Part Number 800-1092-01
Revision: C of 1 November 1983
For: Sun System Release 1.0

Programmer's Reference Manual

for the

Sun Window System

Sun Microsystems, Inc.,
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

3500

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.

Table of Contents

Revision History

Rev	Date	Comments
A	15 July 1983	Preliminary draft release 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.

1. INTRODUCTION

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

- Provision of a processor for each user is a prerequisite for powerful, responsive, cost-effective systems.
- The bit-mapped display in the Sun allows arbitrary fonts and graphics to be used freely to make applications programs easier to learn and use.
- In addition, Sun's RasterOp hardware supports fast and convenient manipulation of image data.
- The mouse pointing device can be used to select operations from menus or to point at text, graphics and icons. Similarly, objects to be operated on can be specified directly and conveniently.
- The up-down encoded keyboard supports sophisticated function-key interfaces, at once simpler and more effective than most command languages.

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

- A uniform interface is provided to varied pixel-oriented devices, which allows convenient incorporation of new devices into the system, and clean access to all these devices by applications programs.
- Device independence is extended on input to user-interface features such as function keys and locators.
- A window management facility keeps track of multiple overlapping windows, allowing their creation and rearrangement at will. It 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.
- Built on all these facilities, an executive and application environment 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.

1.1. Design Goals

The Sun Window System is a ***tool box and parts kit***, rather than 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 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. Layers of Implementation

There are three broad divisions of the window system, reflected in the the structure of this manual. These layers may be identified by the libraries which contain their implementations:

1. The *pixrect* level provides a device-independent interface to pixel operations.
2. The *sunwindow* level implements a manager for overlapping windows, including imaging control, creation and manipulation of windows, and distribution of user inputs.
3. The *suntools* 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*.

1.2.1. Pixrects

Chapter 2 describes the *pixrect* layer of the system. This level generalizes the RasterOP features of the hardware 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 drivers; these 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.

Careful attention is paid to the balance between functionality and efficiency. All pixrects support the same set of operations on their contents. These 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. Where hardware support exists, it is taken advantage of, without sacrificing generality.

All pixrects will 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.

1.2.2. Sunwindows

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 existing concurrently may all access the 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 that one window does not interfere with another's image. One such conflict handled by *sunwindow* (although not the only one) 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, Sun's window system 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.

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.

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 old-fashioned terminal-based programs to run within windows which will handle mouse interactions for them.

1.2.3. Suntools

Chapters 6 through 8 of this manual describe the *suntools* level of the system, which provides an actual user interface.

We refer to an application program which 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 (typically) several windows through which the tool interacts with the user, often including a special *icon*, which is a small form the tool may take to be unobtrusive but still identifiable. Simple examples of tools include a calculator, a bitmap editor, and a terminal emulator. Sun provides a few tools ready-built (several are illustrated in Appendix B), and more will be provided as time passes; customers are expected to provide more to suit their needs.

Some common components of tools are provided by the window system:

- An executive framework which supplies the usual "main loop" of a program, and serves to coordinate the activities of the various subwindows;
- A standard *tool window*, which 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, encompassing adjustment of its size and position (including layering), and inter-subwindow boundary movement;
- Several commonly-used *subwindow* types, which can be instantiated in the tool;
- A standard scheme for laying out those subwindows; and
- A facility which 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;
- The user interface for leaving the window system.

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

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

1.3. Intended Audience and Scope of This Manual

This document is intended for programmers of applications which use window system facilities. It is not intended as a user guide, nor as documentation of the internals of the window system. The user documentation is provided in manual pages for the window system, and for the particular application programs.

This document is primarily a reference manual. However, since no standard references are yet available on window systems, some tutorial information is included. The material in the manual is presented in a roughly bottom-up fashion, with primitive concepts and facilities presented first. This approach minimizes forward references, at the possible cost of leaving what many readers may consider the most interesting material to the end. The manual is not intended to be read linearly front-to-back; an early look at the chapters on tools may serve to motivate much of the rest. Where early sections descend into uninteresting detail, readers are encouraged to skip ahead, and to return only as the need becomes apparent.

1.4. Concepts and Terminology

A few terms are used in this manual with special meanings distinct from their common usage, or to introduce concepts which are specific to this window system. We discuss the most important here.

We use the word *client* to indicate a program which uses facilities of the window system. This is in contrast to *user*, which generally 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 (e.g. Foley and Van Dam, *Fundamentals of Interactive Computer Graphics*).

The position of the mouse is indicated by a *cursor* on the screen; this is any small image which is moved 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; often the two concepts are not distinguished. We wish to keep these concepts distinct; if we need to refer to the type-in location, we will use the term *caret*.)

A *menu* is a list of related choice items displayed on the screen in response to a user mouse-action, and in which the user chooses one 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.

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, which typically displays some identifying image rather than a portion of its contents; it is frequently used for dormant application programs.

2. PIXEL DATA AND OPERATIONS

This section discusses pixel data and operations in the lowest-level output facilities of the Sun window system. These facilities will frequently be accessed indirectly, through higher-level abstractions described in chapters 3 through 8. However, some client implementors will deal at this level, for instance to include new display devices in the window system. The header file */usr/include/pixrect/pixrect_hs.h* includes most of the header files that you need to work at this level of the window system.

2.1. Pixrects

The fundamental object of pixel manipulation in the window system is the *pixrect.* Pixrects are designed along the model of *objects* in an object-oriented programming system such as Smalltalk or CLU. They combine both data and operations, presenting their clients with a simple interface in which a well-defined set of operations produce 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.

A pixrect encapsulates a rectangular array of pixels along with the operations which are defined on that data. The pixrect facility is designed to satisfy two broad objectives:

> A *uniform interface to a variety of devices* provides 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 (how many bits in each, how they are aligned, and how interpreted).
>
> A proper *balance of functionality and efficiency* allows provision of 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. Where specialized hardware exists and can be used for a particular operation, it is; but the generality of operations available is not sacrificed to peculiarities of the hardware.

2.1.1. Pixel Interpretation

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

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 *color map*, 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. A pixrect may be any depth up to 32 bits; it need only specify a large enough color map to cover all possible values (2**n entries for an *n*-bit pixel).

2.1.2. Geometry Structs

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.

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

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

defines a position (x and y coordinates).

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

defines the dimensions (width and height) of an area. Thus, the coordinates of a pixel in a pixrect are integers between 0 and the pixrect's width (or height) – 1.

Leaving a pixrect undefined for the moment,

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

defines a point within a pixrect: it contains a pointer to the pixrect, and a position within it.

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

defines a sub-area within a pixrect: it contains a pointer to the pixrect, an origin for the area, and its width and height.

2.1.3. The Pixrect Struct

The pixrect struct implements a rectangular array of pixels, along with the operations defined on it:

```
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)  ((pr)->pr_size.x)
#define     pr_height(pr) ((pr)->pr_size.y)
```

All other information about the pixrect is data private to it, accessed through the operations addressed in the *pr_ops* struct. In particular, the locations, values, and interpretations of pixels, if stored at all, are reached through *pr_data*.

Memory pixrects have their pixels stored in memory, in a public format. This gives them a privileged position: they are the common representation to and from which all other pixrects may be converted. Thus pixrects of any two kinds may be interconverted by passing through an intermediate memory representation. Memory pixrects are described in section 2.4.

2.1.4. Primary and Secondary Pixrects

More than one pixrect may refer to the same pixels. One pixrect is considered to "manage" the pixels, and is referred to as the *primary* pixrect. For devices that share a resource with many processes, such as the framebuffer, a primary pixrect might be responsible for mapping the shared resource into a process address space. For device that don't share a resource with other processes, such as a memory pixrect, a primary pixrect might be responsible for allocating the single resource. This includes other objects in the same process.

A *secondary* pixrect is formed from a subregion of either a primary or another secondary pixrect. It provides finer clipping control when writing than is possible with a primary pixrect. It also indicates a limited source of pixels when reading.

2.2. Pixrectops

A pixrectops struct is a collection of pointers to procedures, one for each of the operations defined on all pixrects:

```
struct pixrectops {
    int         (*pro_rop)();
    int         (*pro_batchrop)();
    struct      pixrect *(*pro_create)();
    int         (*pro_destroy)();
    int         (*pro_get)();
    int         (*pro_put)();
    int         (*pro_vector)();
    struct      pixrect *(*pro_region)();
};
```

With the exception of the *create* and *region* operations, each procedure returns an int; by convention, a return value of -1 indicates an error. (*Create* and *region* return NULL in case of error.)

Every pixrect is expected to provide each of these operations. Of course the implementations will vary from device to device, but in describing the individual operations, we are interested in their external behavior, which should be the same across the various instantiations. Therefore, in the following sections, we will describe a model procedure, which differs in at least name from every implementation. This bears repeating: *The pixrect-op described at the top of each succeeding subsection is a convenient fiction; there is no single procedure with that name.*

These routines will be used in fairly standard fashions, so macros are defined for more convenient invocation in the expected forms. These macros are described in the same section as the routine they apply to.

In many of the macros which expand to subroutine calls, the argument(s) which point to a pixrect are used several times int he macro expansion. Thus it will be cheaper to use register variables as these arguments. Even though it only appears once in your source, it may get used four or five times in the generated code.

2.2.1. Create/Open

The properties of a non-memory pixrect are described by a 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 only *devicename* currently supported is */dev/bw0.* The returned pixrect covers the entire surface of the device.

Memory pixrects are special and have their own create routine.

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

Create routines for a pixrect may also be found in an existing pixrect of the desired type. Macros are defined to invoke Create given an existing pixrect; these will make a new instance of an existing pixrect type.

```
struct   pixrect *pr_create(pr, w, h)
        struct pixrect *pr;
        int     w, h;
```

extracts the create procedure from *pr*'s pixrectops, and applies it to the list of arguments;

```
struct   pixrect *prs_create(pr, size)
        struct pixrect *pr;
        struct pr_size size;
```

does the same, expanding its *pr_size* argument in the process.

2.2.2. Destroy/Close

```
pr_destroy(pr)
   struct      pixrect *pr;
```

destroys the pixrect pointed to by *pr*, freeing any storage allocated for it. The procedure 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.

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

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

does the same. (These are both defined so that either set of names may be used consistently across all operations.)

2.2.3. Raster Op

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

Rop performs the indicated raster operation from coordinates (*sx, sy*) in the source pixrect *spr*, into the rectangle whose origin is (*dx, dy*) and whose dimensions are *dw* and *dh* in the destination pixrect *dpr*.

The interpretation of the *op* argument is discussed in section 2.2.5, below.

If the source pointer is NULL, it is taken to indicate an infinite source of pixels all 0.

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

expands its *pr_subregion* and *pr_prpos* arguments in the process.

