Principia Softwarica: The Plan 9 Widget Library
`libpanel`
version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Tom Duff

March 24, 2026

Contents

1 Introduction **7**
- 1.1 Motivations . . . 7
- 1.2 The Plan 9 widget library: `libpanel` . . . 7
- 1.3 Other widget libraries . . . 8
- 1.4 Getting started . . . 8
- 1.5 Requirements . . . 9
- 1.6 About this document . . . 9
- 1.7 Copyright . . . 10
- 1.8 Acknowledgments . . . 10

2 Overview **11**
- 2.1 Widget library principles . . . 11
  - 2.1.1 The widget . . . 11
  - 2.1.2 The event loop . . . 11
  - 2.1.3 Layout . . . 11
  - 2.1.4 Scrolling . . . 12
- 2.2 `hellopanel.c` . . . 12
- 2.3 Code organization . . . 14
- 2.4 Software architecture . . . 15
- 2.5 Book structure . . . 17

3 Core Data Structures **18**
- 3.1 `Panel` . . . 18
  - 3.1.1 Widget flags . . . 19
  - 3.1.2 A tree of widgets . . . 20
  - 3.1.3 Widget-specific data . . . 21
  - 3.1.4 Widget display state . . . 21
  - 3.1.5 Main widget methods . . . 22
- 3.2 `Icon` . . . 22
- 3.3 Layout styles . . . 22
  - 3.3.1 Placement . . . 23
  - 3.3.2 Packing . . . 24
  - 3.3.3 Filling . . . 24
  - 3.3.4 Padding . . . 25
- 3.4 `Direction` . . . 25

4 Initialization **26**
- 4.1 `plinit()` . . . 26

5 Drawing **28**
5.1 `pldraw()` . . . 28
5.2 Drawing helpers . . . 29
5.3 Drawing boxes . . . 29

6 Events **31**
6.1 Mouse events . . . 31
6.1.1 Mouse leaving the widget . . . 32
6.1.2 `REMOUSE` . . . 32
6.1.3 Hitting priority . . . 33
6.2 Keyboard events . . . 33

7 Layout **35**
7.1 Layout fields . . . 35
7.2 Computing sizes: `pl_sizereq()` . . . 36
7.2.1 Packing . . . 37
7.2.2 Padding . . . 38
7.3 Computing rectangles: `pl_setrect()` . . . 38
7.3.1 Padding . . . 39
7.3.2 Filling . . . 39
7.3.3 Placing . . . 40
7.3.4 Packing the children . . . 40

8 Basic Widgets **44**
8.1 Label . . . 44
8.1.1 Initializing . . . 44
8.1.2 Drawing . . . 45
8.1.3 Reacting . . . 47
8.1.4 Packing methods . . . 47
8.2 Text entry . . . 47
8.2.1 Initializing . . . 48
8.2.2 Drawing . . . 49
8.2.3 Reacting . . . 50
8.2.4 Packing methods . . . 52
8.2.5 Other methods . . . 52
8.3 Button . . . 53
8.3.1 `Button` . . . 53
8.3.2 Basic button . . . 54
8.3.3 Check button . . . 56
8.3.4 Radio button . . . 58
8.4 Slider . . . 59
8.4.1 Initializing . . . 60
8.4.2 Drawing . . . 60
8.4.3 Reacting . . . 62
8.4.4 Packing methods . . . 63
8.4.5 Other methods . . . 63
8.5 Canvas . . . 64
8.5.1 Initializing . . . 64
8.5.2 Drawing . . . 64
8.5.3 Reacting . . . 65

8.5.4 Packing methods . . . 65

9 Composite Widgets **66**
9.1 Frame . . . 66
9.1.1 Initializing . . . 66
9.1.2 Drawing . . . 67
9.1.3 Reacting . . . 67
9.1.4 Packing methods . . . 67
9.2 Group . . . 68
9.2.1 Initializing . . . 68
9.2.2 Drawing . . . 69
9.2.3 Reacting . . . 69
9.2.4 Packing methods . . . 69

10 Scrollable Widgets **70**
10.1 Scrolling fields . . . 70
10.2 Scrolling methods . . . 71
10.3 Scrollable list . . . 71
10.3.1 Initializing . . . 72
10.3.2 Drawing . . . 73
10.3.3 Reacting . . . 74
10.3.4 Packing methods . . . 75
10.3.5 Scrollbar to scrollee . . . 76
10.4 Scroll bar . . . 77
10.4.1 Initializing . . . 77
10.4.2 Drawing . . . 79
10.4.3 Reacting . . . 79
10.4.4 Packing methods . . . 80
10.4.5 Scrollee to scrollbar . . . 81

11 Menu widgets **82**
11.1 Menu items . . . 82
11.1.1 Initializing . . . 82
11.1.2 Drawing . . . 83
11.1.3 Reacting . . . 83
11.2 Popup menu . . . 83
11.2.1 Initializing . . . 84
11.2.2 Reacting . . . 84
11.2.3 Drawing . . . 85
11.2.4 Packing methods . . . 87
11.3 Pull-down . . . 87
11.3.1 Initializing . . . 88
11.3.2 Drawing . . . 88
11.3.3 Reacting . . . 88
11.3.4 Packing methods . . . 90
11.4 Menu bar . . . 90
11.4.1 Initializing . . . 91

12 Text Widgets **92**
12.1 Message . . . 92
12.1.1 Initializing . . . 92
12.1.2 Drawing . . . 93
12.1.3 Reacting . . . 94
12.1.4 Packing methods . . . 94
12.2 Edit . . . 95
12.2.1 Initializing . . . 95
12.2.2 Drawing . . . 96
12.2.3 Reacting . . . 96
12.2.4 Packing methods . . . 98
12.2.5 Other methods . . . 98
12.3 Text view . . . 100
12.3.1 Initializing . . . 101
12.3.2 Drawing . . . 101
12.3.3 Reacting . . . 102
12.3.4 Packing methods . . . 103
12.4 Completion . . . 103
12.5 Rich text . . . 103
12.5.1 Initializing . . . 104
12.5.2 Drawing . . . 104
12.5.3 Reacting . . . 106
12.5.4 Packing methods . . . 106
12.5.5 Other methods . . . 106

13 Advanced Topics **108**
13.1 copy/paste . . . 108
13.1.1 Widgets hooks . . . 109
13.2 Hooks . . . 111
13.3 Advanced layout . . . 111
13.3.1 `FIXEDX`, `FIXEDY` . . . 111
13.3.2 `MAXX`, `MAXY` . . . 111

14 Conclusion **113**
14.1 Patterns and techniques . . . 113
14.2 Connections to other books . . . 114
14.3 Missing features . . . 114
14.4 Beyond the Plan 9 widget library . . . 114

A Debugging **116**

B Error Management **118**

C Utilities **120**
C.1 Memory management . . . 120

D Extra Code **121**
D.1 `include/gui/` . . . 121
D.1.1 `include/gui/panel.h` . . . 121
D.1.2 `include/gui/rtext.h` . . . 124
D.2 `lib_gui/libpanel/` . . . 125

D.2.1 lib_gui/libpanel/pldefs.h . . . 125
D.2.2 lib_gui/libpanel/init.c . . . 127
D.2.3 lib_gui/libpanel/mem.c . . . 127
D.2.4 lib_gui/libpanel/draw.c . . . 127
D.2.5 lib_gui/libpanel/event.c . . . 128
D.2.6 lib_gui/libpanel/print.c . . . 128
D.2.7 lib_gui/libpanel/label.c . . . 128
D.2.8 lib_gui/libpanel/button.c . . . 129
D.2.9 lib_gui/libpanel/entry.c . . . 129
D.2.10 lib_gui/libpanel/edit.c . . . 130
D.2.11 lib_gui/libpanel/slider.c . . . 131
D.2.12 lib_gui/libpanel/canvas.c . . . 131
D.2.13 lib_gui/libpanel/frame.c . . . 131
D.2.14 lib_gui/libpanel/group.c . . . 131
D.2.15 lib_gui/libpanel/list.c . . . 132
D.2.16 lib_gui/libpanel/message.c . . . 132
D.2.17 lib_gui/libpanel/pack.c . . . 132
D.2.18 lib_gui/libpanel/popup.c . . . 133
D.2.19 lib_gui/libpanel/pulldown.c . . . 133
D.2.20 lib_gui/libpanel/rtext.c . . . 133
D.2.21 lib_gui/libpanel/scroll.c . . . 137
D.2.22 lib_gui/libpanel/scrollbar.c . . . 137
D.2.23 lib_gui/libpanel/snarf.c . . . 138
D.2.24 lib_gui/libpanel/textview.c . . . 138
D.2.25 lib_gui/libpanel/textwin.c . . . 140
D.2.26 lib_gui/libpanel/utf.c . . . 149

Glossary **151**

Index **152**

References **160**

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a widget library.

1.1 Motivations

Why a widget library? A widget library, also known as a GUI toolkit, is the layer that turns a bare windowing system into something a user can actually interact with: buttons, text fields, menus, scroll bars. Because I think you are a better programmer if you fully understand how things work under the hood, and because widget libraries are central to any graphical application, I think they deserve a book.

Every graphical application you use—a web browser, an editor, a file manager—is built on top of a widget library. Yet most programmers treat widgets as black boxes, never looking inside to understand how a button press gets dispatched, how a scroll bar communicates with the list it controls, or how the layout engine decides where to place each element on screen.

This is a pity because a widget library, despite its modest appearance, contains a surprising number of interesting techniques: event-driven programming with dispatching through a tree, a layout engine that negotiates sizes in two passes, object-oriented programming in C using function pointers for polymorphic dispatch, and inter-widget communication protocols for scrolling. These techniques are useful far beyond GUI programming. The event dispatch pattern appears in web browsers (the DOM), the two-pass layout algorithm is analogous to constraint solving, and the OO-in-C pattern is the foundation of many C libraries (e.g., GTK's GObject, the Linux kernel's VFS).

Unlike the windowing system (covered in the WINDOWS book [Pad16c]) which multiplexes the screen among processes, a widget library operates within a single process, organizing its visual elements into a tree of nested widgets. This tree is the central data structure of any widget library—it governs drawing order, event dispatch, and layout computation.

Here are a few questions I hope this book will answer:

- How does a mouse event get dispatched to the appropriate widget?
- How does scrolling work? How do widgets communicate with each other?
- How does a widget library lay out its elements? How does it decide the size and position of each widget?
- How are composite widgets built from simpler ones? How does a widget tree get assembled?

1.2 The Plan 9 widget library: `libpanel`

I will explain in this book the code of the Plan 9 widget library `libpanel`, which contains about 4000 lines of code (LOC). `libpanel` is written entirely in C.

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. `libpanel` was written by Tom Duff specifically to build `mothra`, the Plan 9 web browser. As Duff himself notes, "the design is modeled strongly on Ousterhout's Tcl/Tk, except that the programming language is C and most of Tcl/Tk's automatic behind-the-scenes recalculation and redrawing is missing." In `libpanel`, widgets are called "panels"—hence the library name.

The Plan 9 windowing system `rio` already provides a few rudimentary widget-like features: `menuhit()` for popup menus and `libframe` for editable text frames. However, these are limited and ad-hoc. `libpanel` provides a proper widget tree with a uniform interface for layout, drawing, and event dispatch—a real toolkit rather than a collection of standalone primitives.

At roughly 4000 LOC, `libpanel` is tiny compared to mainstream widget libraries. GTK is hundreds of thousands of lines; Qt is millions. Even the OCaml bindings to GTK (`lablgtk`) contain more code than `libpanel` itself, and `lablgtk` is just a binding, not a widget library. This extreme conciseness makes `libpanel` an ideal subject for a book: every line of code can be explained and understood.

1.3 Other widget libraries

Here are a few widget libraries that I considered for this book, but which I ultimately discarded:

- Xlib is not really a widget library—it is the low-level C interface to the X Window system. Writing a "hello world" with Xlib requires hundreds of lines of boilerplate [Ros88]. The Athena widget set (Xaw), built on top of Xlib and the Xt Intrinsics, is a proper widget library, but it is tied to X Window's complex client/server architecture.

- Tcl/Tk (really Tk, the widget library, paired with the Tcl scripting language) was one of the first toolkits to establish the basic vocabulary of GUI programming: buttons, labels, entries, listboxes, frames, and a geometry manager to lay them out. In some ways, `libpanel` is a C equivalent of Tk—and indeed, Inferno (Plan 9's successor) includes a Tk reimplementation in Limbo.

- GTK (the GIMP Toolkit) is the dominant open source widget library, used by GNOME and many Linux applications. However, GTK is enormous: hundreds of thousands of lines, with a complex GObject type system to simulate object-oriented programming in C.

- Qt is the other major cross-platform toolkit, written in C++ with a custom meta-object compiler (MOC) for signals and slots. Qt was originally developed for KDE and is even larger than GTK, with millions of lines of code.

Figure 1.1 presents a timeline of major widget libraries. `libpanel` belongs to the Plan 9 branch, highlighted in red.

I think `libpanel` represents the best compromise for this book: it implements the essential features of a widget library—a widget tree, layout engine, event dispatch, scrolling, and menus—while still having a small and understandable codebase (about 4000 LOC).

The widget library lineage goes back to the Xerox Alto and Smalltalk in the 1970s, where the MVC (Model-View-Controller) pattern was invented. On the industry side, Microsoft has MFC and WPF, Java has AWT and Swing, and Apple has Cocoa. On the minimal end, Nuklear is a single-header-file GUI library in C, showing that a widget toolkit does not need to be huge. `libpanel` sits squarely in this minimalist tradition.

1.4 Getting started

To play with `libpanel`, you will first need to install the Plan 9 fork used in Principia Softwarica (see `https://www.principia-softwarica.org/getting-started.html`). Once installed, you can compile and run a small

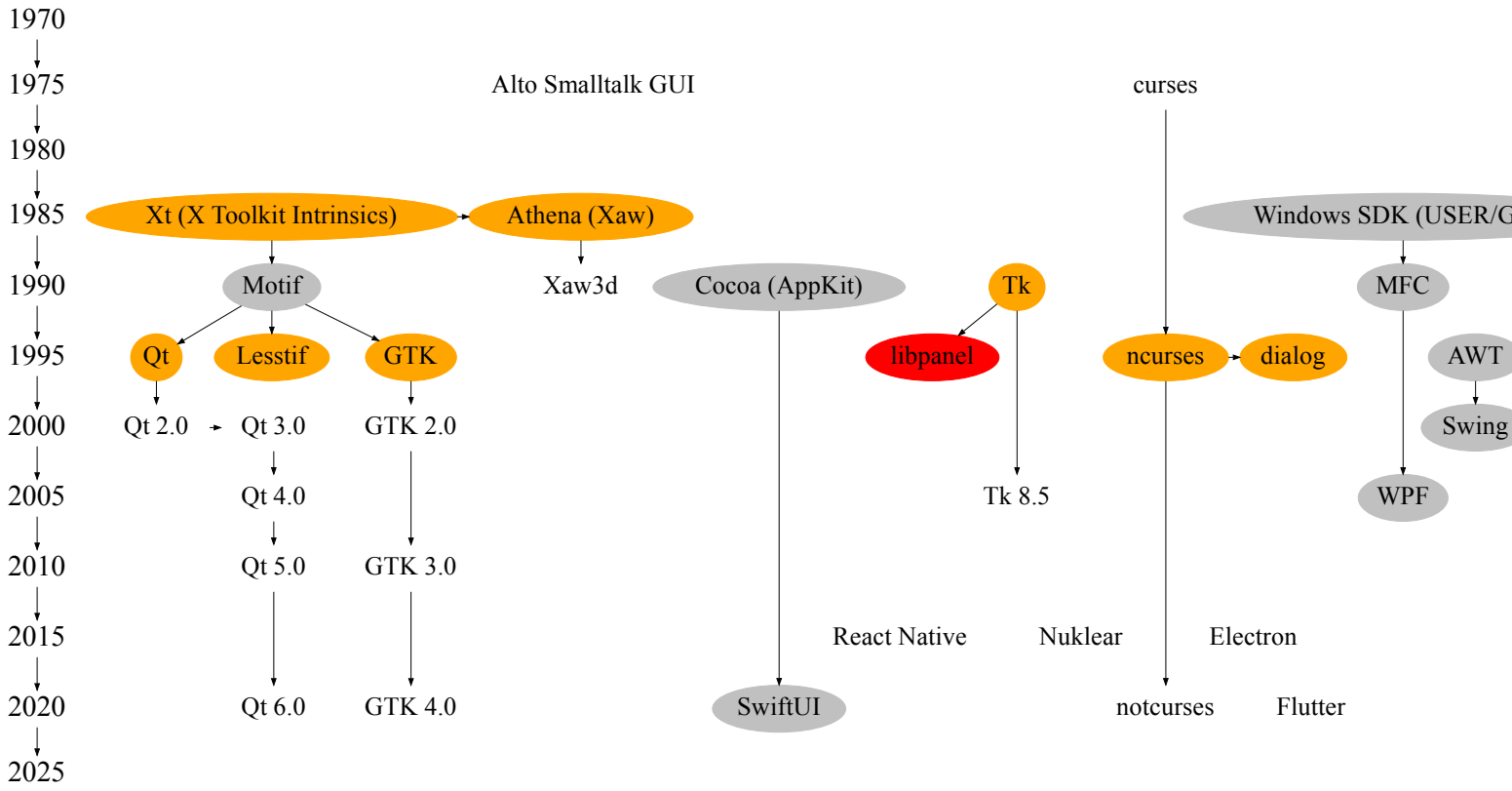


Figure 1.1: Widget libraries timeline

test program called `hellopanel.c` (described in more detail in Chapter 2) with the following commands:

```
1   % cd /lib_gui/libpanel/tests
2   % mk hellopanel
3   5c -c hellopanel.c
4   5l -o hellopanel hellopanel.5
5   % hellopanel
```

Line 2 runs `mk` to compile `hellopanel.c`. The Plan 9 C compiler `5c` produces an object file `hellopanel.5`, and the linker `5l` produces the executable. Line 5 runs the program, which opens a window containing a "Hello, world!" label and a "done" button. Clicking the button exits the program. This is the `libpanel` equivalent of a "hello world."

1.5 Requirements

The reader should be familiar with C and with the basics of Plan 9 graphics programming as covered in the GRAPHICS book [Pad16b] and WINDOWS book [Pad16c]. In particular, you should understand `Image`, `Rectangle`, and the `draw()` operation from `libdraw`, as well as the `Mouse` event structure from `libevent`.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

1.8 Acknowledgments

I would like to acknowledge of course the author of `libpanel`, Tom Duff, who wrote in some sense most of this book.

Chapter 2

Overview

Before showing the source code of `libpanel` in the following chapters, I first give an overview in this chapter of the general principles of a widget library. I also walk through a complete "hello world" program using `libpanel`, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Widget library principles

The following sections explain the core concepts that any widget library must address: what a widget is, how user input flows through the widget tree, how widgets are positioned on screen, and how scrolling works. These concepts apply to most widget libraries, not just `libpanel`.

2.1.1 The widget

A widget (a contraction of "window gadget") is a visual element that the user can see and interact with: a button, a label, a text entry, a scroll bar, a menu. Every widget has at least three responsibilities: it must draw itself on the screen, it must react to user input (mouse clicks, key presses), and it must report its desired size so that the layout engine can position it. In `libpanel`, widgets are represented by the `Panel` structure. Each widget type (label, button, entry, etc.) stores its type-specific data inside `Panel` and provides its own implementations of the draw, react, and size methods. This is essentially object-oriented programming in C: the `Panel` struct acts as a base class with virtual method pointers.

2.1.2 The event loop

A graphical application cannot simply execute a sequence of instructions and exit—it must wait for the user to do something. This leads to a fundamentally different programming model: the event loop, also known as event-driven programming. The program sits in an infinite loop, waiting for events (mouse movements, button clicks, key presses). When an event arrives, the program dispatches it to the appropriate widget, which reacts accordingly. In `libpanel`, the event loop is not hidden inside the library; it is written explicitly in `main()` by the application programmer (as shown in `hellopanel.c` below). The application calls `event()` to wait for an event, then calls `plmouse()` to dispatch it to the widget tree. This keeps the library simple: it does not need to manage the event loop itself.

2.1.3 Layout

One of the trickiest problems in a widget library is deciding where each widget goes and how big it should be. This is the job of the layout engine. In `libpanel`, layout works in two passes: first, each widget reports the

size it would like to have (bottom-up, from leaves to root); then, the available screen rectangle is divided among widgets (top-down, from root to leaves). `libpanel` uses a packing model: widgets are packed against one side of their parent—north (top), south (bottom), east (right), or west (left). The first child packed takes its strip of space, and the remaining rectangle is passed to the next child. This is similar to Tk's `pack` geometry manager.

2.1.4 Scrolling

Scrolling arises when the content of a widget is larger than the rectangle allocated to it. The most visible example is a list widget: it may contain hundreds of items but only have room to display a dozen. Scrolling requires cooperation between two widgets: the scrollable widget (e.g., a list) that knows its total content size, and a scroll bar that shows the user where they are and lets them navigate. The scroll bar must communicate with the scrollable widget to say "show me a different portion," and the scrollable widget must tell the scroll bar "I am now showing this portion of the total." This two-way communication is one of the more interesting design problems in a widget library.

2.2 `hellopanel.c`

This section shows a complete program using `libpanel`. Despite its simplicity, it illustrates all four principles from the previous section: it creates widgets (a frame, a label, a button), runs an event loop, triggers a layout pass, and could easily be extended with scrollable widgets. The program structure follows a pattern common to all `libpanel` applications: (1) initialize the graphics system and the library, (2) build a widget tree, (3) perform an initial layout and draw, (4) enter the event loop.

⟨lib_gui/libpanel/tests/hellopanel.c 12⟩≡

```
// source code from panel.pdf introduction
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <event.h>

#include <panel.h>

Panel *root;

⟨function done 13a⟩

⟨function eresized 13c⟩

void main(void){
    errorneg1 err;
    Event e;
    int i;

    err = initdraw(nil, nil, "hellopanel");
    ⟨main() sanity check err 13b⟩
    einit(Emouse);
    plinit(view->depth);

    root=plframe(nil, NOFLAG);
    pllabel(root, NOFLAG, "Hello, world!");
    plbutton(root, NOFLAG, "done", done);

    eresized(false);

    for(;;) {
```

```
        i=event(&e);
        plmouse(root, &e.mouse);
    }
}
```

Uses plbutton() 54a, plframe() 66a, plinit() 26a, pllabel() 44c, and plmouse() 31a.

The main() function first initializes the Plan 9 graphics system with initdraw() and the event library with einit(Emouse), then initializes libpanel with plinit(). Here einit() is told we are only interested in mouse events; we will see later how to also accept keyboard events. Next, it builds a small widget tree: a frame (the root container) with two children, a label displaying "Hello, world!" and a button labeled "done." The first argument to each constructor is the parent widget (nil for the root, root for the children), so the tree is built implicitly by passing parents to constructors. The call to eresized(false) triggers the initial layout and draw. Then the program enters the event loop: event() blocks until a mouse event arrives, and plmouse() dispatches it down the widget tree to the appropriate widget.

⟨*function* done 13a⟩≡ (12)

```
void done(Panel *p, buttons buttons){
    USED(p, buttons);
    exits(nil);
}
```

The done() function is a *callback*: it is passed as an argument to plbutton() above and will be called when the user clicks the button. This is the basic mechanism for widget-to-application communication in libpanel: the application registers a function pointer, and the widget calls it when the appropriate event occurs.

I will not comment much on error handling code in the rest of this book as it is usually self-explanatory; see the appendix for more information on error management.

⟨main() *sanity check* err 13b⟩≡ (12)

```
if(err < 0)
    sysfatal("initdraw: %r");
```

Finally, the eresized() function is the bridge between the windowing system and the widget library. It is called both at startup (from main()) and whenever the user resizes the window (called back by the graphics system, see the WINDOWS book [Pad16c]). Its job is simple but critical: call plpack() to recompute the layout of the entire widget tree within the new window rectangle view->r, then call pldraw() to redraw everything. These two functions—layout then draw—are the heartbeat of libpanel.

An important design principle of libpanel is that *nothing happens automatically*. Unlike Tk, where changing a widget's text triggers an automatic redisplay, libpanel requires the application to explicitly call plpack() and pldraw() whenever something changes. If you reinitialize a label with plinitlabel(), you must call pldraw() to see the result. If you add a new widget to the tree, you must repack and redraw. This makes the library simpler—no observers, no invalidation queues—at the cost of putting more responsibility on the application programmer.

⟨*function* eresized 13c⟩≡ (12)

```
void eresized(bool new){
    ⟨eresized() if new get a new window 13d⟩
    plpack(root, view->r);
    pldraw(root, view);
}
```

Uses pldraw() 28a and plpack() 35.

⟨eresized() *if* new *get a new window* 13d⟩≡ (13c)

```
if(new && getwindow(display, Refnone) == ERROR_NEG1) {
    fprint(STDERR, "getwindow: %r\n");
    exits("getwindow");
}
```

2.3 Code organization

Table 2.1 presents short descriptions of the source files of `libpanel`, together with the main entities (e.g., structures, functions) the file defines, and the corresponding chapter in this document in which the code contained in the file is primarily discussed.

Function	Ch.	File	Entities
core data structures	3	`panel.h`	`Panel`[18] `Icon`[22d]
internal definitions	3	`pldefs.h`	layout flags, method prototypes
initialization	4	`init.c`	`plinit()`[26a]
drawing	5	`draw.c`	`pldraw()`[28a] `pl\_draw()X` `pl\_relief()X`
mouse/keyboard dispatch	6	`event.c`	`plmouse()`[31a] `plkeyboard()`[34a] `plgrabkb()`[33k]
layout algorithm	7	`pack.c`	`plpack()`[35] `pl\_sizereq()X` `pl\_setrect()X`
label widget	8	`label.c`	`Label`[44a] `pllabel()`[44c]
text entry widget	8	`entry.c`	`Entry`[47g] `plentry()`[48b]
button widget	8	`button.c`	`Button`[53a] `plbutton()`[54a]
slider widget	8	`slider.c`	`Slider`[59d] `plslider()`[60b]
canvas widget	8	`canvas.c`	`Canvas`[64a] `plcanvas()`[64b]
frame container	9	`frame.c`	`plframe()`[66a]
group container	9	`group.c`	`plgroup()`[68d]
scrolling glue	10	`scroll.c`	`plscroll()`[71a] `plsetscroll()`[137b]
scroll bar widget	10	`scrollbar.c`	`Scrollbar`[77d] `plscrollbar()`[77f]
scrollable list	10	`list.c`	`List`[71g] `pllist()`[72b]
popup menu	11	`popup.c`	`Popup`[83c] `plpopup()`[84a]
pull-down menu	11	`pulldown.c`	`Pulldown`[133b] `plpulldown()`[88a] `plmenubar()`[91a]
message widget	12	`message.c`	`Message`[92a] `plmessage()`[92b]
edit widget	12	`edit.c`	`Edit`[95a] `pledit()`[95b]
text view widget	12	`textview.c`	`Textview`[100d] `pltextview()`[101a]
text window widget	12	`textwin.c`	
rich text	12	`rtext.c`	`pl\_rtfmt()X` `pl\_rtdraw()X`
copy/paste	13	`snarf.c`	`plsnarf()`[109a] `plpaste()`[109b]
memory management	C	`mem.c`	`pl\_emalloc()X` `pl\_erealloc()X`
debugging output	C	`print.c`	`pl\_print()X`
UTF-8 utilities	C	`utf.c`	`pl\_nextrune()X` `pl\_runewidth()X`
Total			∼4000 LOC

Table 2.1: Chapters and associated `libpanel` source files.

Every source file includes the same set of headers:

⟨`libpanel` *includes* 14⟩≡ (150c 148d 140a 138a 137 136c 133 132 131 130 129 128 127)

```
#include <u.h>
#include <libc.h>
#include <draw.h>
```

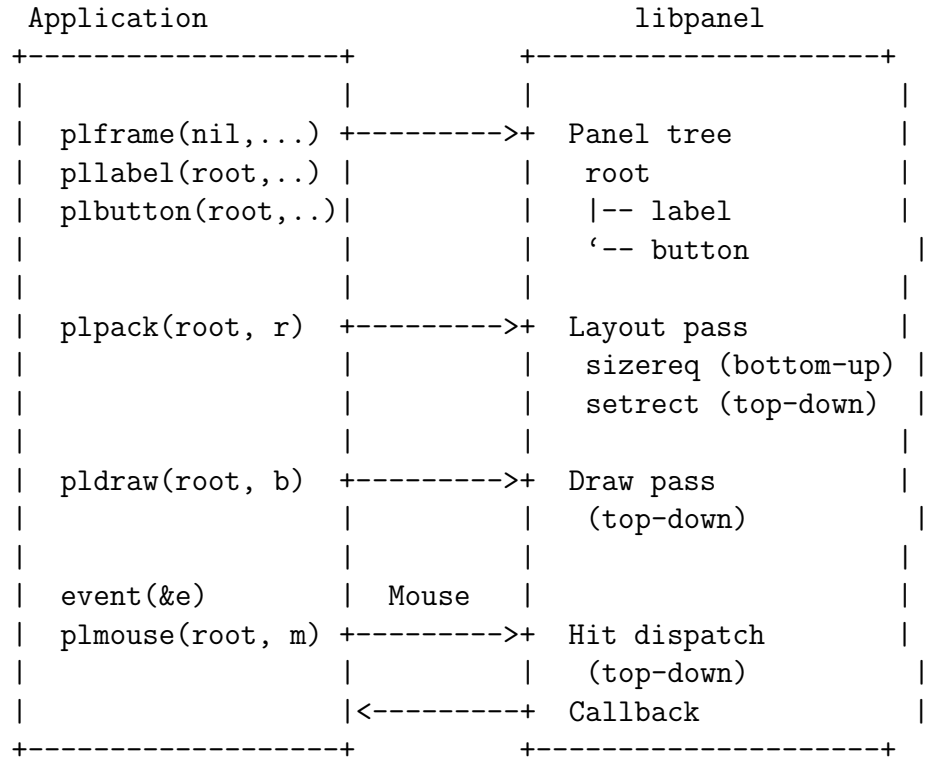


Figure 2.1: Data flow diagram of `libpanel`.

```
#include <event.h>

#include <panel.h>
#include "pldefs.h"
```

The public API is in `panel.h`: this is what application programmers include (as in `hellopanel.c` above). The internal header `pldefs.h` contains the private definitions used by the library implementation—helper functions, internal macros, and the full `Panel` structure with its method pointers.

2.4 Software architecture

Figure 2.1 describes the main data flow of `libpanel`, whereas Figure 2.2 describes the main control flow.

As shown in Figure 2.1, the application builds a tree of `Panel`[18] widgets using constructors like `plframe()`[66a], `pllabel()`[44c], and `plbutton()`[54a]. It then calls `plpack()`[35] to compute the layout and `pldraw()`[28a] to render the tree. During the event loop, mouse events are dispatched into the tree via `plmouse()`[31a], which finds the target widget and triggers its callback.

Figure 2.2 shows the two main flows triggered by the application. The first flow is triggered by `eresized()`[13c] (either at startup or when the window is resized): `plpack()` runs the two-pass layout algorithm—`pl_sizereq()`[36e] walks the tree bottom-up to collect size requests, then `pl_setrect()`[38c] walks top-down to assign rectangles. After layout, `pldraw()` walks the tree top-down, calling `pl_draw()X` on each widget, which dispatches to the type-specific drawing function (e.g., `pl_drawlabel()`[45b], `pl_drawbutton()`[54d]). The second flow is triggered by a mouse event: `plmouse()` calls `pl_hit()X` to find which widget contains the mouse coordinates, then dispatches to the type-specific hit handler (e.g., `pl_hitbutton()`[55b]), which may invoke the application's callback.

`libpanel` uses a consistent naming convention that reveals its architecture. Public functions use the `pl` prefix: `plinit()`[26a], `pldraw()`, `plpack()`, `plmouse()`, `plfree()`[19c]. Widget constructors also use `pl`: `pllabel()`, `plbutton()`, `plframe()`, `pllist()`[72b], etc. Internal functions (not meant for application programmers) use the `pl_` prefix with an underscore: `pl_draw()X` is the internal draw dispatcher, `pl_hit()X` is the internal mouse-hit

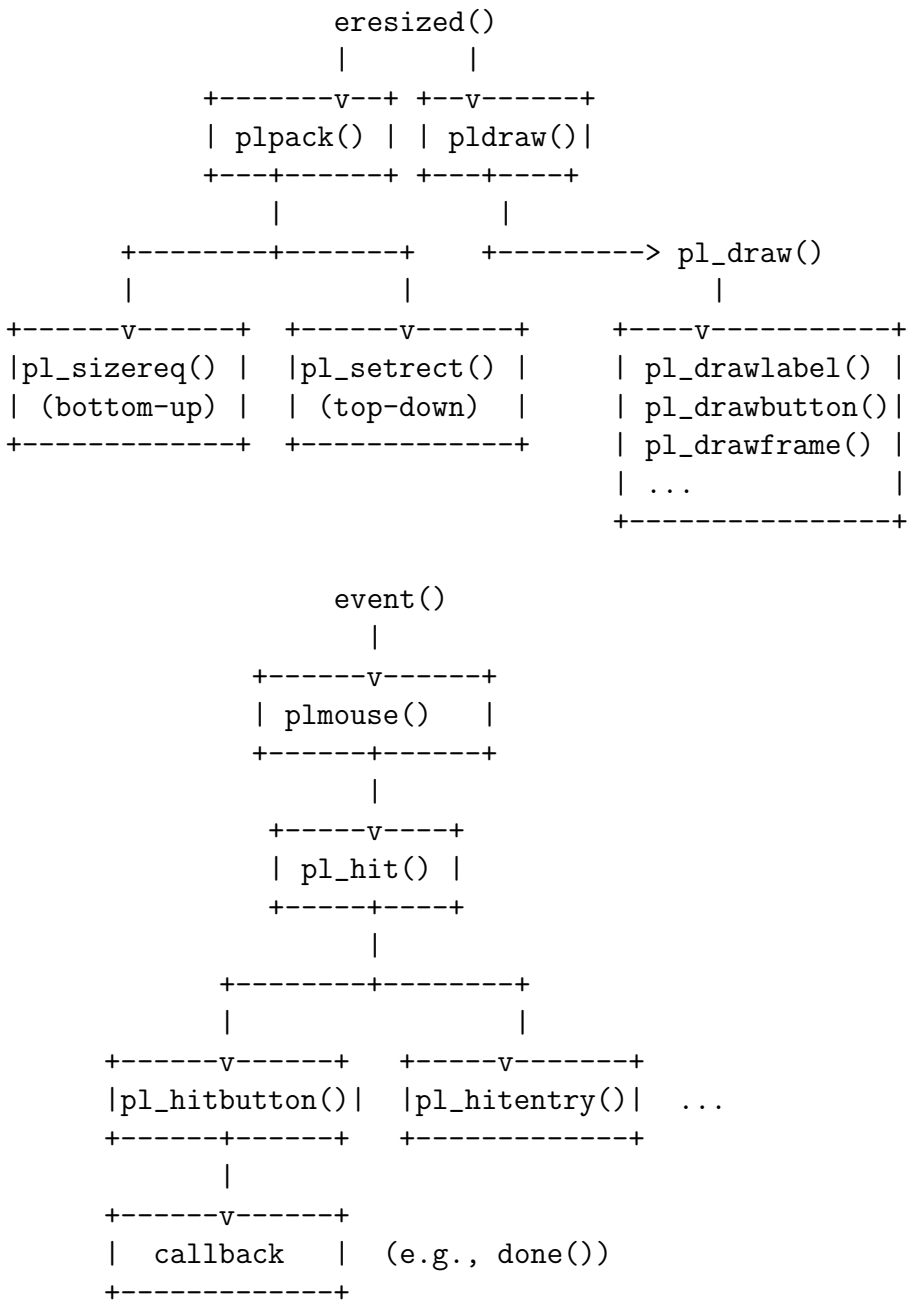


Figure 2.2: Control flow diagram of `libpanel`.

dispatcher, `pl_sizereq()` and `pl_setrect()` are the two layout passes. Each widget type provides its own implementations of these internal methods. For instance, the button widget provides `pl_drawbutton()` and `pl_hitbutton()`. These are stored as function pointers in the `Panel` structure, enabling polymorphic dispatch: calling `pl_draw()X` on any widget automatically calls the right type-specific drawing function.

2.5 Book structure

The rest of this book follows the architecture of `libpanel`. Chapter 3 presents the core data structures, primarily the `Panel` structure and the widget tree. Chapter 4 covers initialization. Chapter 5 covers drawing—how `pldraw()` walks the tree and renders each widget. Chapter 6 covers event handling—how `plmouse()` dispatches mouse events to the right widget. Chapter 7 covers layout—the two-pass algorithm of `plpack()`. The following chapters then go through the individual widget types: basic widgets (label, entry, button, slider, canvas), composite widgets (frame, group), scrollable widgets (list, scroll bar), menu widgets (popup, pull-down, menu bar), and text widgets (message, edit, text view). The final chapters cover advanced topics and the conclusion.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter, I will present the core data structures of `libpanel`: the `Panel`[18] structure (the central widget type), the `Icon`[22d] type, and the layout style constants that control how widgets are arranged within their parent (`PACK`, `PLACE`, `FILL`).

All of these are defined in the `panel.h` header file. The `Panel` structure is the heart of the library — every widget, from a simple label to a complex text editor, is represented by this single structure.

3.1 `Panel`

The `Panel`[18] structure is the universal widget type in `libpanel`. Every widget — whether a label, a button, a text entry, or a scrollbar — is a `Panel`. The overview chunk below shows its overall shape: the two key fields visible here are `Rectangle r`, which defines where the widget is positioned on screen, and `Image *b`, which is the drawing surface. The rest of the fields and methods are defined gradually in the following subsections.

⟨*struct* `Panel` 18⟩≡ (121d)

```
struct Panel{
    // public

    Rectangle r;     /* where the Panel goes */

    ⟨Panel padding fields 25a⟩
    ⟨Panel fixed size field 111d⟩
    ⟨Panel user-specific fields 111a⟩

    /* private below */

    Image *b;     /* where we're drawn */
    ⟨Panel widget-specific data 21d⟩
    ⟨Panel layout fields 36a⟩
    ⟨Panel debugging fields 116a⟩
    ⟨Panel other fields 19d⟩

    // methods
    ⟨Panel main methods 22b⟩
    ⟨Panel packing methods 36f⟩
    ⟨Panel other methods 21e⟩
```

```
    // Extra
    ⟨Panel extra fields 20a⟩
};
```

Note that `Panel.r` (the rectangle) and `Panel.b` (the drawing surface) are not set at construction time — they are filled in later by `plpack()`[35] and `pldraw()`[28a] respectively. This two-phase initialization reflects the separation between layout and drawing described in Chapter 2.

The constructor `pl_newpanel()`[19a] takes a parent widget and an `ndata` size parameter (explained later in Section 3.1.3). It allocates a `Panel`, links it into the parent's child list, and sets safe defaults for all fields.

⟨*function* `pl_newpanel` 19a⟩≡ (127b)
```
Panel *pl_newpanel(Panel *parent, int ndata){
    Panel *v;

    ⟨pl_newpanel() sanity check if can create child on parent 21c⟩
    v=pl_emalloc(sizeof(Panel));

    ⟨pl_newpanel() set tree fields 20e⟩
    ⟨pl_newpanel() set widget-specific data fields 21f⟩
    ⟨pl_newpanel() set other fields 19b⟩

    ⟨pl_newpanel() set default methods 22c⟩

    return v;
}
```
Uses `pl_emalloc()` 120a.

⟨`pl_newpanel()` *set other fields* 19b⟩≡ (19a) 19f▷
```
v->r=Rect(0,0,0,0);
v->b=nil;
```

The destructor `plfree()`[19c] recursively frees a widget and all its children. The chunks it references are defined in the following subsections as each field group is introduced.

⟨*function* `plfree` 19c⟩≡ (127b)
```
void plfree(Panel *p){
    ⟨plfree locals 20f⟩

    if(p==nil)
        return;
    ⟨plfree() if plkbfocus 34b⟩
    ⟨plfree() free the children 21a⟩
    ⟨plfree() free the widget-specific data 21g⟩
    free(p);
}
```

3.1.1 Widget flags

The `flags` field is an `int` used as a bitset. Different bits encode different properties: whether the widget is a leaf, whether it is invisible, and — most importantly — the layout style (see Section 3.3).

