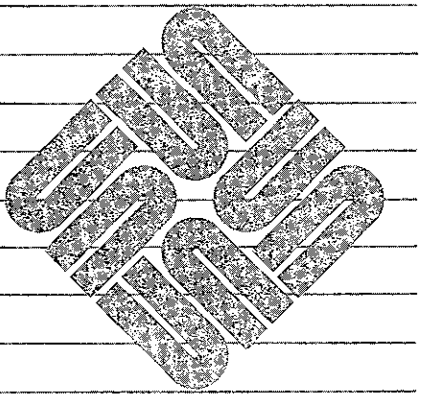
sun
microsystems

Programmer's Reference Manual *for* SunWindows

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Programmer's Reference Manual *for* SunWindows



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1167-01
Revision E of 15 May, 1985

Acknowledgements

A preliminary implementation of the Sun Window System was written at Sun Microsystems, Inc. in December 1982 and January 1983. It incorporated a number of low-level operations and data, including raster operations and fonts, provided by Tom Duff of Lucasfilm, Inc. The present version is a major rework of the preliminary implementation, aimed at generality, extensibility, and reliability.

Trademarks

Revision History

Rev	Date	Comments
A	15 July 1983	Preliminary draft of this Programmer's Reference Manual
B	15 September 1983	0.9 release of this Programmer's Reference Manual
C	1 November 1983	Additions to pixrect creation, input handling, and tool facilities
D	7 January 1984	Many corrections, additions, changes, and deletions to user interface, option subwindow, graphics subwindow, and window manager; changes to sunwindow library to accomodate color and multiple screens, and to the pixrect library to support color pixrectss
E	19 November 1984	2.0 α release of this Programmer's Reference Manual.
F	1 February 1985	2.0 β release. Addition of panel subwindow package. Removal of option subwindow package to Appendix F in favor of panel package; addition of Appendix G showing how to convert programs that use option subwindows to make use of the panel package. Addition of Appendix E on how to write a pixrect driver.
G	15 April 1985	2.0 final release. Many corrections and minor changes.

Contents

Contents

Tables

Figures

Preface

The *Programmer's Reference Manual for SunWindows* provides primarily reference material on SunWindows, the Sun window system. It is intended for programmers of applications using window system facilities.

Manual Contents

The contents of the manual are:

Chapter 1 — *Overview* — Describes basic hardware and software support and the layers of implementation of SunWindows, the ***pixrect*** layer, the ***sunwindow*** layer, and the ***suntool*** layer.

Chapter 2 — *Pixel Data and Operations* — Describes pixel data and operations in the lowest level output facilities of SunWindows, pixrects, pixrectops, memory pixrects, and text facilities for pixrects.

Chapter 3 — *Overlapped Windows: Imaging Facilities* — Explains image generation on windows which may overlap other windows.

Chapter 4 — *Window Manipulation* — Describes the ***sunwindow*** layer facilites for creating, positioning, and controlling windows.

Chapter 5 — *Input to Application Programs* — Discusses how user input is made available to application programs.

Chapter 6 — *Suntool: Tools and Subwindows* — Discusses how to write a tool, and covers creation and destruction of a tool and its subwindows, the strategy for dividing work among subwindows, and the use of routines provided to accomplish that work.

Chapter 7 — *Suntool: Subwindow Packages* — Discusses ***subwindows*** as building blocks in the construction of a tool, covers the currently existing subwindows, and suggests the approach for creating new kinds of subwindows.

Chapter 8 — *The Panel Subwindow Package* — Describes the use of panels, which are subwindows that present information and choices to the application user.

Chapter 9 — *Suntool: User Interface Utilities* Covers user interface utilities, the independent packages for use with the ***suntools*** environment, includes the actual window manipulation routines used by ***tool windows***, the ***icon*** facility, the ***selection*** manager, the ***fullscreen*** access mechanism, and ***menus*** and ***prompts***.

Appendix A — *Rects and Rectlists* — Describes the geometric structures used with the ***sunwindow*** layer and provides a full description of the operations on these structures.

Appendix B — *Sample Tools* — Provides an annotated collection of some simple tools to be used both as illustrations and as templates for client programmers.

Appendix C — *Sample Graphics Programs* — Provides an annotated selection of several graphics programs for writing your own graphics programs; includes code for a bouncing ball demonstration and for a "movie camera" program that displays files as frames from a movie.

Appendix D — *Programming Notes* — Contains useful hints for programmers using the SunWindows library procedures.

Appendix E — *Writing a Pixrect Driver* — Explains how to to construct a device driver for a pixel-addressable device so that it will provide Sun's device-independent interface to the frame buffer.

Apendix F — *Option Subwindow* — Describes a subwindow that implements a type of user interface to application programs. The material here is being phased out; programmers are encouraged to use the panel subwindow instead.

Note: This manual is neither a user guide nor an explanation of the internals of the window system. It presents the material in a bottom-up fashion with primitive concepts and facilities described first. It is not intended to be read linearly front-to-back; glance at the table of contents and the chapters on tools to get a general idea of how to use the rest of the material.

The ***Programmer's Tutorial to SunWindows*** supplies the basics needed to build SunWindows tools.

The ***User's Manual for the Sun Workstation*** provides user information under ***suntools***(1) for SunWindows and under the appropriate entry for the particular application programs. The ***Beginner's Guide to the Sun Workstation*** provides a brief tutorial on general use of the mouse and the SunWindows pop-up menus.

A Note About Special Terms

Several terms in this manual have meanings distinct from their common definitions or introduce concepts that are specific to programming in the SunWindows environment. We discuss the most important here.

The word ***client*** indicates a program that uses window system facilities. This is in contrast to ***user***, which refers to a human.

Terms referring to display hardware, such as ***framebuffer***, ***pixel***, and ***rasterop***, are used in well-established senses; novices who are confused should consult one of the standard texts, such as ***Fundamentals of Interactive Computer Graphics*** by J.D. Foley and A. Van Dam, Addison-Wesley, 1983.

The position of the mouse is indicated by a ***cursor*** on the screen; this is any small image that moves about the screen in response to mouse motions. The term "cursor" is used elsewhere to indicate the location at which type-in will be inserted, or other editor functions performed. The two concepts are not often distinguished. To keep them distinct, we use the term ***caret*** to refer to the type-in location.

A ***menu*** is a list of related choice items displayed on the screen in response to a user mouse-action. The user chooses one menu item by pointing at it with the cursor. Such menus are called ***transient*** or ***pop-up***; they are displayed only while a mouse button is depressed, and are typically used for invoking parameterless operations.

A ***rect*** is a structure that defines a rectangle.

A ***rectlist*** is a structure that defines a list of rects.

Up-down encoded keyboards are devices from which it is possible to receive two distinct signals when a key is pressed and then released.

An ***icon*** is a small form of a window that typically displays an identifying image rather than a portion of the window contents; it is frequently used for dormant application programs. For example, the default icon for a closed Shell Tool is a likeness of a CRT terminal.

Chapter 1

Overview

1.1. What is SunWindows?

SunWindows is the Sun window system. It is a ***tool box and parts kit***, not a closed, finished, end product. Its design emphasizes extensibility, accessibility at multiple layers, and provision of appropriate parts and development tools. Specific applications are provided here both as examples and because they are valuable for further development. The system is designed to be expanded by clients.

The system is explicitly ***layered*** with interfaces at several levels for client programs. There is open access to lower levels, and also convenient and powerful facilities for common requirements at higher levels. For instance, it is always possible for a client to write directly to the screen, although in most circumstances it is preferable to employ higher-level routines.

1.2. Hardware and Software Support

The Sun Microsystems Workstation provides hardware and software support for the construction of high-quality user interfaces. Hardware features include:

- provision of a processor for each user, a prerequisite for powerful, responsive, cost-effective systems;
- a bit-mapped display which allows arbitrary fonts and graphics to be used freely to make applications programs easier to learn and use;
- hardware support of fast and convenient manipulation of image data;
- a mouse pointing device for selecting operations from menus or for pointing at text, graphics and icons; and
- an up-down encoded keyboard that supports sophisticated function-key interfaces at once simpler and more efficient than most command languages.

Sun software is similarly structured to support high-quality interactions. The software features are:

- a uniform interface to varied pixel-oriented devices that allows convenient incorporation of new devices into the system, and clean access to all these devices by application programs;
- extended device independence for input such as function keys and locators, as well as for other user-interface features;

- a window management facility that keeps track of multiple overlapping windows, allowing their creation and rearrangement at will. The facility arbitrates screen access, detects destructive interactions such as overlapping, and initiates repairs. It also serializes and distributes user inputs to the multiple windows, allowing full type-ahead and mouse-ahead; and
- built on all these facilities, an executive and application environment that provides a system for running existing UNIX programs and new applications, taking advantage of icons, menus, prompts, mouse-driven selections, interprocess data exchange, a forms-oriented interface and useful cursor manipulations.

1.3. Layers of Implementation

There are three broad ***layers*** of SunWindows. These layers may be identified by the libraries that contain their implementations. The organization of the reference part of this manual reflects the three layers as described below.

1. The ***pixrect*** level provides a device-independent interface to pixel operations.
2. The ***sunwindow***[1] level implements a manager for overlapping windows, including imaging control, creation and manipulation of windows, and distribution of user inputs.
3. The ***suntool*** level implements a multi-window executive and application environment. In its user interface, it includes a number of relatively independent packages, supporting, for instance, ***menus*** and ***selections***.

1.3.1. Pixrect Layer

Chapter 2 describes the ***pixrect*** layer of the system. This level generalizes RasterOp display functions to arbitrary rectangles of pixels. Peculiarities of specific pixel-oriented devices, such as dimensions, addressing schemes, and pixel size and interpretation, are encapsulated in device-specific implementations, which all present the same uniform interface to clients.

The concept of a pixrect is quite general; it is convenient for referring to a whole display, as well as to the image of a single character in a font. It may also be used to describe the image which tracks the mouse.

There is a balance between functionality and efficiency. All pixrects clip operations that extend beyond their boundaries. Since this may require substantial overhead, clients which can guarantee to stay within bounds may disable this feature. Where hardware support exists, it is taken advantage of without sacrificing generality: all pixrects support the same set of operations on their contents.

These operations include general raster operations on rectangular areas, vectors, batch operations to handle common applications like text, and compact manipulation of constant or regularly-patterned data. A stencil operation provides spatial, two-dimensional masking of the source pixrect with a mask pixrect to control the areas of the destination pixrect to be written.

Color pixrects, as well as monochrome pixrects, are supported. There are uniform operations for accessing a pixrect's ***colormap***. A colormap maps a pixel value to a screen color. The pixel

---

[1] **Note that the term 'sunwindow' refers to the layer or level of implementation while the word 'SunWindows' is the name of the Sun window system.**

planes affected by other operations can be controlled as well. Monochrome pixrects support the same interface as color pixrects. Programs intended primarily for color pixrects usually produce reasonable images on monochrome pixrects, and vice versa.

1.3.2. Sunwindow Layer

Chapters 3 through 5 introduce ***windows*** and operations on them. A window is a rectangular display area, along with the process or processes responsible for its contents. This layer of the system maintains a database of windows which may ***overlap*** in both time and space. These windows may be nested, providing for distinct ***subwindows*** within an application's screen space.

Windows may be created, destroyed, moved, stretched or shrunk, set at different levels in the overlapping structure, and otherwise manipulated. The ***sunwindow*** level of the system provides facilities for performing all these operations. It also allows definition of the image which tracks the mouse while it is in the window, and inquiry and control over the mouse position.

Windows existing concurrently may all access a display; the window system provides locking primitives to guarantee that these accesses do not conflict.

Arbitration between windows is also provided in the allocation of display space. Where one window limits the space available to another, it is necessary to provide ***clipping***, so neither interferes with the other's image. One such conflict handled by the ***sunwindow*** layer arises when windows share the same coordinates on the display: one ***overlaps*** the other.

When one window impacts another window's image without any action on the second window's part, SunWindows informs the affected window of the damage it has suffered, and the areas that ought to be repaired. Windows may either recompute their contents for redisplay, or they may elect to have a full backup of their image in main memory, and merely copy the backup to the display when required.

On color displays, colormap entries are a scarce resource. When shared among multiple applications, they become even more scarce. Arbitration between windows is provided in the allocation of colormap entries. Provisions are made to share portions of the colormap.

Separate collections of windows may reside on separate screens. The user interacts with these multiple screens with his single keyboard and mouse.

User inputs are unified into a single stream at this level, so that actions with the mouse and keyboard can be coordinated. This unified stream is then distributed to different windows, according to user or programmatic indications. Windows may be selective about which input events they will process, and rejected events will be offered to other windows for processing. This enables terminal-based programs to run within windows which will handle mouse interactions for them.

1.3.3. Suntool Layer

Chapters 6 through 9 of this manual describe the ***suntool*** level of the system. While the first two layers provide client interfaces, the ***suntool*** level provides the user interface.

We refer to an application program that is a client of this level of the window system as a ***tool***. This term covers the one or more programs and processes which do the actual application processing. It also refers to the collection of windows through which the tool interacts with the user. This collection often includes a special ***icon***, which is a small form the tool may take to be unobtrusive on the screen but still identifiable. Some examples of tools are a calculator, a bitmap editor, and a terminal emulator. Sun provides a few ready-built tools, several of which are illustrated in Appendix B. Customers can develop their own tools to suit their specific needs.

SunWindows provides some common components of tools:

- an executive framework that supplies the usual "main loop" of a program and coordinates the activities of the various subwindows;
- a standard ***tool window*** that frames the active windows of the tool, identifying it with a name stripe at the top and borders around the subwindows. Each tool window has a facility for manipulating itself in the overlapped window environment. This includes adjusting its size and position, including layering, and moving the boundaries between subwindows;
- several commonly used ***subwindow*** types that can be instantiated in the tool;
- a standard scheme for laying out those subwindows; and
- a facility that provides a default ***icon*** for the tool.

The ***suntools*** program initializes the window environment. It provides for:

- automatic startup of a specified collection of tools;
- dynamic invocation of standard tools;
- management of the default window called the ***root*** window, which underlies all the tools; and
- the user interface for leaving the window system.

Users who wish some other form of environment management can replace the ***suntools*** program, while retaining the tools and supporting utilities.

The facilities provided in the ***suntool*** library are relatively independent; they can be used with window contexts other than ***suntools***. The ***icon*** facility mentioned above is in this category, as are the window manipulation facilities of ***suntools***. There is also a package for presenting ***menus*** to the user and interpreting the response.

Chapter 2

Pixel Data and Operations

This chapter discusses pixel data and operations in the lowest-level output facilities of SunWindows. These facilities will frequently be accessed indirectly, through higher-level abstractions described in chapters 3 through 9. However, some client implementors will deal at this level, for instance to include new display devices in the window system. The header file `<pixrect/pixrect_hs.h>` includes the header files that you need to work at this level of the window system. It will also suffice to include `<suntool/suntool_hs.h>` or `<sunwindow/sunwindow_hs.h>`.

2.1. Pixrects

The fundamental object of pixel manipulation in the window system is the ***pixrect***. A pixrect encapsulates a rectangular array of pixels along with the operations which are defined on that data. Pixrects are designed along the model of ***objects*** in an object-oriented programming system. They combine both data and operations, presenting their clients with a simple interface: a well-defined set of operations produces desired results, and details of representation and implementation are hidden inside the object.

The pixrect presents only its dimensions, a pointer to its operations, and a pointer to private data which those operations may use in performing their tasks. Further, the set of operations is the same across all pixrects, though of course their implementations must differ. This object-oriented style allows similar things which differ in small details to be gathered into a unified framework; it allows clients to use the same approach to all of them, and allows implementors to add new members or improve old ones without disturbing clients.

The pixrect facility satisfies two broad objectives:

- To provide a ***uniform interface to a variety of devices*** for independence from device characteristics where they are irrelevant. Such characteristics include the actual device (pixrects may exist in memory and on printers as well as on displays), the dimensions and addressing schemes of the device, and the definition of the pixels, that is, how many bits in each, how they are aligned, and how interpreted. Color and monochrome devices use the same interface. Programs intended primarily for color pixrects usually produce reasonable images on monochrome pixrects, and vice versa.
- To provide a proper ***balance of functionality and efficiency*** for a full range of pixel operations with performance close to that achieved by direct access to the hardware. Pixrect operations include generalized rasterops, vectors, text and other batch operations, compact manipulation of uniform and regularly-patterned data, as well as single-pixel reads and writes. All provide for clipping to the bounds of the rectangle if desired; this facility may be bypassed by clients which can perform it more efficiently themselves. A stencil function provides spatial masking of the source pixrect with a stencil pixrect to control the areas of the destination pixrect to be written. Where specialized hardware exists and can be used for a particular operation, it is

used, but not at the expense of violating the device-independent interface.

2.1.1. Pixels: Coordinates and Interpretation

Pixels in a pixrect are addressed in two dimensions with the origin in the upper left corner, and x and y increasing to the right and down. The coordinates of a pixel in a pixrect are integers from 0 to the pixrect's width or height minus 1.

A pixrect is characterized by a ***depth***, the number of bits required to hold one pixel. A large class of displays uses a single bit to select black or white (or green or orange, depending on the display technology). On these ***monochrome*** displays and in memory pixrects one bit deep, a 1 indicates ***foreground*** and a 0 ***background***. No further interpretation is applied to memory. The default interpretation on Sun displays is a white background and a black foreground.

Other displays use several bits to identify a color or gray level. Typically, though not necessarily, the pixel value is used as an index into a ***colormap***, where colors may be defined with higher precision than in the pixel. A common arrangement is to use an 8-bit pixel to choose one of 256 colors, each of which is defined in 24 bits, 8 each of red, green and blue. Memory pixrect depths of 1, 8, 16, and 24 are supported. Frame buffer pixrects are either 1 bit or 8 bits (color) per pixel. You can write depth 1 or 8 pixrects to a color frame buffer.

2.1.2. Geometry Structs

As a preliminary to the discussion of pixrects, it is convenient to define a few structs which contain useful geometric information.

The struct that defines a position in coordinates (x, y) is:

```
struct pr_pos  {
        int x, y;
};
```

Leaving a pixrect undefined for the moment, this struct defines a point within a specified pixrect:

```
struct pr_prpos  {
        struct pixrect *pr;
        struct pr_pos pos;
};
```

It contains a pointer to the pixrect and a position within it.

The following struct defines the width and height of an area:

```
struct pr_size  {
        int x, y;
};
```

The following struct defines a sub-area within a pixrect:

```
struct pr_subregion  {
        struct pixrect *pr;
        struct pr_pos pos;
        struct pr_size size;
};
```

It contains a pointer to the pixrect, an origin for the area, and its width and height.

2.1.3. The Pixrect Struct

A particular pixrect is described by a `pixrect` struct. This combines the definition of a rectangular array of pixels and the means of accessing operations for manipulating those pixels:

```
struct pixrect  {
        struct  pixrectops *pr_ops;
        struct  pr_size pr_size;
        int     pr_depth;
        caddr_t pr_data;  ,
};
```

The width and height of the rectangle are given in `pr_size`, and the number of bits in each pixel in `pr_depth`. For programmers more comfortable referring to "width" and "height," there are also two convenient macros:

```
#define pr_width  (pr_size.x)
#define pr_height (pr_size.y)
```

All other information about the pixrect (in particular, the location and values of pixels), is data private to it. Pixels are manipulated only by the set of ***pixrect operations*** described below. These operations will generally use information accessed through `pr_data` to accomplish their tasks.

(This restriction is relaxed somewhat in the case of pixrects whose pixels are stored in memory; this provides an escape to mechanisms outside the pixrect facility for constructing and converting pixrects of differing types. Memory pixrects are described in ***Memory Pixrects***.)

2.2. Operations on Pixrects

Procedures are provided to perform the following operations on pixrects:

- create and destroy a pixrect (`open`, `region` and `destroy`)
- read and write the values of single pixels (`get` and `put`)
- use RasterOp functions to affect multiple pixels in a single operation:
  write from a source to a destination pixrect (`rop`)
  write from a source to a destination under control of a mask (`stencil`)
  replicate a constant source pattern throughout a destination (`replrop`)
  write a batch of sources to different locations in a single destination (`batchrop`)
  draw a straight line of a single source value (`vector`)
- read and write a colormap (`getcolormap`, `putcolormap`)
- select particular bit-planes for manipulation on a color pixrect (`getattributes`, `putattributes`)

Some of these operations are the same for all pixrects, and are implemented by a single procedure. These device-independent procedures are called directly by pixrect clients. Other operations must be implemented differently for each device on which a pixrect may exist. Each pixrect includes a pointer (in its `pr_ops`) to a `pixrectops` structure, that holds the addresses of the particular device-dependent procedures appropriate to that pixrect. This allows clients to access those procedures in a device-independent fashion, by calling through the procedure pointer, rather than naming the procedure directly. To facilitate this indirection, the pixrect facility provides a set of macros which look like simple procedure calls to generic operations, and expand to invocations of the corresponding procedure in the pixrectops structure.

The description of each operation will specify whether it is a true procedure or a macro, since some of the arguments to macros are expanded multiple times, and could cause errors if the arguments contain expressions with side effects. (In fact, two sets of parallel macros are provided, which differ only in whether their arguments use the geometry structs defined above. Each is described with the operation.)

2.2.1. *The Pixrectops Struct*

The pixrectops struct is a collection of pointers to the device-dependent procedures for a particular device:

```
struct pixrectops  {
        int     (*pro_rop)();
        int     (*pro_stencil)();
        int     (*pro_batchrop)();
        int     (*pro_nop)();
        int     (*pro_destroy)();
        int     (*pro_get)();
        int     (*pro_put)();
        int     (*pro_vector)();
        struct  pixrect *(*pro_region)();
        int     (*pro_putcolormap)();
        int     (*pro_getcolormap)();
        int     (*pro_putattributes)();
        int     (*pro_getattributes)();
};
```

All other operations are implemented by device-independent procedures.

2.2.2. *Conventions for Naming Arguments to Pixrect Operations*

In general, the following conventions are used in naming the arguments to pixrect operations:

Table 2-1: Argument Name Conventions

Argument	**Meaning**
d	destination
s	source
x and **y**	left and top origins
w and **h**	width and height

2.2.3. Pixrect Errors

Pixrect procedures which return a pointer to a structure will return NULL when they fail. Otherwise, a return value of PIX_ERR (-1) indicates failure and 0 indicates success. The section describing each library procedure makes note of any exceptions to this convention.

2.2.4. Creation and Destruction of Pixrects

Pixrects are created by the procedures `pr_open` and `mem_create`, by the procedures accessed by the macro `pr_region`, and at compile-time by the macro `mpr_static`. Pixrects are destroyed by the procedures accessed by the macro `pr_destroy`. `mem_create` and `mpr_static` are discussed in the section *Memory Pixrects*; the rest of these are described here.

2.2.4.1. Open: Create a Primary Display Pixrect

The properties of a non-memory pixrect depend on an underlying UNIX device. Thus, when creating the first pixrect for a device you need to open it by a call to:

```
struct  pixrect *pr_open(devicename)
        char    *devicename;
```

The default device name for your display is `/dev/fb` (`fb` stands for *framebuffer*). Any other device name may be used provided that it is a display device, the kernel is configured for it, and it has pixrect support, for example, `/dev/bwone0`, `/dev/bwtwo0`, `/dev/cgone0` or `/dev/cgtwo0`.

`pr_open` does not work for creating a pixrect whose pixels are stored in memory; that function is served by the procedure `mem_create`, discussed in the section *Memory Pixrects.*

`pr_open` returns a pointer to a primary pixrect struct which covers the entire surface of the named device. If it cannot, it returns NULL, and prints a message on *stderr.*

2.2.4.2. Region: Create a Secondary Pixrect

Given an existing pixrect, it is possible to create another pixrect which refers to some or all of the same pixels of the same pixrect. This is called a ***secondary pixrect***, and is created by a call to the procedures invoked by the macros `pr_region` and `prs_region`:

```
#define struct  pixrect *pr_region(pr, x, y, w, h)
        struct  pixrect *pr;
        int     x, y, w, h;

#define struct  pixrect *prs_region(subreg)
        struct  pr_subregion subreg;
```

The existing pixrect is addressed by `pr`; it may be a pixrect created by `pr_open`, `mem_create` or `mpr_static` (a ***primary*** pixrect); or it may be another secondary pixrect created by a previous call to a region operation. The rectangle to be included in the new pixrect is described by `x`, `y`, `w` and `h` in the existing pixrect; (`x`, `y`) in the existing pixrect will map to (0, 0) in the new one. `prs_region` does the same thing, but has all its argument values collected into the single struct `subreg`. Each region procedure returns a pointer to the new pixrect. If it fails, it returns NULL, and prints a message on *stderr*.

If an existing secondary pixrect is provided in the call to the region operation, the result is another secondary pixrect referring to the underlying primary pixrect; there is no further connection between the two secondary pixrects. Generally, the distinction between primary and secondary pixrects is not important; however, no secondary pixrect should ever be used after its primary pixrect is destroyed.

2.2.4.3. Close / Destroy: Release a Pixrect's Resources

The following macros invoke device-dependent procedures to destroy a pixrect, freeing resources that belong to it:

```
#define pr_close(pr)
        struct  pixrect *pr;

#define pr_destroy(pr)
        struct  pixrect *pr;

#define prs_destroy(pr)
        struct  pixrect *pr;
```

The procedure returns 0 if successful, PIX_ERR if it fails. It may be applied to either primary or secondary pixrects. If a primary pixrect is destroyed before secondary pixrects which refer to its pixels, those secondary pixrects are invalidated; attempting any operation but `destroy` on them is an error. The three macros are identical; they are all defined for reasons of history and stylistic consistency.

2.2.5. Single-Pixel Operations

The next two operations manipulate the value of a single pixel.

2.2.5.1. Get: Retrieve the Value of a Single Pixel

The following macros invoke device-dependent procedures to retrieve the value of a single pixel:

```
#define pr_get(pr, x, y)
        struct  pixrect *pr;
        int     x, y;

#define prs_get(srcprpos)
        struct  pr_prpos srcprpos;
```

`pr` indicates the pixrect in which the pixel is to be found; `x` and `y` are the coordinates of the pixel. For `prs_get`, the same arguments are provided in the single struct `srcprpos`. The value of the pixel is returned as a 32-bit integer; if the procedure fails, it returns PIX_ERR.

2.2.5.2. Put: Store a Value into a Single Pixel

The following macros invoke device-dependent procedures to store a value in a single pixel:

```
#define pr_put(pr, x, y, value)
        struct  pixrect *pr;
        int     x, y, value;

#define prs_put(dstprpos, value)
        struct  pr_prpos dstprpos;
        int     value;
```

`pr` indicates the pixrect in which the pixel is to be found; `x` and `y` are the coordinates of the pixel. For `prs_put`, the same arguments are provided in the single struct `dstprpos`. `value` is truncated on the left if necessary, and stored in the indicated pixel. If the procedure fails, it returns PIX_ERR.

2.2.6. Constructing an Op Argument

The multi-pixel operations described in the next section all use a uniform mechanism for specifying the operation which is to produce destination pixel values. This operation is given in the `op` argument and includes several components.

Generally, `op` identifies a RasterOp. This is a logical function of two or three inputs; it computes the value of each pixel in the destination as a function of the previous value of that destination pixel, of a corresponding source pixel, and possibly a corresponding pixel in a mask.

Two other facilities are also specified in the `op` argument:

- a single, constant, source value may be specified as a *color* in `op`, and
- the *clipping* which is normally performed by every pixrect operation may be turned off by setting the PIX_DONTCLIP flag in the `op`.

We describe these three components of the `op` argument in order.

2.2.6.1. Specifying a RasterOp Function

Four bits of the `op` are used to specify one of the 16 distinct logical functions which combine monochrome source and destination pixels to give a monochrome result. This encoding is generalized to pixels of arbitrary depth by specifying that the function is applied to corresponding bits of the pixels in parallel. This emphasizes that the pixrects must be of the same depth. Some functions are much more common than others; the most useful are identified in the table ***Useful Combinations of RasterOps.***

A convenient and intelligible form of encoding the function into four bits is supported by the following definitions:

```
#define PIX_SRC         0x18
#define PIX_DST         0x14
#define PIX_NOT(op)     (0x1E & (~op))
```

PIX_SRC and PIX_DST are defined constants, and PIX_NOT is a macro. Together, they allow a desired function to be specified by performing the corresponding logical operations on the appropriate constants. (The explicit definition of PIX_NOT is required to avoid inverting non-function bits of `op`).

A particular application of these logical operations allows definition of ***set*** and ***clear*** operations. The definition of the ***set*** operation that follows is always true, and hence sets the result:

```
#define PIX_SET (PIX_SRC | PIX_NOT(PIX_SRC))
```

The definition of the ***clear*** operation is always false, and hence clears the result:

```
#define PIX_CLR (PIX_SRC & PIX_NOT(PIX_SRC))
```

Other common RasterOp functions are defined in the following table:

Table 2-2: Useful Combinations of RasterOps

Op with Value	Result
`PIX_SRC`	write (same as source argument)
`PIX_DST`	no-op (same as destination argument)
`PIX_SRC | PIX_DST`	paint (OR of source and destination)
`PIX_SRC & PIX_DST`	mask (AND of source and destination)
`PIX_NOT(PIX_SRC) & PIX_DST`	erase (AND destination with negation of source)
`PIX_NOT(PIX_DST)`	invert area (negate the existing values)
`PIX_SRC ^ PIX_DST`	inverting paint (XOR of source and destination)

2.2.6.2. Ops with a Constant Source Value

In certain cases, it is desirable to specify an infinite supply of pixels, all with the same value. This is done by using NULL for the source pixrect, and encoding a color in bits 5 - 31 of the `op` argument. The following macro supports this encoding:

```
#define PIX_COLOR(color)          ((color)<<5)
```

This macro extracts the color from an op:

```
#define PIX_OPCOLOR(op)           ((op)>>5)
```

If no color is specified in an `op`, 0 appears by default. The color specified in the `op` is used in the case of a null source pixrect or to specify the color of the 'ink' in a depth 1 pixrect.

Note that the color is not part of the ***function*** component of an `op` argument; it should never be part of an argument to PIX_NOT.

The ***color*** component of `op` is also used when a depth 1 pixrect is written to a depth >1 pixrect. In this case:

- if the value of the source pixels = 0, they are painted 0, or background.

- if the value of the source pixels = 1, they are painted ***color***.

If the ***color*** component of `op` is 0 (e.g., because no color was specified), the color will default to -1 (foreground).

2.2.6.3. Controlling Clipping in the RasterOp

Pixrect operations normally clip to the bounds of the operand pixrects. Sometimes this can be done more efficiently by the client at a higher level. If the client can guarantee that only pixels which ought to be visible will be written, it may instruct the pixrect operation to bypass clipping checks, thus speeding its operation. This is done by setting the following flag in the `op` argument:

```
#define PIX_DONTCLIP    0x1
```

The result of a pixrect operation is undefined and may cause a memory fault if PIX_DONTCLIP is set and the operation goes out of bounds.

Note that the PIX_DONTCLIP flag is not part of the ***function*** component of an `op` argument; it should never be part of an argument to PIX_NOT.

2.2.6.4. Examples of Complete Op Argument Specification

A very simple `op` argument will specify that source pixels be written to a destination, clipping as they go:

```
op = PIX_SRC;
```

A more complicated example will be used to affect a rectangle (known to be valid) with a constant red color defined elsewhere. (The function is syntactically correct; it's not clear how useful it is to XOR a constant source with the negation of the OR of the source and destination):

```
op = (PIX_SRC ^ PIX_NOT(PIX_SRC | PIX_DST) ) | PIX_COLOR(red) | PIX_DONTCLIP
```

2.2.7. Multi-Pixel Operations

The following operations all apply to multiple pixels at one time: ***rop, stencil, replrop, batchrop,*** and ***vector***. With the exception of ***vector***, they refer to rectangular areas of pixels. They all use a common mechanism, the `op` argument described in the previous section, to specify how pixels are to be set in the destination.

2.2.7.1. Rop: RasterOp Source to Destination

Device-dependent procedures invoked by the following macros perform the indicated raster operation from a source to a destination pixrect:

```
#define pr_rop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
        struct  pixrect *dpr, *spr;
        int     dx, dy, dw, dh, op, sx, sy;
```

```
#define prs_rop(dstregion, op, srcprpos)
        struct  pr_subregion dstregion;
        int     op;
        struct  pr_prpos srcprpos;
```

`dpr` addresses the destination pixrect, whose pixels will be affected; (`dx`, `dy`) is the origin (the upper-left pixel) of the affected rectangle; `dw` and `dh` are the width and height of that rectangle. `spr` specifies the source pixrect, and (`sx`, `sy`) an origin within it. `spr` may be NULL, to indicate a constant source specified in the `op` argument, as described previously; in this case `sx` and `sy` are ignored. `op` specifies the operation which is performed; its construction is described in preceding sections.

For `prs_rop`, the `dpr`, `dx`, `dy`, `dw` and `dh` arguments are all collected in a `pr_subregion` structure, defined previously under *Geometry Structs*.

Raster operations are clipped to the source dimensions, if those are smaller than the destination size given. *Rop* procedures return PIX_ERR if they fail, 0 if they succeed.

Source and destination pixrects generally must be the same depth. The only exception allows depth=1 pixrects to be sources to a destination of any depth. In this case, source pixels = 0 are interpreted as 0 and source pixels = 1 are written as the maximum value which can be stored in a destination pixel.

2.2.7.2. Stencil: RasterOps through a Mask

Device-dependent procedures invoked by the following macros perform the indicated raster operation from a source to a destination pixrect only in areas specified by a third (stencil) pixrect:

```
#define pr_stencil(dpr,dx,dy,dw,dh,op,stpr,stx,sty,spr,sx,sy)
        struct  pixrect *dpr, *stpr, *spr;
        int     dx,dy,dw,dh,op,stx,sty,sx,sy;

#define prs_stencil(dstregion, op, stenprpos, srcprpos)
        struct  pr_subregion dstregion;
        int     op;
        struct  pr_prpos stenprpos, srcprpos;
```

Stencil is identical to *rop* except that the source pixrect is written through a stencil pixrect which functions as a spatial write-enable mask. The stencil pixrect must be a memory pixrect with depth = 1. The indicated raster operation is applied only to destination pixels where the stencil pixrect is non-zero. Other destination pixels remain unchanged. The rectangle from (`sx`, `sy`) in the source pixrect `spr` is aligned with the rectangle from (`stx`, `sty`) in the stencil pixrect `stpr`, and written to the rectangle at (`dx`, `dy`) with width `dw` and height `dh` in the destination pixrect `dpr`. The source pixrect `spr` may be NULL, in which case the color specified in `op` is painted through the stencil. Clipping restricts painting to the intersection of the destination, stencil and source rectangles. *Stencil* procedures return PIX_ERR if they fail, 0 if they succeed.

2.2.7.3. Replrop: Replicating the Source Pixrect

Often the source for a raster operation consists of a pattern that is used repeatedly, or replicated to cover an area. If a single value is to be written to all pixels in the destination, the best way is to specify that value in the ***color*** component of a ***rop*** operation. But when the pattern is larger than a single pixel, a mechanism is needed for specifying the basic pattern, and how it is to be laid down repeatedly on the destination. The `pr_replrop` procedure replicates a source pattern repeatedly to cover a destination area:

```
pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
        struct  pixrect *dpr, *spr;
        int     dx, dy, dw, dh, op, sx, sy;

#define prs_replrop(dsubreg, op, sprpos)
        struct  pr_subregion dsubreg;
        struct  pr_prpos sprpos;
```

`dpr` indicates the destination pixrect. The area affected is described by the rectangle defined by `dx, dy, dw, dh`. `spr` indicates the source pixrect, and the origin within it is given by `sx, sy`. The corresponding `prs_replrop` macro generates a call to `pr_replrop`, expanding its `dsubreg` into the five destination arguments, and `sprpos` into the three source arguments. `op` specifies the operation to be performed, as described above under ***Constructing Op Arguments.***

The effect of ***replrop*** is the same as though an infinite pixrect were constructed using copies of the source pixrect laid immediately adjacent to each other in both dimensions, and then a ***rop*** was performed from that source to the destination. For instance, a standard gray pattern may be painted across a portion of the screen by constructing a pixrect that contains exactly one tile of the pattern, and by using it as the source pixrect.

The alignment of the pattern on the destination is controlled by the source origin given by `sx, sy`. If these values are 0, then the pattern will have its origin aligned with the position in the destination given by `dx, dy`. Another common method of alignment preserves a global alignment with the destination, for instance, in order to repair a portion of a gray. In this case, the source pixel which should be aligned with the destination position is the one which has the same coordinates as that destination pixel, ***modulo*** the size of the source pixrect. ***replrop*** will perform this modulus operation for its clients, so it suffices in this case to simply copy the destination position (`dx, dy`) into the source position (`sx, sy`).

Replrop procedures return PIX_ERR if they fail, 0 if they succeed. Internally ***replrop*** may use ***rop*** procedures. In this case, ***rop*** errors are detected and returned by ***replrop***.

2.2.7.4. Batch RasterOp: Multiple Source to the Same Destination

Applications such as displaying text perform the same operation from a number of source pixrects to a single destination pixrect in a fashion that is amenable to global optimization. Device-dependent procedures invoked by the following macros perform raster operations on a sequence of sources to successive locations in a common destination pixrect:

```
#define pr_batchrop(dpr, dx, dy, op, items, n)
        struct  pixrect *dpr;
        int     dx, dy, op, n;
        struct  pr_prpos items[ ];

#define prs_batchrop(dstpos, op, items, n)
        struct  pr_prpos dstpos;
        int     op, n;
        struct  pr_prpos items[ ];
```

`items` is an array of `pr_prpos` structures used by a ***batchrop*** procedure as a sequence of source pixrects. Each item in the array specifies a source pixrect and an `advance` in `x` and `y`. The whole of each source pixrect is used, unless it needs to be clipped to fit the destination pixrect: `advance` is used to update the destination position, not as an origin in the source pixrect.

Batchrop procedures take a destination, specified by `dpr`, `dx` and `dy`, or by `dstpos` in the case of `prs_batchrop`; an operation specified in `op`, as described in ***Constructing Op Arguments*** above, and an array of `pr_prpos` addressed by the argument `items`, and whose length is given in the argument `n`.

The destination position is initialized to the position given by `dx` and `dy`. Then, for each `item`, the offsets given in `pos` are added to the previous destination position, and the operation specified by `op` is performed on the source pixrect and the corresponding rectangle whose origin is at the current destination position. Note that the destination position is updated for each item in the batch, and these adjustments are cumulative.

The most common application of ***batchrop*** procedures is in painting text; additional facilities to support this application are described below under ***Text Facilities for Pixrects.*** Note that the definition of ***batchrop*** procedures supports variable-pitch and rotated fonts, and non-roman writing systems, as well as simpler text.

Batchrop procedures return PIX_ERR if they fail, 0 if they succeed. Internally ***batchrop*** may use ***rop*** procedures. In this case, ***rop*** errors are detected and returned by ***batchrop.***

2.2.7.5. Vector: Draw a Straight Line

Device-dependent procedures invoked by the following macros draw a vector one unit wide between two points in the indicated pixrect:

```
#define pr_vector(pr, x0, y0, x1, y1, op, value)
        struct  pixrect *pr;
        int     x0, y0, x1, y1, op, value;

#define prs_vector(pr, pos0, pos1, op, value)
        struct  pixrect *pr;
        struct  pr_pos   pos0, pos1;
        int     op, value;
```

Vector procedures draw a vector in the pixrect indicated by `pr`, with endpoints at (`x0, y0`) and (`x1, y1`), or at `pos0` and `pos1` in the case of `prs_vector`. Portions of the vector lying outside the pixrect are clipped as long as PIX_DONTCLIP is 0 in the `op` argument. The `op` argument is constructed as described previously under ***Constructing Op Arguments***; and `value` specifies the resulting value of pixels in the vector. If the color in `op` is non-zero, it takes precedence over the `value` argument.

2.2.7.6. Draw Curved Shapes (pr_traprop)

`pr_traprop` is an advanced pixrect operation analogous to `pr_rop`. `pr_traprop` operates on a region called a ***trapezon***, rather than on a rectangle.

A trapezon is a region with an irregular boundary. Like a rectangle, a trapezon has four sides: top, bottom, left, and right. The top and bottom sides of a trapezon are straight and horizontal. A trapezon differs from a rectangle in that its left and right sides are irregular curves, called ***falls***, rather than straight lines.

A fall is a line of irregular shape. Vertically, a fall may only move downward. Horizontally, a fall may move to the left or to the right, and this horizontal motion may reverse itself. A fall may also sustain pure horizontal motion, that is, horizontal motion with no vertical motion.

The figures below show a typical trapezon with source and destination pixrects, and some examples of filled regions that were drawn by `pr_traprop`.

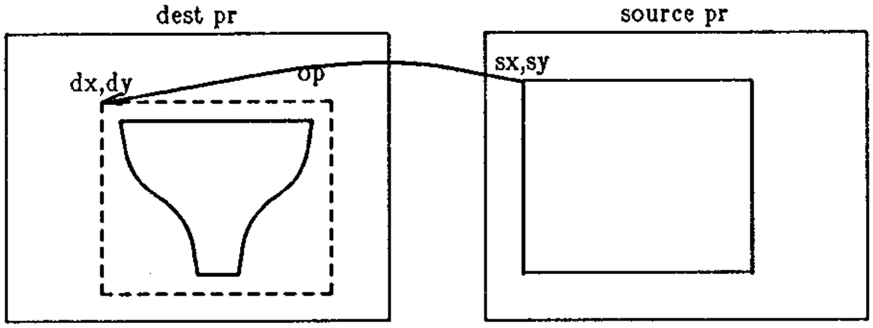


Figure 2-1: Typical trapezon with source and destination pixrects

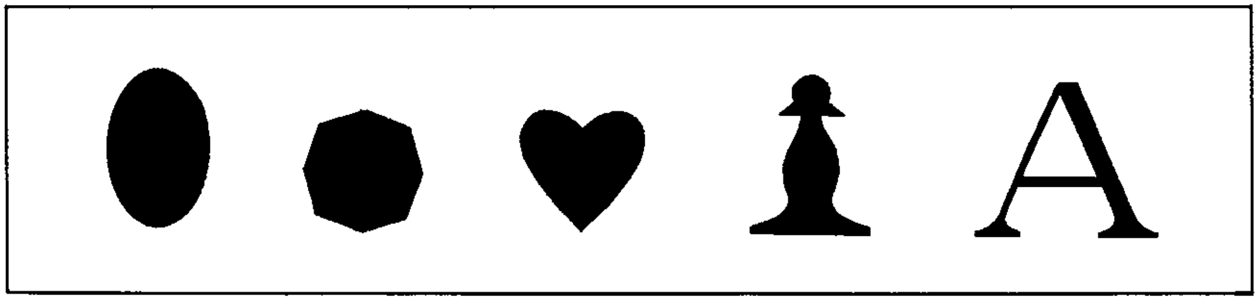


Figure 2-2: Some figures drawn by `pr_traprop`

```
pr_traprop(dpr, dx, dy, t, op, spr, sx, sy)
        struct  pixrect *dpr, *spr;
        struct  pr_trap  t;
        int     dx, dy, sx, sy op;
```

`dpr` and `spr` are pointers to the destination and source pixrects, respectively. `t` is the trapezon to be used. `dx` and `dy` specify an offset into the destination pixrect. `sx` and `sy` specify an offset into the source pixrect. `op` is an op-code as specified previously (see the section entitled ***Constructing an Op Argument***).

```
struct pr_trap {
        struct  pr_fall *left, *right;
        int     y0, y1;
};

struct pr_fall {
        struct  pr_pos pos;
        struct  pr_chain *chain;
};

struct pr_chain {
        struct  pr_chain *next;
        struct  pr_size size;
        int     *bits;
};
```

`pr_traprop` performs a rasterop from the source to the destination, clipped to the trapezon's boundaries. A program must call `pr_traprop` once per trapezon; therefore this procedure must be called at least twice to draw the letter 'A' in the figure ***Some figures drawn by pr_traprop***.

The source pixrect is aligned with the destination pixrect; the pixel at (sx,sy) in the source pixrect goes to the pixel at (dx,dy) in the destination pixrect (see the figure ***Typical trapezon with source and destination pixrects***).

Positions within the trapezon are relative to position (dx,dy) in the destination pixrect. Thus, a position defined as (0,0) in the trapezon would actually be at (dx,dy) in the destination pixrect.

The structure `pr_trap` defines the boundaries of a trapezon. A trapezon consists of pointers to two falls (`*left` and `*right`) and two y coordinates specifying the top and bottom of the trapezon (`y0` and `y1`). Note that the trapezon's top and bottom may be of zero width; `y0` and `y1` may simply serve as points of reference.

Each fall consists of a starting position (`pos`) and a pointer to the head of the list of chains describing the path the fall is to take (`*chain`). A fall may start anywhere above the trapezon and end anywhere below it. `pr_traprop` ignores the portions of a fall that lie above and below the trapezon. If a fall is shorter than the trapezon, `pr_traprop` will clip the trapezon horizontally to the endpoint of the fall in question. The figure ***Trapezon with clipped falls*** illustrates the way this works.

A ***chain*** is a member of a linked list of structures that describes the movement of the fall. Each chain describes a single segment of the fall. Each chain consists of a pointer to the next member of the chain (`*next`), the size of the bounding box for the chain (`size`), and a pointer to a bit vector containing motion commands (`*bits`). Please see the section ***Geometry Structs*** for a description of the `pr_size` structure.

Each chain may specify motion to the right and/or down, or motion to the left and/or down; however, a single chain may not specify both rightward and leftward motion. Remember that motion may not proceed upward, and that straight horizontal motion is permitted.

The x value of the chain's `size` determines the direction of the motion: a positive x value indicates rightward motion, while a negative x value indicates leftward motion. The y value of the chain's `size` must always be positive, since a fall may not move upward (in the direction of negative y).

A chain's bit vector is a command string that tells `pr_traprop` how to draw each segment of the fall. Each set (1) bit in the vector is a command to move one pixel horizontally and each clear (0) bit is a command to move one pixel vertically. The bits within the bit vector are stored in byte order, from most significant bit to least significant bit. This ordering corresponds to the left-to-right ordering of pixels within a memory pixrect.

The fall begins at the starting position specified in `pr_fall`. The motion proceeds downward as specified in the first bit vector in the chain, from the high-order bit to the low-order bit. When the fall reaches the bottom of the bounding box, it continues at the top of the next chain's bounding box. Note that the fall will always begin and end at diagonally opposite corners of a given bounding box.

If a bit vector specifies a segment of the fall that would run outside of the bounding box, `pr_traprop` clips that segment of the fall to the bounding box. This would occur when the sum of the 1's in a chain's bit vector exceeds the chain's x size, or when the sum of the 0's in the chain's bit vector exceeds the chain's y size. When this happens, the segment in question runs along the edge of the bounding box until it reaches the corner of the bounding box diagonally opposite to the corner in which it started.

If the fall is to have a straight vertical segment, the x size of its chain must be 0. If the fall is to have a straight horizontal segment, the y size of its chain must be 0.

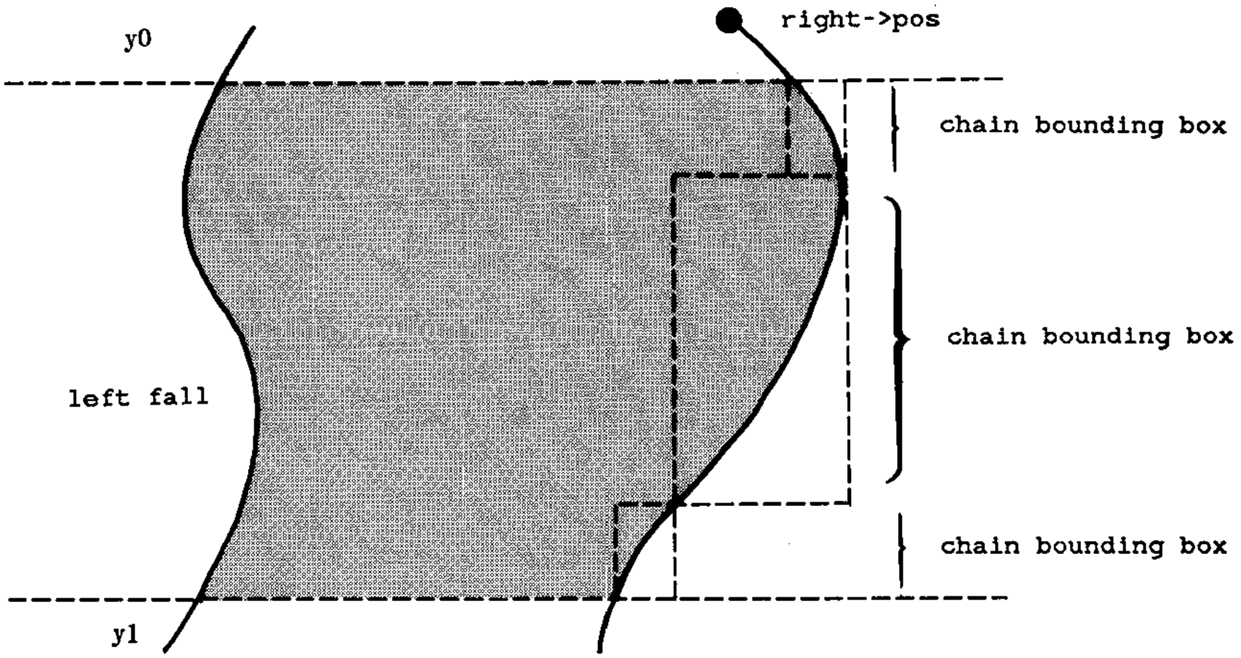


Figure 2-3: Trapezon with clipped falls

The following program draws the octagon shown in the figure *Some figures drawn by pr_traprop*". Make sure to give *cc* the library argument `-lpixrect`.

```
#include <pixrect/pixrect_hs.h>

int     shallowsteep[] = {0xbbbbbbbb, 0xbbbbbbbb, 0x44444444, 0x44444444},
        steepshallow[] = {0x44444444, 0x44444444, 0xbbbbbbbb, 0xbbbbbbbb};

struct pr_chain left1 = {0, {64, 64}, steepshallow},
                left0 = {&left1, {-64, 64}, shallowsteep},
                right1 = {0, {-64, 64}, steepshallow},
                right0 = {&right1, {64, 64}, shallowsteep};

struct pr_fall  left_oct   = {{0, 0}, &left0},
                right_oct  = {{0, 0}, &right0};

struct pr_trap  octagon  = {&left_oct, &right_oct, 0, 128};

main()
{
        pr_traprop(pr_open("/dev/fb"), 576, 450, octagon, PIX_SET, 0, 0, 0);
}
```

`pr_chain` specifies the left lower, the left upper, the right lower, and the right upper sides of the octagon, in that order. `pr_fall` specifies first the left side, then the right side of the octagon.

