sun® microsystems

SunView™ 1 System Programmer's Guide

Part Number: 800-1784-10
Revision A, of 9 May 1988

Contents

Tables

Figures

1

Introduction

1

Introduction

What is SunView?

SunView is a system to support interactive, graphics-based applications running within windows. It consists of two major levels of functionality: the application level and the system level. The system level is described in this document and covers two major areas: the building blocks on which the application level is built and advanced application-related features.

Changes From Release 2.0

SunView is an extension and refinement of SunWindows 2.0, containing many enhancements, bug fixes and new facilities not present in SunWindows. However, the changes preserve source level compatibility between SunWindows 2.0 and SunView.

Organization of Documentation

The 2.0 *SunWindows Reference Manual* has not been reprinted for SunView.

These changes are reflected in the current organization of the SunView documentation. The material on Pixrects from the old *SunWindows Reference Manual* is in a new document titled *Pixrect Reference Manual*. uch of the functionality of the SunWindows window and tool layers has been incorporated into the new SunView interface. The basic SunView interface, intended to meet the needs of simple and moderately complex applications, is documented in the application-level manual, the *SunView 1 Programmer's Guide*.

This document is the *SunView 1 System Programmer's Guide*. It contains a combination of new and old material. Several of its chapters document new facilities such as the Notifier, the Agent, the Selection Service and the defaults package. Also included is low-level material from the old *SunWindows Reference Manual* — e.g. the window manager routines — of interest to implementors of window managers and other advanced applications.

This document is an extension of the application-level manual. You should only delve into this manual if the information in the *SunView 1 Programmer's Guide* manual doesn't answer your needs. Thus, you should read the application-level manual first.

Compatibility

Another consideration is compatibility with future releases. Most of the objects in the *SunView 1 Programmer's Guide* are manipulated through an opaque attribute value interface. Code that uses them will be more portable to future versions of SunView than if it uses the routines documented in this manual which assume particular data structures and explicit parameters. If you do use these routines then the code should be encapsulated so that low-level details are isolated from the rest of your application.

Keep your old documentation

On the way to SunView, we have discarded documentation about the internals of some data structures that were discussed in SunWindows 2.0. In addition, we have discarded documentation about routines whose functionality is now provided by the interface discussed in the *SunView 1 Programmer's Guide*. Thus, if your application is based on the SunWindows programming interface, you should keep your 2.0 documentation. In particular, the following structures are no longer documented (there may by others): `tool`, `pixwin`, `toolsw`, and `toolio`.

Changes in Release 4.0

SunOS Release 4.0 includes further enhancements to the higher-level SunView programmatic interface documented in the *SunView 1 Programmer's Guide*, such as alerts, shadowed subframes, 'Props' actions, and so on . Sun encourages programmers to use these higher-level interfaces in preference to low-level routines documented in this manual whenever possible; this will help to insulate your applications from changes in the low-level window-system interfaces underneath SunView.

2

Overview

2

Overview

2.1. SunView Architecture

From a system point of view, SunView is a two-tiered system, consisting of the *application* and *system* layers:

- The application layer provides a set of high-level objects, including windows of different types, menus, scrollbars, buttons, sliders, etc., which the client can assemble into an application, or *tool.* This layer is sometimes referred to as the *tool layer*. The functionality provided at this level should suffice for most applications. This layer is discussed in the *SunView 1 Programmer's Guide.*
- At the *system* layer a window is presented not as an opaque object but in terms which are familiar to UNIX programmers — as a *device* which the client manipulates through a *file descriptor* returned by an *open* (2) call. This layer is sometimes referred to as the *window device layer*. The manipulation and multiplexing of multiple window devices is the subject of much of this document. The term ''window device'' is often shortened to just *window* in this document.

2.2. Document Outline

This document covers the follow system level topics:

- A system model which presents the levels, components and inter-relationships of the window system.
- A SunView mechanism, called the *Agent*, which includes:
  - notification of window damage and size changes.
  - reading and distribution of input events among windows within a process.
  - posting events with the Agent for delivery to other clients.
- Windows as devices, which includes:
  - reading control options such as asynchronous input and non-blocking input.
- The screen abstraction, called a *desktop*, which includes:
  - Routines to initialize new screens so that SunView may be run on them.
  - Multiple screens accessible by a single user.

- The global input abstraction, called a *workstation*, which includes:
  - environment wide input device instantiation.
  - controlling a variety of system performance and user interface options.
  - extending the *Virtual User Input Device* interface with events of your own design.
- Advanced use of the general notification-based flow of control management mechanism called the *Notifier*, which includes:
  - detection of input pending, output completed and exception occurred on a file descriptor.
  - maintenance of interval timers.
  - dispatching of signal notifications.
  - child process status and control facilities.
  - a client event notification mechanism, which can be thought of as a client-defined signal mechanism.
- The *Selection Service*, for exchanging objects and information between cooperative client, both within and between processes.
- The *defaults* mechanism, for maintaining and querying a database of user-settable options.
- Advanced imaging topics, which include:
  - the repair of damaged portions of your window, when not retained.
  - receiving window damage and size change notifications via `SIGWINCH`.
- The mechanisms used to violate window boundaries. You would use them if you created a menu or prompt package.
- Routines to perform *window management* activities such as open, close, move, stretch, top, bottom, refresh. In addition, there are facilities for invoking new tools and positioning them on the screen.
- Routines to manipulate individual rectangles and lists of rectangular areas. They forms what is essentially an algebra of rectangles, useful in computing window overlap, points in windows, etc.
- Advanced *icon* topics, including displaying them, accessing them from a file, their internal structure, etc..
- Advanced *scrollbar* topics, including calculating and performing your own scroll motions (in a canvas, for example).

Finally, there is an appendix on how to write a line discipline for a new input device that you want to access through SunView. Another appendix covers some programming notes.

3

SunView System Model

3

SunView System Model

This chapter presents the system model of SunView. It discusses the hierarchy of abstractions that make up the window system, the data representations of those abstractions and the packages that manage the components.

3.1. A Hierarchy of Abstractions

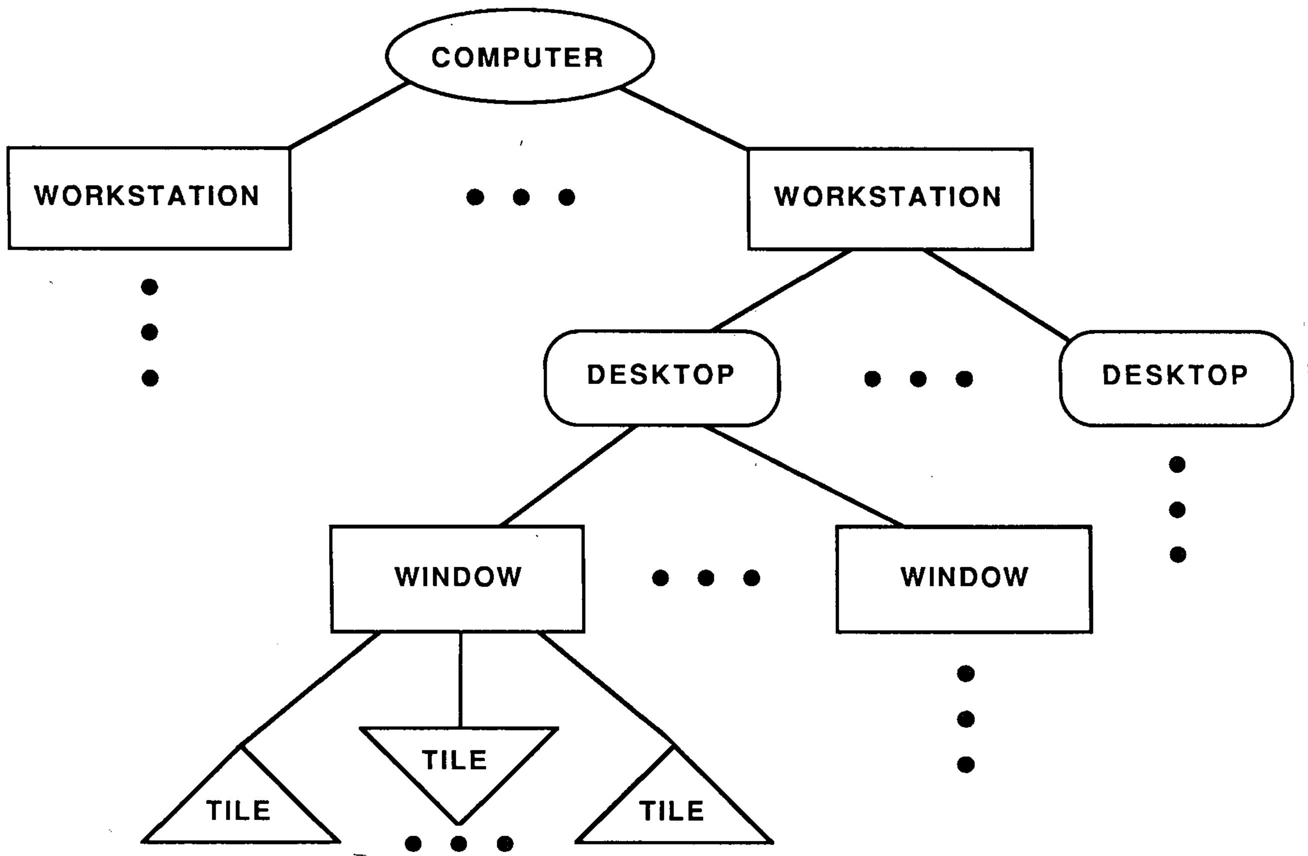
There is a hierarchy of abstractions that make up the window system:

- *Tiles* are used to tile the surface of a window. Tiles don't overlap and may not be nested. For example, a text subwindow with a scrollbar is implemented with separate tiles for both the scrollbar and the text portion of the subwindow.
- Windows are allowed to overlap one another[1] and may be arbitrarily nested. Frames, panels, text subwindows, canvases and the root window are all implemented as windows.
- Screens, sometimes called *desktops*, support multiple windows and represent physical display devices. A screen is covered by the root window.
- *Workstations* support multiple screens that share common user input devices on the behalf a single user. For example, one can slide the cursor between screens.

The figure below shows the hierarchy:

[1] The procedure which lays out subwindows of tools does it so they do not overlap, but this is not an inherent restriction.

Figure 3-1 *SunView System Hierarchy*



Data Managers

The various parts of the system support the management of this hierarchy. They provide the glue between the various components:

- The *window driver*, (currently) residing in the UNIX kernel as a pseudo device driver that is accessed through library routines, supports windows, screens and workstations.
- The pixwin library package allows implementors of specific windows and tiles to access the screen for drawing.
- The Notifier library package is used to support the general flow of control to multiple disjoint clients.
- The Agent library package can be viewed as the SunView-specific extension of the Notifier. The Agent supports tiles and windows.
- The Selection Service is a separate user process that supports the interprocess communication and control of user selection related data. In this role it essentially supports specific tile implementations.

Data Representations

This conceptual model is useful to understand the structure and workings of the system. However, the model doesn't always translate into corresponding objects:

- Tiles are implemented as opaque handles with pixwin regions used to communicate the size and position of the tile to the Agent.
- At the system level, windows are implemented as UNIX *devices* which are represented by *file descriptors*. Window devices are not to be confused with the application level notion of windows which are opaque handles. A *file descriptor* is returned by `open(2)` of an entry in the `/dev` directory. It is manipulated by other system calls, such as `select(2)`, `read(2)`, `ioctl(2)`, and `close(2)`.
- There is a screen structure that describes a limited number of properties of a desktop. However, it is a window file descriptor that is used as the "ticket" into the window driver to get and set screen related data. This is possible because a window is directly associated with a particular screen.
- There is no system object that translates into a workstation. However, like desktop data, workstation-related data is accessed using a window file descriptor. Again, this is because a window is directly associated with a particular screen which is directly associated with a particular workstation. As a side effect of this association, one can use the file descriptor of a panel and asked about workstation related data for the workstation on which the panel resides.

3.2. Model Dynamics

Now that you have been introduced to the players in the window system, let's see how they interact.

Tiles and the Agent

Tiles are quite simple and "lightweight" abstractions. The main reason for having tiles instead of yet another nesting of windows is that file descriptors are relatively heavyweight. There can only be 64 file descriptors open per UNIX process in Sun's release 4.0. As a result, a tile provides only a subset of the functionality of a full-blown window. After telling the Agent that a tile covers a certain portion of the window, the Agent provides the following services:

- The Agent tells you when your tile has been resized.
- The Agent tells you when your tile should be repainted. Optionally, you can tell the Agent to maintain a retained image for your tile from which the Agent can handle the repainting itself.
- The Agent reads input for the tile's window and distributes it to the appropriate tile.
- The Agent notices when tile regions have been entered and exited by the cursor and notifies the tile.

In addition, the Agent is the conduit by which client generated events are passed between tiles. For example, when the scrollbar wants to tell a canvas that it should now scroll, the communications is arranged via the Agent. The Agent, in turn, uses the Notifier to implement the data transfer.

It is your responsibility to lay out your window's tiles so that they don't overlap, even when the window size changes.

Even a window with only a single tile that covers its entire surface may use the Agent and its features.

Windows

Windows are the focus of most of the functionality of the window system. Here is a list of the information about a window maintained by the window system:

- A rectangle refers to the *size* and *position* of a window. Some windows (frames) also utilize an alternative rectangle that describes the iconic position of a window.
- Each window has a series of links that describe the window's position in a hierarchical database, which determines its *overlapping* relationships to other windows. Windows may be arbitrarily nested, providing distinct *subwindows* within an application's screen space.
- Arbitration between windows is provided in the allocation of display space. Where one window limits the space available to another, *clipping*, guarantees that one does not interfere with the other's image. One such conflict arises when windows share the same coordinates on the display: one overlaps the other. Thus, clipping information is associated with each window.
- When one window impacts another window's image without any action on the second window's part, the window system informs the affected window of the damage it has suffered, and the areas that ought to be repaired. To do this the window system maintains a description of the portion of the window of the display that is corrupted as well as the process id of the window's owner.

- On color displays, colormap entries are a scarce resource. When shared among multiple applications, they become even more scarce: there may be simultaneous demand for more colors than the display can support. Arbitration between windows is provided in the allocation of colormap entries. Provisions are made to share portions of the colormap (*colormap segments*). There is colormap information that describes that portion of the colormap assigned to a window.
- Real-time response is important when tracking the cursor, so this is done by the window system. Thus, the image (cursor and optional cross hairs) used to track the mouse when it is in the window is part of the window's data.[2]
- Windows may be selective about which input events they will process, and rejected events will be offered to other windows for processing; you can explicitly designate the window rejected events are first offered to.[3] A mask indicates what keyboard input actions the window should be notified of and there is a similar mask for pick/locator-related actions.
- A window device is read in order to receive the user input events directed at it. So like other input devices a window supports a variety of the input modes, such as blocking or non-blocking, synchronous or asynchronous, etc. In addition, there is a queue of input events that are pending for a window.
- There are 32 bits of data private to the window client stored with the window.

Desktop

Desktop data relates to the physical display:

- The physical display is associated with a UNIX device. The desktop maintains the name of this device.
- The desktop maintains the notion of a default foreground and background color.
- The desktop records the size of the screen.
- The desktop maintains the name of the distinguished root window on itself.
- When multiple screens are part of a workstation, each desktop knows the relative physical placement of its neighboring displays so that the mouse cursor may slide between them.

---

[2] There is only one cursor per window, but the image may be different in different tiles within the window (e.g. scrollbars have different cursors). If so, the different cursor images are dynamically loaded by the user process and thus real time response is not assured.

[3] Not all events are passed on to a designee, for example window-specific events such as `LOC_WINENTER` and `KBD_REQUEST` are not.

Locking

The desktop also arbitrates screen surface access and window database manipulation.

Display Locking

Display locking prevents window processes from interfering with each other in several ways:

- Raster hardware may require several operations to complete a change to the display; one process' use of the hardware is protected from interference by others during this critical interval.
- Changes to the arrangement of windows must be prevented while a process is painting, lest an area be removed from a window as it is being painted.
- A software cursor that the window process does not control (the kernel is usually responsible for the cursor) may have to be removed so that it does not interfere with the window's image.

Window Database Locking

Window database locking is used when doing multiple changes to the window's size, position, or links in the window hierarchy. This prevents any other process from performing a conflicting modification and allows the window system to treat changes as atomic.

Colormap Sharing

On color displays, colormap entries are a limited resource. When shared among multiple applications, colormap usage requires arbitration. Consider the following applications running on the same display at the same time in different windows:

- Application program X needs 64 colors for rendering VLSI images.
- Application program Y needs 32 shades of gray for rendering black and white photographs.
- Application program Z needs 256 colors (assume this is the entire colormap) for rendering full color photographs.

Colormap usage control is handled as follows:

- To determine how X and Y figure out what portion of the colormap they should use (so they don't access each others' entries), the window system provides a resource manager that allocates a *colormap segment* to each window from the *shared colormap*. To reduce duplicate colormap segments, they are named and can be shared among cooperating processes.
- To hide concerns about the correct offset to the start of a colormap segment from routines that access the image, the window system initializes the image of a window with the colormap segment offset. This effectively hides the offset from the application.
- To accommodate Z if its large colormap segment request cannot be granted, Z's colormap is loaded into the hardware, replacing the shared colormap, whenever the cursor is over Z's window. Z's request is not denied even though it is not allocated its own segment in the shared colormap.

Workstations

The domain of a workstation is to manage the global state of input processing. User inputs are unified into a single stream within the window system, so that actions with the user input devices, usually a mouse and a keyboard, can be coordinated. This unified stream is then distributed to different windows, according to user or programmatic indications. To this end a workstation manages the following:

- A workstation needs some number of user input devices to run. A distinguished keyboard device and a distinguished mouse-like device are recognized since these are required for a useful workstation. Non-Sun supported user input devices may be used as these distinguished devices.
- Additional, non-distinguished user input devices, may be managed by a workstation as well.
- The input devices associated with the workstation are polled by the window system. Locator motion causes the cursor to move on the screen. Certain interrupt event sequences are noted. Events are timestamped enqueued on the workstation's input queue based on the time they were generated.
- This input queue is massaged in a variety of ways. If the input queue becomes full, locator motion events on the queue are compressed in order to reduce its size. In addition, locator motion at the head of the queue is (conditionally) collapsed so as to deliver the most up-to-date locator position to applications.
- Based on the state of input focuses and window input masks a window is selected to receive the next event from the head of the input queue. The event is placed on the window device's separate input pending queue and the window's process is awoken.
- The workstation uses a synchronized input mechanism. The main benefit of a synchronized input mechanism is that it removes the input race conditions inherent in a multiple process environment. While a window processes the input event the workstation waits for it to finish before handing out the next event.
- The workstation deals with situations in which a process takes too long to finish processing an input event by pressing on ahead in a partially synchronized mode until the errant process catches up to the user. This prevents a misbehaving process from disabling user interaction with other processes.

4

The Agent & Tiles

4

The Agent & Tiles

This chapter describes how to utilize the Agent to manage tiles for you. It contains the implementation details associated with tiles and the Agent, as introduced in the *SunView System Model* chapter. This chapter uses a text subwindow with a scrollbar as an example of Agent utilization.[4]

4.1. Registering a Tile With the Agent

The Agent is a little funny in that you don't ask it to create a tile for you that it will then manage. In fact tiles are only abstractions. Instead, you create a *pixwin region* and a unique client object and pass these to the Agent to manage on your behalf. The following routine is how this registration is done.

```
int
win_register(client, pw, event_func, destroy_func, flags)
    Notify_client  client;
    Pixwin        *pw;
    Notify_func    event_func;
    Notify_func    destroy_func;
    u_int          flags;

#define PW_RETAIN        0x1
#define PW_FIXED_IMAGE   0x2
#define PW_INPUT_DEFAULT 0x4
#define PW_NO_LOC_ADJUST 0x8
#define PW_REPAINT_ALL   0x10
```

`client` is the handle that the Agent will hand back to you when you are notified of interesting events (see below) by a call to the `event_func` function. `client` is usually the same client handle by which a tile is known to the Notifier. Client handles needs to be unique among all the clients registered with the Notifier.

`pw` is a pixwin opened by `client` and is the pixwin by which the tile writes to the screen. This pixwin could have been created by a call to `pw_open()` if the window has only a single tile that covers its entire surface. More often the tile covers a region of the windows created by a call to `pw_region()`, documented in the *Clipping with Regions* section of the *Imaging Facilities: Pixwins* chapter

[4] The header file `/usr/include/sunwindow/window_hs.h` contains the definitions for the routines in this chapter.

of the *SunView 1 Programmer's Guide*. Regions are themselves pixwins that refer to an area within an existing pixwin.

`flags` control the options utilized by the Agent when managing your tile:

- `PW_RETAIN` — Your tile will be managed as retained. This means that the window system maintains a backup image of your tile in memory from which the screen can be refreshed in case the tile is exposed after being hidden.
- `PW_FIXED_IMAGE` — The underlaying abstraction of the image that your tile is displaying is fixed in size. This means that the client need not be asked to repaint the entire tile on a window size change. Only the newly exposed parts need be repainted.
- `PW_INPUT_DEFAULT` — Usually, the cursor position over a tile indicates which tile input will by sent to. However, if your window has the keyboard focus, the cursor need not be over any tile in your window in order for the window to be sent input. The tile with this flag on will receive input if the cursor is not over any tile in the window. In our example, the text display tile would be created with this flag on because it is the main tile in the window.
- `PW_NO_LOC_ADJUST` — Usually, when the Agent notifies your tile of an event the locator x and y positions contained in your event are adjusted to be relative to the tile's upper left hand corner. Turning this flag on suppresses this action which means that you'll get events in the window's coordinate space.
- `PW_REPAINT_ALL` — Setting this flag causes your tile to be completely repainted when ever the Agent detects that any part of your window needs to be repainted.

`event_func` is the client event notification function for the tile and `destroy_func` is the client destroy function for the tile. The Agent actually sets these functions up with the notifier (see the *Notifier* chapter in the *SunView 1 Programmer's Guide* for a discussion of these two types of notification functions and their calling conventions). In addition, the Agent gets input pending and `SIGWINCH` received (used for repaint and resize detection) notifications from the notifier and posts corresponding events to the appropriate tile. Tiles in the same window need to share the same input pending notification procedure because input is distributed from the kernel at a window granularity. Tiles also share the same input masks, as well as other window data.

Laying Out Tiles

Tiles are used to tile the surface of a window. Tiles may not overlap and may not be nested. As an example, a text subwindow with a scrollbar is implemented with a separate tile for both the scrollbar and the text portion of the subwindow. It is a window owner's responsibility to layout tiles so that they don't overlap. The Agent does nothing for you in this regard, so layout is arranged via conventions among tiles. In our example, there are two tiles, the scrollbar and a text display area. Here is how layout works when scrollbars are involved:

- The text subwindow code creates a vertical scrollbar. The scrollbar code looks at the user's scrollbar defaults and finds out what side to put the scrollbar on and how wide it should be. Given this information it figures out where to place its tile. The scrollbar code registers its new tile with the Agent.
- After creating the scrollbar, the text subwindow code asks the scrollbar what side it is on and how thick it is. Given this information the text subwindow figures out where to place its text display tile. The text subwindow code registers its new tile with the Agent.
- When a window resize notification (sent by the Agent) is received by the scrollbar it knows to hug the side that it is on as it adjusts the size of its region. A similar arrangement is followed by the text display tile.

Dynamically Changing Tile Flags

The following routine lets you dynamically set the tile's flags:

```
int
win_set_flags(client, flags)
    Notify_client  client;
    u_int          flags;
```

A −1 is returned if `client` is not registered, otherwise 0 is returned.

When you set a single flag, it is best to retrieve the state of all the flags first and then operate on the bit that you are changing, then write all the flags back; otherwise, any other flags that are set will be reset. The following routine retrieves the current flags of the tile associated with `client`:

```
u_int
win_get_flags(client)
    Notify_client  client;
```

Extracting Tile Data

Extraction of interesting values from clients of the Agent is done via the following calls:

```
int
win_get_fd(client)
    Notify_client  client;
```

`win_get_fd()` gets the window file descriptor associated with client's tile.

```
Pixwin *
win_get_pixwin(client)
    Notify_client  client;
```

`win_get_pixwin()` gets the pixwin associated with client's tile.

4.2. Notifications From the Agent

Once you register your tile with the Agent, the Agent causes the `event_func` you passed to `win_register()` to be called (''notified'') to handle events. You must write your tile's event notification procedure yourself; the events it might receive are listed in the *Handling Input* chapter in the *SunView 1 Programmer's Guide.*

The calling sequence for any client event notification function is:

```
Notify_value
event_func(client, event, arg, when)
    Notify_client       client;
    Event              *event;
    Notify_arg          arg;
    Notify_event_type   when;
```

`client` is the client handle passed into `win_register()`. `event` is the event your tile is notified of. `arg` is usually `NULL` but depends on `event_id(event)`. In the case of the scrollbar tile notifying the text display tile of a scroll action `arg` is actually defined. `when` is described in the chapter *Advanced Notifier Usage* and is usually `NOTIFY_SAFE`.

What your tile does with events is largely up to you; however, there are a few things to note about certain classes of events.

- For `LOC_RGNENTER` and `LOC_RGNEXIT` to be generated for tiles, `LOC_MOVE`, `LOC_WINENTER` and `LOC_WINEXIT` need to be turned on. Remember that tiles share their window's input mask so they need to cooperate in their use of it.
- Locator coordinate translation is done so that the event is relative to a tile's coordinate system unless disabled by `PW_NO_LOC_ADJUST`.
- On a `WIN_RESIZE` event, you can use `pw_set_region_rect()` to change the size and position of your tile's pixwin region.
- On a `WIN_REPAINT`, you simply repaint your entire tile. The Agent will have set the clipping of your pixwin so that only the minimum portion of the screen will actually appear to repaint. Alternatively, if you have initially told the Agent to maintain a retained image for your tile from which the Agent can handle the repainting itself, you will only get a `WIN_REPAINT` call after a window size change. You won't even get this call if your tile's flags have `PW_FIXED_IMAGE` and `PW_RETAIN` bits turned on.

4.3. Posting Notifications Through the Agent

The Agent is the conduit by which client-generated events are passed between tiles. For example, when the scrollbar wants to tell a canvas that it should now scroll, the communications is arranged via the Agent. The Agent, in turn, uses the Notifier to implement the data transfer.

The Agent follows the lead of the Notifier when it comes to posting events. See the documentation on `notify_post_event()` and `notify_post_event_and_arg()` in the *Advanced Notifier Usage* chapter if you are going to be posting events between tiles.

There are four routines available for posting an event to another tile.

```
Notify_error
win_post_id(client, id, when_hint)
    Notify_client       client;
    short               id;
    Notify_event_type   when_hint;
```

is provided if you want to send an event to a tile and you don't really care about any event data except the `event_id(event)`. The Agent will generate the remainder of the event for you with up-to-date data. `when_hint` is usually `NOTIFY_SAFE`.

A second routine is available if you want to manufacture an event yourself. This is easy if you already have an event in hand.

```
Notify_error
win_post_event(client, event, when_hint)
    Notify_client      client;
    Event             *event;
    Notify_event_type  when_hint;
```

The other two routines parallel the first two but include the capability to pass an arbitrary additional argument to the destination tile. The calling sequence is more complicated because one must make provisions to copy and later free the additional argument in case the delivery of the event is delayed.

```
Notify_error
win_post_id_and_arg(client, id, when_hint, arg,
                    copy_func, release_func)
    Notify_client      client;
    short              id;
    Notify_event_type  when_hint;
    Notify_arg         arg;
    Notify_copy        copy_func;
    Notify_release     release_func;

Notify_error
win_post_event_arg(client, event, when_hint, arg,
                   copy_func, release_func)
    Notify_client      client;
    Event             *event;
    Notify_event_type  when_hint;
    Notify_arg         arg;
    Notify_copy        copy_func;
    Notify_release     release_func;
```

The copy and release functions are covered in the *Advanced Notifier Usage* chapter. After reading about them you will know why you need the following utilities to copy the `event` as well as the `arg`:

```
Notify_arg
win_copy_event(client, arg, event_ptr)
    Notify_client  client;
    Notify_arg     arg;
    Event        **event_ptr;

void
win_free_event(client, arg, event)
    Notify_client  client;
    Notify_arg     arg;
    Event         *event;
```

4.4. Removing a Tile From the Agent

The following call tells the Agent to stop managing the tile associated with `client`.

```
int
win_unregister(client)
    Notify_client client;
```

You should call this from the tile's `destroy_func` that you gave to the Agent in the `win_register()` call. `win_unregister()` also completely removes `client` from having any conditions registered with the Notifier. A –1 is returned if `client` is not registered, otherwise 0 is returned.

5

Windows

5

Windows

This chapter describes the facilities for creating, positioning, and controlling windows. It contains the implementation details associated with window devices, as introduced in the *SunView System Model* chapter.

NOTE *The recommended window programming approach is described in the SunView 1 Programmer's Guide.* You should only resort to the following window device routines if the equivalent isn't available at the higher level. It is possible to use the following routines with a high level SunView `Window` object by passing the file descriptor returned by
`(int) window_get(`*`Window_object,`*` WIN_FD);`

The structure that underlies the operations described in this chapter is maintained within the window system, and is accessible to the client only through system calls and their procedural envelopes; it will not be described here. The window is presented to the client as a *device*; it is represented, like other devices, by a *file descriptor* returned by `open`(2). It is manipulated by other UNIX system calls, such as `select`(2), `read`(2), `ioctl`(2), and `close`(2).[5]

5.1. Window Creation, Destruction, and Reference

As mentioned above, windows are *devices*. As such, they are special files in the `/dev` directory with names of the form "`/dev/win`*n*," where *n* is a decimal number. A window is created by opening one of these devices, and the window name is simply the filename of the opened device.

A New Window

The first process to open a window becomes its *owner*. A process can obtain a window it is guaranteed to own by calling:

```
int
win_getnewwindow()
```

This finds the first unopened window, opens it, and returns a file descriptor which refers to it. If none can be found, it returns −1. A file descriptor, often called the *windowfd*, is the usual handle for a window within the process that opened it.

When a process is finished with a window, it may close it with the standard `close`(2) system call with the window's file descriptor as its argument. As with other file descriptors, a window left open when its owning process terminates

[5] The header file `<sunwindow/window_hs.h>` includes the header files needed to work at this level of the window system. The library `/usr/lib/libsunwindow.a` implements window device routines.

will be closed automatically by the operating system.

Another procedure is most appropriately described at this point, although in fact clients will have little use for it. To find the next available window, `win_getnewwindow()` uses:

```
int
win_nextfree(fd)
    int fd;
```

where `fd` is any valid window file descriptor. The return value is a *window number*, as described in *References to Windows* below; a return value of `WIN_NULLLINK` indicates there is no available unopened window.

An Existing Window

It is possible for more than one process to have a window open at the same time; the section *Providing for Naive Programs* below presents one plausible scenario for using this capability. The window will remain open until all processes which opened it have closed it. The coordination required when several processes have the same window open is described in *Providing for Naive Programs*.

References to Windows

Within the process which created a window, the usual handle on that window is the file descriptor returned by `open`(2) or `win_getnewwindow()`. Outside that process, the file descriptor is not valid; one of two other forms must be used. One form is the *window name* (e.g. `/dev/win12`); the other form is the *window number*, which corresponds to the numeric component of the window name. Both of these references are valid across process boundaries. The *window number* will appear in several contexts below.

Procedures are supplied for converting among various window identifiers. `win_numbertoname()` stores the filename for the window whose number is `winnumber` into the buffer addressed by `name`:

```
win_numbertoname(winnumber, name)
    int winnumber;
    char *name;
```

`name` should be `WIN_NAMESIZE` long, as should all the name buffers in this section.

`win_nametonumber()` returns the window number of the window whose name is passed in `name`:

```
int
win_nametonumber(name)
    char *name;
```

Given a window file descriptor, `win_fdtoname()` stores the corresponding device name into the buffer addressed by `name`:

```
win_fdtoname(windowfd, name)
    int    windowfd;
    char *name;
```

`win_fdtonumber()` returns the window number for the window whose file descriptor is `windowfd`:

```
int
win_fdtonumber(windowfd)
    int windowfd;
```

5.2. Window Geometry

Once a window has been opened, its size and position may be set. The same routines used for this purpose are also helpful for adjusting the screen positions of a window at other times, when the window is to be moved or stretched, for instance. `win_setrect()` copies the `rect` argument into the rectangle of the indicated window:

```
win_setrect(windowfd, rect)
    int   windowfd;
    Rect *rect;
```

This changes its size and/or position on the screen. The coordinates in the `rect` structure are in the coordinate system of the window's parent. The *Rects and Rectlists* chapter explains what is meant by a `rect`. *Setting Window Links* below explains what is meant by a window's "parent." Changing the size of a window that is visible on the screen or changing the window's position so that more of the window is now exposed causes a chain of events which redraws the window. See the section entitled *Damage* in the *Advanced Imaging* chapter.

Querying Dimensions

The window size querying procedures are:

```
win_getrect(windowfd, rect)
    int   windowfd;
    Rect *rect;

win_getsize(windowfd, rect)
    int   windowfd;
    Rect *rect;

short win_getheight(windowfd)
    int  windowfd;

short win_getwidth(windowfd)
    int  windowfd;
```

`win_getrect()` stores the rectangle of the window whose file descriptor is `windowfd` into the `rect`; the origin is relative to that window's parent.

`win_getsize()` is similar, but the rectangle is self-relative — that is, the origin is (0,0).

`win_getheight()` and `win_getwidth()` return the single requested dimension for the indicated window — these are part of the `rect` structure that the other calls return.

The Saved Rect

A window may have an alternate size and location; this facility is useful for storing a window's iconic position that is associated with frames. The alternate rectangle may be read with `win_getsavedrect()`, and written with `win_setsavedrect()`.

```
win_getsavedrect(windowfd, rect)
     int    windowfd;
     Rect *rect;

win_setsavedrect(windowfd, rect)
     int    windowfd;
     Rect *rect;
```

As with `win_getrect()` and `win_setrect()`, the coordinates are relative to the screen.

5.3. The Window Hierarchy

Position in the window database determines the nesting relationships of windows, and therefore their overlapping and obscuring relationships. Once a window has been opened and its size set, the next step in creating a window is to define its relationship to the other windows in the system. This is done by setting links to its neighbors, and inserting it into the window database.

Setting Window Links

The window database is a strict hierarchy. Every window (except the root) has a parent; it also has 0 or more *siblings* and *children*. In the terminology of a family tree, *age* corresponds to *depth* in the layering of windows on the screen: parents underlie their offspring, and older windows underlie younger siblings which intersect them on the display. Parents also enclose their children, which means that any portion of a child's image that is not within its parent's rectangle is clipped. Depth determines overlapping behavior: the *uppermost* image for any point on the screen is the one that gets displayed. Every window has links to its parent, its older and younger siblings, and to its oldest and youngest children.

Windows may exist outside the structure which is being displayed on a screen; they are in this state as they are being set up, for instance.

The links from a window to its neighbors are identified by *link selectors*; the value of a link is a *window number*. An appropriate analogy is to consider the *link selector* as an array index, and the associated *window number* as the value of the indexed element. To accommodate different viewpoints on the structure there are two sets of equivalent selectors defined for the links:

```
WL_PARENT          == WL_ENCLOSING
WL_OLDERSIB        == WL_COVERED
WL_YOUNGERSIB      == WL_COVERING
WL_OLDESTCHILD     == WL_BOTTOMCHILD
WL_YOUNGESTCHILD   == WL_TOPCHILD
```

A link which has no corresponding window, for example, a child link of a "leaf" window, has the value `WIN_NULLLINK`.

When a window is first created, all its links are null. Before it can be used for anything, at least the parent link must be set so that other routines know with which desktop and workstation this window is to be associated. If the window is to be attached to any siblings, those links should be set in the window as well. The individual links of a window may be inspected and changed by the following procedures.

`win_getlink()` returns a window number.

```
int
win_getlink(windowfd, link_selector)
    int windowfd, link_selector;
```

This number is the value of the selected link for the window associated with `windowfd`.

```
win_setlink(windowfd, link_selector, value)
    int windowfd, link_selector, value;
```

`win_setlink()` sets the selected link in the indicated window to be `value`, which should be another window number or `WIN_NULLLINK`. The actual window number to be supplied may come from one of several sources. If the window is one of a related group, all created in the same process, file descriptors will be available for the other windows. Their window numbers may be derived from the file descriptors via `win_fdtonumber()`. The window number for the parent of a new window or group of windows is not immediately obvious, however. The solution is a convention that the `WINDOW_PARENT` environment parameter will be set to the filename of the parent. See `we_setparentwindow()` for a description of this parameter.

Activating the Window

Once a window's links have all been defined, the window is inserted into the tree of windows and attached to its neighbors by a call to

```
win_insert(windowfd)
    int windowfd;
```

This call causes the window to be inserted into the tree, and all its neighbors to be modified to point to it. This is the point at which the window becomes available for display on the screen.

Every window should be inserted after its rectangle(s) and link structure have been set, but the insertion need not be immediate: if a subtree of windows is being defined, it is appropriate to create the window at the root of this subtree, create and insert all of its descendants, and then, when the subtree is fully defined, insert its root window. This activates the whole subtree in a single action, which may result in cleaner display of the whole tree.

Defaults

One need not specify all the sibling links of a window that is being inserted into the display tree. Sibling links may be defaulted as follows (these conventions are applied in order):

- If the `WL_COVERING` sibling link is `WIN_NULLLINK` then the window is put on the top of the heap of windows.

- If the `WL_COVERED` sibling link is `WIN_NULLLINK` then the window is put on the bottom of the heap of windows.
- If the `WL_COVERED` or `WL_COVERING` sibling links are invalid then the window is put on the bottom of the heap of windows.

Once a window has been inserted in the window database, it is available for input and output. At this point, it is appropriate to access the screen with pixwin calls (to draw something in the window!).

Modifying Window Relationships

Windows may be rearranged in the tree. This will change their overlapping relationships. For instance, to bring a window to the top of the heap, it should be moved to the "youngest" position among its siblings. And to guarantee that it is at the top of the display heap, each of its ancestors must likewise be the youngest child of *its* parent.

