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

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Tom Duff

April 28, 2026

Contents

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

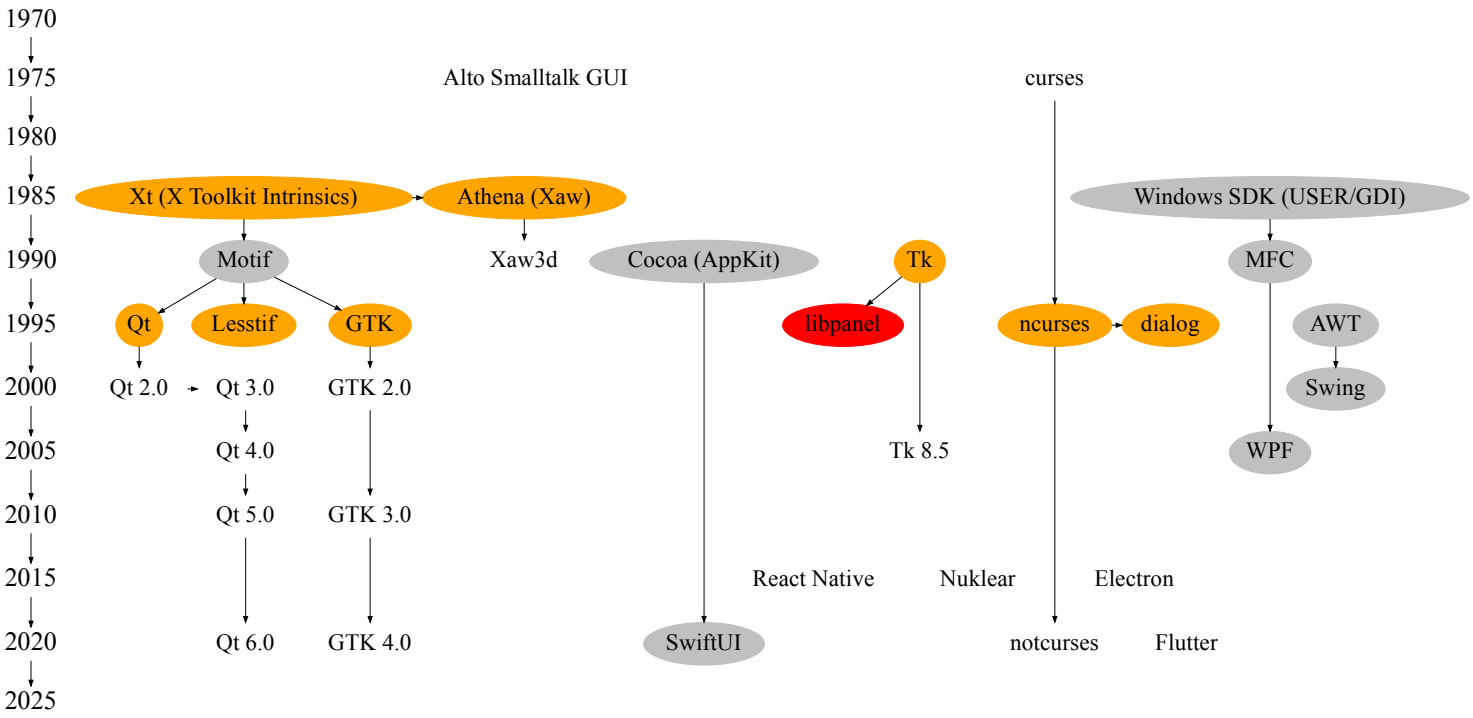


Figure 1.1: Widget libraries timeline

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`.

The single most useful companion document to this book is Tom Duff's "Panel: A Graphical User Interface Toolkit" paper, included in my Plan 9 repository at `lib_gui/docs/panel.pdf`. It describes `libpanel`'s design goals, the widget taxonomy, and the layout model, and is short enough to read in one sitting before diving into the source. The manual page `docs/man/1/panel`[1] documents the API. For the lower-level graphics and event APIs that `libpanel` sits on top of, the most relevant manual pages are `docs/man/2/draw`, `docs/man/2/window`, `docs/man/2/mouse`, and `docs/man/2/keyboard`.

[1] Not present in every Plan 9 distribution; `lib_gui/docs/panel.pdf` covers the same material.

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.

The same pattern recurs in every widget library, dressed differently each time: GTK's `GtkWidget` with GOBJECT virtual dispatch, Qt's `QWidget` with C++ virtual methods, and Cocoa's `NSView` with Objective-C message passing all play the role that `Panel` plays here. The whole lineage descends from Smalltalk-80 (1980), where every screen element was already a class with virtual methods; `libpanel` is essentially that design translated to a language without classes. The Core Data Structures chapter has a full cross-system table.

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.

The choice of whether to expose or hide the event loop is one of the big differentiators between GUI toolkits. Most modern toolkits *hide* it—GTK's `gtk_main`, Qt's `QApplication::exec`, and Cocoa's `NSApp run` all run the loop for you and let you register callbacks. A minority *expose* it, `libpanel` among them, because they want either small code size or explicit frame-by-frame control (the same reason game engines ship with their own frame loop). This is a mild reversal of history: the original Mac Toolbox `WaitNextEvent` (1984) and X Window's `XNextEvent` both handed the loop back to the programmer, and framework-driven toolkits that hide it came later, starting with Motif and Tk in the late 1980s.

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.

Every widget toolkit needs a layout engine, and most use the same two-pass bottom-up-then-top-down shape: GTK's `size_request/size_allocate`, Qt's `sizeHint/setGeometry`, and Android's `onMeasure/onLayout` all have it. Where toolkits differ is the *placement model*—`libpanel` and Tk use *packing* (children consume their parent's space from one edge), CSS gives you flow, flex, and grid, and SwiftUI builds the tree *declaratively*. The packing model was John Ousterhout's invention for Tk in 1988, and Rob Pike carried it almost unchanged into `libpanel`. The Layout chapter has a fuller cross-toolkit comparison.

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.

Other toolkits solve the same problem differently. GTK wraps content in a `GtkScrolledWindow` that handles both scrollbars automatically; Cocoa has `NSScrollView` (inherited from NeXTSTEP, and ultimately from Smalltalk-80's 1980 `ScrolledWindow` idiom); Qt has `QScrollArea`. All of these are *container-based*: a parent widget holds the real content and manages its own scrollbars, hiding the two-way protocol inside the container. `libpanel`'s choice is the opposite—the scroll bar is its own first-class widget that the application assembles. Modern touch-first interfaces (iOS, Android, web) hide the scrollbar entirely except during active scrolling, a third design point that works only because touch gestures replace the scrollbar as the primary navigation tool.

2.1.5 Retained mode versus immediate mode

One last distinction is worth naming explicitly, because it cuts across everything in the previous four subsections. Every widget library in the world sits in one of two camps depending on *what exists in memory between frames*. A *retained-mode* library—the design used by `libpanel`, Tk, GTK, Qt, Cocoa, Swing, WPF, HTML's DOM, and essentially every traditional operating-system toolkit—has the application build a persistent tree of widget objects once and then incrementally mutate it in response to events. The library owns the tree, redraws only what has changed, and dispatches events into it. The widgets *exist* as long-lived C structs (or Java objects, or Objective-C instances, or DOM nodes); clicking a button that is no longer on screen is still a call to a real method on a real object.

An *immediate-mode* library, by contrast, keeps *no* widget objects between frames. The application redescribes the entire UI from scratch on every frame inside its main loop, roughly like this:

```
while running:
    begin_frame()
    if button("OK"):        // draws AND hit-tests in one call
        save_file()
    label("Hello, " + name)
    end_frame()
```

Each call like `button("OK")` both draws the button and returns whether it was clicked this frame, so there is no persistent `Button` object, no tree to mutate, no callback to register, and no invalidation protocol—the state of the UI is always equal to whatever the surrounding code chose to emit. This style was popularized by Casey Muratori's *Dear ImGui* in 2014 and is now standard in game-engine debug and editor UIs, in the Nuklear C library, and in most 3D-accelerated toolkits where redrawing is cheap.

The two styles trade different properties. Retained mode is *economical* (the framework can be told to redraw only what changed), *data-driven* (the tree is the model and can be queried, styled, observed, and walked by a screen reader), and the default choice whenever the widget set needs to persist beyond a single frame. It pays for these properties with an *explicit update protocol*: the programmer must call a refresh after mutating the tree (as `libpanel` does) or the library must watch the tree with signals, observers, or reactive bindings (as GTK, Qt, and WPF do). Immediate mode is *stateless* (no tree lifetime to manage), *declarative* (the UI is whatever the code just said it was), and dramatically simpler to implement—a complete immediate-mode library can fit in a few hundred lines. The cost is that the CPU redraws everything every frame whether or not anything changed, and accessibility is harder because there are no persistent widget objects to hand to assistive software.

`libpanel` is squarely retained-mode, and most of what makes the following chapters interesting is exactly the machinery that retained-mode design requires: a `Panel` structure to hold widget state between frames, a tree-building API to assemble it, a two-pass layout algorithm to resize it when its container changes, a dispatch mechanism to deliver events into it, and cross-linked pointers for scrolling. None of these would exist in an immediate-mode library, and naming the contrast now is what explains why `libpanel`'s design looks the way it does.

The pendulum has swung one more time since. React (Facebook, 2013) and SwiftUI (Apple, 2019) introduced a *third* approach: the programmer writes code that looks immediate-mode ("describe the entire UI every render"), but the framework maintains a retained-mode tree underneath (the *virtual* DOM in React, the *view tree* in SwiftUI) and *diffs* the new description against the old to compute the minimal set of changes. Flutter and Jetpack Compose do the same. The result is the immediate-mode programming experience—no manual tree mutation, no observer callbacks, no invalidation—with retained-mode performance. It is, in a sense, both sides winning at once, though at the cost of a reconciliation engine that neither the retained nor the immediate camp needed.

2.1.6 A widget library as a microcosm of the OS

A final observation that may help readers coming from KERNEL book [Pad14]. A widget library solves, in miniature, the same problems an operating system solves. It *multiplexes* a scarce resource (screen space, like the kernel multiplexes CPU time). It *isolates* clients (each widget has its own rectangle, like each process has its own address space). It *dispatches events* to the right handler (mouse clicks routed to widgets, like interrupts routed to drivers). It *schedules layout* in passes (bottom-up size requests then top-down rectangle assignment, like a two-pass scheduler). And it manages *lifecycles* (create, resize, destroy, like process creation and termination). The patterns are not accidental: both the kernel and the widget library are *resource managers* that must serve multiple untrusting clients fairly, and the design solutions converge because the underlying problem is the same.

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 14a⟩≡

```
// 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 14b⟩

⟨function eresized 15b⟩

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

    err = initdraw(nil, nil, "hellopanel");
    ⟨main() sanity check err 15a⟩
    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()` 59d, `plframe()` 72a, `plinit()` 28a, `pllabel()` 49c, and `plmouse()` 35a.

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 14b⟩≡ (14a)

```
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` 15a⟩≡ (14a)
```
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` 15b⟩≡ (14a)
```
void eresized(bool new){
    ⟨eresized() if new get a new window 15c⟩
    plpack(root, view->r);
    pldraw(root, view);
}
```
Uses `pldraw()` 30a and `plpack()` 40.

⟨`eresized()` *if* `new` *get a new window* 15c⟩≡ (15b)
```
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.

Every source file includes the same set of headers:

⟨`libpanel` *includes* 15d⟩≡ (158c 156d 148a 146a 145 144c 141 140 139 138 137 136 135)
```
#include <u.h>
#include <libc.h>
#include <draw.h>
#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.

Function	Ch.	File	Entities
core data structures	3	`panel.h`	`Panel`[19] `Icon`[24a]
internal definitions	3	`pldefs.h`	layout flags, method prototypes
initialization	4	`init.c`	`plinit()`[28a]
drawing	5	`draw.c`	`pldraw()`[30a] `pl\_draw()X` `pl\_relief()X`
mouse/keyboard dispatch	6	`event.c`	`plmouse()`[35a] `plkeyboard()`[38c] `plgrabkb()`[38b]
layout algorithm	7	`pack.c`	`plpack()`[40] `pl\_sizereq()X` `pl\_setrect()X`
label widget	8	`label.c`	`Label`[49a] `pllabel()`[49c]
text entry widget	8	`entry.c`	`Entry`[52g] `plentry()`[53b]
button widget	8	`button.c`	`Button`[58d] `plbutton()`[59d]
slider widget	8	`slider.c`	`Slider`[65a] `plslider()`[65c]
canvas widget	8	`canvas.c`	`Canvas`[69c] `plcanvas()`[69d]
frame container	9	`frame.c`	`plframe()`[72a]
group container	9	`group.c`	`plgroup()`[74a]
scrolling glue	10	`scroll.c`	`plscroll()`[77a] `plsetscroll()`[145b]
scroll bar widget	10	`scrollbar.c`	`Scrollbar`[83d] `plscrollbar()`[83f]
scrollable list	10	`list.c`	`List`[77g] `pllist()`[78b]
popup menu	11	`popup.c`	`Popup`[90c] `plpopup()`[91a]
pull-down menu	11	`pulldown.c`	`Pulldown`[141b] `plpulldown()`[95c] `plmenubar()`[98d]
message widget	12	`message.c`	`Message`[100a] `plmessage()`[100b]
edit widget	12	`edit.c`	`Edit`[103a] `pledit()`[103b]
text view widget	12	`textview.c`	`Textview`[108d] `pltextview()`[109a]
text window widget	12	`textwin.c`	
rich text	12	`rtext.c`	`pl\_rtfmt()X` `pl\_rtdraw()X`
copy/paste	13	`snarf.c`	`plsnarf()`[117e] `plpaste()`[118a]
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.

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`[19] widgets using constructors like `plframe()`[72a], `pllabel()`[49c], and `plbutton()`[59d]. It then calls `plpack()`[40] to compute the layout and `pldraw()`[30a] to render the tree. During the event loop, mouse events are dispatched into the tree via `plmouse()`[35a], 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()`[15b] (either at startup or when the window is resized): `plpack()` runs the two-pass layout algorithm—`pl_sizereq()`[42a] walks the tree bottom-up to collect size requests, then `pl_setrect()`[44a] walks top-down to assign rectangles.

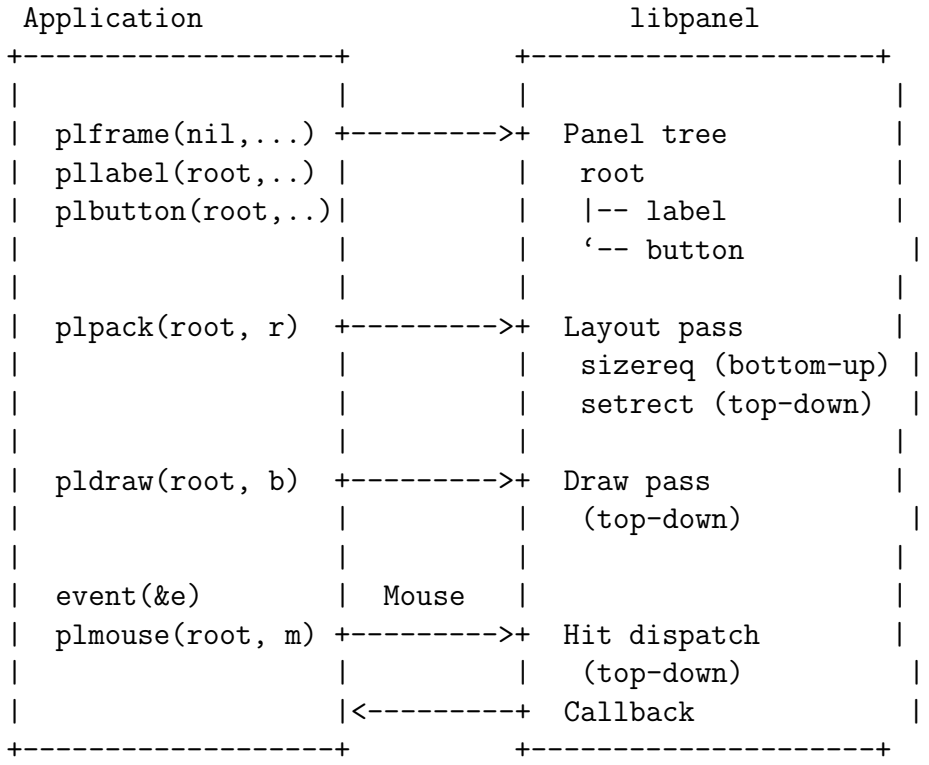


Figure 2.1: Data flow diagram of `libpanel`.

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()`[50b], `pl_drawbutton()`[60b]). 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()`[60d]), which may invoke the application's callback.

`libpanel` uses a consistent naming convention that reveals its architecture. Public functions use the `pl` prefix: `plinit()`[28a], `pldraw()`, `plpack()`, `plmouse()`, `plfree()`[20c]. Widget constructors also use `pl`: `pllabel()`, `plbutton()`, `plframe()`, `pllist()`[78b], 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 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.

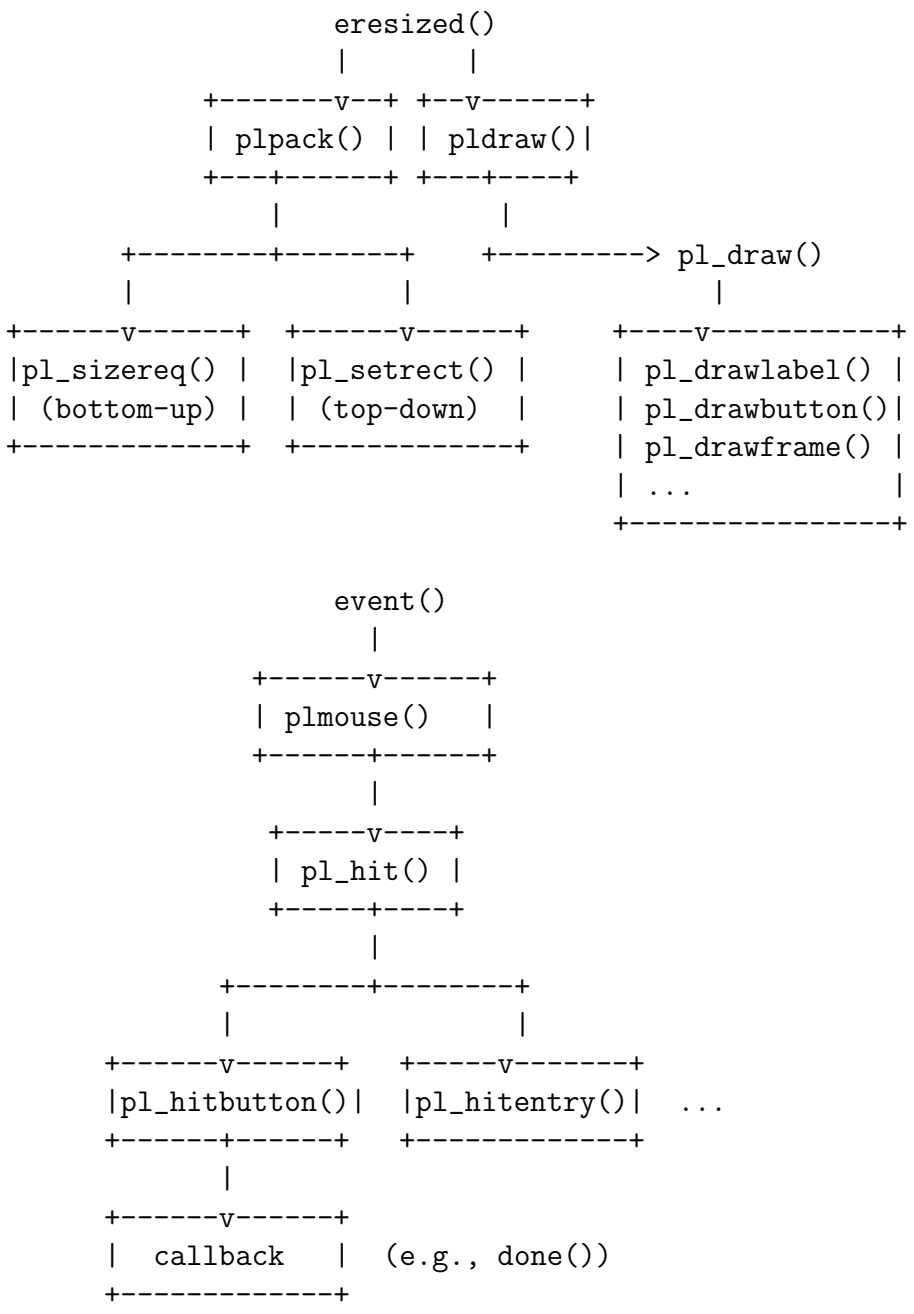


Figure 2.2: Control flow diagram of `libpanel`.

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`[19] structure (the central widget type), the `Icon`[24a] 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`[19] 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` 19⟩≡ (129d)

