4
Frames

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

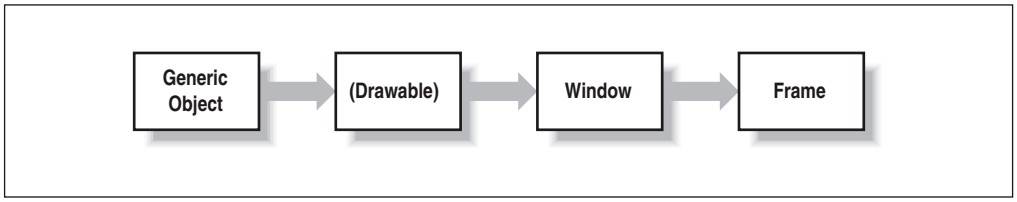

Figure 4-1. Frame package class hierarchy

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

The FRAME package provides the following capabilities:

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

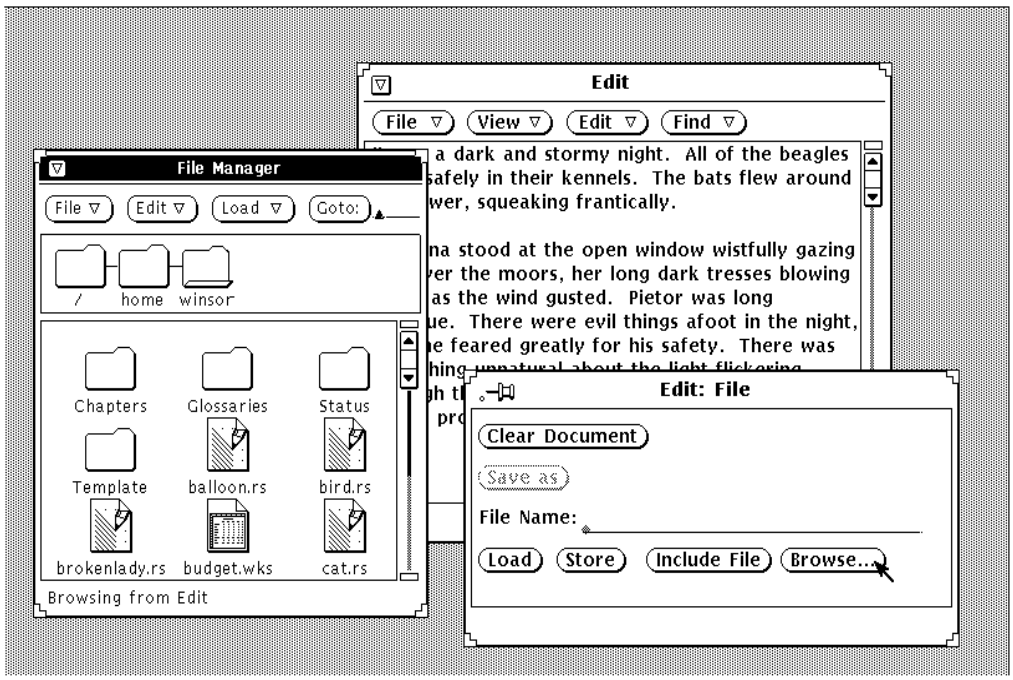

Figure 4-2. Three base frames

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

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

4.1 Types of Frames

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

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

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

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

4.1.1 The Role of the Window Manager

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

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

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

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

4.2 Base Frames

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

Example 4-1. The simple_frame.c program

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

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

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

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

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

4.2.1 XView Initialization and Base Frames

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

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

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

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

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

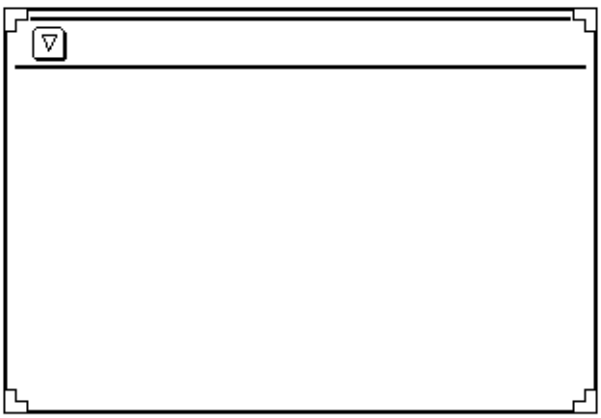

Figure 4-3. Simple base frame created without any FRAME attributes specified

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

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

4.2.2 Headers and Footers

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

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

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

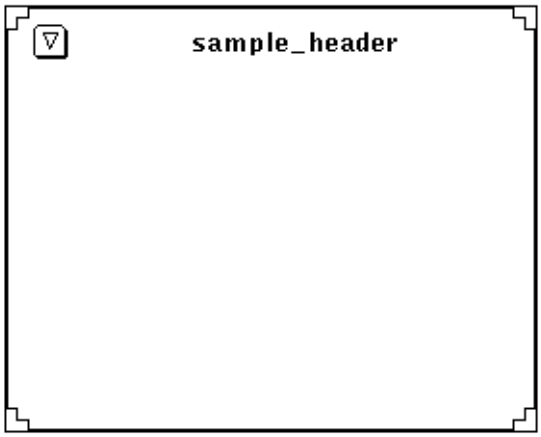

Figure 4-4. A sample header label display in a frame

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

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

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

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

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

Example 4-4. Creating a footer

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

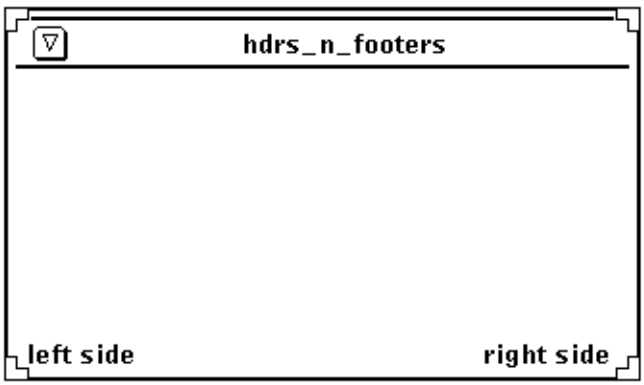

Figure 4-5. Headers and footers on base frame

4.2.3 Closed Base Frames

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

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

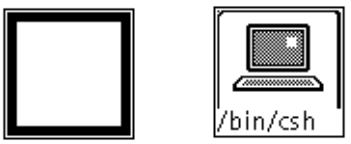

Figure 4-6. Default icon and application icon

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

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

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

*Only base frames have icons associated with them.

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

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

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

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

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

```
extern Icon icon;
Frame       frame;

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

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

4.2.4 Quit Confirmation

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

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

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

4.3 Command Frames

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

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

Example 4-5. Creating a subframe

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

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

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

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

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

    xv_set(subframe, XV_SHOW, TRUE, NULL);

    xv_main_loop(frame);
}
```

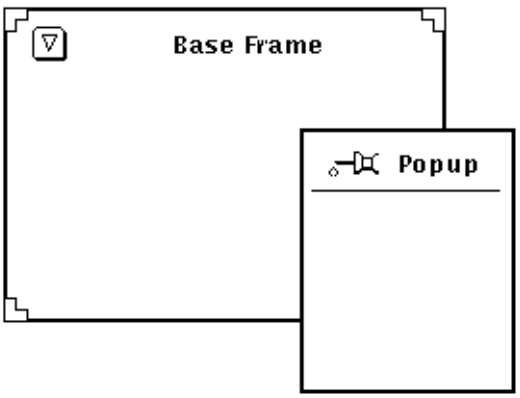

Figure 4-7. Base frame with a command frame

4.3.1 Manually Displaying Frames

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

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

4.3.2 The Pushpin

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

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

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

Example 4-6. Using several frame attributes

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

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

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

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

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

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

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

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

    xv_main_loop(frame);
}
```

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

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

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

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

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

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

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

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

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

4.3.3 The FRAME_DONE_PROC Procedure

If the pushpin is pushed in by the user, the application has no knowledge of this action. However, if the user pulls the pin out, then the toolkit calls the command frame's `FRAME_DONE_PROC` routine. By default, if the parent of command frame is `NULL`, then the frame is unmapped. The programmer can override this behavior by installing another `FRAME_DONE_PROC` routine. This is only needed if you want to check that the frame can be