To accomplish such a modification, the window should first be removed:

```
win_remove(windowfd)
     int windowfd;
```

After the window has been removed from the tree, it is safe to modify its links, and then reinsert it.

A process doing multiple window tree modifications should lock the window tree before it begins. This prevents any other process from performing a conflicting modification. This is done with a call to:

```
win_lockdata(windowfd)
     int windowfd;
```

After all the modifications have been made and the windows reinserted, the lock is released with a call to:

```
win_unlockdata(windowfd)
     int windowfd;
```

Nested pairs of calls to lock and unlock the window tree are permitted. The final unlock call actually releases the lock.

If a client program uses any of the window manager routines, use of `win_lockdata()` and `win_unlockdata()` is not necessary. See the chapter on *Window Management* for more details.

Most routines described in this chapter, including the four above, will block temporarily if another process either has the database locked, or is writing to the screen, and the window adjustment has the possibility of conflicting with the window that is being written.

As a method of deadlock resolution, `SIGXCPU` is sent to a process that spends more that 2 seconds of process virtual time inside a window data lock, and the lock is broken.[6]

[6] The section *Kernel Tuning Options* in the *Workstation* chapter describes how to modify this default number of seconds (see `ws_lock_limit`).

Window Enumeration

There are routines that pass a client-defined procedure to a subset of the tree of windows, and another that returns information about an entire layer of the window tree. They are useful in performing window management operations on groups of windows. The routines and the structures they use and return are listed in <sunwindow/win_enum.h>.

Enumerating Window Offspring

The routines win_enumerate_children() and win_enumerate_subtree() repeatedly call the client's procedure passing it the windowfds of the offspring of the client window, one after another:

```
enum win_enumerator_result
win_enumerate_children(windowfd, proc, args);
    Window_handle  windowfd;
    Enumerator     proc;
    caddr_t        args;

enum win_enumerator_result
win_enumerate_subtree(windowfd, proc, args);
    Window_handle  windowfd;
    Enumerator     proc;
    caddr_t        args;

enum win_enumerator_result \
    { Enum_Normal, Enum_Succeed, Enum_Fail };

typedef enum win_enumerator_result
    (*Enumerator)();
```

windowfd is the window whose children are enumerated (Window_handle is typedef'd to int). Both routines repeatedly call proc(), stopping when told to by proc() or when everything has been enumerated.

proc() is passed a windowfd and args

```
(*proc)(fd, args);
```

It does whatever it wants with the windowfd, then returns win_enumerator_result. If proc() returns Enum_Normal then the enumeration continues; if it returns Enum_Succeed or Enum_Fail then the enumeration halts, and win_enumerate_children or win_enumerate_subtree() returns the same result.

The difference between the two enumeration procedures is that win_enumerate_children() invokes proc() with an fd for each *immediate* descendant of windowfd in oldest-to-youngest order, while win_enumerate_subtree() invokes proc() with a windowfd for the *original* window windowfd and for *all* of its descendants in depth-first, oldest-to-youngest order. The former enumerates windowfd's children, the latter enumerates windowfd and its extended family.

It is possible that win_enumerate_subtree() can run out of file descriptors during its search of the tree if the descendants of windowfd are deeply nested.

Fast Enumeration of the Window Tree

The disadvantage with the above two routines is that they are quite slow. They traverse the window tree, calling `win_getlink()` to find the offspring, then open each window, then call the procedure.

In 3.2 there is a fast window routine, `win_get_tree_layer()`, that returns information about all the children of a window in a single ioctl[7]:

```
win_get_tree_layer(parent, size, buffer);
Window_handle  parent;
u_int           size;
char           *buffer;

typedef struct win_enum_node {
    unsigned char me;
    unsigned char parent;
    unsigned char upper_sib;
    unsigned char lowest_kid;
    unsigned int flags;
#define WIN_NODE_INSERTED      0x1
#define WIN_NODE_OPEN          0X2
#define WIN_NODE_IS_ROOT       0x4
    Rect open_rect;
    Rect icon_rect;
} Win_enum_node;
```

`win_get_tree_layer()` fills `buffer` with `Win_enum_node` information (rects, user_flags, and minimal links) for the children of `window` in oldest-to-youngest order. It returns the number of bytes of `buffer` filled with information, or negative if there is an error.

Unlike `win_enumerate_children()`, `win_get_tree_layer()` returns information for all the children of `windowfd` including those that are have not been installed in the window tree with `win_insert()`; such children will not have the `WIN_NODE_INSERTED` flag set.

5.4. Pixwin Creation and Destruction

A pixwin is the object that you use to access the screen. Its usage has been covered in the *Imaging Facilities: Pixwins* chapter of the *SunView Programmer's Guide* and in the previous chapter on tiles. How to create a pixwin region has also been covered in the same places. Here we cover how a pixwin is generated for a window.

Creation

To create a pixwin, the window to which it will refer must already exist. This task is accomplished with procedures described earlier in this chapter. The pixwin is then created for that window by a call to `pw_open()`:

```
Pixwin *
pw_open(windowfd)
    int windowfd;
```

`pw_open()` takes a file descriptor for the window on which the pixwin is to

[7] `win_get_tree_layer()` will use the slower method if the kernel does not support the ioctl; thus programs that use this can be run on 3.0 systems.

write. A pointer to a pixwin struct is returned. At this point the pixwin describes the exposed area of the window.

Region

To create the pixwin for a tile, call `pw_region()` passing it the pixwin returned from `pw_open()`.

Retained Image

If the client wants a *retained pixwin*, the `pw_prretained` field of the pixwin should be set to point to a memory pixrect of your own creation. If you set this field you need to call `pw_exposed(pw)` afterwards.[8] This updates the pixwin's exposed area list to deal with the memory pixrect; see the *Advanced Imaging* chapter for more information on `pw_expose()`.

Bell

The following routine can be used to beep the keyboard bell and flash a pixwin:

```
win_bell(windowfd, wait_tv, pw)
     int      windowfd;
     struct   timeval wait_tv;
     Pixwin *pw;
```

If `pw` is 0 then there is no flash. `wait_tv` controls the duration of the bell.[9]

Destruction

When a client is finished with a pixwin, it should be released by a call to:

```
pw_close(pw)
     Pixwin *pw;
```

`pw_close()` frees any resource associated with the pixwin, including its `pw_prretained` pixrect if any. If the pixwin has a lock on the screen, it is released.

5.5. Choosing Input

The chapter entitled *Handling Input* in the *SunView Programmer's Guide* describes the window input mechanism. This section describes the file descriptor level interface to setting a window's input masks. This section is very terse, assuming that the concept from *Handling Input* are well understood.

Input Mask

Clients specify which input events they are prepared to process by setting the input masks for each window being read. The calls in this section manipulate input masks.

[8] The best way to manage a retained window is to let the Agent do it (see `win_register()`).

[9] The bell's behavior is controlled by the SunView `defaultsedit` entries *SunView/Audible_Bell* and *Sunview/Visible_Bell*, so the sound and flash can be disabled by the user, regardless of what the call to `win_bell()` specifies.

```
typedef struct inputmask {
      short   im_flags;
#define IM_NEGEVENT      (0x01)  /* send input negative events too */
#define IM_ASCII         (0x10)  /* enable ASCII codes 0-127 */
#define IM_META          (0x20)  /* enable META codes 128-255 */
#define IM_NEGASCII      (0x40)  /* enable negative ASCII codes 0-127 */
#define IM_NEGMETA       (0x80)  /* enable negative META codes 128-255 *,
#define IM_INTRANSIT     (0x400) /* don't surpress locator events when
                                    in-transit over window */
      ...
} Inputmask;
```

Manipulating the Mask Contents

The bit flags defined for the input mask are stored directly in the `im_flags` field. To set a particular event in the input mask use the following macro:

```
win_setinputcodebit(im, code)
    Inputmask *im;
    u_short    code;
```

`win_setinputcodebit()` sets a bit indexed by `code` in the input mask addressed by `im` to 1.

```
win_unsetinputcodebit(im, code)
    Inputmask *im;
    u_short    code;
```

`win_unsetinputcodebit()` resets the bit to zero.

The following macro is used to query the state of an event code in an input mask:

```
int
win_getinputcodebit(im, code)
    Inputmask *im;
    u_short    code;
```

`win_getinputcodebit()` returns non-zero if the bit indexed by `code` in the input mask addressed by `im` is set.

`input_imnull()` initializes an input mask to all zeros:

```
input_imnull(im)
    Inputmask *im;
```

It is critical to initialize the input mask explicitly when the mask is defined as a local procedure variable.

Setting a Mask

The following routines set the keyboard and pick input masks for a window. The different types of masks are discussed in the *Input* chapter.

```
win_set_kbd_mask(windowfd, im)
    int        windowfd;
    Inputmask *im;

win_set_pick_mask(windowfd, im)
    int        windowfd;
    Inputmask *im;
```

Querying a Mask

The following routines get the keyboard and pick input masks for a window.

```
win_get_kbd_mask(windowfd, im)
    int         windowfd;
    Inputmask *im;

win_get_pick_mask(windowfd, im)
    int         windowfd;
    Inputmask *im;
```

The Designee

The designee is that window that input is directed to if the input mask for a window doesn't match a particular event:

```
win_get_designee(windowfd, nextwindownumber)
    int windowfd, *nextwindownumber;

win_set_designee(windowfd, nextwindownumber)
    int windowfd, nextwindownumber;
```

5.6. Reading Input

The recommended way of getting input is to let the Agent notify you of input events (see chapter on tiles). However, there are times when you may want to read input directly, say, when tracking the mouse until one of its buttons goes up. A library routine exists for reading the next input event for a window:

```
int
input_readevent(windowfd, event)
    int     windowfd;
    Event *event;
```

This fills in the `event` struct, and returns 0 if all went well. In case of error, it sets the global variable `errno`, and returns −1; the client should check for this case.

Non-blocking Input

A window can be set to do either blocking or non-blocking reads via a standard `fcntl`(2) system call, as described in `fcntl`(2) (using `_SETFL`) and `fcntl`(5) (using `FNDELAY`). A window defaults to blocking reads. The blocking status of a window can be determined by the `fcntl`(2) system call.

When all events have been read and the window is doing non-blocking I/O, `input_readevent()` returns −1 and the global variable `errno` is set to `EWOULDBLOCK`.

Asynchronous Input

A window process can ask to be sent a `SIGIO` if any input is pending in a window. This option is also enabled via a standard `fcntl`(2) system call, as described in `fcntl`(2) (using `F_SETFL`) and `fcntl`(5) (using `FASYNC`). The programmer can set up a signal handler for `SIGIO` by using the `notify_set_signal_func()` call.[10]

[10] The Notifier handles asynchronous input without you having to set up your own signal handler if you are using the Notifier to determine when there is input for a window.

Events Pending

The number of character in the input queue of a window can be determined via a `FBIONREAD` `ioctl(2)` call. `FBIONREAD` is described in `tty(4)`. Note that the value returned is the number of bytes in the input queue. If you want the number of `Events` then divide by `sizeof(Event)`.

5.7. User Data

Each window has 32 bits of data associated with it. These bits are used to implement a minimal inter-process window-related status-sharing facility. Bits 0x01 through 0x08 are reserved for the basic window system; 0x01 is currently used to indicate if a window is a blanket window. Bits 0x10 through 0x80 are reserved for the user level window manager; 0x10 is currently used to indicate if a window is iconic. Bits 0x100 through 0x80000000 are available for the programmer's use. They is manipulated with the following procedures:

```
int
win_getuserflags(windowfd)
    int windowfd;

int
win_setuserflags(windowfd, flags)
    int windowfd;
    int flags;

int
win_setuserflag(windowfd, flag, value)
    int windowfd;
    int flag;
    int value;
```

`win_getuserflags()` returns the user data. `win_setuserflags()` stores its `flags` argument into the window struct. `win_setuserflag()` uses `flag` as a mask to select one or more flags in the data word, and sets the selected flags on or off as `value` is `TRUE` or `FALSE`.

5.8. Mouse Position

Determining the mouse's current position is treated in the *SunView Programmer's Guide*. The convention for a process tracking the mouse is to arrange to receive an input event every time the mouse moves; the mouse position is passed with *every* user input event a window receives.

The mouse position can be reset under program control; that is, the cursor can be moved on the screen, and the position that is given for the mouse in input events can be reset without the mouse being physically moved on the table top.

```
win_setmouseposition(windowfd, x, y)
    int windowfd, x, y;
```

`win_setmouseposition()` puts the mouse position at (x, y) in the coordinate system of the window indicated by `windowfd`. The result is a jump from the previous position to the new one without touching any points between. Input events occasioned by the move, such as window entry and exit and cursor changes, will be generated. This facility should be used with restraint, as many users are unhappy with a cursor that moves independent of their control.

Occasionally it is necessary to discover which window underlies the cursor, usually because a window is handling input for all its children. The procedure used for this purpose is:

```
int
win_findintersect(windowfd, x, y)
    int windowfd, x, y;
```

where windowfd is the calling window's file descriptor, and (x, y) defines a screen position in that window's coordinate space. The returned value is a window number of a child of the calling window. If a child of the calling window doesn't fall under the given position WIN_NULLLINK is returned.

5.9. Providing for Naive Programs

There is a class of applications that are relatively unsophisticated about the window system, but want to run in windows anyway. For example, a graphics program may want a window in which to run, but doesn't want to know about all the details of creating and positioning it. This section describes a way of allowing for these applications.

Which Window to Use

The window system defines an important environment parameter, WINDOW_GFX. By convention, WINDOW_GFX is set to a string that is the device name of a window in which graphics programs should be run. This window should already be opened and installed in the window tree. Routines exist to read and write this parameter:

```
int
we_getgfxwindow(name)
    char *name

we_setgfxwindow(name)
    char *name
```

we_getgfxwindow() returns a non-zero value if it cannot find a value.

The Blanket Window

A good way to take over an existing window is to create a new window that becomes attached to and covers the existing window. Such a covering window is called a *blanket* window. The covered window will be called the *parent* window in this subsection because of its window tree relationship with a blanket window.[11]

The appropriate way to make use of the blanket window facility is as follows: Using the parent window name from the environment parameter WINDOW_GFX (described above), open(2) the parent window. Get a new window to be used as the blanket window using win_getnewwindow(). Now call:

```
int
win_insertblanket(blanketfd, parentfd)
    int blanketfd, parentfd;
```

A *zero* return value indicates success. As the parent window changes size and position the blanket window will automatically cover the parent.

[11] It's a bad idea to take over an existing window using win_setowner().

To remove the blanket window from on top of the parent window call:

```
win_removeblanket(blanketfd)
     int blanketfd;
```

If the process that owns the window over which the blanket window resides dies before `win_removeblanket()` is called, the blanket window will automatically be removed and destroyed.

A non-zero return value from `win_isblanket()` indicates that `blanketfd` is indeed a blanket window.

```
int
win_isblanket(blanketfd)
     int blanketfd;
```

5.10. Window Ownership

`SIGWINCH` signals are directed to the process that *owns* the window, the owner normally being the process that created the window. The following procedures read from and write to the window:[12] These routines are included for backwards compatibility.

```
int
win_getowner(windowfd)
     int windowfd;

win_setowner(windowfd, pid)
     int windowfd, pid;
```

`win_getowner()` returns the process id of the indicated window owner. If the owner doesn't exist, zero is returned. `win_setowner()` makes the process identified by `pid` the owner of the window indicated by `windowfd`. `win_setowner` causes a `SIGWINCH` to be sent to the new owner.

5.11. Environment Parameters

Environment parameters are used to pass well-established values to an application. They have the valuable property that they can communicate information across several layers of processes, not all of which have to be involved.

Every frame must be given the name of its *parent window*. A frame's parent window is the window in the display tree under which the frame window should be displayed. The environment parameter `WINDOW_PARENT` is set to a string that is the device name of the parent window. For a frame, this will usually be the name of the root window of the window system.

```
we_setparentwindow(windevname)
     char *windevname;
```

sets `WINDOW_PARENT` to `windevname`.

[12] Do not use the two routines in this section for *temporarily* taking over another window.

```
int
we_getparentwindow(windevname)
    char *windevname;
```

gets the value of WINDOW_PARENT into windevname. The length of this string should be at least WIN_NAMESIZE characters long, a constant found in <sunwindow/win_struct.h>. A non-zero return value means that the WINDOW_PARENT parameter couldn't be found.

The environment parameter DEFAULT_FONT should contain the font file name used as the program's default (see pf_default()).

NOTE *This is retained for backwards compatibility. All programs set this variable, but only old-style SunWindows programs, gfx subwindow programs and raw pixwin programs use it to determine which font to use. SunView programs that don't set their own font use the SunView/Font defaults entry; you can use the* **-Wt** *fontname command line frame argument to change the font of SunView programs that allow it.*

5.12. Error Handling

Except as explicitly noted, the procedures described in this section do not return error codes. The standard error reporting mechanism inside the *sunwindow* library is to call an error handling routine that displays a message, typically identifying the ioctl(2) call that detected the error. This error message is somewhat cryptic; Appendix B, *Programming Notes*, has a section on *Error Message Decoding*. After the message display, the calling process resumes execution.

This default error handling routine may be replaced by calling:

```
int (*win_errorhandler(win_error))()
    int (*win_error)();
```

The win_errorhandler() procedure takes the address of one procedure, the new error handler, as an argument and returns the address of another procedure, the old error handler, as a result. Any error handler procedure should be a function that returns an integer.

```
win_error(errnum, winopnum)
    int errnum, winopnum;
```

errnum will be –1 indicating that the actual error number is found in the global errno. winopnum is the ioctl(2) number that defines the window operation that generated the error. See Section B.4, *Error Message Decoding*, in Appendix B, *Programming Notes*.

6

Desktops

6

Desktops

This chapter discusses the calls that affect the screen, or desktop. Some calls affect workstation-related data, i.e., global input related data. This overlap of the conceptual model is purely historical.

Look at `sunview`

Many of the routines in here are used by Sun's window manager, `sunview(1)`. You will find it very helpful to look at the source for `sunview` (it is optional software that must be loaded in `suninstall`) to see how it uses these routines.

6.1. Multiple Screens

Workstations may use multiple displays, and clients may want windows on all of them.[13] Therefore, the window database is a forest, with one tree of windows for each display. There is no overlapping of window trees that belong to different screens. For displays that share the same mouse device, the physical arrangement of the displays can be passed to the window system, and the mouse cursor will pass from one screen to the next as though they were continuous.

The `singlecolor` Structure

The `screen` structure describes attributes of the display screen. First, here is the definition of `singlecolor`, which it uses for the foreground and background colors:

```
struct singlecolor  {
    u_char  red, green, blue;
};
```

[13] There can be as many screens as there are frame buffers on your machine and *dtop* pseudo devices configured into your kernel. The kernel calls screen instances *dtops*.

The screen Structure

Now the `screen` structure:

```
struct screen   {
    char        scr_rootname[SCR_NAMESIZE];
    char        scr_kbdname[SCR_NAMESIZE];
    char        scr_msname[SCR_NAMESIZE];
    char        scr_fbname[SCR_NAMESIZE];
    struct      singlecolor scr_foreground;
    struct      singlcolor scr_background;
    int         scr_flags;
    struct      rect scr_rect;
};

#define SCR_NAMESIZE 20
#define SCR_SWITCHBKGRDFRGRD 0x1
```

`scr_rootname` is the device name of the window which is at the base of the window display tree for the screen; it is often `/dev/win0`.[14] `scr_kbdname` is the device name of the keyboard associated with the screen; the default is `/dev/kbd`. `scr_msname` is the device name of the mouse associated with the screen; the default is `/dev/mouse`. `scr_fbname` is the device name of the frame buffer on which the screen is displayed; the default is `/dev/fb` for the first desktop. `scr_kbdname`, `scr_msname` and `scr_fbname` can have the string "NONE" if no device of the corresponding type is to be associated with the screen. Workstations (hence also desktops) can have additional input devices associated with them; see the section on *User Input Device Control* in the *Workstations* chapter.

`scr_foreground` is three RGB color values that define the foreground color used on the frame buffer; the default is {*colormap size–1*, *colormap size–1*, *colormap size–1*}. `scr_background` is three RGB color values that define the background color used on the frame buffer; the default is `{0, 0, 0}`. The default values of the background and foreground yield a black on white image. `scr_flags` contains boolean flags; the default is 0. `SCR_SWITCHBKGRDFRGRD` is a flag that directs any client of the background and foreground data to switch their positions, thus providing a video reversed image (usually yielding a white on black image). `scr_rect` is the size and position of the screen on the frame buffer; the default is the entire frame buffer surface.

Screen Creation

To create a new screen call:

```
int
win_screennew(screen)
    struct screen *screen;
```

[14] Multiple screen configurations, in particular, will not have `/dev/win0` as the root window on the second screen.

win_screennew() opens and returns a window file descriptor for a root "desktop" window. This new root window resides on the new screen which was defined by the specifications of screen. Any zeroed field in screen tells win_screennew() to use the default value for that field (see above for defaults). Also, see the description of win_initscreenfromargv() below. If −1 is returned, an error message is displayed to indicate that there was some problem creating the screen.

Initializing the screen Structure

The following routine can be called before calling win_screennew():

```
int
win_initscreenfromargv(screen, argv)
     struct   screen *screen;
     char   **argv;
```

win_initscreenfromargv() zeroes the *screen structure, then it parses the relevant command line arguments in argv into *screen. You then call win_screennew() to creates a root window with the desired attributes. The command line arguments allow the user to set all the variables in *screen including the display device, the keyboard device, the mouse device, the foreground and background colors, whether the screen colors should be inverted, and other features. See sunview(1) for semantics and details.

Screen Query

To find out about the screen on which your window is running call:

```
win_screenget(windowfd, screen)
     int            windowfd;
     struct screen *screen;
```

win_screenget() fills in the addressed struct *screen with information for the screen with which the window indicated by windowfd is associated. You can call this from **any** window.

Screen Destruction

To destroy the screen on which your window is running call:

```
win_screendestroy(windowfd)
     int windowfd;
```

win_screendestroy() causes each window owner process (except the invoking process) on the screen associated with windowfd to be sent a SIGTERM signal. This call will block until all the processes have died. If a window owner process hasn't gone away after 15 seconds, it is sent a SIGKILL, which will destroy it.

Screen Position

To tell the window system how multiple screens are arranged call:

```
win_setscreenpositions(windowfd, neighbors)
    int windowfd, neighbors[SCR_POSITIONS];

#define  SCR_NORTH  0
#define  SCR_EAST   1
#define  SCR_SOUTH  2
#define  SCR_WEST   3

#define  SCR_POSITIONS  4
```

`win_setscreenpositions()` informs the window system of the logical layout of multiple screens. This enables the cursor to cross to the appropriate screen. `windowfd`'s window is the root for its screen; the four slots in `neighbors` should be filled in with the window numbers of the root windows for the screens in the corresponding positions. No diagonal neighbors are defined, since they are not strictly neighbors.

`win_getscreenpositions()` fills in `neighbors` with `windowfd`'s screen's neighbors:

```
win_getscreenpositions(windowfd, neighbors)
    int windowfd, neighbors[SCR_POSITIONS];
```

CAUTION **In these routines, `windowfd` must be an fd for the root window.** Most operations on the screen can be done using any windowfd.

Accessing the Root FD

The following code fragment gets the `screen` struct for your window, then opens the window device of the root window:

```
int mywinfd,      rootfd;
struct screen     rootscreen;
...
win_screenget(mywinfd, &rootscreen);
rootfd = open(rootscreen.scr_rootname);
```

7

Workstations

7

Workstations

This chapter discusses the manipulation of workstation data, which comprises mostly global data related to input and input devices. Some calls in the *Desktops* chapter also affect workstations. This overlap is purely historical. This chapter also explains parts of the Virtual User Input Device interface that were not covered in the *Handling Input* chapter of the *SunView 1 Programmer's Guide.* That chapter gave the possible event codes in SunView 1; this chapter explains the mechanism which sets up input devices to generate them.

7.1. Virtual User Input Device

The Virtual User Input Device (*vuid*) is an interface between input devices and their clients. The interface defines an idealized user input device that may not correspond to any existing physical collection of input devices. A client of SunView doesn't access vuid devices directly. Instead, the window system reads vuid devices, serializing input from all the vuid devices and then makes the input available to windows as SunView vuid events.

NOTE *You don't have to write a vuid interface to use your own device in SunView: you can use any input device that generates ASCII. But if your device is hooked up using vuid, then your SunView programs can interface with it using the SunView input event mechanism.*

Since SunView's input system is built on top of vuid, it is explained in some detail.

What Kind of Devices?

Vuid is targeted to input devices that gather command data from humans, e.g., mice, keyboards, tablets, joysticks, light pens, knobs, sliders, buttons, ascii terminals, etc.[15] The vuid interface is not designed to support input devices that produce voluminous amounts of data, such as input scanners, disk drives, voice packets.

Here are some of the properties that are expected of a typical client of vuid, e.g., SunView:

- The client has a richer user interface than can be supported by a simple ASCII terminal.

[15] The appendix titled *Writing a Virtual User Input Device Driver* discusses how to write a device driver that speaks the vuid protocol for a new input device.

- The client serializes multiple input devices being used by the user into a single stream of events.
- The client preserves the entire state of its input so that it may query this state.

Vuid Features

Vuid provides, among others, the following services to clients:

- A client may extend the capabilities of the predefined vuid by adding input devices. A client wants to be able to do this in a way that fits smoothly with its existing input paradigm.
- A client's code may be input device independent. A client can replace the physical device(s) underlying the virtual user input device and not have to change any input or event handlers, only the input device driver. In fact, the vuid interface doesn't care about physical devices. One physical device can masquerade as many logical devices and many physical devices can look like a single logical device.

Vuid Station Codes

This section defines the layout of the address space of vuid station codes. It explains how to extend the vuid address space for your own purposes. The meaning of vuid station codes is covered in the *Handling Input* chapter of the *SunView 1 Programmer's Guide*. The programmatic details of the vuid interface are covered in *Writing a Virtual User Input Device Driver* appendix to this document,

Address Space Layout

The address space for vuid events is 16 bits long, from 0 to 65535 inclusive. It is broken into 256 *segments* that are 256 entries long (`VUID_SEG_SIZE`). The top 8 bits contain a vuid segment identifier value. The bottom 8 bits contain a segment specific value from 0 to 255. Some segments have been predefine and some are available for expansion. Here is how the address space is currently broken down:

- `ASCII_DEVID` (0x00) — ASCII codes, which include `META` codes.
- `TOP_DEVID` (0x01) — Top codes, which are ASCII with the 9th bit on.
- Reserved (0x02 to 0x7B) — for Sun vuid implementations.
- `SUNVIEW_DEVID` (0x7C) — SunView semantic "action" events generated inside SunView by keymapping.
- `PANEL_DEVID` (0x7D) — Panel subwindow package event codes used internally in the panel package (see `<suntool/panel.h>`).
- `SCROLL_DEVID` (0x7E) — Scrollbar package event codes passed to scrollbar clients on interesting scrollbar activity (see `<suntool/scrollbar.h>`).

This device is a bit of a hodge-podge for historical reasons; the middle of the address space has SunView-related events in it (see `<sunwindow/win_input.h>`), and the virtual keyboard and virtual locator are thrown together.

- `WORKSTATION_DEVID` (0x7F) — Virtual keyboard and locator (mouse) related event codes that describe a basic "workstation" device collection (see `<sundev/vuid_event.h>`).
- Reserved for Sun customers (0x80 to 0xFF) — if you are writing a new vuid, you can use a segment in here; see the next section.

Adding a New Segment

`<sundev/vuid_event.h>` is the central registry of virtual user input devices. To allocate a new vuid you must modify this file:

- Choose an unused portion of the address space. Vuids from 0x00 to 0x7F are reserved for use by Sun. Vuids from 0x80 to 0xFF are reserved for Sun customers.
- Add the new device with a `*_DEVID` `#define` in this file. Briefly describe the purpose/usage of the device. Mention the place where more information can be found.
- Add the new device to the `Vuid_device` enumeration with a `VUID_`*devname* entry .
- List the specific event codes in **another** header file that is specific to the new device. `ASCII_DEVID`, `TOP_DEVID` and `WORKSTATION_DEVID` events are listed in `<sundev/vuid_event.h>` for historical reasons.

NOTE *A new vuid device can just as easily be a pure software construction as it can be a set of unique codes emitted by a new physical device driver.*

Input State Access

The complete state of the virtual input device is available. For example, one can ask questions about the up/down state of arbitrary keys.

```
int
win_get_vuid_value(windowfd, id)
    int    windowfd;
    short  id;
```

`id` is one of the event codes from `<sundev/vuid_event.h>` or `<sunwindow/win_input.h>`. `windowfd` can be any window file descriptor.

The result returned for keys is 0 for key is up and 1 for key is down; some vuid events return a range of numbers, such as mouse position. There is no error code for "no such key" because, by definition, the vuid event address space is the entire range of `shorts` and therefore you can't ask an incorrect question. 0 is the default event state.

Unencoded Input

Unencoded keyboard input is supported, for those customers who cannot use the normal keyboard input mechanism .

A new keyboard translation was introduced in 3.2. The type of translation is set by the `KIOCTRANS` ioctl (see `kb`(4S) and `kbd`(4S)). Old values were:

`TR_NONE`	for unencoded keyboard input outside the window system
`TR_ASCII`	for ASCII events (characters and escape sequences) outside the window system
`TR_EVENT`	for window input events inside the window system

A new value is now supported:

`TR_UNTRANS_EVENT`
gives unencoded keyboard values for input events inside the window system.

The client must have `WIN_ASCII_EVENTS` set in the window's input mask; if up-transitions are desired, `WIN_UP_ASCII_EVENTS` must also be set. (See Chapter 6, *Handling Input*, in the *SunView Programmer's Guide* for how to set input masks.)

The number of the key pressed or released will be passed in the event's id `ie_code`, and the direction of the transition will be reported correctly by `event_is_up()` and `event_is_down()` (i.e., the `NEGEVENT` flag in `ie_flags` will be correct). The state of the shiftmask is undefined.

Events for other input (e.g. from the mouse) will be merged in the same stream with keyboard input, in standard window input fashion.

NOTE *Setting the keyboard translation has a global effect — it is not possible to get encoded input in one window and unencoded input in another on the same workstation.*

7.2. User Input Device Control

The number and kind of physical user input devices that can be used to drive SunView is open ended. Here are the controls for manipulating those devices.

Distinguished Devices

A keyboard and a mouse-like device are distinguished and settable directly. The *Desktops* chapter describes how they are specified in the `screen` structure. Here are two calls for changing them explicitly.

```
int
win_setkbd(windowfd, screen)
    int             windowfd;
    struct screen *screen;
```

changes the keyboard associated with `windowfd`'s desktop. Only the data pertinent to the keyboard is used (i.e., `screen->scr_kbdname`).

```
int
win_setms(windowfd, screen)
    int             windowfd;
    struct screen *screen;
```

changes the mouse associated with `windowfd`'s desktop. Only the data pertinent to the mouse is used (i.e., `screen->scr_msname`).

Arbitrary Devices

Arbitrary user input devices may be used to drive a workstation. However, some care must be exercised in selecting the combinations of devices. To install an input device with SunView, call `win_set_input_device()`.

```
int
win_set_input_device(windowfd, inputfd, name)
    int   windowfd;
    int   inputfd;
    char *name;
```

`windowfd` identifies (by association) the workstation on which the input device is to be installed. `name` is used to identify the device on subsequent calls to SunView, e.g., */dev/kbd*. `name` may only be `SCR_NAMESIZE` characters long. Before calling this routine, open the input device and make any `ioctl`(2) calls to it to set it up to your requirements, e.g. possibly setting the speed of the serial port through which the device in coming in on. Pass the open file descriptor in as `inputfd`.

`win_set_input_device()` sends additional `ioctl`(2) calls to make the device operate as a Virtual User Input Device (if that is not its native mode) and operate in non-blocking read mode. The device's unread input is flushed. SunView starts reading from the device. Once `win_set_input_device()` returns, close `inputfd`. This action won't actually close the device; SunView has its own open file descriptor for the device.

Non-Vuid Devices

User input devices that only emit ASCII, and not vuid events, may be used by SunView. If the device does not respond to probing with the vuid ioctls SunView assumes it is an ASCII device and reads it one character at a time. Thus, SunView can handle input from a simple ASCII terminal without modification to any drivers. The routines in the section can be used with vuid or ASCII devices.

Device Removal

To remove an input device from SunView, call `win_remove_input_device()`.

```
int
win_remove_input_device(windowfd, name)
    int   windowfd;
    char *name;
```

`windowfd` identifies the workstation from which to remove the input device. `name` identifies the device. SunView resets the device to its original state.

Device Query

To ask if an input device is being utilized by a workstation, call `win_is_input_device()`.

```
int
win_is_input_device(windowfd, name)
    int   windowfd;
    char *name;
```

`windowfd` identifies the workstation being probed. `name` identifies the device. 0 is returned if the device is not being utilized, 1 is returned if it is, and −1 is returned if there is an error.

Device Enumeration

To ask what all the input devices of the workstation are, call `win_enum_input_device()` which enumerates them all.

```
int
win_enum_input_device(windowfd, func, data)
    int       windowfd;
    int       (*func)();
    caddr_t   data;
```

`windowfd` identifies the workstation being probed. You pass the function `func` which is called once for every input device. The first argument passed to `func()` is a string which is the name of the device. The second argument passed to `func()` is `data`, which can be anything you want. If `func` returns something other than 0 the enumeration is terminated early. `win_enum_input_device()` returns –1 if there was an error during the enumeration, 0 if it went smoothly and 1 if `func` terminated the enumeration early.

7.3. Focus Control

The concept of a split keyboard and pick input focus has been described in the *SunView 1 Programmer's Guide*. The user interface documentation describes it as "click to type" mode. It allows keyboard input events to be directed to a different window than the window that pick (cursor) inputs are sent to. Usually you want the keyboard input focus to stay in one window while the pick input focus is the window under the cursor.

Keyboard Focus Control

The following routine is called when a window gets a `KBD_REQUEST` event and the window doesn't need the keyboard focus.

```
win_refuse_kbd_focus(windowfd)
    int windowfd;
```

The following routine is used to change the keyboard focus. It is only a hint; the target window can refuse the keyboard focus or the user may not be running in click-to-type mode.

```
int
win_set_kbd_focus(windowfd, number)
    int windowfd, number;
```

`number` is the window that you want to have the keyboard focus.

The following routine gets the window number of the window that is currently the keyboard focus.

```
int
win_get_kbd_focus(windowfd)
    int windowfd;
```

Event Specification

This section describes how to programmatically specify which user actions are used as the focus control actions. The `sunview(1)` program has a set of flags to control the keyboard focus.

Setting the Caret Event

One of the ways to change the keyboard focus is to *set* the caret. Setting the focus passes the focus change event through to the application.

```
void
win_set_focus_event(windowfd, fe, shifts)
    int         windowfd;
    Firm_event *fe;
    int         shifts;
```

`windowfd` identifies the workstation. `fe` is a *firm event* pointer; the entire `Firm_event` structure is defined in the appendix titled *Writing a Virtual User Input Device Driver*. Only the `id` and the `value` fields are utilized in this call. The `id` field of `*fe` is set to the identifier of the event that is used to set the keyboard focus, e.g., `MS_LEFT`. The `value` field is set to the value of the event that is used to set the keyboard focus, e.g., 0 (up) or 1 (down). `shifts` is a mask of shift bits that indicate the required state of the shift keys needed in order to have the event described by `fe` treated as the keyboard focus change event. −1 means that you don't care. If you do care, use the same shift bits passed in the `Event` structure as discussed in the *Handling Input* chapter in the *SunView 1 Programmer's Guide*, e.g., `LEFTSHIFT`.

Getting the Caret Event

`win_get_focus_event()` returns the values set by `win_set_focus_event()`.

```
void
win_get_focus_event(windowfd, fe, shifts)
    int         windowfd;
    Firm_event *fe;
    int        *shifts;
```

`*fe` and `*shifts` are filled in with the current values.

Restoring the Caret

Another ways to change the keyboard focus is to *restore* the caret. Restoring the focus swallows the focus change event so that it never makes it to the application. These two routines parallel the focus setting routines described above.

```
void
win_set_swallow_event(windowfd, fe, shifts)
    int         windowfd;
    Firm_event *fe;
    int         shifts;

void
win_get_swallow_event(windowfd, fe, shifts)
    int windowfd;
    Firm_event *fe;
    int *shifts;
```

7.4. Synchronization Control

This section discusses the concept of input synchronization in some detail. It's an important but subtle system mechanism. You should understand it fully before changing the system's default synchronization setting.

When running with input synchronization enabled, only one input event is being consumed, or processed, at a time. This event is called the *current event.* Ownership of the current event is bestowed by SunView to a single process; that process is said to have the *current event lock.* The lock belongs to a process, not a window device and is used to prevent any process from receiving an input event (via a `read(2)` system call) until the the lock has been released. This prevents race conditions between processes. This lets, for example, a user pop a window to the top and start typing to it before its image is drawn and have typing directed to the correct window. Input synchronization allows the process that currently has the lock to change its input mask and have the change recognized immediately so that applications won't miss events when they fall behind the user.

Input synchronization is not to be confused with the management of the input focus. The input focus is that window which is supposed to get the *next* input event and the input synchronization mechanism is used to determine when the *current* input event processing is completed.

The current event lock is acquired upon completing a read of a window device in which an input event was successfully read. For the duration of the lock, the current event lock owner decides what to do based on the current event and does it (or forks a process to do it). There is no notification to the input focus of input pending until the current event lock is released. Thus, there is typically only one process actively reading input at a time (except for rogue polling processes).

Releasing the Current Event Lock

When a process finishes with the current event, the current event lock is released via:

- A `read(2)` of the next event. If the next event is for the window that is doing the read then the lock is released and re-acquired immediately.
- A `select(2)` for input. This is the common case.
- An explicit `win_release_event_lock()` call (see below).

An explicit lock release call is appropriate for an application that knows that it no longer needs to query the state of the virtual input device or change its input mask and is about to do something moderately time consuming. Such an application can explicitly release the lock as soon as it recognizes that the event it has just read is not going to change event distribution.

```
win_release_event_lock(windowfd)
    int windowfd;
```

Current Event Lock Breaking

The current event lock is broken by SunView when the process with it visits the debugger or dies.

