1
XView Package Summary

This section provides a brief introduction to XView and then goes on to introduce each of the XView packages. The packages are described in alphabetical order; their order in this section does not correspond to how they are used in any particular application. Refer to the *XView Programming Manual* for a general introduction to XView and for details on creating and working with XView objects.

The last part of this section provides a description of the Notifier. The Notifier maintains the flow of control in an application. The Notifier controls XView's *notification-based* event handling using *callback procedures* supplied by application programmers. Also refer to the *XView Programming Manual* for more information on the Notifier.

1.1 XView and OPEN LOOK

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.

An important feature of the XView Toolkit is that it implements the OPEN LOOK Graphical User Interface (GUI). The OPEN LOOK GUI provides users with a simple, consistent, and efficient interface.

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 for each application. XView was built based entirely on OPEN LOOK specifications that could be mapped easily into the X Window System.

The visual design of OPEN LOOK is restrained. The design of each component in the user interface is simple and uncomplicated. The interface is based on a few simple concepts that result in a system that is easy to learn initially. And an XView application is relatively simple, and is easy to implement because many of the default values of interface components work well for most applications.

The definitive document on OPEN LOOK for application programmers is the *OPEN LOOK Graphical User Interface Style Guide*.

1.2 XView Packages

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 1-1 shows the XView class hierarchy. 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.

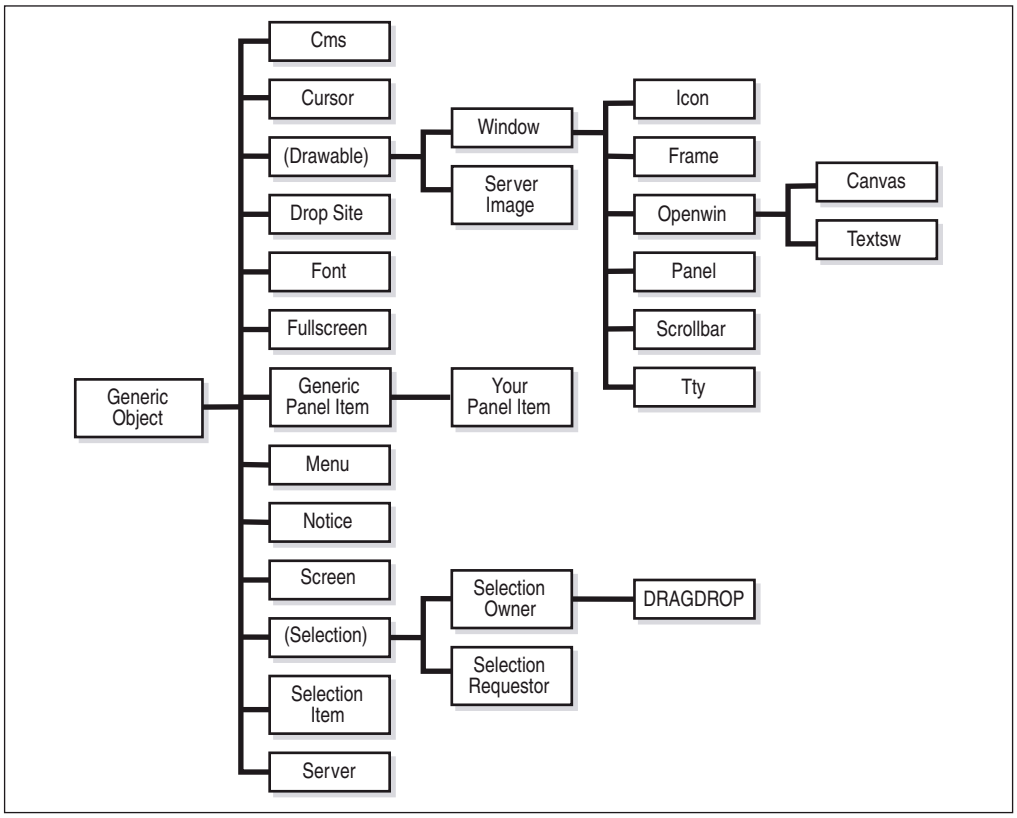

Figure 1-1. XView class hierarchy

Some objects are visual and others are not. Visual objects include windows, scrollbars, frames, panels, and panel items, among others. 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.

