Kiln API reference

Every public symbol of Kiln, a native UI library for C3

Contents

1 Lifecycle 3
2 Windows and title bars 10
3 Ids and state 11
4 Containers and layout 14
5 Layers 23
6 Menus, popovers and tooltips 27
7 Dialogs, sheets and toasts 30
8 Text editing 34
9 Inputs 42
10 Navigation 49
11 Collections 52
12 Status 60
13 Canvas 64
14 Charts 69
15 Background work 77
16 Interaction and input 81
17 Focus 94
18 Text and icons 96
19 Images 105
20 Widgets 109
21 Theme and color 113
22 Motion 134
23 Drawing 136
24 Backends 144
25 Operating system 149
26 Testing 157

This book lists every public symbol of Kiln: its signature, its documentation from the code, and its fields or values. It is generated from the source by `scripts/docs_api.sh` and grouped by topic, in the order of `docs/api-topics.txt`. How the pieces fit together, and why they are built that way, is in the companion book, the Kiln guide (`kiln.pdf`), whose symbol names link to their entries here.

1 Lifecycle

ui::create

```
fn Ui* ui::create(
    Allocator allocator = mem,
    Theme theme = themes::LIGHT,
    int max_elements = DEFAULT_MAX_ELEMENTS,
    FontSet fonts = {})
```

Creates a `Ui` with no platform or renderer attached; a backend's `create_ui` (`sdl::create_ui`, `raylib::create_ui`) makes one that has both. Everything it allocates comes from `allocator`, and `Ui.destroy` gives it all back.

- `max_elements`: Layout elements Clay starts with; doubled by itself when a frame needs more
- `fonts`: Fonts for text; empty uses Kiln's built-in fonts

Ui

```
struct Ui
```

The UI context every widget call goes through: it keeps widget state, focus, the theme, the layout and draw list, and all the memory Kiln uses. Make one with `ui::create`, or with a backend's `create_ui` (`sdl::create_ui`, `raylib::create_ui`) to also read input and draw, and free it with `Ui.destroy`.

Ui.begin

```
fn void Ui.begin(&self, Input* input = null)
```

Starts a frame: takes this frame's input and resets the frame arena. Without `input`, polls the attached platform for it.

Requires:

- `!self.in_frame : "begin() called twice without end()"`

Ui.end

```
fn void Ui.end(&self)
```

Ends the frame: lays it out, builds the draw list, settles hover and captures for the next frame and collects state that has gone unused.

Requires:

- `self.in_frame : "end() without begin()"`
- `self.id_stack.len() == 0 : "push_id without pop_id"`
- `self.dirs.len() == 0 : "a container was opened and not closed"`
- `self.scope_stack.len() == 0 && self.group_stack.len() == 0 : "a focus scope or group was opened and not closed"`

Ui.set_paused

```
fn void Ui.set_paused(&self, bool paused)
```

Marks the UI paused or resumed, calling the matching hook once per change. `Ui.begin` calls it from the input's `paused` flag; an App-style loop calls it before waiting while the platform has no visible surface.

`Ui.gpu_lost`

```
fn void Ui.gpu_lost(&self)
```

The GPU context is gone. Forgets every GPU handle without freeing it, and invalidates the glyph and image textures so they are made again on the next upload. An app that keeps its own GPU resources should do the same in its `gpu_lost` hook.

`Ui.gpu_restored`

```
fn void Ui.gpu_restored(&self)
```

The platform rebuilt its GPU context; textures already re-create lazily, then the app's hook runs.

`Ui.request_quit`

```
fn void Ui.request_quit(&self)
```

Asks an App-style loop to stop after this frame is shown.

`LifecycleFn`

```
alias LifecycleFn = fn void(Ui* ui, void* user)
```

An app hook called when the platform pauses or resumes the UI, or when its GPU context is lost or restored.

`UiLifecycle`

```
struct UiLifecycle
```

The app's lifecycle hooks and their data: set them after `ui::create`. They run outside a frame; do not build widgets from them. A null hook is ignored.

Field	Type	Meaning
`pause`	`LifecycleFn`	the platform has no visible surface (Android backgrounded the app)
`resume`	`LifecycleFn`	the platform is visible again
`gpu_lost`	`LifecycleFn`	the GPU handles are gone; the UI's textures re-create lazily
`gpu_restored`	`LifecycleFn`	the platform rebuilt its GPU context
`user`	`void*`	passed to every hook

`ui::MAX_DT`

```
const float ui::MAX_DT = 0.1
```

The longest frame step the UI takes, in seconds: a suspend or a stall does not jump animations by minutes.

`Ui.wake_after`

```
fn void Ui.wake_after(&self, double seconds)
```

Asks for a frame `seconds` from now even if nothing moves or arrives, as a tooltip waiting out its delay does. The earliest request still to come wins: one made in the frame an earlier deadline arrives replaces it. On a platform that `wakes()`, an App-style loop sleeps until then; on one that doesn't (raylib) the `Ui` counts as animating until then, so frames keep coming.

`Ui.draw`

```
fn void Ui.draw(&self)
```

Draws the frame built by the last `Ui.end` through the attached renderer; does nothing without one.

`Ui.destroy`

```
fn void Ui.destroy(&self)
```

Frees everything, including an attached platform and renderer.

`Ui.frame_new`

```
macro Ui.frame_new(&self, $Type)
```

Zeroed memory for one `$Type` from the frame arena; it lives until the next `Ui.begin`.

`Ui.frame_array`

```
macro Ui.frame_array(&self, $Type, n)
```

Zeroed memory for `n` of `$Type` from the frame arena, as `Ui.frame_new`.

Constant	Type	Value
`ui::DEFAULT_MAX_ELEMENTS`	`int`	`2048`

`app::run`

```
fn void app::run(
    String title,
    int width,
    int height,
    FrameFn frame,
    void* user = null,
    Theme theme = themes::SYSTEM,
    DrawFn draw_under = null,
    DrawFn draw_over = null,
    bool resizable = true,
    FontSet fonts = {},
    String app_id = "",
    char[] icon = {},
    bool title_bar = false,
    BackgroundFn background = null)
```

Opens a window and runs an app in it until the window closes, calling `frame` every frame and drawing the result over the theme's surface color.

- `theme`: `themes::SYSTEM` (the default) follows the OS theme; any other theme overrides it
- `draw_under`: Draws before the UI each frame (backgrounds, custom scenes); raw OpenGL, so it keeps the OpenGL renderer
- `draw_over`: Draws after the UI each frame; raw OpenGL, so it keeps the OpenGL renderer
- `fonts`: The fonts the UI draws with; empty uses `text::default_fonts()`
- `app_id`: The app's .desktop file name without ".desktop" ("glaze"), which Wayland and X11 desktops match the window to for its icon and dock entry; SDL3 only
- `icon`: A PNG or JPEG, such as one $embed-ded, for the window's icon; ignored on the web
- `title_bar`: Draws Kiln's title bar on Wayland and skips libdecor; elsewhere the OS title bar stays and this is ignored
- `background`: Runs instead of the frame while the app is in the background, with the same user pointer

FrameFn

```
alias FrameFn = fn void(Ui* ui, void* user)
```

The frame function `app::run` calls between `Ui.begin` and `Ui.end`, with the `user` pointer given to `app::run`; it builds the UI.

DrawFn

```
alias DrawFn = fn void(Ui* ui, void* user)
```

A function `app::run` calls while drawing a frame, before or after `Ui.draw`, with the `user` pointer given to `app::run`.

BackgroundFn

```
alias BackgroundFn = fn void(Ui* ui, void* user)
```

A function `app::run` calls while the app is in the background, instead of the frame: there is no surface then, but a player or a queue keeps moving.

app::window_override

```
fn void app::window_override(int* width, int* height)
```

KILN_WINDOW=<width>x<height>, if set and well formed, replaces the window size.

app::screenshot_request

```
fn Screenshot app::screenshot_request()
```

Reads the `Screenshot` asked for through KILN_SCREENSHOT, if any; free it with `Screenshot.free`.

Screenshot

```
struct Screenshot
```

A screenshot asked for through the environment: KILN_SCREENSHOT names the PNG file and KILN_SCREENSHOT_FRAME the frame to take it at (default 10). An empty `path` means none was asked for. Get one with `app::screenshot_request` and call `Screenshot.due` after each frame is shown.

Field	Type	Meaning
path	String	
frame	ulong	

Screenshot.due

```
fn bool Screenshot.due(&self, ulong frame)
```

Saves the screen once `frame` reaches the frame asked for and returns true, meaning the app should exit; returns false otherwise.

Screenshot.free

```
fn void Screenshot.free(&self)
```

Frees the path `app::screenshot_request` allocated.

app::frame_times_request

```
fn FrameTimes app::frame_times_request(Clock started)
```

Reads the `FrameTimes` asked for through KILN_FRAME_TIMES, if any; free it with `FrameTimes.free`.

FrameTimes

```
struct FrameTimes
```

Frame times asked for through the environment: KILN_FRAME_TIMES names the CSV file, and KILN_FRAME_TIMES_FRAMES the frames to record (default 300). An empty `path` means none was asked for. While recording, frames come back to back: no idle sleeping and no frame cap.

Field	Type	Meaning
path	String	
frames	int	
started	Clock	when app::run began, for the startup time
startup_ms	double	app::run to the end of the first presented frame
rows	List{double[<5>]}	build, draw, present in ms; draw calls, GL calls

FrameTimes.record

```
fn bool FrameTimes.record(&self, Stats s, double draw_ms, double present_ms)
```

Records one frame: Kiln's own work from its `Stats` (build, layout, paint, the rest of `end`), drawing, presenting (`EndDrawing`, with the buffer swap), and the draw's draw and GL calls. Returns true, and writes the file, once enough frames are recorded.

FrameTimes.free

```
fn void FrameTimes.free(&self)
```

Frees what `app::frame_times_request` allocated.

raylib::create_ui

```
fn Ui* raylib::create_ui(
    Allocator allocator = mem,
    Theme theme = themes::SYSTEM,
    FontSet fonts = {})
```

Creates a `Ui` that reads input from raylib and draws through it, for Game style: Kiln inside your own raylib loop. Call it after InitWindow(); without a window (no display) it ends the program with a message and exit code 1. It follows the OS theme (`themes::SYSTEM`) unless given another `theme`.

```
Ui* ui = raylib::create_ui();
while (!WindowShouldClose())
{
    ui.begin();
    ...widgets...
    ui.end();
    BeginDrawing();
    draw_world();
    ui.draw();
    EndDrawing();
}
ui.destroy();
```

raylib::attach_renderer

```
fn void raylib::attach_renderer(Ui* ui, Allocator allocator = mem)
```

Gives `ui` a raylib renderer when it has none; with one already, does nothing. For a `Ui` made by `ui::create`, as a game that feeds `Ui.begin` its own `Input` makes one, without Kiln's raylib platform.

Requires:

- `rl::is_window_ready() : "open raylib's window first: the renderer needs its GL context"`

raylib::new_renderer

```
fn RaylibRenderer* raylib::new_renderer(Allocator allocator = mem)
```

A raylib renderer: Kiln's draw list through rlgl, in the middle of a raylib frame. Needs raylib's window, whose GL context it draws with.

Requires:

- `rl::is_window_ready() : "open raylib's window first: the renderer needs its GL context"`

raylib::require_window

```
fn void raylib::require_window()
```

Ends the program with a message on stderr and exit code 1 when raylib has no window, as when InitWindow() found no display: without a window there is no graphics context, and loading Kiln's shader and atlas would crash.

sdl::create_ui

```
fn Ui* sdl::create_ui(
    String title,
    int width,
    int height,
    bool resizable = true,
    Allocator allocator = mem,
    Theme theme = themes::SYSTEM,
    FontSet fonts = {},
    String app_id = "",
    char[] icon = {},
    bool title_bar = false,
    bool needs_raw_gl = false)
```

Opens a window of `width` by `height` points titled `title` and a `Ui` drawing into it. On Linux that is SDL_GPU, on Vulkan, with an OpenGL window when no Vulkan device or renderer can be made; elsewhere it is OpenGL until Metal and Direct3D 12 shaders exist. `KILN_RENDERER=gl` forces OpenGL and `KILN_RENDERER=gpu` asks for SDL_GPU. Without a display it says so and exits, as `raylib::create_ui` does.

- `app_id`: The app's .desktop file name without ".desktop", which Wayland and X11 desktops match the window to; empty: SDL's default
- `icon`: A PNG or JPEG for the window's icon; empty: none
- `title_bar`: Draws a window with no OS title bar and a Kiln title bar on Wayland; elsewhere the OS title bar stays
- `needs_raw_gl`: Keeps OpenGL, for an app that draws with raw GL; `app::run` sets it when `draw_under` or `draw_over` is given

sdl::platform_of

```
fn SdlPlatform* sdl::platform_of(Ui* ui)
```

The SDL3 platform of a `Ui` made by `sdl::create_ui`.

2 Windows and title bars

`WindowAction`

```
enum WindowAction : char
```

What the title bar's buttons asked the app to do this frame.

Value	Meaning
`NONE`	no window button was pressed
`MINIMIZE`	the minimize button
`MAXIMIZE`	maximize, or restore when already maximized
`CLOSE`	the close button
`MENU`	a right-click on the bar: the system's window menu, as a native title bar has

`Ui.title_bar`

```
fn WindowAction Ui.title_bar(&self, String title, bool maximized = false)
```

The window's title bar: the title at the start, then the window's minimize, maximize or restore, and close buttons at the end. An App-style app that asked for a Kiln title bar builds it before its own content; the platform's hit test turns the empty bar left of the buttons into a drag region and the window's edges into resize handles. A double-click on the bar maximizes or restores the window, and a right-click on it asks for the system's window menu (`MENU`). Returns the window action asked for this frame, once; the app does it.

`ui::TITLE_BAR_HEIGHT`

```
const float ui::TITLE_BAR_HEIGHT = 46
```

Points the title bar is tall.

`ui::TITLE_BAR_TITLE_WIDTH`

```
const float ui::TITLE_BAR_TITLE_WIDTH = 200
```

Points of the title bar at its start that double-click to maximize or restore, and that are not a drag region.

`ui::TITLE_BAR_BUTTONS_WIDTH`

```
const float ui::TITLE_BAR_BUTTONS_WIDTH = 88
```

Points the three window buttons and their gaps take at the title bar's end: 12 + 24 + 2 + 24 + 2 + 24.

3 Ids and state

Id

```
typedef Id = inline ulong
```

A widget's identity across frames: a hash of its label and the id stack, made by `Ui.make_id`. State, rects, focus and input are keyed by it.

ui::NO_ID

```
const Id ui::NO_ID = 0
```

The empty id: no widget. `Ui.make_id` never returns it.

Ui.make_id

```
fn Id Ui.make_id(&self, String label)
```

The id of `label` in the current scope, counted as used this frame. In a safe build, asking twice for the same id in one frame is reported on stderr. Never returns `ui::NO_ID`.

Ui.make_id_quiet

```
fn Id Ui.make_id_quiet(&self, String label)
```

The id `Ui.make_id` would give `label` here, without counting it as used this frame.

ui::label_text

```
fn String ui::label_text(String label)
```

The part of `label` that is shown: everything before `##`.

Ui.@id

```
macro void Ui.@id(&self, key; @body())
```

Runs the body with `key` pushed on the id stack: a String, or an integer such as a loop index.

Ui.push_id

```
fn void Ui.push_id(&self, String key)
```

Pushes `key` on the id stack, so ids made until the matching `Ui.pop_id` are scoped by it. `Ui.@id` pushes and pops around a body.

Ui.push_id_index

```
fn void Ui.push_id_index(&self, long index)
```

Pushes an integer such as a loop index on the id stack, like `Ui.push_id`. Index 3 and the string "3" give different scopes.

Ui.pop_id

```
fn void Ui.pop_id(&self)
```

Pops the scope the last `Ui.push_id` or `Ui.push_id_index` pushed. Every push needs a pop before `Ui.end`.

Requires:

- `self.id_stack.len() > 0 : "pop_id without push_id"`

Ui.@state

```
macro Ui.@state(&self, $Type, Id id)
```

A pointer to `id`'s state of type `$Type`, zeroed the first time `id` asks. The state lives across frames and is freed after `ui::STATE_DATA_SECONDS` without a call, unless `Ui.keep_state` holds it.

Requires:

- `!self.has_state(id) || self.state_type(id) == $Type::typeid : "this id already has state of another type"`

Ui.@state_owning

```
macro Ui.@state_owning(&self, $Type, Id id, StateDrop drop)
```

`Ui.@state` for state that owns memory: `drop` frees it before the state is collected or the `Ui` is destroyed.

Requires:

- `!self.has_state(id) || self.state_type(id) == $Type::typeid : "this id already has state of another type"`

StateDrop

```
alias StateDrop = fn void(void* state)
```

Frees what a widget's state owns (lists, strings) before the state itself is freed; see `Ui.@state_owning`.

Ui.has_state

```
fn bool Ui.has_state(&self, Id id)
```

Whether `id` has state from `Ui.@state` that hasn't been freed.

Ui.keep_state

```
fn void Ui.keep_state(&self, Id id, bool keep = true)
```

Stops `id`'s state from being collected, for state that must outlive the widget's absence, such as a document's undo history. Does nothing if the id has no state yet.

- `keep`: false lets the state be collected again

Ui.rect_of

```
fn Rect? Ui.rect_of(&self, Id id)
```

Where `id`'s element was placed by the last layout, which runs in `Ui.end`; during a frame this is the previous frame's rect. Only elements given an id keep a rect. Fails with NOT_FOUND if `id` has none, or was not laid out in the last `ui::STATE_RECT_FRAMES` frames.

ui::STATE_RECT_FRAMES

```
const int ui::STATE_RECT_FRAMES = 30
```

A kept rect is dropped after this many frames without being laid out (half a second at 60 fps).

ui::STATE_DATA_SECONDS

```
const float ui::STATE_DATA_SECONDS = 5
```

Seconds a widget's state (springs, scroll, edits) survives without `Ui.@state` asking for it, unless `Ui.keep_state` holds it.

4 Containers and layout

Ui.@row

```
macro void Ui.@row(
    &self,
    String id = "",
    Size width = FIT,
    Size height = FIT,
    float pad = 0,
    float pad_x = 0,
    float pad_y = 0,
    float gap = 0,
    Align align = START,
    Align justify = START,
    Token bg = NO_TOKEN,
    Corners radius = NONE,
    bool border = false,
    bool clip = false; @body())
```

Children side by side.

Ui.@column

```
macro void Ui.@column(
    &self,
    String id = "",
    Size width = FIT,
    Size height = FIT,
    float pad = 0,
    float pad_x = 0,
    float pad_y = 0,
    float gap = 0,
    Align align = START,
    Align justify = START,
    Token bg = NO_TOKEN,
    Corners radius = NONE,
    bool border = false,
    bool clip = false; @body())
```

Children stacked top to bottom.

Ui.@stack

```
macro void Ui.@stack(
    &self,
    Dir dir,
    String id = "",
    Size width = FIT,
    Size height = FIT,
    float pad = 0,
    float gap = 0,
    Align align = START,
    Align justify = START; @body())
```

A row or a column, chosen by `dir`, often with `Ui.at` for responsive layouts.

Ui.@card

```
macro void Ui.@card(
    &self,
    String id = "",
    Size width = FIT,
    Size height = FIT,
    Dir dir = COLUMN,
    float pad = 4,
    float gap = 2,
    Align align = START,
    Align justify = START,
    Elevation shadow = SM; @body())
```

A raised panel: a `SURFACE_RAISED` background, `PANEL` corners, a hairline border and a `shadow` (`SM` by default).

Ui.@box_ex

```
macro void Ui.@box_ex(&self, BoxOpts opts; @body())
```

Runs the body inside a box described by `opts`.

Ui.@scroll

```
macro void Ui.@scroll(
    &self,
    String id,
    ScrollAxis axis = VERTICAL,
    Overscroll overscroll = AUTO,
    Size width = GROW,
    Size height = GROW,
    Dir dir = COLUMN,
    float pad = 0,
    float pad_x = 0,
    float pad_y = 0,
    float gap = 0,
    Align align = START,
    Align justify = START,
    Token bg = NO_TOKEN,
    Corners radius = NONE,
    bool border = false; @body())
```

Runs the body in a box that scrolls its content: with the wheel, a touch or pen drag, its scrollbar, and by bringing focused widgets into view. It grows to fill its parent by default; with `FIT` along its axis it would always fit its content and never scroll.

Ui.@scroll_ex

```
macro void Ui.@scroll_ex(&self, String id, ScrollOpts opts; @body())
```

`Ui.@scroll` with its options in a struct.

ScrollOpts

```
struct ScrollOpts
```

A scroll area's options, for `Ui.@scroll_ex`. The zero value scrolls vertically.

Field	Type	Meaning
`axis`	`ScrollAxis`	
`overscroll`	`Overscroll`	
`box`	`BoxOpts`	the area's box; its id and clip are set by the area

`ScrollAxis`

```
enum ScrollAxis : char
```

Which way a scroll area scrolls. Content is laid out at its natural size along that axis and fitted to the area across it.

Value	Meaning
`VERTICAL`	up and down; text wraps to the area's width
`HORIZONTAL`	sideways
`BOTH`	both ways; the content keeps its natural width

`Overscroll`

```
enum Overscroll : char
```

What an area does with scrolling it can't use because it is at its end (CSS `overscroll-behavior`).

Value	Meaning
`AUTO`	passes it to the enclosing scroll area
`CONTAIN`	keeps it: nothing outside moves

`ScrollInfo`

```
struct ScrollInfo
```

What a scroll area shows, from the last frame.

Field	Type	Meaning
`offset`	`float[<2>]`	how far the content has moved, in points
`range`	`float[<2>]`	how far it can move; 0 where the content fits
`bars`	`float`	the scrollbars' opacity, from 0 hidden to 1

`Ui.scroll_info`

```
fn ScrollInfo Ui.scroll_info(&self, Id id)
```

Where the scroll area `id` stands, such as to show a "back to top" button; all zero for an id that isn't a scroll area.

Ui.is_scroll_area

```
fn bool Ui.is_scroll_area(&self, Id id)
```

Whether `id` is a scroll area, telling "at the top" from "not a scroll area".

Ui.scroll_view

```
fn float[<2>] Ui.scroll_view(&self, Id id)
```

The size of the scroll area `id`'s view from the last layout; zero for an id that isn't a scroll area or before its first layout.

Ui.scroll_to

```
fn void Ui.scroll_to(&self, Id area, float[<2>] offset, bool animate = true)
```

Moves the scroll area `area` so its content is `offset` points along, within its range from the last layout: easing there with `animate`, at once without it or under reduced motion. Does nothing for an id that isn't a scroll area.

ui::WHEEL_STEP

```
const float ui::WHEEL_STEP = 48
```

Points one wheel notch scrolls when the OS has no setting for it.

ui::WHEEL_LINE

```
const float ui::WHEEL_LINE = 16
```

Points per line, for an OS that sets the wheel in lines.

ui::WHEEL_PAGE

```
const float ui::WHEEL_PAGE = -1
```

A `Ui.wheel_step` meaning one view per notch.

ui::WHEEL_LATCH_SECONDS

```
const double ui::WHEEL_LATCH_SECONDS = 0.3
```

Seconds the wheel may pause and still be the same gesture, which stays with the area it started in.

ui::FLING_DECAY_SECONDS

```
const float ui::FLING_DECAY_SECONDS = 0.325
```

Seconds for a fling's speed to fall to about a third: how far content glides after a touch drag.

ui::FLING_STOP_SPEED

```
const float ui::FLING_STOP_SPEED = 20
```

Points per second below which a fling stops.

BoxOpts

```
struct BoxOpts
```

Everything a box can be. Containers fill one in from their named arguments; widgets build one directly and call `Ui.@box_ex`. A zeroed `bg` is a token, not "no background": set it to `ui::NO_TOKEN` for none.

Field	Type	Meaning
`id`	`Id`	NO_ID: anonymous, no rect is kept
`width`	`Size`	
`height`	`Size`	
`dir`	`Dir`	
`pad`	`float`	spacing steps on every side
`pad_x`	`float`	overrides pad left and right when not 0
`pad_y`	`float`	overrides pad top and bottom when not 0
`lead`	`float`	points added to the left padding, for what the box's paint draws there (a toggle's control)
`gap`	`float`	spacing steps between children
`align`	`Align`	across the direction
`justify`	`Align`	along the direction
`bg`	`Token`	NO_TOKEN: nothing drawn
`gradient`	`Gradient*`	a linear gradient in place of `bg` (Tailwind's `bg-linear-*`); copied when the box opens
`radius`	`Corners`	
`shadow`	`Elevation`	drawn only when bg or gradient is set
`border`	`bool`	a 1 pt BORDER outline
`clip`	`bool`	hide what overflows
`clip_x`	`bool`	hide only what overflows left and right, so the box keeps its content's height
`paint`	`PaintFn`	draws under its children, over its own bg or gradient
`paint_data`	`void*`	

PaintFn

```
alias PaintFn = fn void(DrawList* list, Rect rect, void* data)
```

Draws a box into `list` after layout: `rect` is the box's final rectangle and `data` the pointer given with the function (`BoxOpts.paint_data`, or `Ui.custom`'s `data`).

Size

```
struct Size
```

A box's width or height. Build one with `ui::px`, `ui::pct`, `ui::grow` or `ui::fit`, or use `ui::FIT` and `ui::GROW`.

Field	Type	Meaning
kind	SizeKind	
value	float	FIXED: points; PERCENT: 0 to 1
min	float	FIT and GROW: smallest size in points
max	float	FIT and GROW: largest size in points; 0 = no limit

SizeKind

```
enum SizeKind : char
```

How a Size sizes a box along one axis.

Value	Meaning
FIT	wraps the content
GROW	takes the room left, shared with growing siblings
FIXED	exactly value points
PERCENT	value of the parent, less its padding and gaps

ui::px

```
fn Size ui::px(float points)
```

Exactly points.

ui::pct

```
fn Size ui::pct(float percent)
```

percent (0 to 100) of the parent, less its padding and gaps.

ui::grow

```
fn Size ui::grow(float min = 0, float max = 0)
```

Takes the room left, shared with growing siblings, within min and max points (max 0: no limit).

ui::fit

```
fn Size ui::fit(float min = 0, float max = 0)
```

Wraps the content, within min and max points (max 0: no limit).

ui::FIT

```
const Size ui::FIT = { .kind = SizeKind.FIT }
```

Wraps the content, with no limits; the default size of every container.

ui::GROW

```
const Size ui::GROW = { .kind = SizeKind.GROW }
```

Takes the room left, shared with growing siblings, with no limits.

`Dir`

```
enum Dir : char
```

The direction a box lays out its children in.

Value	Meaning
`ROW`	left to right
`COLUMN`	top to bottom

`Align`

```
enum Align : char
```

Where children sit within a box, along its direction (`justify`) or across it (`align`).

Value	Meaning
`START`	left or top
`CENTER`	centered
`END`	right or bottom

`Corners`

```
enum Corners : char
```

The corner radius of a box, picked from the theme's `Radii`.

Value	Meaning
`NONE`	square corners
`CONTROL`	the theme's control radius, as on buttons
`PANEL`	the theme's panel radius, as on cards
`SURFACE`	the theme's surface radius
`FULL`	fully round: a pill, or a circle when square

`Elevation`

```
enum Elevation : char
```

The shadow under a box with a background. Shadows are stronger in a dark theme.

Value	Meaning
`NONE`	no shadow
`SM`	a tight, faint shadow, as on cards
`MD`	a softer shadow, offset further down
`LG`	a wide, deep shadow

Ui.spacer

```
fn void Ui.spacer(&self, Size size = GROW)
```

Empty space `size` long along the parent's direction; by default it takes what the parent has left.

Ui.separator

```
fn void Ui.separator(&self)
```

A 1 pt `BORDER` line across the parent's direction, spanning the parent.

Ui.custom

```
fn void Ui.custom(
    &self,
    Size width,
    Size height,
    PaintFn paint,
    void* data = null,
    Id id = NO_ID,
    float aspect = 0)
```

A box drawn by `paint`, called with the box's final rectangle and `data` after layout. With an `id`, the rect is kept for `Ui.interact`. With an `aspect` above 0, layout keeps the box's width divided by its height at it.

Ui.bp

```
fn Breakpoint Ui.bp(&self)
```

The widest `Breakpoint` whose theme width the layout width reaches.

Ui.at

```
macro Ui.at(&self, Breakpoint at, wide, narrow)
```

`wide` at breakpoint `at` and above, `narrow` below it.