⟨`Panel` *other fields* 19d⟩≡ (18) 21h▷
```
// LEAF | INVIS | ...
int flags;     /* position flags, see below */
```

⟨*constant* `NOFLAG` 19e⟩≡ (121d)
```
#define NOFLAG 0
```

⟨`pl_newpanel()` *set other fields* 19f⟩+≡ (19a) ◁19b 25b▷
```
v->flags=NOFLAG;
```

3.1.2 A tree of widgets

Widgets form a tree: each `Panel`[18] can have children, and each child knows its parent. The children are stored as a singly linked list (via `next` pointers), with `child` pointing to the first child and `echild` pointing to the last (for efficient appending). This is the classic child-sibling tree representation: `child` goes "down" to the first child, `next` goes "right" to the next sibling.

```
Example: a frame with three children (label, entry, button)

                +-------+
          .parent--->| frame |<---.parent--------------.parent
          |          +-------+    |                     |
          |  .child |             |              .echild |
          |         v             |                      v
          | +-------+  .next  +-------+  .next  +--------+  .next
          '-| label | ------> | entry | ------> | button | ------> nil
            +-------+         +-------+         +--------+
```

⟨Panel *extra fields* 20a⟩≡ (18) 20b▷
```
// option<list<ref_own<Panel>>> (next = Panel.next, tail = Panel.echild)
Panel *child;
```

⟨Panel *extra fields* 20b⟩+≡ (18) ◁20a 20c▷
```
// list<ref<Panel>> (head = Panel.child)
Panel *next;     /* It's a list! */
```

⟨Panel *extra fields* 20c⟩+≡ (18) ◁20b 20d▷
```
// option<ref<Panel>>
Panel *echild;
```

⟨Panel *extra fields* 20d⟩+≡ (18) ◁20c
```
// option<ref<Panel>>
Panel *parent;    /* No, it's a tree! */
```

Insertion appends the new widget at the end of the parent's child list, using `echild` as a tail pointer to avoid traversing the entire list. This means children are laid out in the order they were created, which is usually the order the programmer wants.

⟨pl_newpanel() *set tree fields* 20e⟩≡ (19a)
```
v->next=nil;
v->child=nil;
v->echild=nil;

v->parent=parent;
//add_list(v, parent->child)
if(parent){
    if(parent->child==nil)
        parent->child=v;
    else
        parent->echild->next=v;
    parent->echild=v;
}
```

⟨plfree *locals* 20f⟩≡ (19c)
```
Panel *cp, *ncp;
```

⟨plfree() *free the children* 21a⟩≡ (19c)
```
for(cp=p->child;cp;cp=ncp){
    ncp=cp->next;
    plfree(cp);
}
```
Uses plfree() 19c.

Leaf widgets (labels, buttons) cannot have children. The LEAF flag enforces this at construction time: pl_newpanel()[19a] refuses to create a child on a leaf widget.

⟨*constant* LEAF 21b⟩≡ (125e)
```
#define LEAF 0x10000  /* newpanel will refuse to attach children */
```

⟨pl_newpanel() *sanity check if can create child on* parent 21c⟩≡ (19a)
```
if(parent && parent->flags&LEAF){
    fprint(STDERR, "newpanel: can't create child of %s %lux\n",
            parent->kind, (ulong)parent);
    exits("bad newpanel");
}
```
Uses LEAF.

3.1.3 Widget-specific data

Each widget type needs its own private state: a label stores a string, a button stores a callback, a text entry stores an editable buffer. Rather than using a C union (which would make Panel[18] grow with every new widget type), libpanel uses a void *data pointer and a separate free method. The constructor allocates ndata bytes for the widget's private structure, and each widget type casts data to its own type (e.g., LabelData *).

⟨Panel *widget-specific data* 21d⟩≡ (18)
```
// union<ref_own<Label|Entry|Button|...>
void *data;     /* kind-specific data */
```

⟨Panel *other methods* 21e⟩≡ (18) 33b▷
```
void (*free)(Panel *);    /* free fields of data when done */
```

⟨pl_newpanel() *set widget-specific data fields* 21f⟩≡ (19a)
```
if(ndata)
    v->data=pl_emalloc(ndata);
else
    v->data=nil;
v->free=nil;
```
Uses pl_emalloc() 120a.

Note the order: the widget's free method is called before free(p->data), because the method may need to access data to release its own resources (e.g., freeing strings stored inside a LabelData).

⟨plfree() *free the widget-specific data* 21g⟩≡ (19c)
```
if(p->free)
    p->free(p);
if(p->data)
    free(p->data);
```

3.1.4 Widget display state

Widgets can be in different visual states — for example, a button looks different when pressed (DOWN) versus released (UP). The state field tracks this so that the widget can be drawn differently depending on whether it is active or not, and so that mouse event handling can toggle between states.

⟨Panel *other fields* 21h⟩+≡ (18) ◁19d 31b▷
```
// enum<Style>
int state;     /* for hitting & drawing purposes */
```

⟨*enum* `Style` 22a⟩≡ (125e)

```
/*
 * States, also styles
 */
enum{
    // for text entries, buttons
    UP,
    DOWN,
    ⟨Style other cases 45c⟩
};
```

3.1.5 Main widget methods

Here is the object-oriented programming in C at the heart of `libpanel`: each widget has three function pointers that act as virtual methods. `draw` paints the widget, `hit` handles mouse events, and `type` handles keyboard events. Each widget constructor (e.g., `pllabel()`[44c], `plbutton()`[54a]) fills in these pointers with its own implementation. The default values are error-raising stubs set by `pl_newpanel()`[19a], so forgetting to initialize them is caught immediately.

⟨`Panel` *main methods* 22b⟩≡ (18)

```
void (*draw)(Panel *);    /* draw panel and children */

bool (*hit)(Panel *, Mouse *);   /* process mouse event */
void (*type)(Panel *, Rune);   /* process keyboard event */
```

⟨`pl_newpanel()` *set default methods* 22c⟩≡ (19a) 37a▷

```
v->draw=pl_drawerror;
v->hit=pl_hiterror;
v->type=pl_typeerror;
```

Uses `pl_drawerror()` 118b, `pl_hiterror()` 118c, and `pl_typeerror()` 118d.

3.2 `Icon`

Many widgets need to display content that could be either text or a bitmap image — a button might show a label string or an icon graphic, and the programmer should not have to use a different widget type for each case. The `Icon`[22d] type solves this with a simple trick: `typedef void` makes `Icon *` a generic pointer that can hold either an `Image *` or a `char *`, with `pl_drawicon()`[46b] dispatching at runtime. This lets widgets like buttons and labels accept either text or image content through the same interface.

⟨*type* `Icon` 22d⟩≡ (121d)

```
typedef void Icon;   /* Always used as Icon * -- Image or char */
```

3.3 Layout styles

Layout in `libpanel` is controlled by three orthogonal sets of flags, all stored in the `flags` field of `Panel`[18]: packing determines which side of the parent a widget is attached to (north, south, east, west), placement determines where within its allocated space the widget sits (centered, or pushed to a corner), and filling determines whether the widget expands to fill available space horizontally or vertically. This is directly inspired by Tk's packer geometry manager (see Chapter 7). The flags are encoded as bit fields in a single `int`, so they can be combined with bitwise OR.

3.3.1 Placement

Placement controls where a widget sits within the space allocated to it by the packer. By default, widgets are centered (PLACECEN). The compass directions (PLACEN, PLACES, PLACEE, PLACEW and corners) push the widget to an edge or corner of its allocated rectangle.

⟨*constant* PLACE 23a⟩≡ (121d)
```
#define PLACE 0x01e0  /* which side of its space should the Panel adhere to? */
```

⟨*constant* PLACECEN 23b⟩≡ (121d)
```
#define  PLACECEN 0x0000
```

⟨*constant* PLACES 23c⟩≡ (121d)
```
#define  PLACES 0x0020
```

⟨*constant* PLACEE 23d⟩≡ (121d)
```
#define  PLACEE 0x0040
```

⟨*constant* PLACEW 23e⟩≡ (121d)
```
#define  PLACEW 0x0060
```

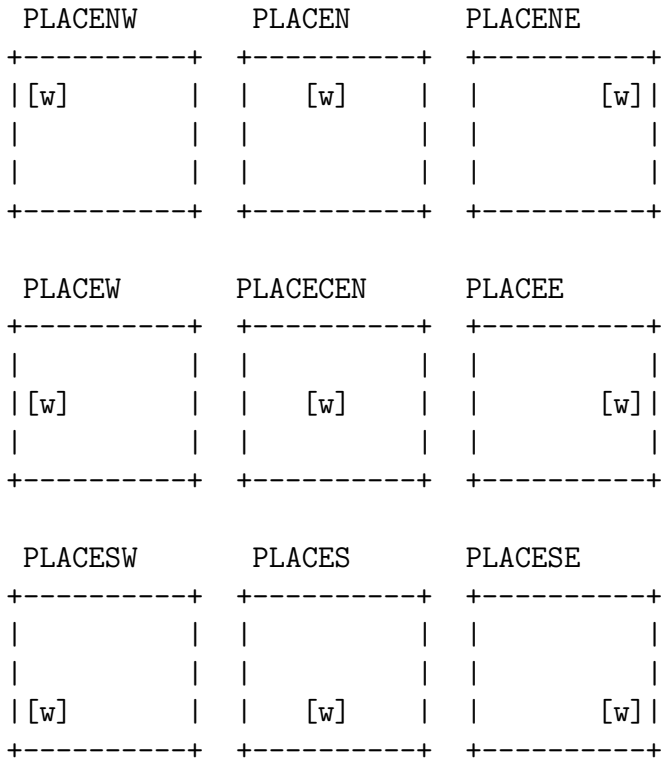
⟨*constant* PLACEN 23f⟩≡ (121d)
```
#define  PLACEN 0x0080
```

⟨*constant* PLACENE 23g⟩≡ (121d)
```
#define  PLACENE 0x00a0
```

⟨*constant* PLACENW 23h⟩≡ (121d)
```
#define  PLACENW 0x00c0
```

⟨*constant* PLACESE 23i⟩≡ (121d)
```
#define  PLACESE 0x00e0
```

⟨*constant* PLACESW 23j⟩≡ (121d)
```
#define  PLACESW 0x0100
```



3.3.2 Packing

Packing determines which side of the parent a widget is attached to and, by extension, how sibling widgets divide the available space. When a widget is packed to the north (`PACKN`), it takes a horizontal strip from the top of the remaining space; when packed to the east (`PACKE`), it takes a vertical strip from the right. This is the same algorithm as Tk's `pack` command.

⟨*constant* PACK 24a⟩≡ (121d)
```
#define PACK 0x0007  /* which side of the parent is the Panel attached to? */
```

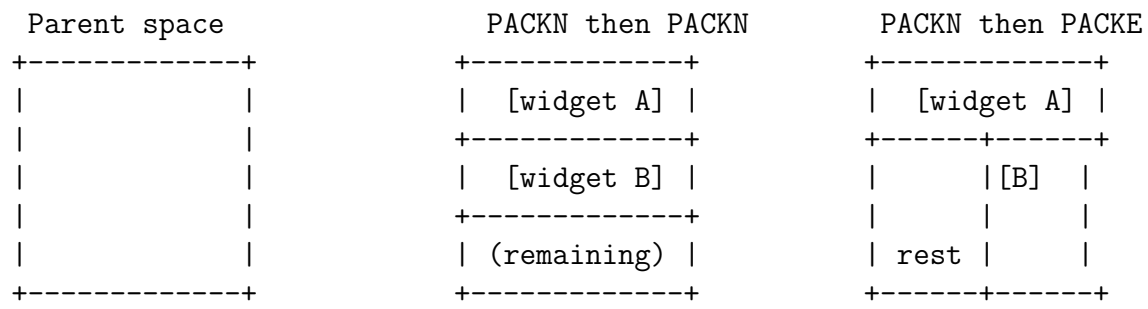
⟨*constant* PACKN 24b⟩≡ (121d)
```
#define  PACKN 0x0000
```

⟨*constant* PACKE 24c⟩≡ (121d)
```
#define  PACKE 0x0001
```

⟨*constant* PACKS 24d⟩≡ (121d)
```
#define  PACKS 0x0002
```

⟨*constant* PACKW 24e⟩≡ (121d)
```
#define  PACKW 0x0003
```

⟨*constant* PACKCEN 24f⟩≡ (121d)
```
#define  PACKCEN 0x0004 /* only used by pulldown */
```



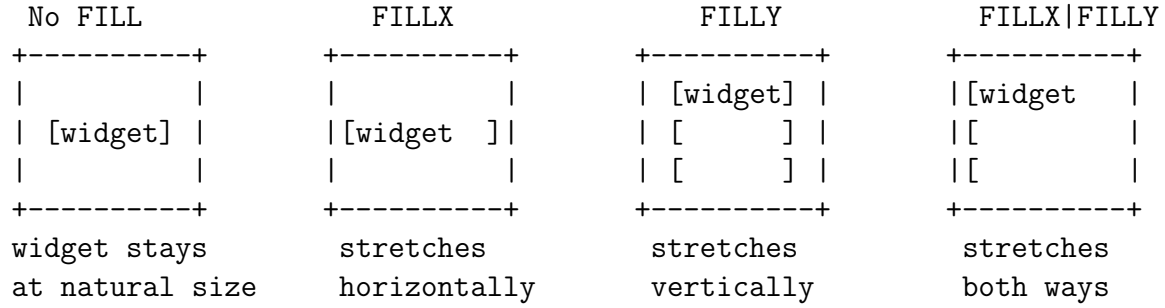
3.3.3 Filling

Filling controls whether a widget grows to fill its allocated space. `FILLX` makes the widget expand horizontally, `FILLY` makes it expand vertically, and both can be combined. Without these flags, a widget only takes up its "natural" size (as reported by `pl_sizereq()`[36e]). `EXPAND` is stronger: it makes the widget claim all *extra* space in the parent, useful for making one widget (like a text area) grow while others (like buttons) stay fixed.

⟨*constant* FILLX 24g⟩≡ (121d)
```
#define FILLX 0x0008  /* grow horizontally to fill the available space */
```

⟨*constant* FILLY 24h⟩≡ (121d)
```
#define FILLY 0x0010  /* grow vertically to fill the available space */
```

⟨*constant* EXPAND 24i⟩≡ (121d)
```
#define EXPAND 0x0200  /* use up all extra space in the parent */
```

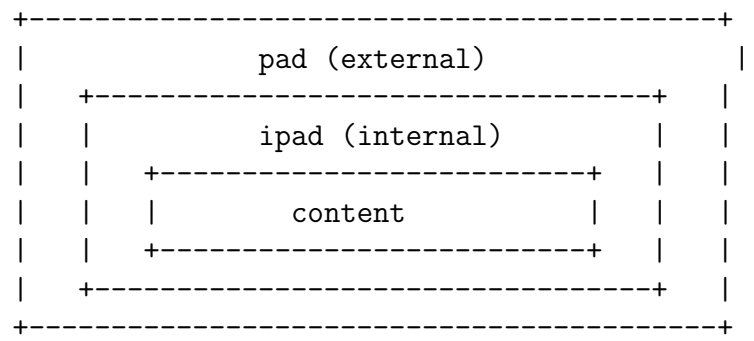


3.3.4 Padding

Padding adds extra space around a widget: `pad` is the external padding (space between this widget and its neighbors), and `ipad` is the internal padding (space between the widget's border and its content). Both are `Vector` (i.e., `Point`) values, giving independent horizontal and vertical padding.

⟨`Panel` *padding fields* 25a⟩≡ (18)
```
Vector ipad;    /* extra space inside and outside */
Vector pad;
```

⟨`pl_newpanel()` *set other fields* 25b⟩+≡ (19a) ◁19f 31c▷
```
v->ipad=Pt(0,0);
v->pad=Pt(0,0);
```



3.4 Direction

Scrollbars and sliders can be oriented horizontally or vertically.

⟨*enum* `Direction` 25c⟩≡ (125e)
```
/*
 * Scrollbar, slider orientations
 */
enum Direction {
    HORIZ,
    VERT
};
```

Chapter 4

Initialization

Before any widget can be drawn, `libpanel` needs to allocate a set of shared color images used throughout the library for drawing borders, backgrounds, and highlights. This is done once at startup by `plinit()`[26a].

4.1 `plinit()`

⟨*function* `plinit` 26a⟩≡ (127a)
```
/*
 * Just a wrapper for all the initialization routines
 */
error0 plinit(int ldepth){
    return pl_drawinit(ldepth);
}
```
Uses `pl_drawinit()` 26d.

The actual work is in `pl_drawinit()`[26d] below, which allocates five shared `Image` values: black, white, dark, light, and a highlight mask. These are used by all widgets for drawing 3D-effect borders (light on top-left, dark on bottom-right) and selection highlights.

⟨*global* `plldepth` 26b⟩≡ (127c)
```
static int plldepth;
```

⟨*globals* `pl_xxx` 26c⟩≡ (127c)
```
static Image *pl_white, *pl_light, *pl_dark, *pl_black, *pl_hilit;
```

⟨*function* `pl_drawinit` 26d⟩≡ (127c)
```
error0 pl_drawinit(int ldepth){

    plldepth=ldepth;

    pl_black=allocimage(display, Rect(0,0,1,1), view->chan, 1, 0x000000FF);
    pl_white=allocimage(display, Rect(0,0,1,1), view->chan, 1, 0xFFFFFFFF);

    pl_dark =allocimage(display, Rect(0,0,1,1), view->chan, 1, DPurpleblue);
    pl_light=allocimagemix(display, DPalebluegreen, DWhite);

    pl_hilit=allocimage(display, Rect(0,0,1,1), CHAN1(CAlpha,8), 1, 0x80);

    if(pl_white==nil || pl_light==nil || pl_black==nil || pl_dark==nil)
        return ERROR_0;
    return OK_1;
}
```
Uses `pl_black-26` 26c, `pl_dark-25` 26c, `pl_hilit-27` 26c, `pl_light-24` 26c, `pl_white-23` 26c, and `plldepth-22` 26b.

Note `pl_hilit`: it is not a color but an alpha mask (50% transparency), used to draw translucent highlights over selected items in lists.

These five images are allocated once and reused by every widget. The 3D relief effect that gives buttons and frames their "raised" or "sunken" appearance is simply `pl_light` on the top-left edges and `pl_dark` on the bottom-right (or reversed, for a sunken look). This is the same technique used in Windows 95, Motif, and early GTK widgets.

Chapter 5

Drawing

Drawing in `libpanel` is a recursive walk down the widget tree: starting from the root, each widget draws itself using its `draw` method, then its children are drawn on top. The rest of this chapter covers the shared drawing helpers that widgets use to render borders and boxes.

5.1 `pldraw()`

The entry point `pldraw()`[28a] takes the root widget and the destination `Image *b` (typically `view`, the screen). It delegates to `pl_draw1()`[28b] and then flushes the display to make the result visible.

⟨*function* `pldraw` 28a⟩≡ (127c)
```
void pldraw(Panel *p, Image *b){
    pl_draw1(p, b);
    flushimage(display, true);
}
```
Uses `pl_draw1()` 28b.

The `flushimage()` call and the `display` global are part of Plan 9's graphics system, explained in the Graphics book [Pad16b].

`pl_draw1()` is the version without the final `flushimage()`, useful when a `hit` callback needs to redraw a sibling widget (e.g., updating a label after a slider moves) without flushing twice.

⟨*function* `pl_draw1` 28b⟩≡ (127c)
```
void pl_draw1(Panel *p, Image *b){
    if(b!=nil)
        pl_drawall(p, b);
}
```
Uses `pl_drawall()` 28c.

`pl_drawall()`[28c] is the recursive core: it stores the drawing surface in `p->b`, calls the widget's `draw` method, then recurses on each child. Every widget in the tree ends up sharing the same `Image *b` (the screen), drawing into its own `p->r` rectangle within it.

⟨*function* `pl_drawall` 28c⟩≡ (127c)
```
void pl_drawall(Panel *p, Image *b){
    ⟨pl_drawall() if invisible widget 87c⟩
    p->b=b;
    // widget-specific method
    p->draw(p);
    for(p=p->child;p;p=p->next)
        // recurse
        pl_draw1(p, b);
}
```
Uses `pl_draw1()` 28b.

5.2 Drawing helpers

`pl_relief()`[29a] draws a 3D relief border around a rectangle using two colors: `ul` (upper-left edges) and `lr` (lower-right edges). When `ul` is light and `lr` is dark, the widget appears raised; swapping them makes it appear sunken. The nested loop fills the diagonal corner pixels where the horizontal and vertical edges meet.

⟨*function* `pl_relief` 29a⟩≡ (127c)
```
void pl_relief(Image *b, Image *ul, Image *lr, Rectangle r, int wid){
    int x, y;

    draw(b, Rect(r.min.x, r.max.y-wid, r.max.x, r.max.y), lr, 0, ZP);/* bottom*/
    draw(b, Rect(r.max.x-wid, r.min.y, r.max.x, r.max.y), lr, 0, ZP);/* right */
    draw(b, Rect(r.min.x, r.min.y, r.min.x+wid, r.max.y), ul, 0, ZP);/* left */
    draw(b, Rect(r.min.x, r.min.y, r.max.x, r.min.y+wid), ul, 0, ZP);/* top */

    for(x=0;x!=wid;x++)
      for(y=wid-1-x;y!=wid;y++){
        draw(b, rectaddpt(Rect(0,0,1,1), Pt(x+r.max.x-wid, y+r.min.y)), lr, 0, ZP);
        draw(b, rectaddpt(Rect(0,0,1,1), Pt(x+r.min.x, y+r.max.y-wid)), lr, 0, ZP);
    }
}
```

The four `draw()` calls paint the four edges of the border:

```
r.min.x          r.max.x
r.min.y +--ul--ul--ul--+
        |ul            |lr
        |ul            |lr
        |ul            |lr
r.max.y +--lr--lr--lr--+
        |<--- wid --->|

        ul = upper-left color (top + left edges)
        lr = lower-right color (bottom + right edges)
```

The nested loop fills the two diagonal corners (top-right and bottom-left) where the `ul` and `lr` edges meet, painting them with `lr` to create a clean mitered join.

5.3 Drawing boxes

Widgets draw their background and border using `pl_box()`[29c] (filled) or `pl_outline()`[29b] (border only). Both dispatch to `pl_boxoutline()`[30a], which uses the widget's `state` (`UP`, `DOWN`, `PASSIVE`, `FRAME`) to choose the appropriate relief colors and border widths. The returned rectangle is the interior area after subtracting the border, which is where the widget should draw its content.

⟨*function* `pl_outline` 29b⟩≡ (127c)
```
Rectangle pl_outline(Image *b, Rectangle r, int style){
    return pl_boxoutline(b, r, style, false);
}
```
Uses `pl_boxoutline()` 30a.

⟨*function* `pl_box` 29c⟩≡ (127c)
```
Rectangle pl_box(Image *b, Rectangle r, int style){
    return pl_boxoutline(b, r, style, true);
}
```
Uses `pl_boxoutline()` 30a.

⟨*function* `pl_boxoutline` 30a⟩≡ (127c)
```
Rectangle pl_boxoutline(Image *b, Rectangle r, int style, bool fill){
    ⟨pl_boxoutline() if plldepth is zero ??⟩
    else
    switch(style){
    ⟨pl_boxoutline() switch style cases 45d⟩
    }
    return insetrect(r, SPACE);
}
```
Uses SPACE-16 30b.

After the relief border, an extra SPACE pixel of padding is subtracted from each side, giving the content a small margin inside the border.

⟨*constant* `SPACE` 30b⟩≡ (127c)
```
#define SPACE 1 /* space inside relief of button or frame */
```

`pl_boxsize()`[30c] is the inverse: given an interior size and a state, it returns how much extra space the border adds. `pl_interior()`[30d] adjusts a point and size inward to account for the border, used when a widget needs to know where to place its content.

⟨*function* `pl_boxsize` 30c⟩≡ (127c)
```
Vector pl_boxsize(Vector interior, int state){
    switch(state){
    ⟨pl_boxsize() switch state cases 47e⟩
    }
    // else
    return Pt(0, 0);
}
```

⟨*function* `pl_interior` 30d⟩≡ (127c)
```
void pl_interior(int state, Point *ul, Vector *size){
    switch(state){
    ⟨pl_interior() switch state cases 30e⟩
    }
}
```

⟨`pl_interior()` *switch state cases* 30e⟩≡ (30d) 30f▷
```
case UP:
case DOWN:
case DOWN1:
case DOWN2:
case DOWN3:
    *ul=addpt(*ul, Pt(BWID+SPACE, BWID+SPACE));
    *size=subpt(*size, Pt(2*(BWID+SPACE), 2*(BWID+SPACE)));
    break;
```
Uses BWID-14 49e, DOWN 45c, DOWN1 22a, DOWN2 22a, DOWN3, SPACE-16 30b, and UP 22a.

⟨`pl_interior()` *switch state cases* 30f⟩+≡ (30d) ◁30e 68c▷
```
case PASSIVE:
    *ul=addpt(*ul, Pt(PWID+SPACE, PWID+SPACE));
    *size=subpt(*size, Pt(2*(PWID+SPACE), 2*(PWID+SPACE)));
    break;
```
Uses PASSIVE 45c, PWID-13 46a, and SPACE-16 30b.

Chapter 6

Events

Event dispatch is the other half of the widget library (drawing being the first). The application's event loop receives raw mouse and keyboard events from Plan 9's event system, then calls `plmouse()`[31a] or `plkeyboard()`[34a] to route them to the right widget.

6.1 Mouse events

`plmouse()`[31a] is called from the application's event loop with the root widget and a `Mouse` event. It finds which leaf widget contains the mouse position using `pl_ptinpanel()`[32a], then calls that widget's `hit` method. It also tracks which widget received the last event (`lastmouse`) so it can send an `OUT` notification when the mouse moves to a different widget.

⟨*function* `plmouse` 31a⟩≡ (128a)
```
void plmouse(Panel *g, Mouse *m){
    Panel* hit;
    Panel* last = g->lastmouse;
    bool remouse;

    ⟨plmouse() if REMOUSE set hit to last 33a⟩
    else{
        hit=pl_ptinpanel(m->xy, g);
        if(last && last!=hit){
            ⟨plmouse() when last!=hit send OUT mouse event 32c⟩
        }
    }
    if(hit){
        // widget-specific method
        remouse=hit->hit(hit, m);
        ⟨plmouse() handle remouse 32e⟩
        g->lastmouse=hit;
    }
    flushimage(display, true);
}
```
Uses `pl_ptinpanel()` 32a.

⟨`Panel` *other fields* 31b⟩+≡ (18) ◁21h 70a▷
```
Panel *lastmouse;    /* who got the last mouse event? */
```

⟨`pl_newpanel()` *set other fields* 31c⟩+≡ (19a) ◁25b 33f▷
```
v->lastmouse=nil;
```

`pl_ptinpanel()` walks the widget tree to find the deepest (most leafward) widget whose rectangle contains the mouse point. When a child and its parent both contain the point, the one with higher priority wins — this is how popup menus can intercept events even when they overlap other widgets.

⟨*function* `pl_ptinpanel` 32a⟩≡ (128a)
```
/*
 * Return the most leafward, highest priority panel containing p
 */
Panel *pl_ptinpanel(Point p, Panel *g){
    Panel *v;

    for(;g;g=g->next)
      if(ptinrect(p, g->r)){
        //recurse
        v=pl_ptinpanel(p, g->child);

        if(v && v->pri(v, p) >= g->pri(g, p))
            return v;
        else
            return g;
      }
    return nil;
}
```
Uses `pl_ptinpanel()` 32a.

6.1.1 Mouse leaving the widget

When the mouse moves from one widget to another, the old widget needs to know — for example, a button should redraw itself in the `UP` state. Rather than adding a separate "mouse leave" method, `libpanel` reuses the `hit` method with an extra `OUT` bit set in `Mouse.buttons`. Since Plan 9 mice have three buttons (bits 0–2), bit 3 is free.

⟨*constant* `OUT` 32b⟩≡ (121d)
```
/*
 * An extra bit in Mouse.buttons
 */
#define OUT 8   /* Mouse.buttons bit, set when mouse leaves Panel */
```

⟨`plmouse()` *when* `last!=hit` *send* `OUT` *mouse event* 32c⟩≡ (31a)
```
m->buttons|=OUT;
last->hit(last, m);
m->buttons&=~OUT;
```

6.1.2 REMOUSE

Sometimes a widget needs to keep receiving mouse events even after the cursor leaves its rectangle — for example, when dragging a scrollbar thumb. The `hit` method signals this by returning `true`, which sets the `REMOUSE` flag on the root widget. On the next event, `plmouse()`[31a] skips `pl_ptinpanel()`[32a] and sends the event directly to the same widget.

⟨*constant* `REMOUSE` 32d⟩≡ (125e)
```
#define REMOUSE 0x40000  /* send next mouse event here, even if not inside */
```

⟨`plmouse()` *handle* `remouse` 32e⟩≡ (31a)
```
if(remouse)
    g->flags|=REMOUSE;
else
    g->flags&=~REMOUSE;
```
Uses `REMOUSE`.

⟨plmouse() *if* REMOUSE *set* hit *to* last 33a⟩≡ (31a)
```
if(g->flags&REMOUSE)
    hit=last;
```
Uses REMOUSE.

6.1.3 Hitting priority

The pri method is another virtual method on Panel[18], returning an integer priority for hit testing. Most widgets use PRI_NORMAL (0). Popup menus use PRI_POPUP (1) so they receive events even when overlapping lower-priority children. Scrollbars use PRI_SCROLLBAR (2) because they sit at the edge of their parent and must win over the content area.

⟨Panel *other methods* 33b⟩+≡ (18) ◁21e 71e▷
```
int (*pri)(Panel *, Point);   /* priority for hitting */
```

⟨*constant* PRI_NORMAL 33c⟩≡ (121d)
```
#define PRI_NORMAL 0  /* ordinary panels */
```

⟨*constant* PRI_POPUP 33d⟩≡ (121d)
```
#define PRI_POPUP 1  /* popup menus */
```

⟨*constant* PRI_SCROLLBAR 33e⟩≡ (121d)
```
#define PRI_SCROLLBAR 2  /* scroll bars */
```

⟨pl_newpanel() *set other fields* 33f⟩+≡ (19a) ◁31c 36b▷
```
v->pri=pl_prinormal;
```
Uses pl_prinormal() 33g.

⟨*function* pl_prinormal 33g⟩≡ (127b)
```
int pl_prinormal(Panel *, Point){
    return PRI_NORMAL;
}
```

⟨*function* pl_pripopup 33h⟩≡ (133a)
```
int pl_pripopup(Panel *, Point){
    return PRI_POPUP;
}
```

⟨*function* pl_priscrollbar 33i⟩≡ (137d)
```
int pl_priscrollbar(Panel *, Point){
    return PRI_SCROLLBAR;
}
```

6.2 Keyboard events

Keyboard events are simpler than mouse events: there is a single global keyboard focus widget (plkbfocus), and all keyboard events go to it. Focus is set implicitly by mouse interaction — only widgets that accept text input (text entries, text editors) call plgrabkb()[33k] in their hit method to claim the focus. Widgets that do not handle keyboard input (labels, buttons, scrollbars) simply never call it.

⟨*global* plkbfocus 33j⟩≡ (121d)
```
Panel *plkbfocus;   /* the panel in keyboard focus */
```

⟨*function* plgrabkb 33k⟩≡ (128a)
```
void plgrabkb(Panel *g){
    plkbfocus=g;
}
```

`plkeyboard()`[34a] is called from the application's event loop with a `Rune` (a Unicode character, see LIBCORE book [Pad16a]). If a widget has the focus, its `type` method is called. The `flushimage()` afterwards ensures that any visual changes (e.g., a new character appearing in a text entry) are immediately displayed.

⟨*function* `plkeyboard` 34a⟩≡ (128a)
```
void plkeyboard(Rune c){
    if(plkbfocus){
        // widget-specific callback
        plkbfocus->type(plkbfocus, c);
        flushimage(display, true);
    }
}
```

When a widget is freed, the focus must be cleared if it pointed to that widget, otherwise `plkeyboard()` would dereference a dangling pointer.

⟨`plfree()` *if* `plkbfocus` 34b⟩≡ (19c)
```
if(p==plkbfocus)
    plkbfocus=nil;
```

We have now seen the three core operations: drawing (top-down tree walk), mouse dispatch (top-down hit test), and keyboard dispatch (direct-to-focus). The remaining piece of the infrastructure is layout: how does `libpanel` decide where each widget goes?

Chapter 7

Layout

Layout is a two-pass process. The first pass, `pl_sizereq()`[36e], walks the tree bottom-up: each widget computes how much space it needs, based on its content and its children's requests. The second pass, `pl_setrect()`[38c], walks the tree top-down: starting from the available window rectangle, each widget is assigned an actual position and size, and the remaining space is divided among its children according to the packing flags.

The entry point `plpack()`[35] runs both passes in sequence. It is called from `eresized()` with `view->r` as the available rectangle.

⟨*function* `plpack` 35⟩≡ (132c)

```
void plpack(Panel *p, Rectangle where){
    pl_sizereq(p);
    pl_setrect(p, where.min, subpt(where.max, where.min));
}
```

Uses `pl_setrect()` 38c and `pl_sizereq()` 36e.

Although `plpack()` is usually called on the root with `view->r`, it can also be called on a subtree that has already been packed. This is useful when a subtree has been modified (e.g., children added or removed) and the program wants to recompute just that portion of the layout without repacking the entire tree. In that case, the `Rectangle` argument should be the subtree's current rectangle.

Note that `pl_setrect()` takes a point and a size vector (`where.min` and `subpt(where.max, where.min)`) rather than a `Rectangle`. This is because the recursive algorithm needs to manipulate the origin (`ul`) and available space (`avail`) independently — advancing `ul` and shrinking `avail` as each child is packed — which is more natural with separate point/vector arithmetic than with rectangle operations.

7.1 Layout fields

Each widget stores three size-related vectors. `sizereq` is the total space the widget requests (children + own decoration + padding). `childreq` is the combined size of all children (computed by `pl_sizesibs()`[37b]). `size` is the final allocated size after the top-down pass adjusts for available space, filling, and expansion.

```
sizereq (bottom-up, what we want):
+-------------------------------------+
| pad                                 |
|  +--------------------------------+ |
|  | ipad                           | |
|  |  +--getsize(childreq)-------+  | |
|  |  | decoration (e.g. border) |  | |
|  |  |  +----childreq--------+  |  | |
|  |  |  | children combined  |  |  | |
|  |  |  +--------------------+  |  | |
```

```
|  |  +------ = sizereq ---------+  |  |
|  |        before padding          |  |
|  +--------------------------------+  |
+---------------------------------------+

size (top-down, what we get):
- starts from sizereq
- minus pad (external padding not included)
- clamped to avail (if not enough space)
- expanded to avail (if FILLX/FILLY set)
```

⟨Panel *layout fields* 36a⟩≡ (18) 36c▷
```
Vector sizereq;     /* size requested by this Panel */
Vector childreq;     /* total size needed by children */
```

⟨pl_newpanel() *set other fields* 36b⟩+≡ (19a) ◁33f 36d▷
```
v->sizereq=Pt(0,0);
```

⟨Panel *layout fields* 36c⟩+≡ (18) ◁36a
```
Vector size;     /* space for this Panel */
```

⟨pl_newpanel() *set other fields* 36d⟩+≡ (19a) ◁36b 70b▷
```
v->size=Pt(0,0);
```

7.2 Computing sizes: pl_sizereq()

The bottom-up pass first recurses into all children, then computes childreq by combining the children's sizes according to their packing direction (via pl_sizesibs()[37b]), and finally asks the widget itself how much space it needs around its children (via the getsize method). Padding is added last.

⟨*function* pl_sizereq 36e⟩≡ (132c)
```
/*
 * Compute the requested size of p and its descendants.
 */
void pl_sizereq(Panel *p){
    Panel *cp;
    ⟨pl_sizeref() other locals 112a⟩

    for(cp=p->child;cp;cp=cp->next){
        // recurse
        pl_sizereq(cp);
        ⟨pl_sizeref() when looping over children, adjust maxsize 112b⟩
    }
    ⟨pl_sizereq() if MAXX or MAXY 112c⟩
    p->childreq=pl_sizesibs(p->child);
    // widget-specific method
    p->sizereq=p->getsize(p, p->childreq);
    ⟨pl_sizereq() adjust size with padding 38b⟩
    ⟨pl_sizereq() if FIXEDX or FIXEDY 111e⟩
}
```
Uses pl_sizereq() 36e and pl_sizesibs() 37b.

⟨Panel *packing methods* 36f⟩≡ (18) 41a▷
```
Vector (*getsize)(Panel *, Vector);  /* return size, given child size */
```

⟨pl_newpanel() *set default methods* 37a⟩+≡ (19a) ◁22c 41b▷
```
v->getsize=pl_getsizeerror;
```
Uses pl_getsizeerror() 118e.

```
pl_sizereq(parent):

1. recurse into children        2. combine children     3. add own decoration
                                   (pl_sizesibs)           (getsize + padding)

child A: sizereq = 40x20       childreq = 40x50        sizereq = 44x56
child B: sizereq = 30x30           (PACKN: max width,      (+2 border
            ^                        sum heights)            +2 ipad + pad)
            |
       (recursed first)
```

7.2.1 Packing

pl_sizesibs()[37b] combines sibling sizes according to their packing direction. Siblings packed north/south stack vertically: their widths are maxed and their heights are summed. Siblings packed east/west stack horizontally: the opposite. This is a recursive walk over the sibling list.

⟨*function* pl_sizesibs 37b⟩≡ (132c)
```
Vector pl_sizesibs(Panel *p){
    Vector s;

    ⟨pl_sizesibs() if no panel 37c⟩
    // recurse
    s=pl_sizesibs(p->next);

    switch(p->flags&PACK){
    ⟨pl_sizesibs() switch packing flags cases 37d⟩
    }
    return s;
}
```
Uses pl_sizesibs() 37b.

⟨pl_sizesibs() *if no panel* 37c⟩≡ (37b)
```
if(p==nil)
    return Pt(0,0);
```

⟨pl_sizesibs() *switch packing flags cases* 37d⟩≡ (37b) 37e▷
```
case PACKN:
case PACKS:
    s.x=pl_max(s.x, p->sizereq.x);
    s.y+=p->sizereq.y;
    break;
```
Uses pl_max() 38a.

⟨pl_sizesibs() *switch packing flags cases* 37e⟩+≡ (37b) ◁37d
```
case PACKE:
case PACKW:
    s.x+=p->sizereq.x;
    s.y=pl_max(s.y, p->sizereq.y);
    break;
```
Uses pl_max() 38a.

⟨*function* `pl_max` 38a⟩≡ (132c)
```
int pl_max(int a, int b){
    return a>b ? a : b;
}
```

```
PACKN/PACKS (vertical stacking)     PACKE/PACKW (horizontal stacking)

+--------+  width = max(A,B)        +----+----+  width = A.x + B.x
|   A    |  height = A.y + B.y      |    |    |  height = max(A,B)
+--------+                          | A  | B  |
|   B    |                          |    |    |
+--------+                          +----+----+
```

7.2.2 Padding

⟨`pl_sizereq()` *adjust size with padding* 38b⟩≡ (36e)
```
p->sizereq=addpt(addpt(p->sizereq, p->ipad), p->pad);
```

```
sizereq after getsize():      sizereq after padding:
+------------------+          +------------------------+
|  widget content  |          | pad                    |
|  + children      |    -->   |  +------------------+  |
+------------------+          |  | ipad             |  |
  getsize(childreq)           |  |  +-----------+   |  |
                              |  |  | content   |   |  |
                              |  |  +-----------+   |  |
                              |  +------------------+  |
                              +------------------------+
```

7.3 Computing rectangles: `pl_setrect()`

The top-down pass takes an upper-left point (`ul`) and available space (`avail`) and assigns `p->r`. The steps are: start from the requested size, subtract external padding, clamp to available space, expand if `FILLX`/`FILLY` flags are set, adjust `ul` for placement, set `p->r`, then recurse into children with the remaining interior space.