```
struct Panel{
    // public

    Rectangle r;     /* where the Panel goes */

    ⟨Panel padding fields 26d⟩
    ⟨Panel fixed size field 120c⟩
    ⟨Panel user-specific fields 119d⟩

    /* private below */

    Image *b;     /* where we're drawn */
    ⟨Panel widget-specific data 22d⟩
    ⟨Panel layout fields 41a⟩
    ⟨Panel debugging fields 124a⟩
    ⟨Panel other fields 20d⟩

    // methods
    ⟨Panel main methods 23b⟩
    ⟨Panel packing methods 42b⟩
    ⟨Panel other methods 22e⟩
```

```
    // Extra
    ⟨Panel extra fields 21a⟩
};
```

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()`[40] and `pldraw()`[30a] respectively. This two-phase initialization reflects the separation between layout and drawing described in Chapter 2.

The constructor `pl_newpanel()`[20a] 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` 20a⟩≡ (135b)
```
Panel *pl_newpanel(Panel *parent, int ndata){
    Panel *v;

    ⟨pl_newpanel() sanity check if can create child on parent 22c⟩
    v=pl_emalloc(sizeof(Panel));

    ⟨pl_newpanel() set tree fields 21e⟩
    ⟨pl_newpanel() set widget-specific data fields 22f⟩
    ⟨pl_newpanel() set other fields 20b⟩

    ⟨pl_newpanel() set default methods 23c⟩

    return v;
}
```
Uses `pl_emalloc()` 128a.

⟨`pl_newpanel()` *set other fields* 20b⟩≡ (20a) 20f▷
```
v->r=Rect(0,0,0,0);
v->b=nil;
```

The destructor `plfree()`[20c] 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` 20c⟩≡ (135b)
```
void plfree(Panel *p){
    ⟨plfree locals 21f⟩

    if(p==nil)
        return;
    ⟨plfree() if plkbfocus 38d⟩
    ⟨plfree() free the children 22a⟩
    ⟨plfree() free the widget-specific data 22g⟩
    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* 20d⟩≡ (19) 22h▷
```
// LEAF | INVIS | ...
int flags;     /* position flags, see below */
```

⟨*constant* `NOFLAG` 20e⟩≡ (129d)
```
#define NOFLAG 0
```

⟨`pl_newpanel()` *set other fields* 20f⟩+≡ (20a) ◁20b 26e▷
```
v->flags=NOFLAG;
```

3.1.2 A tree of widgets

Widgets form a tree: each `Panel`[19] 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* 21a⟩≡ (19) 21b▷
```
// option<list<ref_own<Panel>>> (next = Panel.next, tail = Panel.echild)
Panel *child;
```

⟨Panel *extra fields* 21b⟩+≡ (19) ◁21a 21c▷
```
// list<ref<Panel>> (head = Panel.child)
Panel *next;     /* It's a list! */
```

⟨Panel *extra fields* 21c⟩+≡ (19) ◁21b 21d▷
```
// option<ref<Panel>>
Panel *echild;
```

⟨Panel *extra fields* 21d⟩+≡ (19) ◁21c
```
// 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* 21e⟩≡ (20a)
```
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* 21f⟩≡ (20c)
```
Panel *cp, *ncp;
```

⟨plfree() *free the children* 22a⟩≡ (20c)
```
for(cp=p->child;cp;cp=ncp){
    ncp=cp->next;
    plfree(cp);
}
```
Uses plfree() 20c.

Leaf widgets (labels, buttons) cannot have children. The LEAF flag enforces this at construction time: pl_newpanel()[20a] refuses to create a child on a leaf widget.

⟨*constant* LEAF 22b⟩≡ (133e)
```
#define LEAF 0x10000  /* newpanel will refuse to attach children */
```

⟨pl_newpanel() *sanity check if can create child on* parent 22c⟩≡ (20a)
```
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[19] 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* 22d⟩≡ (19)
```
// union<ref_own<Label|Entry|Button|...>
void *data;     /* kind-specific data */
```

⟨Panel *other methods* 22e⟩≡ (19) 37d▷
```
void (*free)(Panel *);    /* free fields of data when done */
```

⟨pl_newpanel() *set widget-specific data fields* 22f⟩≡ (20a)
```
if(ndata)
    v->data=pl_emalloc(ndata);
else
    v->data=nil;
v->free=nil;
```
Uses pl_emalloc() 128a.

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* 22g⟩≡ (20c)
```
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* 22h⟩+≡ (19) ◁20d 35b▷
```
// enum<Style>
int state;     /* for hitting & drawing purposes */
```

⟨*enum* Style 23a⟩≡ (133e)

```
/*
 * States, also styles
 */
enum{
    // for text entries, buttons
    UP,
    DOWN,
    ⟨Style other cases 50c⟩
};
```

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()`[49c], `plbutton()`[59d]) fills in these pointers with its own implementation. The default values are error-raising stubs set by `pl_newpanel()`[20a], so forgetting to initialize them is caught immediately.

This pattern—a struct of function pointers serving as a vtable—is how C libraries do polymorphism, and it shows up in nearly every serious C codebase under a different name. The variations are mostly about *where* the pointers live and *who* fills them in:

library/system	vtable name	dispatch site
libpanel	Panel.{draw,hit,type}	inline in Panel
Linux kernel VFS	struct file_operations	one per filesystem
Linux kernel	struct device_driver	one per device class
GTK / GObject	GObjectClass	allocated per type id
Qt	vtbl (C++ generated)	compiler-emitted
COM (Windows)	IUnknown vtable	ABI-fixed offset 0
SQLite vtab	sqlite3_module	one per virtual table
stdio FILE	_IO_jump_t (glibc)	inline pointer in FILE
Plan 9 Dev	struct Dev	one per device
Go	itab (interface table)	runtime-built

Two design choices distinguish `libpanel` from the heavier options. (1) The vtable is *inlined* in `Panel` itself rather than living in a separate "class" struct shared by all instances of a kind. This costs roughly 60 extra bytes per widget (eight pointers × 8 bytes) but means the same `Panel` can be *reinitialized* in place to a different widget kind just by overwriting the pointers, which is exactly what the `plinitxxx()` reinitialization pattern relies on. (2) There is no class-id field and no `isa` check: a widget's "type" is simply the set of function pointers it carries. The library never asks "is this a button?"—it just calls `p->hit(p,m)` and lets the right implementation run. The result is OO with zero runtime type information, the absolute minimum the technique needs to work.

⟨`Panel` *main methods* 23b⟩≡ (19)

```
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* 23c⟩≡ (20a) 42c▷

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

Uses `pl_drawerror()` 126b, `pl_hiterror()` 126c, and `pl_typeerror()` 126d.

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`[24a] 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()`[51b] dispatching at runtime. This lets widgets like buttons and labels accept either text or image content through the same interface.

⟨*type* `Icon` 24a⟩≡ (129d)
```
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`[19]: 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` 24b⟩≡ (129d)
```
#define PLACE 0x01e0  /* which side of its space should the Panel adhere to? */
```

⟨*constant* `PLACECEN` 24c⟩≡ (129d)
```
#define  PLACECEN 0x0000
```

⟨*constant* `PLACES` 24d⟩≡ (129d)
```
#define  PLACES 0x0020
```

⟨*constant* `PLACEE` 24e⟩≡ (129d)
```
#define  PLACEE 0x0040
```

⟨*constant* `PLACEW` 24f⟩≡ (129d)
```
#define  PLACEW 0x0060
```

⟨*constant* `PLACEN` 24g⟩≡ (129d)
```
#define  PLACEN 0x0080
```

⟨*constant* `PLACENE` 24h⟩≡ (129d)
```
#define  PLACENE 0x00a0
```

⟨*constant* `PLACENW` 24i⟩≡ (129d)
```
#define  PLACENW 0x00c0
```

⟨*constant* `PLACESE` 24j⟩≡ (129d)
```
#define  PLACESE 0x00e0
```

⟨*constant* `PLACESW` 24k⟩≡ (129d)
```
#define  PLACESW 0x0100
```

```
 PLACENW       PLACEN       PLACENE
+----------+  +----------+  +----------+
|[w]       |  |   [w]    |  |       [w]|
|          |  |          |  |          |
|          |  |          |  |          |
+----------+  +----------+  +----------+

 PLACEW       PLACECEN     PLACEE
+----------+  +----------+  +----------+
|          |  |          |  |          |
|[w]       |  |   [w]    |  |       [w]|
|          |  |          |  |          |
+----------+  +----------+  +----------+

 PLACESW      PLACES       PLACESE
+----------+  +----------+  +----------+
|          |  |          |  |          |
|          |  |          |  |          |
|[w]       |  |   [w]    |  |       [w]|
+----------+  +----------+  +----------+
```

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 25a⟩≡ (129d)
```
#define PACK 0x0007  /* which side of the parent is the Panel attached to? */
```

⟨*constant* PACKN 25b⟩≡ (129d)
```
#define  PACKN 0x0000
```

⟨*constant* PACKE 25c⟩≡ (129d)
```
#define  PACKE 0x0001
```

⟨*constant* PACKS 25d⟩≡ (129d)
```
#define  PACKS 0x0002
```

⟨*constant* PACKW 25e⟩≡ (129d)
```
#define  PACKW 0x0003
```

⟨*constant* PACKCEN 25f⟩≡ (129d)
```
#define  PACKCEN 0x0004 /* only used by pulldown */
```

```
  Parent space              PACKN then PACKN       PACKN then PACKE
 +-------------+           +-------------+        +-------------+
 |             |           |  [widget A] |        |  [widget A] |
 |             |           +-------------+        +------+------+
 |             |           |  [widget B] |        |      |[B]   |
 |             |           +-------------+        |      |      |
 |             |           | (remaining) |        | rest |      |
 +-------------+           +-------------+        +------+------+
```

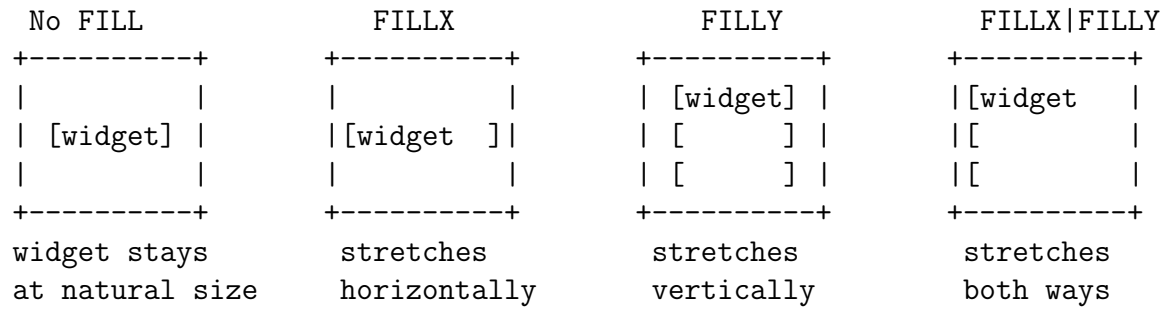
3.3.3 Filling

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()`[42a]). `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` 26a⟩≡ (129d)
```
#define FILLX 0x0008  /* grow horizontally to fill the available space */
```

⟨*constant* `FILLY` 26b⟩≡ (129d)
```
#define FILLY 0x0010  /* grow vertically to fill the available space */
```

⟨*constant* `EXPAND` 26c⟩≡ (129d)
```
#define EXPAND 0x0200  /* use up all extra space in the parent */
```

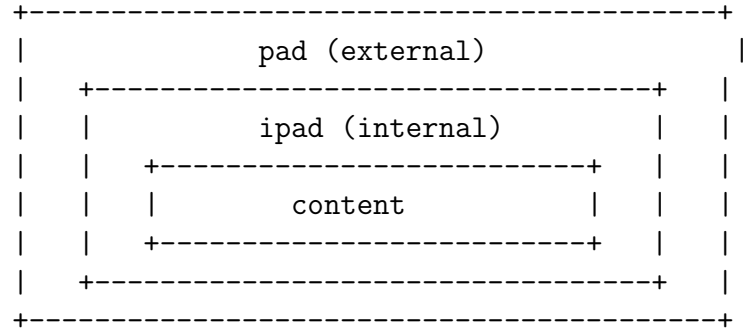


3.3.4 Padding

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* 26d⟩≡ (19)
```
Vector ipad;    /* extra space inside and outside */
Vector pad;
```

⟨`pl_newpanel()` *set other fields* 26e⟩+≡ (20a) ◁20f 35c▷
```
v->ipad=Pt(0,0);
v->pad=Pt(0,0);
```



3.4 Direction

Scrollbars and sliders can be oriented horizontally or vertically.

⟨*enum* Direction 27⟩≡ (133e)

```
/*
 * 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()`[28a].

4.1 `plinit()`

⟨*function* `plinit` 28a⟩≡ (135a)
```
/*
 * Just a wrapper for all the initialization routines
 */
error0 plinit(int ldepth){
    return pl_drawinit(ldepth);
}
```
Uses `pl_drawinit()` 28d.

The actual work is in `pl_drawinit()`[28d] 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` 28b⟩≡ (135c)
```
static int plldepth;
```

⟨*globals* `pl_xxx` 28c⟩≡ (135c)
```
static Image *pl_white, *pl_light, *pl_dark, *pl_black, *pl_hilit;
```

⟨*function* `pl_drawinit` 28d⟩≡ (135c)
```
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` 28c, `pl_dark-25` 28c, `pl_hilit-27` 28c, `pl_light-24` 28c, `pl_white-23` 28c, and `plldepth-22` 28b.

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()`[30a] takes the root widget and the destination `Image *b` (typically `view`, the screen). It delegates to `pl_draw1()`[30b] and then flushes the display to make the result visible.

⟨*function* `pldraw` 30a⟩≡ (135c)
```
void pldraw(Panel *p, Image *b){
    pl_draw1(p, b);
    flushimage(display, true);
}
```
Uses `pl_draw1()` 30b.

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` 30b⟩≡ (135c)
```
void pl_draw1(Panel *p, Image *b){
    if(b!=nil)
        pl_drawall(p, b);
}
```
Uses `pl_drawall()` 30c.

`pl_drawall()`[30c] 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` 30c⟩≡ (135c)
```
void pl_drawall(Panel *p, Image *b){
    ⟨pl_drawall() if invisible widget 94f⟩
    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()` 30b.

The traversal is a preorder depth-first walk: every widget paints itself *before* recursing into its children, so children overlay their parent. On a frame with a label, a scrollable list, and a scroll bar, the draw order is:

```
(1) frame              <- paints its background + relief
     |
     +-- (2) label     <- paints on top of frame
     |
     +-- (3) listbox   <- paints on top of frame
     |        |
     |        +-- (4) visible items
     |
     +-- (5) scrollbar <- paints on top of frame

draw order: 1 -> 2 -> 3 -> 4 -> 5
```

This is the opposite of a bottom-up compositor (the model used by Wayland or Core Animation, where leaves are rendered first and parents assemble them). `libpanel` paints directly on `view`, so a parent must paint first to avoid erasing its own children; each child then happily clobbers the parent's pixels in its `p->r` rectangle.

5.2 Drawing helpers

`pl_relief()`[31] 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` 31⟩≡ (135c)

```
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()`[32b] (filled) or `pl_outline()`[32a] (border only). Both dispatch to `pl_boxoutline()`[32c], 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` 32a⟩≡ (135c)
```
Rectangle pl_outline(Image *b, Rectangle r, int style){
    return pl_boxoutline(b, r, style, false);
}
```
Uses `pl_boxoutline()` 32c.

⟨*function* `pl_box` 32b⟩≡ (135c)
```
Rectangle pl_box(Image *b, Rectangle r, int style){
    return pl_boxoutline(b, r, style, true);
}
```
Uses `pl_boxoutline()` 32c.

⟨*function* `pl_boxoutline` 32c⟩≡ (135c)
```
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 50d⟩
    }
    return insetrect(r, SPACE);
}
```
Uses `SPACE-16` 32d.