2.2.4. Batch Raster Op

Some applications (e.g. 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:

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

struct batchitem {
   struct     pixrect *bi_pr;
   struct     pr_pos  bi_pos;
};
```

Batchrop takes an array of items addressed by the argument *items*, each consisting of a pointer to a source pixrect, and a destination x- and y-offset; the size of the array is indicated by the argument *n*. It performs the raster operation indicated by *op* on each source pixrect, carrying it into a destination pixrect addressed by *dpr*. The whole of each source pixrect is used (unless it needs to be clipped to fit the destination pixrect).

The destination position is initialized to the position given by *dx* and *dy*; then, for each batchitem, the offsets given in *bi_pos* are added to the previous destination position before that *bi_pr* is operated on. That is, a destination position is updated for each item in the batch, and these adjustments are cumulative. (A *batchitem* is not simply a *pr_prpos* to emphasize the fact that the position is *not in the indicated pixrect.*)

The interpretation of the *op* argument is discussed in section 2.2.5, below.

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

expands its destination *pr_prpos* argument.

2.2.5. Constructing Op Arguments for Rop and Batchrop

The rop and batchrop procedures in the pixrectops vector perform raster operations on their source and destination pixrects, with the particular operation specified in the *op* argument to the procedure. A raster operation is one in which each destination pixel is computed as the result of a function of its previous value and of the value of a corresponding source pixel.

For a depth-1 pixrect, such as a black and white display, there are only 16 such functions logically possible, so the operation may be encoded in four bits of the *op* argument. (See the *SUN Programmer's Manual* for a fuller discussion of raster ops.) The four bits used leave one more bit at the low-order end of the value for the clipping flag discussed in *Controlling Clipping in the Raster Op;* thus raster ops are encoded by even values 0 through 30.

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.

2.2.5.1. Constants for Constructing Raster Ops

The encoding of the operation into 4 bits is supported by the following defined constants:

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

These allow the functions to be specified by performing the desired logical operation on the appropriate constant. The explicit definition of PIX_NOT is required since other bits of the operation should not be inverted. When PIX_NOT is used, the full *op* must be specified before setting the PIX_DONTCLIP flag described in *Controlling Clipping in the Raster Op.*

Op with Value	Result
PIX_SRC	Same as source argument (write)
PIX_DST	Same as destination argument (no-op)
PIX_SRC | PIX_DST	OR of source and destination (paint)
PIX_SRC & PIX_DST	AND of source and destination (mask)
PIX_NOT(PIX_SRC) & PIX_DST	AND of negation of source with destination (erase)
PIX_SRC ^ PIX_DST	XOR of source and destination (invert)

A particular application of these logical operations allows definition of *set* and *clear* operations:

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

is always true, and hence sets the result; while

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

is always false, and hence clears the result.

2.2.5.2. Controlling Clipping in the Raster Op

Clipping pixrects in transmission can be very expensive, and often 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 advise *rop* and *batchrop* procedures to bypass clipping checks, thus considerably speeding their operation. This is done by setting the flag

```
#define PIX_DONTCLIP     0x1
```

in the *op* argument.

2.2.5.3. Provision for Inverted Video Pixrects

Two arrays of opcodes provide translation for environments where either the source or the destination data may be inverted:

```
char  pr_reversedst[32],
      pr_reversesrc[32];
```

Indexing the array by an integer less than 32 (*op* plus clipping flag) yields the corresponding *op* and an unchanged clipping flag for the inverted pixrect. The translation is transitive; that is, one may write

```
pr_reversesrc[PIX_SRC]
```

to copy pixels from an inverted source, and

```
pr_reversedst[pr_reversesrc[PIX_SRC & PIX_DST]]
```

to apply a mask from an inverted source to an inverted destination.

2.2.6. Get

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

returns the pixel value at coordinates (x, y) in pixrect pr as a 32-bit unsigned result.

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

expands its *srcprpos* argument.

2.2.7. Put

```
pr_put(pr, x, y, val)
   struct      pixrect *pr;
   int         x, y, val;
```

stores the indicated pixel value at coordinates (x, y) in *pr*.

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

expands its *dstprpos* argument.

2.2.8. Vector

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

draws a vector of unit width between two points in the indicated pixrect, using the indicated operation, and color as a constant source; portions of the vector lying outside the pixrect are clipped.

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

expands its *pos0* and *pos1* arguments in the process.

2.2.9. Region

```
struct pixrect pr_*region(pr, x, y, w, h)
   struct      pixrect *pr;
   int         x, y, w, h;
```

creates a new (secondary) pixrect which refers to an area within an existing one. The pixrect which is to serve as the source is addressed by *pr*; the rectangle to be include in the new pixrect is described by *x, y, w* and *h*, with its upper left pixel at coordinates (0, 0), as with all pixrects.

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

expands its *subreg* argument in the process.

2.3. Independent Procedures

2.3.1. Replicating the Source Pixrect

Often the source for a raster operation consists of a pattern which is used repeatedly, even a single value which fills the whole destination. In these cases, it is easier (and usually faster) to specify the source programmatically than to create a special source pixrect of the exact size to fill the destination. This is done with the *pr_replrop* procedure, declared in *pixrect.h*. This procedure is the same for all pixrects, and is referenced directly, rather than through the pixrectops vector:

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

Dpr indicates the destination pixrect, and the area affected is described by the rectangle (*dx, dy, dw, dh*). The source pixrect is indicated by *spr*, and an origin within it by *sx, sy*.

For instance, to write a solid color in a rectangular region, it suffices to give a single pixel of source, which is replicated throughout the destination. (For monochrome pixrects, this will be equivalent to either PIX_SET or PIX_CLR in the *op* argument to rop.) Similarly, 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 using it as the source pixrect.

The alignment of the pattern on the destination is the same as though the source were a pixrect containing an infinite number of copies of the actual source, both across and down, in which the actual source origin is taken at the position indicated in *spr*. So, if *sx* and *sy* are both 0, the pattern will be aligned with the destination pixrect position. If, on the other hand, what is desired is alignment to some global framework (e.g. to keep a gray pattern aligned across all of a screen), the destination's origin may be copied into the source positions to maintain that alignment.

2.4. Memory Pixrects

Pixrects in memory hold a special position: their format is public, as are the standard operations on them. Thus, a client may construct a memory pixrect using non-pixrect operations, and has access to the memory pixrect operations at all times.

In memory, the pixel stored at the lowest address is the upper-left corner; it 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 that are stored, and so through the whole rectangle.

The declarations required for dealing with memory pixrects are found in the file *memvar.h.*

2.4.1. The Mem_ops Struct

The procedures pointed to by a memory pixrect's *pr_ops* vector are available in a struct named predictably:

```
struct pixrectops mem_ops = {
    mem_rop,
    mem_batchrop,
    mem_create,
    mem_destroy,
    mem_get,
    mem_put,
    mem_vector,
    mem_region
};
```

The procedure declarations are completely consistent with the models described above.

2.4.2. The Mpr_data Struct

The *pr_data* element of a memory pixrect points to an *mpr_data* struct:

```
struct mpr_data {
    int        md_linebytes;
    short      *md_image;
    struct     pr_pos md_offset;
    int        md_primary;
}
```

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 multiple pixrects may include subregions of the same pixels, this quantity cannot be computed from the width of the pixrect — see *Region.*) *Md_image* is the address of the first pixel in the underlying array; it is guaranteed to lie at the beginning of a short word (16 bits). *Md_pos_offset* is the position of the first pixel of this pixrect within the array of pixels addressed by *md_image*. *Md_primary* is 1 if the pixrect is primary and had its image allocated by *calloc*. 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 memory will be freed.

2.4.3. Static Memory Pixrects

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

```
#define  mpr_static(name, w, h, d, image)
```

where *name* is a token to be used in identifying the generated data objects; *w*, *h*, and *d* 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 which contains the pixel values in the correct format.

The macro generates two static 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.

2.5. 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 display format for fonts, with routines for processing them; and routines which take a string of text and a font, and handle various parts of painting that string in a pixrect.

2.5.1. Pixfonts

The file */usr/include/pixrect/pixfont.h* defines the following two structs:

```
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 stored in a pixrect (a separate pixrect for each character) addressed by *pc_pr*. Characters that have no displayable image will have NULL in their entry in *pr_char*. *Pc_home* is the origin of that image (its upper left corner) relative to the character origin. (Characters are normally placed relative to a *baseline*, which is the lowest point on characters without descenders. 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 character origin is changed by this character; that is, the amounts in *pc_adv* added to the current 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 consists of a size used for fonts whose characters are all the same size, and an array of pixchars, indexed by the character code.

2.5.2. Operations on Pixfonts

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

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

opens the file with the given name, which 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.

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

performs the same function for the system default font, a fixed-pitch, 16-point sans serif font with upper-case letters 12 pixels high.

When a process is finished with a font, it should call *pf_close*, to free the memory associated with it. If the environment parameter DEFAULT_FONT is set, it overrides the font file opened by **pf_default*.

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

Note: The external font format is expected to change soon.

2.5.3. Pf_text

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;
```

Where is the destination for the start of the text (baseline, nominal left edge); *op* is the raster operation to be used in writing the text, as described in *Constructing Op Arguments for Rop and Batchrop*; *font* is a pointer to the font in which the text is to be displayed, and *text* is the actual string to be displayed.

Auxiliary procedures used with *pf_text* include

```
struct pr_size pf_textbatch(where, lengthp, font, text)
    struct      batchitem     *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 fills in the array of batchitems and its length, as required by batchrop (see 2.2.4). To do this, it requires a string and a font in which to find source pixrects for the characters.

Pf_textwidth returns a pr_size which contains the total dimension of the string of the first *len* characters in text, when formatted in the indicated font.

3. OVERLAPPED WINDOWS: IMAGING FACILITIES

This chapter and the following two deal with the next layer of the window system, which provides facilities for managing windows with overlapping 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. This layer of the window system is referred to as the *sunwindow* level, from the name of the library which 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; the Sun window system 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. Most of the window system facilities are useful in this case as well.

3.1. Window Issues: Controlled Display Generation

Windows that overlap introduce two new issues, which may be broadly characterized as preventing the window from painting where or when it shouldn't, and 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. 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:

- The raster hardware typically requires 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.
- Finally, a software cursor which is not controlled by the window process (the kernel is usually responsible for the cursor) may have to be removed so that it does not interfere with the window's image.

3.1.2. Damage Repair and Fixups

A window whose image does not all appear 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 (e.g. 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. Fixups are provoked only in response to an action of the window's process, however.

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 required considerable computation, Sun's window system 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 *Pixwins.*

3.1.4. Process Structure

In Sun's window system, access to the screen is performed in the user process. This raises the possibility of a user process running amok and damaging not only its own data, but (at least the display of) other application processes as well. Several compensating factors justify this approach:

- It is consistent with the philosophy espoused in Chapter 1, of providing an open system which clients may access 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.
- The vast majority of clients will use window system library routines which provide protection.

3.1.5. Imaging with Windows

We proceed now to a detailed discussion of imaging with windows. We begin with a description of the basic data structures which are used in this level of the Sun window system: a primitive geometric facility (the *rect*) for describing rectangles, and the basic structure which is used to describe a window on the screen (the *pixwin*), with its associated state and operation vectors.

Creating and destroying *pixwins* is a simple process, which gets a brief discussion.

That is followed by a detailed description of the approach to locking and clipping, which leads naturally into a discussion of library routines which access a *pixwin's* pixels. Detecting and repairing damage is treated next.

3.1.6. Libraries and Header Files

The procedures described in this chapter are provided in the *sunwindow* library (*/usr/lib/libsunwindow.a*). The header file */usr/include/sunwindow/window_hs.h* contains all the *includes* that are required by a program using the facilities described in this chapter.

