goxpyriment — User Manual

christophe@pallier.org

2026-04-06

Contents

goxpyriment User's Manual

This manual explains the key concepts of the library. It assumes you have read the Getting Started guide and want to understand the framework well enough to write experiments confidently.

Table of Contents

---

1. The Experiment Object

Every experiment revolves around a single `*control.Experiment` value. It is a façade that owns the SDL window, renderer, default font, audio device, keyboard and mouse handlers, and data file. You never create these separately.

```
exp := control.NewExperimentFromFlags("My Experiment", control.Black, control.White, 32
defer exp.End()
```

`NewExperimentFromFlags` parses two optional command-line flags:

Flag	Effect
`-w`	Windowed mode: opens a 1024×768 window instead of going fullscreen
`-d N`	Display ID: open on monitor N (-1 = primary display)
`-s N`	Set subject ID to N (integer); written to the data file automatically

`defer exp.End()` must appear immediately after construction. It saves the data file, releases fonts, destroys the window, and shuts down SDL — even if the experiment panics or returns early.

Key fields

```
exp.Screen       *apparatus.Screen           // window + renderer
exp.Keyboard     *apparatus.Keyboard         // keyboard input
exp.Mouse        *apparatus.Mouse            // mouse input
exp.AudioDevice  sdl.AudioDeviceID    // SDL audio device for Sound/Tone
```

```
exp.Audio        *control.AudioManager // high-level audio helpers
exp.Data         *results.DataFile         // output data file
exp.Design       *design.Experiment   // block/trial structure
exp.SubjectID    int
exp.DefaultFont  *ttf.Font
exp.Info         map[string]string    // values collected by GetParticipantInfo, if use
```

---

2. Collecting Participant Information

Before the experiment window opens, you can display a graphical setup dialog that lets the experimenter fill in participant demographics, monitor characteristics, and display mode. The dialog returns the collected values as a `map[string]string`.

```
fields := append(control.StandardFields, control.FullscreenField)
info, err := control.GetParticipantInfo("My Experiment", fields)
if err != nil {
    log.Fatalf("setup cancelled: %v", err)
}
```

Call `GetParticipantInfo` **before** `NewExperiment` or `NewExperimentFromFlags`. It initializes SDL internally, shows the window, and shuts SDL down again before returning, so the subsequent experiment initialization starts from a clean state.

Pre-built field sets

Variable	Fields included
`control.ParticipantFields`	`subject_id`, `age`, `gender`, `handedness`
`control.MonitorFields`	`screen_width_cm`, `viewing_distance_cm`, `refresh_rate_hz`
`control.StandardFields`	`ParticipantFields` + `MonitorFields` combined
`control.FullscreenField`	Checkbox: `fullscreen` (`"true"` / `"false"`)

Defining custom fields

```
fields := []control.InfoField{
    {Name: "subject_id", Label: "Subject ID",      Default: ""},
    {Name: "session",    Label: "Session (1/2/3)", Default: "1"},
    {Name: "room",       Label: "Testing room",    Default: "Lab A"},
    {Name: "fullscreen", Label: "Fullscreen mode", Default: "true",
     Type: control.FieldCheckbox},
}
info, err := control.GetParticipantInfo("My Experiment", fields)
```

Fields of type `FieldText` (the default) render as text input boxes. Fields of type `FieldCheckbox` render as tick-boxes; their value is always `"true"` or `"false"`.

Dialog interaction

Action	Effect
Click a field	Focus it
Type	Append text to the focused field
Backspace	Delete last character
Tab / Shift-Tab	Move focus to next / previous text field
Enter	Confirm (same as clicking OK)
Escape / close window	Cancel — `ErrCancelled` is returned

Session persistence

All values except `subject_id` are saved to `~/.cache/goxpyriment/last_session.json` when the experimenter confirms. They are pre-filled on the next run. `subject_id` is always reset — the experimenter must enter it fresh each session.

Using the results

```go
info, err := control.GetParticipantInfo("My Experiment", fields)
if err != nil {
    log.Fatalf("setup cancelled: %v", err)
}

// Use the fullscreen checkbox to choose the window mode
fullscreen := info["fullscreen"] == "true"
width, height := 0, 0
if !fullscreen {
    width, height = 1024, 768
}

exp := control.NewExperiment("My Experiment", width, height, fullscreen,
    control.Black, control.White, 32)

// Set Info and SubjectID BEFORE Initialize so they are written to the .csv header
exp.SubjectID, _ = strconv.Atoi(info["subject_id"])
exp.Info = info

if err := exp.Initialize(); err != nil {
    log.Fatalf("failed to initialize: %v", err)
}
defer exp.End()
```

When `exp.Info` is non-nil at the time `Initialize()` is called, the collected key/value pairs are automatically written as a `--PARTICIPANT INFO` block in the `.csv` metadata header — no extra call is required.

> **Note:** When using `GetParticipantInfo` you will typically call the lower-level `NewExperiment` + `Initialize()` instead of `NewExperimentFromFlags`, so you can pass the fullscreen/windowed choice from the dialog directly.

---

3. The Run Loop and Error Handling

The `exp.Run` wrapper

All experiment logic runs inside a callback passed to `exp.Run`:

```
err := exp.Run(func() error {
    // your experiment here
    return control.EndLoop
})
if err != nil && !control.IsEndLoop(err) {
    log.Fatalf("experiment error: %v", err)
}
```

`exp.Run` does two important things:

1. **Ensures everything runs on the SDL main thread.** SDL requires that all rendering and event calls happen on the thread that created the window. `exp.Run` guarantees this.
2. **Installs a panic/recover guard.** When the participant presses ESC, the library immediately aborts the trial loop by calling `panic` with an internal sentinel value. `exp.Run` catches this panic, converts it back to an error, and returns it cleanly. Your data is saved.

Returning from the callback

Return value	Meaning
`control.EndLoop`	Normal exit; experiment is done
`nil`	Continue — call the callback again on the next frame
any other `error`	Propagated as the return value of `exp.Run`

In a typical experiment you run the full trial loop inside a single callback invocation and return `control.EndLoop` at the end. The `nil` / "keep looping" pattern is used for frame-by-frame animation; in most cases you will never return `nil`.

You don't need to check every error

Because pressing ESC triggers the panic/recover mechanism, any call that would have returned an error (e.g., `exp.Show`, `exp.Wait`, `exp.Keyboard.WaitKey`) will instead abort

the loop immediately. The error never reaches your code.

This means the following two styles are both correct:

```
// Style A — check errors explicitly (safer for library/production code)
if err := exp.Show(fixation); err != nil {
    return err
}
if err := exp.Wait(500); err != nil {
    return err
}

// Style B — omit error checks inside exp.Run (fine for experiment scripts)
exp.Show(fixation)
exp.Wait(500)
```

Style B works because if something goes wrong (ESC pressed, window closed), the panic/recover mechanism unwinds the call stack before your code can observe an error. Use Style A if you have cleanup that must run on abort; otherwise Style B keeps experiment scripts readable.

---

4. The Coordinate System

All stimulus positions use a **center-origin** system:

```
            +Y (up)
             |
  -X --------+-------- +X
             |
            -Y (down)
```

- `(0, 0)` is the screen center.
- Positive Y is **up** (opposite to SDL's default, which is top-down).
- Units are pixels at the logical resolution.

```
// Center of screen
stimuli.NewTextLine("Hello", 0, 0, control.White)

// 200 px to the right
stimuli.NewCircle(30, control.Red).SetPosition(control.Point(200, 0))

// Bottom-left area
stimuli.NewTextLine("Score", -400, -250, control.White)
```

The conversion to SDL coordinates (top-left origin) is handled internally by `screen.CenterToSDL(x, y)`. You never call this yourself unless you are writing a custom stimulus type.

Logical size

If you call `exp.SetLogicalSize(width, height)`, the coordinate system scales to that virtual resolution regardless of the actual window or screen size. This is useful for making experiments resolution-independent:

```
if err := exp.SetLogicalSize(1920, 1080); err != nil {
    log.Printf("warning: %v", err)
}
// Now (960, 540) is the center-right area, regardless of actual screen size.
```

Display scaling (HiDPI / fractional scaling)

> **Recommendation: set your OS display scaling to 100% before running experiments.**

Most desktop environments (GNOME, KDE, Windows, macOS) allow the user to scale the entire UI — e.g. 125%, 133%, 150% — to make text and icons larger on high-density screens. Goxpyriment does not currently compensate for fractional OS-level scaling. Running with a scale factor other than 100% can cause:

- stimulus positions and sizes to be wrong (coordinates are in logical pixels, but the OS maps them to physical pixels at the scale factor),
- video playback and canvas rendering to appear cropped or misaligned,
- the window to not fill the screen correctly in fullscreen mode.