1.2.1 Object Handles

When you create an object, an 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.

1.2.2 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 the set of all applicable attributes will be of interest.

1.2.3 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 1-1.

Table 1-1. 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. Table 1-2 lists the XView packages. Each of these packages is introduced in this section.

Table 1-2. XView Packages

CANVAS	NOTICE	PANEL_NUMERIC_TEXT
CMS	PANEL	PANEL_SLIDER
CURSOR	PANEL_ITEM	PANEL_TEXT
DRAGDROP	PANEL_BUTTON	SCREEN
DROP_SITE_ITEM	PANEL_CHOICE	SCROLLBAR
FONT	PANEL_CHECK_BOX	SELECTION
FRAME	PANEL_DROP_TARGET_ITEM	SERVER
FULLSCREEN	PANEL_GAUGE	TEXTSW
ICON	PANEL_LIST	TTYSW
MENU	PANEL_MESSAGE	WINDOW
MENU_ITEM	PANEL_MULTILINE_TEXT	

1.3 The CANVAS Package

A canvas is the area in which an application displays graphics and handles its input. An XView canvas object allows the user to view a graphic image that is similar to a painter's canvas. This image may be too large for the window or even too large for the display screen. The viewable portion of the graphic image is part of image's *viewport* or *view window*. Many different views of an 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.

There are three types of windows 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 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.

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. The `owner` of a canvas must be a `FRAME` object.

1.4 The CMS Package

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.

XView applications deal with color by using *colormap segments*. 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. Colormap segments must be applied to windows to assure that the window can access the color you are attempting to draw into.

A colormap segment from the `CMS` package 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.

Applications that use color must include the file <*xview/cms.h*>. The *owner* of a colormap segment is the XView screen object with which the colormap is associated. If an owner is not specified (`NULL` owner), the default screen of the default server is used as the owner.

1.5 The CURSOR Package

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.

To use the `CURSOR` package, include the header file <*xview/cursor.h*>. 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 the owner is `NULL`, then the root window of the default screen is used.

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. There are some OPEN LOOK cursor `define`s prefixed by `OLC_` in <*xview/cursor.h*>.

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.

1.6 The DRAGDROP and DROP_SITE_ITEM Packages

Drag and drop operations are facilitated using two packages: the `DRAGDROP` package and the `DROP_SITE_ITEM` package. `DRAGDROP` is subclassed from the `SELECTION_OWNER` package and represents the source of the drag. `DROP_SITE_ITEM` is subclassed from the Generic package. A drop-site item indicates a destination that is a valid drop-site. A valid drop-site is a region that may either preview a drop or receive a drop. A drag and drop operation, such as the familiar procedure of dragging a file from a folder and dropping it into another folder, allows you to easily transfer data.

Applications need to include the file *<xview/dragdrop.h>* to use these packages. Drag and drop supports *drop previewing* where the drop-site image changes to show that it is a valid drop-site and *drag feedback* where the pointer image (cursor) changes to indicate that an item is being dragged.

Attributes for `DRAGDROP` use a `DND_` prefix. Attributes for `DROP_SITE_ITEM` use a `DROP_SITE` prefix. The owner of either a `DRAGDROP` object, or a `DROP_SITE_ITEM` object is a window.

1.7 The FONT Package

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. So the images, or *glyphs*, represent a character set-defined language used. However, a font may be built to support glyphs for other languages or to provide a set of glyphs. 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, *Helvetica bold 14* refers to the Helvetica font family; bold is the style and 14 is the point size.

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.

*Note that point sizes on workstations are based on pixels, whereas point sizes for typesetters and printers are based on inches.

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."

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 would obtain a handle to the font, if it already exists, rather than open 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.