3.2. Data Structures

3.2.1. Rects

Throughout the window system, 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 */usr/include/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 */usr/include/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* may be found 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*:

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

struct          rect     *rp;
```

These 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. 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; and 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 */usr/include/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;
};
```

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 in screen access macros to call either the *pixwin* software level or (as an optimization) 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 used by screen access macros. These three fields are dynamically altered based on locking and clipping status.

Pw_clipdata is a collection of information of special interest to 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 given here, and then discussed more fully in subsequent sections.

3.2.3. Pixwin_clipdata

```
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];
};

#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. (The interplay between windows and *pixwins* is a continuing story, which picks up again in section 3.3.) The portions of the window's area accessible through the *pixwin* is described by the *pwcd_clipping rectlist*. *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 section 3.4.1 below. Copies of this *pwcd_clipping*, sorted in directions convenient for copy operations, are stored in *pwcd_clippingsorted*.

Pwcd_state can be one of PWCD_NULL (no part of window visible), PWCD_MULTIRECTS (must clip to multiple rectangles), PWCD_SINGLERECT (need clip to only one rectangle) or PWCD_USERDEFINE (the client program will be responsible for setting up the clipping). *Pwcd_prmulti* is the *pixrect* that is used for drawing when there are multiple rectangles involved in the clipping. *Pwcd_prsingle* is the *pixrect* that is used for clipping 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 in order 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 pixwin_clipops {
    int         (*pwco_lock)(),
    int         (*pwco_unlock)(),
    int         (*pwco_reset)(),
    int         (*pwco_getclipping)();
};
```

The *pixwin_ops* struct is a vector of pointers to system-provided procedures which implement correct screen access. These are accessed through macros described in the section entitled *Locking and Clipping.*

3.3. Pixwin Creation and Destruction

In order to create a *pixwin* the window to which it will refer must already be created. This task is accomplished with procedures like *win_getnewwindow* and *win_setrect,* described in the next chapter, 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.

3.4. Locking and Clipping

Before a window process 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 are 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 section 3.5. Some clients will use those routines, but, for efficiency's sake, lock explicitly around a body of screen access operations.

3.4.1. Locking

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

is a macro which 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) which bounds the area to be affected. *Pw_lock* blocks if the lock is unavailable (e.g. 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.

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

is a similar macro, 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 *unlocks*. To handle this situation cleanly, another routine is provided:

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

is a macro which unlocks *pw* until its lockcount has gone to 0. Like *pw_lock* and *pw_unlock*, it 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 window'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 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.

Locking is both a) moderately expensive (it involves two system calls), and b) capable of impacting other processes. Clients with a recognizable 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 might be a line of text, or a series of vectors which have all been computed.

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

- do any significant computation unrelated to displaying its image; and
- invoke any system calls (including other I/O), which might cause it to block.

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 via signal. The idea is that a process shouldn't be aborted for this infraction. However, the display may look bad after this action.

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 area it is valid for this window to 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 section 3.6 below. *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 then 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, modulo locking, 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* set up 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 clobber another window's image.

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 section 3.4.1; 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 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 section 2.2.5); 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.

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

draws a pixel of *value* at (x, y) in the addressed *pixwin*.

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

draws a vector of pixel *value* from (x0, y0) to (x1, y1) in the addressed *pixrect*, using rasterop *op*.

```
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*.)

```
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;
```

These two routines 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, so it is possible for a character to have some pixels to the left of the x-coordinate; and the baseline is the y-coordinate of the lowest pixel of characters without descenders (e.g. 'L', 'o'), so pixels will frequently occur both above and below the baseline in a string.

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.

3.5.2. 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 *pixwin's* will always return *rl_null* in *pw_fixup* because the image is refreshed from *pw_prretained.*

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

returns the value of the pixel at (x, y) in the addressed pixrect.

```
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 *pixwin* pointed to by *pw*, in the rectangle defined by *xs, ys, width, height*, using rasterop function *op*; they are stored in the rectangle with its origin at *xd, yd* in the *pixrect* pointed to by *pr*.

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

Copy is used when both source and destination are in the same *pixwin*.

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 shrank, 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 covers the proper approach to that task.

3.6.1. Handling SIGWINCH

There are several stages to handling a SIGWINCH:

First, the procedure which catches the signal almost always 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, e.g. halfway through a window's repaint for some normal cause. Consequently, the appropriate action in the signal handler is usually to simply set a flag which will be tested somewhere else. Conveniently, a SIGWINCH (like any other signal) will break a process out of a *select* system call, so it is possible for a client which was blocked to be awakened, and, by dint of a little investigation, discover the cause. (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 what exactly 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 the next chapter) can be used to inquire the current dimensions of a window. The previous size must have been remembered, e.g. from when the window was created, or last adjusted. These two sizes are compared to see if the size changed. Upon noticing that its size has changed, a window containing other windows may wish to rearranged 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:

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

pw_damaged 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 opsdata* (in *pwcd_damagedid* as well), and 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.

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 (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.*

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

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

which 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 generated it. The only solution is to fix up *all* windows. Fortunately, that should not be overly expensive, since the damaged areas are comletely and exactly specified by the returned value for *pw_damaged.*

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 this *sunwindow* level of the window system, that allows displaying images on windows which may be overlapped.

The *pixwin* struct was the basic element of discussion in Chapter 3, encapsulating the information necessary for displaying an image on a window, including clipping and damage repair. Another structure underlies the operations described in this chapter; but, since it 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*; and 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 Chapter 3 are required to display output on a window.)

Most of the window manipulations described in this chapter are performed by *ioctl* system calls. However, client programs should use the procedures described in this chapter because they are the supported interface. The header file */usr/include/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 which 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 chapter 3;
- 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 which indicates what user input actions the window should be notified of;
- another window, which is given any input events not used by this window; 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 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* 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;
```

passing it a file descriptor it got by opening */dev/win0*. The return value is a *window number*, as described in 4.2.3 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; section 4.9 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 non-trivial; see the discussion in section 4.9.

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. We introduced the *window name* above; the other form is the *window number*, which corresponds to the numeric component of the window name. Both of these references are valid across process boundaries. The *window number* will appear in several contexts below.

Procedures are supplied for converting the various window identifiers back and forth:

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

stores the filename for the window whose number is *winnumber* into the buffer addressed by *name*, which should be WIN_NAMESIZE long (as should all the name buffers in this section).

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

returns the window number of the window whose name is passed in *name*.

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

given a window file descriptor, stores the corresponding device name into the buffer addressed by *name.*

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

returns the window number for the window whose file descriptor is *windowfd.*

4.3. Window Geometry

Once a window has been opened, its size and position may be set. The same routines that are used for this purpose are also helpful for adjusting the screen positions of a window at other times, e.g. when user-interface actions indicate that it is to be moved or stretched. 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. (Section 4.4.1 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(windowfd, rect)
   int          windowfd;
   struct       rect   *rect;
```

copies the rect argument's data into the rect of the indicated window; 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, e.g. *icons* (see section 8.1). 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. The third 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_YOUNGEST      == WL_TOPCHILD
```

A link which has no corresponding window (a child link of a "leaf" window, for instance) 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:

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

returns a window number, which is the value of the selected link for the window associated with *windowfd.*

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

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, file descriptors will be available for the other windows, and their window numbers may 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 the section entitled *Passing Parameters to a Tool* in the chapter on tools for an example of this environment parameter's usage).

4.4.2. Activating the Window

Once a window's links have all been defined, it 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 re-insert 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;
```

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 which 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 associated with it 32 bits of uninterpreted client data. This is not touched by the basic window system; typically it will be used by the client to store flags. Higher levels of the system may implement minimal inter-window status sharing through this facility. This data is manipulated with the following procedures:

```
win_getuserflags(windowfd)
   int         windowfd;

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

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, and *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 the appendix on rects and rectlists; readers should expect to 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 windows on the screen 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 an area (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 that it hasn't told the window system that it has fixed up.

These facilities can be used by any window manager according to the following strategy:

- The window database is locked and manipulated to accomplish the rearrangement. (*win_lockdata, win_remove, win_setlink, win_setrect, win_insert ...*)
- The old exposed areas for the affected windows are gotten and cached. (*pw_exposed*)
- 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 their signals informing them of the (reduced) damage which must be repaired.

4.7. Relations to Physical Screens

Note: The design calls for multiple concurrent screen support. Currently, only one screen is supported at a time. Also, this entire interface describing the screen is tenative and likely to change.

Multiple displays may be attached to a processor at the same time, and clients may want windows on all of them. Therefore, the window database is a grove, with one tree of windows for each display. (Thus, there is no overlapping of window trees which belong to different screens.) 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 screen  {
   char        scr_name[SCR_NAMESIZE];
   int         scr_type,
   int         scr_reverse;
   struct      rect    scr_rect;
   int         scr_pixeldepth,
   int         scr_pixelsperinch,
   int         scr_colormapsize;
};

#define  SCR_NAMESIZE    20
#define  SCR_SUN1BW      1
```

Scr_name is the device name of the screen (e.g. "*/dev/bw0*"). *Scr_type* is the device type; currently defined types are FBTYPE-SUN1BW and FBTYPE-SUN2BW found in "*/usr/include/sun/fbio.h*". *Scr_rect* is the size of the screen.

The following fields are defined but the window system is ignoring them. *Scr_reverse* is TRUE if the screen is inverted from Sun's default. (More easily remembered, it holds the value which will appear white.) *Scr_pixeldepth*, *Scr_pixelsperinch*, and *Scr_colormapsize* describe physical display characteristics, where *pixeldepth* is in bits, *pixelsperinch* assumes the display pixels are square, and *colormapsize* is the number of entries in the color lookup table (2 for black-and-white displays, with or without colormaps).

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

is used to associate a window with a screen. *Windowfd* is a file descriptor for an existing window; *screen* addresses a screen struct in which the *scr_name*, *scr_type*, and *scr_reverse* entries have been set. A new *desktop* (screen / window-tree combination) is set up, with the indicated window as the root window.

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

fills in the struct addressed *screen* with information for the screen with which the window indicated by *windowfd* is associated. (That can be any window in the tree; the root window is found by the procedure.)

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

completely destroys the desktop of which *windowfd*'s window is the root: it destroys the root window and all windows descended from it, terminating their owner processes (using SIGTERM), and breaks the association set up by *win_screennew*.

```
win_screenpositions(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
```

is used to inform the window system of the physical layout of multiple screens, to enable the cursor to cross to the appropriate one. *Windowfd*'s window is the root for its desktop; 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_screenpositions*, as stated above is not currently implemented.

4.8. Cursor and Mouse Manipulations

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.

The "hot spot" defined by (*cur_xhot, cur_yhot*) is used to associate 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, e.g., an hourglass to indicate that some processing is occurring. This type of cursor should have the hot spot located in the middle of its image so that the user can still use it for pointing without having to guess where the hot spot is.