To disable scaling before running an experiment:

- **GNOME (Ubuntu):** *Settings → Displays → Scale → 100%*
- **KDE:** *System Settings → Display and Monitor → Scale → 100%*
- **Windows:** *Settings → Display → Scale → 100%*
- **macOS:** *System Settings → Displays → Resolution → Default* (or choose a non-HiDPI mode)

You can restore your preferred scaling after the session.

---

5. The Rendering Model

SDL uses a **double-buffered** rendering model. There is an off-screen backbuffer where you draw, and the visible display. You draw to the backbuffer; calling `screen.Update()` (also called a "flip") presents it to the screen, typically synchronized to the vertical retrace (VSYNC).

The three-step cycle for showing one stimulus is:

```
exp.Screen.Clear()       // fill backbuffer with background color
myStim.Draw(exp.Screen)  // draw stimulus onto backbuffer
exp.Screen.Update()      // present to display (VSYNC-blocks)
```

`exp.Show(stim)` does all three in one call and is the right choice for single-stimulus presentations. For cases where you need to draw multiple stimuli simultaneously —

so they appear on screen at the same time — call each `Draw` separately before the single `Update`:

```
// Show fixation cross and stimulus simultaneously
exp.Screen.Clear()
fixation.Draw(exp.Screen)
targetCircle.Draw(exp.Screen)
exp.Screen.Update()
```

The blank screen

`exp.Blank(ms)` clears the screen, flips it (showing a blank), and then waits:

```
exp.Blank(1000) // 1-second blank inter-trial interval
```

This is equivalent to `exp.Screen.Clear() + exp.Screen.Update() + exp.Wait(ms)`.

Never draw outside exp.Run

All rendering must happen inside the `exp.Run` callback — equivalently, on the SDL main thread. Drawing from a goroutine will silently do nothing or crash.

---

6. Timing Architecture

Frame cadence

`screen.Update()` blocks until the display's vertical retrace (VSYNC). On a 60 Hz monitor this is ~16.67 ms; on a 120 Hz monitor ~8.33 ms. This is the fundamental clock of the visual display: every stimulus change is aligned to a frame boundary automatically.

To know your frame duration at runtime:

```
frameDur := exp.Screen.FrameDuration()  // e.g., 16.666ms on a 60 Hz display
fmt.Printf("Frame: %.2f ms\n", frameDur.Seconds()*1000)
```

`exp.Wait` vs `clock.Wait`

Function	Pumps SDL events?	Detects ESC?	Use when
`exp.Wait(ms)`	Yes	Yes	Inside `exp.Run`, between stimuli
`clock.Wait(ms)`	No	No	Short busy-waits, inside animation loops

Always use `exp.Wait` for inter-trial and inter-stimulus intervals — it keeps the OS responsive and responds to ESC. Use `clock.Wait` only inside VSYNC-locked loops that handle events themselves (streams, animation loops).

Sub-millisecond timing is not guaranteed for `exp.Wait`

`exp.Wait` sleeps in 1 ms increments. For coarse delays (ISIs, fixation durations) this is perfectly fine. For frame-accurate stimulus timing — e.g., showing a stimulus for exactly 2 frames — use the stream functions described in Section 10.

Nanosecond event timestamps

SDL3 timestamps every input event **at hardware-interrupt time**, before any application code runs. The timestamp is stored with the event in the SDL event queue, where it remains untouched until your code reads it.

This has a crucial consequence: **it does not matter when you call `GetKeyEventTS`**. Any keypress that occurred — even during `exp.Wait`, between two `ShowTS` calls, or while other computation was running — is waiting in the queue with its exact hardware timestamp. Nothing is lost. This is fundamentally different from a polling-based approach, where a keypress can only be measured from the moment the polling function is called, and any code running before that call inflates the apparent RT.

`exp.ShowTS(stim)` records the SDL nanosecond time of the VSYNC flip. `GetKeyEventTS` retrieves the event's own hardware timestamp. Subtracting the two gives hardware-precision RT regardless of how much code ran in between:

```
onset, _ := exp.ShowTS(stim1)   // record flip timestamp
exp.Wait(500)                    // keypress here is NOT lost
exp.ShowTS(stim2)                // keypress here is also NOT lost
key, keyTS, _ := exp.Keyboard.GetKeyEventTS(responseKeys, -1)
rtToStim1 := int64(keyTS - onset)   // correct even if pressed during Wait
```

This is especially useful when you need RT relative to a specific stimulus in a multi-stimulus sequence — something that `WaitKeysRT` cannot express directly, since it measures from the call site rather than from a recorded onset.

Disabling garbage collection

Go's garbage collector can pause execution for several milliseconds at unpredictable times. The stream and animation functions disable it during the critical loop:

```
old := debug.SetGCPercent(-1)
defer debug.SetGCPercent(old)
```

You do not need to do this yourself for ordinary trial loops; only for high-speed RSVP or animation. The stream and animation functions handle it automatically.

Choosing a timing approach

Use this decision guide before committing to a timing strategy.

Situation	Recommended approach
Coarse ISIs and fixation durations ($\geq$ 100 ms)	`exp.Wait(ms)` — adequate precision, keeps the event loop alive
Stimulus duration measured in frames (e.g. 2-frame mask)	`exp.Show` inside a VSYNC-locked loop, or `PresentStreamOfImages` (Section 10)
Single-stimulus trial, RT relative to onset	`exp.Show(stim)` + `exp.Keyboard.WaitKeysRT(...)` — millisecond RT from call site
Multi-stimulus sequence, RT relative to a specific stimulus	`exp.ShowTS(stim)` + `exp.Keyboard.GetKeyEventTS(...)` — nanosecond timestamps on the same SDL clock; subtract directly
RSVP or rapid animation (frame-accurate presentation of many items)	`PresentStreamOfImages` / `PresentStreamOfTexts` (Section 10) — GC disabled, VSYNC-locked, full timing log
EEG/MEG synchronisation required	Add a TTL pulse via `triggers` immediately after `exp.ShowTS` (Section 15)
Stimulus duration not a multiple of the frame period (e.g. 10 ms, 17 ms)	Variable Refresh Rate mode — see below

OS considerations. Presentation latency depends heavily on whether the display compositor is active — see the next section. For EEG work, always verify onset timing with a photodiode regardless of OS.

When in doubt, use `ShowTS` + `GetKeyEventTS`. The overhead over `Show` + `WaitKeysRT` is negligible, and you get hardware-precision timestamps at no cost.

`ShowTS` vs `FlipTS`. `exp.ShowTS(stim)` is the standard high-level call: it clears the screen, draws the stimulus, flips, and returns the flip timestamp — one line for the common case. `exp.Screen.FlipTS()` is the lower-level primitive that only flips and timestamps, leaving clearing and drawing to you. Use it directly when you need to compose several stimuli with multiple `Draw` calls before a single flip, or when working inside a VRR or VSYNC-locked loop where you manage the render cycle yourself (see the VRR section below).

Display compositor bypass

A compositing window manager blends every application frame with other on-screen elements before sending the result to the display. This adds latency and jitter that can be significant for millisecond-accurate stimulus presentation. How much overhead you incur — and whether you can avoid it — depends on the operating system.

Linux / X11 (fullscreen). SDL3 automatically sets the `_NET_WM_BYPASS_COMPOSITOR` window property when the application enters fullscreen mode. Compliant compositing WMs (KWin, Mutter, Compton) respond by *unredirecting* the fullscreen window: the application's framebuffer is routed directly to the display scan-out pipeline without

compositing. Presentation latency drops below 1 ms — comparable to a compositor-free session. Nothing special needs to be done in goxpyriment; running fullscreen (omit `-w`) is sufficient.

Linux / Wayland (fullscreen). The Wayland compositor is always the intermediary for frame delivery, but modern compositors (KWin 5.21+, GNOME Mutter 44+) support *direct scan-out* for fullscreen applications, which routes the framebuffer to the display hardware with near-zero overhead. In practice the latency is similar to X11 bypass, though this depends on the compositor version and GPU driver.