Ui.width

```
fn float Ui.width(&self)
```

The width of the container being built: the nearest open box that has an id, as it was laid out last frame, or the window's. A component asks it to fit the room its parent gives rather than the window.

Ui.height

```
fn float Ui.height(&self)
```

The height of the container being built, as `Ui.width` finds it.

Ui.width_at

```
macro Ui.width_at(&self, float min, wide, narrow)
```

`wide` when the container is at least `min` points wide (`Ui.width`), `narrow` below: `Ui.at` for the container.

Ui.@container

```
macro Ui.@container(&self, String id = "", float max = 0, float gap = 0; @body())
```

A column that takes the room it is given up to `max` points wide, centred in its parent. It always has an id, so `Ui.width` inside it reads its own width; `id` names it (`"container"` and a count when empty).

ui::PROSE

```
const float ui::PROSE = 640
```

Reading width for running text: about 65 characters.

ui::CONTAINER_SM

```
const float ui::CONTAINER_SM = 640
```

Container widths, as Tailwind's `container` steps.

Constant	Type	Value
ui::CONTAINER_MD	float	768
ui::CONTAINER_LG	float	1024
ui::CONTAINER_XL	float	1280

5 Layers

Ui.@layer

```
macro void Ui.@layer(
    &self,
    String id,
    bool* open,
    Id anchor = NO_ID,
    Side side = BELOW,
    Align align = START,
    float distance = 4,
    bool modal = false,
    Dismiss dismiss = AUTO,
    bool focus = true,
    bool backdrop = true,
    Size width = FIT,
    Size height = FIT,
    Dir dir = COLUMN,
    float pad = 2,
    float gap = 1,
    Token bg = SURFACE_RAISED,
    Corners radius = PANEL,
    Elevation shadow = LG,
    bool border = true,
    bool keep_open = false; @body())
```

Runs the body in a layer floating above the page while `*open` is true, and on while it fades out after `*open` turns false, without input then. With `anchor`, it opens on `side` of that widget, flipping to the other side when it doesn't fit and shifting to stay inside the window; without, it sits in the middle of the window. A modal layer blocks and dims the page. Escape, gamepad B, the system back gesture and a press outside (`dismiss`) set `*open` to false, unless `keep_open`; `Ui.dismissal` reports which.

- `keep_open`: Refuse Escape, B, back and presses outside, such as while the layer holds unsaved work; `Ui.dismissal` still reports them

Ui.@layer_ex

```
macro void Ui.@layer_ex(&self, String id, bool* open, LayerOpts opts; @body())
```

`Ui.@layer` with its options in a struct. The body runs in an id scope named `id`, like `Ui.@id`.

LayerOpts

```
struct LayerOpts
```

A layer's options, for `Ui.@layer_ex`. The zero value is a non-modal layer in the middle of the window that takes focus, closes on a press outside, and draws `box` as given.

Field	Type	Meaning
`anchor`	`Id`	the widget it opens next to, such as `Ui.last_id()`; NO_ID: the window's middle
`side`	`Side`	the side of the anchor it prefers
`align`	`Align`	along the anchor: START, CENTER or END

Field	Type	Meaning
distance	float	points between the anchor and the layer
modal	bool	blocks the page beneath, and dims it unless no_backdrop
dismiss	Dismiss	
no_focus	bool	takes no keyboard focus, as a tooltip: no focus scope
no_backdrop	bool	a modal layer that doesn't dim the page
arrow	bool	points at the anchor with a small triangle on the side facing it
passthrough	bool	takes no pointer: it passes through to what is beneath, as a tooltip
at_edge	bool	without an anchor: against the window's side edge, at align along it, as a toast
edge_offset	float	with at_edge: points further in from the edge, to stack layers
flush	bool	against the window's edges with no margin (a drawer), inside its safe area
at_point	bool	next to point instead of a widget, as a context menu: on side of it, distance away
point	float[<2>]	
back	BackMotion	what the system back gesture does: AUTO by default
keep_open	bool	refuses Escape, B, presses outside, back and swipes; Ui.dismissal still reports them
sheet	bool	a bottom sheet: as wide as the window, on its bottom edge, ignoring the layer margins
sheet_offset	float	with sheet: points it is dragged down from its resting place
box	BoxOpts	the layer's box; its id is the layer's

Side

```
enum Side : char
```

Which side of its anchor a layer opens on. START and END are left and right.

Value	Meaning
BELOW	
ABOVE	
START	
END	
OVER	over the anchor, distance points of the layer above its top: a select's list with its chosen row on the button

Dismiss

```
enum Dismiss : char
```

Whether a press outside a layer closes it.

Value	Meaning
AUTO	non-modal layers close, modal ones stay
OUTSIDE	any press outside closes it, on a modal's backdrop too
NEVER	only its own code, Escape or B close it

Dismissal

```
enum Dismissal : char
```

How the user dismissed a layer or focus scope, or tried to while it was kept open: what `Ui.dismissal` reports. The app closing it, a menu row and a dialog's buttons aren't dismissals.

Value	Meaning
NONE	not dismissed this frame
ESCAPE	Escape, or gamepad B
OUTSIDE	a press outside it, or on a modal's backdrop that dismisses
BACK	the system back gesture or button
SWIPE	a sheet dragged down past its threshold

Ui.dismissal

```
fn Dismissal Ui.dismissal(&self, String id)
```

How the user dismissed the layer or focus scope `id` (as named to `Ui.@layer`, `@dialog`, `@sheet` or `@focus_scope`, in the same id scope), or tried to while it was kept open; `NONE` on other frames. It is reported once, on the frame the app's code first sees `*open` false: for Escape and presses outside, the frame after the key or press.

Ui.current_layer

```
fn Id Ui.current_layer(&self)
```

The floating layer being built, or `ui::NO_ID` on the page.

ui::place_layer

```
fn Placement ui::place_layer(
    Rect anchor,
    float[<2>] size,
    Side side,
    Align align,
    float distance,
    Rect bounds)
```

Where a layer of `size` goes next to `anchor` inside `bounds`: on `side` if it fits there, else on the other side if it fits, else on whichever has more room; then shifted along both axes to stay inside `bounds`.

Placement

```
struct Placement
```

Where `ui::place_layer` puts a layer, and the side it took.

Field	Type	Meaning
pos	float[<2>]	
side	Side	

ui::LAYER_MARGIN

```
const float ui::LAYER_MARGIN = 8
```

Points between a layer and the window's edge.

ui::LAYER_SLIDE

```
const float ui::LAYER_SLIDE = 6
```

Points a layer slides in from its anchor's side while it appears.

ui::LAYER_ARROW_BASE

```
const float ui::LAYER_ARROW_BASE = 16
```

An arrow's size: its base along the layer's edge and its height out from it, in points.

Constant	Type	Value
ui::LAYER_ARROW_HEIGHT	float	8

6 Menus, popovers and tooltips

Ui.@menu

```
macro void Ui.@menu(
    &self,
    String id,
    bool* open,
    Id anchor = NO_ID,
    Side side = BELOW,
    Align align = START,
    Size width = ui::fit(ui::MENU_MIN_WIDTH),
    float distance = 4,
    bool at_point = false,
    float[<2>] point = {}; @body())
```

Runs the body as a menu in a layer while `*open` is true: rows from `Ui.menu_item`, `Ui.menu_separator` and `Ui.@submenu`. With `anchor`, it opens on `side` of that widget, flipping and shifting to stay inside the window, `distance` points from it (with `Side.OVER`, the layer's top is that far above the anchor's); with `at_point`, next to `point` instead, as a context menu. A press outside, Escape, B or choosing a row set `*open` to false.

Ui.menu_item

```
fn bool Ui.menu_item(
    &self,
    String label,
    Icon icon = NONE,
    Key key = NONE,
    bool primary = false,
    bool shift = false,
    bool alt = false,
    bool disabled = false,
    bool* checked = null)
```

A menu row: an optional icon, the label, and the shortcut's label on the right ("Ctrl+S", "⌘S" on macOS) when `key` isn't `Key.NONE`. The label only shows the shortcut; call `Ui.shortcut` for the key itself. With `checked`, the row shows a check mark and choosing it flips the value. True once when chosen, which also closes the menu and the menus it is in.

Ui.menu_separator

```
fn void Ui.menu_separator(&self)
```

A hairline between groups of menu rows.

Ui.@context_menu

```
macro void Ui.@context_menu(&self, String id; @body())
```

A context menu for the widget just built (`Ui.last_id`): it opens at the pointer on a secondary press (the right button), Ctrl+click with Mac keys or a long touch, and at the widget on the Menu key or Shift+F10; another request moves it. The body is a menu's: `Ui.menu_item`, `Ui.menu_separator`, `Ui.@submenu`.

Ui.@submenu

```
macro void Ui.@submenu(
    &self,
    String label,
    Icon icon = NONE,
    bool disabled = false; @body())
```

A menu row that opens the body as a submenu on its END side: when the pointer rests on it for the submenu delay, or at once with Right, Enter, Space or a click. Left or Escape close only the submenu; a row of this menu that the pointer rests on instead closes it too.

Ui.@menu_button

```
macro void Ui.@menu_button(
    &self,
    String label,
    Icon icon = NONE,
    Variant variant = DEFAULT,
    ControlSize size = MD,
    bool disabled = false,
    Align align = START; @body())
```

A button that opens the body as a menu below it; Kiln keeps whether it is open. Use `Ui.@menu` to open a menu from anything else.

Ui.@menu_icon_button

```
macro void Ui.@menu_icon_button(
    &self,
    Icon icon,
    String label,
    Variant variant = GHOST,
    ControlSize size = MD,
    bool disabled = false,
    Align align = END; @body())
```

`Ui.@menu_button` as an icon button; `label` names it for the id and, later, assistive technology.

Ui.@popover

```
macro void Ui.@popover(
    &self,
    String id,
    bool* open,
    Id anchor = NO_ID,
    Side side = BELOW,
    Align align = CENTER,
    bool arrow = false,
    Size width = FIT,
    Size height = FIT,
    float pad = 4,
    float gap = 2; @body())
```

Runs the body as a popover in a layer while `*open` is true: any content on a raised card next to `anchor`, flipping and shifting to stay inside the window. With `arrow`, a small triangle on the side facing the anchor points at it. A press outside, Escape or B set `*open` to false.

`Ui.tooltip`

```
fn bool Ui.tooltip(&self, String text)
```

A tooltip for the widget built just before (`Ui.last_id()`): `text` on a dark chip above it, flipping below. It shows once the pointer has rested on the widget, or the widget has had keyboard focus, for the tooltip delay (the OS's, where it has one); a touch shows it with a long press, which then doesn't click. It hides when that ends, on a press or on Escape, and never takes the pointer or focus. True while it shows.

`ui::MENU_MIN_WIDTH`

```
const float ui::MENU_MIN_WIDTH = 180
```

The narrowest a menu gets, in points.

`ui::MENU_ROW_HEIGHT`

```
const float ui::MENU_ROW_HEIGHT = 32
```

A menu row's height, in points.

7 Dialogs, sheets and toasts

Ui.@dialog

```
macro DialogResult Ui.@dialog(
    &self,
    String id,
    bool* open,
    String title,
    String confirm = "OK",
    String cancel = "Cancel",
    bool destructive = false,
    float width = 400,
    String focus = "",
    bool enter_confirms = true,
    float max_height = 0,
    String note = "",
    bool keep_open = false; @body())
```

Runs the body in a modal dialog while `*open` is true: `title` above it, and below it a button row, `confirm` and `cancel` (no cancelling button when empty), in the platform's order (`Ui.dialog_confirm_first`). Returns how it was answered this frame, and sets `*open` to false then. A `destructive` dialog's confirming button is DANGER. Focus starts on the default button: the confirming one, or the cancelling one in a destructive dialog, so Enter alone never confirms a risky action.

- `focus`: The label of the widget focus starts on, such as a field's; focus the app put inside is kept
- `enter_confirms`: Enter in a single-line field confirms, as Enter elsewhere does; false for a field whose Enter means something else
- `max_height`: The tallest the dialog may be, in points; 0 for no limit. Its body scrolls when it doesn't fit
- `note`: Text at the left of the button row, such as Review's count of files and changes
- `keep_open`: Refuse Escape, B, back and presses outside, such as while the dialog holds unsaved work; its buttons still answer it, and `Ui.dismissal` reports the attempt

DialogResult

```
enum DialogResult : char
```

How a dialog was answered this frame.

Value	Meaning
NONE	not answered this frame
CONFIRMED	the confirming button, or Enter
CANCELLED	the cancelling button, Escape, B, or closed some other way

Ui.@sheet

```
macro void Ui.@sheet(
    &self,
    String id,
    bool* open,
    SheetWide wide = DIALOG,
```

```
    Id anchor = NO_ID,
    Side side = BELOW,
    Align align = END,
    float width = 400,
    float pad = 4,
    float gap = 2,
    float max_height = 0,
    bool keep_open = false; @body())
```

Runs the body in a bottom sheet while `*open` is true: a modal panel at the window's bottom edge, as wide as the window, with a grab handle. Below `MD` the sheet rises from the edge, and dragging its handle down closes it; at `MD` and above the same body is a centred dialog, or a popover beside `anchor` with `wide: POPOVER`. The platform's bottom safe-area inset stays clear under the content, while the sheet's background runs to the edge. Dragging it down past the threshold is a `SWIPE` dismissal.

- `keep_open`: Refuse swipes, Escape, B, back and presses outside; `Ui.dismissal` still reports them

Ui.@sheet_ex

```
macro void Ui.@sheet_ex(&self, String id, bool* open, SheetOpts opts; @body())
```

`Ui.@sheet` with its options in a `SheetOpts`, for an app forwarding its own options. A zero `width` is 400 points; the other zero fields are their zero values (a dialog, `BELOW`, `START`, no padding or gap, and the window's cap).

SheetOpts

```
struct SheetOpts
```

A sheet's options, for `Ui.@sheet_ex`: how it shows at wide widths, where a popover goes, and the dialog's size and padding. `Ui.@sheet` fills in the usual values.

Field	Type	Meaning
`wide`	`SheetWide`	DIALOG at MD and above, or POPOVER by `anchor`
`anchor`	`Id`	the widget a popover opens next to; NO_ID: a dialog
`side`	`Side`	the popover's side of the anchor
`align`	`Align`	how the popover lines up along it
`width`	`float`	the dialog's or popover's width in points; 0 is 400
`pad`	`float`	spacing steps around the body
`gap`	`float`	spacing steps between the body's widgets
`max_height`	`float`	the tallest it may be in points; 0 caps it to the window
`keep_open`	`bool`	refuses swipes, Escape, B, back and presses outside; `Ui.dismissal` still reports them

Ui.sheet_item

```
fn bool Ui.sheet_item(
    &self,
    String label,
    Icon icon = NONE,
```

```
    bool disabled = false,
    bool checked = false,
    Tone tone = Tone.NEUTRAL)
```

A row of a sheet: a full-width action with an icon, a label and an optional check at its end. It is a focus stop, activated by Enter, Space or gamepad A, and reports whether it was chosen. `tone: DANGER` draws a destructive action in the danger color.

SheetWide

```
enum SheetWide : char
```

How a sheet appears at wide widths, at `MD` and above.

Value	Meaning
`DIALOG`	a centred modal card, with the page dimmed behind it
`POPOVER`	a popover beside `anchor`, or a dialog when there is no anchor

Ui.toast

```
fn void Ui.toast(
    &self,
    String text,
    String action = "",
    String id = "",
    float seconds = 4,
    bool notify_if_unfocused = false)
```

Raises a toast with `text` and, optionally, an `action` button; it leaves after `seconds`, sooner when more than `ui::TOAST_LIMIT` show. Give it an `id` to ask `Ui.toast_action(id)` whether the action was pressed. With `notify_if_unfocused`, a toast raised while the window doesn't have the focus also goes out as a system notification (`os::notify`), so it isn't missed. Draw the toasts with `Ui.toasts`.

Ui.toasts

```
fn void Ui.toasts(&self, ToastPlace place = BOTTOM_CENTER)
```

Draws the toasts raised with `Ui.toast`, at `place`, the newest nearest the edge; call it once a frame, after the page, so they sit above it.

Ui.toast_action

```
fn bool Ui.toast_action(&self, String id)
```

True once after the action of the toast raised with `id` was pressed.

Ui.toast_count

```
fn int Ui.toast_count(&self)
```

How many toasts are showing, not counting ones fading out.

`ToastPlace`

```
enum ToastPlace : char
```

Where toasts appear, for `Ui.toasts`.

Value	Meaning
`BOTTOM_CENTER`	GNOME, Android and Material on the web put them here
`BOTTOM_END`	
`TOP_CENTER`	
`TOP_END`	

`NotifyFn`

```
alias NotifyFn = fn void?(String title, String body)
```

A system notification request: `os::notify`, or a stand-in in tests.

`ui::TOAST_LIMIT`

```
const int ui::TOAST_LIMIT = 3
```

At most this many toasts show at once; a new one sends the oldest away.

`ui::TOAST_TEXT`

```
const usz ui::TOAST_TEXT = 160
```

Bytes of a toast's text kept; longer text is cut at a character boundary.

`ui::TOAST_ACTION`

```
const usz ui::TOAST_ACTION = 32
```

Bytes of a toast's action label kept.

`ui::TOAST_WIDTH`

```
const float ui::TOAST_WIDTH = 360
```

The widest a toast gets, in points; longer text wraps.

8 Text editing

TextEdit

```
struct TextEdit
```

A caret, a selection and an undo history for one text.

Field	Type	Meaning
caret	usz	byte offset where typing goes
anchor	usz	the other end of the selection; equal to caret when none
breaks	Breaks	
steps	List{EditStep}	
done	usz	steps[:done] are applied; the rest can be redone
allocator	Allocator	

TextEdit.init

```
fn void TextEdit.init(&self, Allocator allocator)
```

Prepares an empty edit, keeping its boundaries and undo history in `allocator`.

TextEdit.free

```
fn void TextEdit.free(&self)
```

Frees the boundaries and the undo history.

TextEdit.selection

```
fn usz[2] TextEdit.selection(&self)
```

The selection as a byte range, start first; empty when nothing is selected.

TextEdit.has_selection

```
fn bool TextEdit.has_selection(&self)
```

Whether any text is selected.

TextEdit.sync

```
fn void TextEdit.sync(&self, DString* text)
```

Brings the caret, selection and boundaries up to date with `text`, which the app may have changed: positions past the end move to it, and positions inside a grapheme move to its start.

TextEdit.move

```
fn void TextEdit.move(&self, DString* text, EditMove m, bool extend = false)
```

Moves the caret; with `extend` the anchor stays, growing the selection. A plain left or right move from a selection lands on its edge.

`TextEdit.place`

```
fn void TextEdit.place(&self, DString* text, usz pos, bool extend = false)
```

Places the caret at `pos` (a click), or extends the selection there.

`TextEdit.select_word`

```
fn void TextEdit.select_word(&self, DString* text, usz pos)
```

Selects the word, or run of spaces, around `pos` (a double-click).

`TextEdit.select_all`

```
fn void TextEdit.select_all(&self, DString* text)
```

Selects all of the text (a triple-click, Ctrl+A).

`TextEdit.selected`

```
fn String TextEdit.selected(&self, DString* text)
```

The selected text, a view into `text`.

`TextEdit.seal`

```
fn void TextEdit.seal(&self)
```

Typing ends its undo step: the next typed text starts a new one.

`TextEdit.type`

```
fn void TextEdit.type(&self, DString* text, String typed, double time)
```

Replaces the selection with `typed` at `time`, as typing: it joins the last undo step while typing continues, unless a pause or a space after a word starts a new one.

`TextEdit.replace_selection`

```
fn void TextEdit.replace_selection(&self, DString* text, String with, double time)
```

Replaces the selection with `with`, as one undo step (a paste, a cut with `with` empty).

`TextEdit.delete_back`

```
fn void TextEdit.delete_back(
    &self,
    DString* text,
    EditUnit unit,
    double time,
    usz line_start = 0)
```

Deletes the selection, or else `unit` before the caret (Backspace); TO_START deletes back to `line_start`.

`TextEdit.delete_forward`

```
fn void TextEdit.delete_forward(&self, DString* text, EditUnit unit, double time)
```

Deletes the selection, or else `unit` after the caret (Delete).

`TextEdit.undo`

```
fn bool TextEdit.undo(&self, DString* text)
```

Undoes the last step; false when there is none.

`TextEdit.redo`

```
fn bool TextEdit.redo(&self, DString* text)
```

Redoes the last undone step; false when there is none.

`EditMove`

```
enum EditMove : char
```

Where a caret move goes.

Value	Meaning
`LEFT`	the previous grapheme
`RIGHT`	the next grapheme
`WORD_LEFT`	the start of the word before
`WORD_RIGHT`	the start of the next word (Windows, Linux)
`WORD_END`	the end of the word (macOS)
`START`	the start of the text (one line: Home, ⌘←)
`END`	the end of the text

`EditUnit`

```
enum EditUnit : char
```

How much a delete removes, when nothing is selected.

Value	Meaning
`GRAPHEME`	
`WORD`	
`TO_START`	to the start of the line (⌘Backspace): `line_start`

`ui::UNDO_LIMIT`

```
const usz ui::UNDO_LIMIT = 100
```

The most undo steps a `TextEdit` keeps; older ones are dropped.

ui::UNDO_PAUSE

```
const double ui::UNDO_PAUSE = 1
```

Seconds of no typing after which typing starts a new undo step.

Ui.text_field

```
fn TextFieldResult Ui.text_field(
    &self,
    String label,
    DString* value,
    String placeholder = "",
    Icon icon = NONE,
    ControlSize size = MD,
    bool disabled = false,
    bool password = false,
    Size width = GROW,
    bool context_menu = true,
    bool read_only = false,
    InputKind kind = TEXT)
```

A single-line text field editing `value`, the app's string (initialized in `mem` if it isn't yet). `label` names it for its id and, later, assistive technology; `placeholder` shows while it is empty. A `password` field shows bullets and won't copy. A secondary press, a long touch or the Menu key opens its edit menu, unless `context_menu` is false. A `read_only` field can be focused, selected and copied, but not edited.

Ui.set_text_input_area

```
fn void Ui.set_text_input_area(&self, Rect area, float caret)
```

Tells the platform where text entry is happening, in points: the text area and the caret.

Ui.text_field_action

```
fn bool Ui.text_field_action(
    &self,
    String label,
    DString* value,
    FieldAction action,
    bool multiline = false)
```

Does `action` in the field named `label`: the edit menu's Undo, Cut, Copy and the rest.

Ui.text_field_selection_range

```
fn usz[2] Ui.text_field_selection_range(&self, String label)
```

The field's selection as byte offsets, start first; empty when nothing is selected.

Ui.text_field_caret_at

```
fn usz Ui.text_field_caret_at(&self, String label)
```

The field's caret as a byte offset.

Ui.text_field_set_caret

```
fn bool Ui.text_field_set_caret(&self, String label, DString* value, usz at)
```

Puts the field's caret at byte `at` and focuses it.

Ui.text_field_middle_paste

```
fn bool Ui.text_field_middle_paste(
    &self,
    String label,
    DString* value,
    bool multiline = false)
```

Pastes the primary selection at the field's caret.

Ui.text_field_composing

```
fn String Ui.text_field_composing(&self)
```

The input method's current composition, empty when it isn't composing.

TextFieldResult

```
struct TextFieldResult
```

What a text field did this frame.

Field	Type	Meaning
`changed`	`bool`	the text changed
`submitted`	`bool`	Enter was pressed in it
`focused`	`bool`	it has the keyboard focus
`left`	`bool`	the focus left it this frame
`cleared`	`bool`	a search field's or Escape emptied it

Ui.text_area

```
fn TextFieldResult Ui.text_area(
    &self,
    String label,
    DString* value,
    String placeholder = "",
    int rows = 3,
    int max_rows = 8,
    ControlSize size = MD,
    bool disabled = false,
    bool read_only = false,
    Size width = GROW,
    bool context_menu = true)
```

A multi-line text area editing `value`, the app's string (initialized in `mem` if it isn't yet). `label` names it for its id; `placeholder` shows while it is empty. It is `rows` lines tall, grows with its text up to `max_rows` lines, then scrolls. A `read_only` area can be focused, selected and copied, but not edited.

ui::CARET_BLINK_TIMEOUT

```
const double ui::CARET_BLINK_TIMEOUT = 10
```

Seconds without typing after which the caret stops blinking and stays on, as on GNOME.

Ui.field_ex

```
fn TextFieldResult Ui.field_ex(&self, String label, DString* value, FieldOpts o)
```

The text field with all its options: what `Ui.text_field` builds, and what the search and number fields build on.

InputKind

```
enum InputKind : char
```

What a field holds, so the OS can offer the right on-screen keyboard and input method.

Value	Meaning
TEXT	any text
NAME	a person's name
EMAIL	an email address
USERNAME	a user name: not capitalized or corrected
NUMBER	a number: a numeric keypad where there is one

FieldOpts

```
struct FieldOpts
```

A text field's options, for widgets built on it (`Ui.search_field`, `Ui.number_field`).

Field	Type	Meaning
placeholder	String	
icon	Icon	
size	ControlSize	
disabled	bool	
password	bool	
width	Size	ui::GROW for the usual field: a zero Size is FIT
filter	CharFilter	null: every character but controls
filter_data	void*	
trailing	FieldTrailing	null: nothing after the text
trailing_data	void*	
keys	KeySet	more keys the field takes while focused, which its widget reads after it
no_context_menu	bool	no edit menu on a secondary press, a long touch or the Menu key
read_only	bool	selectable and copyable, not editable
multiline	bool	a text area: paste keeps line breaks

Field	Type	Meaning
kind	InputKind	what it holds, for the on-screen keyboard and input method

CharFilter

```
alias CharFilter = fn bool(void* data, uint codepoint)
```

Whether a field accepts `codepoint`, typed or pasted; `data` is `FieldOpts.filter_data`.

FieldTrailing

```
alias FieldTrailing = fn void(Ui* ui, void* data)
```

Builds widgets inside a field's box, after its text: a clear button, steppers. `data` is `FieldOpts.trailing_data`.

Ui.paste_text

```
fn void Ui.paste_text(&self, TextEdit* ed, DString* value, String pasted, FieldOpts* o)
```

Puts pasted text in place of the selection, or at the caret: line breaks become LF in an area and spaces in a one-line field, and the field's filter applies, as when typing.

Ui.offer_primary

```
fn void Ui.offer_primary(&self, Id id, String text, usz[2] range)
```

Makes a field's or a text's selection the primary selection, the text a middle click pastes on Linux and the BSDs, once per selection made: called each frame a non-empty selection stands and the pointer isn't dragging. Never for a password field.

Ui.middle_paste

```
fn bool Ui.middle_paste(
    &self,
    Id id,
    TextEdit* ed,
    DString* value,
    usz at,
    FieldOpts* o)
```

Pastes the primary selection where a middle click landed in an editable field or area, at byte `at`, and focuses it, on Linux and the BSDs unless `Ui.middle_click_paste` is off. True when it pasted.

Ui.set_middle_click_paste

```
fn void Ui.set_middle_click_paste(&self, bool on)
```

Turns pasting the primary selection with a middle click on or off, whatever the OS says, until `Ui.follow_os(MIDDLE_CLICK_PASTE)`.

ui::HAS_PRIMARY_SELECTION

```
const bool ui::HAS_PRIMARY_SELECTION = true
```

Whether the OS has a primary selection that a middle click pastes: Linux and the BSDs.

9 Inputs

`Ui.search_field`

```
fn TextFieldResult Ui.search_field(
    &self,
    String label,
    DString* value,
    String placeholder = "Search",
    ControlSize size = MD,
    bool disabled = false,
    Size width = GROW)
```

A search field editing `value`; `cleared` in the result says the clear button or Escape emptied it this frame (`changed` is set too).

`Ui.number_field`

```
fn NumberResult Ui.number_field(
    &self,
    String label,
    double* value,
    double min = -1e15,
    double max = 1e15,
    double step = 0,
    String format = "%g",
    ControlSize size = MD,
    bool disabled = false,
    Size width = GROW)
```

A number field editing `*value`, between `min` and `max`, in steps of `step` (0: any value; the keys and steppers then move by 1). `format` is a printf format for one double.

Requires:

- `min <= max`

`NumberResult`

```
bitstruct NumberResult : char
```

What a number field did this frame.

Field	Type	Meaning
`changed`	`bool`	the value changed: a step, or typed text committed
`submitted`	`bool`	Enter was pressed in it

`ui::STEP_REPEAT_DELAY`

```
const double ui::STEP_REPEAT_DELAY = 0.4
```

Seconds a held stepper waits before it repeats, and between repeats.