In addition, SunView explicitly breaks the current event lock if an application takes too long to process the event. The time is measured in process virtual time, not real time. This is a quiet lock breaking in that no message is displayed and no signal is sent to the offending process. The duration of the time limit can be set, for the entire workstation, to a range of values:

- 0 — Windows have rampant race conditions. There is poor performance on high speed mouse tracking because the system can't compress mouse motion passed to applications.
- non-zero (approximately 1–10 seconds) — Most synchronization problems go away except when programs exceed the time limit. When SunView detects a process that exceeds the time limit the process temporarily goes into an unsynchronized mode until it catches up with the user.
- large–infinite (greater than 10 seconds) — Synchronization problems don't arise. Unfortunately, the user is locked into just one program at a time echoing/noticing input. The key combination [Setup-I] (the [Setup] key is the [Stop] key on Sun-2 and Sun-3 machines) explicitly breaks the lock.

Getting/Setting the Event Lock Timeout

The default time limit is 2 cpu seconds. You can get the current event lock timeout with a call to `win_get_event_timeout()`:

```
void
win_get_event_timeout(windowfd, tv)
     int             windowfd;
     struct timeval *tv;
```

`*tv` is filed in with the current value.

You can set the current event lock timeout via a call to `win_set_event_timeout()`:

```
void
win_set_event_timeout(windowfd, tv)
     int             windowfd;
     struct timeval *tv;
```

`*tv` is used as the current value.

7.5. Kernel Tuning Options

Some kernel tuning variable are settable using a debugger. However, you are advised not to change these unless you absolutely have to. You can look at the kernel with a debugger to see the default settings of these values, but your are playing with fire.

- `int ws_vq_node_bytes` is the number bytes to use for the input queue. You might increase this number if you find your are getting "Window input queue overflow!" and "Window input queue flushed!" messages. This needs to be modified before starting SunView in order to have any affect.
- `int ws_fast_timeout` is the number of hertz between polls of input devices when in fast mode. SunView polls its input devices at two speeds. The fast mode is the normal polling speed and the slow mode occurs when no action has been detected in the input devices for `ws_fast_poll_duration` hertz. This is all meant to save cpu cycles of useless polling when the user is not doing anything. The system is constantly bouncing between slow and fast polling mode.

ws_fast_timeout should never be 0. Decreasing this number improves interactive cursor tracking at the expense of increased system polling load.

- int ws_slow_timeout is the number of hertz between polls of input devices when in slow mode. ws_slow_timeout should never be 0. Decreasing this number improves interactive cursor tracking at the expense of increased system polling load.

- int ws_fast_poll_duration is discussed above. Increasing this number improves interactive performance at the expense of increased system polling load.

- int ws_loc_still is the number of hertz after which, if the locator has been still, a LOC_STILL event is generated.

- struct timeval ws_lock_limit is the process virtual time limit for a data or display lock. Increasing ws_lock_limit reduces the number of

  ```
  ... lock broken after time limit exceeded ...
  ```

  console messages at the expense of slower response to dealing with lock hogs.

- int ws_check_lock and struct timeval ws_check_time The check for ws_lock_limit doesn't start for ws_check_lock amount of real time after the lock is set. This is done to avoid system overhead for normal short lock intervals. Increasing ws_check_lock reduces system overhead on long lock holding situations at the expense of slower response to dealing with lock hogs.

- int win_disable_shared_locking is a flag that controls whether or not the window driver will try to reduce the overhead of display locking by using a shared memory mechanism. Even though there are no known problems with the shared memory locking mechanism, this variable is available as an escape hatch. If the window system leaves mouse cursor droppings, set this variable to 1. The default is 0. Setting this variable to 1 will result in reduced graphics performance.

- int winclistcharsmax is the maximum number of characters from the operating system's "character buffer" pool that SunView is willing to utilize. Upping this number can reduce "tossed" input situations. Turning on wintossmsg (an int) will print a message if input has to be tossed. winclistcharsmax should only be increased by half again as much as its default.

- int ws_set_favor is a flag that controls whether or not the window driver will try to boost the priority of the window process (and its children) that has the current event lock. The default is 1. In very tight memory situations this dramatically improves interactive performance.

Changing the User Actions that Affect Input

The following is provided so that you can change the user actions for the various real time interrupt actions.

```
typedef struct ws_usr_async {
    short    dont_touch1;
    short    first_id;      /* id of the 1st event */
    int      first_value;   /* value of the 1st event */
    short    second_id;     /* id of the 2nd event */
    int      second_value;  /* value of the 2nd event */
    int      dont_touch2;
} Ws_usr_async

Ws_usr_async ws_break_default = /* Event lock breaking */
     {0, SHIFT_TOP, 1, TOP_FIRST + 'i', 1, 0};
Ws_usr_async ws_stop_default =  /* Stop event */
     {0, SHIFT_TOP, 1, SHIFT_TOP, 0, 0};
Ws_usr_async ws_flush_default = /* Input queue flushing */
     {0, SHIFT_TOP, 1, TOP_FIRST + 'f', 1, 0};
```

8

Advanced Notifier Usage

8

Advanced Notifier Usage

8.1. Overview

This chapter continues the description of the Notifier in *The Notifier* chapter of the *SunView 1 Programmer's Guide*.

Contents

This chapter presents areas which are not of general interest to the majority of SunView application programmers. These include:

- The registration of client, output and exception event handlers.
- Querying for the current address of one of a client's event handlers.
- Interposition of any event handler.
- Controlling the order in which a client receives events.
- Control over the dispatching of events.
- Controlling the order in which clients are notified.
- A list of error codes.
- Restrictions on calls into the Notifier.
- A list of open issues surrounding the Notifier.

Viewpoint

Although the Notifier falls under the umbrella of SunView, the Notifier can be looked at as a library package that is usable separately from the rest of SunView. The viewpoint of this chapter is one in which the Notifier stands alone from SunView. However, there are notes about SunView's usage of the Notifier throughout the chapter.

Further Information

You must read the chapter titled *The Notifier* in the *SunView 1 Programmer's Guide* before you tackle this chapter; it has information and examples about the Notifier and SunView's usage of it. In addition, *The Agent & Tiles* chapter in this manual has further information.

This split description of the Notifier may be a little awkward for advanced users of the Notifier but is much less confusing for the majority of users. You should refer to the index in the *SunView 1 Programmer's Guide* first and then the index in this book when using this material in a reference fashion.

8.2. Notification

This section presents the programming interface to the Notifier that clients use to register event handlers and receive notifications.

Links to the *SunView 1 Programmer's Guide.*

Only those areas not covered in the *Notifier* chapter of the *SunView 1 Programmer's Guide* are presented. In particular, *input pending* refers to the section in the other manual.

The two Notifier chapters are different in that the *SunView 1 Programmer's Guide* considers the Notifier in relation to SunView; thus, for example, in it notifier events are SunView Input Events, so their type is `Event *`. In this chapter, `event` is the more general type `Notify_event`.

Client Events

This section describes how *client events* are handled by the Notifier. From the Notifier's point of view, client events are defined and generated by the client. Client events are not interpreted by the Notifier in any way. The Notifier doesn't detect client events, it just detects UNIX-related events. The Notifier is responsible for dispatching client events to the client's event handler after the event has been *posted* with the Notifier by application code (see the section entitled *Posting* below).

Delivery Times

The Notifier normally sends client event notifications when it is *safe* to do so. This may involve some delay between when an event is posted and when it is delivered. However, a client may ask to always be *immediately* notified of the posting of a client event (see *Posting*, below).

Client Defined Signals

The immediate client event notification mechanism should be viewed as an extension of the UNIX signaling mechanism in which events are client defined signals. However, clients are strongly encouraged to only use safe client event handlers.

Handler Registration

To register a client event handler call:

```
Notify_func
notify_set_event_func(client, event_func, when)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  when;

enum notify_event_type {
    NOTIFY_SAFE=0,
    NOTIFY_IMMEDIATE=1,
};
typedef enum notify_event_type Notify_event_type;
```

`when` indicates whether the event handler will accept notifications only when it is safe (`NOTIFY_SAFE`) or at less restrictive times (`NOTIFY_IMMEDIATE`).[16]

[16] For a rundown of the basics of registering event handlers see the section on *Event Handling* in the *Notifier* chapter of the *SunView 1 Programmer's Guide*.

The Event Handler

The calling sequence of a client event handler is:

```
Notify_value
event_func(client, event, arg, when)
    Notify_client       client;
    Notify_event        event;
    Notify_arg          arg;
    Notify_event_type   when;

typedef caddr_t Notify_arg;
```

in which `client` is the client that called `notify_set_event_func()`. `event` is passed through from `notify_post_event()` (see *Posting*, below). `arg` is an additional argument whose type is dependent on the value of `event` and is completely defined by the client, like `event`. `when` is the actual situation in which `event` is being delivered (`NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`) and may be different from `when_hint` of `notify_post_event()`. The return value is one of `NOTIFY_DONE` or `NOTIFY_IGNORED`.

SunView Usage

You will almost certainly not need to directly register your own client event handler when using SunView. Window objects do this for themselves when they are created. However, note the following:

- A window has a client event handler that you may want to interpose in front of. See the section entitled *Monitoring and Modifying Window Behavior* in the *Notifier* chapter in the *SunView 1 Programmer's Guide*.
- SunView client event handlers are normally registered with `when` equal to `NOTIFY_SAFE`.
- The Agent reads input events from a window's file descriptor and posts them to the client via the `win_post_event()` call. See the section titled *Notifications From the Agent* in *The Agent & Tiles* chapter.

Output Completed Events

Notifications for output completed notifications are similar to input pending notifications, covered in the chapter on the Notifier in the *SunView 1 Programmer's Guide*.

```
Notify_func
notify_set_output_func(client, output_func, fd)
    Notify_client  client;
    Notify_func    output_func;
    int            fd;

Notify_value
output_func(client, fd)
    Notify_client  client;
    int            fd;
```

Exception Occurred Events

Exception occurred notifications are similar to input pending notifications. The only known devices that generate exceptions at this time are stream-based socket connections when an out-of-band byte is available. Thus, a SIGURG signal catcher is set up by the Notifier, much like SIGIO for asynchronous input.

```
Notify_func
notify_set_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;

Notify_value
exception_func(client, fd)
    Notify_client  client;
    int            fd;
```

Getting an Event Handler

Here is the list of routines that allow you to retrieve the value of a client's event handler. The arguments to each notify_get_*_func() function parallel the associated notify_set_*_func() function described elsewhere except for the absence of the event handler function pointer. Thus, we don't describe the arguments in detail here. Refer back to the associated notify_set_*_func() descriptions for details.[17]

A return value of NOTIFY_FUNC_NULL indicates an error. If client is unknown then notify_errno is set to NOTIFY_UNKNOWN_CLIENT. If no event handler is registered for the specified event then notify_errno is set to NOTIFY_NO_CONDITION. Other values of notify_errno are possible, depending on the event, e.g., NOTIFY_BAD_FD if an invalid file descriptor is specified (see the associated notify_set_*_func()).

Here is a list of event handler retrieval routines:

```
Notify_func
notify_get_input_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_func
notify_get_event_func(client, when)
    Notify_client      client;
    Notify_event_type  when;

Notify_func
notify_get_output_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_func
```

[17] It is recommended that you use the Notifier's interposition mechanism instead of trying to do interposition yourself using these notify_get_*_func() routines.

```
notify_get_exception_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_func
notify_get_itimer_func(client, which)
    Notify_client  client;
    int            which;

Notify_func
notify_get_signal_func(client, signal, mode)
    Notify_client       client;
    int                 signal;
    Notify_signal_mode  mode;

Notify_func
notify_get_wait3_func(client, pid)
    Notify_client  client;
    int            pid;

Notify_func
notify_get_destroy_func(client)
    Notify_client  client;
```

8.3. Interposition

There are many reasons why an application might want to interpose a function in the call path to a client's event handler:

- An application may want to use the fact that a client has received a particular notification as a trigger for some application-specific processing.
- An application may want to filter the notifications to a client, thus modifying the client's behavior.
- An application may want to extend the functionality of a client by handling notifications that the client is not programmed to handle.

The Notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client. Here is a typical example of interposition:

- An application creates a client. The client has set up its own client event handler using `notify_set_event_func()`.
- The application tells the Notifier that it wants to interpose its function in front of the client's event handler by calling `notify_interpose_event_func()` (described below).
- When the application's interposed function is called, it tells the Notifier to call the next function, i.e., the client's function, via a call to `notify_next_event_func()` (described below).

Registering an Interposer

The following routines let you interpose your own function in front of a client's event handler. The arguments to each `notify_interpose_*_func()` function parallel the associated `notify_set_*_func()` function described above. Thus, we don't describe the arguments in detail here. Refer back to the associated `notify_set_*_func()` descriptions for details.

NOTE *The one exception to this rule is that the arguments to* `notify_interpose_itimer_func()` *are a subset of the arguments to* `notify_set_itimer_func()`.

```
Notify_error
notify_interpose_input_func(client, input_func, fd)
    Notify_client  client;
    Notify_func    input_func;
    int fd;

Notify_error
notify_interpose_event_func(client, event_func, when)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  when;

Notify_error
notify_interpose_output_func(client, output_func, fd)
    Notify_client  client;
    Notify_func    output_func;
    int            fd;
```

```
Notify_error
notify_interpose_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;

Notify_error
notify_interpose_itimer_func(client, itimer_func, which)
    Notify_client  client;
    Notify_func    itimer_func;
    int            which;

Notify_error
notify_interpose_signal_func(client, signal_func,
                             signal, mode)
    Notify_client       client;
    Notify_func         signal_func;
    int                 signal;
    Notify_signal_mode  mode;

Notify_error
notify_interpose_wait3_func(client, wait3_func, pid)
    Notify_client  client;
    Notify_func    wait3_func;
    int            pid;

Notify_error
notify_interpose_destroy_func(client, destroy_func)
    Notify_client  client;
    Notify_func    destroy_func;
```

The return values from these functions may be one of:

- NOTIFY_OK — The interposition was successful .
- NOTIFY_UNKNOWN_CLIENT — client is not known to the Notifier.
- NOTIFY_NO_CONDITION —nn There is no event handler of the type specified.
- NOTIFY_FUNC_LIMIT — The current implementation allows five levels of interposition for every type of event handler, the original event handler registered by the client plus five interposers. NOTIFY_FUNC_LIMIT indicates that this limit has been exceeded.

If the return value is something other than NOTIFY_OK then notify_errno contains the error code.

Invoking the Next Function

Here is the list of routines that you call from your interposed function in order to invoke the next function in the interposition sequence. The arguments and return value of each notify_next_*_func() function are the same as the arguments passed to the your interposer function. Thus, we don't describe the arguments in detail here. Refer back to the associated event handler descriptions for details.

```
Notify_value
notify_next_input_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_value
notify_next_event_func(client, event, arg, when)
    Notify_client      client;
    Notify_event      *event;
    Notify_arg         arg;
    Notify_event_type  when;

Notify_value
notify_next_output_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_value
notify_next_exception_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_value
notify_next_itimer_func(client, which)
    Notify_client  client;
    int            which;

Notify_value
notify_next_signal_func(client, signal, mode)
    Notify_client       client;
    int                 signal;
    Notify_signal_mode  mode;

Notify_value
notify_next_wait3_func(client, pid, status, rusage)
    Notify_client  client;
    union          wait status;
    struct rusage  rusage;
    int            pid;

Notify_value
notify_next_destroy_func(client, status)
    Notify_client   client;
    Destroy_status  status;
```

Removing an Interposed Function

Here is the list of routines that allow you to remove the interposer function that you installed using a `notify_interpose_*_func()` call. The arguments to each `notify_remove_*_func()` function is exactly the same as the associated `notify_set_*_func()` function described above. Thus, we don't describe the arguments in detail here.

NOTE *The one exception to this rule is that the arguments to* `notify_remove_itimer_func()` *are a subset of the arguments to* `notify_set_itimer_func()`.

```
Notify_error
notify_remove_input_func(client, input_func, fd)
    Notify_client  client;
    Notify_func    input_func;
    int            fd;

Notify_error
notify_remove_event_func(client, event_func, when)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  when;

Notify_error
notify_remove_output_func(client, output_func, fd)
    Notify_client  client;
    Notify_func    output_func;
    int            fd;

Notify_error
notify_remove_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;

Notify_error
notify_remove_itimer_func(client, itimer_func, which)
    Notify_client  client;
    Notify_func    itimer_func;
    int            which;

Notify_error
notify_remove_signal_func(client, signal_func, signal, mode)
    Notify_client       client;
    Notify_func         signal_func;
    int                 signal;
    Notify_signal_mode  mode;

Notify_error
notify_remove_wait3_func(client, wait3_func, pid)
    Notify_client  client;
    Notify_func    wait3_func;
    int            pid;

Notify_error
notify_remove_destroy_func(client, destroy_func)
    Notify_client  client;
    Notify_func    destroy_func;
```

If the return value is something other than `NOTIFY_OK` then `notify_errno` contains the error code. The error codes are the same as those associated with `notify_interpose_*_func()` calls.

8.4. Posting

This section describes how to post client events and destroy events with the Notifier.

Client Events

A client event may be posted with the Notifier at any time. The poster of a client event may suggest to the Notifier when to deliver the event, but this is only a hint. The Notifier will see to it that it is delivered at an appropriate time (more on this below). The call to post a client event is:

```
typedef char * Notify_event;

Notify_error
notify_post_event(client, event, when_hint)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
```

The `client` handle from `notify_set_event_func()` is passed to `notify_post_event()`. `event` is defined and interpreted solely by the `client`. A return code of `NOTIFY_OK` indicates that the notification has been posted. Other values indicate an error condition. `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is unknown to the Notifier. `NOTIFY_NO_CONDITION` indicates that `client` has no client event handler registered with the Notifier.

Delivery Time Hint

Usually it is during the call to `notify_post_event()` that the client event handler is called. Sometimes, however, the notification is queued up for later delivery. The Notifier chooses between these two possibilities by noting which kinds of client event handlers `client` has registered, whether it is safe and what the value of `when_hint` is. Here are the cases broken down by which kinds of client event handlers `client` has registered:

- **Immediate only** — Whether `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE` the event is delivered immediately.
- **Safe only** — Whether `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE` the event is delivered when it is safe.
- **Both safe and immediate** — A client may have both an immediate client event handler as well as a safe client event handler. If `when_hint` is `NOTIFY_SAFE` then the notification is delivered to the safe client event handler when it is safe. If `when_hint` is `NOTIFY_IMMEDIATE` then the notification is delivered to the immediate client event handler right away. If the immediate client event handler returns `NOTIFY_IGNORED` then the same notification will be delivered to the safe client event handler when it is safe.

Actual Delivery Time

For client events, other than knowing which event handler to call, the main function of the Notifier is to know when to make the call. The Notifier defines when it is safe to make a client notification. If it is not safe, then the event is queued up for later delivery. Here are the conventions:

- A client that has registered an immediate client event handler is sent a notification as soon as it is received. The client has complete responsibility for handling the event safely. It is rarely safe to do much of anything when an event is received asynchronously. Usually, just setting a flag that indicates that the event has been received is about the safest thing that can be done.

- A client that has registered a safe client event handler will have a notification queued up for later delivery when the notification was posted during an asynchronous signal notification. Immediate delivery is not safe because your process, just before receiving the signal, may have been executing code at any arbitrary place.
- A client that has registered a safe client event handler will have a notification queued up for later delivery if the client's safe client event handler hasn't returned from processing a previous event. This convention is mainly to prevent the cycle: Notifier notifies *A*, who notifies *B*, who notifies *A*. *A* could have had its data structures torn up when it notified *B* and was not in a state to be reentered.

Implied in these conventions is that a safe client event handler is called immediately from other UNIX event handlers. For example:

- A client's input pending event handler is called by the Notifier.
- Two characters are read by the client's input pending event handler.
- The first character is given to the Notifier to deliver to the client's safe event handler.
- The Notifier immediately delivers the character to the client event handler.
- Returning back to the input pending event handler, the second character is sent. This character is also delivered immediately.

Posting with an Argument

SunView posts a fixed field structure with each event. Sometimes additional data must be passed with an event. for instance when the scrollbar posts an event to its owner to do a scroll. The scrollbars' handle is passed as an argument along with the event. `notify_post_event_and_arg()` provides this argument passing mechanism (see below).

Storage Management

When posting a client event there is the possibility of delivery being delayed. In the case of SunView, the event being posted is a pointer to a structure. The Notifier avoids an invalid (dangling) pointer reference by copying the event if delivery is delayed. It calls routines the client supplies to copy the event information and later to free up the storage the copy uses. `notify_post_event_and_arg()` provides this storage management mechanism.

```
Notify_error
notify_post_event_and_arg(client, event, when_hint, arg,
                          copy_func, release_func)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
    Notify_arg         arg;
    Notify_copy        copy_func;
    Notify_release     release_func;

typedef caddr_t Notify_arg;

typedef Notify_arg (*Notify_copy)();
#define NOTIFY_COPY_NULL    ((Notify_copy)0)

typedef void (*Notify_release)();
#define NOTIFY_RELEASE_NULL ((Notify_release)0)
```

`copy_func()` is called to copy `arg` (and optionally `event`) when `event` and `arg` needed to be queued for later delivery. `release_func()` is called to release the storage allocated during the copy call when `event` and `arg` were no longer needed by the Notifier.

Any of `arg`, `copy_func()` or `release_func()` may be null. If `copy_func` is not `NOTIFY_COPY_NULL` and `arg` is `NULL` then `copy_func()` is called anyway. This allows `event` the opportunity to be copied because `copy_func()` takes a pointer to `event`. The pointed to event may be replaced as a side affect of the copy call. The same applies to a non-`NOTIFY_RELEASE_NULL` release function with a NULL `arg` argument.

The `copy()` and `release()` routines are client-dependent so you must write them yourself. Their calling sequences follow:

```
Notify_arg
copy_func(client, arg, event_ptr)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event  *event_ptr;

void
release_func(client, arg, event)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event   event;
```

SunView Usage

There are Agent calls to post an event to a tile that provide a layer over the posting calls described here (see `win_post_event()` in the chapter entitled *The Agent & Tiles*).

Posting Destroy Events

When a destroy notification is set, the Notifier also sets up a synchronous signal condition for SIGTERM that will generate a DESTROY_PROCESS_DEATH destroy notification. Otherwise, a destroy function will not be called automatically by the Notifier. One or two (depending on whether the client can veto your notification) explicit calls to notify_post_destroy() need be made.

```
Notify_error
notify_post_destroy(client, status, when)
    Notify_client       client;
    Destroy_status      status;
    Notify_event_type   when;
```

NOTIFY_INVAL is returned if status or when is not defined. After notifying a client to destroy itself, all references to client are purged from the Notifier.

Delivery Time

Unlike a client event notification, the Notifier doesn't try to detect when it is safe to post a destroy notification. Thus, a destroy notification can come at any time. It is up to the good judgement of a caller of notify_post_destroy() or notify_die() (described in the section titled *Notifier Control*) to make the call at a time that a client is not likely to be in the middle of accessing its data structures.

Immediate Delivery

If status is DESTROY_CHECKING and when is NOTIFY_IMMEDIATE then notify_post_destroy() may return NOTIFY_DESTROY_VETOED if the client doesn't want to go away.

Safe Delivery

Often you want to tell a client to go away at a safe time. This implies that delivery of the destroy event will be delayed, in which case the return value of notify_post_destroy() can't be NOTIFY_DESTROY_VETOED because the client hasn't been asked yet. To get around this problem the Notifier will flush the destroy event of a checking/destroy pair of events if the checking phase is vetoed. Thus, a common idiom is:

```
(void) notify_post_destroy(client, DESTROY_CHECKING,
                           NOTIFY_SAFE);
(void) notify_post_destroy(client, DESTROY_CLEANUP,
                           NOTIFY_SAFE);
```

8.5. Prioritization

The order in which a particular client's conditions are notified may be controlled by providing a *prioritizer* operation.[18]

The Default Prioritizer

The default prioritizer makes its notifications in this order (any asynchronous or immediate notifications have already been sent):

- Interval timer notifications (`ITIMER_REAL` and then `ITIMER_VIRTUAL`).
- Child process control notifications.
- Synchronous signal notifications by ascending signal numbers.
- Exception file descriptor activity notifications by ascending fd numbers.
- Handle client events by order in which received.
- Output file descriptor activity notifications by ascending fd numbers.
- Input file descriptor activity notifications by ascending fd numbers.

Providing a Prioritizer

This section describes how a client can provide its own prioritizer.

```
Notify_func
notify_set_prioritizer_func(client, prioritizer_func)
    Notify_client   client;
    Notify_func     prioritizer_func;
```

`notify_set_prioritizer_func()` takes an opaque client handle and the function to call before any notifications are sent to `client`. The previous function that would have been called is returned. If this function was never defined then the default prioritization function is returned. If the `prioritizer_func()` argument is `NOTIFY_FUNC_NULL` then no client prioritization is done for `client` and the default prioritizer is used.

The calling sequence of a prioritizer function is:

```
Notify_value
prioritizer_func(client, nfd, ibits_ptr, obits_ptr,
            ebits_ptr, nsig, sigbits_ptr, auto_sigbits_ptr,
            event_count_ptr, events, args)
    Notify_client   client;
    fd_set         *ibits_ptr, *obits_ptr, *ebits_ptr;
    int             nfd, nsig, *sigbits_ptr,
                   *auto_sigbits_ptr, *event_count_ptr;
    Notify_event   *events;
    Notify_arg     *args;
#define SIG_BIT(sig)    (1 << ((sig)-1))
```

in which `client` from `notify_set_prioritizer_func()` are passed to `prioritizer_func()`. In addition, all the notifications that the Notifier is planning on sending to `client` are described in the other parameters. This data reflects only data that `client` has expressed interest in by asking for

[18] It is anticipated that this facility will be rarely used by clients and that a client will rely on the ordering provided by the default prioritizer.

notification of these conditions.

`nfd` describes the maximum number of valid bits in the `fd_set` structures[19] pointed to by `ibits_ptr`, `obits_ptr`, and `ebits_ptr`. `ibits_ptr` points to a bit mask of those file descriptors with input pending for `client`; similarly `obits_ptr` points to a bit mask of file descriptors with output completed, and `ebits_ptr` points to a bit mask of file descriptors on which an exception occurred. `nsig` describes the maximum number of valid bits in the arrays pointed to by `sigbits_ptr` and `auto_sigbits_ptr`. `sigbits_ptr` is a bit mask of signals received for which `client` has a condition registered; the `SIG_BIT` macro can be used to access the correct bit. `auto_sigbits_ptr` is a bit mask of signals received that the Notifier is managing on behalf of `client`. `event_count` is the number of events in the array `events`. `events` is an array of pending client events and `args` is the parallel array of event arguments.

The return value is one of `NOTIFY_DONE` or `NOTIFY_IGNORED`. These have their normal meanings:

- `NOTIFY_DONE` — All of the conditions had notifications sent for them. This implies that no further notifications should be sent to `client` this time around the notification loop. Unsent notifications are preserved for consideration the next time around the notification loop.
- `NOTIFY_IGNORED` — A notification was not sent for one or more of the conditions, i.e., some notifications may have been sent, but not all. This implies that another prioritizer should try to send any remaining notifications to `client`.

Dispatching Events

From within a prioritization routine, the following functions are called to cause the specified notifications to be sent:

```
Notify_error
notify_event(client, event, arg)
    Notify_client  client;
    Notify_event   event;
    Notify_arg     arg;

Notify_error
notify_input(client, fd)
    Notify_client  client;
    int            fd;

Notify_error
notify_output(client, fd)
    Notify_client  client;
    int            fd;

Notify_error
```

[19] With the increase past 32 of the maximum number of file descriptors under SunOS Release 4.0, the masks of FD bits are no longer `int`s but a special structure, defined in `<sys/types.h>`.

```
notify_exception(client, fd)
    Notify_client  client;
    int            fd;

Notify_error
notify_itimer(client, which)
    Notify_client  client;
    int            which;

Notify_error
notify_signal(client, signal)
    Notify_client  client;
    int            signal;

Notify_error
notify_wait3(client)
    Notify_client  client;
```

The Notifier won't send any notifications that it wasn't planning on sending anyway, so one can't use these calls to drive clients programmatically. A return value of NOTIFY_OK indicates that client was sent the notification. A return value of NOTIFY_UNKNOWN_CLIENT indicates that client is not recognized by the Notifier and no notification was sent. A return value of NOTIFY_NO_CONDITION indicates that client does not have the requested notification pending and no notification was sent.

A client may chose to replace the default prioritizer. Alternatively, a client's prioritizer may call the default prioritizer after sending only a few notifications. Any notifications not explicitly sent by a client prioritizer will be sent by the default prioritizer (when called), in their normal turn. Once notified, a client will not receive a duplicate notification for the same event.

Signals indicated by bits in sigbits_ptr should call notify_signal(). Signals in auto_sigbits_ptr need special treatment:

- SIGALRM means that notify_itimer() should be called with a which of ITIMER_REAL.
- SIGVTALRM means that notify_itimer() should be called with a which of ITIMER_VIRTUAL.
- SIGCHLD means that notify_wait3() should be called.

Asynchronous signal notifications, destroy notifications and client event notifications that were delivered right when they were posted do not pass through the prioritizer.

Getting the Prioritizer

notify_get_prioritizer_func() returns the current prioritizer of a client.

```
Notify_func
notify_get_prioritizer_func(client)
    Notify_client client;
```

notify_get_prioritizer_func() takes an opaque client handle. The function that will be called before any notifications are sent to `client` is returned. If this function was never defined for `client` then a default function is returned. A return value of `NOTIFY_FUNC_NULL` indicates an error. If `client` is unknown then `notify_errno` is set to `NOTIFY_UNKNOWN_CLIENT`.

8.6. Notifier Control

The following are the Notifier wide (vs single condition) operations.

Starting

Here is the routine for starting the notification loop of the Notifier:

```
Notify_error
notify_start()
```

This is the main control loop. It is usually called from the main routine of your program after all the clients in your program have registered their event handlers with the Notifier.[20] The return values are:

- NOTIFY_OK — Terminated normally by notify_stop() (see below).
- NOTIFY_NO_CONDITION — There are no conditions registered with the Notifier.
- NOTIFY_INVAL — Tried to call notify_start() before returned from original call, i.e., this call is not reentrant.
- NOTIFY_BADF — One of the file descriptors in one of the conditions is not valid.

Stopping

An application may want to break the Notifier out its main loop after the Notifier finishes sending any pending notifications.

```
Notify_error
notify_stop()
```

This causes notify_start() to return. The return values are NOTIFY_OK (will terminate notify_start()) and NOTIFY_NOT_STARTED (notify_start() not entered).

Mass Destruction

The following routine causes the all client destruction functions to be called immediately with status:

```
Notify_error
notify_die(status)
    Destroy_status  status;
```

This causes the all client destruction functions to be called immediately with status as the reason. The return values are NOTIFY_OK or NOTIFY_DESTROY_VETOED; the latter indicates that someone called notify_veto_destroy() and status was DESTROY_CHECKING. It is then the responsibility of the caller of notify_die() to exit the process, if so desired. See the discussion on notify_post_destroy() for more information.

[20] SunView programs usually call window_main_loop() instead of notify_start().

Scheduling

There is the mechanism for controlling the order in which clients are notified. (Controlling the order in which a particular client's notifications are sent to it is done by that client's prioritizer operation; see the *Prioritization* section earlier.)

```
Notify_func
notify_set_scheduler_func(scheduler_func)
    Notify_func  scheduler_func;
```

`notify_set_scheduler_func()` allows you to arrange the order in which clients are called. (Individual clients can control the order in which their event handlers are called by setting up prioritizers.) `notify_set_scheduler_func()` takes a function to call to do the scheduling of clients. The previous function that would have been called is returned. This returned function will (almost always) be important to store and call later because it is most likely the default scheduler.

Replacement of the default scheduler will be done most often by a client that needs to make sure that other clients don't take too much time servicing all of their notifications. For example, if doing "real-time" cursor tracking in a user process, the tracking client wants to schedule itself ahead of other clients whenever there is input pending on the mouse.

The calling sequence of a scheduler function is:

```
Notify_value
scheduler_func(n, clients)
    int           n;
    Notify_client *clients;
```

in which a list of `n` clients, all of which are slated to receive some notification this time around, are passed into `scheduler_func()`. The scheduler scans `clients` and makes calls to `notify_client()` (see below). Clients so notified should have their slots in `clients` set to `NOTIFY_CLIENT_NULL`. The return value from `scheduler_func()` is one of:

- `NOTIFY_DONE` — All of the clients had a chance to send notifications. This implies that no further clients should be scheduled this time around the notification loop. Unsent notifications are preserved for consideration the next time around the notification loop.
- `NOTIFY_IGNORED` — One or more `clients` were scheduled, i.e., some `clients` may have been scheduled, but not all. This implies that another scheduler should try to schedule any clients in `clients` that are not `NOTIFY_CLIENT_NULL`.

Dispatching Clients

The following routine is called from scheduler routines to cause all the pending notifications for `client` to be sent:

```
Notify_error
notify_client(client)
    Notify_client client;
```

The return value is one of `NOTIFY_OK` (client notified) or `NOTIFY_NO_CONDITION` (no conditions for client, perhaps `notify_client()` was already called with this client handle) or

NOTIFY_UNKNOWN_CLIENT (unknown client).

Getting the Scheduler

The following routine returns the function that will be called to do client scheduling:

```
Notify_func
notify_get_scheduler_func()
```

This function is always defined to at least be the default scheduler.

Client Removal

A client can remove itself from the control of the Notifier with notify_remove():

```
Notify_error
notify_remove(client)
    Notify_client client;
```

notify_remove() is a utility to allow easy removal of a client from the Notifier's control. All references to client are purged from the Notifier. This routine is almost always called by the client itself. The return values are NOTIFY_OK (success) and NOTIFY_UNKNOWN_CLIENT (unknown client).

8.7. Error Codes

This section describes the basic error handling scheme used by the Notifier and lists the meaning of each of the possible error codes. Every call to the Notifier returns a value that indicates success or failure. On an error condition, `notify_errno` describes the failure. `notify_errno` is set by the Notifier much like `errno` is set by UNIX system calls, i.e., `notify_errno` is set only when an error is detected during a call to the Notifier and is not reset to `NOTIFY_OK` on a successful call to the Notifier.

```
enum notify_error {
    ... /* Listed below */
};
typedef enum notify_error Notify_error;

extern Notify_error notify_errno;
```

Here is a complete list of error codes:

- `NOTIFY_OK` — The call was completed successfully.
- `NOTIFY_UNKNOWN_CLIENT` — The `client` argument is not known by the Notifier. A `notify_set_*_func` type call need be done in order for the Notifier to recognize a client.
- `NOTIFY_NO_CONDITION` — A call was made to access the state of a condition but the condition is not set with the Notifier for the given client. This can arise when a `notify_get_*_func()` type call was done before the equivalent `notify_set_*_func()` call was done. Also, the Notifier automatically clears some conditions after they have occurred, e.g., when an interval timer expires.
- `NOTIFY_BAD_ITIMER` — The `which` argument to an interval timer routine was not valid.
- `NOTIFY_BAD_SIGNAL` — The `signal` argument to an signal routine was out of range.
- `NOTIFY_NOT_STARTED` — A call to `notify_stop()` was made but the Notifier was never started.
- `NOTIFY_DESTROY_VETOED` — A client refused to be destroyed during a call to `notify_die()` or `notify_post_destroy()` when `status` was `DESTROY_CHECKING`.
- `NOTIFY_INTERNAL_ERROR` — This error code indicates some internal inconsistency in the Notifier itself has been detected.
- `NOTIFY_SRCH` — The `pid` argument to a child process control routine was not valid.
- `NOTIFY_BADF` — The `fd` argument to an input or output routine was not valid.
- `NOTIFY_NOMEM` — The Notifier dynamically allocates memory from the heap. This error code is generated if the allocator could not get any more memory.

- NOTIFY_INVAL — Some argument to a call to the Notifier contained an invalid argument.
- NOTIFY_FUNC_LIMIT — An attempt to set an interposer function has encountered the limit of the number of interposers allowed for a single condition.

The routine notify_perror() acts just as the library call perror(3).

```
notify_perror(str)
     char *str;
```

notify_perror() prints the string str, followed by a colon and followed by a string that describes notify_errno to stderr.

8.8. Restrictions on Asynchronous Calls into the Notifier

The Notifier takes precautions to protect its data against corruption during calls into it while it is calling out to an asynchronous/immediate event handler. The Notifier may issue an asynchronous notification for an asynchronous signal condition, an immediate client event condition or a destroy condition. Most calls from event handlers back into the Notifier are permitted, but there are some restrictions:

- Some calls are not permitted. In particular, they are:

  ```
  notify_start()
  notify_client()
  ```

- Only a certain number of calls into the Notifier are permitted. This restriction is due to how the Notifier handles memory management in a safe way during asynchronous processing. As a guideline, do not do more than five calls of the `notify_set_*_func()`, `notify_interpose_*_func()` or `notify_post_*()` variety during an asynchronous notification.

- The Notifier is not prepared to handle calls into it from signal catching routines that a client has set up with `signal`(3) or `sigvec`(2).