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` 32d⟩≡ (135c)
```
#define SPACE 1 /* space inside relief of button or frame */
```

`pl_boxsize()`[32e] is the inverse: given an interior size and a state, it returns how much extra space the border adds. `pl_interior()`[32f] 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` 32e⟩≡ (135c)
```
Vector pl_boxsize(Vector interior, int state){
    switch(state){
    ⟨pl_boxsize() switch state cases 52e⟩
    }
    // else
    return Pt(0, 0);
}
```

⟨*function* `pl_interior` 32f⟩≡ (135c)
```
void pl_interior(int state, Point *ul, Vector *size){
    switch(state){
    ⟨pl_interior() switch state cases 33a⟩
    }
}
```

⟨pl_interior() *switch state cases* 33a⟩≡ (32f) 33b ▷

```
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 54e, DOWN 50c, DOWN1 23a, DOWN2 23a, DOWN3, SPACE-16 32d, and UP 23a.

⟨pl_interior() *switch state cases* 33b⟩+≡ (32f) ◁33a 73h ▷

```
case PASSIVE:
    *ul=addpt(*ul, Pt(PWID+SPACE, PWID+SPACE));
    *size=subpt(*size, Pt(2*(PWID+SPACE), 2*(PWID+SPACE)));
    break;
```

Uses PASSIVE 50c, PWID-13 51a, and SPACE-16 32d.

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()`[35a] or `plkeyboard()`[38c] to route them to the right widget.

6.1 Event dispatch principles

Every widget library has the same starting problem: the operating system hands the application a stream of raw input events (mouse moved to (x, y); key pressed; button clicked) and the library must figure out which of the many widgets on screen should receive each event, then invoke the right code inside that widget. Doing this well is most of what distinguishes widget libraries from one another, because it is where the design choices about focus, modality, popups, and event consumption all come together.

Four dispatch models dominate practical widget libraries. The *capture/target/bubble* model, invented for Smalltalk and generalized by the W3C DOM, walks the event from the root of the tree *down* to the leafward widget under the mouse (the capture phase), delivers it there (target), and then walks back *up* toward the root (bubble)—every ancestor gets a chance to intercept on the way down or on the way up. Browsers, React, and most modern web frameworks use this. The *responder chain* model (Apple's Cocoa and UIKit) skips the capture phase: an event is delivered to the view under the mouse, and if that view does not handle it, the event walks upward through parent views until someone claims it—a cleaner single-pass traversal that fits Apple's view-controller architecture. The *centralized procedure* model (Win32 `WndProc` and classic X11 event-handler callbacks) routes every event to a single event procedure on the top-level window, which is then responsible for dispatching internally however it likes—maximally flexible, minimally structured, and the oldest of the four. Finally, the *priority-based* model, used by `libpanel` and a handful of older toolkits such as the X Athena widget set, gives every widget a priority number and picks the highest-priority widget whose rectangle contains the hit-test point, regardless of depth in the tree.

`libpanel`'s priority-based model is the simplest of the four to implement—no phases to walk, no chain to define, no centralized procedure to maintain—but it is also the most unusual to a reader coming from web or Cocoa. Its one big strength shows up with popup menus, tooltips, and modal dialogs: a popup declared as a sibling of a text area still wins hit tests over the text area while visible, because its higher priority trumps depth in the tree. The same trick avoids the need for a separate "capture" phase; you just hand the popup widget a high priority and it intercepts events over whatever it is covering. The price is that priorities are a global property of each widget, so two popups on screen at once have to coordinate their priorities manually—but in practice only one modal element is typically active at a time, so the friction rarely appears.

The rest of this chapter walks through how `libpanel` turns the abstract dispatch model into C code. The mouse path starts at `plmouse()`, which calls `pl_ptinpanel()` to recursively find the highest-priority widget whose rectangle contains the mouse point and then invokes that widget's `hit` method. The chapter also covers the `OUT`-bit trick for notifying a widget that the mouse has left it (reusing the existing `hit` method rather than

adding a separate leave method), the REMOUSE mechanism for widgets that want to keep grabbing events even after the mouse has moved elsewhere, and the keyboard-event counterpart plkeyboard()—which has to use a different strategy because keyboard events do not have a natural hit-test point the way mouse events do.

6.2 Mouse events

plmouse()[35a] 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()[35d], 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 35a⟩≡ (136a)
```
void plmouse(Panel *g, Mouse *m){
    Panel* hit;
    Panel* last = g->lastmouse;
    bool remouse;

    ⟨plmouse() if REMOUSE set hit to last 37c⟩
    else{
        hit=pl_ptinpanel(m->xy, g);
        if(last && last!=hit){
            ⟨plmouse() when last!=hit send OUT mouse event 36b⟩
        }
    }
    if(hit){
        // widget-specific method
        remouse=hit->hit(hit, m);
        ⟨plmouse() handle remouse 37b⟩
        g->lastmouse=hit;
    }
    flushimage(display, true);
}
```
Uses pl_ptinpanel() 35d.

⟨Panel *other fields* 35b⟩+≡ (19) ◁22h 76a▷
```
Panel *lastmouse;    /* who got the last mouse event? */
```

⟨pl_newpanel() *set other fields* 35c⟩+≡ (20a) ◁26e 37h▷
```
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 35d⟩≡ (136a)
```
/*
 * 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()` 35d.

Here is how `pl_ptinpanel()` resolves a click at point `p` on a tree where a popup menu (priority 1) hovers over a text area (priority 0) inside a frame. The click lands inside all three rectangles, so every level of the recursion has a candidate:

```
            frame (pri=0) r = (0,0)-(400,300)
            /          \
           /            \
      textarea (0)     popup (1)  r = (50,80)-(200,180)
      r=(0,20)-(400,300)
                        |
                        *  <- click p=(120,120)
```

```
pl_ptinpanel(p, frame):
    p in frame->r        yes
    recurse on textarea:
        p in textarea->r yes
        recurse (no children) -> nil
        return textarea (pri 0)
    recurse on popup:
        p in popup->r    yes
        recurse (no children) -> nil
        return popup (pri 1)
    compare: popup.pri(1) >= textarea.pri(0) -> popup wins
    return popup
```

The recursion visits children in sibling order but always keeps the *highest priority* hit, which is why a popup declared as a sibling of the textarea still intercepts the click even though the textarea is listed first.

6.2.1 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` 36a⟩≡ (129d)

```
/*
 * 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* 36b⟩≡ (35a)

```
m->buttons|=OUT;
last->hit(last, m);
m->buttons&=~OUT;
```

6.2.2 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()`[35a] skips `pl_ptinpanel()`[35d] and sends the event directly to the same widget.

⟨*constant* `REMOUSE` 37a⟩≡ (133e)
```
#define REMOUSE 0x40000  /* send next mouse event here, even if not inside */
```

⟨`plmouse()` *handle* `remouse` 37b⟩≡ (35a)
```
if(remouse)
    g->flags|=REMOUSE;
else
    g->flags&=~REMOUSE;
```
Uses `REMOUSE`.

⟨`plmouse()` *if* `REMOUSE` *set* `hit` *to* `last` 37c⟩≡ (35a)
```
if(g->flags&REMOUSE)
    hit=last;
```
Uses `REMOUSE`.

6.2.3 Hitting priority

The `pri` method is another virtual method on `Panel`[19], 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* 37d⟩+≡ (19) ◁22e 77e▷
```
int (*pri)(Panel *, Point);   /* priority for hitting */
```

⟨*constant* `PRI_NORMAL` 37e⟩≡ (129d)
```
#define PRI_NORMAL 0  /* ordinary panels */
```

⟨*constant* `PRI_POPUP` 37f⟩≡ (129d)
```
#define PRI_POPUP 1  /* popup menus */
```

⟨*constant* `PRI_SCROLLBAR` 37g⟩≡ (129d)
```
#define PRI_SCROLLBAR 2  /* scroll bars */
```

⟨`pl_newpanel()` *set other fields* 37h⟩+≡ (20a) ◁35c 41b▷
```
v->pri=pl_prinormal;
```
Uses `pl_prinormal()` 37i.

⟨*function* `pl_prinormal` 37i⟩≡ (135b)
```
int pl_prinormal(Panel *, Point){
    return PRI_NORMAL;
}
```

⟨*function* `pl_pripopup` 37j⟩≡ (141a)
```
int pl_pripopup(Panel *, Point){
    return PRI_POPUP;
}
```

⟨*function* `pl_priscrollbar` 37k⟩≡ (145d)
```
int pl_priscrollbar(Panel *, Point){
    return PRI_SCROLLBAR;
}
```

6.3 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()`[38b] in their `hit` method to claim the focus. Widgets that do not handle keyboard input (labels, buttons, scrollbars) simply never call it.

⟨*global* `plkbfocus` 38a⟩≡ (129d)
```
Panel *plkbfocus;   /* the panel in keyboard focus */
```

⟨*function* `plgrabkb` 38b⟩≡ (136a)
```
void plgrabkb(Panel *g){
    plkbfocus=g;
}
```

`plkeyboard()`[38c] 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` 38c⟩≡ (136a)
```
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` 38d⟩≡ (20c)
```
if(p==plkbfocus)
    plkbfocus=nil;
```

Three independent pieces of state together encode "who is currently grabbing input." They live in different places and answer different questions:

```
where stored              what it captures
------------------------  --------------------------------
plkbfocus (global ptr)    keyboard focus: next plkeyboard()
                          goes here, regardless of mouse
root->lastmouse           last widget reached by hit test;
                          used to send OUT on leave
root->flags & REMOUSE     sticky mouse capture: bypass hit
                          test and resend to lastmouse

typical text entry drag:
   click inside entry  -> plkbfocus = entry   (grabkb)
                          lastmouse  = entry
                          REMOUSE    = 0      (hit returned false)
   type 'a'            -> plkeyboard -> entry->type
   click-drag slider   -> lastmouse  = slider
                          REMOUSE    = 1      (hit returned true)
                          plkbfocus  unchanged (still entry!)
   release mouse       -> REMOUSE    = 0
```

Keyboard focus and mouse capture are deliberately decoupled: a user dragging a slider does not lose the typing focus on a text entry elsewhere in the window. The only widget-side work needed to opt into keyboard input is calling `plgrabkb()` from `hit`; everything else is bookkeeping by `plmouse()`[35a] on the root.

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()`[42a], 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()`[44a], 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 bottom-up-then-top-down pattern is universal in GUI toolkits, but the names and the algorithm running inside each pass differ considerably:

toolkit	pass 1 (bottom-up)	pass 2 (top-down)
libpanel	pl_sizereq()	pl_setrect() + packer
Tk	-reqwidth/-reqheight	pack/place/grid manager
GTK	gtk_widget_size_request	gtk_widget_size_allocate
Qt	sizeHint() + minimumSize	QLayout::setGeometry
Android	onMeasure()	onLayout()
Cocoa AppKit	intrinsicContentSize + NSLayoutConstraint	setFrame: + Auto Layout
SwiftUI	preferredSize	place(at:proposal:)
Web (CSS layout)	intrinsic / preferred	flex/grid/normal flow
Win32	WM_GETMINMAXINFO	WM_SIZE / MoveWindow
Flutter	performLayout (single pass with constraints flowing both ways)	same call computes both

Three things follow from looking across the column. First, *everyone* except a true constraint solver runs two passes, because there is no other way to reconcile "what the children want" with "what the parent can give"—one direction must go first. Second, the algorithm in pass 2 is what really distinguishes toolkits: Tk's `pack` (which `libpanel` copies) carves strips from one side at a time, GTK's `GtkBox` is similar, Qt's `QGridLayout` solves a small linear system, CSS Flexbox is a fixed-point loop, and Android can call `onMeasure` multiple times if a child reports a constraint that the parent must propagate back. Third, Flutter is the exception: it merges both passes by passing constraints *down* and sizes *up* in the same recursive call, which is faster but requires every widget to be a pure function of its constraints—no mutable layout state. `libpanel`'s two passes are the simplest possible point on this spectrum.

The entry point `plpack()`[40] runs both passes in sequence. It is called from `eresized()` with `view->r` as the available rectangle.

⟨*function* `plpack` 40⟩≡ (140c)

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

```
}
```
Uses pl_setrect() 44a and pl_sizereq() 42a.

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() [42d]). 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* 41a⟩≡ (19) 41c▷
```
Vector sizereq;     /* size requested by this Panel */
Vector childreq;     /* total size needed by children */
```

⟨pl_newpanel() *set other fields* 41b⟩+≡ (20a) ◁37h 41d▷
```
v->sizereq=Pt(0,0);
```

⟨Panel *layout fields* 41c⟩+≡ (19) ◁41a
```
Vector size;     /* space for this Panel */
```

⟨pl_newpanel() *set other fields* 41d⟩+≡ (20a) ◁41b 76b▷
```
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()`[42d]), 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` 42a⟩≡ (140c)
```
/*
 * Compute the requested size of p and its descendants.
 */
void pl_sizereq(Panel *p){
    Panel *cp;
    ⟨pl_sizeref() other locals 120h⟩

    for(cp=p->child;cp;cp=cp->next){
        // recurse
        pl_sizereq(cp);
        ⟨pl_sizeref() when looping over children, adjust maxsize 120i⟩
    }
    ⟨pl_sizereq() if MAXX or MAXY 120j⟩
    p->childreq=pl_sizesibs(p->child);
    // widget-specific method
    p->sizereq=p->getsize(p, p->childreq);
    ⟨pl_sizereq() adjust size with padding 43e⟩
    ⟨pl_sizereq() if FIXEDX or FIXEDY 120d⟩
}
```
Uses `pl_sizereq()` 42a and `pl_sizesibs()` 42d.

⟨`Panel` *packing methods* 42b⟩≡ (19) 46c▷
```
Vector (*getsize)(Panel *, Vector);  /* return size, given child size */
```

⟨`pl_newpanel()` *set default methods* 42c⟩+≡ (20a) ◁23c 46d▷
```
v->getsize=pl_getsizeerror;
```
Uses `pl_getsizeerror()` 126e.

```
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()`[42d] 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` 42d⟩≡ (140c)
```
Vector pl_sizesibs(Panel *p){
    Vector s;
```

```
    ⟨pl_sizesibs() if no panel 43a⟩
    // recurse
    s=pl_sizesibs(p->next);

    switch(p->flags&PACK){
    ⟨pl_sizesibs() switch packing flags cases 43b⟩
    }
    return s;
}
```

Uses pl_sizesibs() 42d.

⟨pl_sizesibs() *if no panel* 43a⟩≡ (42d)

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

⟨pl_sizesibs() *switch packing flags cases* 43b⟩≡ (42d) 43c ▷

```
case PACKN:
case PACKS:
    s.x=pl_max(s.x, p->sizereq.x);
    s.y+=p->sizereq.y;
    break;
```

Uses pl_max() 43d.

⟨pl_sizesibs() *switch packing flags cases* 43c⟩+≡ (42d) ◁ 43b

```
case PACKE:
case PACKW:
    s.x+=p->sizereq.x;
    s.y=pl_max(s.y, p->sizereq.y);
    break;
```

Uses pl_max() 43d.

⟨*function* pl_max 43d⟩≡ (140c)

```
int pl_max(int a, int b){
    return a>b ? a : b;
}
```

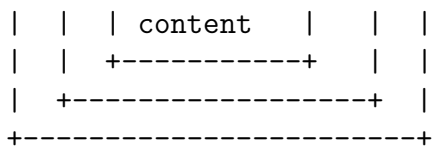
```
PACKN/PACKS (vertical stacking)     PACKE/PACKW (horizontal stacking)

+--------+  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* 43e⟩≡ (42a)

```
p->sizereq=addpt(addpt(p->sizereq, p->ipad), p->pad);
```

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

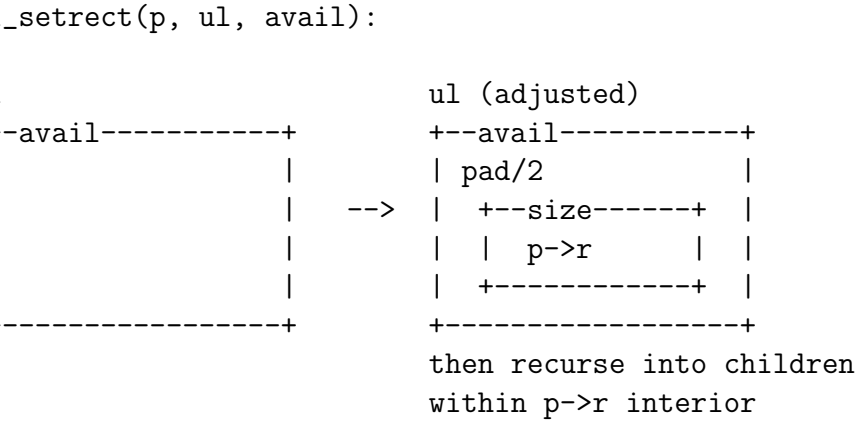


7.3 Computing rectangles: `pl_setrect()`

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` 44a⟩≡ (140c)

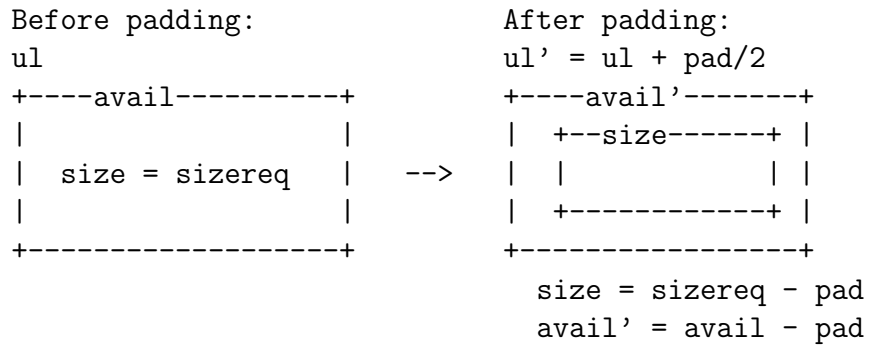
```
/*
 * Set the sizes and rectangles of p and its descendants, given their requested sizes.
 */
void pl_setrect(Panel *p, Point ul, Vector avail){
    ⟨pl_setrect() locals 46a⟩

    // setting p->size
    p->size=p->sizereq;
    ⟨pl_setrect() reduce size for external padding 44b⟩
    ⟨pl_setrect() reduce size if bigger than available 45a⟩
    ⟨pl_setrect() expand size if fill or expand flags 45b⟩

    // setting ul
    ⟨pl_setrect() adjust origin with placement 45c⟩

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

    // recurse
    ⟨pl_setrect() setting the rectangle of the children 46b⟩
}
```



7.3.1 Padding

⟨`pl_setrect()` *reduce size for external padding* 44b⟩≡ (44a)

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



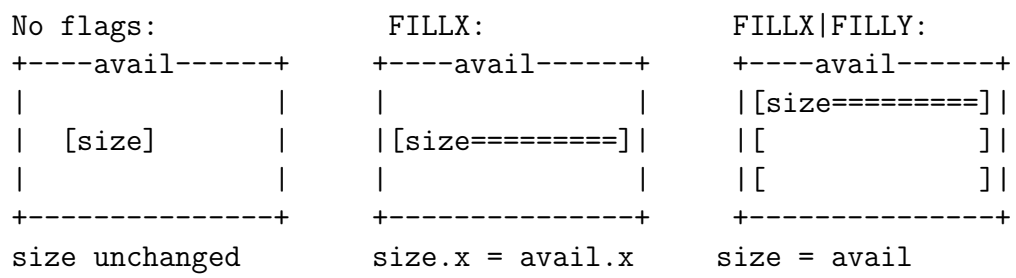
7.3.2 Filling

⟨pl_setrect() *reduce size if bigger than available* 45a⟩≡ (44a)

```
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* 45b⟩≡ (44a)

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

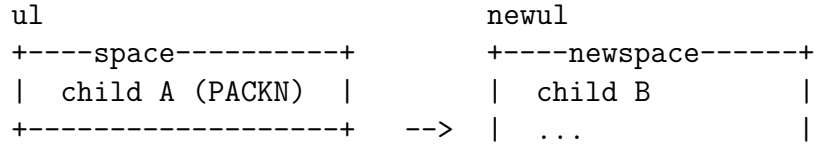


7.3.3 Placing

⟨pl_setrect() *adjust origin with placement* 45c⟩≡ (44a)

```
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



```
|  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* 46a⟩≡ (44a) 48a▷
```
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* 46b⟩≡ (44a)
```
space=p->size;
// widget-specific adjustment method
p->childspace(p, &ul, &space);

⟨pl_setrect() before looping over children, set locals 48b⟩
for(c=p->child;c;c=c->next){
    ⟨pl_setrect() when looping over children, if EXPAND flag 48c⟩
    switch(c->flags&PACK){
    ⟨pl_setrect() when looping over children, switch packing cases 47a⟩
    }
    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* 46c⟩+≡ (19) ◁42b
```
/* child ul & size given our size */
void (*childspace)(Panel *, Point * /**INOUT**/, Vector * /**INOUT**/);
```

⟨pl_newpanel() *set default methods* 46d⟩+≡ (20a) ◁42c 77f▷
```
v->childspace=pl_childspaceerror;
```
Uses pl_childspaceerror() 126f.