dismissed. Suppose that the purpose of the subframe is to query for a filename. If the filename was not given, you might want to display a notice indicating that and allow the user to type in a filename. In this case, the FRAME_DONE_PROC is responsible for doing the necessary checking and deciding whether to display a notice or to take the frame down.

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

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

Example 4-7. The subframe.c program

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

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

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

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

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

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

The call to `xv_set(XV_SHOW, FALSE)` is safe because no action is taken at all if this was not the result of the pin being pulled out. The frame will not go down (i.e., the `xv_set()` is a no-op) if the pin is in or is already out. This gives the programmer the flexibility of using

the same function (something like done_proc()) in other places throughout the application. For example, it could be used in a callback routine for a panel button.

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

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

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

4.3.4 Showing Resize Corners

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

4.3.5 Minimum and Maximum Frame Sizes

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

4.4 Miscellaneous Attributes

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

4.5 Busy Frames

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

```
show_cmd_frame(subframe,event)

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

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

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

4.6 Frame Sizes

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

```
Frame frame;
Rect rect;

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

frame_set_rect(frame, &rect);
```

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

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

If the position of a subframe is queried with:

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

or

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

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

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

4.7 Frame Colors

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

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

This is shown in Example 4-8.

Example 4-8. Changing a frame's color

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

#define RED 0
#define BLUE 1
```

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

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

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

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

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

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

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

    xv_main_loop(frame);
}
```

4.8 Child Windows

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

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

```
Frame subframe;

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

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

```
Canvas canvas;

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

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

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

4.9 Window Loop

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

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

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

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

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

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

4.10 Removing Decorations

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

4.11 Setting Properties and Saving Command-line Options

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

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

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

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

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

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

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

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

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

```
Frame base_frame, second_frame;

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

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

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

4.12 Destroying Frames

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

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

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

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

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

```
Frame   frame;
int    i;

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

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

4.13 Frame Resize and Repaint Events

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

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

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

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

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

4.14 Frame Package Summary

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

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

Table 4-1. Frame Attributes

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

Table 4-1. Frame Attributes (continued)

FRAME_LABEL	FRAME_SHOW_HEADER
FRAME_LEFT_FOOTER	FRAME_SHOW_LABEL
FRAME_MAX_SIZE	FRAME_SHOW_RESIZE_CORNER
FRAME_MIN_SIZE	