8.9. Issues

Here are some issues surrounding the Notifier:

- The layer over the UNIX signal mechanism is not complete. Signal blocking (`sigblock(2)`) can still safely be done in the flow of control of a client to protect critical portions of code as long as the previous signal mask is restored before returning to the Notifier. Signal pausing (`sigpause(2)`) is essentially done by the Notifier. Signal masking (`sigmask(2)`) can be accomplished via multiple `notify_set_signal_func()` calls. Setting up a process signal stack (`sigstack(2)`) can still be done. Setting the signal catcher mask and on-signal-stack flag (`sigvec(2)`) could be done by reaching around the Notifier but is not supported.
- Not all process resources are multiplexed (e.g., `rlimit(2)`, `setjmp(2)`, `umask(2)`, `setquota(2)`, and `setpriority(2)`), only ones that have to do with flow of control multiplexing. Thus, some level of cooperation and understanding need exist between packages in the single process.
- One can make a case for intercepting `close(2)` and `dup(2)` calls so that the Notifier is not waiting on invalid or incorrect file descriptors if a client forgets to remove its conditions from the Notifier before making these calls.
- One can make a case for intercepting `signal(3)` and `sigvec(2)` calls so that the Notifier doesn't get confused by programs that fail to use the Notifier to manage its signals.
- One can make a case for intercepting `setitimer(2)` calls so that the Notifier doesn't get confused by programs that fail to use the Notifier to manage interval timers.
- One can make a case for intercepting `ioctl(2)` calls so that the Notifier doesn't get fouled up by programs that use `FIONBIO` and `FIOASYNC` instead of the equivalent `fcntl(2)` calls.
- One can make a case for intercepting `readv(2)` and `write(2)` just like `read(2)` and `select(2)` so that a program doesn't tie up the process.
- The Notifier is not a lightweight process mechanism that maintains a stack per thread of control. However, if such a mechanism becomes available then the Notifier will still be valuable for its support of notification-based clients.
- Client events are disjoint from UNIX events. This is done to give complete freedom to clients as to how events are defined. One could imagine certain clients wanting to unify client and UNIX events. This could be done with a layer of software on top of the Notifier. A client could define events as pointers to structures that contain event codes and event specific arguments. The event codes would include the equivalents of UNIX event notifications. The event specific arguments would contain, for example, the file descriptor of an input-pending notification. When an input-pending notification from the the Notifier was sent to a client, the client would turn around and post the equivalent client event notification.
- One could imagine extending the Notifier to provide a record and replay mechanism that would drive an application. However, this is not supported by the current interface.

9

The Selection Service and Library

9

The Selection Service and Library

9.1. Introduction

The Selection Service package provides for flexible communication among window applications. It is concerned with aspects of the selection[s] the user has made, and with the status of the user interface which may affect those selections. It has 3 distinct aspects:

1. A server process maintains a clearinghouse of information about the selection, and the function keys which may affect how a selection is made. This process responds to RPC requests for information from clients. Normally, the RPC accesses will be done only by library routines described below; therefore details of that access do not appear in this manual.

2. A library of client routines is provided to communicate with the clearinghouse process and with each other. These routines allow a client to acquire a selection, or yield it to another application, to determine the current holder of a selection, and send or receive requests concerning a selection's contents and attributes.

3. A minimal set of requests is defined for communicating between applications which have some interest in the selection. This set is deliberately separated from the transport mechanism mentioned under (2) above, and the form of a request is carefully separated from its content. This allows applications to treat the definition of what can be said about the selection as open-ended; anything consenting applications agree to can be passed through the Selection Service.

This chapter is primarily concerned with the transport library, and how to use that protocol to accomplish representative application tasks. The (current) set of generic requests is also presented, and used in illustrations.

The next sections is a fast overview of how the Selection Service works, and a walk-through of the complex protocol followed when selections are transferred between client. This is followed by several discursive sections devoted to particular aspects of using the Selection Service, such as reporting function-key transitions, sending requests, acquiring and releasing selections, replying to requests, and debugging selection applications. Throughout these sections, some procedures and data types are mentioned or described. Full documentation for all of these may be found in the reference section which follows. Finally there are two example programs which show how to use the Selection Service for simple queries (*get_selection*) and how a real client of the selection service works (*seln_demo*).

The remainder of the chapter comprises reference material: descriptions of the public data and procedures of the selection library, a list of the defined common attributes, and the complete source of a program to retrieve the contents of a selection and print it on *stdout*.

9.2. Basic Concepts

When a user makes a selection, it is some application program which interprets the mouse and function-key events and resolves them into a particular selection. The Selection Service is involved only as the processing of function-keys and selections spans application windows. Application programs interact with the package in proportion to the sophistication of their requirements. This section is intended to present the information necessary for any use of the Selection Service, and to indicate what further information in the document pertains to various uses of the package.

The selection library deals with four objects under the general term "*selection.*" Most familiar is the *Primary* selection, which is normally indicated on-screen by inverting ("highlighting") its contents. Selections made while a function key is held down (usually indicated with an underscore) are *Secondary* selections. The selection library treats the *Shelf*[21] (the global buffer which is loaded by **Cut**[22] and **Copy** operations, and which may be retrieved by a **Paste** operation) as a third kind of selection. Finally, the insertion point, or *Caret*, is also treated as a selection, even though it has no contents. These are the four *ranks* the selection library deals with: Caret, Primary, Secondary, and Shelf.

Every selection has a *holder*; this is a piece of code within a process which is responsible for operating on the selection and responding to other applications' requests about it. A selection holder is a *client* of the selection library. Typically, a selection client is something like a subwindow; there may be several selection clients within a single process.

Because the selection library uses RPC as well as the SunView input system, it relies on the SunView Notifier to dispatch events; thus any application using the selection library must be built on the notifier system.

9.3. Fast Overview

The simplest use of the Selection Service is to inquire about a selection held by some other application. Programs which never make a selection will not use the facilities described in the rest of this section. Much of the material remaining before the beginning of reference section is likewise irrelevant to these programs: the sections on *Acquiring and Releasing Selections* and *Callback Procedures* pertain only to clients which make selections.

A program which will actually make selections should be a full-fledged client of the selection library. Such an application calls `seln_create()` during the client's initialization routines; if successful, this returns an opaque client handle which is then passed back in subsequent calls to selection library procedures.

---

[21] The shelf is called the *Clipboard* in user-oriented documentation such as the *SunView 1 Beginner's Guide*, and in the text subwindow menu.

[22] Prior to SunOS Release 4.0, the terms *Delete, Put,* and *Get* were used instead of the "industry-standard" **Cut, Copy,** and **Paste.**

Two arguments to this create-procedure specify client *callback procedures* which may be called to perform processing required by external events. These are the `function_proc()` and the `reply_proc()` described below.

The Selection Service vs. the Selection Library

Note the difference here between the Selection Service program, and the selection library. The former is a separate program, `selection_svc(1)`, the latter is the code in a SunView application which knows how to talk to the Selection Service and its other clients. After a client is successfully created, it may call library routines to:

- inquire of the Selection Service how to get in touch with the holder of a particular rank of selection,
- send a request to such a holder (e.g. to find out the contents of a selection),
- inform the Selection Service of any change in the state of the function keys
- acquire a selection, and later,
- release a selection it has acquired.

The client must be prepared to deal with:

- reports from the Selection Service of function key transitions it may be interested in (these are handled by its `function_proc()`)
- requests from other clients about selections it holds (these are handled by its `reply_proc()`).

Finally, when the client is finished, it calls `seln_destroy()` to clean up its selection library resources.

9.4. Selection Service Protocol: an Example

It's important to remember that the Selection Service is an external process which client programs talk to to find out about the selections held by other clients. To clarify the conceptual model, here's an outline of the communication which takes place when a user makes a secondary selection between two windows.

Secondary Selection Between Two Windows

Try the sequence out yourself while you read the description; The following description assumes that you have a standard set-up (focus follows cursor, adjusting selections does not make them pending-delete, left mouse button is select, middle is extend/adjust, etc.). If you have trouble performing the operation, the *SunView 1 Beginner's Guide* has more to say on selections.

1. Move the cursor into one text subwindow (call it "window A"). Note how the border of the subwindow darkens.
2. Click LEFT in window A to select a character. Window A sets the caret and highlights the primary selection you have just made.
3. Hold down the $\boxed{\text{Paste}}$ key (usually $\boxed{\text{L8}}$).
4. Move the cursor into another text subwindow (call it "window B").
5. Click LEFT and MIDDLE in window B to select some text. Note how it underlines the selection to indicate that it is a secondary selection.

6. Release the [Paste] key. Note how the secondary selection from window B appears at the caret in window A.

Here's what happens internally when this sequence of user actions takes place. Assume window A and B have registered as clients of the Selection Service using `seln_create()`. The initial state is no function keys down, and no one holds any of the selections.

Table 9-1 *Selection Service Scenario*

Client A	*Selection Service*	*Client B*	*Notes*
User moves cursor into window A.			
Get KBD_USE event. Call `seln_acquire(A, SELN_CARET)` to acquire caret. Provide feedback to indicate window has keyboard focus. Done automatically by the window packages.			1
	Note caller as holder of Caret; in this case assume no previous holder, so no notification.		
User presses left mouse button (but has not released yet).			
`seln_inquire( UNSPECIFIED)`			2
	Check state of function keys; since all are up, return primary.		
Provide feedback to indicate selection at indicated point.			
User releases left mouse button.			
`seln_acquire(A, SELN_PRIMARY)`			
	Note caller as holder of primary.		
User holds [Paste] key down (but has not released yet).			
`seln_report_event(A,` *down-event*`)`			
	Note [Paste] key is down.		
User moves cursor into window B.			
		Get `KBD_USE` event.	

Table 9-1 *Selection Service Scenario— Continued*

Client A	*Selection Service*	*Client B*	*Notes*
User presses left mouse button (but has not released yet).			
		`seln_acquire( UNSPECIFIED)`	3
	Check state of function keys; since [Paste] is Down, note caller as secondary holder; return secondary.		
		Provide feedback to indicate secondary selection at indicated point If user has held down the [Control] key, secondary selection may be pending-delete.	
User releases left mouse button.			
		no action required	
User presses middle mouse button (but has not released yet).			
		provide feedback for adjustment of secondary selection	
User releases middle mouse button.			
		no action required	
User releases [Paste] key.			
		`seln_report_event(B,` *up-event*`)`	
	Note that this key-up leaves all function keys up. Figure out which of the four selection holders to notify (the primary selection holder, A).		
		B's call to `seln_report_event()` gets notification in return; B calls `seln_figure-_response()`, which says `IGNORE`	

Table 9-1 *Selection Service Scenario— Continued*

Client A	*Selection Service*	*Client B*	*Notes*
Window A's `function_proc()` is called (it was registered with the Selection Service in `seln_create()` to handle reported function key transitions). The `function_proc()` in turn calls `seln_figure_response()`, which tells it to request the contents of the secondary selection from the secondary selection holder. So `function_proc()` calls `seln_init_request()` to set up a request to a) get the contents of the secondary selection, b) commit any pending-delete, and yield secondary; It then calls `seln_request()` to send this request to B.			
		The request arrives as an RPC call to B's `request-proc()` (likewise registered with the Selection Service in `seln_create()`, but the reply proc handles requests about selections). B provides response to each attribute in the request; in response to the request to Yield(secondary), it calls `seln_done( SELN_SECONDARY)`	
	Note there is no holder of the secondary selection.		
		Do pending delete if necessary: remove selection; cache contents; call `seln_acquire( SELN_SHELF)`	
	Note caller as holder of the Shelf.		
		Return from rpc call.	

Table 9-1 *Selection Service Scenario— Continued*

Client A	*Selection Service*	*Client B*	*Notes*
Consume response, inserting contents; if primary selection were pending delete, then would delete primary (superseding B as holder of Shelf if so).			

Notes on the Above Example

1. To be absolutely correct, clients should determine whether they want the Caret when they get a `KBD_USE` event, and acquire it immediately if so. Code to do this would look something like

Client A	*Selection Service*	*Client B*	*Notes*
User moves cursor into window A.			
Get `KBD_USE` event. Call `seln_get_-function_state()`			
	Return state of function keys (all up).		
`seln_acquire( SELN_CARET)`			

But this would involve waking up a client every time the cursor crossed its window; further, the kernel sends spurious `KBD_USE`s on window-crossings when a function key is down; Hence most clients postpone acquiring the Caret until they know they have input. For most clients, input is usually signalled by a key going down (either a function-key or a regular ASCII-key).

2. Some window types acquire the Caret and primary selection at this point, so they simply call `seln_acquire(`*client_handle*`, SELN_UNSPECIFIED)` when a key goes down.

3. This "Point Down" describes what most windows implement, as opposed to the first theoretical exchange. The issue is whether clients call `seln_inquire(`*client*`, SELN_UNSPECIFIED)`, and then acquire the appropriate selection, or whether they should simply call `seln_acquire(`*client*`, SELN_UNSPECIFIED)`.

9.5. Topics in Selection Processing

Reporting Function-Key Transitions

When an application makes a selection, its rank depends on the state of the function keys. (A secondary selection is one made while a function key is held down.) The application which is affected by a function-key transition may not be the one whose window received that transition. Consequently, this system requires that the service be informed of transitions on those function keys that affect the rank of a selection; the service then provides that state information to any application which inquires.

The keys which affect the rank of a selection are [Copy], [Paste], [Cut], and [Find] (ordinarily these are keys [L6], [L8], [L10], and [L9], respectively). If an application program does not include these events in its input mask, then they will fall through to the root window, and be reported by it. But if the application is reading these events for any reason, then it should also report the event to the service. `seln_report_event()` is the most convenient procedure for this purpose; `seln_inform()` does the work at a lower level.

When the service is told a function key has gone up, it will cause calls to the *function_proc* callback procedures of the holders of each selection. For the client that reports the key-up, this will happen during the call to `seln_report_event`; for other holders, it can happen any time control returns to the client's notifier code. The required processing is detailed under *Callback Procedures* below. Programs which never called `seln_create()` can call `seln_report_event()` without incurring any extra processing — they have no *function_proc* to call.)

Two procedures are provided so that clients may interrogate state of the functions keys as stored by the service: `seln_get_function_state()` takes a `Seln_function` and returns `TRUE` or `FALSE` as the service believes that function key is down or not. `seln_functions_state()` takes a pointer to a `Seln_functions_state` buffer (a bit array which will be all 0 if the service believes all function keys are currently up).

Sending Requests to Selection Holders

Clients of the Selection Service inquire of it who holds a particular rank of selection using `seln_inquire();` they then talk directly to each other through request buffers. A client loads an attribute value list of its requests about the rank of selection into these buffers. Requests include things like "What are the contents of the selection?", "What line does the selection start on?", "Please give up the selection (I need to highlight)!", and so forth.

Inside the selection library, a ***request*** is a buffer (a `Seln_request` structure); the following declarations are relevant to the processing that is done with such a buffer:

```
typedef struct  {
    Seln_result         (*consume)();
    char                 *context;
}       Seln_requester;

typedef struct {
    Seln_replier_data    *replier;
    Seln_requester        requester;
    char                 *addressee;
    Seln_rank             rank;
    Seln_result           status;
    unsigned              buf_size;
    char                  data[SELN_BUFSIZE];
}       Seln_request;
```

```
/* VARARGS */
Seln_request *
seln_ask(holder, <attributes>, ... 0)
    Seln_holder     *holder;
    Attr_union       attribute;

/* VARARGS */
void
seln_init_request(buffer, holder, <attributes>, ... 0)
    Seln_request         *buffer;
    Seln_holder          *holder;
    char                 *attributes;

/* VARARGS */
Seln_result
seln_query(holder, reader, context, <attributes>, ... 0)
    Seln_holder     *holder;
    Seln_result    (*reader)();
    char            *context;
    Attr_union       attribute;


Seln_result
reader(buffer)
    Seln_request         *buffer;
```

The selection library routines pass request buffers from clients to the particular client who holds that rank of selection. That client returns its reply in one or more similar buffers. The library is responsible for maintaining the dialogue, but does not have any particular understanding of the requests or their responses.

`seln_query()` (or `seln_ask()`, for clients unwilling to handle replies of more than one buffer) is used to construct a request and send it to the holder of a selection.

There is a lower-level procedure, `seln_request()` which is used to send a pre-constructed buffer containing a request, and an initializer procedure, `seln_init_request()` which can be used to initialize such a buffer.

The data portion of the request buffer is an attribute-value list, copied from the `<attributes> ...` arguments in a call to `seln_query()` or `seln_ask()`. A similar list is returned in the reply, typically with real values replacing placeholders provided by the requester. (It may take several buffers to hold the whole reply list; this case is discussed below.)

The request mechanism is quite general: an attribute-value pair in the request may indicate some action the holder of the selection is requested to perform — even a program to be executed may be passed, as long as the requester and the holder agree on the interpretation.

The header file `<suntool/selection_attributes.h>` defines a base set of request attributes; a copy is printed near the end of this chapter. The most common request attribute is `SELN_REQ_CONTENTS_ASCII`, which requests the holder of the selection to return its contents as an ASCII string.

Long Request Replies

If the reply to a request is very long (more than about 2000 bytes), more than one buffer will be used to return the response. In this case, `seln_ask()` simply returns a pointer to the first buffer and discards the rest.

NOTE *This means that that the attribute value list in the first buffer may not be properly terminated (but it probably will be).*

`seln_query()` should be used if long replies are to be handled gracefully. Rather than returning a buffer, it repeatedly calls a client procedure to handle each buffer in turn. The client passes a pointer to the procedure to be called in the `reader` argument of the call to `seln_query()` (that address appears in the `consume` element of the `Seln_requester struct`.) Such procedures typically need some context information to be saved across their invocations; this is provided for in the `context` element of the `Seln_requester` structure. This is a 32-bit datum provided for the convenience of the `reader` procedure; it may be filled in with literal data or a pointer to some persistent storage; the value will be available in each call to `reader`, and may be modified at will.

Selection holders are responsible for processing and responding to the attributes of a request in the order they appear in the request buffer. Selection holders may not recognize all the attributes in a request; there is a standard response for this case: In place of the unrecognized attribute (and its value, if any), the replier inserts the attribute `SELN_REQ_UNRECOGNIZED`, followed by the original (unrecognized) attribute. This allows heterogeneous applications to negotiate the level at which they will communicate.

A straightforward example of request processing (including code to handle a long reply) is the *get_selection* program, which appears at the end this chapter.

Acquiring and Releasing Selections

Applications in which a selection can be made must be able to tell the service they now hold the selection, and they must be able to release a selection, either on their own initiative, or because another application has asked to acquire it. `Seln_acquire` is used both to request a current holder of the selection to yield, and then to inform the service that the caller now holds that selection. `seln_yield()` is used to yield the selection on the caller's initiative. A request to yield because another application is becoming the holder is handled like other requests; this is discussed under *Callback Procedures* below.

Callback Procedures: Function-Key Notifications

When you register a client with the Selection Service using `seln_create()`, you must supply a *function_proc()* and a *reply_proc()*. The former is called by the Selection Service to report function-key transitions (which may have occurred in other windows) to the client; the latter is called by the Selection Service when this client holds one of the selections and other clients have requests about it.

The selection library calls the client's *function_proc()* when the Selection Service is informed of a function-key transition which leaves all function keys up. This may happen inside the client's call to `seln_report_event()`, if the reporting client holds a selection; otherwise the call will arrive through the RPC mechanism to the `function_proc()`.

The relevant declarations are:

```
typedef enum     {
    SELN_IGNORE, SELN_REQUEST, SELN_FIND,
    SELN_SHELVE, SELN_DELETE
}           Seln_response;

typedef struct   {
    Seln_function       function;
    Seln_rank           addressee_rank;
    Seln_holder         caret;
    Seln_holder         primary;
    Seln_holder         secondary;
    Seln_holder         shelf;
}       Seln_function_buffer;

Seln_response
seln_figure_response(buffer, holder)
    Seln_function_buffer *buffer;
    Seln_holder          **holder;

void
function_proc(client_data, function)
    char                  *client_data;
    Seln_function_buffer *function;
```

`function_proc()` will be called with a copy of the 32 bits of client data originally given as the third argument to `seln_create()`, and a pointer to a `Seln_function_buffer`. The buffer indicates what function is being invoked, which selection the called program is expected to be handling, and what the Selection Service knows about the holders of all four selection ranks (one of whom is the called program). A client will only be called once, even if it holds more than one selection. (In that case, the buffer's `addressee_rank` will contain the first rank the client holds.)

Responding to Selection Requests

The holders of the selections are responsible for coordinating any data transfer and selection-relinquishing among themselves. The procedure `seln_figure_response()` is provided to assist in this task. It takes a pointer to a function buffer such as the second argument to a `function_proc` callback, and a pointer to a pointer to a `Seln_holder`. It returns an indication of the action which this client should take according to the standard interface. It also changes the `addressee_rank` element of that buffer to be the rank which is affected (the destination of a transfer, the item to be deleted, etc.), and if interaction with another holder is required, it stores a pointer to the appropriate `Seln_holder` element in the buffer into the location addressed by the second argument. Here are the details for each return value:

`SELN_IGNORE` — No action is required of this client. Another client may make a request concerning the selection(s) this client holds.

`SELN_REQUEST` This client is expected to request the contents of another selection and insert them in the location indicated by `buffer->addressee_rank`. The holder of the selection that should be retrieved is identified by `*holder`. If `*holder` points to `buffer->secondary`, the request should include `SELN_REQ_YIELD`; if it points to `buffer->primary` or `buffer->secondary`, the request should include `SELN_REQ_COMMIT_PENDING_DELETE`.

Example: the called program holds the Caret and Primary selection; the [Paste] key went up, and there is no Secondary selection. The return value will be `SELN_REQUEST`, `buffer->addressee_rank` will be `SELN_CARET` and `*holder` will be the address of `buffer->shelf`. The client should request the contents of the shelf from that holder.

`SELN_FIND` This client should do a **Find** (if it can). `Buffer->addressee_rank` will be `SELN_CARET`; if `*holder` is not `NULL`, the target of the search is the indicated selection. If `*holder` points to `buffer->secondary`, the request should include `SELN_REQ_YIELD`.

`SELN_SHELVE` This client should acquire the shelf from `*holder` (if that is not `NULL`), and make the current contents of the primary selection (which it holds) be the contents of the shelf.

`SELN_DELETE` This client should delete the contents of the secondary selection if it exists, or else the primary selection, storing those contents on the shelf. `Buffer->addressee_rank` indicates the selection to be deleted; `*holder` indicates the current holder of the shelf, who should be asked to yield.

`Seln_secondary_exists` and `seln_secondary_made` are predicates which may be of use to an application which is not using `seln_figure_response()`. Each takes a `Seln_function_buffer` and returns `TRUE` or `FALSE`. When the user has made a secondary selection and then cancelled it, `seln_secondary_made` will yield `TRUE` while `seln_secondary_exists` will yield `FALSE`. This indicates the function-key action should be ignored.

Callback Procedures: Replying to Requests

The client's *reply_proc()* callback procedure is called when another application makes a request concerning a selection held by this client. It is invoked once for each attribute in the request, plus once for a terminating attribute supplied by the selection library. The relevant declarations are:

```
typedef struct {
    char                    *client_data;
    Seln_rank                rank;
    char                    *context;
    char                   **request_pointer;
    char                   **response_pointer;
}       Seln_replier_data;

Seln_result
reply_proc(item, context, length)
    caddr_t                  item;
    Seln_replier_data       *context;
    int                      length;
```

`reply_proc()` will be called with each of the attributes of the request in turn. `item` is the attribute to be responded to; `context` points to data which may be needed to compute the response, and `length` is the number of bytes remaining in the buffer for the response. `reply_proc()` should write any appropriate response/value for the given attribute into the buffer indicated in `context->response_pointer`, and return.

The fields of `*context` contain, in order:

- the 32 bits of private client data passed as the last argument to `seln_create()`, returned for the client's convenience;
- the rank of the selection this request is concerned with;
- a holder for 32 more bits of context for the replier's convenience (this will typically hold a pointer to data which allows a client to maintain state while generating a multi-buffer response);
- a pointer to a pointer into the request buffer, just after the current item (so that the replier may read the value of this item if relevant). *This pointer should not be modified by reply_proc.*
- a pointer to a pointer into the response buffer, where the value / response (if any) for this item should be stored. This pointer should be updated to point past the end of the response stored. (Note that items and responses should always be multiples of full-words; thus, this pointer should be left at an address which is 0 mod 4.)

After storing the response to one item, `reply_proc` should return `SELN_SUCCESS` and await the next call. When all attributes in a request have been responded to, `reply_proc` will be called one more time with `item == SELN_REQ_END_REQUEST`, to give it a chance to clean up any internal state associated with the request.

Two attributes which are quite likely to be encountered in the processing of a request due to a function-key event, `SELN_REQ_COMMIT_PENDING_DELETE` and `SELN_REQ_YIELD`, are concerned more with the proper handling of secondary selections (rather than the needs of the requesting application), so they are discussed here.

SELN_REQ_COMMIT_PENDING_DELETE indicates that a secondary selection which was made in pending-delete mode should now be deleted. If the recipient does not hold the secondary selection, or the secondary selection is not in pending-delete mode, the replier should ignore the request, i.e., simply return SELN_SUCCESS and await the next call. SELN_REQ_YIELD, with an argument of SELN_SECONDARY, means the secondary selection should be deselected, if it still exists.

Complications on this basic model will now be addressed in order of increasing complexity.

If the request concerns a selection the application does not currently hold, reply_proc should return SELN_DIDNT_HAVE immediately; it will not be called further for that request.

If the request contains an item the client isn't prepared to deal with, reply_proc should return SELN_UNRECOGNIZED immediately; the selection library will take care of manipulating the response buffer to have the standard unrecognized-format, and call back to reply_proc with the next item in the list.

Finally, a response to a request may be larger than the space remaining in the buffer — or larger than several buffers, for that matter. This situation will never arise on items whose response is a single word — the selection library ensures there is room for at least one 4-byte response in the buffer before calling reply_proc.

If a response is too big for the current buffer, the replier should store as much as fits in length bytes, save sufficient information to pick up where it left off in some persistent location, store the address of that information in context->context, and return SELN_CONTINUED. Note that the replier's context information should not be local to reply_proc, since that procedure will exit and be called again before the information is needed.

The selection library will ship the filled buffer to the requester, and prepare a new one for the continuation of the response. It will then call reply_proc again, with the same item and context, and length indicating the space available in the new buffer. reply_proc() should be able to determine from context->context that it has already started this response, and where to continue from. It continues by storing as much of the remainder of the response as fits into the buffer, updating context->response_pointer (and its own context information), and again returning SELN_CONTINUED if the response is not completed. When the end of the response has been stored, including any terminator if one is required, the private context information may be freed, and reply_proc should return SELN_SUCCESS.

The next call to reply_proc will be to respond to the next item in the request if there is one, or else to SELN_REQ_END_REQUEST.

9.6. Debugging and Administrative Facilities

A number of aids to debugging have been included in the system for applications which use the Selection Service. In addition to providing information on how to access holders of selections and maintaining the state of the user-interface keys, the service will respond to requests to display traces of these requests, and to dump its internal state on an output stream. `Seln_debug` is used to turn service tracing on or off; `seln_dump` instructs the service to dump all or part of its state on *stderr*.

A number of library procedures provide formatted dumps of Selection Service structs and enumerated types. These can be found below as `seln_dump_*`.

In debugging an application which uses the Selection Service, it may be convenient to use a separate version of the service whose state is affected only by the application under test. This is done by starting the service with the **-d** flag; that is, by entering **`/usr/bin/selection_svc -d &`** to a shell. The resulting service will use a different RPC program number from the standard version, but be otherwise identical. The two versions of the service may be running at the same time, each responding to its own clients. A client may elect (via `seln_use_test_service()`) to talk to the test service. Thus, it is easy to arrange to have an application under development talking to its own service, while running under a debugger which is talking to a standard service — this keeps the debugger, editors, etc. from interfering with the state maintained by the test service.

The Selection Service depends heavily on remote procedure calls, using Sun's RPC library. It is always possible that the called program has terminated or is not responding for some other reason; this is often detected by a timeout. The standard timeout at this writing is 10 seconds; this is a compromise between allowing for legitimate delays on loaded systems, and minimizing lockups when the called program really won't respond. The delay may be adjusted by a call to `seln_use_timeout`.

9.7. Other Suggestions

Always call `seln_figure_response()` to determine what to do with a request.

Use `seln_report_event()` instead of `seln_inform()` to report events you aren't interested in. Note that the canvas subwindow by default does *not* report SunView function-key transitions; it relies on the events falling through to the frame or root window, which does report the transitions.

If you expect to [Paste] the selection, you must acquire the Caret beforehand.

If You Die

If your application dies, if it holds the shelf then you should save its contents by writing them to a file and calling `seln_hold_file()`. You should also yield the primary and secondary selections.

9.8. Reference Section

The reference material which follows presents first the header files and the public data definitions they contain; then it lists each public procedure in the selection library (in alphabetical order) with its formal parameter declarations, return value, and a brief description of its effect.

Required Header Files

```
#include <sunwindow/attr.h>
#include <suntool/selection_svc.h>
#include <suntool/selection_attributes.h>
```

Enumerated Types

```
typedef enum    {
    SELN_FAILED,      SELN_SUCCESS,       /* basic uses     */
    SELN_NON_EXIST,   SELN_DIDNT_HAVE,    /* special cases  */
    SELN_WRONG_RANK,  SELN_CONTINUED,
    SELN_CANCEL,      SELN_UNRECOGNIZED
}           Seln_result;

typedef enum    {
    SELN_UNKNOWN, SELN_CARET, SELN_PRIMARY,
    SELN_SECONDARY, SELN_SHELF, SELN_UNSPECIFIED
}           Seln_rank;

typedef enum    {
        SELN_FN_ERROR,
    SELN_FN_STOP,  SELN_FN_AGAIN,
    SELN_FN_PROPS, SELN_FN_UNDO,
    SELN_FN_FRONT, SELN_FN_PUT,
    SELN_FN_OPEN,  SELN_FN_GET,
    SELN_FN_FIND,  SELN_FN_DELETE
}           Seln_function;

typedef enum    {
    SELN_NONE, SELN_EXISTS, SELN_FILE
}           Seln_state;

typedef enum    {
    SELN_IGNORE, SELN_REQUEST, SELN_FIND,
    SELN_SHELVE, SELN_DELETE
}       Seln_response;
```

Other Data Definitions

```
typedef char *Seln_client;

typedef struct  {
    Seln_rank           rank;
    Seln_state          state;
    Seln_access         access;
}           Seln_holder;
```

`Seln_holder`

```
typedef struct  {
    Seln_holder         caret;
    Seln_holder         primary;
    Seln_holder         secondary;
    Seln_holder         shelf;
}       Seln_holders_all;

typedef struct  {
    Seln_function       function;
    Seln_rank           addressee_rank;
    Seln_holder         caret;
    Seln_holder         primary;
    Seln_holder         secondary;
    Seln_holder         shelf;
}           Seln_function_buffer;

typedef struct  {
    char                *client_data;
    Seln_rank            rank;
    char                *context;
    char               **request_pointer;
    char               **response_pointer;

}       Seln_replier_data;

typedef struct  {
    Seln_result        (*consume)();
    char                *context;
}       Seln_requester;

#define SELN_BUFSIZE                                    \
        (1500 - sizeof(Seln_replier_data *)             \
              - sizeof(Seln_requester)                  \
              - sizeof(char *)                          \
              - sizeof(Seln_rank)                       \
              - sizeof(Seln_result)                     \
              - sizeof(unsigned))

typedef struct {
    Seln_replier_data   *replier;
    Seln_requester       requester;
    char                *addressee;
    Seln_rank            rank;
    Seln_result          status;
    unsigned            buf_size;
    char                data[SELN_BUFSIZE];
}       Seln_request;
```

Procedure Declarations

```
Seln_rank
seln_acquire(client, asked)
    Seln_client  client;
    Seln_rank    asked;
```

`client` is the opaque handle returned from `seln_create()`; the client uses this call to become the new holder of the selection of rank `asked`. `asked` should be one of `SELN_CARET`, `SELN_PRIMARY`, `SELN_SECONDARY`, or`SELN_SHELF`, `SELN_UNSPECIFIED`. If successful, the rank actually acquired is returned.

If `asked` is `SELN_UNSPECIFIED`, the client indicates it wants whichever of the primary or secondary selections is appropriate given the current state of the function keys; the one acquired can be determined from the return value.

```
/* VARARGS */
Seln_request *
seln_ask(holder, <attributes>, ... 0)
    Seln_holder     *holder;
    Attr_union       attribute;
```

`seln_ask()` looks and acts very much like `seln_query()`; the only difference is that it does not use a callback proc, and so cannot handle replies that require more than a single buffer. If it receives such a long reply, it returns the first buffer, and discards all that follow. The return value is a pointer to a static buffer; in case of error, this will be a valid pointer to a null buffer (*buffer*–>`status = SELN_FAILED`). `seln_ask()` is provided as a slightly simpler interface for applications that refuse to process long replies.

```
void
seln_clear_functions()
```

The Selection Service is told to forget about any function keys it thinks are down, resetting its state to all-up. If it knows of a current secondary selection, the service will tell its holder to yield.

```
Seln_client
seln_create(function_proc, reply_proc, client_data)
    void    (*function_proc) ();
    void    (*reply_proc) ();
    caddr_t       client_data);
```

The selection library is initialized for this client. If this is the first client in its process, an RPC socket is established and the notifier set to recognize incoming calls. Client_data is a 32-bit opaque client value which the Selection Service will pass back in callback procs, as described above. The first two arguments are addresses of client procedures which will be called from the selection library when client processing is required. These occasions are:

- when the service sees a function-key transition which may interest this client, and
- when another process wishes to make a request concerning the selection this client holds,

Details of these procedures are described under *Callback Procs*, above.

```
/* VARARGS */
Seln_result
seln_debug(<attributes> ... 0)
    Seln_attribute  attribute;
```

A debugging routine which requests the service to turn tracing on or off for specified calls. Each attribute identifies a particular call; its value should be 1 if that call is to be traced, 0 if tracing is to be stopped. The attributes are listed with other request-attributes in the first appendix. Tracing is initially off for all calls. When tracing is on, the Selection Service process prints a message on its `stderr` (typically the console) when it enters and leaves the indicated routine.

```
void
seln_destroy(client)
    Seln_client  client;
```

A client created by `seln_create` is destroyed: any selection it may hold is released and various pieces of data associated with the selection mechanism are freed. If this is the last client in this process using the Selection Service the RPC socket is closed and its notification removed.

```
Seln_result
seln_done(client, rank)
    Seln_client  client;
    Seln_rank    rank;
```

Client indicates it is no longer the holder of the selection of the indicated rank. The only cause of failure is absence of the Selection Service. It is not necessary for a client to call this procedure when it has been asked by another client to yield a selection; the service will be completely updated by the acquiring client.

```
void
seln_dump_function_buffer(stream, ptr)
    FILE                 *stream;
    Seln_function_buffer *ptr;
```

A debugging routine which prints a formatted display of a `Seln_function_buffer` struct on the indicated stream.

```
void
seln_dump_function_key(stream, ptr)
    FILE             *stream;
    Seln_function    *ptr;
```

A debugging routine which prints a formatted display of a Seln_function_key transition on the indicated stream.

```
void
seln_dump_holder(stream, ptr)
    FILE             *stream;
    Seln_holder      *ptr;
```

A debugging routine which prints a formatted display of a Seln_holder struct on the indicated stream.

```
void
seln_dump_rank(stream, ptr)
    FILE             *stream;
    Seln_rank        *ptr;
```

A debugging routine which prints a formatted display of a Seln_rank value on the indicated stream.

```
void
seln_dump_response(stream, ptr)
    FILE             *stream;
    Seln_response    *ptr;
```

A debugging routine which prints a formatted display of a Seln_response value on the indicated stream.

```
void
seln_dump_result(stream, ptr)
    FILE             *stream;
    Seln_result      *ptr;
```

A debugging routine which prints a formatted display of a Seln_result value on the indicated stream.

```
void
seln_dump_service(rank)
    Seln_rank      rank;
```

A debugging routine which requests the service to print a formatted display of its internal state on its standard error stream. Rank determines which selection holder is to be dumped; if it is SELN_UNSPECIFIED, all four are printed. In any case, the dump concludes with the state of the function keys and the set of open file descriptors in the service.

```
void
seln_dump_state(stream, ptr)
    FILE            *stream;
    Seln_state      *ptr;
```

A debugging routine which prints a formatted display of a `Seln_state` value on the indicated stream.

```
Seln_response
seln_figure_response(buffer, holder)
    Seln_function_buffer  *buffer;
    Seln_holder           **holder;
```

A procedure to determine the correct response according to the standard user interface when `seln_inform()` returns `*buffer`, or the client's *function_proc* is called with it. The field `addressee_rank` will be modified to indicate the selection which should be affected by this client; `holder` will be set to point to the element of `*buffer` which should be contacted in the ensuing action, and the return value indicates what that action should be.

```
Seln_result
seln_functions_state(buffer)
   Seln_functions_state *buffer;
```

The service is requested to dump the state it is maintaining for the function keys into the bit array addressed by `buffer`. At present, the only commitment made to representation in the buffer is that some bit will be on (== 1) for each function key which is currently down. Thus this call can be used to determine whether any function keys are down, but not which. SELN_SUCCESS is returned unless the service could not be contacted.

```
int
seln_get_function_state(which)
    Seln_function    which;
```

A predicate which returns `TRUE` when the service's state shows the function key indicated by `which` is down and `FALSE` otherwise.

```
Seln_result
seln_hold_file(rank, path)
    Seln_rank        rank;
    char            *path;
```

The Selection Service is requested to act as the holder of the specified `rank`, whose ASCII contents have been written to the file indicated by `path`. This allows a selection to persist longer than the application which made it can maintain it. Most commonly, this will be done by a process which holds the shelf when it is about to terminate.

```
int
seln_holder_same_client(holder, client_data)
    Seln_holder      *holder;
    char             *client_data;
```

A predicate which returns `TRUE` if the holder referred to by `holder` is the same selection client as the one which provided `client_data` as its last argument to `seln_create`.

```
int
seln_holder_same_process(holder)
    Seln_holder *holder;
```

A predicate which returns `TRUE` if the `holder` is a selection client in the same process as the caller. (This procedure is used to short-circuit RPC calls with direct calls in the same address space.)

```
Seln_function_buffer
seln_inform(client, which, down)
    Seln_client       client;
    Seln_function     which;
    int               down;
```

This is the low-level, policy-independent procedure for informing the Selection Service that a function key has changed state. Most clients will prefer to use the higher-level procedure `seln_report_event()`, which handles much of the standard interpretation required.

`Client` is the client handle returned from `seln_create()`; it may be 0 if the client guarantees it will never need to respond to the function transition. `Which` is an element of the `Seln_function` enum defined in `<suntool/selection_svc.h>`; `down` is a boolean which is `TRUE` if the key went down.