⟨*function* `pl_setrect` 38c⟩≡ (132c)
```
/*
 * Set the sizes and rectangles of p and its descendants, given their requested sizes.
 */
void pl_setrect(Panel *p, Point ul, Vector avail){
    ⟨pl_setrect() locals 40b⟩

    // setting p->size
    p->size=p->sizereq;
    ⟨pl_setrect() reduce size for external padding 39a⟩
    ⟨pl_setrect() reduce size if bigger than available 39b⟩
    ⟨pl_setrect() expand size if fill or expand flags 39c⟩

    // setting ul
    ⟨pl_setrect() adjust origin with placement 40a⟩

    // setting p->r
    p->r=Rpt(ul, addpt(ul, p->size));
```

```
    // recurse
    ⟨pl_setrect() setting the rectangle of the children 40c⟩
}
```

```
pl_setrect(p, ul, avail):

ul                          ul (adjusted)
+--avail-----------+        +--avail-----------+
|                  |        | pad/2            |
|                  |   -->  |  +--size------+  |
|                  |        |  |  p->r      |  |
|                  |        |  +------------+  |
+------------------+        +------------------+
                            then recurse into children
                            within p->r interior
```

7.3.1 Padding

⟨pl_setrect() *reduce size for external padding* 39a⟩≡ (38c)

```
p->size=subpt(p->sizereq, p->pad);
ul=addpt(ul, divpt(p->pad, 2));
avail=subpt(avail, p->pad);
```

```
Before padding:               After padding:
ul                            ul' = ul + pad/2
+----avail----------+         +----avail'-------+
|                   |         |  +--size------+ |
|  size = sizereq   |   -->   |  |            | |
|                   |         |  +------------+ |
+-------------------+         +-----------------+
                                size = sizereq - pad
                                avail' = avail - pad
```

7.3.2 Filling

⟨pl_setrect() *reduce size if bigger than available* 39b⟩≡ (38c)

```
if(p->size.x>avail.x)
    p->size.x = avail.x;
if(p->size.y>avail.y)
    p->size.y = avail.y;
```

⟨pl_setrect() *expand size if fill or expand flags* 39c⟩≡ (38c)

```
if(p->flags&(FILLX|EXPAND))
    p->size.x=avail.x;
if(p->flags&(FILLY|EXPAND))
    p->size.y=avail.y;
```

```
No flags:            FILLX:              FILLX|FILLY:
+----avail------+    +----avail------+   +----avail------+
|               |    |               |   |[size=========]|
|  [size]       |    |[size=========]|   |[              ]|
|               |    |               |   |[              ]|
+---------------+    +---------------+   +---------------+
size unchanged       size.x = avail.x    size = avail
```

7.3.3 Placing

⟨pl_setrect() *adjust origin with placement* 40a⟩≡ (38c)
```
switch(p->flags&PLACE){
case PLACECEN: ul.x+=(avail.x-p->size.x)/2; ul.y+=(avail.y-p->size.y)/2; break;
case PLACES: ul.x+=(avail.x-p->size.x)/2; ul.y+= avail.y-p->size.y   ; break;
case PLACEE: ul.x+= avail.x-p->size.x   ; ul.y+=(avail.y-p->size.y)/2; break;
case PLACEW:                              ul.y+=(avail.y-p->size.y)/2; break;
case PLACEN: ul.x+=(avail.x-p->size.x)/2;                              break;
case PLACENE: ul.x+= avail.x-p->size.x   ;                              break;
case PLACENW:   /** Nothing **/                                          break;
case PLACESE: ul.x+= avail.x-p->size.x   ; ul.y+= avail.y-p->size.y   ; break;
case PLACESW:                                 ul.y+= avail.y-p->size.y   ; break;
}
```

```
PACKN: child A takes top strip, then B takes next strip

ul                          newul
+----space----------+       +----newspace------+
|  child A (PACKN)  |       |  child B         |
+-------------------+   -->  |  ...             |
|  remaining space  |       +------------------+
+-------------------+
   avail for A:                space shrinks by A's height
   (space.x, A.sizereq.y)
```

7.3.4 Packing the children

⟨pl_setrect() *locals* 40b⟩≡ (38c) 42a▷
```
Panel *c;
Vector space;
Point newul;
Vector newspace;
```

The child loop allocates space from the parent's interior. For each child, based on its packing direction, it carves out a strip (horizontal or vertical) for the child and recurses, then advances `ul` and shrinks `space` for the next sibling. The `childspace` method lets the widget adjust the starting point and available space (e.g., a frame shrinks the interior to account for its border).

⟨pl_setrect() *setting the rectangle of the children* 40c⟩≡ (38c)
```
space=p->size;
// widget-specific adjustment method
p->childspace(p, &ul, &space);

⟨pl_setrect() before looping over children, set locals 42b⟩
for(c=p->child;c;c=c->next){
    ⟨pl_setrect() when looping over children, if EXPAND flag 42c⟩
    switch(c->flags&PACK){
    ⟨pl_setrect() when looping over children, switch packing cases 41c⟩
    }
    ul=newul;
    space=newspace;
}
```

```
Without childspace:          With childspace (frame):
ul                           ul' (shifted inward)
```

```
+----space----------+          +----space----------+
|                   |          | frame border      |
|  children fill    |          |  +--space'------+ |
|  entire space     |          |  | children     | |
|                   |          |  +--------------+ |
+-------------------+          +-------------------+
```

⟨Panel *packing methods* 41a⟩+≡ (18) ◁36f
```
/* child ul & size given our size */
void (*childspace)(Panel *, Point * /**INOUT**/, Vector * /**INOUT**/);
```

⟨pl_newpanel() *set default methods* 41b⟩+≡ (19a) ◁37a 71f▷
```
v->childspace=pl_childspaceerror;
```
Uses pl_childspaceerror() 118f.

```
PACKN: child at top              PACKS: child at bottom
ul                               ul (unchanged = newul)
+----space.x--------+            +----space.x--------+
|  child (at ul)    |            |                    |
+-------------------+ <- newul   |  remaining space   |
|  remaining space  |            +--------------------+
|  (newspace)       |            |  child (at bottom)|
+-------------------+            +--------------------+
```

⟨pl_setrect() *when looping over children, switch packing cases* 41c⟩≡ (40c) 41d▷
```
case PACKN:
    newul=Pt(ul.x, ul.y+c->sizereq.y);
    newspace=Pt(space.x, space.y-c->sizereq.y);
    pl_setrect(c, ul, Pt(space.x, c->sizereq.y));
    break;
case PACKS:
    newul=ul;
    newspace=Pt(space.x, space.y-c->sizereq.y);
    pl_setrect(c, Pt(ul.x, ul.y+space.y-c->sizereq.y),
        Pt(space.x, c->sizereq.y));
    break;
```
Uses pl_setrect() 38c.

```
PACKW: child at left             PACKE: child at right
ul         newul                 ul (unchanged = newul)
+----+-------------+             +-------------+----+
|    |             |             |             |    |
| ch |  remaining  |             |  remaining  | ch |
|    |  (newspace) |             |  (newspace) |    |
+----+-------------+             +-------------+----+
```

⟨pl_setrect() *when looping over children, switch packing cases* 41d⟩+≡ (40c) ◁41c
```
case PACKW:
    newul=Pt(ul.x+c->sizereq.x, ul.y);
    newspace=Pt(space.x-c->sizereq.x, space.y);
    pl_setrect(c, ul, Pt(c->sizereq.x, space.y));
    break;
case PACKE:
    newul=ul;
```

```
    newspace=Pt(space.x-c->sizereq.x, space.y);
    pl_setrect(c, Pt(ul.x+space.x-c->sizereq.x, ul.y),
        Pt(c->sizereq.x, space.y));
    break;
```

Uses pl_setrect() 38c.

Expanding child

When children have the EXPAND flag, the leftover space (slack = available minus total requested) is distributed among them. pl_getshare()[43] counts how many children want to expand in each direction, and each one gets an equal fraction of the slack. The division is done incrementally (dividing the remaining slack by the remaining count) to handle rounding without losing pixels.

```
Before EXPAND:                  After EXPAND (2 children, both EXPAND):
+-----space=100------+          +-----space=100------+
|  A (req 30) PACKN  |          |  A (30+20=50) PACKN|
+--------------------+          +--------------------+
|  B (req 30) PACKN  |          |  B (30+20=50) PACKN|
+--------------------+          +--------------------+
|  slack = 40        |             slack distributed evenly:
+--------------------+             40 / 2 = 20 each
```

⟨pl_setrect() *locals* 42a⟩+≡ (38c) ◁40b

```
Vector slack, share;
int l;
```

⟨pl_setrect() *before looping over children, set locals* 42b⟩≡ (40c)

```
slack=subpt(space, p->childreq);
share=pl_getshare(p->child);
```

Uses pl_getshare() 43.

⟨pl_setrect() *when looping over children, if* EXPAND *flag* 42c⟩≡ (40c)

```
if(c->flags&EXPAND){
    switch(c->flags&PACK){
    case PACKN:
    case PACKS:
        c->sizereq.x+=slack.x;
        l=slack.y/share.y;
        c->sizereq.y+=l;
        slack.y-=l;
        --share.y;
        break;
    case PACKE:
    case PACKW:
        l=slack.x/share.x;
        c->sizereq.x+=l;
        slack.x-=l;
        --share.x;
        c->sizereq.y+=slack.y;
        break;
    }
}
```

⟨*function* `pl_getshare` 43⟩≡ (132c)

```
Point pl_getshare(Panel *p){
    Point share;

    if(p==nil)
        return Pt(0,0);

    // recurse
    share=pl_getshare(p->next);

    if(p->flags&EXPAND) {
        switch(p->flags&PACK){
        case PACKN:
        case PACKS:
            if(share.x==0) share.x=1;
            share.y++;
            break;
        case PACKE:
        case PACKW:
            share.x++;
            if(share.y==0) share.y=1;
            break;
        }
    }
    return share;
}
```

Uses `pl_getshare()` 43.

Chapter 8

Basic Widgets

Now that we have seen the core infrastructure (data structures, drawing, events, layout), we can look at the individual widgets. This chapter covers the basic leaf widgets: labels, text entries, and buttons. They all have the `LEAF` flag set and cannot have children. Each widget follows the same pattern: a private data structure (e.g., `Label`[44a], `Entry`[47g], `Button`[53a]), a constructor that calls `pl_newpanel()`[19a] and fills in the methods, and implementations of `draw`, `hit`, `type`, `getsize`, and `childspace`.

8.1 Label

A label is the simplest widget: it just displays an `Icon`[22d] (text or image) inside a passive border. Its private data stores only the icon and a placement direction for positioning the content within the label's rectangle.

⟨*struct* `Label` 44a⟩≡ (128c)
```
struct Label{
    // enum<Placement> (default = PLACECEN)
    int placement;
    // ref_own<Icon> = ref_own<Image|string> depending on Panel.flags&BITMAP
    Icon *icon;
};
```

⟨*constant* `BITMAP` 44b⟩≡ (121d)
```
#define BITMAP 0x4000  /* text argument is a bitmap, not a string */
```

8.1.1 Initializing

⟨*function* `pllabel` 44c⟩≡ (128c)
```
Panel *pllabel(Panel *parent, int flags, Icon *icon){
    Panel *p;

    p=pl_newpanel(parent, sizeof(Label));
    plinitlabel(p, flags, icon);
    plplacelabel(p, PLACECEN);
    return p;
}
```
Uses `pl_newpanel()` 19a, `plinitlabel()` 44d, and `plplacelabel()` 45a.

⟨*function* `plinitlabel` 44d⟩≡ (128c)
```
void plinitlabel(Panel *v, int flags, Icon *icon){
    Label* l = v->data;

    v->flags=flags|LEAF;
```

```
    v->draw=pl_drawlabel;
    v->hit=pl_hitlabel;
    v->type=pl_typelabel;

    v->getsize=pl_getsizelabel;
    v->childspace=pl_childspacelabel;

    l->icon=icon;
    //l->placement set in plplacelabel()

    v->kind="label";
}
```

Uses LEAF, pl_childspacelabel() 47f, pl_drawlabel() 45b, pl_getsizelabel() 47d, pl_hitlabel() 47b, and pl_typelabel() 47c.

Recall that v->data was already allocated by pl_newpanel()[19a] with sizeof(Label) bytes, so plinitlabel()[44d] can cast it directly to Label * and fill in the fields. The kind field is a string used for debugging messages (see the Debugging appendix) — it was already referenced by the error message in pl_newpanel().

Notice that the constructor pllabel()[44c] and the initializer plinitlabel() are separate functions. This is a deliberate pattern used by every widget type in libpanel: the plxxx() function allocates a new panel and calls plinitxxx() to fill it in, but plinitxxx() can also be called directly on an existing panel to *reinitialize* it without destroying and recreating it. This is how applications update widget content at runtime—for example, calling plinitlabel(p, flags, "new text") to change a label's text, then pldraw(p, view) to redisplay just that widget.

⟨*function* plplacelabel 45a⟩≡ (128c)

```
void plplacelabel(Panel *p, int placement){
    Label* l = p->data;

    l->placement=placement;
}
```

8.1.2 Drawing

Drawing a label is straightforward: draw a PASSIVE box (light background, thin border), with pl_box()[29c] (which internally calls pl_boxoutline()[30a]), then draw the icon inside it using the label's placement setting.

⟨*function* pl_drawlabel 45b⟩≡ (128c)

```
void pl_drawlabel(Panel *p){
    Label *l = p->data;

    pl_drawicon(p->b, pl_box(p->b, p->r, PASSIVE),
                l->placement, p->flags, l->icon);
}
```

Uses PASSIVE 45c, pl_box() 29c, and pl_drawicon() 46b.

⟨Style *other cases* 45c⟩≡ (22a) 67b ▷

```
// for labels
PASSIVE,
```

The PASSIVE case of pl_boxoutline() (defined in Chapter 5) is shown here because this is where it is first used. In literate programming, definition chunks can be spread across multiple chapters — noweb collects all the pl_boxoutline() switch style cases chunks and assembles them into a single switch statement.

⟨pl_boxoutline() *switch style cases* 45d⟩≡ (30a) 49d ▷

```
case PASSIVE:
    if(fill)
        draw(b, r, pl_light, 0, ZP);
```

```
    r=insetrect(r, PWID);
    if(!fill)
        border(b, r, SPACE, pl_white, ZP);
    break;
```

Uses PASSIVE 45c, PWID-13 46a, SPACE-16 30b, pl_light-24 26c, and pl_white-23 26c.

⟨*constant* PWID 46a⟩≡ (127c)
```
#define PWID 1 /* width of label border */
```

`pl_drawicon()`[46b] is the shared helper that actually renders an `Icon`[22d] (text or bitmap) inside a rectangle. It computes the offset needed to position the icon according to the `stick` placement, adjusts the clip rectangle to avoid drawing outside the widget's bounds, then calls either `draw()` (for bitmaps) or `string()` (for text).

⟨*function* pl_drawicon 46b⟩≡ (127c)
```
void pl_drawicon(Image *b, Rectangle r, int stick, int flags, Icon *s){
    Point ul;
    Vector offs;
    ⟨pl_drawicon() other locals 46e⟩

    ul=r.min;
    offs=subpt(subpt(r.max, r.min), pl_iconsize(flags, s));

    switch(stick){
    ⟨pl_drawicon() switch placement cases, adjust ul 46d⟩
    }

    ⟨pl_drawicon() save and adjust clip rectangle 46f⟩
    if(flags&BITMAP)
        draw(b, Rpt(ul, addpt(ul, pl_iconsize(flags, s))), s, nil, ZP);
    else
        string(b, ul, pl_black, ZP, font, s);
    ⟨pl_drawicon() restore saved clip rectangle 47a⟩
}
```

Uses pl_black-26 26c and pl_iconsize() 46c.

⟨*function* pl_iconsize 46c⟩≡ (127c)
```
Vector pl_iconsize(int flags, Icon *p){
    if(flags&BITMAP)
        return subpt(((Image *)p)->r.max, ((Image *)p)->r.min);
    else
        return stringsize(font, (char *)p);
}
```

⟨pl_drawicon() *switch placement cases, adjust* ul 46d⟩≡ (46b)
```
case PLACENW: /** Nothing **/                break;
case PLACEN: ul.x+=offs.x/2;                 break;
case PLACENE: ul.x+=offs.x;                  break;
case PLACEW:                ul.y+=offs.y/2; break;
case PLACECEN: ul.x+=offs.x/2; ul.y+=offs.y/2; break;
case PLACEE: ul.x+=offs.x;                   break;
case PLACESW:                ul.y+=offs.y;   break;
case PLACES: ul.x+=offs.x/2; ul.y+=offs.y;   break;
case PLACESE: ul.x+=offs.x;   ul.y+=offs.y;   break;
```

⟨pl_drawicon() *other locals* 46e⟩≡ (46b)
```
Rectangle save;
```

⟨pl_drawicon() *save and adjust clip rectangle* 46f⟩≡ (46b)
```
save=b->clipr;
if(!rectclip(&r, save))
    return;
replclipr(b, b->repl, r);
```

⟨pl_drawicon() *restore saved clip rectangle* 47a⟩≡ (46b)
```
replclipr(b, b->repl, save);
```

8.1.3 Reacting

Labels do not react to mouse or keyboard events — the methods are no-ops.

⟨*function* pl_hitlabel 47b⟩≡ (128c)
```
bool pl_hitlabel(Panel *p, Mouse *m){
    USED(p, m);
    return false;
}
```

⟨*function* pl_typelabel 47c⟩≡ (128c)
```
void pl_typelabel(Panel *p, Rune c){
    USED(p, c);
}
```

8.1.4 Packing methods

⟨*function* pl_getsizelabel 47d⟩≡ (128c)
```
Vector pl_getsizelabel(Panel *p, Vector children){
    USED(children);  /* shouldn't have any children */
    return pl_boxsize(pl_iconsize(p->flags, ((Label *)p->data)->icon),
                      PASSIVE);
}
```
Uses PASSIVE 45c, pl_boxsize() 30c, and pl_iconsize() 46c.

⟨pl_boxsize() *switch state cases* 47e⟩≡ (30c) 52d ▷
```
case PASSIVE:
    return addpt(interior, Pt(2*(PWID+SPACE), 2*(PWID+SPACE)));
```
Uses PASSIVE 45c, PWID-13 46a, and SPACE-16 30b.

The childspace method is called by pl_setrect()[38c] during layout (see Chapter 7) to let the widget adjust the interior space available for its children. Since a label is a LEAF widget and has no children, this is a no-op.

⟨*function* pl_childspacelabel 47f⟩≡ (128c)
```
void pl_childspacelabel(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

8.2 Text entry

A text entry is a single-line editable field. Its private data stores the text buffer (entry), a pointer to the current end of text (entp), a pointer to the end of the allocated buffer (eent), a minimum display size, and a callback for when the user presses Enter.

⟨*struct* Entry 47g⟩≡ (129b)
```
struct Entry{
    // ref_own<string>
    char *entry;
    ⟨Entry pointer in entry fields 48a⟩
    Vector minsize;

    void (*hit)(Panel *, char *);
};
```

⟨Entry *pointer in* entry *fields* 48a⟩≡ (47g)
```
// point to \0 in entry
char *entp;
// point to end of entry
char *eent;
```

8.2.1 Initializing

⟨*function* plentry 48b⟩≡ (129b)
```
Panel *plentry(Panel *parent, int flags, int wid, char *str, void (*hit)(Panel *, char *)){
    Panel *v;

    v=pl_newpanel(parent, sizeof(Entry));
    plinitentry(v, flags, wid, str, hit);
    return v;
}
```
Uses pl_newpanel() 19a and plinitentry() 48c.

⟨*function* plinitentry 48c⟩≡ (129b)
```
void plinitentry(Panel *v, int flags, int wid, char *str, void (*hit)(Panel *, char *)){
    Entry *ep = v->data;
    int elen;

    v->flags=flags|LEAF;

    v->draw=pl_drawentry;
    v->hit=pl_hitentry;
    v->type=pl_typeentry;

    v->getsize=pl_getsizeentry;
    v->childspace=pl_childspaceentry;

    ⟨plinitentry() set snarf methods 109c⟩
    ⟨plinitentry() set extra fields 49b⟩
    ⟨plinitentry() set fields in ep 48e⟩

    v->free=pl_freeentry;

    v->kind="entry";
}
```
Uses LEAF, pl_childspaceentry() 52e, pl_drawentry() 49c, pl_freeentry() 48d, pl_getsizeentry() 52c, pl_hitentry() 50e, and pl_typeentry() 51a.

⟨*function* pl_freeentry 48d⟩≡ (129b)
```
void pl_freeentry(Panel *p){
    Entry *ep = p->data;

    free(ep->entry);
    ep->entry = ep->eent = ep->entp = nil;
}
```

⟨plinitentry() *set fields in* ep 48e⟩≡ (48c)
```
ep->minsize=Pt(wid, font->height);

elen=100;
if(str)
    elen+=strlen(str);
ep->entry=pl_erealloc(ep->entry, elen+SLACK);
```

```
  ep->eent=ep->entry+elen;
  strecpy(ep->entry, ep->eent, str ? str : "");
  ep->entp=ep->entry+strlen(ep->entry);

  ep->hit=hit;
```

Uses SLACK-1 110b and pl_erealloc() 120b.

⟨*constant* SLACK*(*(lib_gui/libpanel/textwin.c)*)* 49a⟩≡ (148d)

```
  #define SLACK 100
```

8.2.2 Drawing

The entry is drawn with an UP box when unfocused and a DOWN box when focused (clicked). The UP style has a light background with a raised border; DOWN reverses the relief to look sunken, signaling that the widget is active and accepting input. The text is aligned to the left (PLACEW) if it fits, or to the right (PLACEE) if it overflows, so the end of the text (where the user is typing) stays visible.

⟨plinitentry() *set extra fields* 49b⟩≡ (48c)

```
  v->state=UP;
```

Uses UP 22a.

⟨*function* pl_drawentry 49c⟩≡ (129b)

```
  void pl_drawentry(Panel *p){
      Entry *ep = p->data;
      char *s = ep->entry;
      Rectangle r;

      r=pl_box(p->b, p->r, p->state);

      ⟨pl_drawentry if USERFL 50c⟩
      if(stringwidth(font, s) <= r.max.x-r.min.x)
          pl_drawicon(p->b, r, PLACEW, NOFLAG, s);
      else
          pl_drawicon(p->b, r, PLACEE, NOFLAG, s);
      ⟨pl_drawentry when USERFL if s changed 50d⟩
  }
```

Uses pl_box() 29c and pl_drawicon() 46b.

⟨pl_boxoutline() *switch style cases* 49d⟩+≡ (30a) ◁45d 50a▷

```
  case UP:
      pl_relief(b, pl_white, pl_black, r, BWID);
      r=insetrect(r, BWID);
      if(fill)
          draw(b, r, pl_light, nil, ZP);
      else
          border(b, r, SPACE, pl_white, ZP);
      break;
```

Uses BWID-14 49e, SPACE-16 30b, UP 22a, pl_black-26 26c, pl_light-24 26c, pl_relief() 29a, and pl_white-23 26c.

⟨*constant* BWID 49e⟩≡ (127c)

```
  #define BWID 1 /* width of button relief */
```

⟨pl_boxoutline() *switch style cases* 50a⟩+≡ (30a) ◁49d 67c▷

```
case DOWN:
case DOWN1:
case DOWN2:
case DOWN3:
    pl_relief(b, pl_black, pl_white, r, BWID);
    r=insetrect(r, BWID);
    if(fill)
        draw(b, r, pl_dark, 0, ZP);
    else
        border(b, r, SPACE, pl_black, ZP);
    break;
```

Uses BWID-14 49e, DOWN 45c, DOWN1 22a, DOWN2 22a, DOWN3, SPACE-16 30b, pl_black-26 26c, pl_dark-25 26c, pl_relief() 29a, and pl_white-23 26c.

Drawing secret text

When a text entry is used for passwords, the actual characters should not be visible on screen. If the application sets the USERFL flag on the entry, the drawing code replaces every character with '*' before rendering. The internal buffer is unchanged — only the display is masked.

⟨*constant* USERFL 50b⟩≡ (121d)

```
#define USERFL 0x100000 /* start of user flag */
```

⟨pl_drawentry *if* USERFL 50c⟩≡ (49c)

```
if(p->flags & USERFL){
    char *p;
    s=strdup(s);
    for(p=s; *p; p++)
        *p='*';
}
```

⟨pl_drawentry *when* USERFL *if* s *changed* 50d⟩≡ (49c)

```
if(s != ep->entry)
    free(s);
```

8.2.3 Reacting

When the user clicks on a text entry, the hit method switches the state to DOWN, grabs the keyboard focus with plgrabkb()[33k], and enters a tight loop consuming mouse events until the button is released. The loop is needed because while the left button is held, a middle or right click triggers copy/paste operations (snarf and paste, see Chapter 13). Without the loop, those chord clicks would be lost.

⟨*function* pl_hitentry 50e⟩≡ (129b)

```
bool pl_hitentry(Panel *p, Mouse *m){
    if((m->buttons&7)==CLICK_LEFT){
        p->state=DOWN;
        plgrabkb(p);
        // redraw with changed state
        pldraw(p, p->b);

        while(m->buttons&CLICK_LEFT){
            int old = m->buttons;
            // next mouse event
            *m=emouse();
            ⟨pl_hitentry() handle copy/paste when middle or right click 110c⟩
        }
        // redraw with changed state
```

```
        p->state=UP;
        pldraw(p, p->b);
    }
    return false;
}
```

Uses DOWN 45c, UP 22a, pldraw() 28a, and plgrabkb() 33k.

The type method handles keyboard input: Enter triggers the callback, printable characters are appended to the buffer (with automatic reallocation if needed), and backspace erases the last rune.

⟨*function* pl_typeentry 51a⟩≡ (129b)

```
void pl_typeentry(Panel *p, Rune c){
    Entry *ep = p->data;
    int n;

    switch(c){
    ⟨pl_typeentry() switch rune cases 51b⟩
    }
    // draw updated entry
    pldraw(p, p->b);
}
```

Uses pldraw() 28a.

⟨pl_typeentry() *switch rune cases* 51b⟩≡ (51a) 51c▷

```
case '\n':
case '\r':
    *ep->entp='\0';
    if(ep->hit)
        ep->hit(p, ep->entry);
    return;
```

⟨pl_typeentry() *switch rune cases* 51c⟩+≡ (51a) ◁51b 52a▷

```
default:
    ⟨pl_typeentry() handle special characters 51e⟩
    ep->entp+=runetochar(ep->entp, &c);
    ⟨pl_typeentry() realloc entry if necessary 51d⟩
    *ep->entp='\0';
    break;
```

⟨pl_typeentry() *realloc entry if necessary* 51d⟩≡ (51c)

```
if(ep->entp>ep->eent){
    n=ep->entp-ep->entry;
    ep->entry=pl_erealloc(ep->entry, n+100+SLACK);
    ep->entp=ep->entry+n;
    ep->eent=ep->entp+100;
}
```

Uses SLACK-1 110b and pl_erealloc() 120b.

Special keys

⟨pl_typeentry() *handle special characters* 51e⟩≡ (51c)

```
if(c < 0x20 || (c & 0xFF00) == KF || (c & 0xFF00) == Spec)
    break;
```

⟨pl_typeentry() *switch rune cases* 52a⟩+≡ (51a) ◁51c 52b▷

```
case Kesc:
    plsnarf(p);
    /* no break */
case Kdel: /* clear */
case Kbs: /* ^H: erase character */
    while(ep->entp!=ep->entry && !pl_rune1st(ep->entp[-1]))
        *--ep->entp='\0';
    if(ep->entp!=ep->entry)
        *--ep->entp='\0';
    break;
```

Uses pl_rune1st() 149b and plsnarf() 109a.

⟨pl_typeentry() *switch rune cases* 52b⟩+≡ (51a) ◁52a

```
// case Knack: /* ^U: erase line */
//  ep->entp=ep->entry;
//  *ep->entp='\0';
//  break;
// case Ketb: /* ^W: erase word */
//  while(ep->entp!=ep->entry && !pl_idchar(ep->entp[-1]))
//   --ep->entp;
//  while(ep->entp!=ep->entry && pl_idchar(ep->entp[-1]))
//   --ep->entp;
//  *ep->entp='\0';
//  break;
```

8.2.4 Packing methods

⟨*function* pl_getsizeentry 52c⟩≡ (129b)

```
Vector pl_getsizeentry(Panel *p, Vector children){
    Entry* e = p->data;

    USED(children);
    return pl_boxsize(e->minsize, p->state);
}
```

Uses pl_boxsize() 30c.

⟨pl_boxsize() *switch state cases* 52d⟩+≡ (30c) ◁47e 68a▷

```
case UP:
case DOWN:
case DOWN1:
case DOWN2:
case DOWN3:
    return addpt(interior, Pt(2*(BWID+SPACE), 2*(BWID+SPACE)));
```

Uses BWID-14 49e, DOWN 45c, DOWN1 22a, DOWN2 22a, DOWN3, SPACE-16 30b, and UP 22a.

⟨*function* pl_childspaceentry 52e⟩≡ (129b)

```
void pl_childspaceentry(Panel *p, Point *ul, Vector *size){
    USED(p, ul, size);
}
```

8.2.5 Other methods

⟨*function* plentryval 52f⟩≡ (129b)

```
char *plentryval(Panel *p){
    Entry *ep = p->data;
```

```
    *ep->entp='\0';
    return ep->entry;
}
```

8.3 Button

`libpanel` has three kinds of buttons — regular, check, and radio — sharing the same `Button`[53a] data structure and methods. The `btype` field distinguishes them. They differ mainly in drawing (check buttons have a checkbox, radio buttons have a filled square) and in hit behavior (check toggles, radio ensures mutual exclusion among siblings).

8.3.1 Button

⟨*struct* `Button` 53a⟩≡ (129a)
```
struct Button{
    // enum<ButtonType>
    int btype;   /* button type */

    Icon *icon;   /* what to write on the button */

    ⟨Button other fields 53e⟩

    void (*pl_buttonhit)(Panel *, buttons); /* call back user code on button hit */
    void (*hit)(Panel *, buttons, bool); /* call back user code on check/radio hit */
    ⟨Button other methods 82b⟩
};
```

⟨*constant* `BUTTON` 53b⟩≡ (129a)
```
#define BUTTON 1
```

⟨*constant* `CHECK` 53c⟩≡ (129a)
```
#define CHECK 2
```

⟨*constant* `RADIO` 53d⟩≡ (129a)
```
#define RADIO 3
```

⟨Button *other fields* 53e⟩≡ (53a) 53f▷
```
// state of buttons when hit
buttons buttons;
```

⟨Button *other fields* 53f⟩+≡ (53a) ◁53e 82a▷
```
bool check;   /* for check/radio buttons */
```

⟨*function* `pl_initbtype` 53g⟩≡ (129a)
```
void pl_initbtype(Panel *v, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool), int btype){
    Button *bp = v->data;

    v->flags=flags|LEAF;
    v->state=UP;

    v->draw=pl_drawbutton;
    v->hit=pl_hitbutton;
    v->type=pl_typebutton;

    v->getsize=pl_getsizebutton;
    v->childspace=pl_childspacebutton;
```

```
    bp->btype=btype;
    bp->check=false;
    bp->hit=hit;
    bp->icon=icon;

    switch(btype){
    case BUTTON: v->kind="button"; break;
    case CHECK:  v->kind="checkbutton"; break;
    case RADIO:  v->kind="radiobutton"; break;
    }
}
```

Uses BUTTON-10 53b, CHECK-11 53c, LEAF, RADIO-12 53d, UP 22a, pl_childspacebutton() 56b, pl_drawbutton() 54d, pl_getsizebutton() 56a, pl_hitbutton() 55b, and pl_typebutton() 55c.

8.3.2 Basic button

Initializing

⟨*function* plbutton 54a⟩≡ (129a)

```
Panel *plbutton(Panel *parent, int flags, Icon *icon, void (*hit)(Panel *, buttons)){
    Panel *p;

    p=pl_newpanel(parent, sizeof(Button));
    plinitbutton(p, flags, icon, hit);
    return p;
}
```

Uses pl_newpanel() 19a and plinitbutton() 54b.

⟨*function* plinitbutton 54b⟩≡ (129a)

```
void plinitbutton(Panel *p, int flags, Icon *icon, void (*hit)(Panel *, buttons)){
    Button* b = p->data;

    b->pl_buttonhit=hit;
    pl_initbtype(p, flags, icon, pl_buttonhit, BUTTON);
}
```

Uses BUTTON-10 53b, pl_buttonhit() 54c, and pl_initbtype() 53g.

⟨*function* pl_buttonhit 54c⟩≡ (129a)

```
void pl_buttonhit(Panel *p, buttons buttons, bool check){
    Button* b = p->data;

    USED(check);
    if(b->pl_buttonhit)
        b->pl_buttonhit(p, buttons);
}
```

Drawing

⟨*function* pl_drawbutton 54d⟩≡ (129a)

```
void pl_drawbutton(Panel *p){
    Rectangle r;
    Button *bp = p->data;

    r=pl_box(p->b, p->r, p->state);
    switch(bp->btype){
    ⟨pl_drawbutton() switch button type cases 55a⟩
    }
    pl_drawicon(p->b, r, PLACECEN, p->flags, bp->icon);
```

```
}
```

Uses pl_box() 29c and pl_drawicon() 46b.

⟨pl_drawbutton() *switch button type cases* 55a⟩≡ (54d) 56e▷

```
case BUTTON:
    break;
```

Uses BUTTON-10 53b.

Reacting

The button hit handler follows a three-way pattern common to many widgets: (1) OUT: the mouse left — revert to UP; (2) buttons pressed: switch to DOWN and remember which button; (3) no buttons (release): if we were DOWN, this is a real click. This means dragging out of the button cancels the click — the callback only fires if the user releases *inside* the widget.

⟨*function* pl_hitbutton 55b⟩≡ (129a)

```
bool pl_hitbutton(Panel *p, Mouse *m){
    Button *bp = p->data;
    int oldstate = p->state;
    bool hitme;
    ⟨pl_hitbutton() other locals 59b⟩

    if(m->buttons&OUT){ // mouse leaving the widget
        p->state=UP;
        hitme=false;
    }
    else if(m->buttons&7){ // mouse click inside
        p->state=DOWN;
        hitme=false;
        bp->buttons=m->buttons; // remember for the release event
    }
    else{ /* mouse inside, but no buttons down */ // possibly a release
        hitme=p->state==DOWN;
        p->state=UP;
    }

    if(hitme) {
        switch(bp->btype){
        ⟨pl_hitbutton() switch button type cases 57e⟩
        }
    }
    if(hitme || oldstate!=p->state)
        pldraw(p, p->b);
    if(hitme && bp->hit){
        // user callback
        bp->hit(p, bp->buttons, bp->check);
        p->state=UP;
    }
    return false;
}
```

Uses DOWN 45c, UP 22a, and pldraw() 28a.

⟨*function* pl_typebutton 55c⟩≡ (129a)

```
void pl_typebutton(Panel *g, Rune c){
    USED(g, c);
}
```

Packing methods

⟨*function* pl_getsizebutton 56a⟩≡ (129a)
```
Vector pl_getsizebutton(Panel *p, Vector children){
    Button *bp = p->data;
    Vector s;
    ⟨pl_getsizebutton() other locals 57f⟩

    USED(children);  /* shouldn't have any children */
    s=pl_iconsize(p->flags, bp->icon);
    ⟨pl_getsizebutton() if not a BUTTON 57g⟩
    return pl_boxsize(s, p->state);
}
```
Uses pl_boxsize() 30c and pl_iconsize() 46c.

⟨*function* pl_childspacebutton 56b⟩≡ (129a)
```
void pl_childspacebutton(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

8.3.3 Check button

Initializing

⟨*function* plcheckbutton 56c⟩≡ (129a)
```
Panel *plcheckbutton(Panel *parent, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool)){
    Panel *p;

    p=pl_newpanel(parent, sizeof(Button));
    plinitcheckbutton(p, flags, icon, hit);
    return p;
}
```
Uses pl_newpanel() 19a and plinitcheckbutton() 56d.

⟨*function* plinitcheckbutton 56d⟩≡ (129a)
```
void plinitcheckbutton(Panel *p, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool)){
    pl_initbtype(p, flags, icon, hit, CHECK);
}
```
Uses CHECK-11 53c and pl_initbtype() 53g.

Drawing

⟨pl_drawbutton() *switch button type cases* 56e⟩+≡ (54d) ◁55a 58f▷
```
case CHECK:
    r=pl_check(p->b, r, bp->check);
    break;
```
Uses CHECK-11 53c and pl_check() 56f.

⟨*function* pl_check 56f⟩≡ (127c)
```
Rectangle pl_check(Image *b, Rectangle r, bool val){
    Rectangle remainder = r;

    r.max.x=r.min.x+r.max.y-r.min.y;
    remainder.min.x=r.max.x;
    r=insetrect(r, CKINSET);
    ⟨pl_check() if null plldepth part1 ??⟩
    else
        pl_relief(b, pl_black, pl_white, r, CKWID);
```

```
    r=insetrect(r, CKWID);
    ⟨pl_check() if null plldepth part2 ??⟩
    else
        draw(b, r, pl_light, nil, ZP);
    r=insetrect(r, CKBORDER);
    if(val){
        ⟨pl_check() if checked button 57d⟩
    }
    return remainder;
}
```

Uses CKBORDER-21 57c, CKINSET-20 57a, CKWID-19 57b, pl_black-26 26c, pl_light-24 26c, pl_relief() 29a, and pl_white-23 26c.

⟨*constant* CKINSET 57a⟩≡ (127c)
```
#define CKINSET 1 /* space around check mark frame */
```

⟨*constant* CKWID 57b⟩≡ (127c)
```
#define CKWID 1 /* width of frame around check mark */
```

⟨*constant* CKBORDER 57c⟩≡ (127c)
```
#define CKBORDER 2 /* space around X inside frame */
```

⟨pl_check() *if checked button* 57d⟩≡ (56f)
```
line(b, Pt(r.min.x,   r.min.y+1), Pt(r.max.x-1, r.max.y  ), Endsquare, Endsquare, 0, pl_black, ZP);
line(b, Pt(r.min.x,   r.min.y  ), Pt(r.max.x,   r.max.y  ), Endsquare, Endsquare, 0, pl_black, ZP);
line(b, Pt(r.min.x+1, r.min.y  ), Pt(r.max.x,   r.max.y-1), Endsquare, Endsquare, 0, pl_black, ZP);
line(b, Pt(r.min.x  , r.max.y-2), Pt(r.max.x-1, r.min.y-1), Endsquare, Endsquare, 0, pl_black, ZP);
line(b, Pt(r.min.x,   r.max.y-1), Pt(r.max.x,   r.min.y-1), Endsquare, Endsquare, 0, pl_black, ZP);
line(b, Pt(r.min.x+1, r.max.y-1), Pt(r.max.x,   r.min.y  ), Endsquare, Endsquare, 0, pl_black, ZP);
```

Uses pl_black-26 26c.

Reacting

⟨pl_hitbutton() *switch button type cases* 57e⟩≡ (55b) 59c▷
```
case CHECK:
    if(hitme)
        bp->check=!bp->check;
    break;
```

Uses CHECK-11 53c.

Packing methods

⟨pl_getsizebutton() *other locals* 57f⟩≡ (56a)
```
int ckw;
```

⟨pl_getsizebutton() *if not a* BUTTON 57g⟩≡ (56a)
```
if(bp->btype!=BUTTON){
    ckw=pl_ckwid();
    if(s.y<ckw){
        s.x+=ckw;
        s.y=ckw;
    }
    else s.x+=s.y;
}
```

Uses BUTTON-10 53b and pl_ckwid() 58a.

⟨*function* pl_ckwid 58a⟩≡ (127c)
```
int pl_ckwid(void){
    return 2*(CKINSET+CKSPACE+CKWID)+CKSIZE;
}
```
Uses CKINSET-20 57a, CKSIZE-17 58b, CKSPACE-18 59a, and CKWID-19 57b.

⟨*constant* CKSIZE 58b⟩≡ (127c)
```
#define CKSIZE 3 /* size of check mark */
```

Other methods

⟨*function* plsetbutton 58c⟩≡ (129a)
```
void plsetbutton(Panel *p, int val){
    Button* b = p->data;

    b->check=val;
}
```

8.3.4 Radio button

Initializing

⟨*function* plradiobutton 58d⟩≡ (129a)
```
Panel *plradiobutton(Panel *parent, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool)){
    Panel *p;

    p=pl_newpanel(parent, sizeof(Button));
    plinitradiobutton(p, flags, icon, hit);
    return p;
}
```
Uses pl_newpanel() 19a and plinitradiobutton() 58e.

⟨*function* plinitradiobutton 58e⟩≡ (129a)
```
void plinitradiobutton(Panel *p, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool)){
    pl_initbtype(p, flags, icon, hit, RADIO);
}
```
Uses RADIO-12 53d and pl_initbtype() 53g.