Each of the eight sides of the octagon is half a chain. The two upper left sides correspond to chain `left0`. The bits start out with mostly 1's (0xb is binary 1011) for the shallow uppermost left edge. They turn to mostly 0's (0x4 is binary 0100) for the next edge down, which is steeper.

2.2.7.7. Polygon: Textured Polygons with Holes

`pr_polygon_2` draws a polygon in a pixrect. The polygon can have holes. In addition, you can fill it with an image or a texture. You invoke `pr_polygon_2` as follows:

```
pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op, spr, sx, sy)
        struct  pixrect *dpr, *spr;
        int     dx, dy
        int     nbnds, npts[];
        struct  pr_pos  *vlist;
        int     op, sx, sy;
```

This routine is like `pr_rop` except that `nbnds`, `npts` and `vlist` specify the destination region instead of `(dw,dh)`.

`nbnds` is the number of individual closed boundaries (vertex lists) in the polygon. For example, the polygon may have one boundary for its exterior shape and several boundaries delimiting interior holes. The boundaries may self intersect or intersect each other. Those pixels having an ***odd wrapping number*** are painted. That is, if any line connecting a pixel to infinity crosses an odd number of boundary edges, the pixel will be painted.

For each of the `nbnds` boundaries `npts` specifies the number of points in the boundary. Hence the `npts` array is `nbnds` in length. The `vlist` contains all of the boundary points for all of the boundaries. The number of points in order are `npts[0]+...+npts[nbnds−1]`. `pr_polygon_2` joins the last point and first point to close each boundary.

The `spr` source pixrect fills the interior of the polygon as in `pr_rop`. The position `sx,sy` in `spr` coordinates coincides with position `dx, dy` in `dpr` coordinates. If `sx` = -5 and `sy` = -10, for example, the source pixrect is positioned at (`dx`+5, `dy`+10) in `dpr` coordinates. `pr_polygon_2` clips to both `spr` and `dpr` except in the case of NULL `spr`, where the polygon is filled with the color value in `op`. The source offset `sx,sy` is used to superimpose the source image over the polygon. The `spr` must have depth less than or equal to the depth of `dpr`. A point (`pts[n].x, pts[n].y`) in the boundary of a polygon is mapped to `(dx + pts[n].x, dy + pts[n].y)`.

2.2.8. Colormap Access

A ***colormap*** is a table which translates a pixel value into 8-bit intensities in red, green, and blue. For a pixrect of depth *n*, the corresponding colormap will have 2^n entries. The two most common cases are depth=1 (monochrome with two entries) and depth=8 (with 256 entries). Memory pixrects do not have colormaps.

2.2.8.1. Get Colormap

The following macros invoke device-dependent procedures to read all or part of a colormap into arrays in memory:

```
#define pr_getcolormap(pr, index, count, red, green, blue)
        struct pixrect  *pr;
        int             index, count;
        unsigned char   red[ ], green[ ], blue[ ];

#define prs_getcolormap(pr, index, count, red, green, blue)
        struct pixrect  *pr;
        int             index, count;
        unsigned char   red[ ], green[ ], blue[ ];
```

These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations.

`pr` identifies the pixrect whose colormap is to be read; the `count` entries starting at `index` (zero origin) are read into the three arrays.

For monochrome pixrects the same value is read into corresponding elements of the `red`, `green` and `blue` arrays. These array elements will have their bits either all cleared, indicating ***black***, or all set, indicating ***white***. By default, the 0th (***background***) element is white, and the 1st (***foreground***) element is black. Colormap procedures return -1 if the index or count are out of bounds, and 0 if they succeed.

2.2.8.2. Put Colormap

The following macros invoke device-dependent procedures to store from memory into all or part of a colormap:

```
#define pr_putcolormap(pr, index, count, red, green, blue)
          struct pixrect  *pr;
          int             index, count;
          unsigned char   red[ ], green[ ], blue[ ];

#define prs_putcolormap(pr, index, count, red, green, blue)
          struct pixrect  *pr;
          int             index, count;
          unsigned char   red[ ], green[ ], blue[ ];
```

These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations.

The `count` elements starting at `index` (zero origin) in the colormap for the pixrect identified by `pr` are loaded from corresponding elements of the three arrays.

For monochrome pixrects, the only value considered is `red[0]`. If this value is 0, then the pixrect will be set to a dark background and light foreground. If the value is non-zero, the foreground will be dark, e.g. black-on-white. Monochrome pixrects are dark-on-light by default.

Note: Full functionality of the colormap is not supported for depth=1 pixrects. Colormap changes to depth=1 pixrects apply only to subsequent operations whereas a colormap change to a color device instantly changes all affected pixels on the display surface.

2.2.8.3. Provision for Inverted Video Pixrects

Video inversion is accomplished by manipulation of the colormap of a pixrect. The colormap of a depth=1 pixrect has two elements. The following procedures provide video inversion control:

```
pr_blackonwhite(pr, min, max)
          struct  pixrect *pr;
          int     min, max;

pr_whiteonblack(pr, min, max)
          struct  pixrect *pr;
          int     min, max;

pr_reversevideo(pr, min, max)
          struct  pixrect *pr;
          int     min, max;
```

In each procedure, `pr` identifies the pixrect to be affected; `min` is the lowest index in the colormap, specifying the background color, and `max` is the highest index, specifying the foreground color. These will most often be 0 and 1 for monochrome pixrects; the more general definitions allow colormap-sharing schemes, such as the one described in ***Colormap Sharing***, in the chapter ***Overlapped Windows: Imaging Facilities***.

"Black-on-white" means that zero (background) pixels will be painted at full intensity, which is usually white. `pr_blackonwhite` sets all bits in the entry for colormap location `min` and clears all bits in colormap location `max`.

"White-on-black" means that zero (background) pixels will be painted at minimum intensity, which is usually black. `pr_whiteonblack` clears all bits in colormap location `min` and sets all bits in the entry for colormap location `max`.

`pr_reversevideo` exchanges the `min` and `max` color intensities.

These procedures are ignored for memory pixrects.

Note: These procedures are intended for global foreground/background control, not for local highlighting. For monochrome frame buffers, ***subsequent*** operations will have inverted intensities. For color frame buffers, the colormap is modified immediately, which affects everything in the display.

2.2.9. Attributes for Bitplane Control

In a color pixrect, it is often useful to define bitplanes which may be manipulated independently; operations on one plane leave the other planes of an image unaffected. This is normally done by assigning a plane to a constant bit position in each pixel. Thus, the value of the *i*th bit in all the pixels defines the *i*th bitplane in the image. It is sometimes beneficial to restrict pixrect operations to affect a subset of a pixrect's bitplanes. This is done with a bitplane mask. A bitplane mask value is stored in the pixrect's private data and may be accessed by the attribute operations.

2.2.9.1. Get Attributes

Device-dependent procedures invoked by the following macros retrieve the mask which controls which planes in a pixrect are affected by other pixrect operations:

```
#define pr_getattributes(pr, planes)
        struct  pixrect *pr;
        int     *planes;

#define prs_getattributes(pr, planes)
        struct  pixrect *pr;
        int     *planes;
```

`pr` identifies the pixrect; its current bitplanes mask is stored into the word addressed by `planes`. If `planes` is NULL, no operation is performed.

The two macros are identically defined; both are provided to allow consistent use of the same style of names.

2.2.9.2. Put Attributes

Device-dependent procedures invoked by the following macro manipulate a mask which controls which planes in a pixrect are affected by other pixrect operations:

```
#define pr_putattributes(pr, planes)
        struct  pixrect *pr;
        int     *planes;

#define prs_putattributes(pr, planes)
        struct  pixrect *pr;
        int     *planes;
```

The two macros are identically defined; both are provided to allow consistent use of the same style of names.

`pr` identifies the pixrect to be affected.

The `*planes` argument is a pointer to a bitplane write-enable mask. Only those planes corresponding to mask bits having a value of 1 will be affected by subsequent pixrect operations. If `*planes` is NULL, no operation is performed.

Note: If any `planes` are masked off by a call to `pr_putattributes`, no further write access to those planes is possible until a subsequent call to `pr_putattributes` unmasks them. However, these planes can still be read.

2.2.10. Efficiency Considerations

For maximum execution speed, remember the following points when you write pixrect programs:

- `pr_get` and `pr_put` are relatively slow. For fast random access of pixels it is usually faster to read an area into a memory pixrect and address the pixels directly.
- `pr_rop` is fast for large rectangles.
- `pr_vector` is fast.
- functions run faster when clipping is turned off. Do this only if you can guarantee that all accesses are within the pixrect bounds.
- `pr_rop` is three to five times faster than `pr_stencil`
- `pr_batch_rop` cuts down the overhead of painting many small pixrects.

2.3. Text Facilities for Pixrects

Displaying text is an important task in many applications, so pixrect-level facilities are provided to address it directly. These facilities fall into two main categories: a standard format for describing fonts and character images, with routines for processing them; and a set of routines which take a string of text and a font, and handle various parts of painting that string in a pixrect.

2.3.1. Pixfonts and Pixchars

The following two structures describe fonts and character images for pixrect-level text facilities:

```
struct pixchar  {
        struct  pixrect *pc_pr;
        struct  pr_pos pc_home;
        struct  pr_pos pc_adv;
};

struct pixfont  {
        struct  pr_size pf_defaultsize;
        struct  pixchar pf_char[256];
};
```

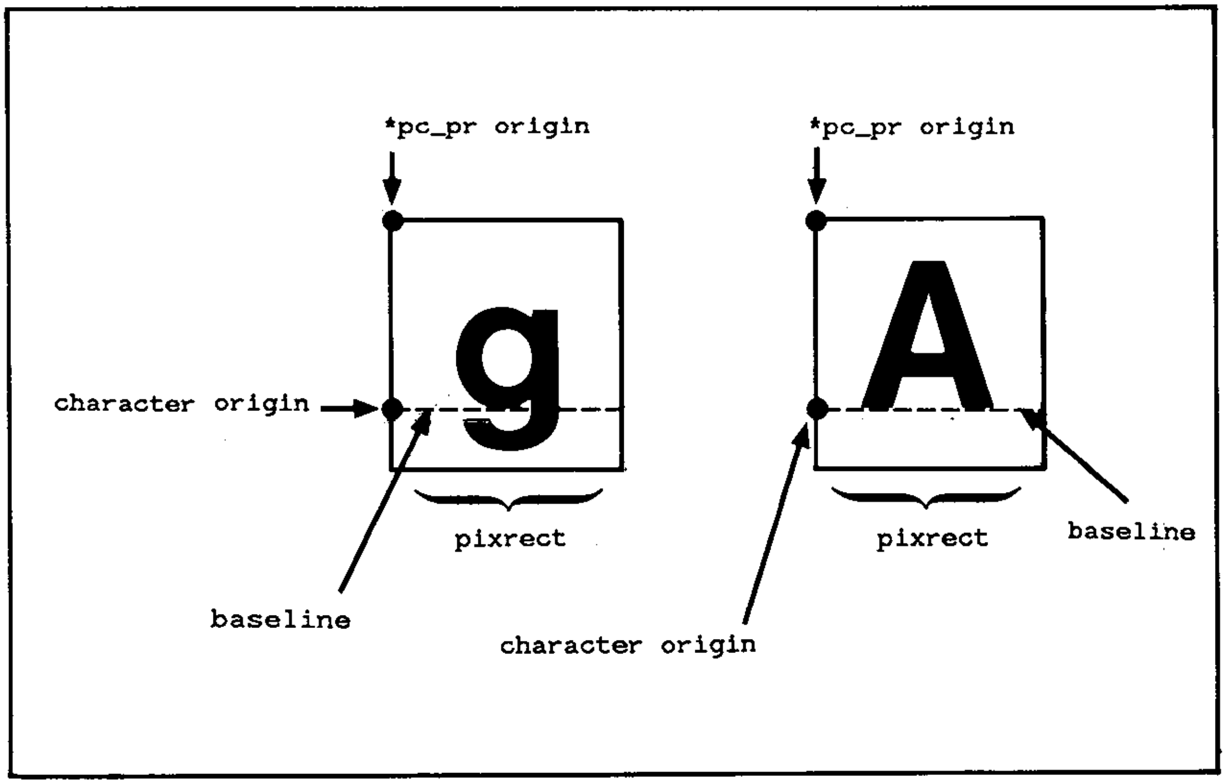
The `pixchar` defines the format of a single character in a font. The actual image of the character is a pixrect (a separate pixrect for each character) addressed by `pc_pr`. The entire pixrect gets painted. Characters that do not have a displayable image will have NULL in their entry in `pc_pr`. `pc_home` is the origin of pixrect `pc_pr` (its upper left corner) relative to the character origin. A character's origin is the leftmost end of its ***baseline***, which is the lowest point on characters without descenders. The figure below illustrates the `pc_pr` origin and the character origin.

The leftmost point on a character is normally its origin, but ***kerning*** or mandatory letter spacing may move the origin right or left of that point. `pc_adv` is the amount the destination position is changed by this character; that is, the amounts in `pc_adv` added to the current character origin will give the origin for the next character. While normal text only advances horizontally, rotated fonts may have a vertical advance. Both are provided for in the font.

A `pixfont` contains an array of `pixchars`, indexed by the character code; it also contains the size (in pixels) of its characters when they are all the same. (If the size of a font's characters varies in one dimension, that value in `pf_defaultsize` will not have anything useful in it; however, the other may still be useful. Thus, for non-rotated variable-pitch fonts, `pf_defaultsize.y` will still indicate the unleaded interline spacing for that font.)

Note: The definition of a `pixfont` is expected to change.

Figure 2-4: Character and pc_pr origins



2.3.2. Operations on Pixfonts

Before a client may use a font, it must ensure that the font has been loaded into virtual memory; this is done with `pf_open`:

```
struct pixfont  *pf_open(name)
    char        *name;
```

This procedure opens the file with the given `name`. The file should be a font file as described in ***vfont***(5): the file is converted to pixfont format, allocating memory for its associated structs and reading in the data for it from disk. A NULL is returned if the font cannot be opened.

The procedure:

```
struct pixfont *pf_default()
```

performs the same function for the system default font, normally a fixed-pitch, 16-point sans serif font with upper-case letters 12 pixels high. If the environment parameter DEFAULT_FONT is set, its value will be taken as the name of the font file to be opened by `pf_default`. The entire path name of the font file must be specified, for example:

```
myfont = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.7");
```

Note: `pf_open` and `pf_default` load a new copy of the font every time they are called, even if the font has already been loaded. To conserve memory, clients may use `pw_pfsysopen`, described in ***Overlapped Windows: Imaging Facilities***, or take care only to open a font once in a process.

When a client is finished with a font, it should call `pf_close` to free the memory associated with it:

```
pf_close(pf)
        struct  pixfont *pf;
```

`pf` should be a font handle returned by a previous call to `pf_open` or `pf_default`.

2.3.3. Pixrect Text Display

Characters are written into a pixrect with the `pf_text` procedure:

```
pf_text(where, op, font, text)
        struct  pr_prpos where;
        int     op;
        struct  pixfont *font;
        char    *text;
```

The `where` argument is the destination for the start of the text (nominal left edge, baseline; see ***Pixfonts***); `op` is the raster operation to be used in writing the text, as described in ***Constructing Op Arguments***; `font` is a pointer to the font in which the text is to be displayed; and `text` is the actual null-terminated string to be displayed. No error indicators are returned. ***Note:*** The color specified in the `op` specifies the color of the ink. The background of the text is painted 0 (background color).

The following procedure paints "transparent" text: it doesn't disturb destination pixels in blank areas of the character's image:

```
pf_ttext(where, op, font, text)
        struct  pr_prpos where;
        int     op;
        struct  pixfont *font;
        char    *text;
```

The arguments to this procedure are the same as for `pf_text`. The characters' bitmaps are used as a stencil, and the color specified in `op` is squirted through the stencil. No error indicators are returned.

(For monochrome pixrects, the same effect can be achieved by using PIX_SRC | PIX_DST as the function in the `op`; this procedure is for color pixrects.)

Auxiliary procedures used with `pf_text` include:

```
struct pr_size pf_textbatch(where, lengthp, font, text)
        struct  pr_pos where[];
        int     *lengthp;
        struct  pixfont *font;
        char    *text;

struct pr_size pf_textwidth(len, font, text)
        int     len;
        struct  pixfont *font;
        char    *text;
```

`pf_textbatch` is used internally by `pf_text`; it constructs an array of `pr_pos` structures and records its length, as required by `batchrop` (see *Batch Raster Op*). `where` should be the address of the array to be filled in, and `lengthp` should point to a maximum length for that array. `text` addresses the null-terminated string to be put in the batch, and `font` refers to the pixfont to be used to display it. When the function returns, `*lengthp` will refer to a word containing the number of `pr_pos` structures actually used for `text`. The `pr_size` returned is the sum of the `pc_adv` fields in their `pixchar` structs.

`pf_textwidth` returns a `pr_size` which contains the sum of the `len` characters in the text of the `pc_adv` in their pixchar structs.

The following routine may be used to find the bounding box for a string of characters in a given font.

```
pf_textbound(bound, len, font, text)
        struct  pr_subregion  *bound;
        int     len;
        struct  pixfont *font;
        char    *text;
```

`bound->pos` is the top-left corner of the bounding box, `bound->size.x` is the width, and `bound->size.y` is the height. `bound->pr` is not modified. `bound->pos` is computed relative to the location of the character origin (base point) of the first character in the text.

2.4. Memory Pixrects

Pixrects which store their pixels in memory, rather than displaying them on some display, are similar to other pixrects but have several special properties. Like all other pixrects, their dimensions are visible in the `pr_size` and `pr_depth` elements of their pixrect struct, and the device-dependent operations appropriate to manipulating them are available through their `pr_ops`. Beyond this, however, the format of the data which describes the particular pixrect is also public: `pr_data` will hold the address of an `mpr_data` struct, described below. Thus, a client may construct and manipulate memory pixrects using non-pixrect operations. There is also a public procedure, `mem_create`, which dynamically allocates a new memory pixrect, and a macro, `mpr_static`, which can be used to generate an initialized memory pixrect in the code of a client program.

2.4.1. The Mpr_data Struct

The `pr_data` element of a memory pixrect points to an `mpr_data` struct, which contains the information needed to deal with a memory pixrect:

```
struct mpr_data  {
        int     md_linebytes;
        short   *md_image;
        struct  pr_pos md_offset;
        short   md_primary;
        short   md_flags;
};
#define MP_DISPLAY
#define MP_REVERSEVIDEO
```

`linebytes` is the number of bytes stored in a row of the primary pixrect. This is the difference in the addresses between two pixels at the same x-coordinate, one row apart. Because a secondary pixrect may not include the full width of its primary pixrect, this quantity cannot be computed from the width of the pixrect — see ***Region***. The actual pixels of a memory pixrect are stored someplace else in memory, usually an array, which `md_image` points to; the format of that area is described in the next section. The creator of the memory pixrect must ensure that `md_image` contains an even address. `md_offset` is the `x,y` position of the first pixel of this pixrect in the array of pixels addressed by `md_image`. `md_primary` is 1 if the pixrect is primary and had its image allocated dynamically (e.g. by `mem_create`). In this case, `md_image` will point to an area not referenced by any other primary pixrect. This flag is interrogated by the `destroy` routine: if it is 1 when that routine is called, the pixrect's image memory will be freed.

(`md_flags` & MP_DISPLAY) is non-zero if this memory pixrect is in fact a display device. Otherwise, it is 0. (`md_flags` & MP_REVERSEVIDEO) is 1 if ***reversevideo*** is currently in effect for the display device. `md_flags` is present to support memory-mapped display devices like the Sun—2 black-and-white video device.

Several macros exist to aid in addressing memory pixrects. The following macro obtains a pointer to the `mpr_data` of a memory pixrect.

```
#define mpr_d(pr)
        ((struct mpr_data *)(pr)->pr_data)
```

The following macro computes the bytes per line of a primary memory pixrect given its width in pixels and the bits per pixel. This includes the padding to word bounds. It is useful for incrementing pixel addresses in the `y` direction.

```
#define mpr_linebytes (width,depth)
        ( ((pr_product(width,depth)+15)>>3) &~1)
```

2.4.2. Pixel Layout in Memory Pixrects

In memory, the upper-left corner pixel is stored at the lowest address. This address must be even. That first pixel is followed by the remaining pixels in the top row, left-to-right. Pixels are stored in successive bits without padding or alignment. For pixels more than 1 bit deep, it is possible for a pixel to cross a byte boundary. However, rows are rounded up to 16-bit boundaries. After any padding for the top row, pixels for the row below are stored, and so on through the whole rectangle. Currently, memory pixrects are only supported for pixels of 1, 8, 16, or 24 bits. If source and destination are both memory pixrects they must have an equal number of bits per pixel.

2.4.3. Creating Memory Pixrects

2.4.3.1. Mem_create

A new primary pixrect is created by a call to the procedure **mem_create**:

```
struct  pixrect *mem_create(w, h, depth)
        int     w, h, depth;
```

w, h, and **depth** specify the width and height in pixels, and depth in bits per pixel, of the new pixrect. Sufficient memory to hold those pixels is allocated and cleared to 0, new **mpr_data** and **pixrect** structs are allocated and initialized, and a pointer to the pixrect is returned. If this can not be done, the return value is NULL.

2.4.3.2. mem_point

The **mem_point** routine builds a pixrect structure that points to a dynamically created image in memory. Client programs may use this routine as an alternative to **mem_create** if the image data is already in memory.

```
struct pixrect *
mem_point(width, height, depth, data)
        int             width, height, depth;
        short          *data;
```

width and **height** are the width and height of the new pixrect, in pixels. **depth** is the depth of the new pixrect, in number of bits per pixel. **data** points to the image to be associated with the pixrect.

2.4.3.3. Static Memory Pixrects

A memory pixrect may be created at compile time by using the `mpr_static` macro:

```
#define mpr_static(name, w, h, depth, image)
        int     w, h, depth;
        short   *image;
```

where `name` is a token to identify the generated data objects; `w`, `h`, and `depth` are the width and height in pixels, and depth in bits of the pixrect; and `image` is the address of an even-byte aligned data object that contains the pixel values in the format described above.

The macro generates two structs:

```
        struct  mpr_data name_data;
        struct  pixrect name;
```

The `mpr_data` is initialized to point to all of the image data passed in; the pixrect then refers to `mem_ops` and to `name_`data. **Note:** Contrary to its name, this macro generates structs of storage class `extern`.

2.5. File I/O Facilities for Pixrects

Sun has specified a file format for files containing raster images. This format is defined by the header file `<rasterfile.h>`. The pixrect library contains routines to perform I/O operations between pixrects and files in the raster file format. This I/O is done using the routines of the C Library Standard I/O package, requiring the caller to include the header file `<stdio.h>`.

The raster file format allows for multiple types of raster images. This means that both unencoded and encoded images are supported. In addition, the pixrect library routines that read and write raster files support the notion of customer defined formats. This support is implemented by passing raster files with non-standard types through filters found in the directory `/usr/lib/rasfilters`. This directory also includes sample source code for a filter that corresponds to one of the standard raster file types.

2.5.1. Writing of Complete Raster Files

The following procedure stores the image described by a pixrect onto a file. It normally returns 0, but if any error occurs it returns PIX_ERR.

```
int
pr_dump(input_pr, output, colormap, type, copy_flag)
        struct pixrect  *input_pr;
        FILE            *output;
        colormap_t      *colormap;
        int             type, copy_flag;
```

The `input_pr` pixrect can be a secondary pixrect. This allows the caller to write a rectangular sub-region of a pixrect by first creating an appropriate `input_pr` via a call to `pr_region`. The output file is specified via `output`. The desired output type should either be one of the following standard types or correspond to a customer provided filter.

```
#define RT_OLD          0
#define RT_STANDARD     1
#define RT_BYTE_ENCODED 2
```

The RT_STANDARD type is the common raster file format in the same sense that memory pixrects are the common pixrect format: every raster file filter is required to read and write this format. The RT_OLD type is very close to the RT_STANDARD type; it was the former standard generated by old versions of Sun software. The RT_BYTE_ENCODED type implements a run-length encoding of bytes of the pixrect image; usually this results in shorter files. Specifying any other output type causes `pr_dump` to pipe a raster file of RT_STANDARD type to the filter named `/usr/lib/rasfilters/convert.`***type***, where ***type*** is the ASCII corresponding to the specified ***type*** in decimal. The output of the filter is then copied to `output`.

It is strongly recommended that customer-defined formats use a type of 100 or more, to avoid conflicts with additions to the set of standard types. To aid in development of filters for customer-defined formats, `pr_dump` recognizes the RT_EXPERIMENTAL type as special, and uses the filter named

```
#define RT_EXPERIMENTAL 65535
```

For pixrects displayed on devices with colormaps, the values of the pixels are not sufficient to recreate the displayed image. Thus, the image's colormap can also be specified in the call to `pr_dump`. If the `colormap` is specified as NULL but `input_pr` is not of depth=1, `pr_dump` will attempt to write the colormap obtained from `input_pr` (via `pr_getcolormap` assuming a 256 element RGB colormap). The following struct is used to specify the colormap associated with `input_pr`:

```
typedef struct {
        int             type;
        int             length;
        unsigned char  *map[3];
} colormap_t;
```

The colormap type should be one of the Sun supported types:

```
#define RMT_NONE        0
#define RMT_EQUAL_RGB   1
```

If the colormap type is RMT_NONE, then the colormap length must be 0. This case usually arises when dealing with monochrome displays and depth=1 pixrects. If the colormap type is RMT_EQUAL_RGB, then the map array should specify the red (`map[0]`), green (`map[1]`) and blue (`map[2]`) colormap values, with each vector in the map array being of the same specified colormap length. For developers of customer-defined formats, the following colormap type is provided but not interpreted by the pixrect software:

```
#define RMT_RAW         2
```

Finally `copy_flag` specifies whether or not `input_pr` should be copied to a temporary pixrect before the image is output. There are two situations in which the `copy_flag` value should be non-zero:

- if the output type is RT_BYTE_ENCODED — This is because the encoding algorithm does the encoding in place and will destroy the image data of `input_pr` if it fails while working on `input_pr` directly.
- if `input_pr` is a pixrect in a framebuffer that is likely to be asynchronously modified — Note that use of `copy_flag` will still not guarantee that the correct image will be output unless the `pr_rop` to copy from the framebuffer is atomic or otherwise made uninterruptable.

2.5.2. Reading of Complete Raster Files

The following procedure can be used to retrieve the image described by a file into a pixrect.

```
struct pixrect *
pr_load(input, colormap)
        FILE            *input;
        colormap_t      *colormap;
```

The raster file's header is read from `input`, a pixrect of the appropriate size is dynamically allocated, the colormap is read and placed in the location addressed by `*colormap`, and finally the image is read into the pixrect and the pixrect returned. If any problems occurs, `pr_load` returns NULL instead.

As with `pr_dump`, if the specified raster file is not of standard type, `pr_load` first runs the file through the appropriate filter to convert it to RT_STANDARD type and then loads the output of the filter.

Additionally, if colormap is NULL, `pr_load` will simply discard any and all colormap information contained in the specified input raster file.

2.5.3. Details of the Raster File Format

A handful of additional routines are available in the pixrect library for manipulating pieces of raster files. In order to understand what they do, it is necessary to understand the exact layout of the raster file format.

The raster file is in three parts: first, a small header containing 8 ints; second, a (possibly empty) set of colormap values; third, the pixel image, stored a line at a time, in increasing `y` order.

The image is essentially laid out in the file the exact way that it would appear in a memory pixrect. In particular, each line of the image is rounded out to a multiple of 16 bits, corresponding to the rounding convention used by the memory pixrects.

The header is defined by the following structure:

```
struct rasterfile {
        int     ras_magic;
        int     ras_width;
        int     ras_height;
        int     ras_depth;
        int     ras_length;
        int     ras_type;
        int     ras_maptype;
        int     ras_maplength;
};
```

The `ras_magic` field always contains the following constant:

```
#define RAS_MAGIC       0x59a66a95
```

The `ras_width`, `ras_height`, and `ras_depth` fields contain the image's width and height in pixels, and its depth in bits per pixel, respectively. The depth is usually either 1 or 8, corresponding to the standard frame buffer depths.

The `ras_length` field contains the length in bytes of the image data. For an unencoded image, this number is computable from the `ras_width`, `ras_height`, and `ras_depth` fields, but for an encoded image it must be explicitly stored in order to be available without decoding the image itself. Note that the length of the header and of the possibly empty colormap values are not included in the value in the `ras_length` field; it is only the image data length. For historical reasons, files of type RT_OLD will usually have a 0 in the `ras_length` field, and software expecting to encounter such files should be prepared to compute the actual image data length if it is needed. The `ras_maptype` and `ras_maplength` fields contain the type and length in bytes of the colormap values, respectively.

If the `ras_maptype` is not RMT_NONE and the `ras_maplength` is not 0, then the colormap values are the `ras_maplength` bytes immediately after the header. These values are either uninterpreted bytes (usually with the `ras_maptype` set to RMT_RAW) or the equal length red, green and blue vectors, in that order (when the `ras_maptype` is RMT_EQUAL_RGB). In the latter case, the `ras_maplength` must be three times the size in bytes of any one of the vectors.

2.5.4. Writing Parts of a Raster File

The following routines are available for writing the various parts of a raster file. Many of these routines are used to implement `pr_dump`. First, the raster file header and the colormap can be written by calling:

```
int
pr_dump_header(output, rh, colormap)
        FILE                    *output;
        struct rasterfile       *rh;
        colormap_t              *colormap;
```

This routine returns PIX_ERR if there is a problem writing the header or the colormap, otherwise it returns 0. If the colormap is NULL, no colormap values are written.

For clients that do not want to explicitly initialize the rasterfile struct the following routine can be used to set up the arguments for `pr_dump_header`:

```
struct pixrect *
pr_dump_init(input_pr, rh, colormap, type, copy_flag)
        struct pixrect          *input_pr;
        struct rasterfile       *rh;
        colormap_t              *colormap;
        int                      type, copy_flag;
```

The arguments to `pr_dump_init` correspond to the arguments to `pr_dump`. However, `pr_dump_init` returns the pixrect to write, rather than actually writing it, and initializes the struct pointed to by `rh` rather than writing it. If colormap is NULL, the `ras_maptype` and `ras_maplength` fields of `rh` will be set to RMT_NONE and 0, respectively.

If any error is detected by `pr_dump_init`, the returned pixrect is NULL. If there is no error and the `copy_flag` is zero, the returned pixrect is simply `input_pr`. However, if `copy_flag` is non-zero, the returned pixrect is dynamically allocated and the caller is responsible for deallocating the returned pixrect after it is no longer needed.

The actual image data can be output via a call to:

```
int
pr_dump_image(pr, output, rh)
        struct pixrect          *pr;
        FILE                    *output;
        struct rasterfile       *rh;
```

This routine returns 0 unless there is an error, in which case it returns PIX_ERR.

Since these routines sequentially advance the output file's write pointer, `pr_dump_image` must be called after `pr_dump_header`.

2.5.5. Reading Parts of a Raster File

The following routines are available for reading the various parts of a raster file. Many of these routines are used to implement `pr_load`. Since these routines sequentially advance the input file's read pointer, rather than doing random seeks in the input file, they should be called in the order presented below.

First, the raster file header can be read by calling:

```
int
pr_load_header(input, rh)
        FILE                    *input;
        struct rasterfile       *rh;
```

This routine reads the header from the specified input, checks it for validity and initializes the specified rasterfile struct from the header. The return value is 0 unless there is an error, in which case it returns PIX_ERR.

If the header indicates that there is a non-empty set of colormap values, they can be read by calling:

```
int
pr_load_colormap(input, rh, colormap)
        FILE                    *input;
        struct rasterfile       *rh;
        colormap_t              *colormap;
```

If the specified colormap is NULL, this routine will skip over the colormap values by reading and discarding them. Note that the caller is responsible for looking at the raster file header and setting up an appropriate colormap struct before calling this routine.

The return value is 0 unless there is an error, in which case it returns PIX_ERR.

Finally, the image can be read by calling:

```
struct pixrect *
pr_load_image(input, rh, colormap)
        FILE                    *input;
        struct rasterfile       *rh;
        colormap_t              *colormap;
```

If the input is a standard raster file type, this routine reads in the image directly. Otherwise, it writes the header, colormap, and image into the appropriate filter and then reads the output of the filter. In this case, both the rasterfile and the colormap structs will be modified as a side-effect of calling this routine. In either case, a pixrect is dynamically allocated to contain the image, the image is read into the pixrect, and the pixrect is returned as the result of calling the routine. If there is an error, the return value is NULL instead of a pixrect containing the image.

If it is known that the image is from a standard raster file type, then it can be read in by calling:

```
struct pixrect *
pr_load_std_image(input, rh, colormap)
        FILE                    *input;
        struct rasterfile       *rh;
```

This routine is identical to `pr_load_image`, except that it will not invoke a filter on non-standard raster file types.

Chapter 3

Overlapped Windows: Imaging Facilities

This chapter and the following two deal with the ***sunwindow*** layer of the window system, which provides facilities for managing windows with overlap and concurrency. This chapter is specifically concerned with generating images in such an environment. Chapter 4 deals with control of the windows, manipulating their size, location, and other structural characteristics. Chapter 5 describes the facilities for serializing multiple input streams and distributing them appropriately to multiple windows. The term "sunwindow layer" comes from the name of the library that contains its implementation.

At this level of the system, a window is treated as a ***device***: it is named by an entry in the `/dev` directory; it is accessed by the `open(2)` system call; and the usual handle on the window is the ***file descriptor*** (or `fd`) returned from that call.

For this chapter, however, a window may be considered as simply a rectangular area with contents maintained by some process. Multiple windows, maintained by independent processes, may coexist on the same screen; SunWindows allows them to ***overlap***, sharing the same (`x`, `y`) coordinates, and proceeding concurrently, while maintaining their separate identities.

Window system facilities may also be used to construct a non-overlapped environment; the window system facilities required are much the same as for constructing an overlapping environment.

3.1. Window Issues: Controlled Display Generation

Multiple windows on a display introduce two new issues, which may be broadly characterized as: 1) preventing the window from painting where or when it shouldn't, and 2) ensuring that it does paint whenever and wherever it should. The first includes ***clipping*** and ***locking***; the latter covers ***damage repair*** and ***fixups***.

3.1.1. Clipping and Locking

Clipping constrains a window to draw only within the boundaries of its portion of the screen. Even this area is subject to changes beyond the control of a window's process — another window may be opened on top of the first, covering part of its contents, or a window may be shrunk to make room for another alongside it. Thus, it is convenient for the window system to maintain up-to-date information on which portions of the screen belong to which windows, and for the windows to consult that information whenever they are about to draw on the screen.

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 should be 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.

Use of explicit locking calls is extremely important for achieving maximum display performance. Clipping and locking are described in more detail in ***Locking and Clipping***.

3.1.2. Damage Repair and Fixups

A window whose image does not appear entirely as it should on the screen is said to be ***damaged***. A common cause of damage is being first overlaid, and then uncovered, by another window. When a window is damaged, a portion of the window's image must be ***repaired***. Note that the requirement for repairing damage may arise at any time; it is completely outside the window's control.

When a process performs some operation which includes reading a portion of its window, for instance copying a part of the image from one region to another to implement scrolling, it may find the source pixels obscured. This necessitates a ***fixup***, in which that portion of the image is regenerated, similar to repairing damage. Unlike damage generation, the need to do some fixup is provoked only in response to an action of the window's process, e.g., scrolling.

3.1.3. Retained Windows

Either form of regeneration may be done by recomputing the image; this approach is reasonable for applications like text where there is some underlying representation from which the display can be recomputed easily. For images which require considerable computation, SunWindows provides a ***retained*** window, whose image is maintained in memory as well as on the display. Such a window may have its image recopied to the display as needed to repair damage. The mechanism for making a window ***retained*** is described in the section entitled ***Pixwins***.

3.1.4. Colormap Sharing

On color displays, colormap entries are a limited resource. When shared among multiple applications, colormap usage requires arbitration. For example, 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, SunWindows 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 knowing the correct offset to the start of a colormap segment from routines that access the image, SunWindow 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 input is directed towards Z's window. Z's request is not denied even though it is not allocated its own segment in the shared colormap.

3.1.5. Process Structure

In SunWindows, access to the screen is performed in each user process, instead of in a single, central, fully debugged screen management process. This increases the possibility of an incorrect user process damaging the display area of other application processes. Several compensating factors justify this approach:

- Clients may access this open system at whichever level is most convenient. Clients who require the ultimate efficiency of direct screen access need not sacrifice the window management functions of the window system.
- Leaving processing in user processes promotes efficiency in both implementation and execution: making and testing extensions and modifications is much easier in user code than in the kernel.

3.1.6. Imaging with Windows

A detailed discussion of imaging with windows follows. We begin with a description of the basic data structures that are used in this level of Sunwindows. These are a primitive geometric facility, the *rect*, for describing rectangles, and the basic structure, the pixwin, that describes a window on the screen with its associated state and operation vectors.

Following is a brief discussion of the simple process of creating and destroying pixwins. This is followed by a detailed description of the approach to locking and clipping, which leads naturally into a discussion of library routines that access a pixwin's pixels. Detecting and repairing damage is treated next.

3.1.7. Libraries and Header Files

The procedures described in this chapter are provided in the ***sunwindow*** library (`/usr/lib/libsunwindow.a`). The header file `<sunwindow/window_hs.h>` contains the declarations that must be `#include`'ed in a program that uses the facilities described in this chapter.

3.2. Data Structures

Here are some data structures used in the implementation of pixwins. Be sure you understand ***rects*** before proceeding. Descriptions of the data structure internals are also provided for additional information.

3.2.1. Rects

Throughout Sunwindows, images are dealt with in rectangular chunks; where complex shapes are required, they are built up out of groups of rectangles. The basic description of a rectangle is the `rect` struct, defined in the header file `<sunwindow/rect.h>`. The same file contains definitions of several useful macros and procedures for dealing with ***rects.***

Where a window is partially obscured, its visible portion generally cannot be described by a simple rectangle; instead a list of non-overlapping rectangular fragments which together cover the visible area is used. This `rectlist` is declared, along with its associated macros and procedures in the file `<sunwindow/rectlist.h>`.

At this point we only discuss the `rect` struct and its most useful macros; a full description of both ***rects*** and ***rectlists*** is in Appendix A.

```
#define coord short

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

In the context of a window, the rectangle lies in a coordinate system whose origin is in the upper left-hand corner, and whose dimensions are given in pixels. Two macros determine an edge not given explicitly in the `rect`. These macros are:

```
#define rect_right(rp)
#define rect_bottom(rp)

struct rect *rp;
```

These macros return the coordinate of the last pixel within the rectangle on the right or bottom, respectively.

3.2.2. Pixwins

Pixwins are the basic imaging elements of the overlapped window system. The window layer of the system uses pixwins to represent pixrects on a window surface. The pixwin thus describes the window image and a set of routines to operate on the window.

A client of the window system has a rectangular window in which it displays information for the user. Because of overlapping, however, it is not always possible to display information in all parts of a client's window. Parts of an image may have to be displayed at some point long after they were generated, as a portion of the window is uncovered. The clipping and repainting necessary to preserve the identity of the rectangular image across interference with other objects on the screen is handled by manipulations on pixwins.

The pixwin struct is defined in `<sunwindow/pixwin.h>`:

```
struct pixwin {
    struct      pixrectops *pw_ops;
    caddr_t     pw_opshandle;
    int         pw_opsx;
    int         pw_opsy;
    struct      rectlist pw_fixup;
    struct      pixrect *pw_pixrect;
    struct      pixrect *pw_prretained;
    struct      pixwin_clipops *pw_clipops;
    struct      pixwin_clipdata *pw_clipdata;
    char        pw_cmsname[20];
};
```

The pixwin refers to a portion of some device, typically a display; the device is identified by `pw_pixrect`.

If the image displayed in the pixwin required a large effort to compute, it will be worth saving a backup copy of the whole image, making the window a ***retained*** window. This is done by creating an appropriate ***memory pixrect*** as described in ***Memory Pixrects***, and storing a pointer to it in `pw_prretained`.

Portions of the image which could not be accessed by an operation which attempted to read pixels from the pixwin are indicated by `pw_fixup`.

`pw_ops` is a pointer to a vector of operations used in screen access macros to call the pixwin software level or, as an optimization, to call the ***pixrect*** software directly. The structure `pixrectops` was discussed in ***Pixrectops***. The `pw_opshandle` is the data handle passed to the operations of `pw_ops`. `pw_opsx` and `pw_opsy` are additional offset information that screen access macros use. These three fields are dynamically altered based on locking and clipping status.

`pw_clipdata` is a collection of information of special interest for locking and clipping. `pw_clipops` points to a vector of operations which are used in locking and clipping. The declarations of these last two structs are discussed more fully in ***pixwin_clipdata Struct***, ***pixwin_clipops Struct***, and subsequent sections.

`pw_cmsname` is the identifier of the colormap segment that this pixwin is currently using. This value should only be accessed via `pw_setcmsname` and `pw_getcmsname` described below.

3.2.3. Pixwin_clipdata Struct

```
struct pixwin_clipdata {
    int         pwcd_windowfd;
    short       pwcd_state;
    struct      rectlist pwcd_clipping;
    int         pwcd_clipid;
    int         pwcd_damagedid;
    int         pwcd_lockcount;
    struct      pixrect *pwcd_prmulti;
    struct      pixrect *pwcd_prsingle;
    struct      pixwin_prlist *pwcd_prl;
    struct      rectlist pwcd_clippingsorted[RECTS_SORTS];
    struct      rect *pwcd_regionrect;
};

#define PWCD_NULL               0
#define PWCD_MULTIRECTS         1
#define PWCD_SINGLERECT         2
#define PWCD_USERDEFINE         3

struct pixwin_prlist {
    struct      pixwin_prlist *prl_next;
    struct      pixrect *prl_pixrect;
    int         prl_x, prl_y;
};
```

`pwcd_windowfd` is a file descriptor for the window being accessed. Within the owning process, it is the standard handle on a window. A description of the interplay between windows and pixwins continues in ***Pixwin Creation and Destruction.*** The portions of the window's area accessible through the pixwin are described by the rectlist `pwcd_clipping` . `pwcd_regionrect`, if not NULL, points to a ***rect*** that is intersected with `pwcd_clipping` to further restrict the portions of the window's area accessible through the pixwin. `pwcd_clipid` and `pwcd_damagedid` identify the most recent rectlists retrieved for a window. `pwcd_lockcount` is a reference count used for nested locking, as described in ***Locking*** below. Copies of `pwcd_clipping`, sorted in directions convenient for copy operations, are stored in `pwcd_clippingsorted`.

`pwcd_state` can be one of the following:

Table 3-1: Clipping State

State	**Meaning**
PWCD_NULL	no part of window visible
PWCD_MULTIRECTS	must clip to multiple rectangles
PWCD_SINGLERECT	need clip to only one rectangle
PWCD_USERDEFINE	the client program will be responsible for setting up the clipping

`pwcd_prmulti` is the pixrect for clipping during drawing when there are multiple rectangles involved in the clipping. `pwcd_prsingle` is the pixrect for clipping during drawing when there is only one rectangle visible.

`pwcd_prl` is a list of pixrects that may be used for clipping when there are multiple rectangles involved. For vector drawing, these clippers *must* be used to maintain stepping integrity across abutting rectangle boundaries. The `prl_x` and `prl_y` fields in the `pixwin_prlist` structure are offsets from the window origin for the associated `prl_pixrect`.

3.2.4. Pixwin_clipops Struct

```
struct   pixwin_clipops {
    int          (*pwco_lock)(),
    int          (*pwco_unlock)(),
    int          (*pwco_reset)(),
    int          (*pwco_getclipping)();
};
```

The `pw_clipops` struct is a vector of pointers to system-provided procedures that implement correct screen access. These are accessed through macros described in ***Locking and Clipping***.

3.3. Pixwin Creation and Destruction

To create a pixwin, the window to which it will refer must already exist. This task is accomplished with procedures like `win_getnewwindow` and `win_setrect`, described in ***Window Manipulation***, or, at a higher level, `tool_create` and `tool_createsubwindow`, described in ***Suntool: Tools and Subwindows***. The pixwin is then created for that window by a call to `pw_open`:

```
struct  pixwin *pw_open(fd)
    int          fd;
```

`pw_open` takes a file descriptor for the window on which the pixwin is to write. A pointer to a pixwin struct is returned. At this point the pixwin describes the exposed area of the window. If the client wants a ***retained pixwin***, `pw_prretained` should be set to point to an appropriately-sized memory pixrect after `pw_open` returns.

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

```
    pw_close(pw)
        struct      pixwin *pw;
```

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

3.3.1. Region Creation

One can use pixwins to clip rectangular regions within a window's own rectangular area. The ***region*** operation creates a new pixwin that refers to an area within an existing pixwin:

```
    struct   pixwin *pw_region(pw, x, y, w, h)
        struct      pixwin *pw;
        int         x, y, w, h;
```

The pixwin which is to serve as the source is addressed by `pw`; `x`, `y`, `w` and `h` describe the rectangle to be included in the new pixwin. The upper left pixel in the returned pixwin is at coordinates (0,0); this pixel has coordinates (`x`, `y`) in the source pixwin.

3.4. Locking and Clipping

Before a window process reads from or writes to the screen, it must satisfy several conditions:

- It should obtain exclusive use of the display hardware,
- The position of windows on the screen should be frozen,
- The window's description of what portions of its window are visible should be up-to-date, and
- The window should confine its activities to those visible areas.

The first three of these requirements is met by ***locking***; the last amounts to ***clipping*** the image the window will write to the bounds of its ***exposed*** area. All are handled implicitly by the access routines described in ***Accessing a Pixwin's Pixels.***

3.4.1. Locking

Locking allows a client program to obtain exclusive use of the display.

Making correct and judicious use of explicit display locking is EXTREMELY important for getting the best display speed possible.

Note that if the client program does not obtain an explicit lock, the window system will. For example, if an application program is to draw one hundred lines, it can either explicitly lock the display once, draw the lines, and unlock explicitly, or it can ignore locking and simply draw the lines. In the latter case, the window system will perform locking and unlocking around each drawing operation, in effect acquiring and releasing the lock one hundred times instead of once.

For efficiency's sake, application programs should lock explicitly around a body of screen access operations.

The `pw_lock` macro:

```
pw_lock(pw, r)
    struct      pixwin *pw;
    struct      rect *r;
```

uses the lock routine pointed to by the window's `pw_clipops` to acquire a lock for the user process that made this call. `pw` addresses the pixwin to be used for the ouput; `r` is the rectangle in the window's coordinate system that bounds the area to be affected. `pw_lock` blocks if the lock is unavailable, for example, if another process currently has the display locked.

Lock operations for a single pixwin may be nested; inner lock operations merely increment a count of locks outstanding, `pwcd_lockcount` in the window's `pw_clipdata` struct. Their affected rectangles must lie within the original lock's.

A similar macro is:

```
pw_unlock(pw)
    struct      pixwin *pw;
```

which decrements the lock count. If this brings it to 0, the lock is actually released.

Since locks may be nested, it is possible for a client procedure to find itself, especially in error handling, with a lock which may require an indefinite number of ***unlock*s**. To handle this situation cleanly, another routine is provided. The following macro sets `pw`'s lock count to 0 and releases its lock:

```
pw_reset(pw)
    struct      pixwin *pw;
```

Like `pw_lock` and `pw_unlock`, `pw_reset` calls a routine addressed in the pixwin's `pixwin_clipops` struct, in this case the one addressed by `pwco_reset`.

Acquisition of a lock has the following effects:

- If the cursor is in conflict with the affected rectangle, it is removed from the screen. While the screen is locked, the cursor will not be moved in such a way as to disrupt any screen accessing.
- Access to the display is restricted to the process acquiring the lock.
- Modification of the database that describes the positions of all the windows on the screen is prevented.
- The id of the most recent clipping information for the window is retrieved, and compared with that stored in `pwcd_clipid` in the pixwin's `pw_clipdata`. If they differ, the routine addressed by `pwco_getclipping` is invoked, to make all the fields in `pw_clipdata` accurately describe the area which may be written into.
- Once the correct clipping is in hand, the `pwcd_state` variable's value determines how to set `pw_ops`, `pw_opshandle`, `pw_opsx` and `pw_opsy`. This setting is done in anticipation of further screen access operations being done before a subsequent unlock. These values can often be set to bypass the pixwin software by going directly to the pixrect level.

Nested locking is cheap, but initial locking is moderately expensive as it involves two system calls. Clients with a group of screen updates to do can gain noticeably by surrounding the group with lock-unlock brackets; then the locking overhead will only be incurred once. An example of such a group is displaying a line of text, or a series of vectors with pre-computed endpoints.

While it has the screen locked, a process should ***not***:

- do any significant computation unrelated to displaying its image;
- invoke any system calls, including other I/O, which might cause it to block; or
- invoke any pixwin calls except `pw_unlock` and those described in ***Accessing a Pixwin's Pixels***. In any case, the lock should not be held longer than about a quarter of a second, even following all these guidelines.

As a deadlock resolution approach, when a display lock is held for more than 10 seconds, the lock is broken. However, the offending process is not notified by signal; the idea is that a process shouldn't be aborted for this infraction. A message is displayed on the console.

3.4.2. Clipping

Output to a window is clipped to the window's `pwcd_clipping` rectlist; this is a series of rectangles which, taken together, cover the valid area that this window may write to. There are two routines which set the pixwin's clipping:

```
pw_exposed(pw)
     struct      pixwin *pw;

pw_damaged(pw)
     struct      pixwin *pw;
```

`pw_damaged` is discussed in ***Damage***. `pw_exposed` is the normal routine for discovering what portion of a window is visible. It retrieves the rectlist describing that area into the pixwin's `pwcd_clipping`, and stores the id identifying it in `pwcd_clipid`. It also stores its own address in the pixwin's `pwco_getclipping`, so that subsequent lock operations will get the correct area description.

Clipping, even more than locking, should normally be left to the library output routines. For the intrepid, the strategy these routines follow is briefly sketched here; the ***rectlist*** data structures and procedures in Appendix A are required reading.

Some procedure will set the pixwin's `pwcd_clipping` so that it contains a ***rectlist*** describing the region which may be painted. This is done by a lock operation which makes a call through `*pwco_getclipping`, or an explicit call to one of `pw_open`, `pw_donedamaged`, `pw_exposed` or `pw_damaged`. This ***rectlist*** is essentially a list of rectangular fragments which together cover the area of interest. As an image is generated, portions of it which lie outside the rectangle list must be masked off, and the remainder written to the window through a pixrect.

The clipping aid `pwcd_prmulti` is set up to be a pixrect which clips for the entire rectangular area of the window. Any clipping using this pixrect must utilize the information in `pwcd_clipping` to do the actual clipping to multiple rectangles.

`pwcd_prl` is set up to parallel each of the rectangles in `pwcd_clipping`. Thus, if one draws to each of the pixrects in this data structure, the image will be correctly clipped. `pwcd_state` is set by examining the makeup of the `pwcd_clipping`. If `pwcd_state` is

PWCD_SINGLERECT, a pixrect is set up in `pwcd_prsingle` also. When this case exists, after `pw_lock` and before `pw_unlock`, most screen accesses will directly access the pixrect level of software. Thus, in this common case, screen access is as fast in the window system as it is on the raw pixrect software outside of the window system. Also, `pwcd_prsingle` is set up with a zero height and width pixrect when `pwcd_state` is PWCD_NULL.

As an escape, none of the pixrect setup described above takes place when `pwcd_state` is PWCD_USERDEFINE. This means that clipping is the responsibility of higher level software.

A client may write to the display with an operation which specifies no clipping:

```
(op | PIX_DONTCLIP)
```

This means that it is doing the clipping at a higher level. Note that clipping data is only valid during the time the client may write to the screen, that is when the window's owner process holds a lock on the screen. If the clipping is done wrong, it is possible to damage another window's image. In particular, the client must clip to *all* of the rectangles in the rectlist, not just the bounding rectangle for the rectlist.

3.5. Accessing a Pixwin's Pixels

Procedures described in this section provide all the normal facilities for output to a window and should be used unless there are special circumstances. Each contains a call to the standard lock procedure, described in *Locking*. Each takes care of clipping to the *rectlist* in `pw_clipping`. Since the routines are used both for painting new material in a window and for repairing damage, they make no assumption about what clipping information should be gotten. Thus, there should be some previous call to either `pw_open`, `pw_donedamaged`, `pw_exposed` or `pw_damaged`, to initialize `pwo_getclipping` correctly.

The procedures described in this section will maintain the memory pixrect for a retained pixwin. That is, they check the window's `pw_prretained`, and if it is not NULL, perform their operation on that data in memory, as well as on the screen.

3.5.1. Write Routines

```
pw_write(pw, xd, yd, width, height, op, pr, xs, ys)
    struct      pixwin *pw;
    int         op, xd, yd, width, height, xs, ys;
    struct      pixrect *pr;

pw_writebackground(pw, xd, yd, width, height, op)
```

Pixels are written to the pixwin `pw` in the rectangle defined by `xd`, `yd`, `width`, and `height`, using rasterop function `op` (as defined in *Constructing an Op Argument*). They are taken from the rectangle with its origin at `xs`, `ys` in the source pixrect pointed to by `pr`. `pw_writebackground` simply supplies a null `pr` which indicates that an infinite source of pixels, all of which are set to zero, is used. The following draws a pixel of `value` at (`x, y`) in the addressed pixwin:

```
pw_put(pw, x, y, value)
    struct      pixwin *pw;
    int         x, y, value;
```

The next draws a vector of pixel `value` from (x0, y0) to (x1, y1) in the addressed pixwin using rasterop `op`:

```
pw_vector(pw, x0, y0, x1, y1, op, value)
    struct      pixwin *pw;
    int         op, x0, y0, x1, y1, value;
```

`pw_rop` performs the indicated rasterop from source to destination:

```
pw_rop(dpw, dx, dy, w, h, op, sp, sx, sy)
        struct  pixwin *dpw;
        struct  pixrect *sp;
        int     dx, dy, w, h, op, sx, sy;
```

For further information, please see *Rop: RasterOp Source to Destination.*

```
pw_replrop(pw, xd, yd, width, height, op, pr, xs, ys)
    struct      pixwin *pw;
    int         op, xd, yd, width, height;
    struct      pixrect *pr;
    int         xs, ys;
```

This procedure uses the indicated raster op function to replicate a pattern (found in the source pixrect) into a destination in a pixwin. For a full discussion of the semantics of this procedure, refer to the description of the equivalent procedure `pr_replrop` in *Pixel Data and Operations.* The following two routines:

```
pw_text(pw, x, y, op, font, s)
    struct      pixwin *pw;
    int         x, y, op;
    struct      pixfont *font;
    char        *s;

pw_char(pw, x, y, op, font, c)
    struct      pixwin *pw;
    int         x, y, op;
    struct      pixfont *font;
    char        c;
```

write a string of characters and a single character respectively, to a pixwin, using rasterop `op` as above. `pw_text` and `pw_char` are distinguished by their own coordinate system: the destination is given as the left edge and ***baseline*** of the first character. The left edge does not take into account any ***kerning*** (character position adjustment depending on its neighbors), so it is possible for a character to have some pixels to the left of the x-coordinate. The baseline is the y-coordinate of the lowest pixel of characters without descenders, 'L' or 'o' for example, so pixels will frequently occur both above and below the baseline in a string. `font` may be NULL in which case the ***system font*** is used.

The system font is the same as the font returned from `pf_default`. In addition, the system font is reference counted and shared between software packages. To get the system font call `pw_pfsysopen`:

```
struct  pixfont *pw_pfsysopen()
```

When you are done with the system font call `pw_pfsysclose`:

```
pw_pfsysclose()
```

Note: A font to be used in `pw_text` is required to have the same `pc_home.y` and character height for all characters in the font.

The following routine:

```
pw_ttext(pw, x, y, op, font, s)
    struct      pixwin *pw;
    int         x, y, op;
    struct      pixfont *font;
    char        *s;
```

is just like `pw_text` except that it writes ***transparent*** text. Transparent text writes the shape of the letters without disturbing the background behind it. This is most useful with color pixwins. Monochrome pixwins can use `pw_text` and a PIX_SRC ¦ PIX_DST op, which is faster.

Applications such as displaying text perform the same operation on a number of pixrects in a fashion that is amenable to global optimization. The batchrop procedure is provided for these situations:

```
pw_batchrop(pw, dx, dy, op, items, n)
    struct      pixwin *pw;
    int         dx, dy, op, n;
    struct      pr_prpos items[ ];
```

`pw_batchrop` is analogous to `pr_batchrop` described in *Pixel Data and Operations.* Please refer to that section for a detailed explanation of `pw_batchrop`.

Stencil ops are like raster ops except that the source pixrect is written through a stencil pixrect which functions as a spatial write enable mask. The indicated raster operation is applied only to destination pixels where the stencil pixrect is non-zero. Other destination pixels remain unchanged.

```
pw_stencil(dpw, dx, dy, dw, dh, op, stpr, stx, sty, spr, sx, sy)
    struct      pixwin *dpw;
    struct      pixrect *stpr, *spr;
    int         dx, dy, dw, dh, op, stx, sty, sx, sy;
```

`pw_stencil` is exactly analogous to `pr_stencil` described in *Pixel Data and Operations.* Refer there for a detailed explanation of `pw_stencil`.

3.5.2. Drawing A Polygon within a Pixwin

The following macro draws a polygon within a pixwin:

```
pw_polygon_2(pw, dx, dy, nbds, npts, vlist, op, spr, sx, sy)
    struct pixwin *pw;
    int dx, dy;
    int nbds;
    int npts[];
    struct pr_pos *vlist;
    int op;
    struct pixrect *spr;
    int sx, sy;
```

You can create a polygon filled with a solid or textured pattern. `pw_polygon_2` is analogous to `pr_polygon_2` described in ***Pixel Data and Operations***. Refer to `pr_polygon_2` for further details on this procedure.

3.5.3. Draw Curved Shapes

`pw_traprop` is a pixwin operation analogous to `pw_rop`. The main difference is that `pw_traprop` operates on a trapezon rather than a rectangle. Refer to the section ***Draw Curved Shapes (pr_traprop)*** for detailed information about trapezons.

The function

```
pr_traprop(dpw, dx, dy, t, op, spr, sx, sy)
    struct pixwin       *dpw;
    struct pr_trap      t;
    struct pixrect      *spr;
    int                 dx, dy, op, sx, sy;
```

writes the source pixrect (`spr`) into the destination pixwin (`dpw`) via the operation `op`. `op` works in the same manner as `pw_rop`. The function then clips the output to the trapezon `t`.

3.5.4. Read and Copy Routines

The following routines use the window as a source of pixels. They may find themselves thwarted by trying to read from a portion of the pixwin which is hidden, and therefore has no pixels. When this happens, `pw_fixup` 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 `pw_prretained`. The following returns the value of the pixel at (`x, y`) in the addressed pixwin:

```
pw_get(pw, x, y)
    struct      pixwin *pw;
    int         x, y,;
```

Pixels are read from the pixwin into a pixrect by:

```
pw_read(pr, xd, yd, width, height, op, pw, xs, ys)
    struct      pixwin *pw;
    int         op, xd, yd, width, height, xs, ys;
    struct      pixrect *pr;
```

Pixels are read from the rectangle defined by `xs`, `ys`, `width`, `height`, in the pixwin pointed to by `pw`, using rasterop function `op`. The pixels are stored in the rectangle with its origin at `xd`, `yd` in the pixrect pointed to by `pr`.

Copy is used when both source and destination are pixwins:

```
pw_copy(dpw, xd, yd, width, height, op, spw, xs, ys)
    struct      pixwin *dpw, *spw;
    int op, xd, yd, width, height, xs, ys;
```

Note: Currently `dpw` and `spw` must be the same pixwin.

3.5.5. Bitplane Control

For pixwins on color display devices, one must be able to restrict access to certain bitplanes.

```
pw_putattributes(pw, planes)
    struct      pixwin *pw;
    int         *planes;
```

`planes` is a bitplane access enable mask. Only those bits of the pixel corresponding to a 1 in the same bit position of `*planes` will be affected by pixwin operations. `pw_putattributes` sets the access enable mask of `pw`. If the `planes` argument is NULL, that attribute value will not be written.

Note: Use `pw_putattributes` with care; it changes the internal state of the pixwin until `pw_putattributes` is next called. Don't forget to restore the internal state once through accessing in this special mode.

```
pw_getattributes(pw, planes)
    struct pixwin *pw;
    int           *planes;
```

retrieves the value of the access enable mask into `*planes`.

3.6. Damage

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 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.

3.6.1. Handling a SIGWINCH Signal

Note: it is a common programming error to try to access the pixwin at the time a SIGWINCH is received, rather than after returning from the SIGWINCH handler. Please read this section and avoid this problem.

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` system call, so it is possible to awaken a client that was blocked, and with a little investigation, discover the cause of the SIGWINCH. See the `select`(2) system call and refer to the `tool_select` mechanism in ***Tool Processing*** for an example of this approach.

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 ***Window Manipulation***) 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.

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)
    struct      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***.

NOTE: 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 Appendix A 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)
    struct      pixwin *pw;
```

When the image is repaired, the client should inform the window system with a call to:

```
pw_donedamaged(pw)
    struct      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`.

3.7. Colormap Manipulation

Pixwins provide an interface to a basic colormap sharing mechanism. Portions of the colormap, ***colormap segments***, are named and can be shared among cooperating processes. Use of a colormap segment, as opposed to the entire colormap, is essentially invisible to clients. Routines that access a pixwin's pixels do not distinguish between windows which use colormap segments and those which use the entire colormap.

3.7.1. Initialization

`pw_open` and `pw_region` both create and return a pixwin. If a colormap segment is already defined for the window of the pixwin, this is the colormap segment used in the new pixwin. However, if the window has no colormap segment defined for it, the ***default colormap segment*** is setup for the pixwin.

The default colormap segment is usually the monochrome colormap segment defined in `<sunwindow/cms_mono.h>`. However, the default colormap segment can be programmatically changed.

```
#define CMS_NAMESIZE    20

struct colormapseg {
    int         cms_size;
    int         cms_addr;
    char        cms_name[CMS_NAMESIZE];
};

struct   cms_map {
    unsigned char       *cm_red;
    unsigned char       *cm_green;
    unsigned char       *cm_blue;
};
```

```
pw_setdefaultcms(cms, map)
    struct      colormapseg *cms;
    struct      cms_map *map;
```

`pw_setdefaultcms` copies the data in `cms` and `map` to serve as the default colormap segment. `cms->cms_name` is the name of the colormap segment (more on names below) and `cms->cms_size` is its size (`cms->cms_addr` should be 0). There are `cms->cms_size` bytes in each of the arrays of `map`. A −1 is returned if `cms->cms_size` is greater than 256. Otherwise, 0 is returned.

```
pw_getdefaultcms(cms, map)
    struct      colormapseg *cms;
    struct      cms_map *map;
```

`pw_getdefaultcms` copies the data in the default colormap segment into the data pointed to by `cms` and `map`. Before the call, the byte pointers in `map` should be initialized to arrays of size `cms->cms_size`. A −1 is returned if `cms->cms_size` is less than the size of the default colormap segment. Otherwise, 0 is returned.

Note: the correct way to access an existing pixwin's colormap is via `pw_putcolormap` and `pw_getcolormap`.

3.7.2. Background and Foreground

Every colormap segment has two distinguished values, its ***background*** and ***foreground***. The background color is defined as the value at the first position of a colormap segment. The foreground color is defined as the value at the last position of a colormap segment (the colormap segment's size minus 1).

The foreground is important in terms of color/monochrome compatibility. Any source color, other than 0, that is written on a monochrome pixrect is translated to the foreground color.

`pw_open` sets the background and foreground of the returned pixwin to be those of the default colormap segment if the pixwin's window has not defined a colormap segment. `pw_region` inherits the background and foreground of the source pixwin.

Here are handy utilities to set two specific colormap segment entries:

```
pw_reversevideo(pw, min, max)
    struct      pixwin *pw;
    int         min, max;

pw_blackonwhite(pw, min, max)
    struct      pixwin *pw;
    int         min, max;

pw_whiteonblack(pw, min, max)
    struct      pixwin *pw;
    int         min, max;
```

`min` and `max` should be the first and last entries, respectively, in the colormap segment. If `min` is the background and `max` is the foreground and `pw` is a color pixwin, these calls do nothing.

3.7.3. A New Colormap Segment

Changing a pixwin's colormap requires two steps. First, the colormap segment must be appropriately named (see `pw_setcmsname`, below). Second, the colormap segment is loaded with the actual colors desired (see `pw_putcolormap`, covered in the next section).

If a colormap segment is not to be shared by another window then the name should be unique. One would certainly want a unique colormap segment if that segment was to be used for colormap animation. A common way to generate a unique name is to append your process id to a more meaningful string that describes the usage of the colormap segment.

If a colormap segment's usage is static in nature, by all means try to use a shared colormap segment definition. There are three basic types of shared colormap segments:

- A colormap segment used by a single program. Sharing occurs when multiple instances of the same program are running. An example of such a program might be a color terminal emulator in which the terminal has a fixed selection of colors.
- A colormap segment used by a group of highly interrelated programs. Sharing occurs whenever two or more programs of this group are running at the same time. An example of such a group might be a series of CAD/CAM programs in which it is common to have multiple programs running at the same time.
- A colormap segment used by a group of unrelated programs. Sharing occurs whenever two or more programs of this group are running. An example of such a colormap segment is CMS_MONOCHROME, as defined in `<sunwindow/cms_mono.h>`. This colormap segment is, by convention, the default colormap. Examples of other colormap segment definitions that could be shared with other windows are in `<sunwindow/cms_*.h>`. These are `cms_rgb.h`, `cms_grays.h`, `cms_mono.h`, and `cms_rainbow.h`.

Remember that colormap entries are scarce so try to share them.

```
pw_setcmsname(pw, cmsname)
    struct      pixwin *pw;
    char        cmsname[CMS_NAMESIZE];
```

`cmsname` is the name that `pw` will call its window's colormap segment. Just setting the name resets the colormap segment to a NULL entry. Usually, the very next call after `pw_setcmsname` should be `pw_putcolormap`, to set the size of the colormap (see section following).

Colormap segments are associated with windows, not pixwins. Each window can have only one colormap segment. Pixwins provide an interface for managing that one colormap segment. Since more than one pixwin may exist per window, care should be taken to avoid changing the colormap segment definition out from underneath another pixwin on the same window.

```
pw_getcmsname(pw, cmsname)
    struct      pixwin *pw;
    char        cmsname[CMS_NAMESIZE];
```

The colormap segment name of `pw` is copied into `cmsname`.

3.7.4. Colormap Access

```
pw_putcolormap(pw, index, count, red, green, blue)
    struct      pixwin *pw;
    int         index, count;
    unsigned char       red[ ], green[ ], blue[ ];
```

Note: Before accessing the colormap, you must call `pw_setcmsname`.

The `count` elements of the pixwin's colormap segment starting at `index` (zero origin) are loaded with the first `count` values in the three arrays. A colormap has three components each indexed by a given pixel value to produce an RGB color. Monochrome pixwins assume red equals green equals blue. Pixrects of depth 8 have colormaps with 256 (2 to the eighth) entries. Background and foreground values are forced to the values defined by the screen if they are the same.

```
pw_getcolormap(pw, index, count, red, green, blue)
    struct      pixwin *pw;
    int         index, count;
    unsigned char       red[ ], green[ ], blue[ ];
```

finds out the state of the colormap segment. The arguments are analogous to those of `pw_putcolormap`.

The utility:

```
pw_cyclecolormap(pw, cycles, index, count)
    struct      pixwin *pw;
    int cycles, index, count;
```

is handy for taking a portion of `pw`'s colormap segment, starting at `index` for `count` entries, and rotating those entries among themselves for `cycles`. A cycle is defined as the `count` shifts it takes one entry to move through every position once.

3.7.5. 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 (see `win_grabio`), 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)
         struct      pixwin *pw;
         struct      rect *r;
```

`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.

Chapter 4

Window Manipulation

This chapter describes the ***sunwindow*** facilities for creating, positioning, and controlling windows. It continues the discussion begun in ***Overlapped Windows: Imaging Facilities***, on the ***sunwindow*** level that allows displaying images on windows which may be overlapped.

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`. It is manipulated by other I/O calls, such as `select`, `read`, `ioctl`, and `close`. `write` to a window is not defined, since all the facilities of the previous chapter on ***Overlapped Windows: Imaging Facilities*** are required to display output on a window.

The header file `<sunwindow/window_hs.h>` includes the header files needed to work at this level of the window system.

4.1. Window Data

The information about a window maintained by the window system includes:

- two rectangles which refer to alternative ***sizes*** and ***positions*** for the window on the screen;
- a series of links that describe the window's position in a hierarchical database, which determines its ***overlapping*** relationships to other windows;
- clipping information used in the processing described in ***Overlapped Windows: Imaging Facilities***;
- the image used to track the mouse when it is in the window;
- the id of the process which should receive SIGWINCH signals for the window (this is the ***owner*** process);
- a mask that indicates what user input actions the window should be notified of;
- another window, which is given any input events that this window does not use; and
- 32 bits of data private to the window client.

4.2. 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.

4.2.1. *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. This is the standard `close`(2) system call with the window's file descriptor as argument. As with other file descriptors, a window left open when its owning process terminates 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 a file descriptor it got by opening `/dev/win0`. 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.

4.2.2. *An Existing Window*

It is possible for more than one process to have a window open at the same time; ***Providing for*** *Naive Programs* 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.***

4.2.3. *References to Windows*

Within the process which created a window, the usual handle on that window is the file descriptor returned by `open` and `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 switching the various window identifiers back and forth. `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;
```

The following returns the window number for the window whose file descriptor is `windowfd`:

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

4.3. 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 user-interface actions indicate that it is to be moved or stretched, for instance. The basic procedures are:

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

win_getsize(windowfd, rect)
    int         windowfd;
    struct      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 the first argument into the rect addressed by the second argument; the origin is relative to that window's parent. *Setting Window Links* explains what is meant by a 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. `win_setrect` copies the rect argument's data into the rect of the indicated window:

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

This changes its size and/or position on the screen. The coordinates are in the coordinate system of the window's parent.

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

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

A window may have an alternate size and location; this facility is useful for ***icons*** (see ***Icons***). The alternate rectangle may be read with `win_getsavedrect`, and written with `win_setsavedrect`. As with `win_getrect` and `win_setrect`, the coordinates are relative to the window's parent.

4.4. 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.

4.4.1. 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. 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. 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 ***Passing Parameters to a Tool*** for an example of this environment parameter's usage.

4.4.2. 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 typically will result in a cleaner display interaction.

Once a window has been inserted in the window database, it is available for input and output. At this point, it is appropriate to call `pw_open` and access the screen.

4.4.3. 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.

Note that if a client program uses any of the window manager routines, use of `win_lockdata` and `win_unlockdata` is not necessary. See *Window Management* in Chapter 9 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 10 seconds of real time inside a window data lock, and the lock is broken.

4.5. 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 client programmer use. This data 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.

4.6. Minimal-Repaint Support

This section has strong connections to the preceding chapter and to Appendix A 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;
    struct       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.

4.7. Multiple Screens

Multiple displays may be simultaneously attached to a workstation, and clients may want windows on all of them. Therefore, the window database is a forest, with one tree of windows for each display. Thus, 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.

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

The `screen` structure describes a client's notion of the display screen. There are also fields indicating the input devices associated with the screen. `scr_rootname` is the device name of the window which is at the base of the window display tree for the screen; the default is `/dev/win0`. `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`. `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. `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.

`win_screennew:`

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

opens and returns a window file descriptor for a root 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.

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

`win_screenget:`

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

fills in the addressed struct `screen` with information for the screen with which the window indicated by `windowfd` is associated.

`win_screendestroy`:

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

causes each window owner process (except the invoking process) on the screen associated with `windowfd` to be sent a SIGTERM signal.

`win_setscreenpositions` informs the window system of the logical layout of multiple screens:

```
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
```

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];
```

`win_setkbd`:

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

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

`win_setms`:

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

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

`win_initscreenfromargv`:

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

can be used to do a standard command line parse of `argv` into `*screen`. `*screen` is first zeroed. The syntax is:

[–d *display device*] [–m *mouse device*] [–k *keyboard device*] [–i] [–f *red green blue*] [–b *red green blue*]

See `suntools(1)` for semantics and details.

4.8. Cursor and Mouse Manipulations

This section describes the interface to the mouse and the cursor that follows the mouse. Both of these are maintained by the window system internals.

4.8.1. Cursors

The cursor is the image which tracks the mouse on the screen:

```
struct cursor  {
    short       cur_xhot, cur_yhot;
    int         cur_function;
    struct      pixrect *cur_shape;
};
#define CUR_MAXIMAGEWORDS       16
```

`cur_shape` points to a memory pixrect which holds the actual image for the cursor. The window system supports a `cur_shape.pr_data->md_image` up to CUR_MAXIMAGEWORDS words. This means that a cursor has a maximum size of 256 pixels, due to alignment constraints inherent in memory pixrects. The pixels in a cursor are usually arranged 16 x 16, although 8 pixels high by 32 wide is also possible.

The "hot spot" defined by (`cur_xhot, cur_yhot`) associates the cursor image, which has height and width, with the mouse position, which is a single point on the screen. The hot spot gives the mouse position an offset from the upper-left corner of the cursor image.

Most cursors have a hot spot whose position is dictated by the image shape: the tip of an arrow, the center of a bullseye, the center of a cross-hair. Cursors can also be used as a status feedback mechanism, an hourglass to indicate that some processing is occurring for instance. This type of cursor should have the hot spot located in the middle of its image so the user has a definite spot for pointing and does not have to guess where the hot spot is.

The function indicated by `cur_function` is a rasterop (as described in *Constructing an Op Argument*), which will be used to paint the cursor. PIX_SRC | PIX_DST is generally effective on light backgrounds, for example in text, but invisible over solid black. PIX_SRC ^ PIX_DST is a reasonable compromise over many different backgrounds, although it does poorly over a gray pattern.

```
win_getcursor(windowfd, cursor)
    int         windowfd;
    struct      cursor *cursor;
```

stores a copy of the cursor that is currently being used on the screen into the buffer addressed by `cursor`. Note that the caller must have set `cursor->cur_shape` to point to a pixrect large enough to hold the cursor image.

```
win_setcursor(windowfd, cursor)
    int         windowfd;
    struct      cursor *cursor;
```

sets the cursor and function that will be used whenever the mouse position is within the indicated window.

If a window process does not want a cursor displayed, the appropriate mechanism is to set the cursor to one whose dimensions are both 0.

Use the following macro as an aid in making your own cursor:

```
DEFINE_CURSOR_FROM_IMAGE(name, hot_x, hot_y, func, image)
```

This macro makes a cursor that is 16 bits wide by 16 bits high. It generates several static structures. The first argument to the macro is the name that will be given to the cursor struct. The second and third arguments are the x and y positions of the hotspot relative to the upper-lefthand corner of the cursor shape. The fourth argument is the RasterOp function used to display the cursor, and the final argument is an array which contains 16 shorts that are the bit pattern of the cursor image. Typically this array will be declared as follows

```
static short cursor_image[] = {
#include "file_generated_by_icontool"
};
```

For example, DEFINE_CURSOR_FROM_IMAGE might be used as follows:

```
#include <suntool/win_cursor.h>
static short hour_glass_image[] = {
#include "hourglass.pr"
};
DEFINE_CURSOR_FROM_IMAGE(hour_glass, 8, 8,
                         PIX_SRC|PIX_NOT, hour_glass_image);
```

This defines a cursor called `hour_glass` which could then be used in some window. For example,

```
win_setcursor(windowfd, &hour_glass);
```

As an alternative, use the following macro; it takes the actual shorts for the image (i1 through i16) rather than the array.

```
DEFINE_CURSOR(name, hot_x, hot_y, func, i1, i2, i3, i4, i5, i6,
              i7, i8, i9, i10, i11, i12, i13, i14, i15, i16)
```

Note that due to the restictions imposed by the C pre-processor, you cannot use a `#include` in the call to the DEFINE_CURSOR macro to obtain `i1` through `i16` from a file.

4.8.2. Mouse Position

Determining the mouse's current position is treated under ***Input to Application Programs.*** We note here that the standard procedure for a process to track the mouse is to arrange to receive an input event every time the mouse moves; and in fact, the mouse position is passed with ***every*** user input 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;
```

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, window entry and exit and cursor changes, will be generated. This facility should be used with restraint, as users are likely to lose a cursor that moves independently 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`) define a screen position in that window's coordinate space. The returned value is a window number. `x` and `y` may lie outside the bounds of the window.

4.9. Providing for Naive Programs

There is a large class of applications that are relatively unsophisticated about the window system, but want to run in windows anyway. For example, a simple-minded 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.

4.9.1. Which Window to Use

SunWindows 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 is already 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.

4.9.2. 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. *Note:* It's a bad idea to take over an existing window using `win_setowner`.

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

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

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

If the process that created the blanket window dies before `win_removeblanket` can be called, the blanket window will automatically be removed and destroyed upon automatic closure of the window device. This automatic closure happens because the only open file descriptor on the window will be in the creating process.

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

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

4.10. Window Ownership

Note: Do not use the two routines in this section for ***temporarily*** taking over another window. These routines are included for backwards compatibility reasons.

SIGWINCH signals are directed to the process that ***owns*** the window, the owner normally being the process that created the window. The following procedures may read from and write to the window:

```
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.

4.11. 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` call that detected the error. 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` number that defines the window operation that generated the error. See *Error Message Decoding* in *Programming Notes* in the appendix.

Chapter 5

Input to Application Programs

This chapter continues the description of the ***sunwindow*** level of the Sun window system. Here we discuss how user input is made available to application programs. Unless otherwise noted, the structures and procedures discussed in this section are found in the header file `/usr/include/sunwindow/win_input.h`.

The window system provides facilities which meet two distinct needs regarding input to an application program:

- A uniform interface to multiple input devices allows programs to deal with varying keyboards and positioning devices, ignoring complexities due to facilities which the programs do not use.
- Several different keyboards are available with Sun systems; they differ in the number and arrangement of keys. At a minimum, some clients will require ASCII characters, one per keystroke. More sophisticated clients will assign special values to non-standard keys (such as "META" characters in the range 0x80 and above). Some clients will assign functions to particular keys on the keyboard, and will distinguish key-down from key-up events.
- The standard positioning device on a Sun workstation is the mouse, which reports a location and the state of three buttons. Alternatively, some clients may use a tablet and stylus, or in place of the stylus, a "puck" with as many as 10 buttons on it.
- In some client systems, the time between input events is significant; for example, when smoothing a user's stylus trace, or assigning special meaning to multiple clicks of a button within a short period.

The window system allows clients with only the simplest requirements to ignore all the complications, while providing more sophisticated clients the facilities they require. The mechanism for accomplishing this is called the ***virtual input device***. This mechanism with its input events is described in *Virtual Input Device*.

The second major section of this chapter describes how user inputs are collected from multiple sources, serialized, and distributed among multiple consumers. Multiple clients are able to accept inputs concurrently, and a slow consumer does not affect other clients' ability to receive their inputs. Type-ahead and mouse-ahead are fully supported.

- Client programs operate under the illusion that they have the user's full attention, leaving the window system to handle the multiplexing. Therefore, a client sees precisely those input events that the user has directed to that application.
- Conversely, the client may require inputs from multiple devices, where the exact sequences across all those devices is significant. The order of mouse and function key events is likely to be significant, for instance. This is provided for via a single unified input stream, rather than requiring polling of multiple streams, which would be unacceptable in a multi-processed environment.

- The distribution of input events takes into account the window's indication of what events it is prepared to handle; other events are redirected, allowing a division of labor among the various components of a system.

5.1. The Virtual Input Device

This section describes the virtual device which generates user input, and how the input is presented to the client process. The device appears as an extended keyboard, different from existing keyboards, but incorporating the common features of most of them. It also incorporates a *locator* which indicates a screen position, and a clock which reports a time in seconds and microseconds.

5.1.1. *Uniform Input Events*

Each user action generates an ***input event***, which is reported in a uniform format regardless of the event. An event is reported in the following struct:

```
struct  inputevent  {
        short       ie_code;
        short       ie_flags;
        short       ie_shiftmask;
        short       ie_locx;
        short       ie_locy;
        struct      timeval ie_time;
};
```

`ie_code` identifies the source of the event, as a switch position on a Virtual Input Device. The exact definition of the codes is given in *Event Codes*. In general, the input events fall into one of three classes: events that generate a single ASCII character; events related to locator motion and window geometry; and events identified with invocation of a special function, usually involving the depression or release of a single special button on the mouse or keyboard. These classes are known as ASCII, pseudo, and function events, respectively.

The information provided by the code in `ie_code` is interpreted according to event flags in `ie_flags`. (See *Event Flags* below.)

The remaining elements of the struct provide general status information which may be useful on any event:

`ie_shiftmask`
is used to report the state of certain shift-keys that is, to modify the meaning of other events.

`ie_locx` and

`ie_locy` provide the position of the locator in the window's coordinate system at the time the event occurred.

`ie_time` provides a timestamp for the event, in the format of a system `timeval`, as defined in `<sys/time.h>`.

5.1.2. Event Codes

Event codes can take on any value in the range from 0 to 65535 inclusive. Of the codes defined in the header file, 256 are assigned to the ASCII event class and the other 128 are partitioned between the pseudo and function event classes. The following constants define the number of codes and the first and last code in the latter two classes:

```
#define VKEY_CODES        128
#define VKEY_FIRST        32512
#define VKEY_LAST         VKEY_FIRST+VKEY_CODES-1
```

5.1.2.1. ASCII Events

The event codes in the range 0 to 255 inclusive are assigned to the ASCII event class. This class is further sub-divided:

```
#define ASCII_FIRST 0
#define ASCII_LAST 127
```

In particular, striking a key which has an obvious ASCII meaning causes the Virtual Input Device to enqueue for the client an event whose code is the 7-bit ASCII character corresponding to that key. Such a key with an obvious ASCII meaning is one in the main typing array labelled with a single letter of the alphabet. This is independent of the physical keyboard actually used. A slight complication occurs because of the presence of both upper- and lower-case characters in ASCII: if the user "shifts" the physical keyboard by depressing the CAPS-LOCK, SHIFT-LOCK, or SHIFT key the `ie_code` contains the shifted ASCII character corresponding to the struck key.

For physical keystations that are mapped to cursor control keys, the current implementation transmits a series of events with codes that correspond to the ANSI X3.64 7-bit ASCII encoding for the cursor control function. For physical keystations that are mapped to function keys, the current implementation transmits a series of events with codes that correspond to an ANSI X3.64 user-definable escape sequence. For further details, see `kbd`(5).

```
#define META_FIRST 128
#define META_LAST 255
```

Event codes from 128 to 255 inclusive are generated when the client has META translation enabled and the user strikes a key that would generate a 7-bit ASCII code while the META key is also depressed. In this case, the event code is the 7-bit ASCII code added to META_FIRST.

5.1.2.2. Function Events

Event codes in the function class correspond to button strikes that do not result in generation of an event code in the ASCII class.

In the function class are the event codes associated with locator buttons:

```
#define BUT(i)
```

A physical locator often has up to 10 buttons connected to it. Alternatively, even though the physical locator does not have any buttons physically available on it, it may have buttons on another device assigned to it. A light pen is an example of such a locator. In either case, each of the *n* buttons (where $0 < n <= 10$) associated with the Virtual Input Device's locator are assigned an event code; the *i-th* button is assigned the code `BUT(i)`. Thus a 3-button mouse reports x and y and buttons $1 - 3$.

In the function class are the event codes associated with keyboard function keys that don't generate single ASCII charaters:

```
#define KEY_LEFT(i)
#define KEY_RIGHT(i)
#define KEY_TOP(i)
#define KEY_BOTTOMLEFT
#define KEY_BOTTOMRIGHT
```

The function keys in the Virtual Input Device define an idealized standard layout that groups keys by location: 16 left, 16 right, 16 top and 2 bottom. While the actual position on the keyboard may be different, it is convenient to provide some grouping for the large number of function keys. The mapping to physical keys on various keyboards is defined in **`<sundev/kbd.h>`** and discussed in `kbd`(5).

5.1.2.3. Pseudo Events

```
#define VKEY_FIRSTPSEUDO
#define VKEY_LASTPSEUDO
```

Event codes in the pseudo class are events that involve locator movement instead of physical button striking. The physical locator constantly provides an (x, y) coordinate position in pixels; this position is transformed by the Virtual Input Device to the coordinate system of the window receiving an event. In order to watch actual locator movement (or lack thereof), the client must be enabled for the events with codes.

```
#define LOC_MOVE
#define LOC_MOVEWHILEBUTDOWN
#define LOC_STILL
```

A LOC_MOVE is reported only when the locator actually moves. Since fast motions may yield non-adjacent locations in consecutive events, the locator tracking mechanism reports the current position at a set sampling rate, currently 40 times per second.

LOC_MOVEWHILEBUTDOWN is like LOC_MOVE but happens only when a button on the locator is down.

A single LOC_STILL event is reported when the locator has been still for a specified period, currently 1/5 of a second.

Clients can be notified when the locator has entered or exited a window via the event codes:

```
#define LOC_WINENTER
#define LOC_WINEXIT
```

5.1.3. Event Flags

Only one event flag is currently defined:

```
#define IE_NEGEVENT
```

indicates the event was "negative." Positive events include depression of any button or key, including buttons on the locator, motion of the locator device while it is available to this client, and entry of the cursor into a window. The only currently defined negative event is the release of a depressed button. Stopping of the locator and locator exit from the window are positive events, distinct from locator motion and window entry. This asymmetry allows a client to be informed of these events without the performance penalty associated with receiving all negative events and then discarding all but these two.

Two macros are defined to inquire about the state of this flag:

```
#define win_inputnegevent(ie)
#define win_inputposevent(ie)
    struct      inputevent *ie;
```

These are TRUE or FALSE if the IE_NEGEVENT bit is 1 or 0 respectively in the input event pointed to by `ie`.

5.1.4. Shift Codes

`ie_shiftmask` contains a set of bit flags which indicate an interesting state when an input event occurs. The most obvious example is the state of the Shift or Control keys when some other key is pressed. Eventually, clients will be able to declare any Virtual Input switch as an "interesting" shift switch. For now, only the following bits are reported:

```
#define  CAPSMASK       0x0001
#define  SHIFTMASK      0x000E
#define  CTRLMASK       0x0030
#define  UPMASK         0x0080
```

These are defined in `<sundev/kbd.h>`, and described in `kbd`(5).

5.2. Reading Input Events

A library routine exists for reading the next input event for a window:

```
int  input_readevent(fd, ie)
    int         fd;
    struct      inputevent *ie;
```

This fills in the indicated 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.

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

A window process can ask to be sent a SIGIO if any input is pending in a window. Enabling this option is also via a standard `fcntl` system call, as described in `fcntl(2)` (using F_SETFL) and `fcntl(5)` (using FASYNC). The programmer can set up a signal catcher for SIGIO by using the `signal(3)` call.

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 `inputevents` then you need to divide by `sizeof(struct inputevent)`.

The recommended normal style for handling input uses blocking I/O and the `select(2)` system call to await both input events and signals such as SIGWINCH. This allows a signal handler to merely set a flag, and leave substantial processing to be performed synchronously when the `select` returns. The `tool_select` mechanism described in chapter 6 illustrates this approach. Using blocking I/O and `read(2)` without a prior `select` forces the client to process SIGWINCH signals entirely in the asynchronous interrupt handler. This necessitates extra care to avoid race conditions and other asynchronous errors.

Non-blocking I/O may be useful in a few circumstances. For example, when tracking the mouse with an image which requires significant computation, it may be desirable to ignore all but the last in a queued sequence of motion events. This is done by reading the events, but not processing them until a non-motion event is found, or until all events are read. Then the most recent mouse location is displayed, but not all the points covered since the last display. 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.

5.3. Input Serialization and Distribution

With the exception of some of the pseudo event codes, the Virtual Input Device described in preceding sections is not logically tied to the Sun window system; the scheme could be used by any system desiring that form of unification. This section is more specific to the window system, since it discusses how events are selected and distributed among the various windows which might use them.

Each user input event is formatted into an `inputevent`, which is then assigned to some recipient. There are three ways a process gets to receive an input event:

- Most commonly, it reads the window which lies under the cursor, and that window has an *input mask* which matches the event. Input masks are described in ***Input Masks.*** If several windows are layered under the cursor, the event is tested first against the input mask of the topmost window.
- If the event does not match the input mask of one window, other windows will be given a chance at it, as described below.
- Much less frequently, a window will be made the recipient of ***all*** input events; this is discussed under `win_grabio` below.

Each window designates another window to be offered events which the first will not accept. By default this is the window's parent; another backstop may be designated in a call to `win_setinputmask`, described in the next section. If an event is offered unsuccessfully to the root window, it is discarded. Windows which are not in the chain of designated recipients never have a chance to accept the event.

If a recipient is found, the locator coordinates are adjusted to the coordinate system of the recipient, and the event is appended to the recipient's input stream. Thus, every window sees a single stream of input events, in the order in which the events happened (and time-stamped, so that the intervals between events can also be computed), and including only the events that window has declared to be of interest.

5.3.1. Input Masks

The input masks facilitate two things:

- Events can be accepted or rejected by classes; for instance, a process may want only ASCII characters.
- The times when events are accepted can be controlled, minimizing the processing required to accept and ignore uninteresting events. For instance, a process may track the mouse only when it is inside one of its windows, or when one of the mouse buttons is down.

Clients specify which input events they are prepared to process by setting the input mask for each window being read.

```
struct  inputmask  {
    short       im_flags;
    char        im_inputcode[IM_CODEARRAYSIZE];
    short       im_shifts;
    short       im_shiftcodes[IM_SHIFTARRAYSIZE];
};
#define IM_CODEARRAYSIZE   (VKEY_CODE/((sizeof char)*BITSPERBYTE))
#define IM_SHIFTARRAYSIZE  ((sizeof short)*BITSPERBYTE)
```

`im_flags` specifies the handling of related groups of input events:

```
#define IM_ASCII
```

indicates that the Virtual Input Device translation should occur.

```
#define IM_ANSI
```

indicates that the process wants keystrokes to be interpreted as ANSI characters and escape sequences: normal ASCII characters are represented by their ASCII code in `ie_code`, described in *Uniform Input Events*. Function keys with a standard interpretation, such as the cursor control keys, are represented by a sequence of input events, whose `ie_codes` are ASCII characters starting with <ESC>. See `kbd`(5) for further details.

```
#define IM_POSASCII
```

indicates that the client only wants to be notified of positive events for ASCII class events, even though IM_NEGEVENT is enabled.

Note: The current implementation automatically enables both IM_ANSI and IM_POSASCII when IM_ASCII is specified.

Requesting a particular function event in addition turns off any ANSI escape-coding for that function event.

```
#define IM_META
```

indicates that META-translation should occur. This means ASCII events that occur while the

META key is depressed are reported with codes in the META range. Note that IM_META does not make sense unless IM_ASCII is enabled.

```
#define IM_NEGEVENT
```

indicates that the client wants to be notified of negative events as well as positive ones. See *Event Flags* for a discussion of positive and negative events.

```
#define IM_UNENCODED
```

indicates that no translation of physical device events should be performed. The Virtual Input Device should not intervene between the window and the user input. In this case, the most significant byte of `ie_code` in an input event is the id number of the device that generated the event, and the least significant byte contains the physical keystation number of the keystation that the user struck. The current device ids are those assigned to the supported keyboards and the id assigned to the mouse

```
#define MOUSE_DEVID 127
```

For unencoded mouse input, the least significant byte of the event code is identical to the least significant byte of the corresponding encoded input event. Note that unencoded pseudo events are associated with the physical locator; that is, a button-push on a tablet puck will generate a different code from a corresponding button-push on a mouse.

`im_inputcode` is an array of bit flags indexed by biased event codes. A 1 in the ***ith*** position of the bit array indicates that the event with code VKEY_FIRST+*i* should be reported. This filter applies in both IM_UNENCODED and IM_ASCII modes.

There are two routines which are of interest here.

```
win_setinputmask(windowfd, acceptmask, flushmask, designee)
    int         windowfd;
    struct      inputmask *acceptmask, *flushmask;
    int         designee;
```

sets the input mask for the window identified by `windowfd`. `acceptmask` addresses the new mask — events it passes will be reported to this window after the call to `win_setinputmask`.

`flushmask` specifies a set of events which should be flushed from this window's input queue. These are events which were accepted by the previous mask, and have already been generated, but not read, by this window. This is a dangerous facility; type-ahead and mouse-ahead will often be lost if it is used. The most obvious application is for confirmations, but these can be better implemented by requiring the confirmation within a short time-out.

Note: If `flushmask` is non-NULL, the current implementation flushes all events from the queue, not just those specified in `flushmask`.

`designee` is the window number, which specifies the next potential recipient for events rejected by this window. If it is set to WIN_NULLLINK (defined in `<sunwindow/win_struct.h>`), it is interpreted as designating the window's parent.

Note: Changing masks in response to some input should be done with caution. There will be a lapse of time between the event which persuades the client it wants a new mask and the time the system interprets the resulting call to `win_setinputmask`. Events which occur in this interval will be passed or discarded according to the old input mask. Thus, it is probably not appropriate to wait for a button down before requesting the corresponding button-up; the button-up may arrive and be discarded before the mask is changed. It's less dangerous to wait until a button goes down to start tracking the mouse, since the client will be caught up as soon as the first motion event arrives. But even here, it's better to ask for the LOC_MOVEWHILEBUTDOWN event, and never change the mask.

The input mask for a window is read with

```
win_getinputmask(windowfd, im, designee)
    int         windowfd;
    struct      inputmask *im;
    int         *designee;
```

The input mask for the window identified by `windowfd` is copied into the buffer addressed by im. The number of the window that is the next possible recipient of input is copied into the integer addressed by `designee`.

We return to `win_input.h` for these routines useful for manipulating input masks. The first three are macros:

```
#define win_setinputcodebit(im, code)
    struct      inputmask *im;
    char        code;
```

sets the bit indexed by `code` in the input mask addressed by `im` to 1.

```
#define win_unsetinputcodebit(im,code)
    struct      inputmask *im;
    char        code;
```

resets the bit to zero. The routine:

```
#define win_getinputcodebit(im, code)
    struct      inputmask *im;
    char        code;
```

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

```
input_imnull(mask)
    struct      inputmask *mask;
```

is a procedure which initializes an input mask to all zeros. It is critical to initialize the input mask explicitly when the mask is defined as a local procedure variable.

5.3.2. Seizing All Inputs

Normally, input events are directed to the window which underlies the cursor at the time the event occurs. Two procedures modify that behavior. 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.

```
win_releaseio(windowfd)
    int         windowfd;
```

undoes the effect of a `win_grabio`, restoring the previous state.

5.4. Event Codes Defined

In the following table are collected together all of the special event code names discussed above. These names define values which appear in the `ie_code` field of an `inputevent`. As the system evolves, the particular value bound to a name is likely to change, thus event codes should be compared to the symbolic names below, not to the current values of those names.

```
#define ASCII_FIRST              (0)
#define ASCII_LAST               (127)
#define META_FIRST               (128)
#define META_LAST                (255)

#define VKEY_CODES               (128)
#define VKEY_FIRST               (32512)

#define VKEY_FIRSTPSEUDO         (VKEY_FIRST)
#define LOC_MOVE                 (VKEY_FIRSTPSEUDO+0)
#define LOC_STILL                (VKEY_FIRSTPSEUDO+1)
#define LOC_WINENTER             (VKEY_FIRSTPSEUDO+2)
#define LOC_WINEXIT              (VKEY_FIRSTPSEUDO+3)
#define LOC_MOVEWHILEBUTDOWN     (VKEY_FIRSTPSEUDO+4)
#define VKEY_LASTPSEUDO          (VKEY_FIRSTPSEUDO+15)

#define VKEY_FIRSTFUNC           (VKEY_LASTSHIFT+1)

#define BUT_FIRST                (VKEY_FIRSTFUNC)
#define BUT(i)                   ((BUT_FIRST)+(i)-1)
#define BUT_LAST                 (BUT_FIRST+9)

#define KEY_LEFTFIRST            ((BUT_LAST)+1)
#define KEY_LEFT(i)              ((KEY_LEFTFIRST)+(i)-1)
#define KEY_LEFTLAST             ((KEY_LEFTFIRST)+15)

#define KEY_RIGHTFIRST           ((KEY_LEFTLAST)+1)
#define KEY_RIGHT(i)             ((KEY_RIGHTFIRST)+(i)-1)
#define KEY_RIGHTLAST            ((KEY_RIGHTFIRST)+15)

#define KEY_TOPFIRST             ((KEY_RIGHTLAST)+1)
#define KEY_TOP(i)               ((KEY_TOPFIRST)+(i)-1)
#define KEY_TOPLAST              ((KEY_TOPFIRST)+15)

#define KEY_BOTTOMLEFT           ((KEY_TOPLAST)+1)
#define KEY_BOTTOMRIGHT          ((KEY_BOTTOMLEFT)+1)

#define VKEY_LASTFUNC            (VKEY_FIRSTFUNC+101)

#define VKEY_LAST               VKEY_FIRST+VKEY_CODES-1
```

There are 3 synonyms for the common case of a 3-button mouse:

```
#define MS_LEFT          BUT(1)
#define MS_MIDDLE        BUT(2)
#define MS_RIGHT         BUT(3)
```

Chapter 6

Suntool: Tools and Subwindows

This chapter introduces the third and highest level of SunWindows, ***suntools***. It discusses how to write a tool: it covers creation and destruction of a tool and its subwindows, the strategy for dividing work among them, and the use of routines provided to accomplish that work.

