sun
microsystems

SunView™ Programmer's Guide

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

Contents

Chapter 1 Introduction **3**

What is SunView? 3

History 4

Release 3.0 4

Release 3.2 4

Code No Longer Supported 5

Chapter 2 The SunView Model **9**

2.1. Objects 9

Window Objects 11

Other Visual Objects 11

2.2. Examples of the use of Objects by Applications 12

2.3. Windows 16

Frames 16

Manipulating Frames Via Menus 18

Subwindows 19

2.4. Input: The Notifier 20

Callback Style of Programming 20

Why a Notification-Based System? 22

Relationship Between the Notifier, Objects, and the Application 22

Calling the Notifier Directly 24

Chapter 3 Interface Outline **27**

Row/Column Space 49

4.8. Attribute Ordering 51

4.9. File Descriptor Usage 54

Chapter 5 Canvases **57**

5.1. Creating and Drawing into a Canvas 58

5.2. Scrolling Canvases 59

5.3. Canvas Model 60

5.4. Repainting 61

Retained Canvases 61

Non-Retained Canvases 62

The Repaint Procedure 62

Retained vs. Non-Retained 62

5.5. Tracking Changes in the Canvas Size 63

Initializing a Canvas 63

5.6. Automatic Sizing of the Canvas 65

5.7. Handling Input in Canvases 66

Default Input Mask 66

Writing Your Own Event Procedure 66

Translating Events from Canvas to Window Space 66

Border Highlighting 67

5.8. Color in Canvases 69

Setting the Colormap Segment 69

Color in Retained Canvases 69

Color in Scrollable Canvases 69

Chapter 6 Handling Input **73**

6.1. Input Environment 74

6.2. Events 75

ASCII Events 77

Locator Button Events 77

Locator Motion Events 77

Window Events 78

7.5. Color ... 103
Introduction to Color ... 103
The Colormap ... 103
Changing the Colormap ... 104
Colormap Segments ... 104
Background and Foreground ... 104
Default Colormap Segment ... 104
Sharing Colormap Segments ... 105
Example: *showcolor* ... 105
Manipulating the Colormap ... 107
Changing a Window's Colormap Segment ... 107
Cycling the Colormap ... 108
Miscellaneous Utilities ... 108
Using Color ... 109
Cursors and Menus ... 109
Is The Application Running on a Color Display? ... 109
Simulating Grayscale on a Color Display ... 109
Double Buffering ... 110
7.6. Plane Groups and the Sun-3/110 ... 112
SunView and Plane Groups ... 112
suntools and Plane Groups ... 113

Chapter 8 Text Subwindows ... **117**
8.1. Creating a Text Subwindow ... 118
8.2. Writing to a Text Subwindow ... 118
8.3. Editing the Contents of a Text Subwindow ... 119
Removing Characters ... 119
Replacing Characters ... 120
The Editing Log ... 120
8.4. Positioning the Text Displayed in a Text Subwindow ... 121
8.5. Finding a Pattern ... 123
8.6. Marking Positions ... 123
8.7. Setting the Primary Selection ... 125

9.4. Messages 147
9.5. Buttons 148
Button Selection 148
Button Notification 148
Button Image Creation Utility 148
9.6. Choices 150
Displaying Choice Items 150
Choice Selection 152
Choice Notification 152
Choice Value 152
Choice Menus 152
9.7. Toggles 156
Displaying Toggles 156
Toggle Selection 156
Toggle Notification 156
Toggle Value 156
Toggle Menus 158
9.8. Text 159
Displaying Text Items 159
Text Selection 160
Text Notification 160
Writing Your Own Notify Procedure 161
Text Value 162
Text Menus 163
9.9. Sliders 164
Displaying Sliders 164
Slider Selection 164
Slider Notification 164
Slider Value 165
9.10. Painting Panels and Individual Items 166
9.11. Iterating Over a Panel's Items 168
9.12. Panel Item Client Data 168
9.13. Event Handling 169

Menu Item Generate Procedure 204

Menu Generate Procedure 205

Pullright Generate Procedure 207

Notify/Action Procedures 208

Flow of Control in `menu_show()` 209

11.8. Interaction with Previously Defined SunView Menus 211

Using an Existing Menu as a Pullright 211

11.9. Initial and Default Selections 212

11.10. User Customizable Attributes 213

Chapter 12 Cursors **217**

12.1. Creating and Modifying Cursors 218

12.2. Copying and Destroying Cursors 218

12.3. Crosshairs 219

12.4. The Cursor Attributes 220

Chapter 13 Icons **225**

13.1. Using Images Generated With *iconedit* 226

13.2. Modifying the Icon's Image 227

13.3. Loading Icon Images At Run Time 227

Chapter 14 Scrollbars **231**

14.1. Scrolling Model 232

14.2. Use of Scrollbars by Application Users 234

14.3. Creating, Destroying and Modifying Scrollbars 235

14.4. Programmatic Scrolling 237

Chapter 15 The Selection Service **241**

15.1. Getting the Primary Selection 242

15.2. Setting the Primary Selection 242

Chapter 16 The Notifier **245**

16.1. When to Use the Notifier 246

16.2. Restrictions 247

Getting Out 266
16.7. Error Handling 267
Error Codes 267
Handling Errors 267
Debugging 267

Chapter 17 Attribute Utilities **271**
17.1. Character Unit Macros 271
17.2. Creating Reusable Attribute Lists 272
Default Attributes 273

Chapter 18 SunView Interface Summary **277**
Canvas Tables 278
Attributes 278
Functions and Macros 279
Cursor Tables 280
Attributes 280
Functions 282
Data Types 283
Icon Tables 286
Attributes 286
Functions and Macros 287
Input Event Tables 288
Event Codes 288
Event Descriptors 289
Input-Related Window Attributes 290
Menu Tables 291
Attributes 291
Item Attributes 294
Functions 296
Notifier Functions Table 298
Panel Tables 301
Attributes 301

A.7. *dctool* 375

A.8. *typein* 381

A.9. Programs that Manipulate Color 385

coloredit 385

animatecolor 390

A.10. Two gfx subwindow–based programs converted to use SunView 397

bounce 397

spheres 402

Appendix B Sun User Interface Conventions **409**

B.1. Program Names 409

B.2. Frame Headers 409

B.3. Menus 409

Capitalization 409

Menus Showing Button Modifiers 410

Interaction with Standard Menus 410

Enable/Disable Menu Items 410

Multi-Column Menus 410

B.4. Panels 410

Buttons 411

List of Non-Exclusive Choices 411

List of Exclusive Choices 411

Binary Choices 412

Text Items 412

Allocation of Function Between Buttons and Menus 412

B.5. Mouse Button Usage 413

Allocation of Function Between Mouse Buttons 413

Using Mouse Buttons for Accelerators 413

B.6. Cursors 413

B.7. Icons 413

Appendix C Converting SunWindows Programs to SunView **417**

Tables

Table 3-1 Reserved Prefixes 29

Table 4-1 Window Usage Examples 33

Table 4-2 SunView File Descriptor Usage 54

Table 6-1 Event Codes 76

Table 6-2 Macros to Get the Event State 81

Table 6-3 Macros to Set the Event State 82

Table 6-4 Event Descriptors 84

Table 7-1 Sample Colormap to Isolate Planes 111

Table 8-1 `Textsw_status` Values 126

Table 8-2 `Textsw_action` Attributes 130

Table 9-1 Text Item Notification 160

Table 9-2 Return Values for Text Item Notify Procedures 161

Table 9-3 Panel Event Handling Attributes 169

Table 9-4 Panel Action Functions 171

Table 10-1 TTY Subwindow Escape Sequences 182

Table 11-1 Attributes To Add Pre-Existing Menu Items 198

Table 11-2 Menu Item Creation Attributes 200

Table 11-3 User Customizable Menu Attributes 213

Table 18-29 TTY Subwindow Attributes 329
Table 18-30 TTY Subwindow Functions 329
Table 18-31 TTY Subwindow Special Escape Sequences 330
Table 18-32 Window Attributes 331
Table 18-33 Frame Attributes 334
Table 18-34 Window Functions and Macros 336
Table 18-35 Command Line Frame Arguments 338

Table C-1 SunWindows ⇒ SunView Equivalences 424

Figures

Figure 2-1 SunView Objects 10

Figure 2-2 Flow of Control in a Conventional Program 20

Figure 2-3 Flow of Control in a Notifier-based Program 21

Figure 2-4 Flow of Input Events in *iconedit,* a SunView Application 23

Figure 11-1 Display Stage of Menu Processing 209

Figure 11-2 Notification Stage of Menu Processing 210

Figure 14-1 Scrolling Model 233

Figure 14-2 Attributes Controlling Scrollbar Appearance 236

Figure 16-1 Overview of Notification 249

Figure 16-2 Flow of Control in Interposition 258

Preface

Audience for this Manual

This manual is addressed to anyone who is interested in writing SunView programs. It assumes that the reader understands the C programming language. Also, before beginning to write your own programs it is a good idea to spend some time in the SunView environment, using the tools and demonstration programs provided with SunView.[1]

Structure

By convention, manuals fall into three categories, *Tutorial*, *User's Guide*, and *Reference*. This manual is a combination of all three.

Tutorial

The *Using Windows* chapter serves as a tutorial introduction to SunView. Through reading it and typing in and modifying its examples, you will be writing simple SunView programs in the proverbial ''10 minutes to SunView'' timeframe. You can then read the later chapters when you need to incorporate the features they describe into your programs.

User's Guide

This entire manual is the user's guide. Start at the beginning, keep reading, and you will understand the SunView model, how SunView programs work, and how to create and use all the different SunView objects in your own window programs.

Reference

Chapter 18, *SunView Interface Summary*, lists all the attributes of the different SunView objects and packages, and the functions and macros to operate on them. Because of the nature of SunView and its use of an attribute value interface, there are a few simple calls but many, many attributes for them. Hence in practice this is all the reference section you will need on a day-to-day basis.

Further Reading

This manual does not teach you how the SunView window system itself works, only how to make working SunView applications. The former is covered, along with many low-level, esoteric, and complex details, in the *SunView System Programmer's Guide.*

[1] All of this is *optional software* that you may not have installed on your system. Consult *Installing UNIX on the Sun Workstation* for more details.

1

Introduction

Introduction 3

What is SunView? 3

History 4

Release 3.0 4

Release 3.2 4

Code No Longer Supported 5

1 Introduction

What is SunView?

SunView (Sun Visual/Integrated Environment for Workstations) is a system to support interactive, graphics-based applications running within windows. It consists of two major areas of functionality: building blocks for output, and a run-time system for managing input. The building blocks include four types of windows:

- *canvases* on which programs can draw,
- *text subwindows* with built in editing capabilities,
- *panels* containing items such as buttons, choice items, and analog sliders, and
- *tty subwindows* in which programs can be run.

Canvases, text subwindows and panels can be scrolled.

The run time system is based on a central *notifier* which distributes input to the appropriate window, and a *window manager* to manage overlapping windows.

The exchange of data between applications running in separate windows (in the same or separate processes) is facilitated by a *selection service.*

The Sun implementations of graphics standards — CGI and CORE — include extensions to run within windows. See the *SunCGI Reference Manual* and the *SunCore Reference Manual,* respectively, for more information.

Code No Longer Supported

The old SunWindows stacking menu package has been supplanted by the SunView walking menu package, described in Chapter 11 of this document. Programmers should convert their applications to use the new package, as the old package may not be included in future releases.

2

The SunView Model

The SunView Model 9

2.1. Objects 9

Window Objects 11

Other Visual Objects 11

2.2. Examples of the use of Objects by Applications 12

2.3. Windows 16

Frames 16

Manipulating Frames Via Menus 18

Subwindows 19

2.4. Input: The Notifier 20

Callback Style of Programming 20

Why a Notification-Based System? 22

Relationship Between the Notifier, Objects, and the Application 22

Calling the Notifier Directly 24

2
The SunView Model

This chapter introduces the conceptual model presented by SunView, covering such basic concepts as *objects*, *windows* and the *notifier*.

It is important that you understand the material in this chapter before you begin to write SunView applications.

2.1. Objects

SunView is an *object-oriented* system. Think of SunView objects as visual building blocks which you use to assemble the user interface to your application. Different types of objects are provided, each with its particular properties; you employ whatever type of object you need for the task at hand.

The most important class of SunView objects are *windows*. Not all objects are windows, however. Other visual objects include cursors, icons, menus and scrollbars.

Technically, an object is a software entity presenting a functional interface. The implementation of the object is not exposed; you manipulate an object by passing its unique identifier, or *handle*, to its associated functions. The style of programmatic interface resulting from this object-oriented approach is outlined in Chapter 2.

The figure on the following page shows the different types and classes of SunView objects.

Window Objects

Window objects include *frames* and *subwindows*. Frames contain non-overlapping subwindows[1] within their borders. Currently, there are four types of subwindows provided by SunView:

- *Panel Subwindow* — A subwindow containing *panel items.*
- *Text Subwindow* — A subwindow containing text.
- *Canvas Subwindow* — A subwindow into which programs can draw.
- *TTY Subwindow* — a terminal emulator, in which commands can be given and programs executed.

The distinctions between frames and subwindows are explained in more detail in the section entitled *Windows* later in this chapter.

Other Visual Objects

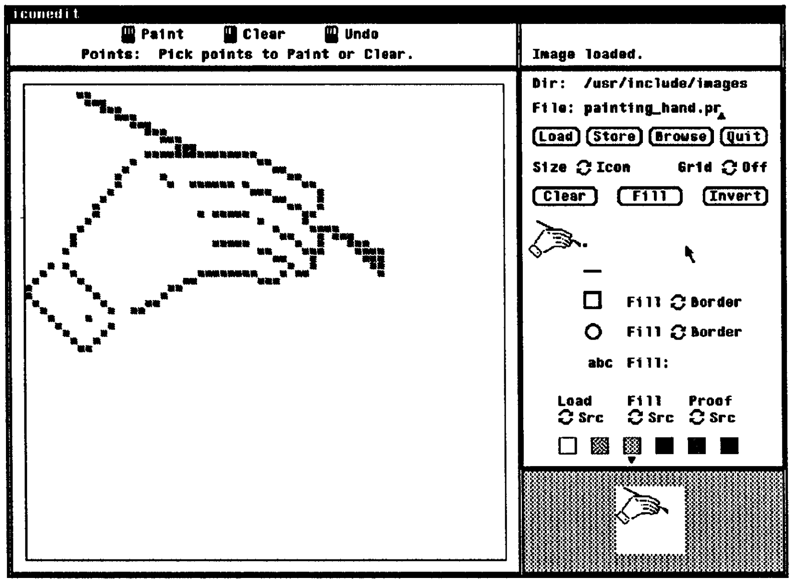
The other types of objects, like windows, are displayed on the screen, but they differ from windows in that they are less general and more tailored to their specific function. They include:

- *Panel Item* — A component of a panel that facilitates a particular type of interaction between the user and the application. Panel items can be moved, displayed or undisplayed under program control. There are several predefined types of items, including *buttons*, *message items*, *choice items*, *text items* and *sliders*.
- *Scrollbar* — An object attached to and displayed within a subwindow through which a user can control which portion of the subwindow's contents are displayed. Both vertical and horizontal scrollbars can be attached to panels and canvases. Text subwindows contain vertical scrollbars by default (they may ~~not contain~~ horizontal scrollbars).
- *Menu* — An object through which a program or a user makes choices and issues commands. By convention in SunView, menus pop up when the user presses the right mouse button. Like windows, menus appear on the screen when needed, and disappear when they have served their purpose. Menus, however, differ from windows in several ways. First, they are more ephemeral — a menu only remains on the screen as long as the menu button remains depressed, in contrast to a window, which remains on the screen until the user indicates he is done or the controlling program explicitly undisplays it. Second, menus are less flexible than windows; they are designed specifically to allow the user to choose from among a list of actions.
- *Cursor* — The object indicating the mouse location on the screen.
- *Icon* — a small (usually 64 x 64 pixel) image representing the application.

The next section gives some examples showing how typical applications make use of SunView objects in their user interface.

[1] It is SunView's window layout policy that enforces non-overlapping subwindows, not some limitation of the system. If you access the window system at a very low level, subwindows can overlap successfully.

The next example shows *iconedit*, a simple bitmap editor for generating images to be used by SunView applications:



Points: Pick points to Paint or Clear.

Image loaded.

Dir: /usr/include/images

File: painting_hand.pr

Size Icon Grid Off

Fill Border

Fill Border

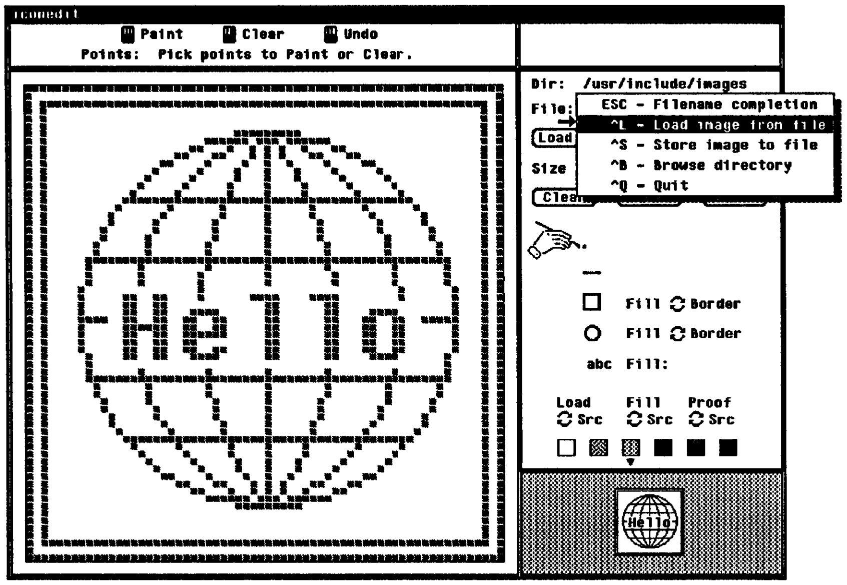
Fill:

Src Src Src

iconedit consists of a frame and five subwindows. From upper left to lower right they are:

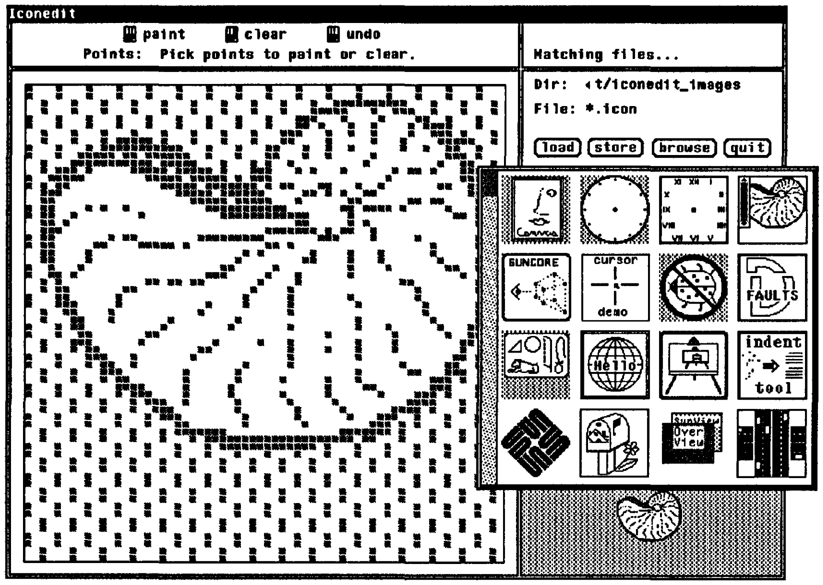
- a panel containing instructions on how to use the mouse;
- a small panel for short messages;
- a canvas for drawing the image;
- a panel containing various items for issuing commands and setting options such as the size of the image being drawn, the drawing mode, etc;
- A small canvas for viewing the icon or cursor actual size.

In the next picture, the user has pressed the menu mouse button over the ''File:'' item in the control panel:



iconedit has displayed a popup menu showing accelerators which the user can type into the item. The purpose of this menu is to both allow the novice user to select the command from the menu and to provide a bridge to more expert use of the application by teaching the use of the accelerators.

iconedit with its popup displayed:


Iconedit

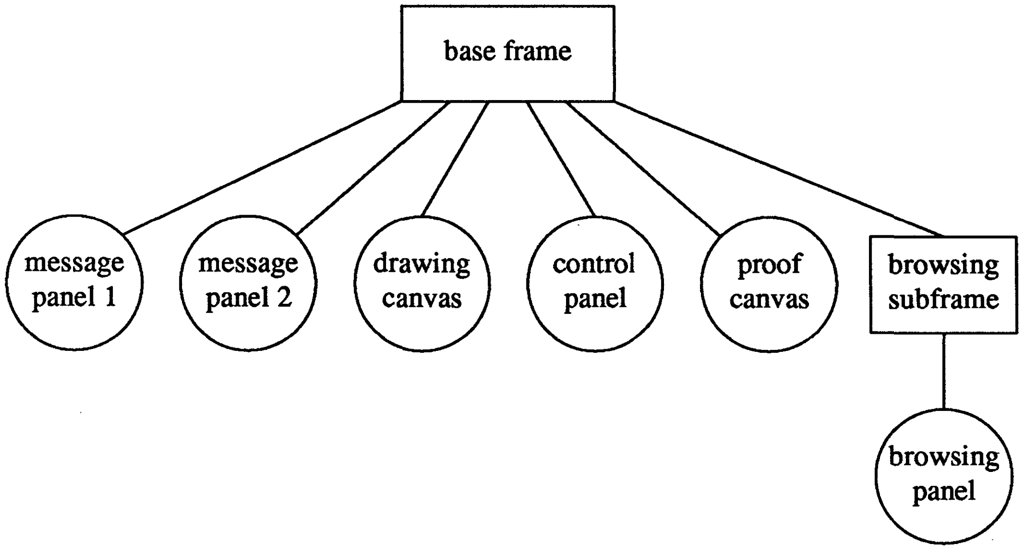
Points: Pick points to paint or clear.

Matching files...

Dir: ‹t/iconedit_images

File: *.icon

The figure below diagrams the structure of *iconedit* as a tree of windows. Frames are shown as rectangles; subwindows as circles:



Both menus contain the *Move*, *Resize*, *Expose*, *Hide* and *Redisplay* commands. *Move* allows the user to change the frame's location. *Resize* allows him to change the window's width and height. *Expose* causes the frame to move in front of the other windows, becoming fully visible on the ''surface'' of the screen, while *Hide* does the opposite, moving the frame behind any other windows occupying the same portion of the screen. *Redisplay* simply causes the window to be displayed again.

When the user is finished working with a base frame he may want to destroy it for good, in which case he would select *Quit*. Or he may want to *Close* the frame, with the anticipation of opening it later and continuing work where he left off. A base frame in its closed state is represented on the screen as a small (usually 64 by 64 pixel) *icon*. The icon is typically a picture indicating the function of the underlying application.

Subframes may not be closed into icons; when the user finishes with a subframe, he simply selects *Done* from the menu. While not destroying the subframe, this causes it to disappear from the screen.

Subwindows

Subwindows differ from frames in several basic ways. Subwindows never exist independently. They are always owned by a frame, and may not themselves own subwindows or subframes. While frames can be moved freely around the screen, subwindows are constrained to fit within the borders of the frame to which they belong. Also in contrast to frames, subwindows are *tiled* — they may not overlap each other within their frame. Within these constraints (which are enforced by a run-time *boundary manager*) subwindows may be moved and resized by either a program or a user.

So far this chapter has discussed the static aspects of the SunView model. The section below outlines the system's model from a dynamic point of view.

Figure 2-3 *Flow of Control in a Notifier-based Program*

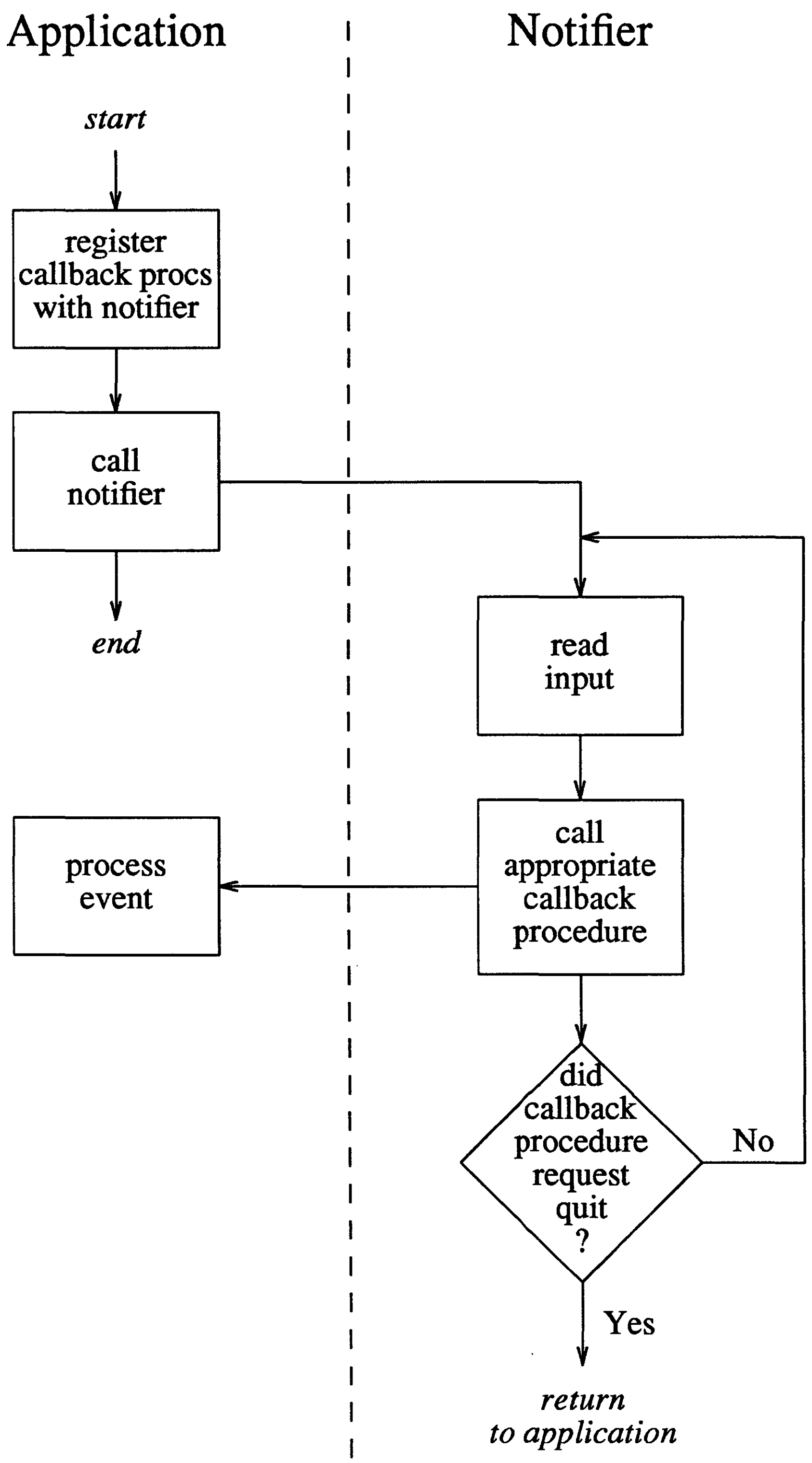
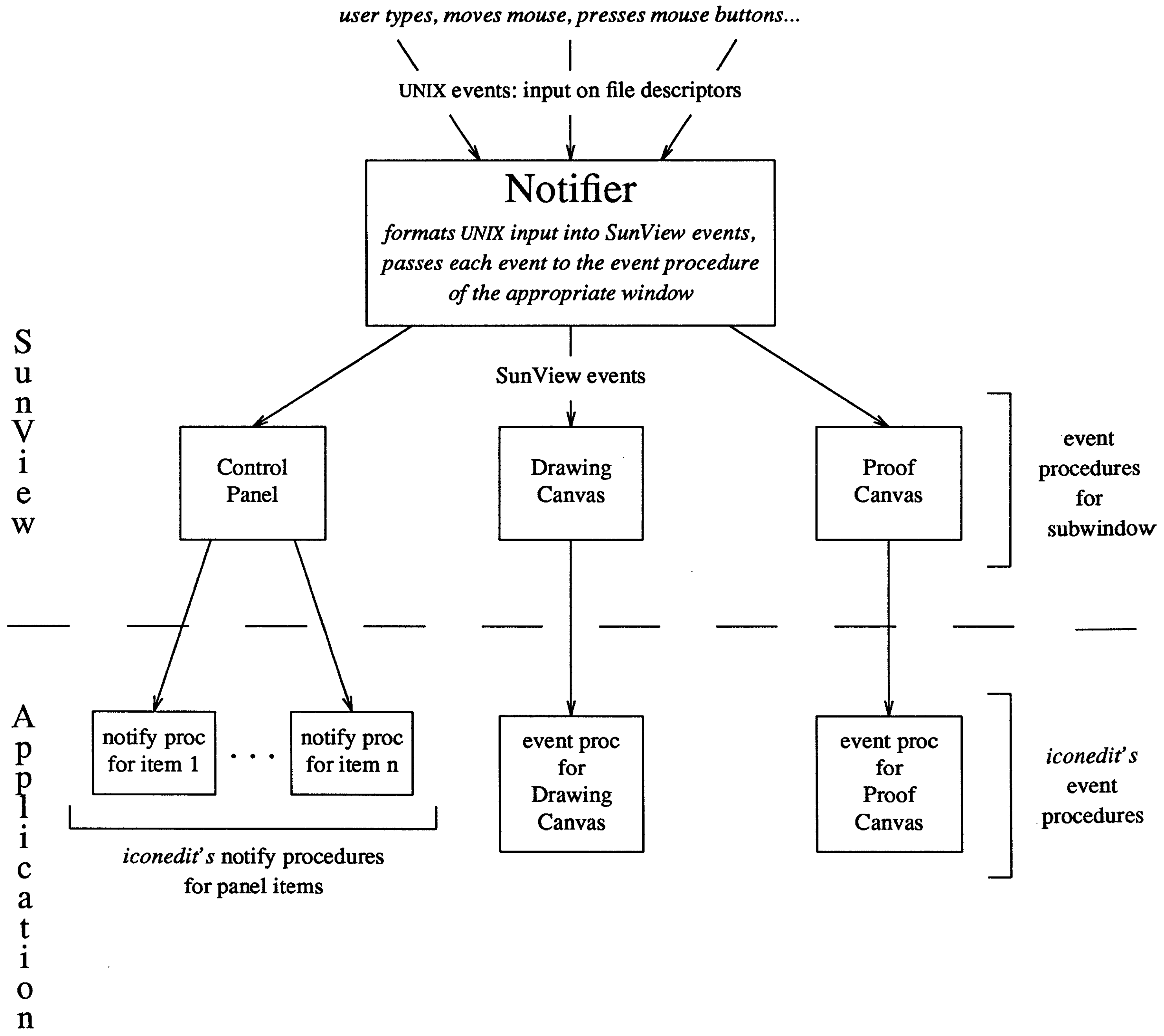

Application
Notifier
start
register callback procs with notifier
call notifier
end
read input
call appropriate callback procedure
process event
did callback procedure request quit ?
No
Yes
return to application

Figure 2-4 *Flow of Input Events in iconedit, a SunView Application*


user types, moves mouse, presses mouse buttons...
UNIX events: input on file descriptors
Notifier
formats UNIX input into SunView events, passes each event to the event procedure of the appropriate window
SunView events
SunView
Control Panel
Drawing Canvas
Proof Canvas
event procedures for subwindow
Application
notify proc for item 1
. . .
notify proc for item n
event proc for Drawing Canvas
event proc for Proof Canvas
iconedit's notify procedures for panel items
iconedit's event procedures

3

Interface Outline

Interface Outline **27**

SunView Libraries 27

Compiling SunView Programs 27

Header Files 27

Object Handles 27

Attribute-based Functions 28

Standard Functions 28

Reserved Namespaces 29

3

Interface Outline

This chapter outlines the SunView interface, the SunView libraries, header files, object handles, attributes and the standard functions applicable to objects of each type.

SunView Libraries

The SunView functions that an application calls are mostly in the library file `/usr/lib/libsuntool.a`. This library includes the code to create and manipulate high-level objects such as frames, panels, scrollbars and icons. These packages in turn call routines in `/usr/lib/libsunwindow.a` to create and manipulate windows and interact with the Notifier. These in turn calls routines in `/usr/lib/libpixrect.a` that do the drawing on the screen.

Compiling SunView Programs

To compile a SunView program you must link in these three libraries, and, because they are built one on top of another, their order is important. For example, to compile a typical SunView application whose source is `testprog.c`, you would type in the command:

```
polar% cc -o testprog testprog.c -lsuntool -lsunwindow -lpixrect
```

Header Files

The basic definitions needed by a SunView application — covering windows, frames, menus, icons and cursors — are obtained by including the header file `<suntool/sunview.h>`. Definitions for the other types of object are found in their own include files — `<suntool/canvas.h>`, `<suntool/text.h>`, `<suntool/panel.h>`, etc.

Object Handles

When you create a SunView object, the creation function returns a *handle* for the object. Later, when you wish to manipulate the object or inquire about its state, you pass its handle to the appropriate function. This reliance on object handles is a way of *information-hiding*. The handles are *opaque* in the sense that you can't ``see through'' them to the actual data structure which represents the object.

Each object type has a corresponding type of handle. The window types of `Frame`, `Canvas`, `Textsw`, `Tty` and `Panel` are grouped under the type `Window`. So, for example, you can declare a panel as either a `Panel` or a `Window`, whichever is most appropriate. The other object types are `Panel_item`, `Menu`, `Scrollbar`, `Cursor`, and `Icon`.

Since C doesn't have an `opaque` type, all the opaque data types mentioned above are `typedef'd` to the UNIX type `caddr_t` (for ``character address

Non-window functions are prefixed with the name of the object. So, to take menus as an example, the standard functions are

- `menu_create()`,
- `menu_get()`,
- `menu_set()`, and
- `menu_destroy()`.

Example

The flavor of the interface is illustrated with the following code fragment, which creates a scrollbar with a width of 10 pixels and a black bubble. Later, the scrollbar's width is changed to 20 pixels. Finally, the scrollbar is destroyed:

```
Scrollbar bar;
bar = scrollbar_create(SCROLL_WIDTH,     10,
                       SCROLL_BAR_COLOR, SCROLL_BLACK,
                       0);
scrollbar_set(bar, SCROLL_WIDTH, 20, 0);
scrollbar_destroy(bar);
```

CAUTION **Note the zero which terminates the attribute lists in the `*_create()` and `*_set()` calls. The most common mistake in using attribute lists is to forget the final zero.** This will not be flagged by the compiler as an error; however, it will cause SunView to generate a run-time error message.

Reserved Namespaces

SunView reserves names beginning with the object types, as well as certain other prefixes, for its own use.

The prefixes listed below should not be used by applications in lower, upper, or mixed case.

Table 3-1 *Reserved Prefixes*

attr_	pixwin_	window_
canvas_	pr_	win_
cursor_	pw_	wmgr_
defaults_	rect_	
ei_	rl_	
es_	scrollbar_	
ev_	scroll_	
event_	seln_	
frame_	textsw_	
icon_	text_	
menu_	toolsw_	
notify_	tool_	
panel_	ttysw_	
pixrect_	tty_	

4

Using Windows

Using Windows 33

4.1. Basic Routines 34

Creating Windows 34

Initiating Event Processing 34

Modifying and Retrieving Window Attributes 34

Destroying Windows 34

4.2. Example: *hello_world* 35

4.3. Example: *simple_panel* 36

Some Frame Attributes 37

Panels 38

Fonts 38

Panel Items 38

4.4. Example: *lister* 39

4.5. Example: *filer* 41

Popups 42

Popup Property Sheet 42

Popup Text Subwindow 43

Popup Confirmer 44

4.6. Example: *image_browser_1* 46

Specifying Subwindow Size 46

Default Subwindow Layout 47

Explicit Subwindow Layout 47

Specifying Subwindow Sizes and Positions 48

4 Using Windows

This chapter describes how to build SunView applications out of frames and subwindows.

The first section presents the basic window routines. Succeeding sections give examples, ranging from the simplest possible application to a moderately useful file manager. For quick reference, the examples are given in the table below:

Table 4-1 *Window Usage Examples*

Example	Description	Illustrates	Page
hello_world	Minimal SunView program.	Compilation, frames.	35
simple_panel	Panel w/message and button.	Basic attributes, panels.	36
lister	Front end to *ls*	Panels, tty subwindows.	39
filer	File manager	Popups, Selection Service.	41
image_browser_1	Displays images	Subwindow layout.	46
image_browser_2	Displays images	Row/column space.	49

Summary Tables

Many window and frame attributes are discussed as they occur in the examples. However, this chapter does not attempt complete coverage of all the attributes. For a comprehensive summary of all attributes see the following tables in Chapter 18, *SunView Interface Summary*:

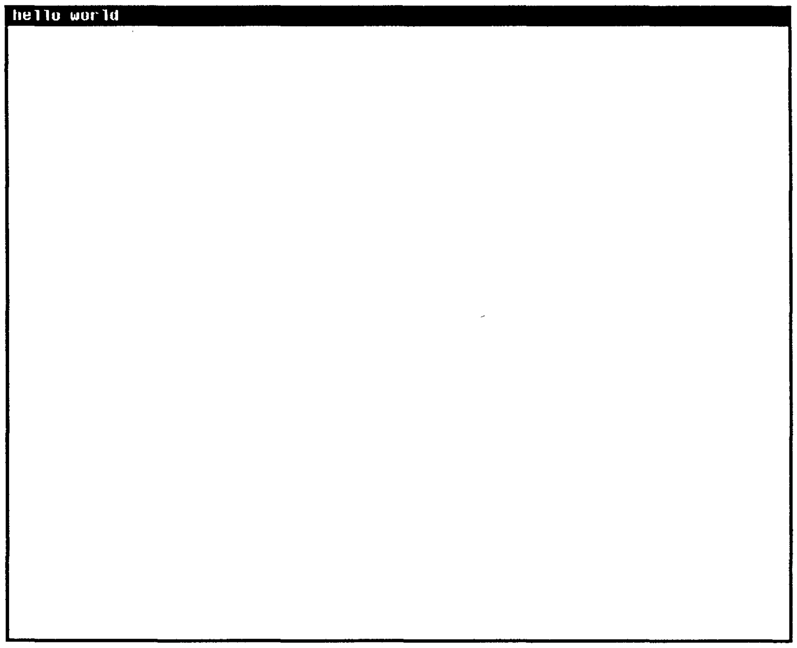
- the *Window Attributes* table begins on page 331;
- the *Frame Attributes* table begins on page 334;
- the *Window Functions and Macros* table begins on page 336;
- the *Command Line Frame Arguments* table begins on page 338.

4.2. Example: *hello_world*

In learning a new programming language or environment, it is useful to begin with a minimal program that simply prints some output. The point of such an exercise is not that the program itself does anything interesting, but that to run it you must master the mechanical details of creating, compiling, loading and running the program. Here's a minimal SunView program:

```
#include <suntool/sunview.h>

main()
{
    Frame frame;
    frame = window_create(0, FRAME, FRAME_LABEL, "hello world", 0);
    window_main_loop(frame);
}
```

The window comes up as shown below — a single frame with the words "hello world" in the frame header:

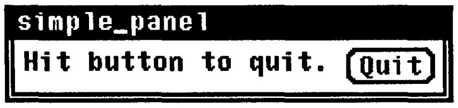


Note that this window is "alive" within the SunView user interface; it can be closed, moved, resized, hidden, etc. When closed, a default icon is displayed, which contains the text from the frame header :

After creating the above program in a file called `hello_world.c`, you would compile it with the command:

```
cc -o hello_world hello_world.c -lsuntool -lsunwindow -lpixrect
```

The code on the previous page creates a frame containing a single panel with a message and a button:



Some Frame Attributes

Below we discuss the attributes in order of appearance.

`FRAME_LABEL`

The string given as the value for `FRAME_LABEL` will appear in a black frame header strip at the top of the frame. To suppress both the label and the frame header, set the attribute `FRAME_SHOW_LABEL` to `FALSE`.

`FRAME_ICON`

The program has used `FRAME_ICON` to specify an icon to be shown when the frame is closed. First, the macro `DEFINE_ICON_FROM_IMAGE()` is used to generate a static icon taking its image from the file `/usr/include/images/hello_world.icon`:

`FRAME_CMDLINE_HELP_PROC`

The next attribute, `FRAME_CMDLINE_HELP_PROC`, takes as its value a function that will be called when the user requests help with the *-WH* command-line argument. This is an example of a *callback procedure*, in which the application registers a function with SunView, and sometime later SunView *calls back* to the registered function.[6] The help function is called with the name by which the program was invoked. In the example, the program's help function `help_proc()` has first printed a message, then called the default frame help function, `frame_cmdline_help()`. (Note that `FRAME_CMDLINE_HELP_PROC` must be specified before `FRAME_ARGS`, or it will be ignored).

`FRAME_ARGS`

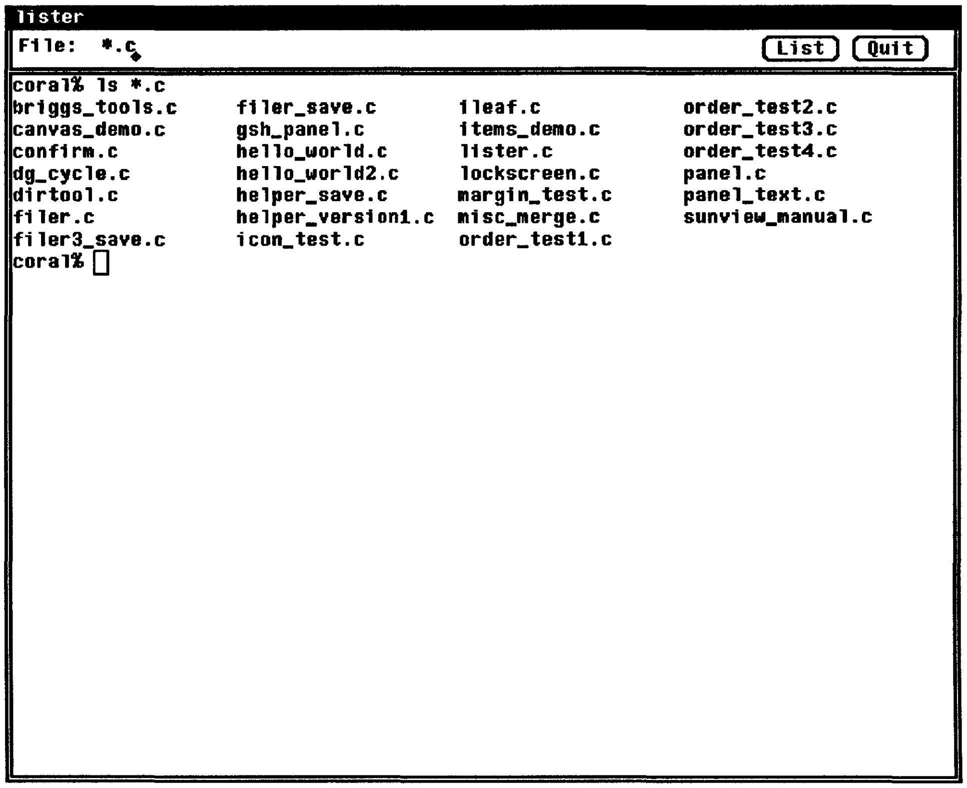
`FRAME_ARGS` is the mechanism by which the application passes command-line arguments given by the user to the frame. There is a set of command line arguments recognized by all frames, allowing the user to control such basic attributes as the dimensions and label of the frame, whether the frame's initial state is open or closed, etc. These arguments begin with *-W*; for a complete list of them see the *Command Line Frame Arguments* table in Chapter 18.[7]

[6] The callback model is described in Section 2.4, *Input: The Notifier.*

[7] As an alternative to `FRAME_ARGS`, you can use `FRAME_ARGC_PTR_ARGV`, which takes a pointer to argc, rather than argc itself. This attribute causes `window_create()` to strip all arguments beginning with *-W* out of argv, and decrement argc accordingly.

4.4. Example: *lister*

Now let's begin to develop a program that actually does something useful. A good candidate is a tool to help manage files. The first version simply lets the user list files in the current directory:



```
coral% ls *.c
briggs_tools.c      filer_save.c        lleaf.c             order_test2.c
canvas_demo.c       gsh_panel.c         items_demo.c        order_test3.c
confirm.c           hello_world.c       lister.c            order_test4.c
dg_cycle.c          hello_world2.c      lockscreen.c        panel.c
dirtool.c           helper_save.c       margin_test.c       panel_text.c
filer.c             helper_version1.c   misc_merge.c        sunview_manual.c
filer3_save.c       icon_test.c         order_test1.c
coral% 
```

The tool presents two subwindows. At the top is a control panel with a text item for the specification of the files to be listed, a List button, and a Quit button.