Drawing

⟨pl_drawbutton() *switch button type cases* 58f⟩+≡ (54d) ◁56e
```
case RADIO:
    r=pl_radio(p->b, r, bp->check);
    break;
```
Uses RADIO-12 53d and pl_radio() 58g.

⟨*function* pl_radio 58g⟩≡ (127c)
```
/*
 * Place a check mark at the left end of r.  Return the unused space.
 * Caller must guarantee that r.max.x-r.min.x>=r.max.y-r.min.y!
 */
Rectangle pl_radio(Image *b, Rectangle r, bool val){
    Rectangle remainder = r;

    r.max.x=r.min.x+r.max.y-r.min.y;
    remainder.min.x=r.max.x;
    r=insetrect(r, CKINSET);
```

```
    ⟨pl_radio() if null plldepth part1 ??⟩
    else
        pl_relief(b, pl_black, pl_white, r, CKWID);
    r=insetrect(r, CKWID);
    ⟨pl_radio() if null plldepth part2 ??⟩
    else
        draw(b, r, pl_light, 0, ZP);
    if(val)
        draw(b, insetrect(r, CKSPACE), pl_black, nil, ZP);
    return remainder;
}
```

Uses CKINSET-20 57a, CKSPACE-18 59a, CKWID-19 57b, pl_black-26 26c, pl_light-24 26c, pl_relief() 29a, and pl_white-23 26c.

⟨*constant* CKSPACE 59a⟩≡ (127c)

```
#define CKSPACE 2 /* space around check mark */
```

Reacting

Radio buttons enforce mutual exclusion: when one is selected, all sibling radio buttons in the same parent are deselected. The code walks the parent's child list looking for other radio buttons (identified by checking `sib->hit==pl_hitbutton` and `btype==RADIO`) and clears their `check` flag.

⟨pl_hitbutton() *other locals* 59b⟩≡ (55b)

```
Panel *sib;
```

⟨pl_hitbutton() *switch button type cases* 59c⟩+≡ (55b) ◁57e

```
case RADIO:
    if(bp->check)
        bp->check=false;
    else{
        if(p->parent){
            for(sib=p->parent->child;sib;sib=sib->next){
                if(sib->hit==pl_hitbutton
                && ((Button *)sib->data)->btype==RADIO
                && ((Button *)sib->data)->check){
                    ((Button *)sib->data)->check=false;
                    pldraw(sib, p->b);
                }
            }
        }
        bp->check=true;
    }
    break;
```

Uses RADIO-12 53d, pl_hitbutton() 55b, and pldraw() 28a.

8.4 Slider

A slider is a rectangular region that the user can drag to select a value. It can be either horizontal or vertical—the direction is inferred from the aspect ratio of the minimum size passed at creation time. The slider's value is stored in screen coordinates (pixels), not in application-domain units; the conversion between logical values and pixel positions happens in `plsetslider` and in the hit callback.

⟨*struct* Slider 59d⟩≡ (131a)

```
struct Slider{
    // enum<Direction>
    int dir;   /* HORIZ or VERT */
    Vector minsize; // range
```

```
    int val;   /* setting, in screen coordinates */
    ⟨Slider other fields 60a⟩

    void (*hit)(Panel *, buttons, int, int); /* call back to user when slider changes */
};
```

⟨Slider *other fields* 60a⟩≡ (59d)
```
buttons buttons;
```

The `buttons` field records which mouse button is currently pressed, so the hit callback can distinguish between button 1, 2, and 3 drags. This is less critical than for buttons since the callback fires continuously during drag rather than once on release.

8.4.1 Initializing

⟨*function* plslider 60b⟩≡ (131a)
```
Panel *plslider(Panel *parent, int flags, Vector size, void (*hit)(Panel *, buttons, int, int)){
    Panel *p;

    p=pl_newpanel(parent, sizeof(Slider));
    plinitslider(p, flags, size, hit);
    return p;
}
```
Uses pl_newpanel() 19a and plinitslider() 60c.

⟨*function* plinitslider 60c⟩≡ (131a)
```
void plinitslider(Panel *v, int flags, Vector size, void (*hit)(Panel *, buttons, int, int)){
    Slider *sp = v->data;

    v->flags=flags|LEAF;
    v->state=UP;

    v->draw=pl_drawslider;
    v->hit=pl_hitslider;
    v->type=pl_typeslider;

    v->getsize=pl_getsizeslider;
    v->childspace=pl_childspaceslider;

    v->r=Rect(0,0,size.x,size.y);

    sp->minsize=size;
    sp->dir=size.x>size.y?HORIZ:VERT;
    sp->hit=hit;

    v->kind="slider";
}
```
Uses HORIZ 125c, LEAF, UP 22a, VERT, pl_childspaceslider() 63d, pl_drawslider() 61a, pl_getsizeslider() 63c, pl_hitslider() 62a, and pl_typeslider() 63b.

8.4.2 Drawing

The slider is drawn as three adjacent rectangles: light, dark, light. The dark region represents the current value. For a horizontal slider, the dark band starts at the left edge and extends to `val`; for a vertical slider, it starts at `val` measured from the top and extends to the bottom (the vertical coordinate is flipped so that "up" means a higher value):

```
Horizontal:                Vertical:
+------+----+-------+     +-------+  <- r.min.y
|light |dark| light |     | light |
+------+----+-------+     +-------+  <- r.max.y - val
^      ^    ^       ^     | dark  |
r.min.x lo   hi   r.max.x +-------+  <- r.max.y
```

⟨*function* `pl_drawslider` 61a⟩≡ (131a)

```
void pl_drawslider(Panel *p){
    Rectangle r;
    Slider *sp = p->data;

    r=pl_box(p->b, p->r, UP);
    switch(sp->dir){
    case HORIZ: pl_sliderupd(p->b, r, sp->dir, 0, sp->val); break;
    case VERT:  pl_sliderupd(p->b, r, sp->dir, r.max.y-sp->val, r.max.y); break;
    }
}
```

Uses `HORIZ` 125c, `UP` 22a, `VERT`, `pl_box()` 29c, and `pl_sliderupd()` 61b.

⟨*function* `pl_sliderupd` 61b⟩≡ (127c)

```
void pl_sliderupd(Image *b, Rectangle r1, int dir, int lo, int hi){
    Rectangle r2, r3;

    r2=r1;
    r3=r1;

    ⟨pl_sliderupd() sanitize lo and hi 61c⟩

    switch(dir){
    case HORIZ:
        r1.max.x=r1.min.x+lo;
        r2.min.x=r1.max.x;
        r2.max.x=r1.min.x+hi;
        if(r2.max.x>r3.max.x) r2.max.x=r3.max.x;
        r3.min.x=r2.max.x;
        break;
    case VERT:
        r1.max.y=r1.min.y+lo;
        r2.min.y=r1.max.y;
        r2.max.y=r1.min.y+hi;
        if(r2.max.y>r3.max.y) r2.max.y=r3.max.y;
        r3.min.y=r2.max.y;
        break;
    }
    draw(b, r1, pl_light, nil, ZP);
    draw(b, r2, pl_dark, nil, ZP);
    draw(b, r3, pl_light, nil, ZP);
}
```

Uses `HORIZ` 125c, `VERT`, `pl_dark-25` 26c, and `pl_light-24` 26c.

The three rectangles are carved from the slider's interior by splitting along `lo` and `hi`: `r1` is the region before the thumb, `r2` is the thumb itself (drawn dark), and `r3` is the region after.

⟨`pl_sliderupd()` *sanitize* `lo` *and* `hi` 61c⟩≡ (61b)

```
if(lo<0) lo=0;
if(hi<=lo) hi=lo+1;
```

8.4.3 Reacting

Unlike buttons, sliders fire the user callback immediately on each mouse movement while a button is pressed, not on release. This gives real-time feedback—essential for scroll bars and volume controls where the user expects to see the effect as they drag. The hit handler converts the mouse position to a pixel offset within the slider's interior:

```
Horizontal slider:
+--[===|===========]--+
^   ^    ^                ^
|  ul   m->xy.x          |
p->r.min             p->r.max

val = m->xy.x - ul.x
len = size.x
```

For vertical sliders, the Y axis is flipped so that moving the mouse up increases the value: `val = ul.y + size.y - m->xy.y`.

⟨*function* `pl_hitslider` 62a⟩≡ (131a)

```
bool pl_hitslider(Panel *p, Mouse *m){
    Slider *sp = p->data;
    int oldstate, oldval;
    int len;
    Point ul;
    Vector size;
    SET(len);

    ul=p->r.min;
    size=subpt(p->r.max, p->r.min);
    pl_interior(p->state, &ul, &size);

    oldstate=p->state;
    oldval=sp->val;

    if(m->buttons&OUT)
        p->state=UP;
    else if(m->buttons&7){
        p->state=DOWN;
        sp->buttons=m->buttons;
        ⟨pl_hitslider() when button, set sp->val and len 62b⟩
        ⟨pl_hitslider() sanitize sp->val 63a⟩
    }
    else /* mouse inside, but no buttons down */
        p->state=UP;

    if(oldval!=sp->val || oldstate!=p->state)
        pldraw(p, p->b);
    if(oldval!=sp->val && sp->hit)
        // user callback
        sp->hit(p, sp->buttons, sp->val, len);
    return false;
}
```

Uses `DOWN` 45c, `UP` 22a, `pl_interior()` 30d, and `pldraw()` 28a.

The callback receives both `val` (current position in pixels) and `len` (total length in pixels), so the application can compute a ratio without needing to query the widget's geometry.

⟨`pl_hitslider()` *when button, set* `sp->val` *and* `len` 62b⟩≡ (62a)

```
if(sp->dir==HORIZ){
    sp->val=m->xy.x-ul.x;
    len=size.x;
}else{
    sp->val=ul.y+size.y-m->xy.y;
    len=size.y;
}
```

Uses HORIZ 125c.

⟨pl_hitslider() *sanitize* sp->val 63a⟩≡ (62a)

```
if(sp->val<0)
    sp->val=0;
else
    if(sp->val>len)
        sp->val=len;
```

⟨*function* pl_typeslider 63b⟩≡ (131a)

```
void pl_typeslider(Panel *p, Rune c){
    USED(p, c);
}
```

8.4.4 Packing methods

⟨*function* pl_getsizeslider 63c⟩≡ (131a)

```
Vector pl_getsizeslider(Panel *p, Vector children){
    USED(children);
    return pl_boxsize(((Slider *)p->data)->minsize, p->state);
}
```

Uses pl_boxsize() 30c.

⟨*function* pl_childspaceslider 63d⟩≡ (131a)

```
void pl_childspaceslider(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

8.4.5 Other methods

plsetslider is the programmatic counterpart of dragging: it converts a logical value/range pair into the pixel coordinate stored in val, scaling proportionally to the widget's current screen size.

⟨*function* plsetslider 63e⟩≡ (131a)

```
void plsetslider(Panel *p, int value, int range){
    Slider *sp = p->data;

    ⟨plsetslider() sanitize value 63f⟩
    if(sp->dir==HORIZ)
        sp->val=value*(p->r.max.x-p->r.min.x)/range;
    else
        sp->val=value*(p->r.max.y-p->r.min.y)/range;
}
```

Uses HORIZ 125c.

⟨plsetslider() *sanitize* value 63f⟩≡ (63e)

```
if(value<0) value=0;
else if(value>range) value=range;
```

8.5 Canvas

Canvas is the "escape hatch" of the widget library. Unlike every other widget, canvas does not implement its own drawing or hit-testing logic. Instead, it stores user-supplied function pointers for both `draw` and `hit`, delegating all behavior to the application. This makes it the equivalent of GTK's `DrawingArea` or Tk's `Canvas`—a blank rectangular region where the application can render arbitrary graphics and handle raw mouse events. The widget itself has no intrinsic size (`pl_getsizecanvas` returns zero), so the application must use `EXPAND` flags or padding to give it screen space.

⟨*struct* `Canvas` 64a⟩≡ (131b)
```
struct Canvas{
    void (*draw)(Panel *);
    void (*hit)(Panel *, Mouse *);
};
```

8.5.1 Initializing

⟨*function* `plcanvas` 64b⟩≡ (131b)
```
Panel *plcanvas(Panel *parent, int flags, void (*draw)(Panel *), void (*hit)(Panel *, Mouse *)){
    Panel *p;

    p=pl_newpanel(parent, sizeof(Canvas));
    plinitcanvas(p, flags, draw, hit);
    return p;
}
```
Uses `pl_newpanel()` 19a and `plinitcanvas()` 64c.

⟨*function* `plinitcanvas` 64c⟩≡ (131b)
```
void plinitcanvas(Panel *v, int flags, void (*draw)(Panel *), void (*hit)(Panel *, Mouse *)){
    Canvas *c = v->data;;

    v->flags=flags|LEAF;

    v->draw=pl_drawcanvas;
    v->hit=pl_hitcanvas;
    v->type=pl_typecanvas;

    v->getsize=pl_getsizecanvas;
    v->childspace=pl_childspacecanvas;

    c->draw=draw;
    c->hit=hit;

    v->kind="canvas";
}
```
Uses `LEAF`, `pl_childspacecanvas()` 65d, `pl_drawcanvas()` 64d, `pl_getsizecanvas()` 65c, `pl_hitcanvas()` 65a, and `pl_typecanvas()` 65b.

8.5.2 Drawing

⟨*function* `pl_drawcanvas` 64d⟩≡ (131b)
```
void pl_drawcanvas(Panel *p){
    Canvas *c = p->data;

    if(c->draw)
        c->draw(p);
}
```

8.5.3 Reacting

⟨*function* `pl_hitcanvas` 65a⟩≡ (131b)

```
int pl_hitcanvas(Panel *p, Mouse *m){
    Canvas *c =p->data;

    if(c->hit)
        c->hit(p, m);
    return false;
}
```

⟨*function* `pl_typecanvas` 65b⟩≡ (131b)

```
void pl_typecanvas(Panel *p, Rune c){
    USED(p, c);
}
```

8.5.4 Packing methods

⟨*function* `pl_getsizecanvas` 65c⟩≡ (131b)

```
Vector pl_getsizecanvas(Panel *p, Vector children){
    USED(p, children);
    return Pt(0,0);
}
```

Because the intrinsic size is zero, the canvas is invisible unless the caller sets `EXPAND` flags (to fill available space) or uses internal padding to force a minimum size.

⟨*function* `pl_childspacecanvas` 65d⟩≡ (131b)

```
void pl_childspacecanvas(Panel *p, Point *ul, Vector *size){
    USED(p, ul, size);
}
```

Chapter 9

Composite Widgets

The previous chapter covered leaf widgets—labels, entries, buttons, sliders, and canvases—that have no children. This chapter presents the composite widgets whose sole purpose is to contain and organize other widgets. There are only two: Frame (a container with a visible 3D border) and Group (an invisible container). Both pass `0` as the data size to `pl_newpanel` because they carry no widget-specific state—their entire purpose is to hold children and participate in layout. The distinction matters for visual structure: use a frame when you want the user to see a group boundary (e.g., a panel of related controls), and a group when you just need to pack several widgets as a unit without any visible separation.

9.1 Frame

A frame is a container that draws a double-relief border around its children—the classic "sunken panel" look. Unlike GTK, which has separate `HBox` and `VBox` containers, libpanel uses the packing flags (`PACKN`, `PACKE`, etc.) on the children themselves to control layout direction, so a single Frame widget serves both purposes. Note the `LEAF` flag is *not* set, allowing children to be added via subsequent `pl_newpanel` calls with this frame as parent.

9.1.1 Initializing

⟨*function* `plframe` 66a⟩≡ (131c)

```
Panel *plframe(Panel *parent, int flags){
    Panel *p;

    p=pl_newpanel(parent, 0); // no widget-specific data
    plinitframe(p, flags);
    return p;
}
```

Uses `pl_newpanel()` 19a and `plinitframe()` 66b.

⟨*function* `plinitframe` 66b⟩≡ (131c)

```
void plinitframe(Panel *v, int flags){
    v->flags=flags;

    v->draw=pl_drawframe;
    v->hit=pl_hitframe;
    v->type=pl_typeframe;

    v->getsize=pl_getsizeframe;
    v->childspace=pl_childspaceframe;

    v->kind="frame";
}
```

Uses `pl_childspaceframe()` 68b, `pl_drawframe()` 67a, `pl_getsizeframe()` 67g, `pl_hitframe()` 67e, and `pl_typeframe()` 67f.

9.1.2 Drawing

⟨*function* `pl_drawframe` 67a⟩≡ (131c)
```
void pl_drawframe(Panel *p){
    pl_box(p->b, p->r, FRAME);
}
```
Uses `FRAME` and `pl_box()` 29c.

⟨`Style` *other cases* 67b⟩+≡ (22a) ◁45c 84c▷
```
// for frames
FRAME
```

`FRAME` is not a real widget state like `UP` or `DOWN`—it is a style constant used only by `pl_box` and `pl_boxoutline` to draw the double-relief border. The border consists of two nested relief bands with opposite light/dark orientation, creating a raised-then-sunken effect:

```
white-black relief (outer, FWID=2 pixels)
  black-white relief (inner, FWID=2 pixels)
    light fill (interior)
```

⟨`pl_boxoutline()` *switch style cases* 67c⟩+≡ (30a) ◁50a
```
case FRAME:
    pl_relief(b, pl_white, pl_black, r, FWID);
    r=insetrect(r, FWID);
    pl_relief(b, pl_black, pl_white, r, FWID);
    r=insetrect(r, FWID);
    if(fill)
        draw(b, r, pl_light, nil, ZP);
    else
        border(b, r, SPACE, pl_white, ZP);
    break;
```
Uses `FRAME`, `FWID-15` 67d, `SPACE-16` 30b, `pl_black-26` 26c, `pl_light-24` 26c, `pl_relief()` 29a, and `pl_white-23` 26c.

⟨*constant* `FWID` 67d⟩≡ (127c)
```
#define FWID 2 /* width of frame relief */
```

9.1.3 Reacting

⟨*function* `pl_hitframe` 67e⟩≡ (131c)
```
bool pl_hitframe(Panel *p, Mouse *m){
    USED(p, m);
    return false;
}
```

⟨*function* `pl_typeframe` 67f⟩≡ (131c)
```
void pl_typeframe(Panel *p, Rune c){
    USED(p, c);
}
```

9.1.4 Packing methods

⟨*function* `pl_getsizeframe` 67g⟩≡ (131c)
```
Vector pl_getsizeframe(Panel *p, Vector children){
    USED(p);
    return pl_boxsize(children, FRAME);
}
```
Uses `FRAME` and `pl_boxsize()` 30c.

⟨pl_boxsize() *switch state cases* 68a⟩+≡ (30c) ◁52d
```
case FRAME:
    return addpt(interior, Pt(4*FWID+2*SPACE, 4*FWID+2*SPACE));
```
Uses FRAME, FWID-15 67d, and SPACE-16 30b.

⟨*function* pl_childspaceframe 68b⟩≡ (131c)
```
void pl_childspaceframe(Panel *p, Point *ul, Vector *size){
    USED(p);
    pl_interior(FRAME, ul, size);
}
```
Uses FRAME and pl_interior() 30d.

⟨pl_interior() *switch state cases* 68c⟩+≡ (30d) ◁30f
```
case FRAME:
    *ul=addpt(*ul, Pt(2*FWID+SPACE, 2*FWID+SPACE));
    *size=subpt(*size, Pt(4*FWID+2*SPACE, 4*FWID+2*SPACE));
```
Uses FRAME, FWID-15 67d, and SPACE-16 30b.

9.2 Group

A group is the minimal composite widget: it contains children but draws nothing and consumes no border space. Its `getsize` returns the children's size unchanged, and `childspace` is a no-op. This is useful when you need a logical grouping of widgets for layout purposes (e.g., packing several widgets as a unit) without any visible decoration.

9.2.1 Initializing

⟨*function* plgroup 68d⟩≡ (131d)
```
Panel *plgroup(Panel *parent, int flags){
    Panel *p;

    p=pl_newpanel(parent, 0);
    plinitgroup(p, flags);
    return p;
}
```
Uses pl_newpanel() 19a and plinitgroup() 68e.

⟨*function* plinitgroup 68e⟩≡ (131d)
```
void plinitgroup(Panel *v, int flags){
    v->flags=flags;

    v->draw=pl_drawgroup;
    v->hit=pl_hitgroup;
    v->type=pl_typegroup;

    v->getsize=pl_getsizegroup;
    v->childspace=pl_childspacegroup;

    v->kind="group";
}
```
Uses pl_childspacegroup() 69e, pl_drawgroup() 69a, pl_getsizegroup() 69d, pl_hitgroup() 69b, and pl_typegroup() 69c.

9.2.2 Drawing

⟨*function* `pl_drawgroup` 69a⟩≡ (131d)
```
void pl_drawgroup(Panel *p){
    USED(p);
}
```

9.2.3 Reacting

⟨*function* `pl_hitgroup` 69b⟩≡ (131d)
```
bool pl_hitgroup(Panel *p, Mouse *m){
    USED(p, m);
    return false;
}
```

⟨*function* `pl_typegroup` 69c⟩≡ (131d)
```
void pl_typegroup(Panel *p, Rune c){
    USED(p, c);
}
```

9.2.4 Packing methods

⟨*function* `pl_getsizegroup` 69d⟩≡ (131d)
```
Vector pl_getsizegroup(Panel *p, Vector children){
    USED(p);
    return children;
}
```

⟨*function* `pl_childspacegroup` 69e⟩≡ (131d)
```
void pl_childspacegroup(Panel *p, Point *ul, Vector *size){
    USED(p, ul, size);
}
```

Chapter 10

Scrollable Widgets

Scrolling is one of the more interesting design problems in a widget library because it requires two-way communication between a pair of widgets: the scrollee (the widget with more content than screen space) and the scroll bar (the widget the user drags to navigate). The scroll bar must tell the scrollee "show a different portion," and the scrollee must tell the scroll bar "I am now showing this portion of the total." In `libpanel`, this communication is wired up with a single call to `plscroll()`[71a], which cross-links the two widgets via pointer fields in `Panel`[18]. After linking, the scrollee and scroll bar communicate through method pointers: `scroll` (from bar to scrollee) and `setscrollbar` (from scrollee to bar).

Unlike GTK, where any widget can be made scrollable by placing it inside a `GtkScrolledWindow`, `libpanel` only supports scrolling for a few specific widget types—lists and text widgets—that implement the `scroll` method. This keeps the design simple but less flexible.

10.1 Scrolling fields

⟨`Panel` *other fields* 70a⟩+≡ (18) ◁31b 71b▷
```
// option<ref<Panel>>
Panel *scrollee;    /* pointer to scrolled window */
// option<ref<Panel>>
Panel *xscroller, *yscroller;   /* pointers to scroll bars */
```

⟨`pl_newpanel()` *set other fields* 70b⟩+≡ (19a) ◁36d 71d▷
```
v->scrollee=nil;
v->xscroller=nil;
v->yscroller=nil;
```

`plscroll()`[71a] establishes the bidirectional link between a scrollee and its scroll bar(s). After this call, the scrollee knows which scroll bars to update (via `xscroller`/`yscroller`), and each scroll bar knows which widget to scroll (via `scrollee`). Either scroller can be `nil` if scrolling is not needed in that direction—in practice, `xscroller` is almost always `nil` since no current widget scrolls horizontally.

```
plscroll(list, nil, scrollbar):

list                        scrollbar
+------------------+        +--------+
| yscroller -------+------->|        |
|                  |<-------+--scrollee
| scroll()         |        | setscrollbar()
+------------------+        +--------+
```

⟨*function* `plscroll` 71a⟩≡ (137c)
```
void plscroll(Panel *scrollee, Panel *xscroller, Panel *yscroller){
    scrollee->xscroller=xscroller;
    scrollee->yscroller=yscroller;
    if(xscroller) xscroller->scrollee=scrollee;
    if(yscroller) yscroller->scrollee=scrollee;
}
```

⟨`Panel` *other fields* 71b⟩+≡ (18) ◁70a
```
Scroll scr;     /* scroll data */
```

⟨*struct* `Scroll` 71c⟩≡ (121d)
```
struct Scroll{
    Point pos;
    Vector size;
};
```

⟨`pl_newpanel()` *set other fields* 71d⟩+≡ (19a) ◁70b
```
v->scr.pos=Pt(0,0);
v->scr.size=Pt(0,0);
```

10.2 Scrolling methods

The two method pointers below implement the two-way protocol. `scroll` is called *on the scrollee* by the scroll bar to say "the user clicked at position `val` of `len` in the bar." The scrollee responds by adjusting what it displays. `setscrollbar` is called *on the scroll bar* by the scrollee to say "I am now showing items `lo` through `hi` out of `len` total." The scroll bar responds by redrawing its thumb.

⟨`Panel` *other methods* 71e⟩+≡ (18) ◁33b 108a▷
```
void (*scroll)(Panel *, int, buttons, int, int); /* scroll bar to scrollee */
void (*setscrollbar)(Panel *, int, int, int); /* scrollee to scroll bar */
```

⟨`pl_newpanel()` *set default methods* 71f⟩+≡ (19a) ◁41b 108b▷
```
v->scroll=pl_scrollerror;
v->setscrollbar=pl_setscrollbarerror;
```
Uses `pl_scrollerror()` 119a and `pl_setscrollbarerror()` 119b.

10.3 Scrollable list

A list displays a scrollable column of text strings. Rather than storing the strings itself, it calls a user-supplied `gen` callback to fetch each item by index—a pattern sometimes called a virtual list. This means the list can display data of arbitrary size without copying it into a widget-owned buffer. The list tracks which range of items is visible (`lo` to the number that fit on screen) and which item is selected (`sel`).

⟨*struct* `List` 71g⟩≡ (132a)
```
struct List{
    int len;   /* # of items in list */
    Vector minsize;

    int lo;    /* indices of first, last items displayed */
    // option<int> (NONE = -1)
    int sel;   /* index of hilited item */
    ⟨List other fields 72a⟩

    char* (*gen)(Panel *, int); /* return text given index or 0 if out of range */
    void (*hit)(Panel *, int, int); /* call user back on hit */

};
```

⟨List *other fields* 72a⟩≡ (71g) 73c▷
```
buttons buttons;
```

10.3.1 Initializing

⟨*function* pllist 72b⟩≡ (132a)
```
Panel *pllist(Panel *parent, int flags, char *(*gen)(Panel *, int), int nlist, void (*hit)(Panel *, int, int)){
    Panel *v;

    v=pl_newpanel(parent, sizeof(List));
    plinitlist(v, flags, gen, nlist, hit);
    return v;
}
```
Uses pl_newpanel() 19a and plinitlist() 72c.

⟨*function* plinitlist 72c⟩≡ (132a)
```
void plinitlist(Panel *v, int flags, char *(*gen)(Panel *, int), int nlist, void (*hit)(Panel *, int, int)){
    List *lp = v->data;
    ⟨plinitlist() other locals 72e⟩

    v->flags=flags|LEAF;
    v->state=UP;

    v->draw=pl_drawlist;
    v->hit=pl_hitlist;
    v->type=pl_typelist;

    v->getsize=pl_getsizelist;
    v->childspace=pl_childspacelist;

    ⟨plinitlist() set fields in lp 72d⟩
    ⟨plinitlist() set scrolling fields 76a⟩

    v->kind="list";
}
```
Uses LEAF, UP 22a, pl_childspacelist() 75c, pl_drawlist() 73d, pl_getsizelist() 75b, pl_hitlist() 74f, and pl_typelist() 75a.

Initialization probes the gen callback to count the total number of items (lp->len) and, unless FILLX or EXPAND is set, measures the widest string to determine the minimum width. The minimum height is nlist * font->height, where nlist is the number of visible items requested by the caller.

⟨plinitlist() *set fields in* lp 72d⟩≡ (72c) 72f▷
```
lp->gen=gen;
lp->hit=hit;
```

⟨plinitlist() *other locals* 72e⟩≡ (72c)
```
int wid, max;
char *str;
```

⟨plinitlist() *set fields in* lp 72f⟩+≡ (72c) ◁72d 73b▷
```
⟨plinitlist() if FILLX or EXPAND 73a⟩
else{
    max=0;
    for(lp->len=0;str=gen(v, lp->len);lp->len++){
        wid=stringwidth(font, str);
        if(wid>max) max=wid;
    }
    lp->minsize=Pt(max, nlist*font->height);
}
```

⟨plinitlist() *if* FILLX *or* EXPAND 73a⟩≡ (72f)
```
if(flags&(FILLX|EXPAND)){
    for(lp->len=0;gen(v, lp->len);lp->len++)
        ;
    lp->minsize=Pt(0, nlist*font->height);
}
```

⟨plinitlist() *set fields in* lp 73b⟩+≡ (72c) ◁72f
```
lp->lo=0;
lp->sel=-1;
```

10.3.2 Drawing

Drawing iterates from item `lo` to however many items fit in the visible rectangle, calling `gen` for each item's text and rendering it with `pl_drawicon()`[46b]. The selected item, if visible, gets a translucent highlight overlay using `pl_highlight()`[74b].

⟨List *other fields* 73c⟩+≡ (71g) ◁72a
```
Rectangle listr;
```

⟨*function* pl_drawlist 73d⟩≡ (132a)
```
void pl_drawlist(Panel *p){
    List *lp = p->data;

    lp->listr=pl_box(p->b, p->r, UP);
    pl_liststrings(p, lp->lo, lp->lo+(lp->listr.max.y-lp->listr.min.y)/font->height,
        lp->listr);
}
```
Uses UP 22a, pl_box() 29c, and pl_liststrings() 73e.

⟨*function* pl_liststrings 73e⟩≡ (132a)
```
void pl_liststrings(Panel *p, int lo, int hi, Rectangle r){
    List *lp = p->data;
    char *s;
    int i;
    ⟨pl_liststrings() other locals 74d⟩

    for(i=lo;i!=hi && (s=lp->gen(p, i));i++){
        r.max.y=r.min.y+font->height;
        pl_drawicon(p->b, r, PLACEW, NOFLAG, s);
        r.min.y+=font->height;
    }
    ⟨pl_liststrings() draw the selection 73f⟩
    ⟨pl_liststrings() set the scrollbar 74e⟩
}
```
Uses pl_drawicon() 46b.

⟨pl_liststrings() *draw the selection* 73f⟩≡ (73e)
```
if(lo<=lp->sel && lp->sel<hi)
    pl_listsel(p, lp->sel, true);
```
Uses pl_listsel() 74a.

⟨*function* pl_listsel 74a⟩≡ (132a)
```
void pl_listsel(Panel *p, int sel, bool on){
    List *lp = p->data;
    int hi;
    Rectangle r;

    hi=lp->lo+(lp->listr.max.y-lp->listr.min.y)/font->height;
    if(lp->lo>=0 && lp->lo<=sel && sel<hi && sel<lp->len){
        r=lp->listr;
        r.min.y+=(sel-lp->lo)*font->height;
        r.max.y=r.min.y+font->height;
        if(on)
            pl_highlight(p->b, r);
        else{
            pl_fill(p->b, r);
            pl_drawicon(p->b, r, PLACEW, NOFLAG, lp->gen(p, sel));
        }
    }
}
```
Uses pl_drawicon() 46b, pl_fill() 74c, and pl_highlight() 74b.

⟨*function* pl_highlight 74b⟩≡ (127c)
```
void pl_highlight(Image *b, Rectangle r){
    draw(b, r, pl_dark, pl_hilit, ZP);
}
```
Uses pl_dark-25 26c and pl_hilit-27 26c.

⟨*function* pl_fill 74c⟩≡ (127c)
```
void pl_fill(Image *b, Rectangle r){
    draw(b, r, plldepth==0? pl_white : pl_light, 0, ZP);
}
```
Uses pl_light-24 26c, pl_white-23 26c, and plldepth-22 26b.

Scrollee to scrollbar

After drawing the visible items, the list notifies its scroll bar (if linked) of the current viewport: items `lo` through `hi` out of `len` total. This is the scrollee-to-scrollbar direction of the protocol—the scroll bar will convert these item indices into pixel positions for its thumb.

⟨pl_liststrings() *other locals* 74d⟩≡ (73e)
```
Panel *sb = p->yscroller;
```

⟨pl_liststrings() *set the scrollbar* 74e⟩≡ (73e)
```
if(sb && sb->setscrollbar)
    sb->setscrollbar(sb, lp->lo,
        lp->lo+(lp->listr.max.y-lp->listr.min.y)/font->height, lp->len);
```

10.3.3 Reacting

The list's hit handler converts the mouse Y coordinate into an item index: `(m->xy.y - ul.y) / font->height + lo`. As the user drags, the selection highlight follows the mouse in real time (old selection unhighlighted, new one highlighted). The user callback fires only on button release, like a regular button.

⟨*function* pl_hitlist 74f⟩≡ (132a)
```
int pl_hitlist(Panel *p, Mouse *m){
    int oldsel;
    bool hitme;
```

```
    Point ul;
    Vector size;
    List *lp = p->data;

    hitme=false;
    ul=p->r.min;
    size=subpt(p->r.max, p->r.min);
    pl_interior(p->state, &ul, &size);
    oldsel=lp->sel;

    if(m->buttons&OUT){
        p->state=UP;
        if(m->buttons&~OUT) lp->sel=-1;
    }
    else if(p->state==DOWN || m->buttons&7){
        lp->sel=(m->xy.y-ul.y)/font->height+lp->lo;
        if(m->buttons&7){
            lp->buttons=m->buttons;
            p->state=DOWN;
        }
        else{ // release
            hitme=true;
            p->state=UP;
        }
    }

    if(oldsel!=lp->sel){
        pl_listsel(p, oldsel, false);
        pl_listsel(p, lp->sel, true);
    }
    if(hitme && 0<=lp->sel && lp->sel<lp->len && lp->hit)
        // user callback
        lp->hit(p, lp->buttons, lp->sel);
    return false;
}
```

Uses DOWN 45c, UP 22a, pl_interior() 30d, and pl_listsel() 74a.

⟨*function* pl_typelist 75a⟩≡ (132a)

```
void pl_typelist(Panel *g, Rune c){
    USED(g, c);
}
```

10.3.4 Packing methods

⟨*function* pl_getsizelist 75b⟩≡ (132a)

```
Vector pl_getsizelist(Panel *p, Vector children){
    USED(children);
    return pl_boxsize(((List *)p->data)->minsize, p->state);
}
```

Uses pl_boxsize() 30c.

⟨*function* pl_childspacelist 75c⟩≡ (132a)

```
void pl_childspacelist(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

10.3.5 Scrollbar to scrollee

This is the other direction of the protocol: when the user clicks the scroll bar, the bar calls `scrollee->scroll()`, which for a list is `pl_scrolllist()`[76b]. The three mouse buttons have different semantics, following the Plan 9 convention: button 1 (left) scrolls up, button 3 (right) scrolls down, and button 2 (middle) jumps to an absolute position. After adjusting `lo`, the function tries to optimize redrawing: if the old and new viewports overlap, it copies the overlapping portion with `pl_cpy()`[77c] (a `draw()`-based blit) and only redraws the newly exposed strip, avoiding a full repaint.

⟨`plinitlist()` *set scrolling fields* 76a⟩≡ (72c)
```
v->scroll=pl_scrolllist;
v->scr.pos=Pt(0,0);
v->scr.size=Pt(0,lp->len);
```
Uses `pl_scrolllist()` 76b.

⟨*function* `pl_scrolllist` 76b⟩≡ (132a)
```
void pl_scrolllist(Panel *p, int dir, buttons buttons, int val, int len){
    List *lp = p->data;
    Point ul;
    Vector size;
    int nlist, nline;
    int oldlo, hi;
    Rectangle r;
    int y;

    oldlo=lp->lo;

    ul=p->r.min;
    size=subpt(p->r.max, p->r.min);
    pl_interior(p->state, &ul, &size);

    nlist=size.y/font->height;

    if(dir==VERT) switch(buttons){
    case CLICK_LEFT:    lp->lo-=nlist*val/len; break;
    case CLICK_MIDDLE:  lp->lo=lp->len*val/len; break;
    case CLICK_RIGHT: lp->lo+=nlist*val/len; break;
    }
    ⟨pl_scrolllist() sanitize lp->lo 77a⟩
    if(lp->lo==oldlo)
        return;
    // else

    p->scr.pos.y=lp->lo;
    r=lp->listr;
    nline=(r.max.y-r.min.y)/font->height;
    hi=lp->lo+nline;

    if(hi<=oldlo || lp->lo>=oldlo+nline){
        // =~ pl_drawlist(), complete redraw
        pl_box(p->b, r, PASSIVE); //BUG? forgot r=pl_box(...)
        pl_liststrings(p, lp->lo, hi, r);
    }
    else
        ⟨pl_scrolllist() optimized drawing by copying 77b⟩
}
```
Uses `PASSIVE` 45c, `VERT`, `pl_box()` 29c, `pl_interior()` 30d, and `pl_liststrings()` 73e.

⟨pl_scrolllist() *sanitize* lp->lo 77a⟩≡ (76b)
```
if(lp->lo<0) lp->lo=0;
if(lp->lo>=lp->len) lp->lo=lp->len-1;
```

⟨pl_scrolllist() *optimized drawing by copying* 77b⟩≡ (76b)
```
if(lp->lo<oldlo){
    y=r.min.y+(oldlo-lp->lo)*font->height;
    pl_cpy(p->b, Pt(r.min.x, y),
        Rect(r.min.x, r.min.y, r.max.x, r.min.y+(hi-oldlo)*font->height));
    r.max.y=y;
    pl_box(p->b, r, PASSIVE);
    pl_liststrings(p, lp->lo, oldlo, r);
}
else{
    pl_cpy(p->b, r.min, Rect(r.min.x, r.min.y+(lp->lo-oldlo)*font->height,
        r.max.x, r.max.y));
    r.min.y=r.min.y+(oldlo+nline-lp->lo)*font->height;
    pl_box(p->b, r, PASSIVE);
    pl_liststrings(p, oldlo+nline, hi, r);
}
```
Uses PASSIVE 45c, pl_box() 29c, pl_cpy() 77c, and pl_liststrings() 73e.

⟨*function* pl_cpy 77c⟩≡ (127c)
```
void pl_cpy(Image *b, Point dst, Rectangle src){
    draw(b, Rpt(dst, addpt(dst, subpt(src.max, src.min))), b, nil, src.min);
}
```

10.4 Scroll bar

The scroll bar is visually similar to a slider—both display a dark thumb on a light track using pl_sliderupd() 61b—but the interaction model is very different. A slider has a single value and calls the application directly; a scroll bar has no value of its own and instead acts as a relay, forwarding user clicks to its linked scrollee. The scroll bar's direction is inferred from its packing flags: packed north or south means horizontal, packed east or west means vertical. The FILLX or FILLY flag is automatically added so the bar stretches along its entire edge.

⟨*struct* Scrollbar 77d⟩≡ (137d)
```
struct Scrollbar{
    // enum<Direction>
    int dir;  /* HORIZ or VERT */
    Point minsize;

    int lo, hi;  /* setting, in screen coordinates */

    ⟨Scrollbar other fields 77e⟩
};
```

⟨Scrollbar *other fields* 77e⟩≡ (77d) 79a▷
```
buttons buttons;  /* saved mouse buttons for transmittal to scrollee */
```

10.4.1 Initializing

⟨*function* plscrollbar 77f⟩≡ (137d)
```
Panel *plscrollbar(Panel *parent, int flags){
    Panel *v;

    v=pl_newpanel(parent, sizeof(Scrollbar));
```

```
    plinitscrollbar(v, flags);
    return v;
}
```
Uses pl_newpanel() 19a and plinitscrollbar() 78a.

⟨*function* plinitscrollbar 78a⟩≡ (137d)
```
void plinitscrollbar(Panel *v, int flags){
    Scrollbar *sp = v->data;

    v->flags=flags|LEAF;
    v->state=UP;

    v->draw=pl_drawscrollbar;
    v->hit=pl_hitscrollbar;
    v->type=pl_typescrollbar;

    v->getsize=pl_getsizescrollbar;
    v->childspace=pl_childspacescrollbar;


    ⟨plinitscrollbar() set extra fields 78b⟩

    v->kind="scrollbar";
}
```
Uses LEAF, UP 22a, pl_childspacescrollbar() 80e, pl_drawscrollbar() 79b, pl_getsizescrollbar() 80d, pl_hitscrollbar() 79c, and pl_typescrollbar() 80c.