```
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* 47a⟩≡ (46b) 47b▷

```
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() 44a.

```
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* 47b⟩+≡ (46b) ◁47a

```
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() 44a.

Expanding child

When children have the EXPAND flag, the leftover space (slack = available minus total requested) is distributed among them. pl_getshare()[48d] 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* 48a⟩+≡ (44a) ◁46a
```
Vector slack, share;
int l;
```

⟨pl_setrect() *before looping over children, set locals* 48b⟩≡ (46b)
```
slack=subpt(space, p->childreq);
share=pl_getshare(p->child);
```
Uses pl_getshare() 48d.

⟨pl_setrect() *when looping over children, if* EXPAND *flag* 48c⟩≡ (46b)
```
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 48d⟩≡ (140c)
```
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() 48d.

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`[49a], `Entry`[52g], `Button`[58d]), a constructor that calls `pl_newpanel()`[20a] 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`[24a] (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` 49a⟩≡ (136c)
```
struct Label{
    // enum<Placement> (default = PLACECEN)
    int placement;
    // ref_own<Icon> = ref_own<Image|string> depending on Panel.flags&BITMAP
    Icon *icon;
};
```

⟨*constant* `BITMAP` 49b⟩≡ (129d)
```
#define BITMAP 0x4000  /* text argument is a bitmap, not a string */
```

8.1.1 Initializing

⟨*function* `pllabel` 49c⟩≡ (136c)
```
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()` 20a, `plinitlabel()` 49d, and `plplacelabel()` 50a.

⟨*function* `plinitlabel` 49d⟩≡ (136c)
```
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() 52f, pl_drawlabel() 50b, pl_getsizelabel() 52d, pl_hitlabel() 52b, and pl_typelabel() 52c.

Recall that `v->data` was already allocated by `pl_newpanel()`[20a] with `sizeof(Label)` bytes, so `plinitlabel()`[49d] 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()`[49c] 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 50a⟩≡ (136c)

```
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()`[32b] (which internally calls `pl_boxoutline()`[32c]), then draw the icon inside it using the label's placement setting.

⟨*function* pl_drawlabel 50b⟩≡ (136c)

```
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 50c, pl_box() 32b, and pl_drawicon() 51b.

⟨Style *other cases* 50c⟩≡ (23a) 72d ▷

```
// 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* 50d⟩≡ (32c) 54d ▷

```
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 50c, PWID-13 51a, SPACE-16 32d, pl_light-24 28c, and pl_white-23 28c.

⟨*constant* PWID 51a⟩≡ (135c)

```
#define PWID 1 /* width of label border */
```

`pl_drawicon()`[51b] is the shared helper that actually renders an `Icon`[24a] (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 51b⟩≡ (135c)

```
void pl_drawicon(Image *b, Rectangle r, int stick, int flags, Icon *s){
    Point ul;
    Vector offs;
    ⟨pl_drawicon() other locals 51e⟩

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

    switch(stick){
    ⟨pl_drawicon() switch placement cases, adjust ul 51d⟩
    }

    ⟨pl_drawicon() save and adjust clip rectangle 51f⟩
    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 52a⟩
}
```

Uses pl_black-26 28c and pl_iconsize() 51c.

⟨*function* pl_iconsize 51c⟩≡ (135c)

```
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 51d⟩≡ (51b)

```
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* 51e⟩≡ (51b)

```
Rectangle save;
```

⟨pl_drawicon() *save and adjust clip rectangle* 51f⟩≡ (51b)

```
save=b->clipr;
if(!rectclip(&r, save))
    return;
replclipr(b, b->repl, r);
```

⟨pl_drawicon() *restore saved clip rectangle* 52a⟩≡ (51b)
```
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 52b⟩≡ (136c)
```
bool pl_hitlabel(Panel *p, Mouse *m){
    USED(p, m);
    return false;
}
```

⟨*function* pl_typelabel 52c⟩≡ (136c)
```
void pl_typelabel(Panel *p, Rune c){
    USED(p, c);
}
```

8.1.4 Packing methods

⟨*function* pl_getsizelabel 52d⟩≡ (136c)
```
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 50c, pl_boxsize() 32e, and pl_iconsize() 51c.

⟨pl_boxsize() *switch state cases* 52e⟩≡ (32e) 58a▷
```
case PASSIVE:
    return addpt(interior, Pt(2*(PWID+SPACE), 2*(PWID+SPACE)));
```
Uses PASSIVE 50c, PWID-13 51a, and SPACE-16 32d.

The childspace method is called by pl_setrect()[44a] 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 52f⟩≡ (136c)
```
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 52g⟩≡ (137b)
```
struct Entry{
    // ref_own<string>
    char *entry;
    ⟨Entry pointer in entry fields 53a⟩
    Vector minsize;

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

⟨Entry *pointer in* entry *fields* 53a⟩≡ (52g)
```
// point to \0 in entry
char *entp;
// point to end of entry
char *eent;
```

8.2.1 Initializing

⟨*function* plentry 53b⟩≡ (137b)
```
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() 20a and plinitentry() 53c.

⟨*function* plinitentry 53c⟩≡ (137b)
```
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 118b⟩
    ⟨plinitentry() set extra fields 54b⟩
    ⟨plinitentry() set fields in ep 53e⟩

    v->free=pl_freeentry;

    v->kind="entry";
}
```
Uses LEAF, pl_childspaceentry() 58b, pl_drawentry() 54c, pl_freeentry() 53d, pl_getsizeentry() 57d, pl_hitentry() 55e, and pl_typeentry() 56a.

⟨*function* pl_freeentry 53d⟩≡ (137b)
```
void pl_freeentry(Panel *p){
    Entry *ep = p->data;

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

⟨plinitentry() *set fields in* ep 53e⟩≡ (53c)
```
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 118e and pl_erealloc() 128b.

⟨*constant* SLACK*(*(lib_gui/libpanel/textwin.c)*)* 54a⟩≡ (156d)
```
  #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* 54b⟩≡ (53c)
```
  v->state=UP;
```

Uses UP 23a.

⟨*function* pl_drawentry 54c⟩≡ (137b)
```
  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 55c⟩
      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 55d⟩
  }
```

Uses pl_box() 32b and pl_drawicon() 51b.

⟨pl_boxoutline() *switch style cases* 54d⟩+≡ (32c) ◁50d 55a▷
```
  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 54e, SPACE-16 32d, UP 23a, pl_black-26 28c, pl_light-24 28c, pl_relief() 31, and pl_white-23 28c.

⟨*constant* BWID 54e⟩≡ (135c)
```
  #define BWID 1 /* width of button relief */
```

⟨pl_boxoutline() *switch style cases* 55a⟩+≡ (32c) ◁54d 73a▷

```
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 54e, DOWN 50c, DOWN1 23a, DOWN2 23a, DOWN3, SPACE-16 32d, pl_black-26 28c, pl_dark-25 28c, pl_relief() 31, and pl_white-23 28c.

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 55b⟩≡ (129d)

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

⟨pl_drawentry *if* USERFL 55c⟩≡ (54c)

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

⟨pl_drawentry *when* USERFL *if* s *changed* 55d⟩≡ (54c)

```
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()[38b], 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 55e⟩≡ (137b)

```
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 119a⟩
        }
        // redraw with changed state
```

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

Uses DOWN 50c, UP 23a, pldraw() 30a, and plgrabkb() 38b.

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 56a⟩≡ (137b)

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

        switch(c){
        ⟨pl_typeentry() switch rune cases 56b⟩
        }
        // draw updated entry
        pldraw(p, p->b);
    }
```

Uses pldraw() 30a.

⟨pl_typeentry() *switch rune cases* 56b⟩≡ (56a) 56c▷

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

⟨pl_typeentry() *switch rune cases* 56c⟩+≡ (56a) ◁56b 57b▷

```
    default:
        ⟨pl_typeentry() handle special characters 57a⟩
        ep->entp+=runetochar(ep->entp, &c);
        ⟨pl_typeentry() realloc entry if necessary 56d⟩
        *ep->entp='\0';
        break;
```

⟨pl_typeentry() *realloc entry if necessary* 56d⟩≡ (56c)

```
    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 118e and pl_erealloc() 128b.

Special keys

The backspace case is more subtle than it looks because `entry` is a UTF-8 *byte* buffer but what the user wants to erase is a *rune*. A single non-ASCII rune can occupy 2, 3, or 4 bytes, and `entp` is a byte pointer, not a rune index. The loop walks backward across any number of UTF-8 trail bytes until it hits a lead byte (recognized by `pl_rune1st`), then erases that lead byte too. For the French word "été" (é = U+00E9 = `0xC3 0xA9`) the buffer and pointers look like this before a backspace on the final é:

```
byte index:    0    1    2    3    4    5    6
entry:      [ 'C3 A9'  't'  'C3 A9'  '\0'  ...]
              '---'       '---'
               e=U+00E9    e=U+00E9
                                ^
                                entp  (byte offset 5, rune offset 3)

backspace:
  step 1: entp[-1]=0xA9, not a rune start -> clear, --entp (byte 4)
  step 2: entp[-1]=0xC3, is  a rune start -> stop
  step 3: final --entp, clear byte 3              (byte 3)
```

After the backspace, `entp` points at byte 3 (rune 2), the buffer contains "ét", and the invariant "`entp` always points at a rune boundary" holds. A naive `--entp` would have left the buffer stuck on the trailing `0xA9` and corrupted the next `runetochar` on insert.

⟨`pl_typeentry()` *handle special characters* 57a⟩≡ (56c)
```
if(c < 0x20 || (c & 0xFF00) == KF || (c & 0xFF00) == Spec)
    break;
```

⟨`pl_typeentry()` *switch rune cases* 57b⟩+≡ (56a) ◁56c 57c▷
```
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()` 157b and `plsnarf()` 117e.

⟨`pl_typeentry()` *switch rune cases* 57c⟩+≡ (56a) ◁57b
```
// 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` 57d⟩≡ (137b)
```
Vector pl_getsizeentry(Panel *p, Vector children){
    Entry* e = p->data;

    USED(children);
    return pl_boxsize(e->minsize, p->state);
}
```
Uses `pl_boxsize()` 32e.

⟨pl_boxsize() *switch state cases* 58a⟩+≡ (32e) ◁52e 73f▷
```
case UP:
case DOWN:
case DOWN1:
case DOWN2:
case DOWN3:
    return addpt(interior, Pt(2*(BWID+SPACE), 2*(BWID+SPACE)));
```
Uses BWID-14 54e, DOWN 50c, DOWN1 23a, DOWN2 23a, DOWN3, SPACE-16 32d, and UP 23a.

⟨*function* pl_childspaceentry 58b⟩≡ (137b)
```
void pl_childspaceentry(Panel *p, Point *ul, Vector *size){
    USED(p, ul, size);
}
```

8.2.5 Other methods

⟨*function* plentryval 58c⟩≡ (137b)
```
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`[58d] 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 58d⟩≡ (137a)
```
struct Button{
    // enum<ButtonType>
    int btype;   /* button type */

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

    ⟨Button other fields 59a⟩

    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 89b⟩
};
```

⟨*constant* BUTTON 58e⟩≡ (137a)
```
#define BUTTON 1
```

⟨*constant* CHECK 58f⟩≡ (137a)
```
#define CHECK 2
```

⟨*constant* RADIO 58g⟩≡ (137a)
```
#define RADIO 3
```

⟨Button *other fields* 59a⟩≡ (58d) 59b▷
```
// state of buttons when hit
buttons buttons;
```

⟨Button *other fields* 59b⟩+≡ (58d) ◁59a 89a▷
```
bool check;   /* for check/radio buttons */
```

⟨*function* pl_initbtype 59c⟩≡ (137a)
```
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 58e, CHECK-11 58f, LEAF, RADIO-12 58g, UP 23a, pl_childspacebutton() 61c, pl_drawbutton() 60b, pl_getsizebutton() 61b, pl_hitbutton() 60d, and pl_typebutton() 61a.

8.3.2 Basic button

Initializing

⟨*function* plbutton 59d⟩≡ (137a)
```
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() 20a and plinitbutton() 59e.

⟨*function* plinitbutton 59e⟩≡ (137a)
```
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 58e, pl_buttonhit() 60a, and pl_initbtype() 59c.

⟨*function* `pl_buttonhit` 60a⟩≡ (137a)

```
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` 60b⟩≡ (137a)

```
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 60c⟩
    }
    pl_drawicon(p->b, r, PLACECEN, p->flags, bp->icon);
}
```

Uses `pl_box()` 32b and `pl_drawicon()` 51b.

⟨`pl_drawbutton()` *switch button type cases* 60c⟩≡ (60b) 62a▷

```
case BUTTON:
    break;
```

Uses `BUTTON-10` 58e.

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` 60d⟩≡ (137a)

```
bool pl_hitbutton(Panel *p, Mouse *m){
    Button *bp = p->data;
    int oldstate = p->state;
    bool hitme;
    ⟨pl_hitbutton() other locals 64d⟩

    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 62g⟩
        }
    }
    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 50c, UP 23a, and pldraw() 30a.

⟨*function* pl_typebutton 61a⟩≡ (137a)

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

Packing methods

⟨*function* pl_getsizebutton 61b⟩≡ (137a)

```
Vector pl_getsizebutton(Panel *p, Vector children){
    Button *bp = p->data;
    Vector s;
    ⟨pl_getsizebutton() other locals 63a⟩

    USED(children);  /* shouldn't have any children */
    s=pl_iconsize(p->flags, bp->icon);
    ⟨pl_getsizebutton() if not a BUTTON 63b⟩
    return pl_boxsize(s, p->state);
}
```

Uses pl_boxsize() 32e and pl_iconsize() 51c.

⟨*function* pl_childspacebutton 61c⟩≡ (137a)

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

8.3.3 Check button

Initializing

⟨*function* plcheckbutton 61d⟩≡ (137a)

```
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() 20a and plinitcheckbutton() 61e.

⟨*function* plinitcheckbutton 61e⟩≡ (137a)

```
void plinitcheckbutton(Panel *p, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool)){
    pl_initbtype(p, flags, icon, hit, CHECK);
}
```

Uses CHECK-11 58f and pl_initbtype() 59c.

Drawing

⟨pl_drawbutton() *switch button type cases* 62a⟩+≡ (60b) ◁60c 64a▷

```
case CHECK:
    r=pl_check(p->b, r, bp->check);
    break;
```

Uses CHECK-11 58f and pl_check() 62b.

⟨*function* pl_check 62b⟩≡ (135c)

```
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 62f⟩
    }
    return remainder;
}
```

Uses CKBORDER-21 62e, CKINSET-20 62c, CKWID-19 62d, pl_black-26 28c, pl_light-24 28c, pl_relief() 31, and pl_white-23 28c.

⟨*constant* CKINSET 62c⟩≡ (135c)

```
#define CKINSET 1 /* space around check mark frame */
```

⟨*constant* CKWID 62d⟩≡ (135c)

```
#define CKWID 1 /* width of frame around check mark */
```

⟨*constant* CKBORDER 62e⟩≡ (135c)

```
#define CKBORDER 2 /* space around X inside frame */
```

⟨pl_check() *if checked button* 62f⟩≡ (62b)

```
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 28c.

Reacting

⟨pl_hitbutton() *switch button type cases* 62g⟩≡ (60d) 64e▷

```
case CHECK:
    if(hitme)
        bp->check=!bp->check;
    break;
```

Uses CHECK-11 58f.

Packing methods

⟨pl_getsizebutton() *other locals* 63a⟩≡ (61b)
```
int ckw;
```

⟨pl_getsizebutton() *if not a* BUTTON 63b⟩≡ (61b)
```
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 58e and pl_ckwid() 63c.

⟨*function* pl_ckwid 63c⟩≡ (135c)
```
int pl_ckwid(void){
    return 2*(CKINSET+CKSPACE+CKWID)+CKSIZE;
}
```
Uses CKINSET-20 62c, CKSIZE-17 63d, CKSPACE-18 64c, and CKWID-19 62d.

⟨*constant* CKSIZE 63d⟩≡ (135c)
```
#define CKSIZE 3 /* size of check mark */
```

Other methods

⟨*function* plsetbutton 63e⟩≡ (137a)
```
void plsetbutton(Panel *p, int val){
    Button* b = p->data;

    b->check=val;
}
```

8.3.4 Radio button

Initializing

⟨*function* plradiobutton 63f⟩≡ (137a)
```
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() 20a and plinitradiobutton() 63g.

⟨*function* plinitradiobutton 63g⟩≡ (137a)
```
void plinitradiobutton(Panel *p, int flags, Icon *icon, void (*hit)(Panel *, buttons, bool)){
    pl_initbtype(p, flags, icon, hit, RADIO);
}
```
Uses RADIO-12 58g and pl_initbtype() 59c.

Drawing

⟨pl_drawbutton() *switch button type cases* 64a⟩+≡ (60b) ◁62a
```
case RADIO:
    r=pl_radio(p->b, r, bp->check);
    break;
```
Uses RADIO-12 58g and pl_radio() 64b.