Linux / KMS–DRM (no display server). The lowest-latency configuration on Linux is to run without any display server at all, from a virtual terminal (`Ctrl+Alt+F2`). Setting the environment variable `SDL_VIDEO_DRIVER=kmsdrm` before launching the experiment directs SDL3 to use the Linux kernel's KMS/DRM subsystem directly, bypassing both X11 and Wayland entirely:

```
SDL_VIDEO_DRIVER=kmsdrm go run myexperiment/main.go
```

This is the recommended configuration for the most demanding timing requirements, such as single-frame subliminal stimulation or high-frequency RSVP.

Windows (fullscreen). SDL3's fullscreen mode creates an exclusive fullscreen surface that bypasses the Desktop Window Manager (DWM), yielding frame-interval jitter typically below 0.3 ms (one standard deviation), comparable to Linux.

macOS. Apple's Metal pipeline always delivers frames through the WindowServer compositor, and no third-party application can bypass it — even in fullscreen. The WindowServer typically adds one frame of fixed latency (8–17 ms depending on refresh rate) and 2–5 ms of frame-interval jitter. This onset uncertainty — larger than many SOAs of experimental interest — makes macOS unsuitable for production data collection. Researchers should prefer Linux or Windows hardware for laboratory deployments; macOS is adequate for pilot testing and paradigm development.

These platform differences can be measured empirically with the `display` and `frames` sub-tests of the bundled `Timing-Tests` suite (see TimingTests.md).

Variable Refresh Rate (VRR) — arbitrary stimulus durations

On a standard 60 Hz monitor, every stimulus duration must be a multiple of 16.67 ms. You cannot present a stimulus for 10 ms or 20 ms — the display simply cannot change in between frame boundaries. Variable Refresh Rate technology (AMD FreeSync, NVIDIA G-Sync, VESA Adaptive-Sync) removes this constraint: the panel holds each frame for exactly as long as the software asks, refreshing only when the next `Present()` call arrives.

Enabling VRR in goxpyriment goxpyriment opens the renderer with VSYNC enabled by default, which locks every flip to a frame boundary. To enter VRR mode, disable VSYNC before the timing-critical section and restore it afterwards:

```
// Switch to VRR mode — Present() now returns immediately.
if err := exp.Screen.SetVSync(0); err != nil {
```

```
    log.Fatalf("VRR not supported: %v", err)
}
defer exp.Screen.SetVSync(1) // always restore

// Pre-load stimulus textures before disabling GC.
stimuli.PreloadVisualOnScreen(exp.Screen, myStim)

old := debug.SetGCPercent(-1)
defer debug.SetGCPercent(old)

// Draw stimulus and present (returns immediately with VSync=0).
exp.Screen.Renderer.SetDrawColor(0, 0, 0, 255)
exp.Screen.Clear()
myStim.Draw(exp.Screen)
onsetNS, _ := exp.Screen.FlipTS()

// Hold for exactly targetDur using a busy-wait (sub-ms precision).
deadline := time.Now().Add(targetDur)
for time.Now().Before(deadline) { /* spin */ }

// Present blank — the display switches immediately on VRR.
exp.Screen.Renderer.SetDrawColor(0, 0, 0, 255)
exp.Screen.Clear()
offsetNS, _ := exp.Screen.FlipTS()

actualDurMs := float64(offsetNS-onsetNS) / 1e6
```

`onsetNS` and `offsetNS` are both `sdl.TicksNS()` timestamps on the SDL nanosecond clock, captured right after each `Present()` returns. Their difference measures the software-controlled stimulus duration. On a VRR display this equals the actual on-screen duration plus a small, constant hardware pipeline latency (GPU scan-out + panel response, typically 1–5 ms) that can be measured independently with the `frames` test and a photodiode.

VRR range and self-diagnosis Every VRR panel has a supported refresh-rate range (e.g. 48–144 Hz on a typical gaming monitor). Requesting a duration shorter than `1000/max_Hz` ms or longer than `1000/min_Hz` ms takes the panel outside its VRR window; the display then falls back to fixed-rate behaviour and durations are again quantised to frame multiples.

You can characterise your panel's VRR window empirically with the `Timing-Tests` suite:

```
go run tests/Timing-Tests/main.go -test vrr -vrr-max-ms 50 -cycles 5 -w
```

This sweeps target durations from 1 ms to 50 ms in 1 ms steps and reports the actual vs. target duration at each step. Duration errors below 0.5 ms across the sweep confirm that VRR is working; large periodic errors confirm that VRR is absent or the

requested duration is outside the supported range. See TimingTests.md — VRR section for detailed interpretation, including how to read the VRR boundary from the output and how to enable FreeSync on Linux.

Requirements

- A VRR-capable monitor (FreeSync, G-Sync Compatible, or Adaptive-Sync).
- VRR enabled in the OS display settings (Linux: DRM/KMS or compositor; Windows: Display Settings → Advanced display → Variable refresh rate).
- The application does not need any special SDL hints; `SetVSync(0)` is sufficient.

Note on tearing On a monitor without VRR, `SetVSync(0)` disables the frame-boundary synchronisation entirely and the GPU writes to the framebuffer while the display is reading it, producing a horizontal tear line. On a VRR monitor this does not happen: the display waits for the GPU's signal before starting each new refresh. If your VRR test shows tearing artefacts, VRR is either not enabled in your OS or not supported by your hardware.

---

7. Input Handling

Keyboard

The `exp.Keyboard` object provides blocking and non-blocking access:

```
// Block until any key — returns the keycode
key, err := exp.Keyboard.Wait()

// Block until a specific key
err := exp.Keyboard.WaitKey(control.K_SPACE)

// Block until one of several keys, with optional timeout (-1 = no timeout)
// Returns 0 on timeout, sdl.EndLoop on ESC/quit
key, err := exp.Keyboard.WaitKeys([]control.Keycode{control.K_F, control.K_J}, 3000)

// Same, but also returns reaction time in milliseconds from the call site
key, rt, err := exp.Keyboard.WaitKeysRT([]control.Keycode{control.K_F, control.K_J}, 30

// Hardware-precision version: returns the SDL event's nanosecond timestamp
key, eventTS, err := exp.Keyboard.GetKeyEventTS([]control.Keycode{control.K_F, control.

// Non-blocking poll — returns 0 if nothing pressed
key, err := exp.Keyboard.Check()

// Query whether a key is physically held down right now (no event queue involvement)
held := exp.Keyboard.IsPressed(control.K_SPACE)

// Wait for KEY_UP on a specific key; returns its SDL hardware timestamp (nanoseconds)
```

```
upTS, err := exp.Keyboard.WaitKeyReleaseTS(key, -1)

// Drain all pending key events (use before a new trial to discard stale presses)
exp.Keyboard.Clear()
```

`GetKeyEventTS` "get" semantics. Unlike the `Wait*` functions, `GetKeyEventTS` reads from the SDL event queue and returns *immediately* if a matching event is already there — it only blocks when the queue has no matching event. This means a keypress that happened during `exp.Wait` or any other intervening code is consumed instantly on the next `GetKeyEventTS` call, with its original hardware timestamp intact.

When to call `Clear()`. Despite living on `exp.Keyboard`, `Clear()` drains the **entire** SDL event queue — keyboard, mouse, gamepad, and everything else — because SDL uses a single shared queue. Call it at the start of a trial (before `ShowTS`) to discard stale events from any device. Do **not** call it after `ShowTS`: the participant may have already responded, and `Clear()` would silently discard that event before `GetKeyEventTS` or `GetPressEventTS` ever sees it.

`WaitKeysRT` vs `GetKeyEventTS`:

`WaitKeysRT` measures elapsed time from the moment the call is made (using `sdl.Ticks()`, millisecond granularity). In a single-stimulus trial this is a good approximation of RT from stimulus onset, but it accumulates any delay between `Show()` returning and `WaitKeysRT` being called.

