goxpyriment — API Reference

Contents

goxpyriment API Reference 1
Package Overview . . . 1
Package `control` . . . 1
Package `stimuli` . . . 8
Package `apparatus` and `results` . . . 13
Package `design` . . . 17
Package `clock` . . . 18
Package `geometry` . . . 19
Package `triggers` . . . 19
Package `assets_embed` . . . 21
Coordinate System . . . 21
Typical Experiment Structure . . . 22

goxpyriment API Reference

This guide documents the complete public API of the `goxpyriment` framework, organized by package.

Package Overview

```
control/      ← experiment lifecycle and orchestration (start here)
stimuli/      ← visual and audio stimulus objects
apparatus/   ← SDL window/renderer, keyboard, mouse, gamepad, gamma corrector
results/      ← experiment data file and output file
design/       ← trial/block structure and randomization
clock/        ← timing utilities
geometry/     ← coordinate conversion helpers
triggers/     ← hardware trigger devices (EEG sync, etc.)
```

Package `control`

Import: `github.com/chrplr/goxpyriment/control`

Boilerplate

Every experiment starts the same way:

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

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

Pre-experiment Setup Dialog

`GetParticipantInfo` opens a graphical SDL window **before** the experiment starts to collect participant demographics, monitor properties, and display preferences. Call it before `NewExperiment` / `NewExperimentFromFlags`.

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

```

// FieldType selects how a field is rendered.
type FieldType int
const (
    FieldText     FieldType = iota // text input box (default)
    FieldCheckbox                  // tick-box; value is "true" or "false"
)

// InfoField describes one entry in the dialog.
type InfoField struct {
    Name    string    // key in the returned map
    Label   string    // displayed label
    Default string    // initial value
    Type    FieldType // FieldText (default) or FieldCheckbox
}
```

Types

Pre-built field sets

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

Function

Function	Description
`GetParticipantInfo(title string, fields []InfoField) (map[string]string, error)`	Shows the dialog and returns collected values. Returns `ErrCancelled` if the user closes or presses Escape without confirming.

Session persistence All values except `subject_id` are saved to `~/.cache/goxpyriment/last_sess` on OK and pre-filled on the next run. `subject_id` is always reset.

```
info, err := control.GetParticipantInfo("My Experiment", fields)
// ...
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 — they are written to the .csv header aut
exp.SubjectID, _ = strconv.Atoi(info["subject_id"])
exp.Info = info

if err := exp.Initialize(); err != nil { log.Fatal(err) }
defer exp.End()
```

Using the fullscreen checkbox and persisting to the data file `Initialize()` writes a `--PARTICIPANT INFO` block to the `.csv` header whenever `exp.Info` is non-nil at that point. No explicit call to `WriteParticipantInfo` is needed.

```
control.ErrCancelled  // returned when the user cancels the dialog
```

Sentinel error

Constructor Functions

Function	Description
`NewExperimentFromFlags(name string, bg, fg Color, fontSize float32) *Experiment`	Creates and fully initializes an experiment from `-w` (windowed 1024×768), `-d N` (display index, -1 = primary), and `-s N` (subject ID) command-line flags. Calls `log.Fatal` on error. **This is the preferred entry point.**
`NewExperiment(name string, width, height int, fullscreen bool, bg, fg Color, fontSize float32) *Experiment`	Lower-level constructor; call `Initialize()` before use.

Lifecycle Methods

Method	Description
`exp.Initialize() error`	Initializes SDL, audio, window, renderer, font, and data file.
`exp.End()`	Cleans up all resources. Always `defer exp.End()` immediately after construction.
`exp.Run(logic func() error) error`	Runs the main trial loop on the SDL main thread. Return `control.EndLoop` to exit cleanly.
`exp.HideCursor() error`	Hides the mouse cursor. Call after `Initialize()` to prevent the cursor from appearing over stimuli.
`exp.ShowCursor() error`	Makes the mouse cursor visible again.