The scroll bar sets a high priority (PRI_SCROLLBAR) for hit testing. This ensures that when the scroll bar overlaps its scrollee's edge (which it always does, since they are packed as siblings), the scroll bar wins the hit test. Without this, clicks near the bar's edge might go to the scrollee instead.

⟨plinitscrollbar() set extra fields 78b⟩≡ (78a) 78c▷
```
v->pri=pl_priscrollbar; // !!
```
Uses pl_priscrollbar() 33i.

⟨plinitscrollbar() set extra fields 78c⟩+≡ (78a) ◁78b 78e▷
```
switch(flags&PACK){
case PACKN:
case PACKS:
    sp->dir=HORIZ;
    sp->minsize=Pt(0, SBWID);
    v->flags|=FILLX;
    break;
case PACKE:
case PACKW:
    sp->dir=VERT;
    sp->minsize=Pt(SBWID, 0);
    v->flags|=FILLY;
    break;
}
```
Uses HORIZ 125c, SBWID-2 78d, and VERT.

⟨*constant* SBWID 78d⟩≡ (137d)
```
#define SBWID 15 /* should come from draw.c? */
```

⟨plinitscrollbar() set extra fields 78e⟩+≡ (78a) ◁78c 81a▷
```
sp->lo=0;
sp->hi=0;
```

10.4.2 Drawing

⟨Scrollbar *other fields* 79a⟩+≡ (77d) ◁77e
```
  Rectangle interior;
```

⟨*function* pl_drawscrollbar 79b⟩≡ (137d)
```
  void pl_drawscrollbar(Panel *p){
      Scrollbar *sp = p->data;

      sp->interior=pl_outline(p->b, p->r, p->state);
      pl_sliderupd(p->b, sp->interior, sp->dir, sp->lo, sp->hi);
  }
```
Uses pl_outline() 29b and pl_sliderupd() 61b.

10.4.3 Reacting

The scroll bar hit handler follows the Plan 9 scrolling convention: button 1 and 3 provide visual feedback by temporarily shifting the thumb (using `pl_sliderupd()`[61b]) to preview where the content will scroll to. The actual scroll happens on button *release*, when `scrollee->scroll()` is called. Button 2 is different: it scrolls immediately (absolute jump), calling `scrollee->scroll()` on every mouse movement for real-time feedback. The handler returns `true` when the button is still held down, which triggers the REMOUSE mechanism (see Chapter 6) so the scroll bar keeps receiving events even if the mouse moves outside its rectangle during a drag.

⟨*function* pl_hitscrollbar 79c⟩≡ (137d)
```
  int pl_hitscrollbar(Panel *g, Mouse *m){
      int oldstate;
      int pos, len, dy;
      Point ul;
      Vector size;
      Scrollbar *sp = g->data;

      ul=g->r.min;
      size=subpt(g->r.max, g->r.min);
      pl_interior(g->state, &ul, &size);
      oldstate=g->state;

      if(!(g->flags & USERFL) && (m->buttons&OUT || !ptinrect(m->xy, g->r))){
          m->buttons&=~OUT;
          g->state=UP;
          goto out;
      }

      if(sp->dir==HORIZ){
          pos=m->xy.x-ul.x;
          len=size.x;
      }
      else{
          pos=m->xy.y-ul.y;
          len=size.y;
      }
      ⟨pl_hitscrollbar() sanitize pos 80a⟩

      if(m->buttons&7){
          g->state=DOWN;
          sp->buttons=m->buttons;
          switch(m->buttons){
          ⟨pl_hitscrollbar() switch buttons cases 80b⟩
          }
```

```
    }
    else{
        if(!(sp->buttons&CLICK_MIDDLE) && g->state==DOWN &&
           g->scrollee && g->scrollee->scroll)
            // scroll hook
            g->scrollee->scroll(g->scrollee, sp->dir, sp->buttons,
                pos, len);
        g->state=UP;
    }
out:
    if(oldstate!=g->state) pldraw(g, g->b);
    return g->state==DOWN;
}
```

Uses DOWN 45c, HORIZ 125c, UP 22a, pl_interior() 30d, and pldraw() 28a.

⟨pl_hitscrollbar() *sanitize* pos 80a⟩≡ (79c)

```
if(pos<0) pos=0;
else if(pos>len) pos=len;
```

⟨pl_hitscrollbar() *switch buttons cases* 80b⟩≡ (79c)

```
case CLICK_LEFT:
    dy=pos*(sp->hi-sp->lo)/len;
    pl_sliderupd(g->b, sp->interior, sp->dir, sp->lo-dy,
        sp->hi-dy);
    break;
case CLICK_MIDDLE:
    if(g->scrollee && g->scrollee->scroll)
        g->scrollee->scroll(g->scrollee, sp->dir,
            m->buttons, pos, len);
    break;
case CLICK_RIGHT:
    dy=pos*(sp->hi-sp->lo)/len;
    pl_sliderupd(g->b, sp->interior, sp->dir, sp->lo+dy,
        sp->hi+dy);
    break;
```

Uses pl_sliderupd() 61b.

⟨*function* pl_typescrollbar 80c⟩≡ (137d)

```
void pl_typescrollbar(Panel *p, Rune c){
    USED(p, c);
}
```

10.4.4 Packing methods

⟨*function* pl_getsizescrollbar 80d⟩≡ (137d)

```
Point pl_getsizescrollbar(Panel *p, Point children){
    USED(children);
    return pl_boxsize(((Scrollbar *)p->data)->minsize, p->state);
}
```

Uses pl_boxsize() 30c.

⟨*function* pl_childspacescrollbar 80e⟩≡ (137d)

```
void pl_childspacescrollbar(Panel *p, Point *ul, Vector *size){
    USED(p, ul, size);
}
```

10.4.5 Scrollee to scrollbar

When the scrollee redraws (e.g., after scrolling), it calls `setscrollbar` on its linked scroll bar to update the thumb position. The function converts from the scrollee's logical coordinates (`lo`, `hi`, `len`—item indices for a list) to the scroll bar's pixel coordinates, then redraws the bar.

⟨`plinitscrollbar()` *set extra fields* 81a⟩+≡ (78a) ◁78e
```
v->setscrollbar=pl_setscrollbarscrollbar;
```
Uses `pl_setscrollbarscrollbar()` 81b.

⟨*function* `pl_setscrollbarscrollbar` 81b⟩≡ (137d)
```
/*
 * Arguments lo, hi and len are in the scrollee's natural coordinates
 */
void pl_setscrollbarscrollbar(Panel *p, int lo, int hi, int len){
    Point ul;
    Vector size;
    int mylen;
    Scrollbar *sp = p->data;

    ul=p->r.min;
    size=subpt(p->r.max, p->r.min);
    pl_interior(p->state, &ul, &size);

    mylen=sp->dir==HORIZ?size.x:size.y;

    if(len==0) len=1;
    sp->lo=lo*mylen/len;
    sp->hi=hi*mylen/len;

    ⟨pl_setscrollbarscrollbar() sanitize sp->lo 81c⟩
    ⟨pl_setscrollbarscrollbar() sanitize sp->hi 81d⟩
    pldraw(p, p->b);
}
```
Uses `HORIZ` 125c, `pl_interior()` 30d, and `pldraw()` 28a.

⟨`pl_setscrollbarscrollbar()` *sanitize* `sp->lo` 81c⟩≡ (81b)
```
if(sp->lo<0) sp->lo=0;
if(sp->lo>=mylen) sp->hi=mylen-1;
```

⟨`pl_setscrollbarscrollbar()` *sanitize* `sp->hi` 81d⟩≡ (81b)
```
if(sp->hi<=sp->lo) sp->hi=sp->lo+1;
if(sp->hi>mylen) sp->hi=mylen;
```

Chapter 11

Menu widgets

Menus are built from the widget primitives already covered. A menu is simply a group of buttons; a popup menu is a container that shows one of three menus depending on which mouse button is pressed; a pull-down is a button that reveals a menu panel when clicked; and a menu bar is a group of pull-downs. This layered construction keeps each piece simple while enabling the full range of menu behaviors.

11.1 Menu items

A menu is not a new widget type—it is a group of regular buttons with two extra fields. Each button stores its `index` in the menu's item array and a `menuhit` callback that receives (buttons, index) instead of (panel, buttons). This indirection lets the application use a single callback for all menu items, distinguishing them by index.

⟨`Button` *other fields* 82a⟩+≡ (53a) ◁53f
```
int index;   /* arg to menuhit */
```

⟨`Button` *other methods* 82b⟩≡ (53a)
```
void (*menuhit)(buttons, int); /* call back user code on menu item hit */
```

11.1.1 Initializing

`plmenu()`[82c] creates a group and populates it with buttons, one per item in the null-terminated `item` array. Each button uses `pl_hitmenu()`[83b] as its hit callback (not the user's callback directly), which extracts the `index` and `menuhit` from the button's data and calls the user's function. The `cflags` argument controls how the buttons are packed within the group—typically `PACKN|FILLX` for a vertical menu or `PACKW` for a horizontal row.

⟨*function* `plmenu` 82c⟩≡ (129a)
```
Panel *plmenu(Panel *parent, int flags, Icon **item, int cflags, void (*hit)(buttons, int)){
    Panel *v;

    v=plgroup(parent, flags);
    plinitmenu(v, flags, item, cflags, hit);
    return v;
}
```
Uses `plgroup()` 68d and `plinitmenu()` 82d.

⟨*function* `plinitmenu` 82d⟩≡ (129a)
```
void plinitmenu(Panel *v, int flags, Icon **item, int cflags, void (*hit)(buttons, int)){
    Panel *p;
    Button* b;
    int i;
```

```
    v->flags=flags;

    ⟨plinitmenu() free child widget if any 83a⟩

    for(i=0;item[i];i++){
        p=plbutton(v, cflags, item[i], pl_hitmenu);
        b = p->data;

        b->menuhit=hit;
        b->index=i;
    }
    v->kind="menu";
}
```

Uses pl_hitmenu() 83b and plbutton() 54a.

⟨plinitmenu() *free child widget if any* 83a⟩≡ (82d)

```
if(v->child){
    plfree(v->child);
    v->child=nil;
}
```

Uses plfree() 19c.

11.1.2 Drawing

11.1.3 Reacting

⟨*function* pl_hitmenu 83b⟩≡ (129a)

```
void pl_hitmenu(Panel *p, buttons buttons){
    Button* b = p->data;
    void (*hit)(int, int) = b->menuhit;

    if(hit)
        hit(buttons, b->index);
}
```

11.2 Popup menu

A popup is an invisible container that intercepts mouse clicks and temporarily displays one of three menu panels depending on which button is pressed—one panel per mouse button, following the Plan 9 convention. The pop array stores the three panels (any of which may be nil to disable that button). The key challenge is that the popup menu must appear on top of existing content and disappear cleanly. The popup solves this by saving the screen area it will cover into save, drawing the menu over it, and restoring the saved image when the menu is dismissed.

⟨*struct* Popup 83c⟩≡ (133a)

```
struct Popup{
    Panel *pop[3];   /* what to pop up */

    //option<ref_own<Image>>
    Image *save;   /* where to save what the popup covers */
};
```

The typical usage pattern is that the popup is the *parent* of the main content widget. For example, in a text editor, the popup contains the edit panel as its child, and the menus as its pop entries. When no button is pressed, mouse events pass through to the child. The popup uses PRI_POPUP priority so that hit testing checks it before its child.

11.2.1 Initializing

⟨*function* `plpopup` 84a⟩≡ (133a)
```
Panel *plpopup(Panel *parent, int flags, Panel *pop0, Panel *pop1, Panel *pop2){
    Panel *v;

    v=pl_newpanel(parent, sizeof(Popup));
    plinitpopup(v, flags, pop0, pop1, pop2);
    return v;
}
```
Uses `pl_newpanel()` 19a and `plinitpopup()` 84b.

⟨*function* `plinitpopup` 84b⟩≡ (133a)
```
void plinitpopup(Panel *v, int flags, Panel *pop0, Panel *pop1, Panel *pop2){
    Popup *pp =v->data;

    v->flags=flags;
    v->state=UP;

    v->draw=pl_drawpopup;
    v->hit=pl_hitpopup;
    v->type=pl_typepopup;

    v->getsize=pl_getsizepopup;
    v->childspace=pl_childspacepopup;

    v->pri=pl_pripopup;

    pp->pop[0]=pop0;
    pp->pop[1]=pop1;
    pp->pop[2]=pop2;
    pp->save=nil;

    v->kind="popup";
}
```
Uses `UP` 22a, `pl_childspacepopup()` 87e, `pl_drawpopup()` 85e, `pl_getsizepopup()` 87d, `pl_hitpopup()` 84d, `pl_pripopup()` 33h, and `pl_typepopup()` 85d.

11.2.2 Reacting

The popup hit handler is a state machine. In the `UP` state, it checks which button was pressed and selects the corresponding menu from `pop[0..2]`. It then packs the menu into the screen rectangle, positions it centered on the mouse cursor (using `plmove()`[86b]), saves the area it covers, and draws the menu. The state transitions to `DOWN1`, `DOWN2`, or `DOWN3` to remember which button triggered the popup. While in a `DOWNX` state, subsequent mouse events are dispatched to the menu panel (via a recursive `plmouse()`[31a] call). When the button is released, the saved image is restored, hiding the menu cleanly.

⟨`Style` *other cases* 84c⟩+≡ (22a) ◁67b
```
DOWN1,
DOWN2,
DOWN3,
```

⟨*function* `pl_hitpopup` 84d⟩≡ (133a)
```
bool pl_hitpopup(Panel *g, Mouse *m){
    Panel *p;
    Popup *pp = g->data;
    ⟨pl_hitpopup() other locals 85f⟩
```

```
    if(g->state==UP){
        switch(m->buttons&7){
        case CLICK_LEFT:   p=pp->pop[0]; g->state=DOWN1; break;
        case CLICK_MIDDLE: p=pp->pop[1]; g->state=DOWN2; break;
        case CLICK_RIGHT:  p=pp->pop[2]; g->state=DOWN3; break;
        ⟨pl_hitpopup() when UP, switch buttons other cases 85a⟩
        }
        ⟨pl_hitpopup() when UP, if no popup menu p 85c⟩
        else if(g->state!=UP){
            ⟨pl_hitpopup() when from UP to DOWNX and valid p 86a⟩
        }
    } else{ // a DOWNX state
        switch(g->state){
        case DOWN1: p=pp->pop[0]; break;
        case DOWN2: p=pp->pop[1]; break;
        case DOWN3: p=pp->pop[2]; break;
        case DOWN:  p=g->child;  break;
        default: SET(p); break;   /* can't happen! */
        }
        ⟨pl_hitpopup() when DOWNX state, if no buttons 86e⟩
    }
    // redispatch mouse event to appropriate menu or child
    plmouse(p, m);

    if((m->buttons&7)==0)
        g->state=UP;
    return (m->buttons&7)!=0;
}
```

Uses DOWN 45c, DOWN1 22a, DOWN2 22a, DOWN3, UP 22a, and plmouse() 31a.

⟨pl_hitpopup() *when* UP, *switch buttons other cases* 85a⟩≡ (84d) 85b▷

```
// just a mouse motion
case 0: p=g->child; break;
```

⟨pl_hitpopup() *when* UP, *switch buttons other cases* 85b⟩+≡ (84d) ◁85a

```
default: p=nil; break;
```

⟨pl_hitpopup() *when* UP, *if no popup menu* p 85c⟩≡ (84d)

```
if(p==nil){
    p=g->child;
    g->state=DOWN;
}
```

Uses DOWN 45c.

⟨*function* pl_typepopup 85d⟩≡ (133a)

```
void pl_typepopup(Panel *g, Rune c){
    USED(g, c);
}
```

11.2.3 Drawing

⟨*function* pl_drawpopup 85e⟩≡ (133a)

```
void pl_drawpopup(Panel *p){
    USED(p);
}
```

⟨pl_hitpopup() *other locals* 85f⟩≡ (84d)

```
Vector d; // delta
```

⟨pl_hitpopup() *when from* UP *to* DOWNX *and valid* p 86a⟩≡ (84d)
```
plpack(p, view->clipr);
⟨pl_hitpopup() when from UP to DOWNX, compute d 86c⟩
plmove(p, d);
pp->save=allocimage(display, p->r, g->b->chan, false, DNofill);
if(pp->save!=nil)
    draw(pp->save, p->r, g->b, nil, p->r.min);
pl_invis(p, false);
pldraw(p, g->b);
```
Uses pl_invis() 87a, pldraw() 28a, plmove() 86b, and plpack() 35.

⟨*function* plmove 86b⟩≡ (132c)
```
/*
 * move an already-packed panel so that p->r=raddp(p->r, d)
 */
void plmove(Panel *p, Point d){
    ⟨plmove() if edit widget special case 86d⟩
    p->r=rectaddpt(p->r, d);
    for(p=p->child;p;p=p->next)
        plmove(p, d);
}
```
Uses plmove() 86b.

⟨pl_hitpopup() *when from* UP *to* DOWNX, *compute* d 86c⟩≡ (86a)
```
if(p->lastmouse)
    d=subpt(m->xy, divpt(addpt(p->lastmouse->r.min,
                p->lastmouse->r.max), 2));
else
    d=subpt(m->xy, divpt(addpt(p->r.min, p->r.max), 2));
if(p->r.min.x+d.x<g->r.min.x) d.x=g->r.min.x-p->r.min.x;
if(p->r.max.x+d.x>g->r.max.x) d.x=g->r.max.x-p->r.max.x;
if(p->r.min.y+d.y<g->r.min.y) d.y=g->r.min.y-p->r.min.y;
if(p->r.max.y+d.y>g->r.max.y) d.y=g->r.max.y-p->r.max.y;
```

⟨plmove() *if edit widget special case* 86d⟩≡ (86b)
```
if(strcmp(p->kind, "edit") == 0) /* sorry */
    plemove(p, d);
```
Uses plemove() 100c.

Hiding

⟨pl_hitpopup() *when* DOWNX *state, if no buttons* 86e⟩≡ (84d)
```
if((m->buttons&7)==0){
    if(g->state!=DOWN){
        if(pp->save!=nil){
            // restore from saved image
            draw(g->b, p->r, pp->save, nil, p->r.min);
            flushimage(display, true);
            freeimage(pp->save);
            pp->save=nil;
        }
        pl_invis(p, true);
    }
    g->state=UP;
}
```
Uses DOWN 45c, UP 22a, and pl_invis() 87a.

⟨*function* `pl_invis` 87a⟩≡ (127c)
```
void pl_invis(Panel *p, bool v){
    for(;p;p=p->next){
        if(v)
            p->flags|=INVIS;
        else
            p->flags&=~INVIS;
        pl_invis(p->child, v);
    }
}
```
Uses `INVIS` and `pl_invis()` 87a.

⟨*constant* `INVIS` 87b⟩≡ (125e)
```
#define INVIS 0x20000  /* don't draw this */
```

⟨`pl_drawall()` *if invisible widget* 87c⟩≡ (28c)
```
if(p->flags&INVIS) return;
```
Uses `INVIS`.

11.2.4 Packing methods

⟨*function* `pl_getsizepopup` 87d⟩≡ (133a)
```
Vector pl_getsizepopup(Panel *g, Vector children){
    USED(g);
    return children;
}
```

⟨*function* `pl_childspacepopup` 87e⟩≡ (133a)
```
void pl_childspacepopup(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

11.3 Pull-down

A pull-down looks like an ordinary button when displayed. When clicked, it reveals a menu panel on one side of the button (controlled by `side`: `PACKN` puts the menu above, `PACKS` below, `PACKE` to the right, `PACKW` to the left). Like the popup, it saves and restores the screen area under the menu. Pull-downs are the building blocks for menu bars: an array of pull-downs packed horizontally across the top of a window, each revealing its menu downward.

⟨*struct* `Pulldown` 87f⟩≡ (133b)
```
struct Pulldown{
    Icon *icon;  /* button label */
    Panel *pull;  /* Panel to pull down */

    // enum<PackingDirection>
    int side;  /* which side of the button to put the panel on */
    //option<ref_own<Image>>
    Image *save;  /* where to save what we draw the panel on */
};
```

11.3.1 Initializing

⟨*function* `plpulldown` 88a⟩≡ (133b)
```
Panel *plpulldown(Panel *parent, int flags, Icon *icon, Panel *pullthis, int side){
    Panel *v;

    v=pl_newpanel(parent, sizeof(Pulldown));
    plinitpulldown(v, flags, icon, pullthis, side);
    return v;
}
```
Uses `pl_newpanel()` 19a and `plinitpulldown()` 88b.

⟨*function* `plinitpulldown` 88b⟩≡ (133b)
```
void plinitpulldown(Panel *v, int flags, Icon *icon, Panel *pullthis, int side){
    Pulldown *pp = v->data;

    v->flags=flags|LEAF;
    v->state=UP;

    v->draw=pl_drawpulldown;
    v->hit=pl_hitpulldown;
    v->type=pl_typepulldown;

    v->getsize=pl_getsizepulldown;
    v->childspace=pl_childspacepulldown;

    pp->icon=icon;
    pp->pull=pullthis;
    pp->side=side;
    pp->save=nil;

    v->kind="pulldown";
}
```
Uses `LEAF`, `UP` 22a, `pl_childspacepulldown()` 90d, `pl_drawpulldown()` 88c, `pl_getsizepulldown()` 90c, `pl_hitpulldown()` 88d, and `pl_typepulldown()` 90b.

11.3.2 Drawing

⟨*function* `pl_drawpulldown` 88c⟩≡ (133b)
```
void pl_drawpulldown(Panel *p){
    Pulldown* pd = p->data;

    pl_drawicon(p->b, pl_box(p->b, p->r, p->state), PLACECEN,
        p->flags, pd->icon);
}
```
Uses `pl_box()` 29c and `pl_drawicon()` 46b.

11.3.3 Reacting

⟨*function* `pl_hitpulldown` 88d⟩≡ (133b)
```
bool pl_hitpulldown(Panel *g, Mouse *m){
    Pulldown *pp = g->data;
    Panel *p = pp->pull;
    Panel *hitme = nil;
    int oldstate;
    Rectangle r;
    bool passon;
```

```
    oldstate=g->state;

    switch(g->state){
    case UP:
        ⟨pl_hitpulldown() when UP, if mouse outside widget 90a⟩
        else
            if(m->buttons&7){
                r=g->b->r;
                p->flags&=~PLACE;
                switch(pp->side){
                ⟨pl_hitpulldown() when UP and buttons, switch side cases 89⟩
                }
                plpack(p, r);
                pp->save=allocimage(display, p->r, g->b->chan, false, DNofill);
                if(pp->save!=nil)
                    draw(pp->save, p->r, g->b, nil, p->r.min);
                pl_invis(p, false);
                pldraw(p, g->b);
                g->state=DOWN;
             }
        break;
    case DOWN:
        if(!ptinrect(m->xy, g->r)){
            switch(pp->side){
            case PACKN: passon=m->xy.y<g->r.min.y; break;
            case PACKS: passon=m->xy.y>=g->r.max.y; break;
            case PACKE: passon=m->xy.x>=g->r.max.x; break;
            case PACKW: passon=m->xy.x<g->r.min.x; break;
            case PACKCEN: passon=true; break;
            default: SET(passon); break;  /* doesn't happen */
            }
            if(passon){
                hitme=p;
                if((m->buttons&7)==0)
                    g->state=UP;
            }
            else g->state=UP;
        }
        else if((m->buttons&7)==0) g->state=UP;
        else hitme=p;

        if(g->state!=DOWN && pp->save){
            draw(g->b, p->r, pp->save, nil, p->r.min);
            freeimage(pp->save);
            pp->save=nil;
            pl_invis(p, true);
            hitme=p;
        }
    }
    if(g->state!=oldstate)
        pldraw(g, g->b);
    if(hitme)
        plmouse(hitme, m);

    return g->state==DOWN;
}
```

Uses DOWN 45c, UP 22a, pl_invis() 87a, pldraw() 28a, plmouse() 31a, and plpack() 35.

⟨pl_hitpulldown() *when* UP *and buttons, switch side cases* 89⟩≡ (88d)

```
case PACKN:
```

```
        r.min.x=g->r.min.x;
        r.max.y=g->r.min.y;
        p->flags|=PLACESW;
        break;
    case PACKS:
        r.min.x=g->r.min.x;
        r.min.y=g->r.max.y;
        p->flags|=PLACENW;
        break;
    case PACKE:
        r.min.x=g->r.max.x;
        r.min.y=g->r.min.y;
        p->flags|=PLACENW;
        break;
    case PACKW:
        r.max.x=g->r.min.x;
        r.min.y=g->r.min.y;
        p->flags|=PLACENE;
        break;
    case PACKCEN:
        r.min=g->r.min;
        p->flags|=PLACENW;
        break;
```

⟨pl_hitpulldown() *when* UP, *if mouse outside widget* 90a⟩≡ (88d)

```
    if(!ptinrect(m->xy, g->r))
        g->state=UP;
```

Uses UP 22a.

⟨*function* pl_typepulldown 90b⟩≡ (133b)

```
    void pl_typepulldown(Panel *p, Rune c){
        USED(p, c);
    }
```

11.3.4 Packing methods

⟨*function* pl_getsizepulldown 90c⟩≡ (133b)

```
    Vector pl_getsizepulldown(Panel *p, Vector children){
        USED(p, children);
        return pl_boxsize(pl_iconsize(p->flags, ((Pulldown *)p->data)->icon), p->state);
    }
```

Uses pl_boxsize() 30c and pl_iconsize() 46c.

⟨*function* pl_childspacepulldown 90d⟩≡ (133b)

```
    void pl_childspacepulldown(Panel *p, Point *ul, Vector *size){
        USED(p, ul, size);
    }
```

11.4 Menu bar

A menu bar is simply a group of pull-down panels. `plmenubar()`[91a] takes a variadic list of (label, menu) pairs, terminated by a `nil` label, and creates one pull-down for each pair. The pull direction is inferred from the packing direction: if the bar items are packed east/west (horizontal bar), the menus drop down (`PACKS`); if packed north/south (vertical bar), the menus cascade to the right (`PACKE`).

11.4.1 Initializing

⟨*function* `plmenubar` 91a⟩≡ (133b)

```
Panel *plmenubar(Panel *parent, int flags, int cflags, Icon *l1, Panel *m1, Icon *l2, ...){
    Panel *v;
    va_list arg;
    Icon *s;
    int pulldir;

    ⟨plmenubar() set pulldir based on cflags 91b⟩
    v=plgroup(parent, flags);

    va_start(arg, cflags);
    while((s=va_arg(arg, Icon *))!=nil)
        plpulldown(v, cflags, s, va_arg(arg, Panel *), pulldir);
    va_end(arg);

    USED(l1, m1, l2); // used for type checking at least the first arg
    v->kind="menubar";
    return v;
}
```

Uses `plgroup()` 68d, `plpulldown()` 88a, and `s`.

⟨`plmenubar()` *set* `pulldir` *based on* `cflags` 91b⟩≡ (91a)

```
switch(cflags&PACK){
case PACKE:
case PACKW:
    pulldir=PACKS;
    break;
case PACKN:
case PACKS:
    pulldir=PACKE;
    break;
default:
    SET(pulldir);
    break;
}
```

Chapter 12

Text Widgets

The basic widgets chapter covered labels (single-line, non-interactive) and text entries (single-line, editable). This chapter presents three more text widgets that handle longer or richer content: Message (multi-line with word wrapping), Edit (scrollable editable text window), and Textview (scrollable formatted text with mixed fonts, images, and embedded panels). The chapter also covers `Rtext`, the linked-list data structure used by Textview to represent formatted text.

12.1 Message

A message widget is like a label, but for longer text: it wraps at word boundaries to fit within a specified width. As Tom Duff notes in the `libpanel` manual, "labels are intended for short pieces of text such as titles; message panels display longer pieces of text, such as error messages or dialog box verbiage, on multiple lines with wrapping at word boundaries." Like labels, messages are non-interactive (hit and type are no-ops).

⟨*struct* `Message` 92a⟩≡ (132b)
```
struct Message{
    // ref_own<string>
    char *text;

    Vector minsize;
};
```

12.1.1 Initializing

⟨*function* `plmessage` 92b⟩≡ (132b)
```
Panel *plmessage(Panel *parent, int flags, int wid, char *msg){
    Panel *v;

    v=pl_newpanel(parent, sizeof(Message));
    plinitmessage(v, flags, wid, msg);
    return v;
}
```
Uses `pl_newpanel()` 19a and `plinitmessage()` 92c.

⟨*function* `plinitmessage` 92c⟩≡ (132b)
```
void plinitmessage(Panel *v, int flags, int wid, char *msg){
    Message *mp = v->data;

    v->flags=flags|LEAF;

    v->draw=pl_drawmessage;
    v->hit=pl_hitmessage;
```

```
    v->type=pl_typemessage;

    v->getsize=pl_getsizemessage;
    v->childspace=pl_childspacemessage;

    mp->text=msg;
    mp->minsize=Pt(wid, font->height);

    v->kind="message";
}
```

Uses LEAF, pl_childspacemessage() 94e, pl_drawmessage() 93a, pl_getsizemessage() 94c, pl_hitmessage() 94a, and pl_typemessage() 94b.

12.1.2 Drawing

The drawing function pl_textmsg()[93b] implements a simple word-wrapping algorithm: it scans forward word by word, accumulating pixel widths, and breaks to a new line when the next word would exceed the available width. If a single word is wider than the entire line, it is placed alone on its line (it may overflow, since messages do not hyphenate).

⟨*function* pl_drawmessage 93a⟩≡ (132b)

```
void pl_drawmessage(Panel *p){
    pl_textmsg(p->b, pl_box(p->b, p->r, PASSIVE), font,
               ((Message *)p->data)->text);
}
```

Uses PASSIVE 45c, pl_box() 29c, and pl_textmsg() 93b.

⟨*function* pl_textmsg 93b⟩≡ (132b)

```
void pl_textmsg(Image *b, Rectangle r, Font *f, char *s){
    char *start, *end; /* of line */
    Point where;
    int lwid, c, wid;

    where=r.min;
    wid=r.max.x-r.min.x;
    do{
        start=s;
        lwid=0;
        end=s;
        do{
            for(;*s!=' ' && *s!='\0';s=pl_nextrune(s)) lwid+=pl_runewidth(f, s);
            if(lwid>wid) break;
            end=s;
            for(;*s==' ';s=pl_nextrune(s)) lwid+=pl_runewidth(f, s);
        }while(*s!='\0');
        if(end==start) /* can't even fit one word on line! */
            end=s;
        c=*end;
        *end='\0';
        string(b, where, display->black, ZP, f, start);
        *end=c;
        where.y+=font->height;
        s=end;
        while(*s==' ') s=pl_nextrune(s);
    }while(*s!='\0');
}
```

Uses pl_nextrune() 150a and pl_runewidth() 150b.

12.1.3 Reacting

⟨*function* pl_hitmessage 94a⟩≡ (132b)
```
bool pl_hitmessage(Panel *g, Mouse *m){
    USED(g, m);
    return false;
}
```

⟨*function* pl_typemessage 94b⟩≡ (132b)
```
void pl_typemessage(Panel *g, Rune c){
    USED(g, c);
}
```

12.1.4 Packing methods

⟨*function* pl_getsizemessage 94c⟩≡ (132b)
```
Vector pl_getsizemessage(Panel *p, Vector children){
    Message *mp = p->data;

    USED(children);
    return pl_boxsize(pl_foldsize(font, mp->text, mp->minsize.x), PASSIVE);
}
```
Uses PASSIVE 45c, pl_boxsize() 30c, and pl_foldsize() 94d.

⟨*function* pl_foldsize 94d⟩≡ (132b)
```
Point pl_foldsize(Font *f, char *s, int wid){
    char *start, *end; /* of line */
    Point size;
    int lwid, ewid;
    size=Pt(0,0);
    do{
        start=s;
        lwid=0;
        end=s;
        ewid=lwid;
        do{
            for(;*s!=' ' && *s!='\0';s=pl_nextrune(s)) lwid+=pl_runewidth(f, s);
            if(lwid>wid) break;
            end=s;
            ewid=lwid;
            for(;*s==' ';s=pl_nextrune(s)) lwid+=pl_runewidth(f, s);
        }while(*s!='\0');
        if(end==start){ /* can't even fit one word on line! */
            ewid=lwid;
            end=s;
        }
        if(ewid>size.x) size.x=ewid;
        size.y+=font->height;
        s=end;
        while(*s==' ') s=pl_nextrune(s);
    }while(*s!='\0');
    return size;
}
```
Uses pl_nextrune() 150a and pl_runewidth() 150b.

⟨*function* pl_childspacemessage 94e⟩≡ (132b)
```
void pl_childspacemessage(Panel *p, Point *ul, Vector *size){
    USED(p, ul, size);
}
```

12.2 Edit

While the message widget handles display-only multi-line text, the edit widget goes further: it is a scrollable, editable text window—the most complex widget in `libpanel`. It delegates most of the heavy lifting to a `Textwin` object (defined in `textwin.c`), which handles the actual text buffer, selection, and rendering. The edit widget itself is a thin wrapper that connects `Textwin` to the `Panel` interface. The edit widget supports mouse-based text selection, keyboard editing (backspace, delete, Escape for cut), chord-based cut/paste (buttons 1+2 for cut, buttons 1+3 for paste), and scrolling via a linked scroll bar. Because `Textwin` does most of the work, the edit widget's methods are short: `pl_drawedit()` creates or reshapes the `Textwin` and highlights the current selection; `pl_hitedit()` delegates to `twselect()` for mouse tracking; `pl_typeedit()` delegates to `twreplace()` for insertions.

⟨*struct* `Edit` 95a⟩≡ (130)
```
struct Edit{
    Point minsize;
    void (*hit)(Panel *);
    int sel0, sel1;
    Textwin *t;
    Rune *text;
    int ntext;
};
```

12.2.1 Initializing

⟨*function* `pledit` 95b⟩≡ (130)
```
Panel *pledit(Panel *parent, int flags, Point minsize, Rune *text, int ntext, void (*hit)(Panel *)){
    Panel *v;

    v=pl_newpanel(parent, sizeof(Edit));
    ((Edit *)v->data)->t=0;
    plinitedit(v, flags, minsize, text, ntext, hit);
    return v;
}
```
Uses `pl_newpanel()` 19a and `plinitedit()` 95c.

⟨*function* `plinitedit` 95c⟩≡ (130)
```
void plinitedit(Panel *v, int flags, Point minsize, Rune *text, int ntext, void (*hit)(Panel *)){
    Edit *ep = v->data;

    v->flags=flags|LEAF;
    v->state=UP;
    v->draw=pl_drawedit;
    v->hit=pl_hitedit;
    v->type=pl_typeedit;
    v->getsize=pl_getsizeedit;
    v->childspace=pl_childspaceedit;
    v->free=pl_freeedit;

    v->snarf=pl_snarfedit;
    v->paste=pl_pasteedit;

    v->kind="edit";
    ep->hit=hit;
    ep->minsize=minsize;
    ep->text=text;
    ep->ntext=ntext;
    if(ep->t!=0) twfree(ep->t);
```

```
    ep->t=0;
    ep->sel0=-1;
    ep->sel1=-1;
    v->scroll=pl_scrolledit;
    v->scr.pos=Pt(0,0);
    v->scr.size=Pt(ntext,0);
}
```

Uses LEAF, UP 22a, pl_childspaceedit() 98b, pl_drawedit() 96b, pl_freeedit() 96a, pl_getsizeedit() 98a, pl_hitedit() 96c, pl_pasteedit() 110e, pl_scrolledit() 98c, pl_snarfedit() 110d, pl_typeedit() 97, and twfree() 148b.

⟨*function* pl_freeedit 96a⟩≡ (130)

```
void pl_freeedit(Panel *p){
    Edit *ep;
    ep=p->data;
    if(ep->t) twfree(ep->t);
    ep->t=0;
}
```

Uses twfree() 148b.

12.2.2 Drawing

⟨*function* pl_drawedit 96b⟩≡ (130)

```
void pl_drawedit(Panel *p){
    Edit *ep;
    Panel *sb;
    ep=p->data;
    if(ep->t==0){
        ep->t=twnew(p->b, font, ep->text, ep->ntext);
        if(ep->t==0){
            fprint(2, "pl_drawedit: can't allocate\n");
            exits("no mem");
        }
    }
    ep->t->b=p->b;
    twreshape(ep->t, p->r);
    twhilite(ep->t, ep->sel0, ep->sel1, 1);
    sb=p->yscroller;
    if(sb && sb->setscrollbar)
        sb->setscrollbar(sb, ep->t->top, ep->t->bot, ep->t->etext-ep->t->text);
}
```

Uses twhilite() 143b, twnew() 148a, and twreshape() 147b.

12.2.3 Reacting

⟨*function* pl_hitedit 96c⟩≡ (130)

```
/*
 * Should do double-clicks:
 * If ep->sel0==ep->sel1 on entry and the
 * call to twselect returns the same selection, then
 * expand selections (| marks possible selection points, ... is expanded selection)
 * <|...|>   <> must nest
 * (|...|)   () must nest
 * [|...|]   [] must nest
 * {|...|}   {} must nest
 * '|...|'   no ' in ...
 * "|...|"   no " in ...
 * \n|...|\n  either newline may be the corresponding end of text
```

```
 *    include the trailing newline in the selection
 * ...|I...  I and ... are characters satisfying pl_idchar(I)
 * ...I|
 */
int pl_hitedit(Panel *p, Mouse *m){
    Edit *ep;
    ep=p->data;
    if(ep->t && m->buttons&1){
        plgrabkb(p);
        ep->t->b=p->b;
        twhilite(ep->t, ep->sel0, ep->sel1, 0);
        twselect(ep->t, m);
        ep->sel0=ep->t->sel0;
        ep->sel1=ep->t->sel1;
        if((m->buttons&7)==3){
            plsnarf(p);
            plepaste(p, 0, 0); /* cut */
        }
        else if((m->buttons&7)==5)
            plpaste(p);
        else if(ep->hit)
            (*ep->hit)(p);
    }
    return 0;
}
```

Uses plepaste() 100b, plgrabkb() 33k, plpaste() 109b, plsnarf() 109a, twhilite() 143b, and twselect() 143c.

⟨*function* pl_typeedit 97⟩≡ (130)

```
void pl_typeedit(Panel *p, Rune c){
    Edit *ep;
    Textwin *t;
    int bot, scrolled;
    Panel *sb;
    ep=p->data;
    t=ep->t;
    if(t==0) return;
    t->b=p->b;
    twhilite(t, ep->sel0, ep->sel1, 0);
    switch(c){
    case Kesc:
        plsnarf(p);
        plepaste(p, 0, 0); /* cut */
        break;
    case Kdel: /* clear */
        ep->sel0=0;
        ep->sel1=plelen(p);
        plepaste(p, 0, 0); /* cut */
        break;
    case Kbs: /* ^H: erase character */
        if(ep->sel0!=0) --ep->sel0;
        twreplace(t, ep->sel0, ep->sel1, 0, 0);
        break;
// case Knack: /* ^U: erase line */
//  while(ep->sel0!=0 && t->text[ep->sel0-1]!='\n') --ep->sel0;
//  twreplace(t, ep->sel0, ep->sel1, 0, 0);
//  break;
// case Ketb: /* ^W: erase word */
//  while(ep->sel0!=0 && !pl_idchar(t->text[ep->sel0-1])) --ep->sel0;
//  while(ep->sel0!=0 && pl_idchar(t->text[ep->sel0-1])) --ep->sel0;
//  twreplace(t, ep->sel0, ep->sel1, 0, 0);
```

```
//  break;
    default:
        if((c & 0xFF00) == KF || (c & 0xFF00) == Spec)
            break;
        twreplace(t, ep->sel0, ep->sel1, &c, 1);
        ++ep->sel0;
        break;
    }
    ep->sel1=ep->sel0;
    /*
     * Scroll up until ep->sel0 is above t->bot.
     */
    scrolled=0;
    do{
        bot=t->bot;
        if(ep->sel0<=bot) break;
        twscroll(t, twpt2rune(t, Pt(t->r.min.x, t->r.min.y+font->height)));
        scrolled++;
    }while(bot!=t->bot);
    if(scrolled){
        sb=p->yscroller;
        if(sb && sb->setscrollbar)
            sb->setscrollbar(sb, t->top, t->bot, t->etext-t->text);
    }
    twhilite(t, ep->sel0, ep->sel1, 1);
}
```

Uses plelen() 99c, plepaste() 100b, plsnarf() 109a, twhilite() 143b, twpt2rune() 140c, twreplace() 146b, and twscroll() 147a.

12.2.4 Packing methods

⟨*function* pl_getsizeedit 98a⟩≡ (130)

```
Point pl_getsizeedit(Panel *p, Point children){
    USED(children);
    return pl_boxsize(((Edit *)p->data)->minsize, p->state);
}
```

Uses pl_boxsize() 30c.

⟨*function* pl_childspaceedit 98b⟩≡ (130)

```
void pl_childspaceedit(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

12.2.5 Other methods

⟨*function* pl_scrolledit 98c⟩≡ (130)

```
void pl_scrolledit(Panel *p, int dir, buttons buttons, int num, int den){
    Edit *ep;
    Textwin *t;
    Panel *sb;
    int index, nline;
    if(dir!=VERT) return;
    ep=p->data;
    t=ep->t;
    if(t==0) return;
    t->b=p->b;
    switch(buttons){
```

```
    default:
        return;
    case 1:  /* top line moves to mouse position */
        nline=(t->r.max.y-t->r.min.y)/t->hgt*num/den;
        index=t->top;
        while(index!=0 && nline!=0)
            if(t->text[--index]=='\n') --nline;
        break;
    case 2:  /* absolute */
        index=(t->etext-t->text)*num/den;
        break;
    case 4:  /* mouse points at new top line */
        index=twpt2rune(t,
            Pt(t->r.min.x, t->r.min.y+(t->r.max.y-t->r.min.y)*num/den));
        break;
    }
    while(index!=0 && t->text[index-1]!='\n') --index;
    if(index!=t->top){
        twhilite(ep->t, ep->sel0, ep->sel1, 0);
        twscroll(t, index);
        p->scr.pos.y=t->top;
        twhilite(ep->t, ep->sel0, ep->sel1, 1);
        sb=p->yscroller;
        if(sb && sb->setscrollbar)
            sb->setscrollbar(sb, t->top, t->bot, t->etext-t->text);
    }
}
```

Uses VERT, twhilite() 143b, twpt2rune() 140c, and twscroll() 147a.