Constant	Type	Value
`ui::STEP_REPEAT_EVERY`	`double`	`0.06`

`Ui.format_number`

```
fn String Ui.format_number(&self, double v, String format)
```

`v` written with `format`, its `.` replaced by `separator`, in the temp allocator.

`Ui.parse_number`

```
fn double? Ui.parse_number(&self, String text)
```

The number `text` holds, written with `.` or the user's separator; a fault when it isn't one.

`Ui.select`

```
fn bool Ui.select(
    &self,
    String label,
    int* selected,
    String[] options,
    ControlSize size = MD,
    bool disabled = false,
    bool full = false)
```

A select choosing `*selected`, an index into `options`; true once when the choice changes. `label` names it and is its id. `full` makes it as wide as its container.

Requires:

- `options.len > 0`

`Ui.date_picker`

```
fn bool Ui.date_picker(
    &self,
    String label,
    Date* date,
    Date min = {},
    Date max = {},
    String placeholder = "Choose a date",
    ControlSize size = MD,
    bool disabled = false,
    bool full = false)
```

A date picker choosing `*date`; true once when the date changes. `label` names it and is its id. The button shows `placeholder` while no date is set. `min` and `max`, when set, bound the dates that can be chosen.

`Date`

```
struct Date
```

A day of the Gregorian calendar; the zero value is no date.

Field	Type	Meaning
`year`	`short`	
`month`	`char`	1–12

Field	Type	Meaning
day	`char`	1–31

`Date.is_set`

```
fn bool Date.is_set(Date self)
```

Whether this is a date rather than the zero value.

`Date.equals`

```
fn bool Date.equals(Date self, Date other)
```

Whether the two are the same day.

`Date.compare`

```
fn int Date.compare(Date self, Date other)
```

Negative when this date is earlier than `other`, zero when they are the same day, positive when later.

`Date.weekday`

```
fn int Date.weekday(Date self)
```

0 Sunday … 6 Saturday.

`Date.add_days`

```
fn Date Date.add_days(Date self, int n)
```

`n` days later, or earlier when negative.

`Date.add_months`

```
fn Date Date.add_months(Date self, int n)
```

`n` months later (or earlier, when negative), on the same day or the month's last when it is shorter: 31 January + 1 is 28 or 29 February.

`Date.month_start`

```
fn Date Date.month_start(Date self)
```

The first day of this date's month.

`Date.days`

```
fn long Date.days(Date self)
```

Days since 1 January 1970: Hinnant's `days_from_civil`.

`ui::date_from_days`

```
fn Date ui::date_from_days(long days)
```

The date `days` days after 1 January 1970: Hinnant's `civil_from_days`.

`ui::today`

```
fn Date ui::today()
```

Today, in the user's time zone.

`ui::is_leap_year`

```
fn bool ui::is_leap_year(int year)
```

Whether `year` has a 29 February: every fourth year, except centuries not divisible by 400.

`ui::days_in_month`

```
fn int ui::days_in_month(int year, int month)
```

How many days `month` (1–12) of `year` has.

`Ui.format_date`

```
fn String Ui.format_date(&self, Date d, Allocator allocator = tmem)
```

`d` as the user's region writes a short date (25/11/2026, 11/25/2026, 2026-11-25), in `allocator`; "" for no date.

`Ui.time_field`

```
fn bool Ui.time_field(
    &self,
    String label,
    TimeOfDay* time,
    int minute_step = 1,
    bool seconds = false,
    ControlSize size = MD,
    bool disabled = false,
    bool full = false)
```

A time field choosing `*time`; true on each frame the time changes. `label` names it and is its id. `minute_step` is what Up and Down move the minute by; `seconds` adds a part for them.

`TimeOfDay`

```
struct TimeOfDay
```

A time of day on a 24-hour clock, with no date and no time zone; the zero value is midnight.

Field	Type	Meaning
hour	char	0–23
minute	char	0–59
second	char	0–59

`TimeOfDay.equals`

```
fn bool TimeOfDay.equals(TimeOfDay self, TimeOfDay other)
```

Whether the two are the same time.

`TimeOfDay.seconds`

```
fn int TimeOfDay.seconds(TimeOfDay self)
```

Seconds since midnight, to compare and sort times.

`ui::now_time`

```
fn TimeOfDay ui::now_time()
```

The time now, in the user's time zone.

`Ui.format_time`

```
fn String Ui.format_time(
    &self,
    TimeOfDay t,
    bool seconds = false,
    Allocator allocator = tmem)
```

`t` as the user's region writes it, in `allocator`: "09:05" on a 24-hour clock, "9:05 AM" on a 12-hour one, with the region's separator and day period words, and the seconds when `seconds` is set.

`Ui.color_picker`

```
fn bool Ui.color_picker(
    &self,
    String label,
    Color* color,
    bool alpha = false,
    Color[] swatches = {},
    ControlSize size = MD,
    bool disabled = false)
```

A color picker choosing `*color`; true on each frame the color changes. `label` names it and is its id. `alpha` adds a bar for the opacity and writes the hex value with it; `swatches` are colors offered to choose with one press.

`Ui.calendar`

```
fn CalendarSettings* Ui.calendar(&self)
```

The user's calendar settings: read from the OS the first time a picker asks, not when the `Ui` starts, so an app without a picker never reads them.

`Ui.set_calendar`

```
fn void Ui.set_calendar(&self, CalendarSettings settings)
```

Uses a copy of `settings` in place of the OS's, as tests do.

Ui.segmented

```
fn bool Ui.segmented(
    &self,
    String id,
    int* selected,
    String[] options,
    Icon[] icons = {},
    bool icons_only = false,
    ControlSize size = MD,
    bool disabled = false)
```

A segmented control choosing one of `options` into `*selected`; true when the choice changed. With `icons` (one per option), a segment shows its icon, and its label too unless `icons_only`; the label still names it.

Requires:

- `icons.len == 0 || icons.len == options.len`

Ui.type_ahead

```
fn int Ui.type_ahead(&self, TypeAhead* ta, String[] labels, int from)
```

Takes this frame's typed text and returns the row of `labels` it leads to, searching from the row after `from` and going round; -1 when nothing was typed or no label matches. Matching ignores ASCII case.

Ui.type_ahead_rows

```
fn int Ui.type_ahead_rows(
    &self,
    TypeAhead* ta,
    int count,
    ListText text,
    void* data,
    int[] order,
    int from)
```

Type-ahead over `count` rows whose labels come from `text`, in display order: `order` maps a position to a row (empty: the same). Returns a position, or -1.

TypeAhead

```
struct TypeAhead
```

What has been typed so far; keep one per list in its state.

Field	Type	Meaning
`typed`	`char[32]`	
`len`	`usz`	
`last`	`double`	when the last character came

ui::TYPE_AHEAD_PAUSE

```
const double ui::TYPE_AHEAD_PAUSE = 1
```

Seconds between typed characters after which type-ahead starts a new search.

10 Navigation

Ui.tabs

```
fn bool Ui.tabs(
    &self,
    String id,
    int* selected,
    String[] labels,
    Icon[] icons = {},
    ControlSize size = MD,
    bool keys = true,
    int[] badges = {})
```

A tab bar choosing one of `labels` into `*selected`; true when the choice changed. With `icons` (one per label), each tab shows its icon before the label. `keys: false` leaves Ctrl+Page Up/Down and the shoulder buttons to another tab bar on the same page. With `badges` (one per label), a tab shows a count after its label (above 0) or a dot (0); none below 0.

Requires:

- `icons.len == 0 || icons.len == labels.len`
- `badges.len == 0 || badges.len == labels.len`
- `labels.len > 0`

Ui.@sidebar

```
macro void Ui.@sidebar(
    &self,
    String id,
    String title = "",
    SidebarMode mode = SidebarMode.AUTO,
    float width = 220,
    bool resizable = false,
    float min_width = 160,
    float max_width = 360,
    bool* drawer = null; @body())
```

A sidebar holding the items and sections its body builds (`Ui.sidebar_item`, `Ui.sidebar_section`); `title` heads it in full mode and in the drawer. `width` is the full sidebar's initial width and the drawer's width. With `resizable`, the full sidebar has a grip at its end edge: dragging it sets the width between `min_width` and `max_width`, and a double-click puts `width` back. Icons mode stays `SIDEBAR_ICONS_WIDTH` wide, and the drawer keeps `width` clamped to leave a strip of the page. With `drawer`, the app holds the drawer's open flag (to open it from a shortcut, say); without it, the sidebar keeps it.

Ui.sidebar_resize_grip

```
fn void Ui.sidebar_resize_grip(
    &self,
    Id panel,
    float* width,
    float current,
    float reset_width,
```

```
    float min_width,
    float max_width)
```

A drag grip at the end edge of `panel`: dragging it sets `width` between `min_width` and `max_width`, and a double-click sets it to `reset_width`. `current` is the width the panel was drawn at, used to seed the drag and place the grip. The caller keeps `width` and reads it on the next frame.

`Ui.sidebar_item`

```
fn bool Ui.sidebar_item(
    &self,
    String label,
    Icon icon,
    bool selected = false,
    int badge = -1)
```

An item of the sidebar being built: `icon` and `label`, tinted with an accent bar when `selected`. True when chosen, which also closes a drawer. `label` is its id. `badge` shows a count at the row's end (above 0) or a dot (0), and a dot on the icon in icons mode; none below 0.

`Ui.sidebar_section`

```
fn void Ui.sidebar_section(&self, String title)
```

A heading over the items that follow: a small label in full mode, a separator in icon mode.

`Ui.sidebar_group`

```
fn SidebarGroupAction Ui.sidebar_group(
    &self,
    String label,
    Icon icon,
    bool* open,
    bool label_opens = false)
```

A group of sidebar items: `icon`, `label` and a chevron, while `*open` says whether the items below it are shown. A click on the chevron toggles; a click on the label toggles too, unless `label_opens`, when it returns `ACTIVATED` for the caller to open the collection's page. Enter and Space follow the label; Left closes and Right opens while the row is focused. In icons mode only the icon shows, with the label as its tooltip.

`SidebarMode`

```
enum SidebarMode : char
```

How a sidebar shows.

Value	Meaning
`AUTO`	by the window: FULL at LG and up, ICONS from MD, DRAWER below
`FULL`	icons and labels
`ICONS`	icons only, the labels as tooltips
`DRAWER`	a menu button that opens the full sidebar over the page

`SidebarGroupAction`

```
enum SidebarGroupAction : char
```

What a sidebar group row did this frame.

Value	Meaning
`NONE`	
`TOGGLED`	the group opened or closed
`ACTIVATED`	the label asked for its page

`ui::SIDEBAR_ICONS_WIDTH`

```
const float ui::SIDEBAR_ICONS_WIDTH = 56
```

Points wide in ICONS mode.

`Ui.@disclosure`

```
macro void Ui.@disclosure(&self, String title, bool* open; @body())
```

A section under a header titled `title` that opens and closes it (`*open`): a click, Enter, Space or gamepad A on the header. The body slides open to its own height. `title` is its id.

`Ui.@accordion`

```
macro void Ui.@accordion(&self, String id, int* open_index; @body())
```

Runs the body's `Ui.@section`s as an accordion: at most one open at a time, its index in `*open_index` (-1: none). Opening a section closes the one open before.

`Ui.@section`

```
macro void Ui.@section(&self, String title; @body())
```

One section of the accordion being built: a disclosure that is open when it is the accordion's open section.

Requires:

- `self.accordion.open != null : "a section belongs inside ui.@accordion"`

11 Collections

`Selection`

```
struct Selection
```

The rows selected in a list or table, by the app's row index, and the anchor Shift ranges start from; `init` it once and `free` it when done.

Field	Type	Meaning
`bits`	`List{ulong}`	
`anchor`	`int`	the row ranges start from; -1 for none
`count`	`int`	rows selected

`Selection.init`

```
fn void Selection.init(&self, Allocator allocator = mem)
```

Gets the selection ready, empty.

`Selection.free`

```
fn void Selection.free(&self)
```

Frees what the selection holds.

`Selection.has`

```
fn bool Selection.has(&self, int i)
```

Whether row `i` is selected.

`Selection.set`

```
fn void Selection.set(&self, int i, bool on = true)
```

Selects row `i` (`on`) or leaves it out.

`Selection.clear`

```
fn void Selection.clear(&self)
```

Selects nothing; the anchor goes too.

`Selection.add_range`

```
fn void Selection.add_range(&self, int a, int b)
```

Selects rows `a` to `b`, both included, in either order.

`Selection.select_all`

```
fn void Selection.select_all(&self, int count)
```

Selects every row of `count`.

`Selection.select_only`

```
fn void Selection.select_only(&self, int i)
```

Selects row `i` alone, and anchors there.

`Selection.first`

```
fn int Selection.first(&self)
```

The first selected row; -1 for none.

`Selection.remove`

```
fn void Selection.remove(&self, int[] rows)
```

Takes `rows` (in any order) out of the selection, after the app removed them from its list: each row after them moves up by the removed rows before it, as the app's indices do. The anchor moves with its row, or is -1 when its row went.

`Selection.@each`

```
macro void Selection.@each(&self; @body())
```

Runs the body for each selected row, in order.

`Ui.@list`

```
macro void Ui.@list(
    &self,
    String id,
    bool dividers = false,
    int* selected = null,
    Selection* selection = null,
    bool reorder = false; @body())
```

A list of the rows its body builds with `Ui.@list_row`, every one of them; for long lists use `Ui.@long_list`. With `selected`, one row is selected; with `selection`, several can be. `dividers` draws hairlines between rows.

`Ui.@list_row`

```
macro void Ui.@list_row(&self, int row; @body())
```

A row of the list being built, for the app's row `row` (rows are built in the order they show). Its body lays out in a row; widgets in it take the pointer but aren't Tab stops of their own.

`Ui.@long_list`

```
macro void Ui.@long_list(
    &self,
    String id,
    int count,
    float row_height,
```

```
    Size height = ui::GROW,
    bool dividers = false,
    int* selected = null,
    Selection* selection = null,
    ListText text = null,
    void* data = null,
    bool reorder = false; @body())
```

A list of `count` rows `row_height` points tall that builds only the rows in view, in a scroll area of its own (`height` tall), so its length costs nothing. The body builds the row it is given. `text`, when set, gives each row's label for type-ahead.

`Ui.@row_menu`

```
macro void Ui.@row_menu(&self, String id; @body())
```

A context menu for every row of the list, long list or table `id`, declared once after it, every frame, in the same id scope:

```
c3 ui.@row_menu("songs"; int row) {
```

```
if (ui.menu_item("Play")) play(row);
```

};

It opens at the pointer on a secondary press or a long touch, and under the row on the Menu key or Shift+F10. The body gets the app's row, not its display position, so a sorted table still names the row the user meant. A widget inside a row with no menu of its own falls back to this menu; one with its own keeps it. The open state and point belong to the list, so the menu stays open when its row scrolls out of view. A list uses either this or a `Ui.@context_menu` on each row, not both.

`ListText`

```
alias ListText = fn String(void* data, int row)
```

The text `row` shows, for type-ahead in a list built row by row.

`Ui.list_activated`

```
fn int Ui.list_activated(&self, String id)
```

The row activated this frame in the list `id` (Enter, a double-click, gamepad A); -1 for none. Ask after the list.

`Ui.list_cursor`

```
fn int Ui.list_cursor(&self, String id)
```

The row the list `id`'s cursor is on; -1 before it has one.

`Ui.list_select`

```
fn void Ui.list_select(&self, String id, int row)
```

Moves the list `id`'s cursor to `row`, if the list has that row.

Ui.list_removed

```
fn void Ui.list_removed(&self, String id, int[] rows)
```

Tells the list or table `id` that the app removed `rows` (its own row indices, as they were) from its data, so the cursor stays on its row, which moves up by the removed rows before it. When the cursor's own row went, it takes the row after the removed ones; in a sorted table, the row shown in its place. Call it when the rows go, with the app's `Selection.remove` for the selection.

Reorder

```
struct Reorder
```

A move of rows within a list: report it with `Ui.list_reordered`, apply it with `Reorder.apply`.

Field	Type	Meaning
`rows`	`int[]`	the rows moved, in their order (valid for the frame it is reported)
`before`	`int`	the row they now go before, which isn't one of them; -1: the end
`count`	`int`	rows in the list, for where rows go at the end (`Reorder.new_index`)

Ui.list_reordered

```
fn Reorder? Ui.list_reordered(&self, String id)
```

The move reported this frame by the list or table `id`; a fault on other frames. Ask after the list.

Reorder.apply

```
macro void Reorder.apply(Reorder self, items)
```

Moves the rows of `items` as the reorder says, in place, keeping the moved rows in their order. It allocates nothing; each moved row costs a rotation of the rows between it and where it lands.

Reorder.new_index

```
fn int Reorder.new_index(&self, int row)
```

Where `row` (before the move) is once the move is applied.

Reorder.moves

```
fn bool Reorder.moves(&self, int row)
```

Whether `row` is one of the rows moved.

Ui.list_row_id

```
fn Id Ui.list_row_id(&self, Id list, int row)
```

The id of `row` in the list `list`.

Ui.table

```
fn void Ui.table(
    &self,
    String id,
    TableColumn[] columns,
    int rows,
    void* data,
    TableText text,
    int* selected = null,
    Selection* selection = null,
    TableSort* sort = null,
    float[] widths = {},
    Size height = GROW,
    float row_height = TABLE_ROW_HEIGHT,
    bool reorder = false,
    int search_column = -1)
```

A table of `rows` rows over `data`, each cell showing `text`. With `selected`, one row is selected; with `selection`, several can be. `sort` pins the sort (the app keeps it); `widths`, one per column (0 for the column's own), pins the dragged widths.

Ui.@table

```
macro void Ui.@table(
    &self,
    String id,
    TableColumn[] columns,
    int rows,
    void* data,
    TableText text,
    int* selected = null,
    Selection* selection = null,
    TableSort* sort = null,
    float[] widths = {},
    Size height = ui::GROW,
    float row_height = ui::TABLE_ROW_HEIGHT,
    bool reorder = false,
    int search_column = -1; @body())
```

`Ui.table` whose body builds each cell: it is given the row and the column. `text` still sorts and finds rows.

TableColumn

```
struct TableColumn
```

A column of a table.

Field	Type	Meaning
`title`	`String`	
`width`	`Size`	px, pct or GROW (FIT is taken as GROW, since rows must line up)
`min_width`	`float`	the narrowest a drag makes it; 0 for 40 points
`align`	`Align`	its cells' text: START, CENTER or END (numbers)

Field	Type	Meaning
`unsortable`	`bool`	its header doesn't sort
`compare`	`TableCompare`	null: the default comparison of the cells' text

`TableSort`

```
struct TableSort
```

How a table is sorted; the zero value is the app's own order.

Field	Type	Meaning
`sorted`	`bool`	
`column`	`int`	
`descending`	`bool`	

`TableText`

```
alias TableText = fn String(void* data, int row, int col)
```

A cell's text: drawn by `Ui.table`, and read by sorting and type-ahead.

`TableCompare`

```
alias TableCompare = fn int(void* data, int a, int b, int col)
```

Negative when row `a` goes before row `b` in column `col`, zero when they tie, positive otherwise.

`Ui.table_changed`

```
fn void Ui.table_changed(&self, String id)
```

Tells the table `id` its rows changed, so it sorts them again next frame.

`Ui.table_sort`

```
fn TableSort Ui.table_sort(&self, String id)
```

The table `id`'s sort, when the table keeps it.

`Ui.table_row_at`

```
fn int Ui.table_row_at(&self, String id, int pos)
```

The row shown at display position `pos` of the table `id`, as last sorted; -1 out of range.

`Ui.compare_cells`

```
fn int Ui.compare_cells(&self, String a, String b)
```

The default order of two cells: numbers as numbers (and before text), other text in the text order.

ui::table_rows_id

```
fn Id ui::table_rows_id(Id table)
```

The id of a table's rows: the list inside it, below the header.

ui::TABLE_ROW_HEIGHT

```
const float ui::TABLE_ROW_HEIGHT = 36
```

Points a table row is tall unless the app says otherwise.

ui::TABLE_HEADER_HEIGHT

```
const float ui::TABLE_HEADER_HEIGHT = 32
```

Points a table's header is tall.

ui::TABLE_RESIZE_GRIP

```
const float ui::TABLE_RESIZE_GRIP = 4
```

Points either side of a column's end edge that take a resize drag.

Ui.@grid

```
macro void Ui.@grid(
    &self,
    String id = "grid",
    int cols = 1,
    float min_cell = 0,
    float gap = 4,
    float gap_x = -1,
    float gap_y = -1,
    bool dense = false,
    bool stretch = true; @body())
```

A grid of the cells its body builds with `Ui.@cell`. Give `cols` for a set number of columns (with `Ui.at` or `Ui.width_at` to change it with the width), or `min_cell` for as many columns as fit cells that wide. `gap` (or `gap_x` and `gap_y`) is in spacing steps, as for boxes.

Ui.@cell

```
macro void Ui.@cell(&self, int span = 1, int start = 0, int rows = 1; @body())
```

A cell of the grid being built: `span` columns wide, starting at column `start` (from 1; 0 for the next free place), `rows` rows tall.

Ui.grid_columns

```
fn int Ui.grid_columns(&self, String id)
```

The number of columns the grid `id` had when last built; 0 before.

ui::GRID_MAX_COLUMNS

```
const int ui::GRID_MAX_COLUMNS = 64
```

The most columns a grid has.

12 Status

`Tone`

```
enum Tone : char
```

What a status widget says: nothing special, information, success, a warning or a danger.

Value	Meaning
`NEUTRAL`	TEXT_SECONDARY on SURFACE_RAISED
`INFO`	ACCENT on ACCENT_SUBTLE
`SUCCESS`	SUCCESS on SUCCESS_SUBTLE
`WARNING`	WARNING on WARNING_SUBTLE
`DANGER`	DANGER on DANGER_SUBTLE

`Tone.strong`

```
fn Token Tone.strong(Tone self)
```

The tone's strong color: its icon, fill, border and, without high contrast, its text.

`Tone.subtle`

```
fn Token Tone.subtle(Tone self)
```

The tone's soft background.

`Tone.icon`

```
fn Icon Tone.icon(Tone self)
```

The tone's icon, for alerts.

`Ui.progress`

```
fn void Ui.progress(
    &self,
    String label,
    float value,
    Tone tone = INFO,
    Size width = GROW,
    bool percent = true)
```

A bar filled to `value`, from 0 to 1, easing there when it changes (at once under reduced motion). With `label`, the label sits above it with the value as a percentage; `label` is also its id.

`Ui.progress_indeterminate`

```
fn void Ui.progress_indeterminate(
    &self,
    String label,
    Tone tone = INFO,
    Size width = GROW)
```

A bar for work whose end isn't known: a segment two fifths of the track slides across, over and over (`Loop.SLIDE`). Under reduced motion it doesn't move: the whole track takes a quarter of the fill color instead.

`ui::PROGRESS_HEIGHT`

```
const float ui::PROGRESS_HEIGHT = 6
```

Points a progress bar's track is tall.

`Ui.set_progress`

```
fn void Ui.set_progress(&self, float value)
```

Shows progress on the app's taskbar button, Dock icon or notification (`os::set_progress`): `value` from 0 to 1, `os::PROGRESS_UNKNOWN` for work whose end isn't known, or `os::PROGRESS_NONE` to take it away when the work is done. Call it each frame with the current value; the OS hears only of changes of a percent or more. Does nothing where the OS has no such place; on Android it shows a notification even without a native window.

`Ui.badge`

```
fn void Ui.badge(&self, String text, Tone tone = NEUTRAL)
```

A pill holding `text` in `tone`: the tone's tint with its color for the text; outlined under high contrast.

`Ui.badge_count`

```
fn void Ui.badge_count(&self, int n, Tone tone = DANGER)
```

A count in a solid pill of `tone`'s color, as on an app icon: "99+" past 99, nothing at 0 or less.

`Ui.badge_dot`

```
fn void Ui.badge_dot(&self, Tone tone = DANGER, bool ping = false)
```

A dot in `tone`'s color; with `ping`, a ring grows from it and fades, over and over (still under reduced motion).

`Ui.count_text`

```
fn String Ui.count_text(&self, int n)
```

The text of a count: "99+" past 99; empty for none.

`Ui.@badged`

```
macro void Ui.@badged(
    &self,
    int count = -1,
    bool dot = false,
    bool ping = false,
    Tone tone = DANGER; @body())
```

Runs the body, any widget, with a badge floating on its top end corner: a count (`count` above 0; "99+" past 99), or a dot (`dot`, which pings with `ping`). With neither, the body shows alone. The body keeps its size, its pointer and its focus.

ui::BADGE_DOT

```
const float ui::BADGE_DOT = 8
```

Points across a badge's dot.

ui::BADGE_HEIGHT

```
const float ui::BADGE_HEIGHT = 18
```

Points a count badge is tall.

Ui.@marked

```
macro void Ui.@marked(
    &self,
    Icon icon,
    Token color,
    String label,
    float inset = MARK_INSET,
    Size width = GROW; @body())
```

Runs the body, any widget, with a status mark floating on its top start corner: a `SURFACE_RAISED` circle `MARK_SIZE` points across, with `icon` in `color` `MARK_INSET` points inside. With `NONE`, the body shows alone, so a mark comes and goes without moving the widget. `label` names the mark for its id; Kiln has no accessible-name API, so it is not a screen-reader name.

ui::MARK_SIZE

```
const float ui::MARK_SIZE = 28
```

Points across a mark's circle.

ui::MARK_INSET

```
const float ui::MARK_INSET = 8
```

Points a mark sits inside its widget's corner.

Ui.@alert

```
macro bool Ui.@alert(
    &self,
    String id,
    Tone tone = INFO,
    String title = "",
    bool* open = null,
    String action = ""; @body())
```

An alert in `tone` with `title` (in semibold, when given) over its body, which builds the message: wrapping text, or any widgets. With `action`, a button at its end; true in the frame it is pressed. With `open`, a that sets `*open` false, and the alert slides shut. `id` names it.

Ui.metric

```
fn void Ui.metric(
    &self,
    String label,
    String value,
    float change = float::nan,
    bool up_is_good = true,
    String detail = "")
```

`value` (already formatted, so the app chooses its units) large under `label`. `change`, a fraction (0.12 for 12%), shows as an arrow and a percentage: green when it goes the good way, red when it doesn't, with `up_is_good` saying which way is good (false for costs or errors); NaN for none. `detail` adds a quiet line under it.

ui::metric_tone

```
fn Token ui::metric_tone(float change, bool up_is_good)
```

The color a change takes: the good way green, the other red, none quiet.

13 Canvas

Ui.@canvas

```
macro void Ui.@canvas(
    &self,
    String id,
    Size width = GROW,
    Size height = ui::px(200); @body())
```

A canvas `width` by `height` whose body draws into `c`: paths (`move_to`, `line_to`, `quad_to`, `cubic_to`, `arc`, `rect`, `circle`), `fill`, `stroke`, `text`, and clips. `id` names it, for `Ui.interact` on `c.id` and for `Ui.canvas_svg`.

Ui.canvas_draw

```
fn void Ui.canvas_draw(&self, Canvas* c, Size width, Size height)
```

Paints an open canvas at `width` by `height`; `Ui.canvas_begin` and `Ui.canvas_end` split `Ui.@canvas`.

Canvas

```
struct Canvas
```

A canvas being drawn this frame; see `Ui.@canvas`.

Field	Type	Meaning
ui	Ui*	
id	Id	
size	float[<2>]	last frame's size in points; the window's before the first layout
points	List{float[<2>]}	
subs	List{SubPath}	
ops	List{CanvasOp}	
path_start	uint	the first subpath of the path being built
building	bool	a subpath is open
cursor	Point	
tolerance	float	points a flattened curve may stray: a quarter pixel

Canvas.move_to

```
fn void Canvas.move_to(&self, Point p)
```

Starts a new subpath at `p`.

Canvas.line_to

```
fn void Canvas.line_to(&self, Point p)
```

A straight line to `p`; a point with NaN in it leaves a gap.

`Canvas.quad_to`

```
fn void Canvas.quad_to(&self, Point c, Point p)
```

A quadratic Bézier curve through control point `c` to `p`.

`Canvas.cubic_to`

```
fn void Canvas.cubic_to(&self, Point c1, Point c2, Point p)
```

A cubic Bézier curve through control points `c1` and `c2` to `p`.