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, or in memory pixmaps, or a server image (a Server_image object), but these objects must have the same display (Server) 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.

1.8 The FRAME Package

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.

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 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.

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.

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 and a pushpin if run under the olwm (OPEN LOOK window manager).

A base frame's *parent* or owner, 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.

1.9 The FULLSCREEN Package

The `FULLSCREEN` package is used to grab the X server, and an instance of it is considered a nonvisual object. Nonvisual objects are not viewed on the screen, but they have a place in the XView object hierarchy. Nonvisual objects are typically used internally by XView and are seldom used directly in an application. Working with `FULLSCREEN` objects may not be essential to all programmers.

1.10 The ICON Package

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.

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.

To use the `ICON` package, include the header file <*xview/icon.h*>. 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 an example of *delayed binding*.

When destroying an icon, the server image associated with the icon is not destroyed—it is the application programmer's responsibility to free the server image and the pixmap associated with the icon if needed.

1.11 The MENU and MENUITEM Packages

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.

1.11.1 Menu Types

There are three different types of menus: *pop-up*, *pulldown*, and *pullright* menus. The general term pop-up menu may describe all three types in certain contexts since menus are *popped up*. However, pulldown and pullright menus have distinct characteristics that make them unique.

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.

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.

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.

1.11.2 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* above). Typically, the 0th item in the menu is the default, but that may be changed either by the application or by the user.

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 (. . .).

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.

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.

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.

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 for a menu's `MENU_ITEM` attribute.

1.12 The NOTICE Package

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.

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 the thread of execution for 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.

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`.

1.13 The PANEL Package

The 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*. 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.

Panels set up and manage their own 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). 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 upon the user pressing 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.

A panel's owner is a frame. All programs that use panels or panel items must include *<xview/panel.h>*.

1.13.1 Panel Items

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.

1.13.1.1 Button Items

A button item allows the user to invoke a command or bring up a menu. 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.

There are several types of panel button items:

- Panel Buttons (shown in Figure 1-2)
- Menu Buttons (shown in Figure 1-3)
- Abbreviated Menu Buttons (shown in Figure 1-4)