The function indicated by *cur_function* is a rasterop (as described in section 2.2.5), which will be used to paint the cursor. (PIX_SRC | PIX_DST is generally effective on light backgrounds, e.g. 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 into the buffer addressed by *cursor* a copy of the cursor which is currently being used on the screen.

```
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.

4.8.2. Mouse Position

Determining the mouse's current position is treated under *Input* in the following chapter. 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 which is given for the mouse in input events can be reset, without the mouse on the table top being physically moved:

```
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 effect is of 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, cursor changes) will be generated. This facility should be used with restraint; 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 fd, and (*x, y*) defines a screen position in that window's coordinate space (it need not actually lie within the window), and the returned value is a window number. *X* and *y* may lie outside the bonds of the window.

4.9. Providing for Naive Programs

There are a large class of applications which 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 two ways of supplying for these applications.

4.9.1. Which Window to Use

An environment parameter defined by the window system is of interest here. By convention, WINDOW_GFX is set to a string which is the device name of a window in which graphics programs should be run. 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. Taking Over an Existing Window

Experience has shown that the following method is not as good as the second approach described in the next section. However, this approach is documented because higher level utilities (emptysw and gfxsw) use this approach and haven't yet been changed to the better one. When these utilities are converted it should be invisible to their clients because the interfaces shouldn't change.

Windows may be opened more than once. This fact can be used to allow a process to temporarily "take over" a window from another. Several issues must be addressed in this sharing of windows:

- The original owner (call it the executive) must inform the newcomer (call it the demo) what window it's getting. This is normally passed via the environment parameter WINDOW_GFX described above.
- The demo, having opened the window for its own use, should make itself the window's owner, so that it will be given relevant SIGWINCHes. See ***win_getowner*** and ***win_setowner*** below. It should then proceed to set up the window as it would a newly-created one saving any window parameters that is changes.
- It is possible for either process to attempt to read from the window, or display on it at the same time; this should normally be avoided, since the results are unpredictable. The most common arrangement is for the executive to leave the window alone until the demo is finished.
- Finally, the trickiest issue is to ensure that the executive gets back full control when the demo is finished. The demo can reset the owner before exiting, or the executive can be catching SIGCHLD, and make itself owner again when the demo goes away. Other properties of the window should be reset upon returning control to the original owner, including the cursor, input mask and input redirection window. If the executive has another window that it has not given up, it may accept user inputs in that window which instruct it to destroy a wayward demo, and recover the window.

SIGWINCH signals are directed to the process which ***owns*** the window. Normally the owner is the process which created it; this may be read and written by the following procedures:

```
int win_getowner(windowfd)
    int         windowfd;

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

Win_getowner returns the process id of the owner of the indicated window. If the owner doesn't exist then zero is returned. ***Win_setowner*** makes the process identified by ***pid*** be the owner of the window indicated by ***windowfd***. ***Win_setowner*** causes an SIGWINCH to be sent to the new owner.

4.9.3. Covering an Existing Window

Another (and probably better) approach is to create a new window that becomes attached to, and covers, an existing window:

- The invoking process (call it the executive) must inform the newcomer (call it the demo) what window (call it the gfx window) to attach itself to. This would be passed via the environment parameter WINDOW_GFX described above.
- The demo, having created a window, would make itself the same size as the gfx window and install itself as the gfx window's top child.
- The executive's only job would be to change its top child window's size whenever the gfx window changes size.
- When the demo finished, the demo window would be destroyed thus leaving the gfx window uncovered.

The main advantage of this scheme is that the problems of restoring the gfx window's state (see previous section) are avoided.

4.10. Error Handling

Except as explicitly noted, the procedures described in this section do not return error codes. The standard error reporting mechanism used inside the *sunwindow* library is to call a procedure which prints a message (typically identifying the *ioctl* call which detected the error), after which the calling process resumes execution.

This default error handling routine may be replaced by calling:

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

That is, *win_errorhandler* is a procedure which 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 which returns an **int**.

```
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 the section entitled *Error Message Decoding* in the appendix about *Programming Notes*).

5. INPUT

In this third chapter devoted to the *sunwindow* level of the Sun window system, we discuss how user input is made available to application programs. The structures and procedures discussed in this section (unless otherwise noted) 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 (e.g. "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 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 the *Virtual Input Device* with its input events, described in the first section of this chapter.

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 acccunt 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.

function keys. The mapping to physical keys on various keyboards is defined in */usr/include/sun/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 moment (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 (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 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.

Caveat: The only negative events currently reported are locator buttons going up.

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 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 occurrs. 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: CAPSMASK SHIFTMASK CTRLMASK UPMASK These are defined in */usr/include/sun/kbd.h*, and described in *cons*(4).

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       input_event   *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 blocking or non-blocking reads via a standard *fcntl* system call, as described in *fctnl*(2) and *fcntl*(5). A window is defaulted to blocking reads. The blocking status of a window can be determined by the *fcntl* system call.

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 7 illustrates this approach. Using blocking I/O and *read*(2) without a prior *select* forces the client to process SIGWINCHes 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 next 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 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 the next section.) 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*, in section 5.3.2 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_UNENCODED
```

indicates that no translation of physical device events should be performed (that is, that the Virtual Input Device not intervene between the window and the user input). In this case, the most significant byte of the code 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 (127). For mouse input, locator motion and locator button events place in the least significant byte of the code the event code used in the corresponding unencoded case minus 512. Note that the 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.

```
        #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*; function keys with a standard interpretation (e.g., 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.

Caveat: The current implementation automatically enables both IM_ANSI and IM_POSASCII when IM_ACSII 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 5.1.3 above for a discussion of positive and negative events.)

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.

Caveat: 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 */usr/include/sunwindow/win_struct.h*), it is interpreted as designating the window's parent.

Caveat: 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. Better still, though, is 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 **int** addressed by designee.

We return to *win_input.h* for three routines useful for manipulating input masks. The first two 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_inputcodebit(im, code)
    struct      inputmask *im;
    char        code;
```

returns true or false as the bit indexed by *code* in the input mask addressed by *im* is 1 or not.

```
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

The actual names of codes which appear in the *ie_code* field of an *inputevent* are:

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


#define VKEY_CODES                (128)
#define VKEY_FIRST                (512)

#define VKEY_FIRSTPSEUDO          (VKEY_FIRST)
#define LOC_MOVE                  (VKEY_FIRSTPSEUDO+ 0) /* No neg event */
#define LOC_STILL                 (VKEY_FIRSTPSEUDO+ 1) /* No neg event */
#define LOC_WINENTER              (VKEY_FIRSTPSEUDO+ 2) /* No neg event */
#define LOC_WINEXIT               (VKEY_FIRSTPSEUDO+ 3) /* No neg event */
#define LOC_MOVEWHILEBUTDOWN                (VKEY_FIRSTPSEUDO+ 4) /* No neg ev
#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)
```

6. SUNTOOL: TOOLS AND SUBWINDOWS

This chapter introduces the third and highest level of the Sun window system. It is at this level that facilities of the lower levels are actually used to build user interfaces. We describe a model for building applications, a number of components which implement commonly-needed portions of such applications, an executive and operating environment which supports that model, and some general-purpose utilities which can be used in this and similar environments.

We refer to an application progam which is a client of this level of the window system as a *tool*. This term covers the one or more processes which do the actual application work. *Tool* 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 provides a few tools ready-built (several are illustrated in Appendix B), and more will be provided as time passes; customers are expected to provide more to suit their particular needs.

Common components of tools that the window system provides include:

- An executive framework which supplies the usual "main loop" of a program, and which serves to coordinate the activities of the various subwindows;
- A standard *tool window*, which 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 facilities for adjusting its size and position (including layering), and subwindow boundary movement.
- several standard *subwindows*, which can be instantiated in the tool;
- a standard scheme for laying out those subwindows; and
- a facility which provides a default *icon*, which is a small form the tool may take to be unobtrusive but still identifiable.

The *suntools* program initializes and oversees 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;
- the user interface for leaving the window system.

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

This chapter discusses how to write a tool: it covers creation and destruction of a tool and its subwindows, the strategy for dividing up work among them, and the use of routines provided to accomplish that work.

Chapter 7 is a discussion of *subwindows*, as building blocks in the construction of a tool. It covers the subwindows currently existing, and also discusses the approach to be followed in creating new kinds of subwindows. We expect the library of available subwindow types to grow, with contributions from both Sun and our customers.

Chapter 8 covers user interface utilities. These are relatively independent packages which can be used with the *suntools* environment, or a similar replacement. They include the actual window manipulation routines used by *tool windows*, the *icon* facility, the *selection* manager (useful for inter-process data exchange), the *fullscreen* access mechanism, and *menus* and *prompts*.

The procedures which support all these facilities are found in the *suntool* library (that is, */usr/lib/libsuntool.a*). They and their data structures are declared in a number of distinct header files, all of which can be included in */usr/include/suntool/tool_hs.h*.

Appendices B and C are an annotated collection of some simple tools, to be used both as illustrations and as templates for client programmers.

Appendix D is a collection of progamming notes and advice.

6.1. Tools Design Philosophy

A typical tool is built as a fairly light-weight *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 of chapter 4; the subwindows form a tree 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-Preemption

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

This non-preemptive 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-preemptive 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 (outlined at the beginning of this chapter).

Subwindows are the workhorses of the *suntool* environment, but most of the work they do is specific to their own tasks, and so 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 done as a tool is started:

- Parameters for this invocation of the tool must be passed to it. Every tool must be given the name of its *parent window*; other parameters that may be given to the tool include a position for it on the screen, whether it should be open or iconic, specification of data files (e.g. fonts) to be used in this invocation, and initializations to be performed.

- The tool should be given its own process and process group. In contrast to the usual procedure when a program is invoked under the shell, the parent process should generally be allowed to go on its way.
- The tool window should be created, with space allocated for it and its various options defined; similarly, its subwindows should be created and positioned in the tool window.
- The UNIX signal system should be initialized to pass appropriate signals to the tool.
- The tool's window should be installed into the display structure.
- Finally, the tool may start its normal processing.

6.2.1. Passing Parameters to the Tool

There are at least three ways parameters may be passed to a tool that is starting up:

- It may have command-line arguments.
- Relatively stable options may be stored in a file (like a user profile).
- Environment parameters may be used for well-established values. They have the valuable property that they can communicate information across several layers of processes, not all of which have to be involved.

The first two parameters passing mechanisms need no special attention here, since they are used just as in non-window UNIX programs. However, the Sun window system itself uses a few environment variables for tool startup. WINDOW_PARENT is set to a string which is the device name of a window's parent; for a tool, this will usually be the name of the root window of the window system. WINDOW_INITIALDATA is set to the coordinates of two rectangles plus one flag; these are the regions for the window while open and closed, and a boolean which is non-zero if the tool should start out iconic.

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

sets WINDOW_PARENT to the name of the parent's window.

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

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

The process that is starting the tool should set WINDOW_INITIALDATA before it forks (*wmgr_forktool* does this; see *Suntools: User Interface Utilities*). After the fork, the newborn tool may interrogate 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.

For tools which are going to be providing windows to other processes to run in, a procedure is provided for unsetting the variable, lest a wayward child process be confused by it:

```
we_clearinitdata()
```

6.2.2. Forking the Tool

A tool will normally have its own process. The creation of that process does not differ significantly from the normal paradigm. 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 system call).

6.2.3. Creating the Tool Window

The pair of procedures *tool_create* and *tool_createsubwindow* carry out the main work of creating a tool with its subwindows. These take a series of parameters which define the object to be created, and return a pointer to an object which encapsulates the interesting information about the tool or its subwindow, as the case may be. That pointer is then passed to a number of other routines which manipulate the object; the client is usually not concerned with the exact definition of the structure.

These create routines include a large part of the processing described in the earlier parts of this manual, so that client programmers need not necessarily concern themselves much with the details of *pixrects* and *pixwins*.

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
```

where

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 flags 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 inital position and size in which the tool in it normal (open) state is to be displayed, in the coordinate system of the tool's parent (typically, the window for the screen); and

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 on making custom icons, in chapter 8; the choice is strictly a matter of convenience vs. ambition. A tool's starting position should almost always be left NULL; it is better to communicate it via the environment parameter-passing mechanisms described above.

