files. Be sure that the first line of every file consists of a domain line at the beginning of each file. You may also have multiple text domains within the same file. To do this, simply start each new domain with the line:

   ```
   domain "domain_name"
   ```

   where *domain_name* is the name of the text domain. Note that the message identifiers and strings of a particular domain must be in the same file under the same domain name.

3. Run the `msgfmt` program on each portable message file to produce the text domain file:

   ```
   msgfmt library_error_string.po
   ```

   This produces the text domain file *library_error_string.mo*, which contains the messages compiled in an internal format. The text domain file is in binary format, and is not interchangeable between CPU architectures. It cannot be edited using an editor, and must be created using `msgfmt`.

4. Move each text domain to the directory under the path specified as follows:

   ```
   <XV_LOCALE_DIR>/locale/LC_MESSAGES
   ```

As mentioned previously, `gettext()` and `dgettext()` search the text domain for the native language text or *key* value, then extract the corresponding foreign language text. The key is a unique name that identifies the string to be retrieved. The key does not have to be the native language string, but the native language string is a good choice since it tends to be self-documenting.

22.2.3 New and Enhanced XView Attributes for gettext()

`NOTICE_MESSAGE_STRING` specifies text to print in a notice. `NOTICE_MESSAGE_STRING` is necessary when using `gettext()` to get a notice message. The value of this attribute is a single `NULL`-terminated string, which may contain "\n" as a line break. `NOTICE_MESSAGE_STRINGS` specifies the text to print in a notice. The value is `NULL`-terminated list of strings, which may contain "\n" as a line break. `NOTICE_MESSAGE_STRINGS_ARRAY_PTR` specifies the text to print in a notice. The value is a variable pointing to a `NULL`-terminated array of strings, which may contain "\n" as a line break.

This example shows how notices are displayed in XView Version 2.

```
result = notice_prompt( panel , NULL,
     NOTICE_BUTTON,           "Update changes" ,    101,
     NOTICE_BUTTON,           "Ignore changes" ,    102,
     NOTICE_BUTTON,           "Destroy everything" ,103,
     NOTICE_MESSAGE_STRINGS,  "Press button,"
                             "then go home" ,
     NULL,
     NULL);
```

This example uses `NOTICE_MESSAGE_STRING` and makes the same notice window as the previous example.

```
result = notice_prompt( panel , NULL,
     NOTICE_BUTTON,           "Update changes", 101,
     NOTICE_BUTTON,           "Ignore changes", 102,
     NOTICE_BUTTON,           "Destroy everything", 103,
     NOTICE_MESSAGE_STRING,   "Press button,\nthen go home" ,
     NULL);
```

This example shows how to embed `gettext()`.

```
result = notice_prompt( panel , NULL,
    NOTICE_BUTTON,   gettext("Update changes" ),    101,
    NOTICE_BUTTON,   gettext("Ignore changes" ),    102,
    NOTICE_BUTTON,   gettext(" Destroy everything" ),    103,
    NOTICE_MESSAGE_STRING, gettext(" Press button,\n then go home" ),
    NULL);
```

22.3 Object Layout and Customization

The layout of screen objects (buttons, panel items, etc.) needs to change when the dimensions of the strings within those objects change. For example, a button string in French might be much longer than the same button string in English, thus requiring other buttons to be repositioned. Another example of object layout changes are objects that have to be repositioned to accommodate locale-specific representations such as date or time. A mechanism is required to alter the layout during localization.

22.3.1 Implicit and Explicit

Object layout of an XView application may be implicit or explicit. Using explicit layout, the application specifies *x* and *y* coordinates for the objects in the application. Implicit layout relies on certain defaults that are built into the toolkit. Thus, if a series of panel items is created, and no layout information is explicitly specified, the panel package will position the items based on default spacing rules. For example, one rule is to start a new row when a panel item extends past the edge of a panel.

Implicit positioning may be all that is required to localize an application. When the locale is switched, the new strings are picked up and the panel package automatically adjusts the layout of the panel items based on the dimensions of the new strings.

Other scenarios may exist, however, where explicit layout is required. One situation where implicit positioning does not provide a satisfactory layout is when switching locales. Another scenario requiring explicit repositioning is when accommodating changes in format when locales are switched. For example, the date representation in the United States is mm/dd/yy, while in India the date is represented as dd/mm/yy. Reliance on such explicit formats requires modification during localization.

The attributes, `XV_USE_DB` and `XV_INSTANCE_NAME`, allow developers to specify customizable attributes. These attributes can be used to specify explicit object layout.

22.3.2 Layout and Customization API

22.3.2.1 XV_LOCALE_DIR

Each application may have one or more locale-specific application defaults files. These files provide layout information and any other data stored under the X Resource Manager that is specific to a given locale.

The directory structure for the locale-specific defaults files used by the layout scheme is:

```
<XV_LOCALE_DIR>/locale/app-defaults/<appname>.db (read first)
<XV_LOCALE_DIR>/locale/app-defaults/<appname>/*.db
```

where *locale* is the current setting of Basic Locale.

Typically an application would have its locale-specific application defaults file (one file) in:

`<XV_LOCALE_DIR>/`*locale*`/app-defaults/<appname>.db`

If the application wants to group its application defaults into multiple files, they can be placed in the directory:

`<XV_LOCALE_DIR>/`*locale*`/app-defaults/<appname>/*.db`

22.3.2.2 XV_USE_DB

`XV_USE_DB` can be used to specify a set of attributes that are to be searched in the X11 Resource Manager database. `XV_USE_DB` takes a `NULL`-terminated list of attribute value pairs as its values. During program execution, each attribute in this `NULL`-terminated list is looked up in the X11 Resource Manager database. If the attribute is not found in the database, then the value specified in the attribute-value pair is used as the default value.

22.3.2.3 XV_INSTANCE_NAME

The attribute `XV_INSTANCE_NAME` is used to associate an instance name with an XView object. The instance name is used to construct the key value (resource name) used by the resource manager to perform the lookup. The key value is constructed by concatenating the instance names of all objects in the current object's lineage, starting with the name of the application or whatever was passed in with the `-name` command-line option. The XView attribute name remains in lowercase.

An example using `XV_INSTANCE_NAME` is shown below. Assume that the application name is "app." The key value is constructed by concatenating "app," "frame1," and "panel1." Thus, resource files containing entries such as the following would be specified as in Example 22-1:

```
app.frame1.panel1.xv_x: 20
app.frame1.panel1.xv_y: 20
```

Example 22-1. Using XV_INSTANCE_NAME

```
(void)xv_init(XV_USE_LOCALE, TRUE,
          XV_INIT_ARGC_PTR_ARGV, &argc, argv,
          ...
          NULL);

frame = (Frame)xv_create(0, FRAME,
          XV_INSTANCE_NAME, "frame1" ,
          NULL);


panel = (Panel)xv_create(frame, PANEL,
          XV_INSTANCE_NAME, "panel1" ,
          XV_USE_DB,
          XV_X, 10,
          XV_Y, 15,
          NULL,
          NULL);
```

If entries corresponding to the attributes XV_X and XV_Y were not specified in the resource file, the default values specified in the attribute-value pair will be used (XV_X will default to 10 and XV_Y will default to 15).

For more detailed information on the X resource manager, refer to Volume One, *Xlib Programming Manual.*

The list of attributes that are customizable is shown below:

```
CANVAS_HEIGHT                          CANVAS_MIN_PAINT_HEIGHT
CANVAS_MIN_PAINT_WIDTH                 CANVAS_WIDTH

PANEL_CHOICE_NCOLS                     PANEL_CHOICE_NROWS
PANEL_DROP_HEIGHT                      PANEL_DROP_WIDTH
PANEL_EXTRA_PAINT_HEIGHT               PANEL_EXTRA_PAINT_WIDTH
PANEL_GAUGE_WIDTH                      PANEL_ITEM_X
PANEL_ITEM_X_GAP                       PANEL_ITEM_Y
PANEL_ITEM_Y_GAP                       PANEL_JUMP_DELTA
PANEL_LABEL_WIDTH                      PANEL_LABEL_X
PANEL_LABEL_Y                          PANEL_LIST_DISPLAY_ROWS
PANEL_LIST_ROW_HEIGHT                  PANEL_LIST_WIDTH
PANEL_MAX_VALUE                        PANEL_MIN_VALUE
PANEL_NEXT_COL                         PANEL_NEXT_ROW
PANEL_SLIDER_WIDTH                     PANEL_TICKS
PANEL_VALUE_DISPLAY_LENGTH             PANEL_VALUE_DISPLAY_WIDTH
PANEL_VALUE_STORED_LENGTH              PANEL_VALUE_X
PANEL_VALUE_Y

WIN_COLUMNS                            WIN_DESIRED_HEIGHT
WIN_DESIRED_WIDTH                      WIN_ROWS

XV_HEIGHT                              XV_WIDTH
XV_X                                   XV_Y
```

22.3.3 Command-line Options

The internationalization command-line options are supported for use with the customizable attributes. The list of command-line options is available in Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual*. The -name command-line option can be used to set the application instance name. For example, if you run:

```
mytool -name big
```

all the resource names of customizable attributes within mytool will use:

```
big.<rest of name>
```

for resource database lookup.

22.4 Internationalization Attribute Summary

The attributes that support internationalization are shown in Table 22-5. This information is described fully in the *XView Reference Manual*.

Table 22-5. Internationalization Attributes

XV_INSTANCE_NAME
XV_LC_BASIC_LOCALE
XV_LC_DISPLAY_LANG
XV_LC_INPUT_LANG
XV_LC_NUMERIC
XV_LC_TIME_FORMAT
XV_LOCALE_DIR
XV_LOCALE_DIR
XV_USE_DB
XV_USE_LOCALE

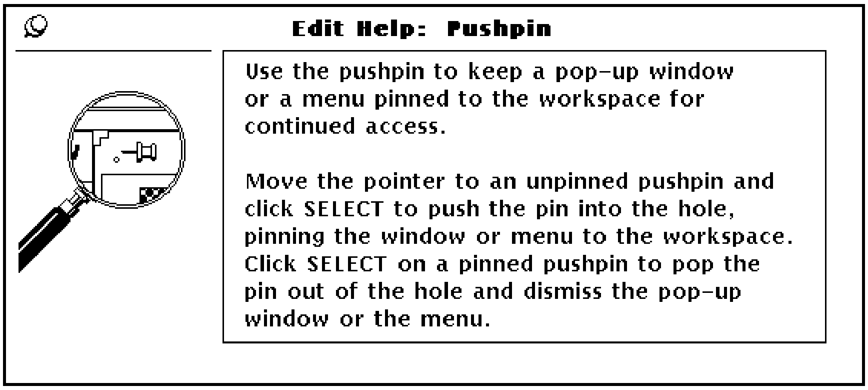
This page intentionally left blank

to preserve original page counts.

23
Help Facilities

This chapter addresses the general-usage Help facilities that apply to all XView packages. The help mechanism is available for the application programmer to register help information in response to user requests.

OPEN LOOK describes a help mechanism that enables the user to get help from anywhere in the user interface. The user requests help using the Help key on the keyboard, or, if one does not exist, the `F1` function key. When this key is used, the window under the pointer receives the `ACTION_HELP` event. In such cases, a *Help window* is displayed as shown in Figure 23-1.



Edit Help: Pushpin

Use the pushpin to keep a pop-up window
or a menu pinned to the workspace for
continued access.

Move the pointer to an unpinned pushpin and
click SELECT to push the pin into the hole,
pinning the window or menu to the workspace.
Click SELECT on a pinned pushpin to pop the
pin out of the hole and dismiss the pop-up
window or the menu.

Figure 23-1. A sample Help window

Note that when you request help on a menu or a menu item, this action will dismiss the menu. This dismissal is required, since the Menu package must relinquish its pointer and keyboard grabs in order for the help request to be processed.

23.1 Using XV_HELP_DATA

The attribute XV_HELP_DATA is used to provide XView with information about where to find the text for the help frame. The attribute's value is a string in the form of:

```
"filename:token"
```

The *filename* is found from a list of directories set in the $HELPPATH environment variable. The actual filename has the *.info* suffix appended to its base name. Thus, the complete path to the filename that contains the help information is:

```
$HELPPATH/filename.info
```

If $HELPPATH is not set, the directory */usr/lib/help* is used. However, if $HELPPATH *is* set, then that path is searched exclusively and /usr/lib/help is *not* searched unless it is explicitly listed in the variable's setting. Therefore, users should always have the default path in their environment setting. Since $HELPPATH may be set to any number of directories, a possible setting would be (for the C shell):

```
setenv HELPPATH /usr/lib/help:/path/to/other/help/directories:.
```

Or, for the Bourne shell:

```
HELPPATH=/usr/lib/help:/path/to/other/help/directories:.
export HELPPATH
```

The trailing "." indicates that the current directory is included.

Once the filename is found, the *token* is searched for within the file. The file must contain the string ":*token*" at the beginning of the line. All the text following the token, starting on the next line, is displayed in the scrolling text subwindow within the help frame. A maximum of 10 lines of text, each terminated by a newline, is visible in the help text subwindow. If the help text exceeds 10 lines, a scrollbar is provided. Each line of help text may not exceed 50 characters. Scrollbars do not appear when a line is too long vertically to fit in the help window. You need to manually type a return in the help text to trim help text lines to fit within the 50-character limit. The text ends when a line is found that starts with a "#" or a ":".

23.2 HELP Key Binding

Servers that know about the XK_HELP keysym because they use a key labeled *Help*, as on Type-4 Sun keyboards, provide help messages when the HELP key is selected. On servers that don't know about XK_HELP, the user cannot get help unless some key is mapped to XK_HELP (OPEN LOOK requires that *some* key be mapped to provide help messages). The user, or system administrator, may define a help key by mapping the resource OpenWindows.KeyboardCommand.Help to a key such as F1.

OPEN LOOK also describes a method to access additional on-line information when a "More" button is placed in the help window. XView allows the More button to be selected with the mouse, or used with an accelerator (see Table 23-1). Selecting the More button allows the application to provide additional help according to the current help context.

Help for a text subwindow is also provided. Table 23-1 shows the default bindings for the various types of help functions provided by OPEN LOOK.

Table 23-1. Modified Help Keystrokes

Key	Type of Help
<Help>	Spot Help
Shift-Help	More Spot Help (an accelerator)
Ctrl-Help	Help on Text
Shift-Ctrl-Help	More Help on Text (an accelerator)

23.2.1 Attaching Help Data

Let's say you have written a program called `my_program`. In this program, there is a panel with a button labeled "Save." The user wants to know exactly what this button will do if selected. The user can select the Help key over the panel item in order to obtain help associated with the item. To support this action, you must attach help data to the panel items in the following manner:

```
 . . .
extern Frame frame;
extern void save_it();
Panel panel;

panel = (Panel)xv_create(frame, PANEL, NULL);

xv_create(panel, PANEL_BUTTON,
    XV_HELP_DATA,        "my_program:save",
    PANEL_LABEL_STRING,  "Save",
    PANEL_NOTIFY_PROC,   save_it,
    NULL);
xv_create(panel, PANEL_TEXT,
    XV_HELP_DATA,        "my_program:target",
    PANEL_LABEL_STRING,  "Filename: ",
    NULL);
 . . .
```

The value for `XV_HELP_DATA` contains the name of the program as the "filename" to search for and the token is `:save`. The filename was chosen based on the name of the application for easier administration of the help files. The next step is to create the file *my_program.info* and add the help text. The file might look like this:

```
:save
When selecting this panel item, all changes made
to the current document will be saved in the target
file.  To change the target file, type in a new
name at the "Filename:" text item.
#

:target
This filename represents the name of the file to use
to load a new file or to store editing changes.  Full
pathnames should be specified to insure that you get
the correct filename.
#
```

The “#” used to terminate help text makes the file more readable. Otherwise, it could be omitted, in which case the text would terminate at the next "*:token*" or the end of the file—whichever comes first.

23.2.2 More Help

The Spot Help window displays a More Help button when “More Help” is available. Alternatively, when Shift-Help is depressed, More Help is requested on the graphical object under the pointer, but the Spot Help window is not displayed. The help mechanism provides More Help when a field is added to the *.info* file, and More help is requested. The More Help field specifies a shell command that is executed to provide additional help information.

```
:spot_help_token:more_help_shell_command
```

The `more_help_shell_command` includes everything from the character after the second colon (:), to the character before the newline (\n). The More Help field should contain a string that is used as an argument for an invocation with the `system` library routine. This executes a command that provides help messages, or executes a help viewer (More Help). For example, an entry in a `.info` file containing a token and description for the topic `back`, and providing more help with the `helpopen` command, contains two fields as well as the Spot Help text.

```
:Back:$OPENWINHOME/bin/helpopen  handbooks/workspace.handbook

Choose Back to send the selected window to the
back of the window stack.
#
```

Prior to executing the system command, if the `-display` switch was specified on the command line, the help mechanism calls `putenv()` on the `DISPLAY` environment variable. This ensures that the help viewer, for example, `helpopen` above, is invoked on the display on which the More Help request was issued.

23.2.3 Text Help

Help information for a selected text string is presented using XView's text help feature and the HELP_STRING_FILENAME attribute. Text help allows XView to search for a string of text and to display help information that is stored in a help file associated with the string of text.

The Ctrl-HELP sequence produces an ACTION_TEXT_HELP event which begins the text help search. The window under the pointer will get an ACTION_HELP event and XView then requests the contents of the primary selection, a text string. XView opens the file associated with the window's HELP_STRING_FILENAME attribute and searches for a matching text string. If a match is found, the text's associated filename and target, which are stored in a second field in the HELP_STRING_FILENAME file, provide Spot Help, and/or More Help. If any condition is not met, then a notice is displayed.

The sequence Shift-Ctrl-Help produces the semantic action ACTION_MORE_TEXT_HELP for More help on a text string.

The HELP_STRING_FILENAME attribute contains the name of a file with a list of string pairs. The file is searched for in the directories listed in the environment variable HELPPATH. Each line in the file contains two words: the first word, which must be less than 128 characters, is the string on which help is available. The second word, which must be less than 64 characters, is of the form "file:target," which XView uses to find the Spot Help text.

HELP_STRING_FILENAME is set on the paint window, or on any of its owners where the help strings are painted.

23.2.4 Displaying Help Manually

If the window corresponds to an XView package that handles its own events, such as Panels or Menus, then XView displays the help frame automatically. All you need to provide is the data for the help message by using the attribute XV_HELP_DATA.

If you are handling events (such as a canvas) in the window, then you are responsible for displaying the help frame as well as providing the help data. The function to accomplish this is xv_help_show(). It takes the following form:

```
xv_help_show(window, help_data, event)
    Xv_Window       window;
    char           *help_data; /* "file:key" */
    Event          *event;
```

The window parameter is an XView window associated with an XView object. The help_data parameter is a string identical to the value of XV_HELP_DATA discussed in the previous section. The event parameter represents the event that caused the need for the help window to be displayed. This event structure may be modified upon return, so it should not be referenced after use.

An event handler for a canvas would have to track help-key events, display the help frame, and provide the text to display in the frame. The following code fragments show how this might be done.

```
...
canvas = xv_create(frame, CANVAS,
    ...
    WIN_CONSUME_EVENTS, ..., ACTION_HELP, ..., NULL,
    WIN_EVENT_PROC, my_event_handler,
    NULL);
...

my_event_handler(window, event)
Xv_Window window;
Event *event;
{
    if (event_action(event) == ACTION_HELP) {
        xv_help_show(window, "canvas:help_info", event);
        return;
    }
    ...
}
```

The meaning of "`canvas:help_info`" is the same as the help data described earlier.

23.2.5 Help File Installation

Once the help file has been written, you should install it in */usr/lib/help* on your system. If you don't, then the user must set the `$HELPPATH` environment variable correctly to point to the path where the file actually resides. Otherwise, the user's request for help will result in a notice that help being posted cannot be found. Further, the file and the path to the file (including directories and links) must be readable and searchable.

If circumstances prevent you from installing the help file in the designated area, it is not reasonable to expect the user to know where the help file is. That is, do not expect that the user has set the `$HELPPATH` variable correctly. You should set the environment for the user. The path must be set to include at least two pathnames: */usr/lib/help* and the path to your help file. Both are needed because the user might request help from XView objects that provide their own help; these help files reside in the default directory and may be needed.

The following code fragment shows how `$HELPPATH` should be initialized to locate help files that do not reside in */usr/lib/help*.

```
#include <stdio.h> /* for BUFSIZ */

#define HELPPATHNAME "/help/directory" /* set this yourself */

main(argc, argv)
int argc;
char *argv[ ];
{
    extern char *getenv();
    char *helppath, buf[BUFSIZ];
```

```
        xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

        /* ... */
        sprintf(buf, "HELPPATH=/usr/lib/help:%s:", HELPPATHNAME);
        if (helppath = getenv("HELPPATH"))
            strcat(buf, helppath);
        putenv(buf);

        /* ... */

        xv_main_loop(...);
    }
```

Notice that we are setting the value of `$HELPPATH` regardless of its previous value. We don't know if the previous value of `$HELPPATH` existed; if it did, we don't know if the necessary pathnames were already in it. It is not worth the effort writing code to parse the string to see if the path exists. Even if it does, we want to be sure it is at the beginning of the variable. This ensures that the path is searched first. Since the same path can be set more than once, no harm is done by prefixing the variable with the desired pathnames.

Further, the use of `putenv()` requires that the `char *` buffer passed be valid throughout the life of the variable (e.g., until it is unset or reset later). In most cases, we would have to use `malloc()` and pass in new memory or use a static variable. However, since this `putenv()` is called from main, the variable space used by `buf` will not be corrupted until `main()` exits. So we don't bother with allocating memory.

Lastly, it should be mentioned that the effects of `putenv()` are temporary; the new path setting affects this process and application only. Each process has its own version of `HELPPATH`, so setting it explicitly for our application does not affect other programs.

23.2.5.1 HELPPATH usage with internationalization

The help mechanism checks the value of the attribute `XV_USE_LOCALE`. If this attribute is set to `TRUE`, the help files are searched for in locale specific directories, according to the setting of the attribute `XV_LC_DISPLAY_LANG`. If `XV_USE_LOCALE` is `TRUE`, `XV_LC_DISPLAY_LANG` is "C," and HELPATH is set to $APPHOME, then the application searches in the specified directory, as well as in the language specific directory for its *.info* files. In this case, the search path includes two directories:

```
$APPHOME/C/help
$APPHOME
```

23.2.5.2 Setting the application name

The attribute `XV_APP_NAME` sets the string to be used by XView as the application's name. (Currently this is only for the help package). The application name can be localized using `gettext` around the string that is set with `XV_APP_NAME`. XView will use the localized string set with `XV_APP_NAME` in the header of spot help windows to indicate which application generated the help window.

This attribute allows for more than one help file while still displaying the same name in the spot help header window.

23.3 Help Package Summary

Table 23-2 lists the attributes and procedures for the `HELP` package. This information is described fully in the *XView Reference Manual*.

Table 23-2. Help Attributes and Procedures

Attributes	Procedures
`XV_HELP_DATA`	`xv_help_show()`
`HELP_STRING_FILENAME`	
`XV_APP_NAME`	

24
Error Recovery

This chapter addresses the general-usage Error Handling facilities that apply to all XView packages. All XView packages, including extensions to existing packages or those packages you write yourself, should address error handling to provide adequate feedback for the programmer.

There are two different types of errors addressed: XView errors and Xlib protocol errors. The two are very different from one another; XView errors are generated by misusing the XView Toolkit in some way, while Xlib protocol errors result from calling Xlib functions incorrectly or using incorrect values to those functions. We'll start with XView errors since, as you may already know, it is easy to generate run-time errors when writing XView applications.

24.1 XView Errors

XView errors are caused by specifying invalid or unknown attributes, objects, or values. Often, XView errors are generated by simply failing to terminate a list of values of attribute-value pairs. Whatever the reason, for an error, the XView internals call the function `xv_error()`.

This function is the highest level interface into the XView error package. Like many of the other XView functions, `xv_error()` takes a `NULL`-terminated attribute-value list of parameters. These parameters, set by the calling function, describe the nature of the error encountered. `xv_error()` then calls either a programmer-supplied function or a default function that prints the nature of the error to `stderr`. Before discussing the details of these routines, let's first address the problem of when you need to use them and to what extent.

XView error handling is most useful during the early development of applications. If you want to force a core dump for debugging purposes or if you want to suppress the standard error messages from being displayed on the controlling tty's `stderr`, then all you need to do is install a general error handling routine. If you are writing your own XView packages or wish to get detailed information about the nature of an XView error, see Section 24.4, "Advanced Error Handling."

First, we'll address the specific task of how to install an error handler for an XView application, then we'll move on to more advanced error handling methods.

24.2 Simple Error Handling

An error handler is installed using the attribute, XV_ERROR_PROC in the call to `xv_init()`.

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
    XV_ERROR_PROC,   my_error_proc,
    NULL);
```

Because `xv_init()` is called only once, the error routine cannot be changed or uninstalled. The routine must be installed at this time, and it remains installed for the life of the application.

The XView error handler is called whenever `xv_error()` is called. That is, when there is an internal error in XView, not if there is an X Protocol error or errors of any other kind.

The following call generates an XView error:

```
xv_set(NULL, PANEL_LABEL, "foo", NULL);
```

The reason for the error is that an invalid XView object was given to `xv_set()`. This may seem like a simplistic example because no one would ever intentionally pass NULL as an object. However, attempting to set attributes for uninitialized objects is not an uncommon error. A more common mistake made by the novice XView programmer is forgetting to use a NULL terminator at the end of the attribute-value list to `xv_set()` or `xv_create()`. This is also a more subtle error that may not always generate an error due to the undefined value of the missing parameter.*

When `xv_error()` is called, it in turn calls the registered error handling routine, which takes the following form:

```
error_proc(object, avlist)
    Xv_object       object;
    Attr_attribute  avlist[ATTR_STANDARD_SIZE];
```

Since XView errors are always generated in response to an internal XView package or routine, there is always an object associated with the error. This is passed as the `object` parameter. The `avlist` provides details about the nature of the error. Note that this is *not* an attribute-value list in the form of the other attribute-value lists used throughout most of the book. It is an array of attributes and values that have been constructed from a NULL-terminated attribute-value list. See Chapter 25, *XView Internals*, for more details.

*A fundamental understanding of variable-argument lists shows that a missing parameter (the NULL parameter in this case) translates into an unpredictable value, which by many coincidences may turn out to be 0.

The following code fragment shows a sample error procedure that supports this interface.

```
error_proc(object, avlist)
Xv_object         object;
Attr_avlist       avlist;
{
    char buf[32];

    printf("%s\nDump core? ", xv_error_format(object, avlist));
    fflush(stdout);
    if (gets(buf) && (buf[0] == 'y' || buf[0] == 'Y'))
        abort(); /* may return if application is trapping SIGIOT */
    return XV_OK;
}
```

24.2.0.1 Using xv_error_format()

The above procedure makes use of the routine `xv_error_format()`. This function returns a pointer to a static `char *` describing the XView error that has occurred. It takes as parameters an XView object and an `Attr_avlist`. Because `xv_error_format()` returns a pointer to a static string, it should be copied into your own buffer if you wish to retain the value since repeated calls overwrite the contents.

Since the parameters to `xv_error_format()` are the same as those passed to the error function, they may be passed on to the format function without further processing. This function is useful if you don't want to parse the `Attr_avlist` yourself, but still wish to print the error message.

This is all very simple if you do not care to put a great deal of work into your error handling routine and parse the `avlist`. Note that we aren't even testing to see if the severity of the error was recoverable or not. If an error occurs, no matter how innocent, we want to trace it down to the offending function call.

24.3 X Error Handling

Catching errors that occur from Xlib or the X server should be done using the methods described in Volume One, *Xlib Programmer's Manual*. This section shows how you can write your own routine that handles Xlib errors as well as errors with the server.

To register an Xlib error handler, you set an XV_X_ERROR_PROC or you may use `XSetErrorHandler()`. Using `XSetErrorHandler()` allows you to register one error handling routine per application. `XSetErrorHandler()` overrides any existing error procedures and makes it necessary to deal with all X errors (by default XView catches some X error events). Using the attribute XV_X_ERROR_PROC in the `xv_init()` call allows XView to continue to filter out some X error events (some of these error events may be not fatal errors caused by the toolkit).

```
int x_error_proc();

xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
    XV_X_ERROR_PROC,  x_error_proc,
    NULL);
```

If an Xlib error occurs, your routine is called with the following parameters:

```
    int
    x_error_proc(dpy, event)
        Display      *dpy;
        XErrorEvent  *event;
```

The routine should return XV_OK if you have handled the error to your satisfaction. In this case, XView will ignore the error and continue on with the execution of the program. If the routine returns XV_ERROR, then XView calls its default error handler, prints an error message to stderr, and exits with a status of 1.

24.4 Advanced Error Handling

If you wish to examine in detail an XView error within your error handling procedure or are implementing your own XView package, the following sections may be useful. Each section addresses the error attributes and their associated values. Using the information described here along with Section 25.2, "Internal Attribute-Value Lists," in Chapter 25, *XView Internals*, you can construct a call to xv_error() and parse an error list passed to your error handling function.

24.4.0.1 Error types

The header file <*xview/xv_error.h*> contains the definitions of the types and functions used by the error package. These error types are used as parameters to xv_error() and from there to your error handling routine. The following list of attributes should be used if you are going to be calling xv_error() or if you wish to parse the Attr_avlist from the error handling routine.

ERROR_BAD_ATTR
: An attribute was specified that is not defined by XView. If the calling function forgets to terminate a list with a NULL, this is most likely the error value passed.

ERROR_BAD_VALUE
: A bad value was provided for an attribute. This includes out of range values, and so on. If you think the value given is correct, check its type. Passing floats when in fact they are read as doubles may cause this problem.

ERROR_CANNOT_GET
: xv_get() was used on an ungettable attribute.

ERROR_CANNOT_SET
: xv_set() was used on an unsettable attribute.

ERROR_CREATE_ONLY
: xv_set() was used to set an attribute that is only valid using xv_create().

ERROR_INVALID_OBJECT
: The `object` parameter to the routine is invalid. Either the object was uninitialized or the object had been (or is in the process of being) destroyed.

ERROR_LAYER
: The layer of software that detected the error. Possible error layers are the operating system, the X server, the XView Toolkit, and the application.

ERROR_PKG
: The toolkit package that detected the error.

ERROR_SERVER_ERROR
: The error detected by the server; takes an `XErrorEvent *` as a value.

ERROR_SEVERITY
: The severity of the error detected. Its value is of type `Error_severity`. This is an enumerated type whose values may be ERROR_RECOVERABLE or ERROR_NON_RECOVERABLE. Unrecoverable errors should definitely cause the program to exit, whereas recoverable errors can cause an exit if you so desire.

ERROR_STRING
: Used by the calling function to `xv_error()` to give a description of the error if necessary. Trailing newlines are stripped.

24.4.1 Calling xv_error()

If you are trying to write your own XView package or add an extension to an existing package, you may need to call `xv_error()`. Calling it is similar to calling `xv_set()`; the first parameter is a handle to an XView object followed by a NULL-terminated attribute-value list consisting of the above attributes. It takes the form:

```
int
xv_error(object, attrs)
    Xv_object object;
```

`object` is the object for which the offending call had taken place. If the programmer called `xv_set()` on a `frame` object and the attributes passed were invalid, then you would call `xv_error()`, passing the `frame` as the object and a set of ERROR_ attribute-value pairs from the above list.

For example, the following call assumes the calling function tried to set an invalid attribute in the FRAME package. (Note that this is only an example, since the FRAME package does not do this.)

```
switch (attribute) {
    case FRAME_FOREGROUND :
        /* ... */
        break;
    case FRAME_OLD_RECT :
        xv_error(frame,
            ERROR_STRING,     "You cannot set this attribute.",
            ERROR_CANNOT_SET, attribute,
            ERROR_PKG,        FRAME,
            NULL);
        break;
    /* ... */
}
```

As you can see, unless you are implementing the internals of an XView package, xv_error() may be of limited use. However, this example demonstrates how xv_error() can be called internally by XView.

24.4.1.1 Error severity

In the above example, the reason to call xv_error() is not a serious one, at least not one that should terminate the program. By default, calling xv_error() is a recoverable error, so in order to specify a non-recoverable error, the call to xv_error() should pass the attribute, ERROR_SEVERITY, and the value ERROR_NON_RECOVERABLE. Unrecoverable errors generally terminate using exit() with a non-zero exit status. However, abort() may be used instead to generate a core image used for debugging.

In any event, if the error handler is called, it should print a warning message and either continue or exit accordingly. If you install your own error routine, the choice is yours. Example 24-1 in the next section, shows how the error severity can be evaluated and acted upon accordingly.

24.4.2 Revisiting the Error Handler

Advanced usage of the ERROR package provides us with the ability to scan the attribute list in search of the causes of the error. With this information, you can print out more useful error messages or display them in a manner other than printing to stderr.

Recall that there are two parameters passed to the function:

```
error_proc(object, avlist)
    Xv_object       object;
    Attr_avlist     avlist;
```

object is the object in which the xv_* call failed to operate. You can get the type of object by calling:

```
Xv_pkg *pkg = (Xv_pkg *)xv_get(object, XV_TYPE);
```

If the error itself pertains to the object, then getting the type of the object may generate another error. You should not attempt to get the package until you test the error code in the Attr_avlist to be sure the error is not with the object parameter. If the error is not due to the object itself, the package returned by xv_get() indicates to which XView package the object belongs. This value matches the same argument as the second parameter to xv_create(). For example, the package returned may be MENU, CANVAS, PANEL_BUTTON, SERVER, etc.

The `Attr_avlist` may be scanned for the ERROR attributes. Example 24-1 shows how this is done. This is a very simplistic example and is for demonstration purposes only. A more complete example may be found in the function `xv_error_format()` in the XView source code.

Example 24-1. Example error parsing function

```
int
my_error_handler(object, avlist)
Xv_object    object;
Attr_avlist  avlist;
{
    Attr_avlist  attrs;
    Error_layer  layer;
    Error_severity severity = ERROR_RECOVERABLE;
    int n = 0;
    char strs[64][7];

    for (attrs = avlist; *attrs && n < 5; attrs = attr_next(attrs)) {
        switch ((int) attrs[0]) {
            case ERROR_BAD_ATTR:
                sprintf(strs[n++], "bad attribute %s",
                    attr_name(attr[1]));
                break;
            case ERROR_BAD_VALUE:
                sprintf(strs[n++],
                    "bad value (0x%x) for attribute %s", attrs[1],
                    attr_name(attrs[2]));
                break;
            case ERROR_INVALID_OBJECT:
                sprintf(strs[n++], "invalid object (%s)",
                    (char *) attrs[1]);
                break;
            case ERROR_STRING:
                sprintf(strs[n++], "%s", (char *) attrs[1]);
                break;
            case ERROR_PKG:
                sprintf(strs[n++], "Package: %s",
                    ((Xv_pkg *)attrs[1])->name);
                break;
            case ERROR_SEVERITY:
                severity = attrs[1];
        }
    }
    strcpy(strs[n++], "Core dump?");
    strs[n] = 0;
    if (notice_prompt(base_frame, (Event *)NULL,
        NOTICE_MESSAGE_STRINGS_ARRAY_PTR, strs,
        NOTICE_BUTTON_YES,              "Yes",
        NOTICE_BUTTON_NO,               "No",
        NULL) == NOTICE_YES)
            abort();
    if (severity == ERROR_NON_RECOVERABLE)
        exit(1);
    return XV_OK;
}
```

This error handling routine sets a set of error messages in an array of buffers. A notice is used to display the messages and prompt the user to generate a core image of the program for debugging. Selecting “Yes” causes `abort()` to be called. The program exits if the severity is non-recoverable, and continues otherwise.

One particular note of interest is the use of the routine `attr_name()`. This is a hypothetical routine that you would have to write to convert the actual enumerated attribute-values into strings that make sense to read. If you write your own XView package with new attributes, you will have to write a routine equivalent to `attr_name()`. The routine should return a static `char *` describing the attribute. If the attribute has no corresponding string (e.g., it is an unknown attribute), it should return a string indicating the integer or hexadecimal value.

This page intentionally left blank

to preserve original page counts.

This page intentionally left blank

to preserve original page counts.

25 XView Internals

This chapter discusses the internal mechanisms that XView uses to implement the existing object classes. The information in this chapter should give you the ability to build your own objects that are "extensible" or are extensions of other existing classes. By writing extensions to XView classes, you can modify their appearance or functionality. By creating new classes, you can create objects that go beyond what the existing XView library provides.

However, you should be forewarned that building XView extensions is not intended to be a solution to every problem. You are strongly encouraged to implement the type of object or enhancements you need using the facilities provided by XView and the existing XView objects. Furthermore, this chapter should be used as an introductory resource. It does not contain enough information to fully explain how all the internal XView objects work, nor does it give you the ability to build an entire library of user interface objects.

If OPEN LOOK compliance is important in your applications, you should be sure that you fully understand the OPEN LOOK specifications before attempting to build new objects or modify existing ones. Because the XView internals do not enforce user interface policy, you could build non-OPEN LOOK-compliant user interface code. However, the existing XView objects were written to conform to OPEN LOOK as much as possible. While you are strongly encouraged to examine the XView source as a model, this chapter only acts as a guide to that model and may not address all issues involved with all XView packages.

In Chapter 2, *The XView Programmer's Model*, we introduced and discussed the hierarchy of XView objects and the use of the basic functions intrinsic to XView: `xv_create()`, `xv_set()`, `xv_get()`, `xv_find()`, and `xv_destroy()`. We will now take a closer look at how that model is utilized by XView internals.

We are going to start with a general discussion of the concepts that XView uses as a framework. The methods described are intrinsic to all XView packages. After that, we examine how attributes and their associated "values" interact with XView and its packages. Once these issues have been addressed, we illustrate how to write your own XView packages and extensions using these concepts.

The Logo package is a simple package that displays an X logo in the middle of a window. The Bitmap package is similar, but it allows the programmer to display an arbitrary bitmap in a window. The Image package is used to demonstrate how to write an extension to an existing XView package. In this case, it is an extension from the server image package found in

Chapter 15, *Nonvisual Objects*. Finally, the Wizzy package shows how to write a panel item extension. Note that the `PANEL` package provides several special attributes that should only be used by panel item extension writers.

25.1 Methods

The *intrinsics* layer of XView is the mechanism that defines and controls the class inheritance model; in other words, it establishes parent/child relationships among XView classes. The XView intrinsics handle the creation, modification, query, and destruction of actual instances of object classes. Each object class contains a *method* (as it is called in object-oriented programming terminology) to respond to any XView-intrinsic request. A method is a function that is written and compiled into the executable program to perform the designated task.

For example, when the programmer calls `xv_create()` to create an instance of a particular class, the XView intrinsics invokes the *initialize method* from that class. For `xv_set()`, it invokes the *set method*, and so on. The XView intrinsics define the methods while the actual classes provide the functions that perform them.

Each *class* is represented programmatically by declaring a data structure consisting of pointers to functions that correspond to each of the methods. When the programmer calls `xv_get()`, the intrinsics de-reference the pointer to the *get* method and call it as a function.

25.1.0.1 Static subclassing

You recall from Chapter 2, *The XView Programmer's Model*, that XView classes are subclassed from one another starting from the *generic* class. This class contains basic information about the object such as its *x,y* position, its geometry specification, its reference count (how many other objects refer to it in some way), what server and display the object is associated with and so on. Most classes share this information and are therefore subclassed from the generic class.

From the generic class, new subclasses are created to describe more specifics about that particular class's appearance, functionality, or other attributes. Subclassing causes each new class to inherit everything from its parent class, so the child class does not need to be redefined or reinitialized. New subclasses define their own methods so that the intrinsics can utilize the parts of new classes that differentiate them from their parent classes. When the programmer calls `xv_create()` to create an instance from a particular class, the intrinsics call the initialize method for each subclass in the hierarchy in succession.

To implement this using the C language, each class is physically defined by a data structure that contains pointers to functions previously written for that class. All this information must be compiled into the XView library (or at least linked with the rest of the object modules at compile time). Because the functions and data structures are pre-written, the subclasses are *static*—they cannot be changed during the execution of the program.* In sum, XView uses *static subclassing*.

25.1.1 Order of Methods

As discussed in Chapter 2, *The XView Programmer's Model*, whenever an object is created, the XView intrinsics initialize each class from top to bottom. Thus, the generic class is first instantiated by calling its initialize method, followed by the next subclass, all the way down until the class of the type requested is instantiated. When completed, an instance of the class has been created with all the default properties of the classes set.

However, the initialization sequence does not stop there. As Figure 25-1 shows, the initialization sequence consists of three phases.

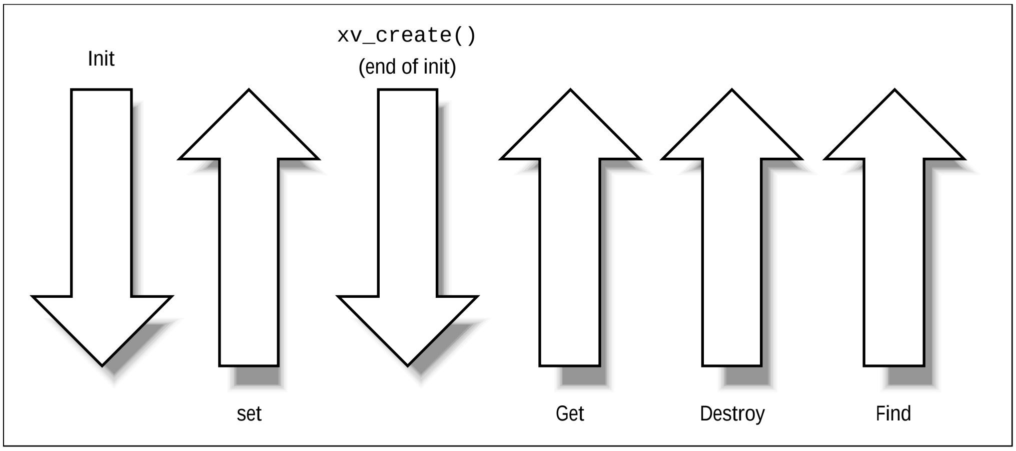


Figure 25-1. Calling order for init, set, get, destroy, and find

After the initialization methods are called for each of the packages, XView calls the *set* methods to handle any attribute-value pairs specified in the programmer's call to `xv_create()`. This is done even if there were no attributes specified. This phase is executed in the *reverse order* of the initialize method, moving from the bottom up.

The final phase of the initialization sequence calls the *set* function again, but in the original order (e.g., from the top down). Here, the *set* method is called with only one attribute, `XV_END_CREATE`. This is the only time that the *set* method is called from the top down. This final phase indicates that the creation phase is over and that the class should resolve any unfinished work. Prior to this point, be careful not to use `xv_set()` on the object being created. During this final phase, you may use `xv_set()` on the object being created. Each phase of this operation is discussed in more detail in later sections.

*Limitations of the C language prevent the ability to do dynamic subclassing.

25.2 Internal Attribute-value Lists

Each of the methods introduced above (with the exception of the destroy method) must deal with attributes and attribute-value lists. Before we begin to discuss the details of how the methods work, the fundamentals of attributes must be understood. This includes the nature of attributes, how their internal values are constructed, the nature of the values *associated* with attributes, package IDs, and so on.

25.2.1 Attribute Values

The semantics of the term *value* can be confusing. There are two *values* that are used when referring to attributes. The type most commonly used is the value *associated* with the attribute. That is, "brown" is the value associated with the attribute `PANEL_ITEM_COLOR`. However, `PANEL_ITEM_COLOR` is declared as an enumerated type that has a "value" just as C variables have values.

The value of an attribute variable contains information about the type of attribute it is, the package it belongs to, how many "value" parameters are associated with it, and the types of those "values." This is all accomplished by setting particular bits within segments of the 32-bit data type, `Attr_attribute`.

The breakdown of the bits in the attribute is shown in Figure 25-2.

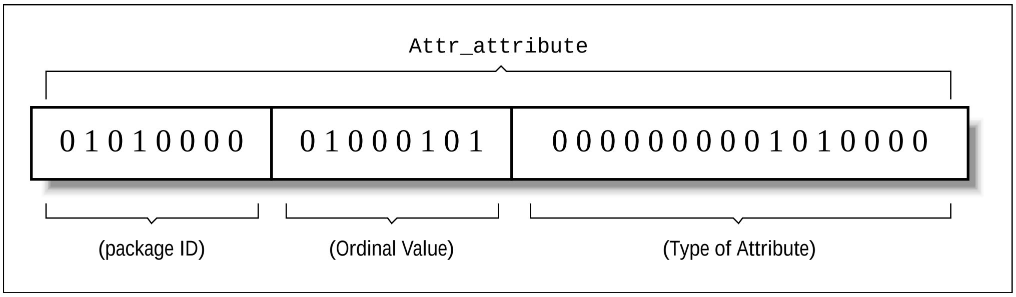


Figure 25-2. The bits in an attribute

All this is done via macros defined in <*xview/attrs.h*> along with a complete listing of all the different types of attributes that may be used.

Let's examine a real attribute, `XV_RECT`. This attribute can be used to set or get the bounding box of an XView object (e.g., x and y, width and height). The attribute is declared in <*xview/generic.h*> as:

```
XV_RECT = XV_ATTR(ATTR_RECT_PTR, 74)
```

The value of XV_RECT expands to a long value set by the macros XV_ATTR and ATTR_RECT_PTR. These are macros whose values are used as masks that can identify the attribute later. The value of 74 is a unique number with respect to the other attributes in the package.

Thus, when the internals of an XView procedure want to determine the value of XV_RECT, they know that the "value" that the programmer specified must be a Rect *. Furthermore, it is known that xv_get() should return the same type. This loose type-checking is by no means enforced. It is used for various reasons, including the ability of the programmer to identify the type of value a particular attribute should take.

The attribute's "type" is also used to determine how many programmer-supplied "values" are associated with it. For example, the attribute PANEL_CHOICE_STRINGS indicates that the value associated with it is a NULL-terminated list of char pointers. We can check this by looking at the value of the attribute in <*xview/panel.h*>:

```
PANEL_CHOICE_STRINGS =
    PANEL_ATTR(ATTR_LIST_INLINE(ATTR_NULL, ATTR_STRING), 22)
```

Here, the use of PANEL_ATTR shows that the attribute is part of the PANEL package. The macro ATTR_LIST_INLINE shows that the value associated with the attribute is a NULL-terminated list. The macro ATTR_STRING shows that it is a list of strings or char pointers.

The list of possible *types* of attributes is in <*xview/attr.h*>.

25.2.2 Creating Attribute Lists

When functions that take attribute-value lists are called, XView internally converts the entire list into an array of Attr_attribute values. The size of the array cannot exceed ATTR_STANDARD_SIZE. This array of attributes and values is assigned the type, Attr_avlist.

The array is created and the actual variable argument list of attributes and values is stored in the array using the function attr_create_list().

Any function whose interface is called with a variable argument list has a corresponding (internal) function that takes the Attr_avlist parameter rather than the variable argument list. This is due to the fact that variable argument lists cannot be passed to subsequent functions reliably. XView overcomes this problem by converting the entire list of attributes and values into the Attr_attribute array and passing it around to functions.

25.2.2.1 Attribute lists within attribute lists

Sometimes, the same attribute-value list is used in multiple calls to xv_create() or xv_set(). Specifying the same list all the time is wasteful since it requires the same processing for each call. In order to optimize and simplify this problem, XView provides an interface for both the XView programmer and the package implementor by introducing the attribute, ATTR_LIST.

The value to the ATTR_LIST attribute is an Attr_avlist, the same type returned by attr_create_list(). The following code fragment shows how this can be used:

```
...
Canvas canvas1, canvas2, canvas3;
Attr_avlist attr_list;

attr_list = attr_create_list(
    WIN_CMS,         cms,
    WIN_EVENT_PROC,  my_event_proc,
    XV_WIDTH,        100,
    NULL);

canvas1 = xv_create(frame, CANVAS,
    ATTR_LIST,            attr_list,
    WIN_BACKGROUND_COLOR, 1,
    NULL);
canvas2 = xv_create(frame, CANVAS,
    ATTR_LIST,            attr_list,
    WIN_BACKGROUND_COLOR, 2,
    NULL);
canvas3 = xv_create(frame, CANVAS,
    ATTR_LIST,            attr_list,
    WIN_BACKGROUND_COLOR, 3,
    NULL);

free(attr_list);
...
```

The only restriction on the use of ATTR_LIST is that it must be the *first* attribute specified in the call to xv_create(), xv_set() or any other xv_* routine that accepts attribute-value lists. Be sure that ATTR_LIST is the first attribute specified when you use it. There are cases when ATTR_LIST may appear to be the first attribute specified, but it is actually not the first attribute. This occurs when you use a macro. For panels, the following are macros: PANEL_CHOICE_STACK, PANEL_CHECK_BOX, and PANEL_TOGGLE. These macros expand to PANEL_CHOICE objects with several associated attributes that will not be visible. These attributes displace ATTR_LIST as the first attribute. These and similar cases should be avoided if you use ATTR_LIST.

Lastly, since attr_create_list() allocates memory, the list should be freed when it is no longer needed.

25.2.3 Interpreting Attributes

When a routine is passed an Attr_avlist, it needs to scan the list looking for attributes of interest. The function may not be interested in all the attributes, so when scanning the list, it needs to skip ahead to successive attributes for evaluation. To facilitate this task, XView provides the macro attr_next() to make scanning Attr_avlist easier. This macro looks at a particular attribute and, by the nature of the attribute itself, knows how many items to scan ahead for the next one. The macro returns a pointer to that next attribute.

The first thing to do when scanning the `Attr_avlist` is to set a pointer to the beginning of the list and then advance forward until you reach the `NULL` attribute indicating the end of the attribute-value pairs.

```
function(param1, param2, avlist)
Xv_opaque    param1;
Xv_opaque    param2;
Attr_avlist  avlist;
{
    Attr_avlist     attrs;
    for (attrs = avlist; attrs[0]; attrs = attr_next(attrs)) {
        switch ((int) attrs[0]) {
            ...
        }
    }
    ...
}
```

The `for()` loop initializes the `attrs` variable to the beginning of the list and tests for the `NULL` attribute. Upon each iteration of the loop, the variable is set to the next attribute in the list. For this to work, `attrs` should never be moved in either direction in the list. To look at the value of a particular index into the attribute list, you should index that position relative to the current value of `attrs`. This is precisely what is done in the `switch()` statement within the loop.

`attr_next()` looks at the "type" of the attribute to determine how many, if any, value parameters are associated with the attribute. For attributes that take *lists* such as `PANEL_CHOICE_STRINGS`, it knows to look ahead for the next `NULL`-valued index in the array. This is why lists may not contain the value `NULL` or 0 as elements in the list. Once a `NULL` or 0 is found, `attr_next()` returns the element following the terminating `NULL`.*

The `switch` looks at index 0 of the `attrs` pointer for the attribute to evaluate. Each `case` in the `switch` statement handles the value that pertains specifically to the package in question. For example, the *set* routine for the `CANVAS` package has the following code fragment:

```
Attr_avlist attr;
for (attr = avlist; attr[0]; attr = attr_next(avlist)) {
    switch ((int) attr[0]) {
        case CANVAS_WIDTH:
            if (canvas->width != (int) attr[1]) {
                width = (int) attr[1];
                new_paint_size = TRUE;
            }
            break;
        case CANVAS_HEIGHT:
            if (canvas->height != (int) attr[1]) {
                height = (int) attr[1];
                new_paint_size = TRUE;
            }
            break;
```

*For portability reasons, `NULL` should always be used rather than 0 to terminate a list.

```
                /* .... */
                default:
                    *status = xv_check_bad_attr(&xv_canvas_pkg, attr);
            }
        }
```

For each attribute, the `case` statement knows what to interpret as the value parameter in the attribute list. The `CANVAS_WIDTH` case sets the `width` variable to the value set in `attr[1]` and assumes it is an `int`.

25.2.4 Checking for Bad Attributes

When the attribute being evaluated in the `switch` statement falls to the default case, there may or may not be something wrong with the attribute. Since the `switch` statement should have had a case for all the known attributes to the package, it is assumed that the attribute that had fallen through probably belongs to another package.

To check for this, the function `xv_check_bad_attr()` is used. The form of the function is:

```
int
xv_check_bad_attr(pkg, attr)
    Xv_pkg         *pkg;
    Attr_attribute  attr;
```

The function checks to see if the attribute in the second parameter belongs to the package specified in the first parameter. If the attribute *does* belong to the package, then an error message is printed and the function returns `XV_OK`. Otherwise, the function does nothing and returns `XV_ERROR`. Yes, this is counter-intuitive, but this value is utilized more appropriately by the *get* method. Details are discussed in Section 25.8.3, "The Bitmap Get Method."

An unknown attribute does not indicate that an error has been made. Remember that packages can be subclassed from other packages, so an attribute may apply to another level of the class hierarchy and will be dealt with at another time by another function.

25.2.4.1 Searching for specific attributes

Rather than scanning the entire `Attr_avlist` looking for one particular attribute, XView provides the convenience function, `attr_find()`. This function takes an `Attr_avlist` and an `Attr_attribute` as parameters and returns the location within the list where the attribute was found.

Here is its implementation:

```
Attr_avlist
attr_find(attrs, attr)
register Attr_avlist attrs;
register Attr_attribute attr;
{
    for (; *attrs; attrs = attr_next(attrs)) {
     if (*attrs == (Xv_opaque) attr)
         break;
    }
    return (attrs);
}
```

25.2.5 Consuming Attributes

Once an attribute has been evaluated, it should be *consumed* so that no other functions may see it. Consuming attributes should not be done if multiple packages (or functions) care to examine the same attribute. The `attr_skip()` macro knows to skip over attributes (and their associated values) that have been consumed.

Attribute consumption is done with the `ATTR_CONSUME()` macro.*

25.3 Customizable Attributes

New attributes that are introduced when you create extensions to XView can be made customizable, via the X resource database, with the function `xv_add_custom_attrs()`. The format of `xv_add_custom_attrs()` is:

```
void
xv_add_custom_attrs(pkg, va_alist)
    Xv_pkg      *pkg;
    va_dcli     va_alist;     /* var args list */
```

The argument *pkg* is the XView package to which the customizable attributes belong. *va_alist* is a `NULL`-terminated list of pairs using the following format:

```
<customizable attribute, attribute resource name>
```

*While the pre-built XView packages *should* consume attributes, few of them actually do. This will change in later releases of XView.

The type of "customizable attribute" is `Attr_attribute`. The type of attribute resource name is `char*`. For example,

```
     xv_add_custom_attrs(pkg,
         <attribute1, attribute1 resource name>,
         <attribute2, attribute2 resource name>,
         <attribute3, attribute3 resource name>,
         <attribute4, attribute4 resource name>,
         ...
         NULL);
```

The attribute resource name is used to construct the key for database lookup when the attribute is used with `XV_USE_DB`.

`xv_add_custom_attrs()` must be called before any of the customizable attributes are used. A good place to call `xv_add_custom_attrs()` would be immediately following `xv_init()`.

For example, you can use `xv_add_custom_attrs()` to make attributes customizable for the package extension called `LOGO` (see Section 25.5, "The Logo Package," in this chapter.) Make the new attributes `LOGO_WIDTH` and `LOGO_HEIGHT` customizable with the following call:

```
     xv_add_custom_attrs(LOGO,
        LOGO_WIDTH, "logo_width",
        LOGO_HEIGHT, "logo_height",
        NULL);
```

The attributes `LOGO_WIDTH` and `LOGO_HEIGHT` can then be customized as in the following example:

```
     logo = xv_create(owner, LOGO,
              XV_USE_DB
                 LOGO_HEIGHT, 300,
                 LOGO_WIDTH, 250,
                 NULL,
              ...
              NULL);
```

The resource names constructed for database lookup for `LOGO_HEIGHT` and `LOGO_WIDTH` will be:

```
   CONCAT_INSTANCE_NAME.logo_height
   CONCAT_INSTANCE_NAME.logo_width
```

where `CONCAT_INSTANCE_NAME` is the concatenation of instance names of all objects in the current object's lineage.

If such entries did exist in the X resource database, then their values will be used for `LOGO_HEIGHT` and `LOGO_WIDTH`. Otherwise, `LOGO_HEIGHT` will default to 300, and `LOGO_WIDTH` to 250.

Currently, support for customizable attributes is provided only for attributes of type `long`, `int`, `boolean`, `char`, and string (`char *`). See Section 22.3, "Object Layout and Customization," in Chapter 22, *Internationalization*, for more details on customizable attributes.

25.4 XView Packages

Earlier, we introduced the concept of XView *methods* and how they are used to define the interaction between a particular class and the XView intrinsics. These methods, along with a set of attributes, macros, types, and functions, collectively make up an XView *package*. The XView library is made up of statically subclassed (pre-built) packages representing user interface objects.

25.4.1 The Xv_pkg Type

Packages (and thus, classes) are declared by creating a global variable of type `Xv_pkg`. This package is defined in <*xview/pkg_public.h*> as:

```
typedef struct _xview_pkg {
    char                *name;
    Attr_attribute       attr_id;
    unsigned             size_of_object;
    struct _xview_pkg   *parent_pkg;
    int                (*init)();
    Xv_opaque          (*set)();
    Xv_opaque          (*get)();
    int                (*destroy)();
    Xv_object          (*find)();
} Xv_pkg;
```

The fields of the `Xv_pkg` type are declared and used as follows:

`name`	The name of the package is a unique, descriptive string. This is useful for debugging the `ERROR` package, and it may also be used in the future to implement resource handling from the resource database. Therefore, it should not contain whitespace or dots (periods). Underscores and hyphens are allowed, but should be avoided for aesthetic reasons. A combination of uppercase and lowercase letters should be used to imply multi-word names (e.g., "`DigitalClock`").
`attr_id`	This is the ID of the package. It is set to a unique number and is used in attributes' values to associate them with the corresponding package. For XView extensions (packages you write), the value should lie between the value for `ATTR_PKG_UNUSED_FIRST` and the value for `ATTR_PKG_UNUSED_LAST`.
`size_of_object`	This is the size of the public part of the object. XView objects are broken down into a public part and a private part. XView is responsible for allocating the public part, while the initialize method is responsible for allocating the private part. The value of the `size_of_object` field is used by the XView intrinsics to know how much space to allocate when creating a new instance of the public part from this class.
`parent_pkg`	This is a pointer to package's parent (the package above the object in the object hierarchy).

`init`	This is the *initialize* method for the package. It is a pointer to a function that returns an error status (`XV_OK` or `XV_ERROR`) depending on whether the initialization process was successful in creating an instance of the object.
`set`	This is the *set* method. This is a pointer to a function that is called when the programmer calls `xv_set()`. The function typically returns an error status (`XV_OK` or `XV_ERROR`) but may return an opaque data type if it chooses.
`get`	This is the *get* method. This is a pointer to a function that is called when the programmer calls `xv_get()`. The function returns the value of the attribute specified in the call to `xv_get()`. A status value may be set indicating an error.
`destroy`	This is the *destroy* method; it is a pointer to a function that is called when the object is destroyed via `xv_destroy()` or when the window manager invokes it in a "save yourself" operation (discussed later). When an object is being destroyed, the function frees any allocated fields of the private data structure. The function returns either `XV_OK` or `XV_ERROR`.
`find`	The *find* method is a pointer to a function that returns a handle to an existing instance of the package specified to `xv_find()`. If no instances of the package with the specified attributes can be found, `NULL` is returned and XView may call the initialize routine depending on the value of `XV_AUTO_CREATE`.

Details about the form of the functions listed above are given later in Section 25.5.2, "The Implementation File."

25.4.2 Public and Private Data

In XView, the object that is made available to the programmer writing XView applications is a public data type defined in the public header file for the package. The only information this type contains is a handle to two data types: the object's parent-data type and the *private* data. The public data type is what `xv_create()` returns to the user. Its nature will become clear in the sample XView package we create later.

The parent data is the public data type of the parent package (the superclass). The private data type is used by the implementation of the XView class. In it, there is a pointer to an object that contains specific information about the object itself. This may include a window, boolean variables, other data structures, and so on. It also contains a pointer back to the public data type.

The *initialize* routine allocates and initializes the fields within the private data type. The initialize routine is also responsible for setting the handles of the public and private data types to one another. Once this double-linking occurs, an instance of the class is complete and the XView intrinsics return a handle to the public type. This is discussed in detail later in Section 25.5.4, "The Initialize Method."

25.5 The Logo Package

This section presents an implementation for a simple package utilizing the concepts introduced so far. This example may help explain some of the more confusing concepts for those still unsure of the material presented.

The example package is called *logo*. All this object does is draw the X logo in the middle of a window. To do this, we require a window, the `Pixmap` containing the X logo, and a `GC` to specify the colors to use when rendering the pixmap. Creating a window is a very complicated task; there's so much to worry about with colormaps, visuals, displays, and screens. Since the XView `WINDOW` package already handles this, the logo class is subclassed from it to take advantage of the `WINDOW` package's capabilities. That package can handle all the window-related details without intervention from the logo package. More generic attributes such as the geometry and position of the object are handled by the `GENERIC` package.

The only thing the logo package needs to concern itself with is providing the data for the bitmap showing the X logo. Once we have that, all we need to do is render it to the window when repaint or `Expose` events take place.

25.5.1 Header Files

Packages usually contain two header files (or *include* files): one that is included by applications that intend to use the package, and another that is included by the source code that implements the package itself.

25.5.1.1 The public header file

The file *logo.h* is the public header file for the logo package.

```
/* logo.h -- public header file for the logo XView class. */
#include <xview/xview.h>
#include <xview/window.h>

extern Xv_pkg logo_pkg;

#define LOGO &logo_pkg

typedef Xv_opaque Logo;

typedef struct {
    Xv_window_struct    parent_data;
    Xv_opaque           private_data;
} Logo_public;
```

Since the logo package is subclassed from the `WINDOW` package, we must include <*xview/window.h*>. We can't include that file unless we include <*xview/xview.h*> first. You'll find that most packages include at least the basic XView header files.

Next, we declare the `logo_pkg` object as an external variable of type `Xv_pkg *`. This is a global variable that we must declare later in the implementation source file where the private data is declared. The `LOGO` definition refers to the address of this global variable. The `Logo` type is a typedef of Xv_opaque. This is basically a convenience type for the benefit of the programmer and follows the style of the other XView packages.

With the #define of LOGO and the declaration of the Logo type, the necessary types are available to make it possible for the programmer to create an instance of the logo object:

```
Logo logo;

logo = xv_create(parent, LOGO, NULL);
```

For the simple logo package, this is all that is necessary to declare in the public header file. There are no attributes specific to the logo package. Had we wanted to provide attributes, their declarations would be here in the public header file. We'll add attributes to the logo package later in the chapter.

The parent parameter in the call to xv_create() for a logo object must be a Frame because the logo is subclassed from the WINDOW package. This is because the FRAME package is the only one that manages subwindow layout.

Finally, the last thing declared in the public header file is the public data type, Logo_public. This is the actual object returned by xv_create(). Since the programmer has no need to reference fields in this data type, an *opaque* data type is sufficient. Therefore, the programmer uses the Logo type.

The Logo_public structure has the two fields discussed earlier: a handle to the parent object and a pointer to the private data. The parent handle is a public data type similar to the logo's public data type. In this case, the public data type for the parent object is Xv_window_struct. We use the real data type for this rather than the *opaque* type, Xv_Window, because XView needs to reference internal fields within it.

The private_data field is a pointer to the actual data structure used by the internals of the logo object. But in the spirit of true object-oriented programming, this type is hidden from the programmer by declaring it to be Xv_opaque.

25.5.1.2 The private header file

The primary purpose of the private header file is to declare the private data structure mentioned above. This file is named with the *_impl.h* suffix implying that it is used by the code that *implements* the logo object methods. Here is the *logo_impl.h* file for the logo package:

```
/* logo_impl.h -- implementation-dependent header file for the
 * logo XView class.
 */
#include "logo.h"

typedef struct {
    Xv_object   public_self;    /* pointer back to self */
    GC          gc;             /* GC to render logo */
    Pixmap      bitmap;         /* xlogo bitmap */
} Logo_private;

#define LOGO_PUBLIC(item)       XV_PUBLIC(item)
#define LOGO_PRIVATE(item)      XV_PRIVATE(Logo_private, Logo_public, item)
```

The public header file is always included in the private header file since it has all the necessary declarations specific to the package and it includes other header files that may be needed.

The private logo structure is declared next. The first field in all private data structures is a handle back to the public data structure. Again, when the *initialize* routine for the package is called, an instance of the private data type is allocated and its `public_self` field is set to the public data type passed. This is shown in Section 25.5.4, "The Initialize Method."

The rest of the fields in the private data structure are those that are specific to the aspects of the logo package that vary from instance to instance. This includes a handle to a `Pixmap` (which is the X logo bitmap) and a `GC`. When multiple instances of the object are created, each instance uses a discrete pixmap and `GC` since each instance of the class may have different window attributes. That is, the programmer may create a logo on a color window and another logo for a monochrome window.

A general rule of thumb is that there should be few, if any, global variables in the implementation of a package. These variables should almost always be declared as fields within the private data type. Therefore, all variables that are needed by the package and that may have different values depending on the instance are declared as fields within the logo's private data structure.

The last two lines of the private header file are:

```
#define LOGO_PUBLIC(item)   XV_PUBLIC(item)
#define LOGO_PRIVATE(item) \
            XV_PRIVATE(Logo_private, Logo_public, item)
```

These macros are used to facilitate the task of cross referencing to and from the public and private data types. Because these types are declared as `Xv_opaque`, typecasting is necessary to coerce the type into a data type needed. They utilize the two XView macros, `XV_PUBLIC` and `XV_PRIVATE`. These macros are defined in *<xview/pkg.h>* as:

```
#define XV_PRIVATE(private_type, public_type, obj) \
    ((private_type *)((public_type *) (obj))->private_data)

#define XV_PUBLIC(obj) ((obj)->public_self)
```

These macros are used frequently in source files that implement an XView package.

25.5.2 The Implementation File

The next task is to declare the logo package and to implement all the methods. This may be done in one or more source files. For maintenance, it is much easier to declare as much as possible in one file and declare all functions as `static`. This is to insure that the functions used by the package are used *only* by the package. However, in the event that more than one file is used to contain all the functions necessary to implement a package, it is impossible to restrict the scope of a function in this manner. XView, therefore, introduces two reserved types that can be used to declare functions for either internal (private) use or public use.

The types `Xv_public` and `Xv_private` are both defined to be `extern` to indicate that they may be called from outside of the files they are declared in. However, the meaning of these types indicates the intended use of the function. Programmers should never call "private" functions, whereas they are allowed to call the public ones. It is assumed that the private functions are those that aid in the implementation of the package. Again, functions should be declared as `static` whenever possible.

25.5.3 The Package Declaration

A package is declared by initializing a global variable of type `Xv_pkg`. All the fields of the data structure are initialized to identify the package. The logo package is declared by creating a `logo_pkg` variable of this type.*

```
Xv_pkg logo_pkg = {
    "Logo",               /* package name */
    ATTR_PKG_UNUSED_FIRST, /* package ID */
    sizeof(Logo_public),   /* size of the public struct */
    WINDOW,               /* subclassed from the WINDOW package */
    logo_init,
    logo_set,
    logo_get,
    logo_destroy,
    NULL                  /* disable the use of xv_find() */
};
```

The package ID is set to `ATTR_PKG_UNUSED_FIRST` because it is assumed that this is the first unused package in the XView library. In short, a package ID only needs to be distinct from the IDs of parent and child packages. While this is the only *requirement* imposed by XView, it is still recommended that all packages have unique package IDs. The value of the package ID must fall within the range `ATTR_PKG_UNUSED_FIRST` through `ATTR_PKG_UNUSED_LAST`.

The `size_of_public` field of the `logo_pkg` is set to the size of the public structure using the C macro `sizeof()`. The XView intrinsics initialize the public structure before calling the initialize method by allocating the number of bytes set by this field.

The parent package from which the logo package is subclassed is set in the `parent_pkg` field. This field is initialized to the `WINDOW` package. You may recall that this is a macro that refers to the address of the `WINDOW` package's global variable: `xv_window_pkg` (implying that the two are interchangeable).

The rest of the fields in the `logo_pkg` structure are initialized to the pointers to the appropriate functions. Notice, however, that the *find* method is disabled by having its field initialized to `NULL`. Any routine that does not apply to a particular package may be set to `NULL`; XView will not try to invoke `NULL` methods.

*This declaration may be done in the file that contains the package implementation. However, for systems with shared libraries, it is advantageous to put this variable declaration and initialization in a file by itself so that the compiler can link it in with the shared libraries. You should consult your compiler and operating system documents for instructions on how to create shared libraries.

The `xv_find()` routine is unset for the logo package because it doesn't make sense to be able to reuse an instance of a logo object. That is, because the logo package is subclassed from the `WINDOW` package, it is impossible to render a window in more than one place on the screen at a time. It is only possible to create multiple instances of this type of object even though the package may display the same logo image.*

This is contrary to the way the font package works, for example. Fonts can be rendered anywhere and they do not contain windows, so referencing the same font instance is reasonable. However, the scope of availability for fonts is restricted to each server. Not all fonts may exist on all servers, and even if they do, they do not share the same `XID`s. Thus, you cannot render a string using a font in a window that resides on a server other than the server from which the font was obtained.

The *find* method is discussed in detail in Section 25.9, "The Find Method," later in this chapter.

25.5.4 The Initialize Method

The *initialize* method is responsible for allocating an instance of the `private_data` structure and linking the private and public structures together. Once the public and private structures have been initialized and linked, an instance of the class has been created. However, it is incomplete because none of the attributes of the class have been set in the new instance. This may be done in the initialize routine and in the set routine, called later.

The function takes the following form:

```
int
init_func(owner, package_public, avlist)
    Xv_opaque       owner;
    Xv_opaque      *package_public;
    Attr_avlist     avlist;
```

The `owner` parameter is the object passed as the owner to the call to `xv_create`. It is declared as `Xv_opaque` here because its actual type varies from package to package. However, the type of the owner is also a public data type. For the logo package, the owner is a `Frame` since frames are used to manage subwindow layout (and the logo package is subclassed from the `WINDOW` package).

The `package_public` parameter is a pointer to the public data type declared in the public header file. For the logo package, this type is `Logo_public`. The XView intrinsics have allocated this data type before calling the routine.

The `avlist` parameter contains the attribute-value pairs specified in the call to `xv_create()`. These may or may not be evaluated from within the initialize routine depending on the nature of the attribute. We'll get to this in a moment.

*It is possible to share the same logo *image*, if not the entire logo object. This can be accomplished using the *Bitmap* package discussed next.

The function returns XV_OK or XV_ERROR depending on whether it was successful allocating and initializing the necessary resources. If there is an error of any kind that should prevent the object from being instantiated, all allocated resources should be freed and the function should return XV_ERROR. If there is an error during any phase of the initialize method, XView calls the *destroy* method for each package (except for the package whose initialize routine actually returned XV_ERROR). The op parameter passed has the value DESTROY_CLEANUP. See Section 25.5.7, “The Destroy Method,” for details.

The following is a listing of the initialize function for the logo package:

```
static int
logo_init(owner, logo_public, avlist)
Xv_opaque       owner;
Logo_public     *logo_public;
Attr_avlist     avlist; /* ignored here */
{
    Logo_private *logo_private = xv_alloc(Logo_private);
    Display *dpy;
    Window win;

    if (!logo_private)
        return XV_ERROR;

    dpy = (Display *)xv_get(owner, XV_DISPLAY);
    win = (Window)xv_get(logo_public, XV_XID);

    /* link the public to the private and vice-versa */
    logo_public->private_data = (Xv_opaque)logo_private;
    logo_private->public_self = (Xv_opaque)logo_public;
    /* create the 1-bit deep pixmap of the X logo */
    if ((logo_private->bitmap = XCreatePixmapFromBitmapData(dpy, win,
        xlogo32_bits, xlogo32_width, xlogo32_height,
        1, 0, 1)) == NULL) {
        free(logo_private);
        return XV_ERROR;
    }
    /* set up event handlers to get resize and repaint events */
    xv_set(logo_public,
        WIN_NOTIFY_SAFE_EVENT_PROC,      logo_redraw,
        WIN_NOTIFY_IMMEDIATE_EVENT_PROC, logo_redraw,
        NULL);

    return XV_OK;
}
```

The first thing this function does is allocate the private data structure for the logo object using the xv_alloc() macro. This macro is defined to be:

```
#define xv_alloc(t) ((t *)xv_calloc((unsigned)1, (unsigned)sizeof(t)))
```

Because xv_calloc() is used, the entire private data structure is allocated and all the fields are initialized to NULL or 0 (thus the analogy to calloc()). The fields of the logo data structure that must be initialized are the bitmap and the GC.

The bitmap is the X logo, a Pixmap created by XCreatePixmapFromBitmapData(). In order to create the pixmap, XCreatePixmapFromBitmapData() requires a pointer to the Display and an X window, both of which can be obtained from the window part of the logo object. Since the initialize phase of XView initializes classes from the generic package down through subclasses, we know the logo's parent (the WINDOW package) has already been initialized and we can use its XID.

While we have initialized the logo's pixmap to use, we cannot initialize the GC for the logo because we do not know the ultimate foreground and background colors of the window. These pixel values are extracted from the window's colormap segment, and although the window for the logo has been created and initialized, its cms has not been assigned yet. This is not done until the window's *set* routine is called. Since the WINDOW package does not evaluate the WIN_CMS attribute in *its* initialize routine, the logo package cannot attempt to read it from the logo package's initialize routine. This must be done later, after the WINDOW package's set routine has had a chance to set the window's cms.

Providing a practical example, consider the following code fragment:

```
cms = xv_create(NULL, CMS,
    CMS_SIZE, 2,
    CMS_NAMED_COLORS, "blue", "red", NULL,
    NULL);

logo = xv_create(frame, LOGO,
    WIN_CMS,  cms,
    NULL);
```

Here, the programmer intended the logo to be rendered in red with a blue background. This is accomplished by creating a colormap segment with two colors: blue and red. During the *initialization* phase of object creation, the WINDOW package creates its window, but only assigns a default colormap segment. It is only during the *set* phase that the window is assigned the colormap segment from the WIN_CMS attribute. Since the set phase is done in reverse order (e.g., the logo's set routine is called before its parent's set routine), the logo package cannot query its window's colors until later. The only opportunity for the logo package to get the window's colors is during the extra call to the set routine in which XV_END_CREATE is passed. The set routine in this case is called in ascending order (top down) and the logo can now query its window's colormap segment.

It is true that if the programmer specifies attribute-value pairs in the call to xv_create(), those pairs are passed to the initialize function in the avlist parameter. However, since the logo package has no attributes of its own for the user to specify, the avlist is ignored in favor of allowing the other packages to deal with attributes. While we could have looked in this attribute list for a WIN_CMS attribute and captured the pixel values from it, a parent package can override or change its mind about which attributes it actually decides to use. Although this may be unlikely, the XView design allows for it to happen and, therefore, XView packages should be written with this possibility in mind.

A general rule of thumb is that packages should not test for attributes from other packages through the avlist. Instead, the preferred method is to use xv_get() to allow the package that is responsible for the attribute to return whatever value it deems appropriate. Note that this is not always true—there are some cases where packages not only look for attributes that don't belong to them, but they override them. The PANEL package, for example, does not allow the programmer to change the foreground and background colors on its window by intercepting or modifying certain color-related attributes. The PANEL package does this in order to prevent the programmer from violating OPEN LOOK.

On the other hand, let's suppose we wanted to restrict the colors used by the logo window to black and white. In this case, we would want to override the WIN_CMS specification if the programmer provided one. We would do this by *consuming* the WIN_CMS attribute (and presumably the WIN_CMS_NAME attribute) from the avlist using the ATTR_CONSUME() macro (see Section 25.2.5, "Consuming Attributes"). The attribute may be consumed here in the initialize routine or later in the set routine. If the WINDOW package chose to consume these attributes, it could have done so before we got to them. However, by consuming them in the logo's initialize routine, we can prevent the WINDOW package from consuming them in its set routine called later. If an attribute that is consumed is just done to prevent another package from evaluating it, chances are that this attribute should not have been set by the programmer. In such a case, a warning should be dispatched via xv_error().

Finally, the last thing done in logo_initialize() is setting the event handlers for the window. This is not intended to track events generated by the user, but to track WIN_REPAINT (Expose) and WIN_RESIZE (ConfigureNotify) events for the logo's window. This is done to determine when and where the logo should be drawn.

The method used to track these events is by using the specified attributes:

```
xv_set(logo_public,
    WIN_NOTIFY_SAFE_EVENT_PROC,      logo_redraw,
    WIN_NOTIFY_IMMEDIATE_EVENT_PROC, logo_redraw,
    NULL);
```

These are *private attributes* (e.g., not for general programmer use) from the WINDOW package specifically for the purpose of having the internals of XView packages be able to specify event handlers that do not conflict with or get overridden by the programmer. The programmer, as you may recall, uses the attribute, WIN_EVENT_PROC to handle events destined for the window. In fact, this attribute will continue to work as expected despite the use of the WIN_NOTIFY_* attributes listed above.

These two private attributes are similar to the Notifier's notify_set_event_func() function. The programmer can interpose on the logo's event function just as described in Chapter 20, *The Notifier*, as usual. In this case, the programmer's interposing functions are called ahead of the logo_redraw() function (by design).

25.5.4.1 The logo_redraw() function

The logo_redraw() function itself does not have anything to do with XView internals or package implementation. However, it is described here so as to keep the continuity of the discussion.

This function simply renders the logo in the object's window. It uses XCopyPlane() to render the logo because the logo Pixmap is known to be one-bit deep whereas the logo's window can be any depth. This is done for the last in a possible series of Expose events as shown:

```
logo_redraw(logo_public, event)
Logo_public     *logo_public;
Event           *event;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);
```

```
        XEvent *xevent = event_xevent(event);

        if (xevent->xany.type == Expose && xevent->xexpose.count == 0) {
            Display *dpy = (Display *)xv_get(logo_public, XV_DISPLAY);
            Window window = (Window)xv_get(logo_public, XV_XID);
            int width = (int)xv_get(logo_public, XV_WIDTH);
            int height = (int)xv_get(logo_public, XV_HEIGHT);
            int x = (width - xlogo32_width)/2;
            int y = (height - xlogo32_height)/2;

            XCopyPlane(dpy, logo_private->bitmap, window, logo_private->gc,
                0, 0, xlogo32_width, xlogo32_height, x, y, 1L);
        } else if (xevent->xany.type == ConfigureNotify)
            XClearArea(xv_get(logo_public, XV_DISPLAY),
                xv_get(logo_public, XV_XID), 0, 0,
                xevent->xconfigure.width, xevent->xconfigure.height, True);
    }
```

The `ConfigureNotify` event is tested to see if the window resized. If it did, the window needs to be cleared and the logo redrawn in the new center of the window. The window is cleared using `XClearArea()` and passing `True` as the last parameter indicating that an `Expose` event should be generated. When the event is delivered, `logo_redraw()` is called again, and the logo is redrawn.

25.5.5 The Set Method

After the initialize routines for all the classes have been called, the *set* method is invoked in reverse order (from the bottom up). That is, the generic package's set routine is called last and the logo's set routine is called first.

The form of the set routine is:

```
Xv_opaque
set_func(pkg_public, avlist)
    Xv_opaque   *pkg_public;
    Attr_avlist  avlist;
```

The first parameter is a handle to the public data type representing the package. The `avlist` parameter is a list of the attributes and values that have not been consumed by the initialize routine or previously called set routines from other packages. In this routine, package-specific attributes are scanned and evaluated, modifying the private data type according to the attributes' values.

Most applications ignore the return value of `xv_set()`, so it is usually sufficient to return `XV_OK`. However, you can return anything you like. For example, your own package may wish to return the previous value of an attribute if `xv_set()` was used to change it. This may not be clearly defined, as `xv_set()` can be called to set many attributes.

Here is the set routine for the logo package:

```
logo_set(logo_public, avlist)
Logo_public *logo_public;
Attr_avlist avlist;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);
    Attr_attribute *attrs;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int) attrs[0]) {
            case XV_END_CREATE : {
                /* this stuff *must* be here rather than in the "init"
                 * routine because the CMS is not loaded into the
                 * window object until the "set" routines are called.
                 */
                Cms cms = xv_get(logo_public, WIN_CMS);
                XGCValues gcvalues;
                Display *dpy =
                    (Display *)xv_get(logo_public, XV_DISPLAY);
                gcvalues.foreground =
                    x(unsigned long)v_get(cms, CMS_FOREGROUND_PIXEL);
                gcvalues.background =
                    (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
                gcvalues.graphics_exposures = False;
                logo_private->gc = XCreateGC(dpy,
                    xv_get(logo_public, XV_XID),
                    GCForeground|GCBackground|GCGraphicsExposures,
                    &gcvalues);
            }
            default :
                xv_check_bad_attr(LOGO, attrs[0]);
                break;
        }

    return XV_OK;
}
```

A handle to the logo's private data structure is needed since the set routine changes the value of fields within that structure. The `LOGO_PRIVATE()` macro is used to get a pointer to the private data from the public object.

Since the logo package has no attributes, there is no need to scan for attributes specific to the logo package or any other package. Recall, however, that we still need to initialize the `GC` for the logo. Therefore, we scan for the `XV_END_CREATE` attribute.

NOTE

The set routine must return `XV_OK` when `XV_END_CREATE` is in the `avlist`. Any other return value causes XView to assume that there was an error in initialization.

At this point in time, the `WINDOW` package has initialized itself completely and we can therefore get the colors from the window's colormap segment. We use `xv_get()` and ask for its `WIN_CMS`.

The example *Bitmap* package goes into more detail about the set routine.

25.5.6 The Get Method

The *get* method is simple: it basically returns the value of the attribute specified in the programmer's call to xv_get(). The form of the get function is:

```
Xv_opaque
get_func(pkg_public, status, attr, avlist)
    pkg_public      *pkg_public;
    int             *status;
    Attr_attribute   attr;
    Attr_avlist      avlist;
```

The attr parameter is the attribute for which the programmer wants the value.* However, there are some attributes used by xv_get() that take an additional parameter. For example, when using CANVAS_NTH_VIEW, an additional int parameter is required to indicate which view to return. For such cases, the additional parameter(s) is in the avlist.

The status parameter must be set by the get routine to XV_ERROR if there is an error.

The calling sequence for the get method is from the bottom up—that is, the specific packages are called first followed by each parent up the chain to the generic object. Each class in the chain is called until one of them sets the status parameter to XV_OK.

If an unknown attribute is requested, status should be set to XV_ERROR, but the function should return XV_OK. This tells XView that the attribute requested does not belong to this package and that it should try the next package in the chain.

Here is the get function for the logo object:

```
logo_get(logo_public, status, attr, args)
Logo_public     *logo_public;
int             *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    *status = xv_check_bad_attr(LOGO, attr);
    return (Xv_opaque)XV_OK;
}
```

Since the logo object has no attributes, it sets the status parameter and returns. As noted earlier, since there are no attributes specific to the logo package, this routine is unnecessary; we could have declared the function pointer as NULL in the Xv_pkg data structure causing the get method for this package to be unused.

A more detailed discussion of the get method, including a discussion on xv_check_bad_attr(), is given in the example *Bitmap* package.

*Remember, xv_get() can only be used to get the value of one attribute.

25.5.7 The Destroy Method

When the programmer calls `xv_destroy()` or if XView decides to destroy objects or classes externally, the *destroy* method for the package is called. The calling sequence for the destroy method is from the bottom up, as it is for the get method.

The main task of the destroy routine is to free the private data type and any other cleanup that may accompany it. This includes freeing allocated data, closing open file descriptors, unlinking temp files, and so on. However, this is not the only reason the destroy method is called; in fact, there are four different reasons or conditions in which the function may be invoked.

The form of the destroy function is:

```
int
destroy_func(pkg_public, status)
    Xv_opaque       *pkg_public;
    Destroy_status   status;
```

As with all the methods, the first parameter is a handle to the public data type for the package. The `status` parameter is of type `Destroy_status`. It describes the condition for which the function has been called. This is the same situation as the `destroy_func()` described in Section 20.9.5, "Modifying a Frame's Destruction."

For the logo object, we need to destroy the allocated pixmap and free the allocated `GC`. The logo's destroy function is:

```
logo_destroy(logo_public, status)
Logo_public     *logo_public;
Destroy_status   status;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);

    if (status == DESTROY_CLEANUP) {
        XFreePixmap(xv_get(logo_public, XV_DISPLAY),
            logo_private->bitmap);
        XFreeGC(xv_get(logo_public, XV_DISPLAY), logo_private->gc);
        free(logo_private);
    }

    return XV_OK;
}
```

Unless the status is `DESTROY_CLEANUP`, nothing is freed. This assures that the instance of the logo object remains intact in case the destroy method was invoked for other reasons. Please consult Chapter 20, *The Notifier*, and Chapter 4, *Frames*, for details on how to handle the other conditions possible for the destroy function.

25.6 Example Program Listing

At this point, we have discussed everything necessary to implement the logo object except for a main application that creates an instance of a logo object.

Writing this application is really no different from the way it is done for any other XView package, as you can see from Example 25-1.

Example 25-1. The logo.c program

```
/* logo.c -- demonstrate the use of the logo package. */
#include <xview/xview.h>
#include <xview/cms.h>
#include "logo.h"

main(argc, argv)
char *argv[ ];
{
    Frame frame;
    Cms cms;
    Logo logo;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    cms = xv_create(NULL, CMS,
        CMS_SIZE,       2,
        CMS_NAMED_COLORS, "blue", "red", NULL,
        NULL);
    logo = xv_create(frame, LOGO,
        XV_WIDTH,       100,
        XV_HEIGHT,      100,
        WIN_CMS,        cms,
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

All the pieces are in play—the `logo` variable is of type `Logo`, the call to `xv_create()` has `LOGO` as the package name, and the owner of the logo is the frame object. The program allocates a colormap segment that has the colors red and blue to demonstrate that colors can be assigned to the logo through the `WINDOW` package attributes. However, if this application is run on a monochrome screen, the output is rendered in black and white, as in Figure 25-3. The frame may be resized by the user and the logo is always redrawn in the middle of its window. This is the task of the `logo_redraw()` routine registered by the logo object in its initialize routine.