On an up-event which leaves all keys up, the service informs the holders of all selections of the transition, and what other parties are affected. If the caller of `seln_inform()` is one of these holders, its notification is returned as the value of the function; other notifications go out as a call on the client's *function_proc* callback procedure (described above under *Callback Procedures*). Only one notification is sent to any single client. If the caller does not hold any selection, or if the transition was not an up which left all function keys up, the return value will be a null `Seln_function_buffer`; `buffer.rank` will be `SELN_UNKNOWN`.

```
/* VARARGS */
void
seln_init_request(buffer, holder, <attributes>, ... 0)
    Seln_request          *buffer;
    Seln_holder           *holder;
    char                  *attributes;
```

This procedure is used to initialize a buffer before calling `seln_request`. (It is also called internally by `seln_ask` and `seln_query`.) It takes a pointer to a request buffer, a pointer to a struct referring to the selection holder to which the request is to be addressed, and a list of attributes which constitute the request to be sent. The attributes are copied into `buffer->data`, and the corresponding size is stored into `buffer->buf_size`. Both elements of `requester_data` are zeroed; if the caller wants to handle long requests, a *consumer-proc* and *context* pointers must be entered in these elements after `seln_init_request` returns.

```
Seln_holder
seln_inquire(rank)
    Seln_rank  rank;
```

A `Seln_holder` struct is returned, containing information which enables the holder of the indicated selection to be contacted. If the `rank` argument is `SELN_UNSPECIFIED`, the Selection Service will return access information for either the primary or secondary selection holder, as warranted by the state of the function keys it knows about; the `rank` element in the returned struct will indicate which is being returned.

This procedure may be called without having called `seln_create()` first. If no contact between this process and the service has been established yet, it will be set up, and then the call will proceed as usual. In this case, return of a null `holder` struct may indicate inaccessibility of the server.

```
Seln_holders_all
seln_inquire_all()
```

A `Seln_holders_all` struct is returned from the Selection Service; it consists of a `Seln_holder` struct for each of the four ranks.

```
Seln_result
reader(buffer)
    Seln_request          *buffer;

/* VARARGS */
Seln_result
seln_query(holder, reader, context, <attributes>, ... 0)
    Seln_holder      *holder;
    Seln_result     (*reader)();
    char             *context;
    Attr_union        attribute;
```

A request is transmitted to the selection holder indicated by the `holder` argument. `Consume` and `context` are used to interpret the response, and are described in the next paragraph. The remainder of the arguments to `seln_query` constitute an attribute value list which is the request. (The last argument should be a 0 to terminate the list.)

The procedure pointed to by `consume` will be called repeatedly with a pointer to each buffer of the reply. The value of the `context` argument will be available in `buffer->requester_data.context` for each buffer. This item is not used by the selection library; it is provided for the convenience of the client. When the reply has been completely processed (or when the consume proc returns something other than `SELN_SUCCESS`), `seln_query` returns.

```
void
seln_report_event(client, event)
    Seln_client_node     *client;
    struct inputevent    *event;

#define SELN_REPORT(event) seln_report_event(0, event)
```

This is a high-level procedure for informing the selection service of a function key transition which may affect the selection. It incorporates some of the policy of the standard user interface, and provides a more convenient interface to `seln_inform()`.

`Client` is the client handle returned from `seln_create`; it may be 0 if the client guarantees it will not need to respond to the function transition. `Event` is a pointer to the `struct inputevent` which reports the transition. `seln_report_event()` generates a corresponding call to `seln_inform()`, and, if the returned struct is not null, passes it to the client's *function_proc* callback procedure (described above under *Callback Procedures*).

`SELN_REPORT` is a macro which takes just an input-event pointer, and calls `seln_report_event` with 0 as a first argument.

```
Seln_result
seln_request(holder, buffer)
    Seln_holder      *holder;
    Seln_request     *buffer;
```

Seln_request is the low-level (policy-independent) mechanism for retrieving information about a selection from the process which holds it. Most clients will access it only indirectly, through seln_ask or seln_query.

Seln_request takes a pointer to a holder (as returned by seln_inquire), and a request constructed in *buffer. The request is transmitted to the indicated selection holder, and the buffer rewritten with its response. Failures in the RPC mechanism will cause a SELN_FAILED return; if the process of the addressed holder is no longer active, the return value will be SELN_NON_EXIST.

Clients which call seln_request directly will find it most convenient to initialize the buffer by a call to seln_init_request.

Request attributes which are not recognized by the selection holder will be returned as the value of the attribute SELN_UNRECOGNIZED. Responses should be provided in the order requests were encountered.

```
int
seln_same_holder(holder1, holder2)
    Seln_holder   *holder1, *holder2;
```

This predicate returns TRUE if holder1 and holder2 refer to the same selection client.

```
int
seln_secondary_exists(buffer)
    Seln_function_buffer    *buffer;
```

This predicate returns TRUE if the function buffer indicates that a secondary selection existed at the time the function key went up.

```
int
seln_secondary_made(buffer)
    Seln_function_buffer    *buffer;
```

This predicate returns TRUE if the function buffer indicates that a secondary selection was made some time since the function key went down (although it may have been cancelled before the key went up).

```
void
seln_use_test_service()
```

The application is set to communicate with a test version of the Selection Service, rather than the standard production version. This call should be made before any selection client is created; this normally means before subwindows in the application process are created.

```
void
seln_use_timeout(seconds)
    int     seconds;
```

The default timeout on subsequent RPC calls from this process is changed to be `seconds` long.

```
void
seln_yield_all()
```

This procedure inquires the holders of all selection, and for each which is held by a client in the calling process, sends a yield request to that client and a Done to the service. It should be called by applications which are about to exit, or to undertake lengthy computations during which they will be unable to respond to requests concerning selections they hold.

9.9. Common Request Attributes

The following is an annotated listing of `<suntool/selection_attributes.h>`.

```
/*  @(#)selection_attributes.h 1.10 85/09/05    */

#ifndef suntool_selection_attributes_DEFINED
#define suntool_selection_attributes_DEFINED

/*
 *  Copyright (c) 1985 by Sun Microsystems, Inc.
 */

#include <sunwindow/attr.h>
/*
 *  Common requests a client may send to a selection-holder
 */
#define ATTR_PKG_SELECTION  ATTR_PKG_SELN_BASE

#define SELN_ATTR(type, n)  ATTR(ATTR_PKG_SELECTION, type, n)

#define SELN_ATTR_LIST(list_type, type, n)  \
    ATTR(ATTR_PKG_SELECTION, ATTR_LIST_INLINE(list_type, type), n)
```

```
/*
 *      Attributes of selections
 */

typedef enum    {

    /*  Simple attributes
     */
    SELN_REQ_BYTESIZE           = SELN_ATTR(ATTR_INT,                     1),
        /*  value is an int giving the number of bytes in the
         *      selection's ascii contents                              */
    SELN_REQ_CONTENTS_ASCII     = SELN_ATTR_LIST(ATTR_NULL, ATTR_CHAR,    2),
        /*  value is a null-terminated list of 4-byte words containing
         *      the selection's ascii contents.  The last word containing
         *      a character of the selection should be followed by a
         *      terminator word whose value is 0.  If the last word of
         *      contents is not full, it should be padded out with NULs */

    SELN_REQ_CONTENTS_PIECES    = SELN_ATTR_LIST(ATTR_NULL, ATTR_CHAR,    3),
        /*  value is a null-terminated list of 4-byte words containing
         *      the selection's contents described in the textsw's
         *      piece-table format.                                     */
    SELN_REQ_FIRST              = SELN_ATTR(ATTR_INT,                     4),
        /*  value is an int giving the number of bytes which precede
         *      the first byte of the selection.                        */
    SELN_REQ_FIRST_UNIT         = SELN_ATTR(ATTR_INT,                     5),
        /*  value is an int giving the number of units of the selection's
         *      current level (line, paragraph, etc.) which precede the
         *      first unit of the selection.                            */
    SELN_REQ_LAST               = SELN_ATTR(ATTR_INT,                     6),
        /*  value is an int giving the byte index of the last byte
         *      of the selection.                                       */
    SELN_REQ_LAST_UNIT          = SELN_ATTR(ATTR_INT,                     7),
        /*  value is an int giving the unit index of the last unit
         *      of the selection at its current level.                  */
    SELN_REQ_LEVEL              = SELN_ATTR(ATTR_INT,                     8),
        /*  value is an int giving the current level of the selection
         *       (See below for #defines of the most useful levels.)    */
    SELN_REQ_FILE_NAME          = SELN_ATTR_LIST(ATTR_NULL, ATTR_CHAR,    9),
        /*  value is a null-terminated list of 4-byte words containing
         *      the name of the file which holds the selection (when the
         *      Selection Service has been asked to hold a selection).
         *      The string is represented exactly like ascii contents.  */
```

```
/* Simple commands (no parameters)
 */
SELN_REQ_COMMIT_PENDING_DELETE
                            = SELN_ATTR(ATTR_NO_VALUE,                65),
    /*  There is no value.  The replier is instructed to delete any
     *      secondary selection made in pending delete mode.         */
SELN_REQ_DELETE             = SELN_ATTR(ATTR_NO_VALUE,                66),
    /*  There is no value.  The replier is instructed to delete the
     *      selection referred to in this request.                   */
SELN_REQ_RESTORE            = SELN_ATTR(ATTR_NO_VALUE,                67),
    /*  There is no value.  The replier is instructed to restore the
     *      selection referred to in this request, if it has maintained
     *      sufficient information to do so.                         */

/* Other commands
 */
SELN_REQ_YIELD              = SELN_ATTR(ATTR_ENUM,                    97),
    /*  The value in the request is not meaningful; in the response,
     *      the value is a Seln_result which is the replier's
     *      return code.  The replier is requested to yield the
     *      selection referred to in this request.  SELN_SUCCESS,
     *      SELN_DIDNT_HAVE, and SELN_WRONG_RANK are legitimate
     *      responses (the latter comes from a holder asked to
     *      yield the primary selection when it knows a function-key
     *      is down).                                                */
SELN_REQ_FAKE_LEVEL         = SELN_ATTR(ATTR_INT,                     98),
    /*  value is an int giving a level to which the selection
     *      should be expanded before processing the remainder of
     *      this request.  The original level should be maintained
     *      on the display, however, and restored as the true level
     *      on completion of the request                             */
SELN_REQ_SET_LEVEL          = SELN_ATTR(ATTR_INT,                     99),
    /*  value is an int giving a level to which the selection
     *      should be set.  This request should affect the true level */

/* Service debugging commands
 */
SELN_TRACE_ACQUIRE          = SELN_ATTR(ATTR_BOOLEAN,                193),
SELN_TRACE_DONE             = SELN_ATTR(ATTR_BOOLEAN,                194),
SELN_TRACE_HOLD_FILE        = SELN_ATTR(ATTR_BOOLEAN,                195),
SELN_TRACE_INFORM           = SELN_ATTR(ATTR_BOOLEAN,                196),
SELN_TRACE_INQUIRE          = SELN_ATTR(ATTR_BOOLEAN,                197),
SELN_TRACE_YIELD            = SELN_ATTR(ATTR_BOOLEAN,                198),
SELN_TRACE_STOP             = SELN_ATTR(ATTR_BOOLEAN,                199),
    /*  value is a boolean (TRUE / FALSE) indicating whether calls
     *      to that procedure in the service should be traced.
     *      TRACE_INQUIRE also controls tracing on seln_inquire_all(). */
SELN_TRACE_DUMP             = SELN_ATTR(ATTR_ENUM,                   200),
    /*  value is a Seln_rank, indicating which selection holder
     *      should be dumped; SELN_UNSPECIFIED indicates all holders. */
```

```
    /*  Close bracket so replier can terminate commands
     *  like FAKE_LEVEL which have scope
     */
    SELN_REQ_END_REQUEST        = SELN_ATTR(ATTR_NO_VALUE,              253),

    /*  Error returnd for failed or unrecognized requests
     */
    SELN_REQ_UNKNOWN            = SELN_ATTR(ATTR_INT,                   254),
    SELN_REQ_FAILED             = SELN_ATTR(ATTR_INT,                   255)

}       Seln_attribute;

/* Meta-levels available for use with SELN_REQ_FAKE/SET_LEVEL.
 *      SELN_LEVEL_LINE is "text line bounded by newline characters,
 *                          including only the terminating newline"
 */
typedef enum {
    SELN_LEVEL_FIRST    = 0x40000001,
    SELN_LEVEL_LINE     = 0x40000101,
    SELN_LEVEL_ALL      = 0x40008001,
    SELN_LEVEL_NEXT     = 0x4000F001,
    SELN_LEVEL_PREVIOUS = 0x4000F002
}       Seln_level;
#endif
```

9.10. Two Program Examples

There are several programs in the *SunView 1 Programmer's Guide* that do a `seln_ask()` for the primary selection. Here are two sample programs that manipulate the selection in more complex ways.

get_selection Code

The following code is from the program *get_selection*, which is part of the release. This program copies the contents of the desired SunView selection to stdout. For more information, consult the `get_selection(1)` man page.

```
#ifndef lint
static char   sccsid[] = "@(#)get_selection.c 10.5 86/05/14";
#endif

/*
 * Copyright (c) 1986 by Sun Microsystems, Inc.
 */

#include <stdio.h>
#include <sys/types.h>
#include <suntool/selection_svc.h>
#include <suntool/selection_attributes.h>

static Seln_result     read_proc();

static int        data_read = 0;

static void                quit();

#ifdef STANDALONE
main(argc, argv)
#else
get_selection_main(argc, argv)
#endif STANDALONE
    int                    argc;
    char                 **argv;
{
    Seln_client            client;
    Seln_holder            holder;
    Seln_rank              rank = SELN_PRIMARY;
    char                   context = 0;
    int                    debugging = FALSE;

    while (--argc) {
        /*  command-line args control rank of desired selection,        */
        /*  use of a debugging service, and rpc timeout                  */
        argv++;
        switch (**argv) {
          case '1':
            rank = SELN_PRIMARY;
            break;
          case '2':
            rank = SELN_SECONDARY;
```

```
          break;
        case '3':
          rank = SELN_SHELF;
          break;
        case 'D':
          seln_use_test_service();
          break;
        case 't':
        case 'T':
          seln_use_timeout(atoi(++argv));
          --argc;
          break;
        default:
          quit("Usage: get_selection [D] [t seconds] [1 | 2 |3]\n");
      }
    }
    /*  find holder of desired selection */
    holder = seln_inquire(rank);
    if (holder.state == SELN_NONE) {
        quit("Selection non-existent, or selection-service failure\n");
    }
    /*  ask for contents, and let callback proc print them */

    (void) seln_query(&holder, read_proc, &context,
                      SELN_REQ_CONTENTS_ASCII, 0, 0);

    if (data_read)
        exit(0);
    else
        exit(1);
}

static void
quit(str)
    char                  *str;
{
    fprintf(stderr, str);
    exit(1);
}

/*
 *      Procedure called with each buffer of data returned in response
 *      to request transmitted by seln_query.
 */
static Seln_result
read_proc(buffer)
    Seln_request          *buffer;
{
    char                  *reply;

    /*  on first buffer, we have to skip the request attribute,
     *  and then make sure we don't repeat on subsequent buffers
     */
```

```
    if (*buffer->requester.context == 0) {
        if (buffer == (Seln_request *) NULL ||
            *((Seln_attribute *) buffer->data) != SELN_REQ_CONTENTS_ASCII) {
            quit("Selection holder error -- unrecognized request\n");
        }
        reply = buffer->data + sizeof (Seln_attribute);
        *buffer->requester.context = 1;
    } else {
        reply = buffer->data;
    }
    fputs(reply, stdout);
    fflush(stdout);
    data_read = 1;
    return SELN_SUCCESS;
}
```

seln_demo

The following program, *seln_demo* gets the selection, but it also sets the selection and responds to appropriate queries about it. It isn't an entirely realistic program, since it doesn't provide selection feedback or use function keys.

It displays a panel with several choices and buttons and a text item. You choose the rank of the selection you wish to set or retrieve first. If you are setting the selection, you may also choose whether you want to literally set the selection or provide the name of a file which contains the selection. Then either type in the selection and push the **Set** button, or just push the **Get** button to retrieve the current selection of the type you chose.

The code has three logical sections: the procedures to create and service the panel, the code to set a selection, and the code to get a selection. The routines to set and get the selection are complicated because they are written to allow arbitrary length selections. Try selecting a 3000 byte piece of text; although you can only see 10 characters of it in the text panel item, the entire selection can be retrieved and/or set.

Large Selections

In order to handle large selections, the selection service breaks them into smaller chunks of about 2000 bytes called buffers. The routines you write must be able to handle a buffer and save enough information so that when they are called again with the next buffer, they can pick up where they left off. *seln_demo* uses the context fields provided in the Selection Service data structures to accomplish this.

```
#ifndef lint
static char sccsid[] = "@(#)seln_demo.c 1.5 88/03/14 Copyr 1986 Sun Micro";
#endif
/*
 * seln_demo.c
 *
 * demonstrate how to use the selection service library
 */

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/seln.h>

static Frame frame;
static Panel panel;

int err = 0;

char *malloc();

/*
 * definitions for the panel
 */

static Panel_item text_item, type_item, source_item, mesg_item;
static Panel_item set_item[3], get_item[3];
static void set_button_proc(), get_button_proc(), change_label_proc();

#define PRIMARY_CHOICE          0       /* get/set the primary selection */
#define SECONDARY_CHOICE        1       /* get/set the secondary selection */
#define SHELF_CHOICE            2       /* get/set the shelf */

#define ITEM_CHOICE             0       /* use the text item literally as the
                                           selection */
#define FROMFILE_CHOICE         1       /* use the text item as the name of a
                                           file which contains the selection */

int selection_type = PRIMARY_CHOICE;
int selection_source = ITEM_CHOICE;

char *text_labels[3][2] = {
        {
                "New primary selection:",
                "File containing new primary selection:"
        },
        {
                "New secondary selection:",
                "File containing new secondary selection:"
        },
        {
                "New shelf:",
                "File containing new shelf:"
```

```
        }
};

char *mesg_labels[3][2] = {
        {
                "Type in a selection and hit the Set Selection button",
                "Type in a filename and hit the Set Selection button"
        },
        {
                "Type in a selection and hit the Set Secondary button",
                "Type in a filename and hit the Set Secondary button"
        },
        {
                "Type in a selection and hit the Set Shelf button",
                "Type in a filename and hit the Set Shelf button"
        }
};

Seln_rank type_to_rank[3] = { SELN_PRIMARY, SELN_SECONDARY, SELN_SHELF };

/*
 * definitions for selection service handlers
 */

static Seln_client s_client;    /* selection client handle */

#define FIRST_BUFFER    0
#define NOT_FIRST_BUFFER        1

char *selection_bufs[3];        /* contents of each of the three selections;
                                   they are set only when the user hits a set
                                   or a get button */

int func_key_proc();
Seln_result reply_proc();
Seln_result read_proc();
```

```
/**********************************************************************/
/* main routine                                                       */
/**********************************************************************/

main(argc, argv)
int argc;
char **argv;
{
        /* create frame first */

        frame = window_create(NULL, FRAME,
                        FRAME_ARGS,     argc, argv,
                        WIN_ERROR_MSG, "Cannot create frame",
                        FRAME_LABEL, "seln_demo",
                        0);

        /* create selection service client before creating subwindows
           (since the panel package also uses selections) */

        s_client = seln_create(func_key_proc, reply_proc, (char *)0);
        if (s_client == NULL) {
                fprintf(stderr, "seln_demo: seln_create failed!\n");
                exit(1);
        }

        /* now create the panel */

        panel = window_create(frame, PANEL,
                        WIN_ERROR_MSG, "Cannot create panel",
                        0);

        init_panel(panel);

        window_fit_height(panel);

        window_fit_height(frame);

        window_main_loop(frame);

        /* yield any selections we have and terminate connection with the
           selection service */

        seln_destroy(s_client);
}
```

```
/**********************************************************************/
/* routines involving setting a selection                             */
/**********************************************************************/

/*
 * acquire the selection type specified by the current panel choices;
 * this will enable requests from other clients which want to get
 * the selection's value, which is specified by the source_item and text_item
 */

static void
set_button_proc(/* args ignored */)
{
        Seln_rank ret;
        char *value = (char *)panel_get_value(text_item);

        if (selection_source == FROMFILE_CHOICE) {
                /* set the selection from a file; the selection service will
                   actually acquire the selection and handle all requests */

                if (seln_hold_file(type_to_rank[selection_type], value)
                                                        != SELN_SUCCESS) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                                "Could not set selection from named file!", 0);
                        err++;
                } else if (err) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                                mesg_labels[selection_type][selection_source],0);
                        err = 0;
                }
                return;
        }
        ret = seln_acquire(s_client, type_to_rank[selection_type]);

        /* check that the selection rank we received is the one we asked for */

        if (ret != type_to_rank[selection_type]) {
                panel_set(mesg_item, PANEL_LABEL_STRING,
                                "Could not acquire selection!", 0);
                err++;
                return;
        }

        set_selection_value(selection_type, selection_source, value);
}

/*
 * copy the new value of the appropriate selection into its
 * buffer so that if the user changes the text item and/or the current
 * selection type, the selection won't mysteriously change
 */

set_selection_value(type, source, value)
```

```
int type, source;
char *value;
{
        if (selection_bufs[type] != NULL)
                free(selection_bufs[type]);
        selection_bufs[type] = malloc(strlen(value) + 1);
        if (selection_bufs[type] == NULL) {
                panel_set(mesg_item, PANEL_LABEL_STRING, "Out of memory!", 0);
                err++;
        } else {
                strcpy(selection_bufs[type], value);
                if (err) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                                                mesg_labels[type][source], 0);
                        err = 0;
                }
        }
}

/*
 * func_key_proc
 *
 * called by the selection service library whenever a change in the state of
 * the function keys requires an action (for instance, put the primary
 * selection on the shelf if the user hit PUT)
 */

func_key_proc(client_data, args)
char *client_data;
Seln_function_buffer *args;
{
        Seln_holder *holder;

        /* use seln_figure_response to decide what action to take */

        switch (seln_figure_response(args, &holder)) {
        case SELN_IGNORE:
                /* don't do anything */
                break;
        case SELN_REQUEST:
                /* expected to make a request */
                break;
        case SELN_SHELVE:
                /* put the primary selection (which we should have) on the
                   shelf */
                if (seln_acquire(s_client, SELN_SHELF) != SELN_SHELF) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                                        "Could not acquire shelf!", 0);
                        err++;
                } else {
                        shelve_primary_selection();
                }
                break;
```

```
        case SELN_FIND:
                /* do a search */
                break;
        case SELN_DELETE:
                /* do a delete */
                break;
        }
}

shelve_primary_selection()
{
        char *value = selection_bufs[PRIMARY_CHOICE];

        if (selection_bufs[SHELF_CHOICE] != NULL)
                free(selection_bufs[SHELF_CHOICE]);
        selection_bufs[SHELF_CHOICE] = malloc(strlen(value)+1);
        if (selection_bufs[SHELF_CHOICE] == NULL) {
                panel_set(mesg_item, PANEL_LABEL_STRING, "Out of memory!", 0);
                err++;
        } else {
                strcpy(selection_bufs[SHELF_CHOICE], value);
        }
}

/*
 * reply_proc
 *
 * called by the selection service library whenever a request comes from
 * another client for one of the selections we currently hold
 */

Seln_result
reply_proc(item, context, length)
Seln_attribute item;
Seln_replier_data *context;
int length;
{
        int size, needed;
        char *seln, *destp;

        /* determine the rank of the request and choose the
           appropriate selection */

        switch (context->rank) {
        case SELN_PRIMARY:
                seln = selection_bufs[PRIMARY_CHOICE];
                break;
        case SELN_SECONDARY:
                seln = selection_bufs[SECONDARY_CHOICE];
                break;
        case SELN_SHELF:
                seln = selection_bufs[SHELF_CHOICE];
                break;
```

```
default:
        seln = NULL;
}

/* process the request */

switch (item) {
case SELN_REQ_CONTENTS_ASCII:
        /* send the selection */

        /* if context->context == NULL then we must start sending
           this selection; if it is not NULL, then the selection
           was too large to fit in one buffer and this call must
           send the next buffer; a pointer to the location to start
           sending from was stored in context->context on the
           previous call */

        if (context->context == NULL) {
                if (seln == NULL)
                        return(SELN_DIDNT_HAVE);
                context->context = seln;
        }
        size = strlen(context->context);
        destp = (char *)context->response_pointer;

        /* compute how much space we need: the length of the selection
           (size), plus 4 bytes for the terminating null word, plus 0
           to 3 bytes to pad the end of the selection to a word
           boundary */

        needed = size + 4;
        if (size % 4 != 0)
                needed += 4 - size % 4;
        if (needed <= length) {
                /* the entire selection fits */
                strcpy(destp, context->context);
                destp += size;
                while ((int)destp % 4 != 0) {
                        /* pad to a word boundary */
                        *destp++ = '\0';
                }
                /* update selection service's pointer so it can
                   determine how much data we are sending */
                context->response_pointer = (char **)destp;
                /* terminate with a NULL word */
                *context->response_pointer++ = 0;
                return(SELN_SUCCESS);
        } else {
                /* selection doesn't fit in a single buffer; rest
                   will be put in different buffers on subsequent
                   calls */
                strncpy(destp, context->context, length);
                destp += length;
```

```
                        context->response_pointer = (char **)destp;
                        context->context += length;
                        return(SELN_CONTINUED);
                }
        case SELN_REQ_YIELD:
                /* deselect the selection we have (turn off highlight, etc.) */

                *context->response_pointer++ = (char *)SELN_SUCCESS;
                return(SELN_SUCCESS);
        case SELN_REQ_BYTESIZE:
                /* send the length of the selection */

                if (seln == NULL)
                        return(SELN_DIDNT_HAVE);
                *context->response_pointer++ = (char *)strlen(seln);
                return(SELN_SUCCESS);
        case SELN_REQ_END_REQUEST:
                /* all attributes have been taken care of; release any
                   internal storage used */
                return(SELN_SUCCESS);
                break;
        default:
                /* unrecognized request */
                return(SELN_UNRECOGNIZED);
        }
        /* NOTREACHED */
}
```

```
/*****************************************************************/
/* routines involving getting a selection                       */
/*****************************************************************/

/*
 * get the value of the selection type specified by the current panel choices
 * from whichever client is currently holding it
 */

static void
get_button_proc(/* args ignored */)
{
        Seln_holder holder;
        int len;
        char context = FIRST_BUFFER; /* context value used when a very long
                                        message is received; see procedure
                                        comment for read_proc */

        if (err) {
                panel_set(mesg_item, PANEL_LABEL_STRING,
                        mesg_labels[selection_type][selection_source], 0);
                err = 0;
        }

        /* determine who has the selection of the rank we want */

        holder = seln_inquire(type_to_rank[selection_type]);
        if (holder.state == SELN_NONE) {
                panel_set(mesg_item, PANEL_LABEL_STRING,
                                "You must make a selection first!", 0);
                err++;
                return;
        }

        /* ask for the length of the selection and then the actual
           selection; read_proc actually reads it in */

        (void) seln_query(&holder, read_proc, &context,
                                SELN_REQ_BYTESIZE, 0,
                                SELN_REQ_CONTENTS_ASCII, 0,
                                0);


        /* display the selection in the panel */

        len = strlen(selection_bufs[selection_type]);
        if (len > (int)panel_get(text_item, PANEL_VALUE_STORED_LENGTH))
                panel_set(text_item, PANEL_VALUE_STORED_LENGTH, len, 0);
        panel_set_value(text_item, selection_bufs[selection_type]);
}

/*
 * called by seln_query for every buffer of information received; short
```

```
 * messages (under about 2000 bytes) will fit into one buffer; for larger
 * messages, read_proc will be called with each buffer in turn; the context
 * pointer passed to seln_query is modified by read_proc so that we will know
 * if this is the first buffer or not
 */

Seln_result
read_proc(buffer)
Seln_request *buffer;
{
        char *reply;    /* pointer to the data in the buffer received */
        unsigned len;   /* amount of data left in the buffer */
        int bytes_to_copy;
        static int selection_have_bytes; /* number of bytes of the selection
                        which have been read; cumulative over all calls for
                        the same selection (it is reset when the first
                        buffer of a selection is read) */
        static int selection_len;       /* total number of bytes in the current
                                   selection */

        if (*buffer->requester.context == FIRST_BUFFER) {

                /* this is the first buffer */

                if (buffer == (Seln_request *)NULL) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                                "Error reading selection - NULL buffer", 0);
                        err++;
                        return(SELN_UNRECOGNIZED);
                }
                reply = buffer->data;
                len = buffer->buf_size;

                /* read in the length of the selection */

                if (*((Seln_attribute *)reply) != SELN_REQ_BYTESIZE) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                              "Error reading selection - unrecognized request",
                              0);
                        err++;
                        return(SELN_UNRECOGNIZED);
                }
                reply += sizeof(Seln_attribute);
                len = buffer->buf_size - sizeof(Seln_attribute);
                selection_len = *(int *)reply;
                reply += sizeof(int); /* this only works since an int is 4
                                     bytes; all values must be padded to
                                     4-byte word boundaries */
                len -= sizeof(int);

                /* create a buffer to store the selection */

                if (selection_bufs[selection_type] != NULL)
```

```
                        free(selection_bufs[selection_type]);
                selection_bufs[selection_type] = malloc(selection_len + 1);
                if (selection_bufs[selection_type] == NULL) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                                                        "Out of memory!", 0);
                        err++;
                        return(SELN_FAILED);
                }
                selection_have_bytes = 0;

                /* start reading the selection */

                if (*(Seln_attribute *)reply != SELN_REQ_CONTENTS_ASCII) {
                        panel_set(mesg_item, PANEL_LABEL_STRING,
                             "Error reading selection - unrecognized request",
                             0);
                        err++;
                        return(SELN_UNRECOGNIZED);
                }
                reply += sizeof(Seln_attribute);
                len -= sizeof(Seln_attribute);
                *buffer->requester.context = NOT_FIRST_BUFFER;
        } else {
                /* this is not the first buffer, so the contents of the buffer
                   is just more of the selection */

                reply = buffer->data;
                len = buffer->buf_size;
        }

        /* copy data from the received buffer to the selection buffer
           allocated above */

        bytes_to_copy = selection_len - selection_have_bytes;
        if (len < bytes_to_copy)
                bytes_to_copy = len;
        strncpy(&selection_bufs[selection_type][selection_have_bytes],
                                                        reply, bytes_to_copy);
        selection_have_bytes += bytes_to_copy;
        if (selection_have_bytes == selection_len)
                selection_bufs[selection_type][selection_len] = '\0';
        return(SELN_SUCCESS);
}
```

```
/*****************************************************************/
/* panel routines                                                */
/*****************************************************************/

/* panel initialization routine */

init_panel(panel)
Panel panel;
{
        mesg_item = panel_create_item(panel, PANEL_MESSAGE,
                    PANEL_LABEL_STRING,
                            mesg_labels[PRIMARY_CHOICE][ITEM_CHOICE],
                    0);
        type_item = panel_create_item(panel, PANEL_CYCLE,
                    PANEL_LABEL_STRING,      "Set/Get: ",
                    PANEL_CHOICE_STRINGS,    "Primary Selection",
                                             "Secondary Selection",
                                             "Shelf",
                                             0,
                    PANEL_NOTIFY_PROC,       change_label_proc,
                    PANEL_LABEL_X,           ATTR_COL(0),
                    PANEL_LABEL_Y,           ATTR_ROW(1),
                    0);
        source_item = panel_create_item(panel, PANEL_CYCLE,
                    PANEL_LABEL_STRING,      "Text item contains:",
                    PANEL_CHOICE_STRINGS,    "Selection",
                                     "Filename Containing Selection",
                                             0,
                    PANEL_NOTIFY_PROC,       change_label_proc,
                    0);
        text_item = panel_create_item(panel, PANEL_TEXT,
                    PANEL_LABEL_STRING,
                            text_labels[PRIMARY_CHOICE][ITEM_CHOICE],
                    PANEL_VALUE_DISPLAY_LENGTH, 20,
                    0);
        set_item[0] = panel_create_item(panel, PANEL_BUTTON,
                    PANEL_LABEL_IMAGE,       panel_button_image(panel,
                                                     "Set Selection", 15,0),
                    PANEL_NOTIFY_PROC,       set_button_proc,
                    PANEL_LABEL_X,           ATTR_COL(0),
                    PANEL_LABEL_Y,           ATTR_ROW(5),
                    0);
        set_item[1] = panel_create_item(panel, PANEL_BUTTON,
                    PANEL_LABEL_IMAGE,       panel_button_image(panel,
                                                     "Set Secondary", 15,0),
                    PANEL_NOTIFY_PROC,       set_button_proc,
                    PANEL_LABEL_X,           ATTR_COL(0),
                    PANEL_LABEL_Y,           ATTR_ROW(5),
                    PANEL_SHOW_ITEM,         FALSE,
                    0);
        set_item[2] = panel_create_item(panel, PANEL_BUTTON,
                    PANEL_LABEL_IMAGE,       panel_button_image(panel,
                                                     "Set Shelf", 15,0),
```

```
                        PANEL_NOTIFY_PROC,      set_button_proc,
                        PANEL_LABEL_X,          ATTR_COL(0),
                        PANEL_LABEL_Y,          ATTR_ROW(5),
                        PANEL_SHOW_ITEM,        FALSE,
                        0);
        get_item[0] = panel_create_item(panel, PANEL_BUTTON,
                        PANEL_LABEL_IMAGE,      panel_button_image(panel,
                                                        "Get Selection", 15,0),
                        PANEL_NOTIFY_PROC,      get_button_proc,
                        PANEL_LABEL_X,          ATTR_COL(20),
                        PANEL_LABEL_Y,          ATTR_ROW(5),
                        0);
        get_item[1] = panel_create_item(panel, PANEL_BUTTON,
                        PANEL_LABEL_IMAGE,      panel_button_image(panel,
                                                        "Get Secondary", 15,0),
                        PANEL_NOTIFY_PROC,      get_button_proc,
                        PANEL_SHOW_ITEM,        FALSE,
                        PANEL_LABEL_X,          ATTR_COL(20),
                        PANEL_LABEL_Y,          ATTR_ROW(5),
                        0);
        get_item[2] = panel_create_item(panel, PANEL_BUTTON,
                        PANEL_LABEL_IMAGE,      panel_button_image(panel,
                                                        "Get Shelf", 15,0),
                        PANEL_NOTIFY_PROC,      get_button_proc,
                        PANEL_SHOW_ITEM,        FALSE,
                        PANEL_LABEL_X,          ATTR_COL(20),
                        PANEL_LABEL_Y,          ATTR_ROW(5),
                        0);
}

/*
 * change the label of the text item to reflect the currently chosen selection
 * type
 */

static void
change_label_proc(item, value, event)
Panel_item item;
int value;
Event *event;
{
        int old_selection_type = selection_type;

        selection_type = (int)panel_get_value(type_item);
        selection_source = (int)panel_get_value(source_item);
        panel_set(text_item, PANEL_LABEL_STRING,
                        text_labels[selection_type][selection_source], 0);
        panel_set(mesg_item, PANEL_LABEL_STRING,
                        mesg_labels[selection_type][selection_source], 0);
        if (old_selection_type != selection_type) {
                panel_set(set_item[old_selection_type],
                                PANEL_SHOW_ITEM, FALSE, 0);
                panel_set(set_item[selection_type],
```

```
                                  PANEL_SHOW_ITEM, TRUE, 0);
                panel_set(get_item[old_selection_type],
                                  PANEL_SHOW_ITEM, FALSE, 0);
                panel_set(get_item[selection_type],
                                  PANEL_SHOW_ITEM, TRUE, 0);
        }
}
```

10

The User Defaults Database

10

The User Defaults Database

Many UNIX programs are *customizable* in that the user can modify their behavior by setting certain parameters checked by the program at startup time. This approach has been extended in SunView to include facilities used by many applications, such as menus, text and scrollbars, as well as applications.

The SunView *user defaults database* is a centralized database for maintaining customization information about different programs and facilities.

This chapter is addressed to programmers who want their programs to make use of the defaults database. For a discussion of the user interface to the defaults database, read the `defaultsedit(1)` manual page in the *SunOS Reference Manual* and see the section on `defaultsedit` in the *SunView 1 Beginner's Guide*.

In this chapter, customizable parameters are referred to as *options*; the values they can be set to are referred to as *values*.

All definitions necessary to use the defaults database may be obtained by including the file `<sunwindow/defaults.h>`.

Why a Centralized Database?

Traditionally, each customizable program has a corresponding *customization file* in the user's home directory. The program reads its customization file at startup time to get the values the user has specified.

Examples of customizable programs are `mail(1)`, `csh(1)`, and `sunview(1)`. The corresponding customization files are `.mailrc`, `.cshrc`, and `.sunview`.

While this method of handling customization works well enough, it can become confusing to the user because:

- Since the information is scattered among programs, it's difficult for the user to determine what options she or he can set.
- Since the format of each customization file is different, the user must find and read documentation for each program he or she wants to customize.
- Even after the user has located the customization file and become familiar with its format, it's often difficult for the user to determine what the legal values are for a particular option.

SunView addresses these problems by providing a centralized database which can be used by any customizable program. The user can view and modify the options in the defaults database with the interactive program `defaultsedit(1)`.

10.1. Overview

The defaults database actually consists of a single *master database* and a *private database* for each user.

The master database contains all the options for each program which uses the defaults database. For each option, the *default value* is given.

The user's private database contains the values she or he has specified using `defaultsedit`. An option's value in the private database takes precedence over the option's default value in the master database.

Application programs retrieve values from the database using the routines described later in this chapter. These routines first search the user's private database for the value. If the value is not found in the private database, then the default value from the master database is returned. Each of these routines specify a fall-back default value which is used if neither database contains the value. It should match the value in the master database.

Master Database Files

The master database is stored in the directory `/usr/lib/defaults` as a number of individual files, each containing the options for one program or package. These files are created with a text editor by the author of the program or package; see Section 10.3, *Creating a* `.d` *File: Example*, later in this chapter. By convention, the file name is the capitalized name of the program or package, with the suffix `.d` — `Mail.d`, `SunView.d`, `Menu.d`, etc.