Presentation Methods

Method	Description
`exp.Show(stim VisualStimulus) error`	Clear → draw → flip. The standard one-call stimulus presentation.
`exp.ShowTS(stim VisualStimulus) (uint64, error)`	Clear → draw → flip, and return the SDL nanosecond timestamp captured immediately after the VSYNC flip. Use with `GetKeyEventTS` for hardware-precision RT measurement.
`exp.ShowInstructions(text string) error`	Display centered text and wait for spacebar.
`exp.Blank(ms int) error`	Clear and flip screen, then wait `ms` milliseconds.

Method	Description
`exp.Wait(ms int) error`	Wait `ms` ms while pumping SDL events (ESC-abortable).
`exp.ShowSplash(waitForKey bool) error`	Show experiment name + version splash.
`exp.Flip() error`	Present the backbuffer (VSYNC-locked when VSync is enabled).

Input

Method	Description
`exp.Keyboard`	`*apparatus.Keyboard` — see Keyboard section
`exp.Mouse`	`*apparatus.Mouse` — see Mouse section
`exp.PollEvents(handle func(sdl.Event) bool) EventState`	Process all pending SDL events; optionally forward to a handler. Returns `EventState` including nanosecond timestamps.
`exp.HandleEvents() (Keycode, uint32, error)`	Convenience wrapper: returns `(key, mouseButton, error)`.

`EventState` now includes SDL event timestamps:

```go
type EventState struct {
    LastKey            sdl.Keycode
    LastMouseButton    uint32
    LastKeyTimestamp   uint64  // SDL nanosecond timestamp of the last key event
    LastMouseTimestamp uint64  // SDL nanosecond timestamp of the last mouse event
    QuitRequested      bool
}
```

Design and Data

Method	Description
`exp.AddDataVariableNames(names []string)`	Register CSV column names for the data file.
`exp.Data.Add(values ...interface{})`	Append a data row. Subject ID is prepended automatically.
`exp.AddBlock(b *design.Block, copies int)`	Add trial blocks to the experiment.
`exp.ShuffleBlocks()`	Randomize block presentation order.
`exp.AddBWSFactor(name string, conditions []interface{})`	Register a between-subjects factor for Latin-square counterbalancing.
`exp.GetPermutedBWSFactorCondition(name string) interface{}`	Return this subject's condition for a BWS factor.

Method	Description
`exp.Design`	`*design.Experiment` — full design object
`exp.Info`	`map[string]string` — values from `GetParticipantInfo`; set before `Initialize()` to persist them automatically to the `.csv` header

Font and Display

Method	Description
`exp.LoadFont(path string, size float32) error`	Load a TTF font from file.
`exp.LoadFontFromMemory(data []byte, size float32) error`	Load a TTF font from a byte slice.
`exp.SetVSync(vsync int) error`	Toggle vertical sync (1 = on, 0 = off).
`exp.SetLogicalSize(w, h int32) error`	Set device-independent logical resolution.
`exp.SetOutputDirectory(dir string)`	Override default data file directory (`~/goxpy_data`).

Gamma Correction

Standard monitors apply a power-law transfer function $L(V) = k \cdot (V/255)^\gamma$ ($\gamma \approx 2.2$ for sRGB displays). Equal steps in RGB values do **not** produce equal steps in physical luminance. Use `SetGamma` to enable inverse-gamma correction.

Method	Description
`exp.SetGamma(gamma float64)`	Install a uniform inverse-gamma corrector. Call once after `Initialize()`.
`exp.CorrectColor(c sdl.Color) sdl.Color`	Apply gamma correction to a color. Returns `c` unchanged when no corrector is set.
`exp.GammaCorrector`	`*apparatus.GammaCorrector` — set directly for per-channel calibration.

```go
// Uniform gamma (typical sRGB monitor)
exp.SetGamma(2.2)

// Per-channel gamma (from photometer measurements)
exp.GammaCorrector = apparatus.NewGammaCorrector(2.1, 2.2, 2.3)

// Use in trial loop — specify colors in linear luminance space (0–255)
disk := stimuli.NewFilledCircle(exp.CorrectColor(control.RGB(128, 128, 128)), radius)
```