⟨*function* pl_radio 64b⟩≡ (135c)
```
/*
 * 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 62c, CKSPACE-18 64c, CKWID-19 62d, pl_black-26 28c, pl_light-24 28c, pl_relief() 31, and pl_white-23 28c.

⟨*constant* CKSPACE 64c⟩≡ (135c)
```
#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* 64d⟩≡ (60d)
```
Panel *sib;
```

⟨pl_hitbutton() *switch button type cases* 64e⟩+≡ (60d) ◁62g
```
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 58g, pl_hitbutton() 60d, and pldraw() 30a.

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 65a⟩≡ (139a)

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

    int val;   /* setting, in screen coordinates */
    ⟨Slider other fields 65b⟩

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

⟨Slider *other fields* 65b⟩≡ (65a)

```
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 65c⟩≡ (139a)

```
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() 20a and plinitslider() 65d.

⟨*function* plinitslider 65d⟩≡ (139a)

```
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 133c, LEAF, UP 23a, VERT, pl_childspaceslider() 68e, pl_drawslider() 66a, pl_getsizeslider() 68d, pl_hitslider() 67b, and pl_typeslider() 68c.

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 66a⟩≡ (139a)

```
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 133c, UP 23a, VERT, pl_box() 32b, and pl_sliderupd() 66b.

⟨*function* pl_sliderupd 66b⟩≡ (135c)

```
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 67a⟩

    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 133c, VERT, pl_dark-25 28c, and pl_light-24 28c.

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 67a⟩≡ (66b)

```
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 67b⟩≡ (139a)

```
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 68a⟩
            ⟨pl_hitslider() sanitize sp->val 68b⟩
        }
        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 50c, UP 23a, pl_interior() 32f, and pldraw() 30a.

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 68a⟩≡ (67b)

```
    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 133c.

⟨pl_hitslider() *sanitize* sp->val 68b⟩≡ (67b)

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

⟨*function* pl_typeslider 68c⟩≡ (139a)

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

8.4.4 Packing methods

⟨*function* pl_getsizeslider 68d⟩≡ (139a)

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

Uses pl_boxsize() 32e.

⟨*function* pl_childspaceslider 68e⟩≡ (139a)

```
    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` 69a⟩≡ (139a)
```
void plsetslider(Panel *p, int value, int range){
    Slider *sp = p->data;

    ⟨plsetslider() sanitize value 69b⟩
    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` 133c.

⟨`plsetslider()` *sanitize* `value` 69b⟩≡ (69a)
```
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` 69c⟩≡ (139b)
```
struct Canvas{
    void (*draw)(Panel *);
    void (*hit)(Panel *, Mouse *);
};
```

8.5.1 Initializing

⟨*function* `plcanvas` 69d⟩≡ (139b)
```
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()` 20a and `plinitcanvas()` 69e.

⟨*function* `plinitcanvas` 69e⟩≡ (139b)
```
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() 70e, pl_drawcanvas() 70a, pl_getsizecanvas() 70d, pl_hitcanvas() 70b, and pl_typecanvas() 70c.

8.5.2 Drawing

⟨*function* pl_drawcanvas 70a⟩≡ (139b)

```
void pl_drawcanvas(Panel *p){
    Canvas *c = p->data;

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

8.5.3 Reacting

⟨*function* pl_hitcanvas 70b⟩≡ (139b)

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

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

⟨*function* pl_typecanvas 70c⟩≡ (139b)

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

8.5.4 Packing methods

⟨*function* pl_getsizecanvas 70d⟩≡ (139b)

```
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 70e⟩≡ (139b)

```
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.

The container vocabulary varies considerably across toolkits, and the differences are a good lens for `libpanel`'s minimalism. The same intent—"put these children in a row, in a column, in a labelled box, in a notebook, in a grid"—is expressed through very different sets of widgets:

intent	libpanel	GTK 3	Qt
row of widgets	plframe + PACKW or PACKE	GtkBox(HORIZONTAL)	QHBoxLayout in QFrame
column of widgets	plframe + PACKN	GtkBox(VERTICAL)	QVBoxLayout in QFrame
no border	plgroup	GtkBox (no border)	QWidget + layout
labelled border	--- (build with pllabel + plframe)	GtkFrame	QGroupBox
2D grid	--- (nest two frames)	GtkGrid	QGridLayout
tabs	--- (build with pulldown + show/hide)	GtkNotebook	QTabWidget
freeform position	--- (use FIXED + pad)	GtkFixed	QWidget at absolute coords
resizable split	--- (manual)	GtkPaned	QSplitter

The absences in the `libpanel` column are deliberate. There are only *two* containers in the whole library—visible (`plframe`) and invisible (`plgroup`)—and direction is a property of the children, not the parent. The

result is that adding a new container kind is a non-event: a vertical menu and a horizontal menu bar use the same `plgroup`, differing only in whether their children carry `PACKN` or `PACKW`. GTK needs `GtkVBox` and `GtkHBox` (and now `GtkBox` with an orientation property because the split was a mistake), Qt needs `QHBoxLayout` and `QVBoxLayout`, etc. The `libpanel` approach scales worse to genuinely 2D layouts (no `GtkGrid` equivalent), but it scales better to "one more menu, in the other direction": zero new lines of library code.

9.1.1 Initializing

⟨*function* `plframe` 72a⟩≡ (139c)

```
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()` 20a and `plinitframe()` 72b.

⟨*function* `plinitframe` 72b⟩≡ (139c)

```
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()` 73g, `pl_drawframe()` 72c, `pl_getsizeframe()` 73e, `pl_hitframe()` 73c, and `pl_typeframe()` 73d.

9.1.2 Drawing

⟨*function* `pl_drawframe` 72c⟩≡ (139c)

```
void pl_drawframe(Panel *p){
    pl_box(p->b, p->r, FRAME);
}
```

Uses `FRAME` and `pl_box()` 32b.

⟨`Style` *other cases* 72d⟩+≡ (23a) ◁50c 92a▷

```
// 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* 73a⟩+≡ (32c) ◁55a
```
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 73b, SPACE-16 32d, pl_black-26 28c, pl_light-24 28c, pl_relief() 31, and pl_white-23 28c.

⟨*constant* FWID 73b⟩≡ (135c)
```
#define FWID 2 /* width of frame relief */
```

9.1.3 Reacting

⟨*function* pl_hitframe 73c⟩≡ (139c)
```
bool pl_hitframe(Panel *p, Mouse *m){
    USED(p, m);
    return false;
}
```

⟨*function* pl_typeframe 73d⟩≡ (139c)
```
void pl_typeframe(Panel *p, Rune c){
    USED(p, c);
}
```

9.1.4 Packing methods

⟨*function* pl_getsizeframe 73e⟩≡ (139c)
```
Vector pl_getsizeframe(Panel *p, Vector children){
    USED(p);
    return pl_boxsize(children, FRAME);
}
```
Uses FRAME and pl_boxsize() 32e.

⟨pl_boxsize() *switch state cases* 73f⟩+≡ (32e) ◁58a
```
case FRAME:
    return addpt(interior, Pt(4*FWID+2*SPACE, 4*FWID+2*SPACE));
```
Uses FRAME, FWID-15 73b, and SPACE-16 32d.

⟨*function* pl_childspaceframe 73g⟩≡ (139c)
```
void pl_childspaceframe(Panel *p, Point *ul, Vector *size){
    USED(p);
    pl_interior(FRAME, ul, size);
}
```
Uses FRAME and pl_interior() 32f.

⟨pl_interior() *switch state cases* 73h⟩+≡ (32f) ◁33b
```
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 73b, and SPACE-16 32d.

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` 74a⟩≡ (139d)
```
Panel *plgroup(Panel *parent, int flags){
    Panel *p;

    p=pl_newpanel(parent, 0);
    plinitgroup(p, flags);
    return p;
}
```
Uses `pl_newpanel()` 20a and `plinitgroup()` 74b.

⟨*function* `plinitgroup` 74b⟩≡ (139d)
```
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()` 75b, `pl_drawgroup()` 74c, `pl_getsizegroup()` 75a, `pl_hitgroup()` 74d, and `pl_typegroup()` 74e.

9.2.2 Drawing

⟨*function* `pl_drawgroup` 74c⟩≡ (139d)
```
void pl_drawgroup(Panel *p){
    USED(p);
}
```

9.2.3 Reacting

⟨*function* `pl_hitgroup` 74d⟩≡ (139d)
```
bool pl_hitgroup(Panel *p, Mouse *m){
    USED(p, m);
    return false;
}
```

⟨*function* `pl_typegroup` 74e⟩≡ (139d)
```
void pl_typegroup(Panel *p, Rune c){
    USED(p, c);
}
```

9.2.4 Packing methods

⟨*function* `pl_getsizegroup` 75a⟩≡ (139d)

```
Vector pl_getsizegroup(Panel *p, Vector children){
    USED(p);
    return children;
}
```

⟨*function* `pl_childspacegroup` 75b⟩≡ (139d)

```
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()`[77a], which cross-links the two widgets via pointer fields in `Panel`[19]. 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* 76a⟩+≡ (19) ◁35b 77b▷

```
// option<ref<Panel>>
Panel *scrollee;    /* pointer to scrolled window */
// option<ref<Panel>>
Panel *xscroller, *yscroller;   /* pointers to scroll bars */
```

⟨`pl_newpanel()` *set other fields* 76b⟩+≡ (20a) ◁41d 77d▷

```
v->scrollee=nil;
v->xscroller=nil;
v->yscroller=nil;
```

`plscroll()`[77a] 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` 77a⟩≡ (145c)
```
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* 77b⟩+≡ (19) ◁76a
```
Scroll scr;     /* scroll data */
```

⟨*struct* `Scroll` 77c⟩≡ (129d)
```
struct Scroll{
    Point pos;
    Vector size;
};
```

⟨`pl_newpanel()` *set other fields* 77d⟩+≡ (20a) ◁76b
```
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* 77e⟩+≡ (19) ◁37d 117a▷
```
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* 77f⟩+≡ (20a) ◁46d 117b▷
```
v->scroll=pl_scrollerror;
v->setscrollbar=pl_setscrollbarerror;
```
Uses `pl_scrollerror()` 127a and `pl_setscrollbarerror()` 127b.

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` 77g⟩≡ (140a)
```
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 78a⟩

    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* 78a⟩≡ (77g) 79c▷
```
buttons buttons;
```

10.3.1 Initializing

⟨*function* pllist 78b⟩≡ (140a)
```
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() 20a and plinitlist() 78c.

⟨*function* plinitlist 78c⟩≡ (140a)
```
void plinitlist(Panel *v, int flags, char *(*gen)(Panel *, int), int nlist, void (*hit)(Panel *, int, int)){
    List *lp = v->data;
    ⟨plinitlist() other locals 78e⟩

    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 78d⟩
    ⟨plinitlist() set scrolling fields 82a⟩

    v->kind="list";
}
```
Uses LEAF, UP 23a, pl_childspacelist() 81c, pl_drawlist() 79d, pl_getsizelist() 81b, pl_hitlist() 80f, and pl_typelist() 81a.

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 78d⟩≡ (78c) 78f▷
```
lp->gen=gen;
lp->hit=hit;
```

⟨plinitlist() *other locals* 78e⟩≡ (78c)
```
int wid, max;
char *str;
```

⟨plinitlist() *set fields in* lp 78f⟩+≡ (78c) ◁78d 79b▷
```
⟨plinitlist() if FILLX or EXPAND 79a⟩
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 79a⟩≡ (78f)
```
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 79b⟩+≡ (78c) ◁78f
```
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()[51b]. The selected item, if visible, gets a translucent highlight overlay using pl_highlight()[80b].

⟨List *other fields* 79c⟩+≡ (77g) ◁78a
```
Rectangle listr;
```

⟨*function* pl_drawlist 79d⟩≡ (140a)
```
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 23a, pl_box() 32b, and pl_liststrings() 79e.

⟨*function* pl_liststrings 79e⟩≡ (140a)
```
void pl_liststrings(Panel *p, int lo, int hi, Rectangle r){
    List *lp = p->data;
    char *s;
    int i;
    ⟨pl_liststrings() other locals 80d⟩

    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 79f⟩
    ⟨pl_liststrings() set the scrollbar 80e⟩
}
```
Uses pl_drawicon() 51b.

⟨pl_liststrings() *draw the selection* 79f⟩≡ (79e)
```
if(lo<=lp->sel && lp->sel<hi)
    pl_listsel(p, lp->sel, true);
```
Uses pl_listsel() 80a.

⟨*function* `pl_listsel` 80a⟩≡ (140a)
```
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()` 51b, `pl_fill()` 80c, and `pl_highlight()` 80b.

⟨*function* `pl_highlight` 80b⟩≡ (135c)
```
void pl_highlight(Image *b, Rectangle r){
    draw(b, r, pl_dark, pl_hilit, ZP);
}
```
Uses `pl_dark-25` 28c and `pl_hilit-27` 28c.

⟨*function* `pl_fill` 80c⟩≡ (135c)
```
void pl_fill(Image *b, Rectangle r){
    draw(b, r, plldepth==0? pl_white : pl_light, 0, ZP);
}
```
Uses `pl_light-24` 28c, `pl_white-23` 28c, and `plldepth-22` 28b.

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* 80d⟩≡ (79e)
```
Panel *sb = p->yscroller;
```

⟨`pl_liststrings()` *set the scrollbar* 80e⟩≡ (79e)
```
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` 80f⟩≡ (140a)
```
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 50c, UP 23a, pl_interior() 32f, and pl_listsel() 80a.

⟨*function* pl_typelist 81a⟩≡ (140a)

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

10.3.4 Packing methods

⟨*function* pl_getsizelist 81b⟩≡ (140a)

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

Uses pl_boxsize() 32e.

⟨*function* pl_childspacelist 81c⟩≡ (140a)

```
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()`[82b]. 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()`[83c] (a `draw()`-based blit) and only redraws the newly exposed strip, avoiding a full repaint.

⟨`plinitlist()` *set scrolling fields* 82a⟩≡ (78c)
```
v->scroll=pl_scrolllist;
v->scr.pos=Pt(0,0);
v->scr.size=Pt(0,lp->len);
```
Uses `pl_scrolllist()` 82b.

⟨*function* `pl_scrolllist` 82b⟩≡ (140a)
```
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 83a⟩
    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 83b⟩
}
```
Uses `PASSIVE` 50c, `VERT`, `pl_box()` 32b, `pl_interior()` 32f, and `pl_liststrings()` 79e.

⟨pl_scrolllist() *sanitize* lp->lo 83a⟩≡ (82b)
```
if(lp->lo<0) lp->lo=0;
if(lp->lo>=lp->len) lp->lo=lp->len-1;
```

⟨pl_scrolllist() *optimized drawing by copying* 83b⟩≡ (82b)
```
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 50c, pl_box() 32b, pl_cpy() 83c, and pl_liststrings() 79e.

⟨*function* pl_cpy 83c⟩≡ (135c)
```
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() 66b—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 83d⟩≡ (145d)
```
struct Scrollbar{
    // enum<Direction>
    int dir;  /* HORIZ or VERT */
    Point minsize;

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

    ⟨Scrollbar other fields 83e⟩
};
```

⟨Scrollbar *other fields* 83e⟩≡ (83d) 85a▷
```
buttons buttons;  /* saved mouse buttons for transmittal to scrollee */
```

10.4.1 Initializing

⟨*function* plscrollbar 83f⟩≡ (145d)
```
Panel *plscrollbar(Panel *parent, int flags){
    Panel *v;

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

```
    plinitscrollbar(v, flags);
    return v;
}
```
Uses pl_newpanel() 20a and plinitscrollbar() 84a.

⟨*function* plinitscrollbar 84a⟩≡ (145d)
```
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 84b⟩

    v->kind="scrollbar";
}
```
Uses LEAF, UP 23a, pl_childspacescrollbar() 87c, pl_drawscrollbar() 85b, pl_getsizescrollbar() 87b, pl_hitscrollbar() 85c, and pl_typescrollbar() 87a.

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* 84b⟩≡ (84a) 84c▷
```
v->pri=pl_priscrollbar; // !!
```
Uses pl_priscrollbar() 37k.

⟨plinitscrollbar() *set extra fields* 84c⟩+≡ (84a) ◁84b 84e▷
```
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 133c, SBWID-2 84d, and VERT.

⟨*constant* SBWID 84d⟩≡ (145d)
```
#define SBWID 15 /* should come from draw.c? */
```

⟨plinitscrollbar() *set extra fields* 84e⟩+≡ (84a) ◁84c 87d▷
```
sp->lo=0;
sp->hi=0;
```

10.4.2 Drawing

⟨Scrollbar *other fields* 85a⟩+≡ (83d) ◁83e
```
  Rectangle interior;
```

⟨*function* pl_drawscrollbar 85b⟩≡ (145d)
```
  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() 32a and pl_sliderupd() 66b.

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()`[66b]) 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 85c⟩≡ (145d)
```
  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 86a⟩

      if(m->buttons&7){
          g->state=DOWN;
          sp->buttons=m->buttons;
          switch(m->buttons){
          ⟨pl_hitscrollbar() switch buttons cases 86b⟩
          }
```

```
    }
    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 50c, HORIZ 133c, UP 23a, pl_interior() 32f, and pldraw() 30a.

⟨pl_hitscrollbar() *sanitize* pos 86a⟩≡ (85c)

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

The trough-to-logical mapping is a single linear transform. The scrollbar's pixel track is [0, len] (the interior height in pixels after stripping the relief border), and the current thumb spans the *pixel* range [sp->lo, sp->hi]. Left-click at pixel pos scrolls *up* by a fraction of the thumb span; right-click scrolls down by the same amount:

```
pixel space (what the user sees)
     0                                      len
     +--------------------------------------+
     |......[==== thumb ====].............|
            ^                ^
            sp->lo           sp->hi
                   ^
                   pos (click)

dy = pos * (sp->hi - sp->lo) / len
   = (click fraction) * (thumb span)

left click:  preview = [sp->lo - dy, sp->hi - dy]  (scroll up)
right click: preview = [sp->lo + dy, sp->hi + dy]  (scroll down)
middle click: absolute jump -> pos/len directly
```

The scrollee later re-interprets sp->lo and sp->hi in its own logical units (byte offsets for a text view, item indices for a list), but the scrollbar itself never knows: its only job is this pixel arithmetic plus a redraw.

⟨pl_hitscrollbar() *switch buttons cases* 86b⟩≡ (85c)

```
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() 66b.

⟨*function* pl_typescrollbar 87a⟩≡ (145d)

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

10.4.4 Packing methods

⟨*function* pl_getsizescrollbar 87b⟩≡ (145d)

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

Uses pl_boxsize() 32e.