The entire logo implementation module is listed in Appendix F, *Example Programs*.

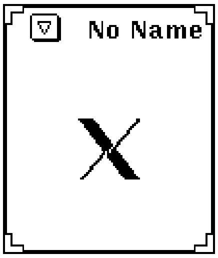


Figure 25-3. Output of logo.c

25.7 Compiling an Implementation File

This is a brief overview of how to compile a file or set of files to implement an XView package. If the package is used only by a particular application that you are writing, you can add the source and object files to your own Makefile or Imakefile just as you would for the sources in your main application. However, if you want to build an XView object and add it to the base XView library for general use, then there are several steps you need to take. You should consult your system manuals for details specific to your system.

First, you should compile your program to generate an object file:

```
cc -c Logo.c
```

You may require additional compilation flags depending on your environment. The include files are presumed to be in the same directory as the source file. If you install them anywhere else, you should change the `#include` directives at the top of the source files to use a different syntax. If you installed the header files in the default XView location (for example, */usr/include/xview*), the `#include` directives should say:

```
#include <xview/logo.h>
```

Anywhere else should have the line:

```
#include <logo.h>
```

If this is the case, your compile line options should include the `-I` parameter.

```
cc -c -I<include_path> Logo.c
```

Once *Logo.c* (the package implementation file) has been compiled, you may compile *logo.c* (the sample application) and link all of them with the default XView library:

```
% cc -c Logo.c
% cc -c logo.c
% cc logo.o Logo.o -lxview -lolgx -lX11 -o logo
```

25.8 The Bitmap Package

The logo package is a very simplistic one since it does virtually nothing but render the X logo in the middle of its window. There are no attributes specific to the logo package to make the package configurable by the programmer. The next example package, the Bitmap package, demonstrates how attributes are defined and used in XView packages.

The Bitmap package is similar to the logo package in that it just displays a bitmap in the middle of a window. However, the Bitmap package provides the programmer with the ability to specify the file containing the bitmap: the create- and set-only attribute, `BITMAP_FILE`. To provide the programmer with the ability to get the pixmap, the attribute `BITMAP_PIXMAP` is available as a get-only attribute.

We cannot get `BITMAP_FILE` because the filename is not retained. Once the bitmap has been loaded, the programmer can get it with the `BITMAP_PIXMAP` attribute. Clearly, the code can easily be modified to support the ability to get the filename or to set the pixmap directly.

The first thing to do is declare these attribute types in the public header file, *bitmap.h*:

```
#include <xview/xview.h>
#include <xview/window.h>

extern Xv_pkg bitmap_pkg;

#define BITMAP &bitmap_pkg

typedef Xv_opaque Bitmap;

#define ATTR_PKG_BITMAP            ATTR_PKG_UNUSED_FIRST
#define BITMAP_ATTR(type, ordinal) ATTR(ATTR_PKG_BITMAP, type, ordinal)

typedef enum {
    BITMAP_FILE     = BITMAP_ATTR(ATTR_STRING, 1),
    BITMAP_PIXMAP   = BITMAP_ATTR(ATTR_OPAQUE, 2), /* get-only */
};

typedef struct {
    Xv_window_struct   parent_data;
    Xv_opaque          private_data;
} Bitmap_public;
```

There are several new aspects to the header file that are used to support the new attributes. First, we define a new macro, `BITMAP_ATTR()`. It is defined as:

```
#define BITMAP_ATTR(type, ordinal) \
    ATTR(ATTR_PKG_BITMAP, type, ordinal)
```

This macro aids in the initialization of the bitmap attribute values by setting the package ID portion of the attribute to be the bitmap package.

Following the macro definitions, the attributes for the bitmap package are defined in the enumerated type definition. `BITMAP_FILE` is declared with `ATTR_STRING` indicating that the programmer-specified value associated with the attribute is a string. The `1` is a unique number to the attributes within the bitmap package, so we start at 1 for the first attribute and increment this number by one for each new attribute.

The private data is declared for the bitmap object in the implementation-specific header file, *bitmap_impl.h*. As you can see, the only changes are new fields used to support the new method of specifying a bitmap filename:

```
#include "bitmap.h"

typedef struct {
    Xv_object   public_self;    /* pointer back to self */
    GC          gc;             /* GC to render logo */
    Pixmap      bitmap;
    int         width, height;  /* ...of pixmap */
} Bitmap_private;

#define BITMAP_PUBLIC(item)  XV_PUBLIC(item)
#define BITMAP_PRIVATE(item)     XV_PRIVATE(Bitmap_private, Bitmap_public, item)
```

The width and height fields are introduced because these are needed to calculate how to center the bitmap on the window. These are fields rather than global variables because their values are unique on a per-instance basis. However, even though these are fields in the private data structure, this does not mean that there *have* to be corresponding attributes.

25.8.1 The Bitmap Initialize Method

The initialization routine does not initialize any fields of the private data since the set routine, which is eventually called, handles the attributes adequately. The basic initialize functionality of allocating the private data and linking the public and private structures together is still done:

```
bitmap_init(owner, bitmap_public, avlist)
Xv_opaque       owner;
Bitmap_public     *bitmap_public;
Attr_avlist     avlist; /* ignored here */
{
    Bitmap_private *bitmap_private = xv_alloc(Bitmap_private);

    if (!bitmap_private)
        return XV_ERROR;

    /* link the public to the private and vice-versa */
    bitmap_public->private_data = (Xv_opaque)bitmap_private;
    bitmap_private->public_self = (Xv_opaque)bitmap_public;

    /* set up event handlers to get resize and repaint events */
    xv_set(bitmap_public,
        WIN_NOTIFY_SAFE_EVENT_PROC,      bitmap_redraw,
        WIN_NOTIFY_IMMEDIATE_EVENT_PROC, bitmap_redraw,
        NULL);

    return XV_OK;
}
```

As you can see, the event handling function is declared and used the same way as for the logo object.

25.8.2 The Bitmap Set Method

The set method looks for the BITMAP_FILE attribute and, when given, reads in the corresponding bitmap file. Since the BITMAP_PIXMAP attribute is a get-only attribute, if given, the programmer is warned of the error via xv_error().

```
bitmap_set(bitmap_public, avlist)
Bitmap_public *bitmap_public;
Attr_avlist avlist;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);
    Attr_attribute *attrs;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int) attrs[0]) {
            case BITMAP_FILE : {
                int val, x, y;
                Display *dpy =
                    (Display *)xv_get(bitmap_public, XV_DISPLAY);
                Window window =
                    (Window)xv_get(bitmap_public, XV_XID);
                Pixmap old = bitmap_private->bitmap;
                if (XReadBitmapFile(dpy, window, attrs[1],
                    &bitmap_private->width, &bitmap_private->height,
                    &bitmap_private->bitmap, &x, &y) != BitmapSuccess)
                {
                    xv_error(bitmap_public,
                        ERROR_STRING, "Unable to load bitmap file",
                        ERROR_PKG,    BITMAP,
                        NULL);
                    bitmap_private->bitmap = old;
                }
                break;
            }
            case BITMAP_PIXMAP :
                xv_error(bitmap_public,
                    ERROR_CANNOT_SET, attrs[0],
                    ERROR_PKG,        BITMAP,
                    NULL);
                break;
            case XV_END_CREATE : {
                /* this stuff *must* be here rather than in the "init"
                 * routine because the CMS is not loaded into the
                 * window object until the "set" routines are called.
                 */
                Cms cms = xv_get(bitmap_public, WIN_CMS);
                XGCValues gcvalues;
                Display *dpy =
                    (Display *)xv_get(bitmap_public, XV_DISPLAY);
                gcvalues.foreground =
                    (unsigned long)xv_get(cms, CMS_FOREGROUND_PIXEL);
                gcvalues.background =
                    (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
                gcvalues.graphics_exposures = False;
                bitmap_private->gc =
                    XCreateGC(dpy, xv_get(bitmap_public, XV_XID),
                        GCForeground|GCBackground|GCGraphicsExposures,
                        &gcvalues);
            }
            default :
                xv_check_bad_attr(BITMAP, attrs[0]);
                break;
        }
    return XV_OK;
}
```

25.8.3 The Bitmap Get Method

The get method supports the BITMAP_PIXMAP attribute by returning a handle to the actual Pixmap used by the bitmap object. The BITMAP_FILE attribute cannot be gotten, but rather than producing an error message—we could do so—instead we just fall through to the default case and set the status parameter to the value returned by xv_check_ bad_attr().

```
bitmap_get(bitmap_public, status, attr, args)
Bitmap_public    *bitmap_public;
int              *status;
Attr_attribute   attr;
Attr_avlist      args;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);

    switch ((int) attr) {
        case BITMAP_PIXMAP :
            return (Xv_opaque)bitmap_private->bitmap;
        case BITMAP_FILE : /* can't get this attribute */
        default :
            *status = xv_check_bad_attr(BITMAP, attr);
            return (Xv_opaque)XV_OK;
    }
}
```

As you may recall, xv_check_bad_attr() checks that the attribute given is part of the package specified in the first parameter. If so, xv_error() is called, warning that the attribute was not handled. This is an error because attribute *does* apply to the package and should have been evaluated appropriately. This happens to be the case in the above scenario; the BITMAP_FILE attribute belongs to the Bitmap package, but we have not handled the attribute. Thus, a warning is printed saying:

```
XView Warning: Bitmap attribute not allowed.
```

Since the bitmap package does not allow the programmer to use xv_get() for the BITMAP_ FILE attribute, this is an appropriate warning.

The `status` variable is set to the return value of `xv_check_bad_attr()` because this is the same value needed by the internals to XView—that is, the function that called this get routine. Recall that the get routine for each package is called until one of them returns `XV_OK` in the `*status` parameter. This is the indicator that the attribute passed applies to that particular package and that the sequence of calling the packages' get functions should cease. Since `BITMAP_FILE` does apply to the Bitmap package, `*status` is set to `XV_OK`. Thus, when the function returns `XV_OK`, the get method stops and returns to the programmer.

As an opposite case, consider what happens when an attribute that does *not* apply to the Bitmap package is evaluated. Let's say the programmer called:

```
int width = (int)xv_get(bitmap, XV_WIDTH);
```

In this case, the `switch` would fall through to the default case and call `xv_check_bad_attr()`. This time, however, the attribute does not belong to the Bitmap package (it belongs to the generic package) and an error message is not printed. `xv_check_bad_attr()` returns `XV_ERROR` indicating that the get method should continue on to the next package in the class hierarchy. Thus, `*status` is set correctly.

This interface for `xv_check_bad_attr()` may seem confusing, but if you follow a simple rule of thumb, it can be made easy. Always have the default case in a `switch` statement set the `status` variable to the return value of `xv_check_bad_attr()`, and always return `XV_OK` from the function itself.

There still exists one problem, but there is no way to overcome it in the current XView API. That is, if the program that called `xv_get()` ever passes a bad or invalid attribute, the function returns `XV_OK` and it is impossible to determine if that value is a legitimate return value. Fortunately, this type of error should be worked out before it ever gets to the end user, so application developers should pay close attention to the error messages printed to `stderr`.

25.8.4 Creating a Bitmap Instance

The rest of the implementation for the Bitmap package may be seen in Appendix F, *Example Programs*. The destroy method is the same as the logo's destroy method and there is no find method for the Bitmap package. In this section, we show a small example program that demonstrates how an application might create an instance from the bitmap package.

Example 25-2. The bitmap.c program

```
/* bitmap.c -- demonstrate the use of the Bitmap package. */
#include <xview/xview.h>
#include <xview/cms.h>
#include "bitmap.h"

main(argc, argv)
char *argv[ ];
{
    Frame frame;
    Cms cms;
    Bitmap bitmap;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

Example 25-2. The bitmap.c program (continued)

```
    if (argc <= 1)
        puts("Specify bitmap filename"), exit(1);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    cms = xv_create(NULL, CMS,
        CMS_SIZE,       2,
        CMS_NAMED_COLORS, "blue", "red", NULL,
        NULL);
    bitmap = xv_create(frame, BITMAP,
        XV_WIDTH,       100,
        XV_HEIGHT,      100,
        WIN_CMS,        cms,
        BITMAP_FILE,    argv[1],
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

25.9 The Find Method

One aspect of the XView intrinsics that we have not yet addressed is the method for retrieving handles to objects of a particular class that already have been created. It is not appropriate to do this for the logo object because it is a window-based object. The need for a programmer to use `xv_find()` to retrieve an existing object stems from the ability to reuse the object in more than one instance.

Window-based objects cannot be rendered in more than one location on the screen, so the find method is not appropriate for packages subclassed from the `WINDOW` package. The most common example of objects that use a find method are those that have no windows associated with them. Fonts, for example, can be rendered anywhere as they are just handles to a type of information. Colormap segments can be shared by different windows, so it is possible to find existing colormap segments and reuse them.

25.9.0.1 To find or not to find

When choosing whether or not you should provide a *find* method to a package that you are writing, you should consider whether or not it makes sense to be able to share the same instance of the object in multiple places. If your XView object is application-specific (i.e., it cannot be used in a general way by "any" application), then you should probably reconsider whether you should make an XView object or try to implement it using existing XView objects or other methods.

Allowing the programmer to use `xv_find()` for a package helps the application keep down its use of system resources, such as memory. It also aids in performance, since the objects are shared among the entire application.

25.9.1 Conceptual Implementation

When the programmer calls `xv_find()`, the *find* method for the package specified (second parameter) is called. The purpose of this function is to cycle through all the objects that have been created of the package's type and find the one that matches the attributes specified. Recall that `xv_find()` may work just like `xv_create()` (see Chapter 2, *The XView Programmer's Model*).

In order to cycle through a list of objects of a particular package, that list must be created and updated every time an object is created or destroyed. That is, each time an object is created, the new object is added to the list, whereas each time an object is destroyed, it is removed from the list. The next issue is where to store this list.

It cannot reside within the package's private data since each instance of the object would have to be updated every time a new instance is created or another destroyed. Instead, we must choose a central location where the list can be obtained directly by the find and initialize procedures. We could choose a global, but private, variable representing the head of the list, but this would cause problems for packages that must have separate lists according to various constraints.

With fonts, for example, each server has unique font IDs and font objects cannot be shared among different servers. So, a list of font objects that have been created could be attached to the server object associated with the font. For colormap segments, since they are dependent upon colormaps, there has to be a separate list of available colormap segments for each available screen on a server (a cms associated with a color screen won't work very well with a cms assigned to a monochrome screen).

For all the XView objects that currently exist and that support `xv_find()`, you may find a different choice of implementation. If you design an XView object that is not unique to each server, you may very well wish not to attach the data to the server object. On the other hand, if the object depends on the unique qualities of the screen within the server (for which there can be many), then you may wish to attach the list head to the screen object associated with a server. Still, your XView object may not even depend on any X- or XView-related information in which case you needn't attach the list to any XView object at all. It may very well be a global variable that you access directly.

If you decide to follow the methods that some of the existing XView packages use, you may wish to attach lists to XView objects directly. Chapter 7, *Panels*, describes these methods, including `XV_KEY_DATA`. We can attach a list of objects from a particular package to another XView object (such as a `Xv_Server` object) using the attribute `XV_KEY_DATA` with the package *identifier* (ID) as the *key** and the head of the list as the data type for the key.

*We don't have to use the package ID as the key, but since it tends to be distinct from the other package IDs, it is a good choice.

25.9.1.1 Scope of list availability

Wherever you decide to attach the list of your XView objects, remember that the list is restricted to the application. It is impossible for `xv_find()` to retrieve an instance of an XView object that was created on a separate application that happens to be running on the same machine. Also, note that while there is one server that may support many applications running concurrently, XView creates an instance of a *server* object on a per-application basis, so attaching lists to a server or screen object does not imply that the list is available outside your application's context.

25.9.2 Actual Implementation

When the programmer calls `xv_find()`, XView starts with the package specified and works its way to the `GENERIC` package until it calls that package's find procedure, if available. If a package returns an object, then XView terminates the calling sequence and returns the object found. If no object is actually returned, XView may automatically create the object by invoking the initialize method just as if the programmer called `xv_create()` rather than `xv_find()`. It will only do this if the attribute `XV_AUTO_CREATE` is `TRUE` (the default). If the programmer sets this attribute to `FALSE`, then `xv_find()` returns `NULL` and the programmer doesn't get an object.

Because of the sequence that XView uses to call the find method for classes, package-specific attributes are considered first, followed by the more generic ones. If the programmer calls `xv_find()` and passes only one attribute-value pair, such as `XV_WIDTH,100` then if no objects of the package type requested is found, the more generic (parent) packages' find methods are called until one returns an object that happens to have a width of `100` pixels.

25.10 The Image Package

Server images (a front end for `Pixmaps`), like fonts, can be rendered anywhere on the screen (in windows, in other server images, and so on), under certain constraints (e.g., window depth and so on). So it is possible to find and reuse instances of server images. However, the current implementation of the server image package does not support a find method. So, we are going to demonstrate how to implement the find method by creating an extension to the server image package that does nothing but support the call to `xv_find()`. The new package is called *Image* and does not have any package-specific attributes.

There are three routines that provide the functionality of the *find* method: the initialize routine, the destroy routine, and the find routine itself. The initialize routine is responsible for creating the list or, if it already exists, adding the newly created object to the list. The destroy routine is responsible for removing the instance of the object being destroyed from the list. Lastly, the find routine is responsible for checking the list for matching attribute-value pairs and returning the matching object.

25.10.0.1 The public image header file

The public header file for the image package is fairly simple:

```
#include <xview/xview.h>
#include <xview/svrimage.h>

extern Xv_pkg image_pkg;

#define IMAGE &image_pkg

typedef Xv_opaque Image;

#define ATTR_PKG_IMAGE        ATTR_PKG_UNUSED_FIRST+1

typedef struct {
    Xv_server_image   parent_data;
    Xv_opaque         private_data;
} Image_public;
```

The `parent_data` field of the public image structure is `Xv_server_image` because the image package is subclassed from `Server_image`.

25.10.0.2 The private image header file

The private header file is equally simple:

```
#include <stdio.h> /* for BUFSIZ */
#include "image.h"

typedef struct _image {
    Xv_object      public_self;          /* pointer back to self */
    char          *filename;             /* for get/find */
    Xv_Screen      screen;               /* need to retain for list */
    struct _image *next;                 /* linked list for find */
} Image_private;

#define IMAGE_PUBLIC(item)      XV_PUBLIC(item)
#define IMAGE_PRIVATE(item)     XV_PRIVATE(Image_private, Image_public, item)
```

The private data structure contains several fields that enable the image package to allow the find method to work. The `filename` field is used to save the filename specified as the value to the `SERVER_IMAGE_BITMAP_FILE` attribute. Because the server image implementation does not save this data, the image package does. The `next` field is used to create a linked list of these objects to attach to the screen object set in the `screen` field. This is the list that the find method uses to find existing objects from a call to `xv_find()`.

25.10.0.3 The image package declaration

The package is initialized in the following way:

```
Xv_pkg image_pkg = {
    "Image",                    /* package name */
    ATTR_PKG_IMAGE,             /* package ID */
    sizeof(Image_public),       /* size of the public struct */
    SERVER_IMAGE,               /* subclassed from the server image */
    image_init,
    image_set,
    image_get,
    image_destroy,
    image_find
};
```

25.10.1 The Image Initialize Method

The task of the initialize routine for the image class is primarily to initialize the private data of the image object and to create and/or add to the linked list of the private data types. The routine is defined as follows:

```
image_init(owner, image_public, avlist)
Xv_Screen       owner;
Image_public   *image_public;
Attr_avlist     avlist;           /* ignored here */
{
    Attr_attribute *attrs;
    Image_private *image_private = xv_alloc(Image_private);
    Image_private *list; /* linked list of image instances */
    Xv_Screen screen = owner? owner : xv_default_screen;

    if (!image_private || !screen)
        return XV_ERROR;

    /* link the public to the private and vice-versa */
    image_public->private_data = (Xv_opaque)image_private;
    image_private->public_self = (Xv_opaque)image_public;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        if (attrs[0] == SERVER_IMAGE_BITMAP_FILE)
    /* you might also want to check that image_private->filename is NULL*/
            image_private->filename =
                strcpy(malloc(strlen(attrs[1])+1), attrs[1]);

    image_private->next = (Image_private *)NULL;
    image_private->screen = screen;

    /* get the list of existing images from the screen */
    if (list = (Image_private *)xv_get(screen,
                    XV_KEY_DATA, ATTR_PKG_IMAGE)) {
        /* follow list till the end */
        while (list->next)
            list = list->next;
        /* assign new image object to end of list */
```

```
            list->next = image_private;
        } else {
            /* no image objects on this screen -- create a new list */
            xv_set(screen,
                XV_KEY_DATA, ATTR_PKG_IMAGE, image_private,
                NULL);
        }
        return XV_OK;
    }
```

The `owner` for the image object is an `Xv_Screen` object just as with the `Server_image` object. Once the private data is allocated and the public and private structures are linked to one another, the fields are initialized. The `avlist` is scanned, checking for `SERVER_IMAGE_BITMAP_FILE`. If the programmer called `xv_create()`, passing that attribute, then we need to find out what its value is so we can store it for later retrieval. Remember, we must do this because the `Server_image` package does not.

Next, the screen object is queried to see if there have been any other Image objects stored. The list returned, if any, is the actual linked list of objects. We need this list in order to append the new instance to the end of it. If the list does not exist, the instance created here is set as the head of the list and is stored in the screen object via `XV_KEY_DATA`. The package ID is used as the key identifier and the new instance is used as the head of the list.

25.10.2 The Image Set Method

The only purpose for the Image's set method is to test to see if the programmer is changing the image's pixmap. If so, that would invalidate the value for `SERVER_IMAGE_BITMAP_FILE`, if set. If not set, then nothing is done.

```
    image_set(image_public, avlist)
    Image_public   *image_public;
    Attr_avlist     avlist;
    {
        Attr_attribute *attrs;
        Image_private *image_private = IMAGE_PRIVATE(image_public);

        /* loop thru attrs looking for anything that would invalidate
         * the fact that the filename is set to a valid file.  If the
         * programmer is assigning a new pixmap or data to this server
         * image, the filename that was originally associated with the
         * object is no longer valid.  Disable for later get/find calls.
         */
        if (image_private->filename)
            for (attrs = avlist; *attrs; attrs = attr_next(attrs))
                if (attrs[0] == SERVER_IMAGE_PIXMAP ||
                    attrs[0] == SERVER_IMAGE_BITS   ||
                    attrs[0] == SERVER_IMAGE_X_BITS) {
                        free(image_private->filename);
                        image_private->filename = NULL;
                }

        return (Xv_opaque)XV_OK;
    }
```

25.10.3 The Image Get Method

The get routine for the Image package provides the ability to return values for SERVER_ IMAGE_BITMAP_FILE and XV_SCREEN. The filename is stored in the Image's private data structure, so we return this value, if set. The function is as follows:

```
image_get(image_public, status, attr, args)
Image_public   *image_public;
int            *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    Image_private *image_private = IMAGE_PRIVATE(image_public);

    switch ((int) attr) {
        case SERVER_IMAGE_BITMAP_FILE :
            return (Xv_opaque)image_private->filename;
        case XV_SCREEN :
            return (Xv_opaque)image_private->screen;
        default :
            *status = xv_check_bad_attr(IMAGE, attr);
            return (Xv_opaque)XV_OK;
    }
}
```

25.10.4 The Image Destroy Method

When an instance of the Image class is destroyed, the destroy procedure is called with the status of DESTROY_CLEANUP. The first parameter to the destroy function is a handle to the object being destroyed. The task of the destroy function for the Image package is to remove the item from the list of items attached to the screen object and free it. Once the object has been freed, all references to the object become invalid. And of course, once the object has been removed from the screen's list, then xv_find() will fail to find it.

```
image_destroy(image_public, status)
Image_public    *image_public;
Destroy_status   status;
{
    Image_private *image_private = IMAGE_PRIVATE(image_public);
    Image_private *list; /* linked list of image instances */
    Xv_Screen screen = image_private->screen;

    if (status == DESTROY_CLEANUP) {
        /* get the list of existing images from the screen */
        list = (Image_private *)xv_get(screen,
                        XV_KEY_DATA, ATTR_PKG_IMAGE);
        if ((Image)XV_PUBLIC(list) == (Image)image_public)
            xv_set(screen,
                XV_KEY_DATA, ATTR_PKG_IMAGE, list->next,
                NULL);
        for ( ; list->next; list = list->next)
            if ((Image)XV_PUBLIC(list->next) == (Image)image_public) {
                list->next = list->next->next;
```

```
                    break;
                }
            if (list->filename)
                free(list->filename);
            free(list);
        }

        return XV_OK;
    }
```

25.10.5 The Image Find Method

The find procedure is the main purpose of the Image package. Its purpose is to find an existing Image object whose attributes match those specified to the programmer's call to `xv_find()`. If there is more than one matching object, the find routine usually returns the first one found because it is simpler to implement it that way. However, this is not required and the object returned may be arbitrary provided that the specified attributes match.

```
image_find(owner, pkg, avlist)
Xv_Screen       owner;
Xv_pkg         *pkg;
Attr_avlist     avlist;            /* ignored here */
{
    Image_private *list; /* linked list of image instances */
    /* this is what the server image package does */
    Xv_Screen screen = owner? owner : xv_default_screen;
    Attr_attribute *attrs;
    /* consider all the attrs we allow "find" to match on */
    int     width = -1, height = -1, depth = -1;
    Pixmap  pixmap = (Pixmap)NULL;
    char   *filename = NULL;

    /* get the list of existing images from the screen */
    list = (Image_private *)xv_get(screen,
                    XV_KEY_DATA, ATTR_PKG_IMAGE);

    if (!list)
        return NULL;

    /* loop thru each attribute requested and save the value
     * associated with it.  Later, we'll loop thru the existing
     * objects looking for the object that has the same values.
     */
    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int)attrs[0]) {
            case XV_WIDTH :
                width = (int)attrs[1];
                break;
            case XV_HEIGHT :
                height = (int)attrs[1];
                break;
            case SERVER_IMAGE_DEPTH :
                depth = (int)attrs[1];
                break;
            case SERVER_IMAGE_PIXMAP :
                pixmap = (Pixmap)attrs[1];
                break;
            case SERVER_IMAGE_BITMAP_FILE :
                filename = (char *)attrs[1];
                break;
            case SERVER_IMAGE_BITS :
            case SERVER_IMAGE_X_BITS :
            case SERVER_IMAGE_COLORMAP :
            case SERVER_IMAGE_SAVE_PIXMAP :
            default :
                return NULL; /* you can't "find" for these attrs */
        }
    /* Now loop thru each object looking for those whose
     * value that match those specified above.
     */
    for ( ; list; list = list->next) {
        /* If it doesn't match, continue to the next object in
         * the list.  Repeat for each requested attribute.
         */
        if (width > -1 &&
            (width != (int)xv_get(XV_PUBLIC(list), XV_WIDTH)))
            continue;
        if (height > -1 &&
            (height != (int)xv_get(XV_PUBLIC(list), XV_HEIGHT)))
            continue;
        if (depth > -1 && (depth != (int)xv_get(XV_PUBLIC(list),
                                    SERVER_IMAGE_DEPTH)))
            continue;
        if (pixmap && (pixmap != (Pixmap)xv_get(XV_PUBLIC(list),
                                    SERVER_IMAGE_PIXMAP)))
            continue;
        if (filename &&
            (!list->filename || strcmp(filename, list->filename)))
            continue;
        /* all matches seemed to be successful, return this object */
        return XV_PUBLIC(list);
    }
    /* nothing found */
    return NULL;
}
```

A find procedure can be implemented in many ways; the one provided above is just one way.

25.10.6 The Image.c Program

Example 25-3 demonstrates one way that the image package can be used. It creates a pixmap based on the bitmap filename given on the command line (a filename must be given). Then it uses `xv_find()` to find the same object first by the same bitmap filename and then again by using the `Pixmap` associated with the image.

While the program functionally does very little, it is intended to demonstrate how multiple server images can be shared in an application via the new Image package.

Example 25-3. The image.c program

```
/* image.c -- demonstrate the use of the image package. */
#include <xview/xview.h>
#include "image.h"

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame;
    Image image1, image2;
    Pixmap pixmap;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    if (argc < 2)
      puts("specify filename"), exit(1);

    /* frame = (Frame)xv_create(NULL, FRAME, NULL); */
    if (!(image1 = xv_create(NULL, IMAGE,
        XV_WIDTH,                  100,
        XV_HEIGHT,                 100,
        SERVER_IMAGE_BITMAP_FILE,  argv[1],
        NULL)))
          puts("unsuccessfully created image1"), exit(1);
    if (!(image2 = xv_find(NULL, IMAGE,
        SERVER_IMAGE_BITMAP_FILE,  argv[1],
      NULL)))
          puts("unsuccessfully created image2"), exit(1);
    printf("image1 %s image2\n",
      (image2 != image1)? "matched" : "didn't match");
    pixmap = (Pixmap)xv_get(image1, SERVER_IMAGE_PIXMAP);
    if (!(image2 = xv_find(NULL, IMAGE,
        SERVER_IMAGE_PIXMAP,  pixmap,
      NULL)))
          puts("unsuccessfully created image2"), exit(1);
    printf("image1 %s image2\n",
      (image2 != image1)? "matched" : "didn't match");

    /* window_fit(frame); */
    /* xv_main_loop(frame); */
}
```

25.11 The Wizzy Package—A Panel Item Extension

This section presents an implementation for a panel item extension. This package is called the Wizzy package; this example does not actually do anything but is presented to show how to make extensions to the existing `PANEL` package. A panel item extension could be used to define a new panel item. For example, using the methods described in this section, you could implement a slider item that selects a range of values, rather than a single value. For this example, you need to be familiar with the previous examples presented in this chapter and you should also read Section 7.3.2, "Panel Item Layout," in Chapter 7, *Panels*.

25.11.1 The Public Wizzy Header File

The public *wizzy.h* header file is defined as follows:

```
/* wizzy.h -- public header file for the Wizzy Xview class. */

#include <xview/xview.h>
#include <xview/panel.h>

extern Xv_pkg   xv_panel_wizzy_pkg;

#define WIZZY   &xv_panel_wizzy_pkg;

typedef Xv_panel_extension_item Wizzy;

#define ATTR_PKG_WIZZY              ATTR_PKG_UNUSED_FIRST
#define WIZZY_ATTR (type, ordinal) ATTR(ATTR_PKG_WIZZY, type, ordinal)

typedef enum {
     WIZZY_OFFSET    = WIZZY_ATTR(ATTR_INT, 1),
     WIZZY_FLAG = WIZZY_ATTR(ATTR_BOOLEAN, 2)
} Wizzy_attr;
```

25.11.2 The Private Wizzy Header File

The private data is declared for the `Wizzy` object in the implementation-specific header file, *wizzy_impl.h*.

```
/* wizzy_impl.h -- private header file for the Wizzy Xview class. */

#include "wizzy.h"

typedef struct {
     Panel_item     public_self;   /* pointer back to self */
     Rect           block;         /* a rect this item's panel value */
     GC             gc;            /* a GC for this item */
     int            offset;        /* an offset for block */
     Panel          panel;         /* Panel that this item is owned by */
     int            flag;          /* some boolean value */
#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
```

```
     int           has_kbd_focus; /* TRUE or FALSE */
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
} Wizzy_private;

#define WIZZY_PUBLIC(item)   XV_PUBLIC(item)
#define WIZZY_PRIVATE(item)  XV_PRIVATE(Wizzy_private, Wizzy, item)

#define BLOCK_WIDTH          16
#define BLOCK_HEIGHT         12
#define INITIAL_OFFSET       10
```

The only entry required for the `Wizzy_private` structure is `public_self`. All other entries illustrate how you could implement the private data structure; they should be replaced with your item's private data requirements.

25.11.3 The Wizzy Package Declaration

The package is initialized in the following way:

```
     #include <xview/wizzy.h>

     extern Xv_pkg xv_panel_item_pkg;

     Pkg_private int wizzy_init();
     Pkg_private Xv_opaque wizzy_set_avlist();
     Pkg_private Xv_opaque wizzy_get_attr();
     Pkg_private int wizzy_destroy();

     Xv_pkg    xv_panel_wizzy_pkg = {
         "Wizzy Item",
         ATTR_PKG_WIZZY,
         sizeof(Wizzy),
         &xv_panel_item_pkg,
         wizzy_init,
         wizzy_set_avlist,
         wizzy_get_attr,
         wizzy_destroy,
         NULL            /* no find proc */
     };
```

25.11.4 The Implementation Files

The implementation file for the Wizzy package is similar to the previous packages. It includes an initialize method, a set method, a get method, a destroy method, and an additional *panel operations vector table*. Panel item handler procedures need to be defined by the package and placed in the *panel operations vector table* (see the description of panel item handler procedures in Section 25.11.9, "Panel Item Handler Procedures"). XView defines the order of procedures in the panel operations vector table. If you do not define a particular function, you should place `NULL` in the appropriate position for the function. There are fifteen procedures, with one additional procedure reserved for future use. Once these procedures are declared, they can be specified in the `Panel_ops` table. The declaration for panel-item handler procedures for the Wizzy Package follow:

```
static void wizzy_begin_preview();
static void wizzy_update_preview();
static void wizzy_accept_preview();
static void wizzy_cancel_preview();
static void wizzy_accept_menu();
static void wizzy_accept_key();
static void wizzy_clear();
static void wizzy_paint();
static void wizzy_resize();
static void wizzy_remove();
static void wizzy_restore();
static void wizzy_layout();
static void wizzy_accept_kbd_focus();
static void wizzy_yield_kbd_focus();
```

The panel item operations table itself is declared as follows:

```
static Panel_ops ops = {
     panel_default_handle_event,   /* handle_event() */
     wizzy_begin_preview,          /* begin_preview() */
     wizzy_update_preview,         /* update_preview() */
     wizzy_cancel_preview,         /* cancel_preview() */
     wizzy_accept_preview,         /* accept_preview() */
     wizzy_accept_menu,            /* accept_menu() */
     wizzy_accept_key,             /* accept_key() */
     wizzy_clear,                  /* clear() */
     wizzy_paint,                  /* paint() */
     wizzy_resize,                 /* resize() */
     wizzy_remove,                 /* remove() */
     wizzy_restore,                /* restore() */
     wizzy_layout,                 /* layout() */

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
     wizzy_accept_kbd_focus,       /* accept_kbd_focus() */
     wizzy_yield_kbd_focus,        /* yield_kbd_focus() */
#else
     NULL,                         /* accept_kbd_focus() */
     NULL,                         /* yield_kbd_focus() */
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
     NULL                          /* reserved for future use */
};
```

The Section 25.11.9, “Panel Item Handler Procedures,” provides a complete explanation of each panel item handler procedure.

25.11.5 The Wizzy Initialize Method

The task of the initialize routine for the Wizzy class is primarily to initialize the private data of the Wizzy object and set any create-only attributes.

```
Pkg_private int
wizzy_init(panel, item, avlist)
     Panel      panel;     /* parent */
```

```
        Panel_item  item; /* this object */
        Attr_avlist avlist;     /* attribute-value pair list */

    {
        Wizzy_public    *item_object = (Wizzy_public *)item; /* this item */
        Display         *display;
        Wizzy_private   *dp;
        XGCValues       gcvalues;
        XID             xid;
        Attr_attribute  *attrs;

        dp = xv_alloc(Wizzy_private);

        /* link the public to the private, link the private to the public */

        item_object->private_data = (Xv_opaque)dp;
        dp->public_self = item;

        /* initialize any non-zero private data members */

        display = (Display *)XV_DISPLAY_FROM_WINDOW(panel);
        xid = (XID)xv_get(panel, XV_XID);
        gcvalues.foreground = BlackPixel(display, 0);
        dp->gc = XCreateGC(display, xid, GCForeground, &values);
        dp->offset = INITIAL_OFFSET;
        dp->panel = panel;

        /* Process any create-only attributes from avlist */

        for ( attrs = avlist; *avlist; attrs = attr_next(attrs)) {
              switch ( (int)attrs[0] )     {

                    /* case <create_only_attr>: */

                    default:
                    break;

                    }
              }

        xv_set(item, PANEL_OPS_VECTOR, &ops,
    #ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
                    PANEL_ACCEPT_KEYSTROKE, TRUE,
    #endif WIZZY_CAN_ACCEPT_KBD_FOCUS
                    NULL);

        return XV_OK;

    }
```

Once the private data is allocated and the public and private structures are linked to one another, the private data fields are initialized. Then the `avlist` is scanned for any create-only attributes.

Next, `xv_set` is used to store the address of the panel operations vector table and allow the Wizzy package to accept keyboard input as specified.

25.11.6 The Wizzy Set Method

The code for the set method is as follows:

```
Pkg_private Xv_opaque
wizzy_set_avlist(item, avlist)
      Panel_item  item;       /* this object */
      Attr_avlist avlist;           /* attribute list */

{

      Wizzy_private     *dp = WIZZY_PRIVATE(item);
      Xv_opaque         result;
      Rect        value_rect;
      Attr_attribute    *attrs;

      /* Parse panel item generic attributes before parsing Wizzy
       * specific attributes and prevent panel_redisplay_item() from
       * being called in item_set_avlist()
       */

      if (*avlist != XV_END_CREATE) {
            xv_set( dp->panel, PANEL_NO_REDISPLAY_ITEM, TRUE, NULL);
            result = xv_super_avlist( item, &xv_wizzy_panel_pkg, avlist);
            xv_set( dp->panel, PANEL_NO_REDISPLAY_ITEM, FALSE, NULL);

            if (result != XV_OK)
                  return (result);

            }

      for ( attrs = avlist; *avlist; attrs = attr_next(attrs)) {
            switch ( (int)attrs[0] )     {

            case WIZZY_OFFSET:
                  dp->offset = attrs[1];
                  break;

            case WIZZY_FLAG:
                  dp->flag = attrs[1];
                  break;

            case XV_END_CREATE:
                  value_rect = *(Rect *)xv_get(item, PANEL_ITEM_VALUE_RECT);
                        rect_construct(&dp->block,
                  value_rect.r_left + dp->offset,
                  value_rect.r_top,
                  BLOCK_WIDTH, BLOCK_HEIGHT);

                  value_rect = rect_bounding(&value_rect, &dp->block);

      /* Note: setting the value rect will cause the item
      * item rect to be recalculated as the enclosing rect
      * containing both the label and value rects.
      */

                  xv_set( item, PANEL_ITEM_VALUE_RECT, &value_rect,
                        NULL);
                  break;

                  default:
                  break;

                  }
            }

        return XV_OK; /* return XV_ERROR if something went very wrong */

    }
```

The purpose of the set method is to set any of the attributes defined in the public header file by storing the corresponding value in the private data structure. In the case of the Wizzy package, `WIZZY_OFFSET` and `WIZZY_FLAG` are the only attributes and they correspond to `offset` and `flag`, respectively.