At the ***suntools*** level, the lower-level facilities are actually used to build user interfaces. This chapter also describes a model for building applications, a number of components that implement commonly-needed portions of such applications, and an executive and operating environment that supports that model.

We refer to an application program that is a client of this SunWindows level as a ***tool***. ***Tool*** covers the one or more processes that do the actual application work. This term also refers to the collection of typically several windows through which the tool interacts with the user. Simple tools might include a calculator, a bitmap editor, and a terminal emulator. Sun Microsystems provides a few ready-built tools, several of which are illustrated in Appendix B. Others may be developed to suit particular needs.

Common SunWindows tool components and their functions include:

- A standard ***tool window*** that frames the ***subwindows*** of the tool, identifying it with a name stripe at the top and borders around the subwindows. Each tool window can adjust its size and position, including layering, and subwindow boundary movement.
- An executive framework that supplies the usual "main loop" of a program, and which coordinates the activities of the various subwindows.
- Several standard subwindows that can be instantiated in the tool.
- A standard scheme for laying out those subwindows.
- A facility that provides a default ***icon***, which is a small form the tool takes to be unobtrusive but still identifiable.

The `suntools` program initializes and oversees the window environment. It provides:

- Automatic startup of a specified collection of tools.
- Dynamic invocation of standard tools through a menu interface.
- Management of the window, called the ***root*** window, which underlies all tools and paints a simple pattern.
- The user interface for leaving the window system.

Users desiring another interface to these functions can replace the `suntools` program, while retaining specific tools.

The procedures that support the facilities described in this chapter and the following two are in the ***suntool*** library, `/usr/lib/libsuntool.a`. These procedures and their data structures are declared in a number of distinct header files, which are included in

`<suntool/tool_hs.h>`.

6.1. Tools Design

A typical tool is built as a ***tool window***, and contained within that, a set of ***subwindows***, which incorporate most of the user interface to the tool's facilities. Each subwindow is a "window" in the sense described in *Window Manipulation*; the subwindows form a subtree rooted at the tool window, and the various tool windows are all children of the ***root*** window associated with the screen.

6.1.1. Non-Pre-emptive Operation

In general, tools should be designed to function in a ***non-pre-emptive*** style: they should wait without consuming resources until given something to do, perform the task expeditiously, and promptly return control to the user. If some task requires extensive processing, a separate process should be forked to run it without blocking the user interface.

This non-pre-emptive style implies that the tool is built as a set of independent procedures, which are invoked as appropriate by a standardized control structure. The basic advice to client programs is, "Wait right there; we'll let you know as soon as we have something for you to do." From a programming point of view, the main function that the tool mechanism provides is the provision of the control structure to implement this non-pre-emptive programming style. The tool window and its subwindows all have the same interface to this control mechanism.

6.1.2. Division of Labor

The tool window performs a few functions directly. These are the user interface functions, which are common to all tools.

Subwindows are the workhorses of the ***suntool*** environment, but most of the work they do is specific to their own tasks, and of little interest here. It is important to understand that a subwindow corresponds to a data type: there will be many instantiations of particular subwindows, quite possibly several in a single tool.

Various types of subwindows are developed as separate packages that can be assembled at a high level. In addition to programmer convenience, this approach promotes a consistent user interface across applications.

The remainder of this chapter divides a tool's existence into two large areas: creation and destruction, and tool-specific aspects of processing.

6.2. Tool Creation

All of the following processing must be performed as a tool is started:

- Parameters for this invocation of the tool are passed to it. Some of the parameters are application specific and some parameters are generic to all tools.
- The tool window is created with space allocated for it and its various options defined; similarly, its subwindows are created and positioned in the tool window.

- The UNIX signal system is initialized to catch appropriate signals, e.g., SIGWINCH, that will be sent to the tool's process.
- The tool's window is installed into the display structure.
- Finally, the tool starts its normal processing.

6.2.1. Tool Attributes

The programming interface to the tool window is based on ***attribute*** manipulation. An attribute is an ***identifier/value*** pair. The identifier is an integer constant. The value is a long word (32 bit) quantity that may be a single numeric quantity or a pointer to other data. Attributes can be gathered together into an ***attribute list***. An attribute list can contain other attribute lists as well.

The tool window has a collection of attributes that can be set to affect the behavior of the tool window. The following table lists each tool attribute, followed by the type of value that may be assigned to it, and a short description of the attribute's meaning. The procedures used to manipulate these attributes are discussed throughout this chapter. Tool attribute identifiers are defined in `<suntool/tool.h>`.

Table 6-1: Summary of Tool Attributes

Summary of Tool Attributes

Name	Value type	Description
`WIN_COLUMNS`	[unsigned int]	This attribute is the width, in columns of characters, of the internal area of a tool that is available to subwindows. A tool is 80 columns by default.
`WIN_LINES`	[unsigned int]	This attribute is the height, in rows of characters, of the internal area of a tool that is available to subwindows. A tool is 34 rows by default.
`WIN_WIDTH`	[unsigned int]	This attribute is the width, in pixels, of a normal sized tool.
`WIN_HEIGHT`	[unsigned int]	This attribute is the height, in pixels, of a normal sized tool.
`WIN_LEFT`	[int]	This attribute is the *x* position of the upper left hand corner, in pixels, of the tool.
`WIN_TOP`	[int]	This attribute is the *y* position of the upper left hand corner, in pixels, of the tool.
`WIN_ICONIC`	[0 or 1]	This attribute is the state of the tool: 0 means normal state (opened) and 1 means iconic state (closed). A tool is open by default.
`WIN_REPAINT_LOCK`	[0 or 1]	This attribute indicates the state of a tool's repaint lock: 0 means repaint as usual and 1 means don't repaint as usual. Turning off the repaint lock or explicitly calling `tool_display` forces a repaint. One might turn on the repaint lock if one was doing a batch of things to the tool and only wanted the tool's image to repaint once at the end of the changes. This attribute is 0 by default.

Summary of Tool Attributes

Name	Value type	Description
`WIN_LAYOUT_LOCK`	[0 or 1]	This attribute indicates the state of a tool's subwindow layout lock, 0 means use the tool's tiling algorithm to lay out the position and size of subwindows, and 1 means don't do any layout. Turning on the layout lock makes subwindow layout the programmer's complete responsibility (see `tool_layoutsubwindows`). This attribute is 0 by default.
`WIN_NAME_STRIPE`	[0 or 1]	This attribute indicates whether the tool has a name stripe at the top of the tool: 1 means yes and 0 means no. This attribute is 1 by default.
`WIN_BOUNDARY_MGR`	[0 or 1]	This attribute indicates whether the user is allowed to try to interactively move the boundary between subwindows of the tool with the mouse. A 1 value means yes and a 0 value means no. This attribute is 0 by default.
`WIN_LABEL`	[char *]	This attribute indicates the string used in the name stripe of the tool. This attribute is NULL by default.
`WIN_ICON`	[struct icon *]	This attribute is the icon used by the tool. Its default value is NULL which means that a default iconic image is displayed.
`WIN_ICON_LEFT`	[int]	This attribute is the x position, in pixels, of the upper left hand corner of an iconic tool.
`WIN_ICON_TOP`	[int]	This attribute is the y position of the upper left hand corner, in pixels, of an iconic tool.
`WIN_ICON_LABEL`	[char *]	This attribute indicates the string used as the text in the icon. This attribute is NULL by default. ***Note: The current implementation of this attribute does not support setting it unless a WIN_ICON has been done already.***

Summary of Tool Attributes

Name	Value type	Description
`WIN_ICON_IMAGE`	[struct pixrect *]	This attribute is the memory pixrect used for the graphic portion of the icon. It's default value is NULL which means that a default iconic image is displayed. ***Note: The current implementation of this attribute does not support setting it unless a** WIN_ICON **has been done already.***
`WIN_ICON_FONT`	[struct pixfont *]	This attribute is the font handle used to display text in the icon. Its default value is NULL, which means that the system default font is displayed. ***Note: The current implementation of this attribute does not support setting it unless a** WIN_ICON **has been done already.***
`WIN_ATTR_LIST`	[char **]	This is a pseudo-attribute that is a list of other attributes. A 0 attribute identifier terminates the list. Querying for this attribute is an error.
`WIN_DEFAULT_CMS`	[0 or 1]	This attribute indicates the state of the ***default colormap segment.*** The default colormap segment is that to which newly created pixwins are initialized. Normally, the default colormap segment is named "monochrome" with its two colors defined by the values set during screen creation. If the value of WIN_DEFAULT_CMS is 1 then the colormap segment currently being used by the tool window is set to be the default colormap segment for the tool's process. This attribute is 0 by default. WIN_DEFAULT_CMS is usually set to 1 from the command line so that the tool window colors, set with WIN_FOREGROUND and WIN_BACKGROUND, are used for all the subwindows as well. ***Note: The current implementation of this attribute does not support resetting it back to 0 once set to 1.***

Summary of Tool Attributes

Name	Value type	Description
`WIN_FOREGROUND`	[struct singlecolor *]	This attribute indicates the foreground color of the tool window. This attribute's default value is the foreground color set during screen creation.
`WIN_BACKGROUND`	[struct singlecolor *]	This attribute indicates the background color of the tool window. This attribute's default value is the background color set during screen creation.

6.2.1.1. The Tool Struct

The tool structure is considered private to the implementation of the tool. Its data should be accessed indirectly via attribute calls. However, in previous versions of the system, programmers were instructed to write code that directly accesses this structure, and not all tool data is directly accessible via the attributes mechanism. Therefore, this section describes the fields of the structure.

Note: Mixing access of the tool structure by direct access (via tool structure field reference) and indirect access (via attributes) will often yield incorrect results. The attribute interface dynamically allocates storage for the fields of the tool struct while the old interface saved whatever the programmer handed it.

The tool struct is defined in `<suntool/tool.h>`. It is:

```
struct tool  {
    short       tl_flags;
    int         tl_windowfd;
    char        *tl_name;
    struct      icon *tl_icon;
    struct      toolio tl_io;
    struct      toolsw *tl_sw;
    struct      pixwin *tl_pixwin;
    struct      rect tl_rectcache;
};
```

`tl_flags` holds state information. Currently, there are 6 defined flags:

```
#define  TOOL_NAMESTRIPE          0x01
#define  TOOL_BOUNDARYMGR         0x02
#define  TOOL_ICONIC              0x04
#define  TOOL_SIGCHLD             0x08
#define  TOOL_SIGWINCHPENDING     0x10
#define  TOOL_DONE                0x20
```

Their meanings are as follows:

TOOL_NAMESTRIPE
: indicates that the tool is to be displayed with a black stripe holding its name at the top of its window.

TOOL_BOUNDARYMGR
: enables the option that allows the user to move inter-subwindow boundaries.

TOOL_ICONIC
: indicates the current state of the tool: 1 = small (*iconic*); 0 = normal (*open*). Note that client programs should *never* set or clear the TOOL_ICONIC flag.

TOOL_SIGCHLD and TOOL_SIGWINCHPENDING
: mean that the tool has received the indicated signal and has not yet performed the processing to deal with it.

TOOL_DONE
: indicates the tool should exit the `tool_select` notification loop.

The last three flags are used during `tool_select` processing described below and should be considered private to the tool implementation.

`tl_windowfd`	holds the file descriptor for a tool's window. This is used for both input and output. It also identifies the window for manipulations on the window database, such as modifying its position or shape. Possible uses of `windowfd` are discussed in chapters 3 through 5.
`tl_name`	addresses the string that can be displayed in the tool's namestripe and default icon.
`tl_rectcache`	holds a rectangle that indicates the size of the tool's window. Because the rectangle is in the tool's coordinate system, the origin will always be (0, 0). This size information is cached so that the tool can tell when its size has changed by comparing the cached rect with the current rect.
`tl_icon`	holds a pointer to the icon struct for this tool.
`tl_pixwin`	addresses the window's pixwin, which is the structure through which the tool accesses the display.
`tl_sw`	points to the first and oldest of the tool's subwindows. The following section discusses these structs.

The tool uses `tl_io` to control notification of input and window change events to itself. ***Toolio Structure*** details this structure type. During tool creation, the fields of this structure are set up with values to do default tool processing.

6.2.2. Tool Initialization Parameters

Tool manager specific parameters are passed through the environment and via the command line. Most programmers can ignore the environment parameters, which are described below in ***Environment Parameters***. However, most programmers do need to deal with command line arguments.

6.2.2.1. Command Line Parsing

The following table lists the command line arguments that the user should be able to pass to a tool on the command line. All tools should be able to accept these arguments and thus they are called ***generic*** tool arguments.

Table 6-2: Generic tool arguments

FLAG	(LONG FLAG)	ARGS	ATTRIBUTE
-Ww	(-width)	column	WIN_COLUMNS
-Wh	(-height)	line	WIN_LINES
-Ws	(-size)	x y	WIN_WIDTH WIN_HEIGHT
-Wp	(-position)	x y	WIN_LEFT WIN_TOP
-WP	(-icon_position)	x y	WIN_ICON_LEFT WIN_ICON_TOP
-Wl	(-label)	"string"	WIN_LABEL
-Wi	(-iconic)		WIN_ICONIC
-Wn	(-no_name_stripe)		WIN_NAME_STRIPE
-Wt	(-font)	filename	
-Wf	(-foreground_color)	red green blue	WIN_FOREGROUND
-Wb	(-background_color)	red green blue	WIN_BACKGROUND
-Wg	(-set_default_color)		WIN_DEFAULT_CMS
-WI	(-icon_image)	filename	WIN_ICON_IMAGE
-WL	(-icon_label)	"string"	WIN_ICON_LABLE
-WT	(-icon_font)	filename	WIN_ICON_FONT
-WH	(-help)		

So that tool builders can parse the command line for generic tool arguments in a uniform way, some utilities are provided.

```
int tool_parse_all(argc_ptr, argv, tool_args_ptr, tool_name)
    int         *argc_ptr;
    char        **argv;
    char        ***tool_args_ptr;
    char        *tool_name;
```

`tool_parse_all` scans the entire length of `argv` for generic tool arguments and builds up an attributes list in `*tool_args_ptr`. It is important to initialize `*tool_args_ptr` to NULL before making this call. As flags and their arguments are successfully parsed `argv` is modified to no longer contain the matched arguments and `*argc_ptr` is decremented. `*argc_ptr` is the count of elements in `argv`. `tool_name` is passed in so that meaningful error messages can be sent to `stderr` if an error is detected in the command line. `tool_parse_all` returns −1 to indicate such an error and 0 to signify success. When an error is detected, it is a good idea to call `tool_usage`.

```
tool_usage(tool_name)
    char        *tool_name;
```

`tool_usage` sends an message to `stderr` listing the command line format of generic tool arguments. `tool_name` is used in formatting the message.

Some programs have reason to not give over control of their command lines to `tool_parse_all`. For these programs, `tool_parse_one` is provided.

```
int tool_parse_one(argc, argv, tool_args_ptr, tool_name)
    int         argc;
    char        **argv;
    char        ***tool_args_ptr;
    char        *tool_name;
```

`tool_parse_one` scans the first string in `argv` for a generic tool argument flag. If it finds one, the attributes list in `*tool_args_ptr` has another attribute added to it. It is important to initialize `*tool_args_ptr` to NULL before calling this routine for the first time. Unlike `tool_parse_all`, `tool_parse_one` doesn't modify `argv` or `argc`. A positive number return value indicates how many arguments from the front of `argv` were used. It is then the callers responsibility to modify `argv` and `argc`. Error reporting is as with `tool_parse_all`.

Some programs want the convenience of `tool_parse_all` but would like to explicitly determine if a particular attribute has been specified by the user from the command line. `Tool_find_attribute` is a utility to help do this.

```
int  tool_find_attribute(tool_args, id, value_ptr)
    char        **tool_args;
    int         id;
    char        **value_ptr;
```

`tool_find_attribute` looks for the attribute identifier `id` in `tool_args`. If the attribute is not found then the return value is 0. If the attribute is found then the return value is 1 and `*value_ptr` is set to the value of the attribute. The storage for `*value_ptr` must later be released via a call to `tool_free_attribute` (described below).

The storage used for the attribute list built up by the calls to `tool_parse_all` and `tool_parse_one` should eventually be freed via a call to `tool_free_attribute_list`.

```
int tool_free_attribute_list(tool_args)
    char        **tool_args;
```

`tool_free_attribute_list` releases the storage used by `tool_args` after releasing all the storage for its component attributes. This call is most often made just after calling `tool_make`.

6.2.3. Creating the Tool Window

The pair of procedures `tool_make` and `tool_createsubwindow` perform the main work of creating a tool with its subwindows. These take a series of parameters that define the object to be created, and return a pointer to an object that encapsulates the information about the tool or a subwindow. That pointer is then passed to a number of other routines that manipulate the object; the client is usually not concerned with the exact definition of the structure.

`tool_make` and `tool_createsubwindow` include a large part of the processing described in the earlier parts of this manual. Thus, client programmers need not necessarily concern themselves much with the details of ***pixwins*** and window devices.

A tool is created by a call to:

```
/* VARARGS */
struct  tool *tool_make(id, value, id, value, ... 0)
    int         id;
    caddr_t     value;
```

`tool_make` takes a variable number of attribute identifier/value pairs, terminated by the special attribute identifier 0. These attributes control the behavior of the tool. A list of valid attributes is available in the section ***Tool Attributes***. `id`'s are the attribute identifiers. `value`'s are the attribute values of the preceding `id`. A tool handle is returned. If the tool handle is

NULL then the call failed. `tool_make` changes the process group of the current process to the current process id.

All `value` arguments passed into `tool_make` are copied. Thus, all subsequent accesses of tool attribute values must use `tool_get_attribute` (see ***Changing the Tools' Attributes***). For example, if you use WIN_ICON to set the tool's icon, changing the icon structure after you passed it into `tool_make` will not change the tool's icon.

There are parameters passed in the environment (see ***Environment Parameters***) that `tool_make` examines during its execution. Attribute arguments to `tool_make` that duplicate environment parameters override the environment parameters. In addition, an attribute specified early in the calling sequence is overridden by a later instance of the same attribute. Thus, the order of attributes in the call to `tool_make` is significant. Here is how attributes should be ordered in the call to `tool_make`:

- Attributes that set the default setting for the tool should come first, e.g., WIN_LABEL and WIN_ICON.
- Attributes that the user has specified from the command line should come next, i.e., specify WIN_ATTR_LIST and its value.
- Attributes that you, as the programmer, are absolutely not going to allow the user to override should come last, e.g., WIN_WIDTH and WIN_HEIGHT if you insist that the tool be started a fixed size.

Here is a sample call to `tool_make` that illustrates the ordering of attributes as described above:

```
tool = tool_make(
        WIN_LABEL,        "Tool 2.0",
        WIN_ICON,         &icon,
        WIN_ATTR_LIST,    tool_args,
        WIN_WIDTH,        200,
        WIN_HEIGHT,       100,
        0);
```

Remember to call `tool_free_attribute_list` after calling `tool_make`.

Creating the tool does not cause it to appear on the screen; a separate step is used for that purpose as described in ***Tool Installation***.

6.2.4. Subwindow Creation

After the tool is created, its subwindows are added to it. This section describes the basic tool subwindow creation procedure. Often, however, you are not providing your own subwindow implementation. Instead, an existing subwindow package is providing the implementation, e.g., a message subwindow or a panel subwindow. Their create procedures, e.g., `msgsw_createtoolsubwindow` or `panel_create`, handle tool subwindow creation for you. If you are not providing your own subwindow implementation then you can skip down to ***Tool Installation***.

```
struct  toolsw *tool_createsubwindow(tool, name, width, height)
    struct      tool *tool;
    char        *name;
    short       width, height;

#define TOOL_SWEXTENDTOEDGE  -1
```

makes a new subwindow, adds it to the list of subwindows for the indicated `tool`, and returns a pointer to the new `toolsw` struct. The `width` and `height` parameters are hints to the layout mechanism indicating what size the windows should be if there is enough room to accommodate them. There are no guarantees about maintaining subwindow size because changing window sizes can ruin any scheme. TOOL_SWEXTENDTOEDGE may be passed for `width` and/or `height`; it allows the subwindow to stretch with its parent in either or both directions. ***Subwindow Layout*** details the subwindow layout algorithm. The `name` is currently unused; it may eventually support the capability to refer to subwindows by name.

The remaining subwindow initialization requires reference to the data structure:

```
struct  toolsw {
    struct      toolsw *ts_next;
    int         ts_windowfd;
    char        *ts_name;
    short       ts_width;
    short       ts_height;
    struct      toolio ts_io;
    int         (*ts_destroy)();
    caddr_t     ts_data;
};
```

The subwindows of a tool are chained on a list with `ts_next` in one subwindow pointing to the next in line, until the list is terminated with a null pointer.

Like the tool window, each subwindow must have an associated open window device; `tool_createsubwindow` stores the file descriptor in `ts_windowfd`.

`ts_name`, `ts_width` and `ts_height` are exactly as in the call to `tool_createsubwindow`.

The tool uses `ts_io` to control notification of input and window change events to the subwindow. Upon subwindow creation, the `ts_io` structure has null values in it that need to be set. This is normally done by the ***create*** routine for a standard subwindow type. ***Toolio Structure*** details this structure.

`ts_destroy` gets called when the tool is being destroyed by `tool_destroy` so that the subwindow may terminate cleanly.

`ts_data` provides 32 bits of uninterpreted data private to the subwindow implementation. Typically, it will be a pointer to information for this instance of the subwindow. That is, all subwindows of the same type will share common interrupt handlers and layout characteristics. Window contents and other information specific to one particular window will all be accessed through this pointer. This is discussed at more length in ***Minimum Standard Subwindow Interface*** in Chapter 7.

6.2.5. Subwindow Layout

By default, subwindows are laid out in their tool's area in a simple left-to-right, top-to-bottom fashion, in the order they are created. A subwindow is placed as high as it can be, and in that space, as far to the left as it can be. The `ts_width` and `ts_height` fields in the `toolsw` structure control the width and height of the subwindow.

The default subwindow layout mechanism breaks down for complicated subwindow layouts. This is how you replace the default subwindow layout mechanism with your own. Include a function named `tool_layoutsubwindows` in your program. Your version of this function will be loaded instead of the function of the same name that the ***suntool*** library contains. `tool_layoutsubwindows` just takes a tool handle and has no return value. It will be called by the tool manager whenever the following occurs:

- The tool's size has changed. This includes the first time that the tool goes to display itself.
- The subwindow boundary manager has changed one of the values of `ts_width` or `ts_height` in a `toolsw` structure.
- The WIN_LAYOUT_LOCK attribute has been set to 0.

You can then use `win_setrect` in your implementation of `tool_layoutsubwindows` to layout the subwindows yourself. Note that just setting WIN_LAYOUT_LOCK to 1 and laying out your subwindows at create time is inadequate because you don't know when to change the subwindow layout.

Three functions return numbers useful for doing subwindow layout:

```
short   tool_stripeheight(tool)
    struct      tool *tool;
```

returns the height in pixels of the tool's name stripe. Note that the `tool` argument cannnot be NULL.

```
short   tool_borderwidth(tool)
    struct      tool *tool;
```

returns the width in pixels of the tool's outside border. If the caller supplies a null `tool` argument, the function returns the default border width.

```
short   tool_subwindowspacing(tool)
    struct      tool *tool;
```

returns the number of pixels that should be left as a margin between subwindows of a tool.

6.2.6. Subwindow Initialization

By the time `tool_createsubwindow` has returned, the subwindow is already inserted in the subtree growing out of the tool window; however, the subwindow will not perform any interesting function until `ts_io` and `ts_data` have been initialized. Normally, `tool_createsubwindow` is not directly called. Instead, the tool subwindow creation procedure for a subwindow type is called. The subwindow specific routine will call `tool_createsubwindow` and then initialize `ts_io` and `ts_data`.

6.2.7. Tool Installation

Once the tool is created and its subwindows have been created, the software interrupt system should be turned on via a call to `signal` as described in ***Window Change Notifications.*** At least SIGWINCH should be caught; if there are inferior processes in any of the subwindows, SIGCHLD should be added with any others as appropriate. Finally, the tool is installed into the display window tree by a call to:

```
tool_install(tool)
    struct      tool *tool;
```

At this point, the tool is operating; in fact, it will probably shortly receive a SIGWINCH asynchronously to paint its window(s) for the first time.

6.2.8. Tool Destruction

Explicitly destroying a tool as it reaches the end of its processing allows the system to reclaim resources and remove the windows gracefully. The procedure to invoke this cleanup is:

```
tool_destroy(tool)
    struct      tool *tool;
```

`tool_destroy` will destroy every subwindow of the indicated tool as part of its processing, so the subwindows need not be destroyed explicitly. Each subwindow's `ts_destroy` procedure gets called, so they can clean up gracefully. The pointer passed to `tool_destroy` must never be dereferenced after that call, since it is no longer valid.

A single subwindow can be destroyed by an explicit call to:

```
tool_destroysubwindow(tool, subwindow)
    struct      tool *tool;
    struct      toolsw *subwindow;
```

6.2.9. Programmatic Tool Creation

This section contains considerations if you are programmatically spawning processes that contain tools.

6.2.9.1. Forking the Tool

A tool has its own process. The creation of that process does not differ significantly from the normal paradigm for process creation. If it is to be started by a menu command or some other procedural interface, it is appropriate for the creating process to do the fork and return from the procedure call. When the child process dies, the parent process should catch the SIGCHLD signal and clean up. See the `wait3`(2) system call. SIGCHLD indicates to a parent process that a child process has changed state.

6.2.9.2. Environment Parameters

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

Every tool must be given the name of its ***parent window***. A tool's parent window is the window in the display tree under which the tool 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 tool, this will usually be the name of the root window of the window system.

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

sets WINDOW_PARENT to `windevname`.

```
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 contains the font file name that will be used as the tool's default (see `pf_default`).

Another parameter, WINDOW_INITIALDATA, describes the screen placement of a tool, and whether it should be open or iconic. WINDOW_INITIALDATA contains the coordinates of two rectangles, as well as one flag. The rectangles describe the placement and size of the open and closed window, and the flag is a boolean that is non-zero if the tool should start out iconic.

The process that is starting the tool may set WINDOW_INITIALDATA before it forks (`wmgr_forktool` does this; see *Suntools: User Interface Utilities*). After the fork, `tool_make` interrogates these variables. The routines to do this are in the library `/usr/lib/libsunwindow.a`.

```
we_setinitdata(rnormal, riconic, iflag)
    struct      rect *rnormal, *riconic;
    int         iflag;
```

sets the environment variable in the parent process, and

```
we_getinitdata(rnormal, riconic, iflag)
    struct      rect *rnormal, *riconic;
    int         *iflag;
```

reads those values in the child process. A non-zero return value means that the WINDOW_INITIALDATA parameter couldn't be found.

A procedure is provided for unsetting WINDOW_INITIALDATA for tools that are going to provide windows for other processes to run in. This procedure prevents a wayward child process from being confused by the incorrectly set environment variable:

```
we_clearinitdata()
```

6.3. Tool Processing

The main loop of a normal tool is encapsulated inside a call to:

```
tool_select(tool, waitprocessesdie)
    struct      tool *tool;
    int         waitprocessesdie;
```

This procedure is the notification distributer used for event-driven program control flow. When some input event, timeout or signal interrupt is detected inside `tool_select`, a call to a notification handler is made, paasing in enough information to identify what happened, and to which window. When the handler returns, `tool_select` awaits another event. The `waitprocessesdie` argument is discussed below in ***Child Process Management.***

6.3.1. Toolio Structure

The `toolio` data structure in each `toolsw` structure holds what is needed for a subwindow to wait for something to happen in the `tool_select` call. The `tool` structure uses the `toolio` data structure within itself to wait for input too. It is defined in `<suntool/tool.h>`.

```
struct  toolio {
    int         tio_inputmask,
    int         tio_outputmask,
    int         tio_exceptmask;
    struct      timeval *tio_timer;
    int         (*tio_handlesigwinch) ();
    int         (*tio_selected) ();
};
```

`tio_inputmask`, `tio_outputmask`, `tio_exceptmask` and `tio_timer` fields are analogous to the last four arguments to the `select` system call. `tio_inputmask` has the bit "1<<*f*" set for each file descriptor *f* on which a window wants to wait for input. Similarly, `tio_outputmask` and `tio_exceptmask` indicate an interest in *f* being ready for writing and having an exceptional condition pending, respectively. There are currently no "exceptional conditions" implemented; this field provides compatibility with the `select` system call.

If `tio_timer` is a non-zero pointer, it specifies a maximum interval to wait for one of the file descriptors in the masks to require attention. If `tio_timer` is a zero pointer, an infinite timeout is assumed. To effect a poll, the `tio_timer` argument should be non_zero, pointing to a `timeval` structure with all zero fields.

`toolio` also contains pointers to the procedures that are called when the tool has received some notification. `tio_handlesigwinch` addresses the procedure that responds to the SIGWINCH signal. This procedure handles repaint requests and window size changes. The general form for such a procedure is:

```
sigwinch_handler(data)
    caddr_t     data;
```

Such procedures take a single argument `data` whose type is context-dependent. For a tool this data is a pointer to the `tool` structure. For a subwindow this data is the `ts_data` value in the `toolsw` structure.

`tio_selected` addresses the procedure which responds to notifications from the `select` system call. The procedure's calling sequence is:

```
io_handler(data, ibits, obits, ebits, timer)
    caddr_t     data;
    int         *ibits,
    int         *obits,
    int         *ebits,
    struct      timeval **timer;
```

In such procedures, the `data` argument is like that of the SIGWINCH handlers described above. The three integer pointers indicate which file descriptors are ready for reads (`*ibits`), writes (`*obits`), or exception-handling (`*ebits`). If `timer` is NULL, this window was not waiting on any timeout. If `*timer` points to a valid `struct timeval` then this window is waiting for a timeout. If both the `(*timer)->tv_sec` and `(*timer)->tv_usec` are zero, the timeout has just happened for this window and should be serviced. The data in the file descriptor masks is not defined if a timeout has occurred.

Before returning from a procedure of this type, the masks and timer must be reset by storing through the pointers passed in the arguments; the values should be consistent with the discussion of the masks and timer pointer above. You may not want to reset the timer if you are using it as a countdown timer, and it still has time remaining on it.

6.3.2. File Descriptor and Timeout Notifications

`tool_select` generates three composite masks by merging the corresponding masks from all of the `toolio` structures in the tool. The input mask is special in that if all the masks in a particular `toolio` structure are zero, an entry in the composite input mask is made for the associated window anyway. `tool_select` also determines the shortest timeout that any of the windows is waiting on. The composite masks and shortest timeout are passed to the `select` system call.

When the `select` system call returns normally, windows that have a match between their masks and the mask of ready file descriptors that have timed out are notified via their `tio_selected` procedure. Each `tio_selected` procedure is called with the complete ready masks, not just the intersection of its own masks and the ready masks. However, a `tio_selected` procedure is called with its own window's timer value.

Each window that has been selected as a result of the `select` system call is notified. The order of notification is not defined. Problems will arise if there are multiple non-cooperating windows waiting on the same device.

It should be noted that timers in this implementation are only approximate. When the `select` system call returns and a timeout hasn't occurred, the `select` is assumed to have been instantaneous. Also, the time taken up with handling notifications is not deducted from the timers.

6.3.3. Window Change Notifications

Clients of the tool interface must catch the SIGWINCH signal. A signal catcher can be set up via the `signal`(3) library call. That catcher is then responsible for notifying the tool package that the signal has arrived. This is done by calling:

```
tool_sigwinch(tool)
    struct      tool *tool;
```

This procedure simply sets the TOOL_SIGWINCHPENDING flag in `tool`. The receipt of any signal has the side effect of causing the `select` system call in `tool_select` to return abnormally. The TOOL_SIGWINCHPENDING flag is noticed and the tool's `tio_handlesigwinch` procedure is called. The default `tio_handlesigwinch` procedure does some processing, which may include changing the subwindow layout, and eventually calls all its subwindows' `tio_handlesigwinch` procedures.

6.3.4. Child Process Maintenance

`tool_select` also gathers up dead children processes of the tool. The `waitprocessesdie` argument to `tool_select` is provided for tools which have separate processes behind some of their subwindows. Such tools must explicitly catch SIGCHLD, the signal that indicates to a parent process that a child process has changed state. Then the signal handler, parallel to a SIGWINCH catcher and `tool_sigwinch`, should call:

```
tool_sigchld(tool)
    struct      tool *tool;
```

This call causes `tool_select` to try to gather up a dead child process via a `wait3` system call (see `wait`(2)). When as many child processes have been gathered up as indicated by the `waitprocessesdie` argument to `tool_select`, `tool_select` returns.

6.3.5. Changing the Tool's Attributes

Tool attributes may be changed even after a tool has been created. `tool_set_attributes` specifies changes to tool attributes.

```
/* VARARGS */
int tool_set_attributes(tool, id, value, id, value, ... 0)
    struct      tool *tool;
    int         id;
    caddr_t     value;
```

`tool_set_attributes` takes a variable number of attribute identifier/value pairs, terminated by the special attribute identifier 0. A list of valid attributes is available in the section *Tool Attributes*. `id`'s are the attribute identifiers. `value`'s are the attribute values of the preceding `id`. This routine returns 0 if all the arguments are OK, −1 otherwise. All feedback is taken care of, e.g., when setting the label, the name stripe is redisplayed. Repainting is only done once at the end of the `tool_set_attributes` call.

All arguments passed into `tool_set_attributes` are copied. Thus, all accesses of attribute values must use `tool_get_attribute`.

```
caddr_t tool_get_attribute(tool, id)
    struct      tool *tool;
    int         id;
```

`tool_get_attribute` allows the programmer to determine the value of the attribute identified by `id` at any time in the life of the tool. The return value of the function is the value of the attribute. If `id` is not understood then −1 is returned. The returned value is either a 32 bit non-dynamically allocated quantity or a pointer to dynamically allocated storage. The type of the return value depends on the attribute and will usually need to be cast into that type. For pointer values, `tool_free_attribute` must be called to release the storage allocated during this call.

```
tool_free_attribute(id, value)
    int         id;
    caddr_t     value;
```

`tool_free_attribute` releases the storage allocated during `tool_get_attribute` or `tool_find_attribute` calls. If `id`'s value is defined as a non-dynamically allocated quantity, then `value` is not freed and this call does nothing.

6.3.6. Terminating Tool Processing

During the time that `tool_select` is acting as the main loop of the program, a call to:

```
tool_done(tool)
    struct      tool *tool;
```

causes the flag TOOL_DONE to be set in `tool`. `tool_select` notices this flag, and then returns gracefully.

6.3.7. Replacing Toolio Operations

Since the `toolio` structure contains procedure pointers in variables, it is possible to customize the behavior of a window by replacing the default values.

Icons that respond to user inputs or that update their image in response to timer or other events, may be implemented by replacing the tool's `tio_selected` procedure. A different subwindow layout scheme may be implemented in a replacement procedure for `tio_handlesigwinch`. Note that these modifications do not require changes to existing libraries; the address of the substitute routine is simply stored in the appropriate slot at run-time. However, the substitute routine must either do all of the processing handled by the original library routine, or the substitute routine should do its special processing and then call the original library routine.

6.3.8. *Boilerplate Tool Code*

Here is the boilerplate code for a simple tool. It illustrates the order in which things should be done in a tool. All of the window related calls have been discussed in this chapter.

```
#include <stdio.h>
#include <suntool/tool_hs.h>
static  struct tool *tool;

main(argc, argv)
        int argc;
        char **argv;
{
        char    **tool_attrs = NULL;
        char    *tool_name = argv[0];
        static  int sigwinchcatcher();

        argv++;
        argc--;
        /* Pick up command line arguments to modify tool behavior */
        if (tool_parse_all(&argc, argv, &tool_attrs, tool_name) == -1) {
                tool_usage(tool_name);
                exit(1);
        }
        /* Get application specific args */
        while (argc > 0 && **argv == '-') {
                /* Parse switches */
                argv++;
                argc--;
        }
        /* Create tool window */
        tool = tool_make(
            WIN_LABEL,          tool_name,
            WIN_ATTR_LIST,      tool_attrs,
            0);
        if (tool == (struct tool *)NULL)
                exit(1);
        tool_free_attribute_list(tool_attrs);
        /* ...Create tool subwindows... */
        /* Install tool in tree of windows */
        (void) signal(SIGWINCH, sigwinchcatcher);
        tool_install(tool);
        /* Run notifier */
        tool_select(tool, 0);
        /* Cleanup */
        tool_destroy(tool);
        exit(0);
}

static
sigwinchcatcher() { tool_sigwinch(tool); }
```

6.3.9. Old Style Tool Creation

`tool_make` is the recommended call to use when creating a tool window. `tool_create` is an out-dated call that used to do this for you. While `tool_create` still works, it is not recommended. Here is `tool_create` documentation.

A tool is created by a call to:

```
struct tool *tool_create(name, flags, normalrect, icon)
    char        *name;
    short       flags;
    struct      rect *normalrect;
    struct      icon *icon;

#define  TOOL_NAMESTRIPE         0x01
#define  TOOL_BOUNDARYMGR        0x02
```

`name`	is the name of the tool. This is what will be displayed in the tool's name stripe if TOOL_NAMESTRIPE is set in the flag's argument. It also appears on the default icon.
`flags`	has the flags TOOL_NAMESTRIPE and/or TOOL_BOUNDARYMGR set as those properties are desired. (TOOL_BOUNDARYMGR enables boundaries that the user can move between subwindows.)
`normalrect`	describes the initial position and size of the tool in its normal open state in the coordinate system of the tool's parent, which is typically the window for the screen.
`icon`	is a pointer to an `icon` struct, if the client wants a special icon.

`normalrect` and the `icon` may be defaulted by passing NULL for their arguments. The default icon is described, along with considerations for making custom icons, in ***Suntool: User Interface Utilities***; the choice is strictly a matter of convenience vs. ambition. A tool's starting position should almost always be left NULL; it could be the result of WE_GETINITDATA that is going into `normalrect`.

Note, `tool_display` is an outdated tool operation that has been taken over by `tool_set_attributes`. During processing, a call to:

```
tool_display(tool)
    struct      tool *tool;
```

redisplays the entire tool. This is useful if some change has been made to the image of the tool itself, for instance if its name or its icon's image have been changed. Normal repaints in response to size changes or damage should not use this procedure. They will be taken care of by SIGWINCH events and their handlers.

Chapter 7

Suntool: Subwindow Packages

This chapter describes ***subwindow packages***, the building blocks for constructing a ***tool***. It presents a guide for building new subwindow packages of general utility and describes the available standard subwindow packages for use with ***suntools***. Refer to ***Suntool: Tools and Subwindows*** for a description of the overall structure of tools and the general notion of a subwindow.

Subwindows, as presented here, are designed to be independent of the particular framework in which they are used. That is, a subwindow is a merger of window handling and application processing which should be valid in frameworks other than the ***tool*** structure and ***suntool*** environment described in the preceding chapter. The design avoids any dependence on those constructs. Thus, a subwindow package can be used in another user interface system written on top of the ***sunwindow*** basic window system. However, subwindow packages all provide a utility for creating a subwindow in the ***tool*** context.

7.1. Minimum Standard Subwindow Interface

This section describes the minimum programming interface one should define when writing a new subwindow package. A subwindow implementation should provide all the facilities described here. This section presents the arguments to the following standard procedures. Each subwindow package need only document any additional arguments passed to its ***create/init*** procedures. There is a set of naming conventions that provides additional consistency between subwindow package interfaces.

For the purpose of example, we use `proto` as the prefix. Other prefixes used in existing subwindow packages include `tty`, `gfx` and `msg`.

Each subwindow package has a structure definition that contains all the data required by a single instance of the subwindow.

```
struct  protosubwindow {
    int         fsw_windowfd;
    struct      pixwin *fsw_pixwin;
  ...
};
```

The structure definition typically has a ***pixwin*** for screen access and a window handle for identification as part of this data. The information that the subwindow's procedures need should be stored in this data structure; this may entail redundantly storing some data that is in the associated containing data structure, such as the `toolsw` struct. Having an object per subwindow allows multiple instantiations of a subwindow package in a single-user process. The following function creates new instances of a proto-subwindow:

```
struct  protosubwindow *protosw_init(windowfd, ...)
    int         windowfd;
```

`windowfd` is to be a proto-subwindow. The "..." indicates that many subwindow packages will require additional set-up arguments. This routine typically opens a pixwin, sets its input mask as described in ***Input to Application Programs***, and dynamically allocates and fills the subwindow's data object. If the returned value is NULL then the operation failed.

```
protosw_done(protosw)
    struct      protosubwindow *protosw;
```

destroys subwindow instance data. Once this procedure is called, the `protosw` pointer should no longer be referenced.

```
protosw_handlesigwinch(protosw)
    struct      protosubwindow *protosw;
```

This procedure handles repaint requests and must also detect and deal with changes in the window size. It is called as an eventual result of some other procedure catching a SIGWINCH.

```
protosw_selected(protosw, ibits, obits, ebits, timer)
    struct      protosubwindow *protosw;
    int         *ibits,
    int         *obits,
    int         *ebits,
    struct      timeval **timer;
```

handles event notifications. Subwindow packages that don't accept input may not have a procedure of this type. The semantics of this procedure are fully described in the preceding chapter in the section entitled ***Toolio Structure***.

```
struct  toolsw *protosw_createtoolsubwindow(tool, name, width, height, ...)
    struct      tool *tool;
    char        *name;
    short       width, height;
```

creates a struct `toolsw` that is a proto-subwindow. `protosw_createtoolsubwindow` is only applicable in the ***tool*** context. It is often the only call that an application program need make to set up a subwindow of a given type. `tool` is the handle on the tool that has already been created. `name` is the name that you want associated with the subwindow. `width` and `height` are the dimensions of the subwindow as wanted by the `tool_createsubwindow` call. The "..." indicates that many subwindow packages will require additional arguments. These additional arguments should parallel those in `protosw_init`. If the returned value is NULL then the operation failed.

`protosw_createtoolsubwindow` takes the window file descriptor it gets from `tool_createsubwindow`, passes it to `protosw_init`, and stores the resulting pointer in the tool subwindow's `ts_data` slot. The addresses of `protosw_handlesigwinch` and `protosw_selected` are stored in the appropriate slots of the `toolio` structure for the tool subwindow, and the address of `protosw_done` is stored in the tool subwindow's `ts_destroy` procedure slot.

Of course, most subwindow packages define functions that perform application-specific processing; the ones described here are merely the permissible minimum.

7.2. Empty Subwindow

The empty subwindow package simply serves as a place holder. It does nothing but paint itself gray. It expects the window it is tending to be taken over by another process as described in *Graphics Subwindow*. When the other process is done with the empty subwindow package, the caretaker process resumes control.

A private data definition that contains instance-specific data defined in `<suntool/emptysw.h>` is:

```
struct  emptysubwindow {
    int          em_windowfd;
    struct       pixwin *em_pixwin;
};
```

`em_windowfd` is the file descriptor of the window that is tended by the empty subwindow. `em_pixwin` is the structure for accessing the screen.

```
struct  toolsw *esw_createtoolsubwindow(tool, name, width, height)
    struct       tool *tool;
    char         *name;
    short        width, height;
```

sets up an empty subwindow in a tool window. If the returned value is NULL then the operation failed. Since `esw_createtoolsubwindow` takes care of setting up the empty subwindow, the reader may not be interested in the remainder of this section.

```
struct  emptysubwindow *esw_init(windowfd)
    int          windowfd;
```

creates a new instance of an empty subwindow. `windowfd` is the window to be tended. If the returned value is NULL then the operation failed.

```
esw_handlesigwinch(esw)
    struct       emptysubwindow *esw;
```

handles SIGWINCH signals. If the process invoking this procedure is the current owner of `esw->em_windowfd`, gray is painted in the window. If it is not the current owner, it checks to see if the current owner is still alive. If the current owner is dead, this process takes over the windows again and paints gray in the window.

```
esw_done(esw)
    struct       emptysubwindow *esw;
```

destroys the subwindow's instance data.

Processes that take over windows should follow guidelines discussed in *Overlapped Windows: Imaging Facilities* concerning the use of the `win_getowner` and `win_setowner` procedures. Preferably, the graphics subwindow interface described below should be used for this activity.

7.3. Graphics Subwindow

The graphics subwindow package is for programs that need a single window in which to draw. Using this subwindow package insulates programmers of this type of program from much of the complexity of the window system.

Users of this interface have the additional benefit of being able to invoke their programs from outside the window system. Thus, you can write one program and have it run both inside and outside the window system. This situation is actually an illusion. What really happens when running outside the window system is that the window system is actually started up and that a single window is created in which the graphics subwindow package runs.

The graphics subwindow can also manage a retained window for the programmer. The programmer need not worry about the fact that he is in an overlapping window situation. A backup copy of the bits on the screen is maintained from which to service any repaint requests.

Appendix C contains programs based on graphics subwindows.

The graphics subwindow can be used in tool building like any of the other subwindow packages described in this chapter. However, the graphics subwindow also provides the ability for a program to run on top of an existing window by using the blanket window mechanism.

The data definition for the instance-specific data defined in `<suntool/gfxsw.h>` is:

```
struct  gfxsubwindow {
        int         gfx_windowfd;
        int         gfx_flags;
        int         gfx_reps;
        struct      pixwin *gfx_pixwin;
        struct      rect gfx_rect;
        caddr_t     gfx_takeoverdata;
};

#define GFX_RESTART      0x01
#define GFX_DAMAGED      0x02
```

`gfx_windowfd` is the file descriptor of the window that is being accessed. `gfx_reps` are the number of repetitions that continuously running (non-blocking) cyclic programs are to execute. `gfx_pixwin` is the structure for accessing the screen. `gfx_rect` is a cached copy of the window's current self relative dimensions. `gfx_takeoverdata` is data private to the graphics subwindow package.

`gfx_flags` contains bits that the client program interprets. The GFX_DAMAGED bit is set by the graphics subwindow package whenever a SIGWINCH has been received. In addition, the GFX_RESTART bit is set if the size of the window has changed or the window is not retained. The client program must examine these flags at the times described below.

GFX_DAMAGED means that `gfxsw_handlesigwinch` should be called. This flag should be examined and acted upon before looking at GFX_RESTART. GFX_RESTART is often interpreted by a graphics program to mean that the image should be scaled to a new window size and that the image should be redrawn. Many continuous programs, graphics demos for instance, redraw from the beginning of a cycle. Other event-driven programs, graphics editors and status windows, for example, redraw from their underlying data descriptions. The GFX_RESTART bit needs to be reset to 0 by the client program before actually doing any redrawing.

7.3.1. In a Tool Window

A graphics subwindow in a ***tool*** context is only applicable for event-driven programs that use the `tool_select` mechanism. Any subwindow in a tool must use this notification mechanism so that all the windows are able to cooperate in the same process.

```
struct  toolsw *gfxsw_createtoolsubwindow(tool, name, width, height, argv)
    struct      tool *tool;
    char        *name;
    short       width, height;
    char        **argv;
```

sets up a graphics subwindow in a tool window. If `argv` is not zero, this array of character pointers is processed like a command line in a standard way to determine whether the window should be made retained "–r" and/or what value should be placed in `gfx_reps` "–n ####". If the returned value is NULL then the operation failed. It is the responsibility of the client to set up `toolsw->ts_io.tio_selected` if the client is to process input through the graphics subwindow.

It is also the responsibility of the client to replace `toolsw->ts_io.tio_handlesigwinch` with the client's own routine to notify the client when something about his window changes. The client `tio_handlesigwinch` will call `gfxsw_interpretesigwinch` described below.

```
gfxsw_getretained(gfxsw);
    struct      gfxsubwindow *gfxsw;
```

can be called to make a graphics subwindow retained if you choose not to do the standard command line parsing provided by `gfxsw_createtoolsubwindow`. It should be called immediately after the graphics subwindow is created. Destroying `gfxsw->gfx_prretained` has the effect of making the window no longer retained.

The procedure:

```
gfxsw_interpretesigwinch(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

is called from the client `tio_handlesigwinch` to give the graphics subwindow package a chance to set the bits in `gfxsw->gfx_flags`. The code in the client `tio_handlesigwinch` then checks the flags and responds appropriately, perhaps by calling the `gfxsw_handlesigwinch` procedure that handles SIGWINCH signals:

```
gfxsw_handlesigwinch(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

If the window is retained and the window has not changed size, this routine fixes up any part of the image that has been damaged. If the window is retained and the window has changed size, this routine frees the old retained pixrect and allocates one of the new size. If the window is not retained, the damaged list associated with the window is thrown away. The GFX_DAMAGED flag is reset to zero in this routine.

The procedure:

```
gfxsw_done(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

destroys the subwindow's instance data.

7.3.2. Overlaying an Existing Window

The graphics subwindow provides the ability for a program to overlay an existing window. The empty subwindow described above is designed to be overlaid.

The following procedure creates a new instance of a graphics subwindow in something other than the ***tool*** context:

```
struct  gfxsubwindow *gfxsw_init(windowfd, argv)
    int         windowfd;
    char        **argv;
```

`windowfd` should be zero; the assumption is that there is some indication in the environment as to which window should be overlayed. See `we_getgfxwindow` in ***Window Manipulation*** for more information. `argv` is like `argv` in `gfxsw_createtoolsubwindow`. In addition, arguments similar to the ones recognized by `win_initscreenfromargv` are parsed. Thus, the program can be directed to run on a particular screen. If the returned value is NULL then the operation failed.

When a screen is created from scratch, window system keyboard and mouse processing are not turned on. `gfxsw_setinputmask` should be called instead of `win_setinputmask` when defining window input (see below) in order to enable window system keyboard and mouse processing. This mechanism is used to allow programs that listen to the standard input to still run when started from outside the window system.