The defaults database itself has two options you can set using `defaultsedit` to control where the master database resides:

- *Directory* is provided so that a group may have its own master database directory in which to do development independently of the standard `/usr/lib/defaults` directory.
- *Private_Directory* is provided so that an individual developer may have his own private master database for development. Note that this directory must have copies (or symbolic links) to all of the `.d` files in `/usr/lib/defaults`, or accesses to the absent files will result in run-time errors.

When the master database is accessed, the defaults routines look for the appropriate `.d` file first in the *Private_directory* (if specified). If the file is not found or the directory not specified, then if a *Directory* is specified it is searched, otherwise the default directory, `/usr/lib/defaults`, is searched.

Private Database Files

A user's private database is stored in the file `.defaults` in the user's home directory. This is where changes the user makes using `defaultsedit` are recorded.[23]

[23] There is rarely any need for the user to edit his `.defaults` file by hand — it is automatically created

There is an option called *Private_only* which allows the user to disable the reading of the master database entirely, thereby reducing program startup time. Note that for this to work, you must make sure that the fall-back values you specify in your program exactly match the values in the master database.

10.2. File Format

The format for both master and private database files is identical.

The first line in the file contains a version number.[24] The rest of the file consists of a number of lines, each line contains either an option name with its associated value or a comment, preceded by a semi-colon (;). Blank lines are also legal.

Option Names

The option names are organized hierarchically, just like files in a file system. Names must always start with a slash character, (/), and each level in the naming hierarchy is separated from the previous level by a slash character. Each name consists of one or more letters (A-Z, a-z), digits (0-9), dollar signs ($), and underscores (_). By convention, the first letter of each name is capitalized.[25]

There are two shorthand notations for option names. First, whenever a line does not start with a slash, the previous node is prepended to the name (this is similar to the treatment of path names in UNIX). So

```
/SunView/Font
        $Help
```

is equivalent to

```
/SunView/Font
/SunView/Font/$Help
```

The second shorthand convention is that any time two slashes in a row are encountered, the option name previously defined at that level is assumed. Each pair of slashes corresponds to one name. So

```
/SunView/Font
//Walking_Menus
//Icon_gravity
```

is equivalent to

```
/SunView/Font
/SunView/Walking_Menus
        /SunView/Icon_gravity
```

and updated by `defaultsedit`. The one time the user needs to edit his `.defaults` file by hand is to disable the defaults *Testmode* option once it has been enabled. See the discussion in Section 10.7, *Error Handling*, below.

[24] The version number is included so that if any incompatible changes are made to the default database format in the future, the library routines can tell when they encounter an older file format.

[25] This convention is just for readability — internally all names are converted to lower case, so the defaults database is insensitive to case.

and

```
        /SunView/Font/Bold
        ///Italic
        ///Size
```

is equivalent to

```
/SunView/Font/Bold
/SunView/Font/Italic
        /SunView/Font/Size
```

Option Values

All option values are stored as strings. They have the same syntax as quoted strings in C. In particular, the backslash character (\) is used as an escape character for inserting other characters into the quoted string. The following backslash escapes are recognized:

Table 10-1 *Defaults Metacharacters*

Backslash Escape	***Prints as:***
`\\`	Backslash
`\"`	Double quote
`\'`	Single quote
`\n`	Newline
`\t`	Tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\ddd`	3 digit octal number specifying a single character

Option values can be up to 10,000 characters long.

Distinguished Names

There are several distinguished names used by `defaultsedit`. See the next section for an example illustrating their usage.

`$Help`

`$Help` allows you to add an explanatory string to be displayed by `defaultsedit` for each option.

`$Enumeration`

An *enumerated option* is one in which the values are explicitly given, such as {True, False}, {Yes, No}, {North, South, East, West} etc.[26] The user selects one of the values using `defaultsedit`.

The way that `defaultsedit` knows that it has encountered an enumerated option is by the level name `$Enumeration`. The values for the enumerated option follow at the same level. Note that you can specify a help string for the entire enumerated option, as well as specifying the value.

[26] There is no limit to the number of values an enumerated option can have.

$Message

$Message allows you to add a one-line message to be displayed by defaultsedit. Use this to make more readable the display of a category with many options by setting off related options with blank lines or headings.

10.3. Creating a .d File: Example

Adding options for a new program to the database corresponds to adding a new first-level option name in the master database, and appears to the user as a new *category* in defaultsedit. You do this by creating the appropriate .d file in /usr/lib/defaults. If the file is in the correct format, and ends in .d, then defaultsedit will automatically display it as a new category.

Let's create such a file for a game called "Space Wars." The options are: the number of friendly and enemy ships, whether or not stars attract ships, the name of the user's ship, and the direction that ships enter the window from.

To conform to the naming convention for master database files, we add the suffix .d to the first-level option name, yielding the filename SpaceWar.d:

```
SunDefaults_Version 2
/SpaceWar
        $Help               "A space ship battle game"
//Friends                "15"
        $Help               "Number of friendly ships"
//Enemies                "15"
        $Help               "Number of enemy ships"
//Gravity                "Yes"
        $Help               "Affects whether star attract ships"
        $Enumeration        ""
        Yes                 ""
        Yes/$Help          "Stars attract ships"
        No                  ""
        No/$Help            "Ships are immune to attraction"
//Name                   "Battlestar"
        $Help               "Name of your space ship"
//Direction              "North"
        $Help               "Starting window border"
        $Enumeration        ""
        North               ""
        North/$Help         "Ships start at north window border"
        South               ""
        South/$Help         "Ships start at south window border"
        East                ""
        East/$Help          "Ships start at east window border"
        West                ""
        West/$Help          "Ships start at west window border"
```

Note that the highest-level option name, /SpaceWar, has no associated value, since it wouldn't make sense to have one. If a database routine tries to access an option which has no value, the special string DEFAULTS_UNDEFINED will be returned.

10.4. Retrieving Option Values

A simple programmatic interface is provided to retrieve option values from the defaults database. All values are stored as strings, and may be retrieved with `defaults_get_string()`. For convenience, similar get routines are provided to retrieve values as integers, characters, or enumerated types. The get routines are described below.

Retrieving String Values

To retrieve a string value, use:

```
char *
defaults_get_string(option_name, default_value, 0)
    char *option_name;
    char *default_value;
```

`option_name` is the name of the option whose value will be retrieved. `default` is a value to return if the option is not found in the database or if the database itself cannot be accessed for any reason. Note that this value should match the default value in the master database. The final argument to all `defaults_get*()` routines is zero.[27] In our Space Wars example in the previous section, we would call

```
ship = defaults_get_string("/SpaceWar/Name",
                           "Battlestar", 0);
```

On return, `ship` would point to the string `Battlestar`.

Suppose you misspelled the option name `Name` as `Nane`. Since `/SpaceWar/Nane` is not in the defaults database, the fall-back value of `Battlestar` will be returned and an error message may be output.[28]

Retrieving Integer Values

To retrieve an integer value, use:

```
int
defaults_get_integer(option_name, default_value, 0)
    char *option_name;
    int   default_value;
```

This function gets the option value associated with `option_name`, treats it as a decimal integer, and returns the integer value. For example, the string `"17"` parses into the number `17` and the string `"-123"` parses into the number `-123`. If `option_name` can't be found, or its associated value can't be parsed, the integer passed in for `default_value` is returned. For example, the call

```
defaults_get_integer("/SpaceWar/Enemys", 15, 0);
```

will return the integer `15`, since "Enemies" was misspelled as "Enemys".

[27] This third argument is not currently used. It is necessary for compatibility with future releases of the defaults database package, which may use the third argument to return status information.

[28] Whether or not the database retrieval routines generate error messages on error conditions depends on the setting of the option *Error_Action*. See Section 10.7, *Error Handling*, below.

The function `defaults_get_integer_check()` is the same as `defaults_get_integer()`, except that it checks that the returned value is within a specified range:

```
int
defaults_get_integer_check(option_name, default_value,
                                  min, max, 0)
     char *option_name;
     int   default_value;
     int   min, max;
```

If the option value is not between `min` and `max`, the integer passed in for `default_value` is returned and an error message may be output.

Retrieving Character Values

To retrieve a character value, use:

```
int
defaults_get_character(option_name, default_value, 0)
     char *option_name;
     char  default_value;
```

`defaults_get_character()` returns the first character from the option value. If the option value contains more than one character, the character passed in for `default_value` is returned and an error message is output.

Retrieving Boolean Values

To retrieve a boolean value,[29] use:

```
Bool
defaults_get_boolean(option_name, default_value, 0)
     char *option_name;
     Bool  default_value;
```

`defaults_get_boolean()` returns `True` if the option value is `True`, `Yes`, `Enabled`, `Set`, `Activated`, or `1` and `False` if the option value is `False`, `No`, `Off`, `Disabled`, `Reset`, `Cleared`, `Deactivated`, or `0`. If the option value is not one of the above, the value passed in for `default_value` is returned and an error message is output.

[29] The definition for `Bool`, found in `<sunwindow/sun.h>`, is: `typedef enum {False = 0, True = 1} Bool;`.

Retrieving Enumerated Values

You can retrieve enumerated option values with `defaults_get_string()`, then use `strcmp(3)` to test which value was returned. As an alternative, you may find it more convenient to define an enumerated type corresponding to the option values, and use `defaults_get_enum()` to return the option value as the corresponding enum. The definition is:

```
int
defaults_get_enum(option_name, pairs)
    char            *option_name;
    Defaults_pairs  pairs[];
```

`pairs` is a pointer to an array of `Defaults_pairs` which contains name-value pairs. `Defaults_pairs` is defined as:

```
typedef struct {
        char    *name;
        int      value;
} Defaults_pairs;
```

The array passed in as `pairs` must be null-terminated. `defaults_get_enum()` returns the `name` associated with the `value` which is the current value of the option. If no match is found, the `value` associated with the last (null) entry is returned.

The following example, using the direction option for our Space Wars example, illustrates the usage of `defaults_get_enum()`:

```
typedef enum {North, South, East, West} directions;
directions dir;
Defaults_pairs direction_pairs [] = {
        "North",  (int) North,
        "South",  (int) South,
        "East",   (int) East,
        "West",   (int) West,
        NULL,     (int) North};      /* Error value */

dir = defaults_get_enum("/SpaceWar/Direction", direction_pairs);
```

Searching for Specific Symbols

To probe the defaults database to see whether or not a particular symbol is stored in it, use the `defaults_exists()` routine. This routine will return `TRUE`, if `path_name` has a value defined in the defaults database. Otherwise a value of `FALSE` will be returned.

```
Bool flag1 = defaults_exists("/SpaceWar/Ship_name", NULL);
Bool flag2 = defaults_exists("/SpaceWar/Fred", NULL);
```

`flag1` has a value of `TRUE`; `flag2` has a value of `FALSE`.

Searching for Specific Values

To find the original value of a particular database entry before the client's personalized database overwrites it, use

```
defaults_get_default(path_name, default_value, status)
```

For example, assume that the master database has the entry

```
/SpaceWar/ShipName "Lollipop"
```

and that the client's private database is

```
/SpaceWar/ShipName "Death Avenger"
```

If you call `defaults_get_string("/SpaceWar/ShipName", "<err>", NULL)"` it will return Lollipop. If, however, the path_name is not in the database, then the default value will be returned.

Retrieving all Values in the Database

To search the database to find all of the values in the database, use the routines `defaults_get_child()` and `defaults_get_sibling()`. The routine `defaults_get_child(path_name, status)` returns the simple name of the the database entry immediately under `path_name`. If you use `defaults_get_child("/SpaceWar", NULL)` it will return `ShipName`. You can use `sprintf`(3S) to construct the full path name:

```
char temp[1000], *child;
child = defaults_get_child("/SpaceWar", NULL);
if (child == NULL) {
    (void)fprintf(stderr, "Error");
    exit(1);
}
sprintf(temp, "%s/%s", "/SpaceWar", child);
```

`temp` would contain `/SpaceWar/ShipName`. A `NULL` value is returned in there is no child.

`defaults_get_sibling(path_name, status)` returns the simple name of the next database entry immediately following pathname at the same level. So, if you use `defaults_get_sibling("/SpaceWar/ShipName", NULL)` it will return `Framus`. This can be assembled into a full path name using `sprintf`(3S).

```
char temp[1000], *sibling;
sibling = defaults_get_sibling("/SpaceWar/ShipName");
if (sibling == NULL) {
    (void)fprintf(stderr, "Error");
    exit(1);
}
sprintf(temp, "%s/%s", "/SpaceWar", sibling);
```

The following program will dump the entire contents of the defaults database along with their associated values.

```
void
dump_defaults(path_name, indent)
    char    *path_name;
    char    *indent;
{
    char   temp[1000];
    char   *child;

    (void)printf("%s%s %s0, indent, path_name,
                (defaults_get_string(path_name, "<err>", NULL ));
    child = defaults_get_child(path_name, NULL);
         if (child == NULL) {
             return;
         }
    len = strlen(indent);
    indent[len] = '';
    indent[len+1] = ' ';
    (void)sprintf(temp, "%s/%s", path_name, child);
    dump_defaults(temp, indent);
          while (sibling = defaults_get_sibling(temp, NULL)
                                          != NULL) {
                (void)sprintf(temp, "%s/%s", path_name,
                                               sibling);
                (dump_defaults(temp, indent);
          }
}

main()
{
    char indent_buf[100] = "";
    dump_defaults("/", indent);
}
```

10.5. Conversion Programs

The defaults package provides a mechanism to convert from an existing customization file, such as `.mailrc`, to the `.d` format used by `defaultsedit`.

You must write a separate program to do the conversion each way. Specify the name of the program converting from the existing customization file to the defaults format as the value of the `$Specialformat_to_defaults` option in the corresponding `.d` file. The program to go the other way is specified as `$Defaults_to_specialformat`.

As an example, at Sun we have written programs to convert from the traditional `.mailrc` file to the defaults format. The file `/usr/lib/defaults/Mail.d` contains the lines:

```
/Mail                          ""
//$Specialformat_to_defaults "mailrc_to_defaults"
//$Defaults_to_specialformat "defaults_to_mailrc"
```

If a program is specified as the value for `$Specialformat_to_defaults`, then `defaultsedit` runs the program the first time it needs to display the options for that category. When the user saves the changes she or he has made to the database, and any changes that were made to the category, then the `$Defaults_to_specialformat` program is run.

To write your own conversion programs, use the following guidelines. Read the customization file into the program. Then, to go from the customization file to `.defaults`, you simply figure out the appropriate option value to set, and set it with the routine `defaults_set_string()`.[30] To go the other way, retrieve the value from the defaults database with the appropriate get routine, then make the appropriate change to the customization file.

Note: Conversion programs should use the master database, regardless of the setting of the `defaultsedit` option *Private-only*. To do this, call the function `defaults_special_mode()` as the first statement of your program.

10.6. Property Sheets

Many window programs have property sheets that the end-user can use to modify the behavior of their programs. You may use the defaults database to store the information set by the user.

The following discussion uses code fragments taken from the *default_filer* program printed at the end of this chapter.

The *filer* program has a property sheet that consists of one item, the flags to pass through to the `ls` command. This property is represented as a string. When the property sheet is popped up it is necessary to read the value from the database:

```
char *filer_flags;
filer_flags = defaults_get_string("/Filer/LsFlags", "-l",
                                                    NULL);
```

When, the user changes the flags property, it is necessary to store the new value into the database. This is done using `defaults_set_string(path_name, new_value, status)`.

For example:

```
filer_flags = code from example;
defaults_set_string("/Filer/LsFlags", filer_flags, NULL);
```

[30] `defaults_set_string()` is documented in `<defaults/defaults.h>`.

This code writes the new value into the defaults database in memory. The new value will not be stored in the file system until the user pushes the **Done** button. When this occurs, the routine `defaults_write_changed(file_name, status)` is called. This routine will write any database values in memory that are different from the ones in the master database to the defaults file `file_name`. If `file_name` has a value of `NULL`, then it will be written out to the user's private defaults database file. So,

```
defaults_write_changed(NULL, NULL);
```

will cause the defaultsvalues to be stored to `~/.defaults`.

If the user pushes the **Reset** button, then you want to reset the property sheet to be the value that the property sheet had when it first came up. A call to `defaults_reread(path_name, status)` will restore the database under path_name from the file system. So,

```
defaults_reread("/Filer", NULL);
```

will restore the defaults database for the filer property sheet. You can obtain the original value by reentering the property sheet display code to obtain the original values.

10.7. Error Handling

The defaults routines report errors by printing messages on the standard error stream `stderr`. The most common cause for getting error messages is that a program that uses the defaults database is copied from somewhere without also copying the associated master defaults database file. While these messages are annoying, in general the program will continue to work, since every routine that accesses the defaults database has a `default_value` argument that will be returned if an option is not present in the database.[31]

Using `defaultsedit`, the user can set two options for the defaults database itself to control error reporting:

Error_Action

Error_Action controls what happens when an error is encountered. Possible values are:

- *Continue*: print an error message and continue execution.
- *Suppress*: no action is taken.
- *Abort*: print an error message and terminate execution on encountering the first error.

Most users will want to set *Error_Action* to either *Continue* or *Suppress*. Use *Suppress* if you are getting all sorts of extraneous defaults error messages. *Abort* is useful for forcing programmers to track down extraneous error messages prior to releasing software to a larger community.

[31] These error messages are not printed when *Private_only* is *True*.

Maximum_Errors

Maximum_Errors puts a limit on the number of error messages which will be printed regardless of the setting of *Error_Action*.

Test_Mode

The option *Test_Mode* is provided to facilitate the testing of software prior to release to a larger community. Use it to check for incorrect values for the `default_value` argument to the get routines. When *Test_Mode* is set to *Enable*, the defaults database is made inaccessible. In this mode, every time an option value is accessed, a diagnostic message is generated and the value passed in as `default_value` is returned.

Note that once enabled, *Test_Mode* can not be disabled using `defaultsedit`. This is one time when you must edit your `.defaults` file by hand, to set the *Test_Mode* option to *Disabled* (or remove the entry altogether).

10.8. Interface Summary

The following table lists and explains all of the procedures that may be used to make use of the defaults database.

Table 10-2 *Default Procedures*

Routine	*Description*

`Bool`
`defaults_exists(path_name, status)`
`    char *path_name;`
`    int  *status;`
	Returns `TRUE` if `path_name` exists in the database.

`Bool`
`defaults_get_boolean(option_name, default, 0)`
`    char *option_name;`
`    Bool  default;`
	Looks up `path_name` in the database and returns `TRUE` if the value is `True`, `Yes`, `On`, `Enabled`, `Set`, `Activated`, or `1`. `FALSE` is returned if the value is `False`, `No`, `Off`, `Disabled`, `Reset`, `Cleared`, `Deactivated`, or `0`. If the value is not one of the above values, then a warning message is displayed and the default is returned.

`char`
`defaults_get_character(option_name, default, 0)`
`    char *option_name;`
`    char  default;`
	Looks up `path_name` in the defaults database and returns the resulting character value. The default value is returned in an error occurs.

`char *`
`defaults_get_child(path_name, status)`
`    char *path_name;`
`    int  *status;`
	Returns a pointer to the simple name associated with the next sibling of `path_name`. `NULL` will be returned if `path_name` does not exist or if `path_name` does not have an next sibling

Table 10-2 *Default Procedures— Continued*

Routine	Description

`char *`
`defaults_get_default(path_name, default, status)`
`    char *path_name;`
`    char *default_value;`
`    int  *status;`
	Returns the value associated with `path_name` prior to being overridden by the clients private database. `default` is returned in any error occurs.

`int`
`defaults_get_enum(option_name, pairs, 0)`
`    char           *option_name;`
`    Defaults_pairs *pairs;`
	Looks up the values associated with `path_name`, scans the pairs table, and returns the associated value. If no match can be found, then an error will be generated and the value associated with the last entry will be returned. (See `defaults_lookup()`.)

`int`
`defaults_get_integer(option_name, default, 0)`
`    char *option_name;`
`    int   default;`
	Looks up `path_name` in the defaults database and returns the resulting integer. The default value is returned if any error occurs.

`int`
`defaults_get_integer_check(option_name, default_value,`
`                           min, max, 0)`
`    char *option_name;`
`    int   default_value;`
`    int   min, max;`
	Looks up `path_name` in the defaults database and returns the resulting value. If the value in the database is not between minimum and maximum (inclusive), then an error message will be printed. The default will be returned if an error occurs.

`char *`
`defaults_get_sibling(path_name, status)`
`    char   *path_name;`
`    int    *status;`
	Returns a pointer to the simple name associated with the next sibling of `path_name`. `NULL` will be returned, if `path_name`does not exist or if `path_name` does not have an next sibling.

`char *`
`defaults_get_string(option_name, default, 0)`
`    char *option_name;`
`    char *default;`
	Looks up `path_name` in the defaults database and returns the string value associated with it. The default will be retuned if any error occurs.

`void`
`defaults_reread(path_name, status)`
`    char  *path_name;`
`    int   *status;`
	Rereads the portion of the database associated with `path_name`.

`defaults_set_character(path_name, value, status)`
`        char *path_name;`
`        char  value;`
`        int  *status;`
	Sets `path_name` to `value`, while `value` is a character.

Table 10-2 *Default Procedures— Continued*

Routine	*Description*

`void`
`defaults_set_enumeration(path_name, value, status)`
`    char    *path_name;`
`    char    *value;`
`    int     *status;`
	Sets `path_name` to value, where `value` is a pointer to a string.

`void`
`defaults_set_integer(path_name, value, status)`
`    char     *path_name;`
`    int      value;`
`    int      *status;`
	Sets `path_name` to `value`, where `value` is an integer.

`void`
`defaults_set_string(path_name, value, status)`
`    char     *path_name;`
`    char     *value;`
`    int      *status;`
	Sets `path_name` to `value`where `value` is a pointer to a string

`void` `defaults_special_mode()`	Causes the database to behave as if the entire master database has been read into memory prior to reading in the private database. This is done to insure that the order of nodes that `defaultsedit` presents is the same as that in the `.d` files, regardless of what the user happens to have set in his or her private database.

`void`
`defaults_write_all(path_name, file_name, status)`
`    char     *path_name;`
`    char     *file_name;`
`    int      *status;`
	Writes out all of the data base nodes from `path_name` and below into `file_name`. `Out_file` is the string name of the file to create. If `file_name` is `NULL`, then env var `DEFAULTS_FILE` will be used.

`void`
`defaults_write_changed(file_name, status)`
`    char    *file_name;`
`    int     *status;`
	Writes out all of the private database entries to `file_name`. Any time a database node is set, it becomes part of the private database. If the value of the `File_Name` is `NULL`, then `DEFAULTS_FILE` will be used.

`void`
`defaults_write_differences(file_name, status)`
`    char    *file_name;`
`    int     *status;`
	Writes our all of the database entries that differ from the master database. `Out_File` is the string name of the file to create. If `file_name` is `NULL`, env var `DEFAULTS_FILE` is used.

10.9. Example Program: *filer* Defaults Version

The following program is a variation of the *filer* program discussed in the *SunView 1 Programmer's Guide*. It uses some of the defaults procedures discussed in this chapter.

```
/*****************************************************************************/
/*                        4.0     filer_default.c                            */
/*****************************************************************************/

#include <suntool/sunview.h>
#include <sunwindow/defaults.h>
#include <suntool/panel.h>
#include <suntool/tty.h>
#include <suntool/textsw.h>
#include <suntool/seln.h>
#include <suntool/alert.h>
#include <sys/stat.h>       /* stat call needed to verify existence of files */

/* these objects are global so their attributes can be modified or retrieved */
Frame           base_frame, edit_frame, ls_flags_frame;
Panel           panel, ls_flags_panel;
Tty             ttysw;
Textsw          editsw;
Panel_item      dir_item, fname_item, filing_mode_item, reset_item, done_item;
int             quit_confirmed_from_panel;
char *compose_ls_options();

#define         MAX_FILENAME_LEN        256
#define         MAX_PATH_LEN            1024

char *getwd();

main(argc, argv)
    int    argc;
    char **argv;
{
    static Notify_value filer_destroy_func();
    void        ls_flags_proc();

    base_frame = window_create(NULL, FRAME,
                 FRAME_ARGS,             argc, argv,
                 FRAME_LABEL,            "filer",
                 FRAME_PROPS_ACTION_PROC, ls_flags_proc,
                 FRAME_PROPS_ACTIVE, TRUE,
                 FRAME_NO_CONFIRM,       TRUE,
                 0);
    (void) notify_interpose_destroy_func(base_frame, filer_destroy_func);

    create_panel_subwindow();
    create_tty_subwindow();
    create_edit_popup();
    create_ls_flags_popup();
    quit_confirmed_from_panel = 0;

    window_main_loop(base_frame);
    exit(0);
}
```

```
create_tty_subwindow()
{
    ttysw = window_create(base_frame, TTY, 0);
}

create_edit_popup()
{
    edit_frame = window_create(base_frame, FRAME,
                 FRAME_SHOW_LABEL, TRUE,
                 0);
    editsw = window_create(edit_frame, TEXTSW, 0);
}

create_panel_subwindow()
{
    void ls_proc(), ls_flags_proc(), quit_proc(), edit_proc(),
        edit_sel_proc(), del_proc();

    char current_dir[MAX_PATH_LEN];

    panel = window_create(base_frame, PANEL, 0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_X,              ATTR_COL(0),
        PANEL_LABEL_Y,              ATTR_ROW(0),
        PANEL_LABEL_IMAGE,          panel_button_image(panel, "List Directory", 0, 0),
        PANEL_NOTIFY_PROC,          ls_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,          panel_button_image(panel, "Set ls flags", 0, 0),
        PANEL_NOTIFY_PROC,          ls_flags_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,          panel_button_image(panel, "Edit", 0, 0),
        PANEL_NOTIFY_PROC,          edit_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,          panel_button_image(panel, "Delete", 0, 0),
        PANEL_NOTIFY_PROC,          del_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,          panel_button_image(panel, "Quit", 0, 0),
        PANEL_NOTIFY_PROC, quit_proc,
        0);

    filing_mode_item = panel_create_item(panel, PANEL_CYCLE,
        PANEL_LABEL_X,              ATTR_COL(0),
        PANEL_LABEL_Y,              ATTR_ROW(1),
        PANEL_LABEL_STRING,         "Filing Mode:",
        PANEL_CHOICE_STRINGS,       "Use \"File:\" item",
                                    "Use Current Selection", 0,
        0);

    (void) panel_create_item(panel, PANEL_MESSAGE,
```

```
        PANEL_LABEL_X,                  ATTR_COL(0),
        PANEL_LABEL_Y,                  ATTR_ROW(2),
        0);

    dir_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,                  ATTR_COL(0),
        PANEL_LABEL_Y,                  ATTR_ROW(3),
        PANEL_VALUE_DISPLAY_LENGTH,     60,
        PANEL_VALUE,                    getwd(current_dir),
        PANEL_LABEL_STRING,             "Directory: ",
        0);

    fname_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,                  ATTR_COL(0),
        PANEL_LABEL_Y,                  ATTR_ROW(4),
        PANEL_LABEL_DISPLAY_LENGTH,     60,
        PANEL_LABEL_STRING,             "File:  ",
        0);

    window_fit_height(panel);

    window_set(panel, PANEL_CARET_ITEM, fname_item, 0);
}

typedef struct Filer {
        char    *flags;
        char    *path;
};
    struct Filer filer_options = {"   ", "/Filer/Options"};
    struct Filer filer_format = {" 1 ", "/Filer/Format"};
    struct Filer filer_sort_order = {" r ", "/Filer/Sort_Order"};
    struct Filer filer_sort_criterion = {" tu", "/Filer/Sort_Criterion"};
    struct Filer filer_directories = {" d ", "/Filer/Sort_Directories"};
    struct Filer filer_recursive = {" R ", "/Filer/Recursive"};
    struct Filer filer_file_type = {" F ", "/Filer/File_Type"};
    struct Filer filer_dot_files = {" a ", "/Filer/Dot_Files"};

create_ls_flags_popup()
{

    void done_proc();
    void reset_proc();

    ls_flags_frame = window_create(base_frame, FRAME, 0);

    ls_flags_panel = window_create(ls_flags_frame, PANEL, 0);

    panel_create_item(ls_flags_panel, PANEL_MESSAGE,
                PANEL_ITEM_X,           ATTR_COL(14),
                PANEL_ITEM_Y,           ATTR_ROW(0),
                PANEL_LABEL_STRING,     "Options for ls command",
                PANEL_CLIENT_DATA,      &filer_options,
                0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                PANEL_ITEM_X,           ATTR_COL(0),
                PANEL_ITEM_Y,           ATTR_ROW(1),
                PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
```

```
                    PANEL_LABEL_STRING,      "Format:                           ",
                    PANEL_CHOICE_STRINGS,    "Short", "Long", 0,
                    PANEL_CLIENT_DATA,       &filer_format,
                    0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                    PANEL_ITEM_X,            ATTR_COL(0),
                    PANEL_ITEM_Y,            ATTR_ROW(2),
                    PANEL_DISPLAY_LEVEL,     PANEL_CURRENT,
                    PANEL_LABEL_STRING,      "Sort Order:                       ",
                    PANEL_CHOICE_STRINGS,    "Descending", "Ascending", 0,
                    PANEL_CLIENT_DATA,       &filer_sort_order,
                    0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                    PANEL_ITEM_X,            ATTR_COL(0),
                    PANEL_ITEM_Y,            ATTR_ROW(3),
                    PANEL_DISPLAY_LEVEL,     PANEL_CURRENT,
                    PANEL_LABEL_STRING,      "Sort criterion:                   ",
                    PANEL_CHOICE_STRINGS,    "Name", "Modification Time",
                                             "Access Time", 0,
                    PANEL_CLIENT_DATA,       &filer_sort_criterion,
                    0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                    PANEL_ITEM_X,            ATTR_COL(0),
                    PANEL_ITEM_Y,            ATTR_ROW(4),
                    PANEL_DISPLAY_LEVEL,     PANEL_CURRENT,
                    PANEL_LABEL_STRING,      "For directories, list:            ",
                    PANEL_CHOICE_STRINGS,    "Contents", "Name Only", 0,
                    PANEL_CLIENT_DATA,       &filer_directories,
                    0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                    PANEL_ITEM_X,            ATTR_COL(0),
                    PANEL_ITEM_Y,            ATTR_ROW(5),
                    PANEL_DISPLAY_LEVEL,     PANEL_CURRENT,
                    PANEL_LABEL_STRING,      "Recursively list subdirectories? ",
                    PANEL_CHOICE_STRINGS,    "No", "Yes", 0,
                    PANEL_CLIENT_DATA,       &filer_recursive,
                    0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                    PANEL_ITEM_X,            ATTR_COL(0),
                    PANEL_ITEM_Y,            ATTR_ROW(6),
                    PANEL_DISPLAY_LEVEL,     PANEL_CURRENT,
                    PANEL_LABEL_STRING,      "List '.' files?                   ",
                    PANEL_CHOICE_STRINGS,    "No", "Yes", 0,
                    PANEL_CLIENT_DATA,       &filer_dot_files,
                    0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                    PANEL_ITEM_X,            ATTR_COL(0),
                    PANEL_ITEM_Y,            ATTR_ROW(6),
                    PANEL_DISPLAY_LEVEL,     PANEL_CURRENT,
                    PANEL_LABEL_STRING,      "Indicate type of file?            ",
                    PANEL_CHOICE_STRINGS,    "No", "Yes", 0,
                    PANEL_CLIENT_DATA,       &filer_file_type,
```

```
                0);

    done_item = panel_create_item(ls_flags_panel, PANEL_BUTTON,
                PANEL_ITEM_X,           ATTR_COL(1),
                PANEL_ITEM_Y,           ATTR_ROW(8),
                PANEL_LABEL_IMAGE,      panel_button_image(panel, "Done", 6, 0),
                PANEL_NOTIFY_PROC,      done_proc,
                0);

    reset_item = panel_create_item(ls_flags_panel, PANEL_BUTTON,
                PANEL_ITEM_X,           ATTR_COL(12),
                PANEL_LABEL_IMAGE,      panel_button_image(panel, "Reset", 6, 0),
                PANEL_NOTIFY_PROC,      reset_proc,
                0);

    (void)compose_ls_options(1);
    window_fit(ls_flags_panel); /* fit panel around its items */
    window_fit(ls_flags_frame); /* fit frame around its panel */
}

char *
compose_ls_options(control)
    int             control;
{
    static char     flags[20];
    char           *ptr;
    char            flag;
    int             first_flag = TRUE;
    Panel_item      item;
    struct Filer   *client_data;
    int             index;

    ptr = flags;
    panel_each_item(ls_flags_panel, item)
        if ((item != done_item) && (item != reset_item)) {
            client_data = (struct Filer *)panel_get(item,
                                                PANEL_CLIENT_DATA, 0);
            switch (control) {
            case 0:
                index = (int)panel_get_value(item);
                defaults_set_integer((char *)client_data->path,
                        (int)index, (int *)NULL);
                break;
            case 1:
                index = defaults_get_integer((char *)client_data->path,
                        (int)1, NULL);
                panel_set_value(item, index);
                break;
        case 2:
            flag = client_data->flags[index];
            if (flag != ' ') {
                if (first_flag) {
                *ptr++     = '-';
                first_flag = FALSE;
                }
               *ptr++ = flag;
                }
            }
```

```
        }
    panel_end_each
    *ptr = '\0';
    return flags;
}

void
ls_proc()
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;
    char cmdstring[100];        /* dir_item's value can be 80, plus flags */

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir((char *)panel_get_value(dir_item));
        strcpy(previous_dir, current_dir);
    }
    sprintf(cmdstring, "ls %s %s/%s\n",
                compose_ls_options(),
                current_dir,
                panel_get_value(fname_item));
    ttysw_input(ttysw, cmdstring, strlen(cmdstring));
}

void
ls_flags_proc()
{
    window_set(ls_flags_frame, WIN_SHOW, TRUE, 0);
}

void
done_proc()
{
    window_set(ls_flags_frame, WIN_SHOW, FALSE, 0);
    (void)compose_ls_options(0);
    defaults_write_changed(NULL, NULL);
}

void
reset_proc()
{
    defaults_reread("/Filer", NULL);
}

/* return a pointer to the current selection */
char *
get_selection()
{
    static char   filename[MAX_FILENAME_LEN];
    Seln_holder   holder;
    Seln_request *buffer;

    holder = seln_inquire(SELN_PRIMARY);
    buffer = seln_ask(&holder, SELN_REQ_CONTENTS_ASCII, 0, 0);
    strncpy(
        filename, buffer->data + sizeof(Seln_attribute), MAX_FILENAME_LEN);
```

```
    return(filename);
}

/* return 1 if file exists, else print error message and return 0 */
stat_file(filename)
    char  *filename;
{
    static char previous_dir[MAX_PATH_LEN];
    char  *current_dir;
    char   this_file[MAX_PATH_LEN];
    struct stat statbuf;

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir((char *)panel_get_value(dir_item));
        strcpy(previous_dir, current_dir);
    }
    sprintf(this_file, "%s/%s", current_dir, filename);
    if (stat(this_file, &statbuf) < 0) {
        char buf[MAX_FILENAME_LEN+11];  /* big enough for message */
        sprintf(buf, "%s not found.", this_file);
        msg(buf, 1);
        return 0;
    }
    return 1;
}

void
edit_proc()
{
    void edit_file_proc(), edit_sel_proc();
    int file_mode = (int)panel_get_value(filing_mode_item);

    if (file_mode) {
        (void)edit_sel_proc();
    } else {
        (void)edit_file_proc();
    }
}

void
edit_file_proc()
{
    char *filename;

    /* return if no selection */
    if (!strlen(filename = (char *)panel_get_value(fname_item))) {
        msg("Please enter a value for \"File:\".", 1);
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    window_set(editsw, TEXTSW_FILE, filename, 0);
```

```
    window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
}

void
edit_sel_proc()
{
    char *filename;

    /* return if no selection */
    if (!strlen(filename = get_selection())) {
        msg("Please select a file to edit.", 0);
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    window_set(editsw, TEXTSW_FILE, filename, 0);

    window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
}

void
del_proc()
{
    char    buf[300];
    char    *filename;
    int     result;
    Event   event;    /* unused */
    int     file_mode = (int)panel_get_value(filing_mode_item);

    /* return if no selection */
    if (file_mode) {
        if (!strlen(filename = get_selection())) {
            msg("Please select a file to delete.", 1);
            return;
        }
    } else {
        if (!strlen(filename = (char *)panel_get_value(fname_item))) {
            msg("Please enter a file name to delete.", 1);
            return;
        }
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    /* user must confirm the delete */
    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
            "Ok to delete file:",
            filename,
            0,
        ALERT_BUTTON_YES,       "Confirm, delete file",
        ALERT_BUTTON_NO,        "Cancel",
        0);
```

```
    switch (result) {
        case ALERT_YES:
            unlink(filename);
            sprintf(buf, "%s deleted.", filename);
            msg(buf, 0);
            break;
        case ALERT_NO:
            break;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            printf("Alert failed, will not delete %s.\n", filename);
            break;
    }
}

int
confirm_quit()
{
    int     result;
    Event   event;   /* unused */
    char    *msg = "Are you sure you want to Quit?";

    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
            "Are you sure you want to Quit?",
            0,
        ALERT_BUTTON_YES,       "Confirm",
        ALERT_BUTTON_NO,        "Cancel",
        0);
    switch (result) {
        case ALERT_YES:
            break;
        case ALERT_NO:
            return 0;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            printf("Alert failed, quitting anyway.\n");
            break;
    }
    return 1;
}

static Notify_value
filer_destroy_func(client, status)
    Notify_client       client;
    Destroy_status      status;
{
    if (status == DESTROY_CHECKING) {
        if (quit_confirmed_from_panel) {
            return(notify_next_destroy_func(client, status));
        } else if (confirm_quit() == 0) {
            (void) notify_veto_destroy((Notify_client)(LINT_CAST(client)));
            return(NOTIFY_DONE);
        }
    }
    return(notify_next_destroy_func(client, status));
}

void
quit_proc()
```

```
{
    if (confirm_quit()) {
        quit_confirmed_from_panel = 1;
        window_destroy(base_frame);
    }
}

msg(msg, beep)
    char *msg;
    int   beep;
{
    char    buf[300];
    int     result;
    Event   event;    /* unused */
    char    *contine_msg = "Press \"Continue\" to proceed.";

    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
            msg,
            contine_msg,
            0,
        ALERT_NO_BEEPING,  (beep) ? 0:1,
        ALERT_BUTTON_YES,  "Continue",
        ALERT_TRIGGER,     ACTION_STOP, /* allow either YES or NO answer */
        0);
    switch (result) {
        case ALERT_YES:
        case ALERT_TRIGGERED: /* result of ACTION_STOP trigger */
            break;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            printf("Alert failed, continuing anyway.\n");
            break;
    }
}
```

11

Advanced Imaging

11

Advanced Imaging

The chapter covers some topics on low level image maintenance. There is also a section on icon manipulation.