`GetKeyEventTS` returns the SDL3 event's own `Timestamp` field — the nanosecond time at which the hardware key-down event was generated. Combined with `exp.ShowTS(stim)`, which records the nanosecond time of the VSYNC flip, reaction time is simply:

```
onset, _ := exp.ShowTS(target)   // nanoseconds at VSYNC flip
key, eventTS, _ := exp.Keyboard.GetKeyEventTS(responseKeys, 3000)
rtNS := int64(eventTS - onset)   // nanoseconds; divide by 1e6 for ms
```

This form is also the only correct approach when multiple stimuli are presented in sequence and you need RT relative to a specific one:

```
onset1, _ := exp.ShowTS(prime)   // prime appears
exp.Wait(500)
exp.ShowTS(target)               // target appears 500 ms later
key, eventTS, _ := exp.Keyboard.GetKeyEventTS(responseKeys, 3000)
rtToPrime := int64(eventTS - onset1)  // RT from prime onset
```

Collecting multiple simultaneous responses — `GetKeyEventsTS`:

In rare cases you may want to capture *all* key events, not just the first. `GetKeyEventsTS` is designed for detecting bilateral responses (e.g. both hands pressing at once) and works in two phases:

1. **Phase 1** — blocks until the first matching key arrives, respecting the timeout exactly like `GetKeyEventTS`.
2. **Phase 2** — waits up to 50 ms for any additional matching keys.

The second phase is necessary because human "simultaneous" presses are rarely truly simultaneous — the two KEY_DOWN events typically arrive 10–50 ms apart. Without this window, a non-blocking drain after the first key would miss the second key almost every time.

```
events, err := exp.Keyboard.GetKeyEventsTS(responseKeys, 3000)
if len(events) > 0 {
    firstKey := events[0].Key
    firstTS  := events[0].TimestampNS
    rtNS     := int64(firstTS - onset)
}
if len(events) == 2 {
    lagNS := int64(events[1].TimestampNS - events[0].TimestampNS)
}
```

In the common single-response case only one event is returned, at the cost of at most 50 ms of extra latency before the function returns. For the typical single-response trial, `GetKeyEventTS` avoids that overhead entirely.

Recording all presses over a fixed duration — `CollectKeyEventsTS`:

When the goal is to record *everything* the participant presses within a known time window (finger-tapping, free-response periods, stimulus-stream logging), use `CollectKeyEventsTS`. Unlike `GetKeyEventsTS`, it always runs for the full duration — it does not return early on the first keypress:

```
// Record every tap of F or J over 5 seconds
exp.Keyboard.Clear()
onset, _ := exp.ShowTS(stim)
events, err := exp.Keyboard.CollectKeyEventsTS(
    []control.Keycode{control.K_F, control.K_J}, 5000)
for _, ev := range events {
    rtNS := int64(ev.TimestampNS - onset)
    fmt.Printf("key %s at %d ms\n", ev.Key.KeyName(), rtNS/1_000_000)
}
```

Pass `keys = nil` to capture any key. Pass `durationMS = 0` to do a non-blocking drain of whatever is already in the queue. Returns an empty (non-nil) slice if nothing was pressed.

Querying whether a key is currently held — `IsPressed`:

`IsPressed` queries SDL's internal scancode state array and returns `true` if the key is physically depressed at the instant of the call, regardless of the event queue. It is useful in animation loops that need to react continuously to a held key, or to verify that the participant has released a key before starting a new trial:

```
for exp.Keyboard.IsPressed(control.K_SPACE) {
    // do something while SPACE is held
    time.Sleep(10 * time.Millisecond)
}
```

Because `IsPressed` does not consume queue events, it can be called freely alongside `GetKeyEventTS` without interfering with the event stream.

Measuring keypress duration — `WaitKeyReleaseTS`:

`WaitKeyReleaseTS` blocks until a KEY_UP event arrives for the specified key and returns its SDL3 hardware timestamp in nanoseconds. Together with the KEY_DOWN timestamp from `GetKeyEventTS`, this gives nanosecond-precision keypress duration:

```
key, downTS, _ := exp.Keyboard.GetKeyEventTS(responseKeys, -1)
// ... optionally update display while key is held ...
upTS, _ := exp.Keyboard.WaitKeyReleaseTS(key, 5000)
durationNS := upTS - downTS           // nanoseconds
durationMS := durationNS / 1_000_000  // milliseconds
```

This is the recommended approach for paradigms where press duration is a dependent variable (e.g., force-choice hold-to-respond, finger-tapping, or hold-triggered responses).

Mouse

```
// Block until any button
btn, err := exp.Mouse.WaitPress()

// RT in milliseconds from call site (mirrors Keyboard.WaitKeysRT)
btn, rt, err := exp.Mouse.WaitPressRT(3000)

// Hardware-precision version: returns the SDL event's nanosecond timestamp
btn, eventTS, err := exp.Mouse.GetPressEventTS(3000)

// Non-blocking poll
btn, err := exp.Mouse.Check()

// Query whether a button is physically held down right now
held := exp.Mouse.IsPressed(sdl.BUTTON_LEFT)

// Wait for MOUSE_BUTTON_UP; returns its SDL hardware timestamp (nanoseconds)
upTS, err := exp.Mouse.WaitButtonReleaseTS(btn, 5000)

// Current cursor position in center-based coordinates
x, y := exp.Mouse.Position()

// Show / hide cursor
exp.Mouse.ShowCursor(false)
```

Button values: `sdl.BUTTON_LEFT`, `sdl.BUTTON_MIDDLE`, `sdl.BUTTON_RIGHT`, `sdl.BUTTON_X1`, `sdl.BUTTON_X2`.

`IsPressed` and `WaitButtonReleaseTS` mirror the keyboard's `IsPressed` and `WaitKeyReleaseTS` and can be used to measure click-hold duration:

```
btn, downTS, _ := exp.Mouse.GetPressEventTS(-1)
upTS, _        := exp.Mouse.WaitButtonReleaseTS(btn, 5000)
durationMS     := int64(upTS-downTS) / 1_000_000
```

Multi-device input — `WaitAnyEventTS`

If you want to accept a response from *either* the keyboard or the mouse (or both), use `exp.WaitAnyEventTS` instead of calling `Keyboard` and `Mouse` separately:

```
onset, _ := exp.ShowTS(stim)

// Accept F or J key, or any mouse button click, up to 3 s
ev, err := exp.WaitAnyEventTS(
    []control.Keycode{control.K_F, control.K_J},
    true,   // catchMouse
    3000,
)

rtNS := int64(ev.TimestampNS - onset)  // hardware-precision RT, nanoseconds
rtMs := rtNS / 1_000_000

switch ev.Device {
case apparatus.DeviceKeyboard:
    fmt.Printf("key %d, RT %d ms\n", ev.Key, rtMs)
case apparatus.DeviceMouse:
    fmt.Printf("mouse button %d, RT %d ms\n", ev.Button, rtMs)
}
```

Pass `keys = nil` to accept any key. Pass `catchMouse = false` to ignore the mouse. On timeout, returns a zero `InputEvent` with `nil` error.

Key codes

Key codes are re-exported in `control` for the most common experiment keys. For anything not listed there, use `go-sdl3/sdl` directly:

```
import "github.com/Zyko0/go-sdl3/sdl"

key, _ := exp.Keyboard.Wait()
if key == sdl.K_A { ... }
```

Response Devices

All of the input methods above are device-specific: `GetKeyEventTS` for keyboards, `GetPressEventTS` for the mouse, and so on. If the experiment should run equally well with different input hardware — a keyboard in one lab, a response box in another — device-specific calls scatter conditional logic throughout the trial loop.

`ResponseDevice` is a single interface that abstracts over all of them:

```go
// in package io
type ResponseDevice interface {
    WaitResponse(ctx context.Context) (Response, error)
    DrainResponses(ctx context.Context) error
    Close() error
}
```

`WaitResponse` blocks until a response is detected and returns a `Response`:

```go
type Response struct {
    Source  apparatus.DeviceKind  // which device fired
    Code    uint32         // key code, button index, or TTL bitmask
    RT      time.Duration  // elapsed from WaitResponse call
    Precise bool           // timing quality (see below)
}
```

Wrap any input device with the matching adapter:

```go
// Keyboard (SDL hardware timestamps → Precise: true)
var rd apparatus.ResponseDevice = &apparatus.KeyboardResponseDevice{KB: exp.Keyboard}

// Mouse (SDL hardware timestamps → Precise: true)
var rd apparatus.ResponseDevice = &apparatus.MouseResponseDevice{M: exp.Mouse}

// TTL response box, e.g. MEGTTLBox (software poll → Precise: false)
box, _ := triggers.NewMEGTTLBox("/dev/ttyACM0")
var rd apparatus.ResponseDevice = apparatus.NewTTLResponseDevice(box, 5*time.Millisecon
```

All three look identical inside a trial loop:

```go
onset, _ := exp.ShowTS(stim)
_ = rd.DrainResponses(ctx)         // clear stale presses from previous trial
resp, err := rd.WaitResponse(ctx)
if err != nil { /* ESC, timeout, etc. */ }

rtMs := resp.RT.Milliseconds()     // always valid
```