`gfx_takeoverdata` in the returned `gfxsubwindow` data structure is not zero in this case. The structure of the data that this pointer refers to is private to the implementation of the graphics subwindow.

When a graphics subwindow has overlayed another window, various signal catching routines are set up if the corresponding signals have no currently defined handler routines.

The `gfxsw_catchsigwinch` procedure is set up as the signal catcher of SIGWINCH:

```
gfxsw_catchsigwinch()
```

It, in turn, calls `gfxsw_interpretesigwinch`.

The `gfxsw_catchsigtstp` procedure is set up as the signal catcher of SIGTSTP:

```
gfxsw_catchsigtstp()
```

The graphics subwindow is removed from the display tree. The pixwin of the graphics subwindow is reset. SIGSTOP is sent to the the graphics subwindow's own process.

The `gfxsw_catchsigcont` procedure is set up as the signal catcher of SIGCONT:

```
gfxsw_catchsigcont()
```

The graphics subwindow is inserted back into the display tree (presumably after `gfxsw_catchsigtstp` removed it).

Continuous programs that never use a select mechanism should examine `gfxsw->gfx_flags` in their main loop. Other programs that would like to use a select mechanism to wait for input/timeout should call:

```
gfxsw_select(gfxsw, selected, ibits, obits, ebits, timer)
    struct      gfxsubwindow *gfxsw;
    int         (*selected)(), ibits, obits, ebits;
    struct      timeval *timer;
```

as a substitute for the `tool_select`. `selected` is the routine that is called when some input or timeout is noticed. Its calling sequence is exactly like `protosw_selected` described at the beginning of this chapter. The only difference in the semantics of this routine and `protosw_selected` is that the `gfxsw->gfx_flags` should be examined and acted upon in `selected`. `selected` may be called with no input pending so that you are able to see the flags when they change.

`ibits`, `obits`, `ebits` and `timer`, as well as `gfxsw` and `selected`, can be thought of as initializing an internal `toolio` structure, which is then fed to the `tool_select` mechanism.

A substitute for the `tool_done` procedure is:

```
gfxsw_selectdone(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

`gfxsw_selectdone` is called from within the `selected` procedure passed to `gfxsw_select`.

Programs that are not using the mouse can call:

```
gfxsw_notusingmouse(gfx)
    struct      gfxsubwindow *gfx;
```

In certain cases, when the graphics subwindow is the only window on the display for instance, some efficiency measures can be taken. In particular, pixwin locking overhead can be reduced.

```
gfxsw_setinputmask(gfx, im_set, im_flush, nextwindownumber, usems, usekbd)
    struct      gfxsubwindow *gfx;
    int         nextwindownumber;
    struct      inputmask *im_set, *im_flush;
    int         usems, usekbd;
```

The calling sequence is essentially that of `win_setinputmask`. `usems` being non-zero means that mouse input is wanted and so the mouse is turned on for the screen (if currently off). `usekbd` being non-zero means that keyboard input is wanted and so the keyboard is turned on for the screen (if currently off). See `gfxsw_init` (above) for a rationale for using `gfxsw_setinputmask` instead of `win_setinputmask`.

```
gfxsw_inputinterrupts(gfx, ie)
    struct      gfxsubwindow *gfx;
    struct      inputevent *ie;
```

This utility looks at `*ie`. If `*ie` is a character that (on a tty) normally does process control (interrupts the process, dumps core, stops the process, terminates the process), it does the similar action. This routine is meant to be a primitive substitute for tty process control while using the window input mechanism.

Remember to call `gfxsw_done` to "give back" the window that was taken over.

7.4. Message Subwindow

The message subwindow is an extremely simple facility. If you are not concerned about the size of the client's object code, or if the client already employs a panel subwindow, you should consider using the panel subwindow with a single message item instead of the message subwindow. This is because the panel subwindow provides superior functionality and a cleaner interface than does the message subwindow. Please see the chapter entitled *The Panel Subwindow Package* for further information on panels.

The message subwindow package displays simple ASCII strings.

A private data definition that contains instance-specific data defined in `<suntool/msgsw.h>` is:

```
struct  msgsubwindow {
    int          msg_windowfd;
    char*        msg_string;
    struct       pixfont *msg_font;
    struct       rect msg_rectcache;
    struct       pixwin *msg_pixwin;
};
```

`msg_windowfd` is the file descriptor of the window that is the message subwindow. `msg_string` is the string being displayed using `msg_font`. Only printable characters and blanks are properly dealt with, ***not*** carriage returns, line feeds or tabs. The implementation uses `msg_rectcache` to help determine if the size of the subwindow has changed. `msg_pixwin` is the structure that accesses the screen.

```
struct  toolsw *msgsw_createtoolsubwindow(tool, name, width, height,
                                            string, font)
    struct       tool *tool;
    char         *name;
    short        width, height;
    char         *string;
    struct       pixfont *font;
```

is the call that sets up a message subwindow in a tool window. `string` is the string being displayed using `font`. If the returned value is NULL then the operation failed. Since `msgsw_createtoolsubwindow` takes care of the set-up of the message subwindow, the reader may not be interested in the remainder of this section, except for `msgsw_setstring`.

The following function creates a new instance of a message subwindow:

```
struct  msgsubwindow *msgsw_init(windowfd, string, font)
    int          windowfd;
    char         *string;
    struct       pixfont *font;
```

`windowfd` identifies the window to be used. `string` is the string being displayed using `font`. If the returned value is NULL then the operation failed.

```
msgsw_setstring(msgsw, string)
    struct       msgsubwindow *msgsw;
    char         *string;
```

changes the existing `msgsw—>msg_string` to `string` and redisplays the window.

```
msgsw_display(msgsw)
    struct       msgsubwindow *msgsw;
```

redisplays the window.

```
msgsw_handlesigwinch(msgsw)
    struct       msgsubwindow *msgsw;
```

is called to handle SIGWINCH signals. It repairs the damage to the window if the window hasn't changed size. If the window has changed size, the string is reformatted into the new size.

```
msgsw_done(msgsw)
    struct       msgsubwindow *msgsw;
```

destroy's the subwindow's instance data.

7.5. Terminal Emulator Subwindow

The terminal emulator subwindow mimics a standard Sun terminal. It accepts most of the same ANSI escape sequences as the Sun terminal (see `cons(4s)` in the *System Interface Manual*). However, certain control sequences cause the terminal emulator subwindow to behave differently from the normal Sun terminal. The table following lists these control sequences and their effects.

Definitions for the use of the terminal emulator subwindow are in `<suntool/ttysw.h>`. *Note:* Only one tty subwindow per process is allowed.

Table 7-1: Differences between Sun terminal and SunWindows tty emulator

Control sequence	*Synopsis*	*Behavior in SunWindows tty emulator*
CTRL-G (0x07)	Bell	Flashes window.
ESC [p	Black on white	No effect.
ESC [q	White on black	No effect.
ESC [0r	Enable vertical wrap mode[3]	No effect.
ESC [s	Reset	No effect.

"ESC" indicates the ASCII escape character (0x1B).

```
struct  toolsw *ttysw_createtoolsubwindow(tool, name, width, height)
    struct      tool *tool;
    char        *name;
    short       width, height;
```

is the call that sets up a terminal emulator subwindow in a tool window. `ttysw_createtoolsubwindow` takes care of setting up the terminal emulator subwindow except for the forking of the program. Thus, clients of this routine may want to ignore the remainder of this section except for the discussion of `ttysw_fork` and perhaps `ttysw_becomeconsole`. `ttysw_createtoolsubwindow` returns NULL on failure.

```
caddr_t ttysw_init(windowfd)
    int         windowfd;
```

creates a new instance of a tty subwindow. `windowfd` is the window that is to be used. `ttysw_init` returns NULL on failure.

```
ttysw_becomeconsole(ttysw)
    caddr_t     ttysw;
```

sets up the terminal emulator to receive any output directed to the console. This should be called after calling `ttysw_init`.

```
ttysw_saveparms(ttyfd)
    int         ttyfd;
```

should be called by the screen initialization program, e.g., `suntools`(1). This saves the characteristics of the terminal `ttyfd` in an environment variable. Terminal emulation

---

[3] Note that the zero in this escape sequence may be replaced by an integer, in order to set up jump scrolling. Positive integer arguments do supply the desired effect.

processes forked from the screen initialization process will get their characteristics from this environment variable; terminal emulation processes started directly from shells get their characteristics from the standard error tty. `ttysw_saveparms` is needed because a screen initialization program is often started from the console, whose characteristics can change due to console redirection.

```
ttysw_handlesigwinch(ttysw)
    caddr_t     ttysw;
```

is called to handle SIGWINCH signals. On a size change, the terminal emulator's display space is reformatted. Also, its process group is notified via SIGWINCH that the size available to it is different. Refer to *TTY-Based Programs in TTY Subwindows.* If there is display damage to be fixed up, the terminal emulator redisplays the image by using character information from its screen description.

```
ttysw_selected(ttysw, ibits, obits, ebits, timer)
    caddr_t     ttysw;
    int         *ibits, *obits, *ebits;
    struct      timeval **timer;
```

reads input and writes output for the terminal emulator. `*ibits`, `*obits` and `*timer` are modified by `ttysw_selected`. See the general discussion of `tio_selected` type procedures in *Minimum Standard Subwindow Interface.*

```
int  ttysw_fork(ttysw, argv, inputmask, outputmask, exceptmask)
    caddr_t     ttysw;
    char        **argv;
    int         *inputmask, *outputmask, *exceptmask;
```

forks the program indicated by `*argv`. The identifier of the forked process is returned. If the returned value is −1 then the operation failed and the global variable `errno` contains the error code. There are the following possibilities:

- If `*argv` is NULL, the user SHELL environment value is used. If this environment parameter is not available, */bin/sh* is used.
- If `*argv` is "−c", this flag and `argv[1]` are passed to a shell as arguments. The shell then runs `argv[1]`. The argument list for this case becomes *shell* `-c argv[1] 0`.
- If `*argv` is not NULL, the program named by `argv[0]` is run with the arguments given in the rest of `argv`. The argument list should be NULL terminated.

The arguments `*inputmask`, `*ouputmask`, `*exceptmask` are dereferenced by `ttysw_fork` and set to the values that the terminal eumlator subwindow manager wants to wait on in a subsequent `select`(2) call.

```
ttysw_done(ttysw)
    caddr_t     ttysw;
```

destroys the subwindow's instance data.

7.5.1. The Tool Specific TTY Subwindow Type

The tool terminal emulator subwindow, called the ***tty tool subwindow***, extends the basic terminal emulator subwindow. A tty tool subwindow is a super class of a straight terminal emulator subwindow. This means that a tty tool subwindow can do what a straight terminal emulator subwindow can, and more. In particular, a tty subwindow knows about tool windows and allows terminal-emulator-based programs to set/get data about the tool window. Also, the user can send window management commands to change the tool window via the keyboard.

The only public access to a tty tool subwindow is its create/destroy procedures, `ttytlsw_createtoolsubwindow` and `ttytlsw_done`. Other than this, think of the subwindow as a straight terminal emulator subwindow.

The following table shows the escape sequences that can be sent to a tty tool subwindow. Do not send these escape sequences to a straight terminal emulator subwindow, because they will be ignored.

Table 7-2: Escape sequences for tty tool subwindow

Escape sequence[4]	*Description*
`\E[1t`	Opens a tool.
`\E[2t`	Closes a tool.
`\E[3t`	Moves the tool with interactive feedback.
`\E[3;TOP;LEFTt`	Moves the tool so that its top left corner is at `TOP;LEFT`. `TOP` and `LEFT` are in pixels.
`\E[4t`	Stretches a tool with interactive feedback.
`\E[4;WIDTH;HTt`	Stretches a tool to `WIDTH` and `HT`. `WIDTH` and `HT` are in pixels.
`\E[5t`	Exposes a hidden tool.
`\E[6t`	Hides a tool.
`\E[7t`	Refreshes the tool window.
`\E[8;ROWS;COLSt`	Stretches the tool so that its width and height are `ROWS` and `COLS`, respectively.
`\E[11t`	Reports if the tool is open or iconic by sending `\E[1t` (open) or `\E[2t` (close) sequence.
`\E[13t`	Reports the tool's position by sending the `\E[3;TOP;LEFT` sequence.
`\E[14t`	Reports the tool's size in pixels by sending the `\E[4;WIDTH;HEIGHT` sequence.
`\E[18t`	Reports the tool's size in characters by sending an `\E[8;ROWS;COLSt` sequence.
`\E[20t`	Reports an icon label by sending an `\E[L` sequence (see below).
`\E[21t`	Reports the tool's namestripe by sending an `\E]1` sequence (see below).
`\E]l<text>\E\ `	Sets the tool's namestripe to `<text>`.
`\E]I<file>\E\ `	Sets the icon to the icon contained in `<file>`.
`\E]L<label>\E\ `	Sets the icon label to `<label>`.
`\E[>OPT; . . .h`	Turns `OPT` on. The only currently defined `OPT` value is 1 (PAGEMODE). For example, `\E[>1h`.
`\E[>OPT; . . .k`	Turns `OPT` off.
`\E[>OPT; . . .l`	Reports the current `OPT` settings by sending an `\E[>OPTl` or `\E[>OPTh` sequence for each defined option.

---

[4] Note that "\E" is the *termcap* specification for <ESC>.

```
struct  toolsw *ttytlsw_createtoolsubwindow(tool, name, width, height)
        struct  tool *tool;
        char    *name;
        short   width, height;
```

`ttytlsw_createtoolsubwindow` has the same calling sequence as `ttysw_createtoolsubwindow`.

```
void    ttytlsw_done(ttysw)
        struct ttysubwindow *ttysw;
```

`ttytlsw_done` destroys the subwindow's instance data.

7.5.2. TTY-Based Programs in TTY Subwindows

TTY-based programs that use ***termcap*** to determine the size of their screen (such as ***more*** and ***vi***) need not know about windows to run under the terminal emulator. The ***termcap*** library routine `tgetent` will return the current number of lines and columns of the terminal emulator subwindow (see `termcap(3x)`). However, if the window size changes while one of these programs is running, the terminal emulator and the client program may disagree about what the terminal size is.

In the case of a size change, the terminal emulator sends a SIGWINCH signal to its process group. If a child process doesn't catch the signal, no harm is done because the default action for SIGWINCH is that the signal be ignored. A child process can catch the signal, and then perform an `ioctl` call to get the correct terminal size. Please refer to the header file `<sys/ioctl.h>` for a complete list of `ioctl` requests.

The terminal emulator and the ***termcap*** library communicate size information through `ioctl` system calls on the pseudo-tty shared by both. The terminal emulator makes a TIOCSSIZE `ioctl` call to set the size of the pseudo-tty. The ***termcap*** library or some other TTY-based program makes a TIOCGSIZE `ioctl` call to get the size of the pseudo-tty.

TTY-based programs running in a TTY subwindow should ***always*** use the `ioctl` TIOCGSIZE operation to determine the current size of the window, even if they use `tgetent`, because the window size could have changed before `tgetent` returns.

```
int  we_getmywindow(windowname)
    char        *windowname;
```

can be called by programs running under a window system pseudo-tty to find out the terminal emulator's window name. This information is passed from the terminal emulator process to a child process through the environment variable WINDOW_ME, which is set to be the subwindow's device name, for example `/dev/win5`. `we_getmywindow` reads the value of WINDOW_ME into `windowname`. A return value of 0 indicates success. `windowname` should point to at least WIN_NAMESIZE characters. This information could be the handle needed for a program to perform some sort of special window management function not provided by the default window manager.

7.5.3. Driving a TTY Subwindow

It is possible to drive the terminal emulator directly. There are procedures which take both input and output.

```
int  ttysw_output(ttysw, addr, len)
     caddr_t      ttysw;
     char         *addr;
     int          len;
```

`ttysw_output` runs the character sequence in `addr` that is `len` characters long through the terminal emulator of `ttysw`. The number of characters accepted is returned.

```
int  ttysw_input(ttysw, addr, len)
     caddr_t      ttysw;
     char         *addr;
     int          len;
```

`ttysw_input` appends the character sequence in `addr` that is `len` characters long onto the input queue of the terminal emulator of `ttysw`. The number of characters accepted is returned.

7.5.4. Extending a TTY Subwindow

Client programs may extend the tty subwindow's interpretation of ANSI escape sequences.

The `ttysubwindow` structure in the header file `<suntool/ttysw_impl.h>` contains a pointer to a function, `ttysw_escapeop`, that handles ANSI X3.64 escape sequences coming in to a tty subwindow. X3.64 escape codes start with \E[# and terminate with an alphabetic character.

You can extend escape code interpretation by replacing the pointer to the `ttysw_escapeop` function with a pointer to a function you provide, according to the following instructions.

The procedure you provide to handle X3.64 escape sequences must have the following calling sequence:

```
int  ttysw_esc_extend(ttysw, c, ac, av)
     struct       ttysubwindow *ttysw;
     char         c;
     int          ac;
     int          *av;
```

The procedure itself may have any name you wish. `ttysw` is the terminal emulator handle. `c` is the character that terminates the escape sequence. `av` is a pointer to an array of integers which are the arguments to the escape sequence. `ac` is the number of integer parameters to the escape sequence.

A return value of TTY_DONE means that the routine handled the sequence. A return value of TTY_OK means that the routine didn't handle the sequence.

If you provide your own routine, please note the following:

> In order to replace `ttysw_escapeop` with your routine, declare a variable (for example, `saveptr`) and assign `ttysw_escapeop` to it. Then assign a pointer that addresses your new routine to `ttysw_escapeop`.

> If your routine cannot process the input escape sequence, it should call `ttysw_escapeop` to handle the sequence in question. This can be done using the pointer previously stored in the variable `saveptr`. `ttysw_escapeop` will return a value that can be delivered in turn to the caller of the new routine.

You can extend the interpretation of ANSI string escape codes in an analogous manner by replacing a pointer to `ttysw_stringop`. ANSI string escape codes begin as follows:

\EP — ANSI Device Control String.

\E] — ANSI Operating System Command.

\E^ — ANSI Privacy Message.

\E_ — ANSI Application Program Command.

ANSI string escape codes terminate with \E\.

The procedure you provide to handle string escape codes must have the following calling sequence:

```
int  ttysw_esc_str_extend(ttysw, strtype, c)
    struct      ttysubwindow *ttysw;
    char        strtype;
    char        c;
```

The procedure itself may have any name you wish. `ttysw` is the terminal emulator handle. `c` is the next character in the string. `strtype` is the string type character (one of `P, ], ^` or `_`).

Unlike `ttysw_esc_extend`, the terminal emulator will call `ttysw_esc_str_extend` for each character in the escape string. A NULL `c` argument indicates the end of the escape string.

A return value of TTY_DONE means that the routine handled the character. A return value of TTY_OK means that the routine didn't handle the character.

As with `ttysw_esc_extend`, your routine should store a pointer to `ttysw_stringop`. If your routine cannot process the input character or string type, it should call `ttysw_stringop` to handle the character in question. Then your routine can deliver the value returned by `ttysw_stringop` to the caller. See above for specific instructions.

Chapter 8

The Panel Subwindow Package

8.1. Introduction

This chapter discusses the panel subwindow package, which supersedes the option subwindow package. We strongly urge you to use this new package instead of the option subwindow package in all your programs. The option subwindow is included in this release (see Appendix F), but will not be included in future releases of Sunwindows. For an example of how to convert a program from the option subwindow package to the panel package, see Appendix G.

This chapter assumes you are familiar with such concepts as ***tools***, ***subwindows***, ***menus***, ***pixrects***, and so on. If you need background information about these and other basics read the ***Sunwindows Tutorial***.

A note on how to use this chapter: The first three sections provide a non-technical introduction to panels and what they are good for. Sections 8.4 through 8.6 introduce the basic concepts and routines needed to create simple panels. Section 8.7 gives a detailed description of the structure and behavior of the different types of panel items; it will prove useful as you begin to create more elaborate panels. As you continue to use panels, you will probably want to refer often to Section 8.14, which provides a comprehensive summary of the many "attributes" at your disposal to manipulate panels.

Programs using panels must include the header file `<suntool/panel.h>`.

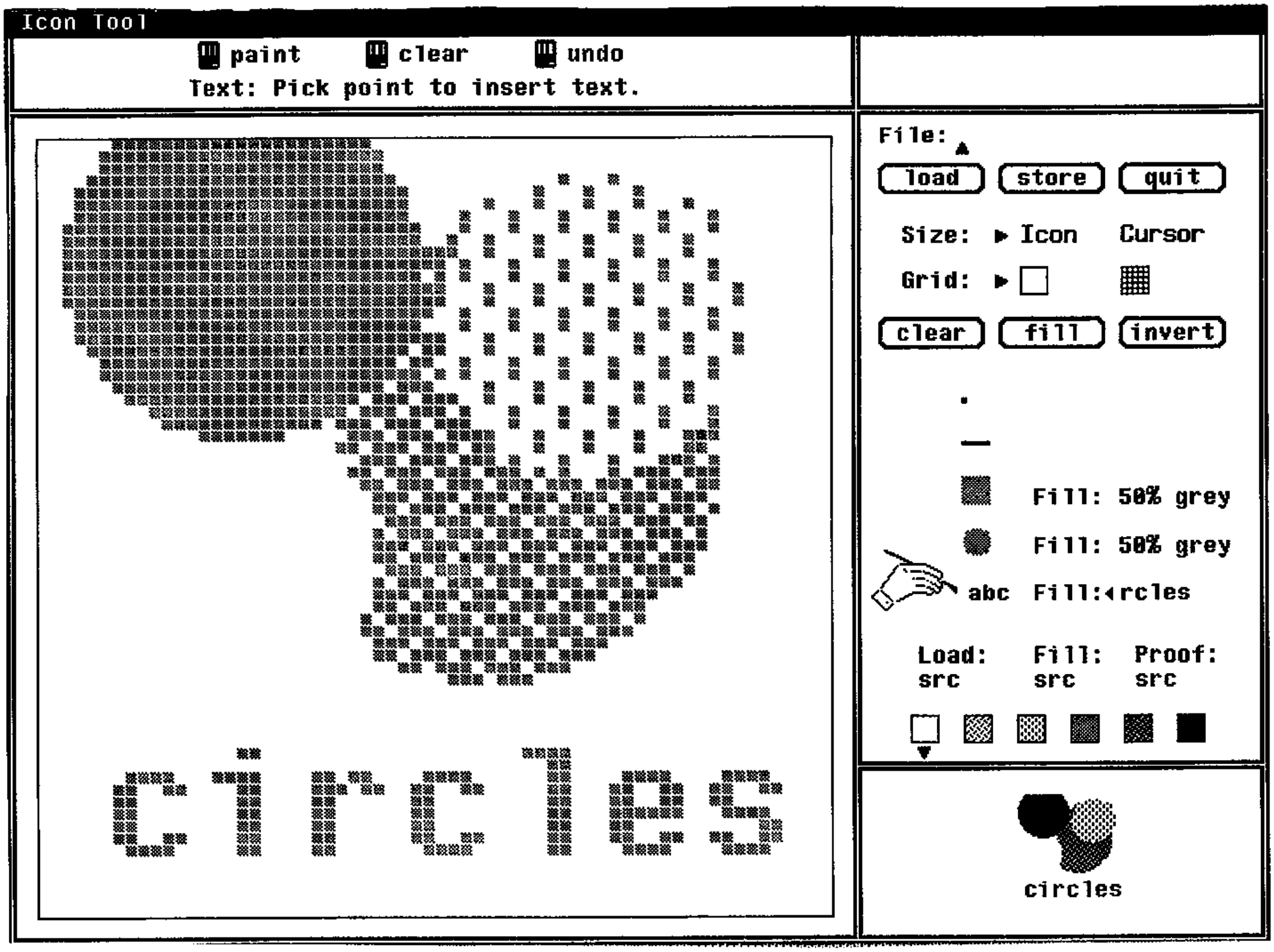
8.2. Definition and Uses of Panels

The word ***panel*** refers to a subwindow containing ***items*** through which the user interacts with a program. Several different types of items are available, including buttons, messages, text, choices, and analog sliders. Panels are quite flexible; you can use them to model a variety of things, such as

- a form, consisting mainly of text items
- a single button representing a command
- a single switch representing the current mode of a program
- a row of pull-down menus
- a complex control panel containing items and menus of many types
- a message window containing status or error messages

See the following figure for a picture of `icontool`, a tool that makes extensive use of panels.

Figure 8-1: `icontool` — a tool that uses panels



Text: Pick point to insert text.

File:

Size: ▶ Icon Cursor

Grid: ▶ ☐ ▦

Fill: 50% grey

Fill: 50% grey

abc Fill:◂rcles

Load: src

Fill: src

Proof: src

Note that nearly all of the windows in `icontool` are panels; the exceptions are the large subwindow at bottom left, and the small subwindow at bottom right.

The characteristics of both panels and items are specified by means of ***attributes***, which are usually set when you create the panel or item. You may also retrieve and modify attributes after creation time.

8.3. Panel Item Types and Their Uses

There are currently six basic types of items: messages, buttons, choices, toggles, text and sliders. Items are made up of one or more displayable components. One component shared by all item types is the ***label***. An item label is either a string or a graphic image (i.e., a pointer to a pixrect).

Button, choice, toggle, and text items also have a menu component. Thus the user may interact with most items in either of two ways: by selecting the item directly (with the left mouse button) or by selecting from the item's menu (with the right mouse button). Item menus are described more fully in ***Description of Each Item Type***.

Each item type is introduced briefly below.

Message Items

The only visible component of a Message item is a label, which may be an image or a string in a specified font. Message items are useful for annotations of all kinds, e.g.,

- titles
- comments
- descriptions
- pictures
- static messages
- dynamic messages, such as error messages.

Message items are selectable, and you may specify a ***notify procedure*** to be called when the item is selected.

Button Items

Button items allow the user of a program to initiate commands. Buttons, like Message items, have a label, are selectable, and have a notify procedure. Button items differ from Message items in that they have visible feedback for tentative and actual selection (see Section 8.8.2 below).

Choice Items

Choice items allow the user to select one choice from a list. The displayed form of choice items can vary radically, depending on their attribute settings. Some of the ways choice items can be presented are:

- a (horizontal or vertical) list of choices, with all choices visible and the current choice indicated by a mark (such as a checkmark).
- a (horizontal or vertical) list of choices, with all choices visible and the current choice in reverse-video.
- a "cycled item", or list of choices with only the current choice visible. Selecting the item causes the next choice in the list to be selected and displayed.

- a binary switch, modelled after a light switch, with an arrow or other shape pointing to one of two strings or images representing the two states.
- a knob, which has a pointer of some sort which turns to indicate one of several choices.
- a pull down menu, with only the label visible until the menu button is pressed.

Behind the flexibility of presentation lies a uniform structure consisting of a label, a list of choices, and, optionally, a corresponding lists of "on-marks" and "off-marks" which indicate which choice is currently selected.

Toggle Items

In appearance and structure, toggle items are identical to choice items. The difference lies in the behavior of the two types of items when selected. In a choice item exactly one element of the list is selected, or ***current***, at a time. A toggle item, on the other hand, is best understood as a list of elements which behave as toggles: each choice may be either on or off, independently of the other choices. Selecting a choice causes it to change state. There is no concept of a single current choice; at any given time all, some, or none of the choices may be selected.

Text Items

Text items are basically type-in fields with optional labels and menus. Notification behavior for text items is more flexible than for the other item types. The notification level can be set so that the notify procedure will be called on each character typed in, only on specified characters, or not at all. This allows a client such as a forms-entry program to process input on a per character, per field, or per screen basis.

Slider Items

Slider items allow the graphical representation and selection of a value within a range. They are appropriate for situations where it is desired to make fine adjustments over a continuous range of values. A familiar model would be a horizontal volume control lever on a stereo panel.

8.4. A Sample Panel

Here is an example of a simple control panel for an imaginary tool which lets you list a directory:

Directory Listing Tool

Directory: List Quit

Below is the routine which creates this panel:

```
#include <suntool/panel.h>

int list_proc(), quit_proc();

make_control_panel()
{
   struct toolsw       *panel_subwindow;
   Panel                panel;
   Panel_item           heading_item, directory_item,
                        list_item, quit_item;

   panel_subwindow = panel_create(tool, 0);
   panel           = (Panel)panel_subwindow->ts_data;

   heading_item =   panel_create_item(panel, PANEL_MESSAGE,
                    PANEL_ITEM_X, PANEL_CU(20),
                    PANEL_ITEM_Y, PANEL_CU(1),
                    PANEL_LABEL_STRING, "Directory Listing Tool",
                    0);

   directory_item = panel_create_item(panel, PANEL_TEXT,
                                      PANEL_ITEM_X, PANEL_CU(5),
                                      PANEL_ITEM_Y, PANEL_CU(3),
                                      PANEL_LABEL_STRING, "Directory:",
                                      PANEL_VALUE_DISPLAY_LENGTH, 20,
                                      0);

   list_item = panel_create_item(panel, PANEL_BUTTON,
                                 PANEL_LABEL_STRING,    "List",
                                 PANEL_NOTIFY_PROC,     list_proc,
                                 0);

   quit_item = panel_create_item(panel, PANEL_BUTTON,
                                 PANEL_LABEL_STRING,    "Quit",
                                 PANEL_NOTIFY_PROC,     quit_proc,
                                 0);

   panel_fit_height(panel,0);    /* adjust panel height to fit the items */
}

list_proc(panel, list_item)
Panel      panel;
Panel_item list_item;
{
 body of procedure...
}
quit_proc(panel, quit_item)
Panel      panel;
Panel_item quit_item;
{
 body of procedure...
}
```

The items are positioned in the panel with the PANEL_ITEM_X and PANEL_ITEM_Y arguments. PANEL_CU is a macro meaning "interpret this number in character units"; so the heading, for example, appears on row 1, column 20. (The first row is row zero, the first column is column zero.) Items which are not explicitly positioned are placed immediately after the lowest, rightmost item; so `list_item` and `quit_item` appear just after the 20-character-long type-in area for `directory_item`.

When the cursor is positioned within a panel, the left mouse button is used to select items. So, to list the directory typed into the **Directory:** field, the user would position the mouse over the word *List* and click the left button.

`list_proc()` and `quit_proc()` are the routines, specified by the client, which will be called by the panel package when the user selects the corresponding item. It is up to these routines to take the appropriate action — in this case list the directory or quit.

8.5. Attributes and Attribute-Lists

There is a large set of attributes applying to panels and to the various item types. For a given call to create or modify a panel or item, only a subset of all the attributes will be of interest. So that only the relevant attributes need be mentioned, the panel routines make use of variable-length ***attribute lists***. An attribute list consists of attribute/value pairs, separated by commas, and ending with a 0. For example, an item at pixel location (5,10), with a label of "Load File " would be described with the attribute list:

```
PANEL_ITEM_X,    5,
PANEL_ITEM_Y,    10,
PANEL_LABEL_STRING,      "Load File: ",
0
```

The order in which different attributes are mentioned is irrelevant. If the same attribute appears more than once in an attribute list, the last value mentioned is the one which takes effect.

Attributes take values of a particular type; thus they may be referred to as ***string-valued***, ***integer-valued***, etc. Values for each attribute also have a particular ***cardinality***, that is, the values are single, a null-terminated list, or, in some cases, a pair. Some examples of attributes of different types are given in the table below.

Table 8-1: Some Sample Panel Attributes

Type	Attribute/Value	Explanation
integer	`PANEL_VALUE, 5`	item's value is 5
boolean	`PANEL_SHOW_ITEM, FALSE`	item is not displayed
string	`PANEL_LABEL_STRING, "Name: "`	item's label is "Name: "
list of strings	`PANEL_CHOICE_STRINGS, "A", "B", "C", 0`	item has 3 choices
image	`PANEL_LABEL_IMAGE, &pixrect1`	item's label is pixrect1
list of images	`PANEL_CHOICE_IMAGES, &pixrect2, &pixrect3, 0`	item has 2 choices
ptr to function	`PANEL_NOTIFY_PROC, f`	"f()" called when item selected

A basic rule to bear in mind when using the panel package is that in setting the value of an item's attribute, the effect will be the same whether the operation is done at creation time or afterwards. In other words, any attribute which you can specify in the `panel_create()` call can also be specified in later calls to `panel_set()`. Thus items exhibit ***dynamic*** behavior: they can be created in one position and moved later, their labels, fonts, or their entire appearance on the screen may be changed, etc.

A special attribute, PANEL_CLIENT_DATA, is provided for the client to use as desired. Some example uses:

- To associate a unique identifier with each item. This is convenient in the case where you have many items, or where you are creating and destroying items dynamically. If you need to pick one item out of all the items, you can store an identifier (or a class) with it via PANEL_CLIENT_DATA, and then query the item directly to find out its identifier or class.
- To associate a pointer to a private structure with an item. One application of this would be to link several items together into a list which is completely under client control.

Throughout this chapter, specific attributes will be mentioned as they arise in the course of the discussion. All of the attributes, along with their types, cardinalities, default values and applications, are summarized in the tables at the end of this chapter.

The discussion which follows will make frequent reference to six data types defined by the panel package; these are listed in the following table.

Table 8-2: Frequently Used Panel Data Types

`Panel`	A pointer to the structure which describes a panel[5].
`Panel_item`	A pointer to the structure which describes a panel item.
`Panel_item_type`	The type of an item, specified when the item is created.
`Panel_attribute`	A constant which specifies a particular attribute.
`Panel_attribute_value`	Type used for retrieving attribute values.
`Panel_setting`	Type returned by `panel_text_notify()`; type of `repaint` argument to `panel_paint()`.

8.6. Creating Panels

Create the subwindow for a panel with the routine:

```
struct toolsw *panel_create(tool, attributes)
        struct tool             *tool;
        <attribute-list>     attributes;
```

---

[5] From the point of view of client programs, `Panel` and `Panel_item` are *opaque* data types, meaning that clients of the panel package cannot "see through" them to the actual data structure. Values of type `Panel` and `Panel_item` are simply used as ***handles*** by which the corresponding object is referenced. One of these opaque handles is returned when you create a panel or item. Later, when you wish to refer to a particular panel or item, you pass its handle to the appropriate panel package routine. (The data types `Panel` and `Panel_item` are actually `typedef`'ed to `char *`.)

`Panel` and `Panel_item` are actually 32-bit pointers, so they may be passed as parameters without efficiency penalty.

In order to manipulate a panel and create items within it, you must have the panel's handle, which is a variable of type Panel. To obtain a `Panel`, first create the panel subwindow, then use the `ts_data` field of that subwindow, as follows:

```
struct toolsw   *panel_subwindow;
       Panel            panel;

panel_subwindow = panel_create(tool, 0);
panel           = (Panel)panel_subwindow->ts_data;
```

A panel, once created, is not linked into the display tree until the call to `tool_install()` is made. The usual usage is to first create the panel, then create the items within the panel, and finally install the tool.

The above call to `panel_create()` is the simplest case, in which no attributes are given. In particular, since the height and width of the panel are not set, the panel will extend to the bottom and right edges of the tool. You can control the dimensions of the panel in either of the two ways described below.

Often you want the panel to be just high enough to encompass all of the items within it. To enable you to achieve this without having to compute the desired dimension, the panel package provides the following mechanism. After creating all of the items, ***and before creating any other subwindows*** in the tool, set the height of the panel to the constant PANEL_FIT_ITEMS, e.g.:

```
panel_set(panel, PANEL_HEIGHT, PANEL_FIT_ITEMS, 0);
```

This causes the panel package to compute the lowest point occupied by any of the panel's items and set the panel height to that point plus a bottom margin, which defaults to four pixels. If you want a different bottom margin, simply include the adjustment by adding or subtracting to PANEL_FIT_ITEMS. So

```
panel_set(panel, PANEL_HEIGHT, PANEL_FIT_ITEMS + 16, 0);
```

produces a bottom margin of 20 pixels, while

```
panel_set(panel, PANEL_HEIGHT, PANEL_FIT_ITEMS - 4, 0);
```

will leave no bottom margin.

The above discussion applies to setting the width of the panel as well. Thus, the call

```
panel_set(panel,
          PANEL_HEIGHT, PANEL_FIT_ITEMS + 6,
          PANEL_WIDTH,  PANEL_FIT_ITEMS,
          0);
```

will yield a panel with a bottom margin of 10 and a right margin of 4.

To ease the syntax for the simple case, two macros are provided:

```
panel_fit_height(panel);
panel_fit_width(panel);
```

These macros extend the panel 4 pixels below the lowest item, and extend the panel 4 pixels to the right of the rightmost item, respectively.

If the default 4 pixel margin is not what you want, you can use `panel_set()` to get the exact margin you want. For example,

```
panel_set(panel, PANEL_HEIGHT, PANEL_FIT_ITEMS + 6, 0)
```

will yield a bottom margin of 10 pixels.

Note that the automatic sizing described above must be done ***after*** creating all the panel's items, and ***before*** creating any other subwindows below or to the right of the panel.

The panel's height and width can also be set explicitly at creation time, as in

```
panel_subwindow = panel_create(tool,
                                PANEL_HEIGHT, PANEL_CU(10),
                                PANEL_WIDTH,  PANEL_CU(20),
                                0);
panel           = (Panel)panel_subwindow->ts_data;
```

which creates a panel 10 high by 20 characters wide.

After creating a panel, you can retrieve its attributes by calling `panel_get()`, and modify certain of its attributes (e.g., change its caret from blinking to non-blinking) by calling `panel_set()`.

The attributes applicable to the panel as a whole, as opposed to the individual items within the panel, are summarized in the ***Panel Attributes*** table (Section 8.14).

8.7. Creating and Positioning Items

8.7.1. Creating Items

Use the routine below to create panel items:

```
Panel_item panel_create_item(panel, item_type, attributes)
        Panel                   panel;
        Panel_item_type         item_type;
        <attribute-list>        attributes;
```

Values for ***item_type*** must be one of PANEL_MESSAGE, PANEL_BUTTON, PANEL_CHOICE, PANEL_TOGGLE, PANEL_TEXT, or PANEL_SLIDER.

Many attributes, such as those relating to item positioning, apply across all of the item types; these are called ***generic*** attributes. A comprehensive summary of these generic attributes is given in the table *Generic Item Attributes* in Section 8.14.

To give just a single illustration, the following call creates a message which is initially "hidden" (not displayed on the screen):

```
delete_msg_item = panel_create_item(panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING, "Warning: you are about to delete all files!",
    PANEL_SHOW_ITEM,     FALSE,
    0);
```

The above message could be displayed later with the call:

```
panel_set(delete_msg_item, PANEL_SHOW_ITEM, TRUE, 0);
```

8.7.2. Positioning Items Within a Panel

Explicit Item Positioning

The position of items within the panel may be specified explicitly by means of the attributes PANEL_ITEM_X and PANEL_ITEM_Y. PANEL_ITEM_X sets the left edge of the item's rectangle (the rectangle which encloses the item's label and value). PANEL_ITEM_Y sets the top edge of the item's rectangle.

All coordinate specification attributes interpret their values in pixel units. For simple panels and forms which do not make heavy use of images and have only one text font, it is usually more convenient to specify positions in character units — columns and rows rather than x's and y's. To this end a macro PANEL_CU() (for ***Character Units***) is provided, which interprets its argument as columns for X attributes or as rows for Y attributes, and converts the value to the corresponding number of pixels, based on the panel's font, as specified by PANEL_FONT. PANEL_CU() takes as its argument any expression yielding an integer. The use of PANEL_CU() as an operand in an expression is restricted to adding a pixel offset (e.g., PANEL_CU+ (5) + 2) as described below. Examples of legal and illegal usage are given in the table below:

Table 8-3: Example uses of the PANEL_CU() macro

Attribute/Value		Interpretation
PANEL_ITEM_X,	5	5 pixels from left
PANEL_ITEM_Y,	10	10 pixels from top
PANEL_ITEM_X,	PANEL_CU(5)	column 5
PANEL_ITEM_X,	PANEL_CU(-5)	column -5
PANEL_ITEM_X,	PANEL_CU(5+2)	column 7
PANEL_ITEM_X,	PANEL_CU(5)+2	two pixels to the right of column 5
PANEL_ITEM_X,	PANEL_CU(5)-1	one pixel to the left of column 5
PANEL_ITEM_Y,	PANEL_CU(10)	row 10
PANEL_ITEM_Y,	PANEL_CU(-10)	row -10
PANEL_ITEM_X,	PANEL_CU(10+2)	row 12
PANEL_ITEM_Y,	PANEL_CU(10)+2	two pixels down from row 10
PANEL_ITEM_Y,	PANEL_CU(10)-1	one pixel up from row 10
PANEL_ITEM_X,	PANEL_CU(10)+PANEL_CU(2)	*illegal*
PANEL_ITEM_X,	2*PANEL_CU(10)	*illegal*

Default Item Positioning

If you create an item without specifying its position, it is placed just to the right of the item on the "lowest row" of the panel, where lowest row is defined as the maximum y-coordinate (PANEL_ITEM_Y) of all the items. So in the absence of specific instructions, items will be placed within the panel in ***reading order*** as they are created: beginning four pixels in from the left and four pixels down from the top, items are located from left to right, top to bottom. If an item will not fit on a row, and more of the item would be visible on the next line, it will be placed on the next row. The number of pixels left blank between items on a line may be specified by PANEL_ITEM_X_GAP, which has a default value of 10. The number of pixels left blank between rows of items may be specified by PANEL_ITEM_Y_GAP, which has a default value of 5.

8.7.3. Laying Out Components Within an Item

You may also specify the layout of the various components within an item, by means of the attributes PANEL_LABEL_X, PANEL_LABEL_Y, PANEL_VALUE_X, PANEL_VALUE_Y, etc. If the components are not explicitly positioned, then the value is placed either eight pixels to the right of the label (if PANEL_LAYOUT has the value PANEL_HORIZONTAL) or four pixels below the label (if PANEL_LAYOUT has the value PANEL_VERTICAL). The default layout is horizontal (PANEL_LAYOUT is PANEL_HORIZONTAL).

8.8. Description of Each Item Type

This section describes each item type in more detail, covering the display options, selection feedback, notification behavior, value, and menu behavior.

Before getting into the different item types, it is worth mentioning that some attributes which apply to items may be set for all items in the panel by setting them when the panel is created. Such attributes include whether items have menus, whether item labels appear in bold, whether items are laid out vertically or horizontally, and whether items are automatically repainted when their attributes are modified (see the table ***Panel Attributes*** in Section 8.14 for a complete list). For example, the call

```
panel_sw = panel_create(panel,
                        PANEL_SHOW_MENU,  FALSE,
                        PANEL_LABEL_BOLD, TRUE,
                        PANEL_LAYOUT,     PANEL_VERTICAL,
                        PANEL_PAINT,      PANEL_NONE,
                        0);
```

overrides the defaults for all the attributes mentioned: any items subsequently created in that panel will not have menus, will have their labels printed in bold, will have their components laid out vertically, and will not be repainted automatically when their attributes are modified.

Note that the panel-wide item attributes mentioned above are only used to supply default values for items which are subsequently created — e.g., you cannot change all the item labels from the default bold font by first creating the items and then setting PANEL_LABEL_BOLD to FALSE for the panel.

A note on the usage of item menus. The panel package is designed to encourage the use of graphic images to convey information, and to allow you to present your interface in a form appropriate to your application. This will result in applications with different styles of panels. The menus are intended to balance this diversity with uniformity. Menus for all item types have a single, standard form and the user selects from them in the same way. In addition, the menus have a ***type symbol*** in their headings, indicating the item type. You can specify an item's menu type symbol via the attribute PANEL_TYPE_IMAGE. The default type symbols for each item type are given below:

- buttons — exclamation point

- choices — single check mark
- toggles with one choice — on/off switch
- toggles with more than one choice — double check mark
- text — pencil

For example, any choice item, regardless of the form it takes on the screen, will have a menu with the current choice checked. So the user, when faced with a new panel containing strange items whose interpretation is not clear, need only look at the menus to see a familiar interface.

Whether an item has a menu or not is controlled by the attribute PANEL_SHOW_MENU, which defaults to FALSE for all item types except choice and toggle items. You can enable or disable menus for all items in a panel by setting this attribute appropriately when you create the panel.

Now we discuss each of the item types in detail.

8.8.1. Messages

Message items are selectable, but there is no selection feedback. Messages also have no value visible on the panel, and no associated menu. A simple example is

```
message_to_mom = panel_create_item(panel, PANEL_MESSAGE,
                              PANEL_LABEL_STRING, "Hi Mom!", 0);
```

You may change the label for a message item (as for any type of item) via the PANEL_LABEL_STRING or PANEL_LABEL_IMAGE attribute, as in the call

```
panel_set(message_to_mom, PANEL_LABEL_STRING, "Bye Mom!", 0);
```

8.8.2. Buttons

Button items have a label and a menu, but no value.

Button Selection Behavior

When the left mouse button is pressed over a button item, the item's rectangle is inverted. When the mouse button is released over a button item, the item's rectangle is painted with a grey background, indicating that the item has been selected and the command is being executed. The grey background is cleared upon return from the notify procedure.

Button Notification Behavior

The procedure specified via the attribute PANEL_NOTIFY_PROC will be called when the item is selected. The notify procedure should declare both the the item and the event as arguments:

```
sample_notify_proc(item, event)
        Panel_item              item;
        struct inputevent       *event;
```

Note that if you need the panel in the notify procedure, you must get it from the item via the attribute PANEL_PARENT_PANEL using `panel_get`. For example,

```
Panel panel;
      panel = (Panel) panel_get(item, PANEL_PARENT_PANEL);
```

Button Menu Behavior

The menu for a button item has for its type symbol an exclamation point, which is meant to convey the idea of a command. The title of a button menu defaults to the item's label. Selection of a button through its menu is equivalent to selection by clicking directly on the label.

Button item menus do not appear by default; to obtain one for a particular item, set the attribute PANEL_SHOW_MENU for the item to TRUE.

Button Image Creation Utility

A routine is provided to create a standardized, button-like image from a string:

```
struct pixrect *panel_button_image(panel, string, width, font)
        Panel           panel;
        char            *string;
        int             width;
        struct pixfont  *font;
```

where `width` indicates the width of the button, in character units. The value returned is a pointer to a pixrect showing the string with a border drawn around it. The border is wide enough to contain the number of characters indicated by `width`.

If `width` is greater than the length of `string`, the string will be centered in the wider border; otherwise the border will be just wide enough to contain the entire string (i.e., the string will not be clipped). The font is given by `font` — if NULL, the font for `panel` is used.

For example, the call

```
panel_create_item(panel, PANEL_BUTTON,
                  PANEL_LABEL_IMAGE,
                  panel_button_image(panel, "Quit", 6, small_font),
                  0);
```

creates an item whose label is the string "Quit", in font `small_font`, centered in a border whose total width is six characters.

8.8.3. Choices

This section covers the general structure and behavior of choice items. For a complete list of the attributes applicable to choice items, see the table ***Choice Item Attributes*** in Section 8.14 below.

Choice items are the most flexible — and complex — item types. Besides the label, they are composed of:

- a list of either image or string choices (specified via the attributes PANEL_CHOICE_IMAGES or PANEL_CHOICE_STRINGS).
- a list of ***mark-images*** — images to be displayed when the corresponding choice is selected (PANEL_MARK_IMAGES). The default mark is a checkmark in a box.
- a list of ***nomark-images*** — images to be displayed when the corresponding choice is not selected (PANEL_NOMARK_IMAGES). The default nomark image is an empty box.

A single choice item may have up to 32 choices.

Displaying Choice Items

The attribute PANEL_DISPLAY_LEVEL determines which of an item's choices are actually displayed on the screen. The display level may be set to:

- PANEL_ALL, all choices are shown
- .PANEL_CURRENT, only the current choice is shown
- PANEL_NONE, no choices are shown. Since the only way of selecting a choice is through the menu, this becomes a label with an associated pop up menu.

If the display level is PANEL_CURRENT or PANEL_ALL, the choices are placed by default horizontally after the label. You can lay them out vertically below the label by setting PANEL_LAYOUT to PANEL_VERTICAL. If you want to place the choices or marks more precisely — in order to model a switch or some other special form — you can do so by setting the appropriate attribute, such as PANEL_CHOICE_XS, PANEL_CHOICE_YS, PANEL_MARK_XS, PANEL_MARK_YS, etc.

A few words about using the various lists in choice items. The list you give for PANEL_CHOICE_STRINGS (or PANEL_CHOICE_IMAGES) determines the item's choices. The parallel lists PANEL_CHOICE_FONTS, PANEL_MARK_IMAGES, PANEL_NOMARK_IMAGES, PANEL_MARK_XS, PANEL_MARK_YS, PANEL_CHOICE_XS, and PANEL_CHOICE_YS, are interpreted with respect to the list of choices. For example, the first font given for PANEL_CHOICE_FONTS will be used to print the first string given for PANEL_CHOICE_STRINGS, the second font will be used for the second string, and so on. Here's an example with several parallel lists:

```
size_item = panel_create_item(osw, PANEL_CHOICE,
                 PANEL_LABEL_X,          10,
                 PANEL_LABEL_Y,          4,
                 PANEL_LABEL_STRING,     "Size:",
                 PANEL_FEEDBACK,         PANEL_MARKED,
                 PANEL_CHOICE_STRINGS,   "Small", "Medium", "Large", 0,
                 PANEL_MARK_IMAGES,      &arrow_pixrect, 0,
                 PANEL_NOMARK_IMAGES,    0,
                 PANEL_CHOICE_XS,        20, 80, 140, 0,
                 PANEL_CHOICE_YS,        5, 0,
                 PANEL_MARK_XS,          10, 70, 130, 0,
                 PANEL_MARK_YS,          5, 0,
                 PANEL_NOTIFY_PROC,      size_proc,
                 0);
```

The above example illustrates the use of abbreviated lists. The item has three choices, "Small", "Medium" and "Large". Several of the parallel lists, however, have fewer than three elements — PANEL_MARK_IMAGES, PANEL_CHOICE_YS and PANEL_MARK_YS all have only one element. When any of the parallel lists are abbreviated in this way, the last element given will be used for the remainder of the choices. So, in the case of the attribute PANEL_CHOICE_YS above, "5,0" serves as an abbreviation for "5,5,5,0". All the choices and mark-images will appear at y coordinate 5, and all the choices will have the image `arrow_pixrect` as their mark-image.

Note: you cannot specify that a choice or mark-image appear at $x = 0$ or $y = 0$ by using the attributes PANEL_CHOICE_XS, PANEL_CHOICE_YS, PANEL_MARK_XS or PANEL_MARK_YS. Since these attributes take null-terminated lists as values, the zero would be interpreted as the terminator for the list. You may achieve the desired effect by setting the positions individually, with the attributes PANEL_CHOICE_X, PANEL_CHOICE_Y, PANEL_MARK_X, or PANEL_MARK_Y, which take as values the number of the choice or mark, followed by the desired position (note: the first choice is number 0).

Choice Selection Behavior

Feedback for choice items comes in two flavors — ***inverted,*** in which the current choice is shown in reverse video, and ***marked,*** in which the current choice is indicated by the presence of a distinguishing mark, such as a check-mark or arrow. The type of feedback is specified by setting PANEL_FEEDBACK to either PANEL_INVERTED or PANEL_MARKED. You may also disable feedback entirely, by setting PANEL_FEEDBACK to PANEL_NONE.

The default feedback is marked, unless the display level (see "Displaying Choice Items" above) is current, in which case the feedback is none.

There are three ways to make a selection from a choice item:

- by clicking on the desired choice directly, making it the new current choice;
- by clicking on the label, which causes the new current choice to be set to the one after the old current choice (or ***before*** if the shift key is pressed while selecting);
- through the associated menu.

Choice Notification Behavior

The procedure specified via the attribute PANEL_NOTIFY_PROC will be called when the item is selected. The notify procedure should declare the panel, the item and the value as formal parameters:

```
sample_notify_proc(item, value, event)
        Panel_item            item;
        int                   value;
        struct inputevent     *event;
```

Choice Value

The value passed to the notify procedure is the ordinal number corresponding to the current choice (the choice which the user has just selected). The first choice has ordinal number zero.

Choice Menu Behavior

Choice item menus may be used to represent menus of two types:

- a menu of commands to be executed, which gives no indication of which command was the last one executed (a ***simple*** menu).

- a menu of choices showing the currently selected choice (a ***checklist***).

Choice and Toggle items are the only item types for which a menu appears by default. To disable the menu for a particular item, set the attribute PANEL_SHOW_MENU for that item to FALSE. Set PANEL_SHOW_MENU_MARK to FALSE to obtain a simple menu, or TRUE to get a checklist.

Note that the number of menu choices, if set by MENU_CHOICE_STRINGS or MENU_CHOICE_IMAGES, must be equal to the number of choices for the item.

8.8.4. Toggles

Toggle items are identical in structure to choice items — they have a label and parallel lists of up to 32 choices, on-marks and off-marks. They differ from choice items in certain aspects of their display options, their selection behavior and the interpretation of their value. These differences are highlighted below.

Displaying Toggle Items

Toggle items may have a PANEL_DISPLAY_LEVEL of either PANEL_ALL — all choices visible, or PANEL_NONE — no choices visible. Since there is no notion of the ***current*** choice for a toggle item, a display level of PANEL_CURRENT is not allowed.

Toggle Selection Behavior

Toggle items, like choice items, may have either ***inverted*** or ***marked*** feedback (see ***Choice Selection Behavior*** above). Specify the feedback you want by setting PANEL_FEEDBACK to either PANEL_INVERTED or PANEL_MARKED (PANEL_NONE is not allowed).

Toggle items may be selected by clicking on the desired choice or through the menu. Selecting a choice causes that choice to toggle on or off, (change state); other choices are not affected.

If there is only one choice, it may be toggled by selecting the label; if there is more than one choice, selecting the label has no effect.

Toggle Notification Behavior

The parameters for the notify procedure are the same as for choice items except that the value passed is a bit mask (see the discussion under ***Toggle Value***, below) instead of an integer:

```
sample_notify_proc(item, value, event)
        Panel_item              item;
        unsigned int            value;
        struct inputevent       *event;
```

Toggle Value

The value passed to the notify procedure is a bit mask representing the state of the choices — if a bit is one, then the corresponding choice was ***on***, if a bit is zero, then the corresponding choice was ***off***. (The least significant bit is bit zero, which maps to choice zero.)

Take as an example an item called "format_item" and the following bit mask definitions:

```
#define LONG_CHOICE      0x00000001
#define SORTED_CHOICE    0x00000002
#define SHOW_ALL_CHOICE  0x00000004
```

The value might be used in the notify procedure as follows:

```
format_notify_proc(panel, format_item, value)
        Panel        panel;
        Panel_item   format_item;
        unsigned int value;
{
   if (value & LONG_CHOICE) {
      <perform some action>
   } else if (value & SORTED_CHOICE) {
      <perform some action>
   } else if (value & SHOW_ALL_CHOICE) {
      <perform some action>
   }
}
```

The value might also be retrieved outside of the notify procedure, as in:

```
unsigned int value;
value = panel_get_value(format_item);
if (value & LONG_CHOICE) {
  ....
}
```

Toggle Menu Behavior

The menu for a toggle item has one of two type symbols preceding its title. If the item has more than one choice, a double check-mark is shown, indicating that more than one choice may be selected at once. If the item has only one choice, then a two-state toggle is shown, indicating that the single choice may be either on or off.

The menu has as many lines as choices, and each line toggles when selected. In other words, the mark indicating "on" (PANEL_MENU_MARK_IMAGE) is alternated with the mark signifying "off" (PANEL_MENU_NOMARK_IMAGE) each time the user selects a given line.

To disable the menu, set PANEL_SHOW_MENU to FALSE.

8.8.5. Text

Displaying Text Items

The value component of a text item (the string which the user enters and edits) is drawn on the screen just after the label. The interpretation of "after" depends on the setting of PANEL_LAYOUT for the item: if PANEL_LAYOUT is PANEL_HORIZONTAL the value is placed to the right of the label, if PANEL_LAYOUT is PANEL_VERTICAL the value comes below the label.

Text Selection Behavior

A panel may have several text items, exactly one of which is ***current*** at any given time. The current text item is the one to which keyboard input is directed, and is indicated by a caret at the end of the item's value. (If PANEL_BLINK_CARET is TRUE, the caret will blink as long as the cursor is in the panel.) Selection of a text item (i.e. pressing and releasing the left mouse button anywhere within the item's rectangle) causes that item to become current. A text item also becomes current if it is restored — i.e. if PANEL_SHOW_ITEM is set to TRUE.

You can find out which text item has the caret, or give the caret to a specified text item, by means of the panel attribute PANEL_CARET_ITEM. The call

```
Panel_item name_item;
panel_set(panel, PANEL_CARET_ITEM, name_item, 0);
```

moves the caret to ***name_item***, while

```
caret_item = (Panel_item)panel_get(panel, PANEL_CARET_ITEM, 0);
```

sets the variable ***caret_item*** to the current item.

Text Notification Behavior

If a procedure is specified via the attribute PANEL_NOTIFY_PROC, it will be called at the appropriate time, as determined by the setting of PANEL_NOTIFY_LEVEL, discussed below. Text notify procedures receive, in addition to the usual panel and item handles, the event containing the input code:

```
sample_notify_proc(item, event)
        Panel_item              item;
        struct inputevent       *event;
```

The input character is referenced by `event->ie_code`.

If you do not specify your own notify procedure, a default procedure will be called:

```
Panel_setting panel_text_notify(item, event);
        Panel_item              item;
        struct inputevent       *event;
```

This procedure causes the caret to move to the next text item on carriage-return or tab, the previous text item on <SHIFT>-carriage-return or <SHIFT>-tab, printable characters to be inserted, and all other characters to be discarded.

You can tailor the notification behavior of each text item to support a variety of interface styles. On one extreme, you may want to process each character as the user types it in. For a different application you may not care about the values as they are typed in, and only want to look at them in response to some other button (e.g., only look at a filename field when the user presses the "Load" button).

The notification behavior of a text item is controlled by the attribute PANEL_NOTIFY_LEVEL. The following table describes its possible settings:

Table 8-4: Notification behavior

Notification Level	Causes Notify Proc to be Called
PANEL_NONE	never
PANEL_NON_PRINTABLE	on each non-printable input character
PANEL_SPECIFIED	if the input char is found in the string given for the attribute PANEL_NOTIFY_STRING.
PANEL_ALL	on each input character

For example, suppose you want to be notified only when the user types <ESC> or <CTRL>-C into an item. Create the item as follows:

```
name_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_STRING,     "Enter Name Here:",
        PANEL_NOTIFY_LEVEL,     PANEL_SPECIFIED,
        PANEL_NOTIFY_STRING,    "\033\03",
        PANEL_NOTIFY_PROC,      name_proc,
        0);
```

PANEL_NOTIFY_LEVEL defaults to PANEL_SPECIFIED, and PANEL_NOTIFY_STRING defaults to "\n\r\t" (i.e., notification on line-feed, carriage-return and tab).

If the user types a character which does not cause the item's notify procedure to be called, then the character, if printable, is appended to the item's value. Non-printable characters which do not cause notification are ignored, except for the user's editing characters, which are applied to the text item's value.

For input characters which cause notification, the value returned by the notify procedure determines what happens to the text item's value and what appears on the screen after the character is input. The following table shows the options for value returned by the notify procedure.

Table 8-5: Possible return values from notify procedures

Value Returned	Action Caused
`PANEL_INSERT`	character is inserted into item's value
`PANEL_NEXT`	caret moves to next text item
`PANEL_PREVIOUS`	caret moves to previous text item
`PANEL_NONE`	ignore the input character

If the notify procedure returns PANEL_INSERT, the character is appended to the value. If the character which is inserted is non-printable, nothing is shown on the screen. The editing characters (erase, erase_word, and erase-line) cause their intended actions to be performed and are ***not*** appended to the value, regardless of what the notify procedure returns.

Text Value

The value of a text item may be set (at creation time or later) with the attribute PANEL_VALUE, as in:

```
panel_create_item(panel, PANEL_TEXT, PANEL_VALUE,
                  "Edward G. Robinson", 0);
```

To retrieve the value of the text item `name_item` and store it into `buffer` (assuming that `name_item` has been created with a PANEL_VALUE_STORED_LENGTH of NAME_ITEM_MAX_LENGTH, so the buffer will not overflow):

```
Panel_item name_item;
char       buffer[NAME_ITEM_MAX_LENGTH];
strcpy(buffer, (char *)panel_get_value(name_item));
```

To set the value of `name_item`:

```
panel_set_value(name_item, "Millard Fillmore");
```

Text Menu Behavior

A menu may be associated with a text item by setting PANEL_SHOW_MENU to TRUE. The menu for a text item has a type symbol of a (very stubby!) pencil, suggesting that the item is a type-in field. The default title is the item's label.

The primary use of text item menus is to make any item-specific "accelerators", or characters which cause special behavior, visible to the user. An example of the use of accelerators may be found in the code below, which was taken from the cursor/icon editor "Icontool". The text item ***fname_item*** holds the name of the file being edited. In addition to typing printable characters, which are appended to the value of the item, the user can type <ESC> for filename completion, <CTRL>-L to load an image from the file, or <CTRL>-S to store an image to the file. The item is created with the call

```
#define ESC 27
#define CTRL_L 12
#define CTRL_S 19

fname_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,                  10,
        PANEL_LABEL_Y,                  6,
        PANEL_VALUE_DISPLAY_LENGTH,     18,
        PANEL_LABEL_FONT,               bold_font,
        PANEL_LABEL_STRING,             "File:",
        PANEL_NOTIFY_LEVEL,             PANEL_ALL,
        PANEL_VALUE_FONT,               bold_font,
        PANEL_NOTIFY_PROC,              fname_proc,
        PANEL_MENU_TITLE_STRING,        " Current File",
        PANEL_MENU_CHOICE_STRINGS,      "ESC    - Filename completion",
                                        "CTRL L - Load image from file",
                                        "CTRL S - Store image to file",
                                        0,
        PANEL_MENU_CHOICE_VALUES,   ESC, CTRL_L, CTRL_S, 0,
        0);
```

The last three attributes specify the menu. PANEL_MENU_TITLE_STRING specifies the menu's title. PANEL_MENU_CHOICE_STRINGS is a null-terminated array of strings to appear as the selectable lines of the menu. The value that the menu returns for each of its lines is specified with the attribute PANEL_MENU_CHOICE_VALUES. So if the menu line "CTRL-L - Load image from file" is selected, the menu will return the value CTRL_L. The value returned by the menu is passed directly to the text item, just as if it had been typed at the keyboard. In other words, the text item makes no distinction between menu-generated values and keyboard-generated characters.

Type-in Behavior

The user's erase, erase-word and kill characters function normally when typing into text items.

The number of characters of the text item's value which are displayable on the screen is set via the attribute PANEL_VALUE_DISPLAY_LENGTH. When characters are entered beyond this length, the value string is scrolled one character to the left, so that the most recently entered character is always visible. As the string scrolls to the left, the leftmost characters move out of the visible display area. The presence of these temporarily hidden characters is indicated by a small left-pointing triangle. The string is scrolled back to the right as excess characters are deleted, until the actual length becomes equal to the displayable length, and the entire string is visible.

The maximum number of characters which can be typed into a text item (independently of how many are displayable) is set via the attribute PANEL_VALUE_STORED_LENGTH. Attempting to enter a character beyond this limit causes the field to overflow, and the character is lost. The value string is blinked to indicate to the user that the text item is not accepting any more characters.

PANEL_VALUE_DISPLAY_LENGTH and PANEL_VALUE_STORED_LENGTH both default to 80. (Note that while the positioning attributes are measured in pixels, these two are measured in characters.)

Caret Manipulation

If a panel contains any text items, then there is a single caret which is associated with one of the text items at any point in time. The caret may be set to a particular text item by calling `panel_set`. The caret may also be rotated through the text items with the two routines:

```
panel_advance_caret(panel)
        Panel   panel;

panel_backup_caret(panel)
        Panel   panel;
```

Advancing past the last text item places the caret at the first text item; backing up past the first text item places the caret at the last text item.

8.8.6. Sliders

Displaying a Slider

A slider has four displayable components: the label, the current value, the slider bar, and the minimum and maximum allowable integral values (the range). When PANEL_SHOW_VALUE is TRUE, the current value is shown in brackets after the label (e.g. "[45]"). The font used to display the value is PANEL_VALUE_FONT.

The slider bar width in pixels is set with PANEL_SLIDER_WIDTH. If you want to specify the width in characters, use the "character units" macro PANEL_CU (Section 8.3). The minimum and maximum allowable values are set with PANEL_MIN_VALUE and PANEL_MAX_VALUE. The width of the slider bar corresponding to the current value is filled with grey. The slider bar is always displayed, unless the item is hidden (i.e., PANEL_SHOW_ITEM is FALSE). When PANEL_SHOW_RANGE is TRUE, the minimum value of the slider (PANEL_MIN_VALUE) is shown to the left of the slider bar and the maximum value (PANEL_MAX_VALUE) is shown to the right of the slider bar.

Selection Behavior

Only the slider bar of a slider may be selected. When the left mouse button is pressed within the slider bar or the mouse is dragged into the slider bar with the left mouse button pressed, the grey shaded area of the bar will advance or retreat to the position of the cursor. If the mouse is dragged left or right within the slider bar, the grey area will be updated appropriately. If the cursor is dragged outside of the slider bar, the original value of the slider (i.e., the value before the left button was pressed) will be restored.

Slider Notification Behavior

The notify procedure for a slider has the form:

```
sample_notify_proc(item, value, event)
        Panel_item            item;
        unsigned int          value;
        struct inputevent     *event;
```

where ***item*** is the item, ***value*** is the new value, and ***event*** is a pointer to the event that caused the notification.

The notification behavior of a slider is controlled by PANEL_NOTIFY_LEVEL. When PANEL_NOTIFY_LEVEL is set to PANEL_DONE, the notify procedure will be called only when the select button is released within the slider bar. When PANEL_NOTIFY_LEVEL is set to PANEL_ALL, the notify procedure will be called whenever the value of the slider is changed. This includes:

- when the select button is first pressed within or dragged into the slider bar,
- each time the mouse is dragged within the slider bar,

when the mouse is dragged outside the slider bar,

- when the select button is released.

Slider Value

The value of a slider is an integer in the range PANEL_MIN_VALUE to PANEL_MAX_VALUE. You can retrieve or set a slider's value with the attribute PANEL_VALUE.

Slider Menu

A slider has no associated menu.

Slider Examples

Below is an example illustrating a slider which might be used to control the brightness of a screen:

```
Panel           panel;
Panel_item      bright_slider;

bright_slider = panel_create_item(panel, PANEL_SLIDER,
                                    PANEL_LABEL_STRING, "Brightness: ",
                                    PANEL_VALUE,        75,
                                    PANEL_MIN_VALUE,    0,
                                    PANEL_MAX_VALUE,    100,
                                    PANEL_SLIDER_WIDTH, 400,
                                    PANEL_SHOW_RANGE,   TRUE,
                                    PANEL_SHOW_VALUE,   TRUE,
                                    0);
```

8.9. Modifying and Retrieving Attributes of Panels or Items

This section describes how to modify or retrieve the current values of attributes of panels or individual panel items which have already been created.

Several examples are given here; for a complete list of the attributes applying to panels and items, see Section 8.14.

Modifying Attributes

A single routine is used to set attributes of both panels and items:

```
panel_set(panel_object, attributes)
        <Panel_item or Panel>       panel_object;
        <attribute_list>            attributes;
```

For example, to move a panel's caret to the text item `name_item`:

```
Panel_item name_item;
panel_set(panel, PANEL_CARET_ITEM, name_item, 0);
```

To set the location of the item `error_message_item` to pixel coordinates (10, 50):

```
Panel_item error_message_item;
panel_set(error_message_item, PANEL_ITEM_X, 10,
                              PANEL_ITEM_Y, 50,
                              0);
```

A macro is provided to ease the syntax for the common operation of setting an item's value:

```
#define panel_set_value(item, value) panel_set(item, PANEL_VALUE, (value), 0)
```

For example, to set the value of the choice item `display_format_item` to the third (counting from zero) choice:

```
Panel_item display_format_item;
panel_set_value(display_format_item,2);
```

Note: The values for string-valued attributes are dynamically allocated when they are set (at creation time or later). If a previous value was present, it is freed after the new string is allocated. This is in contrast to the storage-allocation policy for retrieving attributes, described in the next section.

Retrieving Attributes

A single routine is used to retrieve attributes of both panels and items:

```
Panel_attribute_value panel_get(panel_object, attribute[, optional_arg])
        <Panel_item or Panel>       panel_object;
        Panel_attribute             attribute;
        Panel_attribute             optional_arg;
```

`Panel_get()` is used to retrieve attributes of all types, so the value returned must be coerced into the type appropriate to the attribute being retrieved. For example, to find out whether the caret in a panel is blinking or non-blinking:

```
int caret_is_blinking;
caret_is_blinking = (int)panel_get(panel, PANEL_BLINK_CARET);
```

To find out whether an item is currently being displayed on the screen:

```
int item_is_displayed;
item_is_displayed = (int)panel_get(item, PANEL_SHOW_ITEM);
```

The argument `optional_arg` is used for only a few item attributes. For example, to get the image for a choice item's third (counting from zero) choice:

```
struct pixrect *third_choice_image;
third_choice_image = (struct pixrect *)panel_get(panel, PANEL_CHOICE_IMAGE, 2
```

A macro is provided to ease the syntax for the common operation of retrieving an item's value:

```
#define panel_get_value(item) panel_get(item, PANEL_VALUE)
```

For example, to retrieve the current value of the text item ***comment_item***:

```
Panel_item comment_item;
char           *comments;
comments =     (char *)panel_get_value(comment_item);
```

Note: `panel_get()` and `panel_get_value()` do not dynamically allocate storage for the values they return. If the value returned is a pointer, it points directly into the panel's private data. In the example above, the string pointed to by ***comments*** may change — transparently to the program — as the user types into the panel. It is the programmer's responsibility to copy the information pointed to, if this kind of behavior is to be avoided.

The policy for setting attributes is different: the values for string-valued attributes are dynamically allocated (see the previous section).

8.10. Painting Panels and Individual Items

To repaint either an individual item or an entire panel, use the routine:

```
panel_paint(panel_object, paint_behavior)
        <Panel_item or Panel>  panel_object;
        Panel_setting                    paint_behavior;
```

`paint_behavior` should be either PANEL_CLEAR, which causes the rectangle occupied by the panel or item to be cleared prior to repainting, or PANEL_NO_CLEAR, which causes repainting to be done without any prior clearing.

It is not necessary to call `panel_paint()` explicitly to control the repainting of items. The "repaint behavior" of an item is controlled by the special attribute PANEL_PAINT. PANEL_PAINT has three possible values: PANEL_CLEAR, PANEL_NO_CLEAR, and PANEL_NONE. A value of PANEL_CLEAR means that the item will be automatically cleared and repainted after each call to `panel_set()`. A value of PANEL_NO_CLEAR means that the item will be automatically repainted (without any prior clearing) after each *panel_set()* call. A value of PANEL_NONE means that no automatic repainting will be done.

The default value for PANEL_PAINT is PANEL_CLEAR. Thus, in the default case, you do not need to call `panel_paint()` after calling `panel_set()`. You can set the repaint behavior for an item when the item is created, or for all items in the panel when the panel is created. The item's repaint behavior may ***not*** be reset after the item is created. However, you may temporarily ***override*** an item's repaint behavior on any call to `panel_set()` by giving a different setting for PANEL_PAINT. The following examples show two possible repaint policies:

Example 1:

```
item1 = panel_create_item(panel, PANEL_TEXT,
                    PANEL_LABEL_STRING,          "Enter Name:",
                    PANEL_VALUE_DISPLAY_LENGTH,  10,
                    PANEL_PAINT,                 PANEL_NONE,
                    0);

(install tool, etc...)

panel_set(item1, PANEL_ITEM_X, 10, PANEL_ITEM_Y, 50, 0);
panel_set(item1, PANEL_LABEL_IMAGE, &pixrect1, 0);
panel_set(item1, PANEL_VALUE_DISPLAY_LENGTH, 30, 0);
panel_paint(item1, PANEL_CLEAR);
```

Example 2:

```
item2 = panel_create_item(panel, PANEL_TEXT,
                    PANEL_LABEL_STRING,         "Enter Name:",
                    PANEL_VALUE_DISPLAY_LENGTH, 10,
                    0);
```

(install tool, etc...)

```
panel_set(item2,
          PANEL_ITEM_X, 10,
          PANEL_ITEM_Y, 50,
          PANEL_PAINT,  PANEL_NONE,
          0);
panel_set(item2,
          PANEL_LABEL_IMAGE, &pixrect1,
          PANEL_PAINT,  PANEL_NONE,
          0);
panel_set(item2,
          PANEL_VALUE_DISPLAY_LENGTH, 30,
          0);
```

The above two examples each produce the same effect. In the first example, the item's repaint behavior is set to PANEL_NONE at creation time, so it is not repainted automatically after the `panel_set()` calls, and no repainting occurs until the call to `panel_paint()`. In the second example, the item's repaint behavior is the default, PANEL_CLEAR. This is overridden in the first two `panel_set()` calls, so no repainting occurs. However, it is not overridden in the third call to `panel_set()`, so repainting occurs before that call returns.

As mentioned above, the repaint behavior for all items in a panel can be set when the panel is created, e.g.:

```
panel_create(tool, PANEL_PAINT, PANEL_NONE, 0);
```

All items created in the above panel will have a repaint behavior of PANEL_NONE.

8.11. Destroying Panels and Individual Items

A panel or individual item is destroyed (and its associated dynamic storage freed) with the routine:

```
panel_free(panel_object);
        <Panel_item or Panel>        panel_object;
```

8.12. Creating Reusable Attribute Lists

It may be desirable to create an attribute list which can then be passed to different routines. This can be done in either of two ways, either by creating the list explicitly, or by using the routine `panel_make_list()`.

To create an attribute list explicitly, a program must define a static array of strings, which is initialized (or later filled in with) the desired attribute/value pairs. Note that non-string values must be coerced to type `char *`:

```
static char *attributes[] = {
        PANEL_LABEL_STRING,             "Name: ",
        PANEL_VALUE,                    "Goofy ",
        PANEL_NOTIFY_PROC,              (char *)name_item_proc,
        0 }
```

To make an attribute list dynamically, use:

```
char **panel_make_list(attributes)
<attribute_list> attributes;
```

`panel_make_list()` allocates storage for the list it returns. It is up to the programmer to free this storage when no longer needed.

`Panel_make_list` can be used to support default attributes, e.g.:

```
int             text_proc(), name_proc();
Panel_item      name_item, address_item;
struct pixfont  *big_font, small_font;
char            *defaults;

defaults =      panel_make_list(PANEL_TEXT,
                                PANEL_SHOW_ITEM,     FALSE,
                                PANEL_LABEL_FONT,    big_font,
                                PANEL_VALUE_FONT,    small_font,
                                PANEL_NOTIFY_PROC,   text_proc,
                                0);

name_item =     panel_create_item(PANEL_TEXT
                                  PANEL_ATTRIBUTE_LIST, defaults,
                                  PANEL_NOTIFY_PROC,    name_proc,
                                  0);

address_item =  panel_create_item(
                                  PANEL_ATTRIBUTE_LIST, defaults,
                                  PANEL_SHOW_ITEM,      TRUE,
                                  PANEL_VALUE_FONT,     big_font,
                                  0);
```

The special attribute PANEL_ATTRIBUTE_LIST takes as its value an attribute list. In the above example, first an attribute_list called `defaults` is created. Then, by mentioning `defaults` first in the attribute lists for subsequent item creation calls, each item takes on those default attributes. Subsequent references to an attribute override the setting in `defaults` since the last value mentioned for an attribute is the one which takes effect.

8.13. Summary of Panel Functions

All functions, data types and attributes needed by programs using panels are found in the header file `<suntool/panel.h>`.

Data types:

`Panel`	a pointer to the structure which describes a panel.
`Panel_item`	a pointer to the structure which describes a panel item.
`Panel_item_type`	the type of an item, specified when the item is created.
`Panel_attribute`	a constant which specifies a particular attribute.
`Panel_attribute_value`	type used for retrieving attribute values.
`Panel_setting`	type returned by `panel_text_notify()`; type of `repaint` argument to `panel_paint()`.

Functions:

```
struct toolsw *panel_create(tool, attributes);
        struct tool             *tool;
        <attribute-list>     attributes;

Panel_item panel_create_item(panel, item_type, attributes);
        Panel                   panel;
        Panel_item_typei        item_type;
        <attribute-list>     attributes;

panel_free(panel_object);
        <Panel_item or Panel>       panel_object;

Panel_attribute_value panel_get(panel_object, attribute[, optional_arg])
        <Panel_item or Panel>       panel_object;
        Panel_attribute                     attribute;
        Panel_attribute                     optional_arg;

panel_set(panel_object, attributes)
        <Panel_item or Panel>       panel_object;
        <attribute_list>            attributes;

panel_set_value(item, value)
        Panel_item              item;
        Panel_attribute_value   value;
```

`panel_set_value()` is a macro, defined as:

```
#define panel_set_value(item, value) panel_set(item,
                                        PANEL_VALUE, value, 0)
```

```
Panel_attribute_value panel_get_value(item)
        Panel_item              item;
```

`panel_get_value()` is a macro, defined as:

```
#define panel_get_value(item) panel_get(item, PANEL_VALUE)

panel_paint(panel_object, paint_behavior)
        <Panel_item or Panel>       panel_object;
        Panel_setting                       paint_behavior;

panel_advance_caret(panel)
        Panel panel;

panel_backup_caret(panel)
        Panel panel;

struct pixrect *panel_button_image(panel, string, width, font)
        panel_handle    panel;
        char           *string;
        int             width;
        struct pixfont *font;

char    **panel_make_list(attributes)
        <attribute_list>    attributes;
```

8.14. Tables of Attributes

All of the panel package attributes are summarized in the tables below.

Panel Attributes covers those attributes which apply to the panel as a whole.

Generic Item Attributes cover those attributes that apply to panel items of all types.

Choice and Toggle Item Attributes, *Text Item Attributes*, and *Slider Item Attributes* cover attributes that apply to those specific types of items.

All the tables below use the following naming conventions in the interests of brevity:

- The prefix PANEL has been omitted from the attribute names.
- Under the *Characteristics* heading, the notation **...** after the name of an object means that a list of those objects, terminated by a zero, may appear in the actual code. For example, the PANEL_CHOICE_IMAGES attribute has an ***argument type*** of `struct pixrect * ...`, meaning that this attribute accepts a list of pointers to `pixrect`.
- The notation `get()` *returns* refers to the `panel_get()` function.

Table 8-6: Panel Attributes

Panel Attributes		
Name: PANEL_	*Description*	*Characteristics*
NAME	Subwindow name	*Argument Type:* `char *` `get()` *returns:* NULL *Default Value:* `""`
WIDTH	Subwindow width	*Argument Type:* `int` `get()` *returns:* NULL *Default Value:* `-1`
HEIGHT	Subwindow height	*Argument Type:* `int` `get()` *returns:* NULL *Default Value:* `-1`
FONT	Panel default font	*Argument Type:* `struct pixfont *` `get()` *returns:* `struct pixfont *` *Default Value:* `pw_pfsysopen()`
CARET_ITEM	Item with the caret	*Argument Type:* PANEL_ITEM `get()` *returns:* PANEL_ITEM *Default Value:* first text item

Panel Attributes

Name: PANEL_	*Description*	*Characteristics*
ITEM_X_GAP	Number of x-pixels between items	*Argument Type:* int get() *returns:* int *Default Value:* 10
ITEM_Y_GAP	Number of y-pixels between items	*Argument Type:* int get() *returns:* int *Default Value:* 5
BLINK_CARET	Static or blinking caret	*Argument Type:* int (TRUE/FALSE) get() *returns:* int *Default Value:* TRUE
TIMER_SECS	Number of timer seconds	*Argument Type:* int get() *returns:* int *Default Value:* 0
TIMER_USECS	Number of timer microseconds	*Argument Type:* int get() *returns:* int *Default Value:* 500000
TIMER_PROC	Function to call when timer expires	*Argument Type:* int (*) () get() *returns:* int (*) () *Default Value:* NULL
FIRST_ITEM	First item in the panel	*Argument Type:* N/A (Get only) get() *returns:* PANEL_ITEM *Default Value:* first panel item
SHOW_MENU	Show or don't show the panel menu. Sets the default for subsequent items created in panel.	*Argument Type:* int (TRUE/FALSE) get() *returns:* int *Default Value:* TRUE for choice items, FALSE for all other items
LABEL_BOLD	Bold or regular label string. Sets the default for subsequent items created in panel.	*Argument Type:* int (TRUE/FALSE) get() *returns:* int *Default Value:* FALSE
LAYOUT	Layout of value relative to label. Sets the default for subsequent items created in panel.	*Argument Type:* int (PANEL_HORIZONTA or PANEL_VERTICAL) get() *returns:* NULL *Default Value:* PANEL_HORIZONTAL

Table 8-7: Generic Item Attributes

Generic Item Attributes		
*Name:*PANEL_	*Description*	*Characteristics*
`ITEM_X`	Left edge of item rectangle. If unspecified and label or value positions are fixed, then set to minimum of `PANEL_LABEL_X` and `PANEL_VALUE_X`	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* after lowest, rightmost item
`ITEM_Y`	Top edge of item rectangle. If unspecified and label or value positions are fixed, then set to minimum of `PANEL_LABEL_Y` and `PANEL_VALUE_Y`	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* previous item's `PANEL_ITEM_Y`
`LABEL_X`	Left edge of label. If unspecified and value position is fixed, then set to left of `PANEL_VALUE_X` for horizontal layout, or at `PANEL_VALUE_X` for vertical layout	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* `PANEL_ITEM_X`
`LABEL_Y`	Top edge of label. If unspecified and value position is fixed, then set to `PANEL_VALUE_Y` for horizontal layout, or above `PANEL_VALUE_Y` for vertical layout	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* `PANEL_ITEM_Y`
`VALUE_X`	Left edge of value rectangle. If unspecified and label position is fixed, then set to right of `PANEL_LABEL_X` for horizontal layout, or at `PANEL_LABEL_X` for vertical layout	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* after the label
`VALUE_Y`	Top edge of value rectangle. If unspecified and label position is fixed, then set to `PANEL_LABEL_Y` for horizontal layout, or below `PANEL_LABEL_Y` for vertical layout	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* `PANEL_LABEL_Y`
`LABEL_STRING`	String for the label	*Argument Type:* `char *` `get()` *returns:* `char *` *Default Value:* `""`
`LABEL_IMAGE`	Graphic Image for the label	*Argument Type:* `struct pixrect *` `get()` *returns:* `struct pixrect *` *Default Value:* `NULL`
`LABEL_FONT`	Font for `PANEL_LABEL_STRING`	*Argument Type:* `struct pixfont *` `get()` *returns:* `struct pixfont *` *Default Value:* `PANEL_FONT`

Generic Item Attributes

*Name:*PANEL_	*Description*	*Characteristics*
LABEL_BOLD	Bold or regular label string	*Argument Type:* int (TRUE/FALSE) get() *returns:* int *Default Value:* TRUE
PAINT	Item's painting behavior for panel_set() calls	*Argument Type:* Panel_setting (PANEL_NONE, PANEL_CLEAR, PANEL_NO_CLEAR) get() *returns:* Panel_setting *Default value:* PANEL_CLEAR
NOTIFY_PROC	Function to call when item is selected	*Argument Type:* int (*)() get() *returns:* int (*)() *Default Value:* NULL
SHOW_MENU	Show or don't show the menu	*Argument Type:* int (TRUE/FALSE) get() *returns:* int *Default Value:* TRUE for choice items, FALSE for all other items
MENU_TITLE_STRING	String for the menu title	*Argument Type:* char * get() *returns:* NULL *Default Value:* PANEL_LABEL_STRING
MENU_TITLE_IMAGE	Graphic Image for the menu title	*Argument Type:* struct pixrect * get() *returns:* NULL *Default Value:* PANEL_LABEL_IMAGE
MENU_TITLE_FONT	Font for PANEL_MENU_TITLE_STRING	*Argument Type:* struct pixfont * get() *returns:* struct pixfont *Default Value:* PANEL_FONT
MENU_CHOICE_STRINGS	String for each menu choice. Default is PANEL_CHOICE_STRINGS for choice items, or NULL for other items.	*Argument Type:* char * ... get() *returns:* NULL *Default Value:* item dependent
MENU_CHOICE_IMAGES	Graphic image for each menu choice. Default is PANEL_CHOICE_IMAGES for choice items, PANEL_LABEL_IMAGE for button items, or NULL for other items.	*Argument Type:* struct pixrect * .. get() *returns:* NULL *Default Value:* item dependent
MENU_CHOICE_FONTS	Font for each menu choice string.	*Argument Type:* struct pixfont * get() *returns:* NULL *Default Value:* PANEL_FONT

Generic Item Attributes

Name:PANEL_	*Description*	*Characteristics*
`SHOW_ITEM`	Show or don't show the item	*Argument Type:* `int` (`TRUE/FALSE`) `get()` *returns:* `int` *Default Value:* `TRUE`
`LAYOUT`	Layout of value relative to label	*Argument Type:* `int` (`PANEL_HORIZONTA` or `PANEL_VERTICAL`) `get()` *returns:* `NULL` *Default Value:* `PANEL_HORIZONTAL`
`ITEM_RECT`	Enclosing rectangle for the item	*Argument Type:* N/A (Get only) `get()` *returns:* `struct rect *` *Default Value:* N/A
`NEXT_ITEM`	Next item in the panel	*Argument Type:* N/A (Get only) `get()` *returns:* `Panel_item` *Default Value:* N/A
`CLIENT_DATA`	Uninterpreted data for clients use	*Argument Type:* `caddr_t` `get()` *returns:* `caddr_t` *Default Value:* `NULL`
`PARENT_PANEL`	The panel in which an item is contained	*Argument Type:* N/A (get only) `get()` *returns:* `Panel` *Default Value:* N/A

Table 8-8: Choice and Toggle Item Attributes

Choice and Toggle Item Attributes		
*Name:*PANEL_	*Description*	*Characteristics*
CHOICE_STRINGS	String for each choice, maximum of 32 values	*Argument Type:* char * ... get() *returns:* NULL *Default Value:* "", 0
CHOICE_FONTS	Fonts to use for each choice string	*Argument Type:* struct pixfont * .. get() *returns:* NULL *Default Value:* PANEL_FONT, 0
CHOICE_IMAGES	Graphic image for each choice, maximum of 32 values	*Argument Type:* struct pixrect * .. get() *returns:* NULL *Default Value:* NULL, 0
CHOICE_STRING	String for a particular choice. Argument is the choice number, followed by the choice string.	*Argument Type:* int, char * get() *returns:* char * *Default Value:* N/A
CHOICE_IMAGE	Graphic image for a particular choice. Argument is the choice number, followed by the choice image.	*Argument Type:* int, struct pixrect get() *returns:* struct pixrect * *Default Value:* N/A
CHOICES_BOLD	Bold or regular choice strings	*Argument Type:* int (TRUE/FALSE) get() *returns:* NULL *Default Value:* FALSE
VALUE	If item is a choice, value is ordinal number of current choice (first choice is choice zero). If item is a toggle, value is a bitmask indicating currently selected choices (for example, bit 5 is 1 if 5th choice is selected).	*Argument Type:* int or unsigned get() *returns:* int or unsigned *Default Value:* 0
LAYOUT	Layout of the choices. If PANEL_HORIZONTAL, choices are laid out left to right after the label. If PANEL_VERTICAL, choices are laid out top to bottom under the label.	*Argument Type:* int (PANEL_HORIZONTA or PANEL_VERTICAL) get() *returns:* NULL *Default Value:* PANEL_HORIZONTAL
DISPLAY_LEVEL	How many choices to display. PANEL_NONE displays no choices, PANEL_CURRENT displays the selected choice (N/A for toggles), PANEL_ALL displays all the choices.	*Argument Type:* int (PANEL_NONE, PANEL_CURRENT, PANEL_ALL) get() *returns:* int *Default Value:* PANEL_ALL

Choice and Toggle Item Attributes

*Name:*PANEL_	*Description*	*Characteristics*
FEEDBACK	Feedback to give when a choice is selected. PANEL_NONE gives no feedback, PANEL_MARKED paints the on-mark for the choices, PANEL_INVERTED inverts the choice. If PANEL_DISPLAY_LEVEL is PANEL_CURRENT, the default feedback is PANEL_NONE, otherwise PANEL_MARKED.	*Argument Type:* int (PANEL_NONE, PANEL_MARKED, PANEL_INVERT) get () *returns:* int *Default Value:* depends on PANEL_DISPLAY_LEVEL
MARK_IMAGES	Graphic image to mark each choice with when selected.	*Argument Type:* struct pixrect * ... get () *returns:* NULL *Default Value:* Check in a box
NOMARK_IMAGES	Graphic image to mark each choice with when not selected.	*Argument Type:* struct pixrect * ... get () *returns:* NULL *Default Value:* empty box
MENU_MARK_IMAGE	Graphic image to mark each menu choice with when selected.	*Argument Type:* struct pixrect * get () *returns:* struct pixrect * *Default Value:* Check mark
MENU_NOMARK_IMAGE	Graphic image to mark each menu choice with when not selected.	*Argument Type:* struct pixrect * get () *returns:* struct pixrect * *Default Value:* NULL
SHOW_MENU_MARK	Show or don't show the menu mark for each selected menu choice.	*Argument Type:* int (TRUE/FALSE) get () *returns:* int *Default Value:* TRUE
CHOICE_OFFSET	Offset (in pixels) to place between choices.	*Argument Type:* int get () *returns:* int *Default Value:* NULL
CHOICE_XS	Left edge of each choice.	*Argument Type:* int ... get () *returns:* NULL *Default Value:* determined by PANEL_LAYOUT
CHOICE_YS	Top edge of each choice.	*Argument Type:* int ... get () *returns:* NULL *Default Value:* determined by PANEL_LAYOUT

Choice and Toggle Item Attributes

*Name:*PANEL_	*Description*	*Characteristics*
`MARK_XS`	Left edge of each choice mark.	*Argument Type:* `int ...` `get()` *returns:* `NULL` *Default Value:* determined by `PANEL_LAYOUT`
`MARK_YS`	Top edge of each choice mark.	*Argument Type:* `int ...` `get()` *returns:* `NULL` *Default Value:* determined by `PANEL_LAYOUT`
`CHOICE_X`	Left edge of specifed choice. Argument is choice number followed by desired left edge.	*Argument Type:* `int, int` `get()` *returns:* `int` *Default Value:* determined by `PANEL_LAYOUT`
`CHOICE_Y`	Top edge of specifed choice. Argument is choice number followed by desired top edge.	*Argument Type:* `int, int` `get()` *returns:* `int` *Default Value:* determined by `PANEL_LAYOUT`
`MARK_X`	Left edge of specifed choice mark. Argument is choice mark number followed by desired left edge.	*Argument Type:* `int, int` `get()` *returns:* `int` *Default Value:* determined by `PANEL_LAYOUT`
`MARK_Y`	Top edge of specifed choice mark. Argument is choice mark number followed by desired top edge.	*Argument Type:* `int, int` `get()` *returns:* `int` *Default Value:* determined by `PANEL_LAYOUT`

Table 8-9: Text Item Attributes

Text Item Attributes

*Name:*PANEL_	*Description*	*Characteristics*
VALUE	Initial or new string value for the item	*Argument Type:* `char *` `get()` *returns:* `char *` *Default Value:* `""`
VALUE_FONT	Font to use for the value string	*Argument Type:* `struct pixfont *` `get()` *returns:* `struct pixfont *` *Default Value:* PANEL_LABEL_FONT
NOTIFY_LEVEL	When to call the notify function. PANEL_NONE never notifies, PANEL_SPECIFIED notifies when a character specified in PANEL_NOTIFY_STRING is typed, PANEL_NON_PRINTABLE notifies when a non-printable character is typed, PANEL_ALL notifies on each typed character	*Argument Type:* `int` (PANEL_NONE, PANEL_SPECIFIED, PANEL_NON_PRINTABLE, PANEL_ALL) `get()` *returns:* `int` *Default Value:* PANEL_SPECIFIED
NOTIFY_STRING	String of characters which trigger notification when typed. Applicable only when PANEL_NOTIFY_LEVEL is PANEL_SPECIFIED.	*Argument Type:* `char *` `get()` *returns:* `char *` *Default Value:* `"\n\r\t"`
VALUE_STORED_LENGTH	Maximum number of characters to store in the value string. When the user attempts to enter more than PANEL_VALUE_STORED_LENGTH characters, the type-in string is blinked.	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* 80
VALUE_DISPLAY_LENGTH	Maximum number of characters to display in the panel. When the user enters more than PANEL_VALUE_DISPLAY_LENGTH characters, the type-in string is scrolled and clipped at the left.	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* 80
MASK_CHAR	Character used to mask type-in characters. Use the space character (' ') for no character echo (caret does not advance). Use the null character ('\0') to disable masking.	*Argument Type:* `char` `get()` *returns:* `char` *Default Value:* `'*'`

Table 8-10: Slider Item Attributes

Slider Item Attributes		
*Name:*PANEL_	*Description*	*Characteristics*
VALUE	Initial or new value for the item. The value is in the range PANEL_MIN_VALUE to PANEL_MAX_VALUE.	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* PANEL_MIN_VALUE
SHOW_VALUE	Show or don't show the integer value of the slider.	*Argument Type:* `int` (TRUE/FALSE) `get()` *returns:* `int` *Default Value:* TRUE
SHOW_RANGE	Show or don't show the minimum and maximum slider values.	*Argument Type:* `int` (TRUE/FALSE) `get()` *returns:* `int` *Default Value:* TRUE
VALUE_FONT	Font to use when displaying the value (PANEL_SHOW_VALUE, TRUE).	*Argument Type:* `struct pixfont *` `get()` *returns:* NULL *Default Value:* PANEL_LABEL_FONT
MAX_VALUE	Maximum value of the slider.	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* 100
MIN_VALUE	Minimum value of the slider.	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* 0
SLIDER_WIDTH	Width of the slider bar in pixels.	*Argument Type:* `int` `get()` *returns:* `int` *Default Value:* 100
NOTIFY_LEVEL	When to call the notify function. PANEL_DONE notifies when the select button is released, PANEL_ALL notifies continuously as the select button is dragged.	*Argument Type:* `int` (PANEL_DONE, PANEL_ALL) `get()` *returns:* `int` *Default Value:* PANEL_DONE

Chapter 9

Suntool: User Interface Utilities

This chapter describes the programming interface to a variety of separate packages that implement the user interface of the *suntool* layer. Because these utilities are not tied to the notions of *tool* and *subwindow* as described in previous chapters, they can be used as is, in another user interface system written on top of the *sunwindow* basic window system. For convenience, these utilities are associated directly with the *suntool* software layer.

9.1. 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;
    int         fs_cachedinputnext;
};
```

`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.

```
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.

`fullscreen_destroy` restores `fs`'s cached data:

```
fullscreen_destroy(fs)
    struct       fullscreen *fs;
```

It 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.

9.2. Icons

9.2.1. Icon Display Facility

This section describes an icon display facility. The icon structure is simply a stylized description of a useful class of images. Icons normally serve more to identify an object than display its contents. A typical use of an icon is to identify a currently unused but available tool. Another use might be a graphical depiction of an object, a document, database element, or resource for instance, that a user might want to point at with his mouse. The icon structure is declared in the file `<suntool/icon.h>`:

```
struct  icon {
    short       ic_width;
    short       ic_height;
    struct      pixrect *ic_background;
    struct      rect ic_gfxrect;
    struct      pixrect *ic_mpr;
    struct      rect ic_textrect;
    char        *ic_text;
    struct      pixfont *ic_font;
    int         ic_flags;
};

#define  ICON_BKGRDPAT  0x02
#define  ICON_BKGRDGRY  0x04
#define  ICON_BKGRDCLR  0x08
#define  ICON_BKGRDSET  0x10
```

`ic_width` and `ic_height` describe the full size of the icon. `ic_background` is an optional pattern with which to prepare the image background. `ic_gfxrect` and `ic_textrect` describe two subareas of the icon (icon coordinate system relative), which may overlap. `ic_mpr` addresses a memory pixrect as described in *Memory Pixrects*. `ic_mpr` has the graphic portion of the icon, `ic_text` points to a string, and `ic_font` a font in which to display it. The bits of `ic_flags` are defined above and indicate different ways to prepare the background of the image before adding `ic_mpr` and the text:

ICON_BKGRDPAT	use `ic_background`
ICON_BKGRDGRY	use a standard gray pattern used by the background window (this background is the memory pixrect `tool_bkgrd` defined in `<suntool/tool.h>`).

ICON_BKGRDCLR	clear (white out) the image
ICON_BKGRDSET	set (solid black) the image.

The function:

```
icon_display(icon, pixwin, x, y)
    struct      icon *icon;
    struct      pixwin *pixwin;
    int         x, y;
```

displays `icon` offset (`x, y`) from the origin of `pixwin`. The background is prepared according to `icon->ic_flags`. The graphic portion of the icon is displayed next, followed by the text; thus, if they overlap, the text will come out on top.

There are no strict restrictions on the size of an icon. However, the facility becomes relatively pointless if the icon is too large. Non-uniform icons have esthetic and placement defects. Therefore, a set of standard dimensions should be provided for any particular class of icons. The standards used by clients of tools are defined in `<suntool/tool.h>`. The names of the relevant constants defined there are:

```
TOOL_ICONWIDTH
TOOL_ICONHEIGHT
TOOL_ICONMARGIN
TOOL_ICONIMAGEWIDTH
TOOL_ICONIMAGEHEIGHT
TOOL_ICONIMAGELEFT
TOOL_ICONIMAGETOP
TOOL_ICONTEXTWIDTH
TOOL_ICONTEXTHEIGHT
TOOL_ICONTEXTLEFT
TOOL_ICONTEXTTOP
```

Please consult the header file for the current values of these constants.

The icon constants define the icon to be in an area of specified size, including a margin all around. The graphics and text regions are defined relative to the size of the icon and its margin; the graphics area covers the whole icon inside the margin, and the text covers the bottom 3/4 of that region. The TOOL_ICONIMAGE group of constants and the TOOL_ICONTEXT group of constants hold defaults for generating reasonable images when `ic_gfxrect` and `ic_textrect` respectively are initialized to them.

9.2.2. Making a Static Icon

After creating an icon with the ***icontool***, you can store a description of the image. This description is an ASCII file with two parts. The first part is a comment describing the image. The second part is a list of hexadecimal constants defining the actual pixel values of the image. Note that this file format enables a piece of code to incorporate an icon image at compile time. The code simply does a `#include` of the file containing the image description wherever it initializes the image array passed to `mpr_static`. The pixrect generated by `mpr_static` is then used in the initialization of the icon image structure. An example of such code can be found in the source for the ***icontool***.

A sample icon image description is the file `<images/template.icon>`, which is a template for all the image files in the cursor/icon library. Its contents follow:

```
/* 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. They contain information used 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. The `width`, `height`, and `depth` parameters are used in constructing the pixrect to hold the image, and should be the width, height and depth of the image. **`valid_bits_per_item`** specifies how many of the bits of each hexadecimal constant making up the image are valid, and uses the least significant bits. This sample file describes a cursor sized image on a white background; the image spells out the word **TEMPLATE** using two lines, and has a solid bar at the bottom of the image.