11.1. Handling Fixup

The routines `pw_read()`, `pw_copy()` and `pw_get()` may find themselves thwarted by trying to read from a portion of the pixwin which is hidden, and therefore has no pixels. This can happen with a canvas that you have made non-retained. When this happens, `pw_fixup` (a `struct rectlist`) in the `pixwin` structure will be filled in by the system with the description of the source areas which could not be accessed. The client must then regenerate this part of the image into the destination. Retained pixwins will always return `rl_null` in `pw_fixup` because the image is refreshed from the retained memory pixrect.

The usual strategy when calling `pw_copy()` is to call the following routine to restrict the pixwin's clipping to just that part of the image that needs to be fixed up.

```
pw_restrict_clipping(pw, rl)
    Pixwin   *pw;
    Rectlist *rl;
```

You pass in `&pw->pw_fixup` as `rl`. Now you draw your entire pixwin. Only the parts that need to be repaired are drawn. Now you need to reset your pixwin so that you may access its entire visible surface.

```
pw_exposed(pw)
    Pixwin *pw;
```

`pw_exposed()` is the call that does this.

Dealing with fixup for `pw_read()` or `pw_get()` is really quite ludicrous. One should really run these retained if they are using the screen as a storage medium for their bits.

11.2. Icons

The basic usage of icons is described in the *Icons* chapter of the *SunView Programmer's Guide*. The opaque type `Icon`, and the functions and attributes by which icons are manipulated, are defined in the header file `<suntool/icon.h>`.

Applications such as icon editors or browsers, which need to load icon images at run time, will need to use the functions described in this section. The definitions necessary to use these functions are contained in `<suntool/icon_load.h>`.

Loading Icons Dynamically

You can load an icon's image from a file with the call:

```
int
icon_load(icon, file, error_msg)
    Icon  icon;
    char *file, *error_msg;
```

`Icon` is an icon returned by `icon_create()`; `file` is the name of a file created with *iconedit*. `error_msg` is the address of a buffer (at least 256 characters long) into which `icon_load()` will write a message in the event of an error. If `icon_load()` succeeds, it returns zero; otherwise it returns 1.

The function

```
int
icon_init_from_pr(icon, pr)
    Icon     icon;
    Pixrect *pr;
```

initializes the width and height of the icon's graphics area (attribute `ICON_IMAGE_RECT`) to match the width and height of `pr`. It also initializes the icon's label (attribute `ICON_LABEL`) to `NULL`. The return value is meaningless.

To load an image from a file into a pixrect, use the routine:

```
Pixrect *
icon_load_mpr(file, error_msg)
    char *file, *error_msg;
```

This function allocates a pixrect, and loads it with the image contained in `file`. If no problem is encountered, `icon_load_mpr()` returns a pointer to the new pixrect containing the image. If it can't access or interpret the file, `icon_load_mpr()` writes a message into the buffer pointed to by `error_msg` and returns `NULL`.

Icon File Format

iconedit writes out an ASCII file consisting of two parts: a comment describing the image, and a list of hexadecimal constants defining the actual pixel values of the image. The contents of the file `<images/template.icon>` are reproduced below, as an example:

```
/* Format_version=1, Width=16, Height=16, Depth=1, Valid_bits_per_item=16
 * This file is the template for all images in the cursor/icon library.
 * The first line contains the information needed to properly interpret
 *   the actual bits, which are expected to be used directly by software
 *   that wants to do compile-time binding to an image via a #include.
 * The actual bits must be specified in hex.
 * The default interpretation of the bits below is specified by the
 *   behavior of mpr_static.
 * Note that Valid_bits_per_item uses the least-significant bits.
 * See also: icon_load.h.
 * Description: A cursor that spells "TEMPLATE" using two lines, with a
 *   solid bar at the bottom.
 * Background: White
 */
	0xED2F, 0x49E9, 0x4D2F, 0x4928, 0x4D28, 0x0000, 0x0000, 0x8676,
	0x8924, 0x8F26, 0x8924, 0xE926, 0x0000, 0x0000, 0xFFFF, 0xFFFF
```

The first line of the comment is composed of header parameters, used by the icon loading routines to properly interpret the actual bits of the image. The `format_version` exists to permit further development of the file format in a compatible manner, and should always be 1. Default values for the other header parameters are `Depth=1`, `Width=64`, `Height=64`, `Valid_bits_per_item=16`.

The remainder of the comment can be used for arbitrary descriptive material.

The following function is provided to allow you to preserve this material when rewriting an image file:

```
FILE *
icon_open_header(file, error_msg, info)
    char                *file, *error_msg;
    icon_header_handle  info;

typedef struct icon_header_object {
    int    depth,
           height,
           format_version,
           valid_bits_per_item,
           width;
    long   last_param_pos;
} icon_header_object;
```

`icon_open_header` fills in `info` from `file's` header parameters. `info->last_param_pos` is filled in with the position immediately after the last header parameter that was read. The `FILE *` returned by `icon_open_header()` is left positioned at the end of the header comment. Thus `ftell(icon_open_header())` indicates where the actual bits of the image should begin, and the characters in the range

```
[info->last_param_pos...ftell(icon_open_header()]
```

encompass all of the extra descriptive material contained in the file's header.

11.3. Damage

This section is included for those who can't use the Agent to hide all this complexity. Try to use the Agent, because it is very hard to get the following right.

When a portion of a client's window becomes visible after having been hidden, it is *damaged*. This may arise from several causes. For instance, an overlaying window may have been removed, or the client's window may have been stretched to give it more area. The client is notified that such a region exists either by the events `WIN_REPAINT` or `WIN_RESIZE`, or if the client is not using the Notifier, by the signal `SIGWINCH`; this simply indicates that something about the window has changed in a fashion that probably requires repainting. It is possible that the window has shrunk, and no repainting of the image is required at all, but this is a degenerate case. It is then the client's responsibility to *repair* the damage by painting the appropriate pixels into that area. The following section describes how to do that.

Handling a `SIGWINCH` Signal

There are several stages to handling a `SIGWINCH`. First, in almost all cases, the procedure that catches the signal should *not* immediately try to repair the damage indicated by the signal. Since the signal is a software interrupt, it may easily arrive at an inconvenient time, halfway through a window's repaint for some normal cause, for instance. Consequently, the appropriate action in the signal handler is usually to set a flag which will be tested elsewhere. Conveniently, a `SIGWINCH` is like any other signal; it will break a process out of a `select(2)` system call, so it is possible to awaken a client that was blocked, and with a little investigation, discover the cause of the `SIGWINCH`.

Once a process has discovered that a `SIGWINCH` has occurred and arrived at a state where it's safe to do something about it, it must determine exactly what has changed, and respond appropriately. There are two general possibilities: the window may have changed size, and/or a portion of it may have been uncovered.

`win_getsize()` (described in *Windows*) can be used to inquire the current dimensions of a window. The previous size must have been remembered, for instance from when the window was created or last adjusted. These two sizes are compared to see if the size has changed. Upon noticing that its size has changed, a window containing other windows may wish to rearrange the enclosed windows, for example, by expanding one or more windows to fill a newly opened space.

NOTE *If you are using* `window_main_loop()` *to drive your program, then the* `SIGWINCH` *is translated into a* `WIN_RESIZE` *and/or* `WIN_REPAINT` *event for you. The rest of this discussion still applies.*

Image Fixup

Whether a size change occurred or not, the actual images on the screen must be fixed up. It is possible to simply repaint the whole window at this point — that will certainly repair any damaged areas — but this is often a bad idea because it typically does much more work than necessary.

Therefore, the window should retrieve the description of the damaged area, repair that damage, and inform the system that it has done so: The `pw_damaged()` procedure:

```
pw_damaged(pw)
    Pixwin *pw;
```

is a procedure much like `pw_exposed()`. It fills in `pwcd_clipping` with a `rectlist` describing the area of interest, stores the id of that `rectlist` in the pixwin's `pw_opshandle` and in `pwcd_damagedid` as well. It also stores its own address in `pwco_getclipping`, so that a subsequent lock will check the correct `rectlist`. All the clippers are set up too. Colormap segment offset initialization is done, as described in *Surface Preparation.*

CAUTION **A call to `pw_damaged()` should ALWAYS be made in a sigwinch handling routine. Likewise, `pw_donedamaged()` should ALWAYS be called before returning from the sigwinch handling routine. While a program that runs on monochrome displays may appear to function correctly if this advice is not followed, running such a program on a color display will produce peculiarities in color appearance.**

Now is the time for the client to repaint its window — or at least those portions covered by the damaged `rectlist`; if the regeneration is relatively expensive, that is if the window is large, or its contents complicated, it may be worth restricting the amount of repainting *before* the clipping that the `rectlist` will enforce. This means stepping through the rectangles of the `rectlist`, determining for each what data contributed to its portion of the image, and reconstructing only that portion. See the chapter on rectlists for details about *rectlists.*

For retained pixwins, the following call can be used to copy the image from the backup pixrect to the screen:

```
pw_repairretained(pw)
    Pixwin *pw;
```

When the image is repaired, the client should inform the window system with a call to:

```
pw_donedamaged(pw)
    Pixwin *pw;
```

`pw_donedamaged()` allows the system to discard the `rectlist` describing this damage. It is possible that more damage will have accumulated by this time, and even that some areas will be repainted more than once, but that will be rare.

After calling `pw_donedamaged()`, the pixwin describes the entire visible area of the window.

A process which owns more than one window can receive a `SIGWINCH` for any of them, with no indication of which window generated it. The only solution is to fix up *all* windows. Fortunately, that should not be overly expensive, as only the appropriate damaged areas are returned by `pw_damaged()`.

11.4. Pixwin Offset Control

The following routines control the offset of a pixwin's coordinate space. They can be used for writing in a fixed coordinate space even though the pixwin moves about relative to the window's origin.

```
void
pw_set_x_offset(pw, offset)
    Pixwin *pw;
    int     offset;

void
pw_set_y_offset(pw, offset)
    Pixwin *pw;
    int     offset;

void
pw_set_xy_offset(pw, int x_offset, y_offset)
    Pixwin *pw;
    int     x_offset, y_offset;

int
pw_get_x_offset(pw)
    Pixwin *pw;

int
pw_get_y_offset(pw)
    Pixwin *pw;
```

12

Menus & Prompts

12

Menus & Prompts

This chapter describes routines that you will probably need when writing a menu or prompt package of your own. Note, however, that SunView's menu and alert facilities documented in the *SunView 1 Programmer's Guide* are both comprehensive full-featured packages that handle fullscreen access for you. Also note that you can use `window_loop()` together with one of the SunView window types to create other kinds of full-screen prompts, again without having to manage fullscreen interaction yourself.

12.1. Full Screen Access

To provide certain kinds of feedback to the user, it may be necessary to violate window boundaries. Pop-up menus, prompts and window management are examples of the kind of operations that do this. The *fullscreen* interface provides a mechanism for gaining access to the entire screen in a safe way. The package provides a convenient interface to underlying *sunwindow* primitives. The following structure is defined in `<suntool/fullscreen.h>`:

```
struct  fullscreen {
        int         fs_windowfd;
        struct      rect fs_screenrect;
        struct      pixwin *fs_pixwin;
        struct      cursor fs_cachedcursor;
        struct      inputmask fs_cachedim;   /* Pick mask */
        int         fs_cachedinputnext;
        struct      inputmask fs_cachedkbdim;   /* Kbd mask */
};
```

`fs_windowfd` is the window that created the `fullscreen` object. `fs_screenrect` describes the entire screen's dimensions. `fs_pixwin` is used to access the screen via the pixwin interface. The coordinate space of fullscreen access is the same as `fs_windowfd`'s. Thus, pixwin accesses are not necessarily done in the screen's coordinate space. Also, `fs_screenrect` is in the window's coordinate space. If, for example, the screen is 1024 pixels wide and 800 pixels high, `fs_windowfd` has its left edge at 300 and its top edge at 200, that is, both relative to the screen's upper left-hand corner, then `fs_screenrect` is {–300, –200, 1024, 800}.

The original cursor, `fs_cachedcursor`, input mask, `fs_cachedim`, and the window number of the input redirection window, `fs_cachedinputnext`, are cached and later restored when the fullscreen access object is destroyed.

Initializing Fullscreen Mode

```
struct fullscreen *
fullscreen_init(windowfd)
    int  windowfd;
```

gains full screen access for `windowfd` and caches the window state that is likely to be changed during the lifetime of the fullscreen object. `windowfd` is set to do blocking I/O. A pointer to this object is returned.

During the time that the full screen is being accessed, no other processes can access the screen, and all user input is directed to `fs->fs_windowfd`. Because of this, use fullscreen access infrequently and for only short periods of time.

Releasing Fullscreen Mode

```
fullscreen_destroy(fs)
    struct fullscreen *fs;
```

`fullscreen_destroy()` restores `fs`'s cached data, releases the right to access the full screen and destroys the fullscreen data object. `fs->fs_windowfd`'s input blocking status is returned to its original state.

Seizing All Inputs

Fullscreen access is built out of the grab I/O mechanism described here. This lower level is useful if you wanted to only grab input.

Normally, input events are directed to the window which underlies the cursor at the time the event occurs (or the window with the keyboard focus, if you have split pick/keyboard focus). Two procedures modify this state of affairs.

Grabbing I/O

A window may temporarily seize all inputs by calling:

```
win_grabio(windowfd)
    int  windowfd;
```

The caller's input mask still applies, but it receives input events from the whole screen; no window other than the one identified by `windowfd` will be offered an input event or allowed to write on the screen after this call.

Releasing I/O

```
win_releaseio(windowfd)
    int windowfd;
```

undoes the effect of a `win_grabio()`, restoring the previous state.

12.2. Surface Preparation

In order for a client to ignore the offset of his colormap segment the image of the pixwin must be initialized to the value of the offset. This *surface preparation* is done automatically by pixwins under the following circumstances:

- The routine `pw_damaged()` does surface preparation on the area of the pixwin that is damaged.
- The routine `pw_putcolormap()` does surface preparation over the entire exposed portion of a pixwin if a new colormap segment is being loaded for the first time.

For monochrome displays, nothing is done during surface preparation. For color displays, when the surface is prepared, the low order bits (colormap segment size minus 1) are not modified. This means that surface preparation does not clear the image. Initialization of the image (often clearing) is still the responsibility of client code.

There is a case in which surface preparation must be done explicitly by client code. When window boundaries are knowingly violated, as in the case of pop-up menus, the following procedure must be called to prepare each rectangle on the screen that is to be written upon:

```
pw_preparesurface(pw, rect)
    Pixwin *pw;
    Rect   *rect;
```

`rect` is relative to pw's coordinate system. Most commonly, a saved copy of the area to be written is made so that it can be restored later — see the next section.

Multiple Plane Groups

On machines with multiple plane groups (such as the Sun-3/110 and other machines using the `cgfour`(4S) frame buffer), `pw_preparesurface()` will correctly set up the enable plane so that the `rect` you are drawing in is visible. If you do not use `pw_preparesurface()`, it is possible that part of the area you are drawing on is displaying values from another plane group, so that part of your image will be occluded.

Pixel Caching

If your application violates window boundaries to put up fullscren menus and prompts, it is often desirable to remember the state of the screen before you drew on it and then repair it when you are finished. On machines with multiple plane groups such as the Sun-3/110 you need to restore the state of the enable plane and the bits in the other plane group(s). There are routines to help you do this.

Saving Screen Pixels

This routine saves the screen image where you area about to draw:

```
Pw_pixel_cache *
pw_save_pixels(pw, rect);
    Pixwin *pw;
    Rect   *rect;

typedef struct pw_pixel_cache {
        Rect rect;
        struct pixrect * plane_group[PIX_MAX_PLANE_GROUPS];
} Pw_pixel_cache;
```

`pw_save_pixels()` tries to allocate memory to store the contents of the pixels in `rect`. If it is unable to, it prints out a message on stderr and returns `PW_PIXEL_CACHE_NULL`. If it succeeds, it returns a pointer to a structure which holds the rect `rect` and an array of pixrects with the values of the pixels in `rect` in each plane group.

Restoring Screen Pixels

Then, when you have finished fullscreen access, you restore the image which you drew over with:

```
void
pw_restore_pixels(pw, pc);
    Pixwin         *pw;
    Pw_pixel_cache *pc;
```

`pw_restore_pixels()` restores the state of the screen where you drew. All the information it needs is in the `Pw_pixel_cache` pointer that `pw_save_pixels()` returned.

Fullscreen Drawing Operations

If you use `pw_preparesurface()`, you will be given a homogeneous area on which to draw during fullscreen access. However, for applications such as adjusting the size of windows (''rubber-banding''), you do not want to obscure what is underneath. On the other hand, on a machine with multiple plane groups you want your fullscreen access to be visible no matter what plane groups are being displayed.

The following routines perform the same vector drawing, raster operation and pixwin copying as their counterparts in *Imaging Facilities: Pixwins* in the *SunView 1 Programmer's Guide*. The difference is that these routines guarantee that the operation will happen in all plane groups so it will definitely be visible on-screen.

CAUTION **To save a lot of overhead, these routines make certain assumptions which must be followed.**

Anyone calling these `fullscreen_pw_*` routines must

- have called `fullscreen_init()`
- have not done any surface preparation under the pixels affected
- have not called `pw_lock()`
- use the fullscreen pixwin during this call
- use a `PIX_NOT(PIX_DST)` operation.

```
void
fullscreen_pw_vector(pw, x0, y0, x1, y1, op, value);
    Pixwin *pw;
    int     x0, y0, x1, y1, op, value;

void
fullscreen_pw_write(pw, xw, yw, width, height, op,
                    pr, xr, yr);
    Pixwin  *pw;
    int      xw, yw, width, height, op, xr, yr;
    Pixrect *pr;
```

```
void
fullscreen_pw_copy(pw, xw, yw, width, height, op,
                   pw_src, xr, yr);
    Pixwin *pw, *pw_src;
    int     xw, yw, width, height, op, xr, yr;
```

13

Window Management

13

Window Management

The window management routines provide the standard user interface presented by tool windows:

```
wmgr_open(framefd, rootfd)

wmgr_close(framefd, rootfd)

wmgr_move(framefd)

wmgr_stretch(framefd)

wmgr_top(framefd, rootfd)

wmgr_bottom(framefd, rootfd)

wmgr_refreshwindow(windowfd)
```

`wmgr_open()` opens a frame window from its iconic state to normal size. `wmgr_close()` closes a frame window from its normal size to its iconic size. `wmgr_move()` prompts the user to move a frame window or cancel the operation. If confirmed, the rest of the move interaction, including dragging the window and moving the bits on the screen, is done. `wmgr_stretch()` is like `wmgr_move()`, but it stretches the window instead of moving it. `wmgr_top()` places a frame window on the top of the window stack. `wmgr_bottom()` places the frame window on the bottom of the window stack. `wmgr_refreshwindow()` causes `windowfd` and all its descendant windows to repaint.

The routine `wmgr_changerect()`:

```
wmgr_changerect(feedbackfd, windowfd, event, move, noprompt)
    int     feedbackfd, windowfd;
    Event  *event;
    int     move, noprompt;
```

implements `wmgr_move()` and `wmgr_stretch()`, including the user interaction sequence. `windowfd` is moved (1) or stretched (0) depending on the value of `move`. To accomplish the user interaction, the input event is read from the `feedbackfd` window (usually the same as `windowfd`). The prompt is turned off if `noprompt` is 1.

```
int
wmgr_confirm(windowfd, text)
    int   windowfd;
    char *text;
```

`wmgr_confirm()` implements a layer over the prompt package for a standard confirmation user interface. `text` is put up in a prompt box. If the user confirms with a left mouse button press, then −1 is returned. Otherwise, 0 is returned. The up button event is not consumed.

NOTE *This routine is preserved only for backwards compatibility with versions of the SunOS prior to Release 4.0. You should use the new* package documented in the *SunView 1 Programmer's Guide* for dialogs with the user. `wmgr_confirm()` is not used for user interaction by SunView 1 packages unless the user has disabled *Alerts* in the *Compatibility* category of `defaultsedit(1)`.

Tool Invocation

```
wmgr_forktool(programname, otherargs, rectnormal, recticon,
              iconic)
    char *programname, *otherargs;
    Rect *rectnormal, *recticon;
    int   iconic;
```

is used to fork a new tool that has its normal rectangle set to `rectnormal` and its icon rectangle set to `recticon`. If `iconic` is not zero, the tool is created iconic. `programname` is the name of the file that is to be run and `otherargs` is the command line that you want to pass to the tool. A path search is done to locate the file. Arguments that have embedded white space should be enclosed by double quotes.

Utilities

The utilities described here are some of the low level routines that are used to implement the higher level routines. They may be used to put together a window management user interface different from that provided by tools. If a series of calls is to be made to procedures that manipulate the window tree, the whole sequence should be bracketed by `win_lockdata()` and `win_unlockdata()`, as described in *The Window Hierarchy*.

```
wmgr_completechangerect(windowfd, rectnew, rectoriginal,
                        parentprleft, parentprtop)
    int   windowfd;
    Rect *rectnew, *rectoriginal;
    int   parentprleft, parentprtop;
```

does the work involved with changing the position or size of a window's rect. This involves saving as many bits as possible by copying them on the screen so they don't have to be recomputed. `wmgr_completechangerect()` would be called after some programmatic or user action determined the new window position and size in pixels. `windowfd` is the window being changed. `rectnew` is the window's new rectangle. `rectoriginal` is the window's original rectangle. `parentprleft` and `parentprtop` are the upper-left screen coordinates of the parent of `windowfd`.

```
wmgr_winandchildrenexposed(pixwin, rl)
    Pixwin   *pixwin;
    Rectlist *rl;
```

computes the visible portion of `pixwin->pw_clipdata.pwcd_windowfd` and its descendants and stores it in `rl`. This is done by any window management routine that is going to try to preserve bits across window changes. For example, `wmgr_completechangerect()` calls `wmgr_winandchildrenexposed()` before and after changing the window size/position. The intersection of the two rectlists from the two calls are those bits that could possibly be saved.

```
wmgr_changelevel(windowfd, parentfd, top)
    int  windowfd, parentfd;
    bool top;
```

moves a window to the top or bottom of the heap of windows that are descendants of its parent. `windowfd` identifies the window to be moved; `parentfd` is the file descriptor of that window's parent, and `top` controls whether the window goes to the top (`TRUE`) or bottom (`FALSE`). Unlike `wmgr_top()` and `wmgr_bottom()`, no optimization is performed to reduce the amount of repainting. `wmgr_changelevel()` is used in conjunction with other window rearrangements, which make repainting unlikely. For example, `wmgr_close()` puts the window at the bottom of the window stack after changing its state.

```
#define  WMGR_ICONIC  WUF_WMGR1

wmgr_iswindowopen(windowfd)
    int  windowfd;
```

The user data of `windowfd` reflects the state of the window via the `WMGR_ICONIC` flag. `WUF_WMGR1` is defined in *<sunwindow/win_ioctl.h>* and `WMGR_ICONIC` is defined in *<suntool/wmgr.h>*. `wmgr_iswindowopen()` tests the `WMGR_ICONIC` flag and returns `TRUE` or `FALSE` as the window is open or closed.

Note that client programs should *never* set or clear the WMGR_ICONIC flag.

The `rootfd` window maintains a "next slot" position for both normal tool windows and icon windows in its unused iconic rect data. `wmgr_setrectalloc()` stores the next slot data and `wmgr_getrectalloc()` retrieves it:

```
wmgr_setrectalloc(rootfd, tool_left, tool_top,
                  icon_left, icon_top)
    int    rootfd;
    short  tool_left, tool_top, icon_left, icon_top;

wmgr_getrectalloc(rootfd, tool_left, tool_top,
                  icon_left, icon_top)
    int    rootfd;
    short *tool_left, *tool_top, *icon_left, *icon_top;
```

If you do a `wmgr_setrectalloc()`, make sure that all the values you are not changing were retrieved with `wmgr_getrectalloc()`. In other words, both procedures affect all the values.

13.1. Minimal Repaint Support

This is an extremely advanced subsection used only for those who might want to implement routines similar of the higher level window management routines mentioned above. This section has strong connections to the *Advanced Imaging* chapter and the chapter on *Rects and Rectlists*. Readers should refer to both from here.

Moving windows about on the screen may involve repainting large portions of their image in new places. Often, the existing image can be copied to the new location, saving the cost of regenerating it. Two procedures are provided to support this function:

```
win_computeclipping(windowfd)
    int windowfd;
```

causes the window system to recompute the *exposed* and *damaged* rectlists for the window identified by `windowfd` while withholding the `SIGWINCH` that will tell each owner to repair damage.

```
win_partialrepair(windowfd, r)
    int   windowfd;
    Rect *r;
```

tells the window system to remove the rectangle `r` from the damaged area for the window identified by `windowfd`. This operation is a no-op if `windowfd` has damage accumulated from a previous window database change, but has not told the window system that it has repaired that damage.

Any window manager can use these facilities according to the following strategy:

- The old exposed areas for the affected windows are retrieved and cached. (`pw_exposed()`)
- The window database is locked and manipulated to accomplish the rearrangement. (`win_lockdata()`, `win_remove()`, `win_setlink()`, `win_setrect()`, `win_insert()`,...)
- The new area is computed, retrieved, and intersected with the old. (`win_computeclipping()`, `pw_exposed()`, `rl_intersection()`)

- Pixels in the intersection are copied, and those areas are removed from the subject window's damaged area. (`pw_lock()`, `pr_copy()`, `win_partialrepair()`)
- The window database is unlocked, and any windows still damaged get the signals informing them of the reduced damage which must be repaired.

14

Rects and Rectlists

14

Rects and Rectlists

This chapter describes the geometric structures and operations SunView provides for doing rectangle algebra.

Images are dealt with in rectangular chunks. The basic structure which defines a rectangle is the `Rect`. Where complex shapes are required, they are built up out of groups of rectangles. The structure provided for this purpose is the `Rectlist`.

These structures are defined in the header files `<sunwindow/rect.h>` and `<sunwindow/rectlist.h>`. The library that provides the implementation of the functions of these data types is part of `/usr/lib/libsunwindow.a`.

14.1. Rects

The rect is the basic description of a rectangle, and there are macros and procedures to perform common manipulations on a rect.

```
#define coord short

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

The rectangle lies in a coordinate system whose origin is in the upper left-hand corner and whose dimensions are given in pixels.

Macros on Rects

The same header file defines some interesting macros on rectangles. To determine an edge not given explicitly in the rect:

```
#define rect_right(rp)
#define rect_bottom(rp)
Rect *rp;
```

returns the coordinate of the last pixel within the rectangle on the right or bottom, respectively.

Useful predicates returning `TRUE` or `FALSE`[32] are:

[32] `<sunwindow/rect.h>` defines `bool`, `TRUE` and `FALSE` if they are not already defined.

```
rect_isnull(r)                  /* r's width or height is 0 */
rect_includespoint(r,x,y)       /* (x,y) lies in r          */
rect_equal(r1, r2)              /* r1 and r2 coincide
                                 * exactly                  */
rect_includesrect(r1, r2)       /* every point in r2
                                 * lies in r1         */
rect_intersectsrect(r1, r2)     /* at least one point lies
                                 *  in both r1 and r2 */

    Rect  *r, *r1, *r2;
    coord  x, y;
```

Macros which manipulate dimensions of rectangles are:

```
rect_construct(r, x, y, w, h)
    Rect *r;
    int   x, y, w, h;
```

This fills in r with the indicated origin and dimensions.

```
rect_marginadjust(r, m)
    Rect *r;
    int   m;
```

adds a margin of m pixels on each side of r; that is, r becomes 2*m larger in each dimension.

```
rect_passtoparent(x, y, r)
rect_passtochild(x, y, r)
    coord  x, y;
    Rect   *r;
```

sets the origin of the indicated rect to transform it to the coordinate system of a parent or child rectangle, so that its points are now located relative to the parent or child's origin. x and y are the origin of the parent or child rectangle within *its* parent; these values are added to, or respectively subtracted from, the origin of the rectangle pointed to by r, thus transforming the rectangle to the new coordinate system.

Procedures and External Data for Rects

A null rectangle, that is one whose origin and dimensions are all 0, is defined for convenience:

```
extern struct rect rect_null;
```

The following procedures are also defined in rect.h:

```
Rect
rect_bounding(r1, r2)
    Rect *r1, *r2;
```

This returns the minimal rect that encloses the union of r1 and r2. The returned value is a struct, not a pointer.

```
rect_intersection(r1, r2, rd)
    Rect *r1, *r2, *rd;
```

computes the intersection of r1 and r2, and stores that rect into rd.

```
bool
rect_clipvector(r, x0, y0, x1, y1)
    Rect *r;
    int  *x0, *y0, *x1, *y1;
```

modifies the vector endpoints so they lie entirely within the rect, and returns FALSE if that excludes the whole vector, otherwise it returns TRUE.

NOTE *This procedure should not be used to clip a vector to multiple abutting rectangles. It may not cross the boundaries smoothly.*

```
bool rect_order(r1, r2, sortorder)
    Rect *r1, *r2;
    int   sortorder;
```

returns TRUE if r1 precedes or equals r2 in the indicated ordering:

```
#define RECTS_TOPTOBOTTOM       0
#define RECTS_BOTTOMTOTOP       1
#define RECTS_LEFTTORIGHT       2
#define RECTS_RIGHTTOLEFT       3
```

Two related defined constants are:

```
#define RECTS_UNSORTED  4
```

indicating a "don't-care" order, and

```
#define RECTS_SORTS     4
```

giving the number of sort orders available, for use in allocating arrays and so on.

14.2. Rectlists

A Rectlist is a structure that defines a list of rects. A number of rectangles may be collected into a list that defines an interesting portion of a larger rectangle. An equivalent way of looking at it is that a large rectangle may be fragmented into a number of smaller rectangles, which together comprise all the larger rectangle's interesting portions. A typical application of such a list is to define the portions of one rectangle remaining visible when it is partially obscured by others.

```
typedef struct rectlist {
    coord     rl_x, rl_y;
    Rectnode  *rl_head;
    Rectnode  *rl_tail,
    Rect      rl_bound;
} Rectlist;

typedef struct rectnode {
    Rectnode  *rn_next;
    Rect      rn_rect;
} Rectnode;
```

Each node in the rectlist contains a rectangle which covers one part of the visible whole, along with a pointer to the next node. rl_bound is the minimal bounding rectangle of the union of all the rectangles in the node list. All rectangles in the rectlist are described in the same coordinate system, which may be translated efficiently by modifying rl_x and rl_y.

The routines that manipulate rectlists do their own memory management on rectnodes, creating and freeing them as necessary to adjust the area described by the rectlist.

Macros and Constants Defined on Rectlists

Macros to perform common coordinate transformations are provided:

```
rl_rectoffset(rl, rs, rd)
    Rectlist *rl;
    Rect     *rs, *rd;
```

copies rs into rd, and then adjusts rd's origin by adding the offsets from rl.

```
rl_coordoffset(rl, x, y)
    Rectlist *rl;
    coord     x, y;
```

offsets x and y by the offsets in rl. For instance, it converts a point in one of the rects in the rectnode list of a rectlist to the coordinate system of the rectlist's parent.

Parallel to the macros on rect's, we have:

```
rl_passtoparent(x, y, rl)
rl_passtochild(x, y, rl)
    coord     x, y;
    Rectlist *rl;
```

which add or subtract the given coordinates from the rectlist's rl_x and rl_y to convert the rl into its parent's or child's coordinate system.

Procedures and External Data for Rectlists

An empty rectlist is defined, which should be used to initialize any rectlist before it is operated on:

```
extern struct rectlist rl_null;
```

Procedures are provided for useful predicates and manipulations. The following declarations apply uniformly in the descriptions below:

```
Rectlist *rl, *rl1, *rl2, *rld;
Rect      *r;
coord      x, y;
```

Predicates return TRUE or FALSE. Refer to the following table for specifics.

Table 14-1 *Rectlist Predicates*

Macro	*Returns* TRUE *if*
`rl_empty(rl)`	Contains only null rects
`rl_equal(rl1, rl2)`	The two rectlists describe the same space identically — same fragments in the same order
`rl_includespoint(rl,x,y)`	(x,y) lies within some rect of `rl`
`rl_equalrect(r, rl)`	`rl` has exactly one rect, which is the same as `r`
`rl_boundintersectsrect(r, rl)`	Some point lies both in `r` and in `rl`'s bounding rect

Manipulation procedures operate through side-effects, rather than returning a value. Note that it is legitimate to use a rectlist as both a source and destination in one of these procedures. The source node list will be freed and reallocated appropriately for the result. Refer to the following table for specifics.

Table 14-2 *Rectlist Procedures*

Procedure	*Effect*
`rl_intersection(rl1, rl2, rld)`	Stores into `rld` a rectlist which covers the intersection of `rl1` and `rl2`.
`rl_union(rl1, rl2, rld)`	Stores into `rld` a rectlist which covers the union of `rl1` and `rl2`.
`rl_difference(rl1, rl2, rld)`	Stores into `rld` a rectlist which covers the area of `rl1` not covered by `rl2`.
`rl_coalesce(rl)`	Attempts to shorten `rl` by coalescing some of its fragments. An `rl` whose bounding rect is completely covered by the union of its node rects will be collapsed to a single node; other simple reductions will be found; but the general solution to the problem is not attempted.
`rl_sort(rl, rld, sort)` `int sort;`	`rl` is copied into `rld`, with the node rects arranged in `sort` order.
`rl_rectintersection(r, rl, rld)`	`rld` is filled with a rectlist that covers the intersection of `r` and `rl`.
`rl_rectunion(r, rl, rld)`	`rld` is filled with a rectlist that covers the union of `r` and `rl`.
`rl_rectdifference(r, rl, rld)`	`rld` is filled with a rectlist that covers the portion of `rl` which is not in `r`.
`rl_initwithrect(r, rl)`	Fills in `rl` so that it covers the rect `r`.
`rl_copy(rl, rld)`	Fills in `rld` with a copy of `rl`.
`rl_free(rl)`	Frees the storage allocated to `rl`.
`rl_normalize(rl)`	Resets `rl`'s offsets (`rl_x`, `rl_y`) to be 0 after adjusting the origins of all rects in `rl` accordingly.

15

Scrollbars

15

Scrollbars

Canvases, text subwindows and panels have been designed to work with scrollbars. The text subwindow automatically creates its own vertical scrollbar. For canvases and panels, it is your responsibility to create the scrollbar and pass it in via the attributes `WIN_VERTICAL_SCROLLBAR` or `WIN_HORIZONTAL_SCROLLBAR`.

The chapter on scrollbars in the *SunView Programmer's Guide* covers what most applications need to know about scrollbars.

The material in this chapter will be of interest only if you are writing an application not based on canvases, text subwindows or panels, and you need to communicate with the scrollbar directly as events are received. This chapter is directed to programmers writing software which receives scroll-request events and implements scrolling.

The definitions necessary to use scrollbars are found in the header file `<suntool/scrollbar.h>`

15.1. Basic Scrollbar Management

Registering as a Scrollbar Client

The scrollbar receives events directly from the Notifier. The user makes a scroll request by releasing a mouse button over the scrollbar. The scrollbar's job is to translate the button-up event into a scrolling request event, and send this event, via the Notifier, to its client.

To receive scrolling request events, the client must register itself with the scrollbar via the `SCROLL_NOTIFY_CLIENT` attribute. For example, `panel_1` would register as a client of `bar_1` with the call:

```
scrollbar_set(bar_1, SCROLL_NOTIFY_CLIENT, panel_1, 0);
```

NOTE *Before registering with the scrollbar, the client must register with the Notifier by calling* `win_register()`.

In most applications, such as the panel example above, the client and the scrolling object are identical. However, they may well be distinct. In such a case, if the client wants the scrollbar to keep track of which object the scrollbar is being used with, the client has to inform the scrollbar explicitly of the object which is to be scrolled. This is done by setting the `SCROLL_OBJECT` attribute.

For example, in the text subwindow package, the text subwindow is the client to be notified. Within a given text subwindow there may be many views onto the underlying file. Each of these views has its own scrollbar. So each scrollbar created by the text subwindow will have the text subwindow as `SCROLL_NOTIFY_CLIENT` and the particular view as `SCROLL_OBJECT`. So to create scrollbars for two views, the text subwindow package would call:

```
scrollbar_set(bar_1,
              SCROLL_NOTIFY_CLIENT, textsubwindow_1,
              SCROLL_OBJECT,        view_1,
              0);

scrollbar_set(bar_2,
              SCROLL_NOTIFY_CLIENT, textsubwindow_1,
              SCROLL_OBJECT,        view_2,
              0);
```

Keeping the Scrollbar Informed

The visible portion of the scrolling object is called the *view* into the object The scrollbar displays a *bubble* representing both the location of the view within the object and the size of the view relative to the size of the object. In order to compute the size and location of the bubble, and to compute the new offset into the object after a scroll, the scrollbar needs to know the current lengths of both the object and the view.

The client must keep the scrollbar informed by setting the attributes `SCROLL_OBJECT_LENGTH` and `SCROLL_VIEW_LENGTH`. There are two obvious strategies for when to update this information. You can ensure that the scrollbar is always up-to-date by informing it whenever the lengths in question change. If this is too expensive (because the lengths change too frequently) you can update the scrollbar only when the cursor enters the scrollbar.

This strategy of updating the scrollbar when it is entered can be implemented as follows. When the scrollbar gets a `LOC_RGNENTER` or `LOC_RGNEXIT` event, it causes the event-handling procedure of its notify client to be called with a `SCROLL_ENTER` or `SCROLL_EXIT` event. The client then catches the `SCROLL_ENTER` event and updates the scrollbar, as in the example below. (Note that the scrollbar handle to use for the `scrollbar_set()` call is passed in as `arg`).