However, before parsing attributes specific to the Wizzy object, it is necessary to parse any attributes generic to Panel. This is done by a call to `xv_super_avlist()`. It is also necessary to prevent the parent panel from redisplaying while these attributes are being set. This is taken care of by setting the attribute `PANEL_NO_REDISPLAY_ITEM` to `TRUE` for the parent panel and then resetting it to `FALSE`.

After all attributes are handled, the value rectangle `value_rect` is constructed and the item rectangle is recalculated since it encloses both the label and value rectangles.

25.11.7 The Wizzy Get Method

The get routine for the Wizzy package provides the ability to return values for `WIZZY_OFFSET` and `WIZZY_FLAG`.

```
    Pkg_private Xv_opaque
    wizzy_get_attr(item, status, which_attr, avlist)

        Panel_item          item;
        int                 *status;
        Attr_attribute  which_attr;
        va_list             avlist;

    {

        Wizzy_private    *dp = WIZZY_PRIVATE(item);

        switch ( (int)which_attr ) {
            case WIZZY_OFFSET:
                return (Xv_opaque)dp->offset;
                break;

            case WIZZY_FLAG:
                return (Xv_opaque)dp->flag;
                break;

            default:
                *status = xv_check_bad_attr(WIZZY, attr);
                return (Xv_opaque)XV_OK;

                }
    }
```

25.11.8 The Wizzy Destroy Method

When an instance of the Wizzy class is destroyed, the destroy procedure is called. The first parameter to the destroy function is a handle to the panel item being destroyed. The second parameter is the destroy status (see Chapter 20, *The Notifier*, for more information on `destroy_status`). The task of the destroy function for the Wizzy package is to remove the item from the list of items attached to the panel object and free it. Once the object has been freed, all references to the object become invalid.

```
Pkg_private int
wizzy_destroy(item, status)
      Panel_item      item;
      Destroy_status  status;

{

      Wizzy_private    *dp = WIZZY_PRIVATE(item);

      if ( (status==DESTROY_CHECKING) || (status==DESTROY_YOURSELF) )
            return XV_OK;

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
      wizzy_remove(item);
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS

      free(dp);

      return XV_OK;

}
```

25.11.9 Panel Item Handler Procedures

25.11.9.1 The handle event function

The handle event function allows the application writer to specify a notify procedure for the panel item.

25.11.9.2 The begin preview function

The begin preview function is called when SELECT-down has been detected. Highlight the item to show active feedback but don't actually take any action yet. Private data may be accessed as necessary. The function has the form:

```
static void
wizzy_begin_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.3 The update preview function

The update preview function is called when the pointer has been dragged within the item after begin preview was detected. Adjust highlighting to reflect the new position of the pointer and update the appropriate private data. The function has the form:

```
static void
wizzy_update_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.4 The cancel preview function

The cancel preview function is called when the pointer has been dragged out of the item after begin preview was detected. Remove the active feedback (i.e., de-highlight) and clean up any private data. The function has the form:

```
static void
wizzy_cancel_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.5 The accept preview function

The accept preview function is called when the SELECT button has been released over the item. Remove the active feedback (i.e., de-highlight), paint the busy feedback, perform the action associated with the item, and then remove the busy feedback. Also update any private data as necessary. The function has the form:

```
static void
wizzy_accept_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.6 The accept menu function

The accept menu function is called when the MENU button has been pressed over the item. Show the menu item attached to the item, if any. The function has the form:

```
static void
wizzy_accept_menu (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.7 The accept key function

The accept key function is called when a keyboard event has been detected. Process the key and update the data and/or display as necessary. The function has the form:

```
static void
wizzy_accept_key (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.8 The clear function

The clear function is called whenever the item's rectangle needs to be cleared. Clear the item rectangle and update any private data as needed. An example of this function would be:

```
static void
wizzy_clear (item, event)
     Panel_item item;
     Event      *event;
{

     panel_default_clear_item(item);

}
```

25.11.9.9 The paint function

The paint function is called when the panel needs to be repainted. Do everything necessary to paint the entire item but do not go outside of the rectangle describing the boundaries of the item. An example of this function would be:

```
static void
wizzy_paint(item)
    Panel_item      item;

    Display            *display;
    Wizzy_info         *dp = WIZZY_PRIVATE(item);
    Panel_paint_window *ppw;  /* ptr to Panel_paint_window structure */
    Xv_Window          pw;    /* paint window */
    XID                xid;
    /* Paint the label */
    panel_paint_label(item);

    /* Paint the value.
     * In this wizzy example, we paint something into dp->block.
     */
    display = (Display *) XV_DISPLAY_FROM_WINDOW(dp->panel);
    for (ppw = (Panel_paint_window *)
                xv_get(dp->panel, PANEL_FIRST_PAINT_WINDOW);
         ppw;
         ppw = ppw->next) {
               pw = ppw->pw;    /* pw = the actual window to paint in */
               xid = (XID) xv_get(pw, XV_XID);
               XFillRectangle(display, xid, dp->gc, dp->block.r_left,
                   dp->block.r_top, dp->block.r_width, dp->block.r_height);
    }
}
```

5.11.9.10 The resize function

The resize function is called when the panel has been resized. Recalculate any extend-to-edge dimensions. The function has the form:

```
static void
wizzy_resize (item)
     Panel_item item;
```

5.11.9.11 The remove function

The remove function is called when the item has been made hidden via `xv_set(item, XV_SHOW, FALSE)`. An example function might be:

```
static void
wizzy_remove(item)
    Panel_item      item;
{

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
    Wizzy_info      *dp = WIZZY_PRIVATE(item);
    Panel_status    *panel_status;

    /*
     * Only reassign the keyboard focus to another item
     * if the panel isn't being destroyed.
     */
    panel_status = (Panel_status *) xv_get(dp->panel, PANEL_STATUS);
    if (!panel_status->destroying &&
             xv_get(dp->panel, PANEL_CARET_ITEM) == item)
        (void) panel_advance_caret(dp->panel);
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
}
```

5.11.9.12 The restore function

The restore function is called when the item has been made visible via `xv_set(item, XV_SHOW, TRUE)`. An example function might look like:

```
static void
wizzy_restore(item)
    Panel_item      item;
{

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
    Wizzy_info     *dp = WIZZY_PRIVATE(item);

    /* If no item has the keyboard focus, then give this item the focus */
    if (!xv_get(dp->panel, PANEL_CARET_ITEM))
        xv_set(dp->panel, PANEL_CARET_ITEM, item, 0);
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
}
```

5.11.9.13 The layout function

The layout function is called when the item has been moved. Adjust the coordinates. An example function might look like:

```
     static void
     wizzy_layout(item, deltas)
         Panel_item item;
         Rect       *deltas;
     {
         Wizzy_info     *dp = WIZZY_PRIVATE(item);

         dp->block.r_left += deltas->r_left;
         dp->block.r_top += deltas->r_top;
     }
```

5.11.9.14 Accept keyboard focus function

The accept keyboard focus function is called when the keyboard focus has been set to this item. Change the keyboard focus feedback to active, and update private data as necessary. An example function might look like:

```
static void
wizzy_accept_kbd_focus(item)
    Panel_item      item;
{

    Wizzy_info     *dp = WIZZY_PRIVATE(item);
    Frame      frame;
    int        x;
    int        y;

    dp->has_kbd_focus = TRUE;
    frame = xv_get(dp->panel, WIN_FRAME);
    if (xv_get(dp->panel, PANEL_LAYOUT) == PANEL_HORIZONTAL) {
        xv_set(frame, FRAME_FOCUS_DIRECTION, FRAME_FOCUS_UP, 0);
        x = dp->block.r_left +
             (dp->block.r_width - FRAME_FOCUS_UP_WIDTH)/2;
             y = dp->block.r_top + dp->block.r_height - FRAME_FOCUS_UP_HEIGHT/2;
    } else {
           xv_set(frame, FRAME_FOCUS_DIRECTION, FRAME_FOCUS_RIGHT, 0);
           x = dp->block.r_left - FRAME_FOCUS_RIGHT_WIDTH/2;
            y = dp->block.r_top +
               (dp->block.r_height - FRAME_FOCUS_RIGHT_HEIGHT)/2;
    }
    if (x < 0)
       x = 0;
    if (y < 0)
       y = 0;
    panel_show_focus_win(item, frame, x, y);
}
```

5.11.9.15 The yield keyboard focus function

The yield keyboard focus function is called when the keyboard focus has been removed from this item. Change the keyboard focus back to inactive and update private data as necessary. An example function might look like:

```
static void
wizzy_yield_kbd_focus(item)
    Panel_item     item;
{
    Wizzy_info    *dp = WIZZY_PRIVATE(item);
    Xv_Window      focus_win;
    Frame     frame;

    dp->has_kbd_focus = FALSE;
    frame = xv_get(dp->panel, WIN_FRAME);
    focus_win = xv_get(frame, FRAME_FOCUS_WIN);
    xv_set(focus_win, XV_SHOW, FALSE, 0);
}
```

5.11.10 Panel Item Extension Attributes

Table 25-1 lists the attributes for use with Panel Item extensions. This information is described fully in the *XView Reference Manual*.

Table 25-1. Panel Item Extension Attributes

PANEL_ACCEPT_KEYSTROKE
PANEL_BUSY
PANEL_CURRENT_ITEM
PANEL_FIRST_PAINT_WINDOW
PANEL_FOCUS_PW
PANEL_GINFO
PANEL_ITEM_CREATED

Table 25-1. Panel Item Extension Attributes (continued)

PANEL_ITEM_DEAF
PANEL_ITEM_LABEL_RECT
PANEL_ITEM_VALUE_RECT
PANEL_ITEM_WANTS_ADJUST
PANEL_ITEM_WANTS_ISO
PANEL_ITEM_X_POSITION
PANEL_ITEM_Y_POSITION
PANEL_NO_REDISPLAY_ITEM
PANEL_OPS_VECTOR
PANEL_PRIMARY_FOCUS_ITEM
PANEL_STATUS

This page intentionally left blank

to preserve original page counts.

A
The Selection Service

This appendix describes the compatibility procedures and attributes that support the XView Selection Service. If you are creating a new application that uses selections, refer to Chapter 18, *Selections*. This appendix is provided only for those who need to use the selection service for compatibility reasons. It contains the text on selections that was found in older versions of this book. XView still supports the procedures and attributes described in this appendix. Details on these procedures and attributes may be found in appendices in the *XView Reference Manual*.

The X Window System provides several methods for separate applications to exchange information with one another. One of these methods is the use of the selection service. A *selection* transfers arbitrary information between two clients. An in-depth discussion of the selection mechanism that X provides is discussed in Volume One, *Xlib Programming Manual*. This chapter addresses XView's programmatic interface to the selection service provided by the X server.

While XView provides all the functions for applications to set and get selections of various sorts, OPEN LOOK applications must follow the conventions outlined in the *OPEN LOOK GUI Specification Guide*. All XView packages that have interactive text entry support the ability to make selections and to get selections from the server. In OPEN LOOK, you select objects in basically the same way that you select windows or icons—using the SELECT and ADJUST mouse buttons. OPEN LOOK describes three functions that operate on selected objects: CUT, COPY, and PASTE. These are *core functions* that are accessed from the keyboard.* CUT, COPY, and PASTE operations use the *clipboard* to keep track of selected objects. The clipboard temporarily stores items selected via the selection service; it does not store selected windows or icons since these are not considered selectable objects by the selection service.

*The function keys that are bound to these functions vary from keyboard to keyboard depending on the make and model of the computer.

Table A-1 summarizes text selection for the OPEN LOOK GUI.

Table A-1. Selecting Text

Action	Off Selection	On Selection
Click SELECT	Insert point is set at the pointer location.	When SELECT is released, insert point is set at pointer location and selection is cleared.
Drag SELECT	Text is highlighted as pointer is dragged (wipe-through selection).	Text move pointer is displayed. When you release SELECT, text is moved to pointer location if that location is outside the highlighted area.
Click ADJUST	Extends the highlighting to the pointer location extending either the beginning or the end of the current selection.	Moves the end of the highlighting to the pointer location. Beginning of selection is preserved.
Drag ADJUST	Adjusts an existing selection as the pointer is dragged (wipe-through). Beginning of selection is anchored at insert point.	Adjusts an existing selection as pointer is dragged (wipe-through). Beginning of selection is anchored at insert point.

Note that although OPEN LOOK specifies the selection of graphic objects, no XView objects currently support selection of graphical objects. A possible implementation is to have a canvas object, which has graphic objects displayed in it, set selections based on the event sequences outlined in Table A-1. A *draw* application might consider a drawn geometric shape as a graphic object, whereas a *paint* application might consider the pixels in an arbitrary area as the graphical object.

A.1 The XView Selection Model

The XView selection model is based upon the requestor/owner model of peer-to-peer communications. The *owner* has the data representing the value of its selection, and the *requestor* receives it.

In the X environment, *all* data transferred between an owner and a requestor must transfer via the server. An X client cannot assume that another client can open the same files or even communicate directly. Such assumptions might result in an application that does not work in all network configurations or across heterogeneous computer architectures.

X makes provisions for selections and therefore generates certain events such as `PropertyNotify` when a selection is acquired. For XView to implement its selection service, XView tracks all events that might be generated from selections. Because of this, you cannot use any of the selection mechanisms provided by Xlib. If you do, you will generate events that you will not be able to receive and that XView will be confused about.

The XView selection library deals with four discrete *ranks* under the general term *selection*. Those ranks are: primary, secondary, shelf, and caret. Most familiar is the *primary* selection, which is normally indicated on the screen by inverting (*highlighting*) its contents. Selections made while a function key is held down (usually indicated with an underscore under the selection) are considered *secondary* selections.* These selections are used when the primary selection must be left undisturbed. The *shelf* (or *clipboard*) selection is used by the CUT and COPY operations to load the selection, while the PASTE operation retrieves the selection. Finally, the *caret* (the insertion point for interactive text objects) is also treated as a selection even though it has no contents.

When a user interface element, such as a text subwindow, wants to allow the user to make a selection, it must have a selection *client*. Through this client, the text subwindow can acquire a selection rank (primary, secondary, etc) and provide it with data (text). An application can have many such clients, but there is typically one client per XView object. Each selection rank is associated with each client—a separate client need not be created just to utilize other selection ranks. However, only one client can be the holder of a particular selection rank at any one time. If a client wishes to acquire the selection, the current selection *holder* must *yield* the selection to the new requestor. The client becomes the new holder of the selection and might provide any data it chooses.

A.2 Using the Selection Service

It is not necessary to create a selection service client just to query the holder of a selection or to get its contents. Since this is the most common usage, most of this chapter is dedicated to explaining this level of functionality. Creating a selection is typically used internally by packages in the XView library or by applications that wish to create their own objects that need to communicate via the selection service. Doing this is generally very intricate and complicated and is beyond the scope of this manual.

Nevertheless, to understand how to create a selection service client, it is best to learn how to request information from an existing client. Once you understand what to expect from a selection, you can understand how to create a client that provides information from other requests. Therefore, the bulk of the chapter addresses the process of querying for selections already provided by XView packages or other X-based applications.

OPEN LOOK assigns to function keys the special functions COPY, CUT, and PASTE. The function keys generate and correspond to the XView events `ACTION_CUT`, `ACTION_COPY`, and `ACTION_PASTE`. An OPEN LOOK application checks the state of these keys and modifies the selection rank accordingly. Note that it is the responsibility of the XView application (more specifically, each XView package) to set the state of the server's selection rank according to the state of some function keys. Unless you are writing your own XView package (a topic that this book does not address), you need not concern yourself about it.

*Which function key depends on your particular computer. By default the `L6` function key should work for Sun Workstations or the `F6` key for other computers.

Even though XView packages might query for the state of function keys, this does not interfere with normal event processing. All events that your application has registered to receive are not affected by the selection service. Note, however, that while the selection service might react to the state of these function keys, any action you take as a result of these keys might result in a *dual action* and might confuse the user. For example, if your application is coded to change the font of a text subwindow in the event of an `L8-up` event, you will not only get that event, but the selection service also gets it and will PASTE the contents of the shelf selection.

Whenever using the selection service either as a client or to query the selection from another client, application code must include the header file <*xview/seln.h*>. This file includes the files <*xview/sel_svc.h*> and <*xview/sel_attrs.h*>, which provide external declarations of available functions and data types.

A.3 Getting the Current Selection

Determining the current selection involves two steps: determining the holder of the current selection and getting the actual selection data from the holder. To determine who is holding the selection of a specified rank:

```
Seln_holder
selection_inquire(server, rank)
    Xv_Server server;
    Seln_rank rank;
```

The returned `holder` is used in other selection routines that allow you to access selection data for that rank. For example, the call `selection_ask()` asks for the data held by the holder of the selection. The form of the call is:

```
Seln_request *
selection_ask(server, holder, attrs)
    Xv_Server       server;
    Seln_holder    *holder;
    <attribute-value list>   attrs
```

The `holder` in this case is the address of the holder returned from the call to `selection_inquire()`.

The *server* is important to all the selection service routines because it identifies the X server in which the selection is associated. Because X applications can communicate with more than one server, specifying different servers can result in getting different selections.

The `Seln_rank` specifies which selection type you want. Its value corresponds with the four selection ranks outlined in the beginning of the chapter. However, there are six legal values in this enumerated data type:

`SELN_UNKNOWN`	This is an error value, not a value you would use as a parameter or a legal or known rank.
`SELN_CARET`	The caret selection is used with text subwindows and text panel items. It is usually used to describe where the insertion point is within the corresponding text stream of the object so there is no

	data associated with the caret selection. This selection type is not very widely used.
SELN_PRIMARY	The primary selection is the most widely used and is usually the default type in most user interfaces and applications.
SELN_SECONDARY	As noted earlier, the secondary selection is only used if the primary selection must not be removed or lost—or if it must remain clearly visible (visually selected) on the screen.
SELN_SHELF	The selection buffers are stored in files. Clients of the selection service do not access these files; the selection service just uses them as temporary storage for data. As noted, this is not a good place to hold selections.
SELN_UNSPECIFIED	When this rank is used, either the primary or the secondary selection is used depending on the state of the appropriate function keys. This is usually passed as the `rank` parameter to `selection_inquire()`.

`selection_ask()` returns a pointer to a `Seln_request` data structure. The function goes out and asks the server for the selection associated with the rank described in the `holder` parameter. All the information about the selection is held in this data structure. The attribute-value list following the `holder` parameter describes the attributes of the selection you are interested in.

Before we go on, let's show an example program that illustrates what we have covered so far. The simple program in Example A-1 has a panel button that prints the current primary selection to the standard output. The selection can be held by any client on the server.

Example A-1. The simple_seln.c program

```
/*
 * simple_seln.c -- print the primary selection by pressing the panel
 * button.  The selection may have originated from any window or
 * application on the server.
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/server.h>
#include <xview/seln.h>

Xv_Server server;

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
    Panel       panel;
    void        exit();
    int         print_seln();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

Example A-1. The simple_seln.c program (continued)

```
    frame = (Frame) xv_create(NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);
    panel = (Panel)xv_create(frame, PANEL,
        WIN_WIDTH,              WIN_EXTEND_TO_EDGE,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Print Selection",
        PANEL_NOTIFY_PROC,      print_seln,
        NULL);
    window_fit(panel);
    window_fit(frame);

    server = (Xv_Server)xv_get(xv_get(frame, XV_SCREEN), SCREEN_SERVER);

    xv_main_loop(frame);
}

/*
 * Get the selection using selection_ask().  Note that if the
 * selection is bigger than about 2K, the whole selection will
 * not be gotten with one call, thus this method of getting
 * the selection may not be sufficient for all situations.
 */
int
print_seln(item, event)
Panel_item item;
Event *event;
{
    Seln_holder         holder;
    Seln_request       *response;
    char                text[BUFSIZ];

    /* get the holder of the primary selection */
    holder = selection_inquire(server, SELN_PRIMARY);
    response = selection_ask(server, &holder,
        SELN_REQ_CONTENTS_ASCII, NULL,
        NULL);

    strcpy(text, response->data + sizeof (SELN_REQ_CONTENTS_ASCII));
    printf("---selection---\n%s\n---end seln---\n", text);

    return XV_OK;
}
```

`selection_ask()` does not return until it has contacted the server and gotten a response back from it. This implies that if the server does not respond, the application *blocks* until either a time-out occurs or the selection is received. The attribute-value pair that is passed (`SELN_REQ_CONTENTS_ASCII, NULL`) indicates that we are interested in the ASCII contents of the selection. Whether the selection is successful or not, a pointer to a `Seln_request` structure is returned. If there was an error, the `status` field of the structure will indicate so. If it succeeded, then the selection contents will be in the data field of the structure. All this is clarified in the next section.

A.3.1 The Seln_request Structure

The Seln_request data structure returned from selection_ask() contains information about the selection requested. The pointer returned points to static data that is overwritten on each call. Thus, if you need to save any of this data, it should be copied. The Seln_request structure is defined as follows:

```
typedef struct {
    Seln_replier_data   *replier;
    Seln_requester       requester;
    char                *addressee;
    Seln_rank            rank;
    Seln_result          status;
    unsigned             buf_size;
    char                 data[SELN_BUFSIZE];
} Seln_request;
```

If there is no selection or if the selection fails in any way, the status field in the data structure is set to one of the values in the enumerated type Seln_result. If status is set to SELN_FAILED, then the data field should not be examined as it will not contain any reliable values.

On the other hand, if selection_ask() returns successfully, the same attributes that were passed into the function are copied into the data byte array along with the new values (see Figure A-1).

```
Seln_request  *request;

...

request = selection_ask(server, &holder,
    SELN_REQ_FIRST,          NULL,
    SELN_REQ_LAST,           NULL,
    SELN_REQ_CONTENTS_ASCII, NULL,
    NULL);
```

Figure A-1 shows what is returned assuming that the selection contained the string "Now is the time for all ...." The data field contains all the attributes passed in to selection_ask(), but the attributes and the values are all aligned to 4-byte boundaries. This includes the *string* returned from the selection. If the selection string is not a multiple of 4, then it is NULL-padded. The NULL-terminating byte of the string is required—if the last character of the string aligns to a 4-byte boundary, the NULL-terminator pushes it into the next 4-byte block and three more NULLs are required to align to the next boundary.

The value of buf_size is the number of bytes in the data array that is used by attribute-value pairs including the text selection and alignment padding. The attributes SELN_REQ_FIRST and SELN_REQ_LAST return the first and last indices into the object in which the selection resides.*

*Currently, the text subwindow is the only XView object that responds to these requests—panel text items do not.

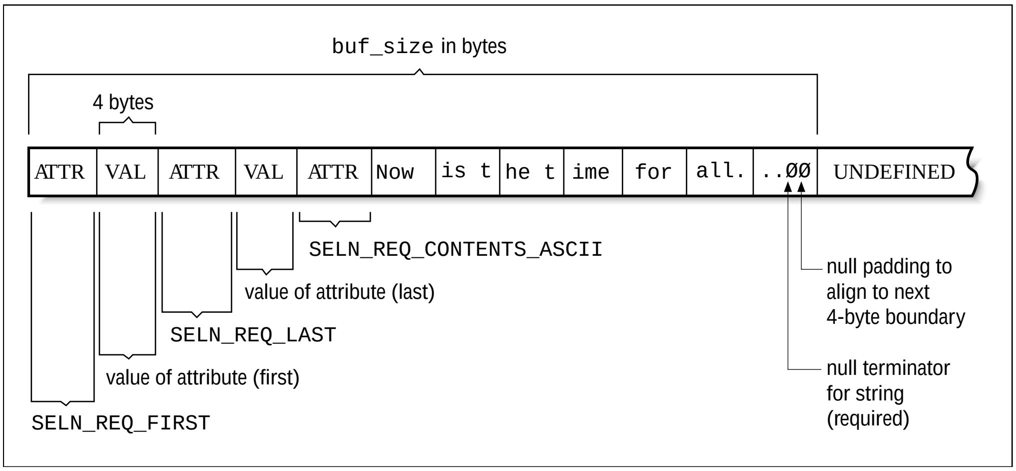

buf_size in bytes
4 bytes
ATTR
VAL
ATTR
VAL
ATTR
Now
is t
he t
ime
for
all.
..ØØ
UNDEFINED
SELN_REQ_CONTENTS_ASCII
value of attribute (last)
SELN_REQ_LAST
value of attribute (first)
SELN_REQ_FIRST
null padding to align to next 4-byte boundary
null terminator for string (required)

Figure A-1. Byte stream after selection_ask() returns the current text selection

To further demonstrate the use of the selection service, we will examine another program (see Example A-2) that is a little more intricate but that still follows the same principles outlined in the first program. The new program, *text_seln.c*, also helps explain some of the new concepts introduced in this section.

text_seln.c contains a text subwindow in which selections can be made. A panel button that prints the current primary selection is also provided. If the selection is made in the text subwindow provided, information is printed about the relationship between the selected text and the rest of the subwindow. The program makes more extensive use of the `selection_ask()` function.

Example A-2. The text_seln.c program

```
/*
 * text_seln.c -- print the primary selection from the server.  If the
 * selection is in a text subwindow, also print information about
 * the line number(s) the selection spans and the indexes of
 * the bytes within the textsw's buffer.
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/textsw.h>
#include <xview/panel.h>
#include <xview/server.h>
#include <xview/seln.h>

Xv_Server       server;
Textsw          textsw;

char *get_selection();

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
```

Example A-2. The text_seln.c program (continued)

```
    Panel       panel;
    void        exit();
    int         print_seln();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);
    panel = (Panel)xv_create(frame, PANEL,
        WIN_WIDTH,              WIN_EXTEND_TO_EDGE,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Get Selection",
        PANEL_NOTIFY_PROC,      print_seln,
        NULL);
    window_fit(panel);

    textsw = (Textsw)xv_create(frame, TEXTSW,
        WIN_X,                  0,
        WIN_BELOW,              panel,
        WIN_ROWS,               10,
        WIN_COLUMNS,            80,
        TEXTSW_FILE_CONTENTS,   "/etc/passwd",
        NULL);
    window_fit(frame);

    server = (Xv_Server)xv_get(xv_get(frame, XV_SCREEN), SCREEN_SERVER);

    xv_main_loop(frame);
}

int
print_seln()
{
    char *text = get_selection();

    if (text)
        printf("---selection---\n%s\n---end seln---\n", text);

    return XV_OK;
}

/*
 * Get the selection using selection_ask().  Note that if the
 * selection is bigger than about 2K, the whole selection will
 * not be gotten with one call, thus this method of getting the
 * selection may not be sufficient.
 */
char *
get_selection()
{
    long                sel_lin_num, lines_selected;
```

Example A-2. The text_seln.c program (continued)

```
    Textsw_index        first, last;
    Seln_holder         holder;
    Seln_result         result;
    int                 len;
    Seln_request        *response;
    static char         selection_buf[BUFSIZ];
    register char       *ptr;

    /* get the holder of the primary selection */
    holder = selection_inquire(server, SELN_PRIMARY);

    /* If the selection occurs in the text subwindow, print lots of
     * info about the selection.
     */
    if (seln_holder_same_client(&holder, textsw)) {
        /* ask for information from the selection service */
        response = selection_ask(server, &holder,
            /* get index of the first and last chars in the textsw */
            SELN_REQ_FIRST,             NULL,
            SELN_REQ_LAST,              NULL,
            /* get the actual selection bytes */
            SELN_REQ_CONTENTS_ASCII,    NULL,
            /* Now fool the textsw to think entire lines are selected */
            SELN_REQ_FAKE_LEVEL,        SELN_LEVEL_LINE,
            /* Get the line numbers of beginning and ending of the
             * selection */
            SELN_REQ_FIRST_UNIT,        NULL,
            SELN_REQ_LAST_UNIT,         NULL,
            NULL);
        /* set the ptr to beginning of data -- SELN_REQ_FIRST */
        ptr = response->data;
        /* "first" is data succeeding SELN_REQ_FIRST -- skip attr */
        first = *(Textsw_index *)(ptr += sizeof(SELN_REQ_FIRST));
        ptr += sizeof(Textsw_index); /* skip over value of "first" */
        /* "last" is data succeeding SELN_REQ_LAST -- skip attr */
        last  = *(Textsw_index *)(ptr += sizeof(SELN_REQ_LAST));
        ptr += sizeof(Textsw_index); /* skip over value of "last" */

        /* advance pointer past SELN_REQ_CONTENTS_ASCII */
        ptr += sizeof(SELN_REQ_CONTENTS_ASCII);
        len = strlen(ptr); /* length of string in response */
        (void) strcpy(selection_buf, ptr);
        /*
         * advance pointer past length of string.  If the string length
         * isn’t aligned to a 4-byte boundary, add the difference in
         * bytes -- then advance pointer passed "value".
         */
        if (len % 4)
            len = len + (4 – (len % 4));
        ptr += len + sizeof(Seln_attribute); /* skip over "value" */

        /* advance pointer past SELN_REQ_FAKE_LEVEL, SELN_LEVEL_LINE */
        ptr += sizeof(SELN_REQ_FAKE_LEVEL) + sizeof(SELN_LEVEL_LINE);

        sel_lin_num = *(long *)(ptr += sizeof(SELN_REQ_FIRST_UNIT));
        ptr += sizeof(long);
```

Example A-2. The text_seln.c program (continued)

```
        lines_selected = *(long *)(ptr += sizeof(SELN_REQ_LAST_UNIT));
        ptr += sizeof(long);

        /* hack to workaround bug with SELN_REQ_LAST_UNIT always
         * returning -1.  Count the lines explicitly in the selection.
         */
        if (lines_selected < 0) {
            register char *p;
            lines_selected++;
            for (p = selection_buf; *p; p++)
                if (*p == '\n')
                    lines_selected++;
        }
        printf("index in textsw: %d–%d, line number(s) = %d–%d\n",
            first+1, last+1, sel_lin_num+1,
            sel_lin_num + lines_selected + 1);
    } else {
        /* the selection does not lie in our text subwindow */
        response = selection_ask(server, &holder,
            SELN_REQ_CONTENTS_ASCII, NULL,
            NULL);
        if (response->status != SELN_SUCCESS) {
            printf("selection_ask() returns %d\n", response–>status);
            return NULL;
        }
        (void) strcpy(selection_buf,
            response->data + sizeof(SELN_REQ_CONTENTS_ASCII));
    }
    return selection_buf;
}
```

There are several points of interest here. In the function `get_selection()`, once the holder of the client has been obtained, it is tested to see if the holder is the text subwindow using `seln_holder_same_client()`. If so, `selection_ask()` is called requesting information specific to the text subwindow. If the text subwindow is not the holder of the selection, then `selection_ask()` is called requesting only the ASCII contents. If there is no selection, then the `status` field of the structure is set to `SELN_FAILED`.

In the case where the holder is the text subwindow, we ask it for the first and last indices of the selection relative to the beginning of the text stream. Note that we might not be able to request this information from any object. For example, if the selection were inside an *xterm*, then this information would not be available and the *xterm*'s selection client would not respond to such requests.

The next attribute (`SELN_REQ_CONTENTS_ASCII`) requests the ASCII contents of the selection actually made. Following that, the attribute-value pair:

```
SELN_REQ_FAKE_LEVEL, SELN_LEVEL_LINE
```

fools the text subwindow into thinking that the user selected an entire line of text (in OPEN LOOK, this would have meant a triple-click with the SELECT mouse button). Had this attribute-value pair been listed *before* the request for ASCII contents, the text returned by the request would have contained the entire line of text on which the selection occurred *regardless of whether the selection began at the beginning of the line*.

The reason we fake the text window into thinking the entire line has been selected is: the attributes SELN_REQ_FIRST_UNIT and SELN_REQ_LAST_UNIT request the line numbers that the selection spans. As the names of the attributes imply, the request is for the first and last *units* selected. Setting the SELN_REQ_FAKE_LEVEL attribute to SELN_LEVEL_LINE indicates that the unit type should be line. Note that we fake the fact that the selection unit is set to line just to get the start and end line numbers of the selection. If we wanted to actually set the level, we would have used SELN_REQ_SET_LEVEL.

After selection_ask() returns a pointer to a Seln_request structure, the values of the requested attributes are found in data, the byte stream. As demonstrated in Example A-2 above, the way to retrieve these values is by moving a pointer along the array:

```
Seln_request *response;
char         *ptr;
long          value;
...
response = selection_ask(server, &holder,
    ATTR1,      NULL,
    ATTR2,      NULL,
    ...
    NULL);
...
/* set the ptr to beginning of data response -- first attribute */
ptr = response->data;
/* value is data succeeding first attribute -- skip over attr */
value = *(long *)(ptr += sizeof(Seln_attribute));
ptr += sizeof(long); /* skip over the size of the type of value */
```

There is no need to test the attributes as you scan data; they are the same attributes that you used in selection_ask() and they remain in the same order. The values you get back are almost always long.* The exception to this is the text string returned when SELN_REQ_CONTENTS_ASCII is specified. However, the text string is padded to a 4-byte boundary to make sure that the alignment is correct. *text_seln.c* demonstrates how this is done.

A.4 Using selection_query()

One problem with using selection_ask() is handling large selections. In this context, “large” means a text string that is long enough so that it, along with all its attributes and values, does not fit in the data byte-stream. Of course, the fewer attributes that are requested, the more text is returned from the selection.

In this case, the problem is that there is an upper limit to the number of bytes that can be retrieved from the selection. There is no guarantee that the user is not going to select a large number of bytes from some arbitrary application on the screen. However, there is another way to get the selection, regardless of how large it is, by using selection_query():

*Architectures whose int type does not equal its long type must be sure to compensate for this.

```
Seln_result
selection_query(server, holder, reader, context, attrs)
    Xv_Server     server;
    Seln_holder   *holder;
    Seln_result   (*reader)();
    char          *context;
    <attribute-value list> attrs
```

The primary feature of this routine is that you provide it with a pointer to a function that does the scanning of the `data` array in the `Seln_request` structure, as demonstrated earlier. Your `reader` function is called by `selection_query()`, and it gets the `Seln_request` structure as the sole parameter to your function. Your function takes the form of:

```
Seln_result
reader(request)
    Seln_request *request;
```

Your function should return `SELN_SUCCESS` provided that you encountered no problems with scanning `request->data`. `selection_query()` returns the same `Seln_result` that your `reader` function returns. Your `reader` function is called by `selection_query()` for each *chunk* of data in the selection.* The flowchart in Figure A-2 shows the sequence of operations.

The program in Example A-3 demonstrates the use of `selection_query()`. It is similar to *text_seln.c*, but this new program also provides for selections from one of three selection ranks. The user chooses the selection rank from the panel choice item. When the Get Selection button is pressed, the current selection from that selection rank is displayed.

The point of the program is to demonstrate the flow of control between `selection_query()` and the client-installed `reader` procedure. The text subwindow in the application loads the file */etc/termcap*. When a selection is made, `selection_query()` is called, which in turn calls the reader procedure. You can make an arbitrarily large selection to show how `read_proc` is called many times. Start by initializing the selection and then *scrolling* the window and *extending* the selection by using the ADJUST mouse button on later text. Select the Get Selection panel button and the output is directed to `stdout`. Because the selection size can be large, the output text is truncated to the first 20 characters of the selection.

Example A-3. The long_seln.c program

```
/*
 * long_seln.c shows how to get an arbitrarily large selection by
 * providing a reading procedure to selection_query().  The panel
 * items allow the user to choose between 3 selection ranks.
 */
#include <xview/xview.h>
#include <xview/textsw.h>
#include <xview/panel.h>
#include <xview/seln.h>
```

*A *chunk* is the largest text string that will fit in the `data` field of the `Seln_request` structure.

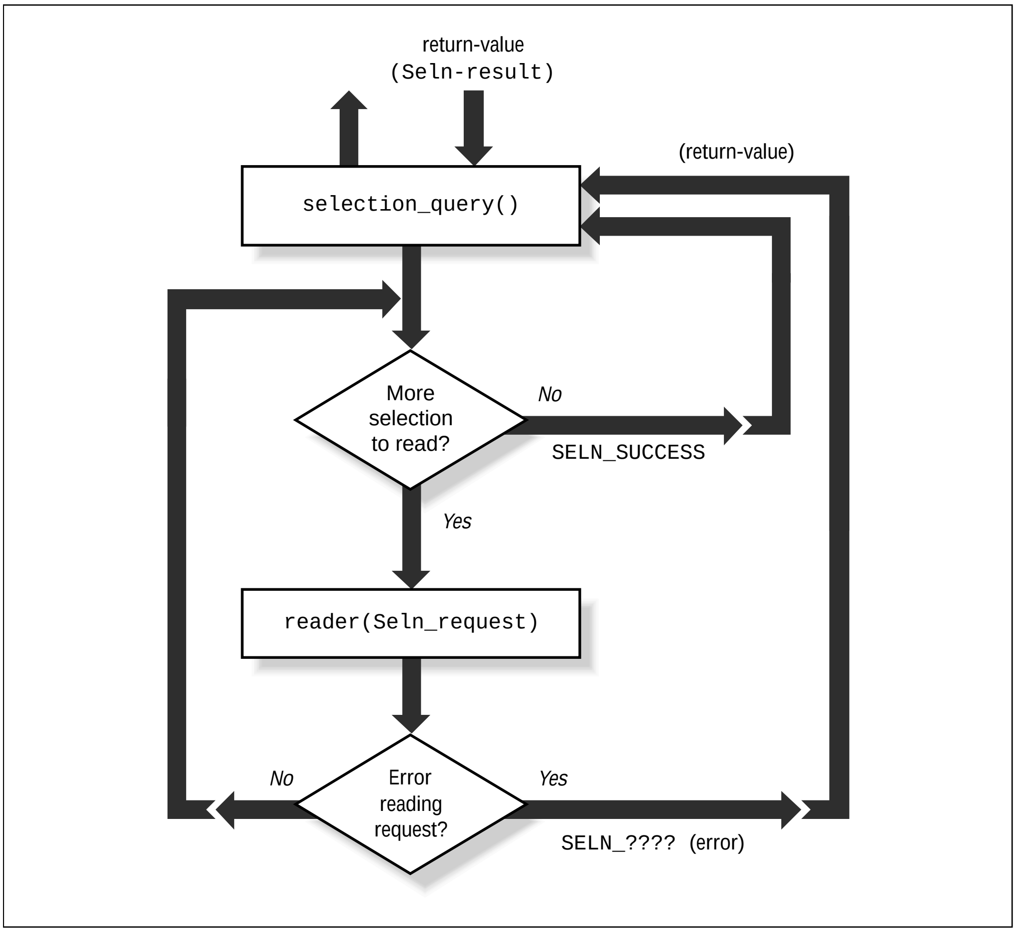


Figure A-2. How selection_query() is used

Example A-3. The long_seln.c program (continued)

```
extern char *malloc();

Seln_rank seln_type = SELN_PRIMARY;

#define FIRST_BUFFER            0
#define NOT_FIRST_BUFFER        !FIRST_BUFFER

char *seln_bufs[3];     /* contents of each of the three selections */

Seln_result read_proc(); /* supplied to selection_query() as reader */

Textsw          textsw;  /* select from this textsw */
Xv_Server       server;
char *get_selection();

void
change_selection(item, value)
```

Example A-3. The long_seln.c program (continued)

```
Panel_item item;
int value;
{
    if (value == 0)
        seln_type = SELN_PRIMARY;
    else if (value == 1)
        seln_type = SELN_SECONDARY;
    else
        seln_type = SELN_SHELF;
}

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
    Panel       panel;
    void        print_seln(), exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    frame = (Frame) xv_create(NULL, FRAME,
        FRAME_LABEL, argv[0],
        NULL);

    panel = (Panel)xv_create(frame, PANEL,
        WIN_WIDTH,              WIN_EXTEND_TO_EDGE,
        NULL);

    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Get Selection",
        PANEL_NOTIFY_PROC,      print_seln,
        NULL);
    (void) xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING,     "Selection Type",
        PANEL_CHOICE_STRINGS,   "Primary", "Secondary", "Shelf", NULL,
        PANEL_NOTIFY_PROC,      change_selection,
        NULL);
    window_fit(panel);

    textsw = (Textsw)xv_create(frame, TEXTSW,
        WIN_X,                  0,
        WIN_BELOW,              panel,
        WIN_ROWS,               10,
        WIN_COLUMNS,            80,
        TEXTSW_FILE_CONTENTS,   "/etc/termcap",
        NULL);
    window_fit(frame);
    server = (Xv_Server)xv_get(xv_get(frame, XV_SCREEN), SCREEN_SERVER);
    xv_main_loop(frame);
}

void
print_seln()
```

Example A-3. The long_seln.c program (continued)

```
{
    char *text = get_selection();

    if (text)
        printf("---seln---\n%.*s [...]\n---end seln---\n", 20, text);
}

/*
 * return the text selected in the current selection rank.  Use
 * selection_query() to guarantee that the entire selection is
 * retrieved.  selection_query() calls our installed routine,
 * read_proc() (see below).
 */
char *
get_selection()
{
    Seln_holder   holder;
    Seln_result   result;
    Seln_request  *response;
    char          context = FIRST_BUFFER;

    holder = selection_inquire(server, seln_type);
    printf("selection type = %s\n",
        seln_type == SELN_PRIMARY? "primary" :
        seln_type == SELN_SECONDARY? "secondary" : "shelf");