The `apparatus.GammaCorrector` type is also available directly:

```
gc := apparatus.NewGammaCorrectorUniform(2.2)
corrected := gc.CorrectColor(sdl.Color{R: 128, G: 128, B: 128, A: 255})
// corrected.R ≈ 186 — the physical digital value for 50% luminance on γ=2.2
```

Colors, Types, and Constants

```
// Named colors
control.Black, White, Red, Green, Blue, Yellow, Magenta, Cyan
control.Gray, DarkGray, LightGray

// Type aliases (so you only need to import "control")
type Color   = sdl.Color
type Keycode = sdl.Keycode
type FPoint  = sdl.FPoint
type FRect   = sdl.FRect

// Constructors
control.RGB(r, g, b uint8) Color
control.RGBA(r, g, b, a uint8) Color
control.Point(x, y float32) FPoint
control.Origin() FPoint  // returns (0, 0)

// Font helpers
control.FontFromFile(path string, size float32) (*ttf.Font, error)
control.FontFromMemory(data []byte, size float32) (*ttf.Font, error)

// Loop control
control.EndLoop          // sentinel error: return from Run callback to exit
control.IsEndLoop(err)   // test whether an error is the EndLoop sentinel

// Keyboard codes (only the exported subset)
control.K_SPACE, K_ESCAPE, K_RETURN, K_BACKSPACE
control.K_UP, K_DOWN, K_LEFT, K_RIGHT
control.K_S, K_D, K_F, K_J, K_K, K_L
control.K_Q, K_R, K_G, K_B, K_Y, K_N, K_P
control.K_1, K_2, K_3, K_4
control.K_KP_1, K_KP_2, K_KP_3, K_KP_4

// Mouse buttons
control.BUTTON_LEFT, BUTTON_RIGHT
```

> **Tip:** For key codes not listed above (e.g. `K_A`), import `go-sdl3/sdl` directly and use `sdl.K_A`.

Audio

```
exp.AudioDevice  // sdl.AudioDeviceID — pass to Sound.PreloadDevice()

// Top-level helper (call before NewExperiment)
control.SetAudioSampleFrames(frames int)  // set audio buffer size (256–2048)
```

Package stimuli

Import: `github.com/chrplr/goxpyriment/stimuli`

Interfaces

```
type Stimulus interface {
    Present(screen *apparatus.Screen, clear, update bool) error
    Preload() error
    Unload() error
}

type VisualStimulus interface {
    Stimulus
    Draw(screen *apparatus.Screen) error
    GetPosition() sdl.FPoint
    SetPosition(pos sdl.FPoint)
}
```

GPU textures are **lazily allocated** on the first `Draw`. To force early allocation (for timing-sensitive code), use:

```
stimuli.PreloadVisualOnScreen(screen, stim)     // single stimulus
stimuli.PreloadAllVisual(screen, []VisualStimulus{...})  // slice
```

Visual Stimuli

Text

Constructor	Description
`NewTextLine(text string, x, y float32, color Color) *TextLine`	Single line of text.
`NewTextBox(text string, width int32, pos FPoint, color Color) *TextBox`	Word-wrapped multi-line text.

Both support a `Font *ttf.Font` field — set it to override the screen default.

Shapes

Constructor	Description
`NewFixCross(size, lineWidth float32, color Color) *FixCross`	Fixation cross centered at (0, 0).
`NewCircle(radius float32, color Color) *Circle`	Filled circle.
`NewRectangle(cx, cy, w, h float32, color Color) *Rectangle`	Filled rectangle.
`NewLine(x1, y1, x2, y2 float32, color Color) *Line`	Line segment.

Images and Video

Constructor/Function	Description
`NewPicture(filePath string, x, y float32) *Picture`	Image loaded from file (PNG, JPG, BMP…).
`NewPictureFromMemory(data []byte, x, y float32) *Picture`	Image loaded from embedded bytes.
`PlayGv(screen, path string, x, y float32) ([]UserEvent, error)`	Play a `.gv` (LZ4-compressed RGBA) video file, VSYNC-locked.
`NewGvVideo(path string) (*GvVideo, error)`	Open a `.gv` file for frame-by-frame access.