⟨*function* plescroll 99a⟩≡ (130)

```
void plescroll(Panel *p, int top){
    Textwin *t;
    t=((Edit*)p->data)->t;
    if(t) twscroll(t, top);
}
```

Uses twscroll() 147a.

⟨*function* plegetsel 99b⟩≡ (130)

```
void plegetsel(Panel *p, int *sel0, int *sel1){
    Edit *ep;
    ep=p->data;
    *sel0=ep->sel0;
    *sel1=ep->sel1;
}
```

⟨*function* plelen 99c⟩≡ (130)

```
int plelen(Panel *p){
    Textwin *t;
    t=((Edit*)p->data)->t;
    if(t==0) return 0;
    return t->etext-t->text;
}
```

⟨*function* pleget 99d⟩≡ (130)

```
Rune *pleget(Panel *p){
    Textwin *t;
    t=((Edit*)p->data)->t;
    if(t==0) return 0;
    return t->text;
}
```

⟨*function* `plesel` 100a⟩≡ (130)
```
void plesel(Panel *p, int sel0, int sel1){
    Edit *ep;
    ep=p->data;
    if(ep->t==0) return;
    ep->t->b=p->b;
    twhilite(ep->t, ep->sel0, ep->sel1, 0);
    ep->sel0=sel0;
    ep->sel1=sel1;
    twhilite(ep->t, ep->sel0, ep->sel1, 1);
}
```
Uses `twhilite()` 143b.

⟨*function* `plepaste` 100b⟩≡ (130)
```
void plepaste(Panel *p, Rune *text, int ntext){
    Edit *ep;
    ep=p->data;
    if(ep->t==0) return;
    ep->t->b=p->b;
    twhilite(ep->t, ep->sel0, ep->sel1, 0);
    twreplace(ep->t, ep->sel0, ep->sel1, text, ntext);
    ep->sel1=ep->sel0+ntext;
    twhilite(ep->t, ep->sel0, ep->sel1, 1);
    p->scr.size.y=ep->t->etext-ep->t->text;
    p->scr.pos.y=ep->t->top;
}
```
Uses `twhilite()` 143b and `twreplace()` 146b.

⟨*function* `plemove` 100c⟩≡ (130)
```
void plemove(Panel *p, Point d){
    Edit *ep;
    ep=p->data;
    if(ep->t && !eqpt(d, Pt(0,0))) twmove(ep->t, d);
}
```
Uses `twmove()` 148c.

12.3 Text view

A text view displays scrollable, multi-font formatted text interspersed with images and embedded panels—it is the widget used by `mothra` to render web pages. The content is represented as a linked list of `Rtext` nodes (see the Rich Text section below), each describing a piece of text with a specific font, or a bitmap, or an inline sub-panel. The text view reformats its content when the window width changes (via `pl_rtfmt()`[134d]), computing line breaks and positions for each `Rtext` node. Drawing then simply iterates through the formatted list, rendering only nodes visible in the current viewport.

⟨*struct* `Textview` 100d⟩≡ (140a)
```
struct Textview{
    void (*hit)(Panel *, int, Rtext *); /* call back to user on hit */
    Rtext *text;   /* text */
    int yoffs;   /* offset of top of screen */
    Rtext *hitword;   /* text to hilite */
    Rtext *hitfirst;  /* first word in range select */
    int twid;   /* text width */
    int thgt;   /* text height */
    Point minsize;   /* smallest acceptible window size */
    buttons buttons;
};
```

12.3.1 Initializing

⟨*function* pltextview 101a⟩≡ (140a)
```
Panel *pltextview(Panel *parent, int flags, Point minsize, Rtext *t, void (*hit)(Panel *, int, Rtext *)){
    Panel *v;
    v=pl_newpanel(parent, sizeof(Textview));
    plinittextview(v, flags, minsize, t, hit);
    return v;
}
```
Uses pl_newpanel() 19a and plinittextview() 101b.

⟨*function* plinittextview 101b⟩≡ (140a)
```
void plinittextview(Panel *v, int flags, Point minsize, Rtext *t, void (*hit)(Panel *, int, Rtext *)){
    Textview *tp = v->data;

    v->flags=flags|LEAF;
    v->state=UP;
    v->draw=pl_drawtextview;
    v->hit=pl_hittextview;
    v->type=pl_typetextview;
    v->getsize=pl_getsizetextview;
    v->childspace=pl_childspacetextview;
    v->kind="textview";
    v->pri=pl_pritextview;
    tp->hit=hit;
    tp->minsize=minsize;
    tp->text=t;
    tp->yoffs=0;
    tp->hitfirst=0;
    tp->hitword=0;
    v->scroll=pl_scrolltextview;
    v->snarf=pl_snarftextview;
    tp->twid=-1;
    v->scr.pos=Pt(0,0);
    v->scr.size=Pt(0,1);
}
```
Uses LEAF, UP 22a, pl_childspacetextview() 103b, pl_drawtextview() 101c, pl_getsizetextview() 103a, pl_hittextview() 102a, pl_pritextview() 139a, pl_scrolltextview() 138d, pl_snarftextview() 139b, and pl_typetextview() 102b.

12.3.2 Drawing

⟨*function* pl_drawtextview 101c⟩≡ (140a)
```
void pl_drawtextview(Panel *p){
    int twid;
    Rectangle r;
    Textview *tp;
    tp=p->data;
    r=pl_outline(p->b, p->r, UP);
    twid=r.max.x-r.min.x;
    if(twid!=tp->twid){
        tp->twid=twid;
        tp->thgt=pl_rtfmt(tp->text, tp->twid);
        p->scr.size.y=tp->thgt;
    }
    p->scr.pos.y=tp->yoffs;
    pl_rtdraw(p->b, r, tp->text, tp->yoffs);
    pl_setscrpos(p, tp, r);
}
```
Uses UP 22a, pl_outline() 29b, pl_rtdraw() 105b, pl_rtfmt() 134d, and pl_setscrpos() 138b.

12.3.3 Reacting

⟨*function* pl_hittextview 102a⟩≡ (140a)

```
int pl_hittextview(Panel *p, Mouse *m){
    Rtext *oldhitword, *oldhitfirst;
    int hitme, oldstate;
    Point ul, size;
    Textview *tp;

    tp=p->data;

    hitme=0;
    oldstate=p->state;
    oldhitword=tp->hitword;
    oldhitfirst=tp->hitfirst;
    if(oldhitword==oldhitfirst)
        pl_passon(oldhitword, m);
    if(m->buttons&OUT)
        p->state=UP;
    else if(m->buttons&7){
        p->state=DOWN;
        tp->buttons=m->buttons;
        if(oldhitword==0 || oldhitword->p==0 || (oldhitword->p->flags&REMOUSE)==0){
            ul=p->r.min;
            size=subpt(p->r.max, p->r.min);
            pl_interior(p->state, &ul, &size);
            tp->hitword=pl_rthit(tp->text, tp->yoffs, m->xy, ul);
            if(tp->hitword==0)
                if(oldhitword!=0 && oldstate==DOWN)
                    tp->hitword=oldhitword;
                else
                    tp->hitfirst=0;
            if(tp->hitword!=0 && oldstate!=DOWN)
                tp->hitfirst=tp->hitword;
        }
    }
    else{
        if(p->state==DOWN) hitme=1;
        p->state=UP;
    }
    if(tp->hitfirst!=oldhitfirst || tp->hitword!=oldhitword){
        plrtseltext(tp->text, tp->hitword, tp->hitfirst);
        pl_drawtextview(p);
        if(tp->hitword==tp->hitfirst)
            pl_passon(tp->hitword, m);
    }
    if(hitme && tp->hit && tp->hitword!=0 && tp->hitword==tp->hitfirst){
        plrtseltext(tp->text, 0, 0);
        pl_drawtextview(p);
        tp->hit(p, tp->buttons, tp->hitword);
        tp->hitword=0;
        tp->hitfirst=0;
    }
    return 0;
}
```

Uses DOWN 45c, REMOUSE, UP 22a, pl_drawtextview() 101c, pl_interior() 30d, pl_passon() 138c, pl_rthit() 106b, and plrtseltext() 106c.

⟨*function* pl_typetextview 102b⟩≡ (140a)

```
void pl_typetextview(Panel *g, Rune c){
```

```
    USED(g, c);
}
```

12.3.4 Packing methods

⟨*function* `pl_getsizetextview` 103a⟩≡ (140a)

```
Point pl_getsizetextview(Panel *p, Point children){
    USED(children);
    return pl_boxsize(((Textview *)p->data)->minsize, p->state);
}
```

Uses `pl_boxsize()` 30c.

⟨*function* `pl_childspacetextview` 103b⟩≡ (140a)

```
void pl_childspacetextview(Panel *g, Point *ul, Vector *size){
    USED(g, ul, size);
}
```

12.4 Completion

12.5 Rich text

`Rtext` ("rich text") is the data structure that feeds the text view widget. Each node in the linked list represents one displayable element: either a string (with a font), a bitmap image, or an inline `Panel`. The `b`, `p`, and `text` fields form a discriminated union: if `b` is non-nil, it is an image; else if `p` is non-nil, it is an inline panel; else `text` is a string rendered with `font`. The `space` and `indent` fields control layout: `space` is the horizontal gap before this node if it is on the same line as its predecessor, and `indent` is the indentation if it starts a new line. Setting `space` to zero prevents a line break between adjacent nodes (useful for multi-font words). The `flags` field marks whether a node is mouse-sensitive (`PL_HOT`) or selected (`PL_SEL`). Three convenience constructors—`plrtstr()`[104c], `plrtbitmap()`[104d], and `plrtpanel()`[104b]—create nodes of each type and append them to a list.

⟨*struct* `Rtext` 103c⟩≡ (121d)

```
struct Rtext{
    int flags;  /* responds to hits? text selection? */
    void *user;  /* user data */
    int space;  /* how much space before, if no break */
    int indent;  /* how much space before, after a break */
    Image *b;  /* what to display, if nonzero */
    Panel *p;  /* what to display, if nonzero and b==0 */
    Font *font;  /* font in which to draw text */
    char *text;  /* what to display, if b==0 and p==0 */
    Rtext *next;  /* next piece */
    /* private below */
    Rtext *nextline; /* links line to line */
    Rtext *last;  /* last, for append */
    Rectangle r;  /* where to draw, if origin were Pt(0,0) */
    int topy;  /* y coord of top of line */
    int wid;  /* not including space */
};
```

12.5.1 Initializing

⟨*function* pl_rtnew 104a⟩≡ (136c)

```
Rtext *pl_rtnew(Rtext **t, int space, int indent, Image *b, Panel *p, Font *f, char *s, int flags, void *user){
    Rtext *new;
    new=pl_emalloc(sizeof(Rtext));
    new->flags=flags;
    new->user=user;
    new->space=space;
    new->indent=indent;
    new->b=b;
    new->p=p;
    new->font=f;
    new->text=s;
    new->next=0;
    new->nextline=0;
    new->r=Rect(0,0,0,0);
    if(*t)
        (*t)->last->next=new;
    else
        *t=new;
    (*t)->last=new;
    return new;
}
```

Uses pl_emalloc() 120a.

⟨*function* plrtpanel 104b⟩≡ (136c)

```
Rtext *plrtpanel(Rtext **t, int space, int indent, Panel *p, void *user){
    return pl_rtnew(t, space, indent, 0, p, 0, 0, 1, user);
}
```

Uses pl_rtnew() 104a.

⟨*function* plrtstr 104c⟩≡ (136c)

```
Rtext *plrtstr(Rtext **t, int space, int indent, Font *f, char *s, int flags, void *user){
    return pl_rtnew(t, space, indent, 0, 0, f, s, flags, user);
}
```

Uses pl_rtnew() 104a.

⟨*function* plrtbitmap 104d⟩≡ (136c)

```
Rtext *plrtbitmap(Rtext **t, int space, int indent, Image *b, int flags, void *user){
    return pl_rtnew(t, space, indent, b, 0, 0, 0, flags, user);
}
```

Uses pl_rtnew() 104a.

⟨*function* plrtfree 104e⟩≡ (136c)

```
void plrtfree(Rtext *t){
    Rtext *next;
    while(t){
        next=t->next;
        free(t);
        t=next;
    }
}
```

12.5.2 Drawing

⟨*constant* PL_HOT 104f⟩≡ (121d)

```
#define PL_HOT  1
```

⟨*constant* PL_SEL 105a⟩≡ (121d)

```
#define PL_SEL  2
```

⟨*function* pl_rtdraw 105b⟩≡ (136c)

```
void pl_rtdraw(Image *b, Rectangle r, Rtext *t, int yoffs){
    static Image *backup;
    Point offs, lp;
    Rectangle dr;
    Image *bb;

    bb = b;
    if(backup==0 || backup->chan!=b->chan || rectinrect(r, backup->r)==0){
        freeimage(backup);
        backup=allocimage(display, bb->r, bb->chan, 0, DNofill);
    }
    if(backup)
        b=backup;
    pl_clr(b, r);
    lp=ZP;
    offs=subpt(r.min, Pt(0, yoffs));
    for(;t;t=t->next) if(!eqrect(t->r, Rect(0,0,0,0))){
        dr=rectaddpt(t->r, offs);
        if(dr.max.y>r.min.y
        && dr.min.y<r.max.y){
            if(t->b){
                draw(b, insetrect(dr, BORD), t->b, 0, t->b->r.min);
                if(t->flags&PL_HOT) border(b, dr, 1, display->black, ZP);
                if(t->flags&PL_SEL)
                    pl_highlight(b, dr);
            }
            else if(t->p){
                plmove(t->p, subpt(dr.min, t->p->r.min));
                pldraw(t->p, b);
                if(b!=bb)
                    pl_stuffbitmap(t->p, bb);
            }
            else{
                string(b, dr.min, display->black, ZP, t->font, t->text);
                if(t->flags&PL_SEL)
                    pl_highlight(b, dr);
                if(t->flags&PL_HOT){
                    if(lp.y+1 != dr.max.y)
                        lp = Pt(dr.min.x, dr.max.y-1);
                    line(b, lp, Pt(dr.max.x, dr.max.y-1),
                        Endsquare, Endsquare, 0,
                        display->black, ZP);
                    lp = Pt(dr.max.x, dr.max.y-1);
                    continue;
                }
            }
            lp=ZP;
        }
    }
    if(b!=bb)
        draw(bb, r, b, 0, r.min);
}
```

Uses BORD-8 134a, pl_clr() 105c, pl_highlight() 74b, pl_stuffbitmap() 135, pldraw() 28a, and plmove() 86b.

⟨*function* pl_clr 105c⟩≡ (127c)

```
void pl_clr(Image *b, Rectangle r){
```

```
    draw(b, r, display->white, 0, ZP);
}
```

⟨*function* `pl_rtredraw` 106a⟩≡ (136c)

```
/*
 * Rectangle r of Image b contains an image of Rtext t, offset by oldoffs.
 * Redraw the text to have offset yoffs.
 */
void pl_rtredraw(Image *b, Rectangle r, Rtext *t, int yoffs, int oldoffs){
    int dy, size;
    dy=oldoffs-yoffs;
    size=r.max.y-r.min.y;
    if(dy>=size || -dy>=size)
        pl_rtdraw(b, r, t, yoffs);
    else if(dy<0){
        pl_reposition(t, b, r.min,
            Rect(r.min.x, r.min.y-dy, r.max.x, r.max.y));
        pl_rtdraw(b, Rect(r.min.x, r.max.y+dy, r.max.x, r.max.y),
            t, yoffs+size+dy);
    }
    else if(dy>0){
        pl_reposition(t, b, Pt(r.min.x, r.min.y+dy),
            Rect(r.min.x, r.min.y, r.max.x, r.max.y-dy));
        pl_rtdraw(b, Rect(r.min.x, r.min.y, r.max.x, r.min.y+dy), t, yoffs);
    }
}
```

Uses `pl_reposition()` 136a and `pl_rtdraw()` 105b.

12.5.3 Reacting

⟨*function* `pl_rthit` 106b⟩≡ (136c)

```
Rtext *pl_rthit(Rtext *t, int yoffs, Point p, Point ul){
    Rectangle r;
    Point lp;
    if(t==0) return 0;
    p.x-=ul.x;
    p.y+=yoffs-ul.y;
    while(t->nextline && t->nextline->topy<=p.y) t=t->nextline;
    lp=ZP;
    for(;t!=0;t=t->next){
        if(t->topy>p.y) return 0;
        r = t->r;
        if((t->flags&PL_HOT) != 0 && t->b == nil && t->p == nil){
            if(lp.y == r.max.y && lp.x < r.min.x)
                r.min.x=lp.x;
            lp=r.max;
        } else
            lp=ZP;
        if(ptinrect(p, r)) return t;
    }
    return 0;
}
```

12.5.4 Packing methods

12.5.5 Other methods

⟨*function* `plrtseltext` 106c⟩≡ (136c)

```
void plrtseltext(Rtext *t, Rtext *s, Rtext *e){
    while(t){
        t->flags &= ~PL_SEL;
        t = t->next;
    }
    if(s==0 || e==0)
        return;
    for(t=s; t!=0 && t!=e; t=t->next)
        ;
    if(t==e){
        for(t=s; t!=e; t=t->next)
            t->flags |= PL_SEL;
    }else{
        for(t=e; t!=s; t=t->next)
            t->flags |= PL_SEL;
    }
    t->flags |= PL_SEL;
}
```

Chapter 13

Advanced Topics

This chapter covers features that cut across multiple widget types: copy/paste (called "snarf" in Plan 9 terminology), the user-data hooks for application-specific state, and advanced layout flags.

13.1 copy/paste

Copy/paste in Plan 9 uses `/dev/snarf` as a shared clipboard (rather than X11's complex selection protocol). The `snarf` and `paste` method pointers on `Panel`[18] allow each widget type to provide its own implementation: entries snarf their text content; edit widgets snarf the current selection. Widgets that do not support copy/paste leave these pointers `nil`.

⟨`Panel` *other methods* 108a⟩+≡ (18) ◁71e
```
char* (*snarf)(Panel *);   /* snarf text from panel */
void (*paste)(Panel *, char *);   /* paste text into panel */
```

⟨`pl_newpanel()` *set default methods* 108b⟩+≡ (19a) ◁71f
```
v->snarf=nil;
v->paste=nil;
```

⟨*function* `plputsnarf` 108c⟩≡ (138a)
```
void plputsnarf(char *s){
    int fd;

    if(s==0 || *s=='\0')
        return;
    if((fd=open("/dev/snarf", OWRITE|OTRUNC))>=0){
        write(fd, s, strlen(s));
        close(fd);
    }
}
```

⟨*function* `plgetsnarf` 108d⟩≡ (138a)
```
char *plgetsnarf(void){
    int fd, n, r;
    char *s;

    if((fd=open("/dev/snarf", OREAD))<0)
        return nil;
    n=0;
    s=nil;
    for(;;){
        s=pl_erealloc(s, n+1024);
        if((r = read(fd, s+n, 1024)) <= 0)
            break;
```

```
        n += r;
    }
    close(fd);
    if(n <= 0){
        free(s);
        return nil;
    }
    s[n] = '\0';
    return s;
}
```

Uses pl_erealloc() 120b.

⟨*function* plsnarf 109a⟩≡ (138a)

```
void plsnarf(Panel *p){
    char *s;

    if(p==0 || p->snarf==0)
        return;
    s=p->snarf(p);
    plputsnarf(s);
    free(s);
}
```

Uses plputsnarf() 108c.

⟨*function* plpaste 109b⟩≡ (138a)

```
void plpaste(Panel *p){
    char *s;

    if(p==0 || p->paste==0)
        return;
    if(s=plgetsnarf()){
        p->paste(p, s);
        free(s);
    }
}
```

Uses plgetsnarf() 108d.

13.1.1 Widgets hooks

Each widget type that supports copy/paste registers its own `snarf` and `paste` implementations during initialization. The entry widget's paste appends text to the buffer; the edit widget's paste replaces the current selection using the `Textwin` API. Note that the entry widget's snarf refuses to copy from password fields (`USERFL` flag).

⟨plinitentry() *set snarf methods* 109c⟩≡ (48c)

```
v->snarf=pl_snarfentry;
v->paste=pl_pasteentry;
```

Uses pl_pasteentry() 110a and pl_snarfentry() 109d.

⟨*function* pl_snarfentry 109d⟩≡ (129b)

```
char *pl_snarfentry(Panel *p){
    Entry *ep;
    int n;

    if(p->flags&USERFL) /* no snarfing from password entry */
        return nil;
    ep=p->data;
    n=ep->entp-ep->entry;
    if(n<=0) return nil;
    return smprint("%.*s", n, ep->entry);
}
```

⟨*function* pl_pasteentry 110a⟩≡ (129b)
```
void pl_pasteentry(Panel *p, char *s){
    Entry *ep;
    char *e;
    int n, m;

    ep=p->data;
    n=ep->entp-ep->entry;
    m=strlen(s);
    e=pl_erealloc(ep->entry,n+m+SLACK);
    ep->entry=e;
    e+=n;
    strncpy(e, s, m);
    e+=m;
    *e='\0';
    ep->entp=ep->eent=e;
    pldraw(p, p->b);
}
```
Uses SLACK-1 110b, pl_erealloc() 120b, and pldraw() 28a.

⟨*constant* SLACK 110b⟩≡ (129b)
```
#define SLACK 7 /* enough for one extra rune and < and a nul */
```

⟨pl_hitentry() *handle copy/paste when middle or right click* 110c⟩≡ (50e)
```
if((old&7)==CLICK_LEFT){
    if((m->buttons&7)==CLICK_LEFT|CLICK_MIDDLE){
        Entry *ep;

        plsnarf(p);

        /* cut */
        ep=p->data;
        ep->entp=ep->entry;
        *ep->entp='\0';
        pldraw(p, p->b);
    }
    if((m->buttons&7)==CLICK_LEFT|CLICK_RIGHT)
        plpaste(p);
}
```
Uses pldraw() 28a, plpaste() 109b, and plsnarf() 109a.

⟨*function* pl_snarfedit 110d⟩≡ (130)
```
char *pl_snarfedit(Panel *p){
    int s0, s1;
    Rune *t;
    t=pleget(p);
    plegetsel(p, &s0, &s1);
    if(t==0 || s0>=s1)
        return nil;
    return smprint("%.*S", s1-s0, t+s0);
}
```
Uses pleget() 99d and plegetsel() 99b.

⟨*function* pl_pasteedit 110e⟩≡ (130)
```
void pl_pasteedit(Panel *p, char *s){
    Rune *t;
    if(t=runesmprint("%s", s)){
        plepaste(p, t, runestrlen(t));
        free(t);
```

```
    }
  }
```
Uses `plepaste()` 100b.

13.2 Hooks

⟨`Panel` *user-specific fields* 111a⟩≡ (18)
```
  int user;     /* available for user */
  void *userp;     /* available for user */
```

These two fields are not used by the library at all—they exist solely for the application programmer. As Tom Duff explains in the `libpanel` manual, "it is often handy to have a common hit function for a group of panels and distinguish amongst them by an index number or private data stored in `user` or `userp`." For example, an application with ten buttons sharing the same callback can store a different index in each button's `user` field to tell them apart.

13.3 Advanced layout

13.3.1 `FIXEDX`, `FIXEDY`

The `FIXED` flag tells `plpack()`[35] to use the widget's `fixedsize` field instead of the size computed from its children. `FIXED` is actually the OR of `FIXEDX` and `FIXEDY`, which can be used independently to fix only one dimension. This is useful for widgets like canvas or images where the application knows the exact pixel size it wants.

⟨*constant* `FIXEDX` 111b⟩≡ (121d)
```
  #define FIXEDX 0x0400
```

⟨*constant* `FIXEDY` 111c⟩≡ (121d)
```
  #define FIXEDY 0x0800
```

⟨`Panel` *fixed size field* 111d⟩≡ (18)
```
  Point fixedsize;    /* size of Panel, if FIXED */
```

⟨`pl_sizereq()` *if* `FIXEDX` *or* `FIXEDY` 111e⟩≡ (36e)
```
  if(p->flags&FIXEDX) p->sizereq.x=p->fixedsize.x;
  if(p->flags&FIXEDY) p->sizereq.y=p->fixedsize.y;
```

⟨*constant* `FIXED` 111f⟩≡ (121d)
```
  #define FIXED 0x0c00  /* don't pass children's size requests through to parent */
```

13.3.2 `MAXX`, `MAXY`

`MAXX` and `MAXY` solve a common aesthetic problem: when several widgets are packed side by side (e.g., a row of buttons packed with `PACKW`), each one gets only as much width as its label requires, producing a ragged result. Setting `MAXX` on each sibling forces them all to the same width—the width of the widest one—giving a clean, uniform appearance. `MAXY` does the same in the vertical direction. The implementation works in two passes over the sibling list within `pl_sizereq()`[36e]: the first pass records the maximum size request, and the second pass overwrites each flagged widget's request with that maximum.

⟨*constant* `MAXX` 111g⟩≡ (121d)
```
  #define MAXX 0x1000  /* make x size as big as biggest sibling's */
```

⟨*constant* `MAXY` 111h⟩≡ (121d)
```
  #define MAXY 0x2000  /* make y size as big as biggest sibling's */
```

⟨`pl_sizeref()` *other locals* 112a⟩≡ (36e)

```
Vector maxsize = Pt(0,0);
```

⟨`pl_sizeref()` *when looping over children, adjust* `maxsize` 112b⟩≡ (36e)

```
if(cp->sizereq.x>maxsize.x) maxsize.x=cp->sizereq.x;
if(cp->sizereq.y>maxsize.y) maxsize.y=cp->sizereq.y;
```

⟨`pl_sizereq()` *if* `MAXX` *or* `MAXY` 112c⟩≡ (36e)

```
for(cp=p->child;cp;cp=cp->next){
    if(cp->flags&MAXX) cp->sizereq.x=maxsize.x;
    if(cp->flags&MAXY) cp->sizereq.y=maxsize.y;
}
```

Chapter 14

Conclusion

You now know how the Plan 9 widget library `libpanel` works—from the tree of `Panel` nodes assembled by the application, through the two-pass layout algorithm that assigns positions, to the event dispatch that routes a mouse click to the right widget's callback—and more generally how many widget libraries work.

Despite being only about 4000 lines of C, `libpanel` implements a complete widget toolkit: a tree of polymorphic widgets dispatched through function pointers (OO in C), a two-pass layout engine modeled on Tk's pack geometry manager, mouse event dispatch with priority-based hit testing, a scrolling protocol that links scroll bars to content widgets, popup and pull-down menus with save/restore screen management, and a formatted text view capable of rendering a web browser's output. The library was written by Tom Duff specifically for `mothra`, the Plan 9 web browser, and its design reflects a deliberate choice: keep everything explicit, nothing automatic, and let the application programmer call `plpack()` and `pldraw()` when things change.

The beauty of `libpanel`'s design is how much it accomplishes with so little machinery. There is no type system beyond C's own structs and function pointers. There is no observer pattern, no signal/slot mechanism, no property binding. Instead, there is a tree of `Panel` nodes, each carrying a handful of method pointers, and a few well-chosen conventions: packing flags for layout, priority levels for hit testing, and a pair of cross-linked pointers for scrolling. These simple building blocks compose into surprisingly rich behavior.

14.1 Patterns and techniques

These techniques apply far beyond GUIs:

- *Composite pattern*: a tree where operations on a parent propagate to its children. The same structure appears in HTML DOMs, 3D scene graphs, compiler ASTs, and file systems—any hierarchy where you need recursive traversal.

- *Function-pointer polymorphism*: each `Panel` carries a table of method pointers (`draw`, `hit`, `scroll`, etc.), so the framework treats all widgets uniformly. This is object-oriented programming in C—the same technique used by the Linux kernel's `file_operations` struct, SQLite's virtual table API, and GObject in GTK. It solves the same problem as virtual methods in C++ or interfaces in Go: dispatching through a uniform API to implementations that differ.

- *Two-pass layout*: first bottom-up (each widget reports its preferred size), then top-down (the parent assigns rectangles). CSS layout, TeX's box-and-glue model, and Android's `onMeasure`/`onLayout` use the same approach. It arises whenever the final arrangement depends on both what children want and what the parent can offer.

- *Two-way protocol*: scrollbars and content widgets call each other's `scroll` method through cross pointers, neither knowing the other's implementation. This mutual-callback decoupling is the same idea as the Observer pattern and two-way data binding in reactive frameworks.

- *Explicit update model*: the application calls `plpack()` and `pldraw()` rather than having the framework detect changes. This pull-based approach also characterizes immediate-mode GUI libraries like Dear ImGui and game render loops. It trades convenience for predictability.

14.2 Connections to other books

- GRAPHICS book [Pad16b] covers `libdraw`, the drawing library on which `libpanel` is built. Every `draw()` call, every `Image`, every `Rectangle` operation used throughout this book comes from `libdraw`.
- WINDOWS book [Pad16c] covers `rio`, the Plan 9 window manager. `libpanel` applications run inside `rio` windows and receive mouse and keyboard events through the event library (`libevent`). The `eresized()` callback, the `view` global, and `getwindow()` are all part of the `rio`/`libdraw` contract.
- LIBCORE book [Pad16a] covers the C library, including `Rune` (Unicode code points) and the UTF-8 string functions used by text widgets.

14.3 Missing features

Compared to modern widget libraries, `libpanel` is deliberately minimal. A few notable absences:

- **No automatic relayout.** As discussed in Chapter 2, the application must explicitly call `plpack()` and `pldraw()` after any change. Modern toolkits (GTK, Qt, Tk) track invalidation and schedule redraws automatically.
- **No horizontal scrolling.** The `xscroller` field exists in `Panel`[18] but no current widget implements it.
- **No theming or styling.** Colors and border widths are hardcoded in `pl_drawinit()`. There is no way to change the look without modifying the library source.
- **Limited text editing.** The edit widget delegates to `Textwin`, which provides basic insert/delete and selection but lacks undo, find/replace, or syntax highlighting.
- **No accessibility.** There is no keyboard navigation between widgets (no tab focus), no screen reader support, and no high-DPI scaling.

14.4 Beyond the Plan 9 widget library

Widget toolkits have grown enormously since `libpanel` was written. Here are some of the features found in modern frameworks:

- *Declarative UI*: frameworks like React, SwiftUI, Jetpack Compose, and Flutter describe the UI as a function of state—the framework diffs the old and new descriptions and updates only what changed. `libpanel` is fully imperative: the application creates widgets, mutates them, and explicitly calls `plpack()`/`pldraw()`.
- *Layout engines*: CSS Flexbox and Grid, Qt's layout managers, and SwiftUI's stacks provide powerful and flexible constraint-based layout. `libpanel`'s Tk-style packer handles the common cases (fill, expand, centering) but cannot express layouts like "these three columns should share space equally" or "this widget should be 30% of the parent's width."
- *Styling and theming*: GTK, Qt, and web frameworks separate appearance from structure through CSS-like stylesheets, themes, and skins. `libpanel` hardcodes colors and border widths in `pl_drawinit()`.

- *Data binding and reactivity*: modern frameworks automatically propagate changes from data models to the UI (and back). When a model value changes, the bound widgets update automatically. `libpanel` has no such mechanism; the application must manually push changes to widgets.

- *GPU-accelerated rendering*: Qt, GTK4, Flutter, and all web browsers composite and render widgets on the GPU, enabling smooth animations, transparency, and complex visual effects. `libpanel` uses software rendering through `libdraw`.

- *Accessibility*: modern toolkits provide accessibility trees that screen readers can traverse, keyboard navigation between all widgets, and high-DPI scaling. This is increasingly a legal requirement (WCAG compliance).

- *Rich text and web content*: embedding a web browser (Chromium, WebKit) as a widget is now commonplace (Electron, WebView). `libpanel`'s `Rtext` rich text widget is far simpler—it was designed for `mothra`'s modest HTML rendering needs.

`libpanel` demonstrates that a useful widget toolkit can be built from a small set of concepts—a widget tree, function-pointer polymorphism, a packing layout algorithm, and event dispatch. The 4000 lines can be read and understood in their entirety, which is rare for a widget library. Modern toolkits are orders of magnitude larger, but the fundamentals—widget hierarchies, layout, event handling, and painting—are the same ones you have studied in this book.

Appendix A

Debugging

Each panel carries a `kind` string (e.g., `"label"`, `"button"`) set at creation time, used only for debugging output.

⟨`Panel` *debugging fields* 116a⟩≡ (18)
```
char *kind;     /* what kind of panel? */
```

`pl_print`[116b] is the entry point for dumping a widget tree. `pl_ipprint`[116d] walks the tree recursively, printing each panel's `kind`, bounding rectangle, packing flags, padding, and sizes with increasing indentation—essentially a textual snapshot of the layout state, invaluable when a widget ends up in the wrong place.

⟨*function* `pl_print` 116b⟩≡ (128b)
```
void pl_print(Panel *p){
    pl_ipprint(p, 0);
}
```
Uses `pl_ipprint()` 116d.

⟨*function* `pl_iprint` 116c⟩≡ (128b)
```
void pl_iprint(int indent, char *fmt, ...){
    char buf[8192];
    va_list arg;

    memset(buf, '\t', indent);
    va_start(arg, fmt);
    write(1, buf, vsnprint(buf+indent, sizeof(buf)-indent, fmt, arg));
    va_end(arg);
}
```

⟨*function* `pl_ipprint` 116d⟩≡ (128b)
```
void pl_ipprint(Panel *p, int n){
    Panel *c;
    char *place, *stick;

    pl_iprint(n, "%s (0x%.8x)\n", p->kind, p);
    pl_iprint(n, "  r=(%d %d, %d %d)\n",
        p->r.min.x, p->r.min.y, p->r.max.x, p->r.max.y);
    switch(p->flags&PACK){
    default: SET(place); break;
    case PACKN: place="n"; break;
    case PACKE: place="e"; break;
    case PACKS: place="s"; break;
    case PACKW: place="w"; break;
    }
    switch(p->flags&PLACE){
    default: SET(stick); break;
    case PLACECEN: stick=""; break;
    case PLACES: stick=" stick s"; break;
```

```
    case PLACEE: stick=" stick e"; break;
    case PLACEW: stick=" stick w"; break;
    case PLACEN: stick=" stick n"; break;
    case PLACENE: stick=" stick ne"; break;
    case PLACENW: stick=" stick nw"; break;
    case PLACESE: stick=" stick se"; break;
    case PLACESW: stick=" stick sw"; break;
    }
    pl_iprint(n, "  place %s%s%s%s%s%s\n",
        place,
        p->flags&FILLX?" fill x":"",
        p->flags&FILLY?" fill y":"",
        stick,
        p->flags&EXPAND?" expand":"",
        p->flags&FIXED?" fixed":"");
    if(!eqpt(p->pad, Pt(0, 0))) pl_iprint(n, "  pad=%d,%d)\n", p->pad.x, p->pad.y);
    if(!eqpt(p->ipad, Pt(0, 0))) pl_iprint(n, "  ipad=%d,%d)\n", p->ipad.x, p->ipad.y);
    pl_iprint(n, "  size=(%d,%d), sizereq=(%d,%d)\n",
        p->size.x, p->size.y, p->sizereq.x, p->sizereq.y);
    for(c=p->child;c;c=c->next)
        pl_ipprint(c, n+1);
}
```

Uses `pl_ipprint()` 116d and `pl_iprint()` 116c.

Appendix B

Error Management

Every `Panel` method pointer (`draw`, `hit`, `type`, `getsize`, `childspace`, `scroll`, `setscrollbar`) is initialized to a default "error" stub that prints the widget kind and aborts. If a widget constructor forgets to install a method, or if client code calls a method that a widget does not support (e.g., `type` on a button), the program crashes immediately with a clear message rather than silently misbehaving.

⟨*function* `pl_unexpected` 118a⟩≡ (127b)

```
void pl_unexpected(Panel *g, char *rou){
    fprint(STDERR, "%s called unexpectedly (%s %lux)\n",
           rou, g->kind, (ulong)g);
    abort();
}
```

⟨*function* `pl_drawerror` 118b⟩≡ (127b)

```
void pl_drawerror(Panel *g){
    pl_unexpected(g, "draw");
}
```

Uses `pl_unexpected()` 118a.

⟨*function* `pl_hiterror` 118c⟩≡ (127b)

```
int pl_hiterror(Panel *g, Mouse *m){
    USED(m);
    pl_unexpected(g, "hit");
    return 0;
}
```

Uses `pl_unexpected()` 118a.

⟨*function* `pl_typeerror` 118d⟩≡ (127b)

```
void pl_typeerror(Panel *g, Rune c){
    USED(c);
    pl_unexpected(g, "type");
}
```

Uses `pl_unexpected()` 118a.

⟨*function* `pl_getsizeerror` 118e⟩≡ (127b)

```
Point pl_getsizeerror(Panel *g, Point childsize){
    pl_unexpected(g, "getsize");
    return childsize;
}
```

Uses `pl_unexpected()` 118a.

⟨*function* `pl_childspaceerror` 118f⟩≡ (127b)

```
void pl_childspaceerror(Panel *g, Point *ul, Vector *size){
    USED(ul, size);
    pl_unexpected(g, "childspace");
}
```

Uses `pl_unexpected()` 118a.

⟨*function* `pl_scrollerror` 119a⟩≡ (127b)

```
void pl_scrollerror(Panel *g, int dir, buttons button, int num, int den){
    USED(dir, button, num, den);
    pl_unexpected(g, "scroll");
}
```

Uses `pl_unexpected()` 118a.

⟨*function* `pl_setscrollbarerror` 119b⟩≡ (127b)

```
void pl_setscrollbarerror(Panel *g, int top, int bot, int den){
    USED(top, bot, den);
    pl_unexpected(g, "setscrollbar");
}
```

Uses `pl_unexpected()` 118a.

Appendix C

Utilities

C.1 Memory management

`pl_emalloc`[120a] and `pl_erealloc`[120b] are fatal-on-failure wrappers around Plan 9's `mallocz` and `realloc`. The `setmalloctag`/`setrealloctag` calls record the caller's PC in the allocation header, so that Plan 9's `leak` tool can attribute leaked blocks back to the actual call site rather than to these wrappers.