Creating the tool does not cause it to appear on the screen; a separate step is used for that purpose after the full tool structure is constructed, as described in *Tool Installation* below. Most tool programmers can skim down to *Subwindow Initialization* below and ignore the details of the *tool* and *toolsw* data structures without missing anything of direct interest.

6.2.4. The Tool Struct

The tool struct is defined in */usr/include/suntool/tool.h*:

```
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, 6 flags are defined:

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

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 indicates that the option to allow the user to move inter-subwindow boundaries is to be enabled. TOOL_ICONIC indicates the current state of the tool: 1 = small (*iconic*); 0 = normal (*open*).

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. These three flags are used during *tool_select* processing (see 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 modifiying its position or shape. The uses of *windowfds* are discussed in chapters 3 through 5 of this manual.

Tl_name addresses the string which can be displayed in the tool's namestripe and default icon.

Tl_rectcache holds a rectangle which 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 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 (oldest) of the tool's subwindows. These structs are discussed in the following section.

Tl_io is used by the tool to control notification of input and window change events to itself. This structure type is discussed in detail in *Toolio Structure.* During tool creation, the fields of this structure are set up with values to do default tool processing.

6.2.5. Subwindow Creation

```
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 as to 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. Details of subwindow layout are discussed in section 6.2.6 below. 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; the file descriptor is stored in *ts_windowfd* by *tool_createsubwindow.*

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, this structure has null values in it that need to be set; this is normally doen by the *create* routine for a standard subwindow type. This structure is discussed in detail in *Toolio Structure.*

Ts_destroy gets called when the tool is being destroyed (*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 *Requirements for Subwindows* in Chapter 7.)

6.2.6. 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.

Subwindows that should be arranged in a more controlled fashion may be rearranged after they have all been created, using the rectangle manipulation facilities described in *Window Geometry.* Three functions return numbers useful to tools doing their own subwindow layout explicitly:

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

returns the height in pixels of the tool's name stripe.

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

returns the width (in pixels) of the tool's outside border

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

returns the number of pixels that should be left as a margin between subwindows of a tool (currently the same as the outside border of the tool).

6.2.7. Subwindow Initialization

By the time *tool_createsubwindow* has returned, the subwindow is already inserted in the tree 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. This will call *tool_createsubwindow* and then initialize *ts_io* and *ts_data.*

6.2.8. Tool Installation

Once the tool is created and its subwindows have been created and installed, the software interrupt system should be turned on via a call to *signal* as described in 6.3.3. 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.

6.2.9. 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. Care should be taken that the pointer passed to *tool_destroy* is never 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;
```

A tool may use this procedure to change its subwindows, while continuing to run.

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, passing in the *toolio* structures of the tool and its subwindows. When the handler returns, *tool_select* awaits another event.

6.3.1. Toolio Structure

The *toolio* data structure holds what is needed for a window to wait for something to happen in the *select* system call. It is defined in */usr/include/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 exactly 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. Similary, *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 compatability 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 zero valued timeval structure.

Toolio also contains pointers to the procedures that are called when some notification has been received by the tool. *Tio_handlesigwinch* addresses the procedure which 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 the tool this data is a pointer to the tool. For a subwindow this data is the *ts_data* value.

Tio_selected addresses the procedure which responds to notifications from the *select* system call. Its general form 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 then 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 occured.

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 from the three *toolio* masks from each 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, then 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. The *tio_selected* procedures are 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.

It should be noted that timers in this implementation are only approximate. When the *select* system call returns and a timeout hasn't occured, the duration of 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 is set up via the *signal* system 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 Maintainence

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 causes *tool_select* to try to gather up a dead child process (via a *wait3* system call). 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 Image

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 — its name or its icon's image have been changed, for instance. 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.

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 resond to user inputs, or that update their image in response to timer or other events, may be implemented by replacing the tool's *tool_selected* procedure. A different subwindow layout scheme may be implemented in a replacement procedure for *tio_handle sigwinch*. 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.

7. SUNTOOL: SUBWINDOW PACKAGES

This chapter describes ***subwindow packages***, the building blocks used to construct a ***tool***. It presents a guide for constructing new subwindow packages of general utility, and describes the available standard subwindow packages for use with ***suntools***. (Refer to the preceding chapter 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 in this section. 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 ***foo*** 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 foosubwindow {
    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 contained 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.

```
struct foosubwindow *foosw_init(windowfd, ...)
    int         windowfd;
```

creates new instances of a foo subwindow. ***Windowfd*** is to be a foo 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 Chapter 5, and dynamically allocates and fills the subwindow's data object.

```
foosw_done(foosw)
    struct      foosubwindow *foosw;
```

destroys subwindow instance data. Once this procedure is called, the ***foosw*** pointer should no longer be referenced.

```
        foosw_handlesigwinch(foosw)
           struct      foosubwindow *foosw;
```

This procedure handles repaint requests and must also detect and deal with changes in the window size. It is called as a result of some other procedure catching a SIGWINCH.

```
        foosw_selected(foosw, ibits, obits, ebits, timer)
           struct      foosubwindow *foosw;
           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 *foosw_createtoolsubwindow(tool, name, width, height, ...)
           struct      tool *tool;
           char        *name;
           short       width, height;
```

creates a struct *toolsw* that is a foo subwindow. *Foosw_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 *foosw_init.*

Foosw_createtoolsubwindow takes the window file descriptor it gets from *tool_createsubwindow,* passes it to *foosw_init,* and stores the resulting pointer in the tool subwindow's *ts_data* slot. The addresses of *foosw_handlesigwinch* and *foosw_selected* are stored in the appropriate slots of the *toolio* structure for the tool subwindow, and the address of *foosw_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 emply 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 (see *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 */usr/include/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. Since *esw_createtoolsubwindow* takes care of set up of 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.

```
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 Chapter 3 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 simply need a display area in which to run. Using this subwindow package insulates programmers of this type of program from much of the complexity of the window system. This subwindow package is unique among subwindow packages in that it doesn't generate the bits for its image. Instead, it provides a mechanism that programs can use to manage their own display area.

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.

The graphics subwindow can be used in tool building like any of the other subwindow packages described in this chapter. However, the ability for a program to take over an existing window from another process is also provided by the graphics subwindow.

The data definition for the instance-specific data defined in */usr/include/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 cyclic continuously running 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_takoverdata* is described in the following section.

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 examines 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 (e.g., graphics demos) will redraw from the beginning of a cycle. Other event driven programs (e.g., graphics editors, status windows) will redraw from their underlying data descriptions. The GFX_RESTART bit needs to be reset to 0 by the client program before actually doing any resetting.

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 then 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 ####". *Toolsw->ts_io.tio_selected* is set up with the client's own routine.

Toolsw->ts_io.tio_handlesigwinch is replaced with the client's own routine. This is so that the client is notified when something about his window changes. The client *tio_handlesigwinch* will call *gfxsw_interpretsigwinch* (which is 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.

```
gfxsw_interpretsigwinch(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

is a procedure that 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:

```
gfxsw_handlesigwinch(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

This procedure handles SIGWINCH signals. 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 then this routine will free the old retained pixrect and allocate one of the new size. If the window is not retained the damaged list associated with the window is thrown away. GFX_DAMAGED flag is reset to zero in this routine.

```
gfxsw_done(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

destroys the subwindow's instance data.

7.3.2. Taking Over an Existing Window

The ability for a program to take over an existing window from another process is provided by the graphics subwindow. The empty subwindow (described above) is designed to be taken over.

```
struct gfxsubwindow *gfxsw_init(windowfd, argv)
    int         windowfd;
    char        **argv;
```

This procedure creates a new instance of a graphics subwindow in something other than the *tool* context. *Windowfd* should be zero; the assumption is that there is some indication in the environment as to which window should be taken over. (See *we_getgfxwindow* in Chapter 4.) *Argv* is like *argv* in *gfxsw_createtoolsubwindow*.

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. Part of this, however, is cached data from of the original owner window: input mask, cursor image, input redirection window and original owner process identifier.

```
gfxsw_catchsigwinch()
```

When a graphics subwindow has taken over a window from another process, this procedure is set up as the signal catcher of SIGWINCH. It, in turn, calls *gfxsw_interpretesigwinch*.

```
gfxsw_cleanup()
```

Also, when a graphics subwindow has taken over a window from another process, *gfxsw_cleanup* is the signal catching routine used to catch SIGINT and SIGHUP. This routine resets the original owner window's state from the graphics subwindow's cached private data.

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 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 *foosw_selected* described at the beginning of this chapter. The only difference in the semantics of this routine and *foosw_selected* is that the *gfxsw->gfx_flags* should be examined and acted upon in *selected*.

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.

```
gfxsw_selectdone(gfxsw)
   struct      gfxsubwindow *gfxsw;
```

is a substitute for the *tool_done*. *gfxsw_selectdone* is called from within the *selected* procedure passed to *gfxsw_select*.

7.4. Message Subwindow

This subwindow package displays simple ASCII strings.

A private data definition that contains instance-specific data defined in */usr/include/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 currently properly dealt with; please no carriage returns, line feeds or tabs (yet). 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*. 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.*

```
    struct messagesubwindow *msgsw_init(windowfd, string, font)
       int         windowfd;
       char        *string;
       struct      pixfont *font;
```

creates a new instance of a message subwindow. *Windowfd* identifies the window to be used. *String* is the string being displayed using *font.*

```
    msgsw_setstring(msgsw, string)
       struct      messagesubwindow *msgsw;
       char        *string;
```

changes the existing *msgsw->msg_string* to *string* and redisplays the window.

```
    msgsw_display(msgsw)
       struct      messagesubwindow *msgsw;
```

redisplays the window.