`Canvas.arc`

```
fn void Canvas.arc(&self, Point center, float radius, float from, float to)
```

An arc of the circle at `center` with `radius`, from angle `from` to `to` (radians, clockwise on screen from the x axis), joined to the path by a line.

`Canvas.close`

```
fn void Canvas.close(&self)
```

Closes the subpath being built back to its start.

`Canvas.rect`

```
fn void Canvas.rect(&self, Rect r)
```

A closed rectangle subpath.

`Canvas.rounded_rect`

```
fn void Canvas.rounded_rect(&self, Rect r, float radius)
```

A closed rectangle subpath with corners of `radius`.

`Canvas.circle`

```
fn void Canvas.circle(&self, Point center, float radius)
```

A closed circle subpath.

`Canvas.fill`

```
fn void Canvas.fill(&self, Color color)
```

Fills the path built since the last fill or stroke in `color`, each subpath as its own polygon (closed or not); then a new path starts. A subpath should not cross itself.

`Canvas.stroke`

```
fn void Canvas.stroke(&self, Color color, StrokeStyle style = {})
```

Strokes the path built since the last fill or stroke in `color` with `style`; then a new path starts.

`StrokeStyle`

```
struct StrokeStyle
```

How a path is stroked. The zero value is a 1-point miter-joined, butt-capped line.

Field	Type	Meaning
`width`	`float`	points; 0 for 1
`join`	`LineJoin`	
`cap`	`LineCap`	
`miter_limit`	`float`	the longest miter, in widths, before it bevels; 0 for 4
`dashes`	`float[]`	dash and gap lengths in points, repeating; empty for a solid line
`dash_offset`	`float`	how far into the pattern the line starts

`LineJoin`

```
enum LineJoin : char
```

How a stroke's segments meet.

Value	Meaning
`MITER`	a sharp corner, beveled past the miter limit
`ROUND`	
`BEVEL`	

`LineCap`

```
enum LineCap : char
```

How a stroke's open ends finish.

Value	Meaning
`BUTT`	flat, at the end point
`ROUND`	a half circle past the end
`SQUARE`	flat, half the width past the end

`Canvas.text`

```
fn void Canvas.text(
    &self,
    Point at,
    String text,
    TextSize size = SM,
    Weight weight = REGULAR,
    Color color = {},
    Align align = START,
    TextBaseline baseline = ALPHABETIC)
```

`text` at `at`, placed by `align` across and `baseline` up and down.

`TextBaseline`

```
enum TextBaseline : char
```

Where text sits vertically against its point.

Value	Meaning
`TOP`	the point is at the top of the line
`MIDDLE`	halfway down the line
`ALPHABETIC`	on the baseline
`BOTTOM`	at the bottom of the line

`Canvas.text_width`

```
fn float Canvas.text_width(
    &self,
    String text,
    TextSize size = SM,
    Weight weight = REGULAR)
```

The width of `text` at `size` and `weight`, in points, for laying out labels.

`Canvas.push_clip`

```
fn void Canvas.push_clip(&self, Rect r)
```

Draws only inside `r` until `Canvas.pop_clip`, within any clip already pushed.

`Canvas.pop_clip`

```
fn void Canvas.pop_clip(&self)
```

Ends the clip of the last `Canvas.push_clip`.

`Canvas.pointer`

```
fn Point? Canvas.pointer(&self)
```

Where the pointer is over the canvas, from its top-left, while it hovers it.

`Ui.@canvas_id`

```
macro void Ui.@canvas_id(
    &self,
    Id id,
    Size width = GROW,
    Size height = ui::px(200); @body())
```

`Ui.@canvas` under an id the caller made, for a widget that interacts with its canvas.

`Canvas.marker`

```
fn void Canvas.marker(&self, Point p, float r, int i, bool shapes)
```

A marker for series `i`: a circle, square, diamond or triangle, by `i` under high contrast.

`Ui.fit_text`

```
fn String Ui.fit_text(
    &self,
    String text,
    float width,
    TextSize size = SM,
    Weight weight = REGULAR)
```

`text` shortened with "…" to fit `width` points at `size`, cut at a character's start; whole when it fits.

`Ui.canvas_svg`

```
fn String Ui.canvas_svg(&self, String id, Allocator allocator)
```

The canvas `id` built this frame as an SVG document, in `allocator`; empty when there is none.

`Canvas.svg`

```
fn String Canvas.svg(&self, Allocator allocator)
```

What the canvas recorded, as an SVG document in `allocator`.

`Point`

```
alias Point = float[<2>]
```

A point or a vector in points: x right, y down.

14 Charts

`Ui.chart`

```
fn int Ui.chart(&self, String id, Series[] series, ChartOpts opts = {})
```

A chart of `series` with `opts`: lines, areas and points over a shared x. `id` names it. Hover, or the arrow keys once it has focus, show a crosshair and a tooltip; the legend hides and shows series. Returns the x index a click or Enter chose this frame, or -1.

`Series`

```
struct Series
```

One set of values in a chart.

Field	Type	Meaning
`name`	`String`	in the legend and the tooltip
`ys`	`double[]`	
`xs`	`double[]`	empty: 0, 1, 2, ...
`kind`	`SeriesKind`	
`color`	`Color`	zero: the next series color
`width`	`float`	the line's width; 0 for 2
`stacked`	`bool`	AREA: on top of the stacked areas before it

`SeriesKind`

```
enum SeriesKind : char
```

How a series is drawn.

Value	Meaning
`LINE`	a line through its points
`AREA`	a line, filled down to the axis or to the series stacked under it
`POINTS`	markers only, as a scatter plot

`ChartOpts`

```
struct ChartOpts
```

A chart's options. The zero value is a chart as wide as its parent, 240 points tall.

Field	Type	Meaning
`width`	`Size`	FIT (the zero value) is taken as GROW
`height`	`float`	points; 0 for 240
`x`	`ChartAxis`	
`y`	`ChartAxis`	

Field	Type	Meaning
legend	LegendPlace	
cursor	int*	the x index shown with a crosshair, shared: hover writes it, -1 for none
zoomable	bool	Ctrl/⌘ and the wheel, a pinch, or Ctrl/⌘ and + or − zoom x; a drag or Shift and the wheel pan; a double-click or Ctrl/⌘ 0 resets
instant	bool	new data shows at once instead of easing in, for live data

ChartAxis

```
struct ChartAxis
```

An axis of a chart. The zero value fits its data, with labels.

Field	Type	Meaning
title	String	over the y axis, under the x axis
hidden	bool	no labels
grid	GridLines	
fixed	bool	min and max are given, not taken from the data
min	double	
max	double	
include_zero	bool	the range reaches 0 (areas always do)
format	AxisFormat	null: numbers with the ticks' decimals
format_data	void*	

GridLines

```
enum GridLines : char
```

Which grid lines an axis draws.

Value	Meaning
AUTO	across the chart for y, none for x
ON	
OFF	

AxisFormat

```
alias AxisFormat = fn String(void* data, double value)
```

A value's label, for axes and tooltips; return a string that lives through the frame (tmem, or a literal).

LegendPlace

```
enum LegendPlace : char
```

Where a chart's legend goes.

Value	Meaning
AUTO	on top, when there are two series or more
NONE	
TOP	
BOTTOM	
END	beside the chart

Ui.bar_chart

```
fn int Ui.bar_chart(
    &self,
    String id,
    String[] categories,
    Series[] series,
    BarOpts opts = {})
```

A bar chart of `series` (their `ys`, one per category) over `categories`, with `opts.id` names it. Hover, or the arrow keys once it has focus, highlight a category and show a tooltip; the legend hides and shows series. Returns the category a click or Enter chose this frame, or -1.

BarOpts

```
struct BarOpts
```

A bar chart's options. The zero value: grouped upright bars, as wide as the parent and 240 points tall.

Field	Type	Meaning
width	Size	FIT (the zero value) is taken as GROW
height	float	points; 0 for 240
values	ChartAxis	the value axis: up the side, or along the bottom when horizontal
layout	BarLayout	
horizontal	bool	categories down the side, bars across
legend	LegendPlace	
cursor	int*	the category shown, shared with other charts; -1 for none

BarLayout

```
enum BarLayout : char
```

How a bar chart's series share a category.

Value	Meaning
GROUPED	side by side
STACKED	one on another, positives up from zero and negatives down

`ui::BAR_BAND`

```
const float ui::BAR_BAND = 0.72
```

The fraction of a category's band its bars take.

`Canvas.bar`

```
fn void Canvas.bar(&self, Rect r, float radius, Side far)
```

A bar from `r`, with the end at `far` rounded by `radius`: TOP, BOTTOM, START (left) or END (right).

`Ui.strip_chart`

```
fn void Ui.strip_chart(
    &self,
    String id,
    StripData* data,
    double window,
    String name = "",
    ChartOpts opts = {})
```

A line of the samples in `data` from the last `window` seconds, plotted against seconds before the newest, so its ticks stay put as it scrolls ("-10s" ... "now"), as a `Ui.chart` with `opts` (its x range and easing are the strip's own; a format of x is given seconds before the newest). `name` names the series.

`StripData`

```
struct StripData
```

A ring buffer of timestamped values for `Ui.strip_chart`.

Field	Type	Meaning
`times`	`double[]`	
`values`	`double[]`	
`head`	`usz`	where the next sample goes
`count`	`usz`	
`allocator`	`Allocator`	

`StripData.init`

```
fn void StripData.init(&self, usz capacity, Allocator allocator = mem)
```

Gets the buffer ready for `capacity` samples; once full, each new one replaces the oldest.

`StripData.free`

```
fn void StripData.free(&self)
```

Frees the buffer.

StripData.push

```
fn void StripData.push(&self, double t, double value)
```

Adds `value` at time `t` (seconds, rising); NaN leaves a gap.

StripData.at

```
fn double[2] StripData.at(&self, usz i)
```

The `i`th sample, oldest first.

Ui.chart_zoom_range

```
fn double[2]? Ui.chart_zoom_range(&self, String id)
```

The x range a zoomable chart `id` shows, when it is zoomed in.

Ui.sparkline

```
fn void Ui.sparkline(
    &self,
    String id,
    double[] values,
    Tone tone = INFO,
    bool area = true,
    Size width = ui::px(120),
    float height = 32)
```

A small chart of `values` in a line, without axes, for a trend beside a number: filled under the line with `area`, and a dot on the last value. `tone` colors it. It takes no pointer and no focus.

Ui.series_color

```
fn Color Ui.series_color(&self, int i)
```

The color of chart series `i`, going round the eight `SERIES_*` tokens.

ui::default_series

```
fn Oklch ui::default_series(usz i, bool dark)
```

Kiln's color for chart series `i` (0 to 7): its hue at one lightness and chroma, a little lighter on dark themes, so no series stands out more than another.

ui::SERIES_HUES

```
const float[8] ui::SERIES_HUES = { 259, 55, 150, 25, 300, 195, 350, 95 }
```

Hues of Kiln's chart palette, in series order: blue, orange, green, red, purple, teal, pink, yellow; neighbors in the order are far apart on the wheel.

ChartFrame

```
struct ChartFrame
```

A chart's layout this frame: the plot area and its scales.

Field	Type	Meaning
`plot`	`Rect`	in the canvas, points
`x`	`Scale`	
`y`	`Scale`	
`xt`	`Ticks`	
`yt`	`Ticks`	

`ChartFrame.to_screen`

```
fn Point ChartFrame.to_screen(&self, double x, double y)
```

The point in the canvas where the data `x`, `y` is drawn.

`ChartFrame.to_data`

```
fn double[2] ChartFrame.to_data(&self, Point p)
```

The data at point `p`.

`Scale`

```
struct Scale
```

A linear map from a data range (`d0` to `d1`) to points (`r0` to `r1`).

Field	Type	Meaning
`d0`	`double`	
`d1`	`double`	
`r0`	`float`	
`r1`	`float`	

`Scale.map`

```
fn float Scale.map(Scale self, double v)
```

`v` in points.

`Scale.invert`

```
fn double Scale.invert(Scale self, float p)
```

The data value at point `p`.

`Ticks`

```
struct Ticks
```

Tick values for an axis: `count` of them from `start`, `step` apart, labelled with `decimals` decimals after dividing by `divisor` and adding `suffix`.

Field	Type	Meaning
`start`	`double`	
`step`	`double`	
`count`	`int`	
`decimals`	`int`	
`divisor`	`double`	1, 1e3, 1e6 or 1e9
`suffix`	`String`	"", "k", "M" or "G"

`ui::nice_ticks`

```
fn Ticks ui::nice_ticks(double min, double max, int target)
```

About `target` ticks covering `min` to `max`: a step of 1, 2 or 5 times a power of ten, the first tick at or below `min` and the last at or above `max`. An empty range is widened around its value.

`Ticks.at`

```
fn double Ticks.at(Ticks self, int i)
```

The value of tick `i`.

`Ticks.end`

```
fn double Ticks.end(Ticks self)
```

The last tick's value.

`Ui.tick_label`

```
fn String Ui.tick_label(&self, Ticks t, double v)
```

The label of value `v` on an axis with ticks `t`: the ticks' decimals and suffix, and the OS's decimal separator.

`Ui.chart_hides`

```
fn bool Ui.chart_hides(&self, String id, int i)
```

Whether the chart `id`'s legend hid series `i`.

`Ui.chart_activated`

```
fn int Ui.chart_activated(&self, String id)
```

The point chosen in the chart `id` (a click, Enter or a gamepad A), or -1.

`Ui.chart_hovered`

```
fn int Ui.chart_hovered(&self, String id)
```

The x index under the chart `id`'s pointer or keys this frame, or -1: its tooltip's index.

`Ui.chart_set_hidden`

```
fn void Ui.chart_set_hidden(&self, String id, int i, bool hidden)
```

Shows or hides series `i` of the chart `id`, as a legend click does.

`Ui.chart_drawn`

```
fn double Ui.chart_drawn(&self, String id, int index)
```

The value the chart `id` drew last frame at `index`, its series one after another: on its way to the data while new data eases in.

`ui::CHART_PICK_RADIUS`

```
const float ui::CHART_PICK_RADIUS = 12
```

Points from the pointer within which a scatter plot's nearest point is picked.

15 Background work

`Ui.job`

```
fn JobState Ui.job(
    &self,
    String id,
    JobFn run,
    void* data,
    ulong key = 0,
    JobFree free_result = null,
    JobFree free_data = null)
```

Starts the job `id` the first time it is asked for, and reports it. A different `key` (a hash of what the job reads, such as a search query) lets go of the old job and starts a new one. `free_result` frees the result when the job is let go of; without it, the result is freed as one block. With `free_data`, the job owns `data` (allocated in `ui::job_allocator()`) and frees it once neither the UI nor the worker holds the job, so the app can let go of a job without waiting for it.

`JobFn`

```
alias JobFn = fn void?(Job* job)
```

A job's work: reads `job.data`, sets `job.result` (allocated in `job.allocator`); a fault fails the job. Runs on a worker thread.

`JobFree`

```
alias JobFree = fn void(void* result, Allocator allocator)
```

Frees a job's result, and whatever it owns, in `allocator`.

`Job`

```
struct Job
```

One job, as its function sees it.

Field	Type	Meaning
`data`	`void*`	the app's input; kept alive and unchanged while the job runs (by the app, or by the job with `free_data`)
`result`	`void*`	set by the function
`allocator`	`Allocator`	thread-safe: for the result
`progress`	`Atomic{float}`	0 to 1, set by the function if it can tell
`cancel`	`Atomic{bool}`	set by the UI when it no longer wants the result
`run`	`JobFn`	
`free_result`	`JobFree`	
`free_data`	`JobFree`	frees `data` with the job, when the job owns it
`refs`	`Atomic{int}`	the UI and the worker
`status`	`Atomic{char}`	a JobStatus

Field	Type	Meaning
`started`	`Atomic{bool}`	a worker picked it up
`error`	`fault`	
`key`	`ulong`	
`asked`	`double`	when the UI last asked about it
`taken`	`bool`	the app took its result
`inline_pending`	`bool`	to run on the UI thread at the next begin()
`platform`	`Platform`	woken when it finishes

`Job.cancelled`

```
fn bool Job.cancelled(&self)
```

True once the UI no longer wants this job's result: long work can stop early.

`Job.set_progress`

```
fn void Job.set_progress(&self, float done)
```

Reports how far along the job is, 0 to 1, for a progress bar.

`JobState`

```
struct JobState
```

A job as the UI sees it in a frame.

Field	Type	Meaning
`status`	`JobStatus`	
`result`	`void*`	with DONE: the function's result, until the job is let go of or taken
`error`	`fault`	with FAILED
`progress`	`float`	

`JobStatus`

```
enum JobStatus : char
```

How a job is doing.

Value	Meaning
`PENDING`	queued or running
`DONE`	finished; its result is ready
`FAILED`	its function returned a fault

`Ui.job_take`

```
fn void* Ui.job_take(&self, String id)
```

Hands a finished job's result to the app, which then frees it; null when there is none.

Ui.job_progress

```
fn float Ui.job_progress(&self, String id)
```

The job `id`'s progress, 0 to 1; 0 when it hasn't started.

Ui.job_status

```
fn int Ui.job_status(&self, String id)
```

The job `id`'s status as a `JobStatus` ordinal.

Ui.job_restart

```
fn void Ui.job_restart(&self, String id)
```

Lets go of the job `id`, cancelling it if it runs; the next `Ui.job` call starts it again.

Ui.set_job_threads

```
fn void Ui.set_job_threads(&self, int threads)
```

Sets how many worker threads run jobs; 0 runs every job on the UI thread at the next `begin()`.

Requires:

- `!self.pool_ready : "set the worker count before the first job starts"`

ui::job_allocator

```
fn Allocator ui::job_allocator()
```

The allocator jobs and their results use: the C heap, safe from any thread. Free a result taken with `Ui.job_take` with it.

ui::MAX_JOB_THREADS

```
const int ui::MAX_JOB_THREADS = 4
```

The most workers a `Ui` starts by default, whatever the CPU count.

Ui.@loading

```
macro void Ui.@loading(&self, String id, bool pending, Size width = ui::FIT; @body())
```

Runs the body as usual, so its size and layout stay put; while `pending`, a skeleton covers it, breathing with `Loop.PULSE`, takes the pointer from it, and its widgets are no focus stops. `id` names the box; `width` sizes it as any box (`ui::GROW` for content that wraps).

Ui.spinner

```
fn void Ui.spinner(&self, ControlSize size = MD, Token color = TEXT_SECONDARY)
```

A spinner the size of a control's icon at `size`, in `color`; it turns with `Loop.SPIN`.

`Ui.skeleton`

```
fn void Ui.skeleton(
    &self,
    Size width = ui::GROW,
    Size height = ui::px(12),
    float radius = 0)
```

A loading block `width` by `height`, filled with the skeleton tone and `radius` corners.

16 Interaction and input

`Ui.interact`

```
fn Interact Ui.interact(
    &self,
    Id id,
    bool focusable = false,
    bool disabled = false,
    bool autofocus = false,
    bool keys = true)
```

Reports what the pointers and keys did to `id` this frame.

- `focusable`: Can take keyboard focus (a focus stop); pressing it focuses it
- `disabled`: Shows a tooltip and stops presses from reaching what's under it, but takes no other input and isn't a focus stop
- `autofocus`: Takes focus the first frame it appears
- `keys`: Enter and Space (and the gamepad A) activate it while it has focus; a text field turns this off and reads the keys itself

`Ui.claim_pointer`

```
fn void Ui.claim_pointer(&self, Id id)
```

Keeps scroll areas from taking the pointer holding `id`: for a widget that drags, such as a slider, so dragging it never scrolls the page. Call it each frame the widget is held; it lasts until the pointer lifts.

`Ui.last_id`

```
fn Id Ui.last_id(&self)
```

The id of the widget that last called `Ui.interact` this frame, such as the button just built, to anchor a `Ui.@layer` to it; `ui::NO_ID` before any has.

`Ui.click_count`

```
fn int Ui.click_count(&self, Id id)
```

How many clicks the pointer holding `id` made in a row: 1, 2 for a double-click, 3 for a triple; 0 when no pointer holds it.

`Interact`

```
bitstruct Interact : ushort
```

What happened to one widget this frame, as `Ui.interact` reports it.

Field	Type	Meaning
`hovered`	`bool`	a mouse or pen is over it, and no other widget holds that pointer
`pressed`	`bool`	a pointer was pressed on it (the first frame only)
`held`	`bool`	a pointer pressed on it is still down
`released`	`bool`	the pointer holding it was released this frame

Field	Type	Meaning
`clicked`	`bool`	released over it without dragging
`dragging`	`bool`	held, and moved past the drag threshold
`focused`	`bool`	has keyboard focus
`activated`	`bool`	Enter, Space or gamepad A while focused
`ring`	`bool`	focused, and focus is being moved by keyboard or gamepad: draw the focus ring
`shown`	`bool`	hover, hold or focus here is only shown, by ui.@show_state: don't act on it
`context`	`bool`	a context menu was asked for on it: a secondary press, a long touch, the Menu key
`long_press`	`bool`	the context menu came from a long touch
`touch`	`bool`	a finger is holding it

`Ui.pointer_of`

```
fn float[<2>]? Ui.pointer_of(&self, Id id)
```

Where the pointer holding `id` is, for dragging; a fault when nothing holds it.

`Ui.press_point_of`

```
fn float[<2>]? Ui.press_point_of(&self, Id id)
```

Where the pointer holding `id` went down: the start of a drag, or where a click landed. A fault when nothing holds it.

`Ui.context_requested`

```
fn bool Ui.context_requested(&self, Id id)
```

True in the frame after a context menu was asked for on `id`: a secondary press on it, Ctrl+click with Mac keys, a long touch, or the Menu key or Shift+F10 while it has focus or holds a selection. `Ui.context_point` says where.

`Ui.middle_clicked`

```
fn bool Ui.middle_clicked(&self, Id id)
```

True in the frame after a middle click landed on `id`; `Ui.middle_point` says where. On Linux and the BSDs a text field pastes the primary selection there.

`Ui.middle_point`

```
fn float[<2>] Ui.middle_point(&self)
```

Where the middle click `Ui.middle_clicked` reports landed.

`Ui.context_point`

```
fn float[<2>] Ui.context_point(&self)
```

Where the context menu asked for this frame opens: the pointer, or the bottom-left of the widget the keyboard asked for.

Ui.has_context

```
fn void Ui.has_context(&self, Id id)
```

Notes that `id` has a context menu this frame, so a long touch on it opens the menu rather than its tooltip.

ui::LONG_PRESS

```
const double ui::LONG_PRESS = 0.5
```

Seconds a touch rests on a widget with a context menu before it opens.

Ui.is_held

```
fn bool Ui.is_held(&self, Id id)
```

Whether a pointer pressed on `id` is still down: true from the frame the press is reported until the release.

Ui.press_of

```
fn float[<2>]? Ui.press_of(&self, Id id)
```

Where the pointer holding `id` was pressed; fails with NOT_FOUND when no pointer holds it.

Ui.wants_pointer

```
fn bool Ui.wants_pointer(&self)
```

Whether any pointer (mouse, pen or touch) is over a widget or held by one, so a game can ignore it; ask after `Ui.end`.

Ui.@show_state

```
macro void Ui.@show_state(&self, Interact state; @body())
```

Runs the body with its widgets showing `state` (hovered, held, focused, ring) as if a pointer or the keyboard were on them: for galleries and screenshots. Nothing fires: no clicks or activations come of it, and real focus and pointer capture are untouched. Disabled widgets stay disabled.

ui::MIN_TOUCH_TARGET

```
const float ui::MIN_TOUCH_TARGET = 44
```

The smallest hit area, in points, a widget gets under a finger; smaller widgets grow to it around their center.

ui::DRAG_THRESHOLD_MOUSE

```
const float ui::DRAG_THRESHOLD_MOUSE = 6
```

How far, in points, a mouse or pen press moves before it becomes a drag instead of a click.

ui::DRAG_THRESHOLD_TOUCH

```
const float ui::DRAG_THRESHOLD_TOUCH = 10
```

How far, in points, a touch moves before it becomes a drag instead of a click.

Input

```
struct Input
```

Everything that happened since the last frame: time, window size, pointers, wheel, keys, modifiers and the first gamepad. A backend fills it each frame; a test or custom backend fills one and passes it to `Ui.begin`.

Field	Type	Meaning
time	double	seconds since some fixed start; 0 lets the Ui add up dt
dt	float	seconds since the last frame
size	float[<2>]	the window's size in logical points; 0 keeps the last size
scale	float	pixels per point; 0 keeps the last scale
unfocused	bool	the window doesn't have the keyboard focus (another app does)
paused	bool	the platform has no visible surface; the app should not build or draw a frame
events	bool	anything happened at all (an idle app may sleep when not)
pointers	Pointer[10]	
pointer_count	usz	
wheel	float[<2>]	wheel movement this frame, x and y
keys_down	KeySet	
keys_pressed	KeySet	went down this frame (including key repeat)
mods	Mods	
pad_down	Gamepad	the first gamepad
pad_pressed	Gamepad	went down this frame
typed	char[64]	UTF-8 the keyboard typed this frame; see Input.text
typed_len	usz	
composing	char[256]	UTF-8 an input method is composing at the caret, not typed yet; see Input.composition
composing_len	usz	
composing_cursor	usz	where the caret is in it, in bytes
composing_clause	usz[2]	the part the input method is converting, as a byte range in it; empty for none
dropped	String[]	the files' paths
dropped_text	String	
drop_pos	float[<2>]	where, in points
dragging	bool	something is being dragged over the window
drag_pos	float[<2>]	where, while dragging, where the platform reports it

Field	Type	Meaning
back	BackGesture	the system's predictive back gesture, if any this frame

Input.pointer

```
fn void Input.pointer(
    &self,
    float[<2>] pos,
    bool down,
    PointerKind kind = MOUSE,
    uint id = 0,
    bool secondary = false,
    bool middle = false)
```

Sets pointer `id`'s position and state, adding it if it isn't listed yet.

Requires:

- `self.pointer_count < MAX_POINTERS || self.find_pointer(id) >= 0 : "too many pointers"`

Input.drop

```
fn void Input.drop(&self, String[] paths, float[<2>] pos)
```

Drops `paths` on the window at `pos` this frame, for tests and for a Game-style app's own drops.

Ui.dropped_files

```
fn String[] Ui.dropped_files(&self)
```

The paths of the files dropped on the window this frame, anywhere; none most frames.

Ui.dropped_text

```
fn String Ui.dropped_text(&self)
```

The text dropped on the window this frame, such as a link; empty most frames.

Ui.dropped_on

```
fn String[] Ui.dropped_on(&self, Id box)
```

The files dropped on `box` this frame: on its rect from last frame.

Ui.text_dropped_on

```
fn String Ui.text_dropped_on(&self, Id box)
```

The text dropped on `box` this frame, as `Ui.dropped_on` finds files.

Ui.drag_over

```
fn bool Ui.drag_over(&self, Id box, DragKind kind = OS_DROP)
```

Whether something of `kind` is being dragged over `box` now, for a highlight that says it can be dropped there: by default files or text from the OS, or else a drag inside the app (`Ui.drag_source`) of that kind.

DragKind

```
alias DragKind = int
```

What a drag carries, as the app sees it: its own kinds start at 1; Kiln's own are below 0.

ui::OS_DROP

```
const DragKind ui::OS_DROP = 0
```

Files and text dragged from the OS, as `Ui.drag_over` takes them by default.

Ui.drag_source

```
fn bool Ui.drag_source(&self, DragKind kind, void* payload, String label, int count = 1)
```

Makes the widget just built (`Ui.last_id`) a drag source: dragging it carries `payload` as `kind`, and a preview showing `label`, with `count` beside it above 1, follows the pointer. A mouse or pen drags once past the drag threshold; a touch rests for a long press first, so a quick swipe still scrolls, and lifting without moving then opens the widget's context menu, if it has one. Escape, or a release over no target, cancels. True while this widget's drag is on. Kiln keeps only the pointer: the payload is the app's, and must live until the drag ends.

Requires:

- `kind != OS_DROP : "kind 0 is the OS's drops"`

Ui.dragged_item

```
fn void*? Ui.dragged_item(&self, DragKind kind)
```

The payload of the drag of `kind` going on now, wherever it is; a fault when there is none.

Ui.dropped_item

```
fn void*? Ui.dropped_item(&self, Id box, DragKind kind)
```

The payload of a drag of `kind` let go over `box` this frame, by its rect from last frame; a fault on any other frame. The app takes it from there.