Below the control panel is a tty subwindow. When the user hits the List button, the program constructs a command string consisting of the string "ls ", followed by the value of the **File:** item, followed by a newline, and inputs the command string to the tty subwindow by calling `ttysw_input()`.

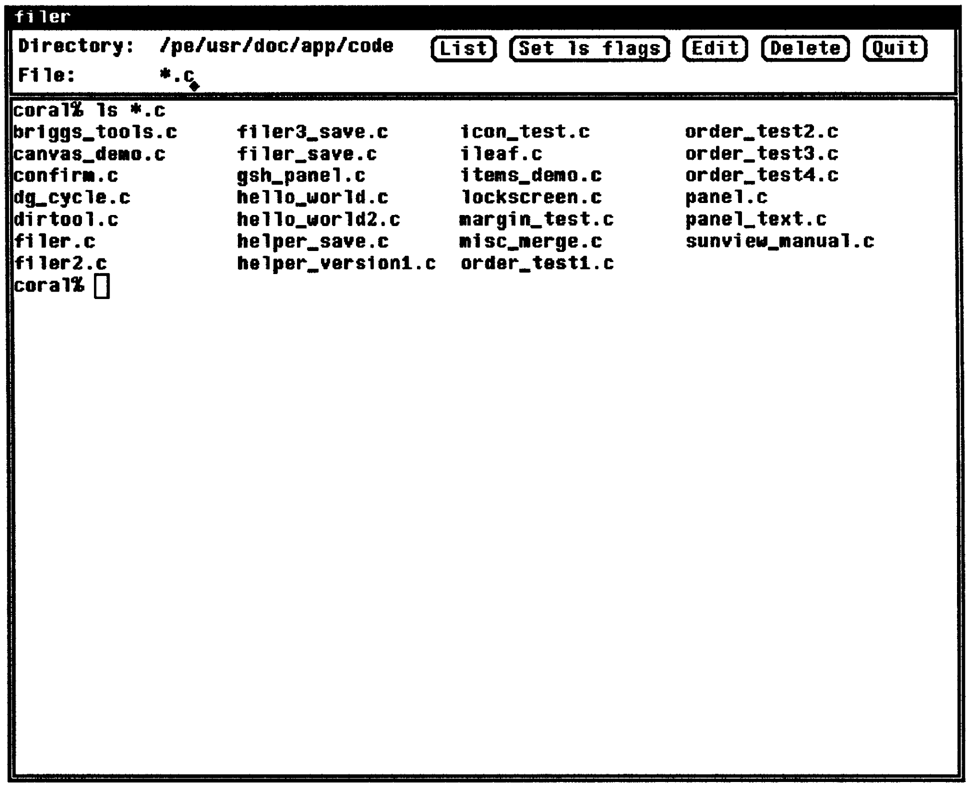
The program is listed in its entirety on the following page.

Note that the frame, the panel and the tty subwindow are all declared as type `Window`. They could just as well have been declared as type `Frame`, `Panel` and `Tty`.

4.5. Example: *filer*

Our next example builds on the simple front end to *ls* given in the previous example to create a more interesting file manipulation tool. This application illustrates the use of the text subwindow, the Selection Service, and *popups* — windows that appear on the screen and disappear dynamically during execution of a program.

In appearance, *filer* is similar to *lister*, with a control panel and tty subwindow. The user specifies the directory and file, and hits the [List] button, causing the *ls* command to be sent to the tty subwindow:



```
coral% ls *.c
briggs_tools.c    filer3_save.c       icon_test.c      order_test2.c
canvas_demo.c     filer_save.c        ileaf.c          order_test3.c
confirm.c         gsh_panel.c         items_demo.c     order_test4.c
dg_cycle.c        hello_world.c       lockscreen.c     panel.c
dirtool.c         hello_world2.c      margin_test.c    panel_text.c
filer.c           helper_save.c       misc_merge.c     sunview_manual.c
filer2.c          helper_version1.c   order_test1.c
coral% 
```

There are three new buttons, each of which illustrates a typical use of popups:

[Set ls flags] a popup *property sheet* for setting options to *ls*;

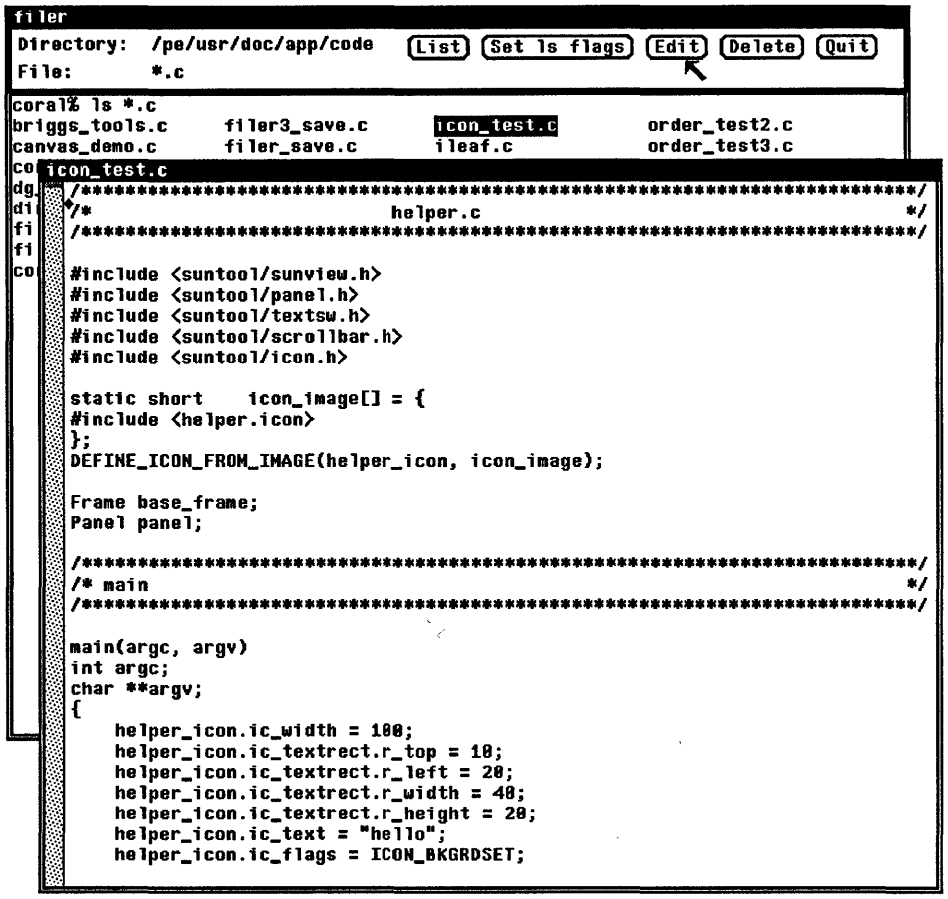
[Edit] a popup text subwindow for browsing and editing files;

[Delete] a popup *confirmer* which forces the user to confirm or cancel.

The three buttons are discussed in the pages that follow. The discussion makes reference to specific routines from the program, which is listed in its entirety as *filer* in Appendix A, *Example Programs*.

Popup Text Subwindow

The [Edit] button gives another illustration of a non-blocking popup. When the user selects a filename and presses the button, a popup text subwindow containing the file appears:



```
Directory:  /pe/usr/doc/app/code
File:       *.c
```

```
coral% ls *.c
briggs_tools.c      filer3_save.c       icon_test.c         order_test2.c
canvas_demo.c       filer_save.c        ileaf.c             order_test3.c
co
dg
di
fi
fi
co
```

```
/*****************************************************************************/
/*                              helper.c                                     */
/*****************************************************************************/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/textsw.h>
#include <suntool/scrollbar.h>
#include <suntool/icon.h>

static short    icon_image[] = {
#include <helper.icon>
};
DEFINE_ICON_FROM_IMAGE(helper_icon, icon_image);

Frame base_frame;
Panel panel;

/*****************************************************************************/
/* main                                                                      */
/*****************************************************************************/

main(argc, argv)
int argc;
char **argv;
{
    helper_icon.ic_width = 100;
    helper_icon.ic_textrect.r_top = 10;
    helper_icon.ic_textrect.r_left = 20;
    helper_icon.ic_textrect.r_width = 40;
    helper_icon.ic_textrect.r_height = 20;
    helper_icon.ic_text = "hello";
    helper_icon.ic_flags = ICON_BKGRDSET;
```

Both the subframe and text subwindow for the popup are created at initialization time with the calls:

```
edit_frame = window_create(base_frame, FRAME,
                           FRAME_SHOW_LABEL, TRUE,
                           0);
editsw = window_create(edit_frame, TEXTSW, 0);
```

When the user hits the [Edit] button the notify procedure `edit_proc()` is invoked. This function first calls the Selection Service to get the name of the file the user has selected.[12] Then it loads the file into the text subwindow, sets the frame header to the filename, and displays the frame with these two calls:

```
window_set(editsw, TEXTSW_FILE, filename, 0);
window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
```

[12] The routine to get the selection is given in Chapter 15. The Selection Service is described in detail in Chapter 9 of the *SunView System Programmer's Guide*.

```
caddr_t
window_loop(subframe)
    Frame subframe;

void
window_return(return_value)
    caddr_t return_value;
```

`window_loop()` causes the popup to be displayed and receive all input directed to the screen. The call will not return until `window_return()` is called from one of the popup's notify procedures. The value passed to `window_return()` as `return_value` will be returned by `window_loop()`. Its interpretation is up to the application — it may be used to indicate whether the command was confirmed, whether a valid file name was entered, etc.

Default Subwindow Layout

The default subwindow layout algorithm is extremely simple. The first subwindow is placed at the upper left corner of the frame (leaving space for the frame's header and a border). If the width of the previously-created subwindow is fixed (i.e. not extend-to-edge), the next subwindow is placed to the right of it. If the width of the previously-created subwindow *is* extend-to-edge, the next subwindow is placed below it, at the left of the frame.

Explicit Subwindow Layout

This default layout algorithm handles only very simple topologies. SunView provides attributes to allow you to specify more complex layouts by explicitly positioning subwindows. You can position one subwindow relative to another by using `WIN_BELOW` and `WIN_RIGHT_OF`. These attributes take as their value the handle of the subwindow you want the new subwindow to be below or to the right of.

image_browser_1, pictured on the preceding page, illustrates the use of `window_fit()` along with explicit subwindow positioning to obtain a particular layout. The relevant calls are shown below:

```
tty = window_create(frame, TTY, WIN_ROWS, 20, WIN_COLUMNS, 30, 0);

control_panel = window_create(frame, PANEL, 0);

(create panel items...)

window_fit(control_panel);

display_panel = window_create(frame, PANEL,
                              WIN_BELOW,    control_panel,
                              WIN_RIGHT_OF, tty,
                              0);
window_fit(frame);
```

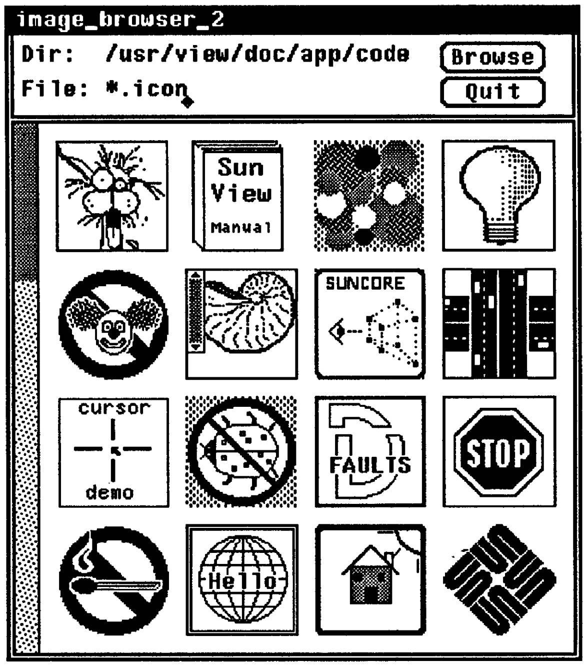
First the tty subwindow is created with a fixed height and width. Then the control panel is created, with no specification of origin or dimensions. Since the width of the previous subwindow was fixed, the control panel is placed by default just to the right. After its items are created, the control panel is shrunk around its items in both dimensions with `window_fit()`. Then the display panel is created and explicitly positioned below the control panel and to the right of the tty subwindow. Both dimensions of the display panel default to `WIN_EXTEND_TO_EDGE`. Finally, `window_fit()` is called to shrink the frame to the width of the control panel and the height of the display panel.[14]

NOTE One thing to watch out for is that `WIN_BELOW` only affects the subwindow's *y* dimension, and `WIN_RIGHT_OF` only affects the *x* dimension.

[14] `window_fit()` causes the window to shrink until it encounters the first fixed border. Subwindows which are extend-to-edge don't stop the shrinking.

4.7. Example: *image_browser_2*

In the next example, when the user specifies a filename and presses [Browse] the images in the files are displayed in a scrollable panel:



The point of this example is to illustrate how you can use *row/column space* to specify the size of a subwindow. The goal was to make the panel just the right size to display a single page of icons, with four rows, four columns, and 10 pixels of white space around each icon.

Row/Column Space

Row/column space refers to a logical grid defining the rows and columns of a window. You can define the row/column space for a window by using the attributes in the following table:

Attribute	Description	Default	Def. in Panels
WIN_BOTTOM_MARGIN	Bottom margin.	0	(same)
WIN_COLUMN_GAP	Space after columns.	0	(same)
WIN_COLUMN_WIDTH	Width of a column.	Width of WIN_FONT.	(same)
WIN_LEFT_MARGIN	Left margin.	5	4
WIN_RIGHT_MARGIN	Right margin.	5	0
WIN_ROW_GAP	Space after rows.	0	5
WIN_ROW_HEIGHT	Height of a row.	Height of WIN_FONT	(same)
WIN_TOP_MARGIN	Top margin.	5[17]	4

[17] In frames with headers, the default for WIN_TOP_MARGIN depends on the system font. With the default system font, it defaults to 17.

4.8. Attribute Ordering

The general rule is that attributes in SunView are evaluated in the order they are given. The following two examples of text subwindow calls illustrate how giving the same attributes in different orders can produce different effects:

```
window_set(textsw, TEXTSW_FILE, "file_1", 0);
window_set(textsw, TEXTSW_FIRST, 20, TEXTSW_FILE, "file_2", 0);

window_set(textsw, TEXTSW_FILE, "file_1", 0);
window_set(textsw, TEXTSW_FILE, "file_2", TEXTSW_FIRST, 20, 0);
```

In the first pair of calls, the index is first set to the 20th character of `file_1`, then `file_2` is loaded, starting at the zeroth character. The second pair of calls first loads `file_2`, then sets the index in `file_2` to 20.

Command-line Arguments

The attribute `FRAME_ARGS` bears special mention. As described in the second example in this chapter, *simple_panel*, this attribute causes the frame to process the command-line arguments given by the user at run time. Some of these arguments correspond to attributes that can be set programmatically — for example, `-Wh` corresponds to `WIN_ROWS`.[19]

The basic rule — that attributes are evaluated in the order given — applies equally to attributes that are explicitly specified in the program and to those that are specified at run time via their command-line equivalents. If a given attribute is specified more than once, the last setting is the one that takes effect, so you can control whether your application or the user will have the last word by specifying attributes after or before `FRAME_ARGS`.

Let's take a couple of examples:

```
window_create(0, FRAME,
                FRAME_ARGS,   argv, argc,
                FRAME_LABEL, "LABEL FROM PROGRAM",
                WIN_ROWS,     10,
                0);

window_create(0, FRAME,
                FRAME_LABEL, "LABEL FROM PROGRAM",
                WIN_ROWS,     10,
                FRAME_ARGS,   argv, argc,
                0);
```

Assume that the program was invoked with a command line containing the following arguments:

```
-Wl "LABEL FROM COMMAND-LINE" -Wh 4
```

[19] For a complete list of these arguments see the *Command Line Frame Arguments* table in Chapter 18.

The Panel Package

The panel package deviates from the norm in that its attributes are generally not order-dependent. For example, you can specify the label of an item before the font, and the font will be used even though it appears after the label.

The only thing to watch out for is that you can't change the font in a single call, as in:

```
panel_set(text_item,
         PANEL_FONT,         font_1,
         PANEL_LABEL_STRING, "Label:",
         PANEL_FONT,         font_2,
         PANEL_VALUE,        "initial value",
         0);
```

The above call will cause both the label and the value for `text_item` to be rendered in `font_2`.

5

Canvases

Canvases 57

5.1. Creating and Drawing into a Canvas 58
5.2. Scrolling Canvases 59
5.3. Canvas Model 60
5.4. Repainting 61
Retained Canvases 61
Non-Retained Canvases 62
The Repaint Procedure 62
Retained vs. Non-Retained 62
5.5. Tracking Changes in the Canvas Size 63
Initializing a Canvas 63
5.6. Automatic Sizing of the Canvas 65
5.7. Handling Input in Canvases 66
Default Input Mask 66
Writing Your Own Event Procedure 66
Translating Events from Canvas to Window Space 66
Border Highlighting 67
5.8. Color in Canvases 69
Setting the Colormap Segment 69
Color in Retained Canvases 69
Color in Scrollable Canvases 69

5 Canvases

The most basic type of subwindow provided by SunView is the *Canvas*. A canvas is essentially a window into which you can draw.

For a demonstration of the various canvas attributes, run the program */usr/demo/canvas_demo*. For examples of canvases that illustrate event handling, run the image editor *iconedit*. *iconedit* uses two canvases, the large drawing canvas on the left, and the small proof area on the lower right.

In order to use canvases you must include the header file `<suntool/canvas.h>`.

Summary Tables

Tables that summarize canvas attributes, functions and macros are in Chapter 18, *SunView Interface Summary*:

- the *Canvas Attributes* table begins on page 278;
- the *Canvas Functions and Macros* table begins on page 279.

5.2. Scrolling Canvases

Many applications need to view and manipulate a large object through a smaller viewing window. To facilitate this SunView provides *scrollbars*, which can be attached to subwindows of type canvas, text or panel.

Example 2:

The code below creates a canvas that is scrollable in both directions:

```
frame  = window_create(NULL, FRAME, 0);
canvas = window_create(frame, CANVAS,
            CANVAS_AUTO_SHRINK,         FALSE,
            CANVAS_WIDTH,               1000,
            CANVAS_HEIGHT,              1000,
            WIN_VERTICAL_SCROLLBAR,     scrollbar_create(0),
            WIN_HORIZONTAL_SCROLLBAR,   scrollbar_create(0),
            0);
```

The distinction between the dimensions of the *canvas* and of the *window* is important. In the above example, we set the canvas width and height to 1000 pixels. Since the dimensions of the canvas subwindow (i.e. `WIN_WIDTH` and `WIN_HEIGHT`) were not explicitly set, the subwindow extends to fill the frame. The frame's dimensions, in turn, were not explicitly set, so it defaults to 25 lines by 80 characters in the default font. The result is a logical canvas roughly the area of the screen, which is viewed through a window about one fourth that size.

NOTE It is necessary to explicitly disable the ''auto-shrink'' feature in the above example. If this were not done, the canvas size would be truncated to the size of the window. See Section 5.6, *Automatic Sizing of the Canvas*.

The Canvas

Think of the canvas itself as a logical surface on which you can draw. The width and height of the canvas are set via the attributes `CANVAS_WIDTH` and `CANVAS_HEIGHT`. So the coordinate system is as shown in the diagram on the previous page, with the origin at the upper left corner and the point (`CANVAS_WIDTH-1, CANVAS_HEIGHT-1`) at the lower right corner. Note that the logical canvas origin is always at (0, 0).

The Canvas Pixwin

As mentioned above, you draw on the canvas by writing into the canvas pixwin, which is retrieved via the `CANVAS_PIXWIN` attribute or the `canvas_pixwin()` macro.

The canvas pixwin is set up to take scrolling into account by performing the transformation from your canvas coordinate system to its pixwin coordinate system. So when you draw into the canvas pixwin using the `pw_*` functions you don't have to do any mapping yourself — the arguments you give should be in the canvas coordinate system.

Between the frame border and the canvas pixwin is a margin, set via the attribute `CANVAS_MARGIN`. This margin defaults to zero pixels, so in the simple case, the canvas pixwin occupies the entire inner area of the window pixwin. If one or more scrollbars are present, the canvas margin begins at the inside border of the scrollbar.

Note the distinction between the pixwin of the canvas (attribute `CANVAS_PIXWIN`) and the pixwin of the window (attribute `WIN_PIXWIN`). The canvas pixwin is one of several regions of the window's pixwin, which also includes the regions occupied by the scrollbars and the margin.

The canvas package manages the canvas pixwin for you. In particular, the clipping list is restricted to the area of the canvas pixwin actually backed by the canvas. This means that you can never draw off the edge of the canvas. For example, if you have set the canvas height to be less than the height of the canvas pixwin, any `pw_*` operations that attempt to draw below the canvas height will be clipped away.

5.4. Repainting

By default, canvases are *retained* — i.e. the canvas package maintains a copy of the bits on the screen in a *backing pixrect*, from which it automatically repaints the screen image when necessary. If you wish to handle repainting yourself, you can defeat this feature.

Retained Canvases

The Canvas Package allocates a backing pixrect the size of the logical canvas. When the canvas width or height changes, a new backing pixrect of the proper dimensions is allocated, the contents of the old pixrect are copied into the new pixrect, and the old pixrect is freed.

5.5. Tracking Changes in the Canvas Size

The client's resize procedure is called whenever the canvas width or height changes. Its form is:

```
sample_resize_proc(canvas, width, height)
    Canvas      canvas;
    int         width;
    int         height;
```

NOTE *You should never repaint the image in the resize procedure, since if there is any new area to be painted, the repaint procedure will be called later.*

There are some subtle points to be aware of related to whether or not the image is fixed size (`CANVAS_FIXED_IMAGE` is `TRUE`). In the default case the image is fixed size, and the repaint procedure will not be called when the canvas gets smaller, since there will be no new canvas area to be repainted. If the image is *not* fixed size, then whenever the canvas size changes, the canvas package assumes that the entire canvas needs to be repainted, and the repaint area will contain the entire canvas.

Initializing a Canvas

Neither the repaint procedure nor the resize procedure will be called until the canvas subwindow has been displayed at least once. This allows you to create and initialize a canvas without having to deal with the resize/repaint procedures. The very first time the canvas is displayed, the resize procedure will be called with the current canvas size. This initial call to the resize procedure allows you to synchronize with the canvas size.

There are several points to note from the example on the previous page. First, since the width and height of the canvas are not specified, they default to the width and height of the window. Second, since the image being drawn is dependent on the size of the canvas, we set `CANVAS_FIXED_IMAGE` to `FALSE`. Third, when the repaint proc is called, we don't bother to draw the specified repaint area, instead we rely on the clipping list to be restricted correctly and simply redraw the entire image.

5.6. Automatic Sizing of the Canvas

Two attributes requiring some explanation are `CANVAS_AUTO_EXPAND` and `CANVAS_AUTO_SHRINK`. Setting both these attributes to `TRUE` allows you to have a drawing area which automatically tracks the size of the window.

If `CANVAS_AUTO_EXPAND` is `TRUE`, the canvas width and height are never allowed to be less than the edges of the canvas pixwin. For example, if you try to set `CANVAS_WIDTH` to a value which is smaller than the width of the canvas pixwin, the value will be automatically expanded (rounded up) to the width of the canvas pixwin.

The main use of `CANVAS_AUTO_EXPAND` is to allow the canvas to grow bigger as the user stretches the window. For example, if the canvas starts out exactly the same size as the canvas pixwin, and the user stretches the window, the canvas pixwin will get bigger, which will cause the canvas itself to expand.

Another point to keep in mind is that whenever you set `CANVAS_AUTO_EXPAND` to `TRUE`, the canvas will be expanded to the edges of the canvas pixwin (if it is smaller to begin with).

`CANVAS_AUTO_SHRINK` is symmetrical to `CANVAS_AUTO_EXPAND`. If `CANVAS_AUTO_SHRINK` is `TRUE`, the canvas width and height are never allowed to be greater than the edges of the canvas pixwin.

Border Highlighting

The SunView convention is that a subwindow indicates that it is accepting keyboard events by highlighted its border. By default, canvas subwindows do not enable any keyboard events, so the border is not highlighted. However, if you explicitly enable keyboard events, by consuming `WIN_ASCII_EVENTS`, the canvas package will highlight the canvas border when it is given the input focus.

5.8. Color in Canvases

You can use color in canvases by specifying a colormap segment for the canvas with the colormap manipulation routines described in Chapter 7.

Setting the Colormap Segment

The first thing to note is that since the canvas pixwin is a region of the `WIN_PIXWIN`, you must also set the colormap segment for the canvas pixwin.

Color in Retained Canvases

If the canvas is retained, then the colormap segment must be set *before* `CANVAS_RETAINED` is set to `TRUE`. This is because the canvas package will determine the depth of the backing pixrect based on depth of the colormap segment defined for the `WIN_PIXWIN`. (If the colormap segment depth is greater than two, then the full depth of the display will be used. Otherwise, the backing pixrect depth will be set to one.)

Since the depth of the backing pixrect is determined when the canvas is created, you must create the canvas with `CANVAS_RETAINED FALSE`, then set the colormap segment, then set `CANVAS_RETAINED` to `TRUE`.

Color in Scrollable Canvases

If the canvas has scrollbars, you need to attach the scrollbars to the canvas *after* the colormap segment has been changed. If the canvas has already been created with scrollbars attached, you should change the colormap, then re-attach the scrollbars. This will insure that the scrollbar pixwin regions use the new colormap segment.

6

Handling Input

Handling Input 73

6.1. Input Environment 74
6.2. Events 75
ASCII Events 77
Locator Button Events 77
Locator Motion Events 77
Window Events 78
Function Key Events 79
Shift Key Events 80
Other Events 80
6.3. Querying and Setting the Event State 81
6.4. Input Distribution 82
Input Focus 82
Input Masks 83
Selection of the Input Recipient 83
6.5. Enabling and Disabling Events 84
Which Mask to Use 85
Setting the Input Mask as a Whole 86
Querying Input Mask State 86
6.6. Releasing the Event Lock 87
6.7. Reading Events Explicitly 87

6 Handling Input

Material Covered

This chapter discusses the SunView input paradigm, including:

- the basic *event* construct;
- the various classes of events —ASCII, function keys, locator buttons, locator motion, window generated events, and so on;
- how to query the state of an event;
- the input focus model distinguishing between *pick* and *keyboard* focuses;
- control of input distribution via *input masks*;
- how to explicitly read events.

While the material in this chapter applies to the window system as a whole, it is of most immediate interest to clients of canvases, who typically will want to handle events themselves.

Header Files

The definitions necessary to use SunView's input facilities are in the header file `<sunwindow/win_input.h>`, which is included by `<sunwindow/window_hs.h>`, which in turn is included by default when you include `<suntool/sunview.h>`.

Related Documentation

The chapter titled *Workstations* in the *SunView System Programmer's Guide* explains the input system at a lower level, covering such topics as how to add user input devices to SunView.

Summary Tables

Tables that summarize input functions and related attributes are in Chapter 18, *SunView Interface Summary*:

- the *Event Codes* table begins on page 288;
- the *Event Descriptors* table begins on page 289;
- the *Input-Related Window Attributes* table begins on page 290.

6.2. Events

Each user action generates an input event, which is passed to your event procedure as an `Event` pointer (type `Event *`). Information encoded as part of the event includes:

- an identifying code, accessed via the macro `event_id();`
- the location of the event in the window's coordinate system, accessed via `event_x()` and `event_y()`.
- a timestamp, accessed via `event_time()`.

Each event code corresponds to some item from a *Virtual User Input Device* (VUID) interface, thus isolating the application from dependence on any particular hardware input device.[26] The VUID appears as an extended keyboard, different from existing keyboards but incorporating the common features of most of them. It also incorporates a *locator* which indicates a screen position. So that you can get window-related input, a window is also thought of as an input device (the fact that a window was entered by the locator, for example, is treated as an event).

Event codes can take on any value in the range 0 through 65535. The values are useful when debugging. The table on the next page lists the predefined event codes and their values.

[26] It is possible to bypass the VUID and receive unencoded events. Refer to the section on *Unencoded Input* in Chapter 7 of the *SunView System Programmer's Guide*.

The next several subsections discuss the different classes of events. When reading about a particular event or class of events, keep in mind that **a window will only receive an event if its input mask has been set to let the event through.** The section on *Enabling and Disabling Events* later in the chapter describes how to control a window's input mask.

ASCII Events

The event codes in the range 0 to 255 inclusive are assigned to the ASCII event class, which includes the standard 7-bit ASCII codes and their 8-bit META counterparts.

Striking a key which has an obvious ASCII meaning — i.e. a key in the main typing array labeled with a single letter — causes the VUID to enqueue for the appropriate window an event whose code is the corresponding 7-bit ASCII character.

The META event codes (128 through 255) are generated when the user strikes a key that would generate a 7-bit ASCII code while the META key is also depressed. In this case, the event code is 128 (`META_FIRST` in the table on the previous page) plus the 7-bit ASCII code.

Locator Button Events

The standard Sun locator is a three button mouse, whose buttons generate the event codes `MS_LEFT`, `MS_MIDDLE` and `MS_RIGHT`.

In general, a physical locator can have up to 10 buttons connected to it. In some cases, while the locator itself may not have any buttons on it, it may have buttons from another device assigned to it. A light pen is an example of such a locator. In any case, each of the buttons associated with the VUID's locator are assigned an event code; the *i-th* button is assigned the code `BUT(i)`.

Thus the event codes `MS_LEFT`, `MS_MIDDLE` and `MS_RIGHT` correspond to `BUT(1)`, `BUT(2)` and `BUT(3)`.

Locator Motion Events

The physical locator constantly provides an (x, y) coordinate position in pixels; this position is transformed by SunView to the coordinate system of the window receiving an event. Locator motion event codes include `LOC_MOVE`, `LOC_DRAG`, `LOC_TRAJECTORY` and `LOC_STILL`.

Since the locator tracking mechanism reports the current position at a set sampling rate — 40 times per second — fast motions will yield non-adjacent locations in consecutive events.

A `LOC_MOVE` event is reported when the locator moves, regardless of the state of the locator buttons. If you only want to know about locator motion when a button is down, enable `LOC_DRAG` instead of `LOC_MOVE`. This will greatly reduce the number of motion events that your application has to process.

When you enable `LOC_MOVE` or `LOC_DRAG` the window system gives you the most current locator position it can, by collapsing consecutive locator motion events into one. This is appropriate for applications such as dragging an image from one point to another, in which the important thing is to keep up with the mouse cursor. On the other hand, for applications in which each point on the cursor trajectory is of interest, such as a program which lets the user draw, it is section of code that might, from the user's perspective, take a long time. If your SIGURG handler is called, set the stop flag and return. In the code that is taking a long time, query the stop flag whenever convenient. When you notice that the stop flag has been set, read an event (see Section 6.6, *Reading Events Explicitly*) — if it is a `WIN_STOP`, then gracefully terminate your long operation.

Function Key Events

The function keys in the VUID define an idealized standard layout that groups keys by location: 15 left, 15 right, 15 top and 2 bottom.[28]

The event codes associated with the function keys are `KEY_LEFT(i)`, `KEY_RIGHT(i)` and `KEY_TOP(i)`, where i ranges from 1 to 15. If you specifically ask for a function key event code then that event code will be passed to your event procedure.

The standard SunView function keys map to event codes as follows:

SunView Function Key	Event Code
Stop	`KEY_LEFT(1)`
Again	`KEY_LEFT(2)`
Props	`KEY_LEFT(3)`
Undo	`KEY_LEFT(4)`
Expose	`KEY_LEFT(5)`
Put	`KEY_LEFT(6)`
Open	`KEY_LEFT(7)`
Get	`KEY_LEFT(8)`
Find	`KEY_LEFT(9)`
Delete	`KEY_LEFT(10)`

If you *don't* specifically ask for a given function key event code, then when the user presses that function key you will get an escape sequence instead of the function key event code (assuming ASCII events have been enabled). For physical keystations that are mapped to cursor control keys, events with codes that correspond to the ANSI X3.64 7-bit ASCII encoding for the cursor control function are transmitted. For physical keystations mapped to other function keys, events with codes that correspond to an ANSI X3.64 user-definable escape sequence are transmitted.

[28] The actual position of the function keys on a given physical keyboard may differ — see `kbd(5)` for details on various keyboards.

6.3. Querying and Setting the Event State

You can query the state associated with an event via the following macros, all of which take as their only argument a pointer to an `Event`.

Table 6-2 *Macros to Get the Event State*

Macro	Returns
`event_id()`	The identifying code of the event. The codes are discussed in the previous section.
`event_is_up()`	TRUE if the event is a button or key event and the state is up.
`event_is_down()`	TRUE if the event is a button or key event and the state is down.
`event_x()`	The x coordinate of the locator in the window's coordinate system at the time the event occurred.
`event_y()`	The y coordinate of the locator in the window's coordinate system at the time the event occurred.
`event_shiftmask()`	The value of predefined shift-keys (described in `kbd`(5)). Possible values: `#define CAPSMASK 0x0001` `#define SHIFTMASK 0x000E` `#define CTRLMASK 0x0030` `#define META_SHIFT_MASK 0x0040`
`event_time()`	The event's timestamp, formatted as a `timeval` struct, as defined in `<sys/time.h>`.
`event_shift_is_down()`	TRUE if one of the shift keys are down.
`event_ctrl_is_down()`	TRUE if the control key is down.
`event_meta_is_down()`	TRUE if the meta key is down.
`event_is_button()`	TRUE if the event is a mouse button.
`event_is_ascii()`	TRUE if the event is in the ASCII range (0 thru 127).
`event_is_meta()`	TRUE if the event is in the META range (128 thru 255).
`event_is_key_left()`	TRUE if the event is any `KEY_LEFT(i)`.
`event_is_key_right()`	TRUE if the event is any `KEY_RIGHT(i)`.
`event_is_key_top()`	TRUE if the event is any `KEY_TOP(i)`.

In addition to the above macros, which tell about the state of a particular event, you can query the state of any button or key via the `WIN_EVENT_STATE` attribute. For example, to find out whether or not the first right function key is down you would call:

```
k1_down = (int)
      window_get(canvas, WIN_EVENT_STATE, KEY_RIGHT(1));
```

The call will return non-zero if the key is down, and zero if the key is up.

For example, the call

```
window_set(win, WIN_MOUSE_XY, 200, 300, 0);
```

sets the cursor to position (200, 300) and sets the pick focus to `win`.

Input Masks

An input mask specifies which events a window will receive (or *consume*) and which it will ignore. In other words, an input mask serves as a read enable mask. Each window has both a *pick input mask*, to specify which pick related events it wants, and a *keyboard input mask*, to specify which keyboard related events it wants.

When a window is the pick focus, its pick mask is used to screen events. When a window is the keyboard focus, its keyboard mask is used to screen events.

The section on *Enabling and Disabling Events* describes how to specify which events a window will consume and which it will ignore.

Selection of the Input Recipient

The Notifier determines which window will receive a given event according to the following algorithm:

- First, the keyboard input mask for the window which is the keyboard focus is checked to see if it wants the event. If so, it becomes the recipient; otherwise the next test is applied.
- Second, the pick input mask for the window which is under the cursor is checked to see if it wants the event. If several windows are layered under the cursor, the event is tested against the pick input mask of the topmost window. If it wants the event then it becomes the recipient; otherwise the next test is applied.
- If the event does not match the pick input mask of the window under the cursor, the event will be offered to that window's *designee*. By default the *designee* is the window's owner. You can set the designee explicitly by calling `window_set()` using the `WIN_INPUT_DESIGNEE` attribute.[30]

  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.
- Occasionally you may want to specify that a given window is to receive *all* events, regardless of their location on the screen. You can do this by setting the `WIN_GRAB_ALL_INPUT` attribute for the window to `TRUE`.

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 ordered stream of time-stamped input events, containing only the events that window has declared to be of interest.

[30] Note that you must give the `WIN_DEVICE_NUMBER` of the window you wish to be the designee, not its handle. This is to allow specifying windows in another user process as the input designee. So the following call would set `win2` to be the designee for `win1`: `window_set(win1, WIN_INPUT_DESIGNEE, window_get(win2, WIN_DEVICE_NUMBER));`

Which Mask to Use

To enable or disable ASCII events, use the keyboard mask. To enable or disable locator motion and button events, use the pick mask.

Function keys are typically associated with the keyboard mask, but sometimes it makes sense to include some function keys in the pick mask — in effect extending the number of buttons associated with the pick device. For example, in the SunView interface the [Again] [Undo] [Put] [Get] [Delete] and [Find] function keys are associated with the keyboard mask, while the [Stop] [Expose] and [Open] keys are associated with the pick mask.

Examples

The event attributes cause precisely the events you specify to be enabled or disabled — the input mask is *not* automatically cleared to an initial state. So if you want to be sure that an input mask will let exactly the events you specify through, you should first clear the mask with the special WIN_NO_EVENTS descriptor. Take, for example, the following two calls:

```
window_set(win, WIN_CONSUME_PICK_EVENTS,
           WIN_MOUSE_BUTTONS, LOC_DRAG, 0,
           0);

window_set(win, WIN_CONSUME_PICK_EVENTS,
           WIN_NO_EVENTS, WIN_MOUSE_BUTTONS, LOC_DRAG, 0,
           0);
```

The first will add the mouse buttons and LOC_DRAG to the existing pick input mask, while the second will set the mask to let *only* the mouse buttons and LOC_DRAG through.

Canvases by default enable LOC_WINENTER, LOC_WINEXIT, LOC_MOVE and the three mouse buttons, MS_LEFT, MS_MIDDLE and MS_RIGHT.[31] You could allow the user to type in text to a canvas by calling:

```
window_set(canvas, WIN_CONSUME_KBD_EVENT, WIN_ASCII_EVENTS, 0);
```

Sometime later you could disable type-in by calling:

```
window_set(canvas, WIN_IGNORE_KBD_EVENT, WIN_ASCII_EVENTS, 0);
```

An application needing to track mouse motion with the button down would enable LOC_DRAG by calling:

```
window_set(canvas, WIN_CONSUME_PICK_EVENT, LOC_DRAG, 0);
```

You can enable or disable the left, right or top function keys as a group via the event descriptors WIN_LEFT_KEYS WIN_RIGHT_KEYS, or WIN_TOP_KEYS. Note that if you want to see the up event you must also ask for WIN_UP_EVENTS, as in:

```
window_set(win, WIN_CONSUME_KBD_EVENTS, WIN_LEFT_KEYS,
           WIN_UP_EVENTS, 0);
```

[31] Note that the canvas package expects to receive these events, and will not function properly if you disable them.

6.6. Releasing the Event Lock

If an operation generated by an input event is going to take a long time — over, say, 5 seconds — call this routine to allow other processes to get input:[32]

```
void
window_release_event_lock(window)
    Window window;
```

6.7. Reading Events Explicitly

There are times when it is appropriate to go get the next event yourself, rather than waiting for it to come through the normal event stream. In particular, when tracking the mouse with an image which requires significant computation, it may be desirable to read events until a particular action — such as a mouse button up — is detected. To read the next input event for a window, use the function:

```
int
window_read_event(window, event)
    Window  window;
    Event  *event;
```

`window_read_event()` fills in the event structure, and returns 0 if all went well. In case of error, it sets the global variable `errno` and returns -1.

`window_read_event()` can be used in either a blocking or non-blocking mode, depending on how the window has been set up.[33]

Note that if you read events in a canvas subwindow yourself, you must translate the event's location to canvas space by calling `canvas_event()`:

```
event_in_canvas_space = canvas_event(event);
```

[32] For more details see the section on synchronization in the *Workstations* chapter of the *SunView System Programmer's Guide.*

[33] See the *Input Control* section in the *Windows* chapter of the *SunView System Programmer's Guide.*

7

Imaging Facilities: Pixwins

Imaging Facilities: Pixwins **91**

7.1. What is a Pixwin? 92

7.2. Accessing a Pixwin's Pixels 92

Obtaining the Window's Pixwin 92

Write Routines 93

Basic RasterOp Operations 93

Other Raster Operations 93

Text Routines 94

Batching and Stenciling Routines 95

Drawing Polygons 96

Drawing Curved Shapes 96

Drawing Lines 96

Read and Copy Routines 97

7.3. Rendering Speed 98

Locking 98

Batching 100

Locking and Batching Interaction 101

7.4. Clipping With Regions 102

7.5. Color 103

Introduction to Color 103

The Colormap 103

Changing the Colormap 104

Colormap Segments 104

7 Imaging Facilities: Pixwins

Material Covered

This chapter describes the *pixwin*, that is the construct you use to draw, or *render*, images in SunView. The most basic use of pixwins is to draw in a canvas subwindow.

In addition to basic pixwin usage, this chapter covers:

- How to boost your rendering speed by *locking* and *batching*
- How to use *regions* for clipping
- How to manipulate the *colormap*
- How to use the *plane groups* on the Sun-3/110

Related Documentation

This chapter is addressed primarily to writers of simple applications using canvas subwindows. For lower level details, see the chapter on *Advanced Imaging* in the *SunView System Programmers Guide.*

The pixwin drawing operations do not directly support high-level graphics operations such as shading, segments, 3-D, linestyles, etc. If your application requires these you should consider some graphics package such as SunCore or SunCGI. Both of these will run in windows — see the *SunCore* and *SunCGI Reference Manuals*.

Header Files

The definitions necessary to use pixwins are in the header file `<sunwindow/pixwin.h>`, which is included by `<sunwindow/window_hs.h>`, which in turn is included by default when you include `<suntool/sunview.h>`.

Summary Tables

Tables that summarize pixwin functions and macros are in Chapter 18, *SunView Interface Summary*:

- the *Pixwin Drawing Functions and Macros* table begins on page 311;
- the *Pixwin Color Manipulation Functions* table begins on page 315.

Write Routines

The following routines allow you to draw areas, backgrounds, vectors, text, polygons, lines, and polylines in a pixwin.

Basic RasterOp Operations

The following are the basic low-level raster operations that draw on the screen. They are common to many imaging systems.

```
pw_write(pw, dx, dy, dw, dh, op, pr, sx, sy)
```
— or —
```
pw_rop(pw, dx, dy, dw, dh, op, pr, sx, sy)
    Pixwin  *pw;
    int      dx, dy, dw, dh, op, sx, sy;
    Pixrect *pr;
```

`pw_write()` and `pw_rop()` are different names for the same procedure. They perform the indicated rasterop from the source pixrect to the destination in the pixwin. Pixels are written to the rectangle defined by `dx, dy, dw,` and `dh` in the pixwin `pw` using rasterop function `op`. `dx` and `dy` are the position of the top left-hand corner of the rectangle, and `dw` and `dh` are the width and height of the rectangle. They are copied from the rectangle with its origin at `sx, sy` in the source pixrect pointed to by `pr`.

`pw_write()` is essential for many window system operations such as scrolling a window, drawing frames and borders, drawing an icon on the screen, etc.

Other Raster Operations

The routines in this section are variations on the basic rasterop routine.

```
pw_writebackground(pw, dx, dy, dw, dh, op)
    Pixwin  *pw;
    int      dx, dy, dw, dh, op;
```

`pw_writebackground()` uses a conceptually infinite set of pixels, all of which are set to zero, as the source. It is often used to clear a canvas pixwin before drawing a new image.[35]

The following routine draws a pixel of `value` at `(x, y)` in the addressed pixwin:

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

Using this routine to draw is very slow and should be avoided. If you use it, be sure to read the later sections on *batching* and *locking*.

[35] Canvases will automatically clear damaged areas if they are set not to be retained, or if the attribute `CANVAS_AUTO_CLEAR` is set. See Chapter 5, *Canvases*, for more information.

NULL in which case the *system font* is used.

The system font is reference counted and shared between software packages. The following routines are provided to open and close the system font:[37]

```
Pixfont *
pw_pfsysopen()

pw_pfsysclose()
```

The following routine:

```
pw_ttext(pw, x, y, op, font, s)
    Pixwin  *pw;
    int      x, y, op;
    Pixfont *font;
    char    *s;
```

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

Batching and Stenciling Routines

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

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

Stencil operations are like raster ops except that the source pixrect is written through a stencil pixrect which functions as a pixel–by–pixel write enable mask. The indicated raster operation is applied only to destination pixels where the stencil pixrect `stpr` is non-zero; other destination pixels remain unchanged.

