6
Handling Input

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.

Handling Input

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

Handling Input

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 */
```

Handling Input

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.

Handling Input

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.

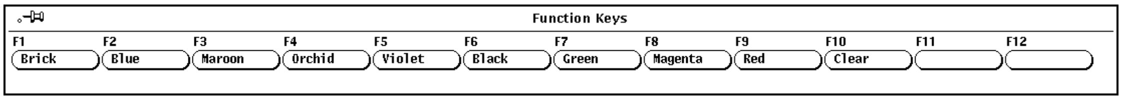

Figure 6-1. A sample function keys window

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.

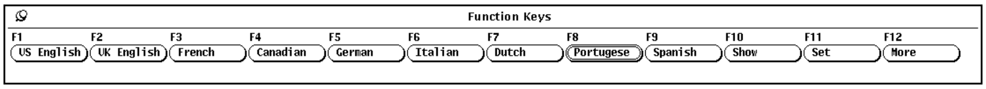

Figure 6-2. Sample function keys window with a MORE key

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 keyboards (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.

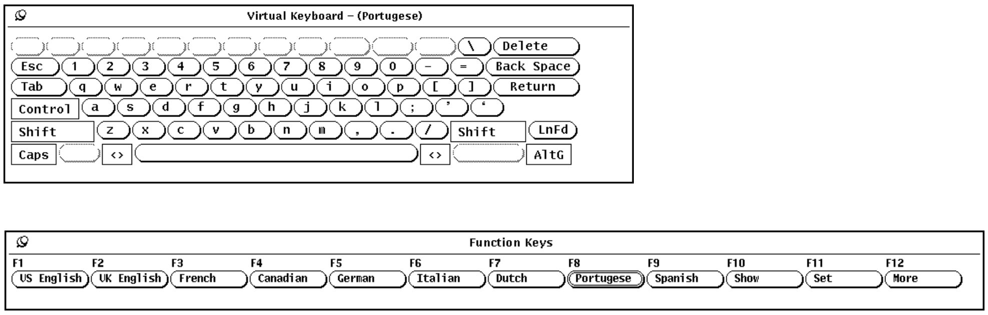

Figure 6-3. Sample virtual keyboard binding

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.

Handling Input

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*.