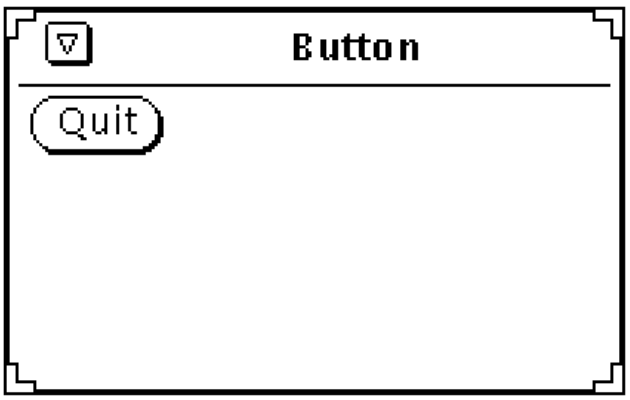

Figure 1-2. Panel item created with PANEL_BUTTON package

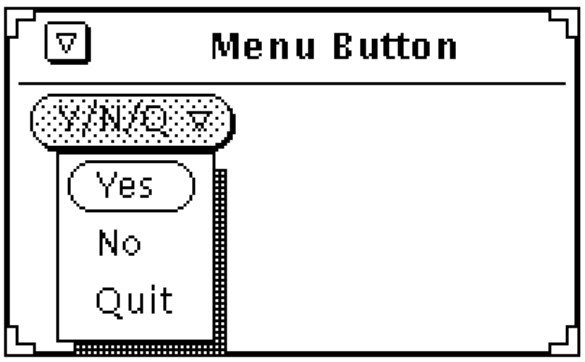

Figure 1-3. Panel button with an attached menu

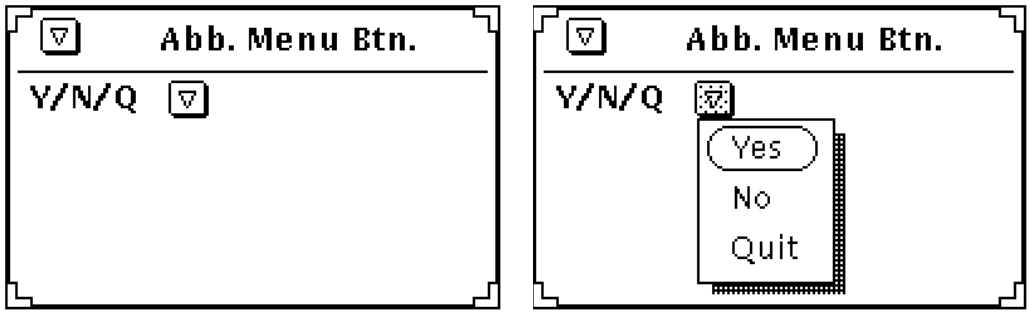

Figure 1-4. Panel item created with the PANEL_ABBREV_MENU_BUTTON package

1.13.1.2 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

The figures below show several, but not all of the different types of choice items.

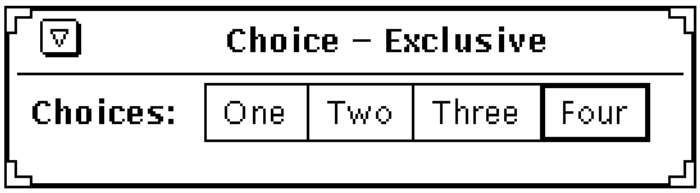

Figure 1-5. Panel item from the PANEL_CHOICE package

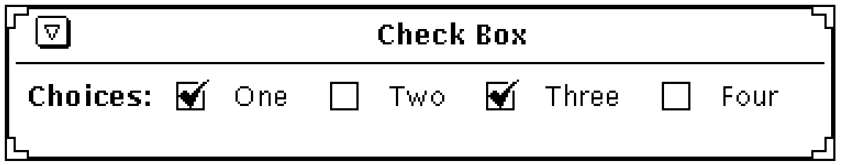

Figure 1-6. Panel item from the PANEL_CHECK_BOX package

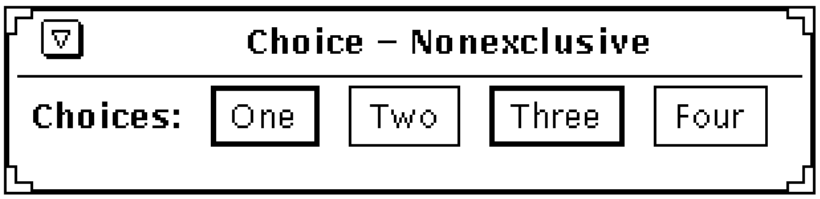

Figure 1-7. Panel item created with PANEL_TOGGLE macro

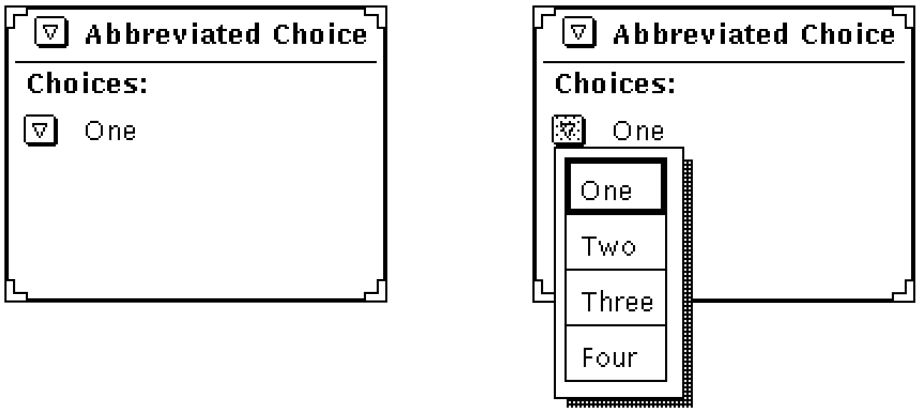

Figure 1-8. Panel item created with PANEL_CHOICE_STACK macro

1.13.1.3 Drop Target Items

A panel drop target item is a bordered image in a panel area that is used to transfer data to or from applications. Before you use a panel drop target item you need to be familiar with the `SELECTION` and `DRAGDROP` packages.