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

[37] The system font can also be obtained by calling `pf_default()`.

Read and Copy Routines

The following routines use the pixwin as a source of pixels. To get the value of the pixel at (x, y) in pixwin pw call:

```
int
pw_get(pw, x, y)
    Pixwin *pw;
    int     x, y;
```

To read pixels from a pixwin into a pixrect call:

```
pw_read(pr, dx, dy, dw, dh, op, pw, sx, sy)
    Pixwin  *pw;
    int      dx, dy, dw, dh, op, sx, sy;
    Pixrect *pr;
```

This routine reads pixels from pw starting at offset (sx, sy), using rasterop op. The pixels are stored in the rectangle with its origin at dx, dy of width dw and height dh in the pixrect pointed to by pr.

When the destination, as well as the source, is a pixwin, use:

```
pw_copy(dpw, dx, dy, dw, dh, op, spw, sx, sy)
    Pixwin *dpw, *spw;
    int     dx, dy, dw, dh, op, sx, sy;
```

dpw and spw must be the same pixwin. Also, only horizontal or vertical copies are supported.

CAUTION **These read and copy routines may find themselves thwarted by trying to read from a portion of a non-retained pixwin which is hidden, and therefore has no pixels. Therefore it is considered advanced usage to call them on a non-retained pixwin; refer to the section entitled *Handling Fixup* in the *SunView System Programmer's Guide*.**

affected rectangles must lie within the rectangles affected by the original lock.

To decrement the lock count, call:

```
pw_unlock(pw)
    Pixwin *pw;
```

When the lock count reaches 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. The following macro sets `pw`'s lock count to 0 and releases its lock:

```
pw_reset(pw)
    Pixwin *pw;
```

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 clipping information for the pixwin is validated and, if necessary, updated.
- In the case of a non-retained pixwin with only a single rectangle visible, the internals of the pixwin mechanism can be set up to bypass the pixwin software by going directly to the pixrect level on subsequent display operations.

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

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

When a display lock is held for more than two seconds of process virtual time, the lock is broken. However, the offending process is not notified by signal, because a process shouldn't be aborted for this infraction. Instead, a message is displayed on the console.

```
#define PW_OP_COUNT(n)   ((Pw_batch_type)(n))
```

So, to have batching and ensure the image on-screen is refreshed after every `n` operations, call:

```
pw_batch(pw, PW_OP_COUNT(n));
```

Clients with a group of screen updates to do can gain noticeably by doing the group as a batch. Also, the locking overhead, discussed above, will only be incurred when the screen is refreshed. An example of such a group is displaying a screen full of text, or a series of vectors with pre-computed endpoints.

In considering how to do batching, it's a good idea to be sensitive to how long the user is staring at a blank screen or an old image, and adjust the rate of screen refresh accordingly.

Locking and Batching Interaction

There are situations in which batching around locking calls makes sense. Consider that

- while batching, locking calls are a no-op;
- if a pixwin is not retained, batching calls are a no-op.

Thus, if your application has a switch to run retained or not, it makes good sense to batch around locking calls. If you batch around locking calls then your application gets the benefit of batching if running retained and the benefit of locking if running non-retained.

Locking around batches, on the other hand, is not very efficient.

7.5. Color

The dicussion which follows is divided into three sections:

- *Introduction to Color*, which introduces the concepts of the colormap and colormap segments,
- *Changing the Colormap*, which describes how to change a colormap segment, and
- *Using Color*, which describes how to make color applications compatible with monochrome and grayscale screens, and how to perform smooth animation by using double buffering.

Introduction to Color

Just as there must be arbitration between different windows to decide what is displayed on the screen when several windows overlap, there must likewise be some process of allocation when several windows want to display different sets of many colors all at once. To understand how this works you need to know how color is handled.

The pixels on a color display are not simply on or off; they take many different values for different colors. On all current Sun color displays[38] each pixel has 8 bits. Such an "8 bit deep" pixel can have any value from 0 to 255. The value in each pixel helps to determine what color appears in that dot on the screen, but it is not in a one-to-one correspondence with the color displayed; otherwise Sun color displays would only be able to display 256 different colors.

The Colormap

Instead, the value of the pixel serves as an index into the *colormap* of the display. The colormap is an array of 256 *colormap entries*. The colormap entry for each index drives the color that is actually displayed for the corresponding pixel value. A colormap entry consists of 8 bits of red intensity, 8 bits of green intensity and 8 bits of blue, packaged into the following structure:

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

Hence a Sun color display is capable of displaying over 16 million colors /(em since each colormap entry has 24 bits — but can only display 256 colors *simultaneously* — because there are only 256 colormap entries.

A Colormap Example

Suppose that in a group of pixels on the screen, some have the value 0 while others have the value 193. All pixels with the same value will be displayed in the same color. The colormap determines what that colormap will be. If entry 0 in the colormap of the screen is

```
    red = 250; green = 0; blue = 3;
```

then the pixels with a value of 0 will come out bright red. If entry 0 in the colormap is changed to

[38] See `cgone`(4S), `cgtwo`(4S) and `cgfour`(4S) in the *UNIX Interface Overview* manual.

the default colormap segment on a per-application basis by invoking the application with certain flags. The *-Wf* flag sets the foreground color, *-Wb* sets the background color, and *-Wg* specifies that the colormap of the frame will be inherited by the frame's subwindows.

The equivalent frame attributes for these flags are `FRAME_FOREGROUND_COLOR`, `FRAME_BACKGROUND_COLOR` and `FRAME_INHERIT_COLORS`.

Sharing Colormap Segments

It is possible for different processes to share a single colormap segment. For some applications, you want to guarantee that your colormap segment is not shared by another process. For example, a colormap segment to be used for animation, as described later in the section on *Double Buffering*, should not be shared. The way to ensure that a colormap segment will not be shared by another window is to give it a unique name. A common way to generate a unique name is to append the process' id to a more meaningful string that describes the usage of the colormap segment.

If a colormap segment's usage is static in nature, then it pays to use a shared colormap segment definition, since colormap entries are scarce. Windows, in the same or different processes, can share the same colormap by referring to it by the same name.

There are three basic types of shared colormap segments:

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

Example: *showcolor*

The program on the following page shows the actual colors in the display's colormap. It should help you see how the window system manages the colormap. Run this program soon after bringing up *suntools*, then run several color graphics programs such as the demos mentioned earlier. Try bringing up different windows with different foreground and background colors, as in:

```
shelltool -Wf 23 182 48 -Wb 255 200 230 -Wg
```

Manipulating the Colormap

The following sections document the routines that implement the techniques described above.

Changing a Window's Colormap Segment

To change a a window's colormap segment, you must:

1) Name the colormap segment with `pw_setcmsname()`.
2) Set the size of the segment by loading the colors with `pw_putcolormap()`.

It is important that these two steps happen in order and together.

You set and retrieve the name of a colormap segment with these two functions:

```
pw_setcmsname(pw, name)
    Pixwin *pw;
    char    name[CMS_NAMESIZE];

pw_getcmsname(pw, name)
    Pixwin *pw;
    char    name[CMS_NAMESIZE];
```

If you set the foreground and background colors (which are entries `count` - 1 and 0 in the colormap segment, respectively) to the same color, the system will change them to the foreground and background colors of *suntools*. In other words, you are prevented from making the foreground and background colors of a pixwin indistinguishable.

Setting the name resets the colormap segment to a `NULL` entry. After calling `pw_setcmsname()`, you must immediately call `pw_putcolormap()` to set the size of the colormap segment and load it with the actual colors desired. `pw_putcolormap()` and the corresponding routine to retrieve the colormap's state, `pw_getcolormap()`, are defined as follows:

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

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

`pw_putcolormap` loads the `count` elements of the pixwin's colormap segment starting at `index` (zero origin) with the first `count` values in the three arrays.

The first time `pw_putcolormap()` is called, after calling `pw_setcmsname()`, the `count` parameter defines the size of the colormap segment. The size of a colormap segment must be a power of 2, and can't be changed unless `pw_setcmsname()` is called with another name. You can call `pw_putcolormap()` subsequently to modify a subrange of the colormap — use a larger value for `index` and a smaller value for `count`.

In Appendix A, *Example Programs*, there is a program called *coloredit* which uses `pw_putcolormap()` to change the colors of its subwindows as the user adjusts sliders for red, green and blue.

Using Color

This section gives some notes on the use of color by cursors and menus, how to make color applications compatible with monochrome and grayscale screens, and how to use double buffering for smooth animation.

Cursors and Menus

Cursors appear in the foreground color, the last color in the pixwin's colormap.

Menus and prompts use *fullscreen access*, covered in Chapter 12, *Menus and Prompts* of the *SunView System Programmer's Guide*. Fullscreen access saves the colors in the first and last entries of the **screen**'s colormap, puts in the foreground and background colors, and displays the menu or prompt. This means that depending on where your application's colormap segment resides in the screen's colormap, some colors in your tool may change whenever menus or prompts are put up. You can allow for this by making the background and foreground colors in your colormap segment the same as the screen's background and foreground.

There are other menu/cursor ''glitches'' that occur when running applications in different plane groups on the Sun-3/110. These are covered in the later section on *Multiple Plane Groups*, and in the *Release 3.2 Manual*.

Is The Application Running on a Color Display?

None of the colormap manipulations described in this chapter causes an error if run on a monochrome display. All colors other than zero map to the foreground color, so if your application displays colored objects on a background of zero, they will appear as black objects on a white foreground on a monochrome display[40]. The window system detects and prevents the foreground and background being the same color on color displays.

However, you may may want to determine at run time whether your application has a color or monochrome display available to it. For example, when displaying a chart, you may want to use patterns if colors are not available. You can determine whether the display is color or monochrome by finding out how deep the pixels are. Each pixwin includes a pointer to a pixrect which represents its pixels on the screen. Pixrects, in turn, have a depth field which holds the number of bits per screen pixel. Thus

```
Pixwin *pw;
int depth = pw->pw_pixrect->pr_depth;
```

will have a value of 1 for windows displayed on monochrome devices, and a value greater than 1 for color screens. Currently, all Sun color displays have 8 bits per pixel.

Simulating Grayscale on a Color Display

There is no way to tell if your application is running on a grayscale monitor, since it runs off the same color board. The grayscale monitor is usually driven from the red output of the color board, so if two colors have different green and blue values but the same red value, they will show up the same on a color display.

[40] Unless you are running with black and white inverted, using the *-i* option to *suntools*.

Table 7-1 *Sample Colormap to Isolate Planes*

Pixel Value	Colormap A *(Only upper planes are "visible")*	Colormap B *(Only lower planes are "visible"*
00 00	blue	blue
00 01	blue	red
00 10	blue	green
00 11	blue	pink
01 00	red	blue
01 01	red	red
01 10	red	green
01 11	red	pink
10 00	green	blue
10 01	green	red
10 10	green	green
10 11	green	pink
11 00	pink	blue
11 01	pink	red
11 10	pink	green
11 11	pink	pink

From the above table, you can see that if colormap A is set (using `pw_putcolormap()`), then no matter what the value in the two lower planes, the color displayed is the same; the value in the upper two planes alone controls the color. So, if you use this colormap while only enabling the two lower planes (by passing `pw_putattributes()` the value `3`), then the values you write into the lower planes won't change what is shown.

When you switch to colormap B, the situation is reversed. Only the values in the lower planes affect what is visible. You would then pass `pw_putattributes()` the value `12` to write to the upper two planes. The two sets of colors need not be the same, so you can switch between two different-colored images.

You would use the same technique to switch between more images and/or to display more colors. You can display two different images, each with 16 different colors, or 8 different monochrome images, or values in between.

Using Double Buffering For Smooth Animation

One application of the above technique is to provide smooth animation. To move an image across the screen, you must draw it in one location, erase it, and redraw it in another. Even on a fast system, the erasing and redrawing is visible. You'd like the object to immediately appear in its new position, without disappearing momentarily. You can do this by alternating two colormaps so that the object disappears in its old location and reappears in a new one. This is called *double buffering*, because you are using the display planes as alternating buffers; as you write to one set of planes, the other set of planes is displayed.

''Glitches'' Visible when Using Plane Groups

For performance reasons, the cursor image is only written in the plane group of the window under it. So, if the cursor's *hot spot* is in a black and white window in the overlay plane and there is an adjacent color window, that part of its image that would lie over the color window is invisible, since it is drawn in the overlay plane but the enable plane is still showing the value in the color buffer. The same disappearance applies in the reverse situation.

When menus and prompts are drawn, the enable plane is set so that they are visible.

NOTE There are other glitches that occur when running applications that have not been compiled under 3.2. Consult the Release 3.2 manual for more information.

suntools and Plane Groups

It is possible to direct *suntools* (1) to only use the color buffer or the overlay plane; it is also possible to start up a second copy of *suntools* (1) in the other plane group, and switch between them using *switcher* (1) or *adjacentscreens* (1). Consult these programs' manual pages for more information.

8

Text Subwindows

Text Subwindows **117**

8.1. Creating a Text Subwindow 118

8.2. Writing to a Text Subwindow 118

8.3. Editing the Contents of a Text Subwindow 119

Removing Characters 119

Replacing Characters 120

The Editing Log 120

8.4. Positioning the Text Displayed in a Text Subwindow 121

8.5. Finding a Pattern 123

8.6. Marking Positions 123

8.7. Setting the Primary Selection 125

8.8. Manipulating the Backing Store 125

Loading a File 125

Saving Edits 126

Storing Edits 126

Discarding Edits 127

Which File is Being Edited? 127

Interactions with the File System 127

8.9. Reading from a Text Subwindow 128

8.10. Notifications from a Text Subwindow 128

8.11. Dealing with Multiple Views 131

8 Text Subwindows

This chapter describes the text subwindow package, which you can use by including the file `<suntool/textsw.h>`.

The basic function of a text subwindow is to interact with the user to display and edit a sequence of ASCII characters. These characters may be stored either in a file or in primary memory. From the programmer's point of view, a text subwindow is an opaque object upon which a set of operations can be performed.

Summary Tables

Tables that summarize text subwindow attributes, status values and functions are in Chapter 18, *SunView Interface Summary*:

- the *Text Subwindow Attributes* table begins on page 321;
- the `Textsw_action` *Attributes* table begins on page 324;
- the `Textsw_status` *Values* table begins on page 325;
- the *Text Subwindow Functions* table begins on page 326.

character of the text:

```
window_set(textsw, TEXTSW_INSERTION_POINT, 2, 0);
```

To cause the insertion point to be placed at the end of the text, set `TEXTSW_INSERTION_POINT` to the special index `TEXTSW_INFINITY`.

8.3. Editing the Contents of a Text Subwindow

Removing Characters

You can remove a contiguous span of characters from a text subwindow by calling:

```
Textsw_index
textsw_delete(textsw, first, last_plus_one)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
```

`first` specifies the first character of the span that will be deleted, while `last_plus_one` specifies the first character after the span that will not be deleted. `first` should be less than, or equal to, `last_plus_one`. To delete to the end of the text, pass the special value `TEXTSW_INFINITY` for `last_plus_one`.

The return value is the number of characters deleted, and is `last_plus_one - first`, unless all or part of the specified span is read-only. In this case, only those characters that are not read-only will be deleted, and the return value will indicate how many such characters there were. If the insertion point is in the span being deleted, it will be left at `first`.

A side-effect of calling `textsw_delete()` is that the deleted characters become the contents of the global Shelf. To remove the characters from the textsw subwindow without affecting the Shelf, call:

```
Textsw_index
textsw_erase(textsw, first, last_plus_one)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
```

Again, the return value is the number of characters removed, and `last_plus_one` can be `TEXTSW_INFINITY`.

You can emulate the behavior of an editing character, such as CTRL-H, with `textsw_edit()`:

```
Textsw_index
textsw_edit(textsw, unit, count, direction)
    Textsw    textsw;
    unsigned  unit, count, direction;
```

Depending on the value of `unit`, this routine will erase either a character, a word, or a line. Set `unit` to:

- `SELN_LEVEL_FIRST` to erase individual characters, Unfortunately, once the edit log has reached its maximum size, no more characters can be inserted into or removed from the text subwindow. In particular, since deletions, as well as insertions, are logged, space cannot be recovered by deleting characters. It is important to understand how the edit log works because because you may want to use a text subwindow with no associated file to implement a temporary scratch area or error message log. If such a text subwindow is used for a long time, the default limit of 20,000 bytes may well be reached, and it will be impossible for either the user or your code to insert any more characters even though there may be only a few characters visible in the text subwindow. Therefore, in such situations it is recommended to set `TEXTSW_MEMORY_MAXIMUM` much higher, say to 200,000.

8.4. Positioning the Text Displayed in a Text Subwindow

Usually there is more text managed by the text subwindow than can be displayed all at once. As a result, it is often necessary to determine the indices of the characters that are being displayed, and to control exactly which portion of the text is being displayed.

Screen Lines and File Lines

When there are long lines in the text it is necessary to draw a distinction between two different definitions of ''line of text''.

A *screen line* reflects what is actually displayed on the screen. A line begins with the leftmost character in the subwindow and continues across until either a newline character or the right edge of the subwindow is encountered. A *file line*, on the other hand, can only be terminated by the newline character. It is defined as the span of characters starting after a newline character (or the beginning of the file) running through the next newline character (or the end of the file).

Whenever the right edge of the subwindow is encountered before the newline, if `TEXTSW_LINE_BREAK_ACTION` is `TEXTSW_WRAP_AT_CHAR`, the next character and its successors will be displayed on the next lower screen line. In this case there would be two screen lines, but only one file line. From the perspective of the display there are two lines; from the perspective of the file only one.

Unless otherwise specified, all text subwindow attributes and procedures use the *file line* definition.

NOTE Line indices have a zero-origin, like the character indices; i.e., the first line has index 0, not 1.

Absolute Positioning

Two attributes are provided to allow you to specify which portion of the text is displayed in the text subwindow.

Setting the attribute `TEXTSW_FIRST` to a given index causes the first character of the line containing the index to become the first character displayed in the text subwindow. Thus the following call causes the text to be positioned so that the first displayed character is the first character of the line which contains index 1000:

```
window_set(textsw, TEXTSW_FIRST, 1000, 0);
```

Conversely, the following call retrieves the index of the first displayed character:

top of the subwindow.

If a particular character should always be at the top of the subwindow, then calling the following routine is more appropriate:

```
void
textsw_normalize_view(textsw, position)
    Textsw       textsw;
    Textsw_index position;
```

NOTE Both of these routines ignore any setting of the attribute `TEXTSW_UPPER_CONTEXT`, whether explicit by client code or implicit via use of the User Defaults Database.

8.5. Finding a Pattern

A common operation performed on text is finding a span of characters that match some specification. Currently, the text subwindow provides only a rudimentary pattern matching facility. You can implement a more powerful pattern matcher by reading the contents of the text subwindow and doing your own matching.

To find the nearest span of characters that match a pattern, call:

```
int
textsw_find_bytes(textsw, first, last_plus_one, buf,
                  buf_len, flags)
Textsw        textsw;
Textsw_index *first, *last_plus_one;
char         *buf;
unsigned      buf_len;
unsigned      flags;
```

The pattern to match is specified by `buf` and `buf_len`. The matcher looks for an exact and literal match — it is sensitive to case, and does not recognize any kind of meta-character in the pattern. `first` specifies the position at which to start the search. If `flags` is 0, the search proceeds forwards through the text, if 1 the search proceeds backwards. The return value is –1 if the pattern cannot be found, else it is some non-negative value, in which case the indices addressed by `first` and `last_plus_one` will have been updated to indicate the span of characters that match the pattern.

8.6. Marking Positions

Often a client wants to keep track of a particular character, or group of characters that are in the text subwindow. Given that arbitrary editing can occur in a text subwindow, and that it is very tedious to intercept and track all of the editing operations applied to a text subwindow, it is often easier to simply place one or more marks at various positions in the text subwindow. These marks are automatically updated by the text subwindow to account for user and client edits. There is no limit to the number of marks you can add.

A new mark is created by calling:

An existing mark is removed by calling:

```
void
textsw_remove_mark(textsw, mark)
    Textsw      textsw;
    Textsw_mark mark;
```

Note that marks are dynamically allocated, and it is the client's responsibility to keep track of them and to remove them when they are no longer needed.

8.7. Setting the Primary Selection

The primary selection may be set by calling:

```
void
textsw_set_selection(textsw, first, last_plus_one, type)
    Textsw       textsw;
    Textsw_index first, last_plus_one;
    unsigned     type;
```

A value of 1 for type means *primary selection*, while a value of 2 means *secondary selection*. There is currently no way to make the specified selection be pending delete. Note that there is no requirement that all or part of the selection be visible; use `textsw_possibly_normalize()` (described previously in Section 8.4) to guarantee visibility.

8.8. Manipulating the Backing Store

The file or memory being edited by a text subwindow is referred to as the *backing store*. Several attributes and functions are provided to allow you to manipulate the backing store of a text subwindow.[43]

Loading a File

You can load a file into a textsw by using `TEXTSW_FILE`, as in:

```
window_set(textsw, TEXTSW_FILE, file_name, 0);
```

CAUTION **If the existing text has been edited, these edits will be lost.** To avoid such loss, first check whether there are any outstanding edits by calling `window_get(textsw, TEXTSW_MODIFIED)`.

The above call to `window_set()` will load the new file with the text positioned so that the first character displayed has the same index as the first character that was displayed in the previous file — which is probably not what you want. To load the file with the first displayed character having its index specified by `position`, use the following:

```
window_set(textsw, TEXTSW_FILE, file_name,
                   TEXTSW_FIRST, position, 0);
```

NOTE The order of these attributes is important. Because attributes are evaluated in the order given, reversing the order would first reposition the existing file, then load the new file. This would cause an unnecessary repaint, and mis-position the old file, if it was shorter than `position`. For a full discussion of attribute ordering, see Section 4.8.

[43] Note that the edit log maintained by the text subwindow package is reset on each operation affecting the backing store. For a description of the edit log, see the discussion at the end of Section 8.3.

```
unsigned
textsw_store_file(textsw, filename, locx, locy)
    Textsw  textsw;
    char   *filename;
    int     locx, locy;
```

Again, `locx` and `locy` are used to position the upper left corner of the message box. The return value is 0 if and only if the store succeeded.

NOTE **By default, this call changes the file that the text subwindow is editing,** so that subsequent saves will save the edits to the new file. To override this policy, set the attribute `TEXTSW_STORE_CHANGES_FILE` to `FALSE`.

Discarding Edits

To discard the edits performed on the contents of a text subwindow, call:

```
void
textsw_reset(textsw, locx, locy)
    Textsw  textsw;
    int     locx, locy;
```

`locx` and `locy` are as above. Note that if the text subwindow contains a file which has not been edited, the effect of `textsw_reset` is to unload the file and replace it by primary memory provided by the text subwindow package; thus the user will see an absolutely empty text subwindow. Alternatively, if the text subwindow already was editing such primary memory then another, virgin, piece of primary memory will be provided and the edited piece will be deallocated.

Which File is Being Edited?

To find out which file the text subwindow is editing, call:

```
int
textsw_append_file_name(textsw, name)
    Textsw  textsw;
    char   *name;
```

If the text subwindow is not editing a file, this routine will return a non-zero value. Otherwise, it will return 0, and also append the name of the file to the end of `name`.

Interactions with the File System

If a text subwindow is editing a file called "foo" and the user selects *Save* from the subwindow's menu (or client code invokes `textsw_save()`), the following sequence of file operations occurs:

- foo is copied to foo%
- The contents of foo% is combined with information from the edit log file (/tmp/EtHost*Host-id*Process*Process-id*Counter*Unsigned-int*) and written over foo (thereby preserving all the permissions, etc)
- the edit log file is removed from /tmp

If "foo" is a symbolic link to "../some_dir/baz", then the backup file is created as "../some_dir/baz%".

Your notification procedure must be careful to either process all of the possible attributes that it can be called with or to pass through the attributes that it does not process to the standard notification procedure. This is important because among the attributes that can be in the *avlist* are those that cause the standard notification procedure to implement the *Expose*, *Hide*, *Open*, *Close*, and *Quit* accelerators of the user interface.

Here is an example of a client notify procedure, and a code fragment demonstrating how it would be used:

```
int (*cached_textsw_notify)();

void
client_notify_proc(textsw, attributes)
        Textsw          textsw;
        Attr_avlist     attributes;
{
        int             pass_on = 0;
        Attr_avlist     attrs;

        for (attrs = attributes; *attrs;
             attrs = attr_next(attrs)) {
            switch ((Textsw_action)(*attrs)) {
              case TEXTSW_ACTION_CAPS_LOCK:
                /* Swallow this attribute */
                ATTR_CONSUME(*attrs);
                break;
              case TEXTSW_ACTION_CHANGED_DIRECTORY:
                /* Monitor the attribute, don't swallow it */
                strcpy(current_directory, (char *)attrs[1]);
                pass_on = !0;
                break;
              default:
                pass_on = !0;
                break;
            }
        }
        if (pass_on)
            cached_textsw_notify(textsw, attributes);
}

cached_textsw_notify =
        (void (*)())window_get(textsw, TEXTSW_NOTIFY_PROC);
window_set(textsw, TEXTSW_NOTIFY_PROC, client_notify_proc);
```

The attribute TEXTSW_ACTION_EDITED_FILE is a slight misnomer, as it is given to the notify procedure *after* the first edit to *any* text, whether or not it came from a file. This notification only happens once per session of edits, where notification of TEXTSW_ACTION_LOADED_FILE is considered to terminate the old session and start a new one.

CAUTION

The attribute TEXTSW_ACTION_LOADED_FILE must be treated very carefully. This is because the notify procedure gets called with this attribute in several situations: after a file is initially loaded, after any successful *Save* menu operation, after a *Reset* menu operation, and during successful calls to textsw_reset(), textsw_save() and textsw_store(). The appropriate response by the procedure is to interpret these notifications as being equivalent to ''the text subwindow is displaying the file named by the provided string value; no edits have been performed on the file yet. In addition, any previously displayed or edited file has been either reset, saved, or stored under another name.'' In later versions of the system, the notifications will be extended to provide finer grain information about exactly what happened to the formerly displayed file. Also, note that an unnecessary save (i.e., there are no edits yet) still results in file operations and in the associated notification.

8.11. Dealing with Multiple Views

By using the *Split view* menu operation, the user can create multiple views of the text being managed by the text subwindow. Although these additional views are usually transparent to the client code controlling the text subwindow, it may occasionally be necessary for a client to deal directly with all of the views. This is accomplished by using the following routines, and the information that split views are simply extra text subwindows that happen to share the text of the original text subwindow.

```
Textsw
textsw_first(textsw)
    Textsw textsw;
```

Given an arbitrary view out of a set of multiple views, textsw_first() returns the first view (currently, this is the original text subwindow that the client created). To move through the other views of the set, call:

```
Textsw
textsw_next(textsw)
    Textsw textsw;
```

Given any view of the set, textsw_next() returns some other member of the set, or NULL if there are none left to enumerate. The following loop is guaranteed to process all of the views in the set:

```
for (textsw=textsw_first(any_split);
     *textsw;
     textsw=textsw_next(textsw)) {
        processing involving textsw;
}
```

9

Panels

Panels 135

9.1. Introduction to Panels and Panel Items 138
Message Items 138
Button Items 138
Choice Items 138
Toggle Items 139
Text Items 139
Slider Items 139
9.2. Basic Panel Routines 140
Creating and Sizing Panels 140
Creating and Positioning Panel Items 140
Explicit Item Positioning 140
Default Item Positioning 141
Laying Out Components Within an Item 141
Modifying Attributes 142
Panel-Wide Item Attributes 143
Retrieving Attributes 144
Destroying Panel Items 144
9.3. Using Scrollbars With Panels 145
Creating Scrollbars 145
Scrolling Panels Which Change Size 145
Detaching Scrollbars from Panels 146
9.4. Messages 147

9

Panels

This chapter describes the panel subwindow package, which you can use by including the file `<suntool/panel.h>`.

Section 1 provides a non-technical introduction to panels. Section 2 introduces the basic concepts and routines needed to use panels. Scrollable panels are covered in Section 3. Sections 4 through 9 describe the different types of panel items in detail, including examples. The examples are also shown on the next two pages.

For examples of complete panels, see the programs *filer*, *image_browser_1* and *image_browser_2*, which are listed in Appendix A and discussed in Chapter 4.

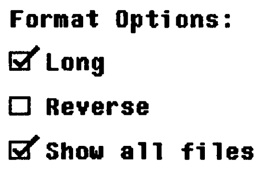
Summary Tables

Tables that summarize panel attributes, functions and macros are in Chapter 18, *SunView Interface Summary*:

- the Panel Attributes table begins on page 301;
- the *Generic Panel Item Attributes* table begins on page 302;
- the *Choice and Toggle Item Attributes* table begins on page 304;
- the *Slider Attributes* table begins on page 306;
- the *Text Item Attributes* table begins on page 307;
- the *Panel Functions and Macros* table begins on page 308.

Page Description | **Example**

157 Toggle (vertical)



Format Options:
☑ Long
☐ Reverse
☑ Show all files

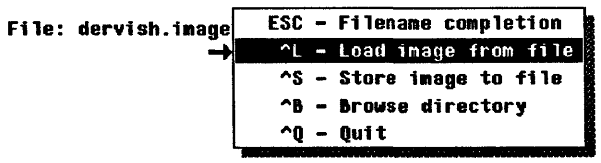
159 Text

Name: Edward G. Robinson

159 Text (masked)

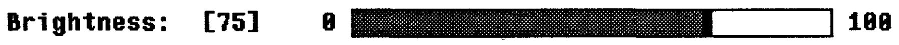
Password: ********

163 Text with menu



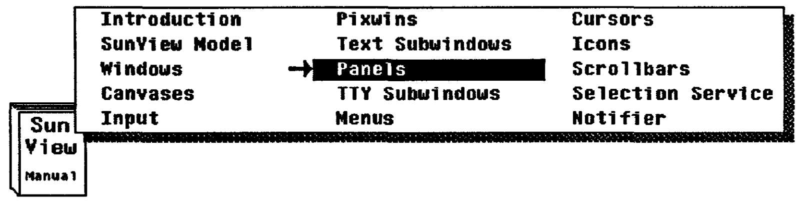
```
File: dervish.image  ESC - Filename completion
                 →    ^L - Load image from file
                      ^S - Store image to file
                      ^B - Browse directory
                      ^Q - Quit
```

165 Slider



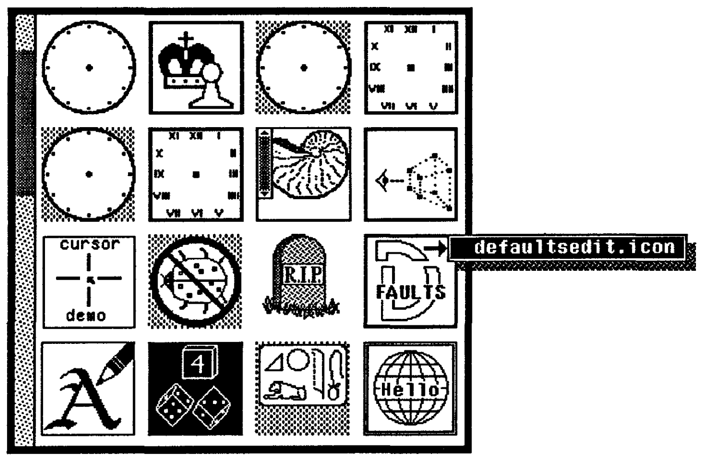
Brightness: [75] 0 ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓░░░░░ 100

149 Button with menu



```
Introduction      Pixwins           Cursors
SunView Model     Text Subwindows   Icons
Windows         → Panels            Scrollbars
Canvases          TTY Subwindows    Selection Service
Input             Menus             Notifier
```

174 Buttons with menus on scrollable panel



- a place holder for a pop-up menu, with only the label visible until the menu button is pressed.

Behind this flexibility of presentation lies a uniform structure consisting of a label, a list of choices, and, optionally, a corresponding lists of *on-marks* and *off-marks* used to indicate which choice is currently selected.

Toggle Items

In appearance and structure, toggle items are identical to choice items. The difference lies in the behavior of the two types of items when selected. In a choice item exactly one element of the list is selected, or *current*, at a time. A toggle item, on the other hand, is best understood as a list of elements which behave as toggles: each choice may be either on or off, independently of the other choices. Selecting a choice causes it to change state. There is no concept of a single current choice; at any given time all, some, or none of the choices may be selected.

Text Items

Text items are basically type-in fields with optional labels and menus. You can specify that your notify procedure be called on each character typed in, only on specified characters, or not at all. This allows an application such as a forms-entry program to process input on a per character, per field, or per screen basis.

Slider Items

Slider items allow the graphical representation and selection of a value within a range. They are appropriate for situations where it is desired to make fine adjustments over a continuous range of values. A familiar model would be a horizontal volume control lever on a stereo panel.

interpret their arguments as rows or columns, respectively, and convert the value to the corresponding number of pixels, based on the panel's font, as specified by WIN_FONT. Compare the two calls below:

```
panel_create_item(panel, PANEL_MESSAGE,
                  PANEL_LABEL_STRING, "Hi!",
                  PANEL_ITEM_X,       10,
                  PANEL_ITEM_Y,       20,
                  0);

panel_create_item(panel, PANEL_MESSAGE,
                  PANEL_LABEL_STRING, "Hi!",
                  PANEL_ITEM_X,       ATTR_COL(10),
                  PANEL_ITEM_Y,       ATTR_ROW(20),
                  0);
```

The first will place the item at pixel location (10,20), while the second will place the item at row 20, column 10.

NOTE

The value computed for ATTR_ROW() includes the top margin, given by WIN_TOP_MARGIN, and the value computed for ATTR_COL() includes the left margin, given by WIN_LEFT_MARGIN. The alternate macros ATTR_ROWS() and ATTR_COLS() are also provided, which compute values that do not include the margins.

Default Item Positioning

If you create an item without specifying its position, it is placed just to the right of the item on the "lowest row" of the panel, where lowest row is defined as the maximum y-coordinate (PANEL_ITEM_Y) of all the items. So in the absence of specific instructions, items will be placed within the panel in *reading order* as they are created: beginning four pixels in from the left and four pixels down from the top, items are located from left to right, top to bottom. If an item will not fit on a row, and more of the item would be visible on the next row, it will be placed on the next row. The number of pixels left blank between items on a row may be specified by PANEL_ITEM_X_GAP, which has a default value of 10. The number of pixels left blank between rows of items may be specified by PANEL_ITEM_Y_GAP, which has a default value of 5.

Laying Out Components Within an Item

You may also specify the layout of the various components within an item, by means of the attributes PANEL_LABEL_X, PANEL_LABEL_Y, PANEL_VALUE_X, PANEL_VALUE_Y, etc. If the components are not explicitly positioned, then the value is placed either eight pixels to the right of the label, if PANEL_LAYOUT is PANEL_HORIZONTAL (the default), or four pixels below the label, if PANEL_LAYOUT is PANEL_VERTICAL.

Panel-Wide Item Attributes

Some attributes which apply to items may be set for all items in the panel by setting them when the panel is created. Such attributes include whether items have menus, whether item labels appear in bold, whether items are laid out vertically or horizontally, and whether items are automatically repainted when their attributes are modified.[47] For example, the call:

```
panel = window_create(frame, PANEL
                        PANEL_SHOW_MENU,  FALSE,
                        PANEL_LABEL_BOLD, TRUE,
                        PANEL_LAYOUT,     PANEL_VERTICAL,
                        PANEL_PAINT,      PANEL_NONE,
                        0);
```

overrides the defaults for all the attributes mentioned: any items subsequently created in that panel will not have menus, will have their labels printed in bold and their components laid out vertically, and will not be repainted automatically when their attributes are modified.

CAUTION **The panel-wide item attributes mentioned above are only used to supply default values for items which are subsequently created.** This means, for example, that you cannot change all the item labels to bold by first creating the items and then setting `PANEL_LABEL_BOLD` to `TRUE` for the panel.

[47] For a complete list of panel-wide item attributes, see the *Panel Attributes* table in Chapter 18.

9.3. Using Scrollbars With Panels

A *scrollable* panel is a large panel which can be viewed through a smaller subwindow by means of scrollbars.

Creating Scrollbars

Scrollbars come in two orientations: vertical and horizontal. The call below creates a panel with both vertical and horizontal scrollbars (as would be desirable in a long, many-columned table, for example):

```
panel = window_create(frame, PANEL,
           WIN_VERTICAL_SCROLLBAR,    scrollbar_create(0),
           WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(0),
           0);
```

The values of the attributes `WIN_VERTICAL_SCROLLBAR` and `WIN_HORIZONTAL_SCROLLBAR` are the scrollbars which are returned by the `scrollbar_create()` calls.[48]

Commonly the scrollbar will remain attached to the panel for the duration of the panel's existence, and there will be no need to modify the scrollbar's attributes. In this simple case, there is no need to save the handle returned by `scrollbar_create()`. If you desire to destroy the scrollbar, modify its attributes, or detach it from one panel and attach it to another, you must either save the handle or retrieve it from the panel.[49] For example, to destroy a panel's vertical scrollbar:

```
scrollbar_destroy(panel_get(panel, WIN_VERTICAL_SCROLLBAR));
panel_set(panel, WIN_VERTICAL_SCROLLBAR, 0, 0);
```

Scrolling Panels Which Change Size

Often panels are used to display information for browsing. *iconedit*, for example, uses a popup panel to allow the user to browse through the images in a directory. The easiest way to do this is to create the panel items anew each time different information is displayed. For example, the *iconedit* function which fills the browsing panel first destroys any existing panel items, then creates an item for each image found.

If you are going to change the size of the panel in this way, you must inform the scrollbar of the new size by calling the function:

```
panel_update_scrolling_size(panel)
    Panel panel;
```

The correct time to call `panel_update_scrolling_size()` is after you have created all the items and given them labels. If you don't update the scrollbar's idea of the panel's size, the size of the scrollbar's bubble will be wrong.

[48] The call `scrollbar_create(0)` produces a default scrollbar. It is usually best to create a default scrollbar and let the user specify how it looks via *defaultsedit*. You can, of course, override the user's default settings by explicitly setting the scrollbar's attributes. For a complete list of scrollbar attributes see Chapter 18, *SunView Interface Summary*.

[49] In order to save the scrollbar's handle or reference any scrollbar attributes you must include the file `<suntool/scrollbar.h>`.

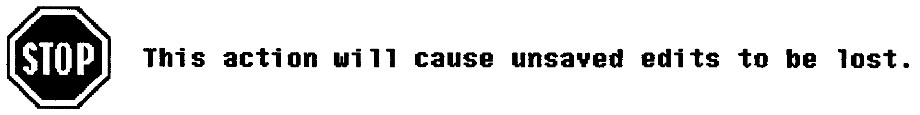
9.4. Messages

Messages are the simplest of the item types. Their only visible component is their label. They have no value or menu.

Message items, like buttons, are selectable and can have notify procedures. The selection behavior of messages differs from that of buttons in that no feedback is given to the user when a message is selected.

Example

In the following example, two message items are used together to give a warning message:

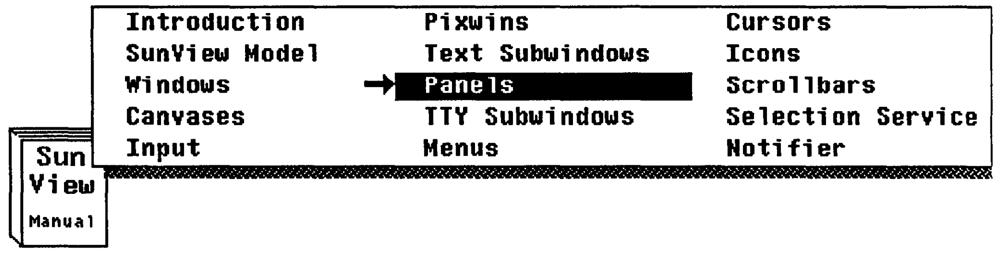


This action will cause unsaved edits to be lost.

```
static short    stop_array[] = {
#include "stopsign.image"
};
mpr_static(stopsign, 64, 64, 1, stop_array);

panel_create_item(panel, PANEL_MESSAGE,
                    PANEL_LABEL_IMAGE, &stopsign,
                    0);

panel_create_item(panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING,
    "This action will cause unsaved edits to be lost.",
    0);
```

You may change the label for a message item (as for any type of item) via `PANEL_LABEL_STRING` or `PANEL_LABEL_IMAGE`.

It is often useful to associate a menu with a button. The example below shows a button representing an online manual. The menu over the button allows the user to bring up the text for the different chapters:

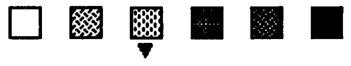


To do this, you must write your own event procedure, as described in Section 9.13, *Event Handling*. On receiving a right mouse button down event, display the menu and take the appropriate action depending on which menu item the user selects. For all other events, call the panel's default event procedure.

Here is the code to create the menu and the button, and the event procedure to display the menu:

```
static short    book_array[] = {
#include "book.image"
};
mpr_static(book, 64, 64, 1, book_array);

Menu menu = menu_create( MENU_NCOLS, 3, MENU_STRINGS,
     "Introduction",  "Pixwins",         "Cursors",
     "SunView Model", "Text Subwindows", "Icons",
     "Windows",       "Panels",          "Scrollbars",
     "Canvases",      "TTY Subwindows",  "Selection Service",
     "Input",         "Menus",           "Notifier", 0,
     0);

panel_create_item(panel, PANEL_BUTTON,
                  PANEL_LABEL_IMAGE, &book,
                  PANEL_EVENT_PROC,  handle_panel_event,
                  0);

handle_panel_event(item, event)
    Panel_item item;
    Event *event;
{
    if (event_id(event) == MS_RIGHT && event_is_down(event)) {
        int chapter = menu_show(book_menu, panel, event, 0);
        switch (chapter) {
            case 1:  /* Introduction */ break;
            case 2:  /* Pixwins      */ break;
                     ...
            case 15: /* Notifier     */ break;
        }
    } else
        panel_default_handle_event(item, event);
}
```

The item below, taken from *iconedit*, shows how parallel lists can be abbreviated:



```
panel_create_item(iced_panel, PANEL_CHOICE,
    PANEL_MARK_IMAGES,          &down_triangle,0,
    PANEL_NOMARK_IMAGES,        0,
    PANEL_CHOICE_IMAGES,        &square_white, &square_25,
                                &square_root,  &square_50,
                                &square_75,    &square_black, 0,
    PANEL_VALUE,                2,
    PANEL_CHOICE_XS,            30, 60, 90, 120, 150, 180, 0,
    PANEL_MARK_XS,              34, 64, 94, 124, 154, 184, 0,
    PANEL_CHOICE_YS,            345, 0,
    PANEL_MARK_YS,              363, 0,
    PANEL_NOTIFY_PROC,          proof_background_proc,
    0);
```

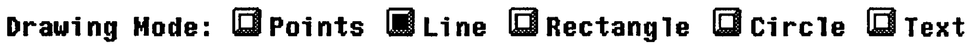
The item has six choices, representing the six available background patterns for the proof area. Note, however, that three of the lists, — `PANEL_MARK_IMAGES`, `PANEL_CHOICE_YS` and `PANEL_MARK_YS` all have only one element. When any of the parallel lists are abbreviated in this way, the last element given will be used for the remainder of the choices. So, the "345, 0" in the example above serves as shorthand for "345, 345, 345, 345, 345, 345, 0". All the choice images will appear at *y* coordinate 345, all the mark images will appear at *y* coordinate 363, and all the choices will have `down_triangle` as their mark image.

NOTE You can't specify that a choice or mark-image appear at $x = 0$ or $y = 0$ by using the attributes `PANEL_CHOICE_XS`, `PANEL_CHOICE_YS`, `PANEL_MARK_XS` or `PANEL_MARK_YS`. Since these attributes take null-terminated lists as values, the zero would be interpreted as the terminator for the list. You may achieve the desired effect by setting the positions individually, with the attributes `PANEL_CHOICE_X`, `PANEL_CHOICE_Y`, `PANEL_MARK_X`, or `PANEL_MARK_Y`, which take as values the number of the choice or mark, followed by the desired position.

Examples

As a basis for our examples we'll take the item in iconedit which allows the user to select the drawing mode. The item could have been presented in several different forms.