```
    msgsw_handlesigwinch(msgsw)
       struct      messagesubwindow *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      messagesubwindow *msgsw;
```

destroys the subwindow's instance data.

7.5. Option Subwindow

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. New item types and extensions to these existing types are both contemplated.

The program *optiontool* is provided as a simple example of the features discussed here. Familiarity with the behavior of the program, and with its source file */usr/suntool/src/optiontool.c,* is helpful in reading this section.

The declarations for the optionsw package are found in the header file */usr/include/sunwtool/optionsw.h.* */usr/include/suntool/tool.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:

```
typedef struct          typed_pair {
        u_int   type;
        caddr_t         value;
}       typed_pair;

#define IM_UNKNOWN      0
#define IM_GRAPHIC      2
#define IM_TEXT         3
#define IM_TEXTVEC      4

typedef struct opt_item {
   struct       opt_item *oi_next;
   u_int        oi_flags;
   struct       rect oi_rect;
   caddr_t      oi_ops;
   caddr_t      oi_data;
}  opt_item;

#define LAY_XFIXED      0x80
#define LAY_YFIXED      0x40
#define LAY_WFIXED      0x20
#define LAY_HFIXED      0x10
```

For a *typed_pair*, *type* indicates what kind of object *value* points to. The current choices are:

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 below).

The four layout flags indicate an aspect of an option item's layout which should not be adjusted (left edge, top edge, width, height). Their use is discussed under *Item Layout and Relocation* below.

7.5.1. Option Subwindow Standard Procedures

This section describes the routines needed to conform to subwindow package norms. These routines follow the general procedures provided in section 7.1.

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

creates a 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. 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* which 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.

```
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.

```
optsw_done(optsw)
    caddr_t optsw;
```

is the cleanup routine for an optionsw. 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.

7.5.2. 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 a pointer to the *opt_item* which describes 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 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 inverting); but any such indication may be cancelled simply by moving the cursor off the item before letting up on the button.

7.5.2.1. Boolean Items

```
opt_item *optsw_bool(optsw, label, init, notify)
    caddr_t     optsw;
    struct typed_pair *label;
    int         init;
    int         (*notify)();
```

creates an item which maintains a boolean (TRUE or FALSE) value. Its *label* contains a pointer to a string (type is IM_TEXT) or to a pixrect (type is IM_GRAPHIC); this is what is displayed in the window for the item. The label is displayed in reverse video whenever the item is TRUE. The value of the item is initially set to *init,* and is changed whenever the user selects the item. Whenever the value of the item is changed by user action, the procedure *notify* is called with the new value, as described in section 7.5.4.

7.5.2.2. Command Items

```
opt_item *optsw_command(optsw, label, notify)
    caddr_t     optsw;
    struct typed_pair *label;
    int         (*notify)();
```

creates an item which invokes the client procedure *notify* when selected by the user; the item has no value. *Label* is as described for a boolean item above.

7.5.2.3. Enumerated Items

```
struct opt_item *optsw_enum(optsw, label, count, init, choices, notify)
    caddr_t     optsw;
    struct typed_pair *label;
    struct typed_pair *choices;
    int         count;
    int         init;
    int         (*notify)();
```

creates an item in which exactly one of a set of choices is in effect at any time. The value is interpreted as an index (0-based) into the choices for the selection. Optsw, label, and notify are ass above. Choices is a vector of images to be displayed for the choices; for now, they must be strings (type is ITEM_VEC). Init is the initial value of the item; it should be at most the size of the choices array - 2 (to avoid the null pointer which terminates the array). Flags will eventually indicate layout options, but for now should be 0.

7.5.2.4. Label Items

```
struct opt_item     *optsw_label(optsw, label)
    caddr_t         optsw;
    struct typed_pair *label;
```

creates an item which does nothing but paint itself. Optsw and label are as above. This item type may be used for including labelling information in the option subwindow.

7.5.2.5. Text Items

```
struct opt_item     *optsw_label(optsw, label, default_value, flags)
    caddr_t         optsw;
    struct typed_pair *label;
    char            *default_value;     int         flags;

  #define OPT_TEXTMASKED
```

creates an item which holds a text value. Optsw and label are as above. Default_value is the initial value of the string. Flags specify attributes of the created item; currently, only the masked attribute is supported. If the OPT_TEXTMASKED flag is set in the call, 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.

There may be multiple text items in an option subwindow. At any time, one of them is active; any keystrokes directed to the option subwindow will be appended to the current active text item. Only displayable characters will be accepted in the item (ASCII codes 040-0176 inclusive). Other characters will be discarded. This is initially the first item created in the option subwindow; the user may select another item to receive keystrokes by clicking either the left or middle mouse button while the cursor is pointing at the new item's label.

The user's erase (character delete) and kill (line delete) characters are available for editing existing text in a text item. The first will delete the last character of the text; the latter will delete the whole string.

Text items do NOT notify their client when their value changes; this would imply calling the client on every keystroke as the user enters data. Rather, clients should use optsw_getvalue to interrogate a text item when the value is needed.

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 below.

Caveat: This is a preliminary release of text items. The user interface will become noticeably less awkward in future releases. For now, the following restrictions apply:

> Currently, the only way of changing which item is active is by selecting the label of the new "active item." There is no feedback to indicate which item is current.
>
> 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 are planned, the actual interface (external procedure definitions and data structures) are designed to accommodate those extensions without change.

7.5.3. Item Layout and Relocation (SIGWINCH Handling)

As each item is created, its width and height are determined and stored in the *oi_rect* element of the item's *opt_item* struct. No left and top positions are assigned at this time. LAY_WFIXED and LAY_HFIXED are set in the item's *oi_flags*, to indicate that the width and height have been set to a value that should not be changed in the layout process. 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 inital window), a layout procedure determines positions for all the items in the window.

At any time after an item has been created and before it is destroyed, the client may set any elements of the item's rectangle by a call to:

```
optsw_setplace(optsw, ip, rp, reformat)
   caddr_t     optsw;
   opt_item    *ip;
   struct      rect *rp;
   bool        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. *Rp* is a pointer to a *rect* struct which specifies the modifications to be performed. Any value other than -1 is copied into the corresponding element of the item's *oi_rect* and the corresponding LAY_?FIXED flag is set in the item's *oi_flags* to prevent its being changed by the layout routine. A value of -1 indicates the existing value and flag are to be left unchanged. If *reformat* is TRUE, the whole window will be re-laid-out, taking the changed item into account; this is appropriate if the window is already displayed. On a batch of changes, it is appropriate to reformat only after the last change.

The rectangle is expressed in standard window-system fashion: pixel coordinates, with (0, 0) the first pixel in the upper left corner. For convenience in laying out string items, two functions convert character columns and lines to the appropriate pixel coordinate:

```
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 'n' 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.

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. This procedure does *not* set the LAY_XFIXED or LAY_YFIXED flags for the item; this allows the item to 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:

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;

subsequent items in the same row will not be positioned to the left of the item's right edge; and

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 or the width (or both) should be fixed.

7.5.4. 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 for specifying the procedure to be invoked in response to selection of a command button.

In any 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;
   opt_item    *item;
   int          value;
{ ... item has just changed to value;  do what you want with it.
}
```

Procedures to be invoked in response to a command button-push have the same form, except there is no *value* parameter.

Note the notification procedure is ***provided by the client,*** and ***invoked by the optionsw package.***

7.5.5. Explicit Client Reading and Writing of Item Values

```
int  optsw_getvalue(ip, dest)
     struct opt_item  *ip;
     caddr_t                   dest;
```

Ip is the item pointer returned by the item's create routine. *Dest* is the address where the value should be stored; it will be cast to the proper type by the specific routine for the item. Optsw_getvalue also returns an integer value. For all but text items, this is the same as the value stored in *dest; for text, it is the length of the value stored. Items which do not have a meaningful value (labels and commands) store and return -1.

```
optsw_setvalue(optsw, ip, value)
     caddr_t                   optsw;
     struct opt_item  *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, and value (appropriately cast) the new value to be assigned to the item.

7.5.6. Miscellany

```
optsw_setfont(optsw, font)
     caddr_t          optsw;
     struct pixfont   *font;
```

resets the font used to paint text labels and values in the option subwindow. Fonts for these objects are determined at the time the item is created, so to be effective, this routine should be called after creation of the option subwindow and before creation of the items to be affected. Different items may use different fonts.

The two procedures remaining in the optionsw interface are of secondary interest. 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, on an fd of the client's choice:

```
optsw_dumpsw(fd, optsw, verbose)
    int         fd;
    caddr_t     optsw;
    bool verbose;

optsw_dumpitem(fd, ip)
    int         fd;
    opt_item    *ip;
```

For each procedure, the client says where to write the dump with the *fd* 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.

7.6. Terminal Emulator Subwindow

This is the subwindow package that provides a Sun Terminal emulator.

The private data definition that contains instance-specific data defined in */usr/include/suntool/ttysw.h* is:

```
struct ttysubwindow {
    int         ttysw_placeholder;
};
```

(*Note: Only one TTY subwindow per process.*)

```
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 all of the set up of 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.*

```
struct ttysubwindow *ttysw_init(windowfd)
    int         windowfd;
```

creates a new instance of a tty subwindow. Windowfd is the window that is to be used.

```
ttysw_becomeconsole(ttysw)
    struct      ttysubwindow *ttysw;
```

sets up the terminal emulator to receive any output directed to the console. This should be called after calling *ttysw_init.*

```
ttysw_handlesigwinch(ttysw)
   struct      ttysubwindow *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 (See the following section). 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)
   struct      ttysubwindow *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 section 7.1.

```
int ttysw_fork(ttysw, argv, inputmask, outputmask, exceptmask)
   struct      ttysubwindow *ttysw;
   char        **argv;
   int         *inputmask, *outputmask, *exceptmask;
```

forks the program indicated by *argv. 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 run.
- If *argv is "–c", this flag and argv[1] are passed to a shell as arguments. The shell then runs argv[1]. (The arg list for this case becomes shell/-c/*(argv+ +)/0). If *argv is not NULL, the program named by argv[0] is run with the arguments given in the rest of argv.

```
ttysw_done(ttysw)
   struct      ttysubwindow *ttysw;
```

destroys the subwindow's instance data.

7.6.1. TTY-Based Programs in TTY Subwindows

TTY-based programs, such as *csh, sh,* and *vi,* which use the *termcap* to determine the size of their screen need not know about windows in order to run reasonably under the terminal emulator. The termcap library will return the current number of lines and columns of the terminal emulator. However, if the user changes his window's size while one of these programs is running, the terminal emulator and the 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 then there is no harm done because the default action for SIGWINCH is that it be ignored. A child process can catch the signal, and then requery the termcap library for the correct terminal size. Unfortunately, no TTY-based programs do this now.

The terminal emulator and a 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. These constants and the data that they pass in the ioctl call are further defined in */usr/include/sys/ioctl.h.*

```
    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 (e.g. */dev/win5*). *We_getmywindow* reads WINDOW_ME's value into *windowname*; a return value of 0 indicates success. 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.

8. SUNTOOL: USER INTERFACE UTILITIES

A variety of separate packages implement the user interface of suntool. These utilities are not tied to the notions of *tool* and *subwindow* described in a previous chapter. Thus, these packages could 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. This chapter describes the programming interface to these packages.

8.1. 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 */usr/include/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 right edge at 200 (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 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. although a global pointer named Sunwindow will keep multiple processes from gaining fullscreen access at the same time.

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(fs)
    struct      fullscreen *fs;
```

restores *fs*'s cached data, releases the right to access the full screen and destroys the fullscreen data object. *Fs->fs_windowfd*'s input blocking status is returned to its original state.

8.2. Icons

This section describes an icon display facility. The icon structure is simply a stylized description of an useful class of images. Icons normally serve more to identify an object than display its contents. A typical use of an icon would be to identify a currently unused but available tool. Another use might be to graphically depict an object (document, database element, resource) that a user might want to point at with his mouse. The icon structure is declared in the file */usr/include/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 relative), which may overlap. *Ic_mpr* addresses a memory pixrect (as described in section 2.4) which 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 */usr/include/suntool/tool.h*).