Default values for header parameters are:

Depth	1
Height	64
Width	64
Valid_bits_per_item	16

As an aid in making your own icon, use the following macro:

```
DEFINE_ICON_FROM_IMAGE(name, image)
```

This macro makes an icon that is ICON_DEFAULT_WIDTH bits wide by ICON_DEFAULT_HEIGHT bits high.

The DEFINE_ICON_FROM_IMAGE macro generates several static structures. The first argument to the macro is the name that will be given to the icon struct. The other argument is an array which contains `(ICON_DEFAULT_HEIGHT*ICON_DEFAULT_WIDTH/16)` shorts that are the bit pattern of the icon image. Typically this array will be declared as follows

```
static short icon_image[] = {
#include "file_generated_by_icontool"
};
```

Note that this macro does not provide access to all of the facilities that can be specified in an icon struct, but it is sufficient for most cases.

9.2.3. Dynamic Icon Loading

The routines used for run-time loading of icon images are declared in `<suntool/icon_load.h>`, along with the associated data structures and constants:

```
#define IL_ERRORMSG_SIZE                      256

typedef struct icon_header_object {
        int     depth,
                height,
                format_version,
                valid_bits_per_item,
                width;
        long    last_param_pos;
} icon_header_object;
typedef icon_header_object       *icon_header_handle;

extern int                        icon_load();
extern int                        icon_init_from_pr();
extern struct pixrect            *icon_load_mpr();
extern int                        icon_read_pr();
extern FILE                      *icon_open_header();
```

These routines all share the following convention about errors: if an error condition arises and the routine takes an `error_msg` parameter, the routine places an appropriate error message into the character array pointed to by `error_msg`, which must be at least IL_ERRORMSG_SIZE characters long.

```
int
icon_load(icon, from_file, error_msg);
        struct icon     *icon;
        char            *from_file, *error_msg;
```

`icon_load` allocates a pixrect for the icon image, loads it from the named file, then copies the file and the dimensions from the pixrect to initialize the icon. Information which is specified in the current pixrect (e.g., the font in which to display text associated with the icon) is set to default values. `from_file` names a file in the format described above; its contents are used to load the pixrect. `error_msg` should point to a buffer for an error message, as described above. If `icon_load` successfully initializes the icon it returns 0, otherwise it returns a non-zero value.

```
int
icon_init_from_pr(icon, pr)
        struct icon             *icon;
        struct pixrect   *pr;
```

`icon_init_from_pr` initializes the icon struct from the specified `pr`. The routine cannot ascertain certain information from a pixrect (e.g., the font for any icon text). As a consequence of this, certain fields in the icon struct are simply set to a default value by `icon_init_from_pr`. The return value of this routine is meaningless.

```
struct pixrect *
icon_load_mpr(from_file, error_msg)
        char            *from_file, *error_msg;
```

`icon_load_mpr` loads the icon image named by `from_file` into a dynamically allocated memory pixrect. If `icon_load_mpr` cannot access the file or the file is not in a valid format it returns NULL.

```
FILE *
icon_open_header(from_file, error_msg, info)
        char                    *from_file, *error_msg;
        icon_header_handle      info;
```

`icon_open_header` allows a client to preserve extra descriptive material when rewriting an icon image file. It is also the front-end routine used by `icon_load_mpr`, and thus `icon_load`. If `icon_open_header` cannot access the file or the file is not in a valid format it returns NULL.

`icon_open_header` fills in `info` from the file's header parameters, except in the case of `info->last_param_pos`. The routine fills it 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. 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 icon image file's header.

9.3. Pop-up Menus

A pop-up menu is a collection of items that a user can choose among by pointing the cursor at the desired item. It is quickly displayed in response to a button push, remains visible as long as the user holds the button down, and disappears as soon as the button is released.

Several menus can be presented at once. They appear to the user as a stack of images with the header of each menu visible, along with the items of the top menu in a vertical list. The user can bring other menus to the top by the same mechanism as choosing an item in the top menu.

A single menu is described by the following structure defined in `<suntool/menu.h>`:

```
struct  menu {
    int         m_imagetype;
    caddr_t     m_imagedata;
    int         m_itemcount;
    struct      menuitem *m_items;
    struct      menu *m_next;
    caddr_t     m_data;
};

#define  MENU_IMAGESTRING  0x0
#define  MENU_GRAPHIC      0x1
```

`m_imagetype` describes the data type of `m_imagedata`. `m_imagedata` is a pointer to the data displayed in the header of the menu. MENU_IMAGESTRING and MENU_GRAPHIC are the only currently defined data types. MENU_IMAGESTRING indicates that `m_imagedata` is a char *. MENU_GRAPHIC indicates that `m_imagedata` is a struct pixrect *. To use pixrects in the menu header, set the `m_imagetype` field to MENU_GRAPHIC, and the `m_imagedata` field to the desired pixrect.

`m_next` addresses the next menu in a stack; it is NULL if this menu is the last or only one in the stack.

`m_data` is private data utilized by the menu package while displaying menus. When you first create the menu you have to set the `m_data` field in the menu struct to zero. To do this, either explicitly set `m_data` to NULL or use `calloc()` instead of `malloc()` to allocate the storage for the menu structure.

`m_items` is an array of `menuitems` whose length is `m_itemcount`.

```
struct  menuitem {
        int         mi_imagetype;
        caddr_t     mi_imagedata;
        caddr_t     mi_data;
};
```

A `menuitem` consists of a display token/data pair. `mi_imagetype` describes the data type of `mi_imagedata`. `mi_imagedata` is a pointer to the data displayed in this item. MENU_IMAGESTRING and MENU_GRAPHIC are the only currently defined data types. MENU_IMAGESTRING indicates that `mi_imagedata` is a char *. MENU_GRAPHIC indicates that `mi_imagedata` is a struct pixrect *. To use pixrects in a menu item, set the `mi_imagetype` field to MENU_GRAPHIC, and the `mi_imagedata` field to the desired pixrect. `mi_data` is private to the creator of the item. Typically, it is an identifier that differentiates this item from others.

A client of the menu package constructs a stack of menus or several, for different situations by allocating menu structures and menuitem arrays and initializing all the fields in them. This involves hooking up all the data structures by setting the various pointers. Button-down on the right mouse button is the standard invocation. Then when a user action initiates menu processing, the client calls:

```
struct  menuitem *menu_display(menuptr, event, iowindowfd)
        struct      menu **menuptr;
        struct      inputevent *event;
        int         iowindowfd;
```

`menuptr` is the address of a menu pointer that points to the first or "top" menu structure in a menu stack. If the user causes the stack order to be rearranged, this indirection allows the menu package to leave the new top of the stack in `*menuptr` upon returning from `menu_display`. The menu package shuffles the stack's `m_next` values to rearrange the stack order. This enables the menu stack to be redisplayed in the order it was left in the last invocation.

`event` is the input event which provoked the menu. The location information in the event (`event->ie_locx`, `event->ie_locy`) controls where the menus will be displayed. `event->ie_code` is the event that is treated as the "menu button;" that is, the menu is displayed until this button goes up. The right mouse button is the usual menu button. The left mouse button is always used as the accelerator to bring rear menus forward. If it wasn't an explicit user action that provoked the call to `menu_display`, these three `event` fields must be loaded with the desired values beforehand.

`iowindowfd` is the file descriptor for the window that is displaying the menu. It is also the window that is read for user input. The `event` location values are relative to this window.

`menu_display` currently uses the mechanism described in ***Full Screen Access.*** `menu_display` temporarily modifies `iowindowfd`'s input mask to allow mouse motion and buttons to be placed on this window's input queue. All the menus in the stack are displayed, and there can only be one stack on the screen at a time. The font used for strings is that returned from `pw_pfsysopen`.

`menu_display` returns the `menuitem`, which was under the cursor when the user released the mouse button, or NULL if the cursor was not over an item.

9.4. Prompt Facility

A prompt facility is sometimes used with menus to tell the user to proceed from his current state. Prompting can also be done without menus. The definitions for the prompt facility are found in `<suntool/menu.h>`:

```
struct  prompt {
    struct      rect prt_rect;
    struct      pixfont *prt_font;
    char        *prt_text;
};

#define  PROMPT_FLEXIBLE  -1
```

`prt_rect` is the rectangle in which the text addressed by `prt_text` will be displayed using `prt_font`. Only printable characters and blanks are properly dealt with. Carriage returns, line feeds or tabs are not. If any of `prt_rect`'s fields are PROMPT_FLEXIBLE, that dimension is automatically chosen by the prompt mechanism to accommodate all the characters in `prt_text`.

```
menu_prompt(prompt, event, iowindowfd)
    struct      prompt *prompt,
    struct      inputevent *event;
    int         iowindowfd;
```

`menu_prompt` displays the indicated prompt (`prompt->prt_rect` is `iowindowfd` relative), and then waits for any input event other than mouse motion. It then removes the prompt, and returns the event which ended the prompt's existence in `event`. `iowindowfd` is the window from which input is taken while the prompt is up. The *fullscreen* access method is used during prompt display.

9.5. Selection Management

This section describes an interface to a ***selection manager*** that is used to coordinate access to a single data entity called the ***current selection***. The current selection is globally accessible by any process, thus providing an inter-tool data exchange mechanism.

In the window system, a common style of command specification is one in which the operand is specified first. The operand is called a ***selection*** since it usually requires that the user select something with the pointing device. A selection is highlighted in some way and persists until an operation removes it programmatically or the user performs some action that causes the selection to be removed.

The header file `<suntool/selection.h>` contains the definitions necessary for using selections. The object that describes a selection is:

```
struct  selection {
    int         sel_type,
    int         sel_items,
    int         sel_itembytes,
    int         sel_pubflags;
    caddr_t     sel_privdata;
};

#define  SELTYPE_NULL  0
#define  SELTYPE_CHAR  1
```

`sel_type` indicates the type of the selection. Currently, SELTYPE_NULL (no selection) and SELTYPE_CHAR (ASCII characters) are the only selection types defined. `sel_items` is the number of items in the selection data. `sel_itembytes` is the number of bytes each item occupies in the selection data. `sel_pubflags` is used to contain publicly understood flags that further describe the selection. `sel_privdata` is used to contain 32 bits worth of data that is only interpreted by processes that understand a particular selection type (i.e. the Selection Manager does not look at `sel_privdata`).

The selection structure contains information about the current selection. The actual data representing the current selection (e.g. the characters in a string if the selection is a string) is application dependent. Both the information in the selection structure and the selection data are stored in a single file on the system (call it the ***selection file***). The Selection Manager is simply a package to help an application read or write the current selection file.

The Selection Manager writes the information from the selection struct to the selection file when `selection_set()` is called. The application is responsible for writing the selection data to the selection file when `sel_write()` is called.

```
selection_set(sel, sel_write, sel_clear, windowfd)
    struct      selection *sel
    int         (*sel_write)();
    int         (*sel_clear)();
    int         windowfd;

sel_write(sel, file)
    struct      selection *sel;
    FILE        *file;

sel_clear(sel, windowfd)
    struct      selection *sel;
    int         windowfd;
```

`selection_set` is used to change the current selection. `sel` describes the selection. `sel_write` is a procedure (which must be provided by the client) that is called by `selection_set()` to store information into the selection. Currently, only `selection_set` calls `sel_write`, but in the future `sel_write` might be called at any time. The `sel_write` procedure takes as arguments `sel`, the selection description handed to `selection_set`, and `file`, a standard FILE pointer. The standard I/O library is used to write the selection data to `file`. `windowfd` is the window that is making the selection.

`sel_clear` is a procedure (which must be provided by the client) that the selection manager would call when it wanted the selection currently being set to be dehighlighted. This could happen when another selection had been made. ***This clear feature is not currently implemented. When implemented this call could come at any time after returning from*** `selection_set`.

```
selection_clear(windowfd)
    int         windowfd;
```

is called when `windowfd` wants to clear the current selection. Ideally, there is only one selection on the screen at a time so that the user doesn't become confused about which operand will be affected by his next command.

Since the `sel_clear` feature is not currently implemented, it is the selection maker's (i.e. the client's) decision as to when to dehilight his selection feedback. The only existing use of the selection mechanism waits for the user to move his cursor out of the window that made the selection before dehilighting it.

```
selection_get(sel_read, windowfd)
    int         (*sel_read)();
    int         windowfd;

sel_read(sel, file)
    struct      selection *sel;
    FILE        *file;
```

`selection_get` is used to find out the current selection. `sel_read` is a procedure (which must be provided by the client) that `selection_get` calls to enable the client to retrieve the selection. `windowfd` is the window that wants to find out about the selection.

When an application calls `selection_get()`, the Selection Manager will read the selection information from the selection file into a selection struct. The Selection Manager will then call the specified `sel_read()` function to allow the application to read the selection data from the selection file. The Selection Manager handles the overhead of insuring that the selection file is open and actually reading in the selection information. The application is responsible for reading in the selection type specific selection data.

The `sel_read` procedure takes as arguments `sel`, the selection description of the current selection, and `file`, a standard FILE pointer. The standard io library is used to read the selection data from `file`. `sel_read` should check the type of the selection and make sure that it is a type with which it can deal. For example, if an application only understands a selection consisting of characters, the `sel_read()` function for the application should check the `sel_type` field of the selection struct it is passed to insure that the selection it is about to read is actually a string of characters (as opposed to, say, a string of bits representing a graphic image).

9.6. Window Management

The procedures in this section implement common functions for managing windows.

9.6.1. Window Manipulation

These routines provide the standard window management user interface presented by tool windows:

```
wmgr_open(toolfd, rootfd)
    int          toolfd, rootfd;

wmgr_close(toolfd, rootfd)
    int          toolfd, rootfd;

wmgr_move(toolfd)
    int          toolfd;

wmgr_stretch(toolfd)
    int          toolfd;

wmgr_top(toolfd, rootfd)
    int          toolfd, rootfd;

wmgr_bottom(toolfd, rootfd)
    int          toolfd, rootfd;

wmgr_refreshwindow(windowfd)
    int          windowfd;
```

In each of the above routines, `toolfd` is a file descriptor for a tool window and `rootfd` is a file descriptor for the root window. `wmgr_open` opens a tool window from its iconic state to normal size. If the window is already open, `wmgr_open` does nothing. `wmgr_close` closes a tool window from its normal size to its iconic size. If the window is already closed, `wmgr_close` does nothing. `wmgr_move` prompts the user to move the tool 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 the tool window on the top of the window stack. `wmgr_bottom` places the tool 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;
    struct       inputevent *event;
    bool         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.

Note: The up button event is not consumed.

The window management package provides menu handling code that ties all the routines in this subsection into the **wmgr_toolmenu**. This provides a convenient way of getting access to the same menu that is presented by a tool window. If you don't like the menu provided (you want to add/subtract/change menu items), define and use a new one. The routines in this section should be all you need to put together a functionally similar window manipulation interface.

```
struct  menu *wmgr_toolmenu;

wmgr_setupmenu(toolfd)
    int         toolfd;

wmgr_handletoolmenuitem(menu, mi, toolfd, rootfd)
    struct      menu *menu;
    struct      menuitem *mi;
    int         toolfd, rootfd;
```

To use the default tool menu, call `wmgr_setupmenu` just before you put up **wmgr_toolmenu**. `wmgr_setupmenu` arranges the menu items depending on the tool state (iconic vs. normal). Passing the menu item returned from `menu_display` to **wmgr_handletoolmenuitem** causes the appropriate menu action to be done.

9.6.2. Tool Invocation

The routines in this section provide tool invocation and default position control.

```
#define  WMGR_SETPOS  -1

wmgr_figuretoolrect(rootfd, rect)
    int         rootfd;
    struct      rect *rect;

wmgr_figureiconrect(rootfd, rect)
    int         rootfd;
    struct      rect *rect;
```

These routines allow windows to be assigned initial positions that don't pile up on top of one another. The `rootfd` window maintains a "next slot" position for both normal tool windows and icon windows (see `wmgr_setrectalloc` below). These procedures assign the next slot to the rect if `rect->r_left` or `rect->r_top` is equal to WMGR_SETPOS. A new slot is chosen and is then available for the next window with an undefined position.

These procedures also assign a default width and height if WMGR_SETPOS is given, again for both normal (tool) and iconic rects. `wmgr_figuretoolrect` currently assigns tool window slots that march from near the top middle of the screen towards the bottom left of the screen. It assigns a window size correct for an 80-column by 34-row terminal emulator window. `wmgr_figureiconrect` currently assigns icon slots that march from the left bottom towards the right of the screen. It assigns icon sizes that are 64 by 64 pixels.

```
wmgr_forktool(programname, otherargs, rectnormal, recticon, iconic)
    char        *programname, *otherargs;
    struct      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.

9.6.3. 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;
    struct      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)
    struct      pixwin *pixwin;
    struct      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 (see above) 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.

Appendix A

Rects and Rectlists

This appendix describes the geometric structures used with the ***sunwindow*** layer and a full description of the operations on these structures. Throughout ***sunwindow***, images are dealt with in rectangular chunks. Where complex shapes are required, they are built up out of groups of rectangles. A ***rect*** is a structure that defines a rectangle. A ***rectlist*** is a structure that defines a list of rects.

The header files `<sunwindow/rect.h>` and `<sunwindow/rectlist.h>` contain the definitions of these structures. The library that provides the implementation of the functions of these data types is part of `/usr/lib/libsunwindow.a`.

Although these structures are presented in terms of ***sunwindow*** usage with pixel units, they are really separate and can be thought of as a rectangle algebra package. Any application that needs such a facility should consider using rects and rectlists.

A.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

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

The rectangle lies in a coordinate system whose origin is in the upper left-hand corner and whose dimensions are given in pixels.

A.1.1. 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)
struct rect *rp;
```

returns the coordinate of the last pixel within the rectangle on the right or bottom, respectively.

Useful predicates returning TRUE or FALSE are:

```
#define bool                unsigned
#define TRUE                1
#define FALSE               0

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        */

    struct  rect *r, *r1, *r2;
    coord   x, y;
```

Macros which manipulate dimensions of rectangles are:

```
rect_construct(r, x, y, w, h)
    struct  rect *r;
    int     x, y, w, h;
```

This fills in `r` with the indicated origin and dimensions.

```
rect_marginadjust(r, m)
    struct  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;
    struct  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.

A.1.2. 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`:

```
struct rect rect_bounding(r1, r2)
    struct  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)
    struct  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)
    struct  rect *r;
    coord   *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)
    struct  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.

A.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.

```
struct rectlist {
    coord   rl_x, rl_y;
    struct  rectnode *rl_head;
    struct  rectnode *rl_tail,
    struct  rect rl_bound;
};

struct rectnode {
    struct  rectnode *rn_next;
    struct  rect rn_rect;
};
```

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.

A.2.1. Macros and Constants Defined on Rectlists

Macros to perform common coordinate transformations are provided:

```
rl_rectoffset(rl, rs, rd)
    struct  rectlist *rl;
    struct  rect *rs, *rd;
```

copies `rs` into `rd`, and then adjusts `rd`'s origin by adding the offsets from `rl`.

```
rl_coordoffset(rl, x, y)
    struct  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;
    struct  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.

A.2.2. 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:

```
    struct  rectlist *rl, *rl1, *rl2, *rld;
    struct  rect *r;
    coord   x, y;
```

Predicates return TRUE or FALSE. Refer to the following table for specifics.

Table A-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 A-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)`	An attempt is made 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.

Appendix B

Sample Tool

This appendix contains the source code for a sample tool program that you can use as a model when you write your own tools. The sample program is the graphics window (*gfxtool.c*), which produces a shell subwindow and an empty subwindow in which graphics programs can run.

For more examples, please see the ***Programmer's Tutorial to SunWindows*** and the source files in the directory `/usr/src/sun/suntool`.

B.1. gfxtool.c Code

Code for *gfxtool.c* follows.

```
#ifndef lint
static  char sccsid[] = "@(#)gfxtool.c 1.1 84/12/21 Copyr 1984 Sun Micro";
#endif

/*
 * Copyright (c) 1984 by Sun Microsystems, Inc.
 */

/*
 *  gfxtool - run a process in a tty subwindow with a separate graphic area
 */

#include <stdio.h>
#include <signal.h>
#include <suntool/tool_hs.h>
#include <suntool/ttysw.h>
#include <suntool/ttytlsw.h>
#include <suntool/emptysw.h>

extern char *getenv();

static  int sigwinchcatcher(), sigchldcatcher(), sigtermcatcher();

static  struct tool *tool;
static  struct  toolsw *tsw;

static short ic_image[258] = {
#include <images/gfxtool.icon>
};
mpr_static(gfxic_mpr, 64, 64, 1, ic_image);

static  struct icon icon = {64, 64, (struct pixrect *)NULL, 0, 0, 64, 64,
            &gfxic_mpr, 0, 0, 0, 0, NULL, (struct pixfont *)NULL,
            ICON_BKGRDCLR};

gfxtool_main(argc, argv)
```

```
        int argc;
        char **argv;
{
        char    **tool_attrs = NULL;
        int     become_console = 0;
        char    *tool_name = argv[0], *tmp_str;
        static  char *label_default = "Graphics Tool 2.0";
        static  char *label_console = " (CONSOLE): ";
        static  char label[150];
        static  char icon_label[30];
        static  char *sh_argv[2] = { (char *)NULL, (char *)NULL };
        struct  toolsw *emptysw;
        char    name[WIN_NAMESIZE];

        argv++;
        argc--;
        /*
         * Pick up command line arguments to modify tool behavior
         */
        if (tool_parse_all(&argc, argv, &tool_attrs, tool_name) == -1) {
                tool_usage(tool_name);
                exit(1);
        }
        /*
         * Get ttysw related args
         */
        while (argc > 0 && **argv == '-') {
                switch (argv[0][1]) {
                case 'C':
                        become_console = 1;
                        break;
                case '?':
                        tool_usage(tool_name);
                        fprintf(stderr, "To make the console use -C0);
                        exit(1);
                default:
                        ;
                }
                argv++;
                argc--;
        }
        if (argc == 0) {
                argv = sh_argv;
                if ((argv[0] = getenv("SHELL")) == NULL)
                        argv[0] = "/bin/sh";
        }
        /*
         * Set default icon label
         */
        if (tool_find_attribute(tool_attrs, WIN_LABEL, &tmp_str)) {
                /* Using tool label supplied on command line */
                strncat(icon_label, tmp_str, sizeof(icon_label));
                tool_free_attribute(WIN_LABEL, tmp_str);
        } else if (become_console)
                strncat(icon_label, "CONSOLE", sizeof(icon_label));
        else
                /* Use program name that is run under ttysw */
                strncat(icon_label, argv[0], sizeof(icon_label));
        /*
         * Buildup tool label
```

```
         */
        strcat(label, label_default);
        if (become_console)
                strcat(label, label_console);
        else
                strcat(label, ": ");
        strncat(label, *argv, sizeof(label)-
            strlen(label_default)-strlen(label_console)-1);
        /*
         * Create tool window
         */
        tool = tool_make(
            WIN_LABEL,                label,
            WIN_NAME_STRIPE,          1,
            WIN_BOUNDARY_MGR,               1,
            WIN_ICON,                 &icon,
            WIN_ICON_LABEL,           icon_label,
            WIN_ATTR_LIST,            tool_attrs,
            0);
        if (tool == (struct tool *)NULL)
                exit(1);
        tool_free_attribute_list(tool_attrs);
        /*
         * Create tty tool subwindow
         */
        tsw = ttytlsw_createtoolsubwindow(tool, "ttysw", TOOL_SWEXTENDTOEDGE,
            200);
        if (tsw == (struct toolsw *)NULL)
                exit(1);
       /* Create empty subwindow for graphics */
       emptysw = esw_createtoolsubwindow(tool, "emptysw",
           TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
       if (emptysw == (struct toolsw *)NULL)
              exit(1);
        /* Install tool in tree of windows */
        (void) signal(SIGWINCH, sigwinchcatcher);
        (void) signal(SIGCHLD, sigchldcatcher);
        (void) signal(SIGTERM, sigtermcatcher);
        tool_install(tool);
        /* Start tty process */
       win_fdtoname(emptysw->ts_windowfd, name);
       we_setgfxwindow(name);
        if (become_console)
                ttysw_becomeconsole(tsw->ts_data);
        if (ttysw_fork(tsw->ts_data, argv, &tsw->ts_io.tio_inputmask,
            &tsw->ts_io.tio_outputmask, &tsw->ts_io.tio_exceptmask) == -1) {
                perror(tool_name);
                exit(1);
        }
        /* Handle input */
        tool_select(tool, 1);
        /* Cleanup */
        tool_destroy(tool);
        exit(0);
}

static
sigchldcatcher()
{
        tool_sigchld(tool);
```

```
}

static
sigwinchcatcher()
{
        tool_sigwinch(tool);
}

static
sigtermcatcher()
{
        /* Special case: Do ttysw related cleanup (e.g., /etc/utmp) */
        ttysw_done(tsw->ts_data);
        exit(0);
}
```

Appendix C

Sample Graphics Programs

Use these sample programs as templates for your own graphics programs. The programs are: a bouncing ball demonstration (*bouncedemo.c*) and a "movie camera" program (*framedemo.c*), which displays files sequentially like movie frames, for example, for producing a rotating globe.

For more sample programs, please see the ***Programmer's Tutorial to SunWindows.***

C.1. bouncedemo.c Source

Code for the *bouncedemo.c* follows.

```
#ifndef lint
static char sccsid[] = "@(#)bouncedemo.c 1.1 84/12/21 SMI";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *	Overview:	Bouncing ball demo in window
 */

#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <sunwindow/rect.h>
#include <sunwindow/rectlist.h>
#include <sunwindow/pixwin.h>
#include <suntool/gfxsw.h>

main(argc, argv)
	int argc;
	char **argv;
{
	short	x, y, vx, vy, z, ylastcount, ylast;
	short	Xmax, Ymax, size;
	struct	rect rect;
	struct	gfxsubwindow *gfx = gfxsw_init(0, argv);

	if (gfx == (struct gfxsubwindow *)0)
		exit(1);
Restart:
	win_getsize(gfx->gfx_windowfd, &rect);
	Xmax = rect_right(&rect);
	Ymax = rect_bottom(&rect);
	if (Xmax < Ymax)
		size = Xmax/29+1;
	else
```

```
                size = Ymax/29+1;
        x=rect.r_left;
        y=rect.r_top;
        vx=4;
        vy=0;
        ylast=0;
        ylastcount=0;
        pw_writebackground(gfx->gfx_pixwin, 0, 0, rect.r_width, rect.r_height,
            PIX_SRC);
        while (gfx->gfx_reps) {
                if (gfx->gfx_flags&GFX_DAMAGED)
                        gfxsw_handlesigwinch(gfx);
                if (gfx->gfx_flags&GFX_RESTART) {
                        gfx->gfx_flags &= ~GFX_RESTART;
                        goto Restart;
                }
                if (y==ylast) {
                        if (ylastcount++ > 5)
                                goto Reset;
                } else {
                        ylast = y;
                        ylastcount = 0;
                }
                pw_writebackground(gfx->gfx_pixwin, x, y, size, size,
                    PIX_NOT(PIX_DST));
                x=x+vx;
                if (x>(Xmax-size)) {
                        /*
                         * Bounce off the right edge
                         */
                        x=2*(Xmax-size)-x;
                        vx= -vx;
                } else if (x<rect.r_left) {
                        /*
                         * bounce off the left edge
                         */
                        x= -x;
                        vx= -vx;
                }
                vy=vy+1;
                y=y+vy;
                if (y>=(Ymax-size)) {
                        /*
                         * bounce off the bottom edge
                         */
                        y=Ymax-size;
                        if (vy<size)
                                vy=1-vy;
                        else
                                vy=vy / size - vy;
                        if (vy==0)
                                goto Reset;
                }
                for (z=0; z<=1000; z++);
                continue;
Reset:
                if (--gfx->gfx_reps <= 0)
                        break;
                x=rect.r_left;
                y=rect.r_top;
```

```
                vx=4;
                vy=0;
                ylast=0;
                ylastcount=0;
        }
        gfxsw_done(gfx);
}
```

C.2. framedemo.c Source

Code for *framedemo.c* follows.

```
#ifndef lint
static char sccsid[] = "@(#)framedemo.c 1.1 84/12/21 SMI";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:       Frame displayer in windows.  Reads in all the
 *                      files of form "frame.xxx" in working directory &
 *                      displays them like a movie.
 *                      See constants below for limits.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/time.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_util.h>
#include <pixrect/bw1var.h>
#include <pixrect/memvar.h>
#include <sunwindow/rect.h>
#include <sunwindow/rectlist.h>
#include <sunwindow/pixwin.h>
#include <sunwindow/win_input.h>
#include <sunwindow/win_struct.h>
#include <suntool/gfxsw.h>

#define         MAXFRAMES       1000
#define         FRAMEWIDTH      256
#define         FRAMEHEIGHT     256
#define         USEC_INC    50000
#define         SEC_INC         1

static  struct pixrect *mpr[MAXFRAMES];
static  struct timeval timeout = {SEC_INC,USEC_INC}, timeleft;
static  char s[] = "frame.xxx";
static  struct gfxsubwindow *gfx;
static  int frames, framenum, ximage, yimage;
static  struct rect rect;

main(argc, argv)
        int argc;
        char **argv;
{
```

```
        int     fd, framedemo_selected();
        struct inputmask im;

        for (frames = 0; frames < MAXFRAMES; frames++) {
                sprintf(&s[6], "%d", frames + 1);
                fd = open(s, O_RDONLY, 0);
                if (fd == -1) {
                        break;
                }
                mpr[frames] = mem_create(FRAMEWIDTH, FRAMEHEIGHT, 1);
                read(fd, mpr_d(mpr[frames])->md_image,
                    FRAMEWIDTH*FRAMEHEIGHT/8);
                close(fd);
                }
        if (frames == 0) {
            printf("Couldn't find any 'frame.xx' files in working directory0);
            return;
        }
        /*
         * Initialize gfxsw
         */
        gfx = gfxsw_init(0, argv);
        if (gfx == (struct gfxsubwindow *)0)
                exit(1);
        /*
         * Set up input mask
         */
        input_imnull(&im);
        im.im_flags |= IM_ASCII;
        gfxsw_setinputmask(gfx, &im, &im, WIN_NULLLINK, 0, 1);
        /*
         * Main loop
         */
        framedemo_nextframe(1);
        timeleft = timeout;
        gfxsw_select(gfx, framedemo_selected, 0, 0, 0, &timeleft);
        /*
         * Cleanup
         */
        gfxsw_done(gfx);
}

framedemo_selected(gfx, ibits, obits, ebits, timer)
        struct gfxsubwindow *gfx;
        int     *ibits, *obits, *ebits;
        struct timeval **timer;
{
        if ((*timer && ((*timer)->tv_sec == 0) && ((*timer)->tv_usec == 0)) ||
            (gfx->gfx_flags & GFX_RESTART)) {
                /*
                 * Our timer expired or restart is true so show next frame
                 */
                if (gfx->gfx_reps)
                        framedemo_nextframe(0);
                else
                        gfxsw_selectdone(gfx);
        }
        if (*ibits & (1 << gfx->gfx_windowfd)) {
                struct inputevent event;
```

```
                /*
                 * Read input from window
                 */
                if (input_readevent(gfx->gfx_windowfd, &event)) {
                        perror("framedemo");
                        return;
                }
                switch (event.ie_code) {
                case 'f': /* faster usec timeout */
                        if (timeout.tv_usec >= USEC_INC)
                                timeout.tv_usec -= USEC_INC;
                        else {
                                if (timeout.tv_sec >= SEC_INC) {
                                        timeout.tv_sec -= SEC_INC;
                                        timeout.tv_usec = 1000000-USEC_INC;
                                }
                        }
                        break;
                case 's': /* slower usec timeout */
                        if (timeout.tv_usec < 1000000-USEC_INC)
                                timeout.tv_usec += USEC_INC;
                        else {
                                timeout.tv_usec = 0;
                                timeout.tv_sec += 1;
                        }
                        break;
                case 'F': /* faster sec timeout */
                        if (timeout.tv_sec >= SEC_INC)
                                timeout.tv_sec -= SEC_INC;
                        break;
                case 'S': /* slower sec timeout */
                        timeout.tv_sec += SEC_INC;
                        break;
                case '?': /* Help */
                        printf("'s' slower usec timeout0f' faster usec timeout0S' slower sec timeout0F' faster sec timeout0);
                        /*
                         * Don't reset timeout
                         */
                        return;
                default:
                        gfxsw_inputinterrupts(gfx, &event);
                }
        }
        *ibits = *obits = *ebits = 0;
        timeleft = timeout;
        *timer = &timeleft;
}

framedemo_nextframe(firsttime)
        int     firsttime;
{
        int     restarting = gfx->gfx_flags&GFX_RESTART;

        if (firsttime || restarting) {
                gfx->gfx_flags &= ~GFX_RESTART;
                win_getsize(gfx->gfx_windowfd, &rect);
                ximage = rect.r_width/2-FRAMEWIDTH/2;
                yimage = rect.r_height/2-FRAMEHEIGHT/2;
                pw_writebackground(gfx->gfx_pixwin, 0, 0,
                    rect.r_width, rect.r_height, PIX_CLR);
```

```
        }
        if (framenum >= frames) {
                framenum = 0;
                gfx->gfx_reps--;
        }
        pw_write(gfx->gfx_pixwin, ximage, yimage, FRAMEWIDTH, FRAMEHEIGHT,
            PIX_SRC, mpr[framenum], 0, 0);
        if (!restarting)
                framenum++;
}
```

Appendix D

Programming Notes

Here are useful hints for programmers who use any of the ***pixrect***, ***sunwindow*** or ***suntool*** libraries.

D.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 SunWindows are accompanied by a description of the nature of the changes and appropriate responses to them.

D.2. Program By Example

We recommend that you try to program by example whenever possible. Take an existing program similar to what you need and modify it. Appendix B contains the source for a sample tool, and Appendix C contains some sample graphics programs. There are many other examples shown in the ***Programmer's Tutorial to SunWindows.***

D.3. Header Files Needed

If you have problems finding the necessary header files for compiling your program, using the examples may help as many of the header files are already included. Moreover, there are certain header files that include most of the header files necessary for working at a certain level. The following table shows these header files.

Include only one of these header files plus whatever extra header files you need. In particular, you'll need to add the header file for each subwindow type that you use, the menu header file if you use menus, the selection header file if you are going to use selections, and so on. However, you'll probably only have to add a single header file for each additional increment of high-level functionality.

Table D-1: Header Files Required

Use	When Working at the Level of
`<suntool/tool_hs.h>`	suntool tool-building facilities; includes headers needed to work at the more primitive layers as well
`<suntool/gfx_hs.h>`	the suntool (standalone or "take over") graphics subwindow facilities; includes headers needed to work at the more primitive layers as well
`<sunwindow/window_hs.h>`	sunwindow basic window facilities layer; includes headers needed to work at the pixrect layer as well
`<pixrect/pixrect_hs.h>`	pixrect display primitives layer

D.4. Lint Libraries

SunWindows provides ***lint libraries*** to help you run `lint` over your program source. `lint` catches argument mismatches and provides better type-checking than the C compiler. `Llib-lpixrect`, `lib-lsunwindow`, and `llib-lsuntool` are the source files to make the actual binary `lint(1)` libraries: `llib-lpixrect.ln`, `llib-lsunwindow.ln`, and `llib-lsuntool.ln`. These files are found on `/usr/lib/lint/`.

D.5. Library Loading Order

When loading programs, remember to load higher level libraries first, that is, `-lsuntool -lsunwindow -lpixrect`.

D.6. Shared Text

The tools released with `suntools` rely on text sharing to reduce the memory working set. This is accomplished by placing the entire collection of tools in a single object file. This has the effect of letting each separate process share the same object code in memory. With many windows active at once this can achieve significant memory savings.

There are trade-offs to using this approach. The main one is that the maximum number of per-process and non-sharable initial data pages tends to be larger. However, the paged virtual memory tends to reduce the effect of this by only having the working set paged in.

The upshot of this is that you may want to either add the tools that you create to the released shared object file or to bundle a few tools together into their own object file. To add tools to the released shared object file, please see `/usr/src/sun/suntool/toolmerge.c`.

D.7. Error Message Decoding

The default error reporting scheme described at the end of ***Window Manipulation*** displays a long hex 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.

D.8. 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 in the section entitled ***Overlapped Windows: Imaging Facilities - Damage***, a process receives a SIGWINCH whenever one of its windows changes state. In particular, as soon as a tool issues a `tool_install`, 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** followed by RETURN. '1c' is the hex number for 28, which is SIGWINCH's number. Re-enable signal breaking by typing **1c:t** followed by 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.

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 D-2: *sunwindow* 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 due to 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 anytime after `main` has been called.

D.9. Sufficient User Memory

To use the `suntool` environment comfortably requires adequate user memory for SunWindows and the Sun UNIX operating system. To achieve the best performance, reconfigure your own kernel, deleting unused device drivers. The procedure is documented in the manual *Installing UNIX on the Sun Workstation.* For a workstation on the network with a single disk drive, you will be able to reclaim significant usable memory.

For the recommended amount of memory, see the manual *Installing UNIX on the Sun Workstation.*

D.10. Coexisting with UNIX

This section discusses how a SunWindows 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 Interface Overview*** and the `signal`(3) and `tty`(4) entries in the ***System Interface Manual for the Sun Workstation***.

This discussion explicitly notes those places where the shells and debuggers interact differently with a tool.

D.10.1. 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 tool calls `tool_create` or `tool_make`, the procedure called changes the tool's process group to its own process number. The following sections describe the effects of this change.

D.10.1.1. 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 `tool_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.

D.10.1.2. 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. 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 SunWindows 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 SunWindows locks, as there is no opportunity to release them. Depending on the lock timeouts, the kernel will eventually break the locks, but 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 (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 SunWindows 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`.

Appendix E

Writing a Pixrect Driver

Sun has defined a common programming interface to pixel addressable devices that enables, in particular, device independent access to all Sun frame buffers. This interface is called the ***pixrect*** interface. Existing Sun supported software systems access the frame buffer through the pixrect interface. Sun encourages customers with other types of frame buffers (or other types of pixel addressable devices) to provide a pixrect interface to these devices.

This appendix describes how to write a pixrect driver. It is assumed that you have already read the chapter on ***Pixel Data and Operations*** in this manual; it describes the programming interface to the basic operations that must be provided in order to generate a complete pixrect implementation. It is also assumed that you have read or will refer to the ***Device Driver Tutorial for the Sun Workstation UNIX System*** for the section on writing the kernel device driver portion of the pixrect implementation.

This appendix contains auxiliary material of interest only to pixrect driver implementers, not programmers accessing the pixrect interface. This document explains how to plug a new pixrect driver into the software architecture so that it may be used in a device independent manner. Also, utilities and conventions that may be of use to the pixrect driver implementor are discussed.

This appendix walks through some of the C language source code for the pixrect driver for the Sun 1 color frame buffer. There is no signifigance to the fact that we are using the Sun 1 color frame buffer as an example. Another pixrect driver would have been just as good.

The actual source code that is presented here is boiler-plate, i.e., almost every pixrect driver will be the same. You should be able to make your own driver just from the documentation alone. However, a complete source example for an existing pixrect driver would probably expedite the development of your own driver. The complete device specific source files for the Sun 1 color frame buffer pixrect driver is available as a source code purchase option (available without a UNIX source license).

This document is germane to release 1.1 of the software for the Sun Workstation. In future releases, any changes that a pixrect driver implementation might need to respond to will be completely documented.

E.1. Glossary

Here are some terms that are used in this document:

- **pixel** - Picture element (single dot). May be any number of bits deep.
- **pixrect driver** - That device specific collection of code that implements a pixel addressable device access method that conforms to the pixrect interface. This includes the device specific code that resides in the UNIX kernel. A pixrect driver is sometimes referred to as a **pixrect implementation**.

- **pixrect library** - That collection of code (device independent and device specific) available to user programs.
- **pixrect kernel device driver** - The code in the UNIX kernel associated with a particular pixrect driver.

E.2. What You'll Need

These are the tools and pieces that you'll need before assembling your pixrect driver:

- You need the correct documentation: ***Writing a Pixrect Driver*, *Programmer's Reference Manual for SunWindows***, and ***Device Driver Tutorial for the Sun Workstation UNIX System*.**
- You need to know how to drive the hardware of your pixel addressable device. The absolute minimum requirements a pixel addressable device must meet is the ability to read and write single pixel values. [One could imagine a device that doesn't even meet the minimum requirements being used as a pixel addressable device. We will not discuss any of the ways that such a device might fake the minumum requirements].
- You must have a UNIX kernel building environment. No extra source is required.
- You must have the released pixrect library file and its accompanying header files. No extra source is required.
- For any pixrect based programs that you'll want to run on your pixel addressable device, you'll need the object and library files from which they are built so that you can load your pixrect driver with these files.

E.3. Implementation Strategy

This is one possible step-by-step approach to implementing a pixrect driver:

- Write and debug pixrect creation and destruction. This involves the pixrect kernel device driver that lets you `open`(2) and `mmap`(2) the physical device from a user process. The private **`cg1_make`** routine must be written. The **`cg1_region`** and **`cg1_destroy`** pixrect operation must be written.
- Write and debug the basic pixel rectangle function. The **`cg1_putattributes`** and **`cg1_putcolormap`** pixrect operations must be written in addition to the **`cg1_rop`** routine.
- Write and debug batchrop routines. The **`cg1_batchrop`** pixrect operation must be written.
- Write and debug vector drawer. The **`cg1_vector`** pixrect operation must be written.
- Write and debug remaining pixrect operations: **`cg1_stencil`**, **`cg1_get`**, **`cg1_put`**, **`cg1_getattributes`** and **`cg1_getcolormap`**.
- Build kernel with minimal basic pixel rectangle function for use by the cursor tracking mechanism in the SunWindows kernel device driver. Also include the colormap access routines for use by the colormap segmentation mechanism in the SunWindows kernel device driver.

- Load and test SunWindows programs with new pixrect driver. Experience has shown that when you are able to run released SunWindow programs that your pixrect driver is in pretty good shape.

E.4. Files Generated

Here is the list of source files generated that implement the example pixrect driver:

- `cg1reg.h` - A header file describing the structure of the raster device. It contains macros used to address the raw device.
- `cg1var.h` - A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.
- `/sys/sundev/cgone.c` - The pixrect kernel device driver code.
- `cg1.c` - The pixrect creation and destruction routines.
- `cg1_region.c` - The region creation routine.
- `pr_makefun.c` - Replaces an existing module and contains the vector of pixrect make operations.
- `cg1_batch.c` - The batchrop routine.
- `cg1_colormap.c` - The colormap access and attribute setting routines .
- `cg1_getput.c` - The single pixel access routines.
- `cg1_rop.c` - The basic pixel rectangle manipulation routine.
- `cg1_stencil.c` - The stencil routine.
- `cg1_vec.c` - The vector drawer.

E.4.1. Memory Mapped Devices

Some devices are memory mapped, e.g., the Sun 2 monochrome video frame buffer. With such devices, their pixels are manipulated directly as main memory; there are no device specific registers through which the pixels are accessed. Memory mapped devices are able to rely on the memory pixrect driver for most of its operations. The only files that the Sun 2 monochrome video frame buffer supplies are:

- `bw2var.h` - A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.
- `/sys/sundev/bwtwo.c` - The pixrect kernel device driver code.
- `bw2.c` - The pixrect creation and destruction routines.

The operations vector for the Sun 2 monochrome pixrect driver is:

```
struct  pixrectops bw2_ops = {
        mem_rop, mem_stencil, mem_batchrop,
        0, bw2_destroy, mem_get, mem_put, mem_vector,
        mem_region, mem_putcolormap, mem_getcolormap,
        mem_putattributes, mem_getattributes
};
```

E.5. Pixrect Private Data

Each pixrect device must have a private data object that contains instance specific data about the state of the driver. It is not acceptible to have global data shared among all the pixrects objects. The device specific portion of the pixrect data must contain certain information:

- An offset from the upper left-hand corner of the pixel device. This offset, plus the width and height of the pixrect from the public portion, is used to determine the clipping rectangle used during pixrect operations.
- A flag for distinguishing between primary and secondary pixrects. Primary pixrects are the owners of dynamically allocated resources shared between primary and secondary pixrects.
- A file descriptor to the pixrect kernel device. Usually, the file descriptor is used while mapping pages into the user process address space so that the device may be addressed. One could imagine a pixrect driver that had some of its pixrect operations implemented inside the kernel. The file descriptor would then be the key to communicating with that portion of the package via `read`(2), `write`(2) and `ioctl`(2) system calls.

Here is other possible data maintained in the pixrect's private data:

- For many devices, a virtual address pointer is part of the private data so that the device can be accessed from user code.
- For color devices, there is a mask to enable access to specific bit planes.
- For monochrome devices, there is a video invert flag. This replaces the colormap of color devices.

E.6. Creation and Destruction

This section covers the code for pixrect object creation and destruction. Code for the Sun 1 color frame buffer pixrect driver is presented as an example.

There are three public pathways to creating a pixrect:

- `pr_open` creates a primary pixrect.
- `pr_region` creates a secondary pixrect which specifies a subregion in an existing pixrect.

There are two public pathways to destroying a pixrect:

- `pr_destroy` destroys a primary or secondary pixrect. Clients of the pixrect interface are responsible for destroying all extant secondary pixrects before destroying the primary pixrect from which they were derived.
- `pr_close` simply calls `pr_destroy`.

E.6.1. Creating a Primary Pixrect

In this section, the private `cg1_make` pixrect operation is described. This is the flow of control for `pr_open`:

- Higher levels of software call `pr_open`, which takes a device file name (e.g., `/dev/cgone0`).

- `pr_open` open(2)s the device and finds out its type and size via an FBIOGTYPE `ioctl(2)` call (see `<sun/fbio.h>`).
- `pr_open` uses the type of pixel addressable device to index into the `pr_makefun` array of procedures (more on this later) and calls the referenced pixrect make function, `cgl_make`.
- `cgl_make` returns the primary pixrect (it workings are discussed below).
- `pr_open` closes its handle on the device and the pixrect is returned.

Here is a partial listing of cgl.c that contains code that is germane to the `cgl_make` procedure. As it is for other code presented in this document, it is here so you can refer back to it as you read the subsequent explanation. Some lines are numbered for reference and normal C comments have been removed in favor of the accompanying text.

```
#include <sys/types.h>
#include <stdio.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_util.h>
#include <pixrect/cglreg.h>
#include <pixrect/cglvar.h>

static     struct pr_devdata *cgldevdata;                       /* cgl.1*/

struct     pixrectops cgl_ops = {                               /* cgl.2*/
     cgl_rop, cgl_stencil, cgl_batchrop, 0, cgl_destroy, cgl_get,
     cgl_put, cgl_vector, cgl_region, cgl_putcolormap, cgl_getcolormap,
     cgl_putattributes, cgl_getattributes,
};

struct pixrect *
cgl_make(fd, size, depth)                                       /* cgl.3*/
     int  fd;                                                   /* cgl.4*/
     struct     pr_size size;
     int  depth;
{
     struct pixrect *pr;
     register struct cglpr *cgpr;                               /* cgl.5*/
     struct     pr_devdata *dd;                                 /* cgl.6*/

     if (depth != CG1_DEPTH || size.x != CG1_WIDTH || size.y != CG1_HEIGHT) {/* cgl.7
          fprintf(stderr, "cgl_make sizes wrong %D %D %D\n",
              depth, size.x, size.y);
          return (0);
     }
     if (!(pr = pr_makefromfd(fd, size, depth, &cgldevdata, &dd,  /* cgl.8*/
         sizeof(struct cglfb), sizeof(struct cglpr), 0)))
          return (0);
     pr->pr_ops = &cgl_ops;                                     /* cgl.9*/
     cgpr = (struct cglpr *)pr->pr_data;                        /*cgl.10*/
     cgpr->cgpr_fd = dd->fd;                                    /*cgl.11*/
     cgpr->cgpr_va = (struct cglfb *)dd->va;                    /*cgl.12*/
     cgpr->cgpr_planes = 255;                                   /*cgl.13*/
     cgpr->cgpr_offset.x = cgpr->cgpr_offset.y = 0;             /*cgl.14*/
     cgl_setreg(cgpr->cgpr_va, CG_STATUS, CG_VIDEOENABLE);      /*cgl.15*/
     return (pr);                                               /*cgl.16*/
}
```

Line cgl.7 does some consistency checking to make sure that the dimensions of the pixel addressable device and the client's idea about the dimensions of the device match.