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`.

1.13.1.4 Panel Extension Items

Panel extension items support additional user defined panel items. Use of this type of panel item is an advanced topic that is covered in Chapter 25, *XView Internals*, in the *XView Programming Manual*.

1.13.1.5 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 slider bar 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.

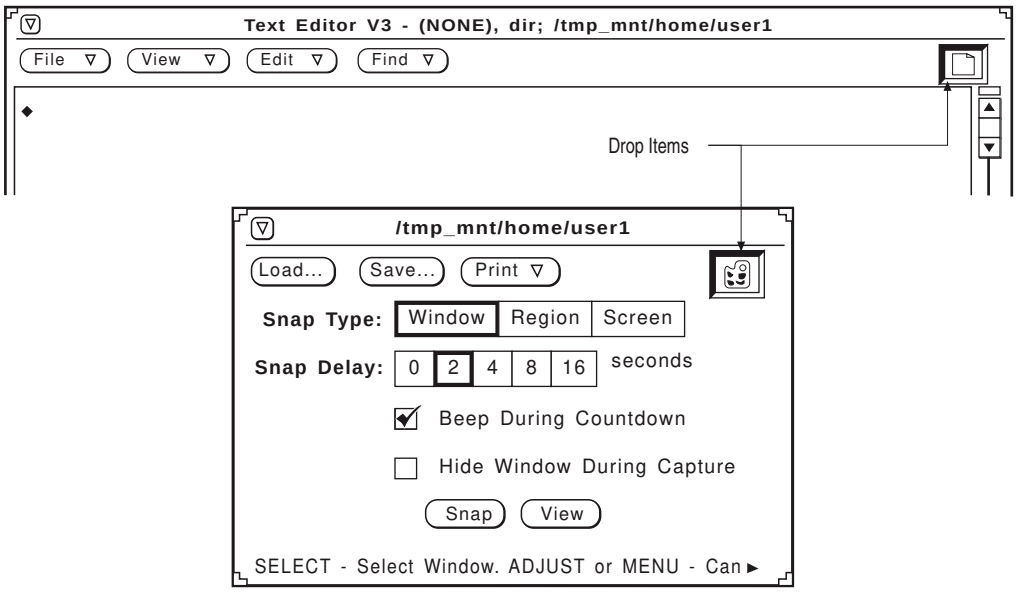

Figure 1-9. Panel drop target items

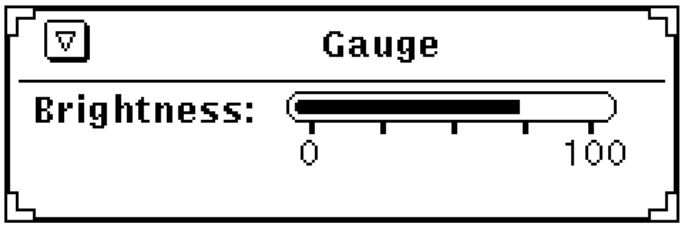

Figure 1-10. Panel item created with the PANEL_GAUGE package

1.13.1.6 List Items—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.

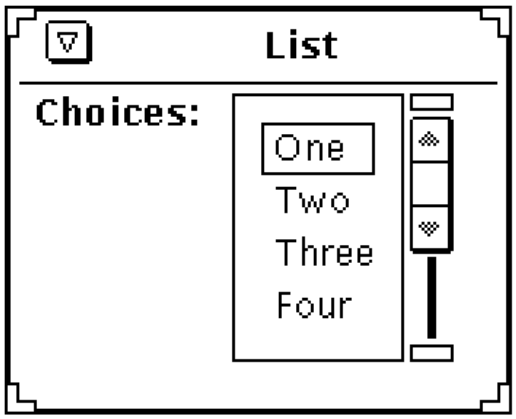

Figure 1-11. Panel list item created with the PANEL_LIST package

1.13.1.7 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.