Ui.drag_point

```
fn float[<2>]? Ui.drag_point(&self, Id box, DragKind kind)
```

Where the pointer of the drag of `kind` is over `box`: for a target that shows where a drop would go.

ui::DRAG_SCROLL_EDGE

```
const float ui::DRAG_SCROLL_EDGE = 32
```

Points from a scroll area's edge within which a drag scrolls it.

ui::DRAG_SCROLL_SPEED

```
const float ui::DRAG_SCROLL_SPEED = 900
```

Points per second a drag scrolls an area at its very edge.

ui::DRAG_LABEL_MAX

```
const usz ui::DRAG_LABEL_MAX = 96
```

Bytes of the preview's label that are kept.

DropBuffer

```
struct DropBuffer
```

What a platform keeps of drops: the paths and text of one still arriving, then those handed to `Input` until the next poll, when they are freed. Nothing is allocated until something is dropped.

Field	Type	Meaning
`allocator`	`Allocator`	
`pending`	`List{String}`	arriving: a drop's files come one event each
`pending_text`	`String`	
`delivered`	`List{String}`	in `Input` this frame
`delivered_text`	`String`	
`pos`	`float[<2>]`	where the drop landed, in points
`dragging`	`bool`	something is over the window
`drag_pos`	`float[<2>]`	where, in points

DropBuffer.init

```
fn void DropBuffer.init(&self, Allocator allocator)
```

Starts empty, allocating in `allocator` once something is dropped.

DropBuffer.next_poll

```
fn void DropBuffer.next_poll(&self)
```

At the start of a poll: frees what the last one handed out.

DropBuffer.add_file

```
fn void DropBuffer.add_file(&self, String path, float[<2>] pos)
```

A dropped file's path, copied, at `pos` in points.

DropBuffer.add_text

```
fn void DropBuffer.add_text(&self, String text, float[<2>] pos)
```

Dropped text, copied, at `pos` in points.

DropBuffer.complete

```
fn void DropBuffer.complete(&self)
```

The drop is complete: what arrived goes to `Input` at this poll's `fill`.

`DropBuffer.fill`

```
fn void DropBuffer.fill(&self, Input* input)
```

This poll's drops and drag into `input`.

`DropBuffer.free`

```
fn void DropBuffer.free(&self)
```

Frees everything it holds, arriving or handed out.

`Input.press`

```
fn void Input.press(&self, Key key)
```

Presses `key` this frame: adds it to both the pressed and the down keys, for tests.

`Input.text`

```
fn String Input.text(&self)
```

What the keyboard typed this frame, as UTF-8.

`Input.add_text`

```
fn void Input.add_text(&self, String text)
```

Adds `text` to what was typed this frame, as far as it fits, for tests and backends.

`Input.composition`

```
fn String Input.composition(&self)
```

What an input method is composing at the caret this frame: shown in the focused field, underlined, but not typed until the input method commits it (then it arrives in `Input.text`). Empty when nothing is being composed.

`Input.compose`

```
fn void Input.compose(&self, String text, usz cursor = usz::max, usz[2] clause = {})
```

Sets what an input method is composing, for backends and tests: `cursor` is where the caret is in it and `clause` the part being converted, both in bytes. Text past what fits is cut at a character's start. The backend sets `events` when it changes; reporting the same composition again isn't one.

`Input.add_codepoint`

```
fn void Input.add_codepoint(&self, uint cp)
```

Adds one typed character, encoded as UTF-8.

`Input.press_pad`

```
fn void Input.press_pad(&self, Gamepad buttons)
```

Presses the gamepad buttons set in `buttons`, for tests.

`Input.is_pressed`

```
fn bool Input.is_pressed(&self, Key key)
```

Whether `key` went down this frame (including key repeat), whether or not a widget used it.

`Input.is_down`

```
fn bool Input.is_down(&self, Key key)
```

Whether `key` is held this frame.

`Pointer`

```
struct Pointer
```

One pointer's state in a frame of `Input`.

Field	Type	Meaning
`id`	`uint`	stable while the pointer exists (a finger, the mouse)
`kind`	`PointerKind`	
`pos`	`float[<2>]`	in logical points, from the window's top-left
`down`	`bool`	primary button, or in contact
`secondary`	`bool`	the secondary button (the right button, a pen's barrel button) held
`middle`	`bool`	the middle button held: on Linux and the BSDs it pastes the primary selection

`PointerKind`

```
enum PointerKind : char
```

What a pointer is; touch never hovers and gets larger hit areas (`ui::MIN_TOUCH_TARGET`).

Value
`MOUSE`
`TOUCH`
`PEN`

`ui::MAX_POINTERS`

```
const usz ui::MAX_POINTERS = 10
```

The most pointers (the mouse, fingers, pens) one `Input` holds and the `Ui` tracks at once.

`Key`

```
enum Key : char
```

The keys Kiln reads: editing and navigation keys, letters, digits (`D0` to `D9`), function keys and a few punctuation keys.

Value	Meaning
TAB	
ENTER	
SPACE	
ESCAPE	
BACKSPACE	
DELETE	
INSERT	
LEFT	
RIGHT	
UP	
DOWN	
HOME	
END	
PAGE_UP	
PAGE_DOWN	
A	
B	
C	
D	
E	
F	
G	
H	
I	
J	
K	
L	
M	
N	
O	
P	
Q	
R	
S	
T	
U	
V	

Value	Meaning
W	
X	
Y	
Z	
D0	
D1	
D2	
D3	
D4	
D5	
D6	
D7	
D8	
D9	
F1	
F2	
F3	
F4	
F5	
F6	
F7	
F8	
F9	
F10	
F11	
F12	
MINUS	
EQUAL	
COMMA	
PERIOD	
SLASH	
MENU	the Menu (application) key, which opens a context menu
NONE	no key: a menu item without a shortcut

KeySet

```
struct KeySet
```

A set of keys, one bit each.

Field	Type	Meaning
bits	ulong[2]	

KeySet.add

```
fn void KeySet.add(&self, Key key)
```

Adds `key` to the set.

KeySet.has

```
fn bool KeySet.has(&self, Key key)
```

Whether `key` is in the set.

KeySet.has_any

```
fn bool KeySet.has_any(&self)
```

Whether the set holds any key.

Gamepad

```
bitstruct Gamepad : ushort
```

Gamepad buttons, by position (A is the bottom face button).

Field	Type	Meaning
a	bool	
b	bool	
x	bool	
y	bool	
lb	bool	
rb	bool	
start	bool	
back	bool	
up	bool	
down	bool	
left	bool	
right	bool	

Mods

```
bitstruct Mods : char
```

The modifier keys held this frame; `super` is Cmd on macOS and the Windows key elsewhere.

Field	Type	Meaning
shift	bool	

Field	Type	Meaning
ctrl	bool	
alt	bool	
super	bool	

Ui.shortcut

```
fn bool Ui.shortcut(
    &self,
    Key key,
    bool primary = false,
    bool shift = false,
    bool alt = false,
    bool local = false)
```

True once when the chord goes down.

- `primary`: Ctrl, or Cmd on macOS
- `local`: Only while focus is inside the scope or group around this call

Ui.use_typing

```
fn void Ui.use_typing(&self)
```

Says a widget used this frame's typing, such as the Space that opened a select's list, so no type-ahead gets it next frame.

Ui.shortcut_label

```
fn String Ui.shortcut_label(
    &self,
    Key key,
    bool primary = false,
    bool shift = false,
    bool alt = false)
```

How the chord reads in a menu: "Ctrl+Shift+S", or "⇧⌘S" on macOS; empty for `Key.NONE`. The string lives in the frame arena.

ui::key_name

```
fn String ui::key_name(Key key)
```

How `key` reads in a shortcut label: "Esc", "Page Up", "←", "S", "5", "F5".

17 Focus

Ui.focus

```
fn void Ui.focus(&self, Id id)
```

Gives keyboard focus to `id`; `ui::NO_ID` clears it. It does not turn the focus ring on; keyboard and gamepad navigation do.

Ui.@focus_scope

```
macro void Ui.@focus_scope(
    &self,
    String id,
    bool* open = null,
    bool trap = true; @body())
```

Runs the body as a focus scope: a layer such as a dialog or menu.

- `open`: Cleared by Escape (or gamepad B) while this is the top layer
- `trap`: Keep Tab and the arrows inside, and make this the top layer

Ui.@focus_group

```
macro void Ui.@focus_group(
    &self,
    String id,
    Axis axis = BOTH,
    bool wrap = false; @body())
```

Runs the body as a focus group: Tab enters it once, at the member that had focus last, and the arrow keys along `axis` move between its members. With `wrap`, they go round from the last member to the first, as in a menu.

Ui.set_group_entry

```
fn void Ui.set_group_entry(&self, Id member)
```

Makes `member` the stop Tab enters the current `Ui.@focus_group` at, such as a custom choice widget's chosen option; the default is the member focused last. Does nothing outside a group.

Axis

```
enum Axis : char
```

The arrow keys that move between the members of a `Ui.@focus_group`; the other arrows leave the group for the nearest focus stop that way.

Value	Meaning
`BOTH`	all four arrows move inside the group
`HORIZONTAL`	left and right
`VERTICAL`	up and down

Ui.key_pressed

```
fn bool Ui.key_pressed(&self, Key key)
```

True once if `key` was pressed this frame and nothing used it yet; the key is then used up. Widgets outside the top layer never get keys.

Ui.pad_pressed

```
fn bool Ui.pad_pressed(&self, Gamepad button)
```

True once if gamepad `button` was pressed this frame and nothing used it yet; the button is then used up. Like `Ui.key_pressed`, widgets outside the top layer never get it, and don't use it up.

Ui.wants_keyboard

```
fn bool Ui.wants_keyboard(&self)
```

Whether the UI is using the keyboard, so a game should leave it alone: a widget has focus moved there by keyboard or gamepad, a text field or a selected text holds the keyboard, or a widget used a key this frame.

Ui.in_top_layer

```
fn bool Ui.in_top_layer(&self)
```

Whether code running now is inside the top layer (or there is no layer).

18 Text and icons

Breaks

```
struct Breaks
```

The boundaries of one string; `Breaks.update` computes them, reusing the memory.

Field	Type	Meaning
`flags`	`List{char}`	per byte offset, 0 to the string's length
`hash`	`ulong`	of the string they are for, to skip recomputing
`len`	`usz`	

`Breaks.init`

```
fn void Breaks.init(&self, Allocator allocator)
```

Prepares the boundaries, keeping their memory in `allocator`.

`Breaks.free`

```
fn void Breaks.free(&self)
```

Frees the boundaries' memory.

`Breaks.update`

```
fn void Breaks.update(&self, String text)
```

Computes the boundaries of `text`, unless they are already for it.

`Breaks.is_grapheme`

```
fn bool Breaks.is_grapheme(&self, usz pos)
```

Whether a grapheme starts or ends at byte `pos`.

`Breaks.is_word`

```
fn bool Breaks.is_word(&self, usz pos)
```

Whether a word boundary falls at byte `pos`.

`Breaks.next_grapheme`

```
fn usz Breaks.next_grapheme(&self, usz pos)
```

The next grapheme boundary after `pos`, or the end.

`Breaks.prev_grapheme`

```
fn usz Breaks.prev_grapheme(&self, usz pos)
```

The grapheme boundary before `pos`, or the start.

Breaks.next_word_end

```
fn usz Breaks.next_word_end(&self, String text, usz pos)
```

The end of the word at or after `pos` (macOS's Option+Right): past any spaces, then to the word's end.

Breaks.next_word_start

```
fn usz Breaks.next_word_start(&self, String text, usz pos)
```

The start of the next word after `pos` (Windows' and Linux's Ctrl+Right): past this word and the spaces after it.

Breaks.prev_word_start

```
fn usz Breaks.prev_word_start(&self, String text, usz pos)
```

The start of the word before `pos` (Ctrl+Left, Option+Left): back over spaces, then to the word's start.

Breaks.word_at

```
fn usz[2] Breaks.word_at(&self, String text, usz pos)
```

The word around `pos`, as a byte range, for a double-click; a run of spaces or punctuation if `pos` is in one.

Ui.text

```
fn void Ui.text(
    &self,
    String text,
    TextSize size = BASE,
    Weight weight = REGULAR,
    Token color = TEXT,
    int max_lines = 0,
    bool wrap = false,
    bool tabular = false,
    Align align = START,
    float opacity = 1,
    bool selectable = false,
    bool ink = false)
```

A run of text at the theme's `size`, in `weight`, colored by `color`.

- `max_lines`: Cut after this many lines and end with …; 0: no limit. Without `wrap`, one line cut with …
- `wrap`: Break lines to fit the width the parent gives
- `tabular`: Equal-width digits, for numbers that change
- `align`: Each line's alignment within the text's box
- `opacity`: 0 to 1; disabled widgets use 0.45
- `selectable`: The user can select it with the pointer and copy it: a message, a path, a title
- `ink`: Ignore the color's RGB and draw black or white, whichever reads on the glass behind it

`Ui.textf`

```
macro void Ui.textf(&self, String fmt)
```

Formats into the frame arena, then shows the result as `Ui.text` does with its defaults.

`Ui.measure_text`

```
fn float[<2>] Ui.measure_text(
    &self,
    String text,
    TextSize size = BASE,
    Weight weight = REGULAR,
    bool tabular = false)
```

The size `text` takes on one line: its width and the line height of `size`.

`TextLines`

```
struct TextLines
```

A text's lines and glyphs, positioned inside `box`.

Field	Type	Meaning
box	Rect	
lines	TextLine[]	
glyphs	LineGlyph[]	
len	usz	the source's length in bytes
breaks	Breaks*	the source's grapheme boundaries

`TextLine`

```
struct TextLine
```

One line: its top and height, its glyphs in visual order, and the bytes it covers.

Field	Type	Meaning
y	float	
height	float	
first	int	its glyphs are glyphs[first:count]
count	int	
right	float	where its last glyph ends
start	usz	the lowest byte it shows
end	usz	the byte after the highest
cut	bool	it ends with the "…" of a cut text

`LineGlyph`

```
struct LineGlyph
```

One glyph on a line: its pen x and advance in points, and the source bytes it shows.

Field	Type	Meaning
x	float	
advance	float	
start	usz	the first byte of its text
end	usz	the byte after it
rtl	bool	
ellipsis	bool	the "..." of a cut line: shows no source bytes

Ui.text_lines

```
fn TextLines Ui.text_lines(
    &self,
    Shaped* s,
    Layout* lay,
    Shaped* ellipsis,
    float size,
    float line_height,
    Rect box,
    Align align,
    usz len,
    Breaks* breaks,
    bool trailing_line = false)
```

Positions the lines of `lay` (laid out from `s`, whose source is `len` bytes long) inside `box`: `size` points per em, `line_height` per line, each line aligned by `align`. `breaks` holds the source's boundaries. With `trailing_line`, an empty line follows the last, for a caret after a final line break, which `text::layout` makes no line for.

TextLines.byte_at

```
fn usz TextLines.byte_at(&self, float x, float y)
```

The grapheme boundary nearest to (`x`, `y`).

TextLines.caret_x

```
fn float TextLines.caret_x(&self, usz i, usz at)
```

The x of a caret at byte `at` on line `i`.

TextLines.line_at

```
fn usz TextLines.line_at(&self, float y)
```

The line at `y`: the first above the text, the last below it.

TextLines.line_of

```
fn usz TextLines.line_of(&self, usz at, bool upstream, String text)
```

The line a caret at byte `at` of `text` shows on. At a soft wrap the same byte ends one line and starts the next: `upstream` picks the line it ends, as after End or a click past the line's end. After a line break, the caret is always on the next line.

`TextLines.edge`

```
fn float TextLines.edge(&self, LineGlyph* g, usz at)
```

The x of byte `at` inside glyph `g` (`g.start <= at <= g.end`): its leading edge at its start (the right edge in a right-to-left run), split evenly by graphemes inside a ligature.

`Ui.selection_rects`

```
fn Rect[] Ui.selection_rects(&self, TextLines* t, usz from, usz to)
```

The rects that show bytes `from` to `to` selected, into the frame arena: one per run of selected glyphs on each line, so a range that crosses directions shows as several. A line the selection continues past is filled to the text's right edge. Ellipsis glyphs are never selected.

`TextSize`

```
enum TextSize : char
```

A step on the theme's type scale, from `XS` (12 points in the built-in themes) to `XXXXL` (36); `BASE` (16) is body text. `Ui.heading` uses `XXXL`, `XXL`, `XL` and `LG` for levels 1 to 4.

Value
XS
SM
BASE
LG
XL
XXL
XXXL
XXXXL

`Weight`

```
enum Weight : char
```

A font weight. `SEMIBOLD` is for headings and button labels; a `FontSet` without a semibold font draws it in the regular font.

Value
REGULAR
SEMIBOLD

`FontSet`

```
struct FontSet
```

What a `Fonts` starts from: one main font per weight and the fallbacks to try after it.

Field	Type	Meaning
`regular`	`char[]`	
`semibold`	`char[]`	empty: use regular
`fallbacks`	`char[][]`	font files in memory, tried in order
`system_fallbacks`	`bool`	also look through the system's font directories
`icons`	`char[]`	the icon font (Lucide); empty: icons draw nothing
`web_fallbacks`	`WebFont[]`	the web: fallback fonts downloaded when text needs them

`text::default_fonts`

```
fn FontSet text::default_fonts()
```

Inter in both weights, Lucide's icons, and the system's fonts for everything else.

`WebFont`

```
struct WebFont
```

A fallback font a web build downloads the first time text needs one of its characters. A page can't look inside a font before downloading it, so `ranges` says which characters it has, written as CSS `unicode-range` is: `"U+0600-06FF, U+0750-077F, U+FB50"`. Other builds ignore web fallbacks.

Field	Type	Meaning
`url`	`String`	relative to the page, or absolute
`ranges`	`String`	

`CodeRange`

```
struct CodeRange
```

Characters from `first` to `last`, both included.

Field	Type	Meaning
`first`	`uint`	
`last`	`uint`	

`text::unicode_ranges`

```
fn CodeRange[]? text::unicode_ranges(String css, Allocator allocator)
```

Code point ranges written as CSS `unicode-range`: entries separated by commas, each `U+` with one hex number, a range `0600-06FF`, or a wildcard `4??` (`400-4FF`); case and spaces don't matter.

`Fonts.add_fallback`

```
fn void? Fonts.add_fallback(&self, char[] data, bool owned)
```

Adds a fallback font after the others, while the app runs: `data` is a font file, which `Fonts` frees when `owned`. Text is shaped again from the next shaping on, so characters it lacked before can come from it. Call it between frames, before any widget asks for text: shaped text from earlier in the frame doesn't survive it.

`text::natural_compare`

```
fn int text::natural_compare(String a, String b)
```

Compares `a` and `b` in natural order, as file managers sort names: runs of digits compare as numbers ("track 2" before "track 10"), letters ignore case, and anything else compares by code point. Strings equal under those rules fall back to comparing their bytes, so the order is total. Negative when `a` goes first, zero when equal, positive otherwise.

`Icon`

```
enum Icon : ushort
```

An icon from the embedded Lucide set, drawn by `Ui.icon` or `Ui.draw_icon`; `NONE` draws nothing.

Value	Associated	Meaning
NONE	{ 0 }	
CHECK	{ 57452 }	check
X	{ 57778 }	x
CHEVRON_UP	{ 57456 }	chevron-up
CHEVRON_DOWN	{ 57453 }	chevron-down
CHEVRON_LEFT	{ 57454 }	chevron-left
CHEVRON_RIGHT	{ 57455 }	chevron-right
CHEVRONS_UP_DOWN	{ 57873 }	chevrons-up-down
PLUS	{ 57661 }	plus
MINUS	{ 57628 }	minus
SEARCH	{ 57681 }	search
MENU	{ 57621 }	menu
SETTINGS	{ 57684 }	settings
PLAY	{ 57660 }	play
PAUSE	{ 57646 }	pause
SKIP_BACK	{ 57695 }	skip-back
SKIP_FORWARD	{ 57696 }	skip-forward
VOLUME_2	{ 57771 }	volume-2
VOLUME_X	{ 57772 }	volume-x
HEART	{ 57586 }	heart
STAR	{ 57718 }	star
TRASH_2	{ 57742 }	trash-2
PENCIL	{ 57849 }	pencil

Value	Associated	Meaning
COPY	{ 57502 }	copy
DOWNLOAD	{ 57522 }	download
UPLOAD	{ 57758 }	upload
INFO	{ 57593 }	info
TRIANGLE_ALERT	{ 57747 }	triangle-alert
CIRCLE_ALERT	{ 57463 }	circle-alert
CIRCLE_CHECK	{ 57894 }	circle-check
EYE	{ 57530 }	eye
EYE_OFF	{ 57531 }	eye-off
HOUSE	{ 57589 }	house
USER	{ 57759 }	user
BELL	{ 57433 }	bell
CALENDAR	{ 57443 }	calendar
FOLDER	{ 57559 }	folder
FILE	{ 57536 }	file
EXTERNAL_LINK	{ 57529 }	external-link
ELLIPSIS	{ 57526 }	ellipsis
ELLIPSIS_VERTICAL	{ 57527 }	ellipsis-vertical
REFRESH_CW	{ 57669 }	refresh-cw
FILTER	{ 57564 }	filter
SLIDERS_HORIZONTAL	{ 58010 }	sliders-horizontal
SUN	{ 57720 }	sun
MOON	{ 57630 }	moon
LOG_OUT	{ 57614 }	log-out
MUSIC	{ 57634 }	music
LIST	{ 57606 }	list
LAYOUT_GRID	{ 57599 }	layout-grid
ARROW_LEFT	{ 57416 }	arrow-left
ARROW_RIGHT	{ 57417 }	arrow-right
MONITOR	{ 57629 }	monitor
SHUFFLE	{ 57694 }	shuffle
REPEAT	{ 57670 }	repeat
REPEAT_1	{ 57853 }	repeat-1
SMARTPHONE	{ 57699 }	smartphone
SPEAKER	{ 57702 }	speaker
LIST_MUSIC	{ 58080 }	list-music
MIC_VOCAL	{ 58185 }	mic-vocal

Value	Associated	Meaning
TAG	{ 57727 }	tag
PIN	{ 57945 }	pin
PIN_OFF	{ 58038 }	pin-off
AUDIO_WAVEFORM	{ 58715 }	audio-waveform

Ui.icon

```
fn void Ui.icon(
    &self,
    Icon icon,
    float size = ICON_SIZE,
    Token color = TEXT,
    float opacity = 1,
    bool ink = false)
```

An icon in a square `size` points wide, colored by `color`. With `ink`, it is black or white, whichever reads on the glass behind it.

Ui.draw_icon

```
fn void Ui.draw_icon(
    &self,
    DrawList* list,
    Icon icon,
    Rect r,
    Color color,
    bool ink = false)
```

Draws `icon` centered in `r`, as large as fits; for widgets' own paint functions.

ui::ICON_SIZE

```
const float ui::ICON_SIZE = 20
```

The size, in points, `Ui.icon` draws an icon at by default.

19 Images

ImageId

```
typedef ImageId = uint
```

A picture made by `Ui.image_create`; 0 is none.

Constant	Type	Value
`ui::NO_IMAGE`	`ImageId`	`0`

Ui.image_create

```
fn ImageId Ui.image_create(&self, int width, int height, char[] rgba)
```

Makes an image `width` × `height` pixels from `rgba`: RGBA8, row by row, sRGB with straight alpha. The pixels are copied, so `rgba` can be freed.

Requires:

- `width > 0 && height > 0 : "an image has at least one pixel"`
- `rgba.len == (sz)width * height * 4 : "four bytes a pixel"`

Ui.image_load

```
fn ImageId Ui.image_load(&self, char[] bytes, int max_size = 0)
```

Makes an image from the bytes of a JPEG or PNG file, decoded in the background: on a job's worker (`os::decode_image`), or by the browser's own decoder on the web. It is `LOADING` until then, drawing a skeleton in its box, and `READY` from the frame after it is decoded, or `FAILED`. Its size is zero until it is ready, so give `ui.image` a size or an `aspect`. `bytes` are copied, so they can be freed at once. `max_size` keeps no more pixels than it can show, as `os::decode_image` does.

Ui.image_load_file

```
fn ImageId Ui.image_load_file(&self, String path, int max_size = 0)
```

Makes an image from a JPEG or PNG file, read and decoded in the background as `Ui.image_load` decodes bytes. A browser has no files to read: there the image is `FAILED`.

Ui.image_state

```
fn ImageState Ui.image_state(&self, ImageId image)
```

Whether `image` is ready to draw, still loading, failed, or none.

ImageState

```
enum ImageState : char
```

Where an image stands: made from pixels or decoded (`READY`), still decoding, or not decodable.

Value	Meaning
`NONE`	no image: `NO_IMAGE`, or a destroyed one

Value	Meaning
LOADING	Ui.image_load is decoding it; it draws a skeleton
READY	
FAILED	its data wasn't a JPEG or PNG Kiln could decode; it draws nothing

Ui.image_update

```
fn void Ui.image_update(&self, ImageId image, char[] rgba)
```

Replaces the pixels of `image`, the same size as before; its smaller copies are made again as they are needed.

Requires:

- `rgba.len == (sz)self.image_size(image).x * self.image_size(image).y * 4 : "the same size, four bytes a pixel"`

Ui.image_destroy

```
fn void Ui.image_destroy(&self, ImageId image)
```

Frees `image`, its pixels and its textures; a load still decoding is cancelled. A freed handle draws nothing.

Ui.image_size

```
fn int[<2>] Ui.image_size(&self, ImageId image)
```

The size of `image` in pixels; zero for none.

Ui.image

```
fn void Ui.image(
    &self,
    ImageId image,
    String label,
    ImageFit fit = FILL,
    ImagePosition position = CENTER,
    float aspect = 0,
    Size width = FIT,
    Size height = FIT,
    Corners radius = NONE,
    float opacity = 1,
    Elevation shadow = NONE)
```

Draws `image` in a box. `label` names it, as HTML's `alt`: it is the image's id and what assistive technology reads; it is never shown. Without a size the box is the picture's own size, a pixel to a point, as in CSS; with one size and no `aspect`, the other follows the picture's shape.

- `fit`: How the picture fills the box (`object-fill`, `-contain`, `-cover`, `-none`, `-scale-down`)
- `position`: Where it sits, and which part `COVER` keeps (`object-center`, `object-top`, ...)
- `aspect`: The box's width divided by its height (`aspect-square` is 1, `aspect-video` 16 / 9); 0 for none
- `radius`: Rounded corners (`rounded-*`); `FULL` makes a square picture a circle
- `opacity`: From 0 (invisible) to 1
- `shadow`: A shadow under the picture (`shadow-*`)

`ImageFit`

```
enum ImageFit : char
```

How a picture fills its box, as CSS's `object-fit`.

Value	Meaning
`FILL`	stretched to the box (CSS's default)
`CONTAIN`	all of it, as large as fits, keeping its shape
`COVER`	the whole box, keeping its shape; what overflows is cut off
`NONE`	its own size, cut off where the box is smaller
`SCALE_DOWN`	NONE when it fits, CONTAIN when it doesn't

`ImagePosition`

```
enum ImagePosition : char
```

Where a picture sits in its box, and which part `COVER` keeps, as CSS's `object-position`.

Value
`CENTER`
`TOP`
`BOTTOM`
`LEFT`
`RIGHT`
`LEFT_TOP`
`RIGHT_TOP`
`LEFT_BOTTOM`
`RIGHT_BOTTOM`

`ImagePosition.along`

```
fn float[<2>] ImagePosition.along(ImagePosition self)
```

How far along each axis `position` puts a picture: 0 at the start, 0.5 in the middle, 1 at the end.

`ImageFitting`

```
struct ImageFitting
```

Where a picture is drawn inside its box, and the part of its texture that shows there.

Field	Type	Meaning
`drawn`	`Rect`	
`tex`	`float[<4>]`	u0, v0, u1, v1, from 0 to 1

ui::image_fit

```
fn ImageFitting ui::image_fit(
    Rect box,
    float[<2>] size,
    ImageFit fit,
    ImagePosition position = CENTER)
```

Fits a picture of `size` points into `box` by `fit`, placed by `position`.

Requires:

- `size.x > 0 && size.y > 0`

ui::image_level

```
fn int ui::image_level(float[<2>] texels, float[<2>] pixels, int levels)
```

The level to draw a picture with: the smallest still at least as many texels as the device pixels it covers, along both axes. `texels` is the part of the level-0 texture shown, `pixels` the device pixels it covers.

ui::MAX_IMAGE_LEVELS

```
const int ui::MAX_IMAGE_LEVELS = 16
```

Levels an image may have: its own size, then each half of the last, down to one pixel.

ui::halve_rgba

```
fn void ui::halve_rgba(char[] src, int sw, int sh, char[] dst, int dw, int dh)
```

Halves RGBA8 pixels: each pixel of `dst`, `dw` × `dh`, the average of the 2 × 2 pixels of `src`, `sw` × `sh`, under it (clamped at an odd edge). Makes an image's smaller levels, and pictures smaller as they are decoded.