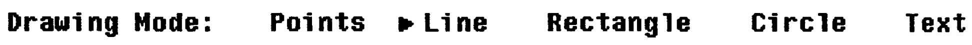
The simplest call would specify the label and choices as strings, and take the defaults for all other attributes. All the choices will be displayed, and the feedback will be marked, with push-buttons for the mark images:



Drawing Mode: ☐ Points ■ Line ☐ Rectangle ☐ Circle ☐ Text

```
panel_create_item(panel, PANEL_CHOICE,
    PANEL_LABEL_STRING,   "Drawing Mode:",
    PANEL_CHOICE_STRINGS, "Points", "Line", "Rectangle",
                          "Circle", "Text", 0,
    0);
```

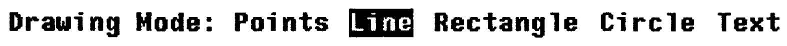
You can specify a custom mark, such as this small pointer, to indicate the current choice:



Drawing Mode: Points ▶ Line Rectangle Circle Text

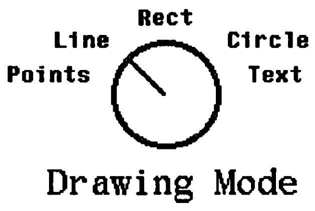
```
static short pointer_array[] = {
#include "pointer.pr"
};
mpr_static(pointer, 16, 16, 1, pointer_array);

panel_create_item(panel, PANEL_CHOICE,
    PANEL_LABEL_STRING,   "Drawing Mode:",
    PANEL_MARK_IMAGES,    &pointer, 0,
    PANEL_NOMARK_IMAGES,  0,
    PANEL_CHOICE_STRINGS, "Points", "Line", "Rectangle",
                          "Circle", "Text", 0,
    0);
```

Setting PANEL_FEEDBACK to PANEL_INVERTED produces:



Drawing Mode: Points Line Rectangle Circle Text

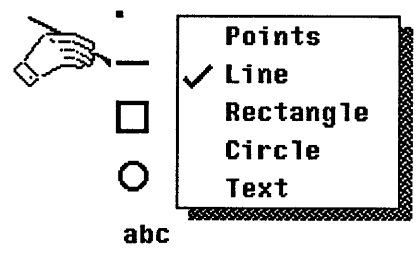
With some effort, you can use a choice item to model a dial:



The way to make a such a dial is to make an image for each dial setting, and use these images as the on-marks. Place the on-marks and the choices explicitly — the on-marks in the center, forming the dial, and the choices around the dial's perimeter:

```
panel_create_item(panel, PANEL_CHOICE,
    PANEL_CHOICE_STRINGS, "Points", "Line", "Rect",
                          "Circle", "Text", 0,
    PANEL_MARK_IMAGES,    &dial_1, &dial_2, &dial_3,
                          &dial_4, &dial_5, 0,
    PANEL_NOMARK_IMAGES,  0,
    PANEL_CHOICE_XS,      7, 34, 82, 133, 145, 0,
    PANEL_CHOICE_YS,      53, 33, 20, 33, 53, 0,
    PANEL_MARK_XS,        66, 0,
    PANEL_MARK_YS,        40, 0,
    PANEL_LABEL_STRING,   "Drawing Mode",
    PANEL_LABEL_X,        30,
    PANEL_LABEL_Y,        65,
    PANEL_LABEL_FONT,
      pf_open("/usr/lib/fonts/fixedwidthfonts/gallant.r.19"),
    0);
```

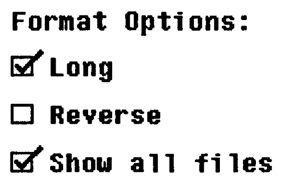
The form which is actually used in *iconedit* employs vertical layout, images for the choices, and strings for the menu:



```
panel_create_item(panel, PANEL_CHOICE,
    PANEL_LAYOUT,              PANEL_VERTICAL,
    PANEL_CHOICE_IMAGES,       &points, &line, &rectangle,
                               &circle, &text, 0,
    PANEL_MENU_CHOICE_STRINGS, "Points", "Line", "Rectangle",
                               "Circle", "Text", 0,
    PANEL_MARK_IMAGES,         &drawing_hand, 0,
    PANEL_NOMARK_IMAGES,       0,
    0);
```

Example

Here's an item which lets you set the *-l*, *-r*, or *-a* flags for the *ls* command:



Format Options:

☑ **Long**

☐ **Reverse**

☑ **Show all files**

```
format_item = panel_create_item(panel, PANEL_TOGGLE,
    PANEL_LABEL_STRING,      "Format Options:",
    PANEL_LAYOUT,            PANEL_VERTICAL,
    PANEL_CHOICE_STRINGS,    "Long",
                             "Reverse",
                             "Show all files",
                             0,
    PANEL_TOGGLE_VALUE,      0, TRUE,
    PANEL_TOGGLE_VALUE,      2, TRUE,
    PANEL_NOTIFY_PROC,       format_notify_proc,
    0);
```

You can get or set the value of a particular choice — including choices beyond the first 32 — with `PANEL_TOGGLE_VALUE`. When used to set the value, this attribute takes two values: the index of the choice to set, and the desired value. In the above example, `PANEL_TOGGLE_VALUE` is used to initialize the first and third choices to `TRUE`. To find out the value of the third choice, you would call:

```
value = (int) panel_get(format_item, PANEL_TOGGLE_VALUE, 2);
```

9.8. Text

Displaying Text Items

The value component of a text item is the string which the user enters and edits. It is drawn on the screen just after the label, as in:

```
Name: Edward G. Robinson
```

```
panel_create_item(panel, PANEL_TEXT,
                  PANEL_LABEL_STRING, "Name:",
                  PANEL_VALUE,        "Edward G. Robinson",
                  0);
```

If PANEL_LAYOUT is set to PANEL_VERTICAL, overriding the default of PANEL_HORIZONTAL, the value will be placed below the label.

The number of characters of the text item's value which are displayable on the screen is set via PANEL_VALUE_DISPLAY_LENGTH, which defaults to 80 characters. When characters are entered beyond this length, the value string is scrolled one character to the left, so that the most recently entered character is always visible. As the string scrolls to the left, the leftmost characters move out of the visible display area. The presence of these temporarily hidden characters is indicated by a small left-pointing triangle. So setting the display length to 12 in the above call would produce:

```
Name: ◂G. Robinson
```

As excess characters are deleted, the string is scrolled back to the right, until the actual length becomes equal to the displayed length, and the entire string is visible.

It is sometimes desirable to have a protected field where the user can enter confidential information. The attribute PANEL_MASK_CHAR is provided for this purpose. When the user enters a character, the character you have specified as the value of PANEL_MASK_CHAR will be displayed in place of the character the user has typed. So setting PANEL_MASK_CHAR to "'*'" would produce:

```
Password: ********
```

If you want to disable character echo entirely, so that the caret does not advance and it is impossible to tell how many characters have been entered, use the space character as the mask. You can remove the mask and display the actual value string at any time by setting the mask to the null character.

The maximum number of characters which can be typed into a text item (independently of how many are displayable) is set via the attribute PANEL_VALUE_STORED_LENGTH. Attempting to enter a character beyond this limit causes the field to overflow, and the character is lost. The value string is blinked to indicate to the user that the text item is not accepting any more characters.

The stored length, like the displayed length, defaults to 80 characters.

What happens when the user types a character? The panel package treats some characters specially. CTRL g and CTRL d are mapped to the SunView functions Get and Delete respectively. When the user types these characters, the panel package notices them and performs the appropriate operation, without passing them on to your notify procedure.

The user's editing characters — *erase*, *erase-word* and *kill* — are also treated specially. If you have asked for the character by including it in `PANEL_NOTIFY_STRING`, the panel package will call your notify procedure. After the notify procedure returns, the appropriate editing operation will be applied to the value string. (Note: the editing characters are never appended to the value string, regardless of the return value of the notify procedure.)

Characters other than the special characters described above are treated as follows. If your notify procedure is *not* called, then the character, if it is printable, is appended to the value string. If it is not printable, it is ignored. If your notify procedure *is* called, what happens to the value string, and whether the caret moves to another text item, is determined by the notify procedure's return value. The following table shows the possible return values:

Table 9-2 *Return Values for Text Item Notify Procedures*

Value Returned	Action Caused
`PANEL_INSERT`	Character is appended to item's value
`PANEL_NEXT`	Caret moves to next text item
`PANEL_PREVIOUS`	Caret moves to previous text item
`PANEL_NONE`	Ignore the input character

If a non-printable character is inserted, it is appended to the value string, but nothing is shown on the screen.

If you don't specify your own notify procedure, the default procedure `panel_text_notify()` will be called at the appropriate time, as determined by the setting of `PANEL_NOTIFY_LEVEL`. This procedure causes the caret to move to the next text item on RETURN or TAB , the caret to move to the previous text item on SHIFT RETURN or SHIFT TAB , printable characters to be inserted, and all other characters to be discarded.

Writing Your Own Notify Procedure

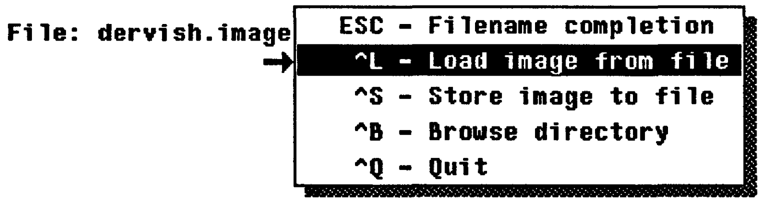
By writing your own notify procedure, you can tailor the notification behavior of a given text item to support a variety of interface styles. On one extreme, you may want to process each character as the user types it in. For a different application you may not care about the characters as they are typed in, and only want to look at the value string in response to some other button. A typical example is getting the value of a filename field when the user presses the Load button.

Text Menus

A menu may be associated with a text item by setting `PANEL_SHOW_MENU` to `TRUE`.

Example

One use of text item menus is to make any item-specific "accelerators", or characters which cause special behavior, visible to the user. This usage of accelerators may be seen in the following example taken from *iconedit*. The item labelled "**File:**" holds the name of the file being edited. In addition to typing printable characters, which are appended to the value of the item, the user can type ESC for filename completion, CTRL l to load an image from the file, CTRL s to store an image to the file, or CTRL b to browse the images in a directory.



```
#define ESC 27
#define CTRL_L 12
#define CTRL_S 19
#define CTRL_Q 17
#define CTRL_B 2

filename_item = panel_create_item(panel, PANEL_TEXT,
  PANEL_LABEL_STRING,          "File:",
  PANEL_NOTIFY_LEVEL,          PANEL_ALL,
  PANEL_NOTIFY_PROC,           filename_proc,
  PANEL_VALUE_DISPLAY_LENGTH,  18,
  PANEL_SHOW_MENU,             TRUE,
  PANEL_MENU_CHOICE_STRINGS,   "ESC    - Filename completion",
                               "CTRL L - Load image from file",
                               "CTRL S - Store image to file",
                               "CTRL B - Browse Directory",
                               "CTRL Q - Quit",
                               0,
  PANEL_MENU_CHOICE_VALUES,    ESC,CTRL_L,CTRL_S,CTRL_B,CTRL_Q, 0,
  0);
```

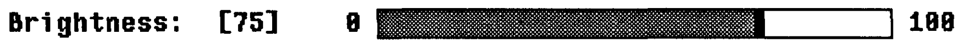
The last two attributes specify the menu. `PANEL_MENU_CHOICE_STRINGS` is a null-terminated array of strings to appear as the selectable lines of the menu. The value that the menu returns for each of its lines is specified via `PANEL_MENU_CHOICE_VALUES`. So if the menu line "**^L — Load image from file**" is selected, the menu will return the value `CTRL_L`. The value returned by the menu is passed directly to the text item, just as if it had been typed at the keyboard.

Slider Value

The value of a slider is an integer in the range `PANEL_MIN_VALUE` to `PANEL_MAX_VALUE`. You can retrieve or set a slider's value with the attribute `PANEL_VALUE`.

Example

Here's a typical slider, which might be used to control the brightness of a screen:



```
panel_create_item(panel, PANEL_SLIDER,
                  PANEL_LABEL_STRING, "Brightness: ",
                  PANEL_VALUE,        75,
                  PANEL_MIN_VALUE,    0,
                  PANEL_MAX_VALUE,    100,
                  PANEL_SLIDER_WIDTH, 300,
                  PANEL_NOTIFY_PROC,  brightness_proc,
                  0);
```

Example 2:

```
item2 = panel_create_item(panel, PANEL_TEXT,
            PANEL_LABEL_STRING,          "Enter Name:",
            PANEL_VALUE_DISPLAY_LENGTH, 10,
            0);
```

(begin processing events, etc...)

```
panel_set(item2,
          PANEL_ITEM_X, 10,
          PANEL_ITEM_Y, 50,
          PANEL_PAINT,  PANEL_NONE,
          0);
panel_set(item2,
          PANEL_LABEL_IMAGE, &pixrect1,
          PANEL_PAINT,       PANEL_NONE,
          0);
panel_set(item2,
          PANEL_VALUE_DISPLAY_LENGTH, 30,
          0);
```

The above two examples each produce the same effect. In the first example, the item's repaint behavior is set to `PANEL_NONE` at creation time, so it is not repainted automatically after the `panel_set()` calls, and no repainting occurs until the call to `panel_paint()`. In the second example, the item's repaint behavior is the default, `PANEL_CLEAR`. This is overridden in the first two `panel_set()` calls, so no repainting occurs. However, it is not overridden in the third call to `panel_set()`, so repainting occurs before that call returns.

As mentioned above, the repaint behavior for all items in a panel can be set when the panel is created, e.g.:

```
window_create(frame, PANEL, PANEL_PAINT, PANEL_NONE, 0);
```

All items created in the above panel will have a repaint behavior of `PANEL_NONE`.

9.13. Event Handling

This section describes how the panel package handles events.[55] If you require a behavior not provided by default, you can write your own event handling procedure for either an individual item or the panel as a whole.

Default Event Handling

Using the default event handling mechanism, events are handled for all the panel items in a uniform way. A single routine reads the events, updates an internal state machine, and maps the event to an *action* to be taken by the item. Actions fall into two categories: *previewing* and *accepting*. The previewing action gives the user visual feedback indicating what will happen when he releases the mouse button. The accepting action causes the item's value to be changed and/or its notify procedure to be called, with the event passed as the last argument.

The default event-to-action mapping is given in the following table:

Event	Action
Left button down or drag in w/left button down	Begin previewing
Drag with left button down	Update previewing
Drag out of item rectangle with left button down	Cancel preview
Left button up	Accept
Right button down	Display menu & accept user's selection
Keystroke	Accept keystroke if text item

What actually happens when an item is told to perform one of the above actions depends on the type of the item. For example, when asked to begin previewing, a button item inverts its label, a message item does nothing, a slider item redraws the shaded area of its slider bar, etc.[56]

Writing Your Own Event Handler

You may want to handle events in a way which is not supported by this default scheme. For example, there is no way to take any action on middle mouse button events. To do so you must extend the event handling functionality by replacing the default event-to-action mapping function for a panel or panel item. Three attributes have been defined for this purpose:

Table 9-3 *Panel Event Handling Attributes*

Attribute	Argument Type	Default Value
`PANEL_EVENT_PROC`	int (*)()	`panel_default_handle_event()`
`PANEL_BACKGROUND_PROC`	int (*)()	`panel_default_handle_event()`
`PANEL_ACCEPT_KEYSTROKE`	boolean	`FALSE`

[55] The general SunView input paradigm, including details on the various events, is covered in Chapter 6, `Handling Input`.

[56] For particulars, see the *Selection* subsection under each item type.

Table 9-4 *Panel Action Functions*

Definition	Description

`panel_accept_key(object, event)`
<Panel or Panel_item> `object;`
`Event *event;`
	Tells a text item to accept a keyboard event. Currently ignored by non-text panel items.

`panel_accept_menu(object, event)`
<Panel or Panel_item> `object;`
`Event *event;`
	Tells an item to display its menu and process the user's selection.

`panel_accept_preview(object, event)`
<Panel or Panel_item> `object;`
`Event *event;`
	Tells an item to do what it is supposed to do when selected, including completing any previewing feedback.

`panel_begin_preview(object, event)`
<Panel or Panel_item> `object;`
`Event *event;`
	Tells an item to begin any feedback which indicates tentative selection.

`panel_cancel_preview(object, event)`
<Panel or Panel_item> `object;`
`Event *event;`
	Tells an item to cancel any previewing feedback.

`panel_update_preview(object, event)`
<Panel or Panel_item> `object;`
`Event *event;`
	Tells an item to update its previewing feedback (e.g. redraw the slider bar for a slider item).

In most of the action routines, only the event's location and shift state are of interest. When previewing, choices, toggles and sliders use the event's location to determine the current value. Choices use the shift state to determine whether to advance or backup the current choice. `panel_accept_key()` is the only action function to make use of the actual event code.

Example

Suppose you are implementing *dbxtool* and want to have the buttons in the command panel execute different commands depending on whether they were selected with the left or middle mouse button. For example, the button labeled (next) might behave as the (step) button if activated with the middle button. When the middle button is depressed, you want to preview an alternate label, and when it is released, you want to execute the dbx command corresponding to the previewed label.

You can get get this functionality by replacing the event procedure for each of the button items in the command panel. This could be done either by specifying a default event procedure for all the items when the panel is created:

```
panel = window_create(frame,PANEL,PANEL_EVENT_PROC,dbx_event_proc,0);
```

or by specifying a the event procedure as each panel item is created:

```
panel_create_item(panel,PANEL_BUTTON,PANEL_EVENT_PROC,dbx_event_proc,
```

Whenever one of the buttons gets an event, `dbx_event_proc()` will be called and can then map the events to actions as it sees fit. The code for the new event procedure is given on the next page. Note the use of `PANEL_CLIENT_DATA` to store the images for the two labels for each item.

```
        /* cancel for some reason */
        case PANEL_EVENT_CANCEL:
          if (panel_get(item, PANEL_LABEL_IMAGE) ==
              dbx_data->middle_pr) {
              /* we were previewing -- cancel it.
              */
              panel_cancel_preview(item, event);
              panel_set(item, PANEL_LABEL_IMAGE, dbx_data->left_pr, 0)
          } else
              /* we weren't previewing, so
              * let the default event proc handle it.
              */
              panel_default_handle_event(item, event);
          break;

        /* some other event */
        default:
          /* we don't care about this event -- let the default
          * event proc handle it.
          */
          panel_default_handle_event(item, event);
    }
}
```

The final step is to modify the notify procedure for each button to perform different actions depending on which mouse button was released. The notify procedure for the step/next button, for example, would look like:

```
next_step_notify_proc(item, event)
    Panel_item item;
    Event      *event;
{
    if (event_id(event) == MS_MIDDLE)
        /* do middle button command, "step" */
    else
        /* do left button command, "next" */
}
```

The browser is implemented as a panel containing buttons having the images as their labels. The buttons are created each time the user wants to browse a different set of images. When each button is created, the name of the file containing the image is stored as the value of the button's `PANEL_CLIENT_DATA`.

Listed below is the event procedure shared by each button. There is a global menu containing a single menu item, `image_menu_item`. If the event is a right mouse button, the display string for this menu item is set to the file name which was previously stored as the button's `PANEL_CLIENT_DATA`. Then the event is adjusted from panel space to window space, and the menu is displayed at the proper coordinates. If the user selects from the menu, the button's notify procedure, `browser_items_notify_proc()`, is called, so the effect is the same whether the item is selected through the menu or directly.

```
browser_items_event_proc(item, event)
    Panel_item item;
    Event *event;
{
    if (event_id(event) == MS_RIGHT) {

        Event *adjusted_event;

        menu_set(image_menu_item,
            MENU_STRING, panel_get(item, PANEL_CLIENT_DATA), 0);

        adjusted_event = panel_window_event(browser, event);

        if (menu_show(image_menu, browser, adjusted_event, 0)) {
            browser_items_notify_proc(item);
            return;
        }
    }
    panel_default_handle_event(item, event);
}
```

Note that for all events other than the right mouse button, the panel's default event procedure is called.

10

TTY Subwindows

TTY Subwindows **179**

10.1. Creating a TTY Subwindow 180

10.2. Driving a TTY Subwindow 180

10.3. TTY Subwindow Escape Sequences 181

Standard ANSI Escape Sequences 181

Special Escape Sequences 181

10

TTY Subwindows

The tty (or *terminal emulator*) subwindow emulates a standard Sun terminal, the principal difference being that the row and column dimensions of a tty subwindow can vary. You can run arbitrary programs in a tty subwindow; perhaps the main use is to run a shell within a window.

To see tty subwindows in use, run the standard tools *shelltool* (1) and *gfxtool* (1).

Programs using tty subwindows must include the file `<suntool/tty.h>`.

Summary Tables

Tables that summarize tty subwindow attributes, functions and macros are in Chapter 18, *SunView Interface Summary*:

- the *TTY Subwindow Attributes* table begins on page 329;
- the *TTY Subwindow Functions* table begins on page 329;
- the *TTY Subwindow Special Escape Sequences* table begins on page 330.

```
int ttysw_output(tty, buf, len)
    Tty tty;
    char *buf;
    int len;
```

`ttysw_output()` runs the character sequence in `buf` that is `len` characters long through the terminal emulator of `tty`. It returns the number of characters accepted. The effect is similar to executing

echo *character_sequence* **> /dev/tty***N*

where `tty`*N* is the pseudo-tty associated with the tty subwindow. One use of `ttysw_output()` is to send the escape sequences listed in the next section to the tty subwindow.

Example: *typein*

Appendix A, *Example Programs* gives the listing for *typein*, a program which uses a tty subwindow to get user input and draws on a canvas based on that input.

10.3. TTY Subwindow Escape Sequences

Standard ANSI Escape Sequences

The tty subwindow accepts the same ANSI escape sequences as the Sun terminal,[59] with the following two exceptions:

- The effect of `CTRL-G` (`0x07`)in a tty subwindow depends on how the user has set the two SunView options *Audible_Bell* and *Visible_Bell* in `defaultsedit`. If *Audible_Bell* is *Enabled*, the bell will ring. If *Visible_Bell* is *Enabled*, the window will flash.
- The sequence `ESC[0r`, which enables vertical wrap mode in the Sun terminal, has no effect in the tty subwindow.

Special Escape Sequences

Escape sequences have been defined by which the user can get and set attributes of both the tty subwindow and the frame which contains it. For example, the user can type an escape sequence to open, close, move or resize the frame, change the label of the frame or the frame's icon, etc. These escape sequences are described in the table on the following page.

Example: *tty_io*

For an example of setting the frame's label via a tty subwindow escape sequence, see the program *tty_io*, listed in Appendix A, *Example Programs*.

[59] see *cons* (4s) in the *System Interface Manual*.

11

Menus

Menus ... **185**

11.1. Basic Menu Usage ... 186

11.2. Components of Menus & Menu Items ... 190

Menus ... 190

Visual Components ... 190

Generate Procedures ... 190

Notify Procedures ... 190

Client Data ... 190

Menu Items ... 190

Menu Items ... 191

Representation on the Screen ... 191

Item Values ... 191

Item Generate Procedures ... 191

Item Action Procedures ... 191

Client Data ... 191

Item Margins ... 192

11.3. Examples ... 193

11.4. Item Creation Attributes ... 200

11.5. Destroying Menus ... 201

11.6. Searching for a Menu Item ... 202

11.7. Callback Procedures ... 203

Generate Procedures ... 203

Menu Item Generate Procedure ... 204

11

Menus

The SunView menu package allows you to chain individual menus together into a collection known as a *walking menu.* A menu contains *menu items,* some of which may have a small arrow pointing to the right. This indicates to the user that if he slides the mouse to the right of that item, a *pullright* menu will appear. Menus can be strung together in this fashion, so that the user ''walks'' to the right down the chain of menus in order to make a selection.

The definitions necessary to use walking menus are found in the file `<suntool/walkmenu.h>`, which is included by default when you include the file `<suntool/sunview.h>`.

The most useful sections to read first are the first three. Section 1 introduces the basic routines and gives some simple examples. Section 2 outlines the components of menus and menu items and introduces common terms. Section 3 gives more examples of using menus. Section 7 is for advanced users who need to understand the subtleties of the callback mechanism.

The listing for *font_menu,* a program which builds on some of the examples given throughout the chapter, is given in Appendix A, *Example Programs.*

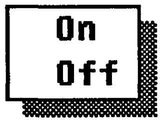
Summary Tables

Tables that summarize menu attributes and functions are in Chapter 18, *SunView Interface Summary*:

- the *Menu Attributes* table begins on page 291;
- the *Menu Item Attributes* table begins on page 294;
- the *Menu Functions* table begins on page 296.

Example 1:

Let's take a very simple example — a menu with two selectable items represented by the strings "On" and "Off":



```
on_off_menu = menu_create(MENU_STRINGS, "On", "Off", 0, 0);
```

The attribute `MENU_STRINGS` takes a list of strings and creates an item for each string. Note that the first zero in the above call terminates the list of strings, and the second zero terminates the entire attribute list.

CAUTION **The menu package, in contrast to the panel package, does not save strings which you pass in.** So you should either pass in the address of a constant, as in the example above, or static storage, or storage which you have dynamically allocated.

Typically you call `menu_show()` from an event procedure,[63] upon receiving the event which is to cause display of the menu. In the code fragment below, we display the menu on right button down:

```
...
case MS_RIGHT:
    menu_show(on_off_menu, window, event, 0);
    break;
...
```

`menu_show()`, by default, returns the value of the item which was selected. If the item was created with `MENU_STRINGS` its value defaults to its ordinal position in the menu, starting with 1.[64] So in the above example, selecting "On" would cause 1 to be returned, while selecting "Off" would cause 2 to be returned.

You can specify that `menu_show()` return the item itself, rather than return the value of the selected item. Do this by setting `MENU_NOTIFY_PROC` to the predefined notify procedure[65] `menu_return_item()`, as in:

```
menu_set(on_off_menu, MENU_NOTIFY_PROC, menu_return_item, 0);
```

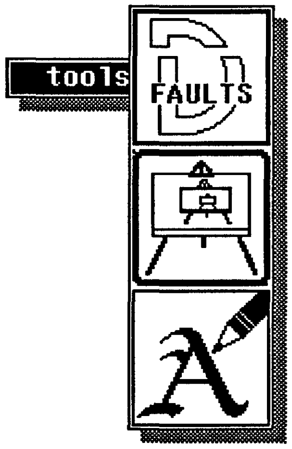
[63] See Chapter 6, *Handling Input* for a discussion of event procedures.

[64] The value of menu items not created with `MENU_STRINGS` defaults to zero. You can explicitly specify the values for menu items via the attributes `MENU_IMAGE_ITEM`, `MENU_STRING_ITEM`, or `MENU_VALUE`.

[65] Notify procedures are covered in detail in Section 7, *Callback Procedures*.

Example 3:

The menu package can accommodate images as well as strings. The example below creates a menu with a single item labelled "tools." When the user pulls right, he brings up a menu showing the icons of three SunView tools — *defaultsedit*, *iconedit*, and *fontedit*.



In order to pass an image into the menu package you need a pointer to a *memory pixrect* containing the image. One common way to create such an image is by first using *iconedit* to create the image and save it to a file. You then include the file in your program, and use the `mpr_static()` macro to create a memory pixrect:

```
static short d_defaults[] = {
#include <images/defaultsedit.icon>
};
mpr_static(defaults_pr, 64, 64, 1, d_defaults);

static short d_icon[] = {
#include <images/iconedit.icon>
};
mpr_static(icon_pr, 64, 64, 1, d_icon);

static short d_font[] = {
#include <images/fontedit.icon>
};
mpr_static(font_pr, 64, 64, 1, d_font);

tool_menu = menu_create(MENU_IMAGES,
                        &defaults_pr, &icon_pr, &font_pr, 0,
                        0);
menu = menu_create(MENU_ITEM,
                     MENU_STRING,    "tools",
                     MENU_PULLRIGHT, tool_menu,
                     0,
                   0);
```

The attribute `MENU_IMAGES` is analogous to `MENU_STRINGS`. It takes a list of images (pointers to pixrects) and creates a menu item for each image.

Menu Items

Representation on the Screen

A menu item is either displayed as a string or an image (a pointer to a pixrect). In the first case the item is referred to as a *string menu item*, in the second as a *image menu item.*

Item Values

Each menu item has a *value*. By default an item's value is the initial ordinal position of the item if it was created with MENU_STRINGS; otherwise the default value is zero. You can set an item's value explicitly when you create the item with MENU_STRING_ITEM or MENU_IMAGE_ITEM. You can also explicitly set an item's value with MENU_VALUE.

As mentioned in Section 1, menu_show(), by default, returns the value of the item the user has selected. Since menu items are counted starting from one, a return value of zero from menu_show() would represent the null selection.[70] However, you may explicitly set the value of a menu item to zero. If you do, then a return value of zero could represent either a legal value for the selected item or an error. To tell whether or not the result was valid, call menu_get() with the boolean MENU_VALID_RESULT. A return value of TRUE means that the result was valid; FALSE means that the value is invalid.

Item Generate Procedures

As with the menu as a whole, you may specify a *generate procedure* for each menu item, to be called just before the item is displayed.

Item Action Procedures

The *action procedure* of a menu item is analogous to the notify procedure of a menu. This is your chance to do something immediately based on the user's selection.

Menu notify procedures and item action procedures differ in when they are called. If the user makes a selection from a pullright, any notify procedures for menus higher up in the chain leading to the pullright will be called, while only those action procedures for menu items at, and to the right of, the selected item will be called.[71]

Client Data

Each menu item has a *client data* field, accessible through MENU_CLIENT_DATA, which is reserved for the application's use. You can use this attribute to associate a unique identifier, or a pointer to a private structure, with each menu item.

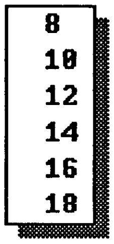
[70] This is why menu items are counted starting with one, rather than zero: so that a zero return value would represent the null selection whether the menu_show() was returning the value of the selected item or the item itself.

[71] Action procedures are discussed in detail in Section 7, *Callback Procedures.*

11.3. Examples

Example 4:

Our next example will show several variations on a simple menu that could be used for selecting font point sizes. The default form is:



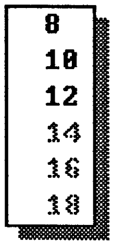
You could create the items with MENU_STRINGS, as in the previous example. Alternately, you could create the menu with no items, then use menu_set() to append the items to the menu:[73]

```
m = menu_create(0);
for (i = 8; i <= 18; i += 2)
    menu_set(m, MENU_STRING_ITEM, int_to_str(i), i, 0);
```

MENU_STRING_ITEM takes as values the item's string and its value.

Now let's see some of the ways in which the appearance of this basic menu can be altered.

By setting MENU_INACTIVE to TRUE for an item, you can "gray out" the item to indicate to the user that it is not currently selectable:



The above menu could be produced by:

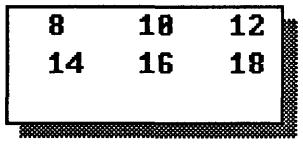
```
for (i = 4; i <= 6; i++) {
    item = menu_get(m, MENU_NTH_ITEM, i);
    menu_set(item, MENU_INACTIVE, TRUE, 0);
}
```

Inactive items do not invert when the cursor passes over them.

[73] Note that using MENU_STRING_ITEM with menu_set() has the effect of an implicit append. Several attributes are provided to explicitly add items to a menu — see the *Attributes To Add Pre-Existing Menu Items* table later in this section.

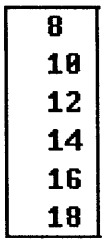
The previous example specified that the menu have 3 columns. Specifying that it have 2 rows via `MENU_NROWS` would have the same effect. Items are laid out from upper left to lower right, in "reading order," regardless of how the layout is specified.

The only time you need to specify both the number of rows and the number of columns is when you want to fix the size of the menu, regardless of how many items it contains. Setting `MENU_NCOLS` to 3 and `MENU_NROWS` to 3 would produce:

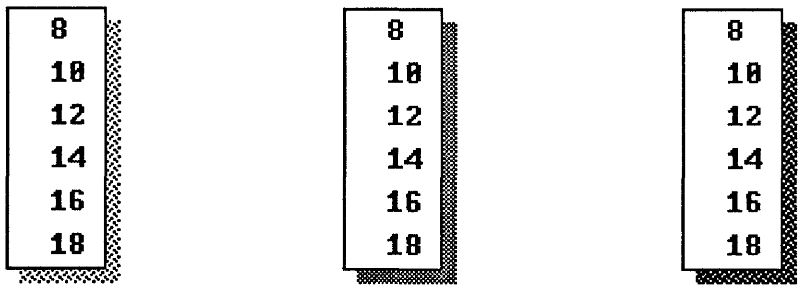


If both dimensions of the menu are fixed and more items are given than will fit, the excess items will not appear.

You can remove the menu's shadow by setting `MENU_SHADOW` to null:

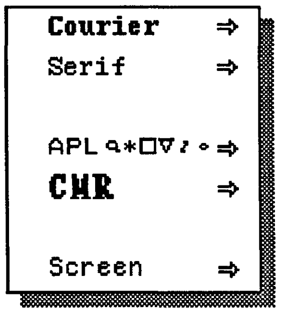


The menu package provides three predefined pixrects for the menu shadow. The call `menu_set(m, MENU_SHADOW, &menu_gray25_pr)` produces the 25 percent gray pattern shown on first menu below. Note that these are pixrects, *not* pixrect pointers. The other two patterns are produced by using `menu_gray50_pr` and `menu_gray75_pr`:



Example 6:

You can insert new items into an existing menu with MENU_INSERT. For example, suppose you want to insert blank lines into the font family menu, to indicate grouping:



You can do this by inserting non-selectable items into the menu:

```
menu_set(family_menu,
         MENU_INSERT,
           2,
           menu_create_item(MENU_STRING, "", MENU_FEEDBACK, FALSE, 0)
         0);

menu_set(family_menu,
         MENU_INSERT,
           5,
           menu_get(family_menu, MENU_NTH_ITEM, 3),
         0);
```

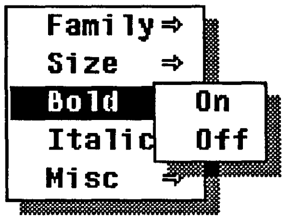
MENU_INSERT takes two values: the number of the item to insert after, and the new item to insert. Disabling MENU_FEEDBACK makes the item non-selectable.

The above example uses menu_create_item() to explicitly create the item to be inserted. Usually menu items are created implicitly, using the attributes described in the *Menu Item Creation Attributes* table in the next section.

NOTE menu_create_item() does *not* set the MENU_RELEASE attribute by default, so that the resulting item will not be automatically destroyed when its parent menu is destroyed. This is in contrast to implicitly created menu items — see Section 5, *Destroying Menus.*

Example 7:

For the next example we will attach the on-off, family and size menus of the previous examples as pullrights to a higher-level menu for selecting fonts:



```
font_menu = menu_create(
              MENU_PULLRIGHT_ITEM, "Family", family_menu,
              MENU_PULLRIGHT_ITEM, "Size",   size_menu,
              MENU_PULLRIGHT_ITEM, "Bold",   on_off_menu,
              MENU_PULLRIGHT_ITEM, "Italic", on_off_menu,
              MENU_PULLRIGHT_ITEM, "Misc",   frame_menu,
              0);
```

MENU_PULLRIGHT_ITEM takes a string and a menu as values. It creates an item represented by the string and with the menu as a pullright.

Note that on_off_menu is used as a pullright for both the bold and the italic menu items, and that the size_menu appears both as a pullright from main level font_menu and from each item in family_menu. This demonstrates that a menu may have more than one parent. However, recursive menus are not allowed — if M1 is a parent of M2, M2 (or any of its children) may not have M1 as a child. Displaying such a recursive menu will probably result in a segmentation fault.

The *Misc* item takes as its pullright the menu which has been retrieved from the frame using WIN_MENU.

The program *font_menu*, included in Appendix A, builds further on the above examples.

11.5. Destroying Menus

Both menus and menu items are destroyed with the function:

```
void
menu_destroy(menu_object)
    <Menu or Menu_item>  menu_object;
```

CAUTION

Watch out for dangling pointers when using a menu item in multiple menus. The attribute MENU_RELEASE (which takes no value) controls whether or not a menu item is automatically destroyed when its parent menu is destroyed. MENU_RELEASE is set by default for menu items created inline via the menu item creation attributes. This can lead to dangling pointers, if the same menu item appears multiple times, because calling menu_destroy() can lead to items being destroyed multiple times. This warning also applies to pullrights which are used multiple times. To prevent this error, remove multiple occurrences of an item or pullright before destroying a menu.

The function menu_destroy_with_proc() lets you specify a procedure to be called every time a particular menu or menu item is about to be destroyed:

```
void
menu_destroy_with_proc(menu_object, destroy_proc)
    <Menu or Menu_item>  menu_object;
    void                 (*destroy_proc)();
```

The destroy procedure is defined as:

```
void
destroy_proc(menu_object, type)
    <Menu or Menu_item>  menu_object;
    Menu_attribute    type;
```

For menus, menu_object is the menu and the type parameter is MENU_MENU; for menu items, menu_object is the item and the type parameter is MENU_ITEM.

11.7. Callback Procedures

When you call `menu_show()`, the menu package displays the menu, gets a selection from the user, and undisplays the menu. The menu package allows you to specify *callback procedures* which will be called at various points during the invocation of the menu. There are three types of callback procedures: *generate procedures* (so named because they are called before the menu or item is displayed, allowing the application to *generate* or modify the menu on the fly), *notify procedures* (for menus) and *action procedures* (for menu items) which are called after the user has made a selection.

Generate Procedures

The first argument to a generate procedure is either a menu or menu item depending on whether it's a `MENU_GEN_PROC` or a `MENU_ITEM_GEN_PROC`. Also passed in is an *operation* indicating at which point in the processing of the menu the generate procedure is being called. The operation parameter is of type `Menu_generate`, and may be `MENU_DISPLAY`, `MENU_DISPLAY_DONE`, `MENU_NOTIFY` or `MENU_NOTIFY_DONE`.[75]

CAUTION **The menu package uses the fullscreen access mechanism when displaying the menu. Writing to the screen while under fullscreen access will probably cause your program to deadlock, so your generate procedure should not access the screen when called with an operation of `MENU_DISPLAY` or `MENU_DISPLAY_DONE`.**

There are three types of generate procedures — *menu item generate procedures*, *menu generate procedures*, and *pullright generate procedures*. A description and example of each is given on the next several pages.

[75] For a detailed explanation of when the generate procedures are called in relation to the other callback procedures, see the diagrams in the next subsection, *Flow of Control in Menu Show*.

Menu Generate Procedure

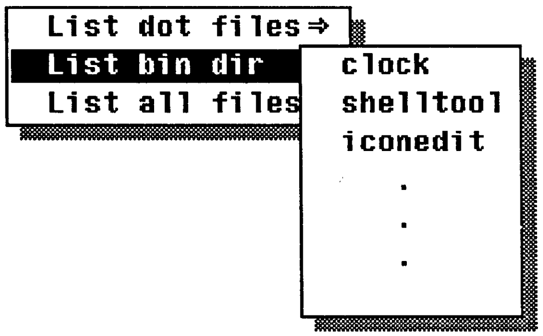
A generate procedure attached to a menu has the form:

```
Menu
menu_gen_proc(m, operation)
    Menu m;
    Menu_generate operation;
```

You can specify a menu generate procedure via the attribute `MENU_GEN_PROC`.

Example 9:

We will take as an example a menu allowing the user to list different groups of files. When the user makes a selection, we generate a menu containing the correct set of files:



The relevant functions are listed on the next page. The first, `initialize_menu()`, creates the three menu items, giving each of them the generate procedure `list_files()`, and a unique identifier as `MENU_CLIENT_DATA`.

Remember that `list_files()` will be called four different times by `menu_show()`.[76] In the first call (`operation` is `MENU_DISPLAY`), it calls the function `get_file_names()` (not shown) to get the appropriate list of file names, and adds each name on the list to the menu.

When `list_files()` is called with an `operation` of `MENU_DISPLAY_DONE` or `MENU_NOTIFY`, the menu is returned unaltered.

The final call has an `operation` of `MENU_NOTIFY_DONE`. This time `list_files()` cleans up by destroying the old menu and returning the handle of a newly created menu with no items.

[76] See the diagrams in the next subsection, *Flow of Control in Menu Show*.

Pullright Generate Procedure

You can postpone the generation of a pullright menu until the user actually pulls right by specifying a a pullright generate procedure. A pullright generate procedure has the form:

```
Menu
pullright_gen_proc(mi, operation)
    Menu_item     mi;
    Menu_generate operation;
```

Note that the pullright generate procedure is passed the item, and returns the menu to be displayed.

You can specify a menu item's pullright generate procedure with a call such as

```
menu_set(menu_item, MENU_GEN_PULLRIGHT_PROC, my_pullright_gen,
```

Alternatively, you can use the attributes `MENU_GEN_PULLRIGHT_IMAGE` or `MENU_GEN_PULLRIGHT_ITEM` to give a menu both an item and the item's generate procedure.

If you want to get the existing menu for an item which has a pullright generate procedure, retrieve the value of the item, as in:

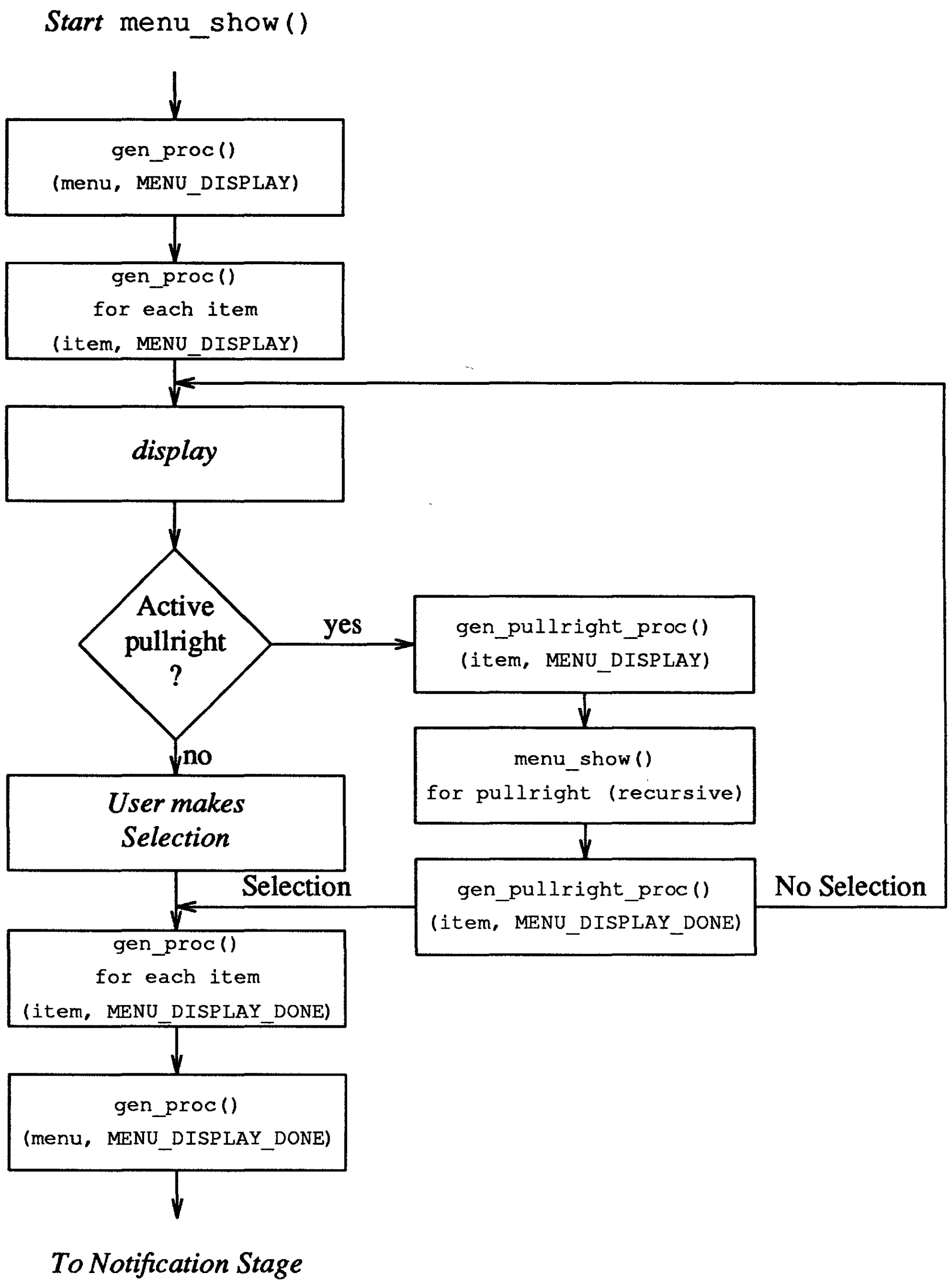
```
menu = menu_get(item, MENU_VALUE);
```

Flow of Control in `menu_show()`

The callback mechanism gives you a great deal of flexibility in creating, combining and modifying menus and menu items. This flexibility comes at the price of some complexity, however. To take advantage of it, it is necessary to understand when the callback procedures are called after you invoke `menu_show()`.