Psychophysics Stimuli

Constructor	Description
`NewGaborPatch(sigma, theta, lambda, phase, psi, gamma float64, bgColor Color, size float32) *GaborPatch`	Static Gabor patch. `theta` in degrees, `lambda` = spatial wavelength in pixels.
`NewDotCloud(radius float32, bgColor, dotColor Color) *DotCloud`	Static random-dot cloud. Call `Make(nDots, dotRadius, gap)` to populate.
`NewRDS(imgSize, innerSize [2]int, shift, gap, scale int) *RDS`	Random-dot stereogram (side-by-side pair).
`NewVisualMask(w, h, dotW, dotH float32, bgColor, dotColor Color, pct int) *VisualMask`	Random-dot masking stimulus. `pct` = dot fill percentage 0–100.

Composite / Interactive

Constructor	Description
`NewThermometerDisplay(size FPoint, nSegments int, state, goal float32) *ThermometerDisplay`	Segmented progress bar. `State` and `Goal` in 0–100.
`NewChoiceGrid(choices []string, maxSelect int, prompt string) *ChoiceGrid`	Multiple-choice button grid (mouse + keyboard). See below.
`NewTextInput(msg string, pos FPoint, boxW float32, bgColor, frameColor, textColor Color) *TextInput`	Free-text keyboard input box. Call `ti.Get(screen, keyboard)`.
`NewMenu(items []string) *Menu`	Numbered keyboard-navigable list. Call `m.Get(screen, keyboard, initialSel)`.

ChoiceGrid

```
cg := stimuli.NewChoiceGrid(choices, maxSelect, prompt)
cg.Cols = 7       // optional: set column count (0 = auto)

selections, err := cg.Get(exp.Screen, exp.Keyboard)
// selections is a []string preserving selection order
```

- `MaxSelect > 0`: auto-submits after N selections.
- `MaxSelect == 0`: participant presses ENTER or SPACE to submit.
- BACKSPACE removes the last selection.
- Both mouse click and matching keypress (single-char labels) activate buttons.

Menu

```
m := stimuli.NewMenu([]string{"Option A", "Option B", "Option C"})
m.Pos = sdl.FPoint{X: 0, Y: 0}    // optional: reposition (default = screen center)
m.HighlightColor = control.Yellow  // optional: override highlight color

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

Navigation: UP/DOWN arrows move the highlight; ENTER or SPACE confirms; number keys 1–9 (0 for tenth) select and confirm directly. The selected item is shown in `HighlightColor` with a > prefix; others use `TextColor`. `LineSpacing` controls vertical item spacing (0 = auto from font height).

Animated / VSYNC-locked Loops

All three functions disable GC, lock to VSYNC, and return `(MotionResult, error)`.

```
type MotionResult struct {
    Key    sdl.Keycode // interrupt key pressed (0 if none)
    Button uint8       // mouse button pressed (0 if none)
```

```go
    RTms   int64      // ms from first frame to response (or total duration on timeout
}
```

Function	Description
`PresentMovingDotCloud(screen, nDots int, dotRadius, cloudRadius float32, center FPoint, speedPxPerSec float32, maxDurationMs int64, interruptKeys []Keycode, catchMouse bool, dotColor, bgColor Color) (MotionResult, error)`	Animated random-dot cloud. Each dot moves at a fixed speed and respawns when it exits the boundary.
`PresentMovingGrating(screen, width, height float32, center FPoint, orientation, spatialFreq, temporalFreq, contrast, bgLuminance float64, maxDurationMs int64, interruptKeys []Keycode, catchMouse bool) (MotionResult, error)`	Drifting sinusoidal grating in a rectangular aperture.
`PresentMovingGabor(screen, size float32, sigma float64, center FPoint, orientation, spatialFreq, temporalFreq, contrast, bgLuminance float64, maxDurationMs int64, interruptKeys []Keycode, catchMouse bool) (MotionResult, error)`	Drifting Gabor patch with Gaussian envelope (alpha-blended edges).

Spatial frequency is in **cycles per pixel** (e.g. `0.05` = one cycle every 20 px). Temporal frequency is in **Hz**. Orientation is in **degrees from horizontal** (0° = vertical bars drifting right).

Stimulus Streams (High-Precision RSVP)

Stream functions disable GC, lock every onset and offset to a VSYNC boundary, and return `([]UserEvent, []TimingLog, error)`.