Timing precision and the `Precise` flag The `Precise` field tells you whether `RT` came from a nanosecond hardware event timestamp or a software poll:

Device	`Precise`	RT accuracy
Keyboard, Mouse, Gamepad	`true`	SDL3 hardware event timestamp, nanosecond resolution — identical to using `GetKeyEventTS` directly
MEGTTLBox, DLPIO8	`false`	`time.Now()` at poll detection; accuracy bounded by poll interval (default 5 ms)

When `Precise` is `true`, the RT in the `Response` is computed from `sdl.TicksNS()` captured at the `WaitResponse` call versus the SDL3 hardware event timestamp — the same nanosecond clock used by `Screen.FlipTS`. It is therefore suitable for stimulus-onset-locked RT:

```
onset, _ := exp.ShowTS(stim)
resp, _ := rd.WaitResponse(ctx)
if resp.Precise {
    // resp.RT was computed from SDL hardware timestamps: nanosecond accuracy
    rtFromOnset := time.Duration(int64(onset) + resp.RT.Nanoseconds())  // approximate
}
```

When `Precise` is `false`, `RT` is still a valid elapsed duration — it just has ~poll-interval jitter (5 ms by default). For typical behavioral RT tasks this is acceptable; for EEG/MEG experiments requiring sub-millisecond synchrony, use the TTL output path for stimulus marking and treat RT as an approximate measure or use a hardware response box with sub-ms timestamping.

---

8. Data Collection

The `.csv` file

`exp.Data` writes to a file named `<expname>_<subjectID>_<timestamp>.csv` in `~/goxpy_data/`. The format is plain CSV with `#`-prefixed comment lines as a metadata header. It opens automatically on `Initialize()` and is flushed to disk when `exp.End()` is called (or when `exp.Data.Save()` is called explicitly).

Declaring column names

Call this once, before the trial loop:

```
exp.AddDataVariableNames([]string{"condition", "response", "rt_ms", "correct"})
```

This writes a `# VARIABLES` header line. **Subject ID is always prepended automatically** — do not include it in the name list.

Adding rows

```
exp.Data.Add(condition, response, rt, correct)
```

Fields are written in the same order as the variable names. Any type that has a meaningful `fmt.Sprint` representation works: `int`, `float64`, `bool`, `string`. The subject ID and a timestamp are prepended automatically. Fields containing commas or quotes are escaped.

Example output

```
# EXPERIMENT: My Experiment
```

```
# SUBJECT: 3
# DATE: 2026-03-22T14:05:11
# VARIABLES: subject_id,condition,response,rt_ms,correct
3,congruent,F,412,true
3,incongruent,J,538,false
```

Saving mid-experiment

For long experiments it is good practice to call `exp.Data.Save()` after each block. This flushes the buffer to disk so that data up to that point is not lost if the experiment crashes.

---

9. Stimuli: Lifecycle and Preloading

GPU textures are lazily allocated

Creating a stimulus (e.g., `stimuli.NewTextLine(...)`) is cheap — it allocates a Go struct and stores the parameters, but does no GPU work. The SDL texture is created on the **first call to `Draw`**. This means:

- You can safely create all your stimuli before the run loop.
- The first presentation of each stimulus will be slightly slower due to texture upload.

For timing-sensitive presentations, preload explicitly:

```
stimuli.PreloadVisualOnScreen(exp.Screen, myStim)
// or, for a slice:
stimuli.PreloadAllVisual(exp.Screen, []stimuli.VisualStimulus{stim1, stim2, stim3})
```

Call this after `exp.Run` starts (the renderer is ready), but before the critical section:

```
err := exp.Run(func() error {
    // Preload everything while showing instructions
    exp.ShowInstructions("Loading, please wait...")
    stimuli.PreloadAllVisual(exp.Screen, stimSlice)

    // Now timing-sensitive trials
    for _, stim := range stimSlice { ... }
    return control.EndLoop
})
```

Releasing textures

If you generate many stimuli dynamically (e.g., hundreds of unique text strings), call `stim.Unload()` after each trial to free the GPU memory:

```
for _, word := range wordList {
    stim := stimuli.NewTextLine(word, 0, 0, control.White)
```

```
    exp.Show(stim)
    exp.Keyboard.Wait()
    stim.Unload()  // release GPU texture
}
```

For a fixed, small set of stimuli created once and reused throughout the experiment, you never need to call `Unload` — `exp.End()` handles cleanup.

Writing a custom stimulus

Any type that implements `VisualStimulus` can be passed to `exp.Show`:

```
type VisualStimulus interface {
    Present(screen *apparatus.Screen, clear, update bool) error
    Preload() error
    Unload() error
    Draw(screen *apparatus.Screen) error
    GetPosition() sdl.FPoint
    SetPosition(pos sdl.FPoint)
}
```

Embed `stimuli.BaseVisual` to get the position methods and no-op `Preload`/`Unload` for free, then implement only `Draw` and `Present`:

```
type MyStimulus struct {
    stimuli.BaseVisual
    // your fields
}

func (m *MyStimulus) Draw(screen *apparatus.Screen) error {
    // draw using screen.Renderer SDL calls
    return nil
}

func (m *MyStimulus) Present(screen *apparatus.Screen, clear, update bool) error {
    return stimuli.PresentDrawable(m, screen, clear, update)
}
```

Interactive widgets

Two stimuli present a UI and collect a structured response from the participant in a blocking loop. Neither is timing-critical, so they use `sdl.WaitEvent` rather than VSYNC polling.

`Menu` — a numbered, keyboard-navigable list:

```
m := stimuli.NewMenu([]string{"Beginner", "Expert", "Practice run"})
// Optional customisation:
m.HighlightColor = sdl.Color{R: 255, G: 220, B: 0, A: 255}
```

```
idx, err := m.Get(exp.Screen, exp.Keyboard, 0)
// idx is 0-based; -1 + sdl.EndLoop on ESC/quit
```

The selected item is shown with a > prefix in `HighlightColor`; all others use `TextColor`. Navigation: UP/DOWN arrows move the highlight; ENTER or SPACE confirms; digit keys 1–9 (0 for tenth) select directly without requiring confirmation.

`ChoiceGrid` — a grid of labelled buttons, activated by mouse click or matching key:

```
cg := stimuli.NewChoiceGrid([]string{"B","C","D","F","G"}, 3, "Pick 3 letters:")
selections, err := cg.Get(exp.Screen, exp.Keyboard)
// selections is []string in press order
```

---

10. High-Precision Streams (RSVP)

For paradigms that present stimuli at high speed — RSVP, attentional blink, priming — the standard `exp.Show` / `exp.Wait` cycle is not precise enough. The stream functions provide frame-accurate presentation:

- GC is **disabled** for the duration of the stream.
- Every onset and offset is aligned to a **VSYNC boundary**.
- A `TimingLog` records the predicted and actual onset for each stimulus.
- All keyboard and mouse events that occur during the stream are captured.

Text stream

The simplest entry point:

```
words := []string{"CHAIR", "RIVER", "TIGER", "CLOCK", "STONE"}
on  := 150 * time.Millisecond
off :=  50 * time.Millisecond

events, logs, err := stimuli.PresentStreamOfText(
    exp.Screen, words, on, off,
    0, 0,          // center of screen
    control.White,
)
```

Image / mixed stimulus stream

For images or any `VisualStimulus`:

```
stims := []stimuli.VisualStimulus{pic1, fixation, pic2, fixation}

// Regular cadence (all stimuli share the same on/off duration)
elements := stimuli.MakeRegularVisualStream(stims, 100*time.Millisecond, 0)

// Irregular cadence (individual onset times and durations, in ms)
```

```
elements, err := stimuli.MakeVisualStream(stims, onsetMs, durationMs)

events, logs, err := stimuli.PresentStreamOfImages(exp.Screen, elements, 0, 0)
```

Reading events from the stream

Each `UserEvent` carries two timestamps:

- `Timestamp` — a `time.Duration` relative to stream start, Go monotonic clock, millisecond precision. Useful for quick inspection.
- `TimestampNS` — the SDL3 hardware event timestamp in nanoseconds, same clock as `Screen.FlipTS`. Use this for sub-millisecond RT computation.

```
for _, ev := range events {
    if ev.Event.Type == sdl.EVENT_KEY_DOWN {
        key := ev.Event.KeyboardEvent().Key
        t   := ev.Timestamp.Milliseconds()  // ms from stream start (coarse)
        fmt.Printf("key %d pressed at %d ms\n", key, t)
    }
}
```

Use `stimuli.FirstKeyPress` to find the first matching key-down event without writing a loop:

```
if ev, ok := stimuli.FirstKeyPress(events, sdl.K_SPACE); ok {
    fmt.Printf("Space pressed at %d ms\n", ev.Timestamp.Milliseconds())
}
```

Both `ev.Timestamp` (Go clock, ms precision) and `ev.TimestampNS` (SDL3 hardware, nanoseconds) are available on the returned `UserEvent`.