ICON_BKGRDCLR clear (white out) the image, or

ICON_BKGRDSET set (solid black) the image.

```
icon_display(icon, pixwin, x, y)
    struct     icon *icon;
    struct     pixwin *pixwin;
    int        x, y;
```

is used to display *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; and non-uniform icons have esthetic and placement defects. Therefore a set of standard dimensions should be provided for any particular class of icons. Here are the standards used by clients of tools (defined in */usr/include/suntool/tool.h*):

```
#define TOOL_ICONWIDTH        64
#define TOOL_ICONHEIGHT       64
#define TOOL_ICONMARGIN        2

#define TOOL_ICONIMAGEWIDTH
#define TOOL_ICONIMAGEHEIGHT
#define TOOL_ICONIMAGELEFT
#define TOOL_ICONIMAGETOP

#define TOOL_ICONTEXTWIDTH
#define TOOL_ICONTEXTHEIGHT
#define TOOL_ICONTEXTLEFT
#define TOOL_ICONTEXTTOP
```

These constants put the icon in a 64-pixel square, including a 2-pixel 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 overlies the bottom 3/4 of that region.

8.3. 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 */usr/include/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
```

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 is the only currently defined image data type and is a character pointer. ***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. ***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 is the only currently defined image data type and is a character pointer. *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). This is done 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. (An example of a menu set is found in *Sample Tools* in the *panetool* program.) Then when a user action initiates menu processing (button-down on the right mouse button is the standard invocation), 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 (top) menu structure in a menu stack. This indirection allows the menu package to leave the new top of the stack (if the user causes the stack order to be rearranged) in **menuptr* upon returning from *menu_display*. The stack's *m_next* values are shuffled by the menu package 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 inputevent which provoked the menu; the location information (*event->ie_locx, event->ie_locy*) in the event 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 menu button is the usual menu button. The left mouse button is always used as the accelerator to bring rear menus forwards). 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.

8.3.1. Prompts

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 */usr/include/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 currently properly dealt with, no carriage returns, line feeds or tabs (yet) please. If any of *prt_rect*'s fields are PROMPT_FLEXIBLE that dimension is automatically chosen by the prompt mechanism to accommodate the number of characters that fix using the given font.

```
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.

8.4. Selection Management

A common style of operation/operand command specification is a non-modal one in which the operand is specified first. In the window system, the operand is called the *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 does some action which causes the selection to be removed.

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.

The header file */usr/include/suntool/selection.h* contains the definition necessary for using selections:

```
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
```

is the object that describes a selection. *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 privately understood data (32 bits worth) that is only understood between implementations of a particular selection type.

The selection structure is not to be confused with actual selection data itself, e.g. the characters in a SELTYPE_CHAR selection.

```
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 that is called 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*, an stdio FILE pointer. The stdio library is used to write the selection data to *file*. *Windowfd* is the window that is making the selection.

Sel_clear is a procedure 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 [see* *above], it is the selection maker'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 that *selection_get* calls to enable the client to retrieve the selection. *Windowfd* is the window that wants to find out about the selection.

The *sel_read* procedure takes as arguments *sel*, the selection description of the current selection, and *file*, a standard io 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.

8.5. Window Management

The following procedures implement common functions for adjusting window relationships. They may be used to provide a window management user interface different from that provided by tools. If a series of calls are to be made to these procedures, the whole sequence should be bracketed by *win_lockdata* / *win_unlockdata*, as described in section 4.4.

```
bool wmgr_changelevelonly(windowfd, parentfd, top)
   int          windowfd, parentfd;
   booltop;
```

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).

```
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. *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 parent of *windowfd*'s upper-left screen coordinates of the

```
wmgr_changelevel(windowfd, parentfd, top)
   int          windowfd, parentfd;
   bool         top;
```

is like *wmgr_changelevelonly,* except that no optimization is performed to reduce the amount of repainting. This is used in conjunction with other window rearrangements, which make repainting unlikely. For example, when the tool window manager makes a tool iconic, it puts it at the bottom of the tool window stack after changing its state.

```
wmgr_refreshwindow(windowfd)
   int         windowfd;
```

causes *windowfd* and all its descendant windows to repaint.

```
wmgr_changestate(windowfd, rootfd, close)
   int         windowfd;
   int         rootfd;
   bool        close;

#define        WMGR_SETPOS      -1
#define        WMGR_ICONIC      WUF_WMGR1
```

changes the window identified by *windowfd* to be closed (iconic) or open, depending on whether *close* is TRUE or FALSE. The user data of *windowfd* reflects the state of the window via the WMGR_ICONIC flag (WUF_WMGR1 is defined in */usr/include/sunwindow/win_ioctl.h* and WMGR_ICONIC is defined in */usr/include/suntool/wmgr.h*).

The following procedures are used to resolve position/size undefined situations for the window's new rectangle:

```
wmgr_figuretoolrect(rootfd, rect)
   int         rootfd;
   struct      rect *rect;
wmgr_figureiconrect(rootfd, rect)
   int         rootfd;
   struct      rect *rect;
```

The *rootfd* window maintains a "next slot" position for both normal tool windows and icon windows. This allows windows to be assigned initial positions that don't pile up on top of one another. 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 tool windows and icon windows.

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* (both of which may have undefined fields). If *iconic* is not zero then the tool is created normal size. *Programname* is the name of the file that is to be run (a path search is done to locate the file) and *otherargs* is the command line that you want to pass to the tool.

Args that have embedded white space should be enclosed by double quotes.

```
    wmgr_iswindowopen(windowfd)
       int          windowfd;
```

tests the WMGR_ICONIC flag (see above) and returns TRUE or FALSE as the window is open or closed.

```
    wmgr_winandchildrenexposed(pixwin, rl)
       struct       pixwin *pixwin;
       struct       rectlist *rl;
```

can be used with your own window management routines to compute the visible portion of *pixwin->pw_clipdata.pwcd_windowfd* and its descendants and store it in *rl*.

9. APPENDIX A: RECTS & RECTLISTS

This appendix describes the geometric structures used with sunwindow and a full description of the operations on these structures. Throughout the 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 *rect.h* and *rectlist.h* are found in */usr/include/sunwindow/*. The library that provides the implmentation of the functions of these data types are 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.

9.1. Rects

The rect is the basic description of a rectangle, and there are macros and proceduress 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.

9.1.1. 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;
```

return 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:

```
rect_construct(r, x, y, w, h)
    struct      rect *r;
```

fills in *r* with the indicated origin and dimensions.

```
rect_marginadjust(r, m)
    struct      rect *r;
```

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 (resp. subtracted from) the origin of the rectangle pointed to by *r*, thus transforming the rectangle to the new coordinate system.

9.1.2. Procedures and Extern Data

A null rectangle (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;
```

returns the minimal rect which 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 the *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, else TRUE. Note: This procedure shouldn't 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, etc.

9.2. Rectlists

A number of rectangles may be collected into a list which 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.

9.2.1. 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*; e.g., 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)       and
rl_passtochild(x, y, rl)
    coord       x, y;
    struct      rectlist *rl;
```

which add (subtract) the given coordinates from the rectlist's *rl_x* and *rl_y* to convert the *rl* into its parent's (child's) coordinate system.

9.2.2. Procedures and Extern Data

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.

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.

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.

10. APPENDIX B: SAMPLE TOOLS

These are sample tools that can be used as starting points for tools of your own. The source files for these and other tools are found on */usr/suntool/src/*tool.c.*

10.1. gfxtool.c Code

```
#ifndef lint
static  char sccsid[] = "@(#)gfxtool.c 1.6 83/10/18 Sun Micro";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:       Graphics Window: A shell subwindow and an empty
 *                      subwindow inwhich graphics programs can run.
 */

#include <sys/types.h>
#include <signal.h>
#include "pixrect/pixrect.h"
#include "pixrect/pixfont.h"
#include "pixrect/pr_util.h"
#include "pixrect/memvar.h"
#include "sunwindow/rect.h"
#include "sunwindow/rectlist.h"
#include "sunwindow/pixwin.h"
#include "sunwindow/win_struct.h"
#include "sunwindow/win_environ.h"
#include "suntool/icon.h"
#include "suntool/tool.h"
#include "suntool/emptysw.h"
#include "suntool/ttysw.h"

static short ic_image[256]={
#include "gfxtool.icon"
};
mpr_static(gfxic_mpr, 64, 64, 1, ic_image);

static  struct icon icon = {64, 64, (struct pixrect *)0, 0, 0, 64, 64,
            &gfxic_mpr, 0, 0, 0, 0, (char *)0, (struct pixfont *)0,
            ICON_BKGRDGRY};

static  int sigwinchcatcher(), sigchldcatcher();

static  struct tool *tool;

gfxtool_main(argc, argv)
        int argc;
        char **argv;
{
        char    *toolname = "Graphics Tool 1.0";
        struct  toolsw *ttysw, *emptysw;
        char    name[WIN_NAMESIZE];
	/*
	 * Create tool window
	 */
	tool = tool_create(toolname, TOOL_NAMESTRIPE|TOOL_BOUNDARYMGR,
	   (struct rect *)0, &icon);
	/*
	 * Create subwindows
	 */
	ttysw = ttysw_createtoolsubwindow(tool, "ttysw",
	   TOOL_SWEXTENDTOEDGE, 200);
	emptysw = esw_createtoolsubwindow(tool, "emptysw",
	   TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
	/*
	 * Setup gfx window environment value.
	 */
	win_fdtoname(emptysw->ts_windowfd, name);
	we_setgfxwindow(name);
	/*
	 * Install tool in tree of windows
	 */
	signal(SIGWINCH, sigwinchcatcher);
	signal(SIGCHLD, sigchldcatcher);
	tool_install(tool);
	/*
	 * Start tty process
	 */
	if (ttysw_fork(ttysw->ts_data, ++argv, &ttysw->ts_io.tio_inputmask,
	   &ttysw->ts_io.tio_outputmask, &ttysw->ts_io.tio_exceptmask) == -1) {
		perror("gfxtool");
		exit(1);
	}
	/*
	 * Handle input
	 */
	tool_select(tool, 1 /* means wait for child process to die*/);
	/*
	 * Cleanup
	 */
	tool_destroy(tool);
	exit(0);
}

static
sigchldcatcher()
{
	tool_sigchld(tool);
}

static
sigwinchcatcher()
```

```
{
        tool_sigwinch(tool);
}
```

10.2. panetool.c Code

```
#ifndef lint
static  char sccsid[] = "@(#)panetool.c 1.8 83/10/18 Sun Micro";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:       Pane Tool: Sample program to illustrate multiple
 *                      subwindows.
 */