```go
type UserEvent struct {
    Event       sdl.Event     // raw SDL event (KeyboardEvent, MouseButtonEvent, …)
    Timestamp   time.Duration // time relative to stream start (Go clock, ms precision)
    TimestampNS uint64        // SDL3 hardware event timestamp, nanoseconds (same clock
}

type TimingLog struct {
    Index        int
    TargetOn     time.Duration
    ActualOnset  time.Duration // Go-clock time of first-frame draw (stream-relative)
    ActualOffset time.Duration // Go-clock time after last on-frame (stream-relative)
    OnsetNS      uint64        // SDL3 nanosecond timestamp of the VSYNC flip that turn
```

```
    OffsetNS     uint64       // SDL3 nanosecond timestamp of the VSYNC flip that turn
}
```

`UserEvent.TimestampNS` and `TimingLog.OnsetNS`/`OffsetNS` are all on the SDL3 nanosecond clock, so reaction times measured during a stream can be computed with full hardware precision:

```
for _, ev := range events {
    if ev.Event.Type == sdl.EVENT_KEY_DOWN {
        // Find the stimulus that was on-screen when the key was pressed
        for _, l := range logs {
            if ev.TimestampNS >= l.OnsetNS && ev.TimestampNS < l.OffsetNS {
                rtNS := int64(ev.TimestampNS - l.OnsetNS)
                fmt.Printf("RT from stimulus %d: %d ms\n", l.Index, rtNS/1_000_000)
            }
        }
    }
}
```

Searching event lists

Function	Description
`FirstKeyPress(events []UserEvent, key sdl.Keycode) (UserEvent, bool)`	Returns the first `KEY_DOWN` event matching key from the slice, plus a found flag.

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

```
// RSVP text stream — simplest entry point
events, logs, err := stimuli.PresentStreamOfText(
    exp.Screen, words, durationOn, durationOff, x, y, color,
)

// Image/mixed stream
elements := stimuli.MakeRegularVisualStream(stims, durationOn, durationOff)
events, logs, err := stimuli.PresentStreamOfImages(exp.Screen, elements, x, y)

// Irregular timing
elements, err := stimuli.MakeVisualStream(stims, onsetMs, durationMs)
events, logs, err := stimuli.PresentStreamOfImages(exp.Screen, elements, x, y)
```

Visual Streams

```go
// Regular timing
elements := stimuli.MakeRegularSoundStream(sounds, durationOn, durationOff)
events, logs, err := stimuli.PlayStreamOfSounds(elements)

// Irregular timing
elements, err := stimuli.MakeSoundStream(sounds, onsetMs, durationMs)
events, logs, err := stimuli.PlayStreamOfSounds(elements)
```

Audio Streams sounds is `[]stimuli.AudioPlayable` — satisfied by both `*Sound` and `*Tone`.

Audio Stimuli

```go
// WAV file
snd := stimuli.NewSound(filePath)
snd.PreloadDevice(exp.AudioDevice)
snd.Play()
snd.Wait()                                    // block until done
snd.PlaySegment(onset, offset, rampSec)       // time-delimited segment with optional f

// Embedded WAV
snd := stimuli.NewSoundFromMemory(data)

// Procedural tone
tone := stimuli.NewTone(frequency, duration, volume)   // duration: time.Duration; volu
tone.PreloadDevice(exp.AudioDevice)
tone.Play()