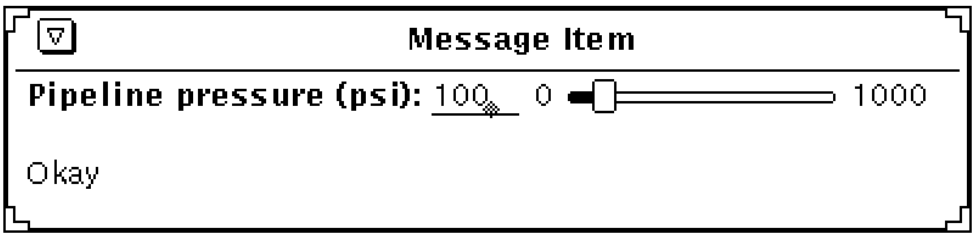

Figure 1-12. Panel item created with PANEL_MESSAGE package

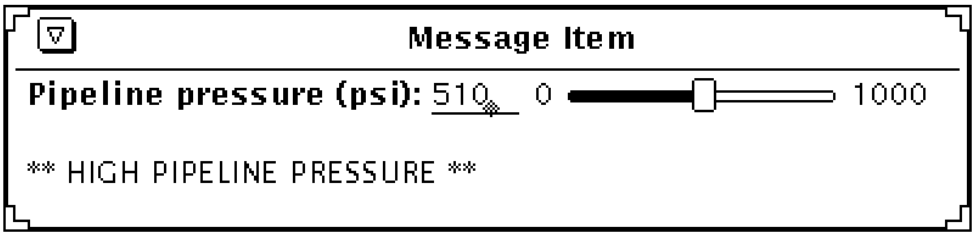

Figure 1-13. Another panel item created with PANEL_MESSAGE package

1.13.1.8 Multiline Text Items

Multiline text items are a special type of panel text item that allow a text field containing multiple lines.

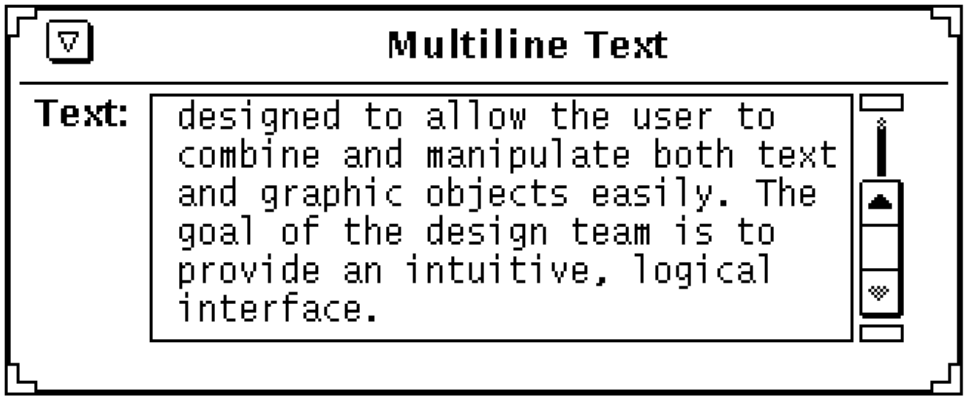

Figure 1-14. Panel item created with the PANEL_MULTILINE_TEXT package

1.13.1.9 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, but there is little programmatic difference between the text item and the numeric text item. You can create a numeric text item using the `PANEL_NUMERIC_TEXT` package.

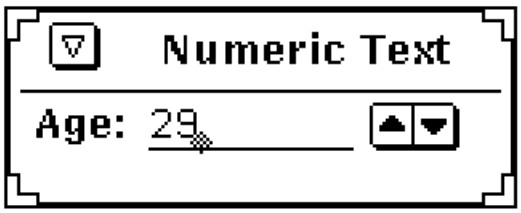

Figure 1-15. Panel item created with the PANEL_NUMERIC_TEXT package

1.13.1.10 Slider Items

Slider items allow the graphical representation and selection of a value within a range as shown in Figure 1-16. 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.