```
Notify_value
panel_event_proc(client, event, arg, type)
    caddr_t            client;
    Event              *event;
    Notify_arg         arg;
    Notify_event_type  type;
{
    switch (event_id(event)) {
      ...
      case SCROLL_ENTER:
          scrollbar_set((Scrollbar)arg,
              SCROLL_OBJECT_LENGTH, current_obj_length,
              SCROLL_VIEW_START,    current_view_start,
              SCROLL_VIEW_LENGTH,   current_view_length,
              0);
          break;
      ...
    }
}
```

The client can interpret the values of SCROLL_OBJECT_LENGTH, SCROLL_VIEW_LENGTH and SCROLL_VIEW_START in whatever units it wants to. For example, the panel package uses pixel units, while the text subwindow package uses character units.

Handling the SCROLL_REQUEST Event

When the user requests a scroll, the scrollbar client's event-handling procedure gets called with an event whose event code is SCROLL_REQUEST. The event proc is passed an argument (arg in the example below) for event-specific data. In the case of scrollbar-related events, arg is a scrollbar handle (type Scrollbar). As in the example below, the client's event proc must switch on the SCROLL_REQUEST event and call a procedure to actually perform the scroll:

```
Notify_value
panel_event_proc(panel, event, arg, type)
    Panel              panel;
    Event              *event;
    Notify_arg         arg;
    Notify_event_type  type;
{
    switch (event_id(event)) {
      ...
      case SCROLL_REQUEST:
        do_scroll(panel, (Scrollbar)arg);
        break;
      ...
    }
}
```

Performing the Scroll

The new offset into the scrolling object is computed by the Scrollbar Package, and is available to the client via the attribute SCROLL_VIEW_START. The client's job is to paint the object starting at the new offset, and to paint the scrollbar reporting the scroll, so that its bubble will be updated to reflect the new offset. So in the simplest case the client's scrolling routine would look something like this:

```
do_scroll(sb)
    Scrollbar sb;
{
    unsigned new_view_start;

    /* paint scrollbar to show bubble in new position */
    scrollbar_paint(sb);

    /* get new offset into object from scrollbar */
    new_view_start = (unsigned) scrollbar_get(sb,
                                    SCROLL_VIEW_START);

    /* client's proc to paint object at new offset */
    paint_object(sb->object, new_view_start);
}
```

If the client has both a horizontal and a vertical scrollbar, it will probably be necessary to distinguish the direction of the scroll, as in:

```
do_scroll(sb)
    Scrollbar sb;
{
    /* paint the scrollbar to show bubble in new position */
    scrollbar_paint(sb);

    /* get new offset into object from scrollbar */

    /*
     * pass the new offset and the direction of the scrollbar
     * into the paint function
     */
    paint_object(sb->object,
                scrollbar_get(sb, SCROLL_VIEW_START),
                scrollbar_get(sb, SCROLL_DIRECTION));
}
```

In order to repaint the screen efficiently, you need to know which bits appear on the screen both before and after the scroll, and thus can be copied to their new location with pw_copy(). To compute the copyable region you will need, in addition to the current offset into the object (SCROLL_VIEW_START), the offset prior to the scroll (SCROLL_LAST_VIEW_START).

Note: you are responsible for repainting the scrollbar after a scroll, with one of the routines described later in *Painting Scrollbars*.

Normalizing the Scroll

The scrollbar package can be utilized in two modes: *normalized* and *un-normalized.* Un-normalized means that when the user makes a scrolling request, it is honored exactly to the pixel, as precisely as resolution permits. In normalized scrolling, the client makes an attempt to put the display in some kind of "normal form" after the scrolling has taken place.

To take panels as an example, this simply means that after a vertical scroll, the Panel Package modifies the value of `SCROLL_VIEW_START` so that the highest item which is either fully or partially visible in the panel is placed with its top edge `SCROLL_MARGIN` pixels from the top of the panel.

Normalization is enabled by setting the `SCROLL_NORMALIZE` attribute for the scrollbar to `TRUE`, and the `SCROLL_MARGIN` attribute to the desired margin. `SCROLL_NORMALIZE` defaults to `TRUE`, and `SCROLL_MARGIN` defaults to four pixels.

Note that the scrollbar package simply keeps track of whether the scrolls it computes are intended to be normalized or not. The client who receives the scroll-request event is responsible for asking the scrollbar whether normalization should be done, and if so, doing it.

After the client computes the normalized offset, it must update the scrollbar by setting the attribute `SCROLL_VIEW_START` to the normalized offset.

Painting Scrollbars

The basic routine to paint a scrollbar is:

```
scrollbar_paint(scrollbar);
    Scrollbar scrollbar;
```

`scrollbar_paint()` repaints only those portions of the scrollbar (page buttons, bar proper, and bubble) which have been modified since they were last painted. To clear and repaint all portions of the bar, use `scrollbar_paint_clear()`.

In addition, the routines `scrollbar_paint_bubble()` and `scrollbar_clear_bubble()` are provided to paint or clear the bubble only.

15.2. Advanced Use of Scrollbars

As indicated previously under *Performing the Scroll*, the client need not be concerned with the details of the scroll request at all — he may simply use the new offset given by the value of the `SCROLL_VIEW_START` attribute. However, the client may want to assume partial or full responsibility for the scroll. He may compute the new offset from scratch himself, or start with the offset computed by the Scrollbar Package and modify it so as not to have text or other information clipped at the top of the window (see the preceding discussion under *Normalizing the Scroll*).

In order to give you complete control over the scroll, attributes are provided to allow you to retrieve all the information about the scroll-request event and the object's state at the time of the event. The attributes of interest include:

Table 15-1 *Scroll-Related Scrollbar Attributes*

Attribute	*Value Type*	*Description*
`SCROLL_LAST_VIEW_START`	int	Offset of view into object prior to scroll. Get only.
`SCROLL_OBJECT`	caddr_t	pointer to the scrollable object.
`SCROLL_OBJECT_LENGTH`	int	Length of scrollable object, in client units (value must be >= 0). Default: 0.
`SCROLL_REQUEST_MOTION`	Scroll_motion	Scrolling motion requested by user.
`SCROLL_REQUEST_OFFSET`	int	Pixel offset of scrolling request into scrollbar. Default: 0.
`SCROLL_VIEW_LENGTH`	int	Length of viewing window, in client units. Default: 0.
`SCROLL_VIEW_START`	int	Current offset into scrollable object, measured in client units. (Value must be >= 0). Default: 0.

Types of Scrolling Motion in Simple Mode

There are three basic types of scrolling motion:

- `SCROLL_ABSOLUTE`. This is the "thumbing" motion requested by the user with the middle button. You can retrieve the number of pixels into the scrollbar of the request (including the page button which may be present) via `SCROLL_REQUEST_OFFSET`.
- `SCROLL_FORWARD`. This is to be interpreted as a request to bring the location of the cursor to the top (left, if horizontal).
- `SCROLL_BACKWARD`. This is to be interpreted as a request to bring the top (left, if horizontal) point to the cursor.

The function which implements scrolling may want to switch on the scrolling motion, to implement different algorithms for each motion. In the following example, `do_absolute_scroll()`, `do_forward_scroll()`, `do_backward_scroll()` and `paint_object()` are procedures written by the client:

```
do_scroll(sb)
    Scrollbar sb;
{
    unsigned new_offset;
    Scroll_motion motion;

    motion = (Scroll_motion) scrollbar_get(sb, SCROLL_MOTION);

    switch (motion) {

      case SCROLL_ABSOLUTE:
         new_offset = do_absolute_scroll(sb);
         break;

      case SCROLL_FORWARD:
         new_offset = do_forward_scroll(sb);
         break;

      case SCROLL_BACKWARD:
         new_offset = do_backward_scroll(sb);
         break;
    }

    /* tell the scrollbar of the new offset */
    scrollbar_set(sb, SCROLL_VIEW_START, new_offset, 0);

    /* paint scrollbar to show bubble in new position */
    scrollbar_paint(sb);

    /* client's repainting proc */
    paint_object(scrollbar_get(sb, SCROLL_OBJECT, 0);
}
```

Types of Scrolling Motion in Advanced Mode

Internally, the scrollbar package distinguishes nine different types of motion, depending on which mouse button the user pressed, the state of the shift key, and the whether the cursor was in the bar, the forward page button or the backward page button. Normally, these motions are mapped onto the three basic motions described above. In order to perform this mapping, the scrollbar package needs to know the distance between lines. You do this by setting the `SCROLL_LINE_HEIGHT` attribute, as in:

```
scrollbar_set(sb, SCROLL_LINE_HEIGHT, 20, 0);
```

NOTE *This is the distance, in pixels, from the top of one line to the top of the succeeding line.*

The scrollbar package can also be used in *advanced mode*, in which case the mapping described above is not performed -- the motion is passed as is to the notify proc. This allows you to interpret each motion exactly as you want.

The following table gives the nine motions, the user action which generates them, and the basic motions which they are mapped onto (if not in advanced mode):

Table 15-2 *Scrollbar Motions*

Motion	*Generated By*	*Mapped to*
ABSOLUTE	middle in bar	ABSOLUTE
POINT_TO_MIN	left in bar	FORWARD
MAX_TO_POINT	shifted left in bar	FORWARD
PAGE_FORWARD	middle in page button	FORWARD
LINE_FORWARD	left in page button	FORWARD
MIN_TO_POINT	right in bar	BACKWARD
POINT_TO_MAX	shifted right in bar	BACKWARD
PAGE_BACKWARD	shifted middle in page button	BACKWARD
LINE_BACKWARD	right in page button	BACKWARD

To operate in advanced mode you must:

- set the attribute SCROLL_ADVANCED_MODE to TRUE.
- Switch on the nine motions in the above table. Note: specifically, SCROLL_FORWARD and SCROLL_BACKWARD must *not* appear in your switch statement. In other words, for basic mode switch on the three basic motions; for advanced mode switch on the nine advanced motions.

A

Writing a Virtual User Input Device Driver

A

Writing a Virtual User Input Device Driver

This section describes what a device driver needs to do in order to conform to the Virtual User Input Device (vuid) interface understood by SunView. This is not a tutorial on writing a device driver; only the vuid related aspects of device driver writing are covered.

A.1. Firm Events

A stream of *firm events* is what your driver is expected to emit when called through the `read()` system call. This stream is simply a byte stream that encodes `Firm_event` structures. A firm event is a structure comprising an ID which indicates what kind of event it is, the value of the event, and a time when this event occurred; it also carries some information that allows the event's eventual consumer to maintain the complete state of its input system.

The `Firm_event` Structure

The `Firm_event` structure is defined in `<sundev/vuid_event.h>`:

```
typedef struct firm_event {
    u_short              id;
    u_char               pair_type;
    u_char               pair;
    int                  value;
    struct timeval       time;
} Firm_event;

#define FE_PAIR_NONE           0
#define FE_PAIR_SET            1
#define FE_PAIR_DELTA          2
#define FE_PAIR_ABSOLUTE       3
```

Here is what the fields in the `Firm_event` mean:

- `id` — is the event's unique identifier. It is either the id of an existing vuid event (if you're trying to emulate part of the vuid) or your one of your own creation (see *Choosing VUID Events*).
- `value` — is the event's value. It is often 0 (up) or 1 (down). For valuators it is a 32 bit integer.
- `time` — is the event's time stamp, i.e., when it occurred. The time stamp is not defined to be meaningful except to compare with other `Firm_event` time stamps. In the kernel, a call to `uniqtime()`, which takes a pointer to a `struct timeval`, gets you a close-to-current unique time. In user

process land, a call to `gettimeofday(2)` gets time from the same source (but it is not made unique).

Pairs

This brings us to `pair_type` and `pair`. These two fields enable a consumer of events to maintain input state in an event-independent way. The pair field is critical for a input state maintenance package, one that is designed to not know anything about the semantics of particular events, to maintain correct data for corresponding absolute, delta and paired state variables. Some examples will make this clear:

- Say you have a tablet emitting absolute locations. Depending on the client, what the absolute location is may be important (say for digitizing) and then again the difference between the current location and the previous location may be of interest (say for computing acceleration while tracking a cursor).
- Say you are keyboard in which the user has typed ^C. Your driver first emits a SHIFT_CTRL event as the control key goes down. Next your driver emits a ^C event (one of the events from the ASCII vuid segment) as the c key goes down. Now the application that you are driving happens to be using the c key as a shift key in some specialized application. The application wants to be able to say to SunView (the maintainer of the input state), ''Is the c key down?'' and get a correct response.

The vuid supports a notion of updating a companion event at the same time that a single event is generated. In the first situations above, the tablet wants to be able to update companion absolute and relative event values with a single event. In the second situations above, the keyboard wants to be able to update companion ^C and c event values with a single event. The vuid supports this notion of updating a companion event in such a way as to be independent from these two particular cases. `pair_type` defines the type of the companion event:

- `FE_PAIR_NONE` — is the common case in which `pair` is not defined, i.e., there is no companion.
- `FE_PAIR_SET` — is used for ASCII controlled events in which `pair` is the uncontrolled *base* event, e.g., ^C and 'c' or 'C', depending on the state of the shift key. The use of this pair type is not restricted to ASCII situations. This pair type simply says to set the *pair*th event in `id`'s vuid segment to be `value`.
- `FE_PAIR_DELTA` — identifies `pair` as the delta companion to `id`. This means that the `pair`th event in `id`'s vuid segment should be set to the delta of `id`'s current value and `value`. One should always create vuid valuator events as delta/absolute pairs. For example, the events LOC_X_DELTA and LOC_X_ABSOLUTE are pairs and the events LOC_Y_DELTA and LOC_Y_ABSOLUTE are pairs. These events are part of the standard WORKSTATION_DEVID segment that define *the* distinguish primary locator motion events.
- `FE_PAIR_ABSOLUTE` — identifies `pair` as the absolute companion to `id`. This means that the `pair`th event in `id`'s vuid segment should be set to the sum of `id`'s current value and `value`. One should always create vuid

valuator events as delta/absolute pairs.

As indicated by the previous discussion, `pair` must be in the same vuid segment as `id`.

Choosing VUID Events

One needs to decide which events the driver is going to emit. If you want to emulate the Sun virtual workstation then you want to emit the same events as the WORKSTATION_DEVID vuid segment. A tablet, for example, can emit absolute locator positions LOC_X_ABSOLUTE and LOC_Y_ABSOLUTE, instead of a mouses relative locator motions LOC_X_DELTA and LOC_Y_DELTA. SunView will uses these to drive the mouse.

If you have a completely new device then you want to create a new vuid segment. This is talked about in the workstations chapter of the *SunView System Programmer's Guide*.

A.2. Device Controls

A vuid driver is expected to respond to a variety of device controls.

Output Mode

Many of you will be starting from an existing device driver that already speaks its own native protocol. You may not want to flush this old protocol in favor of the vuid protocol. In this case you may want to operate in both modes. VUID*FORMAT `ioctls` are used to control which byte stream format that an input device should emit.

```
#define VUIDSFORMAT    _IOW(v, 1, int)
#define VUIDGFORMAT    _IOR(v, 2, int)

#define VUID_NATIVE     0
#define VUID_FIRM_EVENT 1
```

VUIDSFORMAT sets the input device byte stream format to one of:

- VUID_NATIVE — The device's native byte stream format (it may be vuid).
- VUID_FIRM_EVENT — The byte stream format is `Firm_events`.

An errno of `ENOTTY` or `EINVAL` indicates that a device can't speak `Firm_events`.

VUIDSFORMAT gets the input device byte stream format.

Device Instancing

VUID*ADDR ioctls are used to control which address a particular virtual user input device segment has. This is used to have an instancing capability, e.g., a second mouse. One would:

- Take the current mouse driver, which emits events in the WORKSTATION_DEVID vuid segment.
- Define a new vuid segment, say LOC2_DEVID.
- Add LOC2_X_ABSOLUTE, LOC2_Y_ABSOLUTE, LOC2_X_DELTA and LOC2_Y_DELTA to the LOC2_DEVID vuid segment at the same offset from the beginning of the segment as LOC_X_ABSOLUTE, LOC_Y_ABSOLUTE, LOC_X_DELTA and LOC_Y_DELTA in the WORKSTATION_DEVID.

- Command a mouse to emit events using LOC2_DEVID's segment address and the mouse's original low byte segment offsets. Thus, it would be emitting LOC2_X_DELTA and LOC2_Y_DELTA, which is what your application would eventually receive.

Here is the VUID*ADDR commands common data structure and command definitions:

```
typedef struct  vuid_addr_probe {
        short   base;
        union   {
                short   next;
                short   current;
        } data;
} Vuid_addr_probe;

#define VUIDSADDR    _IOW(v, 3, struct  vuid_addr_probe)
#define VUIDGADDR    _IOWR(v, 4, struct  vuid_addr_probe)
```

VUIDSADDR is used to set an alternative vuid segment. `base` is the vuid device addr that you are changing. A vuid device addr is the vuid segment id shifted into it's high byte position. `data.next` is the new vuid device addr that should be used instead of `base`. An errno of `ENOTTY` indicates that a device can't deal with these commands. An errno of `ENODEV` indicates that the requested virtual device has no events generated for it by this physical device.

VUIDGADDR is used to get an current value of a vuid segment. `base` is the default vuid device addr that you are asking about. `data.current` is the current vuid device addr that is being used instead of `base`.

The implementation of these `ioctls` is optional. If you don't do it then your device wouldn't be able to support multiple instances.

Input Controls

Your device needs to support non-blocking reads in order to run with SunView 3.0. This means that the `read`(2) system call returns `EWOULDBLOCK` when no input is available.

In addition, your driver should support the `select`(2) system call and asynchronous input notification (sending SIGIO when input pending). However, your driver will still run without these two things in 3.0 SunView.

A.3. Example

The following example is parts of code taken from the Sun 3.0 mouse driver. It illustrates some of the points made above.

```
/* Copyright (c) 1985 by Sun Microsystems, Inc.  */
```

<elided material>

```
#include "../sundev/vuid_event.h"

/*
 * Mouse select management is done by utilizing the tty mechanism.
 * We place a single character on the tty raw input queue whenever
```

```
 * there is some amount of mouse data available to be read.  Once,
 * all the data has been read, the tty raw input queue is flushed.
 *
 * Note: It is done in order to get around the fact that line
 * disiplines don't have select operations because they are always
 * expected to be ttys that stuff characters when they get them onto
 * a queue.
 *
 * Note: We use spl5 for the mouse because it is functionally the
 * same as spl6 and the tty mechanism is using spl5.  The original
 * code that was doing its own select processing was using spl6.
 */
#define spl_ms  spl5

/* Software mouse registers */
struct  ms_softc {
        struct  mousebuf {
                short   mb_size;        /* size (in mouseinfo units) of buf */
                short   mb_off;         /* current offset in buffer */
                struct  mouseinfo {
                        char    mi_x, mi_y;
                        char    mi_buttons;
#define MS_HW_BUT1              0x4     /* left button position */
#define MS_HW_BUT2              0x2     /* middle button position */
#define MS_HW_BUT3              0x1     /* right button position */
                        struct  timeval mi_time; /* timestamp */
                } mb_info[1];           /* however many samples */
        } *ms_buf;
        short   ms_bufbytes;            /* buffer size (in bytes) */
        short   ms_flags;               /* currently unused */
        short   ms_oldoff;              /* index into mousebuf */
        short   ms_oldoff1;             /* at mi_x, mi_y or mi_buttons... */
        short   ms_readformat;          /* format of read stream */
#define MS_3BYTE_FORMAT VUID_NATIVE     /* 3 byte format (buts/x/y) */
#define MS_VUID_FORMAT  VUID_FIRM_EVENT /* vuid Firm_event format */
        short   ms_vuidaddr;            /* vuid addr for MS_VUID_FORMAT */
        short   ms_vuidcount;           /* count of unread firm events */
        short   ms_samplecount;         /* count of unread mouseinfo samples */
        char    ms_readbuttons;         /* button state as of last read */
};

struct  msdata {
        struct  ms_softc msd_softc;
        struct  tty *msd_tp;
```

<elided material>

```
};
struct  msdata msdata[NMS];
struct  msdata *mstptomsd();
```

<elided material>

```
/* Open a mouse.  Calls sets mouse line characteristics */
/* ARGSUSED */
msopen(dev, tp)
        dev_t dev;
        struct  tty *tp;
{
        register int err, i;
        struct sgttyb sg;
        register struct mousebuf *b;
        register struct ms_softc *ms;
        register struct msdata *msd;
        caddr_t zmemall();
        register struct cdevsw *dp;

        /* See if tp is being used to drive ms already.  */
        for (i = 0;i < NMS; ++i)
                if (msdata[i].msd_tp == tp)
                        return(0);
        /* Get next free msdata */
        for (i = 0;i < NMS; ++i)
                if (msdata[i].msd_tp == 0)
                        goto found;
        return(EBUSY);
found:
        /* Open tty */
        if (err = ttyopen(dev, tp))
                return(err);
        /* Setup tty flags */
        dp = &cdevsw[major(dev)];
        if (err = (*dp->d_ioctl) (dev, TIOCGETP, (caddr_t)&sg, 0))
                goto error;
        sg.sg_flags = RAW+ANYP;
        sg.sg_ispeed = sg.sg_ospeed = B1200;
        if (err = (*dp->d_ioctl) (dev, TIOCSETP, (caddr_t)&sg, 0))
                goto error;
        /* Set up private data */
        msd = &msdata[i];
        msd->msd_xnext = 1;
        msd->msd_tp = tp;
        ms = &msd->msd_softc;
        /* Allocate buffer and initialize data */
        if (ms->ms_buf == 0) {
                ms->ms_bufbytes = MS_BUF_BYTES;
                b = (struct mousebuf *)zmemall(memall, ms->ms_bufbytes);
                if (b == 0) {
                        err = EINVAL;
                        goto error;
                }
                b->mb_size = 1 + (ms->ms_bufbytes-sizeof (struct mousebuf))
                                 / sizeof (struct mouseinfo);
                ms->ms_buf = b;
                ms->ms_vuidaddr = VKEY_FIRST;
                msflush(msd);
```

```
        }
        return (0);
error:
        bzero((caddr_t)msd, sizeof (*msd));
        bzero((caddr_t)ms, sizeof (*ms));
        return (err);
}

/*
 * Close the mouse
 */
msclose(tp)
        struct  tty *tp;
{
        register struct msdata *msd = mstptomsd(tp);
        register struct ms_softc *ms;

        if (msd == 0)
                return;
        ms = &msd->msd_softc;
        /* Free mouse buffer */
        if (ms->ms_buf != NULL)
                wmemfree((caddr_t)ms->ms_buf, ms->ms_bufbytes);
        /* Close tty */
        ttyclose(tp);
        /* Zero structures */
        bzero((caddr_t)msd, sizeof (*msd));
        bzero((caddr_t)ms, sizeof (*ms));
}

/*
 * Read from the mouse buffer
 */
msread(tp, uio)
        struct  tty *tp;
        struct uio *uio;
{
        register struct msdata *msd = mstptomsd(tp);
        register struct ms_softc *ms;
        register struct mousebuf *b;
        register struct mouseinfo *mi;
        register int error = 0, pri, send_event, hwbit;
        register char c;
        Firm_event fe;

        if (msd == 0)
                return(EINVAL);
        ms = &msd->msd_softc;
        b = ms->ms_buf;
        pri = spl_ms();
        /*
         * Wait on tty raw queue if this queue is empty since the tty is
         * controlling the select/wakeup/sleep stuff.
```

```
	 */
	while (tp->t_rawq.c_cc <= 0) {
		if (tp->t_state&TS_NBIO) {
			(void) splx(pri);
			return (EWOULDBLOCK);
		}
		sleep((caddr_t)&tp->t_rawq, TTIPRI);
	}
	while (!error && (ms->ms_oldoff1 || ms->ms_oldoff != b->mb_off)) {
		mi = &b->mb_info[ms->ms_oldoff];
		switch (ms->ms_readformat) {

		case MS_3BYTE_FORMAT:
<elided material>
			break;

		case MS_VUID_FORMAT:
			if (uio->uio_resid < sizeof (Firm_event))
				goto done;
			send_event = 0;
			switch (ms->ms_oldoff1++) {

			case 0: /* Send x if changed */
				if (mi->mi_x != 0) {
					fe.id = vuid_id_addr(ms->ms_vuidaddr) |
					    vuid_id_offset(LOC_X_DELTA);
					fe.pair_type = FE_PAIR_ABSOLUTE;
					fe.pair = LOC_X_ABSOLUTE;
					fe.value = mi->mi_x;
					send_event = 1;
				}
				break;

			case 1: /* Send y if changed */
				if (mi->mi_y != 0) {
					fe.id = vuid_id_addr(ms->ms_vuidaddr) |
					    vuid_id_offset(LOC_Y_DELTA);
					fe.pair_type = FE_PAIR_ABSOLUTE;
					fe.pair = LOC_Y_ABSOLUTE;
					fe.value = -mi->mi_y;
					send_event = 1;
				}
				break;

			default:/* Send buttons if changed */
				hwbit = MS_HW_BUT1 >> (ms->ms_oldoff1 - 3);
				if ((ms->ms_readbuttons & hwbit) !=
				     (mi->mi_buttons & hwbit)) {
					fe.id = vuid_id_addr(ms->ms_vuidaddr) |
					    vuid_id_offset(
						BUT(1) + (ms->ms_oldoff1 - 3));
					fe.pair_type = FE_PAIR_NONE;
					fe.pair = 0;
```

```
                                        /* Update read buttons and set value */
                                        if (mi->mi_buttons & hwbit) {
                                                fe.value = 0;
                                                ms->ms_readbuttons |= hwbit;
                                        } else {
                                                fe.value = 1;
                                                ms->ms_readbuttons &= ~hwbit;
                                        }
                                        send_event = 1;
                                }
                                /* Increment mouse buffer pointer */
                                if (ms->ms_oldoff1 == 5) {
                                        ms->ms_oldoff++;
                                        if (ms->ms_oldoff >= b->mb_size)
                                                ms->ms_oldoff = 0;
                                        ms->ms_oldoff1 = 0;
                                }
                                break;

                        }
                        if (send_event) {
                                fe.time = mi->mi_time;
                                ms->ms_vuidcount--;
                                /* lower pri to avoid mouse droppings */
                                (void) splx(pri);
                                error = uiomove(&fe, sizeof(fe), UIO_READ, uio);
                                /* spl_ms should return same priority as pri */
                                pri = spl_ms();
                        }
                        break;

                }
        }
done:
        /* Flush tty if no more to read */
        if ((ms->ms_oldoff1 == 0) && (ms->ms_oldoff == b->mb_off))
                ttyflush(tp, FREAD);
        /* Release protection AFTER ttyflush or will get out of sync with tty */
        (void) splx(pri);
        return (0);
}

/* Mouse ioctl */
msioctl(tp, cmd, data, flag)
        struct  tty *tp;
        int cmd;
        caddr_t data;
        int flag;
{
        register struct msdata *msd = mstptomsd(tp);
        register struct ms_softc *ms;
        int     err = 0, num;
        register int buf_off, read_off;
```

```
	Vuid_addr_probe *addr_probe;

	if (msd == 0)
		return(EINVAL);
	ms = &msd->msd_softc;
	switch (cmd) {
	case FIONREAD:
		switch (ms->ms_readformat) {
		case MS_3BYTE_FORMAT:
			*(int *)data = ms->ms_samplecount;
			break;

		case MS_VUID_FORMAT:
			*(int *)data = sizeof (Firm_event) * ms->ms_vuidcount;
			break;
		}
		break;

	case VUIDSFORMAT:
		if (*(int *)data == ms->ms_readformat)
			break;
		ms->ms_readformat = *(int *)data;
		/*
		 * Flush mouse buffer because otherwise ms_*counts
		 * get out of sync and some of the offsets can too.
		 */
		msflush(msd);
		break;

	case VUIDGFORMAT:
		*(int *)data = ms->ms_readformat;
		break;

	case VUIDSADDR:
		addr_probe = (Vuid_addr_probe *)data;
		if (addr_probe->base != VKEY_FIRST) {
			err = ENODEV;
			break;
		}
		ms->ms_vuidaddr = addr_probe->data.next;
		break;

	case VUIDGADDR:
		addr_probe = (Vuid_addr_probe *)data;
		if (addr_probe->base != VKEY_FIRST) {
			err = ENODEV;
			break;
		}
		addr_probe->data.current = ms->ms_vuidaddr;
		break;

	case TIOCSETD:
		/*
```

```
                 * Don't let the line discipline change once it has been set
                 * to a mouse.  Changing the ldisc causes msclose to be called
                 * even if the ldisc of the tp is the same.
                 * We can't let this happen because the window system may have
                 * a handle on the mouse buffer.
                 * The basic problem is one of having anything depending on
                 * the continued existence of ldisc related data.
                 * The fix is to have:
                 * 1) a way of handing data to the dependent entity, and
                 * 2) notifying the dependent entity that the ldisc
                 * has been closed.
                 */
                break;
```

(<elided material>

```
        default:
                err = ttioctl(tp, cmd, data, flag);
        }
        return (err);
}

msflush(msd)
        register struct msdata *msd;
{
        register struct ms_softc *ms = &msd->msd_softc;
        int s = spl_ms();
```

<elided material>

```
        ttyflush(msd->msd_tp, FREAD);
        (void) splx(s);
}
```

<elided material>

```
/* Called with next byte of mouse data */
/*ARGSUSED*/
msinput(c, tp)
        register char c;
        struct tty *tp;
{
        int s = spl5();
```

<elided material>

```
        /* Place data on circular buffer */

        if (wake)
                /* Place character on tty raw input queue to trigger select */
                ttyinput('\0', msd->msd_tp);
        (void) splx(s);
}
```

```
/* Match tp to msdata */
struct  msdata *
mstptomsd(tp)
        struct  tty *tp;
{
        register i;

        /* Get next free msdata */
        for (i = 0;i < NMS; ++i)
                if (msdata[i].msd_tp == tp)
                        return(&msdata[i]);
        printf("mstptomsd called with unknown tp %X\n", tp);
        return(0);
}
```

B

Programming Notes

B

Programming Notes

Here are useful hints for programmers who use SunView.

B.1. What Is Supported?

In each release, there may be some difference between the documentation and the actual product implementation. The documentation describes the supported implementation. In general, the documentation indicates where features are only partially implemented, and in which directions future extensions may be expected. Any necessary modifications to SunView are accompanied by a description of the nature of the changes and appropriate responses to them.

B.2. Library Loading Order

When loading programs, remember to load higher level libraries first, that is, **-lsuntool -lsunwindow -lpixrect**.

B.3. Shared Libraries vs. Shared Text

Starting with release 3.2 of SunOS, the tools in SunView were distributed as two huge *toolmerge* files in order to reduce the working set requirements of SunView. Instead of having many different programs running, each including its own copy of the `libsuntool.a`, `libsunwindow.a` and `libpixrect.a` libraries, several copies of one large program would run at once.

Shared Libraries

With the advent of shared libraries in SunOS release 4.0, toolmerging is unnecessary in most cases. By default, each tool is compiled and linked to use shared libraries, so the code of the SunView 1 libraries is still shared.[33] This also has the benefit of greatly reducing the size of SunView 1 binaries. Another benefit is that programs will automatically benefit from improvements in the SunView 1 libraries in each release, without recompilation.

You may find old programs or `Makefiles` with references to `-DMERGE` or `-DSTANDALONE` in them.

Should you want to compile a program with static libraries, just supply the `-Bstatic` flag to the link editor or C compiler (read the `ld(1)` man page for details). One reason to use static libraries is that it simplifies part of the debugging process.

[33] The standard C library and many others are shared also.

B.4. Error Message Decoding

The default error reporting scheme described in Section 5.12, *Error Handling*, displays a long hexadecimal number which is the `ioctl` number associated with the error. You can turn this number into a more meaningful operation name by:

- turning the two least significant digits into a decimal number;
- searching `/usr/include/sunwindow/win_ioctl.h` for occurrences of this number; and
- noting the `ioctl` operation associated with this number.

This can provides a quick hint as to what is being complained about without resorting to a debugger.

B.5. Debugging Hints

When debugging non-terminal oriented programs in the window system, there are some things that you should know to make things easier.

As discussed mentioned in passing a process receives a `SIGWINCH` whenever one of its windows changes state. In particular, as soon as a frame is shown, the kernel sends it a `SIGWINCH`. When running as the child of a debugger, the `SIGWINCH` is sent to the parent debugger instead of to the tool. By default, `dbx` simply propagates the `SIGWINCH` to the tool, while `adb` traps, leaving the tool suspended until the user continues from `adb`. This behavior is not peculiar to `SIGWINCH`: `adb` traps all signals by default, while `dbx` has an initial list of signals (including `SIGWINCH`) that are passed on to the child process. You can instruct `adb` to pass `SIGWINCH` on to the child process by typing **`1c:i`** [Return]. `1c` is the hex number for 28, which is `SIGWINCH`'s number. Re-enable signal breaking by typing **`1c:t`** [Return]. You can instruct `dbx` to trap on a signal by using the `catch` command.

For further details, see the entries for the individual debuggers in the *User's Manual for the Sun Workstation*. In addition, `ptrace`(2) describes the fine points of how kernel signal delivery is modified while a program is being debugged.

The two debuggers differ also in their abilities to interrupt programs built using tool windows. `dbx` knows how to interrupt these programs, but `adb` doesn't. See *Signals from the Control Terminal* below for an explanation.

Disabling Locking

Another situation specific to the window system is that various forms of locking are done that can get in the way of smooth debugging while working at low levels of the system. There are variables in the *sunwindow* library that disable the actual locking. These variables can be turned on from a debugger:

Table B-1 *Variables for Disabling Locking*

Variable/Action	
`int pixwindebug`	When not zero this causes the immediate release of the display lock after locking so that the debugger is not continually getting hung by being blocked on writes to screen. Display garbage can result because of this action.
`int win_lockdatadebug`	When not zero, the data lock is never actually locked, preventing the debugger from being continually hung during block writes to the screen. Unpredictable things may result because of this action that can't properly be described in this context.
`int win_grabiodebug`	When not zero will not actually acquire exclusive I/O access rights so that the debugger wouldn't get hung by being blocked on writes to the screen and not be able to receive input. The debugged process will only be able to do normal display locking and be able to get input only in the normal way.
`int fullscreendebug`	Like `win_grabiodebug` but applies to the fullscreen access package.

Change these variables only during debugging. You can set them any time after `main` has been called.

B.6. Sufficient User Memory

To use the SunView environment comfortably requires adequate user memory for SunView and SunOS, Sun's operating system. To achieve the best performance, **you must** reconfigure your own kernel, deleting unused device drivers and possibly reducing some tuning parameters. The procedure is documented in the manual *Installing the SunOS*. You will be able to reclaim a significant amount of usable main memory.

B.7. Coexisting with UNIX

This section discusses how a SunView tool interacts with traditional UNIX features in the areas of process groups, signal handling, job control and terminal emulation. If you are not familiar with these concepts, read the appropriate portions (*Process Groups, Signals*) of the *System Services Overview*, and the `signal(3)` and `tty(4)` entries in the *SunOS Reference Manual.*

This discussion explicitly notes those places where the shells and debuggers interact differently with a tool.

Tool Initialization and Process Groups

System calls made by the library code in a tool affect the signals that will be sent to the tool. A tool acts like any program when first started: it inherits the process group and control terminal group from its parent process. However, when a frame is created, the tool changes its process group to its own process number. The following sections describe the effects of this change.

Signals from the Control Terminal

When the C shell (see `csh(1)`) starts a program, it changes the process group of the child to the child's process number. In addition, if that program is started in the foreground, the C shell also modifies the process group of the control terminal to match the child's new process group. Thus, if the tool was started from the C shell, the process group modification done by `window_create()` has no effect.

The Bourne Shell (see `sh(1)`) and the standard debuggers do not modify their child's process and control terminal groups. Furthermore, both the Bourne Shell and `adb(1)` are ill-prepared for the child to perform such modification. They do not propagate signals such as `SIGINT` to the child because they assume that the child is in the same control terminal group as they are. The bottom-line is that when a tool is executed by such a parent, typing interrupt characters at the parent process does not affect the child, and vice versa. For example, if the user types an interrupt character at `adb` while it is debugging a tool, the tool is not interrupted. Although `dbx(1)` does not modify its child's process group, it is prepared for the child to do so.

Job Control and the C Shell

The terminal driver and C shell job control interact differently with tools. First, let us examine what happens to programs using the graphics subwindow library package[34] When the user types an interrupt character on the control terminal, a signal is sent to the executing program. When the signal is a `SIGTSTP`, the `gfxsw` library code sees this signal and releases any SunView locks that it might have and removes the graphics from the screen before it actually suspends the program. If the program is later continued, the graphics are restored to the screen.

However, when the user types the C shell's `stop` command to interrupt the executing program, the C shell sends a `SIGSTOP` to the program and the `gfxsw` library code has no chance to clean up. This causes problems when the code has acquired any of the SunView locks, as there is no opportunity to release them. Depending on the lock timeouts, the kernel will eventually break the locks, but

[34] The gfxsubwindow is an out-dated package used only as an example.

until then, the entire screen is unavailable to other programs and the user. To avoid this problem, the user should send the C shell `kill` command with the **–TSTP** option instead of using `stop`.

The situation for tools parallels that of the `gfxsw` code. Thus a tool that wants to interact nicely with job control must receive the signals related to job control >L SIGINT , (`SIGQUIT`, and `SIGTSTP`) and release any locks it has acquired. If the tool is later continued, the tool must receive a `SIGCONT` so that it can reacquire the locks before resuming the window operations it was executing. The tool will still be susceptible to the same problems as the `gfxsw` code when it is sent a `SIGSTOP`.

A final note: the user often relies on job control without realizing it; the expectation is that typing interrupt characters will halt a program. Of course, even programs that do not use SunView facilities, such as a program that opens the terminal in ''raw'' mode, have to provide a way to terminate the program. A program using the `gfxsw` package that reads any input can provide limited job control by calling `gfxsw_inputinterrupts`.

Index

Notes

Notes

Notes

Notes

Notes