// One-shot helper (no preload needed)
stimuli.PlaySoundFromMemory(exp.AudioDevice, data)

// Embedded feedback sounds (via assets_embed)
import "github.com/chrplr/goxpyriment/assets_embed"
stimuli.PlaySoundFromMemory(exp.AudioDevice, assets_embed.BuzzerWav)
stimuli.PlaySoundFromMemory(exp.AudioDevice, assets_embed.CorrectWav)
```

Package `apparatus` and `results`

Import: `github.com/chrplr/goxpyriment/apparatus` (screen, input, gamma) Import: `github.com/chrplr/goxpyriment/results` (data file)

In normal experiments you access apparatus types through `exp.Screen`, `exp.Keyboard`, `exp.Mouse`, and `exp.Data`. Direct use of `apparatus` is only needed when writing custom stimulus types.

Screen

All stimulus positions use a **center-origin coordinate system**: `(0, 0)` is the screen center; positive Y is upward.

```
screen.CenterToSDL(x, y float32) (float32, float32)  // convert to SDL top-left coords
screen.MousePosition() (float32, float32)             // current cursor in center coor
screen.Clear() error                                  // fill with background color
screen.Update() error                                 // present (VSYNC-blocks)
screen.Flip() error                                   // alias for Update
screen.FlipTS() (uint64, error)                       // present + return SDL nanoseco
screen.FrameDuration() time.Duration                  // nominal frame duration (falls
screen.SetLogicalSize(w, h int32) error
screen.SetVSync(vsync int) error
screen.DisplayInfo() apparatus.DisplayInfo                   // monitor properties
screen.Destroy()
```

`FlipTS` returns `sdl.TicksNS()` captured immediately after `SDL_RenderPresent`. This timestamp is on the same nanosecond clock as SDL3 event timestamps, so `int64(event.Timestamp - onsetNS)` gives hardware-precision reaction time without any polling latency.

Keyboard

```
key, err := exp.Keyboard.Wait()                                          // any key
key, err := exp.Keyboard.WaitKey(control.K_SPACE)                       // specific key
key, err := exp.Keyboard.WaitKeys(keys, timeoutMS)                       // first of several k
key, rt, err := exp.Keyboard.WaitKeysRT(keys, timeoutMS)                 // with RT in ms from
key, ts, err := exp.Keyboard.GetKeyEventTS(keys, timeoutMS)              // with SDL event tim
events, err := exp.Keyboard.GetKeyEventsTS(keys, timeoutMS)              // first key + 50 ms
events, err := exp.Keyboard.CollectKeyEventsTS(keys, durationMS)         // all keys during fu
key, err := exp.Keyboard.Check()                                         // non-blocking poll
held := exp.Keyboard.IsPressed(key)                                      // true if key is phy
upTS, err := exp.Keyboard.WaitKeyReleaseTS(key, timeoutMS)               // wait for KEY_UP; r
exp.Keyboard.Clear()                                                     // drain SDL event qu
```

`WaitKeys` and `WaitKeysRT` return `0, nil` on timeout; return `sdl.EndLoop` on ESC or window close.

`IsPressed` queries SDL's scancode state array — no event queue involvement. `WaitKeyReleaseTS` returns the KEY_UP hardware timestamp so that `upTS - downTS` gives nanosecond-precision press duration.

`GetKeyEventsTS` — two-phase collection for bilateral responses. The timeout governs how long to wait for the *first* key. After the first key arrives, the function waits an additional 50 ms for any second key before returning. This extra window is necessary because human "simultaneous" bilateral presses (e.g. both hands at once) arrive 10–50 ms apart — a non-blocking drain after the first key would miss the second key almost every time. Use `GetKeyEventTS` for ordinary single-key trials to avoid the 50 ms overhead.

GetKeyEventTS returns the SDL3 KeyboardEvent.Timestamp field — the nanosecond time at which the hardware key-down event was generated, on the same clock as sdl.TicksNS() and Screen.FlipTS(). This allows computing reaction time from any specific stimulus onset without manual arithmetic:

```
onset, _ := exp.ShowTS(stim1)    // nanoseconds at VSYNC flip
exp.Wait(500)
exp.ShowTS(stim2)
key, eventTS, _ := exp.Keyboard.GetKeyEventTS(responseKeys, -1)
rtToStim1 := int64(eventTS - onset)  // nanoseconds
```

Mouse

```
x, y := exp.Mouse.Position()                               // current position (center c
btn, err := exp.Mouse.WaitPress()                          // block until button pressed
btn, rt, err := exp.Mouse.WaitPressRT(timeoutMS)           // with RT in ms from call si
btn, ts, err := exp.Mouse.GetPressEventTS(timeoutMS)       // with SDL event timestamp (
btn, err := exp.Mouse.Check()                              // non-blocking poll
held := exp.Mouse.IsPressed(sdl.BUTTON_LEFT)               // true if button is physical
upTS, err := exp.Mouse.WaitButtonReleaseTS(btn, timeoutMS) // wait for MOUSE_BUTTON_UP;
exp.Mouse.ShowCursor(show bool) error
```

WaitPressRT mirrors Keyboard.WaitKeysRT: reaction time is measured in milliseconds from the call site. GetPressEventTS returns the SDL3 hardware event timestamp in nanoseconds, suitable for use with ShowTS. IsPressed and WaitButtonReleaseTS mirror the keyboard's IsPressed and WaitKeyReleaseTS.

GamePad

```
pads, err := apparatus.GetGamePads()                                   // enumerate conn
defer pads[0].Close()

btn, err := pads[0].WaitPress()                               // block until any butto
btn, ts, err := pads[0].GetPressEventTS(timeoutMS)           // with SDL event timesta
```

GetPressEventTS returns the GamepadButtonEvent.Timestamp field — same nanosecond clock as Screen.FlipTS and keyboard/mouse event timestamps.

Unified Input — `WaitAnyEventTS`

When the response device is not fixed in advance (keyboard *or* mouse click), use the method on Experiment:

```
// Accept F or J key, or any mouse button, timeout after 3 s
ev, err := exp.WaitAnyEventTS(
    []control.Keycode{control.K_F, control.K_J},
    true,   // catchMouse
```

```
    3000,
)
```

Returns an `apparatus.InputEvent`:

```
type InputEvent struct {
    Device       apparatus.DeviceKind     // DeviceKeyboard | DeviceMouse | DeviceGame
    Key          sdl.Keycode       // non-zero for keyboard events
    Button       uint32            // non-zero for mouse events
    GamepadButton sdl.GamepadButton // non-zero for gamepad events
    TimestampNS  uint64            // SDL3 hardware timestamp, nanoseconds
}
```

`TimestampNS` is on the same clock as `ShowTS`, so RT computation is identical regardless of device:

```
onset, _ := exp.ShowTS(stim)
ev, _ := exp.WaitAnyEventTS(keys, true, -1)
rtNS := int64(ev.TimestampNS - onset)
```

Pass `keys = nil` to accept any key. Pass `catchMouse = false` to ignore the mouse. On timeout, returns a zero `InputEvent` and `nil` error. On ESC or quit, returns `sdl.EndLoop`.

ResponseDevice

`ResponseDevice` is a unified interface over all participant-input hardware — SDL-event-driven devices (keyboard, mouse, gamepad) **and** polled TTL devices (MEGTTLBox, DLPIO8). It is the recommended abstraction when the experiment design does not commit to a specific input modality.