Computing RT from stream events

Because `UserEvent.TimestampNS` and `TimingLog.OnsetNS` are on the same SDL nanosecond clock, reaction time from a specific stimulus is exact:

```
for _, ev := range events {
    if ev.Event.Type == sdl.EVENT_KEY_DOWN {
        for _, l := range logs {
            if ev.TimestampNS >= l.OnsetNS && (l.OffsetNS == 0 || ev.TimestampNS < l.Of
                rtNS := int64(ev.TimestampNS - l.OnsetNS)
                fmt.Printf("RT from stimulus %d: %d ms\n", l.Index, rtNS/1_000_000)
            }
        }
    }
}
```

Reading the timing log

```
for i, l := range logs {
    jitter := (l.ActualOnset - l.TargetOn).Milliseconds()
    fmt.Printf("stimulus %d: target %d ms, actual %d ms, jitter %d ms\n",
        i, l.TargetOn.Milliseconds(), l.ActualOnset.Milliseconds(), jitter)
}
```

`OnsetNS` and `OffsetNS` give the SDL3 nanosecond timestamps of the actual VSYNC flips that turned each stimulus on and off. These are zero for stimuli that had zero frames in their on or off period.

Jitter below ±1 frame (±8–17 ms depending on monitor) is normal and expected. Larger jitter indicates system load or GPU driver issues.

Platform timing notes

`PresentStreamOfImages` provides the best timing accuracy that the operating system and display driver allow. The characteristics differ by platform:

Linux

- Without a compositing window manager (e.g. a plain X11 session with no compositor), SDL3's `Present()` blocks until the next VSYNC boundary. Onset jitter is typically **< 1 ms**.
- With a Wayland compositor or an X11 compositing WM (KWin, Mutter, Picom), the compositor controls buffer swaps. Onset jitter is typically **1–3 ms**; the compositor may add one frame (~17 ms at 60 Hz) of fixed latency.
- For the most reliable timing on Linux, disable the compositor or use a plain X11 session.

macOS (Metal)

- The macOS WindowServer compositor is **always active** — exclusive fullscreen does not bypass it. SDL3 submits each frame to the compositor, which forwards it to the display at the next VSYNC.
- `FlipTS` captures `sdl.TicksNS()` immediately after `Present()` returns; this reflects when the frame was *submitted to the compositor*, not when photons reached the screen. The additional compositor latency is typically **0–1 frames** (~0–17 ms at 60 Hz) and is consistent across trials, so it does not inflate RT variance but does add a fixed bias.
- Onset jitter (as measured by the Go monotonic clock) is typically **2–5 ms** on macOS, owing to Metal's internal frame-pacing algorithm.

Windows

- In **exclusive fullscreen** mode (`fullscreen=true`), the DWM (Desktop Window Manager) compositor is bypassed. Timing behaviour is similar to Linux without a compositor — jitter is typically **< 1 ms**.
- In **windowed mode**, DWM is always active and adds approximately one frame of compositor latency. Jitter is typically **1–3 ms**.

- For the most reliable timing on Windows, always run in fullscreen mode.

What `OnsetNS` measures

`TimingLog.OnsetNS` is recorded immediately after `screen.FlipTS()` returns, not at the moment photons reach the screen. On top of the compositor latency noted above, there is additional hardware pipeline latency (GPU scan-out, cable propagation, display panel response) of typically **0–2 frames**. This latency is constant across trials and does not affect within-experiment RT precision, but it must be accounted for if absolute (photodiode-verified) onset times are required.

Minimum duration

Because presentation is VSYNC-locked, durations shorter than one frame period (e.g. < 16.7 ms at 60 Hz) are rounded up to the nearest whole frame. A stimulus requested for 50 ms on a 60 Hz display is shown for exactly **3 frames (50.0 ms)**; one requested for 60 ms is shown for **4 frames (66.7 ms)**.

Audio stream

The same model applies to sounds:

```
// All tones must be pre-loaded before the stream
for _, t := range tones {
    t.PreloadDevice(exp.AudioDevice)
}

elements := stimuli.MakeRegularSoundStream(tones, 200*time.Millisecond, 100*time.Millis
events, logs, err := stimuli.PlayStreamOfSounds(elements)
```

---

11. Audio

Sounds from files or embedded bytes

```
// From a file
snd := stimuli.NewSound("ping.wav")
snd.PreloadDevice(exp.AudioDevice)
snd.Play()
snd.Wait()   // block until playback finishes

// From embedded bytes (go:embed)
//go:embed assets/beep.wav
var beepWav []byte

snd := stimuli.NewSoundFromMemory(beepWav)
snd.PreloadDevice(exp.AudioDevice)
snd.Play()
```

Procedural tones

```
tone := stimuli.NewTone(1000.0, 200*time.Millisecond, 200) // 1 kHz, 200 ms, medium vol
tone.PreloadDevice(exp.AudioDevice)
tone.Play()
```

Volume is 0–255. Use `PreloadDevice` once; then call `Play` repeatedly.

Segment playback with fade

Play only part of a longer sound, with smooth fade-in and fade-out:

```
snd.PlaySegment(1.5, 3.0, 0.05) // play 1.5–3.0 s, 50 ms ramps
```

This is useful for stimuli like vowels extracted from longer recordings.

Built-in feedback sounds

```
exp.Audio.PlayBuzzer()   // play the embedded error/incorrect sound asynchronously
exp.Audio.PlayCorrect()  // play the embedded correct/reward sound asynchronously
```

Both return immediately; the sound plays in the background.

Synchronous vs asynchronous

`snd.Play()` starts playback and returns immediately. `snd.Wait()` blocks until it finishes. For trial timing where you need to know when a sound ended, call both:

```
snd.Play()
doSomethingElse()
snd.Wait()  // wait here if needed
```

For fire-and-forget feedback sounds, call `snd.Play()` without `snd.Wait()`.

---

12. Experimental Design and Randomization

When to use `design.Block` / `design.Trial`

The `design` package provides a structured `Experiment → Block → Trial` hierarchy with string-keyed factors. Use it when:

- You have a multi-block experiment and want `exp.Design.Blocks` to drive the loop.
- You want between-subjects Latin-square counterbalancing (`AddBWSFactor`).

For simple single-block experiments, a plain Go slice of a custom struct is often more readable (and type-safe). Both approaches are valid.

Building a design

```
block := design.NewBlock("main")
for _, cond := range []string{"congruent", "incongruent"} {
    t := design.NewTrial()
    t.SetFactor("condition", cond)
    t.SetFactor("soa", 200)
    block.AddTrial(t, 20, false)  // 20 copies, appended in order
}
block.ShuffleTrials()
exp.AddBlock(block, 1)
```

Iterate at runtime:

```
for _, blk := range exp.Design.Blocks {
    for _, trial := range blk.Trials {
        cond := trial.GetFactor("condition").(string)
        soa  := trial.GetFactor("soa").(int)
        // ...
    }
}
```

Randomization helpers

The `design` package provides randomization functions that work on any slice type:

```
// In-place shuffle (generic — works on any []T)
design.ShuffleList(mySlice)

// Random integer in [a, b] inclusive
design.RandInt(500, 1500)

// Random element from a slice
word := design.RandElement(wordList)

// Weighted coin flip
if design.CoinFlip(0.75) { /* 75% chance */ }

// Truncated normal sample in [a, b]
design.RandNorm(800.0, 1200.0)

// Shuffled integer range [first, last]
order := design.RandIntSequence(0, len(stimuli)-1)
```

Between-subjects counterbalancing

```
// Register once during setup
exp.AddBWSFactor("mapping", []interface{}{"F=left/J=right", "F=right/J=left"})
```

```
// At runtime — returns the condition assigned to this subject's ID
mapping := exp.GetPermutedBWSFactorCondition("mapping").(string)
```

The assignment follows a balanced Latin square so that conditions rotate across subjects (subject 1 → condition A, subject 2 → condition B, subject 3 → condition A, …).