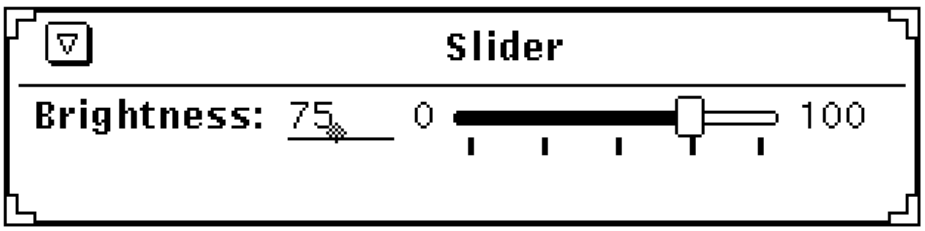

Figure 1-16. Panel item created with the PANEL_SLIDER package

1.13.1.11 Text Items

A panel text item contains as its value a `NULL`-terminated string. Typically, it contains only printable ASCII 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.

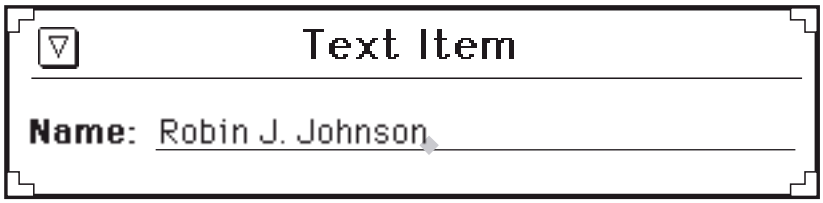

Figure 1-17. Panel item created with the PANEL_TEXT package

1.13.2 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.

Scrollable panels are created the same way panels are, but the package name to use is `SCROLLABLE_PANEL`. However, the scrollable panel package does not create the scrollbars, you must create them separately.

1.14 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>*. 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>*).

1.15 The SCROLLBAR Package

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).

The definitions necessary to use scrollbars are found in the header file *<xview/scrollbar.h>*. The `owner` must be an object subclassed from the `OPENWIN` package or the `FRAME` package. 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.

1.16 The SELECTION Package

The X Window System provides several methods for applications to exchange information with one another. One of these methods is the use of the selections. A *selection* transfers arbitrary information between two clients. XView Version 3 provides a selection mechanism that is implemented using the `SELECTION` and `SELECTION_ITEM` packages. The selection package and its sub classes, including: the `SELECTION_REQUESTOR` package and the `SELECTION_OWNER` package, allow data to be move between applications or within an application. These packages replace the selection service used in previous versions of XView, which required special functions and structures. The old selection service is still supported; it is described in Appendix A, *The Selection Service*, of the *XView Programming Manual*.

1.17 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.

When making any reference to Xv_Server objects, applications should include <*xview/server.h*>. There is no owner for a server, the owner parameter is ignored and you may pass NULL.

1.18 The TEXTSW Package

This TEXTSW package allows a user or client to display and edit a sequence of ASCII characters. A text contains a vertical scrollbar but may not contain a horizontal scrollbar. The vertical scrollbar can be used to split views into several views. 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.

Applications need to include the file <*xview/textsw.h*> to use text subwindows.

1.19 The TTYSW Package

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.

Programs using tty subwindows must include the file <*xview/tty.h*>. 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.

1.20 The Notifier

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* input and *event-driven* input, and consider how they affect where the flow of control resides within a program.

1.20.1 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.

1.20.2 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. Further, there are other sources of events such as the window system itself and 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.

1.20.3 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.

1.20.4 How the Notifier Works

When you specify *callbacks* or *notify procedures*, the XView object specified is said to be the *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.

1.20.4.1 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).