Ui.forget_image_textures

```
fn void Ui.forget_image_textures(&self)
```

Forgets every image's texture without freeing it, after the renderer lost its GPU context (as `Atlas.forget_textures` does for glyphs); each is uploaded again from its pixels the next time it is drawn.

20 Widgets

Ui.heading

```
fn void Ui.heading(&self, String text, int level = 1, bool selectable = false)
```

A heading; `level` 1 is the largest, 4 or any other value the smallest. Wraps; `selectable` as for `Ui.text`.

Ui.secondary

```
fn void Ui.secondary(&self, String text, bool selectable = false)
```

Quieter supporting text: smaller, in the `TEXT_SECONDARY` token. Wraps; `selectable` as for `Ui.text`.

Ui.link

```
fn bool Ui.link(&self, String text, bool disabled = false)
```

Text that acts like a button: true when clicked, or on Enter, Space or gamepad A while focused.

Ui.button

```
fn bool Ui.button(
    &self,
    String label,
    Variant variant = DEFAULT,
    ControlSize size = MD,
    bool disabled = false,
    Icon icon = NONE,
    Icon icon_end = NONE,
    bool full = false,
    bool loading = false)
```

A button with a label; true when clicked, or on Enter, Space or gamepad A while focused.

Ui.button_ex

```
fn bool Ui.button_ex(&self, String label, ButtonOpts opts)
```

`Ui.button` with its options in a `ButtonOpts`, for presets and forwarding.

ButtonOpts

```
struct ButtonOpts
```

A button's options as a struct, for `Ui.button_ex`: presets kept in a variable and forwarded.

Field	Type	Meaning
variant	Variant	
size	ControlSize	
disabled	bool	
icon	Icon	before the label

Field	Type	Meaning
`icon_end`	`Icon`	after the label
`full`	`bool`	as wide as the parent allows
`loading`	`bool`	a spinner over it, keeping its size; it doesn't click

`Variant`

```
enum Variant : char
```

What a button means, which picks its fill, border and label colors.

Value	Meaning
`DEFAULT`	neutral fill
`PRIMARY`	the accent: the main action
`OUTLINE`	a hairline, no fill
`GHOST`	no fill until hovered: toolbars
`DANGER`	destructive actions

`ControlSize`

```
enum ControlSize : char
```

A button's size: its height, padding, text size and icon size.

Value	Meaning
`MD`	36 points high, the default
`SM`	28 points high
`LG`	44 points high, with larger text

`Ui.icon_button`

```
fn bool Ui.icon_button(
    &self,
    Icon icon,
    String label,
    Variant variant = GHOST,
    ControlSize size = MD,
    bool disabled = false,
    int badge = -1)
```

A square button showing only `icon`; `label` names it and is its id. `badge` puts a badge on its corner: a count above 0, a dot at 0, none below (`Ui.@badged`).

`Ui.checkbox`

```
fn bool Ui.checkbox(
    &self,
    String label,
    bool* value,
```

```
    bool disabled = false,
    bool mixed = false)
```

A checkbox; true when a click, Enter, Space or gamepad A changed `value`.
- `mixed`: Shows a minus: some, not all, of what it stands for is on. A click turns it on

`Ui.toggle`

```
fn bool Ui.toggle(&self, String label, bool* value, bool disabled = false)
```

A switch for something that takes effect at once; true when it changed `value`.

`Ui.radio`

```
fn bool Ui.radio(&self, String label, bool selected, bool disabled = false)
```

One option of a radio group, for custom layouts inside `Ui.@focus_group`; true when it was chosen. `Ui.radio_group` is the usual way.

`Ui.radio_group`

```
fn bool Ui.radio_group(
    &self,
    String id,
    int* selected,
    String[] options,
    Dir dir = COLUMN,
    bool disabled = false)
```

A group of radio buttons choosing one of `options` into `selected`; true when the choice changed.

`Ui.slider`

```
fn SliderResult Ui.slider(
    &self,
    String label,
    float* value,
    float min = 0,
    float max = 1,
    float step = 0,
    String format = "",
    bool disabled = false)
```

A slider over [min, max]. `label`'s shown part is drawn before the track ("##name" draws none); `step` 0 is continuous; a non-empty `format` shows the value after the track.

Requires:
- `min <= max : "a slider's min is above its max"`

`SliderResult`

```
bitstruct SliderResult : char
```

What `Ui.slider` did this frame.

Field	Type	Meaning
changed	bool	the value moved this frame
released	bool	the user let go: the end of a drag, or a keyboard or gamepad step

ui::DISABLED_OPACITY

```
const float ui::DISABLED_OPACITY = 0.45
```

The opacity a disabled button, toggle or slider draws at.

Ui.show_demo

```
fn void Ui.show_demo(&self, bool* open = null)
```

Shows Kiln's demo panel: widgets to try and a theme editor. `open`, when given, gets a close button and is cleared by it.

21 Theme and color

Theme

```
struct Theme
```

Everything that sets the look: the built-in token colors, the spacing step, corner radii, the type scale and the layout breakpoints. A theme is a plain value; start from `themes::LIGHT` or `themes::DARK` and make it current with `Ui.set_theme`.

The anonymous member is a union of `colors`, a `ThemeColors` with one field per built-in token, and `by_token`, the same colors as an array indexed by `Token`. `radii` holds the `Radius` scale in points, `text` the `TextStyle` of each `TextSize`, and `breakpoints` the least layout width, in points, of each `Breakpoint`.

Field	Type	Meaning
`dark`	`bool`	custom tokens take their dark value; accent companions are derived for a dark surface
anonymous		
`spacing_base`	`float`	points per spacing step
`radius`	`Radii`	corner radii by kind of thing, for `Corners`
`radii`	`float[7]`	the `Radius` scale, in points
`text`	`TextStyle[8]`	the `TextSize` scale
`breakpoints`	`float[5]`	least layout width, in points, of each `Breakpoint`
`backdrop_opacity`	`float`	how strongly BACKDROP dims the page behind a modal layer; 0 for none
`follow_system`	`bool`	only in `themes::SYSTEM`: the `Ui` resolves it from the OS and follows changes

ThemeColors

```
struct ThemeColors
```

The colors of a `Theme`, one field per built-in `Token`, in token order. An accent companion (`accent_hover`, `accent_pressed`, `accent_subtle`, `on_accent`, `focus_ring`, `selection`) left all zero is derived from `accent`, and a status tint (`success_subtle`, `warning_subtle`, `danger_subtle`) from its status color.

Field	Type	Meaning
`surface`	`Oklch`	
`surface_raised`	`Oklch`	
`background`	`Oklch`	
`text`	`Oklch`	
`text_secondary`	`Oklch`	
`text_disabled`	`Oklch`	
`border`	`Oklch`	

Field	Type	Meaning
track	Oklch	
accent	Oklch	
accent_hover	Oklch	
accent_pressed	Oklch	
accent_subtle	Oklch	
on_accent	Oklch	
focus_ring	Oklch	
danger	Oklch	
success	Oklch	
warning	Oklch	
backdrop	Oklch	
selection	Oklch	
success_subtle	Oklch	
warning_subtle	Oklch	
danger_subtle	Oklch	
series	Oklch[8]	left all zero: Kiln's palette for light or dark

Token

```
constdef Token : ushort
```

A color by its role. Widgets, boxes, text and icons name a token and `Ui.c` gives its color in the current `Theme`; `Ui.push_token` overrides one and `Ui.add_token` registers new ones beside the built-in values.

Value	Associated	Meaning
SURFACE	0	window and page background
SURFACE_RAISED	1	cards, panels, popovers
BACKGROUND	2	controls' resting fill (fields, buttons' neutral variant)
TEXT	3	body text
TEXT_SECONDARY	4	quieter supporting text
TEXT_DISABLED	5	text of disabled controls
BORDER	6	outlines and separators
TRACK	7	the unfilled part of switches, sliders, progress bars
ACCENT	8	the brand color of primary and selected controls
ACCENT_HOVER	9	ACCENT under the pointer
ACCENT_PRESSED	10	ACCENT while pressed
ACCENT_SUBTLE	11	tinted backgrounds: selected rows, soft buttons
ON_ACCENT	12	text and icons drawn on ACCENT
FOCUS_RING	13	the ring around the focused control

Value	Associated	Meaning
DANGER	14	destructive actions and errors
SUCCESS	15	
WARNING	16	
BACKDROP	17	dims the page behind a modal layer, at the theme's backdrop_opacity
SELECTION	18	the background of selected text; derived from ACCENT
SUCCESS_SUBTLE	19	tinted backgrounds of success alerts and badges; derived from SUCCESS
WARNING_SUBTLE	20	the same for WARNING
DANGER_SUBTLE	21	the same for DANGER
SERIES_1	22	chart series, in order: eight hues apart enough to tell side by side
SERIES_2	23	
SERIES_3	24	
SERIES_4	25	
SERIES_5	26	
SERIES_6	27	
SERIES_7	28	
SERIES_8	29	

ui::NO_TOKEN

```
const Token ui::NO_TOKEN = 65535
```

No color: a box whose `bg` is `NO_TOKEN` draws no fill.

Radii

```
struct Radii
```

Radii for the three kinds of rounded thing a widget draws.

Field	Type	Meaning
control	float	buttons, and boxes with radius: CONTROL
panel	float	cards, and boxes with radius: PANEL
surface	float	boxes with radius: SURFACE

Radius

```
enum Radius : char
```

A step on the theme's corner-radius scale, from `XS` (2 points in the built-in themes) to `XXXL` (24); `Ui.radius` gives its value.

Value
XS
SM

Value
MD
LG
XL
XXL
XXXL

`TextStyle`

```
struct TextStyle
```

The font size and line height of one `TextSize`, in points.

Field	Type	Meaning
size	float	
line_height	float	

`Breakpoint`

```
enum Breakpoint : char
```

A layout width threshold, from `BASE` (any width) to `XL` (1280 points in the built-in themes). `Ui.bp` gives the widest one the window reaches and `Ui.at` picks a value by it.

Value
BASE
SM
MD
LG
XL

`themes::LIGHT`

```
const Theme themes::LIGHT
```

The light theme: zinc surfaces, a blue accent, a 4-point spacing step.

`themes::DARK`

```
const Theme themes::DARK
```

The dark theme: `LIGHT` with dark zinc surfaces and lighter accent and status colors.

`themes::SYSTEM`

```
const Theme themes::SYSTEM = { .follow_system = true }
```

Not a theme but a request to follow the operating system: a `Ui` given it (the default of `app::run`, `sdl::create_ui` and `raylib::create_ui`) uses `themes::from_os` and keeps following the OS setting. Pass any other theme to override it.

themes::from_os

```
fn Theme themes::from_os()
```

The built-in theme that matches the operating system: `DARK` when the user prefers dark interfaces, `LIGHT` otherwise, with the system's accent color in place of the blue one and its companions derived from it, and no backdrop dim on macOS, which doesn't dim behind modal windows. `LIGHT` when the OS can't be asked (see `os::theme`).

themes::with_os_accent

```
fn Theme themes::with_os_accent(Theme theme)
```

`theme` with the OS's accent color over its own, when the OS has one, as an app with its own palette may want (`Ui.set_themes(..., os_accent: true)`).

Ui.set_theme

```
fn void Ui.set_theme(&self, Theme theme)
```

Makes `theme` current: resolves every token, built-in and custom, and drops every override pushed with `Ui.push_token`. `themes::SYSTEM` makes the colors follow the operating system: its light or dark theme and accent (`themes::from_os`) now, and again at the start of a frame after the OS setting changes. It also makes the OS's settings follow it: reduced motion, scrollbars, the wheel step, the text size and the rest, unless the app pins one (`Ui.set_reduced_motion`, `Ui.set_always_show_scrollbars`, `Ui.set_wheel_step`...). Another theme stops only the colors following: an app with its own palette keeps the user's settings. For a light and a dark palette that switch with the OS, see `Ui.set_themes`.

Ui.set_themes

```
fn void Ui.set_themes(&self, Theme light, Theme dark, bool os_accent = false)
```

Makes `light` or `dark` current by the OS's color scheme (no preference counts as light), and again at the start of a frame after the OS switches. With `os_accent`, the OS's accent lies over the palette's when the OS has one. The OS's settings follow it, as with `themes::SYSTEM`. `Ui.set_theme` stops the switching.

Ui.follow_os_settings

```
fn void Ui.follow_os_settings(&self, bool on = true)
```

Makes the OS's settings follow it, or with `false` stops them, whatever the colors; pinned ones stay the app's.

Ui.os_changed

```
fn void Ui.os_changed(&self)
```

What follows an OS change, as `Ui.begin` calls it when the OS reports one: the settings, if they follow the OS, and the colors, if they do (`themes::SYSTEM` or `Ui.set_themes`).

`OsSetting`

```
enum OsSetting : char
```

An OS setting the `Ui` follows while its theme is `themes::SYSTEM`, unless the app pins it.

Value	Meaning
REDUCED_MOTION	Ui.reduced_motion
SCROLLBARS	Ui.always_show_scrollbars
WHEEL_STEP	Ui.wheel_step
TOOLTIP_DELAY	Ui.tooltip_delay
SUBMENU_DELAY	Ui.submenu_delay
DIALOG_ORDER	Ui.dialog_confirm_first
CARET_BLINK	Ui.caret_blink
DOUBLE_CLICK	Ui.double_click_time
TEXT_KEYS	Ui.text_keys
DECIMAL_SEPARATOR	Ui.decimal_separator
LIST_PLACEMENT	Ui.list_placement
TEXT_ORDER	Ui.text_order
HIGH_CONTRAST	Ui.high_contrast
REDUCED_TRANSPARENCY	Ui.reduced_transparency
TEXT_SCALE	Ui.text_scale
MIDDLE_CLICK_PASTE	Ui.middle_click_paste

`Ui.set_reduced_motion`

```
fn void Ui.set_reduced_motion(&self, bool on)
```

Turns reduced motion on or off for this app, whatever the OS says, until `Ui.follow_os(REDUCED_MOTION)`. Prefer a calmer design to ignoring a user who asked for less motion.

`Ui.set_always_show_scrollbars`

```
fn void Ui.set_always_show_scrollbars(&self, bool on)
```

Keeps scrollbars shown (true) or fading in on use (false), whatever the OS says, until `Ui.follow_os(SCROLLBARS)`.

`Ui.set_wheel_step`

```
fn void Ui.set_wheel_step(&self, float points)
```

Sets the points one wheel notch scrolls (`ui::WHEEL_PAGE` for one view), whatever the OS says, until `Ui.follow_os(WHEEL_STEP)`.

Ui.set_tooltip_delay

```
fn void Ui.set_tooltip_delay(&self, float seconds)
```

Sets the seconds a tooltip waits before it shows, whatever the OS says, until `Ui.follow_os(TOOLTIP_DELAY)`.

Ui.set_submenu_delay

```
fn void Ui.set_submenu_delay(&self, float seconds)
```

Sets the seconds the pointer rests on a menu row before its submenu opens, whatever the OS says, until `Ui.follow_os(SUBMENU_DELAY)`.

Ui.set_dialog_confirm_first

```
fn void Ui.set_dialog_confirm_first(&self, bool first)
```

Puts a dialog's confirming button first (true) or last, whatever the platform does, until `Ui.follow_os(DIALOG_ORDER)`.

Ui.set_caret_blink

```
fn void Ui.set_caret_blink(&self, float seconds)
```

Sets the seconds the caret stays on, then off (-1: no blinking), whatever the OS says, until `Ui.follow_os(CARET_BLINK)`.

Ui.set_double_click_time

```
fn void Ui.set_double_click_time(&self, float seconds)
```

Sets the seconds between the clicks of a double-click, whatever the OS says, until `Ui.follow_os(DOUBLE_CLICK)`.

Ui.set_text_keys

```
fn void Ui.set_text_keys(&self, TextKeys keys)
```

Sets the keys text fields answer to, whatever the platform is, until `Ui.follow_os(TEXT_KEYS)`.

TextKeys

```
enum TextKeys : char
```

The keys a text field answers to: Windows' and Linux's, or macOS's.

Value	Meaning
PC	Ctrl+arrows by word, Home and End, Ctrl for commands
MAC	Option+arrows by word, Command+arrows to the line's ends, Command for commands

ui::platform_text_keys

```
fn TextKeys ui::platform_text_keys()
```

The platform's text keys: MAC on macOS, PC elsewhere.

Ui.set_decimal_separator

```
fn void Ui.set_decimal_separator(&self, uint codepoint)
```

Sets the character numbers use between their whole and fractional parts (a code point, such as ','), whatever the OS says, until `Ui.follow_os(DECIMAL_SEPARATOR)`.

Ui.set_list_placement

```
fn void Ui.set_list_placement(&self, ListPlacement placement)
```

Sets where a select opens its list, whatever the platform is, until `Ui.follow_os(LIST_PLACEMENT)`.

ListPlacement

```
enum ListPlacement : char
```

Where a select opens its list.

Value	Meaning
BELOW	under the button (Windows, Linux, Android, the web)
OVER	over the button, with the chosen row where the button is (macOS)

Ui.set_high_contrast

```
fn void Ui.set_high_contrast(&self, bool on)
```

Asks for more contrast (true) or the usual look, whatever the OS says, until `Ui.follow_os(HIGH_CONTRAST)`. Status widgets outline themselves under it, and badges take the body text color; prefer adapting a design to ignoring a user who asked for more contrast.

Ui.set_reduced_transparency

```
fn void Ui.set_reduced_transparency(&self, bool on)
```

Asks for surfaces that hide what is behind them (true) or the usual transparency, whatever the OS says, until `Ui.follow_os(REDUCED_TRANSPARENCY)`. A glass dock raises its tint and stays solid enough to read on a photo; a design may also drop its blur. The platform layer can pin this from the OS.

Ui.set_text_scale

```
fn void Ui.set_text_scale(&self, float scale)
```

Scales the whole UI by `scale` (1 is the design's size), whatever the OS says, until `Ui.follow_os(TEXT_SCALE)`. Text, controls and spacing grow together, as CSS sizes in rem do with the browser's font size, so nothing overflows; the window holds less. Prefer offering larger text to ignoring a user who asked for it.

ui::MIN_TEXT_SCALE

```
const float ui::MIN_TEXT_SCALE = 0.5
```

The smallest and largest text scale Kiln applies; the OS's and the app's are kept between them.

Constant	Type	Value
ui::MAX_TEXT_SCALE	float	3

Ui.set_text_order

```
fn void Ui.set_text_order(&self, TextOrder order)
```

Sets how sorted text is ordered, whatever the OS offers, until `Ui.follow_os(TEXT_ORDER)`.

TextOrder

```
enum TextOrder : char
```

How sorted text is ordered.

Value	Meaning
NATURAL	Kiln's natural order (`text::natural_compare`): the same on every machine
SYSTEM	the OS's order for the user's language (`os::collate`), where the OS has one

ui::system_text_order

```
fn TextOrder ui::system_text_order()
```

`SYSTEM` where the OS compares text (Windows, macOS), `NATURAL` elsewhere.

Ui.compare_text

```
fn int Ui.compare_text(&self, String a, String b)
```

Compares `a` and `b` in the `Ui`'s text order: negative when `a` goes first, zero when equal, positive otherwise.

ui::platform_list_placement

```
fn ListPlacement ui::platform_list_placement()
```

The platform's placement: `OVER` on macOS, `BELOW` elsewhere.

ui::DEFAULT_CARET_BLINK

```
const float ui::DEFAULT_CARET_BLINK = 0.5
```

Seconds the caret stays on, then off, when the OS has no setting for it.

ui::DEFAULT_DOUBLE_CLICK

```
const float ui::DEFAULT_DOUBLE_CLICK = 0.4
```

Seconds between the clicks of a double-click when the OS has no setting for it.

ui::platform_confirm_first

```
fn bool ui::platform_confirm_first()
```

The platform's dialog button order: the confirming button first on Windows and on KDE (Qt's layout), last elsewhere (macOS, GNOME and most Linux desktops, Android, the web). On Linux the desktop comes from XDG_CURRENT_DESKTOP.

ui::DEFAULT_TOOLTIP_DELAY

```
const float ui::DEFAULT_TOOLTIP_DELAY = 0.5
```

Seconds a tooltip waits when the OS has no setting for it.

ui::DEFAULT_SUBMENU_DELAY

```
const float ui::DEFAULT_SUBMENU_DELAY = 0.2
```

Seconds a submenu waits when the OS has no setting for it.

Ui.follow_os

```
fn void Ui.follow_os(&self, OsSetting setting)
```

Lets go of a pinned setting: it follows the OS again, at once while the theme is `themes::SYSTEM`.

Ui.@theme

```
macro void Ui.@theme(&self, Theme theme; @body())
```

Runs the body with `theme` and puts the current theme back after; token overrides pushed around it still hold afterwards. A gallery shows light and dark side by side this way: widgets read their colors while the frame is built.

Overrides pushed inside the body must be popped inside it.

Ui.c

```
fn Color Ui.c(&self, Token token)
```

The resolved color of `token`.

Requires:

- `(sz)token < self.token_colors.len() : "unknown token"`

Ui.push_token

```
fn void Ui.push_token(&self, Token token, Oklch value)
```

Overrides `token` with `value` until the matching `Ui.pop_token`. Overriding `ACCENT` also re-derives its hover, pressed, subtle, on-accent and focus colors, and overriding a status color re-derives its tint.

Requires:

- `(sz)token < self.token_values.len() : "unknown token"`

Ui.pop_token

```
fn void Ui.pop_token(&self)
```

Ends the latest `Ui.push_token`, giving the token (and, for `ACCENT`, its companions) back the colors they had before it.

Requires:

- `self.overrides.len() > 0 : "pop_token without push_token"`

`Ui.add_token`

```
fn Token Ui.add_token(&self, String name, Oklch light, Oklch dark)
```

Registers a token with a value for light themes and one for dark themes. `name` documents the token at the call site and isn't stored; the returned token is what code uses.

`Ui.space`

```
fn float Ui.space(&self, float n)
```

`n` spacing steps in points.

`Ui.radius`

```
fn float Ui.radius(&self, Radius r)
```

The current theme's corner radius for step `r` of the `Radius` scale, in points.

`Ui.text_style`

```
fn TextStyle Ui.text_style(&self, TextSize s)
```

The current theme's font size and line height for `s`.

`Color`

```
struct Color
```

An 8-bit sRGB color with straight (not premultiplied) alpha, what paint functions and backends draw with. `Ui.c` gives a token's `Color`.

Field	Type	Meaning
r	char	
g	char	
b	char	
a	char	opacity, 0 to 255

`Color.alpha`

```
fn Color Color.alpha(Color self, float factor)
```

The same color with its opacity scaled by `factor`.

`Color.mix`

```
fn Color Color.mix(Color self, Color other, float t)
```

Blends toward `other` by `t` (0 keeps this color, 1 gives `other`), in OKLab so the midpoint looks halfway.

`Color.equals`

```
fn bool Color.equals(Color self, Color other)
```

Whether both colors have the same four channels; also the == operator.

`Color.to_oklch`

```
fn Oklch Color.to_oklch(Color self)
```

The color in OKLCH, ignoring its alpha.

`Color.to_oklab`

```
fn Oklab Color.to_oklab(Color self)
```

The color in OKLab, ignoring its alpha.

`Color.to_hsv`

```
fn Hsv Color.to_hsv(Color self)
```

The color in HSV, ignoring its alpha. A grey has hue 0.

`Color.to_hex`

```
fn String Color.to_hex(Color self, bool alpha = false, Allocator allocator = tmem)
```

"#3B82F6", or with `alpha` "#3B82F680", in `allocator`.

`ui::color_from_hex`

```
fn Color? ui::color_from_hex(String s)
```

The color written as "#rgb", "#rgba", "#rrggbb" or "#rrggbbaa", in either case, with or without the "#" and with spaces around; NOT_FOUND for anything else. Without an alpha, the color is opaque.

`Hsv`

```
struct Hsv
```

A color as hue (0–360 degrees), saturation and value (0–1): what a color picker shows.

Field	Type	Meaning
h	float	
s	float	
v	float	

`Hsv.to_color`

```
fn Color Hsv.to_color(Hsv self, char alpha = 255)
```

The color this HSV names, with `alpha` (0 to 255).

`ui::WHITE`

```
const Oklch ui::WHITE = { 1, 0, 0 }
```

Pure white in OKLCH.

`ui::BLACK`

```
const Oklch ui::BLACK = { 0, 0, 0 }
```

Pure black in OKLCH (all zero, so `Oklch.is_zero` holds for it).

`Oklch`

```
struct Oklch
```

A color in OKLCH: perceptual lightness, chroma and hue. Themes and tokens are written in it; `Oklch.to_color` gives the `Color` to draw.

Field	Type	Meaning
`l`	`float`	lightness, 0 to 1
`c`	`float`	chroma, 0 to about 0.4
`h`	`float`	hue in degrees

`Oklch.to_color`

```
fn Color Oklch.to_color(Oklch self, float alpha = 1)
```

Converts to sRGB, mapping out-of-gamut colors as CSS does.

- `alpha`: Opacity, 0 to 1

`Oklch.lighten`

```
fn Oklch Oklch.lighten(Oklch self, float dl)
```

The same hue and chroma, `dl` lighter (negative: darker), kept within 0 to 1.

`Oklch.is_zero`

```
fn bool Oklch.is_zero(Oklch self)
```

Whether all three components are 0. A `ThemeColors` accent companion left zero is derived from the accent.

`Oklch.to_oklab`

```
fn Oklab Oklch.to_oklab(Oklch self)
```

The same color in OKLab.

`Oklab`

```
struct Oklab
```

A color in OKLab, the rectangular form of `Oklch`, where straight-line blends and distances are perceptually even.

Field	Type	Meaning
`l`	`float`	lightness, 0 to 1
`a`	`float`	green (negative) to red (positive)
`b`	`float`	blue (negative) to yellow (positive)

`Oklab.to_oklch`

```
fn Oklch Oklab.to_oklch(Oklab self)
```

The same color in OKLCH; a gray (chroma near 0) gets hue 0.

`Oklab.to_linear_srgb`

```
fn float[<3>] Oklab.to_linear_srgb(Oklab self)
```

The same color as linear-light sRGB red, green and blue, unclamped: an out-of-gamut color has components below 0 or above 1.

`ui::oklab_from_linear_srgb`

```
fn Oklab ui::oklab_from_linear_srgb(float[<3>] rgb)
```

The OKLab color of `rgb`, linear-light sRGB red, green and blue.