Constrained shuffling

For designs where you need to prevent the same condition from appearing more than N times in a row, use block-level `AddTrial` with `randomPosition: true`, which inserts each trial at a random position rather than appending:

```
for _, cond := range conditions {
    t := design.NewTrial()
    t.SetFactor("condition", cond)
    block.AddTrial(t, 5, true)  // 5 copies each, randomly interleaved during construct
}
```

---

13. Animated Stimuli

Three functions run self-contained VSYNC-locked animation loops. All three:

- Disable GC for the duration of the loop.
- Drain the SDL event queue before the first frame.
- Return a `MotionResult` with the response key, mouse button, and reaction time.

```
type MotionResult struct {
    Key    sdl.Keycode  // interrupt key pressed (0 if none)
    Button uint8        // mouse button (0 if none)
    RTms   int64        // ms from first frame to response; total elapsed on timeout
}
```

Moving dot cloud

```
result, err := stimuli.PresentMovingDotCloud(
    exp.Screen,
    100,                          // number of dots
    3.0,                          // dot radius (px)
    150.0,                        // cloud radius (px)
    control.Origin(),             // center at (0, 0)
    150.0,                        // speed in px/sec
    5000,                         // max duration ms (0 = infinite)
    []control.Keycode{control.K_SPACE}, // interrupt keys
    false,                        // catch mouse?
    control.White,                // dot color
    control.Color{A: 0},          // background (transparent)
```

```
)
fmt.Printf("RT: %d ms\n", result.RTms)
```

Drifting grating

```
result, err := stimuli.PresentMovingGrating(
    exp.Screen,
    400, 400,              // width, height (px)
    control.Origin(),      // center
    0.0,                   // orientation (degrees; 0 = vertical bars drifting right)
    0.05,                  // spatial frequency (cycles per pixel)
    2.0,                   // temporal frequency (Hz)
    0.8,                   // contrast [0, 1]
    0.5,                   // mean luminance [0, 1]
    3000,                  // max duration ms
    []control.Keycode{control.K_SPACE},
    false,
)
```

Drifting Gabor patch

```
result, err := stimuli.PresentMovingGabor(
    exp.Screen,
    200,                   // bounding box size (px); at least 6×sigma
    30.0,                  // sigma (Gaussian SD in px)
    control.Origin(),      // center
    45.0,                  // orientation
    0.05,                  // spatial frequency (cycles/px)
    3.0,                   // temporal frequency (Hz)
    0.9,                   // contrast
    0.5,                   // mean luminance
    0,                     // max duration (0 = until response)
    []control.Keycode{control.K_SPACE},
    false,
)
```

---

14. Gamma Correction and Luminance Linearity

The gamma problem

Standard monitors apply a power-law transfer function when converting digital RGB values to physical luminance:

`L(V) = k · (V/255)^γ` $\gamma \approx 2.2$ for sRGB/LCD

This means equal steps in RGB values do **not** produce equal steps in physical luminance (cd/m²). For example, with $\gamma = 2.2$:

RGB value	Physical luminance (% of max)
0	0%
64	4.9%
128	21.6%
192	52.2%
255	100%

This matters in psychophysics experiments where luminance values are specified as physical quantities (e.g. contrast masks, threshold estimation, magnitude estimation).

Inverse-gamma LUT

goxpyriment corrects for this by pre-computing a 256-entry look-up table (LUT) per channel. For each desired linear luminance value v (0–255), the LUT stores the digital value required to achieve it:

`LUT[v] = 255 · (v/255)^(1/γ)`

Applying the LUT before sending a color to the renderer means the monitor's forward gamma converts it back to the intended linear value.

Enabling gamma correction

```
// After exp.Initialize() and before the trial loop:
exp.SetGamma(2.2)   // typical sRGB monitor

// Or with per-channel calibration (from photometer measurements):
exp.GammaCorrector = apparatus.NewGammaCorrector(2.1, 2.2, 2.3)
```

Then wrap every color with `exp.CorrectColor`:

```
// Specify colors in linear luminance space (0–255).
// exp.CorrectColor maps them to the physical digital values.
disk := stimuli.NewFilledCircle(exp.CorrectColor(control.RGB(128, 128, 128)), radius)
```

When `SetGamma` has not been called, `CorrectColor` is a no-op that returns the color unchanged, so it is safe to always call it.

Measuring your monitor's gamma

The most accurate approach is photometer-based: measure luminance at several RGB levels, fit the model L(V) = k·(V/255)^γ, and pass the resulting γ to `SetGamma`. Without a photometer, $\gamma = 2.2$ is a reasonable default for modern sRGB displays.

Practical example

The `examples/Magnitude-Estimation-Luminosity` example accepts a `-gamma` flag:

```
go run main.go -gamma 2.2
```

With the flag set, the 7 luminance levels (10, 25, 50, 100, 150, 200, 255) are treated as linear targets and corrected before display, so they produce physically uniform luminance steps.

---

15. Hardware Triggers and TTL Devices

EEG and MEG recordings require a synchronisation signal — a short TTL pulse sent at the exact moment a stimulus is presented — so that electrophysiological data can be time-locked to experimental events. The `triggers` package provides this, along with the ability to read TTL inputs from response hardware such as fiber-optic response pads.

Concepts

All trigger devices in goxpyriment share two interfaces:

- **`OutputTTLDevice`** — send trigger codes (set lines HIGH/LOW, generate pulses)
- **`InputTTLDevice`** — read response button states (poll or block)

Lines are **0-indexed (0–7)**. Bit N of a bitmask drives line N.

Sending triggers (EEG event codes)

```
import (
    "github.com/chrplr/goxpyriment/triggers"
    "time"
)

// Auto-detect a DLP-IO8-G; falls back to NullOutputTTLDevice (no-op) if absent.
out, _, err := triggers.AutoDetectDLPIO8()
if err != nil { log.Fatal(err) }
defer out.Close()

// In the trial loop:
onset, _ := exp.ShowTS(stim)
out.Pulse(0, 10*time.Millisecond)  // 10 ms pulse on line 0 = EEG event marker
```

For the NeuroSpin MEGTTLBox:

```
box, err := triggers.NewMEGTTLBox("/dev/ttyACM0")
defer box.Close()
```

```
box.Pulse(0, 5*time.Millisecond)   // single line
box.PulseMask(0b00000011, 5*time.Millisecond)  // lines 0 and 1 simultaneously
```

Timing advice

The output pulse should be sent as close as possible to `exp.ShowTS(stim)`. Because both `Screen.FlipTS` and the trigger output use the system's real-time clock, the latency between the VSYNC flip and the TTL edge is typically under 1 ms. The EEG amplifier records this edge, and the exact onset can be recovered by subtracting `int64(triggerEdgeNS - onsetNS)` if the amplifier's sample clock is synchronised.

Reading response inputs (FORP pads)

The MEGTTLBox wires a fiber-optic response pad (fORP) to its 8 TTL input lines. Use `WaitForInput` directly or via the `ResponseDevice` wrapper (section 7):

```
// Via InputTTLDevice directly
_ = box.DrainInputs(ctx)                    // clear stale presses
mask, rt, err := box.WaitForInput(ctx)
buttons := triggers.DecodeMask(mask)
for _, b := range buttons {
    fmt.Println(b)  // e.g. "left blue", "right red"
}

// Via ResponseDevice
rd := apparatus.NewTTLResponseDevice(box, 5*time.Millisecond)
_ = rd.DrainResponses(ctx)
resp, _ := rd.WaitResponse(ctx)
// resp.Code == bitmask, resp.Precise == false (software poll)
```

fORP button mapping (NeuroSpin wiring):

FORPButton constant	Line	Arduino pin	STI channel
`FORPLeftBlue`	0	D22	STI007
`FORPLeftYellow`	1	D23	STI008
`FORPLeftGreen`	2	D24	STI009
`FORPLeftRed`	3	D25	STI010
`FORPRightBlue`	4	D26	STI012
`FORPRightYellow`	5	D27	STI013
`FORPRightGreen`	6	D28	STI014
`FORPRightRed`	7	D29	STI015

Supported devices

Device	Type	Output	Input	Notes
DLP-IO8-G	`DLPIO8`	✓	✓	USB-CDC serial; ASCII protocol
NeuroSpin MEGTTLBox	`MEGTTLBox`	✓	✓	Arduino Mega; binary protocol; fORP input
LPT parallel port	`ParallelPort`	✓	—	Linux only; ppdev ioctl
None / fallback	`NullOutputTTLDevice`	✓ (no-op)	—	Safe default; returned by `AutoDetectDL-PIO8`

---

16. Display Compositor Bypass

A compositing window manager (compositor) intercepts every application frame and blends it with other on-screen elements before sending the result to the display. This introduces additional latency and jitter that are problematic for millisecond-accurate stimulus presentation. The degree to which `goxpyriment` can avoid this overhead depends on the operating system.