    /* result is based on the return value of read_proc() */
    result = selection_query(server, &holder, read_proc, &context,
        SELN_REQ_BYTESIZE,              NULL,
        SELN_REQ_CONTENTS_ASCII,        NULL,
        NULL);
    if (result == SELN_FAILED) {
        puts("couldn't get selection");
        return NULL;
    }

    return seln_bufs[seln_type];
}

/*
 * Called by selection_query for every buffer of information received.
 * Short messages (under about 2000 bytes) will fit into one buffer.
 * For larger messages, read_proc is called for each buffer in the
 * selection.  The context pointer passed to selection_query is
 * modified by read_proc so that we know if this is the first buffer
 * or not.
 */
Seln_result
read_proc(response)
Seln_request *response;
{
    char *reply;  /* pointer to the data in the response received */
    long seln_len; /* total number of bytes in the selection */
    static long seln_have_bytes;
        /* number of bytes of the selection
         * which have been read; cumulative over all calls for
```

Example A-3. The long_seln.c program (continued)

```
         * the same selection (it is reset when the first
         * response of a selection is read)
         */

    printf("read_proc status: %s (%d)\n",
        response->status == SELN_FAILED? "failed" :
        response->status == SELN_SUCCESS? "succeeded" :
        response->status == SELN_CONTINUED? "continued" : "???",
        response->status);
    if (*response->requester.context == FIRST_BUFFER) {
        reply = response->data;

        /* read in the length of the selection -- first attribute.
         * advance "reply" passed attribute to point to actual data.
         */
        reply += sizeof(SELN_REQ_BYTESIZE);
        /* set seln_len to actual data now. (bytes selected) */
        seln_len = *(int *)reply;
        printf("selection size is %ld bytes\n", seln_len);
        /* advance "reply" to next attribute in list */
        reply += sizeof(long);

        /* create a buffer large enough to store entire selection */
        if (seln_bufs[seln_type] != NULL)
            free(seln_bufs[seln_type]);
        if (!(seln_bufs[seln_type] = malloc(seln_len + 1))) {
            puts("out of memory");
            return(SELN_FAILED);
        }
        seln_have_bytes = 0;

        /* move "reply" passed attribute so it points to contents */
        reply += sizeof(SELN_REQ_CONTENTS_ASCII);
        *response->requester.context = NOT_FIRST_BUFFER;
    } else {
        /* this is not the first buffer, so the contents of the
         * response is just more of the selection
         */
        reply = response->data;
    }

    /* copy data from received to the seln buffer allocated above */
    (void) strcpy(&seln_bufs[seln_type][seln_have_bytes], reply);
    seln_have_bytes += strlen(reply);

    return SELN_SUCCESS;
}
```

A.5 Selection Package Summary

Table A-2 lists the procedures and macros in the Selection Service. Table A-3 lists the Selection Service Attributes.

Table A-2. Selection Service Procedures

Selection Procedures	
`selection_acquire()`	`selection_inform()`
`selection_ask()`	`selection_init_request()`
`selection_clear_functions()`	`selection_inquire()`
`selection_create()`	`selection_inquire_all()`
`selection_destroy()`	`selection_query()`
`selection_done()`	`selection_report_event()`
`selection_figure_response()`	`selection_request()`
`selection_hold_file()`	`selection_yield_all()`

Table A-3. Selection Service Attributes

Selection Attributes	Advanced Selection Attributes
SELN_REQ_BYTESIZE	SELN_REQ_COMMIT_PENDING_DELETE
SELN_REQ_CONTENTS_ASCII	SELN_REQ_CONTENTS_PIECES
SELN_REQ_DELETE	SELN_REQ_FAKE_LEVEL
SELN_REQ_END_REQUEST	SELN_REQ_FIRST
SELN_REQ_FILE_NAME	SELN_REQ_FIRST_UNIT
SELN_REQ_YIELD	SELN_REQ_LAST
	SELN_REQ_LAST_UNIT
	SELN_REQ_LEVEL
	SELN_REQ_RESTORE
	SELN_REQ_SET_LEVEL

This page intentionally left blank

to preserve original page counts.

B
The notice_prompt Function

This section describes the XView compatibility procedure `notice_prompt()`. If you need to create a new notice, use the `NOTICE` package described in Chapter 12, *Notices*. The information in this chapter describes the old notice interface that is supported for compatibility with older XView versions.

A notice is a pop-up window that notifies the user of a problem or asks a question that requires an immediate response. The notice grabs the entire screen so no other windows or applications can receive input until the user responds to the notice.

Notices are implemented using the `FULLSCREEN` package to grab the keyboard and pointer events from the server. (The `FULLSCREEN` package is described in Chapter 15, *Nonvisual Objects*.) The notice window, which owns the fullscreen object, is a nonrectangular transient X window with the X-window attribute `override_redirect`* set. When the notice is created, the notice window is immediately displayed. When the user responds to one of the available choices, the notice session ends.

B.1 Creating and Displaying Notices

To use the `NOTICE` package in applications, the header file *<xview/notice.h>* must be included. Notices are special XView objects because they are not created via `xv_create()`. Also, they cannot be modified using `xv_set()`. Notices are created using the special procedure `notice_prompt()`:

```
int
notice_prompt(owner, event, attrs)
    Xv_Window   owner;
    Event      *event;
    attributes ...
```

When creating a notice, the owner must be a valid XView object that has a window associated with it. This can be a panel or a frame, but it is typically the window that causes the notice to be created. If the user tries to type in a read-only text subwindow, a notice might appear from that window informing the user of the error. The `event` might be `NULL` if you are not using `NOTICE_TRIGGER` (see Section 12.1.2, "Notice Triggers").

*`override_redirect` tells the window manager to not provide window decorations.

Because the notice window is not a part of any other XView package and it does not allow window-specific attributes, you cannot use any generic, common or window attributes to configure the notice window; you can only use `NOTICE_*` attributes.

Notice windows are explicitly specified by OPEN LOOK and cannot be modified. If you wish to create a notice-type interface that is not OPEN LOOK compliant (which is not recommended), you need to learn more about the fullscreen object described in Chapter 15, *Non-visual Objects*.

Your application has control over the messages that are displayed in the notice window as well as the choices available to the user as responses. `notice_prompt()` creates a window, grabs the server, waits for the user to make a selection on one of the available button choices, then destroys the window. You never have a handle to the notice *object* itself—only the resulting choice made by the user. The result is the return value of the `notice_prompt()` function.

A very simple case of a notice prompt is demonstrated in Example B-1.

Example B-1. The simple_notice.c program

```
/*
 * simple_notice.c -- Demonstrate the use of notices.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Xv_opaque   my_notify_proc();

    /*
     * Initialize XView, create a frame, a panel and one panel button.
     */
    xv_init(XV_INIT_ARGS, argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        NULL);

    /* make sure everything looks good */
    window_fit(panel);
    window_fit(frame);

    /* start window event processing */
    xv_main_loop(frame);
}

/*
```

Example B-1. The simple_notice.c program (continued)

```
 * my_notify_proc() -- called when the user selects the Quit button.
 *      The notice appears as a result of notice_prompt().  Here
 *      the user must chooses YES or NO to confirm or deny quitting.
 */
Xv_opaque
my_notify_proc(item, event)
Panel_item  item;
Event      *event;
{
    int        result;

    result = notice_prompt(panel, NULL,
        NOTICE_FOCUS_XY,        event_x(event), event_y(event),
        NOTICE_MESSAGE_STRINGS, "Do you really want to quit?", NULL,
        NOTICE_BUTTON_YES,      "Yes",
        NOTICE_BUTTON_NO,       "No",
        NULL);

    if (result == NOTICE_YES)
        exit(0);
}
```

The program *simple_notice.c* contains a panel with a Quit button. When the user selects the Quit button, a notice pops up to prompt the user for confirmation. What the user sees is shown in Figure B-1. If the user presses "Yes," the program exits.

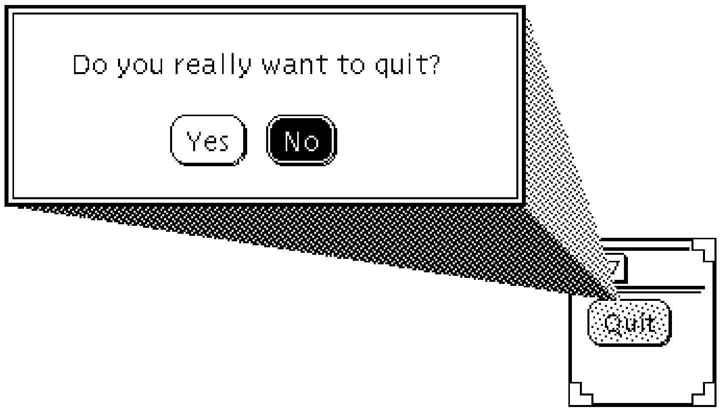


Figure B-1. Output of simple_notice.c while the notice is up

The position from which the notice shadow emanates is described by the attribute NOTICE_FOCUS_XY. This value defaults to the current mouse position when the application calls `notice_prompt()`. As shown in *simple_notice.c*, the point from which the notice shadow emanates appears to be the same position as the location where the panel button was selected. Due to possible delays with the X server, by the time the `notice_prompt()` routine gets called, the location of the mouse may have moved from the place where the panel button was selected. To be sure that the notice prompt appears to emanate from the original mouse-down location, we use the coordinates of the mouse position from the `event` structure. The values for `NOTICE_FOCUS_XY` are relative to the origin of the window passed as the first parameter to `notice_prompt()`.

B.1.1 Response Choices and Values

Two responses are normally present whenever a notice appears: "Yes" and "No." These are defined for convenience in *<xview/notice.h>*:

```
#define NOTICE_YES  1
#define NOTICE_NO   0
```

These are the return values that `notice_prompt()` might return that correspond directly to the attributes `NOTICE_BUTTON_YES` and `NOTICE_BUTTON_NO`. As shown in *simple_notice.c*, these are the only two choices made available to the user.

These two choices are special in another way: they respond to *accelerator* keys. That is, the RETURN key can be used instead of selecting the `NOTICE_BUTTON_YES` button with the pointer, and the STOP key can be used instead of selecting `NOTICE_BUTTON_NO`.

Also, when these choices are used, the cursor is immediately bound to the button associated with `NOTICE_BUTTON_YES` because this is the *default* response to the notice. As a hint to the programmer, it is always desirable to word all questions so the default answer is "Yes."

It is quite common for the application to have more than one appropriate response to some kind of notice prompt. Suppose that your application is an editor of some kind. If the user selects the Quit button and there have been changes to the file that have not been accounted for, you might wish to inform the user and allow more than one response: quit, updating changes; quit, ignoring changes; or cancel the quit all together. To implement these new choices, use the `NOTICE_BUTTON` attribute to define the choices available:

```
result = notice_prompt(panel, NULL,
    NOTICE_MESSAGE_STRINGS,
        "There have been modifications since your last update",
        "Would you like to quit or continue editing?",
        NULL,
    NOTICE_BUTTON,    "Quit, Update changes",    101,
    NOTICE_BUTTON,    "Quit, Ignore changes",    102,
    NOTICE_BUTTON,    "Continue Editing",        103,
    NULL);
```

The `NOTICE_BUTTON` attribute takes two parameters: the button label* and the return value if that button is selected. In this case, the possible return values for the call to `notice_prompt` are 101, 102 and 103 (in addition to possible error return values). The application should make its decision on how to proceed based on the return value.

*The button can display text only; no graphic images can be displayed.

Because the NOTICE_BUTTON attribute is used, there is no default choice and no accelerators associated with the notice; the user must use the pointer to select one of the available choices.

B.1.2 Notice Triggers

If you want to assign accelerators to notice buttons, or if you find it necessary to give the user the choice of using mouse buttons or keyboard events to respond to a notice, you can identify *triggers* that cause the notice to return. The value returned in this case is NOTICE_TRIGGERED, and the event that caused the trigger will be in the Event * passed in the call to notice_prompt(). When triggers are not used, the Event * can be NULL.

Example B-2 shows how one might use the NOTICE_TRIGGER to get a particular event:

Example B-2. The trigger_notice.c program

```
/*
 * trigger_notice.c -- Demonstrate the use of triggers in notices.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

main(argc,argv)
int     argc;
char    *argv[];
{
    Frame       frame;
    Panel       panel;
    Xv_opaque   my_notify_proc();
    extern void exit();

    /*
     * Initialize XView, create a frame, a panel and one panel button.
     */
    xv_init(XV_INIT_ARGS, argc, argv, NULL);
    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Move",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        NULL);

    /* make sure everything looks good */
    window_fit(panel);
    window_fit(frame);

    /* start window event processing */
    xv_main_loop(frame);
}
```

Example B-2. The trigger_notice.c program (continued)

```
/*
 * my_notify_proc() -- called when the user selects the "Move"
 * panel button.  Put up a notice_prompt to get new coordinates
 * to move the main window.
 */
Xv_opaque
my_notify_proc(item, event)
Panel_item  item;
Event       *event;
{
    int         result, x, y;
    Panel       panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);
    Frame       frame = (Frame)xv_get(panel, XV_OWNER);

    x = event_x(event), y = event_y(event);
    printf("original click relative to panel: %d, %d0, x, y);
    result = notice_prompt(panel, event,
        NOTICE_FOCUS_XY,        x, y,
        NOTICE_MESSAGE_STRINGS,
            "You may move the window to a new location specified by",
            "clicking the Left Mouse Button somewhere on the screen",
            "or cancel this operation by selecting
            NULL,
        NOTICE_BUTTON_YES,      "cancel",
        NOTICE_TRIGGER,         MS_LEFT,
        NOTICE_NO_BEEPING,      TRUE,
        NULL);

    if (result == NOTICE_TRIGGERED) {
        x = event_x(event) + (int)xv_get(frame, XV_X);
        y = event_y(event) + (int)xv_get(frame, XV_Y);
        printf("screen x,y: %d, %d0, x, y);
        xv_set(frame, XV_X, x, XV_Y, y, NULL);
    }
}
```

When this program is run and the user selects the Move panel button, a notice is displayed instructing the user to select a new position for the application window. When the user selects a new location, the window frame moves to that position. Note that the window manager adds a title bar and other decorations around the frame; do not expect the upper-left corner of the frame to move to the new position. The real frame's origin is moved to the new position, and the frame's decorations are moved as well but not aligned to the same values (it will be somewhat higher).

When `notice_prompt()` returns, the `Event` structure that was passed to it contains the event that triggered the notice to return. The *x* and *y* coordinates in the `Event` structure are relative to the origin of the notice owner window.

To translate these coordinates to screen-specific coordinates, save the original event location and add to that the (*x*, *y*) coordinates returned when `notice_prompt()` returns as well as the current coordinates of the frame (main application).

Before leaving *trigger_notice.c*, we should mention the attribute NOTICE_NO_BEEPING that is used to prevent the notice from beeping when it is displayed. Beeping the screen is usually done when there is an error condition you wish to alert the user about. In this example, there is no error condition—it is a simple dialog with the user.

B.2 Another Example

In the previous example, we used many of the attributes covered in this section in addition to using some generic and common attributes for the panel items. Example B-3 goes a little further to demonstrate how the NOTICE package works in conjunction with the rest of XView. It creates a frame, a panel with two panel buttons and a message item. Initially, only the Quit button and the Commit button are displayed. When the user selects either button, a notice pops up asking the user to confirm or cancel the proposed action. If the user confirms quitting the program, the program quits. Otherwise, the result, either Confirmed or Canceled, is displayed as the text of the message item.

Example B-3. The notice.c program

```
/*
 * notice.c --
 * This application creates a frame, a panel, and 3 panel buttons.
 * A message button, a Quit button (to exit the program) and a
 * dummy "commit" button.  Extra data is attached to the panel
 * items by the use of XV_KEY_DATA.  The callback routine for the
 * Quit and Commit buttons is generalized enough that it can apply
 * to either button (or any arbitrary button) because it extracts
 * the expected "data" (via XV_KEY_DATA) from whatever panel
 * button might have called it.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

/*
 * assign "data" to panel items using XV_KEY_DATA ... attach the
 * message panel item, a prompt string specific for the panel
 * item’s notice_prompt, and a callback function if the user
 * chooses "yes".
 */
#define MSG_ITEM        10 /* any arbitrary integer */
#define NOTICE_PROMPT   11
#define CALLBACK_FUNC   12

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Panel_item  msg_item;
    Xv_opaque   my_notify_proc();
    extern int  exit();
    /*
     * Initialize XView, and create frame, panel and buttons.
     */
    xv_init(XV_INIT_ARGS, argc, argv, NULL);
    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    msg_item = (Panel_item)xv_create(panel, PANEL_MESSAGE, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        XV_KEY_DATA,            MSG_ITEM,       msg_item,
        /*
         * attach a prompt specific for this button used by
         * notice_prompt()
         */
        XV_KEY_DATA,            NOTICE_PROMPT,  "Really Quit?",
        /*
         * a callback function to call if the user answers "yes"
         * to prompt
         */
        XV_KEY_DATA,            CALLBACK_FUNC,  exit,
        NULL);
    /*
     * now that the Quit button is under the message item,
     * layout horizontally
     */
    xv_set(panel, PANEL_LAYOUT, PANEL_HORIZONTAL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Commit...",
        PANEL_NOTIFY_PROC,      my_notify_proc,
        XV_KEY_DATA,            MSG_ITEM,       msg_item,
        /*
         * attach a prompt specific for this button used by
         * notice_prompt()
         */
        XV_KEY_DATA,            NOTICE_PROMPT,  "Update all changes?",
        /*
         * Note there is no callback func here, but one could be
         * written
         */
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * my_notify_proc()
 * The notice appears as a result of notice_prompt().
 * The "key data" associated with the panel item is extracted via
```

Example B-3. The notice.c program (continued)

```
 * xv_get().  The resulting choice is displayed in the panel
 * message item.
 */
Xv_opaque
my_notify_proc(item, event)
Panel_item  item;
Event      *event;
{
    int          result;
    int        (*func)();
    char        *prompt;
    Panel_item   msg_item;
    Panel        panel;

    func = (int(*)())xv_get(item, XV_KEY_DATA, CALLBACK_FUNC);
    prompt = (char *)xv_get(item, XV_KEY_DATA, NOTICE_PROMPT);
    msg_item = (Panel_item)xv_get(item, XV_KEY_DATA, MSG_ITEM);
    panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);
    /*
     *  Create the notice and get a response.
     */
    result = notice_prompt(panel, NULL,
        NOTICE_FOCUS_XY,        event_x(event), event_y(event),
        NOTICE_MESSAGE_STRINGS,
                prompt,
                "Press YES to confirm",
                "Press NO to cancel",
                NULL,
        NOTICE_BUTTON_YES,      "YES",
        NOTICE_BUTTON_NO,       "NO",
        NULL);

    switch(result) {
        case NOTICE_YES:
            xv_set(msg_item, PANEL_LABEL_STRING, "Confirmed", NULL);
            if (func)
                (*func)();
            break;
        case NOTICE_NO:
            xv_set(msg_item, PANEL_LABEL_STRING, "Cancelled", NULL);
            break;
        case NOTICE_FAILED:
            xv_set(msg_item, PANEL_LABEL_STRING, "unable to pop-up",
              NULL);
            break;
        default:
            xv_set(msg_item, PANEL_LABEL_STRING, "unknown choice",
              NULL);
    }
}
```

B.3 Notice Package Summary

Table B-1 lists the attributes, procedures and macros for the NOTICE package. This information is described fully in the appendices of the *XView Reference Manual*.

Table B-1. Notice Attributes, Procedures, and Macros

Attributes	Procedures and Macros
NOTICE_BUTTON	notice_prompt()
NOTICE_BUTTON_NO	
NOTICE_BUTTON_YES	
NOTICE_FOCUS_XY	
NOTICE_FONT	
NOTICE_MESSAGE_STRINGS	
NOTICE_MESSAGE_STRINGS_ARRAY_PTR	
NOTICE_NO_BEEPING	
NOTICE_TRIGGER	

C
Mouseless Model Keyboard Mappings

This appendix lists information for the Mouseless Model, including the following:

- Resource mappings for the Mouseless Model key bindings.
- Semantic actions for the Mouseless Model.
- SunView1 to Mouseless Model keyboard command mappings.

C.1 Mouseless Model Resources

Each action's mapping is determined by the value of a resource. The name of the resource is:

```
OpenWindows.KeyboardCommand.XViewSemanticAction
```

XViewSemanticAction is the name of the XView semantic action, without the `ACTION_` prefix. For the resource names, the underscore naming paradigm is changed to the capitalized paradigm (see example). Each value for a resource has the form:

```
mapping[,mapping...]
```

and each mapping is of the form:

KeysymName[+*Modifier* . . .]

In other words, each mapping is separated by a comma, and if the keysym is modified, then each modifier is separated by a plus sign. A modifier is either "Shift," "Ctrl," "Alt," or "Meta." Note that when giving alphabetic characters as keysyms, the case of the KeysymName is pertinent. For uppercase characters, use the uppercase alpha keysym, for example "L", instead of the lowercase with a "Shift" modifier. When an alphabetic character is not modified by shift, then use the lowercase alpha keysym (e.g., l+Meta). Unmodified ASCII keyboard commands should not be listed.

C.1.1 SunView1 Mappings

The following keyboard mappings are always loaded regardless of the setting of `OpenWindows.KeyboardCommands`.

C.1.1.1 Keyboard core functions

```
OpenWindows.KeyboardCommand.Stop: L1
OpenWindows.KeyboardCommand.Again: a+Meta,a+Ctrl+Meta,L2
OpenWindows.KeyboardCommand.Props: L3
OpenWindows.KeyboardCommand.Undo: u+Meta,L4
OpenWindows.KeyboardCommand.Copy: c+Meta,L6
OpenWindows.KeyboardCommand.Paste: v+Meta,L8
OpenWindows.KeyboardCommand.FindForward: f+Meta,L9
OpenWindows.KeyboardCommand.FindBackward: F+Meta,L9+Shift
OpenWindows.KeyboardCommand.Cut: x+Meta,L10
OpenWindows.KeyboardCommand.Help: Help
OpenWindows.KeyboardCommand.MoreHelp: Help+Shift
OpenWindows.KeyboardCommand.TextHelp: Help+Ctrl
OpenWindows.KeyboardCommand.MoreTextHelp: Help+Shift+Ctrl
OpenWindows.KeyboardCommand.DefaultAction: Return+Meta
OpenWindows.KeyboardCommand.CopyThenPaste: p+Meta
OpenWindows.KeyboardCommand.Translate: R2
```

C.1.1.2 Local navigation commands

```
OpenWindows.KeyboardCommand.Up: p+Ctrl,N+Ctrl,Up,R8,Up+Shift
OpenWindows.KeyboardCommand.Down: n+Ctrl,P+Ctrl,Down,R14,Down+Shift
OpenWindows.KeyboardCommand.Left: b+Ctrl,F+Ctrl,Left,R10,Left+Shift
OpenWindows.KeyboardCommand.Right: f+Ctrl,B+Ctrl,Right,R12,Right+Shift
OpenWindows.KeyboardCommand.JumpLeft: comma+Ctrl,greater+Ctrl
OpenWindows.KeyboardCommand.JumpRight: period+Ctrl
OpenWindows.KeyboardCommand.GoPageBackward: R9
OpenWindows.KeyboardCommand.GoPageForward: R15
OpenWindows.KeyboardCommand.GoWordForward: slash+Ctrl,less+Ctrl
OpenWindows.KeyboardCommand.LineStart: a+Ctrl,E+Ctrl
OpenWindows.KeyboardCommand.LineEnd: e+Ctrl,A+Ctrl
OpenWindows.KeyboardCommand.GoLineForward: apostrophe+Ctrl,R11
OpenWindows.KeyboardCommand.DataStart: Home,R7,Return+Shift+Ctrl,Home+Shift
OpenWindows.KeyboardCommand.DataEnd: End,R13,Return+Ctrl,End+Shift
```

C.1.1.3 Text editing commands

```
OpenWindows.KeyboardCommand.SelectFieldForward: Tab+Ctrl
OpenWindows.KeyboardCommand.SelectFieldBackward: Tab+Shift+Ctrl
OpenWindows.KeyboardCommand.EraseCharBackward: Delete,BackSpace
OpenWindows.KeyboardCommand.EraseCharForward: Delete+Shift,BackSpace+Shift
OpenWindows.KeyboardCommand.EraseWordBackward: w+Ctrl
OpenWindows.KeyboardCommand.EraseWordForward: W+Ctrl
OpenWindows.KeyboardCommand.EraseLineBackward: u+Ctrl
OpenWindows.KeyboardCommand.EraseLineEnd: U+Ctrl
OpenWindows.KeyboardCommand.MatchDelimiter: d+Meta
OpenWindows.KeyboardCommand.Empty: e+Meta,e+Ctrl+Meta
OpenWindows.KeyboardCommand.IncludeFile: i+Meta
OpenWindows.KeyboardCommand.Insert: Insert
OpenWindows.KeyboardCommand.Load: l+Meta
OpenWindows.KeyboardCommand.Store: s+Meta
```

C.1.2 Basic Mappings

When the `OpenWindows.KeyboardCommands` resource is set to Basic *or* Full, the following keyboard mappings are loaded.

C.1.2.1 Local navigation commands

```
OpenWindows.KeyboardCommand.Up: Up
OpenWindows.KeyboardCommand.Down: Down
OpenWindows.KeyboardCommand.Left: Left
OpenWindows.KeyboardCommand.Right: Right
OpenWindows.KeyboardCommand.JumpUp: Up+Ctrl
OpenWindows.KeyboardCommand.JumpDown: Down+Ctrl
OpenWindows.KeyboardCommand.JumpLeft: Left+Ctrl
OpenWindows.KeyboardCommand.JumpRight: Right+Ctrl
OpenWindows.KeyboardCommand.PaneUp: R9
OpenWindows.KeyboardCommand.PaneDown: R15
OpenWindows.KeyboardCommand.PaneLeft: R9+Ctrl
OpenWindows.KeyboardCommand.PaneRight: R15+Ctrl
OpenWindows.KeyboardCommand.RowStart: Home,R7
OpenWindows.KeyboardCommand.RowEnd: End,R13
OpenWindows.KeyboardCommand.DataStart: Home+Ctrl,R7+Ctrl
OpenWindows.KeyboardCommand.DataEnd: End+Ctrl,R13+Ctrl
```

C.1.2.2 Text editing commands

```
OpenWindows.KeyboardCommand.SelectUp: Up+Shift
OpenWindows.KeyboardCommand.SelectDown: Down+Shift
OpenWindows.KeyboardCommand.SelectLeft: Left+Shift
OpenWindows.KeyboardCommand.SelectRight: Right+Shift
OpenWindows.KeyboardCommand.SelectJumpUp: Up+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectJumpDown: Down+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectJumpLeft: Left+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectJumpRight: Right+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectRowStart: Home+Shift,R7+Shift
OpenWindows.KeyboardCommand.SelectRowEnd: End+Shift,R13+Shift
OpenWindows.KeyboardCommand.SelectPaneUp: R9+Shift
OpenWindows.KeyboardCommand.SelectPaneDown: R15+Shift
OpenWindows.KeyboardCommand.SelectPaneLeft: R9+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectPaneRight: R15+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectDataStart: Home+Shift+Ctrl,R7+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectDataEnd: End+Shift+Ctrl,R13+Shift+Ctrl
OpenWindows.KeyboardCommand.SelectAll: End+Shift+Meta
OpenWindows.KeyboardCommand.SelectNextField: Tab+Meta
OpenWindows.KeyboardCommand.SelectPreviousField: Tab+Shift+Meta
OpenWindows.KeyboardCommand.ScrollUp: Up+Alt
OpenWindows.KeyboardCommand.ScrollDown: Down+Alt
OpenWindows.KeyboardCommand.ScrollLeft: Left+Alt
OpenWindows.KeyboardCommand.ScrollRight: Right+Alt
OpenWindows.KeyboardCommand.ScrollJumpUp: Up+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollJumpDown: Down+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollJumpLeft: Left+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollJumpRight: Right+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollRowStart: Home+Alt,R7+Alt
OpenWindows.KeyboardCommand.ScrollRowEnd: End+Alt,R13+Alt
OpenWindows.KeyboardCommand.ScrollPaneUp: R9+Alt
OpenWindows.KeyboardCommand.ScrollPaneDown: R15+Alt
OpenWindows.KeyboardCommand.ScrollPaneLeft: R9+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollPaneRight: R15+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollDataStart: Home+Alt+Ctrl,R7+Alt+Ctrl
OpenWindows.KeyboardCommand.ScrollDataEnd: End+Alt+Ctrl,R13+Alt+Ctrl
OpenWindows.KeyboardCommand.EraseCharBackward: Delete,BackSpace
OpenWindows.KeyboardCommand.EraseCharForward: Delete+Shift,BackSpace+Shift
OpenWindows.KeyboardCommand.EraseLine: Delete+Meta,BackSpace+Meta
```

C.1.3 Full Mouseless Mappings

When the `OpenWindows.KeyboardCommands` resource is set to Full, the following keyboard mappings are loaded.

C.1.3.1 Keyboard core functions

```
OpenWindows.KeyboardCommand.Adjust: Insert+Alt
OpenWindows.KeyboardCommand.Menu: space+Alt
OpenWindows.KeyboardCommand.InputFocusHelp: question+Ctrl
OpenWindows.KeyboardCommand.QuoteNextKey: q+Alt
OpenWindows.KeyboardCommand.SuspendMouseless: z+Alt
OpenWindows.KeyboardCommand.ResumeMouseless: Z+Alt
OpenWindows.KeyboardCommand.JumpMouseToInputFocus: j+Alt
```

C.1.3.2 Global navigation commands

```
OpenWindows.KeyboardCommand.NextElement: Tab+Ctrl
OpenWindows.KeyboardCommand.PreviousElement: Tab+Shift+Ctrl
OpenWindows.KeyboardCommand.NextPane: a+Alt
OpenWindows.KeyboardCommand.PreviousPane: A+Alt
```

C.1.3.3 Miscellaneous navigation commands

```
OpenWindows.KeyboardCommand.PanelStart: bracketleft+Ctrl
OpenWindows.KeyboardCommand.PanelEnd: bracketright+Ctrl
OpenWindows.KeyboardCommand.VerticalScrollbarMenu: v+Alt
OpenWindows.KeyboardCommand.HorizontalScrollbarMenu: h+Alt
OpenWindows.KeyboardCommand.PaneBackground: b+Alt
```

C.2 Mouseless Model Keyboard Semantic Actions

The following semantic actions are defined to support the Mouseless Model.

```
#define ACTION_ACCELERATOR                (XVIEW_FIRST+74)     /* 31818 */
#define ACTION_DELETE_SELECTION           (XVIEW_FIRST+75)     /* 31819 */
#define ACTION_ERASE_LINE                 (XVIEW_FIRST+76)     /* 31820 */
#define ACTION_HORIZONTAL_SCROLLBAR_MENU  (XVIEW_FIRST+77)     /* 31821 */
#define ACTION_INPUT_FOCUS_HELP           (XVIEW_FIRST+78)     /* 31822 */
#define ACTION_JUMP_DOWN                  (XVIEW_FIRST+79)     /* 31823 */
#define ACTION_JUMP_MOUSE_TO_INPUT_FOCUS  (XVIEW_FIRST+80)     /* 31824 */
#define ACTION_JUMP_UP                    (XVIEW_FIRST+81)     /* 31825 */
#define ACTION_MORE_HELP                  (XVIEW_FIRST+82)     /* 31826 */
#define ACTION_MORE_TEXT_HELP             (XVIEW_FIRST+83)     /* 31827 */
#define ACTION_NEXT_ELEMENT               (XVIEW_FIRST+84)     /* 31828 */
#define ACTION_NEXT_PANE                  (XVIEW_FIRST+85)     /* 31829 */
#define ACTION_PANE_BACKGROUND            (XVIEW_FIRST+86)     /* 31830 */
#define ACTION_PANE_LEFT                  (XVIEW_FIRST+87)     /* 31831 */
#define ACTION_PANE_RIGHT                 (XVIEW_FIRST+88)     /* 31832 */
#define ACTION_PANEL_START                (XVIEW_FIRST+89)     /* 31833 */
#define ACTION_PANEL_END                  (XVIEW_FIRST+90)     /* 31834 */
#define ACTION_PREVIOUS_ELEMENT           (XVIEW_FIRST+91)     /* 31835 */
#define ACTION_PREVIOUS_PANE              (XVIEW_FIRST+92)     /* 31836 */
#define ACTION_QUOTE_NEXT_KEY             (XVIEW_FIRST+93)     /* 31837 */
#define ACTION_RESUME_MOUSELESS           (XVIEW_FIRST+94)     /* 31838 */
#define ACTION_SCROLL_DATA_END            (XVIEW_FIRST+95)     /* 31839 */
#define ACTION_SCROLL_DATA_START          (XVIEW_FIRST+96)     /* 31840 */
#define ACTION_SCROLL_DOWN                (XVIEW_FIRST+97)     /* 31841 */
#define ACTION_SCROLL_JUMP_DOWN           (XVIEW_FIRST+98)     /* 31842 */
#define ACTION_SCROLL_JUMP_LEFT           (XVIEW_FIRST+99)     /* 31843 */
#define ACTION_SCROLL_JUMP_RIGHT          (XVIEW_FIRST+100)    /* 31844 */
#define ACTION_SCROLL_JUMP_UP             (XVIEW_FIRST+101)    /* 31845 */
#define ACTION_SCROLL_LEFT                (XVIEW_FIRST+102)    /* 31846 */
#define ACTION_SCROLL_LINE_END            (XVIEW_FIRST+103)    /* 31847 */
#define ACTION_SCROLL_LINE_START          (XVIEW_FIRST+104)    /* 31848 */
#define ACTION_SCROLL_RIGHT               (XVIEW_FIRST+105)    /* 31849 */
#define ACTION_SCROLL_PANE_DOWN           (XVIEW_FIRST+106)    /* 31850 */
#define ACTION_SCROLL_PANE_LEFT           (XVIEW_FIRST+107)    /* 31851 */
#define ACTION_SCROLL_PANE_RIGHT          (XVIEW_FIRST+108)    /* 31852 */
```

```
#define ACTION_SCROLL_PANE_UP              (XVIEW_FIRST+109)    /* 31853 */
#define ACTION_SCROLL_UP                   (XVIEW_FIRST+110)    /* 31854 */
#define ACTION_SELECT_ALL                  (XVIEW_FIRST+111)    /* 31855 */
#define ACTION_SELECT_DATA_END             (XVIEW_FIRST+112)    /* 31856 */
#define ACTION_SELECT_DATA_START           (XVIEW_FIRST+113)    /* 31857 */
#define ACTION_SELECT_DOWN                 (XVIEW_FIRST+114)    /* 31858 */
#define ACTION_SELECT_JUMP_DOWN            (XVIEW_FIRST+115)    /* 31859 */
#define ACTION_SELECT_JUMP_LEFT            (XVIEW_FIRST+116)    /* 31860 */
#define ACTION_SELECT_JUMP_RIGHT           (XVIEW_FIRST+117)    /* 31861 */
#define ACTION_SELECT_JUMP_UP              (XVIEW_FIRST+118)    /* 31862 */
#define ACTION_SELECT_LEFT                 (XVIEW_FIRST+119)    /* 31863 */
#define ACTION_SELECT_LINE_END             (XVIEW_FIRST+120)    /* 31864 */
#define ACTION_SELECT_LINE_START           (XVIEW_FIRST+121)    /* 31865 */
#define ACTION_SELECT_RIGHT                (XVIEW_FIRST+122)    /* 31866 */
#define ACTION_SELECT_PANE_DOWN            (XVIEW_FIRST+123)    /* 31867 */
#define ACTION_SELECT_PANE_LEFT            (XVIEW_FIRST+124)    /* 31868 */
#define ACTION_SELECT_PANE_RIGHT           (XVIEW_FIRST+125)    /* 31869 */
#define ACTION_SELECT_PANE_UP              (XVIEW_FIRST+126)    /* 31870 */
#define ACTION_SELECT_UP                   (XVIEW_FIRST+127)    /* 31871 */
#define ACTION_SUSPEND_MOUSELESS           (XVIEW_FIRST+128)    /* 31872 */
#define ACTION_TEXT_HELP                   (XVIEW_FIRST+129)    /* 31873 */
#define ACTION_TRANSLATE                   (XVIEW_FIRST+130)    /* 31874 */
#define ACTION_VERTICAL_SCROLLBAR_MENU     (XVIEW_FIRST+131)    /* 31875 */
```

C.3 SunView1 Mappings for the Mouseless Model

The following mappings are defined to map SunView1 keyboard commands to the Mouseless Model keyboard semantic actions. This section covers the SunView1 keyboard commands that have the same function in both SunView1 and in the XView Mouseless Model.