⟨*function* pl_childspacescrollbar 87c⟩≡ (145d)

```
  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* 87d⟩+≡ (84a) ◁84e

```
  v->setscrollbar=pl_setscrollbarscrollbar;
```

Uses pl_setscrollbarscrollbar() 87e.

⟨*function* pl_setscrollbarscrollbar 87e⟩≡ (145d)

```
  /*
   * 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 88a⟩
      ⟨pl_setscrollbarscrollbar() sanitize sp->hi 88b⟩
      pldraw(p, p->b);
  }
```

Uses HORIZ 133c, pl_interior() 32f, and pldraw() 30a.

⟨pl_setscrollbarscrollbar() *sanitize* sp->lo 88a⟩≡ (87e)

```
if(sp->lo<0) sp->lo=0;
if(sp->lo>=mylen) sp->hi=mylen-1;
```

⟨pl_setscrollbarscrollbar() *sanitize* sp->hi 88b⟩≡ (87e)

```
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* 89a⟩+≡ (58d) ◁59b
```
int index;   /* arg to menuhit */
```

⟨`Button` *other methods* 89b⟩≡ (58d)
```
void (*menuhit)(buttons, int); /* call back user code on menu item hit */
```

11.1.1 Initializing

`plmenu()`[89c] creates a group and populates it with buttons, one per item in the null-terminated `item` array. Each button uses `pl_hitmenu()`[90b] 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` 89c⟩≡ (137a)
```
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()` 74a and `plinitmenu()` 89d.

⟨*function* `plinitmenu` 89d⟩≡ (137a)
```
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 90a⟩

    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() 90b and plbutton() 59d.

⟨plinitmenu() *free child widget if any* 90a⟩≡ (89d)

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

Uses plfree() 20c.

11.1.2 Drawing

11.1.3 Reacting

⟨*function* pl_hitmenu 90b⟩≡ (137a)

```
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 90c⟩≡ (141a)

```
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 91a⟩≡ (141a)

```
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() 20a and plinitpopup() 91b.

⟨*function* plinitpopup 91b⟩≡ (141a)

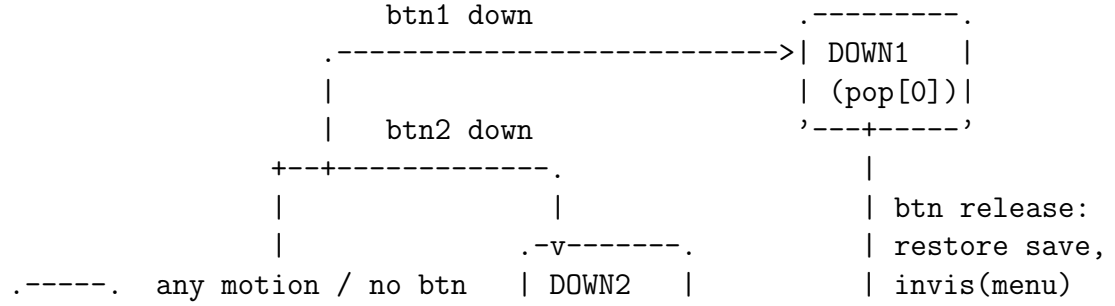
```
void plinitpopup(Panel *v, int flags, Panel *pop0, Panel *pop1, Panel *pop2){
    Popup *pp =v->data;

    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";
}
```

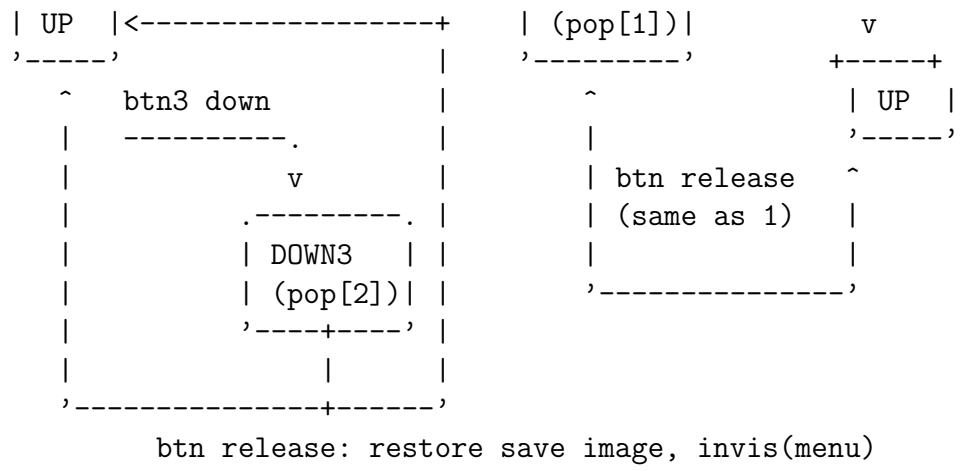
Uses UP 23a, pl_childspacepopup() 95a, pl_drawpopup() 93e, pl_getsizepopup() 94g, pl_hitpopup() 92b, pl_pripopup() 37j, and pl_typepopup() 93d.

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()[93h]), 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()[35a] call). When the button is released, the saved image is restored, hiding the menu cleanly.

Drawn out, the four states and their transitions look like this:





Three things make this state machine work in only one `hit` call. (1) The state *is* the answer to "which menu do I forward to?"—no separate "current menu" field is needed because `DOWN1`/`DOWN2`/`DOWN3` already encode it. (2) The save-and-restore is symmetric on the `UP`→`DOWNX` edge (allocate save image, blit screen into it, draw menu) and the `DOWNX`→`UP` edge (blit save image back, free it), so the popup leaves no residue when dismissed. (3) The menu panels in `pop[]` are not children of the popup widget—they live in `INVIS` limbo until needed, which is why the `INVIS` flag exists at all. Without it, the recursive `pl_drawall` would render the menus all the time, in the wrong place.

⟨`Style` *other cases* 92a⟩+≡ (23a) ◁72d
```
DOWN1,
DOWN2,
DOWN3,
```

⟨*function* `pl_hitpopup` 92b⟩≡ (141a)
```
bool pl_hitpopup(Panel *g, Mouse *m){
    Panel *p;
    Popup *pp = g->data;
    ⟨pl_hitpopup() other locals 93f⟩

    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 93a⟩
        }
        ⟨pl_hitpopup() when UP, if no popup menu p 93c⟩
        else if(g->state!=UP){
            ⟨pl_hitpopup() when from UP to DOWNX and valid p 93g⟩
        }
    } 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 94c⟩
    }
    // 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 50c, DOWN1 23a, DOWN2 23a, DOWN3, UP 23a, and plmouse() 35a.

⟨pl_hitpopup() *when* UP, *switch buttons other cases* 93a⟩≡ (92b) 93b▷

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

⟨pl_hitpopup() *when* UP, *switch buttons other cases* 93b⟩+≡ (92b) ◁93a

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

⟨pl_hitpopup() *when* UP, *if no popup menu* p 93c⟩≡ (92b)

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

Uses DOWN 50c.

⟨*function* pl_typepopup 93d⟩≡ (141a)

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

11.2.3 Drawing

⟨*function* pl_drawpopup 93e⟩≡ (141a)

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

⟨pl_hitpopup() *other locals* 93f⟩≡ (92b)

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

⟨pl_hitpopup() *when from* UP *to* DOWNX *and valid* p 93g⟩≡ (92b)

```
plpack(p, view->clipr);
⟨pl_hitpopup() when from UP to DOWNX, compute d 94a⟩
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() 94d, pldraw() 30a, plmove() 93h, and plpack() 40.

⟨*function* plmove 93h⟩≡ (140c)

```
/*
 * 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 94b⟩
    p->r=rectaddpt(p->r, d);
    for(p=p->child;p;p=p->next)
        plmove(p, d);
}
```

Uses plmove() 93h.

⟨pl_hitpopup() *when from* UP *to* DOWNX, *compute* d 94a⟩≡ (93g)
```
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* 94b⟩≡ (93h)
```
if(strcmp(p->kind, "edit") == 0) /* sorry */
    plemove(p, d);
```
Uses plemove() 108c.

Hiding

⟨pl_hitpopup() *when* DOWNX *state, if no buttons* 94c⟩≡ (92b)
```
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 50c, UP 23a, and pl_invis() 94d.

⟨*function* pl_invis 94d⟩≡ (135c)
```
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() 94d.

⟨*constant* INVIS 94e⟩≡ (133e)
```
#define INVIS 0x20000  /* don't draw this */
```

⟨pl_drawall() *if invisible widget* 94f⟩≡ (30c)
```
if(p->flags&INVIS) return;
```
Uses INVIS.

11.2.4 Packing methods

⟨*function* pl_getsizepopup 94g⟩≡ (141a)
```
Vector pl_getsizepopup(Panel *g, Vector children){
    USED(g);
    return children;
}
```

⟨*function* `pl_childspacepopup` 95a⟩≡ (141a)
```
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` 95b⟩≡ (141b)
```
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` 95c⟩≡ (141b)
```
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()` 20a and `plinitpulldown()` 95d.

⟨*function* `plinitpulldown` 95d⟩≡ (141b)
```
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 23a, pl_childspacepulldown() 98c, pl_drawpulldown() 96a, pl_getsizepulldown() 98b, pl_hitpulldown() 96b, and pl_typepulldown() 98a.

11.3.2 Drawing

⟨*function* pl_drawpulldown 96a⟩≡ (141b)

```
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() 32b and pl_drawicon() 51b.

11.3.3 Reacting

⟨*function* pl_hitpulldown 96b⟩≡ (141b)

```
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 97b⟩
        else
            if(m->buttons&7){
                r=g->b->r;
                p->flags&=~PLACE;
                switch(pp->side){
                ⟨pl_hitpulldown() when UP and buttons, switch side cases 97a⟩
                }
                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 50c, UP 23a, pl_invis() 94d, pldraw() 30a, plmouse() 35a, and plpack() 40.

⟨pl_hitpulldown() *when* UP *and buttons, switch side cases* 97a⟩≡ (96b)

```
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* 97b⟩≡ (96b)

```
if(!ptinrect(m->xy, g->r))
    g->state=UP;
```

Uses UP 23a.

⟨*function* `pl_typepulldown` 98a⟩≡ (141b)
```
void pl_typepulldown(Panel *p, Rune c){
    USED(p, c);
}
```

11.3.4 Packing methods

⟨*function* `pl_getsizepulldown` 98b⟩≡ (141b)
```
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()` 32e and `pl_iconsize()` 51c.

⟨*function* `pl_childspacepulldown` 98c⟩≡ (141b)
```
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()`[98d] 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` 98d⟩≡ (141b)
```
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 98e⟩
    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()` 74a, `plpulldown()` 95c, and `s`.

⟨`plmenubar()` *set* `pulldir` *based on* `cflags` 98e⟩≡ (98d)
```
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` 100a⟩≡ (140b)
```
struct Message{
    // ref_own<string>
    char *text;

    Vector minsize;
};
```

12.1.1 Initializing

⟨*function* `plmessage` 100b⟩≡ (140b)
```
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()` 20a and `plinitmessage()` 100c.

⟨*function* `plinitmessage` 100c⟩≡ (140b)
```
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() 102e, pl_drawmessage() 101a, pl_getsizemessage() 102c, pl_hitmessage() 102a, and pl_typemessage() 102b.

12.1.2 Drawing

The drawing function pl_textmsg()[101b] 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 101a⟩≡ (140b)

```
void pl_drawmessage(Panel *p){
    pl_textmsg(p->b, pl_box(p->b, p->r, PASSIVE), font,
               ((Message *)p->data)->text);
}
```

Uses PASSIVE 50c, pl_box() 32b, and pl_textmsg() 101b.

⟨*function* pl_textmsg 101b⟩≡ (140b)

```
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() 158a and pl_runewidth() 158b.

12.1.3 Reacting

⟨*function* pl_hitmessage 102a⟩≡ (140b)
```
bool pl_hitmessage(Panel *g, Mouse *m){
    USED(g, m);
    return false;
}
```

⟨*function* pl_typemessage 102b⟩≡ (140b)
```
void pl_typemessage(Panel *g, Rune c){
    USED(g, c);
}
```

12.1.4 Packing methods

⟨*function* pl_getsizemessage 102c⟩≡ (140b)
```
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 50c, pl_boxsize() 32e, and pl_foldsize() 102d.

⟨*function* pl_foldsize 102d⟩≡ (140b)
```
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() 158a and pl_runewidth() 158b.

⟨*function* pl_childspacemessage 102e⟩≡ (140b)
```
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` 103a⟩≡ (138)
```
struct Edit{
    Point minsize;
    void (*hit)(Panel *);
    int sel0, sel1;
    Textwin *t;
    Rune *text;
    int ntext;
};
```

12.2.1 Initializing

⟨*function* `pledit` 103b⟩≡ (138)
```
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()` 20a and `plinitedit()` 103c.

⟨*function* `plinitedit` 103c⟩≡ (138)
```
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 23a, pl_childspaceedit() 106b, pl_drawedit() 104b, pl_freeedit() 104a, pl_getsizeedit() 106a, pl_hitedit() 104c, pl_pasteedit() 119c, pl_scrolledit() 106c, pl_snarfedit() 119b, pl_typeedit() 105, and twfree() 156b.

⟨*function* pl_freeedit 104a⟩≡ (138)

```
void pl_freeedit(Panel *p){
    Edit *ep;
    ep=p->data;
    if(ep->t) twfree(ep->t);
    ep->t=0;
}
```

Uses twfree() 156b.

12.2.2 Drawing

⟨*function* pl_drawedit 104b⟩≡ (138)

```
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() 151b, twnew() 156a, and twreshape() 155b.

12.2.3 Reacting

⟨*function* pl_hitedit 104c⟩≡ (138)

```
/*
 * 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() 108b, plgrabkb() 38b, plpaste() 118a, plsnarf() 117e, twhilite() 151b, and twselect() 151c.

⟨*function* pl_typeedit 105⟩≡ (138)

```
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() 107c, plepaste() 108b, plsnarf() 117e, twhilite() 151b, twpt2rune() 148c, twreplace() 154b, and twscroll() 155a.

12.2.4 Packing methods

⟨*function* pl_getsizeedit 106a⟩≡ (138)

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

Uses pl_boxsize() 32e.

⟨*function* pl_childspaceedit 106b⟩≡ (138)

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

12.2.5 Other methods

⟨*function* pl_scrolledit 106c⟩≡ (138)

```
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() 151b, twpt2rune() 148c, and twscroll() 155a.

⟨*function* plescroll 107a⟩≡ (138)

```
void plescroll(Panel *p, int top){
    Textwin *t;
    t=((Edit*)p->data)->t;
    if(t) twscroll(t, top);
}
```

Uses twscroll() 155a.

⟨*function* plegetsel 107b⟩≡ (138)

```
void plegetsel(Panel *p, int *sel0, int *sel1){
    Edit *ep;
    ep=p->data;
    *sel0=ep->sel0;
    *sel1=ep->sel1;
}
```

⟨*function* plelen 107c⟩≡ (138)

```
int plelen(Panel *p){
    Textwin *t;
    t=((Edit*)p->data)->t;
    if(t==0) return 0;
    return t->etext-t->text;
}
```

⟨*function* pleget 107d⟩≡ (138)

```
Rune *pleget(Panel *p){
    Textwin *t;
    t=((Edit*)p->data)->t;
    if(t==0) return 0;
    return t->text;
}
```

⟨*function* `plesel` 108a⟩≡ (138)
```
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()` 151b.

⟨*function* `plepaste` 108b⟩≡ (138)
```
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()` 151b and `twreplace()` 154b.

⟨*function* `plemove` 108c⟩≡ (138)
```
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()` 156c.

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()`[142d]), 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` 108d⟩≡ (148a)
```
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 109a⟩≡ (148a)
```
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() 20a and plinittextview() 109b.

⟨*function* plinittextview 109b⟩≡ (148a)
```
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 23a, pl_childspacetextview() 111b, pl_drawtextview() 109c, pl_getsizetextview() 111a, pl_hittextview() 110a, pl_pritextview() 147a, pl_scrolltextview() 146d, pl_snarftextview() 147b, and pl_typetextview() 110b.

12.3.2 Drawing

⟨*function* pl_drawtextview 109c⟩≡ (148a)
```
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 23a, pl_outline() 32a, pl_rtdraw() 113g, pl_rtfmt() 142d, and pl_setscrpos() 146b.

12.3.3 Reacting

⟨*function* pl_hittextview 110a⟩≡ (148a)

```
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 50c, REMOUSE, UP 23a, pl_drawtextview() 109c, pl_interior() 32f, pl_passon() 146c, pl_rthit() 115a, and plrtseltext() 115b.

⟨*function* pl_typetextview 110b⟩≡ (148a)

```
void pl_typetextview(Panel *g, Rune c){
```

```
    USED(g, c);
}
```

12.3.4 Packing methods

⟨*function* `pl_getsizetextview` 111a⟩≡ (148a)

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

Uses `pl_boxsize()` 32e.

⟨*function* `pl_childspacetextview` 111b⟩≡ (148a)

```
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()`[113b], `plrtbitmap()`[113c], and `plrtpanel()`[113a]—create nodes of each type and append them to a list.

A worked example clarifies how a rendered HTML fragment becomes a chain of `Rtext` nodes. Suppose `mothra` is asked to display the very simple page:

```
<p>Click <b>here</b> for the <img src="logo.png"> manual.</p>
```

After parsing, the text view would hold something like this:

```
text -> [Rtext "Click"]            font=roman flags=0
         |
         v space=4 indent=0
        [Rtext "here"]             font=bold  flags=PL_HOT
         |
         v space=4 indent=0
        [Rtext "for the"]          font=roman flags=0
         |
         v space=4 indent=0
        [Rtext (logo.png)]         b=Image*   flags=0
         |
         v space=4 indent=0
        [Rtext "manual."]          font=roman flags=0
         |
```

```
  v
nil
```

A few details fall out of looking at this list. (1) A *font change* between two adjacent words is just a new node with a different `font` field; the surrounding nodes do not need to know about it, because the layout pass (`pl_rtfmt()`[142d]) walks the chain and asks each node for its own width. (2) An *image* is structurally identical to a piece of text—same struct, just `b` non-nil instead of `text`—which is what lets `mothra` mix bitmaps and prose inline without a separate widget hierarchy. (3) A *hyperlink* is the `PL_HOT` flag plus the `user` field holding the URL; the underline drawn under the hot word in `pl_rtdraw()`[113g] is the only visual cue. (4) The `space` field is what allows "manual." (a noun glued to a period) to stay together: setting `space=0` before the period would make `pl_rtfmt()` treat the two as one word for line-breaking purposes. The same trick is what lets `mothra` render "`plinitlabel()`" as a single token even though the parentheses are in a different font.

⟨*struct* `Rtext` 112a⟩≡ (129d)

```
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` 112b⟩≡ (144c)

```
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()` 128a.

⟨*function* plrtpanel 113a⟩≡ (144c)
```
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() 112b.

⟨*function* plrtstr 113b⟩≡ (144c)
```
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() 112b.