```
struct *pixrect
pr_makefromfd(fd, size, depth, devdata, curdd,
              mmapbytes, privdatabytes, mmapoffsetbytes)
    struct  pr_size size;
    struct  pr_devdata **devdata, **curdd;
    int     fd, depth, mmapbytes, privdatabytes, mmapoffsetbytes;
    int mmapbytes, privdatabytes, mmapoffsetbytes);
```

Line cg1.8 calls the pixrect library routine `pr_makefromfd` to do most of the work:

- Allocates a `struct pixrect` object using the `calloc` library call. The pixrect is filled in with `size` and `depth` parameters.
- Allocates an object of the size `privdatabytes` using the `calloc` library call and placing a pointer to it in the `pr_data` field of the allocated pixrect.
- dup(2)s the passed in file descriptor `fd` so that when the caller closes the file descriptor the device wouldn't close.
- `valloc`(2)s the amount of space `mmapbytes`.
- `mmap`(2)s the space returned from `valloc` to the device.
- If an error is detected during any of the above calls, an error is written to `stderr`. A NULL pixrect handle is returned in this case.
- Returns the allocated pixrect.

This brings us to the issue of minimizing resources used by the pixrect driver. `pr_open`, and thus `cg1_make`, can be (and are) called many times thus creating a situation in which there are many primary pixrects open at a time. A pixrect should maintain an open file descriptor and (usually) a non-trivial amount of virtual address space mapped into the user process's address space. Both the number of open file descriptors (default 20 and max 30), the virtual address space (max 16 megabytes) and the disk swap space needed to support the virtual memory (configurable) are finite resources. However, multiple open pixrects can share all these resources.

The pixrect library supports a resource sharing mechanism, part of which is implemented in `pr_makefromfd`. The `devdata` parameter passed to `pr_makefromfd` is the head of a linked list of `pr_devdata` structures of which there is one per pixrect driver. It is sufficant to say that through the data maintained on this list, sharing of the scarce resources described above can be accomplished.

The `curdd` parameter passed to `pr_makefromfd` is set to be the `pr_devdata` structure that applies to the device indentified by `fd`.

Lines cg1.9 through cg1.14 are concerned with initializing the pixrect's private data with dynamic information described in `dd` (`curdd` in the previous paragraph) and static information about the pixel addressable device.

Line cg1.15 is where the video signal for the device is enabled. By convention, very raster device should make sure that it is enabled.

E.6.2. Creating a Secondary Pixrect

In this section, the `cgl_region` pixrect operation is described. Here is all of `cgl_region.c`.

```
struct pixrect *
cgl_region(src)
     struct pr_subregion src;
{
     register struct pixrect *pr;
     register struct cglpr *scgpr = cgl_d(src.pr), *cgpr;
     int zero = 0;

     pr_clip(&src, &zero);                                          /* cgl_region.1*/
     if ((pr = (struct pixrect *)calloc(1, sizeof (struct pixrect))) == 0)/* cgl_regi
          return (0);
     if ((cgpr = (struct cglpr *)calloc(1, sizeof (struct cglpr))) == 0) {/* cgl_regi
          free(pr);
          return (0);
     }
     pr->pr_ops = &cgl_ops;                                         /* cgl_region.4*/
     pr->pr_size = src.size;                                        /* cgl_region.5*/
     pr->pr_depth = CG1_DEPTH;                                      /* cgl_region.6*/
     pr->pr_data = (caddr_t)cgpr;                                   /* cgl_region.7*/
     cgpr->cgpr_fd = -1;                                            /* cgl_region.8*/
     cgpr->cgpr_va = scgpr->cgpr_va;                                /* cgl_region.9*/
     cgpr->cgpr_planes = scgpr->cgpr_planes;                        /*cgl_region.10*/
     cgpr->cgpr_offset.x = scgpr->cgpr_offset.x + src.pos.x;        /*cgl_region.11*/
     cgpr->cgpr_offset.y = scgpr->cgpr_offset.y + src.pos.y;        /*cgl_region.12*/
     return (pr);
}
```

`cgl_region` is less complex then `cgl_make`. The first thing done is to clip the requested subregion to fall within the source pixrect (line cgl_region.1).

```
pr_clip(dstp, srcp)
    struct  pr_subregion *dstp;
    struct  pr_prpos *srcp;
```

`pr_clip` adjusts the position and size of `dstp`, the destination pixrect subregion, to fall within `dstp->pr`. If `*scrp`, the source pixrect position, is not zero then the position of the source is clipped to fall within `dstp`.

Next, objects are allocated for the pixrect and the pixel addressable device's private data (line cgl_region.2 and cgl_region.3). Then, similarly to the later part of `cgl_make`, the two new data objects are initialized (lines cgl_region.4 through cgl_region.12). One thing to note is that the cgl driver uses a −1 in the file desriptor field of the pixrect's private data to indicate that this pixrect is secondary (line cgl_region.8).

E.6.3. Destroying a Pixrect

In this section, the `cgl_destroy` pixrect operation is described. It works on secondary and primary pixrects. Here is more of cgl.c.

```
cg1_destroy(pr)
        struct pixrect *pr;
{
        register struct cg1pr *cgpr;

        if (pr == 0)
                return (0);
        if (cgpr = cg1_d(pr)) {                                         /*cg1.30*/
                if (cgpr->cgpr_fd != -1) {                              /*cg1.31*/
                        pr_unmakefromfd(cgpr->cgpr_fd, &cg1devdata);    /*cg1.32*/
                }
                free(cgpr);                                             /*cg1.33*/
        }
        free(pr);                                                       /*cg1.34*/
        return (0);
}
```

Note that dynamic memory is freed (lines cg1.33 and cg1.34). Also, note that only a primary pixrect (as indicated by a file descriptor that is not −1) invokes a call to `pr_unmakefromfd` (line cg1.32).

```
pr_unmakefromfd(fd, devdata)
       struct  pr_devdata **devdata;
       int     fd;
```

This pixrect library routine is the counterpart of `pr_makefromfd`. If the device identified by the file descriptor `fd` has no more pixrects associated with it (as determined from `devdata`) then the resources associated with it are released. Note: Actually this is misleading. In the current release (2.0), munmap(2) and `vfree`(3) are not implemented. Thus the virtual memory allocated by `pr_makefromfd` cannot safely be released. As a result, `pr_unmakefromfd` never releases virtual memory. The virtual memory will be reused in `pr_makefromfd` on subsequent calls.

E.6.4. The pr_makefun Operations Vector

As mentioned above, `pr_open` calls `cg1_make` through the `pr_makefun` procedure vector. This is what `pr_makefun` looks like (it is the sole contents of pr_makefun.c):

```
#include <pixrect/pixrect_hs.h>
#include <sun/fbio.h>
#include <sys/ioctl.h>

struct    pixrect *(*(pr_makefun[FBTYPE_LASTPLUSONE]))() = {
        (struct pixrect *(*)())bw1_make,
        (struct pixrect *(*)())cg1_make,
        (struct pixrect *(*)())bw2_make,
        (struct pixrect *(*)())cg2_make,
       0/*(struct pixrect *(*)())bw3_make*/,
       0/*(struct pixrect *(*)())cg3_make*/,
       0/*(struct pixrect *(*)())bw4_make*/,
       0/*(struct pixrect *(*)())cg4_make*/,
       0/*(struct pixrect *(*)())FBTYPE_NOTSUN1_make*/,       /*pr_makefun.1*/
       0/*(struct pixrect *(*)())FBTYPE_NOTSUN2_make*/,
       0/*(struct pixrect *(*)())FBTYPE_NOTSUN3_make*/,
/* uncomment the above as the functions become available */
};
```

When adding some new pixrect driver, you need to assign it some unused constant from `<sun/fbio.h>`, e.g., FBTYPE_NOTSUN1. This then becomes the device identifier for your new pixrect driver. You need to generate a new version of the source file pr_makefun.c with the above data structure except that the array entry `pr_makefun`[FBTYPE_NOTSUN1] would contain the pixrect make procedure for your FBTYPE_NOTSUN1 pixrect driver (line pr_makefun.1). The old `pr_makefun.o` in the pixrect library could be replaced with your new `pr_makefun.o` using `ar`(1).

E.7. Pixrect Kernel Device Driver

A pixrect kernel device driver supports the pixel addressable device as a fullblown UNIX device. It also supports use of this device by the SunWindows driver so that the cursor can be tracked and the colormap loaded within the kernel. The document ***Device Driver Tutorial for the Sun Workstation UNIX System*** contains the details of device driver construction. It also contains an overview.

The code in this section comes from `cgone.c`. In the kernel, suffixes that end with a number (like cg1) confuse the conventions surrounding device driver names. A number suffix refers to the minor device number of a device. Therefore, in our example, `cg1` becomes `cgone` where the naming has something to do with the pixrect kernel device driver.

E.7.1. Configurable Device Support

Raster devices typically hang off a high speed bus (e.g., Multibus) or are plugged into a high speed communications port. At kernel building time the UNIX auto-configuration mechanism is told what devices to expect and where they should be found. At boot time the auto-configuration mechanism checks to see if each of the devices it expects are present.

This section deals with the auto-configuration aspects of the driver. This driver is written in the conventional style that supports multiple units of the same device type. It is recommended that you follow this style even if you aren't anticipating multiple pixel addressable devices of your type on a single UNIX system.

```
#include "cgone.h"
#include "win.h"
#include "../machine/pte.h"
#include "../h/param.h"
#include "../h/systm.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/proc.h"
#include "../h/buf.h"
#include "../h/conf.h"
#include "../h/file.h"
#include "../h/uio.h"
#include "../h/ioctl.h"
#include "../sun/mmu.h"
#include "../sun/fbio.h"
#include "../sundev/mbvar.h"
#include "../pixrect/pixrect.h"
#include "../pixrect/pr_util.h"
#include "../pixrect/cg1reg.h"
#include "../pixrect/cg1var.h"

#define CG1SIZE (sizeof (struct cg1fb))

int  cgoneprobe(), cgoneintr();
struct    mb_device *cgoneinfo[NCGONE];                           /* cgone.2*/
u_long    cgonestd[] = { 0xe8000, 0xec000, 0 };                   /* cgone.3*/
struct    mb_driver cgonedriver = {                               /* cgone.4*/
     cgoneprobe, 0, 0, 0, 0, cgoneintr, cgonestd, 0, CG1SIZE,
     "cgone", cgoneinfo, 0, 0, 0,
};

cgoneprobe(reg, unit)                                             /* cgone.5*/
     caddr_t reg;
     int  unit;
{
     /*
      * if (found device at address reg) return (CG1SIZE);
      * else return (0);
      */
}

cgoneintr()
{
     return(fbintr(NCGONE, cgoneinfo, cgoneintclear));            /* cgone.6*/
}

cgoneintclear(cg1fb)                                              /* cgone.7*/
     struct    cg1fb *cg1fb;
{
     cg1_intclear(cg1fb);                                         /* cgone.8*/
}
```

This is how the driver is plugged into the auto-configuration mechanism. `/etc/config` reads a line in the configuration file for a Sun 1 color frame buffer:

```
device          cgone0 at mb0 csr 0xe8000 priority 3
```

An external reference to `cgonedriver` (line cgone.4) is made in a table maintained by the auto-configuration mechanism. At boot time, if the auto-configuration mechanism can resolve the reference to `cgonedriver` then the contents of this structure are used to configure in the device:

- `cgoneprobe` - The name of the probe procedure (line cgone.5).
- `cgoneintr` - The name of the interrupt procedure (line cgone.6).
- `cgonestd` - A list of standard physical addresses at which the device may be located (line cgone.3).
- CG1SIZE - The size in bytes of the address space of the device.
- `"cgone"` - The prefix of the device. Used in status and error messages.
- `cgoneinfo` - The array of devices pointers of the driver's type (line cgone.2).
- The other field's defaults suffice for most pixel addressable devices.

`cgoneprobe` is called to let the driver decide if the virtual address at `reg` is indeed a device that this driver recognizes as one of its own. The `unit` argument is the minor device number of this device. Writing a good probe routine can be difficult. The trick is to use some idiosyncrasy of the device that differentiates it from others. The real driver for the Sun 1 color frame buffer determines that it is addressing a Sun 1 color frame buffer by setting it up to invert the data written to it and reading back the result. The details of this code are not germane to this discussion and is not included. Zero is returned if the probe fails and CG1SIZE is returned if the probe succeeds.

`cgoneintr` is called when an interrupt is generated at the beginning of the vertical retrace. There are a variety of things that one might want to syncronize with a such an interrupt, e.g., load the colormap or move the cursor. Currently, the utility `fbintr` simply disables the interrupt from happening again (line cgone.6).

```
int
fbintr(numdevs, mb_devs, intclear)
    int     numdevs;
    struct  mb_device **mb_devs;
    int     (*intclear)();
```

`numdevs` is the maximum number of devices of these type configured. `mb_devs` is the array of devices descriptions. `intclear` is called back to actually turn off the interrupt for a particular device. `intclear` must have the same calling sequence as `cgoneintclear` (line cgone.7), i.e., it take the virtual address of the device to disable interrupts. `cg1_intclear` (line cgone.8) is a macro that actually disables the interrupts of `cg1fb`.

E.7.2. Open

When an open system call is made at the user level `cgoneopen` is called.

```
cgoneopen(dev, flag)
     dev_t dev;
{
     return(fbopen(dev, flag, NCGONE, cgoneinfo));
}
```

`cgoneopen` uses the utility `fbopen`.

```
int
fbopen(dev, flag, numdevs, mb_devs)
    dev_t   dev;
    int     flag, numdevs;
    struct  mb_device **mb_devs;
```

`fbopen` checks to see if `dev` is available for openning. If not the error ENXIO is returned. If `flag` doesn't ask for write position (FWRITE) then the error EINVAL is returned. Normally, zero is returned on a successful open.

E.7.3. Mmap

The memory map routine in a device driver is responsible for returning a single physical page number of a portion of a device.

```
/*ARGSUSED*/
cgonemmap(dev, off, prot)
     dev_t dev;
     off_t off;
     int prot;
{
     return(fbmmap(dev, off, prot, NCGONE, cgoneinfo, CG1SIZE));
}
```

`cgonemmap` used the utility `fbmmap`.

```
int
fbmmap(dev, off, prot, numdevs, mb_devs, size)
    dev_t   dev;
    off_t   off;
    int     prot, numdevs, size;
    struct  mb_device **mb_devs;
```

The parameters to `fbmmap` are similar to `fbopen`. However, `off` is the offset in bytes from the beginning of the device. `prot` is passed through but currently not used.

E.7.4. Ioctl

A pixrect kernel device driver must respond to two input/output control requests:

- FBIOGTYPE — Describe the characteristics of the pixel addressable device.
- FBIOGPIXRECT — Hand out a pixrect that may be used in the kernel. This ioctl call is made from within the kernel. This is only required of frame buffers.

```
#if NWIN > 0                                                          /* cgone.9*/
#define  CG1_OPS  &cg1_ops
struct    pixrectops cg1_ops = {
     cg1_rop,                                                         /*cgone.10*/
     cg1_putcolormap,
};
#else
#define  CG1_OPS  (struct pixrectops *)0
#endif

struct    cg1pr cgoneprdatadefault =
    { 0, 0, 255, 0, 0 };
struct    pixrect cgonepixrectdefault =
    { CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };

struct    pixrect cgonepixrect[NCGONE];                               /*cgone.11*/
struct    cg1pr cgoneprdata[NCGONE];

/*ARGSUSED*/
cgoneioctl(dev, cmd, data, flag)
     dev_t dev;
     caddr_t data;
{
     register int unit = minor(dev);

     switch (cmd) {
     case FBIOGTYPE: {
          register struct fbtype *fb = (struct fbtype *)data;

          fb->fb_type = FBTYPE_SUN1COLOR;
          fb->fb_height = CG1_HEIGHT;
          fb->fb_width = CG1_WIDTH;
          fb->fb_depth = 8;
          fb->fb_cmsize = 256;
          fb->fb_size = CG1_HEIGHT*CG1_WIDTH;
          break;
          }
     case FBIOGPIXRECT: {
          register struct fbpixrect *fbpr = (struct fbpixrect *)data;
          register struct cg1fb *cg1fb =
              (struct cg1fb *)cgoneinfo[(unit)]->md_addr;

          fbpr->fbpr_pixrect = &cgonepixrect[unit];                   /*cgone.12*/
          cgonepixrect[unit] = cgonepixrectdefault;                   /*cgone.13*/
          fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];/*cgone.14*/
          cgoneprdata[unit] = cgoneprdatadefault;                     /*cgone.15*/
          cgoneprdata[unit].cgpr_va = cg1fb;                          /*cgone.16*/

          cg1_setreg(cg1fb, CG_FUNCREG, CG_VIDEOENABLE);              /*cgone.17*/
          cg1_intclear(cg1fb);                                        /*cgone.18*/
          break;
          }

     default:
          return (ENOTTY);
     }
     return (0);
}
```

The SunWindows driver isn't configured into the system when NWIN = 0 (line cgone.9). When there is no SunWindows driver, don't reference the pixrect operations `cg1_rop` and `cg1_putcolormap`. The kernel version of `cg1_rop` (line cgone.10) only needs to be able to read and write memory pixrects for cursor management. Thus, you can

> `#ifndef KERNEL`
> *code not associated with reading and writing memory pixrects*
> `#endif KERNEL`

to reduce the size of the code.

Memory for pixrect public (struct `pixrect`) and private (struct `cg1pr`) objects is provided by arrays of each (line cgone.11) NCGONE long. A device `n` in these correspond to device `n` in `cgoneinfo`.

Lines cgone.12 through cgone.16 initialize a pixrect for a particular device. This ioctl call should enable video for a frame buffer (line cgone.17) and disable interrupts as well (line cgone.18).

E.7.5. Close

When the device is no longer being referenced, `cgoneclose` is called. All that is done is that the pixrect data structures of the device are zeroed.

```
cgoneclose(dev, flag)
        dev_t dev;
{
        register int unit = minor(dev);

        if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
                bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cg1pr));
                bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
        }
}

#endif
```

E.7.6. Plugging Your Driver into UNIX

You need to add the device driver procedures to `cdevsw` in `/sys/sun/conf.c` after assigning a new major device number to your driver:

```
#include "cgone.h"
#if NCGONE > 0
int  cgoneopen(), cgonemmap(), cgoneioctl();
int  cgoneclose();
#else
#define   cgoneopen  nodev
#define   cgonemmap  nodev
#define   cgoneioctl  nodev
#define   cgoneclose  nodev
#endif

        {
         cgoneopen,  cgoneclose,  nodev,  nodev,  /*14*/
         cgoneioctl,  nodev,  nodev,  0,
         seltrue,  cgonemmap,
        },
```

Also, you need to add the new files associated with your driver to `/sys/conf/files.sun`:

```
pixrect/cg1_colormap.c  optional cgone win device-driver
pixrect/cg1_rop.c                optional cgone win device-driver
sundev/cgone.c                   optional cgone device-driver
```

E.8. Access Utilities

This section describes utilities used by pixrect drivers. The pixrect header files `memvar.h`, `pixrect.h` and `pr_util.h` contain useful macros that you should familiarize yourself with; they are not documented here.

`pr_clip` modifies `src->pos`, `dst->pos` and `dst->size` so that all references are to valid bits.

```
pr_clip(dstp, srcp)
    struct  pr_subregion *dst;
    struct  pr_prpos *src;
```

`src->pr` may be NULL.

Two operations on operations, `reversesrc` and `reversedst`, are provided for adjusting the operation code to take into account video reversing of monochrome pixrects of either the source or the destination.

```
char        pr_reversedst[16];
char        pr_reversesrc[16];
```

These are implemented by table lookup in which the index into the tables is `(op>>1)&0xF` where `op` is the operation passed into pixrect public procedures. This process can be iterated, e.g., `pr_reversedst[pr_reversesrc[op]]`.

E.9. Rop

These are the major cases to be considered with the `pwo_rop` operation:

- Case 1 -- we are the source for the pixel rectangle operation, but not the destination. This is a pixel rectangle operation from the frame buffer to another kind of pixrect. If the destination is not memory, then we will go indirect by allocating a memory temporary, and then asking the destination to operate from there into itself.
- Case 2 -- writing to your frame buffer. This consists of 4 different cases depending on where the data is coming from: from nothing, from memory, from some other pixrect, and from the frame buffer itself. When the source is some other pixrect, other than memory, ask the other pixrect to read itself into temporary memory to make the problem easier.

E.10. Batchrop

A simple batchrop implementation could iterate on the batch items and call rop for each. Even in a more sophisticated implementation, while iterating on the batch items, you might also choose to bail out by calling rop when the source is skewed, or if clipping causes you to chop off in left-x direction.

E.11. Vector

There are some notable special cases that you should consider when drawing vectors:

- Handle length 1 or 2 vectors by just drawing endpoints.
- If vector is horizontal, use fast algorithm.
- If vector is vertical, use fast algorithm.

E.11.1. Importance of Proper Clipping

The hard part in vector drawing is clipping, which is done against the rectangle of the destination quickly and with proper interpolation so that the jaggies in the vectors are independent of clipping.

E.12. Colormap

Each color raster device has its own way of setting and getting the colormap.

E.12.1. Monochrome

For monochrome raster devices, when `pr_putcolormap` is called, the convention is that if `red[0]` is zero then the display is light on dark, otherwise dark on light. For monochrome raster devices, when `pr_getcolormap` is called, the convention is that if the display is light on dark then zero is stored in `red[0]`, `green[0]` and `blue[0]` and -1 is stored in other positions in the color map. Otherwise, if the display is dark on light, then zero and -1 are reversed.

E.13. Attributes

`pwo_getattributes` and `pwo_putattributes` operations get/set a bitplane mask in color pixrects.

E.13.1. Monochrome

Monochrome devices ignore `pr_putattribute` calls that are setting the bitplane mask. Monochrome devices always return 1 when `pr_getattribute` asking for the bitplane mask.

E.14. Pixel

`pwo_get` and `pwo_put` operations get/set a single pixel.

E.15. Stencil

In its most efficient implementation, stencil code parallels rop code, all the while considering the 2 dimensional stencil. One way to implement stencil is to use rops. We pay a small efficiency penalty for this. You may not consider writing the special purpose code worthwhile for the bitmap stencils since they probably won't get used nearly as much as rop. Here's the basic idea (Temp is a temporary memory pixrect):

```
Temp = Dest
Temp = Dest op Source
Temp = Temp & Stencil
Dest = Dest & ~Stencil
Dest = Dest | Temp

i.e., Dest = (Dest & ~Stencil) | ((Dest op Source) & Stencil)
```

Appendix F

Option Subwindow

NOTE: The option subwindow package is included in this release, but will ***not*** be included in future releases of SunWindows. We recommend that client programs instead use the panel subwindow package (see the chapter ***Panel Subwindow Package***). Appendix G, following, describes how to convert existing programs from the option subwindow package to the panel subwindow package.

An option subwindow (***optionsw***) presents a mouse-and-display-oriented user interface for setting parameters and invoking commands in an application program. It is the window system analog to entering command-line arguments and typing mnemonic commands to an application.

An option subwindow contains a number of items of various types, each of which corresponds to one parameter. Existing item types include labels, booleans, enumerated choices, text parameters, and command buttons.

The program `optiontool` is provided as a simple example of the features discussed here.

The declarations for the ***optionsw*** package are found in the header file `<suntool/optionsw.h>`. The file `<suntool/tool_hs.h>` can be included to provide the support header files for `optionsw.h`. `optionsw.h` includes declarations of all the public procedures, as well as the following structures and their associated defined constants. The first provides a counted buffer for a text item's value to be stored into:

```
struct  string_buf {
    u_int       limit;
    char        *data;
};
```

`data` should point to an array of chars to be used as the buffer, and `limit` should be set to the size of that buffer. Use of this structure is described with `optsw_getvalue` in ***Explicit Client Reading and Writing or Item Values*** below.

The second is used to identify the type as well as the value of a reference:

```
struct  typed_pair {
    u_int       type;
    caddr_t     value;
};

#define IM_GRAPHIC      2
#define IM_TEXT         3
#define IM_TEXTVEC      4
```

`type` indicates what kind of object `value` points to. The current choices are indicated in the following table.

Table F-1: Option Image Types

Type	Value Should Be
IM_GRAPHIC	(struct pixrect*)
IM_TEXT	(char *)
IM_TEXTVEC	(char **)

In the TEXTVEC case, `value` points to the first element of an array of string pointers; the last element of the array should be a NULL pointer. These are currently used only in enumerated items described in ***Enumerated Items***.

F.1. Option Subwindow Standard Procedures

This section describes the routines needed to conform to subwindow package norms. These routines follow the general procedures provided in ***Minimum Standard Subwindow Interface***.

```
struct  toolsw *optsw_createtoolsubwindow(tool, name, width, height)
    struct      tool *tool;
    char        *name;
    short       width, height;
```

creates an option subwindow within a `tool`. The handle `toolsw->ts_data` is used for the `optsw` argument in calls to other procedures of the ***optionsw*** package to identify the affected window and its private data. If the returned value is NULL then the operation failed. The remainder of this section is of interest only to clients outside the ***tool*** system.

In contexts other than a ***tool***, `optsw_init` must be called explicitly. Similarly, provisions must be made for using the rest of the routines in this section.

```
caddr_t optsw_init(fd)
    int         fd;
```

`optsw_init` takes an `fd` that identifies the window to be used for the optionsw, and returns an opaque pointer, which identifies the created optionsw in future calls to the package. If the returned value is NULL then the operation failed.

```
optsw_handlesigwinch(optsw)
    caddr_t     optsw;
```

is called to handle SIGWINCH signals. It repairs the damage to the window, and if the window has changed size, reformats the options as described below.

```
optsw_selected(optsw, ibits, obits, ebits, timer)
    caddr_t     optsw;
    int         *ibits, *obits, *ebits;
    struct      timevalue **timer;
```

is called to handle user inputs.

The cleanup routine for an ***optionsw*** is:

```
optsw_done(optsw)
    caddr_t     optsw;
```

It frees all storage allocated for the subwindow and its items. Of course, the client should not attempt to use any pointer associated with the optionsw or its items after a call to this routine.

F.2. Option Items

Once an *optionsw* is created, it may be populated with option items. Each item is created by a call to the create routine for the desired type; this creates the item, adds it to the items for the *optionsw*, and returns an item handle (an opaque pointer which identifies it).

In some general aspects, all items in the *optionsw* exhibit the same behavior. The left or middle mouse button indicates an item to be manipulated; the right button is left to the menu function. Pressing one of the first two buttons gets the *optionsw*'s attention, and releasing it actually completes a user-input event to which some item may respond. While the button is held down, the cursor may be slid around over the window, and each item it passes over will indicate its readiness to respond, typically by a reverse video display. Any such indication may be canceled simply by moving the cursor off the item before letting up on the button.

Each item is identified on the screen by a *label*, which may be either text or a picture provided by the client. This label is passed to the item creation routine in a `typed_pair` struct. In the graphic case (`type` == IM_GRAPHIC), the pixrect passed pointer is used without further consideration by the optionsw implementation — the client may even change the image after the item is created. For text labels (`type` == IM_TEXT), several defaults provide a uniform style with minimal client effort. Text labels are displayed in a bold-face version of the current font. (The current font for the option subwindow starts as the window's default font, and may be reset for each item, as described under `optsw_setfont` in *Miscellany* below.) The text of the label is modified to indicate the type of the item visually:

> Boolean items are surrounded by square brackets: "**[text]**"
>
> Commands are surrounded by parentheses: "**(text)**"
>
> Enumerated items have a colon appended to their label, and braces surrounding the set of their values: "**text:** { choice1 choice2 choice3 }"
>
> Text items have a colon appended to their label: "**text:** <value>"
>
> Label items have their exact text presented in the bold face: "**text**".

The text of the label is copied by the optionsw implementation; it may not be modified by the client after the item is created.

Clients which find these defaults too restrictive are free to generate their own labels (by using `pf_text` into a memory pixrect, for example) and pass them in as type IM_GRAPHIC.

F.2.1. Boolean Items

The following procedure creates an item which maintains a boolean (TRUE or FALSE) value:

```
caddr_t optsw_bool(optsw, label, init, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    int         init;
    int         (*notify)();
```

Its `label` contains a pointer to a `typed_pair` as described above. The label is displayed in reverse video whenever the item is TRUE. The value of the item is initially set to `init`, and is toggled whenever the user selects the item. (It may also be set by a call to **`optsw_setvalue`**, as described below.) Whenever user action changes the value of the item, the procedure `notify` is called with the new value, as described in ***Client Notification Procedures.*** This argument may be NULL to indicate that no notification is desired.

F.2.2. Command Items

The following procedure creates an item that invokes the client procedure **`notify`** when selected by the user:

```
caddr_t optsw_command(optsw, label, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    int         (*notify)();
```

The created item has no value. All three arguments are the same as their couterparts in `optsw_bool`.

F.2.3. Enumerated Items

The following procedure creates an item in which exactly one of a set of choices is in effect at any time:

```
caddr_t optsw_enum(optsw, label, choices, flags, init, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    struct      typed_pair *choices;
    int         flags;
    int         init;
    int         (*notify)();
```

The value is interpreted as a 0-based index into the choices for the selection. **`optsw`**, **`label`**, and `notify` are as above. **`choices`** is a vector of images to be displayed for the choices; for now its type must be TEXT_VEC. This means that the data pointer for **`choices`** addresses an array of string pointers, one for each possible choice plus a NULL indicating the end of the array. **`init`** is the initial value of the item; it should be at most the size of the **`choices`** array minus 2 (to avoid the null pointer which terminates the array). **`flags`** should be 0.

F.2.4. Label Items

The following procedure creates an item which does nothing but paint itself. This item type may be used to include labeling information in the option subwindow.

```
caddr_t optsw_label(optsw, label)
    caddr_t     optsw;
    struct      typed_pair *label;
```

`optsw` and `label` are as above.

F.2.5. Text Items

The following procedures create an item which holds a text value:

```
caddr_t optsw_text(optsw, label, default_value, flags, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    char        *default_value;
    int flags;
    int (*notify)();

    #define OPT_TEXTMASKED
```

`optsw`, `label`, and `notify` are as above. `default_value` is the initial value of the item. `flags` specify attributes of the created item; currently, only the **masked** attribute is supported. If OPT_TEXTMASKED in flags is set, each character of the text item will be displayed as an asterisk. This feature is useful for text parameters which should not be displayed, such as passwords. The true value of the item is returned by `optsw_getvalue` described below. `notify` is like the procedures of the other item-creation routines. It is called whenever the value of the text item is changed, except by a call to `optsw_setvalue`. Its arguments are handles for the optionsw and the item. `optsw_getvalue` should be used to actually retrieve the new value. This parameter to `optsw_text` may be NULL to indicate 'no notification.'

There may be multiple text items in an option subwindow. At any time, one of them "has the caret." Any keystrokes directed to the option subwindow will be directed to this item. The item that has the caret is indicated by a box around its label. Initially, this is the first text item created in the option subwindow. The user may set the caret in another item by clicking either the left or middle mouse button while the cursor is pointing at the new item's label.

The caret may also be determined and reset programmatically by calls to the following procedures:

```
caddr_t optsw_getcaret(optsw)
    caddr_t     optsw;
```

returns an item handle for the item that currently has the caret.

```
caddr_t optsw_setcaret(optsw, ip)
    caddr_t     optsw;
    caddr_t     ip;
```

sets the caret on the item indicated by `ip`, and returns `ip` if successful. Otherwise, it returns NULL. `ip` should be a handle on a text item.

Only displayable characters will be accepted in the item (ASCII codes 040–0176 inclusive). The user's erase (character delete) and kill (line delete) characters are available for editing existing text. The first will delete the last character of the text; the latter will delete the whole string. Other characters will be discarded.

Text items will expand to fit the remainder of their option subwindow's width. This may be more polymorphism than clients desire. See the discussion under ***Item Layout and Relocation*** below.

Note: This release of text items includes the following restrictions:

- Values of text parameters are restricted to a single line of text, less than 1000 characters long. Characters which extend beyond the item's right edge will not be displayed, although they are entered and edited the same as visible characters.

- Text items may be edited only at their ends. The available operations are: add a character to the end, delete a character from the end, and delete the whole value.

While significant extension to the functionality of text items is planned, the actual interface (the external procedure definitions and data structures) are designed to accommodate those extensions without change.

F.3. Item Layout and Relocation — SIGWINCH Handling

As each item is created, its width and height are determined and stored in the item's private data. No left and top positions are assigned at this time. Later, whenever a signal is received which indicates that the size of the subwindow has changed (in particular, when the tool is first displayed, and the size grows from 0 to the initial window), a layout procedure determines positions for all the items in the window.

The default layout procedure starts in the upper-left corner of the subwindow and places items in successive positions to the right, and then in successive rows down the window. Item positions are not normally fixed; items may be repositioned if the window is later laid out again with a different size.

If an item is encountered with either of its top or left edges fixed, that specification is accepted without further consideration — it is possible to lay one item down on top of a previously positioned item, or to position it out of sight to the right or below the subwindow boundary.

Positioning of subsequent items after an item with a fixed position may be affected in three ways:

1. The top of the row in which the item appears may move down, but not up, for the rest of the items in the row.
2. Subsequent items in the same row will not be positioned to the left of the item's right edge.
3. Items in subsequent rows will not be positioned above the bottom of the fixed item.

If an item is encountered which does not have fixed width (currently, only a text item), an attempt will be made to expand the item to fill the remaining width in the option subwindow. This is done through a rather simple-minded negotiation between the general layout procedure and the flexible item. If both the position and width of the item are flexible, the result of this negotiation may not be very satisfactory to observers. In most cases, the position, the width, or both should be fixed.

At any time between an item's creation and its destruction, the client may inquire or modify its current size and position. This is done via the following two procedures:

```
optsw_getplace(optsw, ip, place)
  caddr_t       optsw;
  caddr_t       ip;
  struct        item_place *place;

optsw_setplace(optsw, ip, place, reformat)
  caddr_t       optsw;
  caddr_t       ip;
  struct        item_place *place;
  int           reformat;
```

`optsw` is the handle returned by `optsw_init`. `ip` is the pointer to an `opt_item` struct returned by the item's create routine. `place` is a pointer to a `struct item_place` described below.

The `optsw_setplace` arguments are parallel to those of `optsw_getplace`. `place` is a pointer to a struct `item_place`, which contains a rect and four boolean flags indicating that a value is to be fixed for that item. The `reformat` argument indicates that the window is to be laid out and displayed anew, taking the changed item into account. This should generally be done any time after the window has been opened, since the item is already displayed, but it may be postponed if a series of adjustments are to be made; in that case, it is appropriate to reformat only after the last item's place is set.

The following struct is also described in `optionsw.h`:

```
struct  item_place  {
    struct      rect rect;
    struct      {
        x : 1;
        y : 1;
        w : 1;
        h : 1;
    }   fixed;
};
```

`rect` indicates the current size and position of the item, and the four bit-fields `fixed.x`, `fixed.y`, `fixed.w`, and `fixed.h` are TRUE if the corresponding dimension may not be adjusted by the layout procedure.

For convenience in laying out string items, two functions convert character columns and lines to the appropriate pixel coordinates:

```
int  optsw_coltox(optsw, col)
    caddr_t     optsw;
    int         col;

int  optsw_linetoy(optsw, line)
    caddr_t     optsw;
    int         line;
```

The dimensions used in calculating these coordinates are the width of the character 'a' in the optionsw's default font and the nominal height of that font, that is, the distance between baselines of successive unleaded lines of text. Both columns and rows start at 0.

F.4. Client Notification Procedures

Most item types provide a mechanism for notifying clients that the value of an item has been changed by the user. The same general mechanism is used to specify the procedure to be invoked in response to selection of a command button.

In each case, a pointer to a procedure is passed to the item-creation routine and stored with the item. This procedure pointer may be zero, in which case there is no client notification. When appropriate, this ***notification*** procedure is invoked by ***optionsw*** code with arguments to identify the affected subwindow and item, and the new value assigned to the item. The general form for these procedures is:

```
notify(optsw, item, value)
    caddr_t     optsw;
    caddr_t     item;
    int         value;
{ ... processing to respond to item's new value.}
```

Procedures to be invoked in response to a command button-push have the same form, except there is no `value` parameter. Notification of changes to text items also omit the `value` parameter.

Note that the notification procedure is ***provided by the client*** and ***invoked by the optionsw package.***

F.5. Explicit Client Reading and Writing of Item Values

Clients may read the current value of an item by calling the procedure:

```
int  optsw_getvalue(ip, dest)
    caddr_t     ip;
    caddr_t     dest;
```

`ip` is the item handle which identifies the item whose value is sought; `dest` is the address of the destination in which the value is to be stored. For items with a numeric value, `dest` should actually be a pointer to an `int`; the value will be stored in the indicated `int`, and returned as the value of the function. Items which have no value (commands, labels) store and return −1.

For text items, `dest` should be a pointer to a struct `string_buf`, whose `limit` is the length of the associated `data` array. `optsw_getvalue` will store characters from the value of the indicated item into `(*dest->data)`, and return the number of characters stored. If there is room, a terminating NULL character will be written, and a later call to `optsw_getvalue` will store characters starting at the beginning of the item's value. Otherwise, the data buffer will be filled and the returned count will be equal to `dest->limit`; the next call to `optsw_getvalue` for this item will resume storing characters with the first character not reported in the previous call. Multiple calls to `optsw_getvalue` may thus be used to retrieve a long value through a short buffer. Eventually, there will be room to store a null character, and the whole value will have been reported; the next call to `optsw_getvalue` for this item will restart at the beginning of the value.

Clients may set the value of an item by calling:

```
optsw_setvalue(optsw, ip, value)
    caddr_t     optsw;
    caddr_t     ip;
    caddr_t     value;
```

`optsw` is the opaque handle on the option subwindow; it enables repainting of the modified item. `ip` indicates the item to be modified, `value` should be an appropriate value for the item, which is then cast to `caddr_t`. That is, booleans and enumerateds should provide an **int** (or **unsigned**); text items should provide a **(char *)**. For example, if `optsw_setvalue` is being used to change a boolean item, `value` could be:

```
(caddr_t) FALSE
```

F.6. Miscellany

Clients may inquire and set the font that is being used for displaying item labels and values. Fonts for these objects are determined at the time the object is created; different items may use different fonts. Thus, the client may create an object, change the font, create more objects which will use the new font, and then change the font back (or to a third value) for succeeding items.

```
struct  pixfont *optsw_getfont(optsw)
        caddr_t      optsw;
```

returns the current font for the indicated `optsw`.

```
optsw_setfont(optsw, font)
        caddr_t      optsw;
        struct       pixfont *font;
```

sets the `optsw`'s font to be `font`.

Given an item in an optionsw, the routine:

```
optsw_nextitem(optsw, ip)
        caddr_t      optsw;
        caddr_t      ip;
```

returns a handle for the next item in sequence. If `ip` is NULL, the first item in the window will be returned; if `ip` refers to the last item in the optionsw, NULL is returned.

The routine:

```
optsw_removeitems(optsw, ip, count, reformat)
        caddr_t      optsw;
        caddr_t      ip;
        int          count;
        int          reformat:
```

removes at most `count` items from `optsw`, making them inaccessible to the user, but not destroying them. They may be restored later by a call to `optsw_restoreitems`. The subwindow is redisplayed without them if reformat is TRUE. The number of items so removed is returned; this may be less than count if the items in the subwindow are exhausted before `count` has been removed.

Starting at the item indicated by `ip`, the routine:

```
optsw_restoreitems(optsw, ip, count, reformat)
        caddr_t      optsw;
        caddr_t      ip;
        int          count;
        int          reformat:
```

restores at most `count` items in `osw` and returns the number restored. This may be left than `count` if all extant for the optionsw are exhausted, or an item which is not currently removed is encountered, first. The subwindow is redisplayed with the restored items if reformat is TRUE.

For assistance in implementing applications which use option subwindows, two routines are provided which print a formatted display of the optionsw and/or its items, to a stream of the client's choice:

```
optsw_dumpsw(stream, optsw, verbose)
    FILE        *stream;
    caddr_t     optsw;
    bool        verbose;

optsw_dumpitem(file, ip)
    FILE        *file;
    caddr_t     *ip;
```

For each procedure, the client says where to write the dump with the `stream` argument, and identifies the object to be dumped with the `optsw` or `ip` argument. If `verbose` is true, `optsw_dumpsw` will dump all the items of the optionsw.

Appendix A

Converting from Option Subwindow to Panel Subwindow

This appendix provides help in converting programs which were originally written using the optionsubwindow package (which will not appear in the next release of SunWindows) to the newer panel package. First, an outline of the steps involved in the conversion process is given. Then a simple program is presented in two versions: the first using the optionsubwindow package, the second using the panel package.

Here are the steps involved in converting from option subwindows to panels:

1. Header file to include:

   Use the include file `<panel.h>` instead of `<optionsw.h>`.

2. Tool creation:

   Use `tool_make()` instead of `tool_create()`.

3. Optionsubwindow/panel creation:

   Use `panel_create()` instead of `optsw_createtoolsubwindow()`.

4. Item creation:

   Use `panel_create_item()` for items of all types, instead of the type-specific routines `optsw_label()`, `optsw_bool()`, `optsw_text()`, `optsw_enum()` and `optsw_command()`.

   The arguments to the optsw item-creation routines (label, initial value, notify proc) are replaced by a list of attributes. The `typed_pair` structs required in the optsw routines as wrappers for strings and pixrect pointers are not needed in the panel package; instead you pass the string or pixrect pointer to the `panel_create_item()` directly, as part of the attribute list.

5. Item placement:

   To fix the location of an item, use the item placement attributes PANEL_ITEM_X and PANEL_ITEM_Y rather than `optsw_setplace()` with an item_place struct. If you want to specify the location `optsw_setplace()`, the `place` struct and `optsw_linetoy()` and `optsw_coltox()` are replaced by location attributes and the PANEL_CU() macro.

6. Notify proc parameters:

   The parameters passed to the notify procs differ in the two packages. Whereas the optionsubwindow handle (type `caddr_t`) is passed to optionsubwindow notify procs, the panel handle (type `Panel`) is NOT passed to panel notify procs. Also, in the panel package the input event is passed to the notify procs for all item types.

   Below are the parameters for the notify procs of each item type in the panel package. The types of the parameters are:

```
Panel_item         item;
int                value;
struct inputevent *event;

button_notify_proc  (item, event);
message_notify_proc (item, event);
text_notify_proc    (item, event);
choice_notify_proc  (item, value, event);
toggle_notify_proc  (item, value, event);
slider_notify_proc  (item, value, event);
```

If you need the panel in a notify proc, you have to get it from the item via the attribute PANEL_PARENT_PANEL, as in:

```
Panel panel;
panel = (Panel) panel_get(item, PANEL_PARENT_PANEL);
```

7. Reading and writing of item values:

   `optsw_getvalue()` and `optsw_setvalue()` can be replaced by `panel_get_value()` and `panel_set_value()`. The `string_buf` struct used in the optionsubwindow package for retrieving text values is no longer needed: `panel_get_value()` returns a pointer to the text value directly.

8. Setting and Retrieving fonts:

   The optionsubwindow routines `optsw_getfont()` and `optsw_setfont()` map onto the more general panel routines `panel_get()` and `panel_set()`. For example, to set a panel's font the two calls might read:

   ```
   optsw_setfont(optsw, font);

   panel_set(panel, PANEL_FONT, font);
   ```

9. Rendering items visible and invisible:

   In the optionsubwindow package, items are rendered invisible by calling `optsw_removeitems()`, and visible by calling `optsw_restoreitems()`. In the panel package, both of these states are achieved by calling `panel_set()`, with the appropriate value to the `PANEL_SHOW_ITEM` attribute. For example, to hide an item, the two calls might read:

   ```
   optsw_removeitems(optsw, item, 1, TRUE);

   panel_set(item, PANEL_SHOW_ITEM, FALSE);
   ```

   The calls to make the item visible again might read:

   ```
   optsw_restoreitems(optsw, item, 1, TRUE);

   panel_set(panel, PANEL_SHOW_ITEM, TRUE);
   ```

We now present a simple program first using the optionsubwindow package, and then modified to use the panel package.

The program creates a tool consisting of a single subwindow, which represents an extremely simple "voter registration form". There is a heading ("Please Enter Information"), a field for the user's name, and a "party affiliation" item allowing the user to choose between ***Democrat, republican*** and ***Independent***. Finally, there is an item labelled ***Quit***. When the user selects this item the function `quit_proc()` is called, which retrieves the current values of the name and party_affiliation items, and passes them to the function `store()`. The function `store()` is not given below; it simply represents the process of storing the information acquired through the optionsubwindow or panel.

The layout of the form is as follows:

```
------------------------------------------------------------------------
          Please Enter Information

 Name:
 Party Affiliation: Democrat  Republican  Independent
 Quit
------------------------------------------------------------------------
```

First, the option subwindow version of the program:

```
#include <suntool/tool_hs.h>
#include <suntool/optionsw.h>

static struct tool   *tool;
static char          *tool_name = "Voting Registration Form";
static struct toolsw *tsw;
static caddr_t        osw;

static caddr_t title_item;
static caddr_t name_item;
static caddr_t party_item;
static caddr_t quit_item;

static int quit_notify_proc();

static struct typed_pair title_label    = {IM_TEXT, "Please Enter Information"
static struct typed_pair name_label     = {IM_TEXT, "Name");
static struct typed_pair party_label    = {IM_TEXT, "Party Affiliation");
static char *choice_values[]            = { "Democrat",
                                            "Republican",
                                            "Independent" };
static struct typed_pair party_choices = {IM_TEXTVEC, (caddr_t)choice_values)
static struct typed_pair quit_label     = {IM_TEXT,"Quit");

main()

   struct item_place place;

   /* create the tool and the optionsubwindow */
   tool = tool_create(tool_name, TOOL_NAMESTRIPE, NULL, NULL);
   tsw = optsw_createtoolsubwindow(tool,"optsw",
                                  TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
   osw = tsw->ts_data;

   /* create the items */
   title_item = optsw_label(osw, &title_label);
   name_item  = optsw_text(osw, &name_label, "John Q. Public", 0, NULL);
   party_item = optsw_enum(osw, &party_label, & party_choices, 0, 0, NULL);
   quit_item  = optsw_command(osw, &quit_label, quit_proc);

   /* now fix the locations of all the items */
   place.fixed.x = palce.fixed.y = TRUE;
   place.rect.r_left = optsw_coltox(osw, 10);
   place.rect.r_top  = optsw_linetoy(osw, 1);
   optsw_setplace(osw, title_item, &place, FALSE);

   place.rect.r_left = optsw_coltox(osw, 1);
   place.rect.r_top  = optsw_linetoy(osw, 3);
   optsw_setplace(osw, name_item, &place, FALSE);

   place.rect.r_left = optsw_coltox(osw, 30);
   place.rect.r_top  = optsw_linetoy(osw, 3);
```

```
    optsw_setplace(osw, party_item, &place, FALSE);

    place.rect.r_left = optsw_coltox(osw, 1);
    place.rect.r_top  = optsw_linetoy(osw, 4);
    optsw_setplace(osw, quit_item, &place, FALSE);

    signal(SIGWINCH, sigwinched);
    tool_install(tool);
    tool_select(tool, 0);
    tool_destroy(tool);
    exit(0);
}

static
sigwinched()
{
    tool_sigwinch(tool);
}

static
quit_notify_proc()
{
    int party_value;
    char[MAX_NAME_LENGTH] buffer;
    struct string_buf name_buf;

    name_buf.limit = MAX_NAME_LENGTH;
    name_buf.data  = buffer;

    optsw_getvalue(party_item, &party_value);
    optsw_getvalue(name_item, &name_buf);
    store(party_value, name_buf.data);
}
```

The Panel Package version of the Voter Form program follows.

```
#include <suntool/tool_hs.h>
#include <suntool/panel.h>

static struct tool   *tool;
static char          *name = "Voting Registration Form";
static struct toolsw *panel_sw;
static Panel          Panel;

static caddr_t title_item;
static caddr_t name_item;
static caddr_t party_item;
static caddr_t quit_item;

static int quit_notify_proc();

#define MAX_NAME_LENGTH         20

main()

   /* create the tool and the panel */
   tool     = tool_make(WIN_NAME_STRIPE, TRUE,
                        WIN_LABEL,       tool_name,
                        0);
   panel_sw = panel_create(tool, 0);
   panel    = (Panel) panel_sw->ts_data;

   /* create the items */
   title_item = panel_create_item(panel, PANEL_MESSAGE,
                PANEL_ITEM_X,       PANEL_CU(10),
                PANEL_ITEM_Y,       PANEL_CU(1),
                PANEL_LABEL_STRING, "Please Enter Information",
                0);

   name_item  = panel_create_item(panel, PANEL_TEXT,
                PANEL_ITEM_X,       PANEL_CU(1),
                PANEL_ITEM_Y,       PANEL_CU(3),
                PANEL_LABEL_STRING,        "Name:",
                PANEL_VALUE_STORED_LENGTH, MAX_NAME_LENGTH,
                PANEL_VALUE,               "John Q. Public",
                0);

   party_item = panel_create_item(panel, PANEL_CHOICE,
                PANEL_ITEM_X,       PANEL_CU(30),
                PANEL_ITEM_Y,       PANEL_CU(3),
                PANEL_LABEL_STRING,  "Party Affiliation:",
                PANEL_CHOICE_STRINGS, "Democrat",
                                      "Republican",
                                      "Independent",
                                      0,
                0);

   quit_item  = panel_create_item(panel, PANEL_BUTTON,
```

```
                    PANEL_ITEM_X,          PANEL_CU(1),
                    PANEL_ITEM_Y,          PANEL_CU(4),
                    PANEL_LABEL_STRING,     "Quit",
                    PANEL_NOTIFY_PROC,      quit_notify_proc,
                    0);

        signal(SIGWINCH, sigwinched);
        tool_install(tool);
        tool_select(tool, 0);
        tool_destroy(tool);
        exit(0);
}

static
sigwinched()
{
        tool_sigwinch(tool);
}

static
quit_notify_proc(panel, item)
Panel      panel;
Panel_item item;
{
        int party_value;
        char[MAX_NAME_LENGTH] name_buf;

        party_value = (int) panel_get_value(party_item);
        strcpy(name_buf,(char *) panel_get_value(name_item));
        store(party_value, name_buf);
}
```

Index