For purposes of explanation, the diagrams below divide the process of displaying a menu and getting the user's selection into two stages, the *display stage* and the *notification stage.*

Figure 11-1 *Display Stage of Menu Processing*


Start menu_show()
gen_proc() (menu, MENU_DISPLAY)
gen_proc() for each item (item, MENU_DISPLAY)
display
Active pullright ?
yes
no
gen_pullright_proc() (item, MENU_DISPLAY)
menu_show() for pullright (recursive)
gen_pullright_proc() (item, MENU_DISPLAY_DONE)
No Selection
Selection
User makes Selection
gen_proc() for each item (item, MENU_DISPLAY_DONE)
gen_proc() (menu, MENU_DISPLAY_DONE)
To Notification Stage

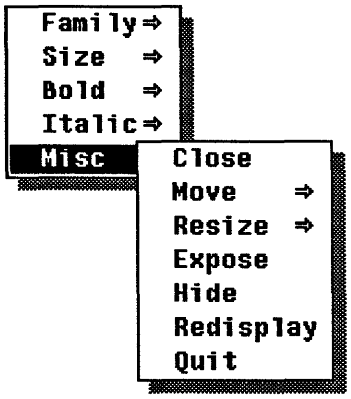
11.8. Interaction with Previously Defined SunView Menus

Walking Menus for frames, tty subwindows and text subwindows can be customized.[77] All menu items in these menus are ''position-independent'' — in other words the menus do not count on a given item having a certain position or being located in a particular menu. This makes it possible for you to safely add new items (including pullright submenus) to an existing menu.[78]

NOTE You should not use the client data field of items created by SunView packages, because the packages have pre-empted it for their own use.

Using an Existing Menu as a Pullright

The program *font_menu*, listed in Appendix A, shows how you can replace an existing menu with your own menu which has the original menu as a pullright. Making use of several of the examples given earlier in the chapter, it creates a font menu which allows the user to select the font family, point size, and whether or not the font is bold or italic. The last item, labelled *Misc*, brings up the original frame menu:



[77] Remember that in order to have these packages use walking menus the user must have enabled the SunView *Walking_Menus* option in *defaultsedit*.

11.10. User Customizable Attributes

The user can specify the values of certain menu attributes using *defaultsedit*. When a menu is created, for attributes not explicitly specified by the application program, the menu package retrieves the values set by the user from the defaults database maintained by *defaultsedit*. This allows the user the ability to tailor, to some extent, the appearance and behavior of menus across different applications. For example, he may want to change the type of shadow, or expand the menu margin, and so on.

The attributes under *defaultsedit* control are listed in the following table.

Table 11-3 *User Customizable Menu Attributes*

Attribute	default	Description
MENU_BOXED	FALSE	If TRUE, a single-pixel box will be drawn around each menu item.
MENU_DEFAULT_SELECTION	MENU_DEFAULT	MENU_SELECTED or MENU_DEFAULT.
MENU_FONT	screen.b.12	Menu's font.
MENU_INITIAL_SELECTION	MENU_DEFAULT	MENU_SELECTED or MENU_DEFAULT.
MENU_INITIAL_SELECTION_SELECTED	FALSE	If TRUE, menu comes up with its initial selection highlighted. If FALSE, menu comes up with the cursor "standing off" to the left.
MENU_INITIAL_SELECTION_EXPANDED	TRUE	If TRUE, when the menu pops up, it automatically expands to select the initial selection.
MENU_JUMP_AFTER_NO_SELECTION	FALSE	If TRUE, cursor jumps back to its original position after no selection made.
MENU_JUMP_AFTER_SELECTION	FALSE	If TRUE, cursor jumps back to its original position after selection made.
MENU_MARGIN	1	The margin around each item.
MENU_LEFT_MARGIN	16	For each string item, margin in addition to MENU_MARGIN on left between menu's border and text.
MENU_PULLRIGHT_DELTA	9999	# of pixels the user must move the cursor to the right to cause a pullright menu to pop up.
MENU_RIGHT_MARGIN	6	For each string item, margin in addition to MENU_MARGIN on right between menu's border and text.
MENU_SHADOW	50% grey	Pattern for menu's shadow.

12

Cursors

Cursors **217**

12.1. Creating and Modifying Cursors 218

12.2. Copying and Destroying Cursors 218

12.3. Crosshairs 219

12.4. The Cursor Attributes 220

12 Cursors

This chapter describes how to create and manipulate *cursors*. A cursor is an image that tracks the mouse on the display. Each window in SunView has its own cursor, which you can change with the cursor package.

The definitions necessary to use cursors are found in the include file `<sunwindow/win_cursor.h>`, which is included by default when you include the file `<suntool/sunview.h>`.

A demo showing the effects of the various cursor attributes can be seen by running */usr/demo/cursor_demo*. The source for this is in */usr/src/sun/suntool/cursor_demo.c*.

Summary Tables

Tables that summarize cursor attributes and functions are in Chapter 18, *SunView Interface Summary*:

- the *Cursor Attributes* table begins on page 280;
- the *Cursor Functions* table begins on page 282.

```
short my_pixrect_data[] = {
#include "file_from_iconedit"
};
mpr_static(my_pixrect, 16, 16, 1, my_pixrect_data);

Canvas canvas;

init_my_canvas()
{
    canvas = window_create(frame, CANVAS,
            WIN_CURSOR, cursor_create(CURSOR_IMAGE, &my_pixrect, 0),
            0);
}
```

In this example we create a cursor "on the fly" and pass it into the `window_create()` routine for use with our canvas. The attribute `CURSOR_IMAGE` is set to the new pixrect we want to use (which could be a diamond or bullseye, for example). All of the other cursor attributes default to the value shown in the attribute table.

Example 2: Changing the Cursor on an Existing Window

Suppose you have already created a window and you want to change its cursor. Let's say you want to change the drawing op to `PIX_SRC`:

```
Cursor  cursor;

cursor = window_get(my_window, WIN_CURSOR);
cursor_set(cursor, CURSOR_OP, PIX_SRC, 0);
window_set(my_window, WIN_CURSOR, cursor, 0);
```

CAUTION

The cursor returned by `window_get()` is a pointer to a static cursor that is shared by all the windows in your application. So, for example, saving the cursor returned by `window_get()` and then making other window system calls might result in the saved cursor being overwritten.[80]

It is safe to get the cursor, modify it with `cursor_set()` and then put the cursor back. If there is any chance that the static cursor will be overwritten, you should use `cursor_copy()` to make a copy of the cursor, then use `cursor_destroy()` when you are done.

12.3. Crosshairs

Crosshairs are horizontal and vertical lines whose intersection tracks the location of the mouse. You can control the appearance of both the horizontal and vertical crosshairs along with the cursor image. For example, you can create a cursor that only shows the cursor image, or only the horizontal crosshair, or both the horizontal and vertical crosshairs and the cursor image. By default both the crosshairs are turned off and only the cursor image is displayed.

Example 3: Turning on the Crosshairs

Suppose you have a canvas window in which you want to turn on both the horizontal and vertical crosshairs. This can be done by getting the cursor from the window and setting the `CURSOR_SHOW_CROSSHAIRS` attribute:

[80] Note that this would happen if one of the routines you call happens to call `window_get()` of `WIN_CURSOR`.

CURSOR_CROSSHAIR_LENGTH	If you don't want the crosshairs to cover the entire window (or screen), you can set the length of both crosshairs with `CURSOR_CROSSHAIR_LENGTH`. The value of this attribute is actually half the total crosshair length. For example, if you want the crosshairs to be 400 pixels wide and high, set the `CURSOR_CROSSHAIR_LENGTH` to 200. You can restore the extend-to-edge length by giving a value of `CURSOR_TO_EDGE` for `CURSOR_CROSSHAIR_LENGTH`.
CURSOR_CROSSHAIR_BORDER_GRAVITY	If the crosshair border gravity is enabled, the crosshairs will "stick" to the edge of the window (or screen). This is only interesting if the `CURSOR_CROSSHAIR_LENGTH` is not set to `CURSOR_TO_EDGE`. With border gravity turned on, each half of each crosshair will be attached to the edge of the window. With the cursor image displayed, this feature might be useful to help the user line up the cursor to a grid drawn on the edges of the window.
CURSOR_CROSSHAIR_GAP	If you don't want the halves of each crosshair to touch, you can set the `CURSOR_CROSSHAIR_GAP` to the half-length of space to leave between each crosshair half. If you set `CURSOR_CROSSHAIR_GAP` to `CURSOR_TO_EDGE`, the crosshairs will back off to the edge of the `CURSOR_IMAGE` rectangle.

13

Icons

Icons **225**

13.1. Using Images Generated With *iconedit* 226

13.2. Modifying the Icon's Image 227

13.3. Loading Icon Images At Run Time 227

13

Icons

An *icon* is a small (usually 64 by 64 pixel) picture representing a base frame in its closed state. The icon is typically a picture indicating the function of the underlying application.

The definitions necessary to use icons are found in the file `<suntool/icon.h>`, which is included by default when you include the file `<suntool/sunview.h>`.

Summary Tables

Tables that summarize icon attributes, functions and macros are in Chapter 18, *SunView Interface Summary*:

- the *Icon Attributes* table begins on page 286;
- the *Icon Functions and Macros* table begins on page 287.

13.2. Modifying the Icon's Image

It is often useful to change the icon's image dynamically, rather than simply using the icon as a static placeholder. When *mailtool* receives new mail, for example, it lets the user know by modifying its icon to show a letter arrived in the mailbox. `clocktool` uses its icon to represent a moving clock face.

The steps to follow in modifying an icon's image are:

- get the frame's icon (attribute `WINDOW_ICON`);
- get the icon's pixrect (attribute `ICON_IMAGE`);
- modify the pixrect as desired, or substitute a new pixrect;
- give the pixrect with the new image back to the icon;
- give the new icon back to the frame.

For example:

```
modify_icon(frame);
    Frame frame;

    Icon icon;
    Pixrect *pr;

    icon = (Icon) window_get(frame, WIN_ICON);
    pr = (Pixrect *) icon_get(icon, ICON_IMAGE);
    ...
    (modify pr)
    ...
    icon_set(icon, ICON_IMAGE, pr, 0);
    window_set(frame, WIN_ICON, icon, 0);
}
```

13.3. Loading Icon Images At Run Time

Often it is sufficient to define the image for a program's icon at compile time, with `mpr_static()`. However, you may want to allow the user to create his own icon images, and give the names of the files containing the images to your program as command-line arguments. Then you can load the images from the files the user has specified. Routines to load icon images from files at run time are described in Chapter 11 of the *SunView System Programmer's Guide.*

14

Scrollbars

Scrollbars **231**

14.1. Scrolling Model 232

14.2. Use of Scrollbars by Application Users 234

14.3. Creating, Destroying and Modifying Scrollbars 235

14.4. Programmatic Scrolling 237

14

Scrollbars

The canvas, text and panel subwindows have been designed to work with scrollbars. The text subwindow automatically creates its own vertical scrollbar. For canvases and panels, it is your responsibility to create the scrollbar and pass it in via the attributes `WIN_VERTICAL_SCROLLBAR` or `WIN_HORIZONTAL_SCROLLBAR`.

Section 1 describes how the user interacts with scrollbars. Basic scrollbar usage is covered in Section 2, and programmatic scrolling is covered in Section 3.

You may want to use scrollbars in an application not based on canvases, text subwindows or panels, in which case you must manage the interaction with the scrollbar directly. For an explanation of how to do this, see the *Scrollbars* chapter in the *SunView System Programmer's Guide*.

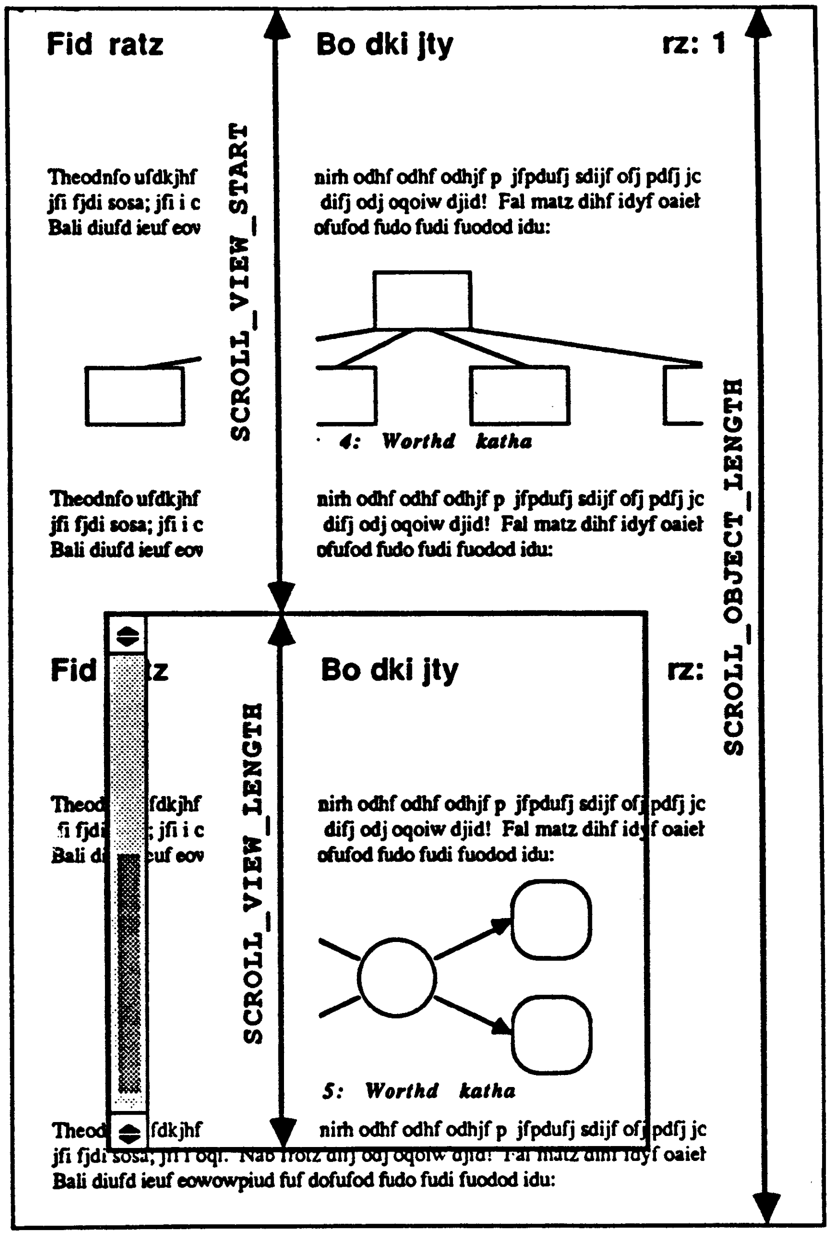
The definitions necessary to use scrollbars are found in the header file `<suntool/scrollbar.h>`

Summary Tables

Tables that summarize scrollbar attributes and functions are in Chapter 18, *SunView Interface Summary*:

- the *Scrollbar Attributes* table begins on page 317;
- the *Scrollbar Functions* table begins on page 320.

Figure 14-1 *Scrolling Model*



The above figure shows a two-page document being viewed within a window roughly half the size of the document. The three view-space attributes SCROLL_OBJECT_LENGTH, SCROLL_VIEW_LENGTH, and SCROLL_VIEW_START are shown superimposed on the document. Note the relative size and position of the bubble within the scrollbar — it is roughly half the size of the window and positioned near the bottom.

14.3. Creating, Destroying and Modifying Scrollbars

Scrollbars are created and destroyed with `scrollbar_create()` and `scrollbar_destroy()`. To take the simplest possible example, you get a default scrollbar (vertical, on the left edge of the subwindow, etc.) by calling:

```
bar = scrollbar_create(0);
```

You would destroy the scrollbar with the call:

```
scrollbar_destroy(bar);
```

The appearance and behavior of a given scrollbar is determined by the values of its attributes. Here's an example of a non-default scrollbar:

```
bar_1 = scrollbar_create(
        SCROLL_PLACEMENT,             SCROLL_EAST,
        SCROLL_BUBBLE_COLOR,          SCROLL_BLACK,
        SCROLL_BAR_DISPLAY_LEVEL,     SCROLL_ACTIVE,
        SCROLL_BUBBLE_DISPLAY_LEVEL,  SCROLL_ACTIVE,
        SCROLL_THICKNESS,             20,
        SCROLL_BUBBLE_MARGIN,         4,
        0),
```

In the above call, setting `SCROLL_PLACEMENT` to `SCROLL_EAST` will cause the scrollbar to appear on the right edge of the subwindow. The scrollbar will be 20 pixels wide with a black bubble 4 pixels from each edge of the bar. The bar and bubble will be shown only when the cursor is in the scrollbar.

You can modify and retrieve the attributes of a scrollbar with the two routines:

```
scrollbar_set(scrollbar, attributes)
    Scrollbar  scrollbar;
    <attribute-list> attributes;

caddr_t
scrollbar_get(scrollbar, attribute)
    Scrollbar           scrollbar;
    Scrollbar_attribute attributes;
```

If the `scrollbar` parameter is `NULL`, `scrollbar_get()` returns 0.

`SCROLL_RECT`, `SCROLL_THICKNESS`, `SCROLL_HEIGHT`, and `SCROLL_WIDTH` do not have valid values until the scrollbar is passed into the subwindow. As a work-around for this problem, the special symbol `SCROLLBAR` has been provided. You can determine the default thickness of a scrollbar before it has been attached to a subwindow with the call:

```
thickness = (int) scrollbar_get(SCROLLBAR, SCROLL_THICKNESS);
```

This convention is currently only implemented for `SCROLL_THICKNESS`.

The figure on the following page shows some of the attributes controlling the visual appearance of a scrollbar.[83]

[83] For a complete list of the scrollbar attributes see the *Scrollbar Attributes* table in Chapter 18.

14.4. Programmatic Scrolling

To scroll to a given location from your program, call:

```
scrollbar_scroll_to(scrollbar, new_view_start)
    Scrollbar scrollbar;
    long      new_view_start;
```

This routine saves the current value of `SCROLL_VIEW_START` as `SCROLL_LAST_VIEW_START`, sets `SCROLL_VIEW_START` to the value passed in as `new_view_start`, and posts a scroll event to the scrollbar's client (i.e. the canvas, panel or textsubwindow) via the notifier. This has the same effect as if the user had requested a scroll to `new_view_start`.

15

The Selection Service

The Selection Service **241**

15.1. Getting the Primary Selection 242

15.2. Setting the Primary Selection 242

15

The Selection Service

The Selection Service provides for flexible communication among window applications. You can use the Selection Service to query and manipulate the selections the user has made.

This chapter gives only the simplest example of using the Selection Service. To find out more about the Selection Service and the other functionality it provides, refer to Chapter 9 of the *SunView System Programmer's Guide.*

The definitions necessary to use the Selection Service are found in the include file `<suntool/seln.h>`.

16

The Notifier

The Notifier 245

16.1. When to Use the Notifier 246
16.2. Restrictions 247
16.3. Overview 249
16.4. Event Handling 250
Child Process Control Events 250
``Reaping'' Dead Processes 250
Results from a Process 251
Input Pending Events (pipes) 252
Example: Reading a Pipe 252
Closing the Pipe 252
Signal Events 253
A `signal()` Replacement for Notifier Compatibility 253
Example: Writing to a Pipe 254
Asynchronous Event Handling 255
Timeout Events 255
Example: Periodic Feedback 256
Polling 257
Checking the Interval Timer 257
Turning the Interval Timer Off 257
16.5. Interposition 258
How Interposition Works 258
Monitoring a Frame's State 259

16

The Notifier

The Notifier is a general-purpose mechanism for distributing events to a collection of clients within a process. It detects events in which its clients have expressed an interest, and dispatches these events to the proper clients, queuing client processing so that clients respond to events in a predictable order.

An overview of the notification-based model is given in Chapter 2, *The SunView Model.*

To encourage the porting of existing applications, the Notifier has provisions to allow programs to run in the Notifier environment without inverting their control structure. See Section 5, *Porting Programs to SunView.*

Header Files

The definitions for the Notifier are contained in the file `<sunwindow/notify.h>`, which will be included indirectly when you include `<suntool/sunview.h>`.[84]

Related Documentation

This chapter will suffice for the majority of SunView applications. See the chapters titled *Advanced Notifier Usage* and *The Agent and Tiles* in the *SunView System Programmer's Guide* for more information on the Notifier and SunView's usage of it. When looking up Notifier-related information, look first in the index to this book, then in the index to the *SunView System Programmer's Guide.*

Summary Table

The *Notifier Functions* table begins on page 298 in Chapter 18, *SunView Interface Summary.*

[84] For those programmers utilizing the Notifier outside of SunView (a perfectly reasonable thing to do), the code that implements the Notifier is found in */usr/lib/libsunwindow*.

16.2. Restrictions

The Notifier imposes some restrictions on its clients which designers should be aware of when developing software to work in the Notifier environment. These restrictions exist so that the application and the Notifier don't interfere with each other. More precisely, since the Notifier is multiplexing access to user process resources, the application needs to respect this effort so as not to violate the sharing mechanism.

Don't Call...

Assuming an environment with multiple clients with an unknown notifier usage pattern, you should not use any of the following system calls or C library routines:[85]

signal (3)
: The Notifier is catching signals on the behalf of its clients. If you set up your own signal handler over the one that the Notifier has set up then the Notifier will never notice the signal. Use `notify_set_signal_func()` instead of *signal* (3).

sigvec (2)
: The same applies for *sigvec* (2) as does for *signal* (3), above.

setitimer (2)
: The Notifier is managing two of the process's interval timers on the behalf of its many clients. If you access an interval timer directly, the Notifier could miss a timeout. Use `notify_set_itimer_func()` instead of *setitimer* (2).

alarm (3)
: Because *alarm* (3) sets the process's interval timer directly, the same applies for *alarm* (3) as does for *setitimer* (2), above.

getitimer (2)
: When using a notifier managed interval timer, you should call `notify_itimer_value()` to get its current status. Otherwise, you can get inaccurate results.

wait3 (2)
: The Notifier notices child process state changes on behalf of its clients. If you do your own *wait3* (2) then the notifier may never notice the change in a child process or you may get a change of state for a child process in which you have no interest. Use `notify_set_wait3_func()` instead of *wait3* (2).

wait (2)
: The same applies for *wait* (2) as does for *wait3* (2), above.

ioctl (2) (..., FIONBIO, ...)
: This call sets the blocking status of a file descriptor. The Notifier needs to know the blocking status of a file descriptor in order to determine if there is

[85] A future release may provide modified versions of some of these forbidden routines that will allow their use without restriction. However, the restrictions described in *Don't catch ...* (below) will continue to be germane. The section of this chapter titled *A* `signal()` *Replacement for Notifier Compatibility* provides a code patch for programs that catch signals.

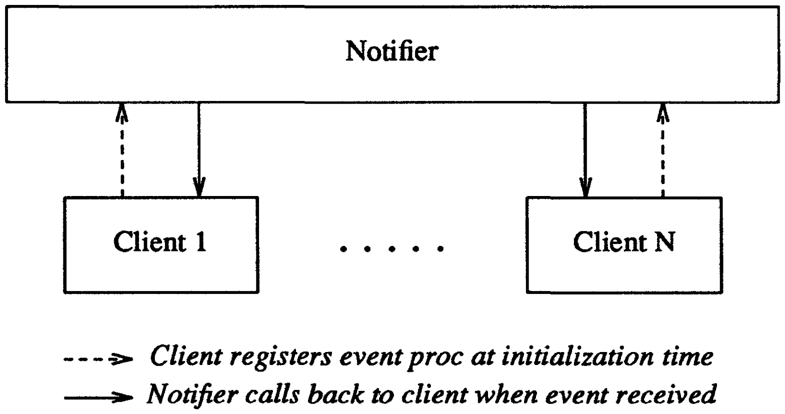
16.3. Overview

How the Notifier Works

Before it can receive events, a client must advise the Notifier of the types of events in which it is interested. It does this by registering an *event handler* function (which it must supply) for each type of event in which it is interested. When an event occurs, the Notifier calls the event handler appropriate to the type of event.

The figure below shows an overview of how the notification mechanism works.

Figure 16-1 *Overview of Notification*


Notifier
Client 1
.....
Client N
---> Client registers event proc at initialization time
——> Notifier calls back to client when event received

Client Handles

The Notifier uses a *client handle* as the unique identifier for a given client. The Notifier, without interpreting the client handle in any way, uses it to associate each event with the event handler for a given client.

The only requirement for a client handle is that it must be unique. Since a program text address or the address of an allocated data block are guaranteed to be unique, they can be used. Since stack addresses are not in general guaranteed to be unique they should not be used. SunView uses the object handles returned from *window_create()* as notifier client handles.

Types of Interaction

Client interaction with the Notifier falls into the following functional areas:

- Event handling — A client may receive events and respond to them via *event handlers.* Event handlers do the bulk of the work in the Notifier environment. The various types of events are in the Section 4, *Event Handling.*
- Interposition — A client may request that the Notifier install a special type of event handler (supplied by the client) to be inserted (or *interposed*) ahead of the current event handler for a given type of event and client. This allows clients to screen incoming events and redirect them, and to monitor and change the status of other clients. Examples of interposition may be found in the Section 5, *Monitoring and Modifying Window Behavior.*
- Notifier control — A client may exercise control over when dispatching of events occurs. See the section entitled *Porting Programs to SunView.*

```
#include <sunwindow/notify.h>

static  int my_client_object;
static  int *me = &my_client_object;

    int pid;

    if ((pid = my_fork()))
        (void) notify_set_wait3_func(me, notify_default_wait3, pid);
    /* Start dispatching events */
    (void) notify_start();
```

This is sufficient to have your child process reaped on its death. The Notifier automatically removes a dead process's wait3 event handler from its internal data structures.

NOTE The use of me as a client handle is arbitrary, but illustrates one method of generating a unique client handle.

Results from a Process

A more interesting application might actually receive some results from the process it forked. In this case, the application would supply its own wait3 event handler[87]. For example:

```
#include <sunwindow/notify.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

static Notify_value my_wait3_handler();

    /* Register a wait3 event handler */
    (void) notify_set_wait3_func(me, my_wait3_handler, pid);
    /* Start dispatching events */
    (void) notify_start();

static Notify_value
my_wait3_handler(me, pid, status, rusage)
    int *me;
    int pid;
    union wait *status;
    struct rusage *rusage;
{
    if (WIFEXITED(*status)) {
        /* Child process exited with return code */
        my_return_code_handler(me, status->w_retcode);
        /* Tell the notifier that you handled this event */
        return (NOTIFY_DONE);
    }
    /* Tell the notifier that you ignored this event */
    return (NOTIFY_IGNORED);
}
```

[87] See the *wait* (2) manual page for details of the `union wait` and the `struct rusage`.

Signal Events

Signals are UNIX software interrupts. The Notifier multiplexes access to the UNIX signal mechanism. A client may ask to be notified that a UNIX signal occurred either when it is received (asynchronously) and/or later during normal processing (synchronously).

Clients may define and register a signal event handler to respond to any UNIX signal desired. However, many of the signals that you might catch in a traditional UNIX program may be being caught for you by the Notifier (see *Don't catch* above).

CAUTION **Clients of the Notifier must not directly catch any UNIX signals using *signal* (2) or *sigvec* (3). Regardless of whether clients choose synchronous or asynchronous signal notification, they must use the signal event mechanism described in this section. See the Section 2, *Restrictions*.**

You can register a signal event handler which the Notifier will call whenever a signal has been caught by calling:

```
Notify_func
notify_set_signal_func(client, signal_func, signal, when)
    Notify_client client;
    Notify_func signal_func;
    int signal;
    Notify_signal_mode when;
```

`when` can be either NOTIFY_SYNC or NOTIFY_ASYNC. NOTIFY_SYNC causes notification during normal processing, that is, the delivering of the signal is delayed, so that your program doesn't receive it at an arbitrary time. NOTIFY_ASYNC causes notification immediately as the signal is received, — this mode mimics the UNIX *signal(3)* semantics.

A `signal()` Replacement for Notifier Compatibility

You should rewrite applications to use `notify_set_signal_func()`. However, in some cases the following code segment

```
#include <sunwindow/notify.h>

int
(*signal(sig, fn))()
        int     sig, (*fn)();
    {
        return ((int(*)())notify_set_signal_func(
            sig, fn, sig, NOTIFY_ASYNC));
    }
```

may be used to replace the C library version of `signal` (2) with one that is somewhat compatible with the Notifier. This code converts `signal()` calls into `notify_set_signal_func()` calls. Explicitly loading this code will override the loading of the C library's version of `signal()`. This approach works only if all the signal handlers registered by `signal()` only look at the first argument passed to them when a signal is received. Also, no Notifier client handle may be a small integer.

Asynchronous Event Handling

An asynchronous signal notification can come at any time (unless blocked using *sigblock* (2)). This means that the client can be executing code at any arbitrary place. Great care must be exercised during asynchronous processing.

It is rarely safe to do much of anything in response to an asynchronous signal. Unless your program has taken steps to protect its data from asynchronous access, the only safe thing to do is to set a flag indicating that the signal has been received.

When in an asynchronous signal event handler, the signal context and signal code is available from the follow routines:

```
int
notify_get_signal_code()

struct  sigcontext *
notify_get_signal_context()
```

The return values of these routines are undefined if called from a synchronous signal event handler.

Timeout Events

A client may require notification of an expired timer based on real time (approximate elapsed wall clock time; ITIMER_REAL) or on process virtual time (CPU time used by this process; ITIMER_VIRTUAL). To receive this type of notification, the client must define and register a timeout event handler.

```
Notify_func
notify_set_itimer_func(client, itimer_func, which, value,
        ovalue)
    Notify_client client;
    Notify_func itimer_func;
    int which;
    struct itimerval *value, *ovalue;
```

The semantics of `which`, `value` and `ovalue` parallel the arguments to *setitimer* (2) (see the *getitimer* (2) manual page). `which` is either ITIMER_REAL or ITIMER_VIRTUAL.

Polling

Interval timers can be used to set up a polling situation. There is a special `value` argument to `notify_set_itimer_func()` that tells the Notifier to call you as often and as quickly as possible. This `value` is the address of the following constant:

```
struct itimerval NOTIFY_POLLING_ITIMER; /*{{0,1},{0,1}}*/
```

This high speed polling can consume all of your machine's available CPU time, but may be appropriate for high speed animation. It is used in the program *spheres*, which shows one way to convert and old SunWindows gfx subwindow-based program to SunView. *spheres* is explained in Appendix C, *Converting SunWindows Programs to SunView*, and listed in full in Appendix A, *Example Programs.*

Checking the Interval Timer

The following function checks on the state of an interval timer by returning its current state in the structure pointed to by `value`.

```
Notify_error
notify_itimer_value(client, which, value)
    Notify_client client;
    int which;
    struct itimerval *value;
```

Turning the Interval Timer Off

If you specify an interval timer with its `it_interval` structure set to `{0, 0}`, the Notifier flushes any knowledge of the interval timer after it delivers the timeout notification. Otherwise, supplying a NULL interval timer pointer to `notify_set_itimer_func()` will turn the timer off.

Monitoring a Frame's State

You can notice when a frame opens or closes by interposing in front of the frame's *client event* handler. The client event handler is a SunView specific event handler which is built on top of the Notifier's general client event mechanism.[92] To install an interposer call the following routine:

```
Notify_error
notify_interpose_event_func(client, event_func, type)
    Notify_client client;
    Notify_func event_func;
    Notify_event_type type;
```

`client` must be the handle of the Notifier client in front of which you are interposing. In SunView, this is the handle returned from `window_create()`.[93] `type` is always NOTIFY_SAFE for SunView clients.

Example: Interposing on Open/Close

Let's say that the application is displaying some animation, and wants to do the necessary computation only when the frame is open. It can use interposition to notice when the frame opens or closes.

The program *spheres* (which shows one way to convert an old SunWindows gfx subwindow-based program to SunView) uses this technique to stop shading an image when its frame is closed. It is explained in Appendix C, *Converting SunWindows Programs to SunView*, and listed in full in Appendix A, *Example Programs.*

Another example appears on the following page.

Note the the call to `notify_next_event_func()`, which transfers control to the frame's client event handler through the Notifier. `notify_next_event_func()` takes the same arguments as the interposer.

[92] The stream of events sent to a client event handler is described in Chapter 6, *Handling Input.*

[93] It could also be the handle returned from the call to `scrollbar_create()`.

Discarding the Default Action

In the example on the preceding page, you wanted the base event handler to handle the event (so that the frame gets closed/opened). If the interposed function replaces the base event handler, and you don't want the base event handler to be called at all, your interposed procedure not not call `notify_next_event()`. For example, your interposed function might handle scroll events itself, so you would not want the base event handler to perform an additional scroll.

Interposing on Resize Events

Another common use of interposition is to give your application more control over the layout of its subwindows. The code is very similar. You call `notify_interpose_event_func()` to interpose your event handler. In the event handler, the following fragment could be used:

```
value = notify_next_event_func(frame, event, arg, type);
if (event_id(event) == WIN_RESIZE)
    resize(frame);
return(value);
```

Let the default event handler handle the event, then check if the event is a resize event. If so, call your own `resize()` procedure to lay out the subwindows.

NOTE *A* `WIN_RESIZE` *event is* not *generated until the frame is resized. If you want your resize procedure to be called when the window first appears you must do so yourself. This is different from a canvas with the* `CANVAS_RESIZE` *attribute set, whose resize procedure is called the first time the canvas is displayed.*

If the user manually adjusts subwindow sizes using CTRL middle mouse button, no `WIN_RESIZE` event is generated. You can disallow subwindow resizing by setting the `FRAME_SUBWINDOWS_ADJUSTABLE` attribute to `FALSE`.

Example: *resize_demo*

The program *resize_demo* shows how to achieve more complex window layouts than possible using window layout attributes. It is listed in Appendix A, *Example Programs.*

A Typical Destroy Handler

A typical destroy handler looks like the following:

```
Notify_value
common_destroy_func(client, status)
    Notify_client client;
    Destroy_status status;
{
    if (status == DESTROY_CHECKING) {
        if (/* Don't want to go away now */)
            notify_veto_destroy(client);
    } else {
        /* Always release external commitments */
        if (status == DESTROY_CLEANUP)
            /* Conditionally release internal resources */
    }
    return (NOTIFY_DONE);
}
```

Example: Interposing a Client Destroy Handler

Now we can present the example of interposing in front of the frame's client destroy event handler. In addition to doing our own confirmation, we prevent double confirmation by suppressing the frame's default confirmation.

Note that after having the destroy OKed by the user, we call `notify_next_destroy_func()` before returning. This allows other subwindows to request confirmation.

The code appears on the following page.

16.6. Porting Programs to SunView

Most programs that are ported to SunView are not notification-based. They are traditional programs that maintain strict control over the inner control loop. Much of the state of such programs is preserved on the stack in the form of local variables. The Notifier supports this form of programming so that you can use SunView packages without inverting the control structure of your program to be notification-based.

Explicit Dispatching

The simplest way to convert a program to coexist with the Notifier is called *explicit dispatching*. This approach replaces the call to `window_main_loop()`, which usually doesn't return until the application terminates, with the following bit of code:

```
#include <suntool/sunview.h>

static  int my_done;

    extern Notify_error notify_dispatch();

    /* Make the frame visible on the screen */
    window_set(frame, WIN_SHOW, TRUE, 0);
    while (!my_done) {
        ...
        /* Dispatch events managed by the notifier */
        (void) notify_dispatch();
        ...
    }
```

`notify_dispatch()` goes once around the Notifier's internal loop, dispatches any pending events, and returns. You should try to have `notify_dispatch()` called at least once every 1/4 second so that good interactive response with SunView windows can be maintained.

The program *bounce* (which shows one way to convert an old SunWindows gfx subwindow–based program to SunView) uses explicit dispatching. It is explained in Appendix C, *Converting SunWindows Programs to SunView*, and given in full in Appendix A, *Example Programs*.

Implicit Dispatching

Explicit dispatching is good when you are performing some computationally intensive processing and you want to occasionally give the user a chance to interact with your program. There is another method of interacting with the Notifier that is useful when you simply want the Notifier to take care of its clients and block until there is something of interest to you. This is called *implicit dispatching*.

This time, we replace the call to `window_main_loop()` with the following bit of code:

16.7. Error Handling

Error Codes

Every call to a notifier routine returns a value that indicates success or failure. Routines that return an enumerated type called `Notify_error` deliver NOTIFY_OK (zero) to indicate a successful operation, while any other value indicates failure. Routines that return function pointers deliver a non-null value to indicate success, while a value of NOTIFY_FUNC_NULL indicates an error condition.

When an error occurs, the global variable `notify_errno` describes the failure. The Notifier sets `notify_errno` much like UNIX system calls set the global `errno`; that is, the Notifier only sets `notify_errno` when it detects an error and does not reset it to NOTIFY_OK on a successful operation. A table in the *SunView System Programmer's Guide* lists each possible value of `notify_errno` and its meaning.

Handling Errors

Most of the errors returned from the Notifier indicate a programmer error, e.g., the arguments are not valid. Often the best approach for the client is to print a message if the return value is non-zero and exit. The procedure `notify_perror()` takes a string which is printed to `stderr`, followed by a colon, followed by a terse description of `notify_errno`. This is done in a manner analogous to the UNIX *perror(3)* call.

Debugging

Here are some debugging hints that may prove useful when programming:

NOTIFY_ERROR_ABORT=YES
: Setting the environment variable NOTIFY_ERROR_ABORT to YES will cause the Notifier to abort with a core dump when the Notifier detects an error. This is useful if there is some race condition that produces notifier error messages that you are having a hard time tracking down.

Stop in `notify_perror()` or *fprintf*(3S)
: If you are getting notifier error messages, but don't know from where, try putting a break point on the entry to either `notify_perror()` or *fprintf*(3S). Trace the stack to see what provoked the message.

The following call can be made from the debugger or your program to dump a printout of the state of the Notifier:

```
void
notify_dump(client, type, file)
    Notify_client client;
    int type;
    FILE *file;
```

The state of `client` is dumped to `file` based on the value of `type`. If `client` is 0 then all clients are dumped. If `type` is 1 then all the registered event handlers are dumped. If `type` is 2 then all the events pending for delivery are dumped. If `type` is 0 then both the registered event handlers and the events pending for delivery are dumped. If `file` is 1 then *stdout* is assumed. If `file` is 2 then *stderr* is assumed. To be able to call `notify_dump()` you need to reference it from some place in your program so that it gets loaded into your binary.

17

Attribute Utilities

Attribute Utilities **271**

17.1. Character Unit Macros 271

17.2. Creating Reusable Attribute Lists 272

Default Attributes 273

17

Attribute Utilities

This chapter describes macros and functions that are provided as utilities to be used with attributes.

17.1. Character Unit Macros

By default in SunView, coordinate specification attributes interpret their values in pixel units. For applications that don't make heavy use of images, it is usually more convenient to specify positions in character units — columns and rows rather than xs and ys. To this end two macros `ATTR_ROW()` and `ATTR_COL()` are provided, which interpret their arguments as rows or columns, respectively, and convert the value to the corresponding number of pixels, based on the subwindow's font, as specified by `WIN_FONT`. `ATTR_ROW()` and `ATTR_COL()` take as arguments any expression yielding an integer. The use of these macros as an operand in an expression is restricted to adding a pixel offset (e.g., `ATTR_ROW(5) + 2`). Examples of legal and illegal usage are given in the table below.

Table 17-1 *Example uses of the* `ATTR_ROW()` *and* `ATTR_COL()` *macros*

Attribute/Value	Interpretation
`PANEL_ITEM_X, 5`	5 pixels from left
`PANEL_ITEM_Y, 10`	10 pixels from top
`PANEL_ITEM_X, ATTR_COL(5)`	column 5
`PANEL_ITEM_X, ATTR_COL(-5)`	column -5
`PANEL_ITEM_X, ATTR_COL(5+2)`	column 7
`PANEL_ITEM_X, ATTR_COL(5)+2`	2 pixels to right of col 5
`PANEL_ITEM_X, ATTR_COL(5)-1`	1 pixel to left of col 5
`PANEL_ITEM_Y, ATTR_ROW(10)`	row 10
`PANEL_ITEM_Y, ATTR_ROW(-10)`	row -10
`PANEL_ITEM_Y, ATTR_ROW(10+2)`	row 12
`PANEL_ITEM_Y, ATTR_ROW(10)+2`	2 pixels down from row 10
`PANEL_ITEM_Y, ATTR_ROW(10)-1`	1 pixel up from row 10
`PANEL_ITEM_X, ATTR_COL(10)+ATTR_COL(2)`	*illegal*
`PANEL_ITEM_X, 2*ATTR_COL(10)`	*illegal*

NOTE `ATTR_ROW()` and `ATTR_COL()` treat their arguments as character *positions* rather than *lengths*. In other words, when you use `ATTR_ROW(5)`, the pixel value that is computed includes the top margin. Similarly, the pixel value computed using `ATTR_COL(5)` includes the left margin.

Default Attributes

The code below shows how to use `attr_create_list()` in conjunction with the attribute `ATTR_LIST` to support default attributes in a panel.

```
int text_proc(), name_proc();
Panel_item name_item, address_item;
Pixfont *big_font, *small_font;
Attr_avlist defaults;

defaults =      attr_create_list(
                    PANEL_SHOW_ITEM,     FALSE,
                    PANEL_LABEL_FONT,    big_font,
                    PANEL_VALUE_FONT,    small_font,
                    PANEL_NOTIFY_PROC,   text_proc,
                    0);

name_item =     panel_create_item(PANEL_TEXT,
                    ATTR_LIST,           defaults,
                    PANEL_NOTIFY_PROC,   name_proc,
                    0);

address_item =  panel_create_item(PANEL_TEXT,
                    ATTR_LIST,           defaults,
                    PANEL_SHOW_ITEM,     TRUE,
                    PANEL_VALUE_FONT,    big_font,
                    0);
```

The special attribute `ATTR_LIST` takes as its value an attribute list. In the above example, first an attribute list called `defaults` is created. Then, by mentioning `defaults` first in the attribute lists for subsequent item creation calls, each item takes on those default attributes. Subsequent references to an attribute override the setting in `defaults` since the last value mentioned for an attribute is the one which takes effect.

18

SunView Interface Summary

SunView Interface Summary 277

Canvas Tables 278

Attributes 278

Functions and Macros 279

Cursor Tables 280

Attributes 280

Functions 282

Data Types 283

Icon Tables 286

Attributes 286

Functions and Macros 287

Input Event Tables 288

Event Codes 288

Event Descriptors 289

Input-Related Window Attributes 290

Menu Tables 291

Attributes 291

Item Attributes 294

Functions 296

Notifier Functions Table 298

Panel Tables 301

Attributes 301

Generic Panel Item Attributes 302

18

SunView Interface Summary

This chapter contains tables summarizing the data types, functions and attributes which comprise the SunView programmatic interface.[1]

The tables correspond to the chapters in this book, but are in *alphabetical* order: Canvases, Cursors, Data Types, Icons, Input (including events and input-related window atttributes), Menus, the Notifier, Panels, Pixwins, Scrollbars, Text Subwindows, TTY Subwindows and Windows (including frames and frame command line arguments).

Note that the order of the chapters is different than the order of the tables. The chapter on Windows comes first, followed by Canvases, Input, Pixwins, Text Subwindows, Panels, Tty Subwindows, Menus, Cursors, Icons, Scrollbars, the Selection Service, the Notifier.

Within each topic, the attribute tables come first, then the functions and macros, then miscellaneous tables.

To help distinguish where one table ends and another begins, the start of each table is marked with a horizontal grey bar.

[1] This chapter does not include a table for the Selection Service functions; see the *SunView System Programmer's Guide* for a complete discussion of the Selection Service interface.

Table 18-2 *Canvas Functions and Macros*

Definition	Description

`Event *`
`canvas_event(canvas, event)`
`    Canvas canvas;`
`    Event *event;`
	Translates the coordinates of `event` from the space of the canvas subwindow to the space of the logical canvas (which may be larger and scrollable).

`Pixwin *`
`canvas_pixwin(canvas)`
`    Canvas canvas;`
	Returns the pixwin to use when drawing into the canvas with the `pw_*()` routines.