Constant	Type	Value
`themes::ZINC_50`	`Oklch`	`{ 0.9850, 0, 0 }`
`themes::ZINC_100`	`Oklch`	`{ 0.9670, 0.001, 286.375 }`
`themes::ZINC_200`	`Oklch`	`{ 0.9200, 0.004, 286.32 }`
`themes::ZINC_300`	`Oklch`	`{ 0.8710, 0.006, 286.286 }`
`themes::ZINC_400`	`Oklch`	`{ 0.7050, 0.015, 286.067 }`
`themes::ZINC_500`	`Oklch`	`{ 0.5520, 0.016, 285.938 }`
`themes::ZINC_600`	`Oklch`	`{ 0.4420, 0.017, 285.786 }`
`themes::ZINC_700`	`Oklch`	`{ 0.3700, 0.013, 285.805 }`
`themes::ZINC_800`	`Oklch`	`{ 0.2740, 0.006, 286.033 }`
`themes::ZINC_900`	`Oklch`	`{ 0.2100, 0.006, 285.885 }`
`themes::ZINC_950`	`Oklch`	`{ 0.1410, 0.005, 285.823 }`
`themes::BLUE_50`	`Oklch`	`{ 0.9700, 0.014, 254.604 }`
`themes::BLUE_100`	`Oklch`	`{ 0.9320, 0.032, 255.585 }`
`themes::BLUE_200`	`Oklch`	`{ 0.8820, 0.059, 254.128 }`
`themes::BLUE_300`	`Oklch`	`{ 0.8090, 0.105, 251.813 }`
`themes::BLUE_400`	`Oklch`	`{ 0.7070, 0.165, 254.624 }`
`themes::BLUE_500`	`Oklch`	`{ 0.6230, 0.214, 259.815 }`
`themes::BLUE_600`	`Oklch`	`{ 0.5460, 0.245, 262.881 }`

Constant	Type	Value
themes::BLUE_700	Oklch	{ 0.4880, 0.243, 264.376 }
themes::BLUE_800	Oklch	{ 0.4240, 0.199, 265.638 }
themes::BLUE_900	Oklch	{ 0.3790, 0.146, 265.522 }
themes::BLUE_950	Oklch	{ 0.2820, 0.091, 267.935 }
themes::RED_50	Oklch	{ 0.9710, 0.013, 17.38 }
themes::RED_100	Oklch	{ 0.9360, 0.032, 17.717 }
themes::RED_200	Oklch	{ 0.8850, 0.062, 18.334 }
themes::RED_300	Oklch	{ 0.8080, 0.114, 19.571 }
themes::RED_400	Oklch	{ 0.7040, 0.191, 22.216 }
themes::RED_500	Oklch	{ 0.6370, 0.237, 25.331 }
themes::RED_600	Oklch	{ 0.5770, 0.245, 27.325 }
themes::RED_700	Oklch	{ 0.5050, 0.213, 27.518 }
themes::RED_800	Oklch	{ 0.4440, 0.177, 26.899 }
themes::RED_900	Oklch	{ 0.3960, 0.141, 25.723 }
themes::RED_950	Oklch	{ 0.2580, 0.092, 26.042 }
themes::GREEN_50	Oklch	{ 0.9820, 0.018, 155.826 }
themes::GREEN_100	Oklch	{ 0.9620, 0.044, 156.743 }
themes::GREEN_200	Oklch	{ 0.9250, 0.084, 155.995 }
themes::GREEN_300	Oklch	{ 0.8710, 0.15, 154.449 }
themes::GREEN_400	Oklch	{ 0.7920, 0.209, 151.711 }
themes::GREEN_500	Oklch	{ 0.7230, 0.219, 149.579 }
themes::GREEN_600	Oklch	{ 0.6270, 0.194, 149.214 }
themes::GREEN_700	Oklch	{ 0.5270, 0.154, 150.069 }
themes::GREEN_800	Oklch	{ 0.4480, 0.119, 151.328 }
themes::GREEN_900	Oklch	{ 0.3930, 0.095, 152.535 }
themes::GREEN_950	Oklch	{ 0.2660, 0.065, 152.934 }
themes::AMBER_50	Oklch	{ 0.9870, 0.022, 95.277 }
themes::AMBER_100	Oklch	{ 0.9620, 0.059, 95.617 }
themes::AMBER_200	Oklch	{ 0.9240, 0.12, 95.746 }
themes::AMBER_300	Oklch	{ 0.8790, 0.169, 91.605 }
themes::AMBER_400	Oklch	{ 0.8280, 0.189, 84.429 }
themes::AMBER_500	Oklch	{ 0.7690, 0.188, 70.08 }
themes::AMBER_600	Oklch	{ 0.6660, 0.179, 58.318 }
themes::AMBER_700	Oklch	{ 0.5550, 0.163, 48.998 }
themes::AMBER_800	Oklch	{ 0.4730, 0.137, 46.201 }
themes::AMBER_900	Oklch	{ 0.4140, 0.112, 45.904 }
themes::AMBER_950	Oklch	{ 0.2790, 0.077, 45.635 }

Constant	Type	Value
themes::SLATE_50	Oklch	{ 0.9840, 0.003, 247.858 }
themes::SLATE_100	Oklch	{ 0.9680, 0.007, 247.896 }
themes::SLATE_200	Oklch	{ 0.9290, 0.013, 255.508 }
themes::SLATE_300	Oklch	{ 0.8690, 0.022, 252.894 }
themes::SLATE_400	Oklch	{ 0.7040, 0.04, 256.788 }
themes::SLATE_500	Oklch	{ 0.5540, 0.046, 257.417 }
themes::SLATE_600	Oklch	{ 0.4460, 0.043, 257.281 }
themes::SLATE_700	Oklch	{ 0.3720, 0.044, 257.287 }
themes::SLATE_800	Oklch	{ 0.2790, 0.041, 260.031 }
themes::SLATE_900	Oklch	{ 0.2080, 0.042, 265.755 }
themes::SLATE_950	Oklch	{ 0.1290, 0.042, 264.695 }
themes::GRAY_50	Oklch	{ 0.9850, 0.002, 247.839 }
themes::GRAY_100	Oklch	{ 0.9670, 0.003, 264.542 }
themes::GRAY_200	Oklch	{ 0.9280, 0.006, 264.531 }
themes::GRAY_300	Oklch	{ 0.8720, 0.01, 258.338 }
themes::GRAY_400	Oklch	{ 0.7070, 0.022, 261.325 }
themes::GRAY_500	Oklch	{ 0.5510, 0.027, 264.364 }
themes::GRAY_600	Oklch	{ 0.4460, 0.03, 256.802 }
themes::GRAY_700	Oklch	{ 0.3730, 0.034, 259.733 }
themes::GRAY_800	Oklch	{ 0.2780, 0.033, 256.848 }
themes::GRAY_900	Oklch	{ 0.2100, 0.034, 264.665 }
themes::GRAY_950	Oklch	{ 0.1300, 0.028, 261.692 }
themes::NEUTRAL_50	Oklch	{ 0.9850, 0, 0 }
themes::NEUTRAL_100	Oklch	{ 0.9700, 0, 0 }
themes::NEUTRAL_200	Oklch	{ 0.9220, 0, 0 }
themes::NEUTRAL_300	Oklch	{ 0.8700, 0, 0 }
themes::NEUTRAL_400	Oklch	{ 0.7080, 0, 0 }
themes::NEUTRAL_500	Oklch	{ 0.5560, 0, 0 }
themes::NEUTRAL_600	Oklch	{ 0.4390, 0, 0 }
themes::NEUTRAL_700	Oklch	{ 0.3710, 0, 0 }
themes::NEUTRAL_800	Oklch	{ 0.2690, 0, 0 }
themes::NEUTRAL_900	Oklch	{ 0.2050, 0, 0 }
themes::NEUTRAL_950	Oklch	{ 0.1450, 0, 0 }
themes::STONE_50	Oklch	{ 0.9850, 0.001, 106.423 }
themes::STONE_100	Oklch	{ 0.9700, 0.001, 106.424 }
themes::STONE_200	Oklch	{ 0.9230, 0.003, 48.717 }
themes::STONE_300	Oklch	{ 0.8690, 0.005, 56.366 }

Constant	Type	Value
themes::STONE_400	Oklch	{ 0.7090, 0.01, 56.259 }
themes::STONE_500	Oklch	{ 0.5530, 0.013, 58.071 }
themes::STONE_600	Oklch	{ 0.4440, 0.011, 73.639 }
themes::STONE_700	Oklch	{ 0.3740, 0.01, 67.558 }
themes::STONE_800	Oklch	{ 0.2680, 0.007, 34.298 }
themes::STONE_900	Oklch	{ 0.2160, 0.006, 56.043 }
themes::STONE_950	Oklch	{ 0.1470, 0.004, 49.25 }
themes::ORANGE_50	Oklch	{ 0.9800, 0.016, 73.684 }
themes::ORANGE_100	Oklch	{ 0.9540, 0.038, 75.164 }
themes::ORANGE_200	Oklch	{ 0.9010, 0.076, 70.697 }
themes::ORANGE_300	Oklch	{ 0.8370, 0.128, 66.29 }
themes::ORANGE_400	Oklch	{ 0.7500, 0.183, 55.934 }
themes::ORANGE_500	Oklch	{ 0.7050, 0.213, 47.604 }
themes::ORANGE_600	Oklch	{ 0.6460, 0.222, 41.116 }
themes::ORANGE_700	Oklch	{ 0.5530, 0.195, 38.402 }
themes::ORANGE_800	Oklch	{ 0.4700, 0.157, 37.304 }
themes::ORANGE_900	Oklch	{ 0.4080, 0.123, 38.172 }
themes::ORANGE_950	Oklch	{ 0.2660, 0.079, 36.259 }
themes::YELLOW_50	Oklch	{ 0.9870, 0.026, 102.212 }
themes::YELLOW_100	Oklch	{ 0.9730, 0.071, 103.193 }
themes::YELLOW_200	Oklch	{ 0.9450, 0.129, 101.54 }
themes::YELLOW_300	Oklch	{ 0.9050, 0.182, 98.111 }
themes::YELLOW_400	Oklch	{ 0.8520, 0.199, 91.936 }
themes::YELLOW_500	Oklch	{ 0.7950, 0.184, 86.047 }
themes::YELLOW_600	Oklch	{ 0.6810, 0.162, 75.834 }
themes::YELLOW_700	Oklch	{ 0.5540, 0.135, 66.442 }
themes::YELLOW_800	Oklch	{ 0.4760, 0.114, 61.907 }
themes::YELLOW_900	Oklch	{ 0.4210, 0.095, 57.708 }
themes::YELLOW_950	Oklch	{ 0.2860, 0.066, 53.813 }
themes::LIME_50	Oklch	{ 0.9860, 0.031, 120.757 }
themes::LIME_100	Oklch	{ 0.9670, 0.067, 122.328 }
themes::LIME_200	Oklch	{ 0.9380, 0.127, 124.321 }
themes::LIME_300	Oklch	{ 0.8970, 0.196, 126.665 }
themes::LIME_400	Oklch	{ 0.8410, 0.238, 128.85 }
themes::LIME_500	Oklch	{ 0.7680, 0.233, 130.85 }
themes::LIME_600	Oklch	{ 0.6480, 0.2, 131.684 }
themes::LIME_700	Oklch	{ 0.5320, 0.157, 131.589 }

Constant	Type	Value
themes::LIME_800	Oklch	{ 0.4530, 0.124, 130.933 }
themes::LIME_900	Oklch	{ 0.4050, 0.101, 131.063 }
themes::LIME_950	Oklch	{ 0.2740, 0.072, 132.109 }
themes::EMERALD_50	Oklch	{ 0.9790, 0.021, 166.113 }
themes::EMERALD_100	Oklch	{ 0.9500, 0.052, 163.051 }
themes::EMERALD_200	Oklch	{ 0.9050, 0.093, 164.15 }
themes::EMERALD_300	Oklch	{ 0.8450, 0.143, 164.978 }
themes::EMERALD_400	Oklch	{ 0.7650, 0.177, 163.223 }
themes::EMERALD_500	Oklch	{ 0.6960, 0.17, 162.48 }
themes::EMERALD_600	Oklch	{ 0.5960, 0.145, 163.225 }
themes::EMERALD_700	Oklch	{ 0.5080, 0.118, 165.612 }
themes::EMERALD_800	Oklch	{ 0.4320, 0.095, 166.913 }
themes::EMERALD_900	Oklch	{ 0.3780, 0.077, 168.94 }
themes::EMERALD_950	Oklch	{ 0.2620, 0.051, 172.552 }
themes::TEAL_50	Oklch	{ 0.9840, 0.014, 180.72 }
themes::TEAL_100	Oklch	{ 0.9530, 0.051, 180.801 }
themes::TEAL_200	Oklch	{ 0.9100, 0.096, 180.426 }
themes::TEAL_300	Oklch	{ 0.8550, 0.138, 181.071 }
themes::TEAL_400	Oklch	{ 0.7770, 0.152, 181.912 }
themes::TEAL_500	Oklch	{ 0.7040, 0.14, 182.503 }
themes::TEAL_600	Oklch	{ 0.6000, 0.118, 184.704 }
themes::TEAL_700	Oklch	{ 0.5110, 0.096, 186.391 }
themes::TEAL_800	Oklch	{ 0.4370, 0.078, 188.216 }
themes::TEAL_900	Oklch	{ 0.3860, 0.063, 188.416 }
themes::TEAL_950	Oklch	{ 0.2770, 0.046, 192.524 }
themes::CYAN_50	Oklch	{ 0.9840, 0.019, 200.873 }
themes::CYAN_100	Oklch	{ 0.9560, 0.045, 203.388 }
themes::CYAN_200	Oklch	{ 0.9170, 0.08, 205.041 }
themes::CYAN_300	Oklch	{ 0.8650, 0.127, 207.078 }
themes::CYAN_400	Oklch	{ 0.7890, 0.154, 211.53 }
themes::CYAN_500	Oklch	{ 0.7150, 0.143, 215.221 }
themes::CYAN_600	Oklch	{ 0.6090, 0.126, 221.723 }
themes::CYAN_700	Oklch	{ 0.5200, 0.105, 223.128 }
themes::CYAN_800	Oklch	{ 0.4500, 0.085, 224.283 }
themes::CYAN_900	Oklch	{ 0.3980, 0.07, 227.392 }
themes::CYAN_950	Oklch	{ 0.3020, 0.056, 229.695 }
themes::SKY_50	Oklch	{ 0.9770, 0.013, 236.62 }

Constant	Type	Value
themes::SKY_100	Oklch	{ 0.9510, 0.026, 236.824 }
themes::SKY_200	Oklch	{ 0.9010, 0.058, 230.902 }
themes::SKY_300	Oklch	{ 0.8280, 0.111, 230.318 }
themes::SKY_400	Oklch	{ 0.7460, 0.16, 232.661 }
themes::SKY_500	Oklch	{ 0.6850, 0.169, 237.323 }
themes::SKY_600	Oklch	{ 0.5880, 0.158, 241.966 }
themes::SKY_700	Oklch	{ 0.5000, 0.134, 242.749 }
themes::SKY_800	Oklch	{ 0.4430, 0.11, 240.79 }
themes::SKY_900	Oklch	{ 0.3910, 0.09, 240.876 }
themes::SKY_950	Oklch	{ 0.2930, 0.066, 243.157 }
themes::INDIGO_50	Oklch	{ 0.9620, 0.018, 272.314 }
themes::INDIGO_100	Oklch	{ 0.9300, 0.034, 272.788 }
themes::INDIGO_200	Oklch	{ 0.8700, 0.065, 274.039 }
themes::INDIGO_300	Oklch	{ 0.7850, 0.115, 274.713 }
themes::INDIGO_400	Oklch	{ 0.6730, 0.182, 276.935 }
themes::INDIGO_500	Oklch	{ 0.5850, 0.233, 277.117 }
themes::INDIGO_600	Oklch	{ 0.5110, 0.262, 276.966 }
themes::INDIGO_700	Oklch	{ 0.4570, 0.24, 277.023 }
themes::INDIGO_800	Oklch	{ 0.3980, 0.195, 277.366 }
themes::INDIGO_900	Oklch	{ 0.3590, 0.144, 278.697 }
themes::INDIGO_950	Oklch	{ 0.2570, 0.09, 281.288 }
themes::VIOLET_50	Oklch	{ 0.9690, 0.016, 293.756 }
themes::VIOLET_100	Oklch	{ 0.9430, 0.029, 294.588 }
themes::VIOLET_200	Oklch	{ 0.8940, 0.057, 293.283 }
themes::VIOLET_300	Oklch	{ 0.8110, 0.111, 293.571 }
themes::VIOLET_400	Oklch	{ 0.7020, 0.183, 293.541 }
themes::VIOLET_500	Oklch	{ 0.6060, 0.25, 292.717 }
themes::VIOLET_600	Oklch	{ 0.5410, 0.281, 293.009 }
themes::VIOLET_700	Oklch	{ 0.4910, 0.27, 292.581 }
themes::VIOLET_800	Oklch	{ 0.4320, 0.232, 292.759 }
themes::VIOLET_900	Oklch	{ 0.3800, 0.189, 293.745 }
themes::VIOLET_950	Oklch	{ 0.2830, 0.141, 291.089 }
themes::PURPLE_50	Oklch	{ 0.9770, 0.014, 308.299 }
themes::PURPLE_100	Oklch	{ 0.9460, 0.033, 307.174 }
themes::PURPLE_200	Oklch	{ 0.9020, 0.063, 306.703 }
themes::PURPLE_300	Oklch	{ 0.8270, 0.119, 306.383 }
themes::PURPLE_400	Oklch	{ 0.7140, 0.203, 305.504 }

Constant	Type	Value
themes::PURPLE_500	Oklch	{ 0.6270, 0.265, 303.9 }
themes::PURPLE_600	Oklch	{ 0.5580, 0.288, 302.321 }
themes::PURPLE_700	Oklch	{ 0.4960, 0.265, 301.924 }
themes::PURPLE_800	Oklch	{ 0.4380, 0.218, 303.724 }
themes::PURPLE_900	Oklch	{ 0.3810, 0.176, 304.987 }
themes::PURPLE_950	Oklch	{ 0.2910, 0.149, 302.717 }
themes::FUCHSIA_50	Oklch	{ 0.9770, 0.017, 320.058 }
themes::FUCHSIA_100	Oklch	{ 0.9520, 0.037, 318.852 }
themes::FUCHSIA_200	Oklch	{ 0.9030, 0.076, 319.62 }
themes::FUCHSIA_300	Oklch	{ 0.8330, 0.145, 321.434 }
themes::FUCHSIA_400	Oklch	{ 0.7400, 0.238, 322.16 }
themes::FUCHSIA_500	Oklch	{ 0.6670, 0.295, 322.15 }
themes::FUCHSIA_600	Oklch	{ 0.5910, 0.293, 322.896 }
themes::FUCHSIA_700	Oklch	{ 0.5180, 0.253, 323.949 }
themes::FUCHSIA_800	Oklch	{ 0.4520, 0.211, 324.591 }
themes::FUCHSIA_900	Oklch	{ 0.4010, 0.17, 325.612 }
themes::FUCHSIA_950	Oklch	{ 0.2930, 0.136, 325.661 }
themes::PINK_50	Oklch	{ 0.9710, 0.014, 343.198 }
themes::PINK_100	Oklch	{ 0.9480, 0.028, 342.258 }
themes::PINK_200	Oklch	{ 0.8990, 0.061, 343.231 }
themes::PINK_300	Oklch	{ 0.8230, 0.12, 346.018 }
themes::PINK_400	Oklch	{ 0.7180, 0.202, 349.761 }
themes::PINK_500	Oklch	{ 0.6560, 0.241, 354.308 }
themes::PINK_600	Oklch	{ 0.5920, 0.249, 0.584 }
themes::PINK_700	Oklch	{ 0.5250, 0.223, 3.958 }
themes::PINK_800	Oklch	{ 0.4590, 0.187, 3.815 }
themes::PINK_900	Oklch	{ 0.4080, 0.153, 2.432 }
themes::PINK_950	Oklch	{ 0.2840, 0.109, 3.907 }
themes::ROSE_50	Oklch	{ 0.9690, 0.015, 12.422 }
themes::ROSE_100	Oklch	{ 0.9410, 0.03, 12.58 }
themes::ROSE_200	Oklch	{ 0.8920, 0.058, 10.001 }
themes::ROSE_300	Oklch	{ 0.8100, 0.117, 11.638 }
themes::ROSE_400	Oklch	{ 0.7120, 0.194, 13.428 }
themes::ROSE_500	Oklch	{ 0.6450, 0.246, 16.439 }
themes::ROSE_600	Oklch	{ 0.5860, 0.253, 17.585 }
themes::ROSE_700	Oklch	{ 0.5140, 0.222, 16.935 }
themes::ROSE_800	Oklch	{ 0.4550, 0.188, 13.697 }

Constant	Type	Value
themes::ROSE_900	Oklch	{ 0.4100, 0.159, 10.272 }
themes::ROSE_950	Oklch	{ 0.2710, 0.105, 12.094 }

22 Motion

`Spring`

```
struct Spring
```

An animated value that chases its target like a spring. Keep one in a widget's state, call `Spring.go` once a frame and draw with `value`.

Field	Type	Meaning
`value`	`float`	where the spring is now; what you draw with
`velocity`	`float`	units per second
`target`	`float`	where it is heading

`Spring.go`

```
fn void Spring.go(&self, float target, Motion motion)
```

Moves toward `target` by one frame of `motion`, and keeps the Ui animating until the spring settles. Outside a frame, or with reduced motion on, jumps straight there.

`Spring.settled`

```
fn bool Spring.settled(&self)
```

Whether the spring is at rest on its target.

`Spring.reset`

```
fn void Spring.reset(&self, float value)
```

Puts the spring at `value`, at rest.

`Spring.step`

```
fn void Spring.step(&self, float dt, float duration, float bounce)
```

Advances the spring by `dt` seconds with a motion's `duration` and `bounce`, without touching the Ui. Substeps keep a long frame stable, and a step covers at most `MAX_STEP`: the first frame after an app has slept has the whole sleep as its `dt`.

Requires:

- `duration > 0 : "a spring needs a duration"`

`Motion`

```
enum Motion : char
```

A motion token: the duration in seconds and the bounce (0 never overshoots) of one kind of animation, passed to `Spring.go`.

Value	Associated	Meaning
`ROW`	`{ 0.1, 0 }`	a selected row, a key step, a press

Value	Associated	Meaning
HOVER	{ 0.14, 0 }	hover color, edges
FOCUS	{ 0.14, 0 }	focus ring appearing
STATE	{ 0.18, 0.15 }	check, selection, switch
DISCLOSURE	{ 0.32, 0 }	content opening, such as a section expanding
POPOVER	{ 0.28, 0.15 }	a small floating layer appearing
DIALOG	{ 0.34, 0.15 }	a modal layer appearing
EXIT	{ 0.16, 0 }	any floating layer leaving
SCROLL	{ 0.2, 0 }	a scroll area settling after a wheel step or a jump

Loop

```
enum Loop : char
```

A motion that repeats while something waits: one period per turn or breath.

Value	Associated	Meaning
SPIN	{ 0.9 }	a spinner's turn
PULSE	{ 1.6 }	a skeleton's breath
SLIDE	{ 1.4 }	an indeterminate progress bar's pass
PING	{ 1.5 }	a notification dot's ring, growing and fading

Ui.loop_phase

```
fn float Ui.loop_phase(&self, Loop loop)
```

Where `loop` is in its period this frame, 0 to 1, and keeps frames coming while it is asked for. Under reduced motion it holds still: at the start of a turn, a pass or a ring, and halfway into a breath (a quarter of its period). Widgets drop a slide or a ring under it rather than show it frozen.

ui::MAX_STEP

```
const float ui::MAX_STEP = 0.1
```

The longest step a spring takes in one frame, in seconds.

23 Drawing

ui::draw_spinner

```
fn void ui::draw_spinner(
    DrawList* list,
    float[<2>] center,
    float size,
    float phase,
    Color color)
```

Eight dots round a circle `size` points across at `center`; the one at `phase` (0 to 1 round the circle) is the brightest, and the dots behind it fade.

DrawList

```
struct DrawList
```

What the core hands a renderer each frame: vertices, 16-bit indices and the `DrawCmd` batches that draw them. A `PaintFn` adds shapes to it with `DrawList.rect` and the other drawing methods.

DrawList.rect

```
fn void DrawList.rect(&self, Rect r, Color color, float radius = 0)
```

A filled rectangle with rounded corners.

DrawList.gradient_rect

```
fn void DrawList.gradient_rect(&self, Rect r, Gradient g, float radius = 0)
```

A rectangle with rounded corners filled with `g`. The gradient line runs through the center, as long as CSS makes it (|w·sin a| + |h·cos a|), so its start and end touch the box's edges at any angle. The box is drawn as strips (a grid for a diagonal) whose vertices carry the gradient's color there; each carries the whole shape, so corners, edges and clipping are a rect's, and the renderers blend linearly between them, close enough to the OKLab blend at this many steps. All in the page's one draw call.

DrawList.hue_rect

```
fn void DrawList.hue_rect(&self, Rect r, float radius)
```

A rounded rectangle filled with every hue from left to right, red to red through yellow, green, cyan, blue and magenta, for a colour picker's hue bar. Each of the six bands blends linearly between two of those, which is exactly how HSV moves between them.

DrawList.quad4

```
fn void DrawList.quad4(
    &self,
    Rect box,
    float[<2>] center,
    Color[4] colors,
```

```
    float[<4>] sdf,
    bool flip = false)
```

DrawList.quad with a color at each corner: top left, top right, bottom right, bottom left. Its two triangles meet on the diagonal from the top left, or with flip from the top right.

Gradient

```
struct Gradient
```

A linear gradient, as CSS's linear-gradient and Tailwind's bg-linear-*: the colors change along a line through the box's center at angle, and are the same across it. Between stops they blend in OKLab, as Tailwind's do, so two saturated colors don't pass through grey.

Field	Type	Meaning
angle	float	degrees, as CSS: 0 runs up, 90 right, 180 down, 270 left
stops	GradientStop[4]	in order along the line, at rising
count	int	the stops used, 2 to 4

GradientStop

```
struct GradientStop
```

One color of a gradient and where it sits on the gradient line: 0 at its start, 1 at its end.

Field	Type	Meaning
color	Color	
at	float	

Gradient.at

```
fn Color Gradient.at(&self, float t)
```

The color at t on the gradient line: the first stop's before it, the last's after it.

ui::gradient

```
fn Gradient ui::gradient(float angle, Color from, Color to)
```

A gradient from from at the start to to at the end (from-* to-*).

ui::gradient3

```
fn Gradient ui::gradient3(float angle, Color from, Color via, Color to)
```

A gradient through via in the middle (from-* via-* to-*).

Constant	Type	Value
ui::TO_TOP	float	0
ui::TO_TOP_RIGHT	float	45
ui::TO_RIGHT	float	90

Constant	Type	Value
ui::TO_BOTTOM_RIGHT	float	135
ui::TO_BOTTOM	float	180
ui::TO_BOTTOM_LEFT	float	225
ui::TO_LEFT	float	270
ui::TO_TOP_LEFT	float	315

DrawList.border

```
fn void DrawList.border(&self, Rect r, Color color, float width, float radius = 0)
```

An outline `width` points wide, drawn inside `r`.

DrawList.circle

```
fn void DrawList.circle(&self, float[<2>] center, float radius, Color color)
```

A filled circle of `radius` points around `center`.

DrawList.triangle

```
fn void DrawList.triangle(&self, Rect r, Side toward, Color color)
```

A filled triangle filling `r`, pointing at its `toward` edge: the point at that edge's middle, the base along the opposite edge, as a popover's arrow. Antialiased like the other shapes.

DrawList.image

```
fn void DrawList.image(
    &self,
    Rect r,
    float[<4>] tex,
    Color color,
    float radius,
    TextureId texture,
    TexturePending pending)
```

A picture filling `r`, its texture's `tex` rectangle (u0, v0, u1, v1, from 0 to 1) stretched over it, with corners rounded by `radius`: the edge is antialiased as a shape's is, so a circle comes out smooth. `texture` and `pending` name the texture as `DrawList.cmd_for` takes them.

DrawList.mesh_begin

```
fn void DrawList.mesh_begin(&self, usz vertices)
```

Starts a mesh of up to `vertices` vertices in the current clip, in MESH mode: add them with `DrawList.mesh_vertex` and their triangles with `DrawList.mesh_triangle`, without changing the clip in between. A mesh has at most `ui::MAX_BATCH_VERTICES` vertices.

Requires:

- `vertices <= MAX_BATCH_VERTICES`

DrawList.mesh_vertex

```
fn ushort DrawList.mesh_vertex(&self, float[<2>] pos, Color color)
```

Adds a mesh vertex and returns its index for `DrawList.mesh_triangle`.

DrawList.mesh_triangle

```
fn void DrawList.mesh_triangle(&self, ushort a, ushort b, ushort c)
```

Adds a triangle between three vertices `DrawList.mesh_vertex` returned.

DrawList.mesh_vertices

```
fn ushort DrawList.mesh_vertices(&self, Vertex[] vertices)
```

Adds mesh vertices at once (only their position and color count) and returns the first one's index.

DrawList.mesh_triangles

```
fn void DrawList.mesh_triangles(&self, ushort[] indices)
```

Adds triangles at once: three indices each, as `DrawList.mesh_vertex` and `DrawList.mesh_vertices` returned them.

DrawList.fill_polygon

```
fn void DrawList.fill_polygon(
    &self,
    Point[] points,
    Color color,
    float fringe,
    Allocator scratch)
```

Fills the simple polygon `pts` (convex or concave, not crossing itself) in `color`, with an antialiased fringe `fringe` points wide.

DrawList.stroke_polyline

```
fn void DrawList.stroke_polyline(
    &self,
    Point[] points,
    bool closed,
    Color color,
    StrokeStyle* style,
    float fringe,
    Allocator scratch)
```

Strokes the polyline `pts` (closed or open) in `color` with `style`, its edges antialiased over `fringe` points. A line thinner than the fringe is drawn as wide as the fringe with its alpha scaled down, so it fades rather than breaks.