```
#define ACTION_CANCEL                   ACTION_STOP
#define ACTION_DATA_END                 ACTION_GO_DOCUMENT_END
#define ACTION_DATA_START               ACTION_GO_DOCUMENT_START
#define ACTION_DEFAULT_ACTION           ACTION_DO_IT
#define ACTION_DOWN                     ACTION_GO_COLUMN_FORWARD
#define ACTION_JUMP_LEFT                ACTION_GO_WORD_BACKWARD
#define ACTION_JUMP_RIGHT               ACTION_GO_WORD_END
#define ACTION_LEFT                     ACTION_GO_CHAR_BACKWARD
#define ACTION_LINE_END                 ACTION_ROW_END
#define ACTION_LINE_START               ACTION_ROW_START
#define ACTION_PANE_DOWN                ACTION_GO_PAGE_FORWARD
#define ACTION_PANE_UP                  ACTION_GO_PAGE_BACKWARD
#define ACTION_PARAGRAPH_DOWN           ACTION_JUMP_DOWN
#define ACTION_PARAGRAPH_UP             ACTION_JUMP_UP
#define ACTION_RIGHT                    ACTION_GO_CHAR_FORWARD
#define ACTION_ROW_END                  ACTION_GO_LINE_END
```

```
#define ACTION_ROW_START                ACTION_GO_LINE_BACKWARD
#define ACTION_SCROLL_CHAR_BACKWARD     ACTION_SCROLL_LEFT
#define ACTION_SCROLL_CHAR_FORWARD      ACTION_SCROLL_RIGHT
#define ACTION_SCROLL_COLUMN_BACKWARD   ACTION_SCROLL_UP
#define ACTION_SCROLL_COLUMN_FORWARD    ACTION_SCROLL_DOWN
#define ACTION_SCROLL_DOCUMENT_END      ACTION_SCROLL_DATA_END
#define ACTION_SCROLL_DOCUMENT_START    ACTION_SCROLL_DATA_START
#define ACTION_SCROLL_ROW_END           ACTION_SCROLL_LINE_END
#define ACTION_SCROLL_ROW_START         ACTION_SCROLL_LINE_START
#define ACTION_SCROLL_PARAGRAPH_DOWN    ACTION_SCROLL_JUMP_DOWN
#define ACTION_SCROLL_PARAGRAPH_UP      ACTION_SCROLL_JUMP_UP
#define ACTION_SCROLL_WORD_BACKWARD     ACTION_SCROLL_JUMP_LEFT
#define ACTION_SCROLL_WORD_END          ACTION_SCROLL_JUMP_RIGHT
#define ACTION_SELECT_CHAR_BACKWARD     ACTION_SELECT_LEFT
#define ACTION_SELECT_CHAR_FORWARD      ACTION_SELECT_RIGHT
#define ACTION_SELECT_COLUMN_BACKWARD   ACTION_SELECT_UP
#define ACTION_SELECT_COLUMN_FORWARD    ACTION_SELECT_DOWN
#define ACTION_SELECT_DOCUMENT_END      ACTION_SELECT_DATA_END
#define ACTION_SELECT_DOCUMENT_START    ACTION_SELECT_DATA_START
#define ACTION_SELECT_NEXT_FIELD        ACTION_SELECT_FIELD_FORWARD
#define ACTION_SELECT_PREVIOUS_FIELD    ACTION_SELECT_FIELD_BACKWARD
#define ACTION_SELECT_ROW_END           ACTION_SELECT_LINE_END
#define ACTION_SELECT_ROW_START         ACTION_SELECT_LINE_START
#define ACTION_SELECT_PARAGRAPH_DOWN    ACTION_SELECT_JUMP_DOWN
#define ACTION_SELECT_PARAGRAPH_UP      ACTION_SELECT_JUMP_UP
#define ACTION_SELECT_WORD_BACKWARD     ACTION_SELECT_JUMP_LEFT
#define ACTION_SELECT_WORD_END          ACTION_SELECT_JUMP_RIGHT
#define ACTION_UP                       ACTION_GO_COLUMN_BACKWARD
```

D
Version 3.2 and the File Chooser

The XView *File Chooser* is an implementation of the OPEN LOOK Application File Choosing Specification. The Application File Choosing Specification supercedes the traditional OPEN LOOK Style Guide on the subject of opening and saving files from an application, without the use of the File Manager. The file chooser also lets the user easily navigate through the file system. Note that the file chooser is not intended to replace the File Manager; any application supporting the file chooser is also required by the File Choosing Specification to support drag and drop with the File Manager, in accordance with the OPEN LOOK Drag and Drop Specification.

File choosers provide a simple and consistent interface for opening and saving files to the UNIX File System. File choosers also provide a Go To Menu for going to different directories which allows the user to easily select a directory from a predefined list, an application defined list or a dynamicly created history list. Normally a file chooser will be presented from an application's File Menu. The user will then have selections for opening a document, saving a document, or saving a document to a new name. Once one of these selections is made, a file chooser object is presented.

D.1 Creating File Choosers

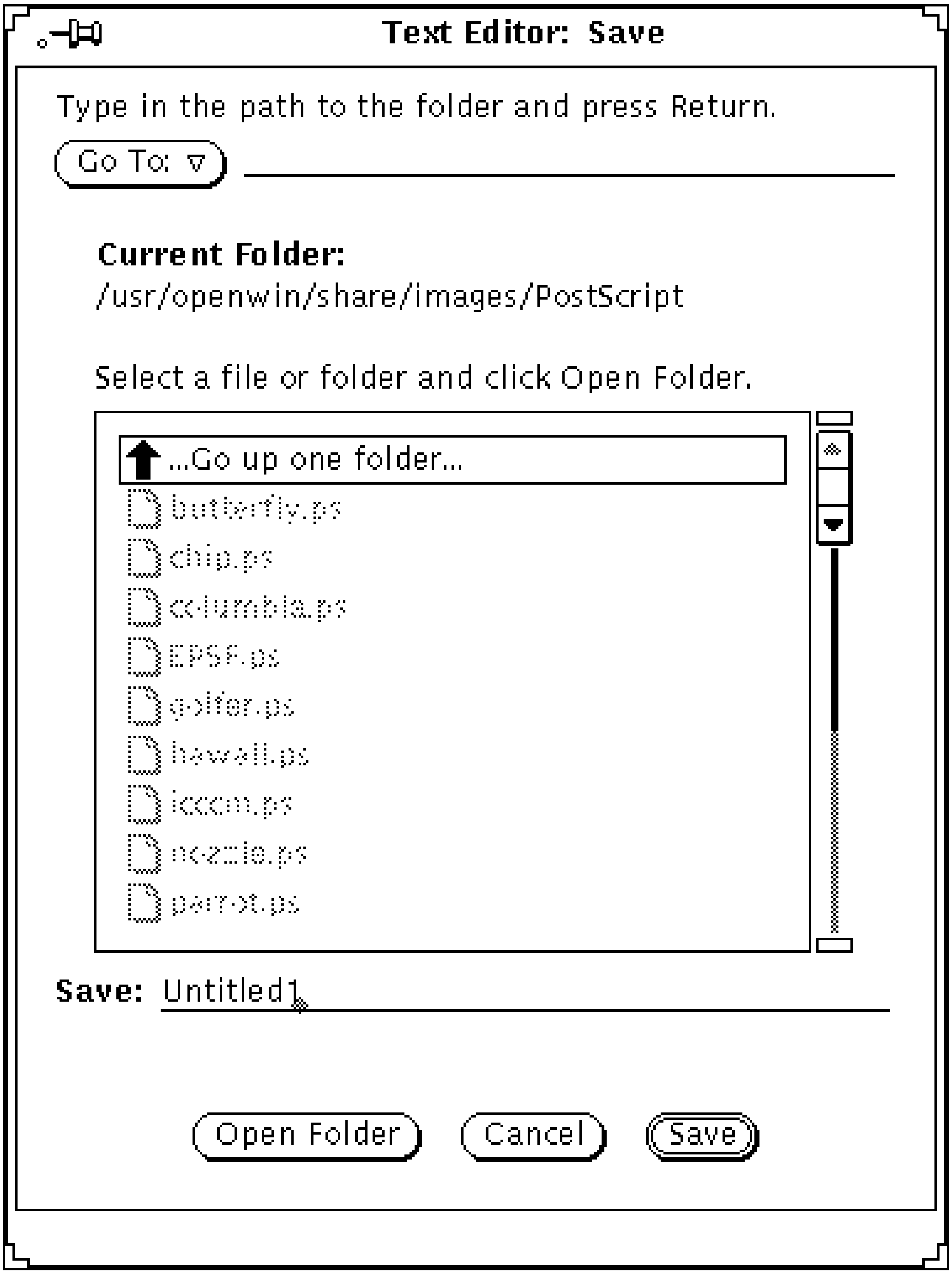
To use the `FILE_CHOOSER` package, include the header file <*xview/file_chsr.h*>. The object type is `File_chooser`. The class hierarchy for the `FILE_CHOOSER` package is:

```
[Generic] -> [(Drawable)] -> [Window] -> [Frame_cmd] -> [File Chooser]
```

File choosers come in three types:

- `FILE_CHOOSER_OPEN`
- `FILE_CHOOSER_SAVE`
- `FILE_CHOOSER_SAVEAS`

Figure D-1 shows an example of an OPEN file chooser.


Text Editor: Save

Type in the path to the folder and press Return.

Current Folder:
/usr/openwin/share/images/PostScript

Select a file or folder and click Open Folder.

...Go up one folder...
butterfly.ps
chip.ps
columbia.ps
EPSF.ps
golfer.ps
hawaii.ps
icon.ps
nozzle.ps
parrot.ps

Save: Untitled1

Figure D-1. File chooser Save dialog

Figure D-2 shows an example of a SAVE file chooser which is similar to a SAVE AS file chooser.

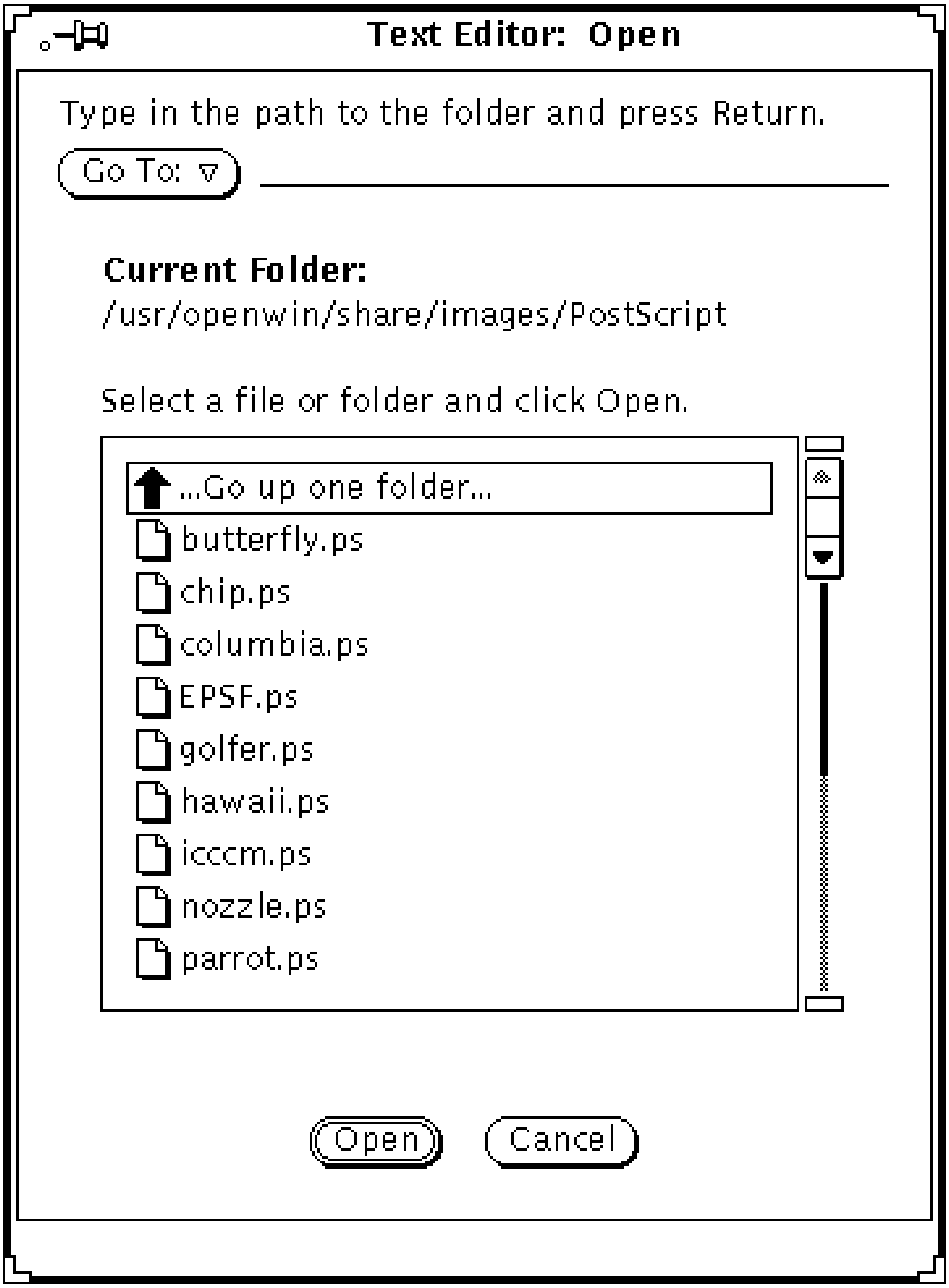


Figure D-2. File chooser Open dialog

These dialogs are presented from an application's file pop-up menu. The user may select one of the predefined package choices: "Open", "Save", "Save As", or an application defined choice such as "Import" or "Include". Once the file chooser is open, the user is presented with a frame containing several elements for navigating the folder hierarchy. These elements include the Go To menu, the Current Folder field, the scrolling list, and the Open, Save or Save As button area, which also includes a Cancel button. The Save and Save As file chooser also include a type in field below the scrolling list where the saved file is named. Refer to the figures for the relative layouts of these file chooser elements. Note: the packages that make up these elements are public and include: `FILE_LIST`, `PATH_NAME`, `HISTORY_LIST` and `HISTORY_MENU`. You can uses these packages for applications that require functionality similar to that provided by any of these file chooser components. Refer to Section A.13, "File Chooser Components" for information on using these packages. File chooser types are specified at create time using the `FILE_CHOOSER_TYPE` attribute. Alternatively, convenience macros allow you to create each of the specified file chooser types as shown below:

```
File_chooser  open_chsr;
File_chooser  save_chsr;
File_chooser  saveas_chsr;

open_chsr   = (File_chooser) xv_create(owner,
                               FILE_CHOOSER_OPEN_DIALOG, NULL);
save_chsr   = (File_chooser) xv_create(owner,
                               FILE_CHOOSER_SAVE_DIALOG, NULL);
saveas_chsr = (File_chooser) xv_create(owner,
                               FILE_CHOOSER_SAVEAS_DIALOG, NULL);
```

The owner of a `File_chooser` should be a base frame, but it can also be the root window, `NULL`, just as with a command frame.

D.2 Using a File Chooser

Given the information provided so far, we can demonstrate to how to install a file chooser for a menu item. Example D-1 shows a portion of a program which installs a File chooser on a menu item. The complete program is found in Appendix F, *Example Programs*.

Example D-1. Portion of file_chooser.c program

```
/*
 * Demonstrate the XView File Chooser
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/scrollbar.h>
#include <xview/file_chsr.h>

static Attr_attribute  MY_KEY;

typedef struct {
    Frame    frame;
```

Example D-1. Portion of file_chooser.c program (continued)

```
    Panel    panel;
    Panel_button_item  file_button;
    Menu    file_menu;
    Textsw    textsw;
    File_chooser  open;
    File_chooser  save;
    File_chooser  saveas;
    File_chooser  import;
    char *    doc_name;
} My_ui;

static void  my_open_notify();
static void  my_save_notify();
static void  my_saveas_notify();
static void  my_import_notify();
static int  my_open_callback();
static int  my_save_callback();

void
main( argc, argv )
     int argc;
     char **argv;
{
    My_ui ui = {0};


    (void) xv_init ( XV_INIT_ARGC_PTR_ARGV,
                      &argc, argv, NULL );
    MY_KEY = xv_unique_key();

    ui.file_menu
  = xv_create(XV_NULL, MENU,
        MENU_ITEM,
          MENU_STRING,    "Open...",
          MENU_NOTIFY_PROC,  my_open_notify,
          NULL,
        MENU_ITEM,
          MENU_STRING,    "Import...",
          MENU_NOTIFY_PROC,  my_import_notify,
          NULL,
        MENU_ITEM,
          MENU_STRING,    "Save...",
          MENU_NOTIFY_PROC,  my_save_notify,
          NULL,
        MENU_ITEM,
          MENU_STRING,    "Save As...",
          MENU_NOTIFY_PROC,  my_saveas_notify,
          NULL,
        XV_KEY_DATA,    MY_KEY, &ui,
        NULL);

    ui.frame = xv_create(XV_NULL, FRAME,
       XV_LABEL,    "Demo Text Editor",
       FRAME_SHOW_FOOTER,  TRUE,
       NULL );
```

Example D-1. Portion of file_chooser.c program (continued)

```
    ui.panel = xv_create(ui.frame, PANEL, NULL);

    ui.file_button = xv_create(ui.panel, PANEL_BUTTON,
             PANEL_LABEL_STRING,  "File",
             PANEL_ITEM_MENU,    ui.file_menu,
             NULL);

    window_fit_height( ui.panel );

    ui.textsw = xv_create(ui.frame, TEXTSW,
        XV_X,    0,
        WIN_BELOW,  ui.panel,
        NULL);

    xv_main_loop( ui.frame );
    exit( 0 );
}

/*
 * Picked "Open" off of File Menu.
 */
static void
my_open_notify( menu,  mi )
     Menu menu;
     Menu_item mi;
{
    My_ui *ui = (My_ui *)xv_get(menu, XV_KEY_DATA, MY_KEY);

    if ( !ui->open ) {
  ui->open
      = xv_create(ui->frame, FILE_CHOOSER_OPEN_DIALOG,
      XV_LABEL,      "Text Editor:  Open",
      FILE_CHOOSER_NOTIFY_FUNC,  my_open_callback,
      XV_KEY_DATA,      MY_KEY, ui,
      NULL);
    }

    xv_set(ui->open, XV_SHOW, TRUE, NULL);
}

/*
 * Picked OPEN off of File Menu.
 *  See Appendix F for full example
 */

my_open_callback( fc, path, file, client_data )
     File_chooser fc;
     char *path;
     char *file;
     Xv_opaque client_data;
{
    My_ui *ui = (My_ui *)xv_get(fc, XV_KEY_DATA, MY_KEY);
    Textsw_status status;
    char buf[512];

    xv_set(fc, FRAME_BUSY, TRUE, NULL);
```

Example D-1. Portion of file_chooser.c program (continued)

```
    xv_set(ui->textsw,
     TEXTSW_STATUS, &status,
     TEXTSW_FILE,  path,
     TEXTSW_FIRST, 0,
     NULL);

    if ( status != TEXTSW_STATUS_OKAY ) {
  window_bell( ui->frame );
  xv_set( ui->frame,
         FRAME_LEFT_FOOTER, "Unable to load file!",
         NULL);
  xv_set(fc, FRAME_BUSY, FALSE, NULL);
  return XV_ERROR;
    }

    /* Set current doc name on the Save popup. */
    (void) sprintf(buf, "%s.1", file);
    if ( ui->saveas )
  xv_set(ui->saveas, FILE_CHOOSER_DOC_NAME, buf, NULL);
    else {
  if ( ui->doc_name )
      free( ui->doc_name );
  ui->doc_name = strdup( buf );
    }

    (void) sprintf(buf, "Demo Text Editor – %s", file);
    xv_set(ui->frame,
     XV_LABEL,  buf,
     NULL);

    xv_set(fc, FRAME_BUSY, FALSE, NULL);

    return XV_OK;
}
.
.
.
```

D.3 Notification from a File Chooser

When a user presses the “Open” or “Save” button on a File chooser, the `FILE_CHOOSER` package first tries to validate the given path name. If the path name passes the validation, the client is notified using the notify function specified with `FILE_CHOOSER_NOTIFY_FUNC`. If the attribute `FILE_CHOOSER_NO_CONFIRM` is set to `TRUE`, the validation step does not occur.

The callbacks for the Open and Save operations are different. The Open Callback takes the form:

```
int
open_callback( fc, path, file, client_data )
  File_chooser  fc;
```

```
    char *        path;
    char *        file;
    Xv_opaque     client_data;
```

The `path` argument is the full path to the file. Use the attribute `FILE_CHOOSER_DIRECTORY` to obtain the path. The `file` argument is strictly the file-name portion of the path. The `client_data` arugment is the `client_data` field set for the row in the display list from the `FILE_CHOOSER_FILTER_FUNC`.

The Save and Save As callbacks take the form:

```
int
save_callback( fc, path, stats )
   File_chooser   fc;
   char *         path;
   struct stat *  stats;
```

The `path` parameter is the full path of the file to be saved. The `stats` parameter is a pointer to the file's `stat` structure, if the file exists, or `NULL` if the file does not exist.

D.4 Controlling the File Chooser Display List

The view of the file system that the `FILE_CHOOSER` displays in the scrolling list (the file list) can be controlled by the client using a system of three callbacks and/or a regular expression string. Using these file list controls, the file chooser can gray out files that are not selectable. An application may choose to gray out object files (.o files) or files otherwise not openable for the particular application.

The flow of control for loading a new directory into the file chooser's file list is as follows:

1. Call the callback specified with the `FILE_CHOOSER_CD_FUNC` attribute with an `op` of `FILE_CHOOSER_BEFORE_CD`. If `XV_ERROR` is returned, leave the list empty and return.
2. Match each file against the regular expression given with the `FILE_CHOOSER_FIL-TER_STRING` attribute This attribute is used primarily with the OPEN type file chooser to gray out files in the file list that cannot be opened.
3. If the mask specified with `FILE_CHOOSER_FILTER_MASK` matches the file attributes, then invoke the `FILE_CHOOSER_FILTER_FUNC`. If the callback *filter-func* returns `FILE_CHOOSER_IGNORE`, gray out the file name.
4. After reading all the files, sort them using the order specified by the callback installed with `FILE_CHOOSER_COMPARE_FUNC`.
5. After sorting, insert the list of files into the `FILE_LIST` object without repainting and call the `FILE_CHOOSER_CD_FUNC` with an `op` of `FILE_CHOOSER_AFTER_CD`.
6. Finally, redisplay the list of files as the new contents of File chooser. This also adds the current folder into the Goto Menu.

These steps, and control sequence for the attributes in the file chooser are summarized in the following control flow diagram. More details are presented in the following sections.

```
                                   /* FILE_CHOOSER_CD_FUNC */
if ( call cd_func(before-cd) != XV_OK )
   return

while ( more files ) {
                                   /* FILE_CHOOSER_FILTER_STRING */
    matched = current file / filter-string
                                   /* FILE_CHOOSER_MATCH_GLYPH */
    assign default glyph
                                   /* FILE_CHOOSER_FILTER_MASK */
                                   /* FILE_CHOOSER_FILTER_FUNC */
    if ( current file in filter-mask )
      call filter-func

    if ( file accepted )           /* FILE_CHOOSER_ABREV_VIEW */
      add to list
}
                                   /* FILE_CHOOSER_COMPARE_FUNC */
sort the list, using the compare_func

insert the new list of files into Scrolling List
                                   /* FILE_CHOOSER_CD_FUNC */
call cd_func(after-cd)

redisplay Scrolling List to user
```

D.4.1 Monitoring Directory Changes

An application may require the flexibility to intercept a change of directory or to modify the list of files after the file list has been built and before it is shown to the user. Setting the FILE_CHOOSER_CD_FUNC attribute installs a callback that is issued twice during a directory change. The first time, it is called with a File_chooser_op of FILE_CHOOSER_BEFORE_CD, this allows the client to return XV_ERROR to prevent the directory change. The change directory function could be called at this stage to veto a change of directory if the application treats a directory specially. For example, Interleaf uses a directory to store book files, that make up a single "document". Thus a Interleaf application could open a directory as a document rather than the individual files in the directory.

The second invocation of the change directory function callback is after the dispay list has been built. The callback's op parameter is set to FILE_CHOOSER_AFTER_CD. This allows the client to modify the file list before the user sees it. If xv_set calls are used on the file list from inside the change directory callback, be sure to set PANEL_PAINT for the file list to PANEL_NONE.

D.4.2 Filtering

Files which are not "openable" by the application will be grayed out in the disply list. The application may select files for special filtering using `FILE_CHOOSER_FILTER_STRING` and/or `FILE_CHOOSER_FILTER_MASK`.

Using `FILE_CHOOSER_FILTER_STRING`, an application may specify a regular expression to match parts of a file name such as a common postfix (for example `.c` for C source files). Files that match this expression will then be grayed out in the file list. Along with this regular expression matching, the application may specify a `Server_image` that automatically is displayed for each file that matches. The `Server_image` for matched files is specified by `FILE_CHOOSER_MATCH_GLYPH` and `FILE_CHOOSER_MATCH_GLYPH_MASK`. By default, the filter string is set to `NULL` which disables pattern matching entirely.

Another method for filtering the file list is by specifing a callback that allows the application to qualify each file and return information such as a `Server_image` to be displayed with the file and a `client_data` field to return to the "open" notify callback. The filter callback is installed using `FILE_CHOOSER_FILTER_FUNC`. The files for wich this callback gets issued can be selected with the `FILE_CHOOSER_FILTER_MASK` attribute which takes an `or`'d list of type `File_chooser_filter_mask` flags.

D.4.3 File Chooser Sorting

The default sorting algorithm used by the `FILE_CHOOSER` package is alpha-numeric without case sensitivity. If a different sort algorithm is appropriate for a particular application, the different sort routine may be installed with the `FILE_CHOOSER_COMPARE_FUNC` attribute. This attribute works using the `qsort(3)` library routine.

XView provides several built-in comparison functions for ascending and descending case-sensitive or case-insensitive sorts. Also, the comparison functions are given arguments of type `File_chooser_row` which includes information such as the file name, the associated `stat(3)` structure, the `strxfrm(3)` version of the string and the `File_chooser_op` of `FILE_CHOOSER_MATCHED` or `FILE_CHOOSER_NOT_MATCHED` with respect to the `FILE_CHOOSER_FILTER_STRING`. This provides a highly flexible and efficient means of sorting files.

D.5 Modifying the Display List

There are several attributes that allow the client to modify the display of files in a file chooser. A file chooser's default values comply with the OPEN LOOK Application File Choosing Specification. Note that the File Choosing Specification requires the client to provide an appropriate human interface to revert to the default behavior if the client changes the default behavior (for example, the application should provide a toggle that turns off the display of dot files if the application displays dot files.)

D.5.1 Dot Files

Since most dot files are created by applications and not directly by the user, the file chooser does not show dot files by default (also, user testing has shown that dot files often confuse end-users). If an application needs to show dot files in the file list display, the attribute `FILE_CHOOSER_SHOW_DOT_FILES` toggles their display.

D.5.2 Abbreviated View

Some applications may only need to show those files that are relevant to the application. In this case, files that are not openable may be removed from the file list, rather than being displayed as grayed out items. This functionality may be turned on by setting the `FILE_CHOOSER_ABBREV_VIEW` attribute to `TRUE`. Warning: use of this attribute is recommended only for applications that expect very technical users who will understand that files not being displayed is not a problem.

D.6 File Chooser Customization

The application has the option of adding permanent application-specific entries into the file chooser Go To menu. The attribute `FILE_CHOOSER_APP_DIR` sets the values for these entries.

Some applications may need to do use a non-standard configuration with the Go To menu. This is achieved by creating a custom `History_list` object and attaching it to a file chooser with the `FILE_CHOOSER_HISTORY_LIST` attribute. Once installed, the `FILE_CHOOSER` package will add "Recent Entries" to the installed list as the user navigates the file system.

Other modifications for the Go To history list and menu can be accomplished using the `HISTORY_LIST` and `HISTORY_MENU` attributes. The last section of this chapter list the `HISTORY` attributes. Refer to the XView Reference Manual for additional information on these attributes.

The Save and Save As FILE_CHOOSERs can include a default name in the field for the File type in. You specify this default name using the FILE_CHOOSER_DOC_NAME attribute. For a Save file chooser, the default document name should be Untitled1. For a Save As File Chooser, the default document name should be "*current document*".1.

In some circumstances, the application may wish to save more than one file to a directory. The file chooser Save dialog does not provide for this very well, so there is an option to gray out the Save typein while leaving the Save button available to the user. This is toggled with the FILE_CHOOSER_SAVE_TO_DIR attribute.

Since the file chooser has elements that are public it is possible to get handles to the objects that make up the file chooser, either by the FILE_CHOOSER_CHILD interface or by PANEL_EACH_ITEM, etc. For example, the file chooser's FILE_LIST object is available in this manner. Keep in mind that values that are not gotten or set via the parent FILE_CHOOSER API are *not* guaranteed to be compatible across releases of the XView toolkit.

The attribute FILE_CHOOSER_AUTO_UPDATE modifies the action of a FILE_CHOOSER. This attribute tells the FILE_CHOOSER to re-read its current directory only when FILE_CHOOSER_UPDATE is explicitly called, or when the user performs some action to read the directory.

D.7 Customizing the File Chooser Dialog

The OPEN LOOK Application File Choosing Specification defines two methods for OPEN LOOK-Compliant file chooser customization. The first, and simplest, is to modify the Open dialog and define a custom File Chooser, such as an "Import" or an "Include" File Chooser. Using the attributes supplied for adding a custom button, this change is not difficult. For example,

```
File_chooser open_chsr;

open_chsr = (File_chooser) xv_create(owner,
                        FILE_CHOOSER_OPEN_DIALOG,
                        FILE_CHOOSER_CUSTOMIZE_OPEN,
                          "Import",
                          "Select a file or folder and click Import",
                        FILE_CHOOSER_SELECT_FILES,
                        NULL);
```

The first argument supplied to the attribute FILE_CHOOSER_CUSTOMIZE_OPEN is the label for the new custom button. The second argument is the line of text that is placed above the Scrolling List. The third argument is an enum value that represents whether this custom operation can deal with files (that is, if the item can only use files this argument is set to FILE_CHOOSER_SELECT_FILES, otherwise, if both files and directories are valid it is set to FILE_CHOOSER_SELECT_ALL).

The second way to customize the XView file chooser is to add custom controls to the file chooser panel. Implementing this is a multi-step process.

The OPEN LOOK Application File Choosing Specification mandates that custom controls be placed under the Scrolling List (and below the Save typein), but above the buttons at the bottom of the dialog. To implement this, the FILE_CHOOSER package reserves an area, called the *extension rectangle*, in part of the dialog space. By default, the extension rectangle has a height of 0 pixels; the client make may this area a specific size using the FILE_CHOOSER_EXTEN_HEIGHT attribute.

The FILE_CHOOSER package leaves much of the responsibility for layout and sizing of custom controls in the hands of the client. The client has the responsibility of positioning custom controls within the extention rectangle area during layout and resizing. The callback installed with the attribute FILE_CHOOSER_EXTEN_FUNC handles resizing and layout.

Other responsibilities of the client when adding custom controls include adjusting the default and minimum sizes for the Frame.

Example D-2 illustrates adding an extension item to a file chooser.

Example D-2. An extension item program

```
File_chooser fc;
Panel        panel;
Panel_item   item;
int          item_width;
int          item_height;
int          frame_width;
int          frame_height;

panel = xv_get(fc, FRAME_CMD_PANEL);

item = xv_create(panel, PANEL_CHOICE,
         PANEL_LABEL_STRING,    "Hidden Files:",
         PANEL_CHOICE_STRINGS,  "Hide", "Show", NULL,
         PANEL_NOTIFY_PROC,      my_show_dot_files_proc,
         NULL);

item_width = (int) xv_get(item, XV_WIDTH);
item_height = (int) xv_get(item, XV_HEIGHT);


/*
* Adjust Frame default size to make room for the extension item.
*/
frame_width = (int) xv_get(fc, XV_WIDTH);
frame_height = (int) xv_get(fc, XV_HEIGHT);
xv_set(fc,
     XV_WIDTH,  MAX(frame_width, (item_width + xv_cols(panel, 4))),
     XV_HEIGHT, frame_height + item_height,
     NULL);


/*
* Adjust Frame Min Size.  provide for at least 2
* columns on either side of the extension item.
*/
xv_get(fc, FRAME_MIN_SIZE, &frame_width, &frame_height);
xv_set(fc,
```

Example D-2. An extension item program (continued)

```
    FRAME_MIN_SIZE, MAX( frame_width, (item_width + xv_cols(panel, 4))),
                    frame_height + item_height,
    NULL);


/* Tell file chooser to reserve layout space for it */

xv_set(fc,
     FILE_CHOOSER_EXTEN_HEIGHT,  item_height,
     FILE_CHOOSER_EXTEN_FUNC,    my_exten_func,
     XV_KEY_DATA, EXTEN_ITEM_KEY, item,
     NULL);


/*   [.................]  */


static int
my_exten_func( fc, frame_rect, exten_rect,
               left_edge, right_edge, max_height )
     File_chooser fc;
     Rect *frame_rect;
     Rect *exten_rect;
     int left_edge;
     int right_edge;
     int max_height;
{
    Panel_item item = (Panel_item) xv_get(fc,
                                        XV_KEY_DATA, EXTEN_ITEM_KEY);
    int item_width;

    item_width = (int) xv_get(item, XV_WIDTH);

/*
* show item centered in frame.
*/
    xv_set(item,
     XV_X,  (frame_rect->r_width - item_width) / 2,
     XV_Y,  exten_rect->r_top,
     PANEL_PAINT, PANEL_NONE,
     NULL);

    return -1;    /* (-1) means exten height didn't change */
}
```

D.7.1 File Chooser Components

Internally, the file chooser is uses three high-level interfaces: the FILE_LIST package, the PATH_NAME packge, and the HISTORY package. The FILE_LIST package is a subclass of PANEL_LIST; it handles all of the navigation and display of the UNIX File System. The PATH_NAME package is a subclass of PANEL_TEXT; it handles shell variable and tilde expansion. The HISTORY package implements a shareable command history through the XView

Menu and Menu_item packages (the Go To menus in the file chooser use this). If your file manipulation needs do not fall within the scope of the OPEN LOOK Application File Choosing Specification, you have the option of designing your own dialog with the same components the File uses. Since the FILE_CHOOSER package implements a particular look and feel, which may not suffice for all application needs, the component parts are made available.

```
[Generic] -> [Panel_item] -> [Panel_list_item] -> [File_list]
[Generic] -> [Panel_item] -> [Panel_text_item] -> [Path_name]
[Generic] -> [History_menu]
[Generic] -> [History_list]
```

D.8 Version 3.2 Additions

This section lists additional changes for available with XView Version 3.2 and new releases.

D.8.1 New Panel List Attributes for Version 3.2

Version 3.2 of XView offers a new and improved method for adding entries to a PANEL_LIST. Version 3.2 offers the following new attributes for panel lists:

```
PANEL_LIST_INACTIVE
PANEL_LIST_DELETE_INACTIVE_ROWS
PANEL_LIST_DO_DBL_CLICK
PANEL_LIST_MASK_GLYPH
PANEL_LIST_MASK_GLYPHS
PANEL_LIST_ROW_VALUES
PANEL_LIST_EXTENSION_DATA
PANEL_LIST_EXTENSION_DATAS
```

D.8.1.1 Adding new list entries

The attribute PANEL_LIST_ROW_VALUES offers an improved performance method of getting/setting row values in a PANEL_LIST. This attribute takes the row number, a pointer to a Panel_list_row_values array, and a count of how many rows in the array. The enum Panel_list_row_values is defined as shown below:

```
typedef struct {
char * string;
Server_image glyph;
Server_image mask_glyph;
Xv_font font;
Xv_opaque client_data;
Xv_opaque extension_data;
unsigned inactive : 1;
unsigned selected : 1;
} Panel_list_row_values;
```

On `get`, the arguments remain the same, the array passed in gets filled in by the `PANEL_LIST` package. The return value is the number of rows that were successfully filled in.

Example D-3 shows a portion of a program which uses the new `PANEL_LIST` insertion method.

Example D-3. Program that adds values to a panel list

```
/*
 * Demonstrate the use of the PANEL_LIST_ROW_VALUES attribute
 */

#include <stdio.h>
#include <xview/xview.h>
#include <xview/font.h>
#include <xview/panel.h>


static Attr_attribute MY_KEY;

static void  my_clear_proc();
static void  my_load_proc();
static void  my_print_proc();

typedef struct {
    Frame frame;
    Panel_list_item list;
    Xv_font font;
} My_ui;


void
main ( argc, argv )
     int argc;
     char **argv;
{
    Panel panel;
    My_ui ui;

    (void) xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    MY_KEY = xv_unique_key();

    ui.frame = xv_create ( XV_NULL, FRAME,
        XV_LABEL,    "New Load",
        FRAME_SHOW_FOOTER,  TRUE,
        NULL );
    panel = xv_create ( ui.frame, PANEL, NULL );


    (void) xv_create ( panel, PANEL_BUTTON,
          PANEL_LABEL_STRING,    "Load",
          PANEL_NOTIFY_PROC,    my_load_proc,
          XV_KEY_DATA,      MY_KEY, &ui,
          NULL );

    (void) xv_create ( panel, PANEL_BUTTON,
          PANEL_LABEL_STRING,    "Clear",
```

Example D-3. Program that adds values to a panel list (continued)

```
          PANEL_NOTIFY_PROC,    my_clear_proc,
          XV_KEY_DATA,      MY_KEY, &ui,
          NULL );

    (void) xv_create( panel, PANEL_BUTTON,
          PANEL_LABEL_STRING,    "Print Selected Row",
          PANEL_NOTIFY_PROC,    my_print_proc,
          XV_KEY_DATA,      MY_KEY, &ui,
          NULL );

    ui.list = xv_create( panel, PANEL_LIST,
      PANEL_LIST_WIDTH,    300,
      PANEL_LIST_DISPLAY_ROWS,  8,
      NULL);

    ui.font = xv_create(XV_NULL, FONT,
      FONT_FAMILY,  FONT_FAMILY_DEFAULT,
      FONT_STYLE,  FONT_STYLE_BOLD,
      NULL);

    window_fit ( panel );
    window_fit ( ui.frame );
    xv_main_loop ( ui.frame );
}


static void
my_clear_proc( item, event )
     Panel_item item;
     Event *event;
{
    My_ui *ui = (My_ui *)xv_get(item, XV_KEY_DATA, MY_KEY);
    int rows = (int)xv_get(ui->list, PANEL_LIST_NROWS);

    if ( rows > 0 )
  xv_set(ui->list,
         PANEL_LIST_DELETE_ROWS,  0, rows,
         NULL);
    xv_set(ui->frame, FRAME_LEFT_FOOTER, "", NULL);
}


static void
my_load_proc( item, event )
     Panel_item item;
     Event *event;
{
    My_ui *ui = (My_ui *)xv_get(item, XV_KEY_DATA, MY_KEY);
    int ii;
    Panel_list_row_values vals[1000];
    char buf[64];

    xv_set(ui->frame, FRAME_BUSY, TRUE, NULL);

    for(ii=0; ii<1000; ++ii) {
  vals[ii].string = "Testing PANEL_LIST_ROW_VALUES";
```

Example D-3. Program that adds values to a panel list (continued)

```
  vals[ii].font = ui->font;
  vals[ii].glyph = XV_NULL;
  vals[ii].mask_glyph = XV_NULL;
  vals[ii].client_data = ui->font;
  vals[ii].selected = FALSE;
  vals[ii].inactive = FALSE;
    }

    xv_set(ui->list,
     PANEL_LIST_ROW_VALUES,  0, &vals, 1000,
     NULL);

    (void) sprintf(buf, "%d rows in list",
       (int) xv_get(ui->list, PANEL_LIST_NROWS)
       );
    xv_set(ui->frame,
     FRAME_LEFT_FOOTER,  buf,
     FRAME_BUSY, FALSE,
     NULL);
}



static void
my_print_proc( item, event )
     Panel_item item;
     Event *event;
{
    My_ui *ui = (My_ui *)xv_get(item, XV_KEY_DATA, MY_KEY);
    Panel_list_row_values vals;
    int row = (int) xv_get(ui->list, PANEL_LIST_FIRST_SELECTED);
    int count;

    count = (int) xv_get(ui->list, PANEL_LIST_ROW_VALUES, row, &vals, 1);

    if ( count != 1 ) {
  window_bell( ui->frame );
  xv_set(ui->frame,
        FRAME_LEFT_FOOTER,  "Unable to get row",
        FRAME_BUSY,     FALSE,
        NULL);
  return;
    }

    printf( "Row Number %d:0, row );
    printf( "   String: %s0, vals.string );
    printf( "   Selected: %d0, vals.selected );
    printf( "   Inactive: %d0, vals.inactive );
}
```

D.8.1.2 Other panel list changes

This section describes the additional changes for the panel list package.

The attribute `PANEL_LIST_INACTIVE` "Grays out" a row in a `PANEL_LIST`. Note that a row that is inactive cannot be selected. Also, mouseless model navigation is not effected by the inactive state of the individual rows.

The attribute `PANEL_LIST_DELETE_INACTIVE_ROWS` deletes all inactive rows from the list. This is similar to `PANEL_LIST_DELETE_SELECTED_ROWS` attribute.

The `PANEL_LIST_DO_DBL_CLICK` attribute tells `PANEL_LIST` to interpret two select events that occur within the timout value as a double-click instead of as a second select or a deselect (depending on the current mode of the list). The timeout value is specified with `OpenWindows.MulticlickTimeout` If true, the `PANEL_LIST` will deliver a new op called `PANEL_LIST_OP_DBL_CLICK` instead of the normal `PANEL_LIST_OP_SELECT` or `PANEL_LIST_OP_DESELECT`.

The attributes `PANEL_LIST_MASK_GLYPH` and `PANEL_LIST_MASK_GLYPHS` tell the `PANEL_LIST` to use the given `Server_image` as a clip mask for the corresponding `PANEL_LIST_GLYPH`. The `Server_image` supplied must be of depth 1. `PANEL_LIST_MASK_GLYPHS` is like like `PANEL_LIST_MASK_GLYPH`, but takes a `NULL` terminated list of glyphs rather than a row and a single handle.

The attributes `PANEL_LIST_EXTENSION_DATA` and `PANEL_LIST_EXTENSION_DATAS` are the same as `PANEL_LIST_CLIENT_DATA`, except they are reserved for package implementors. This is used by the `FILE_LIST` package.

D.9 Keyboard Menu Accelerators

The keyboard menu accelerator functionality in XView provides attributes to associate an accelerator with a menu item. Keyboard menu accelerators can be used to invoke menu commands directly without having to display the menu in which the commands appear. These accelerators provide a more efficient path to familiar menu functionality.

Accelerators are global to a frame of an application. The accelerator key strokes are displayed on the right side of the menu item. The diamond symbol represents the meta key. If there are qualifiers such as Control (ctrl), Shift (shift) or Alt (alt), indicated on the menu item, those keys are to be used in conjunction with the accelerator key and possibly the Meta key (if the diamond symbol exists on the menu item).

If a menu is pinned, the accelerators will not be displayed on the menu, as a way to conserve screen space. Although the accelerators are not visible on the menu, they are still active and can be used as long as the input focus is in the application frame where the menu was first brought up.

Menu accelerators are activated if the resource `OpenWindows.MenuAccelerators` is set to True (the default case for OpenWindows Version 3.2). This resource can be set using the "Keyboard" category of the OpenWindows(Version 3.2) Workspace Properties program. Setting "Keyboard Menu Equivalents" to "Application + Window" or "Application Only" sets the OpenWindows.MenuAccelerators resource to True, while the “None” setting will set the resource to False.

D.9.0.1 Frame package menu accelerator attributes

The attributes FRAME_MENUS, FRAME_MENU_COUNT, FRAME_MENU_ADD, and FRAME_MENU_DELETE are used in conjunction with menu accelerators. They are used to inform the frame object which menus will be used on the frame - this is required because the frame has to know what menu accelerators to detect. Any menus with accelerators that are used anywhere on the frame, i.e. on panels, textsw, canvas will not work until they are registered using these attributes.

The attribute FRAME_MENUS replaces the current menu list with the one passed on the avlist. For get, this return a pointer to the current list of menus. The list returned should not be modified by the application.

```
Menu   *menu_list;

xv_set(frame1, FRAME_MENUS,
                edit_menu, load_menu, NULL,
               NULL);

menu_list = (Menu *)xv_get(frame1, FRAME_MENUS);
```

The number of menus can be obtained with FRAME_MENU_COUNT.

```
int   menu_count;

menu_count = (int)xv_get(frame2, FRAME_MENU_COUNT);
```

The attribute FRAME_MENU_COUNT returns the current number of menus registered on the frame via FRAME_MENUS, FRAME_MENU_ADD, or FRAME_MENU_DELETE.

The attribute FRAME_MENU_ADD appends to the list of accelerated menus on the frame. For example:

```
xv_set(frame1,
     FRAME_MENU_ADD, print_menu,
     NULL);
```

The attribute FRAME_MENU_DELETE deletes from the list of accelerated menus on the frame.