`Event *`
`canvas_window_event(canvas, event)`
`    Canvas canvas;`
`    Event *event;`
	Translates the coordinates of `event` to the space of the canvas subwindow from the space of the logical canvas.

Table 18-3 *Cursor Attributes— Continued*

Attribute	Value Type	Description
CURSOR_VERT_HAIR_COLOR	int	See CURSOR_CROSSHAIR_COLOR
CURSOR_VERT_HAIR_GAP	int	See CURSOR_CROSSHAIR_GAP.
CURSOR_VERT_HAIR_LENGTH	int	See CURSOR_CROSSHAIR_LENGTH.
CURSOR_VERT_HAIR_OP	int	Raster op for drawing vertical crosshair. Default: PIX_SRC.
CURSOR_VERT_HAIR_THICKNESS	int	See CURSOR_CROSSHAIR_THICKNESS.
CURSOR_XHOT	int	Hot spot x coordinate. Default: 0.
CURSOR_YHOT	int	Hot spot y coordinate. Default: 0.

Table 18-5 *Data Types*

Data Type	Description
`Canvas`	Pointer to an opaque structure which describes a canvas.
`Cursor`	Pointer to an opaque structure which describes a cursor.
`Destroy_status`	Enumeration: DESTROY_PROCESS_DEATH, DESTROY_CHECKING, or DESTROY_CLEANUP.
`Event`	The structure which describes an input event: `typedef struct inputevent { short ie_code; short ie_flags; short ie_shiftmask; short ie_locx; short ie_locy; struct timeval ie_time; } Event;`
`Frame`	Pointer to an opaque structure which describes a frame.
`Icon`	Pointer to an opaque structure which describes a icon.
`Inputmask`	Mask specifying which input events a window will receive.
`Menu`	Pointer to an opaque structure which describes a menu.
`Menu_attribute`	One of the menu attributes (MENU_*).
`Menu_generate`	Enumerated type of the `operation` parameter passed to generate procs: MENU_CREATE, MENU_DESTROY, MENU_NOTIFY_CREATE or MENU_NOTIFY_DESTROY.
`Menu_item`	Pointer to an opaque structure which describes a menu item.
`Notify_arg`	Opaque client optional argument.
`Notify_destroy`	Enumeration: NOTIFY_SAFE, NOTIFY_IMMEDIATE. (See also `Notify_event_type`).
`Notify_event`	Opaque client event.
`Notify_error`	Enumeration of errors for notifier functions: NOTIFY_OK, NOTIFY_UNKNOWN_CLIENT, NOTIFY_NO_CONDITION, NOTIFY_BAD_ITIMER, NOTIFY_BAD_SIGNAL, NOTIFY_NOT_STARTED, NOTIFY_DESTROY_VETOED, NOTIFY_INTERNAL_ERROR, NOTIFY_SRCH,

Table 18-5 *Data Types— Continued*

Data Type	Description
	`typedef struct rectnode {` `Rectnode *rn_next;` `Rect rn_rect;` `} Rectnode;`
`Scroll_motion`	Enumerated type representing possible scrolling motions: SCROLL_ABSOLUTE, SCROLL_FORWARD, SCROLL_MAX_TO_POINT, SCROLL_PAGE_FORWARD, SCROLL_LINE_FORWARD, SCROLL_BACKWARD, SCROLL_POINT_TO_MAX, SCROLL_PAGE_BACKWARD, or SCROLL_LINE_BACKWARD.
`Scrollbar`	The opaque handle for a scrollbar.
`Scrollbar_attribute`	One of the scrollbar attributes (SCROLL_*).
`Scrollbar_setting`	The value of an enumerated type scrollbar attribute.
`Textsw`	Pointer to an opaque structure which describes a text subwindow.
`Textsw_index`	An index for a character within a text subwindow.
`Textsw_enum`	Enumerated type for various text subwindow attribute values: TEXTSW_ALWAYS, TEXTSW_NEVER, TEXTSW_ONLY, TEXTSW_IF_AUTO_SCROLL, TEXTSW_CLIP, TEXTSW_WRAP_AT_CHAR, TEXTSW_WRAP_AT_WORD.
`Textsw_status`	Enumeration describing the status of text subwindow operations: TEXTSW_STATUS_OKAY, TEXTSW_STATUS_BAD_ATTR, TEXTSW_STATUS_BAD_ATTR_VALUE, TEXTSW_STATUS_CANNOT_ALLOCATE, TEXTSW_STATUS_CANNOT_OPEN_INPUT, or TEXTSW_STATUS_OTHER_ERROR,
`Tty`	Pointer to an opaque structure which describes a tty subwindow.
`Window`	Pointer to an opaque structure which describes a window.
`Window_attribute`	One of the window attributes (WIN_*).
`Window_type`	Type of window, retrieved via the WIN_TYPE attribute. One of: FRAME_TYPE, PANEL_TYPE, CANVAS_TYPE, TEXTSW_TYPE, or TTY_TYPE.

Table 18-7 *Icon Functions and Macros*

Definition	Description

`Icon`
`icon_create(attributes)`
    <attribute-list> `attributes;`
	Creates and returns the opaque handle to an icon.

`int`
`icon_destroy(icon)`
    `Icon icon;`
	Destroys `icon`.

`caddr_t`
`icon_get(icon, attribute)`
    `Icon icon;`
    `Icon_attribute attribute;`
	Retrieves the value for an attribute of `icon`.

`int`
`icon_set(icon, attributes)`
    `Icon icon;`
    <attribute-list> `attributes;`
	Sets the value for one or more attributes of `icon`. `attributes` is a null-terminated attribute list.

`extern static struct mpr_data`
`DEFINE_ICON_FROM_IMAGE(name, image)`
    `static short   icon_image[];`
	Macro that creates a static memory pixrect `icon` from `image`; the latter typically is generated by including a file created by `iconedit`. **Note:** you must pass the *address* of `icon` to the icon routines, since the `Icon` object is a pointer.

Table 18-9 *Event Descriptors*

Event Descriptor	Explanation
`WIN_NO_EVENTS`	Clears input mask — no events will be accepted. Note: the effect is the same whether used with a *consume* or an *ignore* attribute. A new window has a cleared input mask.
`WIN_ASCII_EVENTS`	All ASCII events. ASCII events that occur while the META key is depressed are reported with codes in the META range. In addition, cursor control keys and function keys are reported as ANSI escape sequences: a sequence of events whose codes are ASCII characters, beginning with <ESC>.
`WIN_IN_TRANSIT_EVENTS`	Enables immediate `LOC_MOVE`, `LOC_WINENTER`, and `LOC_WINEXIT` events. Pick mask only. Off by default.
`WIN_LEFT_KEYS`	The left function keys, KEY_LEFT(1) — KEY_LEFT(15).
`WIN_MOUSE_BUTTONS`	Shorthand for `MS_RIGHT`, `MS_MIDDLE` and `MS_LEFT`. Also sets or resets `WIN_UP_EVENTS`.
`WIN_RIGHT_KEYS`	The right function keys, KEY_RIGHT(1) — KEY_RIGHT(15).
`WIN_TOP_KEYS`	The top function keys, KEY_TOP(1) — KEY_TOP(15).
`WIN_UP_ASCII_EVENTS`	Causes the matching up transitions to normal ASCII events to be reported — if you see an 'a' go down, you'll eventually see the matching 'a' up.
`WIN_UP_EVENTS`	Causes up transitions to be reported for button and function key events being consumed.

Table 18-11 *Menu Attributes*

Attribute	Value Type	Description
MENU_ACTION_IMAGE	Pixrect *, action proc	Create image menu item with action proc. Set only.
MENU_ACTION_ITEM	char *, action proc	Create string menu item with action proc. Set only.
MENU_APPEND_ITEM	Menu_item	Append item to end of menu. Set only.
MENU_BOXED	boolean	If TRUE, a single-pixel box will be drawn around every menu item.
MENU_CLIENT_DATA	caddr_t	For client's use.
MENU_DESCEND_FIRST	(no value)	For menu_find(). If given, search will be depth first, else search will be "deferred".
MENU_DEFAULT	int	Default menu item as a position.
MENU_DEFAULT_ITEM	Menu_item	Default menu item as opaque handle.
MENU_DEFAULT_SELECTION	enum	Either MENU_SELECTED or MENU_DEFAULT.
MENU_FIRST_EVENT	Event *	The event which was initially passed into menu_show(). Get only. (Note that the event's contents can be modified.)
MENU_FONT	Pixfont *	Menu's font.
MENU_GEN_PROC	(procedure)	Client's function called to generate the menu. `Menu gen_proc(m, op)` `Menu m;` `Menu_generate op;`
MENU_GEN_PULLRIGHT_IMAGE	Pixrect *, gen proc	Create image menu item with generate proc for pullright. Set only.
MENU_GEN_PULLRIGHT_ITEM	char *, gen proc	Create string menu item with generate proc for pullright. Set only.
MENU_IMAGE_ITEM	Pixrect *, value	Create image menu item with value. Set only.
MENU_IMAGES	list of Pixrect *	Create multiple image menu items. Set only.
MENU_INITIAL_SELECTION	enum	Either MENU_SELECTED or MENU_DEFAULT.
MENU_INITIAL_SELECTION_EXPANDED	boolean	If TRUE, when the menu pops up, it automatically expands to select the initial selection.

Table 18-11 *Menu Attributes— Continued*

Attribute	Value Type	Description
		Default: 9999.
MENU_PULLRIGHT_IMAGE	Pixrect *, Menu	Create image menu item with pullright. Set only.
MENU_PULLRIGHT_ITEM	char *, Menu	Create string menu item with pullright. Set only.
MENU_REMOVE	int	Remove the nth item. Set only.
MENU_REMOVE_ITEM	Menu_item	Remove the specified item. Set only.
MENU_REPLACE	int, Menu_item	Replace nth item with specified item. Set only.
MENU_REPLACE_ITEM	Menu_item, Menu_item	The item given as first value is replaced with the one given as the second value in the menu (the old item is not replaced in any other menus it may appear in). Set only.
MENU_RIGHT_MARGIN	int	For each string item, margin in addition to MENU_MARGIN on right between menu's border and text.
MENU_SELECTED	int	Last selected item, as a position in menu.
MENU_SELECTED_ITEM	Menu_item	Last selected item, as the item's handle.
MENU_SHADOW	Pixrect *	Pattern for the shadow to be painted behind the menu. If 0, no shadow is painted. Predefined shadow pixrects you can use: menu_gray25_pr, menu_gray50_pr, and menu_gray75_pr.
MENU_STRINGS	list of char *	Create multiple string menu items. Set only.
MENU_STRING_ITEM	char *, value	Create string menu item with value. Set only.
MENU_TITLE_IMAGE	Pixrect *	Create image title item. Set only.
MENU_TITLE_ITEM	char *	Create string title item. Set only.
MENU_TYPE	enum	Get only; returns MENU_MENU.
MENU_VALID_RESULT	boolean	Tells whether a zero return value represents a legitimate value.

Table 18-12 *Menu Item Attributes— Continued*

Attribute	Value Type	Description
		menu's border and text.
MENU_MARGIN†	int	Margin in pixels around the item.
MENU_PARENT†	Menu	The menu containing the item.
MENU_PULLRIGHT	Menu	Item's pullright menu.
MENU_PULLRIGHT_IMAGE†	Pixrect *, Menu	Modifies appropriate fields in item. Set only.
MENU_PULLRIGHT_ITEM†	char *, Menu	Modifies appropriate fields in item. Set only.
MENU_RELEASE	(no value)	The item will be automatically destroyed when its parent menu is destroyed (default for items created inline).
MENU_RELEASE_IMAGE	(no value)	The string or pixrect associated with the item will be freed when the item is destroyed.
MENU_RIGHT_MARGIN†	int	Margin in addition of MENU_MARGIN on right between menu's border and text.
MENU_SELECTED†	boolean	If TRUE, the item is currently selected.
MENU_STRING†	char *	Item's string.
MENU_STRING_ITEM†	char *, value	Modifies appropriate fields in item. Set only.
MENU_TYPE†	enum	Get only, returns MENU_ITEM.
MENU_VALUE	caddr_t	Item's value.

Table 18-13 *Menu Functions— Continued*

Definition	Description

`caddr_t menu_show_using_fd(menu, fd, event)`
`    Menu menu;`
`    int fd;`
`    Event *event;`
	Provided for compatibility with SunWindows 2.0. Allows you to display a menu within a window using the windowfd.

`caddr_t`
`menu_return_item(menu, menu_item)`
`    Menu menu;`
`    Menu_item menu_item;`
	Predefined notify proc which, if given as the value for `MENU_NOTIFY_PROC`, causes `menu_show()` to return the handle of the selected item, rather than its value.

`caddr_t`
`menu_return_value(menu, menu_item)`
`    Menu menu;`
`    Menu_item menu_item;`
	Default notify proc for menus. Causes `menu_show()` to return the value of the selected item.

Table 18-14 *Notifier Functions— Continued*

Definition	Description

`Notify_value`
`notify_next_event_func(client, event, arg, type)`
`  Notify_client client;`
`  Event *event;`
`  Notify_arg arg;`
`  Notify_event_type type;`
	Calls the next event handler for `client`.

`Notify_error`
`notify_no_dispatch()`
	Prevents the notifier from dispatching events from within the call to *read*(2) or *select*(2).

`notify_perror(s)`
`   char *s;`
	Analogous to the UNIX *perror(3)* call. `s` is printed to `stderr`, followed by a terse description of `notify_errno()`.

`Notify_func`
`notify_set_destroy_func(client, destroy_func)`
`    Notify_client client;`
`    Notify_func destroy_func;`
	Registers `destroy_func()` with the notifier. `destroy_func()` will be called when a destroy event is posted to `client` or when the process receives a `SIGTERM` signal.

`Notify_func`
`notify_set_exception_func(client, exception_func, fd)`
`    Notify_client client;`
`    Notify_func exception_func;`
`    int fd;`
	Registers the exception handler `exception_func()` with the notifier. The only known devices that generate exceptions at this time are stream-based socket connections when an out-of-band byte is available.

`Notify_func`
`notify_set_input_func(client, input_func, fd)`
`    Notify_client client;`
`    Notify_func input_func;`
`    int fd;`
	Registers `input_func()` with the notifier. `input_func()` will be called whenever there is input pending on `fd`.

`Notify_func`
`notify_set_itimer_func(client, itimer_func, which,`
`                       value, ovalue)`
`    Notify_client client;`
`    Notify_func itimer_func;`
`    int which;`
`    struct itimerval *value, *ovalue;`
	Registers the timeout event handler `itimer_func()` with the notifier. The semantics of `which`, `value` and `ovalue` parallel the arguments to *setitimer* (see *getitimer* manual page). `which` is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

Table 18-15 *Panel Attributes*

Attribute	Value Type	Description
PANEL_ACCEPT_KEYSTROKE	boolean	If TRUE, keystroke events are passed to the panel's PANEL_BACKGROUND_PROC. Default: FALSE.
PANEL_BACKGROUND_PROC	(procedure)	Event handling procedure called when an event falls on the background of the panel. Form: `background_proc(panel, event)` `Panel panel` `Event *event`
PANEL_BLINK_CARET	boolean	If TRUE, the caret blinks. Default: TRUE.
PANEL_CARET_ITEM	Panel_item	Text item which currently has the caret. Default: first text item.
PANEL_EVENT_PROC	(procedure)	Event handling procedure for panel items. Sets the default for subsequent items created in panel. Form: `event_proc(item, event)` `Panel_item item` `Event *event`
PANEL_FIRST_ITEM	Panel_item	First item in the panel. Get only.
PANEL_ITEM_X_GAP	int	Number of x-pixels between items. Default: 10.
PANEL_ITEM_Y_GAP	int	Number of y-pixels between items. Default: 10.
PANEL_LABEL_BOLD	boolean	If TRUE, item's label is rendered in bold. Sets the default for subsequent items created in panel. Default: FALSE.
PANEL_LAYOUT	Panel_setting	Layout of item's value relative to the label. PANEL_HORIZONTAL (default) or PANEL_VERTICAL.
PANEL_SHOW_MENU	boolean	If TRUE, the menu for the item is enabled. Sets the default for subsequent items created in panel.

Table 18-16 *Generic Panel Item Attributes— Continued*

Attribute	Value Type	Description
PANEL_MENU_TITLE_FONT	Pixfont *	Font for PANEL_MENU_TITLE_STRING.
PANEL_MENU_TITLE_IMAGE	Pixrect *	Image for the menu title.
PANEL_MENU_TITLE_STRING	char *	String for the menu title.
PANEL_NEXT_ITEM	Panel_item	Next item in the panel. Get only.
PANEL_NOTIFY_PROC	(procedure)	Function to call when item is selected. Form for button and text items: `notify_proc(item, event)` `Panel_item item` `Event *event` Choice and slider items have an additional parameter for the current value: `notify_proc(item, value, event)` `Panel_item item` `int value` `Event *event` For toggle items, the `value` parameter is of type `unsigned int`.
PANEL_PAINT	Panel_setting	Item's painting behavior for `panel_set()` calls. One of: PANEL_NONE, PANEL_CLEAR, or PANEL_NO_CLEAR.
PANEL_PARENT_PANEL	Panel	The panel which contains the item.
PANEL_SHOW_ITEM	boolean	Whether or not to show the item. Default: TRUE.
PANEL_SHOW_MENU	boolean	If TRUE, the menu for the item is enabled.
PANEL_VALUE_X	int	Left edge of value. If unspecified and label position is fixed, then set to right of PANEL_LABEL_X for horizontal layout, or at PANEL_LABEL_X for vertical layout. Default: after the label.
PANEL_VALUE_Y	int	Top edge of value. If unspecified and label position is fixed, then set to PANEL_LABEL_Y for horizontal layout, or below PANEL_LABEL_Y for vertical layout. Default: PANEL_LABEL_Y.

Table 18-17 *Choice and Toggle Item Attributes— Continued*

Attribute	Value Type	Description
`PANEL_MARK_Y`	int, int	Second argument is top edge of choice mark specified by first argument.
`PANEL_MARK_YS`	list of int	Top edge of each choice mark. Create, set.
`PANEL_MENU_MARK_IMAGE`	Pixrect *	Image to mark each menu choice with when selected.
`PANEL_MENU_NOMARK_IMAGE`	Pixrect *	Image to mark each menu choice with when not selected.
`PANEL_NOMARK_IMAGES`	list of Pixrect *	Image to mark each choice with when not selected. Create, set. Default is push-button image: `<images/panel_choice_on.pr>`.
`PANEL_SHOW_MENU_MARK`	boolean	Show or don't show the menu mark for each selected choice. Default: `TRUE`.
`PANEL_TOGGLE_VALUE`	int, int	Value of a particular toggle choice. Second argument is value of choice specified by first argument.
`PANEL_VALUE`	int or unsigned	If item is a choice, value is ordinal position (from 0) of current choice. If item is a toggle, value is a bitmask indicating currently selected choices (e.g., bit 5 is 1 if 5th choice selected).

Table 18-19 *Text Item Attributes*

Attribute	Value Type	Description
PANEL_MASK_CHAR	char	Character used to mask type-in characters. Use the space character for no character echo (caret does not advance). Use the null character to disable masking.
PANEL_NOTIFY_LEVEL	Panel_setting	When to call the notify function. One of PANEL_NONE, PANEL_NON_PRINTABLE, PANEL_SPECIFIED of PANEL_ALL. Default: PANEL_SPECIFIED (see *Text Notification*).
PANEL_NOTIFY_STRING	char *	String of characters which trigger notification when typed. Applies only when PANEL_NOTIFY_LEVEL is PANEL_SPECIFIED.
PANEL_VALUE_STORED_LENGTH	int	Max number of characters to store in the value string. Default: 80.
PANEL_VALUE_DISPLAY_LENGTH	int	Max number of characters to display in the panel. Default: 80.
PANEL_VALUE	char *	Initial or new string value for the item.
PANEL_VALUE_FONT	Pixfont *	Font to use for the value string.

Table 18-20 *Panel Functions and Macros— Continued*

Definition	Description

`panel_default_handle_event(object, event)`
    <Panel or Panel_item> `object;`
    `Event *event;`
	The default event proc for panel items (`PANEL_EVENT_PROC`) and for the panel's background (`PANEL_BACKGROUND_PROC`). Implements the standard event-to-action mapping for the item types.

`panel_destroy_item(item)`
    `Panel_item item;`
	Destroys `item`.

`panel_each_item(panel, item)`
    `Panel panel;`
    `Panel_item item;`
	Macro to iterate over each item in a panel. The corresponding macro `panel_end_each` closes the loop opened by `panel_each_item()`.

`Event *`
`panel_event(panel, event)`
    `Panel panel;`
    `Event *event;`
	Translates the coordinates of `event` from the space of the panel subwindow to the space of the logical panel (which may be larger and scrollable).

`caddr_t`
`panel_get(item, attribute[, optional_arg])`
    `Panel_item item;`
    `Panel_attribute attribute;`
    `Panel_attribute optional_arg;`
	Retrieve the value of an attribute for `item`. `optional_arg` is used for a few attributes which require additional information, such as `PANEL_CHOICE_IMAGE`, `PANEL_CHOICE_STRING`, `PANEL_CHOICE_X`, `PANEL_CHOICE_Y`, `PANEL_MARK_X`, `PANEL_MARK_Y`, `PANEL_TOGGLE_VALUE`.

`caddr_t`
`panel_get_value(item)`
    `Panel_item item;`
	A macro, defined as:
	`panel_get(item, PANEL_VALUE)`

`panel_paint(panel_object, paint_behavior)`
    <Panel_item or Panel> `panel_object;`
    `Panel_setting paint_behavior;`
	Paints an item or an entire panel. `paint_behavior` can be either `PANEL_CLEAR` or `PANEL_NO_CLEAR`. `PANEL_CLEAR` causes the area occupied by the panel or item to be cleared prior to painting.

`panel_set(item, attributes)`
    `Panel_item item;`
    <attribute-list> `attributes;`
	Sets the value of one or more panel attributes. `attributes` is a null-terminated attribute list.

`panel_set_value(item, value)`
    `Panel_item item;`
    `caddr_t value;`
	A macro, defined as:
	`panel_set(item, PANEL_VALUE, value, 0)`

`Panel_setting`
`panel_text_notify(item, event)`
    `Panel_item item`
    `Event      *event`
	Default notify procedure for panel text items. Causes caret to advance on CR or tab, caret to backup on shift-CR or shift-tab, printable characters to be inserted into `item's` value, and all other characters to be discarded.

Table 18-21 *Pixwin Drawing Functions and Macros*

Definition	Description

`pw_batch(pw, n)`
`    Pixwin        *pw;`
`    Pw_batch_type  n;`
	Tells the batching mechanism to refresh the screen every n display operations.

`pw_batch_off(pw)`
`    Pixwin *pw;`
	A macro to turn batching off in pw.

`pw_batch_on(pw)`
`    Pixwin *pw;`
	A macro to turn batching on in pw.

`pw_batchrop(pw, dx, dy, op, items, n)`
`    Pixwin          *pw;`
`    int              dx, dy, op, n;`
`    struct pr_prpos  items[];`
	See the *Pixrect Reference Manual* for a full explanation of this function.

`pw_char(pw, x, y, op, font, c)`
`    Pixwin  *pw;`
`    int      x, y, op;`
`    Pixfont *font;`
`    char     c;`
	Writes character c into pw using the rasterop op. The left edge and baseline of c will be written at location (x, y).

`pw_close(pw)`
`    Pixwin *pw;`
	Frees any dynamic storage associated with pw, including its retained memory pixrect, if any.

`pw_copy(dpw, dx, dy, dw, dh,`
`        op, spw, sx, sy)`
`    Pixwin *dpw, *spw;`
`    int     op, dx, dy, dw, dh, sx, sy;`
	Copies pixels from spw to dpw. Currently spw and dpw must be the same. This routine will cause problems if spw is obscured.

`int`
`pw_get(pw, x, y)`
`    Pixwin *pw;`
`    int     x, y;`
	Returns the value of the pixel at (x, y) in pw.

`Pixwin *`
`pw_get_region_rect(pw, r)`
`    Pixwin *pw;`
`    Rect *r;`
	Retrieves the rectangle occupied by the region pw into the rect pointed to by r.

Table 18-21 *Pixwin Drawing Functions and Macros— Continued*

Definition	Description

`pw_read(pr, dx, dy, dw, dh, op, pw, sx, sy)`
`    Pixwin  *pw;`
`    int      op, dx, dy, dw, dh, sx, sy;`
`    Pixrect *pr;`
	Reads pixels from the pixwin pw starting at offset (sx, sy), using rasterop op. The pixels are stored in the rectangle (dx, dy, dw, dh) in the pixrect pointed to by pr.

`Pixwin *`
`pw_region(pw, x, y, width, height)`
`    Pixwin *pw;`
`    int     x, y, w, h;`
	Creates a new pixwin refering to an area within the existing pixwin pw. The origin of the new region is given by (x, y), the dimensions by width and height.

`pw_replrop(pw, dx, dy, dw, dh,`
`           op, pr, sx, sy)`
`    Pixwin  *pw;`
`    int      dx, dy, dw, dh, op, sx, sy;`
`    Pixrect *pr;`
	Replicates a pattern from a pixrect into a pixwin.

`pw_reset(pw)`
`    Pixwin *pw;`
	Macro which sets pw's lock count to 0 and releases its lock.

`pw_rop(pw, dx, dy, dw, dh,`
`        op, sp, sx, sy)`
`    Pixwin  *pw;`
`    Pixrect *sp;`
`    int      dx, dy, dw, dh, op, sx, sy;`
	Performs the rasterop op from the source pixrect sp to the destination pixwin pw.

`Pixwin *`
`pw_set_region_rect(pw, r, use_same_pr)`
`    Pixwin       *pw;`
`    Rect         *r;`
`    unsigned int  use_same_pr;`
	The position and size of the region pw are set to the rect *r.
	If use_same_pr is 0 a new retained pixrect is allocated for the region.

`pw_show(pw)`
`    Pixwin *pw;`
	Macro to refresh the screen while batching, without affecting the batching mode.

`pw_stencil(dpw, dx, dy, dw, dh, op,`
`           stpr, stx, sty, spr, sx, sy)`
`    Pixwin  *dpw;`
`    int      dx, dy, dw, dh, op, stx, sty, sx, sy;`
`    Pixrect *stpr, *spr;`
	Like pw_write(), except that the source pixrect spr is written through the stencil pixrect stpr, which functions as a spatial write enable mask. The raster operation op is only applied to destination pixels where the stpr is non-zero; other destination pixels remain unchanged.

`pw_text(pw, x, y, op, font, s)`
`    Pixwin  *pw;`
`    int      x, y, op;`
`    Pixfont *font;`
`    char    *s;`
	Writes the string s into pw using the rasterop op.
	The left edge and baseline of the first character in s will appear at coordinates (x, y).

Table 18-22 *Pixwin Color Manipulation Functions*

Definition	Description
`pw_blackonwhite(pw, min, max)` `Pixwin *pw;` `int min, max;`	Sets the foreground to black, the background to white, for pixwin `pw`. `min` and `max` should be the first and last entries, respectively, in `pw`'s colormap segment.
`pw_cyclecolormap(pw, cycles, index, count)` `Pixwin *pw;` `int cycles, index, count;`	Rotates the portion of `pw`'s colormap segment starting at `index` for `count` entries, rotating those entries among themselves `cycles` times.
`pw_getattributes(pw, planes)` `Pixwin *pw;` `int *planes;`	Retrieves the value of pw's access enable mask into the integer addressed by `planes`.
`pw_getcmsname(pw, cmsname)` `Pixwin *pw;` `char cmsname[CMS_NAMESIZE];`	Copies the colormap segment name of `pw` into `cmsname`.
`pw_getcolormap(pw, index, count, red, green, blue)` `Pixwin *pw;` `int index, count;` `unsigned char red[], green[], blue[];`	Retrieves the state of pw's colormap. The `count` elements of the pixwin's colormap segment starting at `index` (0 origin) are loaded into the first `count` values in the three arrays.
`pw_getdefaultcms(cms, map)` `struct colormapseg *cms;` `struct cms_map *map;`	Copies the data in the default colormap segment into the data pointed to by `cms` and `map`. Before the call, the byte pointers in `map` should be initialized to arrays of size 256.
`pw_putattributes(pw, planes)` `Pixwin *pw;` `int *planes;`	Sets the access enable mask of `pw`. Only those bits of the pixel corresponding to a 1 in the same bit position of `*planes` will be affected by pixwin operations.
`pw_putcolormap(pw, index, count, red, green, blue)` `Pixwin *pw;` `int index, count;` `unsigned char red[], green[], blue[];`	Sets the state of `pw`'s colormap. The `count` elements of the pixwin's colormap segment starting at `index` (0 origin) are loaded from the first `count` values in the three arrays.
`pw_reversevideo(pw, min, max)` `Pixwin *pw;` `int min, max;`	Reverses the foreground and background colors of `pw` `min` and `max` should be the first and last entries, respectively, in the colormap segment.

Table 18-23 *Scrollbar Attributes*

Attribute	Value Type	Description
SCROLL_ABSOLUTE_CURSOR	Cursor	Cursor to display on middle button down. Default: Right triangle if vert., down triangle if horiz.
SCROLL_ACTIVE_CURSOR	Cursor	Cursor to display when cursor is in bar rect. Default: Right arrow if vertical, down arrow if horiz.
SCROLL_ADVANCED_MODE	boolean	Whether notify proc reports all nine motions. Default: FALSE.
SCROLL_BACKWARD_CURSOR	Cursor	Cursor to display on right button down. Default: up arrow if vertical, left arrow if horiz.
SCROLL_BAR_COLOR	Scrollbar_setting	Color of bar, SCROLL_GREY (default) or SCROLL_WHITE.
SCROLL_BAR_DISPLAY_LEVEL	Scrollbar_setting	When bar is displayed. SCROLL_ALWAYS: always displayed SCROLL_ACTIVE: only displayed when cursor is in bar rect SCROLL_NEVER: never displayed Default: SCROLL_ALWAYS.
SCROLL_BORDER	boolean	Whether the scrollbar has a border.
SCROLL_BUBBLE_COLOR	Scrollbar_setting	Color of bubble, SCROLL_GREY (default) or SCROLL_BLACK.
SCROLL_BUBBLE_DISPLAY_LEVEL	Scrollbar_setting	When bubble is displayed. SCROLL_ALWAYS: always displayed SCROLL_ACTIVE: only displayed when cursor is in bar rect SCROLL_NEVER: never displayed Default: SCROLL_ALWAYS.
SCROLL_BUBBLE_MARGIN	int	Margin on each side of bubble in bar. Default: 0.
SCROLL_DIRECTION	Scrollbar_setting	Orientation of bar, SCROLL_VERTICAL (default) or SCROLL_HORIZONTAL.
SCROLL_END_POINT_AREA	int	The distance, in pixels, from the end of the scrollbar that forces a scroll to the beginning (or end) of the file. Default: 6.
SCROLL_FORWARD_CURSOR	Cursor	Cursor to display on left button down. Default: down arrow if vertical, right arrow if horiz.
SCROLL_GAP	int	Gap between lines. Default: current value of SCROLL_MARGIN.
SCROLL_HEIGHT	int	r_height for scrollbar's rect.

Table 18-23 *Scrollbar Attributes— Continued*

Attribute	Value Type	Description
SCROLL_REQUEST_MOTION	Scroll_motion	Scrolling motion requested by user.
SCROLL_REQUEST_OFFSET	int	Pixel offset of scrolling request into scrollbar. Default: 0.
SCROLL_THICKNESS	int	Thickness of bar. Default: 14.
SCROLL_TO_GRID	boolean	Whether the client wants scrolling aligned to multiples of SCROLL_LINE_HEIGHT. Default: FALSE.
SCROLL_TOP	int	r_top for scrollbar's rect.
SCROLL_VIEW_LENGTH	int	Length of viewing window, in client units. Default: 0.
SCROLL_VIEW_START	int	Current offset into scrollable object (client units). (Value must be > 0). Default: 0.
SCROLL_WIDTH	int	r_width for scrollbar's rect.

Table 18-25 *Text Subwindow Attributes*

Attribute	Value Type	Description
TEXTSW_ADJUST_IS_PENDING_DELETE	boolean	When TRUE, adjusting a selection causes the selection to be pending-delete. Default: FALSE.
TEXTSW_AGAIN_RECORDING	boolean	When FALSE, changes made to the textsw are not repeated when user invokes AGAIN. By disabling when not needed (e.g. for program- driven error logs) you can reduce memory overhead. Default: TRUE.
TEXTSW_AUTO_INDENT	boolean	When TRUE, a new line is automatically indented to match the previous line. Default: FALSE.
TEXTSW_AUTO_SCROLL_BY	int	Number of lines to scroll when type-in moves insert point below the view. Default: 1. Create, get.
TEXTSW_BLINK_CARET	boolean	Determines whether the caret blinks. Default: FALSE.
TEXTSW_BROWSING	boolean	When TRUE, prevents editing of the displayed text. If another file is loaded in, browsing stays on. Default: FALSE.
TEXTSW_CHECKPOINT_FREQUENCY	int	Number of edits between checkpoints. Set to 0 to disable checkpointing. Default: 0.
TEXTSW_CLIENT_DATA	char *	Pointer to arbitrary client data. Default: NULL.
TEXTSW_CONFIRM_OVERWRITE	boolean	A request to write to an existing file will require user confirmation. Default: TRUE.
TEXTSW_CONTENTS	char *	Contents of text subwindow. Default: NULL. Create, get. For create, specifies the initial contents for non-file textsw. Get needs additional parameters: window_get(textsw, TEXTSW_CONTENTS, pos, buf, buf_len) Return value is next position to read at. buf[0..buf_len-1] is filled with the characters from textsw beginning at index pos, and is null-terminated only if there were too few characters to fill the buffer.
TEXTSW_CONTROL_CHARS_USE_FONT	boolean	If FALSE, control characters always display as an up arrow followed by a character, instead of whatever glyph is in the current font. Default: FALSE.
TEXTSW_DISABLE_CD	boolean	Stops textsw from changing current working directory (and grays out the associated items in the menu). Default: FALSE.

Table 18-25 *Text Subwindow Attributes— Continued*

Attribute	Value Type	Description
TEXTSW_MODIFIED	boolean	Whether or not the textsw has been modified. Get only.
TEXTSW_MULTI_CLICK_SPACE	int	Max # of pixels that can be between successive mouse clicks and still have the clicks be considered a multi-click. Default: 3.
TEXTSW_MULTI_CLICK_TIMEOUT	int	Max # of milliseconds that can be between successive mouse clicks and still have the clicks be considered a multi-click. Default: 390.
TEXTSW_NOTIFY_PROC	(procedure)	Notify procedure. Form is: `void` `notify_proc(textsw, avlist)` `Textsw textsw` `Attr_avlist avlist` Default: NULL, meaning standard procedure.
TEXTSW_READ_ONLY	boolean	When TRUE, prevents editing of the displayed text. If another file is loaded in, READ_ONLY is turned off again. Default: FALSE.
TEXTSW_SCROLLBAR	Scrollbar	Scrollbar to use for text subwindow scrolling. NULL means no scrollbar. Default: A scrollbar with default attributes. Note: text subwindow has a scrollbar by default, so you would only use this to get no scrollbar, or to get the scrollbar handle.
TEXTSW_STATUS	Textsw_status *	If set, specifies the address of a variable of type Textsw_status into which a value is written that reflects what happened during the call to window_create(). (For possible values, see the *Textsw_status Values* table).
TEXTSW_STORE_CHANGES_FILE	boolean	If TRUE, Store changes the file being edited to that named as the target of the Store. If FALSE, Store does not affect which file is being edited. Default: TRUE.
TEXTSW_STORE_SELF_IS_SAVE	boolean	Causes textsw to interpret a Store to the name of the current file as a Save. Default: FALSE. Create, get.
TEXTSW_UPDATE_SCROLLBAR	(no value)	Causes text subwindow to update the bubble in the scrollbar. Set only — get returns NULL.
TEXTSW_UPPER_CONTEXT	int	Min # of lines to maintain between the start of the selection and top of view. -1 means to defeat the normal actions. Default: 2.

Table 18-27 `Textsw_status` *Values*

Value	Description
`TEXTSW_STATUS_OKAY`	The operation encountered no problems.
`TEXTSW_STATUS_BAD_ATTR`	The attribute list contained an illegal or unrecognized attribute.
`TEXTSW_STATUS_BAD_ATTR_VALUE`	The attribute list contained an illegal value for an attribute, usually an out of range value for an enumeration.
`TEXTSW_STATUS_CANNOT_ALLOCATE`	A call to calloc(2) or malloc(2) failed.
`TEXTSW_STATUS_CANNOT_OPEN_INPUT`	The specified input file does not exist or cannot be accessed.
`TEXTSW_STATUS_OTHER_ERROR`	The operation encountered a problem not covered by any of the other error indications.

Table 18-28 *Text Subwindow Functions— Continued*

Definition	Description

`Textsw_index`
`textsw_find_mark(textsw, mark)`
`    Textsw      textsw;`
`    Textsw_mark mark;`
	Returns the current position of `mark`.

`Textsw`
`textsw_first(textsw)`
`    Textsw textsw;`
	Returns the first view into `textsw`.

`Textsw_index`
`textsw_index_for_file_line(textsw, line)`
`    Textsw textsw;`
`    int    line;`
	Returns the character index for the first character in the line given by `line`.

`int`
`textsw_insert(textsw, buf, buf_len)`
`    Textsw  textsw;`
`    char    *buf;`
`    int     buf_len;`
	Inserts characters in `buf` into `textsw` at the current insertion point. The number of characters actually inserted is returned — this will equal `buf_len` unless there was a memory allocation failure.

`Textsw`
`textsw_next(textsw)`
`    Textsw textsw;`
	Returns the next view in the set of views into `textsw`.

`void`
`textsw_normalize_view(textsw, position)`
`    Textsw       textsw;`
`    Textsw_index position;`
	Repositions the text so that the character at `position` is visible and at the top of the subwindow.

`void`
`textsw_possibly_normalize(textsw, position)`
`    Textsw       textsw;`
`    Textsw_index position;`
	If the character at `position` is already visible, this function does nothing. If it is not visible, it repositions the text so that it is visible and at the top of the subwindow.

`void`
`textsw_remove_mark(textsw, mark)`
`    Textsw      textsw;`
`    Textsw_mark mark;`
	Removes an existing mark from `textsw`.

Table 18-29 *TTY Subwindow Attributes*

Attribute	Type	Description
`TTY_ARGV`	char **	Argument vector: name of the program running in the tty subwindow, followed by arguments for that program.
`TTY_CONSOLE`	boolean	If `TRUE`, tty subwindow is console. Set only. Default: `FALSE`.
`TTY_PAGE_MODE`	boolean	If `TRUE`, output will stop after each page. Default: `FALSE`.
`TTY_QUIT_ON_CHILD_DEATH`	boolean	If `TRUE`, subwindow quits when its child terminates. Set only. Default: `FALSE`.

Table 18-30 *TTY Subwindow Functions*

Definition	Description

`int`
`ttysw_input(tty, buf, len)`
`    Tty tty;`
`    char *buf;`
`    int len;`
	Appends `len` number of characters from `buf` onto `tty`'s *input* queue. It returns the number of characters accepted.

`int`
`ttysw_output(tty, buf, len)`
`    Tty tty;`
`    char *buf;`
`    int len;`
	Appends `len` number of characters from `buf` onto `tty`'s *output* queue, i.e. they are sent through the terminal emulator to the TTY. It returns the number of characters accepted.

Table 18-32 *Window Attributes*

Attribute	Value Type	Description
`WIN_BELOW`	Window	Causes the window to be laid out below window given as the value.
`WIN_BOTTOM_MARGIN`	int	Margin at bottom of window.
`WIN_CLIENT_DATA`	caddr_t	Client's private data — for your use.
`WIN_COLUMNS`	int	Window's width (including left and right margins) in columns.
`WIN_COLUMN_GAP`	int	Gap between columns in the window.
`WIN_COLUMN_WIDTH`	int	Width of a column in the window.
`WIN_CONSUME_KBD_EVENT`	short	Window will receive this event.
`WIN_CONSUME_KBD_EVENTS`	list of short	Null terminated list of events window will receive. Create, set.
`WIN_CONSUME_PICK_EVENT`	short	Window will receive this pick event.
`WIN_CONSUME_PICK_EVENTS`	list of short	Null terminated list of pick events window will receive. Create, set.
`WIN_CURSOR`	Cursor	The window's cursor. Note: the pointer returned by `window_get()` points to per-process static storage.
`WIN_DEVICE_NAME`	char *	UNIX device name associated with window, consisting of a string and numeric part, e.g. `win10`. Get only.
`WIN_DEVICE_NUMBER`	int	Numeric component of device name. Get only.
`WIN_ERROR_MSG`	char *	Error message to print before exit(1). Create only.
`WIN_EVENT_PROC`	(procedure)	Client's callback procedure which receives input events: `Notify_value` `event_proc(win, event, arg)` `Window window;` `Event *event;` `caddr_t arg;` Note: In current release does not work for frames.
`WIN_EVENT_STATE`	short	Gets the state of the specified event code. For buttons and keys, zero means "up", non-zero means "down". Get only.
`WIN_FD`	int	The UNIX file descriptor for the window. Get only.

Table 18-32 *Window Attributes— Continued*

Attribute	Value Type	Description
WIN_PERCENT_HEIGHT	int	Sets a subwindow's height as a percentage of the frame's height.
WIN_PERCENT_WIDTH	int	Sets a subwindow's width as a percentage of the frame's width.
WIN_PICK_INPUT_MASK	Inputmask *	Window's pick inputmask. **Note:** the pointer returned by `window_get()` points to per-process static storage.
WIN_PIXWIN	Pixwin *	The window's pixwin. Get only.
WIN_RECT	Rect *	Rect of the window. For frames, same as FRAME_OPEN_RECT. **Note:** the pointer returned by `window_get()` for this attribute points to per-process static storage.
WIN_RIGHT_MARGIN	int	Margin at right of window.
WIN_RIGHT_OF	Window	Causes the window to be laid out just to the right of the window given as the value.
WIN_ROW_GAP	int	Gap between rows in the window.
WIN_ROW_HEIGHT	int	Height of a row in the window.
WIN_ROWS	int	Window's height (including top and bottom margins) in rows.
WIN_SCREEN_RECT	Rect *	Rect of the screen containing the window. Get only. **Note:** the pointer returned by `window_get()` for this attribute points to per-process static storage.
WIN_SHOW	boolean	Causes the window to be displayed or undisplayed.
WIN_TOP_MARGIN	int	Margin at top of window.
WIN_TYPE	Window_type	Type of window. One of FRAME_TYPE, PANEL_TYPE, CANVAS_TYPE, TEXTSW_TYPE or TTY_TYPE. Get only.
WIN_VERTICAL_SCROLLBAR	Scrollbar	Vertical scrollbar.
WIN_WIDTH	int	Window's width in pixels. Value of WIN_EXTEND_TO_EDGE causes subwindow to extend to right edge of frame. Default: WIN_EXTEND_TO_EDGE.
WIN_X	int	x position of window, relative to owner.
WIN_Y	int	y position of window, relative to owner.

Table 18-33 *Frame Attributes— Continued*

Attribute	Value Type	Description
FRAME_NTH_SUBWINDOW	int	Returns frame's nth (from 0) subwindow. Get only.
FRAME_NTH_WINDOW	int	Returns frame's nth (from 0) window, regardless of whether the window is a frame or a subwindow. Get only.
FRAME_OPEN_RECT	Rect *	Frame's rect when open.
FRAME_SHOW_LABEL	boolean	Whether the label is shown. Default: TRUE for base frames, FALSE for subframes.
FRAME_SUBWINDOWS_ADJUSTABLE	boolean	User can move subwindow boundaries. Default: TRUE.

Table 18-34 *Window Functions and Macros— Continued*

Definition	Description

`int`
`window_read_event(window, event)`
`    Window window;`
`    Event *event;`
	Reads the next input event for `window`. In case of error, sets the global variable `errno` and returns -1.

`void`
`window_refuse_kbd_focus(window)`
`    Window window;`
	When your event handler receives a `KBD_REQUEST` event, call this function if you do not want your window to become the keyboard focus.

`void`
`window_release_event_lock(window)`
`    Window window;`
	Releases the event lock, allowing other processes to receive input.

`void`
`window_return(value)`
`    caddr_t value;`
	Usually called from one of the application's panel item notify procs. Causes `window_loop()` to return.

`window_set(win, attributes)`
`    Window win;`
`    `*<attribute-list>* `attributes;`
	Sets the value of one or more of `win`'s attributes. `attributes` is a null-terminated attribute list.