#include <sys/types.h>
#include <sys/time.h>
#include <signal.h>
#include "pixrect/pixrect.h"
#include "pixrect/pixfont.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/icon.h"
#include "suntool/tool.h"
#include "suntool/msgsw.h"
#include "suntool/menu.h"

static  int sigwinchcatcher();

static  struct tool *tool;

static  char charbuf[4];

struct  menuitem m3_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m3_menubody = {
        MENU_IMAGESTRING, "M3", sizeof(m3_items) / sizeof(struct menuitem), m3_ite
struct  menuitem m2_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m2_menubody = {
        MENU_IMAGESTRING, "M2", sizeof(m2_items) / sizeof(struct menuitem),
        m2_items, &m3_menubody, 0};
struct  menuitem m1_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m1_menubody = {
        MENU_IMAGESTRING, "M1", sizeof(m1_items) / sizeof(struct menuitem),
        m1_items, &m2_menubody, 0};
struct  menu *stack1menutop = &m1_menubody;

struct  menuitem m4_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m4_menubody = {
        MENU_IMAGESTRING, "M4", sizeof(m4_items) / sizeof(struct menuitem),
          m4_items, 0, 0 };
struct  menuitem m5_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m5_menubody = {
          MENU_IMAGESTRING, "M5", sizeof(m5_items) / sizeof(struct menuitem),
          m5_items, &m4_menubody, 0};
struct  menuitem m6_items[] = { MENU_IMAGESTRING, "Menu Item",0};
struct  menu m6_menubody = {
          MENU_IMAGESTRING, "M6", sizeof(m6_items) / sizeof(struct menuitem),
          m6_items, &m5_menubody, 0};
struct  menu *stack2menutop = &m6_menubody;
int     menutoggle;

main(argc, argv)
        int argc;
        char **argv;
{
        char    *toolname = "Pane Tool 1.0 (A sample tool)";
        struct  toolsw *paneNW, *paneNE, *paneSW, *paneSE;
        extern struct pixfont *pf_sys;

        /*
         * Create tool window
         */
        tool = tool_create(toolname, TOOL_NAMESTRIPE|TOOL_BOUNDARYMGR,
           (struct rect *) 0, (struct icon *) 0);
        /*
         * Create msg subwindows
         */
        paneNW = msgsw_createtoolsubwindow(tool, "paneNW",
           100, 100, "Raw keyboard input", pf_sys);
        paneNE = msgsw_createtoolsubwindow(tool, "paneNE",
           TOOL_SWEXTENDTOEDGE, 100,
           "Key input here redirected to NW subwindow", pf_sys);
        paneSW = msgsw_createtoolsubwindow(tool, "paneSW",
           100, TOOL_SWEXTENDTOEDGE, "Display alternating menu stacks",pf_sys);
        paneSE = msgsw_createtoolsubwindow(tool, "paneSE",
           TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE,
           "Try moving subwindow boundaries", pf_sys);
        /*
         * Raw input and flushing
         */
        {
        struct  inputmask im;
        int     paneNW_selected();

        input_imnull(&im);
        im.im_flags |= IM_UNENCODED;
        win_setinputmask(paneNW->ts_windowfd, &im,  &im, WIN_NULLLINK);
        paneNW->ts_io.tio_selected = paneNW_selected;
        }
        /*
         * Input redirection
         */
        {
        struct  inputmask im;

        win_getinputmask(paneNE->ts_windowfd, &im, 0);
        win_setinputmask(paneNE->ts_windowfd, &im,  (struct inputmask *) 0,
            win_fdtonumber(paneNW->ts_windowfd));
        }
        /*
         * Multi menu stacks
         */
        {
        struct  inputmask im;
        int     paneSW_selected();

        input_imnull(&im);
        win_setinputcodebit(&im, MENU_BUT);
        win_setinputmask(paneSW->ts_windowfd, &im,  &im, WIN_NULLLINK);
        paneSW->ts_io.tio_selected = paneSW_selected;
        }
        /*
         * Install tool in tree of windows
         */
        signal(SIGWINCH, sigwinchcatcher);
        tool_install(tool);
        /*
         * Handle input
         */
        tool_select(tool, 0);
        /*
         * Cleanup
         */
        tool_destroy(tool);
        exit(0);
}

paneNW_selected(msgsw, ibits, obits, ebits, timer)
        struct  msgsubwindow *msgsw;
        int     *ibits, *obits, *ebits;
        struct  timeval **timer;
{
        struct  inputevent event;
        int     error;

        error = input_readevent(msgsw->msg_windowfd, &event);
        if (error < 0) {
                perror("panetool");
                return;
```

```
		}
		charbuf[0] = 'c';
		charbuf[1] = ':';
		charbuf[2] = (char) event.ie_code&0X7f;
		charbuf[3] = ' ';
		msgsw_setstring(msgsw, charbuf);
		*ibits = *obits + *ebits + 0;
}

paneSW_selected(msgsw, ibits, obits, ebits, timer)
		struct	msgsubwindow *msgsw;
		int	*ibits, *obits, *ebits;
		struct	timeval **timer;
{
		struct	inputevent event;
		int	error;
		extern struct menuitem *menu_display();

		error = input_readevent(msgsw->msg_windowfd, &event);
		if (error < 0) {
			perror("panetool");
			return;
		}
		(void) menu_display((menutoggle)? &stack1menutop: &stack2menutop,
		   &event, msgsw->msg_windowfd);
		menutoggle = !menutoggle;
		*ibits = *obits + *ebits + 0;
}

static
sigwinchcatcher()
{
		tool_sigwinch(tool);
}
```

11. APPENDIX C: SAMPLE GRAPHICS PROGRAMS

These are sample graphics programs that can be used as starting points for graphics programs of your own. The source files for these and other graphics demos are found on */usr/suntool/src/ *demo.c*.

11.1. bouncedemo.c Code

```
#ifndef lint
static  char sccsid[] = "@(#)bouncedemo.c 1.5 83/08/26 Sun Micro";
#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);

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);
}
```

11.2. framedemo.c Code

```
#ifndef lint
static  char sccsid[] = "@(#)framedemo.c 1.7 83/09/30 Sun Micro";
#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 ("take over" kind)
         */
        gfx = gfxsw_init(0, argv);
        /*
         * Set up input mask
         */
        input_imnull(&im);
        im.im_flags |= IM_ASCII;
        win_setinputmask(gfx->gfx_windowfd, &im, &im, WIN_NULLLINK);
        /*
         * 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 timeo
                /*
                 * Don't reset timeout
                 */
                return;
        default: {}
        }
}
*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++;
}
```

12. APPENDIX D: PROGRAMMING NOTES

Here are useful hints for programmers that use any of the pixrect, sunwindow or suntool libraries.

12.1. What Is Supported?

The code is the ultimate description of what programs actually do, but the documentation is the description of what is supported. Client programmers who use facilities discovered in header files or through the grapevine may have useful applications running much sooner than it they operated by the book; but they do so at the risk of having their work invalidated.

In early releases such as this, there may be significant discrepancies between the design (and the documentation derived from it), and what is actually implemented. In general, we have tried to indicate where features are only partially implemented, and in which directions future extensions may be expected.

Even in completed portions of the system, the possibility remains that even defined interfaces will change in response to new requirements or newly-discovered constraints. Such modifications will not be undertaken lightly, and should generally be accompanied by a description of the nature of the changes, and appropriate responses to them.

12.2. 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 some sample tools and Appendix C contains some sample graphics programs. The source for these and other sample tools and graphics programs are available on */usr/suntool/src/ *.c.*

12.3. Header Files Needed

It can sometimes be hard to find the header files needed to compile your program. This can be particularly hard in the window system because of the multiple layers of software and the large numbers of header files. Programming by example helps in some respects because a lot of header files are included already.

To alleviate the problem a bit, certain header files exist that include most of the header files necessary for working at a certain level. These header files are:

- */usr/include/pixrect/pixrect_hs.h* - include this header file if you are working at the pixrect display primitives layer.
- */usr/include/sunwindow/window_hs.h* - include this header file if you are working at the sunwindow basic window facilities layer. This will include headers needed to work at the pixrect layer as well.
- */usr/include/suntool/tool_hs.h* - include this header file if you are working with the suntool tool building facilities. This will include headers needed to work at the more primitive layers as well.

- */usr/include/suntool/gfx_hs.h* - include this header file if you are working with the suntool (standalone or "take over") graphics subwindow facilities. This will include headers needed to work at the more primitive layers as well.

The idea is to include only one of the above 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, etc. However, you'll probably only have to add a single header file for each additional increment of high level functionality.

12.4. Lint Libraries

You can do better type-checking than the C compiler and catch argument mismatches in your program by running *lint* over your program source. The Sun window system provides *lint libraries* to allow you to do this. *Llib-lpixrect*, *llib-lsunwindow*, and *llib-lsuntool* are the source files to make the actual binary lint libraries: *llib-lpixrect.ln*, *llib-lsunwindow.ln*, and *llib-lsuntool.ln*. These files are found on */usr/lib/lint/*.

12.5. Library Loading Order

When loading programs remember to load higher level libraries first, i.e. *-lsuntool -lsunwindow -lpixrect*.

12.6. 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 using this approach. The main one is that the maximum number of per-process (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 discussion 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.

12.7. Error Message Decoding

The default error reporting scheme described at the end of *Window Manipulation* prints out 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.

Doing this can give you a quick hint as to what is being complained about without resorting to a debugger.

12.8. Debugging Hints

When debugging non-terminal oriented programs in the window system there are some things that you should know to make things easier.

First, the program being debugged breaks to *adb* when a signal is received. This can be annoying with window programs because SIGWINCH is used to notify windows of certain changes in its state. *Adb*, however, has a way of disabling breaking to the debugger when a particular signal is received. To disable this, type "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.

Another window system specific situation 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 actual locking; these can be turned on from a debugger:

- *int pixwindebug* - When not zero will immediately release 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 will not acquire data lock so that the debugger is not continually getting hung by being blocked on writes to screen. Unpredictable things can result because of this action that can't properly be described in this context. However, this is *unlikely*.
- *int win_grabiodebug* - When not zero will not actually acquire exclusive io access rights so that the debugger wouldn't get hung by being blocked on writes to screen and not able to receive input. The debugged process will only be able to do normal display locking and be able to only get input in the normal way.

Change these variables only during debugging, when not changing them becomes a problem and when you know what you're doing!

12.9. Sufficient User Memory

To use the suntool environment comfortably with the released set of tools requires about 600K of user memory after booting UNIX. Comfort means acceptable response from *vi* while *make* is running a compilation in another window for example. This is achievable in the current 0.9 release on model 100U's with 1 megabyte of memory. You have to reconfigure your own kernel, deleting unused device drivers. The procedure is documented in the *System Manager's Manual*. For a workstation on the network with a single disk drive you will be able to reclaim about 60K of usable memory.

The recommended amount of memory is 2 megabytes. This gives excellent performance with room to accommodate future releases.

13. INDEX

The following index provides references to programming variables, constants, types, macros, programs, and function and procedure names used in the Sun window system. It gives section numbers where the best documentation of the term may be found.

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:

Please list typographical Errors by page number and actual text of the error.

Technical Errors:

Please list errors of fact by page number and actual text of the error.

Content:

Did this guide meet your needs? If not, please indicate what you think should be added or deleted in order to do so. Please comment on any material which you feel should be present but is not. Is there material which is in other manuals, but would be more convenient if it were in this manual?

Layout and Style:

Did you find the organization of this guide useful? If not, how would you rearrange things? Do you find the style of this manual pleasing or irritating? What would you like to see different?