```
xv_set(frame1,
     FRAME_MENU_DELETE, print_menu,
     NULL);
```

D.9.0.2 The menu attributes for menu accelerators

The following attributes provide functionality to define menu accelerators for menu items. The basic difference for menu accelerators is that when an accelerator (eg. Meta+l) is pressed, the menu is not brought up. Instead, the procedures that are normally called when one selects a menu item, are called directly.

Each new attribute below corresponds to one of many ways of creating a menu item. The attribute `MENU_STRINGS_AND_ACCELERATORS` and `MENU_ACTION_ACCELERATOR` are for static creation of menu items for a given menu.

The attribute `MENU_ACCELERATOR`, when used in a create or set call, set an accelerator on a menu item:

```
Menu   menu;

menu = xv_create(NULL, MENU
             MENU_ITEM,
                 MENU_STRING, "Load",
                 MENU_NOTIFY_PROC, load_proc,
                 MENU_ACCELERATOR, "Meta+l",
             NULL,
             NULL);
```

or

```
Menu_item  load_item;

load_item = xv_create(NULL, MENU_ITEM,
                  MENU_STRING, "Load",
                  MENU_NOTIFY_PROC, load_proc,
                  MENU_ACCELERATOR, "Meta+l",
                  NULL);
```

The accelerator string will be copied by XView. A get returns the accelerator string. The string returned should not be modified.

The attribute `MENU_ACTION_ACCELERATOR` can be used to create a menu item with a given label, notify procedure, and accelerator:

```
xv_set(menu,
         MENU_ACTION_ACCELERATOR,
             "Load", load_proc, "Meta+L",
         NULL);
```

The menu item label string, the first argument, will not be copied by XView. The accelerator string, argument three, will be copied.

`MENU_STRINGS_AND_ACCELERATORS` can be used to create a menu item with a given label and accelerator:

```
xv_set(menu, MENU_NOTIFY_PROC, file_proc,
             MENU_STRINGS_AND_ACCELERATORS,
                "Load", "Meta+l",
                "Print", "Meta+p",
                "Include","Meta+Ctrl+i",
             NULL,
             NULL);
```

The accelerator strings will be copied by XView. The menu item label strings will not be copied.

For all the attributes above, if an accelerator is *changed* with `xv_set`, the attribute `FRAME_MENUS` must be set again before the change to the accelerator will take effect.

Callback procedures registered using `MENU_NOTIFY_PROC`, `MENU_GEN_PROC`, and `MENU_DONE_PROC` will be called the same way for menu accelerators as if the menu item was selected using the menu directly.

The accelerators can be specified in a number of ways. These specifications are a combination of Xt, OLIT, and XView syntaxes:

```
Xt:             [modifier...] '<Key>'key
OLIT:           [OLITmodifier...] '<'key'>'
XView:          [modifier ['+' modifier] '+'] key
modifier:       'Meta' | 'Shift' | 'Alt' | 'Hyper' | 'Ctrl' |
OLITmodifier:   modifier | 'm' | 's' | 'a' | 'h' | 'c'
```

Key: all print characters and X keysym names (e.g. 'return', 'tab', 'comma', 'period', etc . . .)

Note: keysym names consist of the entries in <X11/keysymdef.h> without the “XK_” prefix. For example:

```
Meta+Shift+a
Meta+comma
Meta <Key>c
Shift Meta <Key>I
m <z>
```

D.9.0.3 Resources

The resource <appname>.<menu item instance name>.accelerator can be used to override the accelerator specified using the attributes described above.

For example, in the application 'foo', if we have:

```
load_item = xv_create(NULL, MENUITEM,
                            XV_INSTANCE_NAME, "load",
                            MENU_STRING, "load",
                            MENU_ACCELERATOR, "Meta+l",
                            MENU_NOTIFY_PROC, load_action,
                      NULL);
```

The "Meta+l” can be overriden by having the following entry in your X resource database:

```
foo.load.accelerator:Meta+b
```

The load action will be done when “Meta+b” is pressed instead of “Meta+l”. Note: `XV_INSTANCE_NAME` will need to be used to give the menu item an instance name.

D.9.0.4 Core set menu accelerators

Various menu accelerators for "common features" of OpenWindows (e.g. Print, Save, Quit, Undo, Paste, etc.) are called the core set menu accelerators. These accelerators are special in that they are specified using the string: `"coreset <core set name>"` instead of the syntax mentioned above. For example:

```
load_item = xv_create(NULL, MENUITEM,
                      MENU_STRING, "Open",
                      MENU_ACCELERATOR, "coreset Open",
                      MENU_NOTIFY_PROC, load_action,
                      NULL);
```

The core set menu accelerators will be activated only if the X resources:

```
OpenWindows.MenuAccelerator.<core set name>:<accelerator string>
```

are present. For example:

```
OpenWindows.MenuAccelerator.Open:Meta<key>o
```

This means that the key combination meta-o is the menu accelerator for the "open" action. These resources can also be used to rebind the core set accelerators. The advantage of core set accelerators is that all applications that use them will have uniform accelerators (use the same key bindings for the same functions). Rebinding using the resource above will affect all such applications.

The list of possible values of <core set name> and the default case-sensitive resource bindings for `OpenWindows.MenuAccelerator.<core set name>` are as follows:

Core set accelerator	Default binding
`BoldFont`	`Shift Meta <Key>B`
`Copy`	`Meta<Key>c`
`Cut`	`Meta <Key>x`
`Find`	`Meta <Key>f`
`ItalicFont`	`Shift Meta <Key>I`
`New`	`Meta <Key>n`
`NormalFont`	`Shift Meta <Key>N`
`Open`	`Meta <Key>o`
`Paste`	`Meta <Key>v`
`Print`	`Meta <Key>p`
`Props`	`Meta <Key>i`
`Redo`	`Shift Meta <Key>Z`
`Save`	`Meta <Key>s`
`SelectAll`	`Meta <Key>a`
`Typeface`	`Meta <Key>t`
`Undo`	`Meta <Key>z`

In OpenWindows 3.2, the above bindings are set by default upon startup.

D.9.0.5 Events

When a menu is brought up using `menu_show()` and a menu item is selected, it is possible to query the menu using the attribute `MENU_FIRST_EVENT` to obtain the event `(event_action(event) == ACTION_MENU)` that was passed into `menu_show()`, presumably the event that caused the menu to be brought up.

```
Event  *menu_event;

menu_event =  (Event *)xv_get(menu, MENU_FIRST_EVENT);
```

When a menu item notify procedure is called using a menu accelerator, `MENU_FIRST_EVENT` will return the event corresponding to the accelerator. That is, the event that corresponds to "Meta+l". The action id, obtained using the `event_action()` macro, will be `ACTION_ACCELERATOR`. The window id in the event will be the subwindow that had input focus (and received the key event) at the time.

D.10 File Chooser and Version 3.2 Additions Summary

Table D-1 lists the procedures macros for the File Chooser. Table D-2 lists the attributes and macros the File Chooser. This section also lists the attributes for the `HISTORY`, `FILE_LIST` and `PATH` packages, as well as the XView Version 3.2 and newer `PANEL_LIST` and `MENU` accelerator additions. This information is described fully in the *XView Reference Manual*.

Table D-1. File Chooser Procedures and Macros

Procedures and Macros	
fchsr_case_ascend_compare()	FILE_CHOOSER_OPEN_DIALOG
fchsr_case_descend_compare()	FILE_CHOOSER_SAVE_DIALOG
fchsr_no_case_ascend_compare()	FILE_CHOOSER_SAVEAS_DIALOG
fchsr_no_case_descend_compare()	FILE_CHOOSER_NULL_COMPARE
	FILE_CHOOSER_DEFAULT_COMPARE

Table D-2. File Chooser Attributes

FILE_CHOOSER_ABBREV_VIEW	FILE_CHOOSER_FILTER_MASK
FILE_CHOOSER_APP_DIR	FILE_CHOOSER_FILTER_STRING
FILE_CHOOSER_AUTO_UPDATE	FILE_CHOOSER_HISTORY_LIST
FILE_CHOOSER_CD_FUNC	FILE_CHOOSER_MATCH_GLYPH
FILE_CHOOSER_CHILD	FILE_CHOOSER_MATCH_GLYPH_MASK
FILE_CHOOSER_COMPARE_FUNC	FILE_CHOOSER_NOTIFY_FUNC
FILE_CHOOSER_CUSTOMIZE_OPEN	FILE_CHOOSER_NO_CONFIRM
FILE_CHOOSER_DIRECTORY	FILE_CHOOSER_SAVE_TO_DIR
FILE_CHOOSER_DOC_NAME	FILE_CHOOSER_SHOW_DOT_FILES

Table D-2. File Chooser Attributes (continued)

FILE_CHOOSER_EXTEN_FUNC	FILE_CHOOSER_TYPE
FILE_CHOOSER_EXTEN_HEIGHT	FILE_CHOOSER_UPDATE
FILE_CHOOSER_FILTER_FUNC	

Table D-3. History and History Menu Attributes

HISTORY_ADD_FIXED_ENTRY	HISTORY_MENU_HISTORY_LIST
HISTORY_ADD_ROLLING_ENTRY	HISTORY_MENU_OBJECT
HISTORY_DUPLICATE_LABELS	HISTORY_NOTIFY_PROC
HISTORY_DUPLICATE_VALUES	HISTORY_ROLLING_COUNT
HISTORY_FIXED_COUNT	HISTORY_ROLLING_MAXIMUM
HISTORY_INACTIVE	HISTORY_VALUE
HISTORY_LABEL	

Table D-4. File List Attributes

FILE_LIST_ABBREV_VIEW	FILE_LIST_MATCH_GLYPH
FILE_LIST_AUTO_UPDATE	FILE_LIST_MATCH_GLYPH_MASK
FILE_LIST_CHANGE_DIR_FUNC	FILE_LIST_ROW_TYPE
FILE_LIST_COMPARE_FUNC	FILE_LIST_SHOW_DIR
FILE_LIST_DIRECTORY	FILE_LIST_SHOW_DOT_FILES
FILE_LIST_DOTDOT_STRING	FILE_LIST_UPDATE
FILE_LIST_FILTER_FUNC	FILE_LIST_USE_FRAME
FILE_LIST_FILTER_MASK	
FILE_LIST_FILTER_STRING	

Table D-5. Path Attributes

PATH_IS_DIRECTORY
PATH_USE_FRAME
PATH_RELATIVE_TO
PATH_LAST_VALIDATED

Table D-6. Version 3.2 Panel List Attributes

PANEL_LIST_INACTIVE	PANEL_LIST_MASK_GLYPHS
PANEL_LIST_DELETE_INACTIVE_ROWS	PANEL_LIST_ROW_VALUES
PANEL_LIST_DO_DBL_CLICK	PANEL_LIST_EXTENSION_DATA
PANEL_LIST_MASK_GLYPH	PANEL_LIST_EXTENSION_DATAS

Table D-7. Version 3.2 Menu Accelerator Attributes

FRAME_MENUS	MENU_ACCELERATOR
FRAME_MENU_COUNT	MENU_ACTION_ACCELERATOR
FRAME_MENU_ADD	MENU_STRINGS_AND_ACCELERATORS
FRAME_MENU_DELETE	

E OPEN LOOK User-interface Compliance

This appendix lists the ways that the XView Toolkit is not compliant with the *OPEN LOOK Graphical User Interface Functional Specification*. It is not a complete list of the ways that OpenWindows 3.0 is not an OPEN LOOK UI-compliant environment. OPEN LOOK UI compliance has two components: toolkit compliance and environment compliance.

An OPEN LOOK UI-compliant toolkit allows a developer to write an application that will be OPEN LOOK UI-compliant if run with an OPEN LOOK UI window manager. The toolkit might also support the application running successfully with, for example, a MOTIF™ window manager, but in such a configuration, the application would not be OPEN LOOK UI-compliant. An OPEN LOOK UI-compliant environment consists of an OPEN LOOK UI window manager, file manager, workspace properties window, and other such utility programs. To guarantee an OPEN LOOK UI application, the developer must write the application with an OPEN LOOK UI-compliant toolkit *and* run the application in an OPEN LOOK UI-compliant environment.

This list is in three parts. The first part consists of those features missing from XView 3.0 that are specified as Level 1 OPEN LOOK UI features. The second part lists some of the Level 2 OPEN LOOK UI features supported by XView 3.0. The third part lists the rest of the Level 2 OPEN LOOK UI features, which are not supported by XView 3.0.

E.1 Level 1 Features Not Supported in XView 3.0

The Level 1 features listed on the following pages are not supported in XView 3.0.

E.1.1 Keyboard and Mouse Customization

XView 3.0 hard-codes the bindings for function keys, mouse buttons, and mouse modifiers that OPEN LOOK UI says the user should be able to customize.

An OPEN LOOK UI toolkit should allow the user to specify the keys used for `CUT`, `COPY`, `PASTE`, `PROPERTIES`, `UNDO`, `CANCEL`, `DEFAULTACTION`, `NEXTFIELD`, and `PREVFIELD`.

- In XView 3.0, these key bindings are hard-coded to `L10`, `L6`, `L8`, `L3`, `L4`, `L1`, `Return`, `Tab`, and `Shift-Tab`.

An OPEN LOOK UI toolkit should allow the user to change the mouse buttons used for `SELECT`, `ADJUST`, and `MENU`; the specified default mouse button bindings are `LEFT`, `MIDDLE`, and `RIGHT`.

- In XView 3.0, the specified default mouse button bindings are hard-coded.

An OPEN LOOK UI toolkit should allow the user to change the mouse modifiers used for `SETMENUDEFAULT`, `DUPLICATE`, `PAN`, and `CONSTRAIN`. The specified default modified mouse actions are `Control-RIGHT` for `SETMENUDEFAULT`, `Control-LEFT` for `DUPLICATE`, `Meta-LEFT` for `PAN`, and `LEFT-and-MIDDLE-chorded` for `CONSTRAIN`.

- In XView 3.0, the specified defaults for `SETMENUDEFAULT` and `DUPLICATE` are hard-coded, and `PAN` and `CONSTRAIN` are not supported.

In an OPEN LOOK UI toolkit, clicking `ADJUST` when there is no selection will set an initial insert point (the same as clicking `SELECT`).

- In XView 3.0, this selects a single character.

In an OPEN LOOK UI toolkit, the user can bind a modified mouse action to selecting a single character, but the default binding is `NONE`.

- In XView 3.0, selecting a single character is hard-coded to clicking `ADJUST` when there is no selection.

E.1.2 Default Buttons in Pop Ups

In an OPEN LOOK UI toolkit, notices, command windows, and property windows must always have a "default button," even when there is only one button, and that the `DEFAULT-ACTION` keyboard accelerator always invokes the default button.

- XView 3.0 does not provide this automatically, but applications can implement this feature using XView primitives.

E.1.3 Help

In an OPEN LOOK UI toolkit, help text can have bold text, italic text, and glyphs (small pictures).

- XView 3.0 does not support this.

E.1.4 Window Background

In an OPEN LOOK UI toolkit, window backgrounds are used to access the Window menu, to select a window, and to move it by dragging.

- XView 3.0 does not support this.

E.1.5 Notices

In an OPEN LOOK UI toolkit, notices do not freeze the screen; input to other applications is always possible.

- In XView 3.0, making notices that freeze the screen is determined by the user (application programmer).

In an OPEN LOOK UI toolkit, each window of an application displays the standard busy pattern in the header when a notice is displayed.

- XView 3.0 this is determined by the user (application programmer).

E.1.6 Text Functions

In an OPEN LOOK UI toolkit, an `UNDO` after an `UNDO` reverses the effect of the `UNDO`, restoring the original state.

- In XView 3.0, the second `UNDO` undoes the next-previous edit. There is no way in XView 3.0 to reverse the effect of an `UNDO`.

E.1.7 Control Items

In an OPEN LOOK UI toolkit, an abbreviated menu button can have a text field to the right of the menu button. The text field is used to add items to the menu.

- XView 3.0 the application level supports this behavior as an option.

In an OPEN LOOK UI toolkit, menu buttons (and abbreviated menu buttons) highlight on `MENU` down, and change to the standard busy pattern on `MENU-up` in stay-up mode.

- XView 3.0 does not provide this.

In an OPEN LOOK UI toolkit, a default setting for exclusive and non-exclusive settings is only displayed when the controls are used on a menu.

- XView 3.0 indicates a default setting for exclusive and non-exclusive settings even when the controls are used in command and property windows.

In an OPEN LOOK UI toolkit, an indeterminate state is defined on exclusive and non-exclusive settings.

- XView 3.0 does not support this.

In an OPEN LOOK UI toolkit, the bold border width on exclusive and non-exclusive settings is adjusted (to either 2 or 3 pixels) depending on the display resolution.

- XView 3.0 does not provide this.

E.1.8 Property Windows

In an OPEN LOOK UI toolkit, there is a required Settings pop-up menu for a property window.

- XView 3.0 does not provide this automatically, but applications can implement this menu themselves.

In an OPEN LOOK UI toolkit, property windows have two required buttons, `Apply` and `Reset`, and an optional button, `Set Default`.

- Again, XView 3.0 does not provide this automatically, but applications can implement this feature using XView primitives.

In an OPEN LOOK UI toolkit, there is a way to have two active selections when using property windows.

- XView 3.0 does not support this.

In an OPEN LOOK UI toolkit, when there are two active selections, the `Apply` button becomes a menu button with `Original Selection` and `New Selection` items.

- XView 3.0 does not support this.

E.2 Level 2 Features Supported in XView 3.0

The following Level 2 features are supported in XView 3.0.

- Abbreviated buttons.
- Nonstandard basic windows.
- Numeric text fields with increment/decrement buttons.
- Some keyboard accelerators.
- Splittable panes.
  - *Missing*: Dimming the pane's border or its contents to indicate that the pane is about to disappear when removing a split pane using cable anchors.
- Split View and Join Views items on Scrollbar menu.
- Dragging text to move/copy.

- Quick Move and Quick Duplicate on (most) text.
- Some Level 2 Workspace Properties.
- Blocking pop-up windows.
- Multi-line text areas.
- Read-only gauges.
- Automatic scrolling.
- View must be updated while scrollbar elevator is dragged.
- Glyphs in scrolling lists.
- Sliders with end-boxes and tickmarks.
- Vertical sliders.
- Soft function keys.
- Window scaling.

E.3 Level 2 Features Not Supported in XView 3.0

The following Level 2 features are not supported in XView 3.0.

- Change bars in property windows.
- Edit menu for text fields.
- Menus containing more than one type of control.
- Resizable panes.
- Selectable panes.
- Minimum scrollbar.
- Page-oriented scrollbar.
- Panning.
- Hierarchical scrolling lists.

F
Example Programs

This appendix contains nine example programs that supplement the programs in the chapters:

- *item_move.c*
- *scroll_cells2.c*
- *menu_dir2.c*
- *type_font.c*
- *fonts.c*
- *x_draw.c*
- *Logo.c*
- *Bitmap.c*
- *panel_dnd.c*

Some of these programs are extensions to programs presented earlier in this book; they are listed here to demonstrate extended usage. Other programs in this appendix attempt to integrate features from unrelated XView packages that exceeded the scope of a particular chapter.

F.1 item_move.c

The first program demonstrates how you can use an event handler within a panel to allow the user to create panel items, move them around within the panel, and delete them. Chapter 7, *Panels*, discusses specifics about how panels work. Chapter 5, *Canvases and Openwin*, discusses the canvas and openwin issues, since the panel is subclassed from those packages.

Example F-1. The item_move.c program

```
/*
 * item_move.c
 *    Move items around in a panel using an interpose event handler
 * specific to the panel.  Two panels are created -- the left panel
 * contains panel buttons that allow you to create certain types of
 * items that are put in the second panel.  Use the MENU (right)
 * mouse button to move items around in the second panel.
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/panel.h>

/* We need handles to the base frame and a panel -- instead of
 * using global variables, we're going to attach the objects to
 * the objects which need to reference them.  Attach using
 * XV_KEY_DATA -- here are the keys.
 */
#define PACKAGE_KEY     100
#define FRAME_KEY       101
#define PANEL_KEY       102

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame         frame;
    Panel         panel;
    Xv_Window          window;
    Panel_item    create_text, item;
    Notify_value  my_event_proc();
    int           create_item();
    char          buf[64];

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    sprintf(buf, "%s:  Use MENU (Right) Button To Move Items", argv[0]);
    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            buf,
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);
    /*
     * Create panel for known panel items.  Layout panel vertically.
     */
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    /*
     * Create text panel item, attach the frame as client data for
     * use by the notify procedure create_item().  Text items inherit
     * the layout of "label" and "value" from its parent panel.
     * override for the text item by setting PANEL_LAYOUT explicitly.
     */
    create_text = (Panel_item)xv_create(panel, PANEL_TEXT,
      XV_X,             0,
      XV_Y,             20,
        PANEL_LABEL_STRING,     "Create Button:",
        PANEL_NOTIFY_PROC,      create_item,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_VALUE_DISPLAY_LENGTH,     10,
        NULL);
    /*
     * Create panel button to determine which type of button to create --
     * a button, message, or text item.  See create_item().
     */
    item = (Panel_item)xv_create(panel, PANEL_CHOICE,
      XV_X,             0,
      XV_Y,             50,
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_LABEL_STRING,     "Item type",
        PANEL_CHOICE_STRINGS,   "Button", "Message", "Text", NULL,
        NULL);
    window_fit(panel);

    /* Create a new panel to be used for panel creation.  The panel
     * from above is no longer referenced.  The panel created here
     * is the panel used throughout the rest of this program.  To
     * add confusion, "panel" is used as the handle of this panel, too.
     * The panel referenced in WIN_RIGHT_OF and XV_HEIGHT is the old
     * one since the new one hasn't been created yet.
     */
    panel = (Panel)xv_create(frame, PANEL,
        WIN_RIGHT_OF,           canvas_paint_window(panel),
        XV_WIDTH,               300,
        XV_HEIGHT,              xv_get(panel, XV_HEIGHT),
        WIN_BORDER,          TRUE,
        XV_KEY_DATA,            PANEL_KEY,      panel,
        NULL);

    /* Install event handling routine for the panel.  This must be done
     *      by an interpose function to make sure that the ACTION_MENU
     *      event is not consumed by the first panel before it has a chance
     *      to get to the second panel's event proc:  my_event_proc.
     */
    notify_interpose_event_func(panel, my_event_proc, NOTIFY_SAFE);

    /* attach various items to the text item for text_select() */
    xv_set(create_text,
        XV_KEY_DATA,            FRAME_KEY,      frame,
        XV_KEY_DATA,            PACKAGE_KEY,    item,
        XV_KEY_DATA,            PANEL_KEY,      panel,
        NULL);
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * Process events for panel's subwindow.  This routine gets -all-
 * events that occur in the panel subwindow but passes them on to
 * the normal event dispatcher when the interposed function has been
 * completed.  The notify function, my_event_proc, is only
 * interested in MENU button events that happen on top of panel items.
 * When the user clicks and _drags_ the MENU button on a panel item,
 * the item is moved to where the mouse moves to.
 */
Notify_value
my_event_proc(panel, event, arg, type)
Panel                   panel;
```

```
Event            *event;
Notify_arg        arg;
Notify_event_type type;
{
    static Panel_item   item;
    static int  x_offset, y_offset;
    Frame       frame = (Frame)xv_get(panel, XV_OWNER);
    Rect        *rect, *item_rect;
    char        buf[64];

    /*
     * If the mouse is dragging an item, reset its new location.
     */
    if (event_action(event) == LOC_DRAG && item) {
        Panel_item pi;
        Rect r;
        /*
         * Get the rect of item, then *copy* it -- never change data
         * returned by xv_get().  Modify the copied rect reflecting
         * new X,Y position of panel item and check to see if it
         * intersects with any existing panel items.
         */
        rect = (Rect *)xv_get(item, XV_RECT);
        rect_construct(&r, /* see <xview/rect.h> for macros */
            rect->r_left, rect->r_top, rect->r_width, rect->r_height);
        r.r_left = event->ie_locx - x_offset;
        r.r_top = event->ie_locy - y_offset;
        PANEL_EACH_ITEM(panel, pi)
            if (pi == item)
                continue;
            /* don't let panel items overlap */
            item_rect = (Rect *)xv_get(pi, XV_RECT);
            if (rect_intersectsrect(item_rect, &r))
                return;
        PANEL_END_EACH
        /* no overlap -- move panel item. */
        xv_set(item,
            PANEL_ITEM_X, r.r_left,
            PANEL_ITEM_Y, r.r_top,
            NULL);
    }

    /* If it's not the MENU button, we're not interested,
     * so allow the event to be passed on to the notifier
     * for normal event handling.
     */
    if (event_action(event) != ACTION_MENU) {
      notify_next_event_func(panel, (Notify_event) event, arg, type);
      return;
    }

    /*
     * next two cases is MENU button just-down or just-released
     */
    if (event_is_down(event)) {
        /* Right (MENU) button down on an item -- determine panel item */
```

Example F-1. The item_move.c program (continued)

```
      if ( (xv_get((panel), PANEL_FIRST_ITEM) ) == NULL ) {
              sprintf(buf, "No panel items are currently in the panel.");
            xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);
      }
        PANEL_EACH_ITEM(panel, item)
            rect = (Rect *)xv_get(item, XV_RECT);
            if (rect_includespoint(rect,
                event->ie_locx, event->ie_locy)) {
                x_offset = event->ie_locx - rect->r_left;
                y_offset = event->ie_locy - rect->r_top;
              sprintf(buf, "Panel item found.");
            xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);
                break;
            }
          else {
              sprintf(buf, "The cursor is not over any panel item.");
            xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);
            }
        PANEL_END_EACH
        if (item)
            sprintf(buf, "Moving item: '%s'",
                (char *)xv_get(item, PANEL_LABEL_STRING));
        else
            buf[0] = 0;
    } else if (item) {
        char *name = (char *)xv_get(item, PANEL_LABEL_STRING);

        /* test if item is inside panel by comparing XV_RECTs */
        rect = (Rect *)xv_get(panel, XV_RECT);
        if (!rect_includespoint(rect,
            event->ie_locx + rect->r_left,
            event->ie_locy + rect->r_top)) {
            /* item is outside the panel -- remove item */
            xv_destroy(item);
            sprintf(buf, "Removed '%s' from panel", name);
        } else
            sprintf(buf, "'%s' moved to %d %d", name,
                (int)xv_get(item, XV_X), (int)xv_get(item, XV_Y));
        /* set "item" to null so that new drag
         * events don't attempt to move old item.
         */
        item = NULL;
    }
    xv_set(frame, FRAME_LEFT_FOOTER, buf, NULL);
}

/*
 * Callback routine for all panel buttons.
 * If the panel item is the text item, determine the name of the new
 * panel button the user wishes to create.  Loop through all the
 * existing panel items looking for one with the same label.  If so,
 * return PANEL_NONE and set the frame's footer with an error message.
 * Otherwise, create a new panel item with the label, reset the text
 * item value and return PANEL_NEXT.
 */
int
```

Example F-1. The item_move.c program (continued)

```
create_item(item, event)
Panel_item item;
Event *event;
{
    Xv_pkg      *pkg;
    Panel       panel = (Panel)xv_get(item, XV_KEY_DATA, PANEL_KEY);
    Frame       frame = (Frame)xv_get(item, XV_KEY_DATA, FRAME_KEY);
    Panel_item  pi, pkg_item;
    char        buf[64];
    int         selected();

    pkg_item = (Panel_item)xv_get(item, XV_KEY_DATA, PACKAGE_KEY);
    (void) strncpy(buf, (char *)xv_get(item, PANEL_VALUE), sizeof buf);
    if (!buf[0])
        return PANEL_NONE;
    switch((int)xv_get(pkg_item, PANEL_VALUE)) {
        case 1: pkg = PANEL_MESSAGE; break;
        case 2: pkg = PANEL_TEXT; break;
        default: pkg = PANEL_BUTTON;
    }
    /* loop thru all panel items and check for item with same name */
    PANEL_EACH_ITEM(panel, pi)
        if (!strcmp(buf, (char *)xv_get(pi, PANEL_LABEL_STRING))) {
            xv_set(frame, FRAME_LEFT_FOOTER, "Label Taken", NULL);
            return PANEL_NONE;
        }
    PANEL_END_EACH
    (void) xv_create(panel, pkg,
        PANEL_LABEL_STRING,     buf,
        PANEL_NOTIFY_PROC,      selected,
        XV_KEY_DATA,            FRAME_KEY,      frame,
        /* only for text items, but doesn’t affect other items */
        PANEL_VALUE_DISPLAY_LENGTH, 10,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        NULL);
    xv_set(item, PANEL_VALUE, "", NULL);
    return PANEL_NEXT;
}

/*
 * For panel buttons. return XV_OK or XV_ERROR if the item was
 * selected using the left mouse button or not.
 */
int
selected(item, event)
Panel_item item;
Event *event;
{
    Frame       frame = (Frame)xv_get(item, XV_KEY_DATA, FRAME_KEY);
    char        buf[64];

    if (event_action(event) == ACTION_SELECT) {
        sprintf(buf, "’%s’ selected", xv_get(item, PANEL_LABEL_STRING));
        xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);
        return XV_OK;
    }
    return XV_ERROR;
}
```

F.2 scroll_cells2.c

This program is based heavily on *scroll_cells.c*, which is found in Chapter 10, *Scrollbars*. This version of the program deals with resize events. When a resize occurs, the `object`, `view`, and `page` length attributes are set to correctly reflect the size of the scrollbar with respect to the object it scrolls.

Example F-2. The scroll_cells2.c program

```
/*
 * scroll_cells2.c -- scroll a bitmap of cells around in a canvas.
 * This is a simplified version of scroll_cells.c graphically. That
 * is, it does not display icons, just rows and columns of cells.
 * The difference with this version is that it attempts to accommodate
 * resize events not addressed in the scroll_cells.c.
 * This new function is at the end of the file.
 */
#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>   /* Using Xlib graphics */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>
#include <xview/xv_xrect.h>

#define CELL_WIDTH              64
#define CELL_HEIGHT             64
#define CELLS_PER_HOR_PAGE      5 /* when paging w/scrollbar */
#define CELLS_PER_VER_PAGE      5 /* when paging w/scrollbar */
#define CELLS_PER_ROW           16
#define CELLS_PER_COL           16

Pixmap          cell_map;       /* pixmap copied onto canvas window */
Scrollbar       horiz_scrollbar;
Scrollbar       vert_scrollbar;
GC              gc;             /* General usage GC */

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame       frame;
    Canvas      canvas;
    void        repaint_proc(), resize_proc();

    /* Initialize, create frame and canvas... */
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

Example F-2. The scroll_cells2.c program (continued)

```
    frame = xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);

    canvas = xv_create(frame, CANVAS,
        /* make subwindow the size of a "page" */
        XV_WIDTH,               CELL_WIDTH * CELLS_PER_HOR_PAGE,
        XV_HEIGHT,              CELL_HEIGHT * CELLS_PER_VER_PAGE,
        /* canvas is same size as window */
        CANVAS_WIDTH,           CELL_WIDTH * CELLS_PER_HOR_PAGE,
        CANVAS_HEIGHT,          CELL_HEIGHT * CELLS_PER_VER_PAGE,
        /* don't retain window -- we'll repaint it all the time */
        CANVAS_RETAINED,        FALSE,
        /* We're using Xlib graphics calls in repaint_proc() */
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    repaint_proc,
        CANVAS_RESIZE_PROC,     resize_proc,
        OPENWIN_AUTO_CLEAR,     FALSE,
        NULL);

    /*
     * Create scrollbars attached to the canvas. When user clicks
     * on cable, page by the page size (PAGE_LENGTH). Scrolling
     * should move cell by cell, not by one pixel (PIXELS_PER_UNIT).
     */
    vert_scrollbar = xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,            SCROLLBAR_VERTICAL,
        SCROLLBAR_PIXELS_PER_UNIT,      CELL_HEIGHT,
        NULL);
    horiz_scrollbar = xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,            SCROLLBAR_HORIZONTAL,
        SCROLLBAR_PIXELS_PER_UNIT,      CELL_WIDTH,
        NULL);

    /*
     * create pixmap and draw cells into it.  This portion of the
     * program could use XCopyArea to render real bitmaps whose sizes
     * do not exceed whatever CELL_WIDTH and CELL_HEIGHT are defined
     * to be.  The cell_map will be copied into the window via
     * XCopyPlane in the repaint procedure.
     */
    {
        short           x, y, pt = 0;
        XPoint          points[256];
        XGCValues       gcvalues;
        Display *dpy = (Display *)xv_get(canvas, XV_DISPLAY);

        cell_map = XCreatePixmap(dpy, DefaultRootWindow(dpy),
            CELLS_PER_ROW * CELL_WIDTH + 1,
            CELLS_PER_COL * CELL_HEIGHT + 1,
            1); /* We only need a 1-bit deep pixmap */

        /* Create the gc for the cell_map -- since it is 1-bit deep,
         * use 0 and 1 for fg/bg values.  Also, limit number of
         * events generated by setting graphics exposures to False.
         */
        gcvalues.graphics_exposures = False;
        gcvalues.background = 0;
        gcvalues.foreground = 1;
        gc = XCreateGC(dpy, cell_map,
            GCForeground|GCBackground|GCGraphicsExposures, &gcvalues);

        /* dot every other pixel */
        for (x = 0; x <= CELL_WIDTH * CELLS_PER_ROW; x += 2)
            for (y = 0; y <= CELL_HEIGHT * CELLS_PER_COL; y += 2) {
                if (x % CELL_WIDTH != 0 && y % CELL_HEIGHT != 0)
                    continue;
                points[pt].x = x, points[pt].y = y;
                if (++pt == sizeof points / sizeof points[0]) {
                    XDrawPoints(dpy, cell_map, gc,
                        points, pt, CoordModeOrigin);
                    pt = 0;
                }
            }
        if (pt != sizeof points) /* flush out the remaining points */
            XDrawPoints(dpy, cell_map, gc,
                        points, pt, CoordModeOrigin);
        /* label each cell indicating the its coordinates */
        for (x = 0; x < CELLS_PER_ROW; x++)
            for (y = 0; y < CELLS_PER_COL; y++) {
                char buf[8];
                sprintf(buf, "%d,%d", x+1, y+1);
                XDrawString(dpy, cell_map, gc,
                    x * CELL_WIDTH + 5, y * CELL_HEIGHT + 25,
                    buf, strlen(buf));
            }
        /* we're now done with the cell_map, so free gc and
         * create a new one based on the window that will use it.
         */
        XFreeGC(dpy, gc);
        gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
        gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
        gcvalues.plane_mask = 1L;
        gc = XCreateGC(dpy, DefaultRootWindow(dpy),
            GCForeground|GCBackground|GCGraphicsExposures, &gcvalues);
    }

    /* shrink frame to minimal size and start notifier */
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * The repaint procedure is called whenever repainting is needed in
 * a paint window.  Since the canvas is not retained, this routine
 * is going to be called any time the user scrolls the canvas.  The
 * canvas will handle repainting the portion of the canvas that
 * was in view and has scrolled onto another viewable portion of
 * the window.  The xrects parameter will cover the new areas that
 * were not in view before and have just scrolled into view.  If
 * the window resizes or if the window is exposed by other windows
 * disappearing or cycling thru the window tree, then the number
 * of xrects will be more than one and we'll have to copy the new
 * areas one by one.  Clipping isn't necessary since the areas to
 * be rendered are set by the xrects value.
 */
void
repaint_proc(canvas, paint_window, dpy, win, xrects)
Canvas          canvas;
Xv_Window       paint_window;
Display         *dpy;
Window          win;
Xv_xrectlist    *xrects;
{
    int x, y;

    x = (int)xv_get(horiz_scrollbar, SCROLLBAR_VIEW_START);
    y = (int)xv_get(vert_scrollbar, SCROLLBAR_VIEW_START);

    for (xrects->count--; xrects->count >= 0; xrects->count--) {
        printf("top-left cell = %d, %d -- %d,%d %d,%d0, x+1, y+1,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y,
            xrects->rect_array[xrects->count].width,
            xrects->rect_array[xrects->count].height);

        XCopyPlane(dpy, cell_map, win, gc,
            x * CELL_WIDTH,
            y * CELL_HEIGHT,
            xv_get(paint_window, XV_WIDTH),
            xv_get(paint_window, XV_HEIGHT),
            0, 0, 1L);
    }
}
/*
 * If the application is resized, then we may wish to reset the
 * paging and viewing parameters for the scrollbars.
 */
void
resize_proc(canvas, new_width, new_height)
Canvas canvas;
int new_width, new_height;
{
    int page_w = (int)(new_width/CELL_WIDTH);
    int page_h = (int)(new_height/CELL_HEIGHT);

    if (!vert_scrollbar || !horiz_scrollbar)
        return;

    printf("new width/height in cells: w = %d, h = %d0,
        page_w, page_h);

    xv_set(horiz_scrollbar,
        SCROLLBAR_OBJECT_LENGTH,        CELLS_PER_ROW,
        SCROLLBAR_PAGE_LENGTH,          page_w,
        SCROLLBAR_VIEW_LENGTH,          page_w,
        NULL);
```

Example F-2. The scroll_cells2.c program (continued)

```
    xv_set(vert_scrollbar,
        SCROLLBAR_OBJECT_LENGTH,        CELLS_PER_COL,
        SCROLLBAR_PAGE_LENGTH,          page_h,
        SCROLLBAR_VIEW_LENGTH,          page_h,
        NULL);
}
```

F.3 menu_dir2.c

In Chapter 11, *Menus*, the program *menu_dir.c* demonstrates the use of an XView menu in a canvas subwindow. A menu is brought up with the MENU mouse button and displays menu choices representing the files in the directory. The problem with *menu_dir.c* is that the entire menu cascade is created for all the subdirectories at the very beginning of the program. If the directory stack is very deep, it could take a very long time to build. You could also run out of memory in the process. Further, if the contents of the directory tree is dynamic, the menu entries could become invalid over time.

These problems are solved in *menu_dir2.c* because it creates only the top-level menu. For each directory entry under the top-level, rather than creating an associated pullright menu, a `MENU_GEN_PULLRIGHT` procedure is specified. This routine creates that menu only at the time it is needed. So, when the user invokes the menu and tries to descend into a submenu representing a subdirectory in the directory tree, only then is the directory entry searched and a new submenu created. When the user backs out of the menu, the menu is destroyed—attempting to re-enter the submenu causes the process to be repeated.

An exercise for the ambitious programmer would be to modify this program so that the submenus are not destroyed until the entire menu cascade has been dismissed. This enhancement would optimize the perceived performance of the program for the user because, once a directory subpath has been searched and a menu created, the menu is cached so that re-entry into the same submenu would be instantaneous.

Example F-3. The menu_dir2.c program

```
/*
 * menu_dir2.c -
 * Demonstrate the use of an XView menu in a canvas subwindow.
 * A menu is brought up with the MENU mouse button and displays
 * menu choices representing the files in the directory.  If a
 * directory entry is found, a new pullright item is created with
 * that subdir as the pullright menu’s contents.  This implementation
 * creates directories on an as-needed basis.  Thus, we provide a
 * MENU_GEN_PULLRIGHT procedure.
 *
 * argv[1] indicates which directory to start from.
 */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <sys/stat.h>
#include <sys/dir.h>
```

Example F-3. The menu_dir2.c program (continued)

```
#include <X11/Xos.h>
#ifndef MAXPATHLEN
#include <sys/param.h>
#endif /* MAXPATHLEN */

Frame   frame;

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Canvas      canvas;
    extern void exit();
    void        my_event_proc();
    Menu        menu;
    Menu_item   mi, add_path_to_menu();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME,
        FRAME_LABEL,            argv[1]? argv[1] : "cwd",
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);
    canvas = (Canvas)xv_create(frame, CANVAS,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       400,
        XV_HEIGHT,      100,
        NULL);

    mi = add_path_to_menu(argc > 1? argv[1] : ".");
    menu = (Menu)xv_get(mi, MENU_PULLRIGHT);
    /* We no longer need the item since we have the menu from it */
    xv_destroy(mi);

    /* associate the menu to the canvas win for easy etreival */
    xv_set(canvas_paint_window(canvas),
        WIN_CONSUME_EVENTS,     WIN_MOUSE_BUTTONS, NULL,
        WIN_EVENT_PROC,         my_event_proc,
        WIN_CLIENT_DATA,        menu,
        NULL);

    window_fit(frame);
    window_main_loop(frame);
}

/*
 * my_action_proc - display the selected item in the frame footer.
 */
void
my_action_proc(menu, menu_item)
Menu    menu;
Menu_item       menu_item;
{
    xv_set(frame,
        FRAME_LEFT_FOOTER,      xv_get(menu_item, MENU_STRING),
        NULL);
```

Example F-3. The menu_dir2.c program (continued)

```
}

/*
 * Call menu_show() to display menu on right mouse button push.
 */
void
my_event_proc(canvas, event)
Canvas  canvas;
Event *event;
{
    if ((event_id(event) == MS_RIGHT) && event_is_down(event)) {
        Menu menu = (Menu)xv_get(canvas, WIN_CLIENT_DATA);
        menu_show(menu, canvas, event, NULL);
    }
}

/*
 * return an allocated char * that points to the last item in a path.
 */
char *
getfilename(path)
char *path;
{
    char *p;

    if (p = rindex(path, '/'))
        p++;
    else
        p = path;
    return strcpy(malloc(strlen(p)+1), p);
}

/* gen_pullright() is called in the following order:
 *   Pullright menu needs to be displayed. (MENU_PULLRIGHT)
 *   Menu is about to be dismissed (MENU_DISPLAY_DONE)
 *      User made a selection (before menu notify function)
 *      After the notify routine has been called.
 * The above order is done whether or not the user makes a
 * menu selection.
 */
Menu
gen_pullright(mi, op)
Menu_item mi;
Menu_generate op;
{
    Menu menu;
    Menu_item new, old = mi;
    char buf[MAXPATHLEN];

    if (op == MENU_DISPLAY) {
        menu = (Menu)xv_get(mi, MENU_PARENT);
        sprintf(buf, "%s/%s",
            xv_get(menu, MENU_CLIENT_DATA), xv_get(mi, MENU_STRING));
        /* get old menu and free it -- we're going to build another */
        if (menu = (Menu)xv_get(mi, MENU_PULLRIGHT)) {
            free(xv_get(menu, MENU_CLIENT_DATA));
            xv_destroy(menu);
        }
        if (new = add_path_to_menu(buf)) {
            menu = (Menu)xv_get(new, MENU_PULLRIGHT);
            xv_destroy(new);
            return menu;
        }
    }
    if (!(menu = (Menu)xv_get(mi, MENU_PULLRIGHT)))
            menu = (Menu)xv_create(NULL, MENU,
                MENU_STRINGS, "Couldn't build a menu.", NULL,
                NULL);
    return menu;
}

/*
 * The path passed in is scanned via readdir().  For each file in the
 * path, a menu item is created and inserted into a new menu.  That
 * new menu is made the PULLRIGHT_MENU of a newly created panel item
 * for the path item originally passed it.  Since this routine is
 * recursive, a new menu is created for each subdirectory under the
 * original path.
 */
Menu_item
add_path_to_menu(path)
char *path;
{
    DIR                 *dirp;
    struct direct       *dp;
    struct stat         s_buf;
    Menu_item           mi;
    Menu                next_menu;
    char                buf[MAXPATHLEN];
    static int          recursion;

    /* don't add a folder to the list if user can't read it */
    if (stat(path, &s_buf) == -1 || !(s_buf.st_mode & S_IREAD))
        return NULL;
    if (s_buf.st_mode & S_IFDIR) {
        int cnt = 0;
        if (!(dirp = opendir(path)))
            /* don't bother adding to list if we can't scan it */
            return NULL;
        if (recursion)
            return (Menu_item)-1;
        recursion++;
        next_menu = (Menu)xv_create(XV_NULL, MENU, NULL);
        while (dp = readdir(dirp))
            if (strcmp(dp->d_name, ".") && strcmp(dp->d_name, "..")) {
                (void) sprintf(buf, "%s/%s", path, dp–>d_name);
                mi = add_path_to_menu(buf);
                if (!mi || mi == (Menu_item)-1) {
                    int do_gen_pullright = (mi == (Menu_item)-1);
                    /* unreadable file or dir - deactivate item */
                    mi = (Menu_item)xv_create(XV_NULL, MENUITEM,
                        MENU_STRING,  getfilename(dp->d_name),
```

Example F-3. The menu_dir2.c program (continued)

```
                        MENU_RELEASE,
                        MENU_RELEASE_IMAGE,
                        NULL);
                    if (do_gen_pullright)
                        xv_set(mi,
                            MENU_GEN_PULLRIGHT, gen_pullright,
                            NULL);
                    else
                        xv_set(mi, MENU_INACTIVE, TRUE, NULL);
                }
                xv_set(next_menu, MENU_APPEND_ITEM, mi, NULL);
                cnt++;
            }
        closedir(dirp);
        mi = (Menu_item)xv_create(XV_NULL, MENUITEM,
            MENU_STRING,        getfilename(path),
            MENU_RELEASE,
            MENU_RELEASE_IMAGE,
            MENU_NOTIFY_PROC,   my_action_proc,
            NULL);
        if (!cnt) {
            xv_destroy(next_menu);
            /* An empty or unsearchable directory - deactivate item */
            xv_set(mi, MENU_INACTIVE, TRUE, NULL);
        } else {
            xv_set(next_menu,
                MENU_TITLE_ITEM, strcpy(malloc(strlen(path)+1), path),
                MENU_CLIENT_DATA, strcpy(malloc(strlen(path)+1), path),
                NULL);
            xv_set(mi, MENU_PULLRIGHT, next_menu, NULL);
        }
        recursion--;
        return mi;
    }
    return (Menu_item)xv_create(NULL, MENUITEM,
        MENU_STRING,            getfilename(path),
        MENU_RELEASE,
        MENU_RELEASE_IMAGE,
        MENU_NOTIFY_PROC,       my_action_proc,
        NULL);
}
```

F.4 type_font.c

This very simple program captures keyboard events in a canvas and uses `XDrawString()` to render what the user types. It also looks for backspacing. This is not intended to replace the text subwindow in any way, but rather to demonstrate how some text rendering functions can be implemented. See Chapter 16, *Fonts*, for more information about fonts.