A

Example Programs

Example Programs **341**

A.1. *filer* 341

A.2. *image_browser_1* 350

A.3. *image_browser_2* 354

A.4. *tty_io* 359

A.5. *font_menu* 362

A.6. *resize_demo* 370

A.7. *dctool* 375

A.8. *typein* 381

A.9. Programs that Manipulate Color 385

coloredit 385

animatecolor 390

A.10. Two gfx subwindow–based programs converted to use SunView 397

bounce 397

spheres 402

A

Example Programs

A.1. *filer*

This program is discussed in Chapter 4, *Using Windows*. It displays a listing in a tty subwindow, which is manipulated by panel items.

One of the panel buttons makes a popup window appear. Another uses the selection service to determine what file name the user has selected, and creates a popup text subwindow where that file is displayed.

```
create_panel_subwindow()
{
    void ls_proc(), ls_flags_proc(), quit_proc(), edit_proc(), del_proc();

    char current_dir[MAX_PATH_LEN];

    panel = window_create(base_frame, PANEL, 0);

    dir_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,              ATTR_COL(0),
        PANEL_LABEL_Y,              ATTR_ROW(0),
        PANEL_VALUE_DISPLAY_LENGTH, 22,
        PANEL_VALUE,                getwd(current_dir),
        PANEL_LABEL_STRING,         "Directory: ",
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel,"List",0,0),
        PANEL_NOTIFY_PROC, ls_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel,"Set ls flags",0,0),
        PANEL_NOTIFY_PROC, ls_flags_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel,"Edit",0,0),
        PANEL_NOTIFY_PROC, edit_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel,"Delete",0,0),
        PANEL_NOTIFY_PROC, del_proc,
        0);

    (void) panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel,"Quit",0,0),
        PANEL_NOTIFY_PROC, quit_proc,
        0);

    fname_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,              ATTR_COL(0),
        PANEL_LABEL_Y,              ATTR_ROW(1),
        PANEL_VALUE_DISPLAY_LENGTH, 22,
        PANEL_LABEL_STRING,         "File:      ",
        0);

    msg_item = panel_create_item(panel, PANEL_MESSAGE, 0);

    window_fit_height(panel);

    window_set(panel, PANEL_CARET_ITEM, fname_item, 0);
}
```

```
    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                   PANEL_ITEM_X,         ATTR_COL(0),
                   PANEL_ITEM_Y,         ATTR_ROW(6),
                   PANEL_DISPLAY_LEVEL,  PANEL_CURRENT,
                   PANEL_LABEL_STRING,   "List '.' files?                 ",
                   PANEL_CHOICE_STRINGS, "No", "Yes", 0,
                   PANEL_CLIENT_DATA,    " a ",
                   0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
                   PANEL_ITEM_X,         ATTR_COL(0),
                   PANEL_ITEM_Y,         ATTR_ROW(7),
                   PANEL_DISPLAY_LEVEL,  PANEL_CURRENT,
                   PANEL_LABEL_STRING,   "Indicate type of file?          ",
                   PANEL_CHOICE_STRINGS, "No", "Yes", 0,
                   PANEL_CLIENT_DATA,    " F ",
                   0);

    window_fit(ls_flags_panel);  /* fit panel around its items */
    window_fit(ls_flags_frame);  /* fit frame around the panel */
}

char *
compose_ls_options()
{
   static char flags[20];
   char *ptr;
   char flag;
   int first_flag = TRUE;
   Panel_item item;
   char *client_data;
   int index;

   ptr = flags;

   panel_each_item(ls_flags_panel, item)
       client_data = panel_get(item, PANEL_CLIENT_DATA, 0);
       index = (int)panel_get_value(item);
       flag = client_data[index];
       if (flag != ' ') {
           if (first_flag) {
              *ptr++     = '-';
              first_flag = FALSE;
           }
           *ptr++ = flag;
       }
   panel_end_each
   *ptr = '\0';
   return flags;
}

void
ls_proc()
{
   static char previous_dir[MAX_PATH_LEN];
   char *current_dir;
   char cmdstring[100];          /* dir_item's value can be 80, plus flags */
```

```
    /* return if file not found */
    if (!stat_file(filename))
        return;

    msg(""); /* clear any old messages */

    window_set(editsw, TEXTSW_FILE, filename, 0);

    window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
}

void
del_proc()
{
    char buf[300];
    char *filename;

    /* return if no selection */
    if (!strlen(filename = get_selection())) {
        msg("Please select a file to delete.");
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    msg(""); /* clear any old messages */

    /* usr must confirm the delete */
    sprintf(buf, "Ok to delete %s?", filename);
    if (confirm_yes(buf)) {
        unlink(filename);
        sprintf(buf, "%s deleted.", filename);
        msg(buf);
    }
}

void
quit_proc()
{
    window_destroy(base_frame);
}

msg(msg)
char *msg;
{
    panel_set(msg_item, PANEL_LABEL_STRING, msg, 0);
}
```

```
        pr = panel_button_image(panel, "NO", 3, 0);
        width = 2 * pr->pr_width + 10;
    }

    /* center the yes/no or ok buttons under the message */
    r = (Rect *) panel_get(message_item, PANEL_ITEM_RECT);
    left = (r->r_width - width) / 2;
    if (left < 0)
        left = 0;
    top = rect_bottom(r) + 5;

    if (ok_only) {
        panel_create_item(panel, PANEL_BUTTON,
            PANEL_ITEM_X, left, PANEL_ITEM_Y, top,
            PANEL_LABEL_IMAGE, pr,
            PANEL_CLIENT_DATA, 1,
            PANEL_NOTIFY_PROC, yes_no_ok,
        0);
    } else {
        panel_create_item(panel, PANEL_BUTTON,
            PANEL_ITEM_X, left, PANEL_ITEM_Y, top,
            PANEL_LABEL_IMAGE, pr,
            PANEL_CLIENT_DATA, 0,
            PANEL_NOTIFY_PROC, yes_no_ok,
            0);
        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_IMAGE, panel_button_image(panel, "YES", 3, 0),
            PANEL_CLIENT_DATA, 1,
            PANEL_NOTIFY_PROC, yes_no_ok,
            0);
    }

    window_fit(panel);
    window_fit(confirmer);

    /* center the confirmer frame on the screen */
    r      = (Rect *) window_get(confirmer, WIN_SCREEN_RECT);
    width  = (int) window_get(confirmer, WIN_WIDTH);
    height = (int) window_get(confirmer, WIN_HEIGHT);
    left   = (r->r_width - width) / 2;
    top    = (r->r_height - height) / 2;
    if (left < 0)
        left = 0;
    if (top < 0)
        top = 0;
    window_set(confirmer, WIN_X, left, WIN_Y, top, 0);

    return confirmer;
}

static void
yes_no_ok(item, event)
Panel_item      item;
Event           *event;
{
    window_return(panel_get(item, PANEL_CLIENT_DATA));
}
```

```
/*****************************************************************************/
/*                              image_browser_1.c                            */
/*****************************************************************************/

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

Frame frame;
Panel control_panel, display_panel;
Tty   tty;

Panel_item dir_item, fname_item, image_item;

ls_proc(), show_proc(), quit_proc();

char *get_selection();

#define MAX_PATH_LEN 1024
#define MAX_FILENAME_LEN 256

main(argc, argv)
int argc;
char **argv;
{
    frame = window_create(NULL, FRAME,
                          FRAME_ARGS,  argc, argv,
                          FRAME_LABEL, "image_browser",
                          0);
    init_tty();
    init_control_panel();
    window_fit(frame);
    init_display_panel();
    window_main_loop(frame);
    exit(0);
}

init_tty()
{
    tty = window_create(frame, TTY, WIN_COLUMNS, 30, WIN_ROWS, 20, 0);
}

init_display_panel()
{
    display_panel = window_create(frame, PANEL,
                          WIN_BELOW,    control_panel,
                          WIN_RIGHT_OF, tty,
                          0);
    image_item = panel_create_item(display_panel, PANEL_MESSAGE, 0);
}
```

```
ls_proc()
{
   static char previous_dir[MAX_PATH_LEN];
   char *current_dir;
   char cmdstring[100];

   current_dir = (char *)panel_get_value(dir_item);

   if (strcmp(current_dir, previous_dir)) {
       chdir(current_dir);
       sprintf(cmdstring, "cd %s0, current_dir);
       ttysw_input(tty, cmdstring, strlen(cmdstring));
       strcpy(previous_dir, current_dir);
   }

   sprintf(cmdstring, "ls -1 %s\n", panel_get_value(fname_item));
   ttysw_input(tty, cmdstring, strlen(cmdstring));
}

quit_proc()
{
   window_destroy(frame);
}

show_proc()
{
    char *filename;

    if (!strlen(filename = get_selection()))
        return;

    load_image(filename);
}

load_image(filename)
char *filename;
{
    Pixrect *image;
    char error_msg[IL_ERRORMSG_SIZE];

    if (image = icon_load_mpr(filename, error_msg)) {
        panel_set(image_item,
                  PANEL_ITEM_X,      ATTR_COL(5),
                  PANEL_ITEM_Y,      ATTR_ROW(4),
                  PANEL_LABEL_IMAGE, image,
                  0);
    }
}
```

```
/**************************************************************************/
/*                           image_browser_2.c                           */
/**************************************************************************/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/tty.h>
#include <stdio.h>
#include <suntool/icon_load.h>
#include <suntool/seln.h>
#include <suntool/expand_name.h>
#include <suntool/scrollbar.h>

static char namebuf[100];
static int file_count, image_count;
static struct namelist *name_list;
#define get_name(i) name_list->names[(i)]

Frame      frame;
Panel      control_panel, display_panel;
Tty        tty;
Panel_item dir_item, fname_item, image_item;

show_proc(), browse_proc(), quit_proc();
Pixrect *get_image();
char *get_selection();

#define MAX_PATH_LEN 1024
#define MAX_FILENAME_LEN 256

main(argc, argv)
int argc;
char **argv;
{
    frame = window_create(NULL, FRAME,
                          FRAME_ARGS, argc, argv,
                          FRAME_LABEL, "image_browser_2",
                          0);
    init_control_panel();
    init_display_panel();
    window_set(control_panel,
               WIN_WIDTH, window_get(display_panel, WIN_WIDTH, 0),
               0);
    window_fit(frame);
    window_main_loop(frame);
    exit(0);
}

init_control_panel()
{
    char current_dir[MAX_PATH_LEN];

    control_panel = window_create(frame, PANEL, 0);

    dir_item = panel_create_item(control_panel, PANEL_TEXT,
        PANEL_LABEL_X,              ATTR_COL(0),
        PANEL_LABEL_Y,              ATTR_ROW(0),
        PANEL_VALUE_DISPLAY_LENGTH, 23,
set_directory()
{
   static char previous_dir[MAX_PATH_LEN];
   char *current_dir;

   current_dir = (char *)panel_get_value(dir_item);

   if (strcmp(current_dir, previous_dir)) {
       chdir(current_dir);
       strcpy(previous_dir, current_dir);
   }
}

Pixrect *
get_image(i)
int i;
{
   char error_msg[IL_ERRORMSG_SIZE];
   return (icon_load_mpr(get_name(i), error_msg));
}

match_files()
{
   char *val;

   val = (char *)panel_get_value(fname_item);
   strcpy(namebuf, val);
   name_list  = expand_name(namebuf);
   file_count = name_list->count;
}

load_image(filename)
char *filename;
{
    Pixrect *image;
    char error_msg[IL_ERRORMSG_SIZE];

    if (image = icon_load_mpr(filename, error_msg)) {
        panel_set(image_item,
                  PANEL_ITEM_X,      ATTR_COL(5),
                  PANEL_ITEM_Y,      ATTR_ROW(4),
                  PANEL_LABEL_IMAGE, image,
                  0);
    }
}
```

A.4. *tty_io*

The following program demonstrates the use of `ttysw_input()`, `ttysw_output()` and TTY escape sequences. These functions are explained in Chapter 10, *TTY Subwindows*.

tty_io creates a panel and a tty subwindow. You can send arbitrary character sequences to the latter as input or output by manipulating panel items. There is also a button that sends the current time within the escape sequence to set the frame label. Try sending different sequences to the tty subwindow. Press CTRL-R to see the difference between what appears on the screen and what was input to the pseudo-tty. Also try starting the tool with a program such as *vi* as a command line argument.

```
    /* Assume rest of arguments are for tty subwindow, except FRAME_ARGS leaves the
     * program_name as argv[0], and we don't want to pass this to the tty subwindow.
     */
    argv++;
    tty = window_create(frame, TTY, TTY_ARGV, argv, 0);

    window_fit(frame);

    ttysw_input(tty, "echo my pseudo-tty is `tty`\n", 28);

    window_main_loop(frame);
    exit(0);
}

static void
input_text(item, event)
    Panel_item      item;
    Event          *event;
{
    strcpy(tmp_buf, (char *) panel_get_value(text_item));
    ttysw_input(tty, tmp_buf, strlen(tmp_buf));
}

static void
output_text(item, event)
    Panel_item      item;
    Event          *event;
{
    strcpy(tmp_buf, (char *) panel_get_value(text_item));
    ttysw_output(tty, tmp_buf, strlen(tmp_buf));
}

static void
output_time(item, event)
    Panel_item      item;
    Event          *event;
{
#include <sys/time.h>
#define ASCTIMELEN      26

    struct timeval  tp;

    /* construct escape sequence to set frame label */
    tmp_buf[0] = '\033';
    tmp_buf[1] = ']';
    tmp_buf[2] = 'l';
    tmp_buf[2 + ASCTIMELEN + 1] = '\033';
    tmp_buf[2 + ASCTIMELEN + 2] = '\\';
    gettimeofday(&tp, NULL);
    strncpy(&tmp_buf[3], ctime(&tp.tv_sec), ASCTIMELEN);
    ttysw_output(tty, tmp_buf, ASCTIMELEN + 5);
}
```

```
/*****************************************************************************/
/*                   font_menu.c                                             */
/*****************************************************************************/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/walkmenu.h>

void set_family(), set_size(), set_on_off(), toggle_on_off(), open_fonts();
Menu new_menu(), initialize_on_off();
char *int_to_str();
extern char * sprintf();
extern char * malloc();

Panel_item feedback_item;
char *family, *size, *bold, *italic;
Pixfont *cour, *serif, *apl, *cmr, *screen;

/*****************************************************************************/
/* main                                                                      */
/* First create the base frame, the feedback panel and feedback item. The    */
/* feedback item is initialized to "gallant 8".                              */
/* Then get the frame's menu, call new_menu() to create a new menu with the  */
/* original frame menu as a pullright, and give the new menu to the frame.   */
/*****************************************************************************/

main(argc, argv)
        int argc;
        char *argv[];
{
    Frame frame;
    Panel panel;
    Menu  menu;
    int   defaults;

    frame = window_create(NULL, FRAME, FRAME_LABEL, "Menu Test -- Try frame menu.", 0);
    panel = window_create(frame, PANEL, WIN_ROWS, 1, 0);
    feedback_item = panel_create_item(panel, PANEL_MESSAGE, PANEL_LABEL_STRING, "", 0);

   family = "Gallant", size = "8", bold = italic = "";
   update_feedback();

    /* remember if user gave -d flag */
    if (argc >= 2) defaults = strcmp(argv[1], "-d") == 0;

    menu = (Menu)window_get(frame, WIN_MENU);
    menu = new_menu(menu, defaults);
    window_set(frame, WIN_MENU, menu, 0);

    window_main_loop(frame);
}
```

```
                    MENU_ITEM,
                       MENU_STRING, "Screen",
                       MENU_FONT,   screen,
                       0,
                    MENU_NOTIFY_PROC, set_family,
                  0),
                MENU_PULLRIGHT_ITEM,
                  "Size", size_menu = menu_create(0),
                MENU_ITEM,
                  MENU_STRING,      "Bold",
                  MENU_PULLRIGHT,   on_off_menu,
                  MENU_NOTIFY_PROC, toggle_on_off,
                  MENU_CLIENT_DATA, &bold,
                  0,
                MENU_ITEM,
                  MENU_STRING,      "Italic",
                  MENU_PULLRIGHT,   on_off_menu,
                  MENU_NOTIFY_PROC, toggle_on_off,
                  MENU_CLIENT_DATA, &italic,
                  0,
                0);

/* give each item in the family menu the size menu as a pullright */
for (i = (int)menu_get(family_menu, MENU_NITEMS); i > 0; --i)
    menu_set(menu_get(family_menu, MENU_NTH_ITEM, i),
             MENU_PULLRIGHT, size_menu, 0);

/* put non-selectable lines inbetween groups of items in family menu */
menu_set(family_menu,
         MENU_INSERT, 2, menu_create_item(MENU_STRING,     "-------",
                                          MENU_INACTIVE,   TRUE,
                                          0),
         0);
menu_set(family_menu,
         MENU_INSERT, 5, menu_get(family_menu, MENU_NTH_ITEM, 3),
         0);

/* The size menu was created with no items.  Now give it items representing */
/*  the point sizes 8, 10, 12, 14, 16, and 18.                             */
for (i = 8; i <= 18; i += 2)
    menu_set(size_menu, MENU_STRING_ITEM, int_to_str(i), i, 0);

/* give the size menu a notify proc to update the feedback */
menu_set(size_menu, MENU_NOTIFY_PROC, set_size, 0);
```

```
/******************************************************************************/
/* set_size -- notify proc for the size menu.                                 */
/******************************************************************************/

/*ARGSUSED*/
void
set_size(m, mi)
        Menu m;
        Menu_item mi;
{
    size =  menu_get(mi, MENU_STRING);
    update_feedback();
}

/*****************************************************************************/
/* initialize_on_off -- generate proc for the on_off menu.                    */
/* The on-off menu is a pullright of both the bold and the italic menus.      */
/* We want it to toggle -- if its parent was on, it should come up with       */
/* "Off" selected, and vice-versa.  We can do that by first getting the       */
/* parent menu item, then, indirectly through its client data attribute,      */
/* seeing if the string representing the bold or italic state is null.       */
/* If the string was null, we set the first item ("On") to be selected,       */
/* else we set the second item ("Off") to be selected.                        */
/******************************************************************************/

Menu
initialize_on_off(m, op)
        Menu m; Menu_generate op;
{
    Menu_item parent_mi;
    char **name;

    if (op != MENU_CREATE) return (m);

    parent_mi = (Menu_item)menu_get(m, MENU_PARENT);
    name = (char **)menu_get(parent_mi, MENU_CLIENT_DATA);

    if (**name == NULL)
        menu_set(m, MENU_SELECTED, 1, 0);
    else
        menu_set(m, MENU_SELECTED, 2, 0);
    return (m);
}
```

```
update_feedback()
{
    char buf[30];

    sprintf(buf, "%s %s %s %s", bold, italic, family, size);
    panel_set(feedback_item, PANEL_LABEL_STRING, buf, 0);
}

char *
int_to_str(n)
{
    char *r = malloc(4);
    sprintf(r, "%d", n);
    return (r);
}

void
open_fonts()
{
    cour = pf_open("/usr/lib/fonts/fixedwidthfonts/cour.r.10");
    serif = pf_open("/usr/lib/fonts/fixedwidthfonts/serif.r.10");
    apl = pf_open("/usr/lib/fonts/fixedwidthfonts/apl.r.10");
    cmr = pf_open("/usr/lib/fonts/fixedwidthfonts/cmr.b.8");
    screen = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.11");
}
```

```
/****************************************************************************/
/*                          resize_demo.c                                   */
/****************************************************************************/

#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/scrollbar.h>

Canvas Canvas_1, Canvas_2, Canvas_3, Canvas_4;
Pixwin *Pixwin_1, *Pixwin_2, *Pixwin_3, *Pixwin_4;
Rect framerect;
PIXFONT *font;

extern char * sprintf();
/*
 * font macros:
 *      font_offset(font) gives the vertical distance between
 *                        the font origin and the top left corner
 *                        of the bounding box of the string displayed
 *                        (see Text Facilities for Pixrects in the
 *                        Pixrect Reference Manual)
 *      font_height(font) gives the height of the font
 */

#define font_offset(font)       (-font->pf_char['n'].pc_home.y)
#define font_height(font)       (font->pf_defaultsize.y)

/*
 * SunView-dependent size definitions
 */

#define LEFT_MARGIN     5               /* margin on left side of frame */
#define RIGHT_MARGIN    5               /* margin on right side of frame */
#define BOTTOM_MARGIN   5               /* margin on bottom of frame */
#define SUBWINDOW_SPACING       5       /* space in between adjacent
                                           subwindows */

/*
 * application-dependent size definitions
 */

#define CANVAS_1_WIDTH  320             /* width in pixels of canvas 1 */
#define CANVAS_1_HEIGHT 160             /* height in pixels of canvas 1 */
#define CANVAS_3_COLUMNS        30      /* width in characters of canvas 3 */
```

```
/*
 * catch_resize
 *
 * interposed function which checks all input events passed to the frame
 * for resize events; if it finds one,  resize() is called to refit
 * the subwindows; checking is done AFTER the frame processes the
 * event because if the frame changes its size due to this event (because
 * the window has been opened or closed for instance) we want to fit
 * the subwindows to the new size
 */

static Notify_value
catch_resize(frame, event, arg, type)
    Frame frame;
    Event *event;
    Notify_arg arg;
    Notify_event_type type;
{
        Notify_value value;

        value = notify_next_event_func(frame, event, arg, type);
        if (event_id(event) == WIN_RESIZE)
                resize(frame);
        return(value);
}
/*
 * resize -- fit the subwindows of the frame to its new size
 */

resize(frame)
    Frame frame;
{
        Rect *r;
        int canvas_3_width;     /* the width in pixels of canvas 3 */
        int stripeheight;       /* the height of the frame's name stripe */

        /* if the window is iconic, don't do anything */

        if ((int)window_get(frame, FRAME_CLOSED))
                return;

        /* find out our new size parameters */

        r = (Rect *) window_get(frame, WIN_RECT);
        framerect = *r;
        stripeheight = (int) window_get(frame, WIN_TOP_MARGIN);

        canvas_3_width = CANVAS_3_COLUMNS * font->pf_defaultsize.x
                + (int) scrollbar_get(SCROLLBAR, SCROLL_THICKNESS);
        window_set(Canvas_2,
                WIN_X,          0,
                WIN_Y,          0,
                WIN_WIDTH,      framerect.r_width - canvas_3_width
                                - LEFT_MARGIN - SUBWINDOW_SPACING
                                - RIGHT_MARGIN,
                WIN_HEIGHT,     framerect.r_height - CANVAS_1_HEIGHT
                                - stripeheight - SUBWINDOW_SPACING -
                                BOTTOM_MARGIN,
```

A.7. *dctool*

dctool is a simple reverse-polish notation calculator which demonstrates how to use pipes to write a SunView-based front end for an existing non-SunView program. *dctool* consists of a panel with buttons for each digit, the four arithmetic operations, and an enter key. The digits you hit are displayed in a message item and are sent via a pipe to *dc(1)*, the UNIX desk calculator. When *dc* computes an answer, it is sent back to *dctool* via a second pipe and it is displayed.

Note also the use of a single notify procedure for all of the digit buttons. The actual digit associated with each button is stored as the client data for each panel item, so the notify procedure can determine which button was pressed by looking at the client data. This value is then passed directly to dc. The operation buttons also all use a single notify procedure.

When you run *dctool*, remember that it is a reverse-polish notation calculator. For instance, to compute 3 * 5 you must hit the buttons 3, Enter, 5, and * in that order. If you prefer infix notation, you could easily adapt *dctool* to use *bc(1)* instead of *dc(1)*.

```
    window_main_loop(frame);
    exit(0);
}

static
create_panel_items(panel)
    Panel           panel;
{
    int             c;
    char            name[2];
    static void     digit_proc(), op_proc();
    static struct {
        int             col, row;
    }               positions[10] = {
                { 0, 3 }, { 0, 0 }, { 6, 0 }, { 12, 0 },
                          { 0, 1 }, { 6, 1 }, { 12, 1 },
                          { 0, 2 }, { 6, 2 }, { 12, 2 }
                 };

    name[1] = ' ';
    for (c = 0; c < 10; c++) {
        name[0] = c + '0';
        digit_item[c] = panel_create_item(panel, PANEL_BUTTON,
          PANEL_LABEL_IMAGE, panel_button_image(panel, name, 3, 0),
          PANEL_NOTIFY_PROC, digit_proc,
          PANEL_CLIENT_DATA, (caddr_t) (c + '0'),
          PANEL_LABEL_X,     ATTR_COL(positions[c].col),
          PANEL_LABEL_Y,     ATTR_ROW(positions[c].row),
          0);
    }
    add_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "+", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '+',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(0),
        0);
    sub_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "-", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '-',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(1),
        0);
    mul_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "*", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '*',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(2),
        0);
    div_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "/", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '/',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(3),
        0);
    if (pipe(pipeto) < 0 || pipe(pipefrom) < 0) {
        perror("calc");
        exit(1);
    }
    switch (childpid = fork()) {

    case -1:
        perror("calc");
        exit(1);

    case 0:                     /* this is the child process */
        /* use dup2 to set the child's stdin and stdout to the pipes */
        dup2(pipeto[0], 0);
        dup2(pipefrom[1], 1);

        /*
         * close all other fds (except stderr) since the child
         * process doesn't know about or need them
         */

        numfds = getdtablesize();
        for (c = 3; c < numfds; c++)
            close(c);

        /* exec the child process */

        execl("/usr/bin/dc", "dc", 0);
        perror("calc (child)"); /* shouldn't get here */
        exit(1);

    default:                    /* this is the parent */
        close(pipeto[0]);
        close(pipefrom[1]);
        tochild = pipeto[1];
        fp_tochild = fdopen(tochild, "w");
        fromchild = pipefrom[0];
        fp_fromchild = fdopen(fromchild, "r");

        /*
         * the pipe to dc must be unbuffered or dc will not get
         * any data until 1024 characters have been sent
         */

        setbuf(fp_tochild, NULL);
        break;
    }
}

/*
 * notify proc called whenever there is data to read on the pipe
 * from the child process; in this case it is an answer from dc,
 * so we display it
 */

static          Notify_value
pipe_reader(frame, fd)
    Frame           frame;
    int             fd;
```

A.8. *typein*

This program shows how to replace the functionality of the Graphics Tool and gfxsw package previously available under SunWindows. *typein* provides a tty emulator for interaction with the user and a canvas to draw on. To demonstrate it, a simple application is included which allows the user to input coordinates in the tty emulator and then draws the vectors in the canvas.

typein uses a tty subwindow and a canvas. Normally, the tty subwindow is used to allow a child process to run in a window; in this case, we would like the same process to write in that window. To accomplish this, we create two pipes and fork a child process called *loopback*. Loopback simply reads data from its stdin and one of the pipes, and sends it back out on the other pipe and stdout respectively. The parent process (*typein*) uses `dup2` to use the pipes as stdin and stdout, so that whatever it writes on stdout is sent to loopback which sends it to the tty emulator. When the user types something in the emulator, it is sent to loopback which sends it via a pipe to typein which reads it on stdin.

NOTE When using this mechanism, be careful of the following problems. First, you must use the Notifier (unlike the old gfxsw). Second, if you use the standard I/O package, be sure to either use `fflush` carefully or to remove all buffering with `setbuf` because the package will think you are sending data to a file and not to a tty. Finally, be sure you never block on a read because the program will hang (either use non-blocking I/O or only read one line at a time).

In a future release of SunView a better facility will be provided to accomplish the same functionality.

```
        pw = canvas_pixwin(canvas);

        /* set up a notify proc so that whenever there is input to read on
           stdin (fd 0), we are called to read it */

        notify_set_input_func(frame, read_input, STDIN_FD);

        printf("Enter first coordinate: ");
        fflush(stdout);
        window_main_loop(frame);
}

/* this section implements a simple application which reads coordinates and
   draws vectors; it uses a state machine to keep track of what number to
   read next */

#define GET_X_1         0
#define GET_Y_1         1
#define GET_X_2         2
#define GET_Y_2         3
int state = GET_X_1;
int x1, y1, x2, y2;

Notify_value
read_input(client, fd)
Notify_client client;
int fd;
{
        char buf[512];
        char *ptr, *gets();

        ptr = gets(buf);        /* only read one line per call so that we
                                   don't ever block */
        switch (state) {
        case GET_X_1:
                if (sscanf(buf, "%d", &x1) != 1) {
                        printf("Illegal value!0? ");
                        fflush(stdout);
                } else {
                        printf("y? ");
                        fflush(stdout);
                        state++;
                }
                break;
        case GET_Y_1:
                if (sscanf(buf, "%d", &y1) != 1) {
                        printf("Illegal value!0? ");
                        fflush(stdout);
                } else {
                        printf("Enter second coordinate:0? ");
                        fflush(stdout);
                        state++;
                }
                break;
        case GET_X_2:
                if (sscanf(buf, "%d", &x2) != 1) {
                        printf("Illegal value!0? ");
                        fflush(stdout);
```

A.9. Programs that Manipulate Color

The following two programs work with color. You can run them on a monochrome workstation to no ill-effect, but you won't see much of interest.

The techniques employed by these two programs are explained in the *Color* section of Chapter 7, *Imaging Facilities: Pixwins.*

When using these programs, try invoking them with different colors using the frame's command line arguments. Also, run *showcolor* (listed in the pixwin chapter) to see how the screen's colormap changes as different color programs are run simultaneously.

coloredit

The first program, *coloredit*, puts up sliders that let the user modify its colors.

```
/* create a reusable attribute list for my slider attributes */
sliderdefaults = attr_create_list(
                        PANEL_SHOW_ITEM,        TRUE,
                        PANEL_MIN_VALUE,        0,
                        PANEL_MAX_VALUE,        255,
                        PANEL_SLIDER_WIDTH,     512,
                        PANEL_SHOW_RANGE,       TRUE,
                        PANEL_SHOW_VALUE,       TRUE,
                        PANEL_NOTIFY_LEVEL,     PANEL_ALL,
                        0);

panel_create_item(panel, PANEL_CYCLE,
                  PANEL_LABEL_STRING,   "Edit colormap:",
                  PANEL_VALUE,          MYCANVAS,
                  PANEL_CHOICE_STRINGS, "Frame", "Panel", "Canvas", 0,
                  PANEL_NOTIFY_PROC,    editcms,
                  0);

text_item = panel_create_item(panel, PANEL_TEXT,
                        PANEL_VALUE_DISPLAY_LENGTH,     CMS_NAMESIZE,
                        PANEL_VALUE_STORED_LENGTH,      CMS_NAMESIZE,
                        0);

color_item = panel_create_item(panel, PANEL_SLIDER,
                               ATTR_LIST,               sliderdefaults,
                               PANEL_LABEL_STRING,      "color:",
                               PANEL_NOTIFY_PROC,       set_color,
                               0);

red_item = panel_create_item(panel, PANEL_SLIDER,
                             ATTR_LIST,                 sliderdefaults,
                             PANEL_LABEL_STRING,        "  red:",
                             PANEL_NOTIFY_PROC,         change_value,
                             0);

green_item = panel_create_item(panel, PANEL_SLIDER,
                               ATTR_LIST,               sliderdefaults,
                               PANEL_LABEL_STRING, "green:",
                               PANEL_NOTIFY_PROC, change_value,
                               0);

blue_item = panel_create_item(panel, PANEL_SLIDER,
                              ATTR_LIST,                sliderdefaults,
                              PANEL_LABEL_STRING, " blue:",
                              PANEL_NOTIFY_PROC, change_value,
                              0);

panel_create_item(panel, PANEL_BUTTON,
                  PANEL_LABEL_IMAGE,
                  panel_button_image(panel, "Cycle colormap", 12, NULL),
                  PANEL_NOTIFY_PROC, cycle,
                  0);

window_fit(panel);
window_fit_width(base_frame);

/* free the slider attribute list */
free(sliderdefaults);
```

```
        pw_getcolormap(pw, 0, cms.cms_size, red, green, blue);

        panel_set(color_item,
                  PANEL_VALUE, 0,
                  PANEL_MAX_VALUE, cms.cms_size - 1,
                  0);
        /* call the proc to set the colors */
        set_color(NULL, 0, NULL);
}

int             cur_color;
/* ARGSUSED */
static void
set_color(item, color, event)
        Panel_item      item;
        unsigned int    color;
        struct inputevent *event;
{
        panel_set_value(red_item, red[color]);
        panel_set_value(green_item, green[color]);
        panel_set_value(blue_item, blue[color]);
        cur_color = (unsigned char) color;
}

/* ARGSUSED */
static void
change_value(item, value, event)
        Panel_item      item;
        int             value;
        struct inputevent *event;
{
        if (item == red_item)
                red[cur_color] = (unsigned char) value;
        else if (item == green_item)
                green[cur_color] = (unsigned char) value;
        else
                blue[cur_color] = (unsigned char) value;

        /* pw_putcolormap expects arrays of colors, but this only sets one color */
        pw_putcolormap(pw, cur_color, 1,
                       &red[cur_color], &green[cur_color], &blue[cur_color]);
}

/* ARGSUSED */
static void
cycle(item, event)
        Panel_item      item;
        Event          *event;
{
        pw_cyclecolormap(pw, 1, 0, mycms_sizes[cur_cms]);
}
```

```
/****************************************************************/
/* animatecolor.c                                               */
/****************************************************************/

#include <suntool/sunview.h>
#include <suntool/canvas.h>

/****************************************************************/
/* You set MYCOLORS & MYNBITS according to how many colors     */
/* you are using; rest is just boilerplate, more or less;      */
/* after it you define your colors.                            */
/****************************************************************/
/*
 * define the colors I want in the canvas; max 16, must be a
 * power of 2
 */
#define MYCOLORS        4
/*
 * define the number of bits my colors take up -- MYCOLORS log 2;
 * maximum for animation to be possible is half screen's bits per
 * pixel -- 4 bits on current Sun color displays.
 */
#define MYNBITS         2
/*
 * to "hide" one set of planes while displaying another takes a
 * large cms -- the square of the number of colors
 */
#define MYCMS_SIZE      (MYCOLORS * MYCOLORS)

/*
 * when you write out a color pixel, you must write the color in
 * the appropriate planes.  This macro writes it in both sets
 */
#define usecolor(i)     ( (i) | ((i) << colorstuff.colorbits) )

struct colorstuff {
    /* desired colors */
    unsigned char   redcolors[MYCOLORS];
    unsigned char   greencolors[MYCOLORS];
    unsigned char   bluecolors[MYCOLORS];
    /* number of bits the desired colors take up */
    int             colorbits;
    /* colormap segment size */
    int             cms_size;
    /* 2 colormaps to support it */
    unsigned char   red[2][MYCMS_SIZE];
    unsigned char   green[2][MYCMS_SIZE];
    unsigned char   blue[2][MYCMS_SIZE];
    /* 2 masks to support it */
    int             enable_0_mask;
    int             enable_1_mask;
    /* current colormap -- 0 or 1 */
    int             cur_buff;
    /* plane mask to control which planes are written to */
    int             plane_mask;
};
```

```
/* draw the squares, then swap the colormap to animate them */
/* ARGSUSED */
static          Notify_value
my_draw(client, itimer_type)
    Notify_client   client;
    int             itimer_type;
{
#define SQUARESIZE      50
#define MAX_VEL         (SQUARESIZE / 5)
    /* number of squares to animate */
#define NUMBER  (MYCOLORS - 1)

    static int      posx[NUMBER], posy[NUMBER];
    static int      vx[NUMBER], vy[NUMBER];
    static int      prevposx[NUMBER], prevposy[NUMBER];
    int             i;

    /* set the plane mask to be that which we are not viewing */
    pw_putattributes(pw, (colorstuff.cur_buff == 1) ?
      &(colorstuff.enable_1_mask): &(colorstuff.enable_0_mask));

    /* write to invisible planes */
    for (i = 0; i < NUMBER; i++) {
        if (!times_drawn) {
            /* first time drawing */

            posx[i] = (i + 1) * 100;
            posy[i] = 50;
            vx[i] = r(-MAX_VEL, MAX_VEL);
            vy[i] = r(-MAX_VEL, MAX_VEL);
        }
        if (abs(vx[i]) > MAX_VEL) {
            printf("Weird value (%d) for vx[%d]0, vx[i], i);
            vx[i] = r(-MAX_VEL, MAX_VEL);
        }
        posx[i] = posx[i] + vx[i];
        if (posx[i] < 0) {
            /* Bounce off the left wall */
            posx[i] = 0;
            vx[i] = -vx[i];
        } else if (posx[i] > Xmax - SQUARESIZE) {
            /* Bounce off the right wall */
            vx[i] = -vx[i];
            posx[i] = posx[i] + vx[i];
        }
        posy[i] = posy[i] + vy[i];
        if (posy[i] > Ymax - SQUARESIZE) {
            /* Bounce off the top */
            posy[i] = Ymax - SQUARESIZE;
            vy[i] = -vy[i];
        } else if (posy[i] < 0) {
            /* Bounce off the bottom */
            posy[i] = 0;
            vy[i] = -vy[i];
        }
        /* draw the square you can't see */
        pw_rop(pw, posx[i], posy[i], SQUARESIZE, SQUARESIZE,
               PIX_SRC | PIX_COLOR(usecolor(i + 1)), NULL, 0, 0);
    Ymax = height - 1;
}

/*
 * Do double buffering by changing the write enable planes and
 * the color maps. The application draws into a buffer which is
 * not visible and when the buffers are swapped the invisible one
 * become visible and the other become invis.
 *
 * Start out drawing into buffer 1 which is the low-order buffer;
 * ie. the low-order planes. Things would not work if this is not
 * done because the devices start out be drawing with color 1
 * which will only hit the low-order planes.
 *
 * Init double buffering: Allocate color maps for both buffers. Fill
 * in color maps.
 */


doublebuff_init(colorstuff)
    struct colorstuff *colorstuff;
{
    /*
     * user has defined desired colors.  Set them up in the two
     * colormap segments
     */
    int             index_1;
    int             index_2;
    int             i;
    char            cmsname[CMS_NAMESIZE];

    /* name colormap something unique */
    sprintf(cmsname, "animatecolor%D", getpid());
    pw_setcmsname(pw, cmsname);

    /*
     * for each index in each color table, figure out how it maps
     * into the original color table.
     */
    for (i = 0; i < colorstuff->cms_size; i++) {
        /*
         * first colormap will show color X whenever low order
         * bits of color index are X
         */
        index_1 = i & ((1 << colorstuff->colorbits) - 1);
        /*
         * second colormap will show color X whenever high order
         * bits of color index are X
         */
        index_2 = i >> colorstuff->colorbits;

        colorstuff->red[0][i] = colorstuff->redcolors[index_1];
        colorstuff->green[0][i] = colorstuff->greencolors[index_1];
        colorstuff->blue[0][i] = colorstuff->bluecolors[index_1];

        colorstuff->red[1][i] = colorstuff->redcolors[index_2];
        colorstuff->green[1][i] = colorstuff->greencolors[index_2];
        colorstuff->blue[1][i] = colorstuff->bluecolors[index_2];
```

A.10. Two gfx subwindow–based programs converted to use SunView

The following two programs are the Sun demo programs *bouncedemo* and *spheresdemo* converted from using `gfxsw_init()` to canvases in SunView.

The code for the SunWindows–based programs is in */usr/src/sun/suntools* so you can contrast that code with the SunView versions printed here.

Techniques used to convert programs such as these to SunView are described in appendix C, *Converting SunWindows Programs to SunView.*

bounce

The first program is *bouncedemo* converted to draw in a canvas and to call `notify_dispatch()` periodically. Like the original *bouncedemo*, it restarts drawing after any damage (if not retained) or resizing.

```
                        0);
        for (--argc, ++argv; *argv; argv++) {
                /*
                 * handle the arguments that gfxsw_init(0, argv) used to do
                 * for you
                 */
                if (strcmp(*argv, "-r") == 0)
                        retained = 1;
                if (strcmp(*argv, "-n") == 0)
                        if (argc > 1) {
                                (void) sscanf(*(++argv), "%hD", &gfx->gfx_reps);
                                argc++;
                        }
        }

        canvas = window_create(frame, CANVAS,
                                CANVAS_RETAINED, retained,
                                CANVAS_RESIZE_PROC, resize_proc,
                                WIN_ERROR_MSG, "Can't create canvas",
                                0);

        /* only need to define a repaint proc if not retained */
        if (!retained) {
                window_set(canvas,
                           CANVAS_REPAINT_PROC, repaint_proc,
                           0);
        }
        pw = canvas_pixwin(canvas);

        gfx->gfx_pixwin = canvas_pixwin(canvas);

        /* Interpose my proc so I know that the tool is going away. */
        (void) notify_interpose_destroy_func(frame, my_notice_destroy);

        /*
         * Note: instead of window_main_loop, just show the frame. The
         * drawing loop is in control, not the notifier.
         */
        window_set(frame, WIN_SHOW, TRUE, 0);

Restart:
        rect = (Rect *) window_get(canvas, WIN_RECT);
        Xmax = rect_right(rect);
        Ymax = rect_bottom(rect);
        if (Xmax < Ymax)
                size = Xmax / 29 + 1;
        else
                size = Ymax / 29 + 1;
        /*
         * the following were always 0 in a gfx subwindow (bouncedemo
         * is confused on this point
         */
        x = 0;
        y = 0;

        vx = 4;
        vy = 0;
        ylast = 0;
```

```
                                if (vy == 0)
                                        goto Reset;
                        }
                        for (z = 0; z <= 1000; z++);
                        continue;
Reset:
                        if (--gfx->gfx_reps <= 0)
                                break;
                        x = 0;
                        y = 0;
                        vx = 4;
                        vy = 0;
                        ylast = 0;
                        ylastcount = 0;
                }
}

static void
repaint_proc( /* Ignore args */ )
{
        /* if repainting is required, just restart */
        gfx->gfx_flags |= GFX_RESTART;
}

static void
resize_proc( /* Ignore args */ )
{
        gfx->gfx_flags |= GFX_RESTART;
}

/* this is straight from the Notifier chapter */
static          Notify_value
my_notice_destroy(frame, status)
        Frame           frame;
        Destroy_status  status;
{
        if (status != DESTROY_CHECKING) {
                /* set my flag so that I terminate my loop soon */
                my_done = 1;
                /* Stop the notifier if blocked on read or select */
                (void) notify_stop();
        }
        /* Let frame get destroy event */
        return (notify_next_destroy_func(frame, status));
}
```

```
/*************************************************************/
/*                        spheres.c                          */
/* Draw a bunch of shaded spheres.  Algorithm by Tom Duff,     */
/* Lucasfilm Ltd., 1982.  Old SunWindows spheresdeom program   */
/* revised to use SunView canvas instead of gfxsw.            */
/*************************************************************/

#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <sunwindow/cms_rainbow.h>

static Notify_value my_frame_interposer();
static Notify_value my_animation();
static void     sphere();
static void     demoflushbuf();

#define ITIMER_NULL     ((struct itimerval *)0)

/* (NX, NY, NZ) is the light source vector -- length should be 100 */
#define NX 48
#define NY -36
#define NZ 80

#define BUF_BITWIDTH    16

static struct pixrect *mpr;
static int      width;
static int      height;
static int      counter;
static Frame    frame;
static Canvas   canvas;
static int      cmssize;
static Pixwin  *pw;

static short    spheres_image[256] = {
#include "spheres.icon"
};

mpr_static(spheres_pixrect, 64, 64, 1, spheres_image);

main(argc, argv)
    int             argc;
    char          **argv;
{
    char          **args;
    int             usefullgray = 0;
    Icon            icon;

    icon = icon_create(ICON_IMAGE, &spheres_pixrect, 0);
    frame = window_create(NULL, FRAME,
                         FRAME_LABEL,          "spheres",
                         FRAME_ICON,           icon,
                         FRAME_ARGC_PTR_ARGV,  &argc, argv,
                         0);
    canvas = window_create(frame, CANVAS,
      CANVAS_AUTO_EXPAND,       0,
      CANVAS_AUTO_SHRINK,       0,
      CANVAS_AUTO_CLEAR,        0,
        for (y = -maxy; y <= maxy; y++) {
            mark = r(0, radius * 100) <= NX * x + NY * y
                + NZ * sqroot(radius * radius - x * x - y * y);
            if (mark)
                pr_put(mpr, xbuf, y + y0, color);
        }
        if (xbuf == (mpr->pr_width - 1)) {
            demoflushbuf(mpr, PIX_SRC | PIX_DST, x + x0 - mpr->pr_width, pw);
            xbuf = 0;
            x++;
            return (NOTIFY_DONE);
        }
        x++;
    }
    if (x >= radius)
        demoflushbuf(mpr, PIX_SRC | PIX_DST, x + x0 - (xbuf + 2), pw);
    return (NOTIFY_DONE);
}

static void
demoflushbuf(mpr, op, x, pixwin)
    struct pixrect *mpr;
    int             op;
    int             x;
    struct pixwin  *pixwin;
{
    register u_char *sptr, *end;

    sptr = mprd8_addr(mpr_d(mpr), 0, 0, mpr->pr_depth);
    end = mprd8_addr(mpr_d(mpr), mpr->pr_width - 1,
                     mpr->pr_height - 1, mpr->pr_depth);
    /* Flush the mpr to the pixwin. */
    pw_write(pixwin, x, 0, mpr->pr_width, mpr->pr_height, op,
             mpr, 0, 0);
    /* Clear mpr with 0's */
    while (sptr <= end)
        *sptr++ = 0;
    /* Let user interact with tool */
    notify_dispatch();
}

static int
setuprainbowcolormap(pw)
    Pixwin          *pw;
{
    register u_char red[CMS_RAINBOWSIZE];
    register u_char green[CMS_RAINBOWSIZE];
    register u_char blue[CMS_RAINBOWSIZE];

    /* Initialize to rainbow cms. */
    pw_setcmsname(pw, CMS_RAINBOW);
    cms_rainbowsetup(red, green, blue);
    pw_putcolormap(pw, 0, CMS_RAINBOWSIZE, red, green, blue);
    return (CMS_RAINBOWSIZE);
}

static int
setupfullgraycolormap(pw)
```

B

Sun User Interface Conventions

Sun User Interface Conventions **409**

B.1. Program Names 409

B.2. Frame Headers 409

B.3. Menus 409

Capitalization 409

Menus Showing Button Modifiers 410

Interaction with Standard Menus 410

Enable/Disable Menu Items 410

Multi-Column Menus 410

B.4. Panels 410

Buttons 411

List of Non-Exclusive Choices 411

List of Exclusive Choices 411

Binary Choices 412

Text Items 412

Allocation of Function Between Buttons and Menus 412

B.5. Mouse Button Usage 413

Allocation of Function Between Mouse Buttons 413

Using Mouse Buttons for Accelerators 413

B.6. Cursors 413

B.7. Icons 413

B

Sun User Interface Conventions

The window programs released by Sun follow some standard user interface conventions. These conventions are described here so that, if you choose, you can design your interfaces with them in mind.