⟨*function* `pl_emalloc` 120a⟩≡ (127b)

```
void *pl_emalloc(int n){
    void *v;

    v=mallocz(n, 1);
    if(v==nil){
        fprint(STDERR, "Can't malloc!\n");
        exits("no mem");
    }
    setmalloctag(v, getcallerpc(&n));
    return v;
}
```

⟨*function* `pl_erealloc` 120b⟩≡ (127b)

```
void *pl_erealloc(void *v, int n)
{
    v=realloc(v, n);
    if(v==nil){
        fprint(STDERR, "Can't realloc!\n");
        exits("no mem");
    }
    setrealloctag(v, getcallerpc(&v));
    return v;
}
```

Appendix D

Extra Code

D.1 include/gui/

⟨lib_gui/libpanel/tests/panels.c 121a⟩≡

⟨lib_gui/libpanel/tests/scrltest.c 121b⟩≡

⟨lib_gui/libpanel/tests/panels_test.c 121c⟩≡

D.1.1 include/gui/panel.h

⟨include/gui/panel.h 121d⟩≡

```
#pragma src "/sys/src/libpanel"
#pragma lib "libpanel.a"

⟨type Icon 22d⟩

// forward decls
typedef struct Panel Panel;  /* a Graphical User Interface element */
typedef struct Scroll Scroll;
typedef struct Rtext Rtext;  /* formattable text */
typedef struct Idol Idol;  /* A picture/text combo */

⟨struct Scroll 71c⟩
⟨struct Rtext 103c⟩
⟨struct Panel 18⟩

/*
 * Panel flags
 */
⟨constant PACK 24a⟩
⟨constant PACKN 24b⟩
⟨constant PACKE 24c⟩
⟨constant PACKS 24d⟩
⟨constant PACKW 24e⟩

⟨constant PACKCEN 24f⟩

⟨constant FILLX 24g⟩
⟨constant FILLY 24h⟩

⟨constant PLACE 23a⟩
⟨constant PLACECEN 23b⟩
```

⟨*constant* PLACES 23c⟩
⟨*constant* PLACEE 23d⟩
⟨*constant* PLACEW 23e⟩
⟨*constant* PLACEN 23f⟩
⟨*constant* PLACENE 23g⟩
⟨*constant* PLACENW 23h⟩
⟨*constant* PLACESE 23i⟩
⟨*constant* PLACESW 23j⟩

⟨*constant* EXPAND 24i⟩

⟨*constant* FIXED 111f⟩
⟨*constant* FIXEDX 111b⟩
⟨*constant* FIXEDY 111c⟩

⟨*constant* MAXX 111g⟩
⟨*constant* MAXY 111h⟩

⟨*constant* BITMAP 44b⟩
⟨*constant* USERFL 50b⟩

⟨*constant* OUT 32b⟩

```
/*
 * Priorities
 */
```
⟨*constant* PRI_NORMAL 33c⟩
⟨*constant* PRI_POPUP 33d⟩
⟨*constant* PRI_SCROLLBAR 33e⟩

```
/* Rtext.flags */
```
⟨*constant* PL_HOT 104f⟩
⟨*constant* PL_SEL 105a⟩

⟨*global* plkbfocus 33j⟩

```
//pad's stuff
```
⟨*constant* NOFLAG 19e⟩

```
// Initialization
int plinit(int);   /* initialization */

// Memory
void plfree(Panel *);   /* give back space */

// Drawing
void pldraw(Panel *, Image *);  /* display the panel on the bitmap */

// Events
void plkeyboard(Rune);   /* send a keyboard event to the appropriate Panel */
void plmouse(Panel *, Mouse *);  /* send a Mouse event to a Panel tree */

// Packing
void plpack(Panel *, Rectangle); /* figure out where to put the Panel & children */
void plmove(Panel *, Point);  /* move an already-packed panel to a new location */


//XXX
void plgrabkb(Panel *);   /* this Panel should receive keyboard events */
void plscroll(Panel *, Panel *, Panel *); /* link up scroll bars */
```

```
void plepaste(Panel *, Rune *, int); /* paste in an edit window */

/*
 * Panel creation & reinitialization functions
 */
Panel *pllabel(Panel *pl, int, Icon *);

Panel *plbutton(Panel *pl, int, Icon *, void (*)(Panel *pl, int));
Panel *plcheckbutton(Panel *pl, int, Icon *, void (*)(Panel *pl, int, int));
Panel *plradiobutton(Panel *pl, int, Icon *, void (*)(Panel *pl, int, int));

Panel *plentry(Panel *pl, int, int, char *, void (*)(Panel *pl, char *));
Panel *pledit(Panel *, int, Point, Rune *, int, void (*)(Panel *));
Panel *plmessage(Panel *pl, int, int, char *);

Panel *plslider(Panel *pl, int, Point, void(*)(Panel *pl, int, int, int));

Panel *plcanvas(Panel *pl, int, void (*)(Panel *), void (*)(Panel *pl, Mouse *));

Panel *plgroup(Panel *pl, int);
Panel *plframe(Panel *pl, int);
Panel *pllist(Panel *pl, int, char *(*)(Panel *, int), int, void(*)(Panel *pl, int, int));
Panel *plidollist(Panel*, int, Point, Font*, Idol*, void (*)(Panel*, int, void*));

Panel *plmenu(Panel *pl, int, Icon **, int, void (*)(int, int));
Panel *plmenubar(Panel *pl, int, int, Icon *, Panel *pl, Icon *, ...);
Panel *plpopup(Panel *pl, int, Panel *pl, Panel *pl, Panel *pl);
Panel *plpulldown(Panel *pl, int, Icon *, Panel *pl, int);

Panel *plscrollbar(Panel *plparent, int flags);
Panel *pltextview(Panel *, int, Point, Rtext *, void (*)(Panel *, int, Rtext *));


// setters
void plplacelabel(Panel *, int); /* label placement */
void plsetbutton(Panel *, int);  /* set or clear the mark on a button */
void plsetslider(Panel *, int, int); /* set the value of a slider */
void plesel(Panel *, int, int);  /* set the selection in an edit window */
void plescroll(Panel *, int);  /* scroll an edit window */
void plsetscroll(Panel *, Scroll); /* set scrolling information */

// getters
char *plentryval(Panel *);  /* entry delivers its value */
Rune *pleget(Panel *);   /* get the text from an edit window */
int plelen(Panel *);   /* get the length of the text from an edit window */
void plegetsel(Panel *, int *, int *); /* get the selection from an edit window */
Scroll plgetscroll(Panel *);  /* get scrolling information from panel */


// plinitxxx()
void plinitlabel(Panel *, int, Icon *);

void plinitbutton(Panel *, int, Icon *, void (*)(Panel *, int));
void plinitcheckbutton(Panel *, int, Icon *, void (*)(Panel *, int, int));
void plinitradiobutton(Panel *, int, Icon *, void (*)(Panel *, int, int));

void plinitcanvas(Panel *, int, void (*)(Panel *), void (*)(Panel *, Mouse *));
void plinitedit(Panel *, int, Point, Rune *, int, void (*)(Panel *));
void plinitentry(Panel *, int, int, char *, void (*)(Panel *, char *));
void plinitframe(Panel *, int);
```

```
void plinitgroup(Panel *, int);
void plinitidollist(Panel*, int, Point, Font*, Idol*, void (*)(Panel*, int, void*));
void plinitlist(Panel *, int, char *(*)(Panel *, int), int, void(*)(Panel *, int, int));
void plinitmenu(Panel *, int, Icon **, int, void (*)(int, int));
void plinitmessage(Panel *, int, int, char *);
void plinitpopup(Panel *, int, Panel *, Panel *, Panel *);
void plinitpulldown(Panel *, int, Icon *, Panel *, int);
void plinitscrollbar(Panel *parent, int flags);
void plinitslider(Panel *, int, Point, void(*)(Panel *, int, int, int));
void plinittextview(Panel *, int, Point, Rtext *, void (*)(Panel *, int, Rtext *));

/*
 * Rtext constructors & destructor
 */
Rtext *plrtstr(Rtext **, int, int, Font *, char *, int, void *);
Rtext *plrtbitmap(Rtext **, int, int, Image *, int, void *);
Rtext *plrtpanel(Rtext **, int, int, Panel *, void *);
void plrtfree(Rtext *);
void plrtseltext(Rtext *, Rtext *, Rtext *);
char *plrtsnarftext(Rtext *);

int plgetpostextview(Panel *);
void plsetpostextview(Panel *, int);

/*
 * Idols
 */
Idol *plmkidol(Idol**, Image*, Image*, char*, void*);
void plfreeidol(Idol*);
Point plidolsize(Idol*, Font*, int);
void *plidollistgetsel(Panel*);

/*
 * Snarf
 */
void plputsnarf(char *);
char *plgetsnarf(void);
void plsnarf(Panel *);   /* snarf a panel */
void plpaste(Panel *);   /* paste a panel */
```

D.1.2 include/gui/rtext.h

⟨*constant* PL_NOPBIT 124a⟩≡ (125b)
```
#define PL_NOPBIT 4
```

⟨*constant* PL_NARGBIT 124b⟩≡ (125b)
```
#define PL_NARGBIT 12
```

⟨*constant* PL_ARGMASK 124c⟩≡ (125b)
```
#define PL_ARGMASK ((1<<PL_NARGBIT)-1)
```

⟨*function* PL_SPECIAL 124d⟩≡ (125b)
```
#define PL_SPECIAL(op) (((-1<<PL_NOPBIT)|op)<<PL_NARGBIT)
```

⟨*function* PL_OP 124e⟩≡ (125b)
```
#define PL_OP(t) ((t)&~PL_ARGMASK)
```

⟨*function* PL_ARG 124f⟩≡ (125b)
```
#define PL_ARG(t) ((t)&PL_ARGMASK)
```

⟨*constant* PL_TAB 125a⟩≡ (125b)
```
#define PL_TAB  PL_SPECIAL(0)  /* # of tab stops before text */
```

⟨include/gui/rtext.h 125b⟩≡
```
/*
 * Rtext definitions
 */
⟨constant PL_NOPBIT 124a⟩
⟨constant PL_NARGBIT 124b⟩
⟨constant PL_ARGMASK 124c⟩
⟨function PL_SPECIAL 124d⟩

⟨function PL_OP 124e⟩
⟨function PL_ARG 124f⟩
⟨constant PL_TAB 125a⟩

void pltabsize(int, int);   /* set min tab and tab size */
```

D.2 lib_gui/libpanel/

D.2.1 lib_gui/libpanel/pldefs.h

⟨*enum* _anon_ (lib_gui/libpanel/pldefs.h)2 125c⟩≡ (125e)
```
/*
 * Scroll flags
 */
enum{
    SCROLLUP,
    SCROLLDOWN,
    SCROLLABSY,
    SCROLLLEFT,
    SCROLLRIGHT,
    SCROLLABSX,
};
```

⟨*struct* Textwin 125d⟩≡ (125e)
```
struct Textwin{
    Rune *text, *etext, *eslack; /* text, with some slack off the end */
    int top, bot;   /* range of runes visible on screen */
    int sel0, sel1;   /* selection */
    Point *loc, *eloc;  /* ul corners of visible runes (+1 more at end!) */
    Image *b;   /* bitmap the text is drawn in */
    Rectangle r;   /* rectangle the text is drawn in */
    Font *font;   /* font text is drawn in */
    int hgt;   /* same as font->height */
    int tabstop;   /* tab settings are every tabstop pixels */
    int mintab;   /* the minimum size of a tab */
};
```

⟨lib_gui/libpanel/pldefs.h 125e⟩≡
```
/*
 * Definitions for internal use only
 */

/*
 * Variable-font text routines
 * These could make a separate library.
```

```
 */
int pl_rtfmt(Rtext *, int);
void pl_rtdraw(Image *, Rectangle, Rtext *, int);
void pl_rtredraw(Image *, Rectangle, Rtext *, int, int);
Rtext *pl_rthit(Rtext *, int, Point, Point);

⟨constant LEAF 21b⟩
⟨constant INVIS 87b⟩
⟨constant REMOUSE 32d⟩

⟨enum Style 22a⟩
⟨enum _anon_ (lib_gui/libpanel/pldefs.h)2 125c⟩
⟨enum Direction 25c⟩

Panel *pl_newpanel(Panel *, int); /* make a new Panel, given parent & data size */
void *pl_emalloc(int);   /* allocate some space, exit on error */
void *pl_erealloc(void*,int);  /* reallocate some space, exit on error */

void pl_print(Panel *);   /* print a Panel tree */
Panel *pl_ptinpanel(Point, Panel *); /* highest-priority subpanel containing point */

/*
 * Drawing primitives
 */
int pl_drawinit(int);
Rectangle pl_box(Image *, Rectangle, int);
Rectangle pl_outline(Image *, Rectangle, int);
Point pl_boxsize(Point, int);
void pl_interior(int, Point *, Point *);
void pl_drawicon(Image *, Rectangle, int, int, Icon *);
Rectangle pl_check(Image *, Rectangle, int);
Rectangle pl_radio(Image *, Rectangle, int);
int pl_ckwid(void);
void pl_sliderupd(Image *, Rectangle, int, int, int);
void pl_invis(Panel *, bool);
Point pl_iconsize(int, Icon *);
void pl_highlight(Image *, Rectangle);
void pl_clr(Image *, Rectangle);
void pl_fill(Image *, Rectangle);
void pl_cpy(Image *, Point, Rectangle);

/*
 * Rune mangling functions
 */
int pl_idchar(int);
int pl_rune1st(int);
char *pl_nextrune(char *);
int pl_runewidth(Font *, char *);

/*
 * Fixed-font Text-window routines
 * These could be separated out into a separate library.
 */
typedef struct Textwin Textwin;
⟨struct Textwin 125d⟩

Textwin *twnew(Image *, Font *, Rune *, int);
void twfree(Textwin *);
void twhilite(Textwin *, int, int, int);
void twselect(Textwin *, Mouse *);
```

```
void twreplace(Textwin *, int, int, Rune *, int);
void twscroll(Textwin *, int);
int twpt2rune(Textwin *, Point);
void twreshape(Textwin *, Rectangle);
void twmove(Textwin *, Point);
void plemove(Panel *, Point);
```

Uses Textwin 125e.

D.2.2 lib_gui/libpanel/init.c

⟨lib_gui/libpanel/init.c 127a⟩≡
⟨libpanel *includes* 14⟩

⟨*function* plinit 26a⟩

D.2.3 lib_gui/libpanel/mem.c

⟨lib_gui/libpanel/mem.c 127b⟩≡
⟨libpanel *includes* 14⟩

⟨*function* pl_emalloc 120a⟩
⟨*function* pl_erealloc 120b⟩

⟨*function* pl_unexpected 118a⟩
⟨*function* pl_drawerror 118b⟩
⟨*function* pl_hiterror 118c⟩
⟨*function* pl_typeerror 118d⟩
⟨*function* pl_getsizeerror 118e⟩
⟨*function* pl_childspaceerror 118f⟩
⟨*function* pl_scrollerror 119a⟩
⟨*function* pl_setscrollbarerror 119b⟩

⟨*function* pl_prinormal 33g⟩

⟨*function* pl_newpanel 19a⟩
⟨*function* plfree 19c⟩

D.2.4 lib_gui/libpanel/draw.c

⟨lib_gui/libpanel/draw.c 127c⟩≡
⟨libpanel *includes* 14⟩

⟨*constant* PWID 46a⟩
⟨*constant* BWID 49e⟩
⟨*constant* FWID 67d⟩
⟨*constant* SPACE 30b⟩
⟨*constant* CKSIZE 58b⟩
⟨*constant* CKSPACE 59a⟩
⟨*constant* CKWID 57b⟩
⟨*constant* CKINSET 57a⟩
⟨*constant* CKBORDER 57c⟩

⟨*global* plldepth 26b⟩

⟨*globals* pl_xxx 26c⟩

⟨*function* pl_drawinit 26d⟩

⟨*function* `pl_relief` 29a⟩
⟨*function* `pl_boxoutline` 30a⟩
⟨*function* `pl_outline` 29b⟩
⟨*function* `pl_box` 29c⟩
⟨*function* `pl_boxsize` 30c⟩
⟨*function* `pl_interior` 30d⟩

⟨*function* `pl_drawicon` 46b⟩
⟨*function* `pl_radio` 58g⟩
⟨*function* `pl_check` 56f⟩
⟨*function* `pl_ckwid` 58a⟩
⟨*function* `pl_sliderupd` 61b⟩

```
void pl_draw1(Panel *p, Image *b);
```

⟨*function* `pl_drawall` 28c⟩
⟨*function* `pl_draw1` 28b⟩
⟨*function* `pldraw` 28a⟩
⟨*function* `pl_invis` 87a⟩
⟨*function* `pl_iconsize` 46c⟩
⟨*function* `pl_highlight` 74b⟩
⟨*function* `pl_clr` 105c⟩
⟨*function* `pl_fill` 74c⟩
⟨*function* `pl_cpy` 77c⟩

D.2.5 lib_gui/libpanel/event.c

⟨`lib_gui/libpanel/event.c` 128a⟩≡
⟨`libpanel` *includes* 14⟩

⟨*function* `plgrabkb` 33k⟩
⟨*function* `plkeyboard` 34a⟩

⟨*function* `pl_ptinpanel` 32a⟩
⟨*function* `plmouse` 31a⟩

D.2.6 lib_gui/libpanel/print.c

⟨`lib_gui/libpanel/print.c` 128b⟩≡
⟨`libpanel` *includes* 14⟩

⟨*function* `pl_iprint` 116c⟩
⟨*function* `pl_ipprint` 116d⟩
⟨*function* `pl_print` 116b⟩

D.2.7 lib_gui/libpanel/label.c

⟨`lib_gui/libpanel/label.c` 128c⟩≡
⟨`libpanel` *includes* 14⟩

```
typedef struct Label Label;
```

⟨*struct* `Label` 44a⟩

⟨*function* `pl_drawlabel` 45b⟩
⟨*function* `pl_hitlabel` 47b⟩
⟨*function* `pl_typelabel` 47c⟩

⟨*function* `pl_getsizelabel` 47d⟩
⟨*function* `pl_childspacelabel` 47f⟩
⟨*function* `plinitlabel` 44d⟩
⟨*function* `pllabel` 44c⟩
⟨*function* `plplacelabel` 45a⟩

Uses `Label` 44a.

D.2.8 lib_gui/libpanel/button.c

⟨`lib_gui/libpanel/button.c` 129a⟩≡

```
⟨libpanel includes 14⟩

typedef struct Button Button;

⟨struct Button 53a⟩

/*
 * Button types
 */
⟨constant BUTTON 53b⟩
⟨constant CHECK 53c⟩
⟨constant RADIO 53d⟩

⟨function pl_drawbutton 54d⟩
⟨function pl_hitbutton 55b⟩
⟨function pl_typebutton 55c⟩
⟨function pl_getsizebutton 56a⟩
⟨function pl_childspacebutton 56b⟩
⟨function pl_initbtype 53g⟩
⟨function pl_buttonhit 54c⟩
⟨function plinitbutton 54b⟩
⟨function plinitcheckbutton 56d⟩
⟨function plinitradiobutton 58e⟩
⟨function plbutton 54a⟩
⟨function plcheckbutton 56c⟩
⟨function plradiobutton 58d⟩

⟨function pl_hitmenu 83b⟩
⟨function plinitmenu 82d⟩
⟨function plmenu 82c⟩
⟨function plsetbutton 58c⟩
```

Uses `Button` 53a.

D.2.9 lib_gui/libpanel/entry.c

⟨`lib_gui/libpanel/entry.c` 129b⟩≡

```
⟨libpanel includes 14⟩
#include <keyboard.h>

typedef struct Entry Entry;

⟨struct Entry 47g⟩
⟨constant SLACK 110b⟩

⟨function pl_snarfentry 109d⟩
⟨function pl_pasteentry 110a⟩
⟨function pl_drawentry 49c⟩
```

⟨*function* `pl_hitentry` 50e⟩
⟨*function* `pl_typeentry` 51a⟩
⟨*function* `pl_getsizeentry` 52c⟩
⟨*function* `pl_childspaceentry` 52e⟩
⟨*function* `pl_freeentry` 48d⟩
⟨*function* `plinitentry` 48c⟩
⟨*function* `plentry` 48b⟩
⟨*function* `plentryval` 52f⟩

Uses `Entry` 47g.

D.2.10 `lib_gui/libpanel/edit.c`

⟨`lib_gui/libpanel/edit.c` 130⟩≡

```
/*
 * Interface includes:
 * void plescroll(Panel *p, int top);
 *  move the given character position onto the top line
 * void plegetsel(Panel *p, int *sel0, int *sel1);
 *  read the selection back
 * int plelen(Panel *p);
 *  read the length of the text back
 * Rune *pleget(Panel *p);
 *  get a pointer to the text
 * void plesel(Panel *p, int sel0, int sel1);
 *  set the selection -- adjusts hiliting
 * void plepaste(Panel *p, Rune *text, int ntext);
 *  replace the selection with the given text
 */
```

⟨`libpanel` *includes* 14⟩

```
#include <keyboard.h>

typedef struct Edit Edit;
```

⟨*struct* `Edit` 95a⟩
⟨*function* `pl_drawedit` 96b⟩

⟨*function* `pl_snarfedit` 110d⟩
⟨*function* `pl_pasteedit` 110e⟩

⟨*function* `pl_hitedit` 96c⟩
⟨*function* `pl_scrolledit` 98c⟩
⟨*function* `pl_typeedit` 97⟩
⟨*function* `pl_getsizeedit` 98a⟩
⟨*function* `pl_childspaceedit` 98b⟩
⟨*function* `pl_freeedit` 96a⟩
⟨*function* `plinitedit` 95c⟩
⟨*function* `pledit` 95b⟩
⟨*function* `plescroll` 99a⟩
⟨*function* `plegetsel` 99b⟩
⟨*function* `plelen` 99c⟩
⟨*function* `pleget` 99d⟩
⟨*function* `plesel` 100a⟩
⟨*function* `plepaste` 100b⟩
⟨*function* `plemove` 100c⟩

Uses `Edit` 95a.

D.2.11 lib_gui/libpanel/slider.c

⟨lib_gui/libpanel/slider.c 131a⟩≡
  ⟨libpanel *includes* 14⟩

```
typedef struct Slider Slider;
```

  ⟨*struct* Slider 59d⟩

  ⟨*function* pl_drawslider 61a⟩
  ⟨*function* pl_hitslider 62a⟩
  ⟨*function* pl_typeslider 63b⟩
  ⟨*function* pl_getsizeslider 63c⟩
  ⟨*function* pl_childspaceslider 63d⟩
  ⟨*function* plinitslider 60c⟩
  ⟨*function* plslider 60b⟩
  ⟨*function* plsetslider 63e⟩

Uses Slider 59d.

D.2.12 lib_gui/libpanel/canvas.c

⟨lib_gui/libpanel/canvas.c 131b⟩≡
  ⟨libpanel *includes* 14⟩

```
typedef struct Canvas Canvas;
```

  ⟨*struct* Canvas 64a⟩

  ⟨*function* pl_drawcanvas 64d⟩
  ⟨*function* pl_hitcanvas 65a⟩
  ⟨*function* pl_typecanvas 65b⟩
  ⟨*function* pl_getsizecanvas 65c⟩
  ⟨*function* pl_childspacecanvas 65d⟩
  ⟨*function* plinitcanvas 64c⟩
  ⟨*function* plcanvas 64b⟩

Uses Canvas 64a.

D.2.13 lib_gui/libpanel/frame.c

⟨lib_gui/libpanel/frame.c 131c⟩≡
  ⟨libpanel *includes* 14⟩

  ⟨*function* pl_drawframe 67a⟩
  ⟨*function* pl_hitframe 67e⟩
  ⟨*function* pl_typeframe 67f⟩
  ⟨*function* pl_getsizeframe 67g⟩
  ⟨*function* pl_childspaceframe 68b⟩
  ⟨*function* plinitframe 66b⟩
  ⟨*function* plframe 66a⟩

D.2.14 lib_gui/libpanel/group.c

⟨lib_gui/libpanel/group.c 131d⟩≡
  ⟨libpanel *includes* 14⟩

  ⟨*function* pl_drawgroup 69a⟩
  ⟨*function* pl_hitgroup 69b⟩

⟨*function* `pl_typegroup` 69c⟩
⟨*function* `pl_getsizegroup` 69d⟩
⟨*function* `pl_childspacegroup` 69e⟩
⟨*function* `plinitgroup` 68e⟩
⟨*function* `plgroup` 68d⟩

D.2.15 `lib_gui/libpanel/list.c`

⟨`lib_gui/libpanel/list.c` 132a⟩≡
⟨`libpanel` *includes* 14⟩

```
typedef struct List List;
```

⟨*struct* `List` 71g⟩

⟨*function* `pl_listsel` 74a⟩
⟨*function* `pl_liststrings` 73e⟩
⟨*function* `pl_drawlist` 73d⟩
⟨*function* `pl_hitlist` 74f⟩
⟨*function* `pl_scrolllist` 76b⟩
⟨*function* `pl_typelist` 75a⟩
⟨*function* `pl_getsizelist` 75b⟩
⟨*function* `pl_childspacelist` 75c⟩
⟨*function* `plinitlist` 72c⟩
⟨*function* `pllist` 72b⟩

Uses `List` 71g.

D.2.16 `lib_gui/libpanel/message.c`

⟨`lib_gui/libpanel/message.c` 132b⟩≡
⟨`libpanel` *includes* 14⟩

```
typedef struct Message Message;
```

⟨*struct* `Message` 92a⟩

⟨*function* `pl_textmsg` 93b⟩
⟨*function* `pl_foldsize` 94d⟩
⟨*function* `pl_drawmessage` 93a⟩
⟨*function* `pl_hitmessage` 94a⟩
⟨*function* `pl_typemessage` 94b⟩
⟨*function* `pl_getsizemessage` 94c⟩
⟨*function* `pl_childspacemessage` 94e⟩
⟨*function* `plinitmessage` 92c⟩
⟨*function* `plmessage` 92b⟩

Uses `Message` 92a.

D.2.17 `lib_gui/libpanel/pack.c`

⟨`lib_gui/libpanel/pack.c` 132c⟩≡
⟨`libpanel` *includes* 14⟩

⟨*function* `pl_max` 38a⟩
⟨*function* `pl_sizesibs` 37b⟩
⟨*function* `pl_sizereq` 36e⟩
⟨*function* `pl_getshare` 43⟩

⟨*function* `pl_setrect` 38c⟩
⟨*function* `plpack` 35⟩
⟨*function* `plmove` 86b⟩

D.2.18 lib_gui/libpanel/popup.c

⟨`lib_gui/libpanel/popup.c` 133a⟩≡
```
/*
 * popup
 * looks like a group, except diverts hits on certain buttons to
 * panels that it temporarily pops up.
 */
```
⟨`libpanel` *includes* 14⟩

```
typedef struct Popup Popup;
```

⟨*struct* `Popup` 83c⟩

⟨*function* `pl_drawpopup` 85e⟩
⟨*function* `pl_hitpopup` 84d⟩
⟨*function* `pl_typepopup` 85d⟩
⟨*function* `pl_getsizepopup` 87d⟩
⟨*function* `pl_childspacepopup` 87e⟩
⟨*function* `pl_pripopup` 33h⟩
⟨*function* `plinitpopup` 84b⟩
⟨*function* `plpopup` 84a⟩

Uses `Popup` 83c.

D.2.19 lib_gui/libpanel/pulldown.c

⟨`lib_gui/libpanel/pulldown.c` 133b⟩≡
```
/*
 * pulldown
 * makes a button that pops up a panel when hit
 */
```
⟨`libpanel` *includes* 14⟩

```
typedef struct Pulldown Pulldown;
```

⟨*struct* `Pulldown` 87f⟩

⟨*function* `pl_drawpulldown` 88c⟩
⟨*function* `pl_hitpulldown` 88d⟩
⟨*function* `pl_typepulldown` 90b⟩
⟨*function* `pl_getsizepulldown` 90c⟩
⟨*function* `pl_childspacepulldown` 90d⟩
⟨*function* `plinitpulldown` 88b⟩
⟨*function* `plpulldown` 88a⟩
⟨*function* `plmenubar` 91a⟩

Uses `Pulldown` 133b.

D.2.20 lib_gui/libpanel/rtext.c

⟨*constant* `LEAD` 133c⟩≡ (136c)
```
#define LEAD 4 /* extra space between lines */
```

⟨*constant* BORD 134a⟩≡ (136c)
```
#define BORD 2 /* extra border for images */
```

⟨*function* pltabsize 134b⟩≡ (136c)
```
void pltabsize(int min, int size){
    pl_tabmin=min;
    pl_tabsize=size;
}
```
Uses pl_tabmin 136c and pl_tabsize 136c.

⟨*function* pl_space 134c⟩≡ (136c)
```
int pl_space(int space, int pos, int indent){
    if(space>=0) return space;
    switch(PL_OP(space)){
    default:
        return 0;
    case PL_TAB:
        return ((pos-indent+pl_tabmin)/pl_tabsize+PL_ARG(space))*pl_tabsize+indent-pos;
    }
}
```
Uses pl_tabmin 136c and pl_tabsize 136c.

⟨*function* pl_rtfmt 134d⟩≡ (136c)
```
/*
 * initialize rectangles & nextlines of text starting at t,
 * galley width is wid.  Returns the total length of the text
 */
int pl_rtfmt(Rtext *t, int wid){
    Rtext *tp, *eline;
    int ascent, descent, x, space, a, d, w, topy, indent;
    Point p;
    p=Pt(0,0);
    eline=t;
    while(t){
        ascent=0;
        descent=0;
        indent=space=pl_space(t->indent, 0, 0);
        x=0;
        tp=t;
        for(;;){
            if(tp->b){
                a=tp->b->r.max.y-tp->b->r.min.y+BORD;
                d=BORD;
                w=tp->b->r.max.x-tp->b->r.min.x+BORD*2;
            }
            else if(tp->p){
                /* what if plpack fails? */
                plpack(tp->p, Rect(0,0,wid,wid));
                plmove(tp->p, subpt(Pt(0,0), tp->p->r.min));
                a=tp->p->r.max.y-tp->p->r.min.y;
                d=0;
                w=tp->p->r.max.x-tp->p->r.min.x;
            }
            else{
                a=tp->font->ascent;
                d=tp->font->height-a;
                w=tp->wid=stringwidth(tp->font, tp->text);
            }
            if(x+w+space>wid) break;
            if(a>ascent) ascent=a;
```

```
            if(d>descent) descent=d;
            x+=w+space;
            tp=tp->next;
            if(tp==0){
                eline=0;
                break;
            }
            space=pl_space(tp->space, x, indent);
            if(space) eline=tp;
        }
        if(eline==t){ /* No progress!  Force fit the first block! */
            if(tp==t){
                if(a>ascent) ascent=a;
                if(d>descent) descent=d;
                eline=tp->next;
            }else
                eline=tp;
        }
        topy=p.y;
        p.y+=ascent;
        p.x=indent=pl_space(t->indent, 0, 0);
        for(;;){
            t->topy=topy;
            t->r.min.x=p.x;
            if(t->b){
                t->r.max.y=p.y+BORD;
                t->r.min.y=p.y-(t->b->r.max.y-t->b->r.min.y)-BORD;
                p.x+=(t->b->r.max.x-t->b->r.min.x)+BORD*2;
            }
            else if(t->p){
                t->r.max.y=p.y;
                t->r.min.y=p.y-t->p->r.max.y;
                p.x+=t->p->r.max.x;
            }
            else{
                t->r.min.y=p.y-t->font->ascent;
                t->r.max.y=t->r.min.y+t->font->height;
                p.x+=t->wid;
            }
            t->r.max.x=p.x;
            t->nextline=eline;
            t=t->next;
            if(t==eline) break;
            p.x+=pl_space(t->space, p.x, indent);
        }
        p.y+=descent+LEAD;
    }
    return p.y;
}
```

Uses BORD-8 134a, LEAD-7 133c, pl_space() 134c, plmove() 86b, and plpack() 35.

⟨*function* pl_stuffbitmap 135⟩≡ (136c)

```
/*
 * If we draw the text in a backup bitmap and copy it onto the screen,
 * the bitmap pointers in all the subpanels point to the wrong bitmap.
 * This code fixes them.
 */
void pl_stuffbitmap(Panel *p, Image *b){
    p->b=b;
    for(p=p->child;p;p=p->next)
```

```
        pl_stuffbitmap(p, b);
  }
```

Uses pl_stuffbitmap() 135.

⟨*function* pl_reposition 136a⟩≡ (136c)

```
  /*
   * Reposition text already drawn in the window.
   * We just move the pixels and update the positions of any
   * enclosed panels
   */
  void pl_reposition(Rtext *t, Image *b, Point p, Rectangle r){
      Point offs;
      pl_cpy(b, p, r);
      offs=subpt(p, r.min);
      for(;t;t=t->next)
          if(!eqrect(t->r, Rect(0,0,0,0)) && !t->b && t->p)
              plmove(t->p, offs);
  }
```

Uses pl_cpy() 77c and plmove() 86b.

⟨*function* plrtsnarftext 136b⟩≡ (136c)

```
  char *plrtsnarftext(Rtext *w){
      char *b, *p, *e, *t;
      int n;

      b=p=e=0;
      for(; w; w = w->next){
          if((w->flags&PL_SEL)==0 || w->text==0)
              continue;
          n = strlen(w->text)+64;
          if(p+n >= e){
              n = (p+n+64)-b;
              t = pl_erealloc(b, n);
              p = t+(p-b);
              e = t+n;
              b = t;
          }
          if(w->space == 0)
              p += sprint(p, "%s", w->text);
          else if(w->space > 0)
              p += sprint(p, " %s", w->text);
          else if(PL_OP(w->space) == PL_TAB)
              p += sprint(p, "\t%s", w->text);
          if(w->nextline == w->next)
              p += sprint(p, "\n");
      }
      return b;
  }
```

Uses pl_erealloc() 120b.

⟨lib_gui/libpanel/rtext.c 136c⟩≡

```
  /*
   * Rich text with images.
   * Should there be an offset field, to do subscripts & kerning?
   */
  ⟨libpanel includes 14⟩
  #include "rtext.h"

  ⟨constant LEAD 133c⟩
```

⟨*constant* BORD 134a⟩

⟨*function* pl_rtnew 104a⟩
⟨*function* plrtpanel 104b⟩
⟨*function* plrtstr 104c⟩
⟨*function* plrtbitmap 104d⟩
⟨*function* plrtfree 104e⟩
```
int pl_tabmin;
int pl_tabsize;
```
⟨*function* pltabsize 134b⟩
⟨*function* pl_space 134c⟩
⟨*function* pl_rtfmt 134d⟩

⟨*function* pl_stuffbitmap 135⟩

⟨*function* pl_rtdraw 105b⟩
⟨*function* pl_reposition 136a⟩
⟨*function* pl_rtredraw 106a⟩
⟨*function* pl_rthit 106b⟩

⟨*function* plrtseltext 106c⟩

⟨*function* plrtsnarftext 136b⟩

D.2.21 lib_gui/libpanel/scroll.c

⟨*function* plgetscroll 137a⟩≡ (137c)
```
Scroll plgetscroll(Panel *p){
    return p->scr;
}
```

⟨*function* plsetscroll 137b⟩≡ (137c)
```
void plsetscroll(Panel *p, Scroll s){
    if(p->scroll){
        if(s.size.x) p->scroll(p, HORIZ, 2, s.pos.x, s.size.x);
        if(s.size.y) p->scroll(p, VERT, 2, s.pos.y, s.size.y);
    }
}
```
Uses HORIZ 125c and VERT.

⟨lib_gui/libpanel/scroll.c 137c⟩≡
⟨libpanel *includes* 14⟩

⟨*function* plscroll 71a⟩
⟨*function* plgetscroll 137a⟩
⟨*function* plsetscroll 137b⟩

D.2.22 lib_gui/libpanel/scrollbar.c

⟨lib_gui/libpanel/scrollbar.c 137d⟩≡
⟨libpanel *includes* 14⟩

```
typedef struct Scrollbar Scrollbar;
```

⟨*struct* Scrollbar 77d⟩
⟨*constant* SBWID 78d⟩

⟨*function* pl_drawscrollbar 79b⟩

⟨*function* `pl_hitscrollbar` 79c⟩
⟨*function* `pl_typescrollbar` 80c⟩
⟨*function* `pl_getsizescrollbar` 80d⟩
⟨*function* `pl_childspacescrollbar` 80e⟩
⟨*function* `pl_setscrollbarscrollbar` 81b⟩
⟨*function* `pl_priscrollbar` 33i⟩
⟨*function* `plinitscrollbar` 78a⟩
⟨*function* `plscrollbar` 77f⟩

Uses `Scrollbar` 77d.

D.2.23 lib_gui/libpanel/snarf.c

⟨`lib_gui/libpanel/snarf.c` 138a⟩≡
⟨`libpanel` *includes* 14⟩

⟨*function* `plputsnarf` 108c⟩
⟨*function* `plgetsnarf` 108d⟩
⟨*function* `plsnarf` 109a⟩
⟨*function* `plpaste` 109b⟩

D.2.24 lib_gui/libpanel/textview.c

⟨*function* `pl_setscrpos` 138b⟩≡ (140a)

```
void pl_setscrpos(Panel *p, Textview *tp, Rectangle r){
    Panel *sb;
    int lo, hi;
    lo=tp->yoffs;
    hi=lo+r.max.y-r.min.y; /* wrong? */
    sb=p->yscroller;
    if(sb && sb->setscrollbar) sb->setscrollbar(sb, lo, hi, tp->thgt);
}
```

⟨*function* `pl_passon` 138c⟩≡ (140a)

```
/*
 * If t is a panel word, pass the mouse event on to it
 */
void pl_passon(Rtext *t, Mouse *m){
    if(t && t->b==0 && t->p!=0)
        plmouse(t->p, m);
}
```

Uses `plmouse()` 31a.

⟨*function* `pl_scrolltextview` 138d⟩≡ (140a)

```
void pl_scrolltextview(Panel *p, int dir, int buttons, int num, int den){
    int yoffs;
    Point ul, size;
    Textview *tp;
    Rectangle r;
    if(dir!=VERT) return;

    tp=p->data;
    ul=p->r.min;
    size=subpt(p->r.max, p->r.min);
    pl_interior(p->state, &ul, &size);
    switch(buttons){
    default:
        SET(yoffs);
```

```
            break;
        case 1:  /* left -- top moves to pointer */
            yoffs=(vlong)tp->yoffs-num*size.y/den;
            if(yoffs<0) yoffs=0;
            break;
        case 2:  /* middle -- absolute index of file */
            yoffs=(vlong)tp->thgt*num/den;
            break;
        case 4:  /* right -- line pointed at moves to top */
            yoffs=tp->yoffs+(vlong)num*size.y/den;
            if(yoffs>tp->thgt) yoffs=tp->thgt;
            break;
        }
        if(yoffs!=tp->yoffs){
            r=pl_outline(p->b, p->r, p->state);
            pl_rtredraw(p->b, r, tp->text, yoffs, tp->yoffs);
            p->scr.pos.y=tp->yoffs=yoffs;
            pl_setscrpos(p, tp, r);
        }
    }
```

Uses VERT, pl_interior() 30d, pl_outline() 29b, pl_rtredraw() 106a, and pl_setscrpos() 138b.

⟨*function* pl_pritextview 139a⟩≡ (140a)

```
    /*
     * Priority depends on what thing inside the panel we're pointing at.
     */
    int pl_pritextview(Panel *p, Point xy){
        Point ul, size;
        Textview *tp;
        Rtext *h;
        tp=p->data;
        ul=p->r.min;
        size=subpt(p->r.max, p->r.min);
        pl_interior(p->state, &ul, &size);
        h=pl_rthit(tp->text, tp->yoffs, xy, ul);
        if(h && h->b==0 && h->p!=0){
            p=pl_ptinpanel(xy, h->p);
            if(p) return p->pri(p, xy);
        }
        return PRI_NORMAL;
    }
```

Uses pl_interior() 30d, pl_ptinpanel() 32a, and pl_rthit() 106b.

⟨*function* pl_snarftextview 139b⟩≡ (140a)

```
    char* pl_snarftextview(Panel *p){
        return plrtsnarftext(((Textview *)p->data)->text);
    }
```

Uses plrtsnarftext() 136b.

⟨*function* plgetpostextview 139c⟩≡ (140a)

```
    int plgetpostextview(Panel *p){
        return ((Textview *)p->data)->yoffs;
    }
```

⟨*function* plsetpostextview 139d⟩≡ (140a)

```
    void plsetpostextview(Panel *p, int yoffs){
        ((Textview *)p->data)->yoffs=yoffs;
        pldraw(p, p->b);
    }
```

Uses pldraw() 28a.