On **Linux with X11**, SDL3 automatically sets the `_NET_WM_BYPASS_COMPOSITOR` window property when the application enters fullscreen mode. Compliant compositors (KWin, Mutter, Compton) respond by *unredirecting* the fullscreen window — routing its framebuffer directly to the display scan-out pipeline without compositing. The result is presentation latency below 1 ms, comparable to a compositor-free session. On **Linux with Wayland**, the compositor is always the intermediary for frame delivery, but modern compositors (KWin 5.21+, GNOME Mutter 44+) support *direct scan-out* for fullscreen applications, achieving similar low-latency behavior in practice. The best achievable timing on Linux is obtained by running without any display server: setting the environment variable `SDL_VIDEO_DRIVER=kmsdrm` before launching the experiment directs SDL3 to communicate with the Linux KMS/DRM subsystem directly, bypassing X11 and Wayland entirely. This configuration — typically used from a virtual terminal (`Ctrl+Alt+F2`) — is recommended for the most demanding timing requirements such as single-frame subliminal stimulation or high-frequency RSVP.

On **Windows**, SDL3's fullscreen mode creates an exclusive fullscreen surface that bypasses the Desktop Window Manager (DWM), again yielding frame-interval jitter typically below 0.3 ms (one standard deviation), comparable to Linux.

macOS is the exception. Apple's Metal rendering pipeline always delivers frames to the display through the WindowServer compositor, and no third-party application can bypass it — even in fullscreen. The WindowServer typically adds one frame of fixed latency (8–17 ms depending on refresh rate) and 2–5 ms of frame-interval jitter.

This onset uncertainty — larger than many SOAs of experimental interest — makes macOS unsuitable for production data collection. Researchers should prefer Linux or Windows hardware for laboratory deployments of `goxpyriment`; macOS is adequate for pilot testing and paradigm development.

These platform differences can be quantified empirically with the `display` and `frames` sub-tests of the bundled `Timing-Tests` suite.

---

17. Variable Refresh Rate (VRR) Stimulus Presentation

Standard displays refresh at a fixed rate (e.g., 60 Hz), which quantises every stimulus duration to multiples of the frame period (16.67 ms at 60 Hz). A researcher wishing to present a stimulus for 10 ms, 17 ms, or 23 ms cannot do so on such a display without relying on temporal dithering or hardware frame-blending techniques that introduce their own artefacts. Variable Refresh Rate (VRR) technology — marketed as AMD FreeSync, NVIDIA G-Sync, or the VESA Adaptive-Sync standard — removes this constraint by allowing the display to hold each frame for an arbitrary duration and refresh only when instructed to do so by the GPU.

`goxpyriment` exposes VRR through a single API call: `exp.Screen.SetVSync(0)` disables the VSYNC block, so that every subsequent `SDL_RenderPresent` call returns immediately. The following pattern then achieves arbitrary stimulus durations with duration error typically below 0.5 ms (one standard deviation):

1. Draw the stimulus and call `screen.FlipTS()`, which presents the frame and records `onsetNS` (an SDL nanosecond timestamp) immediately after `Present()` returns.
2. Busy-wait for the desired duration d: spin until `time.Now() >= onset + d`. The last ~500 µs use a CPU spin loop (already used in the trigger timing routines) to eliminate OS scheduling jitter.
3. Draw the blank screen, call `FlipTS` again to record `offsetNS`.

The actual software-controlled duration is `(offsetNS - onsetNS) / 1e6` ms. On a VRR-capable display this equals the true on-screen duration plus a small, constant hardware pipeline latency (GPU scan-out and panel response, typically 1–5 ms) that can be calibrated once with a photodiode using the `frames` timing sub-test of the `Timing-Tests` suite.

Every VRR panel supports VRR only within a finite refresh-rate window (e.g., 48–144 Hz, corresponding to frame durations of 6.9–20.8 ms for a typical gaming monitor). Durations outside this window cause the panel to revert to fixed-rate behaviour, re-introducing quantisation. The `Timing-Tests` suite includes a dedicated `vrr` sub-test that sweeps target durations from 1 ms to a user-specified maximum in 1 ms steps and reports duration errors at each step, directly revealing the panel's VRR window in the output data.

On a display without VRR support, disabling VSYNC produces frame tearing. The `vrr` sub-test detects this automatically: on a non-VRR display the duration errors clus-

ter at multiples of the frame period rather than being uniformly small, providing an unambiguous diagnostic.

---

18. Putting It All Together

Here is a skeleton that illustrates how the concepts compose in a realistic experiment:

```
package main

import (
    "fmt"
    "log"

    "github.com/chrplr/goxpyriment/control"
    "github.com/chrplr/goxpyriment/design"
    "github.com/chrplr/goxpyriment/stimuli"
)

const (
    NReps           = 10
    FixationDuration = 500
    ResponseTimeout  = 3000
)

type trialDef struct {
    word  string
    color control.Color
    name  string
}

func main() {
    exp := control.NewExperimentFromFlags("Color Word Task", control.Black, control.Whi
    defer exp.End()

    exp.AddDataVariableNames([]string{"trial", "word", "ink", "key", "rt_ms", "correct"

    // Build trial list
    words  := []string{"RED", "GREEN", "BLUE"}
    colors := []control.Color{control.Red, control.Green, control.Blue}
    names  := []string{"red", "green", "blue"}
    keys   := []control.Keycode{control.K_R, control.K_G, control.K_B}

    var trials []trialDef
    for _, word := range words {
        for j := range colors {
            trials = append(trials, trialDef{word, colors[j], names[j]})
        }
```

```
    }
    for i := 0; i < NReps-1; i++ {
        trials = append(trials, trials[:9]...)
    }
    design.ShuffleList(trials)

    // Preload stimuli
    fixation := stimuli.NewFixCross(20, 2, control.White)

    err := exp.Run(func() error {
        // Preload fixation cross (it will be used every trial)
        stimuli.PreloadVisualOnScreen(exp.Screen, fixation)

        exp.ShowInstructions(
            "Name the INK COLOR of each word.\n\n" +
            "R = Red   G = Green   B = Blue\n\n" +
            "Press SPACE to start.",
        )

        for i, t := range trials {
            // Fixation
            exp.Show(fixation)
            exp.Wait(FixationDuration)

            // Stimulus
            stim := stimuli.NewTextLine(t.word, 0, 0, t.color)
            exp.Show(stim)

            // Response
            key, rt, _ := exp.Keyboard.WaitKeysRT(keys, ResponseTimeout)

            // Score
            correct := false
            for j, k := range keys {
                if key == k && names[j] == t.name {
                    correct = true
                    break
                }
            }

            exp.Data.Add(i, t.word, t.name, key, rt, correct)
            fmt.Printf("trial %3d  %s/%s  rt=%dms  %v\n", i, t.word, t.name, rt, correc

            if !correct {
                exp.Audio.PlayBuzzer()
            }
```

```
            // Release texture (new TextLine each trial)
            stim.Unload()

            exp.Blank(500)
        }

        return control.EndLoop
    })

    if err != nil && !control.IsEndLoop(err) {
        log.Fatalf("experiment error: %v", err)
    }
}
```

Key patterns illustrated:

- `NewExperimentFromFlags` + `defer exp.End()` at the top.
- `exp.AddDataVariableNames` before the run loop.
- `design.ShuffleList` for trial randomization.
- `stimuli.PreloadVisualOnScreen` inside `exp.Run` for timing-sensitive stimuli.
- `exp.Show` for single stimuli; `exp.Blank` for ITIs.
- `exp.Keyboard.WaitKeysRT` for response + RT.
- `stim.Unload()` for dynamically created stimuli.
- `exp.Audio.PlayBuzzer()` for feedback.
- `return control.EndLoop` at the end.

For the complete function signatures and type definitions, see the API Reference. For more worked examples, browse the `examples/` directory.