Ui.draw_text_at

```
fn void Ui.draw_text_at(
    &self,
```

```
    DrawList* list,
    Shaped* shaped,
    Point at,
    TextSize size,
    Weight weight,
    Color color,
    Align align = START,
    TextBaseline baseline = ALPHABETIC)
```

Draws `shaped` text on one line at `at`, placed by `align` across and `baseline` up and down, for paint functions that draw labels.

`Ui.baseline_y`

```
fn float Ui.baseline_y(
    &self,
    float y,
    TextSize size,
    Weight weight,
    TextBaseline baseline)
```

The baseline of text at `size` and `weight` placed at height `y` by `baseline`.

`ui::triangle_distance`

```
fn float ui::triangle_distance(float[<2>] p, float[<2>] half_size, int kind)
```

The signed distance from `p` to a triangle filling the box of `half_size` around the origin, its point at the middle of one edge: `kind` 1 down, 2 up, 3 left, 4 right (the SHAPE vertices' negative stroke). Renderers evaluate it per pixel, as they do the rounded box.

`DrawList.shadow`

```
fn void DrawList.shadow(
    &self,
    Rect r,
    Color color,
    float blur,
    float radius = 0,
    float offset_y = 0)
```

A soft shadow under `r`, spread `blur` points and moved down by `offset_y`.

`DrawList.glass`

```
fn void DrawList.glass(&self, Rect r, Color tint, float blur = 10)
```

Adds a glass capsule: a live backdrop material for a floating navigation dock. The renderer samples the scene behind `r`, blurs and tints it, and draws the rim and shadow. Renderers without backdrop support draw the box's solid surface instead. `blur` is in points.

Constant	Type	Value
`ui::MODE_GLASS`	`float`	`4`
`ui::MODE_INK_TEXT`	`float`	`5`

ui::GLASS_TEXTURE

```
const TextureId ui::GLASS_TEXTURE = 4294967295
```

A command texture that marks a glass quad rather than a sampled image.

DrawList.focus_ring

```
fn void DrawList.focus_ring(&self, Rect r, Color color, float radius, float t)
```

A focus ring: a 2 pt outline 2 pt outside `r`, with corner radius `radius` + 4, and `color`'s alpha scaled by `t` (animate `t` with a `Spring` and `Motion` FOCUS).

DrawList.glyph

```
fn void DrawList.glyph(
    &self,
    Rect r,
    float[<4>] uv,
    Color color,
    uint page,
    bool ink = false)
```

A glyph quad: `r` in points, `uv` the glyph's pixel rectangle on atlas page `page` (x0, y0, x1, y1). `Ui.draw` swaps in the page's texture and normalizes the coordinates once the pages are uploaded.

DrawList.quad

```
fn void DrawList.quad(
    &self,
    Rect box,
    float[<2>] center,
    Color color,
    float[<4>] sdf,
    float blur)
```

Adds one quad covering `box` (already grown for blur and antialiasing), with `center` and `sdf` describing the shape inside it.

DrawList.push_clip

```
fn void DrawList.push_clip(&self, Rect rect)
```

Restricts drawing to `rect` (within the current clip) until `DrawList.pop_clip`.

DrawList.pop_clip

```
fn void DrawList.pop_clip(&self)
```

Restores the clip that was current before the last `DrawList.push_clip`.

DrawList.clip

```
fn Rect DrawList.clip(&self)
```

The current clip rectangle, or `ui::NO_CLIP` when none is pushed.

`ui::NO_CLIP`

```
const Rect ui::NO_CLIP = { -1e9, -1e9, 2e9, 2e9 }
```

The clip rectangle with nothing pushed: large enough to clip nothing.

`Vertex`

```
struct Vertex
```

One corner of a quad in a `DrawList`.

Field	Type	Meaning
`pos`	`float[<2>]`	logical points
`uv`	`float[<2>]`	SHAPE, IMAGE: position relative to the shape's center; TEXT: texture coordinates
`color`	`Color`	
`sdf`	`float[<4>]`	SHAPE: half width, half height, corner radius, stroke width (0 = filled); IMAGE: the same three, then the texture's u
`blur`	`float`	SHAPE: shadow softness in points, replacing the 1-pixel edge ramp; IMAGE: the texture's v
`clip`	`float[<4>]`	the clip rectangle's corners in points (x0, y0, x1, y1); fragments outside it are dropped
`mode`	`float`	the `DrawMode` ordinal: how its fragments are colored

`DrawCmd`

```
struct DrawCmd
```

A batch of quads and mesh triangles a renderer draws in one call. Each vertex carries its own clip and mode, so a batch ends only where the texture changes or its vertex range is full.

Field	Type	Meaning
`texture`	`TextureId`	0 while only shapes and meshes are in it, which need none
`pending`	`TexturePending`	what `texture` stands for until `Ui.draw` swaps in a real texture
`vertex_base`	`uint`	added to every index of this command
`first_index`	`uint`	where its indices start in the list's indices
`index_count`	`uint`	six per quad, three per mesh triangle

`TexturePending`

```
enum TexturePending : char
```

What a command's texture stands for while the frame is built: the real textures are made when it is drawn.

Value	Meaning
`NONE`	a real texture, or none
`ATLAS_PAGE`	a glyph-atlas page, plus one
`IMAGE`	an `ImageId` and one of its levels (`image_texture_ref`)

`DrawMode`

```
enum DrawMode : char
```

How a renderer colors the fragments of a `DrawCmd`.

Value	Meaning
SHAPE	a rounded box from the vertex's sdf, filled or stroked, maybe blurred
TEXT	the vertex color, its alpha scaled by the texture's red channel (glyph coverage)
IMAGE	the texture's RGBA times the vertex color, inside a rounded box as SHAPE's, antialiased
MESH	any triangles, the vertex color interpolated; an alpha-0 fringe antialiases their edges
GLASS	a live backdrop material: blur, tint and rim from the scene behind
INK_TEXT	text whose color comes from the backdrop behind it: black or white, whichever reads

Constant	Type	Value
ui::MODE_SHAPE	float	0
ui::MODE_TEXT	float	1
ui::MODE_IMAGE	float	2
ui::MODE_MESH	float	3

`TextureId`

```
alias TextureId = uint
```

A renderer's handle for a texture, made by `Renderer.texture_create`; 0 means none.

24 Backends

Renderer

```
interface Renderer
{
    <* Starts a frame: clears the drawable to `clear`, `pixel_width` × `pixel_height` pixels. *>
    fn void begin_frame(Color clear, int pixel_width, int pixel_height);
    <* Presents the frame the renderer drew. *>
    fn void present();
    <* Reads the drawable just drawn, top row first, as RGBA8 into `out`. *>
    fn void read_pixels(char[] out, int width, int height);
    <* Creates a `width` × `height` texture in `format`, filled from `pixels` when it is not empty. *>
    fn TextureId texture_create(int width, int height, PixelFormat format, char[] pixels);
    <* Replaces the pixel rectangle `area` of `tex` with `pixels`, packed row by row. *>
    fn void texture_update(TextureId tex, Rect area, char[] pixels);
    <* Frees `tex`. *>
    fn void texture_destroy(TextureId tex);
    <* Whether render() can draw a live backdrop glass command. *>
    fn bool supports_glass();
    <* Draws `list` over a viewport `width` × `height` logical points, `scale` pixels per point. *>
    fn void render(DrawList* list, float width, float height, float scale);
    <* The GPU context is gone: forget every GPU handle without freeing it. *>
    fn void on_gpu_lost();
    <* Frees the renderer and every texture it still holds. *>
    fn void destroy();
}
```

The drawing backend: owns textures and draws the `DrawList` that `Ui.draw` hands it. `GlRenderer`, `RaylibRenderer` and `SoftwareRenderer` implement it.

Platform

```
interface Platform
{
    <*
     Fills `input` with this frame's events. With `wait`, first sleeps until the
     next event, a `wake()`, or `until` (seconds on `time()`'s clock; 0 for no
     deadline), on a platform that `wakes()`; one that doesn't may ignore `until`.
    *>
    fn void poll(Input* input, bool wait, double until);
    <* Wakes a waiting poll from another thread. *>
    fn void wake();
    <*
     True when `wake()` wakes a waiting poll from another thread and `poll` sleeps
     only until `until`, so the `Ui` can let an App-style loop sleep through
     deadlines and running jobs. False where it can't (raylib): frames then keep
     coming until they pass.
    *>
    fn bool wakes();
    <* Seconds since the platform started. *>
    fn double time();
    <* Pixels per logical point on the current display. *>
    fn float dpi_scale();
    <* The screen edges that are covered or unsafe to draw controls under. *>
    fn Insets insets();
    <* Turns text entry (the on-screen keyboard, an input method) on or off for a field, saying what
it holds and where. *>
    fn void text_input(bool on, TextInput input);
    <* Moves a focused field's text area and caret while text entry is on: after typing, scrolling,
the window moving. *>
    fn void text_input_area(Rect area, float caret);
```

```
    <* Ends what an input method is composing, as when focus moves; the input method commits or drops
it. *>
    fn void end_composition();
    <* Tells the platform whether the app has an in-app back target (a layer or a page), so it can
register the OS's back gesture; ignored where there is none. *>
    fn void back_target(bool active);
    <* A copy of the clipboard's text, made with `allocator`; empty when there is none. *>
    fn String clipboard_get(Allocator allocator);
    <* Whether the clipboard holds text, asked without reading it where the OS can tell (Android would
show its clipboard preview on a read). *>
    fn bool clipboard_has();
    <* Puts `text` on the clipboard. *>
    fn void clipboard_set(String text);
    <*
     A copy of the primary selection, the text last selected in any app, made with
     `allocator`: what a middle click pastes on Linux and the BSDs. Empty where the
     OS has none.
    *>
    fn String primary_get(Allocator allocator);
    <* Makes `text` the primary selection; does nothing where the OS has none. *>
    fn void primary_set(String text);
    <* Shows `cursor` as the mouse cursor. *>
    fn void set_cursor(Cursor cursor);
    <* The OS's handle of the window (an HWND on Windows, an NSWindow on macOS, the X11 window or the
Wayland surface on Linux); null for none. *>
    fn void* native_window();
    <* Reads the whole file at `path` into memory made with `allocator`. *>
    fn char[]? load_asset(String path, Allocator allocator);
    <* Frees the platform. *>
    fn void destroy();
}
```

The system backend: input, time, clipboard, cursor and assets for a `Ui`. `Ui.begin` polls it when no `Input` is passed. `RaylibPlatform`, `SdlPlatform` and `HeadlessPlatform` implement it.

`PixelFormat`

```
enum PixelFormat : char
```

The pixel layout of a texture a `Renderer` creates.

Value	Meaning
R8	coverage, for the glyph atlas
RGBA8	8-bit red, green, blue, alpha

`Cursor`

```
enum Cursor : char
```

A mouse cursor shape; `Ui.end` passes it to `Platform.set_cursor` when it changes.

Value	Meaning
ARROW	
POINTER	a pointing hand, over something clickable
TEXT	an I-beam, over editable text
RESIZE_EW	left-right resize

Value	Meaning
RESIZE_NS	up-down resize
NOT_ALLOWED	

Insets

```
struct Insets
```

Safe-area insets: cutouts, gesture bars, the on-screen keyboard.

Field	Type	Meaning
top	float	
right	float	
bottom	float	
left	float	

TextInput

```
struct TextInput
```

A focused field's text entry, as the platform hears of it: `area` is the field's text in window points and `caret` the caret's x from its left, for placing an input method's candidate window (clear of the field) or a Steam Deck's keyboard.

Field	Type	Meaning
area	Rect	
caret	float	
kind	InputKind	
password	bool	
multiline	bool	Enter makes a new line

RaylibPlatform

```
struct RaylibPlatform
```

The `Platform` for raylib windows; `raylib::create_ui` attaches one.

RaylibRenderer

```
struct RaylibRenderer
```

The `Renderer` that draws through raylib's OpenGL layer; `raylib::create_ui` attaches one.

SdlPlatform

```
struct SdlPlatform
```

The `Platform` for an SDL3 window; `sdl::create_ui` attaches one.

SdlPlatform.set_title

```
fn void SdlPlatform.set_title(&self, String title)
```

Sets the window's title.

SdlPlatform.set_size

```
fn void SdlPlatform.set_size(&self, int width, int height)
```

Resizes the window in points.

SdlPlatform.set_min_size

```
fn void SdlPlatform.set_min_size(&self, int width, int height)
```

Sets the smallest size the window accepts, in points.

GlRenderer

```
struct GlRenderer
```

The `Renderer` that draws through OpenGL directly.

SdlGpuRenderer

```
struct SdlGpuRenderer
```

The `Renderer` for an SDL_GPU device and its claimed window; `sdl::create_ui` attaches one when `KILN_RENDERER=gpu`, and `sdlgpu::new_renderer` makes one for a `Ui` of your own.

gl::load

```
fn void? gl::load(LoadFn load)
```

Looks up every GL function the renderer calls in the current context; fails, naming it, when one is missing.

LoadFn

```
alias LoadFn = fn void*(ZString name)
```

Looks up a GL function by name in the current context; null when it has none.

gl::new_renderer

```
fn GlRenderer* gl::new_renderer(Allocator allocator)
```

A renderer for the current context, which `gl::load` has loaded the functions of; `Ui.destroy` frees it.

gl::MISSING_FUNCTION

```
fault gl::MISSING_FUNCTION
```

`gl::load` found no function of that name in the driver; it says which.

sdlgpu::new_renderer

```
fn SdlGpuRenderer* sdlgpu::new_renderer(
    Allocator allocator,
    SdlGPUDevice* device,
    SdlWindow* window)
```

Opens a GPU renderer for a claimed SDL window; null when a GPU object can't be made.

25 Operating system

os::theme

```
fn SystemTheme? os::theme()
```

The system's appearance settings: dark or light, the accent color, reduced motion and high contrast. Settings the OS doesn't have stay at their zero value (`has_accent` false, no preference); fails with `os::UNSUPPORTED` when the OS can't be asked at all, such as a Linux session without a D-Bus session bus or desktop portal.

os::decimal_separator

```
fn uint os::decimal_separator()
```

The character the user's region writes between a number's whole and fractional parts (`.` in English, `,` in German), as a code point; 0 when the OS has no setting. It is read from the locale the app started with; Android reads its `DecimalFormatSymbols`.

os::calendar_settings

```
fn CalendarSettings os::calendar_settings(Allocator allocator)
```

The calendar settings of the user's region, read from the locale the app started with. What the OS doesn't give falls back to English names, Monday first, a 24-hour clock written 09:30, AM and PM, and 2026-11-25. Free it with `CalendarSettings.free`.

os::english_calendar

```
fn CalendarSettings os::english_calendar(
    Allocator allocator,
    int first_weekday = 0,
    bool hour12 = true,
    DateOrder order = MDY,
    String separator = "/")
```

English settings in `allocator`: what tests pin, and what a missing setting falls back to.

CalendarSettings

```
struct CalendarSettings
```

What `calendar_settings` reads from the OS. Its strings belong to the allocator it was given.

Field	Type	Meaning
`first_weekday`	`int`	0 Sunday ... 6 Saturday
`hour12`	`bool`	the clock shows AM and PM
`order`	`DateOrder`	
`date_separator`	`String`	between a short date's parts: "/", ".", "-"
`day_unpadded`	`bool`	the day is written without a leading zero: 5, not 05
`month_unpadded`	`bool`	

Field	Type	Meaning
`months`	`String[12]`	standalone names, as a calendar's title shows them
`months_short`	`String[12]`	
`weekdays_short`	`String[7]`	from Sunday
`time_separator`	`String`	between the hour, minute and second: ":", "."
`am`	`String`	the day period words of a 12-hour clock: "AM", "a. m.", " "
`pm`	`String`	
`period_first`	`bool`	the day period comes before the hour: " 3:30"
`storage`	`char[]`	the block every string above points into

`CalendarSettings.free`

```
fn void CalendarSettings.free(&self, Allocator allocator)
```

Frees the strings, which `allocator` holds.

`DateOrder`

```
enum DateOrder : char
```

The order of a short date's parts: 2026-11-25, 25/11/2026, 11/25/2026.

Value
`YMD`
`DMY`
`MDY`

`os::collate`

```
fn int? os::collate(String a, String b)
```

Compares two strings the way the user's language sorts them, as the platform's file manager does: digits as numbers, case ignored. Negative when `a` goes first, zero when equal, positive otherwise. Windows uses `CompareStringEx`, macOS `localizedStandardCompare:`, Android 24 and later ICU's `RuleBasedCollator` with numeric collation; elsewhere it is `UNSUPPORTED`, and callers fall back to `text::natural_compare`.

`os::set_progress`

```
fn void? os::set_progress(void* window, float value)
```

Shows a long task's progress on the app's taskbar button, Dock icon or notification: `value` from 0 to 1, `PROGRESS_NONE` to take it away, or `PROGRESS_UNKNOWN` for work whose end isn't known. `window` is the platform's handle of the window (`Platform.native_window`). Windows uses the taskbar (`ITaskbarList3`), macOS the Dock tile, Linux the `com.canonical.Unity.LauncherEntry` signal that KDE Plasma, Ubuntu's dock and Dash to Dock show, for the desktop file `os::set_desktop_id` names, and Android an ongoing notification (ignoring `window`; the first call asks for the notification permission). Fails with `UNSUPPORTED` where there is no such place, or when Android notifications are not allowed.

os::PROGRESS_NONE

```
const float os::PROGRESS_NONE = -1
```

os::set_progress's value that takes the progress away.

os::PROGRESS_UNKNOWN

```
const float os::PROGRESS_UNKNOWN = -2
```

os::set_progress's value for work whose end isn't known.

os::set_desktop_id

```
fn void os::set_desktop_id(String id)
```

Names the app's desktop file ("org.example.Player" for org.example.Player.desktop), which Linux docks match the progress to; by default it is the executable's name. Keep the string alive while the app runs (a literal is).

os::launcher_uri

```
fn String os::launcher_uri(String id, Allocator allocator)
```

The URI LauncherEntry names an app by: "application://" and the desktop file's name.

SystemTheme

```
struct SystemTheme
```

The appearance settings the OS reports, from os::theme.

Field	Type	Meaning
scheme	ColorScheme	
has_accent	bool	false when the OS has no accent color setting
accent	Oklch	the user's accent color, when has_accent is true
reduced_motion	bool	the user asked for less animation
high_contrast	bool	the user asked for more contrast
always_show_scrollbars	bool	the user turned off scrollbars that fade in on use
wheel_lines	uint	lines per wheel notch; 0 without a setting, uint::max for a page
tooltip_delay	float	seconds the pointer rests before a tooltip shows; 0 without a setting
submenu_delay	float	seconds the pointer rests on a menu row before its submenu opens; 0 without a setting
caret_blink	float	seconds the caret stays on, then off; 0 without a setting, -1 when the user turned blinking off
double_click_time	float	seconds between the clicks of a double-click; 0 without a setting
text_scale	float	the size of text the user chose, 1 for the OS's default; 0 without a setting

Field	Type	Meaning
`no_middle_click_paste`	`bool`	the user turned off pasting the selected text with a middle click (Linux)

`ColorScheme`

```
enum ColorScheme : char
```

Whether the user prefers dark or light interfaces.

Value	Meaning
`NO_PREFERENCE`	the OS has no setting, or the user chose none
`LIGHT`	
`DARK`	

`os::UNSUPPORTED`

```
fault os::UNSUPPORTED
```

The OS, or this machine, has no way to answer the call.

`Pixels`

```
struct Pixels
```

A decoded picture: RGBA8, row by row, sRGB with straight alpha.

Field	Type	Meaning
`width`	`int`	
`height`	`int`	
`rgba`	`char[]`	

`Pixels.free`

```
fn void Pixels.free(&self, Allocator allocator = mem)
```

Frees the pixels, which were allocated in `allocator`.

`os::decode_image`

```
fn Pixels? os::decode_image(char[] bytes, int max_size = 0, Allocator allocator = mem)
```

Decodes a JPEG (baseline or progressive; grayscale, color or CMYK) or a PNG (any bit depth, a palette, transparency, interlaced) to RGBA8. With `max_size`, the picture is halved, each pixel the average of four, until neither side is more than twice `max_size`, so a list of album covers keeps no more pixels than it can show.

`os::decode_image_file`

```
fn Pixels? os::decode_image_file(
    String path,
```

```
    int max_size = 0,
    Allocator allocator = mem)
```

Decodes the JPEG or PNG file at `path`, as `decode_image` does.

os::window_icon

```
fn Pixels? os::window_icon(char[] bytes, Allocator allocator = mem)
```

The window icon `app::run` takes: `bytes`, a PNG or JPEG, decoded to RGBA8 in `allocator`; none when `bytes` is empty, or in the browser, where a page's icon is its own. Bytes that don't decode print a warning on stderr, and the window opens without an icon.

os::MAX_IMAGE_SIDE

```
const int os::MAX_IMAGE_SIDE = 8192
```

The longest side, in pixels, a picture may have to be decoded.

os::NOT_AN_IMAGE

```
fault os::NOT_AN_IMAGE
```

The data isn't a JPEG or PNG, or it is damaged or cut short.

os::IMAGE_TOO_LARGE

```
fault os::IMAGE_TOO_LARGE
```

The picture is more than `MAX_IMAGE_SIDE` pixels on a side; its header says so before anything is allocated.

os::theme_changed

```
fn bool os::theme_changed()
```

Whether the theme settings changed since the last call; cheap and non-blocking, so a `Ui` asks it at the start of every frame while it follows the system. Always false where `os::theme` is unsupported.

os::open_file_dialog

```
fn FileDialog os::open_file_dialog(
    String title = "",
    FileFilter[] filters = {},
    bool multiple = false,
    String folder = "")
```

Opens the system's dialog for choosing files to open. It doesn't block: keep the returned `FileDialog` and ask `FileDialog.result` each frame.

- `filters`: What the dialog offers to show; empty shows every file
- `multiple`: Lets the user choose more than one file
- `folder`: The folder the dialog starts in; empty lets the OS choose

os::save_file_dialog

```
fn FileDialog os::save_file_dialog(
    String title = "",
    String name = "",
    FileFilter[] filters = {},
    String folder = "")
```

Opens the system's dialog for choosing where to save a file; the result is one path. It doesn't block: ask `FileDialog.result` each frame.

- `name`: The file name the dialog suggests

os::choose_folder_dialog

```
fn FileDialog os::choose_folder_dialog(String title = "", String folder = "")
```

Opens the system's dialog for choosing a folder; the result is one path. It doesn't block.

FileDialog

```
struct FileDialog
```

An open file dialog, kept in the app's state and asked each frame with `FileDialog.result`. The zero value is no dialog.

Field	Type	Meaning
id	uint	

FileDialog.result

```
fn String[]? FileDialog.result(&self, Allocator allocator = mem)
```

The chosen paths, once: after it returns them (or reports `os::CANCELLED` or `os::UNSUPPORTED`), the handle is empty and reports `os::NO_DIALOG`. Fails with `os::PENDING` while the dialog is open. The paths are copied into `allocator`; free them with `os::free_paths`.

FileFilter

```
struct FileFilter
```

A filter in a file dialog: a name the user sees and the patterns it matches.

Field	Type	Meaning
name	String	"Images"
patterns	String[]	{ "*.png", "*.jpg" }

os::free_paths

```
fn void os::free_paths(String[] paths, Allocator allocator = mem)
```

Frees paths returned by `FileDialog.result`.

os::save_file

```
fn void? os::save_file(String path, char[] data)
```

Writes `data` to `path`, a path a save dialog returned. On the desktop it is a plain file write. A web page can't write to the user's disk, so there it writes to the file the user picked, in browsers with a save picker, or starts a download named after the path; it doesn't wait for either.

os::PENDING

```
fault os::PENDING
```

The dialog is still open.

os::CANCELLED

```
fault os::CANCELLED
```

The user closed the dialog without choosing.

os::NO_DIALOG

```
fault os::NO_DIALOG
```

The handle has no dialog: never started, or its result was already taken.

FileDialog.is_open

```
fn bool FileDialog.is_open(&self)
```

Whether the dialog is still open: started, and not yet answered.

FileDialog.close

```
fn void FileDialog.close(&self)
```

Lets go of the dialog: an answer that arrives later is dropped, and the handle is empty. Use it before starting another dialog in the same handle, so the first one's place is freed. (A portal dialog already on screen stays there until the user closes it.)

os::notify

```
fn void? os::notify(String title, String body = "")
```

Shows a system notification with `title` and `body`. It waits only for the OS to accept the notification, not for it to appear. Fails with `os::UNSUPPORTED` where the OS can't show one, such as a macOS program outside an app bundle.

os::share

```
fn void? os::share(String text)
```

Shows the OS share sheet with `text`. Android opens the ACTION_SEND chooser; elsewhere it fails with `os::UNSUPPORTED`.

os::open_url

```
fn void? os::open_url(String url)
```

Opens `url` with the user's default handler: a browser for http and https, the mail app for mailto, the file manager or viewer for file. Other URL schemes fail with `os::UNSUPPORTED`, so an app can't be made to launch arbitrary handlers. It doesn't wait for the handler to start.

26 Testing

software::create_ui

```
fn Ui* software::create_ui(
    int width,
    int height,
    float scale = 1,
    Allocator allocator = mem,
    Theme theme = themes::LIGHT,
    FontSet fonts = {},
    Input[] script = {})
```

A Ui with no window: the headless platform (playing `script`, then empty frames) and the software renderer, drawing `width` × `height` points at `scale` pixels per point.

software::attach_renderer

```
fn void software::attach_renderer(Ui* ui, Allocator allocator = mem)
```

Gives `ui` a software renderer when it has none; with one already, does nothing. For a `Ui` made by `ui::create` and fed its own `Input`: each frame draws at that input's size and scale, and `software::draw` and `software::renderer_of` then work as on a `Ui` from `software::create_ui`.

software::new_renderer

```
fn SoftwareRenderer* software::new_renderer(Allocator allocator = mem)
```

A software renderer: draws on the CPU into an RGBA8 framebuffer, the same pixels on every machine.

software::draw

```
fn void software::draw(Ui* ui)
```

Draws the Ui's last frame over its `SURFACE` color; the pixels are then in `software::renderer_of(ui).framebuffer()`.

software::renderer_of

```
fn SoftwareRenderer* software::renderer_of(Ui* ui)
```

The renderer of a Ui made by `software::create_ui`.

software::platform_of

```
fn HeadlessPlatform* software::platform_of(Ui* ui)
```

The headless platform of a Ui made by `software::create_ui`.

SoftwareRenderer

```
struct SoftwareRenderer
```

The `Renderer` that draws into an RGBA8 framebuffer in memory, the same pixels on every machine; `software::create_ui` attaches one and `software::renderer_of` returns it.

`SoftwareRenderer.framebuffer`

```
fn char[] SoftwareRenderer.framebuffer(&self)
```

The framebuffer from the last render(), RGBA8, `width × height`.

`HeadlessPlatform`

```
struct HeadlessPlatform
```

The `Platform` with no window: it plays a script of `Input` frames, then empty ones, 1/60 s apart; `software::create_ui` attaches one. With `can_wake` set, it stands in for a platform that sleeps until a deadline: a waiting poll with one moves the clock straight to it.

`Ui.stats`

```
fn Stats Ui.stats(&self)
```

The last finished frame's numbers.

`Stats`

```
struct Stats
```

What the last finished frame cost: allocations through the Ui's allocator, caches, draw list sizes and where the time went. `Ui.stats` returns it.

Field	Type	Meaning
`allocations`	`usz`	acquires and resizes in the frame
`bytes`	`usz`	bytes they asked for
`frees`	`usz`	releases in the frame
`shaped_texts`	`usz`	entries in the shaped-text cache
`atlas_glyphs`	`usz`	glyphs in the atlas
`atlas_pages`	`usz`	glyph atlas textures
`vertices`	`usz`	in the draw list
`commands`	`usz`	draw commands in the draw list
`elements`	`int`	Clay elements opened
`build_ms`	`float`	from `Ui.begin` to `Ui.end`: the app's widget code
`layout_ms`	`float`	Clay's layout, and reading the rects back
`paint_ms`	`float`	painting the draw list, canvases tessellated
`end_ms`	`float`	the rest of `Ui.end`: scrolling, focus, layers, collecting state
`draw_ms`	`float`	`Ui.draw`: the atlas upload and the renderer (CPU time; the GPU works on after)
`draw_calls`	`uint`	`Ui.draw`: the renderer's draw calls (a GPU renderer's; 0 for the software one)
`gl_calls`	`uint`	and every graphics call it made, draws included