⟨*function* plrtbitmap 113c⟩≡ (144c)
```
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() 112b.

⟨*function* plrtfree 113d⟩≡ (144c)
```
void plrtfree(Rtext *t){
    Rtext *next;
    while(t){
        next=t->next;
        free(t);
        t=next;
    }
}
```

12.5.2 Drawing

⟨*constant* PL_HOT 113e⟩≡ (129d)
```
#define PL_HOT  1
```

⟨*constant* PL_SEL 113f⟩≡ (129d)
```
#define PL_SEL  2
```

⟨*function* pl_rtdraw 113g⟩≡ (144c)
```
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 142a, pl_clr() 114a, pl_highlight() 80b, pl_stuffbitmap() 143, pldraw() 30a, and plmove() 93h.

⟨*function* pl_clr 114a⟩≡ (135c)

```
void pl_clr(Image *b, Rectangle r){
    draw(b, r, display->white, 0, ZP);
}
```

⟨*function* pl_rtredraw 114b⟩≡ (144c)

```
/*
 * 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()` 144a and `pl_rtdraw()` 113g.

12.5.3 Reacting

⟨*function* `pl_rthit` 115a⟩≡ (144c)

```
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` 115b⟩≡ (144c)

```
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`[19] 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`.

The clipboard is one of the few user-level IPC primitives every GUI has, and the implementations differ wildly in complexity:

system	interface	paid complexity
\plan	read/write /dev/snarf	none; it is a file
X11	ICCCM selection protocol (PRIMARY, CLIPBOARD, SECONDARY) + targets	enormous: clients negotiate format, own the selection, respond to events
Wayland	wl_data_device protocol	simpler than X11 but still client/server negotiation
macOS	NSPasteboard / UIPasteboard	multi-format, promised content, universal clipboard
Windows	Open/Close/SetClipboardData + format negotiation	multi-format with delayed rendering
Web	navigator.clipboard + Clipboard API + events	async, requires user gesture, MIME
Android	ClipboardManager / ClipData	MIME items, drag/drop unification

Plan 9's choice has two virtues that fall out of treating the clipboard as a regular file. First, *any* program can participate—a shell script can do `echo hello world > /dev/snarf` to put text on the clipboard, or `cat /dev/snarf` to read it; no GUI library required. Second, there is no protocol to lose: the clipboard has only one format (UTF-8 text), so there is nothing to negotiate. The cost is exactly that single format: Plan 9 cannot copy a styled range, an image, or a file reference, the way every other system in the table can. For `libpanel`'s

use cases (text entries, code, web page snippets), text is enough; for a modern desktop where users expect to copy a styled fragment from a browser into a word processor, it would not be.

⟨Panel *other methods* 117a⟩+≡ (19) ◁77e
```
char* (*snarf)(Panel *);   /* snarf text from panel */
void (*paste)(Panel *, char *);   /* paste text into panel */
```

⟨pl_newpanel() *set default methods* 117b⟩+≡ (20a) ◁77f
```
v->snarf=nil;
v->paste=nil;
```

⟨*function* plputsnarf 117c⟩≡ (146a)
```
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 117d⟩≡ (146a)
```
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() 128b.

⟨*function* plsnarf 117e⟩≡ (146a)
```
void plsnarf(Panel *p){
    char *s;

    if(p==0 || p->snarf==0)
        return;
    s=p->snarf(p);
    plputsnarf(s);
    free(s);
}
```
Uses plputsnarf() 117c.

⟨*function* `plpaste` 118a⟩≡ (146a)
```
void plpaste(Panel *p){
    char *s;

    if(p==0 || p->paste==0)
        return;
    if(s=plgetsnarf()){
        p->paste(p, s);
        free(s);
    }
}
```
Uses `plgetsnarf()` 117d.

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* 118b⟩≡ (53c)
```
v->snarf=pl_snarfentry;
v->paste=pl_pasteentry;
```
Uses `pl_pasteentry()` 118d and `pl_snarfentry()` 118c.

⟨*function* `pl_snarfentry` 118c⟩≡ (137b)
```
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` 118d⟩≡ (137b)
```
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` 118e, `pl_erealloc()` 128b, and `pldraw()` 30a.

⟨*constant* `SLACK` 118e⟩≡ (137b)
```
#define SLACK 7 /* enough for one extra rune and < and a nul */
```

⟨pl_hitentry() *handle copy/paste when middle or right click* 119a⟩≡ (55e)

```
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() 30a, plpaste() 118a, and plsnarf() 117e.

⟨*function* pl_snarfedit 119b⟩≡ (138)

```
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() 107d and plegetsel() 107b.

⟨*function* pl_pasteedit 119c⟩≡ (138)

```
void pl_pasteedit(Panel *p, char *s){
    Rune *t;
    if(t=runesmprint("%s", s)){
        plepaste(p, t, runestrlen(t));
        free(t);
    }
}
```

Uses plepaste() 108b.

13.2 Hooks

⟨Panel *user-specific fields* 119d⟩≡ (19)

```
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()[40] 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 120a⟩≡ (129d)
```
#define FIXEDX 0x0400
```

⟨*constant* FIXEDY 120b⟩≡ (129d)
```
#define FIXEDY 0x0800
```

⟨Panel *fixed size field* 120c⟩≡ (19)
```
Point fixedsize;    /* size of Panel, if FIXED */
```

⟨pl_sizereq() *if* FIXEDX *or* FIXEDY 120d⟩≡ (42a)
```
if(p->flags&FIXEDX) p->sizereq.x=p->fixedsize.x;
if(p->flags&FIXEDY) p->sizereq.y=p->fixedsize.y;
```

⟨*constant* FIXED 120e⟩≡ (129d)
```
#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()[42a]: the first pass records the maximum size request, and the second pass overwrites each flagged widget's request with that maximum.

⟨*constant* MAXX 120f⟩≡ (129d)
```
#define MAXX 0x1000  /* make x size as big as biggest sibling's */
```

⟨*constant* MAXY 120g⟩≡ (129d)
```
#define MAXY 0x2000  /* make y size as big as biggest sibling's */
```

⟨pl_sizeref() *other locals* 120h⟩≡ (42a)
```
Vector maxsize = Pt(0,0);
```

⟨pl_sizeref() *when looping over children, adjust* maxsize 120i⟩≡ (42a)
```
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 120j⟩≡ (42a)
```
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 auto-refresh on mutation. Retained-mode toolkits like GTK, Qt, and WPF instead watch the widget tree via signals, observers, or reactive bindings—they pay an ongoing cost in framework complexity for the convenience of "just works." `libpanel`'s design is simpler to implement and more predictable, at the price of putting the responsibility on the application programmer. See Section 2.1.5 for the broader retained-versus-immediate-mode contrast.

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`[19] 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* 124a⟩≡ (19)
```
char *kind;     /* what kind of panel? */
```

`pl_print`[124b] is the entry point for dumping a widget tree. `pl_ipprint`[124d] 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` 124b⟩≡ (136b)
```
void pl_print(Panel *p){
    pl_ipprint(p, 0);
}
```
Uses `pl_ipprint()` 124d.

⟨*function* `pl_iprint` 124c⟩≡ (136b)
```
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` 124d⟩≡ (136b)
```
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()` 124d and `pl_iprint()` 124c.

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` 126a⟩≡ (135b)
```
void pl_unexpected(Panel *g, char *rou){
    fprint(STDERR, "%s called unexpectedly (%s %lux)\n",
           rou, g->kind, (ulong)g);
    abort();
}
```

⟨*function* `pl_drawerror` 126b⟩≡ (135b)
```
void pl_drawerror(Panel *g){
    pl_unexpected(g, "draw");
}
```
Uses `pl_unexpected()` 126a.

⟨*function* `pl_hiterror` 126c⟩≡ (135b)
```
int pl_hiterror(Panel *g, Mouse *m){
    USED(m);
    pl_unexpected(g, "hit");
    return 0;
}
```
Uses `pl_unexpected()` 126a.

⟨*function* `pl_typeerror` 126d⟩≡ (135b)
```
void pl_typeerror(Panel *g, Rune c){
    USED(c);
    pl_unexpected(g, "type");
}
```
Uses `pl_unexpected()` 126a.

⟨*function* `pl_getsizeerror` 126e⟩≡ (135b)
```
Point pl_getsizeerror(Panel *g, Point childsize){
    pl_unexpected(g, "getsize");
    return childsize;
}
```
Uses `pl_unexpected()` 126a.

⟨*function* `pl_childspaceerror` 126f⟩≡ (135b)
```
void pl_childspaceerror(Panel *g, Point *ul, Vector *size){
    USED(ul, size);
    pl_unexpected(g, "childspace");
}
```
Uses `pl_unexpected()` 126a.

⟨*function* `pl_scrollerror` 127a⟩≡ (135b)

```
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()` 126a.

⟨*function* `pl_setscrollbarerror` 127b⟩≡ (135b)

```
void pl_setscrollbarerror(Panel *g, int top, int bot, int den){
    USED(top, bot, den);
    pl_unexpected(g, "setscrollbar");
}
```

Uses `pl_unexpected()` 126a.

Appendix C

Utilities

C.1 Memory management

`pl_emalloc`[128a] and `pl_erealloc`[128b] 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` 128a⟩≡ (135b)

```
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` 128b⟩≡ (135b)

```
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 129a⟩≡

⟨lib_gui/libpanel/tests/scrltest.c 129b⟩≡

⟨lib_gui/libpanel/tests/panels_test.c 129c⟩≡

D.1.1 include/gui/panel.h

⟨include/gui/panel.h 129d⟩≡

```
#pragma src "/sys/src/libpanel"
#pragma lib "libpanel.a"

⟨type Icon 24a⟩

// 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 77c⟩
⟨struct Rtext 112a⟩
⟨struct Panel 19⟩

/*
 * Panel flags
 */
⟨constant PACK 25a⟩
⟨constant PACKN 25b⟩
⟨constant PACKE 25c⟩
⟨constant PACKS 25d⟩
⟨constant PACKW 25e⟩

⟨constant PACKCEN 25f⟩

⟨constant FILLX 26a⟩
⟨constant FILLY 26b⟩

⟨constant PLACE 24b⟩
⟨constant PLACECEN 24c⟩
```

⟨*constant* PLACES 24d⟩
⟨*constant* PLACEE 24e⟩
⟨*constant* PLACEW 24f⟩
⟨*constant* PLACEN 24g⟩
⟨*constant* PLACENE 24h⟩
⟨*constant* PLACENW 24i⟩
⟨*constant* PLACESE 24j⟩
⟨*constant* PLACESW 24k⟩

⟨*constant* EXPAND 26c⟩

⟨*constant* FIXED 120e⟩
⟨*constant* FIXEDX 120a⟩
⟨*constant* FIXEDY 120b⟩

⟨*constant* MAXX 120f⟩
⟨*constant* MAXY 120g⟩

⟨*constant* BITMAP 49b⟩
⟨*constant* USERFL 55b⟩

⟨*constant* OUT 36a⟩

```
/*
 * Priorities
 */
```
⟨*constant* PRI_NORMAL 37e⟩
⟨*constant* PRI_POPUP 37f⟩
⟨*constant* PRI_SCROLLBAR 37g⟩

```
/* Rtext.flags */
```
⟨*constant* PL_HOT 113e⟩
⟨*constant* PL_SEL 113f⟩

⟨*global* plkbfocus 38a⟩

```
//pad's stuff
```
⟨*constant* NOFLAG 20e⟩

```
// 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 132a⟩≡ (133b)
```
#define PL_NOPBIT 4
```

⟨*constant* PL_NARGBIT 132b⟩≡ (133b)
```
#define PL_NARGBIT 12
```

⟨*constant* PL_ARGMASK 132c⟩≡ (133b)
```
#define PL_ARGMASK ((1<<PL_NARGBIT)-1)
```

⟨*function* PL_SPECIAL 132d⟩≡ (133b)
```
#define PL_SPECIAL(op) (((-1<<PL_NOPBIT)|op)<<PL_NARGBIT)
```

⟨*function* PL_OP 132e⟩≡ (133b)
```
#define PL_OP(t) ((t)&~PL_ARGMASK)
```

⟨*function* PL_ARG 132f⟩≡ (133b)
```
#define PL_ARG(t) ((t)&PL_ARGMASK)
```

⟨*constant* PL_TAB 133a⟩≡ (133b)
```
#define PL_TAB  PL_SPECIAL(0)  /* # of tab stops before text */
```

⟨include/gui/rtext.h 133b⟩≡

```
/*
 * Rtext definitions
 */
⟨constant PL_NOPBIT 132a⟩
⟨constant PL_NARGBIT 132b⟩
⟨constant PL_ARGMASK 132c⟩
⟨function PL_SPECIAL 132d⟩

⟨function PL_OP 132e⟩
⟨function PL_ARG 132f⟩
⟨constant PL_TAB 133a⟩

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 133c⟩≡ (133e)
```
/*
 * Scroll flags
 */
enum{
    SCROLLUP,
    SCROLLDOWN,
    SCROLLABSY,
    SCROLLLEFT,
    SCROLLRIGHT,
    SCROLLABSX,
};
```

⟨*struct* Textwin 133d⟩≡ (133e)
```
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 133e⟩≡
```
/*
 * 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 22b⟩
⟨constant INVIS 94e⟩
⟨constant REMOUSE 37a⟩

⟨enum Style 23a⟩
⟨enum _anon_ (lib_gui/libpanel/pldefs.h)2 133c⟩
⟨enum Direction 27⟩

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 133d⟩

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 133e.

D.2.2 lib_gui/libpanel/init.c

⟨lib_gui/libpanel/init.c 135a⟩≡
⟨libpanel *includes* 15d⟩

⟨*function* plinit 28a⟩

D.2.3 lib_gui/libpanel/mem.c

⟨lib_gui/libpanel/mem.c 135b⟩≡
⟨libpanel *includes* 15d⟩

⟨*function* pl_emalloc 128a⟩
⟨*function* pl_erealloc 128b⟩

⟨*function* pl_unexpected 126a⟩
⟨*function* pl_drawerror 126b⟩
⟨*function* pl_hiterror 126c⟩
⟨*function* pl_typeerror 126d⟩
⟨*function* pl_getsizeerror 126e⟩
⟨*function* pl_childspaceerror 126f⟩
⟨*function* pl_scrollerror 127a⟩
⟨*function* pl_setscrollbarerror 127b⟩

⟨*function* pl_prinormal 37i⟩

⟨*function* pl_newpanel 20a⟩
⟨*function* plfree 20c⟩

D.2.4 lib_gui/libpanel/draw.c

⟨lib_gui/libpanel/draw.c 135c⟩≡
⟨libpanel *includes* 15d⟩

⟨*constant* PWID 51a⟩
⟨*constant* BWID 54e⟩
⟨*constant* FWID 73b⟩
⟨*constant* SPACE 32d⟩
⟨*constant* CKSIZE 63d⟩
⟨*constant* CKSPACE 64c⟩
⟨*constant* CKWID 62d⟩
⟨*constant* CKINSET 62c⟩
⟨*constant* CKBORDER 62e⟩

⟨*global* plldepth 28b⟩

⟨*globals* pl_xxx 28c⟩

⟨*function* pl_drawinit 28d⟩

⟨*function* `pl_relief` 31⟩
⟨*function* `pl_boxoutline` 32c⟩
⟨*function* `pl_outline` 32a⟩
⟨*function* `pl_box` 32b⟩
⟨*function* `pl_boxsize` 32e⟩
⟨*function* `pl_interior` 32f⟩

⟨*function* `pl_drawicon` 51b⟩
⟨*function* `pl_radio` 64b⟩
⟨*function* `pl_check` 62b⟩
⟨*function* `pl_ckwid` 63c⟩
⟨*function* `pl_sliderupd` 66b⟩

```
void pl_draw1(Panel *p, Image *b);
```

⟨*function* `pl_drawall` 30c⟩
⟨*function* `pl_draw1` 30b⟩
⟨*function* `pldraw` 30a⟩
⟨*function* `pl_invis` 94d⟩
⟨*function* `pl_iconsize` 51c⟩
⟨*function* `pl_highlight` 80b⟩
⟨*function* `pl_clr` 114a⟩
⟨*function* `pl_fill` 80c⟩
⟨*function* `pl_cpy` 83c⟩

D.2.5 lib_gui/libpanel/event.c

⟨`lib_gui/libpanel/event.c` 136a⟩≡
⟨`libpanel` *includes* 15d⟩

⟨*function* `plgrabkb` 38b⟩
⟨*function* `plkeyboard` 38c⟩

⟨*function* `pl_ptinpanel` 35d⟩
⟨*function* `plmouse` 35a⟩

D.2.6 lib_gui/libpanel/print.c

⟨`lib_gui/libpanel/print.c` 136b⟩≡
⟨`libpanel` *includes* 15d⟩

⟨*function* `pl_iprint` 124c⟩
⟨*function* `pl_ipprint` 124d⟩
⟨*function* `pl_print` 124b⟩

D.2.7 lib_gui/libpanel/label.c

⟨`lib_gui/libpanel/label.c` 136c⟩≡
⟨`libpanel` *includes* 15d⟩

```
typedef struct Label Label;
```

⟨*struct* `Label` 49a⟩

⟨*function* `pl_drawlabel` 50b⟩
⟨*function* `pl_hitlabel` 52b⟩
⟨*function* `pl_typelabel` 52c⟩

⟨*function* `pl_getsizelabel` 52d⟩
⟨*function* `pl_childspacelabel` 52f⟩
⟨*function* `plinitlabel` 49d⟩
⟨*function* `pllabel` 49c⟩
⟨*function* `plplacelabel` 50a⟩

Uses `Label` 49a.

D.2.8 lib_gui/libpanel/button.c

⟨`lib_gui/libpanel/button.c` 137a⟩≡

```
⟨libpanel includes 15d⟩

typedef struct Button Button;

⟨struct Button 58d⟩

/*
 * Button types
 */
⟨constant BUTTON 58e⟩
⟨constant CHECK 58f⟩
⟨constant RADIO 58g⟩

⟨function pl_drawbutton 60b⟩
⟨function pl_hitbutton 60d⟩
⟨function pl_typebutton 61a⟩
⟨function pl_getsizebutton 61b⟩
⟨function pl_childspacebutton 61c⟩
⟨function pl_initbtype 59c⟩
⟨function pl_buttonhit 60a⟩
⟨function plinitbutton 59e⟩
⟨function plinitcheckbutton 61e⟩
⟨function plinitradiobutton 63g⟩
⟨function plbutton 59d⟩
⟨function plcheckbutton 61d⟩
⟨function plradiobutton 63f⟩

⟨function pl_hitmenu 90b⟩
⟨function plinitmenu 89d⟩
⟨function plmenu 89c⟩
⟨function plsetbutton 63e⟩
```

Uses `Button` 58d.