Example F-4. The type_font.c program

```
/*
 * simple_font.c -- very simple program showing how to render text
 * using fonts loaded by XView.
 */
#include <ctype.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/font.h>

Display *dpy;
GC      gc;
XFontStruct *font_info;

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Frame       frame;
    Panel       panel;
    Canvas      canvas;
    XGCValues   gcvalues;
    Xv_Font     font;
    void        my_event_proc();
    extern void exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);

    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    window_fit(panel);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               400,
        XV_HEIGHT,              200,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        NULL);
    xv_set(canvas_paint_window(canvas),
        WIN_EVENT_PROC,         my_event_proc,
        NULL);

    window_fit(frame);

    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    font = (Xv_Font)xv_get(frame, XV_FONT);
    font_info = (XFontStruct *)xv_get(font, FONT_INFO);
    gcvalues.font = (Font)xv_get(font, XV_XID);
    gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
    gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
    gcvalues.graphics_exposures = False;
    gc = XCreateGC(dpy, RootWindow(dpy, DefaultScreen(dpy)),
        GCForeground | GCBackground | GCFont | GCGraphicsExposures, &gcvalues);

    xv_main_loop(frame);
}

void
my_event_proc(win, event)
Xv_Window win;
Event *event;
{
    static int x = 10, y = 10;
    Window xwin = (Window)xv_get(win, XV_XID);
    char c;

    if (event_is_up(event))
        return;

    if (event_is_ascii(event)) {
        c = (char)event_id(event);
        if (c == ’0 || c == ’) {
            y += font_info->max_bounds.ascent +
                        font_info->max_bounds.descent;
            x = 10;
        } else if (c == 7 || c == 127) { /* backspace or delete */
            if (x > 10)
                x -= XTextWidth(font_info, "m", 1);
            /* use XDrawImageString to overwrite previous text */
            XDrawImageString(dpy, xwin, gc, x, y, "  ", 2);
        } else {
            XDrawString(dpy, xwin, gc, x, y, &c, 1);
            x += XTextWidth(font_info, &c, 1);
        }
    } else if (event_action(event) == ACTION_SELECT) {
        x = event_x(event);
        y = event_y(event);
    }
}
```

F.5 fonts.c

This program is similar to *type_font.c* above. However, *fonts.c* provides an interface for the user to pick and choose from a subset of the font families and styles available on the X server. If a font "name" is specified, then the family, style and size choices are ignored. Using the SELECT button on the canvas window positions the current typing location at the *x,y* coordinates of the button-down event. The characters typed are printed in the current font.

Example F-5. The fonts.c program

```
/*
 * fonts.c -- provide an interface for the user to pick and choose
 * between font families and styles known to XView.  The program
 * provides several panel buttons to choose between font types, and
 * a canvas window in which the user can type.  The characters typed
 * are printed in the current font.  If a font "name" is specified,
 * then the family, style and size are ignored.  Using the SELECT
 * button on the canvas window positions the current typing location
 * at the x,y coordinates of the button-down event.
 */
#include <ctype.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/font.h>

Display *dpy;
GC      gc;
XFontStruct *cur_font;
Panel_item family_item, style_item, scale_item, name_item;
int canvas_width;

main(argc, argv)
int     argc;
char    *argv[];
{
    Frame       frame;
    Panel       panel;
    Canvas      canvas;
    XGCValues   gcvalues;
    Xv_Font     font;
    void        change_font();
    void        my_event_proc(), my_resize_proc();
    int         change_font_by_name();
    extern void exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);
```

Example F-5. The fonts.c program (continued)

```
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    family_item = (Panel_item)xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING,     "Family",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_CHOICE_STRINGS,
            FONT_FAMILY_DEFAULT, FONT_FAMILY_DEFAULT_FIXEDWIDTH,
            FONT_FAMILY_LUCIDA, FONT_FAMILY_LUCIDA_FIXEDWIDTH,
            FONT_FAMILY_ROMAN, FONT_FAMILY_SERIF, FONT_FAMILY_COUR,
            FONT_FAMILY_CMR, FONT_FAMILY_GALLENT,
            FONT_FAMILY_OLGLYPH, FONT_FAMILY_OLCURSOR, NULL,
        PANEL_NOTIFY_PROC,      change_font,
        NULL);
    style_item = (Panel_item)xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING,     "Style",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_CHOICE_STRINGS,
            FONT_STYLE_DEFAULT, FONT_STYLE_NORMAL, FONT_STYLE_BOLD,
            FONT_STYLE_ITALIC, FONT_STYLE_BOLD_ITALIC, NULL,
        PANEL_NOTIFY_PROC,      change_font,
        NULL);
    scale_item = (Panel_item)xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING,     "Scale",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_CHOICE_STRINGS,
            "Small", "Medium", "Large", "X-Large", NULL,
        PANEL_NOTIFY_PROC,      change_font,
        NULL);
    name_item = (Panel_item)xv_create(panel, PANEL_TEXT,
        PANEL_LABEL_STRING,     "Font Name:",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_VALUE_DISPLAY_LENGTH, 20,
        PANEL_NOTIFY_PROC,      change_font_by_name,
        NULL);
    window_fit(panel);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               400,
        XV_HEIGHT,              200,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_RESIZE_PROC,     my_resize_proc,
        NULL);
    xv_set(canvas_paint_window(canvas),
        WIN_EVENT_PROC,         my_event_proc,
        WIN_CONSUME_EVENT,      LOC_WINENTER,
        NULL);

    window_fit(frame);
    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    font = (Xv_Font)xv_get(frame, XV_FONT);
    cur_font = (XFontStruct *)xv_get(font, FONT_INFO);
    xv_set(frame, FRAME_LEFT_FOOTER, xv_get(font, FONT_NAME), NULL);

    gcvalues.font = cur_font->fid;
    gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
    gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
    gcvalues.graphics_exposures = False;
    gc = XCreateGC(dpy, RootWindow(dpy, DefaultScreen(dpy)),
        GCForeground | GCBackground | GCFont | GCGraphicsExposures,
        &gcvalues);

    xv_main_loop(frame);
}

void
my_event_proc(win, event)
Xv_Window win;
Event *event;
{
    static int x = 10, y = 10;
    Window xwin = (Window)xv_get(win, XV_XID);
    char c;

    if (event_is_up(event))
        return;

    if (event_is_ascii(event)) {
        c = (char)event_action(event);
        XDrawString(dpy, xwin, gc, x, y, &c, 1);
        /* advance x to next position.  If over edge, linewrap */
        if ((x += XTextWidth(cur_font, &c, 1)) >= canvas_width) {
            y += cur_font->max_bounds.ascent +
                 cur_font->max_bounds.descent;
            x = 10;
        }
    } else if (event_action(event) == ACTION_SELECT) {
        x = event_x(event);
        y = event_y(event);
    } else if (event_action(event) == LOC_WINENTER)
        win_set_kbd_focus(win, xwin);
}

/*
 * check resizing so we know how wide to allow the user to type.
 */
void
my_resize_proc(canvas, width, height)
Canvas canvas;
int width, height;
{
    canvas_width = width;
}
```

Example F-5. The fonts.c program (continued)

```
void
change_font(item, value, event)
Panel_item   item;
Event       *event;
{
    static int  family, style, scale;
    char        buf[128];
    Frame       frame;
    char        *family_name;
    char        *style_name;
    int         scale_value;
    Xv_Font     font;

    frame = (Frame)xv_get(xv_get(item, PANEL_PARENT_PANEL), XV_OWNER);
    family_name = (char *)xv_get(family_item, PANEL_CHOICE_STRING,
                                xv_get(family_item, PANEL_VALUE));
    style_name = (char *)xv_get(style_item, PANEL_CHOICE_STRING,
                                xv_get(style_item, PANEL_VALUE));
    scale_value = (int) xv_get(scale_item, PANEL_VALUE);
    xv_set(frame, FRAME_BUSY, TRUE, NULL);
    font = (Xv_Font)xv_find(frame, FONT,
        FONT_FAMILY,    family_name,
        FONT_STYLE,     style_name,
        /* scale_value happens to coincide with Window_rescale_state */
        FONT_SCALE,     scale_value,
        /*
         * If run on a server that cannot rescale fonts, only font
         * sizes that exist should be passed
         */
        FONT_SIZES_FOR_SCALE, 12, 14, 16, 22,
        NULL);
    xv_set(frame, FRAME_BUSY, FALSE, NULL);

    if (!font) {
        if (item == family_item) {
            sprintf(buf, "cannot load '%s'", family_name);
            xv_set(family_item, PANEL_VALUE, family, NULL);
        } else if (item == style_item) {
            sprintf(buf, "cannot load '%s'", style_name);
            xv_set(style_item, PANEL_VALUE, style, NULL);
        } else {
            sprintf(buf, "Not available in %s scale.",
                xv_get(scale_item, PANEL_CHOICE_STRING, scale));
            xv_set(scale_item, PANEL_VALUE, scale, NULL);
        }
        xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);
        return;
    }
    if (item == family_item)
        family = value;
    else if (item == style_item)
        style = value;
    else
        scale = value;
    cur_font = (XFontStruct *)xv_get(font, FONT_INFO);
    XSetFont(dpy, gc, cur_font->fid);
```

Example F-5. The fonts.c program (continued)

```
    sprintf(buf, "Current font: %s", xv_get(font, FONT_NAME));
    xv_set(frame, FRAME_LEFT_FOOTER, buf, NULL);
}

change_font_by_name(item, event)
Panel_item item;
Event *event;
{
    char buf[128];
    char *name = (char *)xv_get(item, PANEL_VALUE);
    Frame frame = (Frame)xv_get(xv_get(item, XV_OWNER), XV_OWNER);
    Xv_Font font;

    xv_set(frame, FRAME_BUSY, TRUE, NULL);
    font = (Xv_Font)font = (Xv_Font)xv_find(frame, FONT,
        FONT_NAME,      name,
        NULL);
    xv_set(frame, FRAME_BUSY, FALSE, NULL);

    if (!font) {
        sprintf(buf, "cannot load ’%s’", name);
        xv_set(frame, FRAME_RIGHT_FOOTER, buf, NULL);
        return PANEL_NONE;
    }
    cur_font = (XFontStruct *)xv_get(font, FONT_INFO);
    XSetFont(dpy, gc, cur_font->fid);
    sprintf(buf, "Current font: %s", xv_get(font, FONT_NAME));
    xv_set(frame, FRAME_LEFT_FOOTER, buf, NULL);
    return PANEL_NONE;
}
```

F.6 x_draw.c

This program uses several Xlib drawing functions to draw various types of geometric objects. We integrate the XView color model (see Chapter 21, *Color*) to render each object in a different color.

Example F-6. The x_draw.c program

```
/*
 * x_draw.c -- demonstrates the use of Xlib drawing functions
 * inside an XView canvas.  Color is used, but not required.
 */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/cms.h>
#include <xview/xv_xrect.h>

/* indices into color table renders specified colors. */
#define WHITE   0
#define RED     1
#define GREEN   2
```

Example F-6. The x_draw.c program (continued)

```
#define BLUE    3
#define ORANGE  4
#define AQUA    5
#define PINK    6
#define BLACK   7

GC gc;                 /* GC used for Xlib drawing */
unsigned long *colors; /* the color table */

/*
 * initialize cms data to support colors specified above.  Assign
 * data to new cms -- use either static or dynamic cms depending
 * on -dynamic command line switch.
 */
main(argc, argv)
int     argc;
char    *argv[ ];
{
    static char stipple_bits[ ] = {0xAA, 0xAA, 0x55, 0x55};
    static Xv_singlecolor cms_colors[ ] = {
        { 255, 255, 255 },
        { 255, 0, 0 },
        { 0, 255, 0 },
        { 0, 0, 255 },
        { 250, 130, 80 },
        { 30, 230, 250 },
        { 230, 30, 250 },
    };
    Cms         cms;
    Frame       frame;
    Canvas      canvas;
    XFontStruct *font;
    Display     *display;
    XGCValues   gc_val;
    XID         xid;
    void        canvas_repaint();
    Xv_cmsdata  cms_data;
    int         use_dynamic = FALSE;

    /* Create windows */
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    if (*++argv && !strcmp(*argv, "-dynamic"))
        use_dynamic = TRUE;

    frame = xv_create(NULL,FRAME,
        FRAME_LABEL,    "xv_canvas_x_draw",
        XV_WIDTH,       400,
        XV_HEIGHT,      300,
        NULL);

    cms = xv_create(NULL, CMS,
        CMS_SIZE,       7,
        CMS_TYPE,       use_dynamic? XV_DYNAMIC_CMS : XV_STATIC_CMS,
        CMS_COLORS,     cms_colors,
        NULL);
```

Example F-6. The x_draw.c program (continued)

```
    canvas = xv_create(frame, CANVAS,
        CANVAS_REPAINT_PROC,    canvas_repaint,
        CANVAS_X_PAINT_WINDOW,  TRUE,
/*      WIN_DYNAMIC_VISUAL,     use_dynamic,  */
      XV_VISUAL_CLASS,  PseudoColor,
        WIN_CMS,                cms,
        NULL);

    /* Get display and xid */
    display = (Display *)xv_get(frame, XV_DISPLAY);
    xid = (XID)xv_get(canvas_paint_window(canvas), XV_XID);

    if (!(font = XLoadQueryFont(display, "fixed"))) {
        puts("cannot load fixed font");
        exit(1);
    }

    /* Create and initialize GC */
    gc_val.font = font->fid;
    gc_val.stipple =
        XCreateBitmapFromData(display, xid, stipple_bits, 16, 2);
    gc = XCreateGC(display, xid, GCFont | GCStipple, &gc_val);

    /* get the colormap from the canvas now that
     * the cms has been installed
     */
    colors = (unsigned long *)xv_get(canvas, WIN_X_COLOR_INDICES);

    /* Start event loop */
    xv_main_loop(frame);
}

/*
 * Draws onto the canvas using Xlib drawing functions.
 */
void
canvas_repaint(canvas, pw, display, xid, xrects)
Canvas          canvas;
Xv_Window       pw;
Display         *display;
Window          xid;
Xv_xrectlist    *xrects;
{
    static XPoint box[ ] = {
        {0,0}, {100,100}, {0,-100}, {-100,100}, {0,-100}
    };
    static XPoint points[ ] = {
        {0,0}, /* this point to be overwritten below */
        {25,0}, {25,0}, {25,0}, {25,0}, {-100,25},
        {25,0}, {25,0}, {25,0}, {25,0}, {-100,25},
        {25,0}, {25,0}, {25,0}, {25,0}, {-100,25},
        {25,0}, {25,0}, {25,0}, {25,0}, {-100,25},
        {25,0}, {25,0}, {25,0}, {25,0}, {-100,25},
    };

    XSetForeground(display, gc, colors[RED]);
```

Example F-6. The x_draw.c program (continued)

```
    XDrawString(display, xid, gc, 30, 20, "XFillRectangle", 14);
    XFillRectangle(display, xid, gc, 25, 25, 100, 100);
    XSetFunction(display, gc, GXinvert);
    XFillRectangle(display, xid, gc, 50, 50, 50, 50);
    XSetFunction(display, gc, GXcopy);

    XSetForeground(display, gc, colors[BLACK]);
    XDrawString(display, xid, gc, 155, 20, "XFillRect - stipple", 19);
    XSetFillStyle(display, gc, FillStippled);
    XFillRectangle(display, xid, gc, 150, 25, 100, 100);
    XSetFillStyle(display, gc, FillSolid);

    XSetForeground(display, gc, colors[BLUE]);
    XDrawString(display, xid, gc, 280, 20, "XDrawPoints", 11);
    points[0].x = 275; points[0].y = 25;
    XDrawPoints(display, xid, gc, points,
        sizeof(points)/sizeof(XPoint), CoordModePrevious);

    XSetForeground(display, gc, colors[ORANGE]);
    XDrawString(display, xid, gc, 30, 145, "XDrawLine - solid", 17);
    XDrawLine(display, xid, gc, 25, 150, 125, 250);
    XDrawLine(display, xid, gc, 25, 250, 125, 150);

    XSetForeground(display, gc, colors[AQUA]);
    XDrawString(display, xid, gc, 155, 145, "XDrawLine - dashed", 18);
    XSetLineAttributes(display, gc, 5,
        LineDoubleDash, CapButt, JoinMiter);
    XDrawLine(display, xid, gc, 150, 150, 250, 250);
    XDrawLine(display, xid, gc, 150, 250, 250, 150);
    XSetLineAttributes(display, gc, 0, LineSolid, CapButt, JoinMiter);

    XSetForeground(display, gc, colors[PINK]);
    XDrawString(display, xid, gc, 280, 145, "XDrawLines", 10);
    box[0].x = 275; box[0].y = 150;
    XDrawLines(display, xid, gc, box, 5, CoordModePrevious);

    XSetForeground(display, gc, colors[GREEN]);
    XDrawRectangle(display, xid, gc,
        5, 5, xv_get(pw, XV_WIDTH)-10, xv_get(pw, XV_HEIGHT)-10);
    XDrawRectangle(display, xid, gc,
        7, 7, xv_get(pw, XV_WIDTH)-14, xv_get(pw, XV_HEIGHT)-14);
}
```

F.7 The Logo.c Module

In Chapter 25, *XView Internals*, the methods for writing XView extensions is discussed. Example F-7 contains all the functions outlined in the chapter. The chapter also contains listings of the header files required by this module.

Example F-7. The Logo.c module

```
/*
 * Logo.c -- a XView object class that paints an X logo in a window.
 * This object is subclassed from the window object to take advantage
 * of the window it creates.  This object has no attributes, so the
 * set and get functions are virtually empty.  The only internal
 * fields used by this object are a GC and a Pixmap.  The GC is used
 * to paint the Pixmap into the window.  The window object has no GC
 * associated with it or we would have inherited it.  This will
 * probably go away in the next version of XView.
 */
#include "logo_impl.h"
#include <xview/notify.h>
#include <xview/cms.h>
#include <X11/bitmaps/xlogo32>

/* declare the "methods" used by the logo class. */
static int logo_init(), logo_destroy();
static Xv_opaque logo_set(), logo_get();
static void logo_repaint();

Xv_pkg logo_pkg = {
    "Logo",                     /* package name */
    ATTR_PKG_UNUSED_FIRST,      /* package ID */
    sizeof(Logo_public),        /* size of the public struct */
    WINDOW,                     /* subclassed from the window package */
    logo_init,
    logo_set,
    logo_get,
    logo_destroy,
    NULL                        /* disable the use of xv_find() */
};

/* the only thing this object does is paint an X into its own window.
 * This is the event handling routine that is used to check for
 * Expose or Configure event requests.  the configure event clears
 * the window and the "expose" event causes a repaint of the X image.
 * The GC has its foreground and background colros set from the
 * CMS of the window from which this logo object is subclassed.
 */
static void
logo_redraw(logo_public, event)
Logo_public     *logo_public;
Event           *event;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);
    XEvent *xevent = event_xevent(event);

    if (xevent->xany.type == Expose && xevent->xexpose.count == 0) {
```

Example F-7. The Logo.c module (continued)

```
        Display *dpy = (Display *)xv_get(logo_public, XV_DISPLAY);
        Window window = (Window)xv_get(logo_public, XV_XID);
        int width = (int)xv_get(logo_public, XV_WIDTH);
        int height = (int)xv_get(logo_public, XV_HEIGHT);
        int x = (width - xlogo32_width)/2;
        int y = (height - xlogo32_height)/2;

        XCopyPlane(dpy, logo_private->bitmap, window, logo_private->gc,
            0, 0, xlogo32_width, xlogo32_height, x, y, 1L);
    } else if (xevent->xany.type == ConfigureNotify)
        XClearArea(xv_get(logo_public, XV_DISPLAY),
            xv_get(logo_public, XV_XID), 0, 0,
            xevent->xconfigure.width, xevent->xconfigure.height, True);
}

/* initialize the logo object -- create (alloc) an instance of it.
 * There are two parts to an object class: a public part and a private
 * part.  Each contains a pointer to the other, so link the two
 * together and initialize the remaining fields of the logo data
 * structure.  This includes creating the Xlogo pixmap.  However,
 * we do no initialize the logo’s GC because it is dependent on its
 * window’s cms and that isn’t assigned to the window till the "set"
 * method.  See logo_set() below.
 */
static int
logo_init(owner, logo_public, avlist)
Xv_opaque       owner;
Logo_public     *logo_public;
Attr_avlist     avlist; /* ignored here */
{
    Logo_private *logo_private = xv_alloc(Logo_private);
    Display *dpy;
    Window win;

    if (!logo_private)
        return XV_ERROR;

    dpy = (Display *)xv_get(owner, XV_DISPLAY);
    win = (Window)xv_get(logo_public, XV_XID);

    /* link the public to the private and vice-versa */
    logo_public->private_data = (Xv_opaque)logo_private;
    logo_private->public_self = (Xv_opaque)logo_public;

    /* create the 1-bit deep pixmap of the X logo */
    if ((logo_private->bitmap = XCreatePixmapFromBitmapData(dpy, win,
        xlogo32_bits, xlogo32_width, xlogo32_height,
        1, 0, 1)) == NULL) {
        free(logo_private);
        return XV_ERROR;
    }
    /* set up event handlers to get resize and repaint events */
    xv_set(logo_public,
        WIN_NOTIFY_SAFE_EVENT_PROC,      logo_redraw,
        WIN_NOTIFY_IMMEDIATE_EVENT_PROC, logo_redraw,
        NULL);
    return XV_OK;
}

/* logo_set() -- the function called to set attributes in a logo
 * object.  This function is called when a logo is created after
 * the init routine as well as when the programmer calls xv_set.
 */
static Xv_opaque
logo_set(logo_public, avlist)
Logo_public *logo_public;
Attr_avlist avlist;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);
    Attr_attribute *attrs;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int) attrs[0]) {
            case XV_END_CREATE : {
                /* this stuff *must* be here rather than in the "init"
                 * routine because the CMS is not loaded into the
                 * window object until the "set" routines are called.
                 */
                Cms cms = xv_get(logo_public, WIN_CMS);
                XGCValues gcvalues;
                Display *dpy =
                    (Display *)xv_get(logo_public, XV_DISPLAY);
                gcvalues.foreground =
                    x(unsigned long)v_get(cms, CMS_FOREGROUND_PIXEL);
                gcvalues.background =
                    (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
                gcvalues.graphics_exposures = False;
                logo_private->gc = XCreateGC(dpy,
                    xv_get(logo_public, XV_XID),
                    GCForeground|GCBackground|GCGraphicsExposures,
                    &gcvalues);
            }
            default :
                xv_check_bad_attr(LOGO, attrs[0]);
                break;
        }

    return XV_OK;
}

/* logo_get() -- There are no logo attributes to get, so just return */
static Xv_opaque
logo_get(logo_public, status, attr, args)
Logo_public     *logo_public;
int             *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    *status = xv_check_bad_attr(LOGO, attr);
    return (Xv_opaque)XV_OK;
}
```

Example F-7. The Logo.c module (continued)

```
/* destroy method: free the pixmap and the GC before freeing the object */
static int
logo_destroy(logo_public, status)
Logo_public     *logo_public;
Destroy_status   status;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);

    if (status == DESTROY_CLEANUP) {
        XFreePixmap(xv_get(logo_public, XV_DISPLAY),
            logo_private->bitmap);
        XFreeGC(xv_get(logo_public, XV_DISPLAY), logo_private->gc);
        free(logo_private);
    }

    return XV_OK;
}
```

F.8 The Bitmap.c Module

In Chapter 25, *XView Internals*, the Bitmap package was introduced, but not fully listed. The following listing contains the entire Bitmap package implementation except for its header files (which are listed in the chapter).

Example F-8. The Bitmap.c module

```
/*
 * Bitmap.c -- an XView object class that displays an arbitrary
 * pixmap.  This is similar to the Logo object, but the programmer
 * may specify the bitmap to use via the BITMAP_FILE attribute.
 */
#include "bitmap_impl.h"
#include <xview/notify.h>
#include <xview/cms.h>
#include <X11/Xutil.h>

/* declare the "methods" used by the bitmap class. */
static int bitmap_init(), bitmap_destroy();
static Xv_opaque bitmap_set(), bitmap_get();
static void bitmap_repaint();

Xv_pkg bitmap_pkg = {
    "Bitmap2",              /* package name */
    ATTR_PKG_BITMAP,        /* package ID */
    sizeof(Bitmap_public),  /* size of the public struct */
    WINDOW,                 /* subclassed from the window package */
    bitmap_init,
    bitmap_set,
    bitmap_get,
    bitmap_destroy,
    NULL                    /* disable the use of xv_find() */
```

```
};

static void
bitmap_redraw(bitmap_public, event)
Bitmap_public     *bitmap_public;
Event           *event;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);
    XEvent *xevent = event_xevent(event);

    if (bitmap_private->bitmap &&
        xevent->xany.type == Expose && xevent->xexpose.count == 0) {
        Display *dpy = (Display *)xv_get(bitmap_public, XV_DISPLAY);
        Window window = (Window)xv_get(bitmap_public, XV_XID);
        int width = (int)xv_get(bitmap_public, XV_WIDTH);
        int height = (int)xv_get(bitmap_public, XV_HEIGHT);
        int x = (width - bitmap_private->width)/2;
        int y = (height - bitmap_private->height)/2;

        XCopyPlane(dpy, bitmap_private->bitmap, window,
            bitmap_private->gc, 0, 0, bitmap_private->width,
            bitmap_private->height, x, y, 1L);
    } else if (xevent->xany.type == ConfigureNotify)
        XClearArea(xv_get(bitmap_public, XV_DISPLAY),
            xv_get(bitmap_public, XV_XID), 0, 0,
            xevent->xconfigure.width, xevent->xconfigure.height,
            True);
}

/* initialize the bitmap object by creating (alloc) an instance
 * of it.  There are two parts to an object class: a public part
 * and a private part.  Each contains a pointer to the other, so
 * link the two together and initialize the remaining fields of
 * the bitmap data structure.  Do no initialize the bitmap's GC
 * because it is dependent on its window's cms and that isn't
 * assigned to the window till the "set" method.  Also, wait till
 * till the "set" method to initialize the bitmap file specified.
 */
static int
bitmap_init(owner, bitmap_public, avlist)
Xv_opaque       owner;
Bitmap_public     *bitmap_public;
Attr_avlist     avlist; /* ignored here */
{
    Bitmap_private *bitmap_private = xv_alloc(Bitmap_private);

    if (!bitmap_private)
        return XV_ERROR;

    /* link the public to the private and vice-versa */
    bitmap_public->private_data = (Xv_opaque)bitmap_private;
    bitmap_private->public_self = (Xv_opaque)bitmap_public;

    /* set up event handlers to get resize and repaint events */
    xv_set(bitmap_public,
        WIN_NOTIFY_SAFE_EVENT_PROC,      bitmap_redraw,
        WIN_NOTIFY_IMMEDIATE_EVENT_PROC, bitmap_redraw,
        NULL);

    return XV_OK;
}

/* bitmap_set() -- the function called to set attributes in a bitmap
 * object.  This function is called when a bitmap is created after
 * the init routine as well as when the programmer calls xv_set.
 */
static Xv_opaque
bitmap_set(bitmap_public, avlist)
Bitmap_public *bitmap_public;
Attr_avlist avlist;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);
    Attr_attribute *attrs;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int) attrs[0]) {
            case BITMAP_FILE : {
                int val, x, y;
                Display *dpy =
                    (Display *)xv_get(bitmap_public, XV_DISPLAY);
                Window window =
                    (Window)xv_get(bitmap_public, XV_XID);
                Pixmap old = bitmap_private->bitmap;
                if (XReadBitmapFile(dpy, window, attrs[1],
                    &bitmap_private->width, &bitmap_private->height,
                    &bitmap_private->bitmap, &x, &y) != BitmapSuccess)
                {
                    xv_error(bitmap_public,
                        ERROR_STRING, "Unable to load bitmap file",
                        ERROR_PKG,    BITMAP,
                        NULL);
                    bitmap_private->bitmap = old;
                }
                break;
            }
            case BITMAP_PIXMAP :
                xv_error(bitmap_public,
                    ERROR_CANNOT_SET, attrs[0],
                    ERROR_PKG,        BITMAP,
                    NULL);
                break;
            case XV_END_CREATE : {
                /* this stuff *must* be here rather than in the "init"
                 * routine because the CMS is not loaded into the
                 * window object until the "set" routines are called.
                 */
                Cms cms = xv_get(bitmap_public, WIN_CMS);
                XGCValues gcvalues;
                Display *dpy =
                    (Display *)xv_get(bitmap_public, XV_DISPLAY);
                gcvalues.foreground =
                    (unsigned long)xv_get(cms, CMS_FOREGROUND_PIXEL);
```

Example F-8. The Bitmap.c module (continued)

```
                gcvalues.background =
                    (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
                gcvalues.graphics_exposures = False;
                bitmap_private->gc =
                    XCreateGC(dpy, xv_get(bitmap_public, XV_XID),
                        GCForeground|GCBackground|GCGraphicsExposures,
                        &gcvalues);
            }
            default :
                xv_check_bad_attr(BITMAP, attrs[0]);
                break;
        }
    return XV_OK;
}

static Xv_opaque
bitmap_get(bitmap_public, status, attr, args)
Bitmap_public     *bitmap_public;
int               *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);

    switch ((int) attr) {
        case BITMAP_PIXMAP :
            return (Xv_opaque)bitmap_private->bitmap;
        case BITMAP_FILE : /* can't get this attribute */
        default :
            *status = xv_check_bad_attr(BITMAP, attr);
            return (Xv_opaque)XV_OK;
    }
}

/* destroy method: free the pixmap and the GC before freeing object */
static int
bitmap_destroy(bitmap_public, status)
Bitmap_public     *bitmap_public;
Destroy_status   status;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);

    if (status == DESTROY_CLEANUP) {
        if (bitmap_private->bitmap)
            XFreePixmap(xv_get(bitmap_public, XV_DISPLAY),
                bitmap_private->bitmap);
        XFreeGC(xv_get(bitmap_public, XV_DISPLAY),
            bitmap_private->gc);
        free(bitmap_private);
    }
    return XV_OK;
}
```

F.9 The panel_dnd.c Program

In Chapter 7, *Panels*, a Drop Target Items is described. This program creates and uses a Panel Drop Target Item.

Example F-9. The panel_dnd.c program

```
/*
 * panel_dnd.c - provides text fields, a textsw, and
 * several drop targets to demonstate ways to receive
 * and illustrate drag and drop operations.
 */

#include <malloc.h>
#include <xview/xview.h>
#include <xview/dragdrop.h>
#include <xview/panel.h>
#include <xview/svrimage.h>
#include <xview/textsw.h>

static unsigned short normal_bitmap[ ] = {
#include "normal.icon"
};

static unsigned short busy_bitmap[ ] = {
#include "busy.icon"
};

static unsigned short normal2_bitmap[ ] = {
#include "normal2.icon"
};

static unsigned short busy2_bitmap[ ] = {
#include "busy2.icon"
};

static char * dnd_codes[7] = {
    "OK",
    "Error",
    "Illegal Target",
    "Timeout",
    "Unable to obtain selection",
    "Dropped on root window",
    "*** Unknown return code"
};

Frame frame;
Panel panel;
Panel_item drop_target[3];
Drag_drop dnd;


/*ARGSUSED*/
static void
hide_drop_targets(item, event)
    Panel_item item;
    Event *event;
{
    int         i;
    int         show;

    show = !xv_get(drop_target[0], XV_SHOW);
    for (i=0; i<=2; i++)
      xv_set(drop_target[i],
             XV_SHOW, show,
             0);
    xv_set(item,
      PANEL_LABEL_STRING, show ? "Hide drop targets" : "Show drop targets",
      0);
}


/*ARGSUSED*/
static void
inactivate_drop_targets(item, event)
    Panel_item item;
    Event *event;
{
    int         i;
    int         inactive;

    inactive = !xv_get(drop_target[0], PANEL_INACTIVE);
    for (i=0; i<=2; i++)
      xv_set(drop_target[i],
             PANEL_INACTIVE, inactive,
             0);
    xv_set(item,
      PANEL_LABEL_STRING, inactive ? "Activate drop targets" :
          "Inactivate drop targets",
      0);
}


static void
get_primary_selection(sel_req)
    Selection_requestor sel_req;
{
    long        length;
    int               format;
    char       *sel_string;
    char       *string;

    xv_set(sel_req, SEL_TYPE, XA_STRING, 0);
    string = (char *) xv_get(sel_req, SEL_DATA, &length, &format);
    if (length != SEL_ERROR) {
      /* Create a NULL-terminated version of 'string' */
      sel_string = (char *) xv_calloc(1, length+1);
      strncpy(sel_string, string, length);
      /* Print out primary selection string */
      printf("Primary selection=
    } else
      printf("*** Unable to get primary selection.0);
```

```
}


static int
drop_target_notify_proc(item, value, event)
    Panel_item       item;
    unsigned int    value;
    Event      *event;
{
    Selection_requestor sel_req;

    printf("(drop_target_notify_proc) %s action= ",
      xv_get(item, PANEL_LABEL_STRING));
    sel_req = xv_get(item, PANEL_DROP_SEL_REQ);
    switch (event_action(event)) {
      case ACTION_DRAG_COPY:
      printf("ACTION_DRAG_COPY,0);
      get_primary_selection(sel_req);
      break;
      case ACTION_DRAG_MOVE:
      printf("ACTION_DRAG_MOVE0);
      get_primary_selection(sel_req);
      break;
      case LOC_DRAG:
      printf("LOC_DRAG, result= %s0, dnd_codes[MIN(value, 6)]);
      break;
      default:
      printf("%d0, event_action(event));
      break;
    }
    return XV_OK;
}


main(argc, argv)
    int argc;
    char **argv;
{
    Server_image    busy_glyph[2];
    Server_image    normal_glyph[2];
    Panel      panel2;

    xv_init(XV_INIT_ARGS, argc, argv, 0);

    frame = xv_create(NULL, FRAME,
      FRAME_LABEL, "Drag and Drop Test",
      0);
    panel = xv_create(frame, PANEL,
      PANEL_LAYOUT, PANEL_VERTICAL,
      0);
    xv_create(panel, PANEL_BUTTON,
      PANEL_LABEL_STRING, "Inactivate Drop Targets",
      PANEL_NOTIFY_PROC, inactivate_drop_targets,
      0);
    xv_create(panel, PANEL_BUTTON,
      PANEL_LABEL_STRING, "Hide Drop Targets",
      PANEL_NOTIFY_PROC, hide_drop_targets,
      0);
    xv_create(panel, PANEL_TEXT,
      PANEL_LABEL_STRING, "Text field #1:",
      PANEL_VALUE, "Hello world!",
      PANEL_VALUE_DISPLAY_LENGTH, 20,
      0);
    xv_create(panel, PANEL_TEXT,
      PANEL_LABEL_STRING, "Text field #2:",
      PANEL_VALUE_DISPLAY_LENGTH, 20,
      0);
    dnd = xv_create(panel, DRAGDROP, 0);
    xv_create(dnd, SELECTION_ITEM,
      SEL_DATA, "dnd selection data",
      0);
    normal_glyph[0] = xv_create(NULL, SERVER_IMAGE,
      XV_HEIGHT, 64,
      XV_WIDTH, 64,
      SERVER_IMAGE_DEPTH, 1,
      SERVER_IMAGE_BITS, normal_bitmap,
      0);
    busy_glyph[0] = xv_create(NULL, SERVER_IMAGE,
      XV_HEIGHT, 64,
      XV_WIDTH, 64,
      SERVER_IMAGE_DEPTH, 1,
      SERVER_IMAGE_BITS, busy_bitmap,
      0),
    normal_glyph[1] = xv_create(NULL, SERVER_IMAGE,
      XV_HEIGHT, 64,
      XV_WIDTH, 64,
      SERVER_IMAGE_DEPTH, 1,
      SERVER_IMAGE_BITS, normal2_bitmap,
      0);
    busy_glyph[1] = xv_create(NULL, SERVER_IMAGE,
      XV_HEIGHT, 64,
      XV_WIDTH, 64,
      SERVER_IMAGE_DEPTH, 1,
      SERVER_IMAGE_BITS, busy2_bitmap,
      0),
    drop_target[0] = xv_create(panel, PANEL_DROP_TARGET,
      PANEL_LABEL_STRING, "Full Drop Target:",
      PANEL_NOTIFY_PROC, drop_target_notify_proc,
      PANEL_DROP_DND, dnd,
      PANEL_DROP_FULL, TRUE,
      PANEL_DROP_GLYPH, normal_glyph[0],
      PANEL_DROP_BUSY_GLYPH, busy_glyph[0],
      0);
    xv_create(panel, PANEL_DROP_TARGET,
      PANEL_LABEL_STRING, "Full Drop Target #2:",
      PANEL_NOTIFY_PROC, drop_target_notify_proc,
      PANEL_DROP_DND, dnd,
      PANEL_DROP_FULL, TRUE,
      PANEL_DROP_GLYPH, normal_glyph[1],
      PANEL_DROP_BUSY_GLYPH, busy_glyph[1],
      0);
    drop_target[1] = xv_create(panel, PANEL_DROP_TARGET,
```

Example F-9. The panel_dnd.c program (continued)

```
      PANEL_LABEL_STRING, "Default Empty Drop Target:",
      PANEL_NOTIFY_PROC, drop_target_notify_proc,
      PANEL_DROP_SITE_DEFAULT, TRUE,
      0);
    if (xv_get(drop_target[1], PANEL_DROP_SITE_DEFAULT) != TRUE) {
      printf("PANEL_DROP_SITE_DEFAULT failed0);
      exit(1);
    }
    drop_target[2] = xv_create(panel, PANEL_DROP_TARGET,
      PANEL_LABEL_STRING, "Default Full Drop Target:",
      PANEL_NOTIFY_PROC, drop_target_notify_proc,
      PANEL_DROP_DND, dnd,
      PANEL_DROP_FULL, TRUE,
      0);
    window_fit(panel);

    xv_create(frame, TEXTSW,
      WIN_ROWS, 4,
      WIN_COLUMNS, 10,
      0);
    panel2 = xv_create(frame, PANEL,
      PANEL_LAYOUT, PANEL_VERTICAL,
      0);
    xv_create(panel2, PANEL_MESSAGE,
      PANEL_LABEL_STRING, "New panel",
      0);
    xv_create(panel2, PANEL_TEXT,
      PANEL_LABEL_STRING, "Text field:",
      PANEL_VALUE_DISPLAY_LENGTH, 20,
      0);
    window_fit(panel2);

    window_fit(frame);

    xv_main_loop(frame);
    exit(0);
}
```

Index