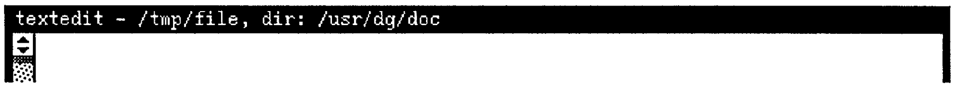
B.1. Program Names

Here are some guidelines for naming programs:

- A window-based version of an existing tty-based program has *tool* appended to the end of the existing program. For example *mailtool* is a window-based version of the tty-based program *mail.*
- A program without a tty version should not end with *tool.* Thus the icon editor is called *iconedit* and not *icontool.*
- Since tools are normally invoked from command files or menus, descriptive names are better than short cryptic ones. Thus *iconedit* is better than *ied.*

B.2. Frame Headers

The frame header should contain the name of the program, optionally followed by a dash and additional information, as in:

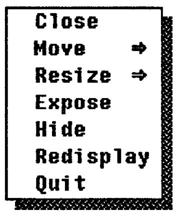


```
textedit - /tmp/file, dir: /usr/dg/doc
```

B.3. Menus

Capitalization

The words in menus should be capitalized as they would be in a chapter heading:

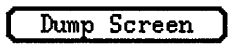


This convention can be bent when the names in the menu correspond to already existing, non-capitalized command names.

Buttons

The proper use of buttons is to allow the user to initiate commands. Button items should not be used to represent categories, modes or options — for these kinds of choices that imply a change of state, you should use toggle, choice or cycle items, as described in the next three sections.

When creating a button, use the routine `panel_button_image()` to create a button-like image, as in:

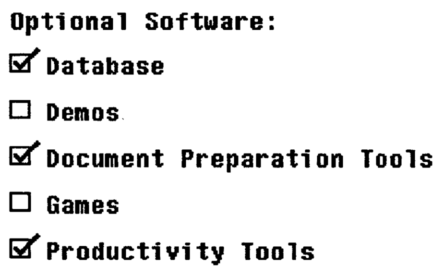


As with menu entries, capitalize buttons unless the button name matches something else (for example, *dbx* commands in *dbxtool*). If the button's meaning can be modified by CTRL or SHIFT these modifiers should be indicated in the button's menu. (For an example, see the picture of the **Reply** menu from *mailtool*, at the top of the preceding page.)

In most cases, a button will remain visible all the time. However, when a tool has different states, and a button can only be used in some of those states, it is usually best to make the button invisible when it can not be invoked. Thus in *mailtool*, the Cancel button only appears when a letter is being composed.

List of Non-Exclusive Choices

A list of choices in which more than one choice can be selected at a time is best implemented with the item type `PANEL_TOGGLE`. The default for toggles is a list of check boxes:



Optional Software:

☑ Database

☐ Demos

☑ Document Preparation Tools

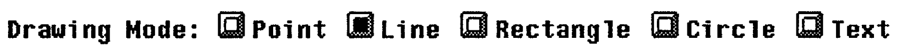
☐ Games

☑ Productivity Tools

The example shows a vertical list; vertical or horizontal are both acceptable..

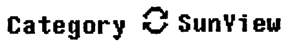
List of Exclusive Choices

A list of choices in which only one choice can be selected at a time can be displayed with all choices visible or with only the current choice visible. To show all the choices, use the item type `PANEL_CHOICE`. The default for choice items is a list of square pushbuttons, with the current choice marked by a darkened pushbutton:



Drawing Mode: □ Point ■ Line □ Rectangle □ Circle □ Text

To show only the current choice, use `PANEL_CYCLE`. This item type provides a symbol consisting of two circular arrows, which indicate to the user that he can cycle through choices, and serves to distinguish cycle items from text items:



Category ⟳ SunView

B.5. Mouse Button Usage

Allocation of Function Between Mouse Buttons

Use of mouse buttons should be consistent with the rest of SunView. The left button should only be used to make selections. The right button should only be used to bring up menus./**

There is some discretion involved in the use of the middle button, however. In most of SunView, the middle button is used to adjust a selection. In text and shell windows, for example, the left button is used to mark the starting point of a selection, and the middle button is used to extend the selection. Similarly, in a pixel editor that allowed you to select regions, clicking the left button on a region could select just that region, and clicking the middle button on another region could add that region to the selection. On the other hand, in a tool that allowed you to move objects, the middle button could move an object, and CTRL middle button could re-size it, which would be consistent with the way icons and frames are moved and re-sized. As a third alternative, in the *iconedit* drawing program the left button draws pixels (which is a kind of selecting) and the middle button erases.

The best use of the middle button is still being discussed. Future versions of this guideline may specify more exactly how the middle button should be used. For now, the most common use is to extend the selection, and the next-most common is to move a graphic object.

Using Mouse Buttons for Accelerators

It is acceptable to use the mouse buttons as accelerators for common operations. The only caveat is that any accelerators should also be available from a menu or panel item. Thus in SunView clicking on a tool with the middle button moves the tool, but you can also move a tool using the frame menu.

Some operations, on the other hand, cannot be invoked from a menu or panel button. In such cases the mouse is the only means of invoking the operation. For example, in *iconedit* you use the mouse for drawing, and the drawing operations are not available from a menu or button.

B.6. Cursors

An application program should not do anything other than change the shape of the cursor when the cursor is moved into a new window. *textedit* presents a good example of using the cursor to alert the user that input is interpreted differently in different regions: The cursor is a thin diagonal arrow in the textsubwindow, a fat horizontal arrow in the scrollbar, and a diamond in the scrollbar buttons.

B.7. Icons

Tools should pack as much information as possible into their icons. *clock* and *perfmeter* are examples of tools that make good use of icon real estate. *textedit* is an example of a tool that could make better use of its icon. For example, it could contain a representation of the text being editing in a 1 point font. Small as that is, you can tell at a glance if you are editing C code or a mail message.

[97] People who want to hold the mouse with their left hand can put the "menu button" on the left and the "select button" on the right by setting the *Left_Handed* option in the *Input* category of *defaultsedit*.

C

Converting SunWindows Programs to SunView

Converting SunWindows Programs to SunView **417**

C.1. Converting Tools 418

General Comments 418

Programming Style Changes 418

Object `typedefs` 418

Attribute Value Interface 418

New Objects 419

Canvas subwindows 419

Text subwindows 419

Scrollbars 419

Objects in Common between SunView and SunWindows 420

Cursors 420

Icons 420

Menus 420

Input Events 421

Setting up Input Event Handling 421

Sigwinch handling 422

Windows 422

Panels 422

Signals 423

Prompts 423

C.2. Converting gfxsubwindow–based code 425

Basic Steps 425

C

Converting SunWindows Programs to SunView

This appendix gives some guidelines for converting programs written using SunWindows to SunView. There are two classes of programs covered: those that create a tool and subwindows, and programs that call `gfxsw_init()` to take over an existing window or the console.

Programs that fall outside these classes are probably UNIX style programs that do not use windows at all. The conversion of such programs is in effect the subject of this whole manual. If you want to convert such a program to SunView, pay particular attention to the *SunView Model* chapter, and the specific discussion of Notifier interaction in *Porting Programs to SunView* in the *Notifier* chapter. You may also find some of the discussion in the *Converting gfxsubwindow–based code* section of this appendix helpful.

New Objects

Most of the data types in the above list are objects new in SunView. Many objects in SunWindows correspond to objects in SunView, for example:

```
tool    ⇒  Frame
ttysw   ⇒  Tty
```

Some objects such as the graphics subwindow and empty subwindow are not supported in SunView[98]. There are new objects that partially take their place.

Canvas subwindows

The canvas subwindow is a general-purpose drawing subwindow, which can replace gfx subwindows and empty subwindows. The size of the canvas you draw on need not be the same as the size of the window it is displayed in; you can create scrollbars to let the user adjust the visible part of the canvas. For a demonstration of the various canvas attributes, run the program */usr/demo/canvas_demo*

Text subwindows

These allow for the display and editing of text in a scrollable window. The user can perform various actions on the text, including saving the text, searching in the text, and editing the text without the programmer having to deal with these interactions.

Since there was no such window in SunWindows, your application may have had to use a gfx subwindow, a set of panel message items, or some strange technique involving `ttysw_input()` or piping to a tty subwindow to display text; the text subwindow can replace all these uses.

Scrollbars

Scrollbars can be attached to windows. In particular, the use of scrollbars with retained canvases makes it very easy to draw a fixed-size image without regard for window size changes.

[98] You can still compile and run code that uses these, but Sun does not intend to develop them further.

Menus also have their own routines and are created via function calls instead of being user-loaded data structures. They use the pointer type `Menu` for their handles instead of `struct menuptr`. One way to create them is to write a special `menu_init()` proc which loads them into their structures correctly. In your `menu_init()`, you have something like

```
m1_items = menu_create(
                    MENU_STRING_ITEM, "insert",      INSERT,
                    MENU_STRING_ITEM, "copy",        COPY  ,
                    MENU_STRING_ITEM, "replace",     REPLACE,
                    MENU_STRING_ITEM, "move",        XLATE ,
                    MENU_STRING_ITEM, "delete",      DELETE,
                    MENU_STRING_ITEM, "HELP",        DRAW_HELP,
          0);
```

Menu values from `menu_get()` or `menu_show()` are returned as `caddr_t`'s. Be sure your types match.

NOTE The old `menu_display()` and the new `menu_show()` routines have a different order for the arguments.

Input Events

The `inputevent` structure has not changed. However, you no longer have to generate events yourself in ''selected'' routines via calls to `input_readevent()`. Instead, windows now have event handlers that are passed pointers to `Event` structures.

There are a number of macros for making input events easier to deal with in SunView, so instead of having something like `ie->ie_code` you have `event_id(ie)`, resulting in more readable code.

`Event` types are not pointers, so you have to distinguish between

```
Event *ie;
```

and

```
Event ie;
```

in your code. You can use either, because the event functions don't just manipulate a handle as, for example, the cursor functions do. See *Object Handles* in Chapter 3, ***Programmatic Interface***, for an explanation of when handles are pointers and when not.

Setting up Input Event Handling

All the input events can be set up from the `window_create()` call or `window_set()` calls. Calls to `win_*inputmask()` are all replaced by these `window_set()` and `window_create()` calls.

The distinction between ''pick'' and ''keyboard'' events is new in SunView, having been added to support the notion of a split input focus.

CAUTION **Be careful that when you are setting mouse events, you are modifying the `WIN_*_PICK_EVENTS` and when you are setting keyboard events you modify `WIN_*_KEYBOARD_EVENTS`. You may get inconsistent results if you modify pick events on the keyboard mask.**

Signals

If you are catching signals, then you should read the documentation on signals in the *Restrictions* section of Chapter 16, *The Notifier*. There are several that the Notifier now catches on your behalf.

You should no longer be catching `SIGWINCH` signals. If you do, your program may never appear on the screen as it will start catching the signals and redrawing endlessly on the screen, which may not be visible.

Prompts

Instead of using the `menu_prompt()` facility of SunWindows, you can use popup subframes and `window_loop(popup_frame)` when prompting the user. One of the example programs in Chapter 4, *Windows*, uses these to implement a confirmer popup.

`menu_prompt()` is documented here for completeness. The definitions used by `menu_prompt()` are:

```
struct prompt {
    Rect     prt_rect;
    Pixfont *prt_font;
    char    *prt_text;
}

menu_prompt(prompt, event, windowfd)
    struct prompt     *prompt;
    struct inputevent *event;
    int                windowfd;
```

`menu_prompt()` displays the string addressed by `prompt->prt_text` using the font `prompt->prt_font`. `prompt->prt_rect` is relative to `windowfd`. If either the `r_width` or the `r_height` fields of `prompt->prt_rect` has the value `PROMPT_FLEXIBLE`, that dimension is chosen to accommodate all the characters in `prompt->prt_text`.

The fullscreen access method is used to display the prompt. After displaying the prompt, `menu_prompt()` waits for any input event other than mouse motion. It then removes the prompt, and returns the event which caused the return in `event`. `windowfd` is the file descriptor of the window from which input is taken while the prompt is up.

C.2. Converting gfxsubwindow–based code

Programs that run in gfxsubwindows are designed to take over an existing window. In SunView you must create a tool for such programs to run in. One limitation of this approach is that the SunView version of the application must run under `suntools`; the old `gfxsw_init()` call would create a SunWindows environment if run on the "bare" Sun console. One major advantage gained by moving to SunView is that your code can use scrollbars.

Basic Steps

- Include `<suntool/sunview.h>` and `<suntool/canvas.h>`.
- Remove all window-related `#include` statements; these will probably be included by `sunview.h`.
- Declare a `Frame` and a `Canvas`.
- Replace `gfxsw_init()` with calls to create the frame and canvas.

Replacing Tool Interaction

Styles of Damage Checking

Many gfx subwindow programs (and many of the Sun demos) call `gfxsw_init()` to take over a window, then run in a loop as they compute and draw an image in the gfx subwindow. At some point in the loop they check for damage to or alteration of the size of the gfx subwindow and handle it accordingly.

In SunView, the coexistence of your program with the window system is less hidden from you. Read the chapter *SunView Model* to understand how this coexistence works. In converting programs, you must ensure the Notifier runs at regular intervals so that window events such as close, quit, etc. are handled appropriately.

Consult the chapter on the Notifier for more information.

You can either (1) set up your program so that, after initialization, control passes to the Notifier, which you have set up to call your imaging/computation routine periodically, or (2) let control continue to pass to your code, and change the program to call the Notifier at regular intervals.

Either the Notifier Takes Over

In the first case, you set up your imaging/computation routine as a function that is called when a timer expires. Do this by calling `notify_set_itimer_func()`. If you want your imaging/computation routine to blaze away non-stop (causing other programs to run more sluggishly), you request the timer function be called as soon as the Notifier has handled window events for you by giving the timer the special value `&NOTIFY_POLLING_ITIMER`.

```
(void) notify_set_itimer_func(frame, my_animation,
        ITIMER_REAL, &NOTIFY_POLLING_ITIMER, ITIMER_NULL);
```

If your code `sleep()`'s on a regular basis, then you should be able to modify it so that the Notifier calls your imaging/computation routine at the same interval.

The program *spheres* in Appendix A, *Example Programs*, is an example of this style of interaction.

canvas you draw in are available through the canvas attributes `CANVAS_WIDTH` and `CANVAS_HEIGHT`. The fields of the `gfx->gfxsw_rect` correspond to these attributes as follows:

`coord   r_left, r_top;`	are both = 0
`short   r_width, r_height;`	are the `CANVAS_WIDTH` and `CANVAS_HEIGHT` attributes.

As described above, you can use your own `GFX_RESTART` and `GFX_REPAINT` flags.

If you care about the gfxsw command line arguments, insert code into your program's `argv,argc` parsing loop to handle the gfx options that used to be taken care of for you. The *bounce* program has reasonable code to do this.

Finishing Up

If your imaging routine is in control and periodically calls the Notifier, then when the window is quit your routine must know that this has occurred. Otherwise, the imaging routine will continue to draw in a window that has been destroyed, and you will see error messages like

```
WIN ioctl number C0146720: Bad file number
```

until you kill the program.

What you must do is interpose in front of the frame's destroy event handler so that your program will know when the frame goes away. See the item on *Getting out* in *Porting Programs to SunView* in the *Notifier* chapter.

If your program exits on its own, then it can call `window_done()` to destroy its windows. This will invoke your interposed notice-destroy routine (which may or may not matter depending on what it does). It will also call the standard

```
Press the left mouse button to confirm Quit...
```

confirmer unless you set `FRAME_NO_CONFIRM`.

Miscellaneous

`gfxsw_getretained()` is equivalent to the `CANVAS_RETAINED` attribute. Canvases are retained by default.

`gfxsw_init()` doesn't consume the gfxsw command line options `-r`, `-n` *Number_of_repetitions*, etc; your code may do strange things with its arguments to deal with this.

Index

A

action procedure for menu item, 191
alarm, 247
ASCII events, 77
asynchronous signal notification, 255
ATTR_COL, 50, 140, 271
ATTR_COLS, 272
ATTR_LIST, 273
ATTR_ROW, 50, 140, 271
ATTR_ROWS, 272
attribute functions
  attr_create_list(), 272
attribute lists
  creating reusable lists, 272
  default attributes, 273
  overview, 28
attribute ordering, 51

B

base frame, 16
boundary manager, 19
BUT(), 77
button image constructor, 148
button panel item, 138, 148 *thru* 150
buttons with menus, 149

C

callback style of programming, 20
canvas
  automatic sizing, 65
  backing pixrect, 61
  canvas space vs. window space, 66
  color in canvases, 69
  coordinate system, 61
  default input mask, 66
  definition of, 57
  handling input, 66
  interface summary, **278**
  model, 60
  monochrome on a Sun-3/110, 112
  non-retained, 62
  pixwin, 58, 61
  repaint procedure, 62
  repainting, 61
  resize procedure, 63
  retained, 61

canvas, *continued*
  scrolling, 59
  table of attributes, 278
  table of functions and macros, 279
  tracking changes in size, 63
  writing your own event procedure, 66
canvas attributes, **278**
  CANVAS_AUTO_CLEAR, 62, 278
  CANVAS_AUTO_EXPAND, 65, 278
  CANVAS_AUTO_SHRINK, 65, 278
  CANVAS_FAST_MONO, 112, 278
  CANVAS_FIXED_IMAGE, 63, 278
  CANVAS_HEIGHT, 61, 278
  CANVAS_MARGIN, 278
  CANVAS_PIXWIN, 61, 92, 278
  CANVAS_REPAINT_PROC, 62, 278
  CANVAS_RESIZE_PROC, 278
  CANVAS_RETAINED, 62, 278
  CANVAS_WIDTH, 61, 65, 278
canvas functions and macros, **279**
  canvas_event(), 66, 279
  canvas_pixwin(), 58, 61, 92, 279
  canvas_window_event(), 66, 279
canvas subwindow package, 57 *thru* 70
CAPSMASK, 81, 82
character unit macros
  ATTR_COL, 50, 140, 271
  ATTR_COLS, 272
  ATTR_ROW, 50, 140, 271
  ATTR_ROWS, 272
child process control using the Notifier, 250
choice panel item, 138, 150 *thru* 155
classes of windows, 16
client handles for the Notifier, 249
clipping in a pixwin, 102
code examples, see *example programs*
color, **103**
  advanced colormap manipulation example program, 385
  animation, 111, 390
  background color of pixwin, 104
  color during fullscreen access, 109
  colormap, 103
  colormap access, 107
  colormap segment, 104
  cursors and menus, 109
  default colormap segment, 104
  determining if display is color, 109
  double buffering, 110, 111

event, *continued*
- repaint and resize event codes, 78
- shift key event codes, 80
- timeout, 255
- window entry and window exit event codes, 78

event codes, **288**
- BUT(), 77
- KBD_DONE, 78
- KBD_REQUEST, 78
- KBD_USE, 78
- KEY_LEFT, 79
- KEY_RIGHT, 79
- KEY_TOP, 79
- LOC_DRAG, 66, 77
- LOC_MOVE, 77
- LOC_RGNENTER, 78
- LOC_RGNEXIT, 78
- LOC_STILL, 77
- LOC_TRAJECTORY, 77
- LOC_WINENTER, 78
- LOC_WINEXIT, 78
- MS_LEFT, 77
- MS_MIDDLE, 77
- MS_RIGHT, 77
- PANEL_EVENT_CANCEL, 170
- PANEL_EVENT_DRAG_IN, 170
- PANEL_EVENT_MOVE_IN, 170
- SHIFT_CAPSLOCK, 80
- SHIFT_CTRL, 80
- SHIFT_LEFT, 80
- SHIFT_LOCK, 80
- SHIFT_META, 80
- SHIFT_RIGHT, 80
- WIN_REPAINT, 66, 78
- WIN_RESIZE, 66, 78, 261
- WIN_STOP, 78

event descriptors, **289**
- WIN_ASCII_EVENTS, 66, 84, 289
- WIN_IN_TRANSIT_EVENTS, 84, 289
- WIN_MOUSE_BUTTONS, 84, 289
- WIN_NO_EVENTS, 84, 289
- WIN_UP_ASCII_EVENTS, 84, 289
- WIN_UP_EVENTS, 84, 289

event handling
- at the Notifier level, 250
- in canvases, 66
- in panels, 169

event procedure
- writing your own for a canvas, 66
- writing your own for a panel item, 171

event state retrieval macros
- event_ctrl_is_down(), 81
- event_id(), 81
- event_is_ascii(), 81
- event_is_button(), 81
- event_is_down(), 81
- event_is_key_left(), 81
- event_is_key_right(), 81
- event_is_key_top(), 81
- event_is_meta(), 81
- event_is_up(), 81
- event_meta_is_down(), 81
- event_shift_is_down(), 81

event state retrieval macros, *continued*
- event_shiftmask(), 81
- event_time(), 81
- event_x(), 81
- event_y(), 81

event state setting macros
- event_set_down(), 82
- event_set_id(), 82
- event_set_shiftmask(), 82
- event_set_up(), 82
- event_set_x(), 82
- event_set_y(), 82

event stream, 74

example programs, **341**
- *animatecolor*, 112, 390
- *bounce*, 397
- color manipulation, 385
- *coloredit*, 107, 385
- colormap manipulation, 385
- creating menus, 362
- *dctool*, 375
- discussion of *image_browser_1* program, 46
- discussion of *image_browser_2*, 49
- discussion of simple file manager, 41
- *filer*, 41, 341
- *font_menu*, 362
- gfx subwindow-based demos converted to SunView, 397
- gfxsw_init to SunView, 397, 402
- *image_browser_1*, 350
- *image_browser_2*, 354
- list files in tty subwindow, 39
- minimal SunView program, 35
- notify_dispatch(), 397
- notify_set_itimer_func(), 402
- *resize_demo*, 48, 261, 370
- row/column space in a window, 354
- *showcolor*, 105
- simple file manager, 341
- simple panel window, 36
- *spheres*, 402
- subwindow layout, 350
- tty subwindow escape sequences, 359
- tty subwindow I/O, 359
- *tty_io*, 359
- *typein*, 381

F

fcntl, 248, 252

file descriptor usage, 54

filer, 41

flow of control in notifier-based programs, 21

font functions
- pf_default(), 38, 95
- pf_open(), 38

frame
- command line frame attributes, 338
- definition of, 16
- frame header, 18
- layout of subwindows within a frame, 47, 48, 261, 370
- menus, 18
- modifying destruction using the Notifier, 262
- modifying open/close using the Notifier, 259
- table of attributes, 334

it_interval struct, 257

K

KBD_DONE, 78
KBD_REQUEST, 78
KBD_USE, 78
KEY_LEFT, 79
KEY_RIGHT, 79
KEY_TOP, 79
keyboard focus, 78

L

layout of items within a panel, 140
layout of subwindows within a frame, 47
libraries used in SunView, 27
LOC_DRAG, 77
LOC_MOVE, 77
LOC_RGNENTER, 78
LOC_RGNEXIT, 78
LOC_STILL, 77
LOC_TRAJECTORY, 77
LOC_WINENTER, 78
LOC_WINEXIT, 78
locator, 77
locator motion event codes, 77

M

Menu, 285
menu
  attributes to add pre-existing menu items, 198
  basic usage, 186
  callback procedures, 203
  client data, 190
  default selection, 212
  destruction, 201
  display stage of menu processing, 209
  example program, 362
  for panel items, 138
  generate procedure, 203
  inactive items, 193
  initial selection, 212
  interface summary, 291
  notification stage of menu processing, 210
  notify procedure, 190, 208
  pullright, 185
  searching for a menu item, 202
  shadow, 190
  table of attributes, 291
  table of functions, 296
  table of menu item attributes, 294
  user customizable attributes, 213
menu attributes, **291**
  MENU_ACTION_IMAGE, 200, 291, 294
  MENU_ACTION_ITEM, 200, 291, 294
  MENU_ACTION_PROC, 294
  MENU_APPEND_ITEM, 198, 291, 294
  MENU_BOXED, 194, 213, 291, 294
  MENU_CLIENT_DATA, 190, 291, 294
  MENU_DEFAULT, 291
  MENU_DEFAULT_ITEM, 212, 291
  MENU_DEFAULT_SELECTION, 212, 213, 291
menu attributes, *continued*
  MENU_DESCEND_FIRST, 202, 291
  MENU_FEEDBACK, 197, 294
  MENU_FIRST_EVENT, 291
  MENU_FONT, 190, 196, 213, 291, 294
  MENU_GEN_PROC, 204, 205, 291, 294
  MENU_GEN_PROC_IMAGE, 294
  MENU_GEN_PROC_ITEM, 294
  MENU_GEN_PULLRIGHT, 294
  MENU_GEN_PULLRIGHT_IMAGE, 200, 207, 291, 294
  MENU_GEN_PULLRIGHT_ITEM, 200, 207, 291, 294
  MENU_GEN_PULLRIGHT_PROC, 207
  MENU_IMAGE, 294
  MENU_IMAGE_ITEM, 187, 191, 200, 291, 294
  MENU_IMAGES, 189, 200, 291
  MENU_INACTIVE, 193, 294
  MENU_INITIAL_SELECTION, 212, 213, 291
  MENU_INITIAL_SELECTION_EXPANDED, 213, 292
  MENU_INITIAL_SELECTION_SELECTED, 213, 292
  MENU_INSERT, 197, 198, 292
  MENU_INSERT_ITEM, 198, 292
  MENU_INVERT, 294
  MENU_ITEM, 188, 196, 200, 292
  MENU_JUMP_AFTER_NO_SELECTION, 213, 292
  MENU_JUMP_AFTER_SELECTION, 213, 292
  MENU_LAST_EVENT, 292
  MENU_LEFT_MARGIN, 192, 213, 292, 295
  MENU_MARGIN, 192, 194, 213, 292, 295
  MENU_NCOLS, 194, 292
  MENU_NITEMS, 196, 292
  MENU_NOTIFY_PROC, 292
  MENU_NROWS, 194, 292
  MENU_NTH_ITEM, 196, 292
  MENU_PARENT, 292, 295
  MENU_PULLRIGHT, 188, 196, 295
  MENU_PULLRIGHT_DELTA, 213, 293
  MENU_PULLRIGHT_IMAGE, 200, 293, 295
  MENU_PULLRIGHT_ITEM, 199, 200, 293, 295
  MENU_RELEASE, 197, 201, 295
  MENU_RELEASE_IMAGE, 295
  MENU_REMOVE, 198, 293
  MENU_REMOVE_ITEM, 198, 293
  MENU_REPLACE, 198, 293
  MENU_REPLACE_ITEM, 198, 293
  MENU_RIGHT_MARGIN, 192, 213, 293, 295
  MENU_SELECTED, 293, 295
  MENU_SELECTED_ITEM, 212, 293
  MENU_SHADOW, 190, 195, 213, 293
  MENU_STRING, 188, 196, 295
  MENU_STRING_ITEM, 187, 191, 200, 293, 295
  MENU_STRINGS, 200, 293
  MENU_TITLE_IMAGE, 190, 293
  MENU_TITLE_ITEM, 190, 293
  MENU_TYPE, 293, 295
  MENU_VALID_RESULT, 293
  MENU_VALUE, 187, 191, 295
menu callback procedures
  generate procedures, 203
  notify and action procedures, 208
menu data types
  Menu, 285
  Menu_generate, 203, 285
  Menu_item, 285
menu functions, **296**

panel, *continued*
- data types, 285
- default event-to-action mapping, 169
- definition of, 138
- event handling mechanism, 169
- interface summary, 301
- item label, 138
- item menu, 138
- iterating over all items in a panel, 168
- modifying attributes, 142
- painting, 166
- panel space vs. window space, 174
- panel-wide item attributes, 143
- positioning items within a panel, 140
- retrieving attributes, 144
- simple panel window example, 36
- table of attributes, 301
- table of functions and macros, 308
- table of generic panel item attributes, 302
- using scrollbars with, 145

panel attribute settings
- PANEL_ALL, 150, 156, 160, 164
- PANEL_CLEAR, 166
- PANEL_CURRENT, 150
- PANEL_DONE, 164
- PANEL_HORIZONTAL, 141, 159
- PANEL_INVERTED, 152
- PANEL_MARKED, 152
- PANEL_NO_CLEAR, 166
- PANEL_NON_PRINTABLE, 160
- PANEL_NONE, 150, 152, 156, 160, 166
- PANEL_SPECIFIED, 160
- PANEL_VERTICAL, 141, 150, 159

panel attributes, **301**
- PANEL_ACCEPT_KEYSTROKE, 169, 170, 301, 302
- PANEL_BACKGROUND_PROC, 169, 170, 301
- PANEL_BLINK_CARET, 144, 301
- PANEL_CARET_ITEM, 142, 160, 301
- PANEL_CHOICE_FONTS, 304
- PANEL_CHOICE_IMAGE, 144, 304
- PANEL_CHOICE_IMAGES, 150, 304
- PANEL_CHOICE_STRING, 304
- PANEL_CHOICE_STRINGS, 150, 304
- PANEL_CHOICE_X, 304
- PANEL_CHOICE_XS, 150, 304
- PANEL_CHOICE_Y, 304
- PANEL_CHOICE_YS, 304
- PANEL_CHOICE_YS,, 150
- PANEL_CHOICES_BOLD, 304
- PANEL_CLIENT_DATA, 168, 302
- PANEL_DISPLAY_LEVEL, 150, 156, 304
- PANEL_EVENT_PROC, 169, 171, 301, 302
- PANEL_FEEDBACK, 152, 304
- PANEL_FIRST_ITEM, 168, 301
- PANEL_ITEM_RECT, 302
- PANEL_ITEM_X, 140, 302
- PANEL_ITEM_X_GAP, 141, 301
- PANEL_ITEM_Y, 140, 302
- PANEL_ITEM_Y_GAP, 141, 301
- PANEL_LABEL_BOLD, 143, 301, 302
- PANEL_LABEL_FONT, 302
- PANEL_LABEL_IMAGE, 147, 302
- PANEL_LABEL_STRING, 147, 302
- PANEL_LABEL_X, 141, 302

panel attributes, *continued*
- PANEL_LABEL_Y, 141, 302
- PANEL_LAYOUT, 141, 143, 150, 159, 301, 302, 304
- PANEL_MARK_IMAGES, 150, 304
- PANEL_MARK_X, 304
- PANEL_MARK_XS, 150, 305
- PANEL_MARK_Y, 305
- PANEL_MARK_YS, 150, 305
- PANEL_MASK_CHAR, 307
- PANEL_MAX_VALUE, 164, 165, 306
- PANEL_MENU_CHOICE_FONTS, 302
- PANEL_MENU_CHOICE_IMAGES, 302
- PANEL_MENU_CHOICE_STRINGS, 163, 302
- PANEL_MENU_CHOICE_VALUES, 163, 303
- PANEL_MENU_MARK_IMAGE, 158, 305
- PANEL_MENU_NOMARK_IMAGE, 158, 305
- PANEL_MENU_TITLE_FONT, 303
- PANEL_MENU_TITLE_IMAGE, 303
- PANEL_MENU_TITLE_STRING, 303
- PANEL_MIN_VALUE, 164, 165, 306
- PANEL_NEXT_ITEM, 168, 303
- PANEL_NOMARK_IMAGES, 150
- PANEL_NOTIFY_LEVEL, 160, 164, 306, 307
- PANEL_NOTIFY_PROC, 148, 152, 160, 303
- PANEL_NOTIFY_STRING, 160, 307
- PANEL_PAINT, 143, 166, 303
- PANEL_PARENT_PANEL, 303
- PANEL_SHOW_ITEM, 144, 160, 303
- PANEL_SHOW_MENU, 143, 301, 303
- PANEL_SHOW_MENU_MARK, 152, 305
- PANEL_SHOW_RANGE, 164, 306
- PANEL_SHOW_VALUE, 164, 306
- PANEL_SLIDER_WIDTH, 164, 306
- PANEL_TOGGLE_VALUE, 305
- PANEL_VALUE, 165, 305, 306, 307
- PANEL_VALUE_DISPLAY_LENGTH, 159, 307
- PANEL_VALUE_FONT, 306, 307
- PANEL_VALUE_STORED_LENGTH, 159, 307
- PANEL_VALUE_X, 141, 303
- PANEL_VALUE_Y, 141, 303

panel data types
- Panel, 285
- Panel_attribute, 285
- Panel_item, 285
- Panel_setting, 285

panel functions and macros, **308**
- panel_accept_key(), 171, 308
- panel_accept_menu(), 171, 308
- panel_accept_preview(), 171, 308
- panel_advance_caret(), 160, 308
- panel_backup_caret(), 160, 308
- panel_begin_preview(), 171, 308
- panel_button_image(), 148, 308
- panel_cancel_preview(), 171, 308
- panel_create_item(), 38, 140, 308
- panel_default_handle_event(), 170, 309
- panel_destroy_item(), 144, 309
- panel_each_item(), 309
- panel_event(), 174, 309
- panel_get(), 144, 309
- panel_get_value(), 144
- panel_paint(), 166, 309
- panel_set(), 142, 309
- panel_set_value(), 142

popup windows, *continued*
  non-blocking, 42
porting programs to SunView, 265
  SunWindows-based, **417**, see *converting programs to SunView*
programmatic scrolling, 237
pty (pseudo-tty), 54
pw_batch, 100, 311

R

reading events, 87
Rect struct, 48
refusing the keyboard input focus, 78
regions of a pixwin, 102
registering an event handler with the Notifier, 250
releasing the event lock, 87
rendering speed, 98
reserved namespaces, 29
restrictions on use of UNIX facilities by SunView applications, 247
row/column space in a window, 49
  example program, 354

S

sample programs, see *example programs*
Scroll_motion, 285
Scrollbar, 285
scrollbar
  interface summary, 317
  table of attributes, 317
  table of functions, 320
  use with canvases, 59
  use with panels, 145
scrollbar attributes, **317**
  SCROLL_ABSOLUTE_CURSOR, 317
  SCROLL_ACTIVE_CURSOR, 317
  SCROLL_ADVANCED_MODE, 317
  SCROLL_BACKWARD_CURSOR, 317
  SCROLL_BAR_COLOR, 317
  SCROLL_BAR_DISPLAY_LEVEL, 317
  SCROLL_BORDER, 317
  SCROLL_BUBBLE_COLOR, 317
  SCROLL_BUBBLE_DISPLAY_LEVEL, 317
  SCROLL_BUBBLE_MARGIN, 317
  SCROLL_DIRECTION, 317
  SCROLL_END_POINT_AREA, 317
  SCROLL_FORWARD_CURSOR, 317
  SCROLL_GAP, 317
  SCROLL_HEIGHT, 318
  SCROLL_LAST_VIEW_START, 318
  SCROLL_LEFT, 318
  SCROLL_LINE_HEIGHT, 318
  SCROLL_MARGIN, 318
  SCROLL_MARK, 318
  SCROLL_NORMALIZE, 318
  SCROLL_NOTIFY_CLIENT, 318
  SCROLL_OBJECT, 318
  SCROLL_OBJECT_LENGTH, 318
  SCROLL_PAGE_BUTTON_LENGTH, 318
  SCROLL_PAGE_BUTTONS, 318
  SCROLL_PAINT_BUTTONS_PROC, 318
  SCROLL_PIXWIN, 318
  SCROLL_PLACEMENT, 235, 318
scrollbar attributes, *continued*
  SCROLL_RECT, 318
  SCROLL_REPEAT_TIME, 319
  SCROLL_REQUEST_MOTION, 319
  SCROLL_REQUEST_OFFSET, 319
  SCROLL_THICKNESS, 319
  SCROLL_TO_GRID, 319
  SCROLL_TOP, 319
  SCROLL_VIEW_LENGTH, 319
  SCROLL_VIEW_START, 319
  SCROLL_WIDTH, 319
scrollbar data types
  Scroll_motion, 285
  Scrollbar, 285
  Scrollbar_attribute, 285
  Scrollbar_attribute_value, 285
  Scrollbar_setting, 285
scrollbar functions, **320**
  scrollbar_clear_bubble(), 320
  scrollbar_create(), 145, 235, 320
  scrollbar_destroy(), 235
  scrollbar_get(), 235
  scrollbar_paint(), 320
  scrollbar_paint_bubble(), 320
  scrollbar_paint_clear(), 320
  scrollbar_scroll_to(), 237, 320
  scrollbar_set(), 235, 320
scrollbar package, 231 *thru* 237
Scrollbar_attribute, 285
Scrollbar_attribute_value, 285
Scrollbar_setting, 285
selection of panel items
  buttons, 148
  choices, 152
  sliders, 164
  text, 160
  toggles, 156
Selection Service, 241 *thru* 242
SELN_LEVEL_FIRST, 119
SELN_LEVEL_LINE, 119
setting position of mouse cursor, 83
SHIFT_CAPSLOCK, 80
SHIFT_CTRL, 80
SHIFT_LEFT, 80
SHIFT_LOCK, 80
SHIFT_META, 80
SHIFT_RIGHT, 80
SHIFTMASK, 81, 82
showcolor, 105
SIGALRM, 248
sigblock, 255
SIGCHLD, 248
SIGCONT, 246
SIGIO, 248
signal, 247, 253
signals — use with notifier, 253
SIGPIPE, 254
SIGTERM, 248
SIGURG, 78, 248
sigvec, 247
SIGVTALRM, 248

text subwindow data types, *continued*
  Textsw, 285
  Textsw_index, 118, 285
  Textsw_status, 126, 285
text subwindow functions, **326**
  textsw_add_mark(), 124, 326
  textsw_append_file_name(), 127, 326
  textsw_delete(), 119, 326
  textsw_edit(), 119, 326
  textsw_erase(), 119, 326
  textsw_file_lines_visible(), 122, 326
  textsw_find_bytes(), 123, 326
  textsw_find_mark(), 124, 327
  textsw_first(), 131, 327
  textsw_index_for_file_line(), 122, 327
  textsw_insert(), 118, 327
  textsw_next(), 131, 327
  textsw_normalize_view(), 123, 327
  textsw_possibly_normalize(), 122, 327
  textsw_remove_mark(), 125, 327
  textsw_replace_bytes(), 120, 328
  textsw_reset(), 127, 131, 328
  textsw_save(), 126, 131, 328
  textsw_screen_line_count(), 122, 328
  textsw_scroll_lines(), 122, 328
  textsw_set_selection(), 125, 328
  textsw_store(), 131
  textsw_store_file(), 127, 328
text subwindow status values
  TEXTSW_STATUS_BAD_ATTR, 126
  TEXTSW_STATUS_BAD_ATTR_VALUE, 126
  TEXTSW_STATUS_CANNOT_ALLOCATE, 126
  TEXTSW_STATUS_CANNOT_OPEN_INPUT, 126
  TEXTSW_STATUS_OKAY, 126
  TEXTSW_STATUS_OTHER_ERROR, 126
Textsw, 285
Textsw_action, 128
Textsw_action attributes, **324**
  TEXTSW_ACTION_CAPS_LOCK, 324
  TEXTSW_ACTION_CHANGED_DIRECTORY, 324
  TEXTSW_ACTION_EDITED_FILE, 324
  TEXTSW_ACTION_FILE_IS_READONLY, 324
  TEXTSW_ACTION_LOADED_FILE, 324
  TEXTSW_ACTION_CAPS_LOCK, 130
  TEXTSW_ACTION_CHANGED_DIRECTORY, 130
  TEXTSW_ACTION_EDITED_FILE, 130, 131
  TEXTSW_ACTION_FILE_IS_READONLY, 130
  TEXTSW_ACTION_LOADED_FILE, 130, 131
  TEXTSW_ACTION_TOOL_CLOSE, 130
  TEXTSW_ACTION_TOOL_DESTROY, 130
  TEXTSW_ACTION_TOOL_MGR, 130
  TEXTSW_ACTION_TOOL_QUIT, 130
  TEXTSW_ACTION_USING_MEMORY, 130
  TEXTSW_ACTION_TOOL_CLOSE, 324
  TEXTSW_ACTION_TOOL_DESTROY, 324
  TEXTSW_ACTION_TOOL_MGR, 324
  TEXTSW_ACTION_TOOL_QUIT, 324
  TEXTSW_ACTION_USING_MEMORY, 324
Textsw_index, 118, 285
TEXTSW_INFINITY, 119, 120
Textsw_status, 126, 285
Textsw_status values, **325**
  TEXTSW_STATUS_BAD_ATTR, 325

Textsw_status values, *continued*
  TEXTSW_STATUS_BAD_ATTR_VALUE, 325
  TEXTSW_STATUS_CANNOT_ALLOCATE, 325
  TEXTSW_STATUS_CANNOT_OPEN_INPUT, 325
  TEXTSW_STATUS_OKAY, 325
  TEXTSW_STATUS_OTHER_ERROR, 325
timeout events, 255
toggle panel item, 139, 156 *thru* 158
translating events from canvas space to window space, 66
translating events from panel space to window space, 174
tty subwindow
  example program, 359
  example program to list files, 39
  input/output to tty subwindow, 180
  interface summary, 329
  overview, 179
  special escape sequences, 181
  standard escape sequences, 181
  table of functions, 329
  table of special escape sequences, 330
tty subwindow attributes
  TTY_ARGV, 329
  TTY_CONSOLE, 329
  TTY_PAGE_MODE, 329
  TTY_QUIT_ON_CHILD_DEATH, 329
tty subwindow functions, **329**
  example program, 359
  ttysw_input(), 39, 180, 329
  ttysw_output(), 181, 329

U

UNIX system calls
  alarm, 247
  fcntl, 248, 252
  free<(3), 272
  getitimer, 247
  ioctl, 248
  perror, 267
  sigblock, 255
  signal, 247, 253
  sigvec, 247
  wait, 247
  wait3, 247, 250
UNIX system calls not to be used under SunView, 247

V

views in text subwindows, 131
Virtual User Input Device, 75
VUID, 75

W

wait, 247
wait3, 247, 250
WIN_ASCII_EVENTS, 84, 289
WIN_EXTEND_TO_EDGE, 46, 47
WIN_IN_TRANSIT_EVENTS, 84, 289
WIN_LEFT_KEYS, 84, 289
WIN_MOUSE_BUTTONS, 84, 289
WIN_NO_EVENTS, 84, 289
WIN_REPAINT, 78
WIN_RESIZE, 78

Revision History

Revision	Date	Comments
– 02	17 February 1986	First release of the SunView Programmer's Guide as part of release 3.0.
– 10	15 October 1986	Rewritten and expanded for release 3.2

Notes

Notes

Notes