⟨lib_gui/libpanel/textview.c 140a⟩≡
```
/*
 * Fonted text viewer, calls out to code in rtext.c
 *
 * Should redo this to copy the already-visible parts on scrolling & only
 * update the newly appearing stuff -- then the offscreen assembly bitmap can go away.
 */
⟨libpanel includes 14⟩

typedef struct Textview Textview;
⟨struct Textview 100d⟩

⟨function pl_setscrpos 138b⟩
⟨function pl_drawtextview 101c⟩
⟨function pl_passon 138c⟩
⟨function pl_hittextview 102a⟩
⟨function pl_scrolltextview 138d⟩
⟨function pl_typetextview 102b⟩
⟨function pl_getsizetextview 103a⟩
⟨function pl_childspacetextview 103b⟩
⟨function pl_pritextview 139a⟩

⟨function pl_snarftextview 139b⟩

⟨function plinittextview 101b⟩
⟨function pltextview 101a⟩
⟨function plgetpostextview 139c⟩
⟨function plsetpostextview 139d⟩
```
Uses Textview 100d.

D.2.25 lib_gui/libpanel/textwin.c

⟨function tw_before 140b⟩≡ (148d)
```
/*
 * Is text at point a before or after that at point b?
 */
int tw_before(Textwin *t, Point a, Point b){
    return a.y<b.y || a.y<b.y+t->hgt && a.x<b.x;
}
```

⟨function twpt2rune 140c⟩≡ (148d)
```
/*
 * Return the character index indicated by point p, or -1
 * if its off-screen.  The screen must be up-to-date.
 *
 * Linear search should be binary search.
 */
int twpt2rune(Textwin *t, Point p){
    Point *el, *lp;
    el=t->loc+(t->bot-t->top);
    for(lp=t->loc;lp!=el;lp++)
        if(tw_before(t, p, *lp)){
            if(lp==t->loc) return t->top;
            return lp-t->loc+t->top-1;
        }
    return t->bot;
}
```
Uses tw_before() 140b.

⟨*function* tw_rune2pt 141a⟩≡ (148d)

```
/*
 * Return ul corner of the character with the given index
 */
Point tw_rune2pt(Textwin *t, int i){
    if(i<t->top) return t->r.min;
    if(i>t->bot) return t->r.max;
    return t->loc[i-t->top];
}
```

⟨*function* tw_storeloc 141b⟩≡ (148d)

```
/*
 * Store p at t->loc[l], extending t->loc if necessary
 */
void tw_storeloc(Textwin *t, int l, Point p){
    int nloc;
    if(l>=t->eloc-t->loc){
        nloc=l+SLACK;
        t->loc=pl_erealloc(t->loc, nloc*sizeof(Point));
        t->eloc=t->loc+nloc;
    }
    t->loc[l]=p;
}
```

Uses SLACK-9 49a and pl_erealloc() 120b.

⟨*function* tw_setloc 141c⟩≡ (148d)

```
/*
 * Set the locations at which the given runes should appear.
 * Returns the index of the first rune not set, which might not
 * be last because we reached the bottom of the window.
 *
 * N.B. this zaps the loc of r[last], so that value should be saved first,
 * if it's important.
 */
int tw_setloc(Textwin *t, int first, int last, Point ul){
    Rune *r, *er;
    int x, dt, lp;
    char buf[UTFmax+1];
    er=t->text+last;
    for(r=t->text+first,lp=first-t->top;r!=er && ul.y+t->hgt<=t->r.max.y;r++,lp++){
        tw_storeloc(t, lp, ul);
        switch(*r){
        case '\n':
            ul.x=t->r.min.x;
            ul.y+=t->hgt;
            break;
        case '\t':
            x=ul.x-t->r.min.x+t->mintab+t->tabstop;
            x-=x%t->tabstop;
            ul.x=x+t->r.min.x;
            if(ul.x>t->r.max.x){
                ul.x=t->r.min.x;
                ul.y+=t->hgt;
                tw_storeloc(t, lp, ul);
                if(ul.y+t->hgt>t->r.max.y) return r-t->text;
                ul.x+=+t->tabstop;
            }
            break;
        default:
            buf[runetochar(buf, r)]='\0';
```

```
            dt=stringwidth(t->font, buf);
            ul.x+=dt;
            if(ul.x>t->r.max.x){
                ul.x=t->r.min.x;
                ul.y+=t->hgt;
                tw_storeloc(t, lp, ul);
                if(ul.y+t->hgt>t->r.max.y) return r-t->text;
                ul.x+=dt;
            }
            break;
        }
    }
    tw_storeloc(t, lp, ul);
    return r-t->text;
}
```

Uses `tw_storeloc()` 141b.

⟨*function* `tw_draw` 142⟩≡ (148d)

```
/*
 * Draw the given runes at their locations.
 * Bug -- saving up multiple characters would
 * reduce the number of calls to string,
 * and probably make this a lot faster.
 */
void tw_draw(Textwin *t, int first, int last){
    Rune *r, *er;
    Point *lp, ul, ur;
    char buf[UTFmax+1];
    if(first<t->top) first=t->top;
    if(last>t->bot) last=t->bot;
    if(last<=first) return;
    er=t->text+last;
    for(r=t->text+first,lp=t->loc+(first-t->top);r!=er;r++,lp++){
        if(lp->y+t->hgt>t->r.max.y){
            fprint(2, "chr %C, index %ld of %d, loc %d %d, off bottom\n",
                *r, lp-t->loc, t->bot-t->top, lp->x, lp->y);
            return;
        }
        switch(*r){
        case '\n':
            ur=*lp;
            break;
        case '\t':
            ur=*lp;
            if(lp[1].y!=lp[0].y)
                ul=Pt(t->r.min.x, lp[1].y);
            else
                ul=*lp;
            pl_clr(t->b, Rpt(ul, Pt(lp[1].x, ul.y+t->hgt)));
            break;
        default:
            buf[runetochar(buf, r)]='\0';
    /***/  pl_clr(t->b, Rpt(*lp, addpt(*lp, stringsize(t->font, buf))));
            ur=string(t->b, *lp, display->black, ZP, t->font, buf);
            break;
        }
        if(lp[1].y!=lp[0].y)
    /***/  pl_clr(t->b, Rpt(ur, Pt(t->r.max.x, ur.y+t->hgt)));
    }
}
```

Uses pl_clr() 105c.

⟨*function* tw_hilitep 143a⟩≡ (148d)
```
/*
 * Hilight the characters with tops between ul and ur
 */
void tw_hilitep(Textwin *t, Point ul, Point ur){
    Point swap;
    int y;
    if(tw_before(t, ur, ul)){ swap=ul; ul=ur; ur=swap;}
    y=ul.y+t->hgt;
    if(y>t->r.max.y) y=t->r.max.y;
    if(ul.y==ur.y)
        pl_highlight(t->b, Rpt(ul, Pt(ur.x, y)));
    else{
        pl_highlight(t->b, Rpt(ul, Pt(t->r.max.x, y)));
        ul=Pt(t->r.min.x, y);
        pl_highlight(t->b, Rpt(ul, Pt(t->r.max.x, ur.y)));
        ul=Pt(t->r.min.x, ur.y);
        y=ur.y+t->hgt;
        if(y>t->r.max.y) y=t->r.max.y;
        pl_highlight(t->b, Rpt(ul, Pt(ur.x, y)));
    }
}
```
Uses pl_highlight() 74b and tw_before() 140b.

⟨*function* twhilite 143b⟩≡ (148d)
```
/*
 * Hilite/unhilite the given range of characters
 */
void twhilite(Textwin *t, int sel0, int sel1, int on){
    Point ul, ur;
    int swap, y;
    if(sel1<sel0){ swap=sel0; sel0=sel1; sel1=swap; }
    if(sel1<t->top || t->bot<sel0) return;
    if(sel0<t->top) sel0=t->top;
    if(sel1>t->bot) sel1=t->bot;
    if(!on){
        if(sel1==sel0){
            ul=t->loc[sel0-t->top];
            y=ul.y+t->hgt;
            if(y>t->r.max.y) y=t->r.max.y;
            pl_clr(t->b, Rpt(ul, Pt(ul.x+1, y)));
        }else
            tw_draw(t, sel0, sel1);
        return;
    }
    ul=t->loc[sel0-t->top];
    if(sel1==sel0)
        ur=addpt(ul, Pt(1, 0));
    else
        ur=t->loc[sel1-t->top];
    tw_hilitep(t, ul, ur);
}
```
Uses pl_clr() 105c, tw_draw() 142, and tw_hilitep() 143a.

⟨*function* twselect 143c⟩≡ (148d)
```
/*
 * Set t->sel[01] from mouse input.
 * Also hilites the selection.
```

```
 * Caller should unhilite the previous
 * selection before calling this.
 */
void twselect(Textwin *t, Mouse *m){
    int sel0, sel1, newsel;
    Point p0, p1, newp;
    sel0=sel1=twpt2rune(t, m->xy);
    p0=tw_rune2pt(t, sel0);
    p1=addpt(p0, Pt(1, 0));
    twhilite(t, sel0, sel1, 1);
    for(;;){
        flushimage(display, 1);
        *m=emouse();
        if((m->buttons&7)!=1) break;
        newsel=twpt2rune(t, m->xy);
        newp=tw_rune2pt(t, newsel);
        if(eqpt(newp, p0)) newp=addpt(newp, Pt(1, 0));
        if(!eqpt(newp, p1)){
            if((sel0<=sel1 && sel1<newsel) || (newsel<sel1 && sel1<sel0))
                tw_hilitep(t, p1, newp);
            else if((sel0<=newsel && newsel<sel1) || (sel1<newsel && newsel<=sel0)){
                twhilite(t, sel1, newsel, 0);
                if(newsel==sel0)
                    tw_hilitep(t, p0, newp);
            }else if((newsel<sel0 && sel0<=sel1) || (sel1<sel0 && sel0<=newsel)){
                twhilite(t, sel0, sel1, 0);
                tw_hilitep(t, p0, newp);
            }
            sel1=newsel;
            p1=newp;
        }
    }
    if(sel0<=sel1){
        t->sel0=sel0;
        t->sel1=sel1;
    }
    else{
        t->sel0=sel1;
        t->sel1=sel0;
    }
}
```

Uses `tw_hilitep()` 143a, `tw_rune2pt()` 141a, `twhilite()` 143b, and `twpt2rune()` 140c.

⟨*function* `tw_clrend` 144a⟩≡ (148d)

```
/*
 * Clear the area following the last displayed character
 */
void tw_clrend(Textwin *t){
    Point ul;
    int y;
    ul=t->loc[t->bot-t->top];
    y=ul.y+t->hgt;
    if(y>t->r.max.y) y=t->r.max.y;
    pl_clr(t->b, Rpt(ul, Pt(t->r.max.x, y)));
    ul=Pt(t->r.min.x, y);
    pl_clr(t->b, Rpt(ul, t->r.max));
}
```

Uses `pl_clr()` 105c.

⟨*function* `tw_moverect` 144b⟩≡ (148d)

```
/*
 * Move part of a line of text, truncating the source or padding
 * the destination on the right if necessary.
 */
void tw_moverect(Textwin *t, Point uld, Point urd, Point uls, Point urs){
    int sw, dw, d;
    if(urs.y!=uls.y) urs=Pt(t->r.max.x, uls.y);
    if(urd.y!=uld.y) urd=Pt(t->r.max.x, uld.y);
    sw=uls.x-urs.x;
    dw=uld.x-urd.x;
    if(dw>sw){
        d=dw-sw;
        pl_clr(t->b, Rect(urd.x-d, urd.y, urd.x, urd.y+t->hgt));
        dw=sw;
    }
    pl_cpy(t->b, uld, Rpt(uls, Pt(uls.x+dw, uls.y+t->hgt)));
}
```

Uses `pl_clr()` 105c and `pl_cpy()` 77c.

⟨*function* `tw_moveup` 145a⟩≡ (148d)

```
/*
 * Move a block of characters up or to the left:
 * Identify contiguous runs of characters whose width doesn't change, and
 * move them in one bitblt per run.
 * If we get to a point where source and destination are x-aligned,
 * they will remain x-aligned for the rest of the block.
 * Then, if they are y-aligned, they're already in the right place.
 * Otherwise, we can move them in three bitblts; one if all the
 * remaining characters are on one line.
 */
void tw_moveup(Textwin *t, Point *dp, Point *sp, Point *esp){
    Point uld, uls;   /* upper left of destination/source */
    int y;
    while(sp!=esp && sp->x!=dp->x){
        uld=*dp;
        uls=*sp;
        while(sp!=esp && sp->y==uls.y && dp->y==uld.y && sp->x-uls.x==dp->x-uld.x){
            sp++;
            dp++;
        }
        tw_moverect(t, uld, *dp, uls, *sp);
    }
    if(sp==esp || esp->y==dp->y) return;
    if(esp->y==sp->y){ /* one line only */
        pl_cpy(t->b, *dp, Rpt(*sp, Pt(esp->x, sp->y+t->hgt)));
        return;
    }
    y=sp->y+t->hgt;
    pl_cpy(t->b, *dp, Rpt(*sp, Pt(t->r.max.x, y)));
    pl_cpy(t->b, Pt(t->r.min.x, dp->y+t->hgt),
        Rect(t->r.min.x, y, t->r.max.x, esp->y));
    y=dp->y+esp->y-sp->y;
    pl_cpy(t->b, Pt(t->r.min.x, y),
        Rect(t->r.min.x, esp->y, esp->x, esp->y+t->hgt));
}
```

Uses `pl_cpy()` 77c and `tw_moverect()` 144b.

⟨*function* `tw_movedn` 145b⟩≡ (148d)

```
/*
 * Same as above, but moving down and in reverse order, so as not to overwrite stuff
```

```
 * not moved yet.
 */
void tw_movedn(Textwin *t, Point *dp, Point *bsp, Point *esp){
    Point *sp, urs, urd;
    int dy;
    dp+=esp-bsp;
    sp=esp;
    dy=dp->y-sp->y;
    while(sp!=bsp && dp[-1].x==sp[-1].x){
        --dp;
        --sp;
    }
    if(dy!=0){
        if(sp->y==esp->y)
            pl_cpy(t->b, *dp, Rect(sp->x, sp->y, esp->x, esp->y+t->hgt));
        else{
            pl_cpy(t->b, Pt(t->r.min.x, sp->x+dy),
                Rect(t->r.min.x, sp->y, esp->x, esp->y+t->hgt));
            pl_cpy(t->b, Pt(t->r.min.x, dp->y+t->hgt),
                Rect(t->r.min.x, sp->y+t->hgt, t->r.max.x, esp->y));
            pl_cpy(t->b, *dp,
                Rect(sp->x, sp->y, t->r.max.x, sp->y+t->hgt));
        }
    }
    while(sp!=bsp){
        urd=*dp;
        urs=*sp;
        while(sp!=bsp && sp[-1].y==sp[0].y && dp[-1].y==dp[0].y
           && sp[-1].x-sp[0].x==dp[-1].x-dp[0].x){
            --sp;
            --dp;
        }
        tw_moverect(t, *dp, urd, *sp, urs);
    }
}
```

Uses `pl_cpy()` 77c and `tw_moverect()` 144b.

⟨*function* `tw_relocate` 146a⟩≡ (148d)

```
/*
 * Move the given range of characters, already drawn on
 * the given textwin, to the given location.
 * Start and end must both index characters that are initially on-screen.
 */
void tw_relocate(Textwin *t, int first, int last, Point dst){
    Point *srcloc;
    int nbyte;
    if(first<t->top || last<first || t->bot<last) return;
    nbyte=(last-first+1)*sizeof(Point);
    srcloc=pl_emalloc(nbyte);
    memmove(srcloc, &t->loc[first-t->top], nbyte);
    tw_setloc(t, first, last, dst);
    if(tw_before(t, dst, srcloc[0]))
        tw_moveup(t, t->loc+first-t->top, srcloc, srcloc+(last-first));
    else
        tw_movedn(t, t->loc+first-t->top, srcloc, srcloc+(last-first));
}
```

Uses `pl_emalloc()` 120a, `tw_before()` 140b, `tw_movedn()` 145b, `tw_moveup()` 145a, and `tw_setloc()` 141c.

⟨*function* `twreplace` 146b⟩≡ (148d)

```
/*
```

```
 * Replace the runes with indices from r0 to r1-1 with the text
 * pointed to by text, and with length ntext.
 * Open up a hole in t->text, t->loc.
 * Insert new text, calculate their locs (save the extra loc that's overwritten first)
 * (swap saved & overwritten locs)
 * move tail.
 * calc locs and draw new text after tail, if necessary.
 * draw new text, if necessary
 */
void twreplace(Textwin *t, int r0, int r1, Rune *ins, int nins){
    int olen, nlen, tlen, dtop;
    olen=t->etext-t->text;
    nlen=olen+nins-(r1-r0);
    tlen=t->eslack-t->text;
    if(nlen>tlen){
        tlen=nlen+SLACK;
        t->text=pl_erealloc(t->text, tlen*sizeof(Rune));
        t->eslack=t->text+tlen;
    }
    if(olen!=nlen)
        memmove(t->text+r0+nins, t->text+r1, (olen-r1)*sizeof(Rune));
    if(nins!=0) /* ins can be 0 if nins==0 */
        memmove(t->text+r0, ins, nins*sizeof(Rune));
    t->etext=t->text+nlen;
    if(r0>t->bot)  /* insertion is completely below visible text */
        return;
    if(r1<t->top){  /* insertion is completely above visible text */
        dtop=nlen-olen;
        t->top+=dtop;
        t->bot+=dtop;
        return;
    }
    if(1 || t->bot<=r0+nins){ /* no useful text on screen below r0 */
        if(r0<=t->top) /* no useful text above, either */
            t->top=r0;
        t->bot=tw_setloc(t, r0, nlen, t->loc[r0-t->top]);
        tw_draw(t, r0, t->bot);
        tw_clrend(t);
        return;
    }
    /*
     * code for case where there is useful text below is missing (see '1 ||' above)
     */
}
```

Uses SLACK-9 49a, pl_erealloc() 120b, tw_clrend() 144a, tw_draw() 142, and tw_setloc() 141c.

⟨*function* twscroll 147a⟩≡ (148d)

```
/*
 * This works but is stupid.
 */
void twscroll(Textwin *t, int top){
    while(top!=0 && t->text[top-1]!='\n') --top;
    t->top=top;
    t->bot=tw_setloc(t, top, t->etext-t->text, t->r.min);
    tw_draw(t, t->top, t->bot);
    tw_clrend(t);
}
```

Uses tw_clrend() 144a, tw_draw() 142, and tw_setloc() 141c.

⟨*function* twreshape 147b⟩≡ (148d)

```
void twreshape(Textwin *t, Rectangle r){
    t->r=r;
    t->bot=tw_setloc(t, t->top, t->etext-t->text, t->r.min);
    tw_draw(t, t->top, t->bot);
    tw_clrend(t);
}
```

Uses tw_clrend() 144a, tw_draw() 142, and tw_setloc() 141c.

⟨*function* twnew 148a⟩≡ (148d)

```
Textwin *twnew(Image *b, Font *f, Rune *text, int ntext){
    Textwin *t;
    t=pl_emalloc(sizeof(Textwin));
    t->text=pl_emalloc((ntext+SLACK)*sizeof(Rune));
    t->loc=pl_emalloc(SLACK*sizeof(Point));
    t->eloc=t->loc+SLACK;
    t->etext=t->text+ntext;
    t->eslack=t->etext+SLACK;
    if(ntext) memmove(t->text, text, ntext*sizeof(Rune));
    t->top=0;
    t->bot=0;
    t->sel0=0;
    t->sel1=0;
    t->b=b;
    t->font=f;
    t->hgt=f->height;
    t->mintab=stringwidth(f, "0");
    t->tabstop=8*t->mintab;
    return t;
}
```

Uses SLACK-9 49a and pl_emalloc() 120a.

⟨*function* twfree 148b⟩≡ (148d)

```
void twfree(Textwin *t){
    free(t->loc);
    free(t->text);
    free(t);
}
```

⟨*function* twmove 148c⟩≡ (148d)

```
/*
 * Correct the character locations in a textwin after the panel is moved.
 * This horrid hack would not be necessary if loc values were relative
 * to the panel, rather than absolute.
 */
void twmove(Textwin *t, Point d){
    Point *lp;
    t->r = rectaddpt(t->r, d);
    for(lp=t->loc; lp<t->eloc; lp++)
        *lp = addpt(*lp, d);
}
```

⟨lib_gui/libpanel/textwin.c 148d⟩≡

```
/*
 * Text windows
 * void twhilite(Textwin *t, int sel0, int sel1, int on)
 *  hilite (on=1) or unhilite (on=0) a range of characters
 * void twselect(Textwin *t, Mouse *m)
 *  set t->sel0, t->sel1 from mouse input.
 *  Also hilites selection.
 *  Caller should first unhilite previous selection.
```

```
 * void twreplace(Textwin *t, int r0, int r1, Rune *ins, int nins)
 *  Replace the given range of characters with the given insertion.
 *  Caller should unhilite selection while this is called.
 * void twscroll(Textwin *t, int top)
 *  Character with index top moves to the top line of the screen.
 * int twpt2rune(Textwin *t, Point p)
 *  which character is displayed at point p?
 * void twreshape(Textwin *t, Rectangle r)
 *  save r and redraw the text
 * Textwin *twnew(Bitmap *b, Font *f, Rune *text, int ntext)
 *  create a new text window
 * void twfree(Textwin *t)
 *  get rid of a surplus Textwin
 */
```

⟨libpanel *includes* 14⟩

⟨*constant* SLACK*(*(lib_gui/libpanel/textwin.c)*)* 49a⟩

⟨*function* tw_before 140b⟩
⟨*function* twpt2rune 140c⟩
⟨*function* tw_rune2pt 141a⟩
⟨*function* tw_storeloc 141b⟩
⟨*function* tw_setloc 141c⟩
⟨*function* tw_draw 142⟩
⟨*function* tw_hilitep 143a⟩
⟨*function* twhilite 143b⟩
⟨*function* twselect 143c⟩
⟨*function* tw_clrend 144a⟩
⟨*function* tw_moverect 144b⟩
⟨*function* tw_moveup 145a⟩
⟨*function* tw_movedn 145b⟩
⟨*function* tw_relocate 146a⟩
⟨*function* twreplace 146b⟩
⟨*function* twscroll 147a⟩
⟨*function* twreshape 147b⟩
⟨*function* twnew 148a⟩
⟨*function* twfree 148b⟩
⟨*function* twmove 148c⟩

D.2.26 lib_gui/libpanel/utf.c

⟨*function* pl_idchar 149a⟩≡ (150c)

```
/*
 * This is the same definition that 8½ uses
 */
int pl_idchar(int c){
    if(c<=' '
    || 0x7F<=c && c<=0xA0
    || utfrune("!\"#$%&'()*+,-./:;<=>?@‘[\\]^{|}~", c))
        return 0;
    return 1;
}
```

⟨*function* pl_rune1st 149b⟩≡ (150c)

```
int pl_rune1st(int c){
    return (c&0xc0)!=0x80;
}
```

⟨*function* `pl_nextrune` 150a⟩≡ (150c)

```
char *pl_nextrune(char *s){
    do s++; while(!pl_rune1st(*s));
    return s;
}
```

Uses `pl_rune1st()` 149b.

⟨*function* `pl_runewidth` 150b⟩≡ (150c)

```
int pl_runewidth(Font *f, char *s){
    char r[4], *t;
    t=r;
    do *t++=*s++; while(!pl_rune1st(*s));
    *t='\0';
    return stringwidth(f, r);
}
```

Uses `pl_rune1st()` 149b.

⟨`lib_gui/libpanel/utf.c` 150c⟩≡

⟨`libpanel` *includes* 14⟩

⟨*function* `pl_idchar` 149a⟩
⟨*function* `pl_rune1st` 149b⟩
⟨*function* `pl_nextrune` 150a⟩
⟨*function* `pl_runewidth` 150b⟩

Glossary

LOC = Lines Of Code

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

BORD-8: 105b, 134a, 134d
Button: 53a, 129a
BUTTON-10: 53b, 53g, 54b, 55a, 57g
Button.btype: 53a
Button.buttons: 53e
Button.check: 53f
Button.hit: 53a
Button.icon: 53a
Button.index: 82a
Button.menuhit: 82b
Button.pl_buttonhit: 53a
Button (typedef): 129a
BWID-14: 30e, 49d, 49e, 50a, 52d
Canvas: 64a, 131b
Canvas.draw: 64a
Canvas.hit: 64a
Canvas (typedef): 131b
CHECK-11: 53c, 53g, 56d, 56e, 57e
CKBORDER-21: 56f, 57c
CKINSET-20: 56f, 57a, 58a, 58g
CKSIZE-17: 58a, 58b
CKSPACE-18: 58a, 58g, 59a
CKWID-19: 56f, 57b, 58a, 58g
done(): 13a
DOWN: 30e, 45c, 50a, 50e, 52d, 55b, 62a, 74f, 79c, 84d, 85c, 86e, 88d, 102a
DOWN1: 22a, 30e, 50a, 52d, 84d
DOWN2: 22a, 30e, 50a, 52d, 84d
DOWN3: 30e, 50a, 52d, 84d
Edit: 95a, 130
Edit.hit: 95a
Edit.minsize: 95a
Edit.ntext: 95a
Edit.sel0: 95a
Edit.sel1: 95a
Edit.t: 95a
Edit.text: 95a
Edit (typedef): 130
Entry: 47g, 129b

Entry.eent: 48a
Entry.entp: 48a
Entry.entry: 47g
Entry.hit: 47g
Entry.minsize: 47g
Entry (typedef): 129b
eresized(): 13c
FRAME: 67a, 67c, 67g, 68a, 68b, 68c
FWID-15: 67c, 67d, 68a, 68c
HITME: 125e
HORIZ: 60c, 61a, 61b, 62b, 63e, 78c, 79c, 81b, 125c, 137b
INVIS: 87a, 87c
Label: 44a, 128c
Label.icon: 44a
Label.placement: 44a
Label (typedef): 128c
LEAD-7: 133c, 134d
LEAF: 21c, 44d, 48c, 53g, 60c, 64c, 72c, 78a, 88b, 92c, 95c, 101b
List: 71g, 132a
List.buttons: 72a
List.gen: 71g
List.hit: 71g
List.len: 71g
List.listr: 73c
List.lo: 71g
List.minsize: 71g
List.sel: 71g
List (typedef): 132a
main-3(): 12
Message: 92a, 132b
Message.minsize: 92a
Message.text: 92a
Message (typedef): 132b
PASSIVE: 30f, 45b, 45c, 45d, 47d, 47e, 76b, 77b, 93a, 94c
plbutton(): 12, 54a, 82d
plcanvas(): 64b
plcheckbutton(): 56c
pldraw(): 13c, 28a, 50e, 51a, 55b, 59c, 62a, 79c, 81b, 86a, 88d, 105b, 110a, 110c, 139d
pledit(): 95b
pleget(): 99d, 110d
plegetsel(): 99b, 110d
plelen(): 97, 99c
plemove(): 86d, 100c
plentry(): 48b
plentryval(): 52f
plepaste(): 96c, 97, 100b, 110e
plescroll(): 99a
plesel(): 100a
plframe(): 12, 66a

plfree(): 19c, 21a, 83a
plgetpostextview(): 139c
plgetscroll(): 137a
plgetsnarf(): 108d, 109b
plgrabkb(): 33k, 50e, 96c
plgroup(): 68d, 82c, 91a
plinit(): 12, 26a
plinitbutton(): 54a, 54b
plinitcanvas(): 64b, 64c
plinitcheckbutton(): 56c, 56d
plinitedit(): 95b, 95c
plinitentry(): 48b, 48c
plinitframe(): 66a, 66b
plinitgroup(): 68d, 68e
plinitlabel(): 44c, 44d
plinitlist(): 72b, 72c
plinitmenu(): 82c, 82d
plinitmessage(): 92b, 92c
plinitpopup(): 84a, 84b
plinitpulldown(): 88a, 88b
plinitradiobutton(): 58d, 58e
plinitscrollbar(): 77f, 78a
plinitslider(): 60b, 60c
plinittextview(): 101a, 101b
plkeyboard(): 34a
pllabel(): 12, 44c
plldepth-22: 26b, 26d, 74c
pllist(): 72b
plmenu(): 82c
plmenubar(): 91a
plmessage(): 92b
plmouse(): 12, 31a, 84d, 88d, 138c
plmove(): 86a, 86b, 86b, 105b, 134d, 136a
plpack(): 13c, 35, 86a, 88d, 134d
plpaste(): 96c, 109b, 110c
plplacelabel(): 44c, 45a
plpopup(): 84a
plpulldown(): 88a, 91a
plputsnarf(): 108c, 109a
plradiobutton(): 58d
plrtbitmap(): 104d
plrtfree(): 104e
plrtpanel(): 104b
plrtseltext(): 102a, 106c
plrtsnarftext(): 136b, 139b
plrtstr(): 104c
plscroll(): 71a
plscrollbar(): 77f
plsetbutton(): 58c

plsetpostextview(): 139d
plsetscroll(): 137b
plsetslider(): 63e
plslider(): 60b
plsnarf(): 52a, 96c, 97, 109a, 110c
pltabsize(): 134b
pltextview(): 101a
pl_black-26: 26c, 26d, 46b, 49d, 50a, 56f, 57d, 58g, 67c
pl_box(): 29c, 45b, 49c, 54d, 61a, 67a, 73d, 76b, 77b, 88c, 93a
pl_boxoutline(): 29b, 29c, 30a
pl_boxsize(): 30c, 47d, 52c, 56a, 63c, 67g, 75b, 80d, 90c, 94c, 98a, 103a
pl_buttonhit(): 54b, 54c
pl_check(): 56e, 56f
pl_childspacebutton(): 53g, 56b
pl_childspacecanvas(): 64c, 65d
pl_childspaceedit(): 95c, 98b
pl_childspaceentry(): 48c, 52e
pl_childspaceerror(): 41b, 118f
pl_childspaceframe(): 66b, 68b
pl_childspacegroup(): 68e, 69e
pl_childspacelabel(): 44d, 47f
pl_childspacelist(): 72c, 75c
pl_childspacemessage(): 92c, 94e
pl_childspacepopup(): 84b, 87e
pl_childspacepulldown(): 88b, 90d
pl_childspacescrollbar(): 78a, 80e
pl_childspaceslider(): 60c, 63d
pl_childspacetextview(): 101b, 103b
pl_ckwid(): 57g, 58a
pl_clr(): 105b, 105c, 142, 143b, 144a, 144b
pl_cpy(): 77b, 77c, 136a, 144b, 145a, 145b
pl_dark-25: 26c, 26d, 50a, 61b, 74b
pl_draw1(): 28a, 28b, 28c
pl_drawall(): 28b, 28c
pl_drawbutton(): 53g, 54d
pl_drawcanvas(): 64c, 64d
pl_drawedit(): 95c, 96b
pl_drawentry(): 48c, 49c
pl_drawerror(): 22c, 118b
pl_drawframe(): 66b, 67a
pl_drawgroup(): 68e, 69a
pl_drawicon(): 45b, 46b, 49c, 54d, 73e, 74a, 88c
pl_drawinit(): 26a, 26d
pl_drawlabel(): 44d, 45b
pl_drawlist(): 72c, 73d
pl_drawmessage(): 92c, 93a
pl_drawpopup(): 84b, 85e
pl_drawpulldown(): 88b, 88c
pl_drawscrollbar(): 78a, 79b

pl_drawslider(): 60c, 61a
pl_drawtextview(): 101b, 101c, 102a
pl_emalloc(): 19a, 21f, 104a, 120a, 146a, 148a
pl_erealloc(): 48e, 51d, 108d, 110a, 120b, 136b, 141b, 146b
pl_fill(): 74a, 74c
pl_foldsize(): 94c, 94d
pl_freeedit(): 95c, 96a
pl_freeentry(): 48c, 48d
pl_getshare(): 42b, 43, 43
pl_getsizebutton(): 53g, 56a
pl_getsizecanvas(): 64c, 65c
pl_getsizeedit(): 95c, 98a
pl_getsizeentry(): 48c, 52c
pl_getsizeerror(): 37a, 118e
pl_getsizeframe(): 66b, 67g
pl_getsizegroup(): 68e, 69d
pl_getsizelabel(): 44d, 47d
pl_getsizelist(): 72c, 75b
pl_getsizemessage(): 92c, 94c
pl_getsizepopup(): 84b, 87d
pl_getsizepulldown(): 88b, 90c
pl_getsizescrollbar(): 78a, 80d
pl_getsizeslider(): 60c, 63c
pl_getsizetextview(): 101b, 103a
pl_highlight(): 74a, 74b, 105b, 143a
pl_hilit-27: 26c, 26d, 74b
pl_hitbutton(): 53g, 55b, 59c
pl_hitcanvas(): 64c, 65a
pl_hitedit(): 95c, 96c
pl_hitentry(): 48c, 50e
pl_hiterror(): 22c, 118c
pl_hitframe(): 66b, 67e
pl_hitgroup(): 68e, 69b
pl_hitlabel(): 44d, 47b
pl_hitlist(): 72c, 74f
pl_hitmenu(): 82d, 83b
pl_hitmessage(): 92c, 94a
pl_hitpopup(): 84b, 84d
pl_hitpulldown(): 88b, 88d
pl_hitscrollbar(): 78a, 79c
pl_hitslider(): 60c, 62a
pl_hittextview(): 101b, 102a
pl_iconsize(): 46b, 46c, 47d, 56a, 90c
pl_idchar(): 149a
pl_initbtype(): 53g, 54b, 56d, 58e
pl_interior(): 30d, 62a, 68b, 74f, 76b, 79c, 81b, 102a, 138d, 139a
pl_invis(): 86a, 86e, 87a, 87a, 88d
pl_ipprint(): 116b, 116d, 116d
pl_iprint(): 116c, 116d

pl_light-24: 26c, 26d, 45d, 49d, 56f, 58g, 61b, 67c, 74c
pl_listsel(): 73f, 74a, 74f
pl_liststrings(): 73d, 73e, 76b, 77b
pl_max(): 37d, 37e, 38a
pl_newpanel(): 19a, 44c, 48b, 54a, 56c, 58d, 60b, 64b, 66a, 68d, 72b, 77f, 84a, 88a, 92b, 95b, 101a
pl_nextrune(): 93b, 94d, 150a
pl_outline(): 29b, 79b, 101c, 138d
pl_passon(): 102a, 138c
pl_pasteedit(): 95c, 110e
pl_pasteentry(): 109c, 110a
pl_prinormal(): 33f, 33g
pl_print(): 116b
pl_pripopup(): 33h, 84b
pl_priscrollbar(): 33i, 78b
pl_pritextview(): 101b, 139a
pl_ptinpanel(): 31a, 32a, 32a, 139a
pl_radio(): 58f, 58g
pl_relief(): 29a, 49d, 50a, 56f, 58g, 67c
pl_reposition(): 106a, 136a
pl_rtdraw(): 101c, 105b, 106a
pl_rtfmt(): 101c, 134d
pl_rthit(): 102a, 106b, 139a
pl_rtnew(): 104a, 104b, 104c, 104d
pl_rtredraw(): 106a, 138d
pl_rune1st(): 52a, 149b, 150a, 150b
pl_runewidth(): 93b, 94d, 150b
pl_scrolledit(): 95c, 98c
pl_scrollerror(): 71f, 119a
pl_scrolllist(): 76a, 76b
pl_scrolltextview(): 101b, 138d
pl_setrect(): 35, 38c, 41c, 41d
pl_setscrollbarerror(): 71f, 119b
pl_setscrollbarscrollbar(): 81a, 81b
pl_setscrpos(): 101c, 138b, 138d
pl_sizereq(): 35, 36e, 36e
pl_sizesibs(): 36e, 37b, 37b
pl_sliderupd(): 61a, 61b, 79b, 80b
pl_snarfedit(): 95c, 110d
pl_snarfentry(): 109c, 109d
pl_snarftextview(): 101b, 139b
pl_space(): 134c, 134d
pl_stuffbitmap(): 105b, 135, 135
pl_tabmin: 134b, 134c, 136c
pl_tabsize: 134b, 134c, 136c
pl_textmsg(): 93a, 93b
pl_typebutton(): 53g, 55c
pl_typecanvas(): 64c, 65b
pl_typeedit(): 95c, 97
pl_typeentry(): 48c, 51a

pl_typeerror(): 22c, 118d
pl_typeframe(): 66b, 67f
pl_typegroup(): 68e, 69c
pl_typelabel(): 44d, 47c
pl_typelist(): 72c, 75a
pl_typemessage(): 92c, 94b
pl_typepopup(): 84b, 85d
pl_typepulldown(): 88b, 90b
pl_typescrollbar(): 78a, 80c
pl_typeslider(): 60c, 63b
pl_typetextview(): 101b, 102b
pl_unexpected(): 118a, 118b, 118c, 118d, 118e, 118f, 119a, 119b
pl_white-23: 26c, 26d, 45d, 49d, 50a, 56f, 58g, 67c, 74c
Popup: 83c, 133a
Popup.pop: 83c
Popup.save: 83c
Popup (typedef): 133a
Pulldown: 133b, 133b
Pulldown (typedef): 133b
PWID-13: 30f, 45d, 46a, 47e
RADIO-12: 53d, 53g, 58e, 58f, 59c
REMOUSE: 32e, 33a, 102a
root: 12
s: 91a
SBWID-2: 78c, 78d
SCROLLABSX: 125c
Scrollbar: 77d, 137d
Scrollbar.buttons: 77e
Scrollbar.dir: 77d
Scrollbar.hi: 77d
Scrollbar.interior: 79a
Scrollbar.lo: 77d
Scrollbar.minsize: 77d
Scrollbar (typedef): 137d
SCROLLLEFT: 125c
SCROLLRIGHT: 125c
SCROLLUP: 22a
SLACK-1: 48e, 51d, 110a, 110b
SLACK-9: 49a, 141b, 146b, 148a
Slider: 59d, 131a
Slider.buttons: 60a
Slider.dir: 59d
Slider.hit: 59d
Slider.minsize: 59d
Slider.val: 59d
Slider (typedef): 131a
SPACE-16: 30a, 30b, 30e, 30f, 45d, 47e, 49d, 50a, 52d, 67c, 68a, 68c
Textview: 100d, 140a
Textview.buttons: 100d

Textview.hit: 100d
Textview.hitfirst: 100d
Textview.hitword: 100d
Textview.minsize: 100d
Textview.text: 100d
Textview.thgt: 100d
Textview.twid: 100d
Textview.yoffs: 100d
Textview (typedef): 140a
Textwin: 125e, 125e
Textwin.b: 125e
Textwin.bot: 125e
Textwin.eloc: 125e
Textwin.eslack: 125e
Textwin.etext: 125e
Textwin.font: 125e
Textwin.hgt: 125e
Textwin.loc: 125e
Textwin.r: 125e
Textwin.sel0: 125e
Textwin.sel1: 125e
Textwin.tabstop: 125e
Textwin.text: 125e
Textwin.top: 125e
Textwin (typedef): 125e
twfree(): 95c, 96a, 148b
twhilite(): 96b, 96c, 97, 98c, 100a, 100b, 143b, 143c
twmove(): 100c, 148c
twnew(): 96b, 148a
twpt2rune(): 97, 98c, 140c, 143c
twreplace(): 97, 100b, 146b
twreshape(): 96b, 147b
twscroll(): 97, 98c, 99a, 147a
twselect(): 96c, 143c
tw_before(): 140b, 140c, 143a, 146a
tw_clrend(): 144a, 146b, 147a, 147b
tw_draw(): 142, 143b, 146b, 147a, 147b
tw_hilitep(): 143a, 143b, 143c
tw_movedn(): 145b, 146a
tw_moverect(): 144b, 145a, 145b
tw_moveup(): 145a, 146a
tw_relocate(): 146a
tw_rune2pt(): 141a, 143c
tw_setloc(): 141c, 146a, 146b, 147a, 147b
tw_storeloc(): 141b, 141c
UP: 22a, 30e, 49b, 49d, 50e, 52d, 53g, 55b, 60c, 61a, 62a, 72c, 73d, 74f, 78a, 79c, 84b, 84d, 86e, 88b, 88d, 90a, 95c, 101b, 101c, 102a
VERT: 60c, 61a, 61b, 76b, 78c, 98c, 137b, 138d
__anon_enum_1: 22a

__anon_enum_3: 125c

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 9

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 9

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 34, 114

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 9, 28, 114

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 7, 9, 13, 114

[Ros88] David Rosenthal. A simple X11 client program -or- how hard can it really be to write "hello, world"? In *USENIX Winter Conference*, pages 229–242, 1988. cited page(s) 8