D.2.9 lib_gui/libpanel/entry.c

⟨`lib_gui/libpanel/entry.c` 137b⟩≡

```
⟨libpanel includes 15d⟩
#include <keyboard.h>

typedef struct Entry Entry;

⟨struct Entry 52g⟩
⟨constant SLACK 118e⟩

⟨function pl_snarfentry 118c⟩
⟨function pl_pasteentry 118d⟩
⟨function pl_drawentry 54c⟩
```

⟨*function* `pl_hitentry` 55e⟩
⟨*function* `pl_typeentry` 56a⟩
⟨*function* `pl_getsizeentry` 57d⟩
⟨*function* `pl_childspaceentry` 58b⟩
⟨*function* `pl_freeentry` 53d⟩
⟨*function* `plinitentry` 53c⟩
⟨*function* `plentry` 53b⟩
⟨*function* `plentryval` 58c⟩

Uses `Entry` 52g.

D.2.10 `lib_gui/libpanel/edit.c`

⟨`lib_gui/libpanel/edit.c` 138⟩≡

```
/*
 * 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* 15d⟩

```
#include <keyboard.h>

typedef struct Edit Edit;
```

⟨*struct* `Edit` 103a⟩
⟨*function* `pl_drawedit` 104b⟩

⟨*function* `pl_snarfedit` 119b⟩
⟨*function* `pl_pasteedit` 119c⟩

⟨*function* `pl_hitedit` 104c⟩
⟨*function* `pl_scrolledit` 106c⟩
⟨*function* `pl_typeedit` 105⟩
⟨*function* `pl_getsizeedit` 106a⟩
⟨*function* `pl_childspaceedit` 106b⟩
⟨*function* `pl_freeedit` 104a⟩
⟨*function* `plinitedit` 103c⟩
⟨*function* `pledit` 103b⟩
⟨*function* `plescroll` 107a⟩
⟨*function* `plegetsel` 107b⟩
⟨*function* `plelen` 107c⟩
⟨*function* `pleget` 107d⟩
⟨*function* `plesel` 108a⟩
⟨*function* `plepaste` 108b⟩
⟨*function* `plemove` 108c⟩

Uses `Edit` 103a.

D.2.11 lib_gui/libpanel/slider.c

⟨lib_gui/libpanel/slider.c 139a⟩≡
  ⟨libpanel *includes* 15d⟩

```
typedef struct Slider Slider;
```

⟨*struct* Slider 65a⟩

⟨*function* pl_drawslider 66a⟩
⟨*function* pl_hitslider 67b⟩
⟨*function* pl_typeslider 68c⟩
⟨*function* pl_getsizeslider 68d⟩
⟨*function* pl_childspaceslider 68e⟩
⟨*function* plinitslider 65d⟩
⟨*function* plslider 65c⟩
⟨*function* plsetslider 69a⟩

Uses Slider 65a.

D.2.12 lib_gui/libpanel/canvas.c

⟨lib_gui/libpanel/canvas.c 139b⟩≡
  ⟨libpanel *includes* 15d⟩

```
typedef struct Canvas Canvas;
```

⟨*struct* Canvas 69c⟩

⟨*function* pl_drawcanvas 70a⟩
⟨*function* pl_hitcanvas 70b⟩
⟨*function* pl_typecanvas 70c⟩
⟨*function* pl_getsizecanvas 70d⟩
⟨*function* pl_childspacecanvas 70e⟩
⟨*function* plinitcanvas 69e⟩
⟨*function* plcanvas 69d⟩

Uses Canvas 69c.

D.2.13 lib_gui/libpanel/frame.c

⟨lib_gui/libpanel/frame.c 139c⟩≡
  ⟨libpanel *includes* 15d⟩

⟨*function* pl_drawframe 72c⟩
⟨*function* pl_hitframe 73c⟩
⟨*function* pl_typeframe 73d⟩
⟨*function* pl_getsizeframe 73e⟩
⟨*function* pl_childspaceframe 73g⟩
⟨*function* plinitframe 72b⟩
⟨*function* plframe 72a⟩

D.2.14 lib_gui/libpanel/group.c

⟨lib_gui/libpanel/group.c 139d⟩≡
  ⟨libpanel *includes* 15d⟩

⟨*function* pl_drawgroup 74c⟩
⟨*function* pl_hitgroup 74d⟩

⟨*function* `pl_typegroup` 74e⟩
⟨*function* `pl_getsizegroup` 75a⟩
⟨*function* `pl_childspacegroup` 75b⟩
⟨*function* `plinitgroup` 74b⟩
⟨*function* `plgroup` 74a⟩

D.2.15 `lib_gui/libpanel/list.c`

⟨`lib_gui/libpanel/list.c` 140a⟩≡
⟨`libpanel` *includes* 15d⟩

```
typedef struct List List;
```

⟨*struct* `List` 77g⟩

⟨*function* `pl_listsel` 80a⟩
⟨*function* `pl_liststrings` 79e⟩
⟨*function* `pl_drawlist` 79d⟩
⟨*function* `pl_hitlist` 80f⟩
⟨*function* `pl_scrolllist` 82b⟩
⟨*function* `pl_typelist` 81a⟩
⟨*function* `pl_getsizelist` 81b⟩
⟨*function* `pl_childspacelist` 81c⟩
⟨*function* `plinitlist` 78c⟩
⟨*function* `pllist` 78b⟩

Uses `List` 77g.

D.2.16 `lib_gui/libpanel/message.c`

⟨`lib_gui/libpanel/message.c` 140b⟩≡
⟨`libpanel` *includes* 15d⟩

```
typedef struct Message Message;
```

⟨*struct* `Message` 100a⟩

⟨*function* `pl_textmsg` 101b⟩
⟨*function* `pl_foldsize` 102d⟩
⟨*function* `pl_drawmessage` 101a⟩
⟨*function* `pl_hitmessage` 102a⟩
⟨*function* `pl_typemessage` 102b⟩
⟨*function* `pl_getsizemessage` 102c⟩
⟨*function* `pl_childspacemessage` 102e⟩
⟨*function* `plinitmessage` 100c⟩
⟨*function* `plmessage` 100b⟩

Uses `Message` 100a.

D.2.17 `lib_gui/libpanel/pack.c`

⟨`lib_gui/libpanel/pack.c` 140c⟩≡
⟨`libpanel` *includes* 15d⟩

⟨*function* `pl_max` 43d⟩
⟨*function* `pl_sizesibs` 42d⟩
⟨*function* `pl_sizereq` 42a⟩
⟨*function* `pl_getshare` 48d⟩

⟨*function* `pl_setrect` 44a⟩
⟨*function* `plpack` 40⟩
⟨*function* `plmove` 93h⟩

D.2.18 lib_gui/libpanel/popup.c

⟨`lib_gui/libpanel/popup.c` 141a⟩≡
```
/*
 * popup
 * looks like a group, except diverts hits on certain buttons to
 * panels that it temporarily pops up.
 */
```
⟨`libpanel` *includes* 15d⟩

```
typedef struct Popup Popup;
```

⟨*struct* `Popup` 90c⟩

⟨*function* `pl_drawpopup` 93e⟩
⟨*function* `pl_hitpopup` 92b⟩
⟨*function* `pl_typepopup` 93d⟩
⟨*function* `pl_getsizepopup` 94g⟩
⟨*function* `pl_childspacepopup` 95a⟩
⟨*function* `pl_pripopup` 37j⟩
⟨*function* `plinitpopup` 91b⟩
⟨*function* `plpopup` 91a⟩

Uses `Popup` 90c.

D.2.19 lib_gui/libpanel/pulldown.c

⟨`lib_gui/libpanel/pulldown.c` 141b⟩≡
```
/*
 * pulldown
 * makes a button that pops up a panel when hit
 */
```
⟨`libpanel` *includes* 15d⟩

```
typedef struct Pulldown Pulldown;
```

⟨*struct* `Pulldown` 95b⟩

⟨*function* `pl_drawpulldown` 96a⟩
⟨*function* `pl_hitpulldown` 96b⟩
⟨*function* `pl_typepulldown` 98a⟩
⟨*function* `pl_getsizepulldown` 98b⟩
⟨*function* `pl_childspacepulldown` 98c⟩
⟨*function* `plinitpulldown` 95d⟩
⟨*function* `plpulldown` 95c⟩
⟨*function* `plmenubar` 98d⟩

Uses `Pulldown` 141b.

D.2.20 lib_gui/libpanel/rtext.c

⟨*constant* `LEAD` 141c⟩≡ (144c)
```
#define LEAD 4 /* extra space between lines */
```

⟨*constant* BORD 142a⟩≡ (144c)
```
#define BORD 2 /* extra border for images */
```

⟨*function* pltabsize 142b⟩≡ (144c)
```
void pltabsize(int min, int size){
    pl_tabmin=min;
    pl_tabsize=size;
}
```
Uses pl_tabmin 144c and pl_tabsize 144c.

⟨*function* pl_space 142c⟩≡ (144c)
```
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 144c and pl_tabsize 144c.

⟨*function* pl_rtfmt 142d⟩≡ (144c)
```
/*
 * 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 142a, LEAD-7 141c, pl_space() 142c, plmove() 93h, and plpack() 40.

⟨*function* pl_stuffbitmap 143⟩≡ (144c)

```
/*
 * 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() 143.

⟨*function* pl_reposition 144a⟩≡ (144c)

```
  /*
   * 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() 83c and plmove() 93h.

⟨*function* plrtsnarftext 144b⟩≡ (144c)

```
  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() 128b.

⟨lib_gui/libpanel/rtext.c 144c⟩≡

```
  /*
   * Rich text with images.
   * Should there be an offset field, to do subscripts & kerning?
   */
  ⟨libpanel includes 15d⟩
  #include "rtext.h"

  ⟨constant LEAD 141c⟩
```

⟨*constant* BORD 142a⟩

⟨*function* pl_rtnew 112b⟩
⟨*function* plrtpanel 113a⟩
⟨*function* plrtstr 113b⟩
⟨*function* plrtbitmap 113c⟩
⟨*function* plrtfree 113d⟩
```
int pl_tabmin;
int pl_tabsize;
```
⟨*function* pltabsize 142b⟩
⟨*function* pl_space 142c⟩
⟨*function* pl_rtfmt 142d⟩

⟨*function* pl_stuffbitmap 143⟩

⟨*function* pl_rtdraw 113g⟩
⟨*function* pl_reposition 144a⟩
⟨*function* pl_rtredraw 114b⟩
⟨*function* pl_rthit 115a⟩

⟨*function* plrtseltext 115b⟩

⟨*function* plrtsnarftext 144b⟩

D.2.21 lib_gui/libpanel/scroll.c

⟨*function* plgetscroll 145a⟩≡ (145c)
```
Scroll plgetscroll(Panel *p){
    return p->scr;
}
```

⟨*function* plsetscroll 145b⟩≡ (145c)
```
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 133c and VERT.

⟨lib_gui/libpanel/scroll.c 145c⟩≡
⟨libpanel *includes* 15d⟩

⟨*function* plscroll 77a⟩
⟨*function* plgetscroll 145a⟩
⟨*function* plsetscroll 145b⟩

D.2.22 lib_gui/libpanel/scrollbar.c

⟨lib_gui/libpanel/scrollbar.c 145d⟩≡
⟨libpanel *includes* 15d⟩

```
typedef struct Scrollbar Scrollbar;
```

⟨*struct* Scrollbar 83d⟩
⟨*constant* SBWID 84d⟩

⟨*function* pl_drawscrollbar 85b⟩

⟨*function* `pl_hitscrollbar` 85c⟩
⟨*function* `pl_typescrollbar` 87a⟩
⟨*function* `pl_getsizescrollbar` 87b⟩
⟨*function* `pl_childspacescrollbar` 87c⟩
⟨*function* `pl_setscrollbarscrollbar` 87e⟩
⟨*function* `pl_priscrollbar` 37k⟩
⟨*function* `plinitscrollbar` 84a⟩
⟨*function* `plscrollbar` 83f⟩

Uses `Scrollbar` 83d.

D.2.23 lib_gui/libpanel/snarf.c

⟨`lib_gui/libpanel/snarf.c` 146a⟩≡
⟨`libpanel` *includes* 15d⟩

⟨*function* `plputsnarf` 117c⟩
⟨*function* `plgetsnarf` 117d⟩
⟨*function* `plsnarf` 117e⟩
⟨*function* `plpaste` 118a⟩

D.2.24 lib_gui/libpanel/textview.c

⟨*function* `pl_setscrpos` 146b⟩≡ (148a)

```
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` 146c⟩≡ (148a)

```
/*
 * 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()` 35a.

⟨*function* `pl_scrolltextview` 146d⟩≡ (148a)

```
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() 32f, pl_outline() 32a, pl_rtredraw() 114b, and pl_setscrpos() 146b.

⟨*function* pl_pritextview 147a⟩≡ (148a)

```
    /*
     * 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() 32f, pl_ptinpanel() 35d, and pl_rthit() 115a.

⟨*function* pl_snarftextview 147b⟩≡ (148a)

```
    char* pl_snarftextview(Panel *p){
        return plrtsnarftext(((Textview *)p->data)->text);
    }
```

Uses plrtsnarftext() 144b.

⟨*function* plgetpostextview 147c⟩≡ (148a)

```
    int plgetpostextview(Panel *p){
        return ((Textview *)p->data)->yoffs;
    }
```

⟨*function* plsetpostextview 147d⟩≡ (148a)

```
    void plsetpostextview(Panel *p, int yoffs){
        ((Textview *)p->data)->yoffs=yoffs;
        pldraw(p, p->b);
    }
```

Uses pldraw() 30a.

⟨lib_gui/libpanel/textview.c 148a⟩≡
```
/*
 * 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 15d⟩

typedef struct Textview Textview;
⟨struct Textview 108d⟩

⟨function pl_setscrpos 146b⟩
⟨function pl_drawtextview 109c⟩
⟨function pl_passon 146c⟩
⟨function pl_hittextview 110a⟩
⟨function pl_scrolltextview 146d⟩
⟨function pl_typetextview 110b⟩
⟨function pl_getsizetextview 111a⟩
⟨function pl_childspacetextview 111b⟩
⟨function pl_pritextview 147a⟩

⟨function pl_snarftextview 147b⟩

⟨function plinittextview 109b⟩
⟨function pltextview 109a⟩
⟨function plgetpostextview 147c⟩
⟨function plsetpostextview 147d⟩
```
Uses Textview 108d.

D.2.25 lib_gui/libpanel/textwin.c

⟨function tw_before 148b⟩≡ (156d)
```
/*
 * 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 148c⟩≡ (156d)
```
/*
 * 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() 148b.

⟨*function* tw_rune2pt 149a⟩≡ (156d)

```
/*
 * 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 149b⟩≡ (156d)

```
/*
 * 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 54a and pl_erealloc() 128b.

⟨*function* tw_setloc 149c⟩≡ (156d)

```
/*
 * 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()` 149b.

⟨*function* `tw_draw` 150⟩≡ (156d)

```
/*
 * 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() 114a.

⟨*function* tw_hilitep 151a⟩≡ (156d)

```
/*
 * 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() 80b and tw_before() 148b.

⟨*function* twhilite 151b⟩≡ (156d)

```
/*
 * 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() 114a, tw_draw() 150, and tw_hilitep() 151a.

⟨*function* twselect 151c⟩≡ (156d)

```
/*
 * 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()` 151a, `tw_rune2pt()` 149a, `twhilite()` 151b, and `twpt2rune()` 148c.

⟨*function* `tw_clrend` 152a⟩≡ (156d)

```
/*
 * 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()` 114a.

⟨*function* `tw_moverect` 152b⟩≡ (156d)

```
/*
 * 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()` 114a and `pl_cpy()` 83c.

⟨*function* `tw_moveup` 153a⟩≡ (156d)

```
/*
 * 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()` 83c and `tw_moverect()` 152b.

⟨*function* `tw_movedn` 153b⟩≡ (156d)

```
/*
 * 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()` 83c and `tw_moverect()` 152b.

⟨*function* `tw_relocate` 154a⟩≡ (156d)

```
/*
 * 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()` 128a, `tw_before()` 148b, `tw_movedn()` 153b, `tw_moveup()` 153a, and `tw_setloc()` 149c.

⟨*function* `twreplace` 154b⟩≡ (156d)

```
/*
```

```
 * 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 54a, pl_erealloc() 128b, tw_clrend() 152a, tw_draw() 150, and tw_setloc() 149c.

⟨*function* twscroll 155a⟩≡ (156d)

```
/*
 * 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() 152a, tw_draw() 150, and tw_setloc() 149c.

⟨*function* twreshape 155b⟩≡ (156d)

```
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() 152a, tw_draw() 150, and tw_setloc() 149c.

⟨*function* twnew 156a⟩≡ (156d)

```
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 54a and pl_emalloc() 128a.

⟨*function* twfree 156b⟩≡ (156d)

```
void twfree(Textwin *t){
    free(t->loc);
    free(t->text);
    free(t);
}
```

⟨*function* twmove 156c⟩≡ (156d)

```
/*
 * 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 156d⟩≡

```
/*
 * 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* 15d⟩

⟨*constant* SLACK*(*(lib_gui/libpanel/textwin.c)*)* 54a⟩

⟨*function* tw_before 148b⟩
⟨*function* twpt2rune 148c⟩
⟨*function* tw_rune2pt 149a⟩
⟨*function* tw_storeloc 149b⟩
⟨*function* tw_setloc 149c⟩
⟨*function* tw_draw 150⟩
⟨*function* tw_hilitep 151a⟩
⟨*function* twhilite 151b⟩
⟨*function* twselect 151c⟩
⟨*function* tw_clrend 152a⟩
⟨*function* tw_moverect 152b⟩
⟨*function* tw_moveup 153a⟩
⟨*function* tw_movedn 153b⟩
⟨*function* tw_relocate 154a⟩
⟨*function* twreplace 154b⟩
⟨*function* twscroll 155a⟩
⟨*function* twreshape 155b⟩
⟨*function* twnew 156a⟩
⟨*function* twfree 156b⟩
⟨*function* twmove 156c⟩

D.2.26 lib_gui/libpanel/utf.c

⟨*function* pl_idchar 157a⟩≡ (158c)

```
/*
 * 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 157b⟩≡ (158c)

```
int pl_rune1st(int c){
    return (c&0xc0)!=0x80;
}
```

⟨*function* `pl_nextrune` 158a⟩≡ (158c)

```
char *pl_nextrune(char *s){
    do s++; while(!pl_rune1st(*s));
    return s;
}
```

Uses `pl_rune1st()` 157b.

⟨*function* `pl_runewidth` 158b⟩≡ (158c)

```
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()` 157b.

⟨`lib_gui/libpanel/utf.c` 158c⟩≡

⟨`libpanel` *includes* 15d⟩

⟨*function* `pl_idchar` 157a⟩
⟨*function* `pl_rune1st` 157b⟩
⟨*function* `pl_nextrune` 158a⟩
⟨*function* `pl_runewidth` 158b⟩

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.

__anon_enum_3: 133c

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 10

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 10

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 13

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 38, 122

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 9, 30, 122

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 7, 9, 15, 122

[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