```
type ResponseDevice interface {
    WaitResponse(ctx context.Context) (Response, error)
    DrainResponses(ctx context.Context) error
    Close() error
}

type Response struct {
    Source  apparatus.DeviceKind  // DeviceKeyboard | DeviceMouse | DeviceGamepad | Dev
    Code    uint32         // SDL Keycode, mouse button, gamepad button, or TTL bitmask
    RT      time.Duration  // elapsed from WaitResponse call to detection
    Precise bool           // true = hardware event timestamp; false = software poll
}
```

`Response.Precise` distinguishes two timing regimes:

Device	Precise	RT origin
Keyboard, Mouse, Gamepad	`true`	SDL3 hardware event timestamp (nanosecond)

Device	Precise	RT origin
MEGTTLBox, DLPIO8	`false`	`time.Now()` at poll detection (±poll interval, ~5 ms)

Construct wrappers with the provided adapters:

```
// SDL-event-driven devices
rd := &apparatus.KeyboardResponseDevice{KB: exp.Keyboard}
rd := &apparatus.MouseResponseDevice{M: exp.Mouse}
rd := &apparatus.GamepadResponseDevice{GP: pad}

// Polled TTL device (MEGTTLBox, DLPIO8, or any type with ReadAll/DrainInputs)
box, _ := triggers.NewMEGTTLBox("/dev/ttyACM0")
rd := apparatus.NewTTLResponseDevice(box, 5*time.Millisecond)
```

Usage in a trial loop:

```
onset, _ := exp.ShowTS(stim)
_ = rd.DrainResponses(ctx)
resp, err := rd.WaitResponse(ctx)
// resp.RT is always valid; resp.Precise tells you whether to trust nanosecond accuracy
```

DataFile

```
exp.Data.Add(field1, field2, ...)                // append a data row
exp.Data.AddVariableNames([]string{...})         // write column header
exp.Data.WriteDisplayInfo(info)                  // append display metadata as comments
exp.Data.WriteParticipantInfo(info)              // append --PARTICIPANT INFO block (calle
exp.Data.WriteEndTime()                          // append end time + duration
```

Output is written to `~/goxpy_data/<expname>_<subjectID>_<timestamp>.csv` (a CSV with #-prefixed metadata header).

Package `design`

Import: `github.com/chrplr/goxpyriment/design`

Data Structures

```
// Trial — one experimental trial
trial := design.NewTrial()
trial.SetFactor("condition", "congruent")
trial.GetFactor("condition")   // → "congruent"
trial.Copy()                   // deep copy
```

```go
// Block — a sequence of trials
block := design.NewBlock("Practice")
block.SetFactor("type", "practice")
block.AddTrial(trial, copies, randomPosition)
block.ShuffleTrials()

// Experiment design (separate from control.Experiment)
exp.Design  // *design.Experiment — contains Blocks, DataVariableNames, etc.
```

Randomization

```go
design.RandInt(a, b int) int                            // random int in [a, b]
design.RandElement(list []T) T                          // random element (generic)
design.CoinFlip(headBias float64) bool                  // weighted coin flip
design.RandNorm(a, b float64) float64                   // truncated normal in [a, b]
design.ShuffleList(list []T)                            // in-place Fisher-Yates shuffle (g
design.MakeMultipliedShuffledList(list []T, n int) []T  // n shuffled copies concatenat
design.RandIntSequence(first, last int) []int           // shuffled range [first, last]
```

Latin Square (Between-Subjects Counterbalancing)

```go
// Registration
exp.AddBWSFactor("handedness", []interface{}{"left", "right"})

// At runtime — returns the condition for the current subject
condition := exp.GetPermutedBWSFactorCondition("handedness")

// Low-level
square, err := design.LatinSquare(elements, design.PBalancedLatinSquare)
square, err := design.LatinSquareInts(n, design.PCycledLatinSquare)
// permutation types: design.PBalancedLatinSquare, PCycledLatinSquare, PRandom
```

Package `clock`

Import: `github.com/chrplr/goxpyriment/clock`

```go
clock.Wait(ms int)                    // block for ms milliseconds
clock.GetTime() int64                 // ms since package first used
clock.GetTimeNS() int64               // nanoseconds since package first used

c := clock.NewClock()                 // clock relative to "now"
c.NowMillis() int64                   // ms elapsed
c.NowNanos() int64                    // nanoseconds elapsed
c.Now() time.Duration
```

```
c.Reset()                                // restart reference
c.SleepUntil(target time.Duration)     // sleep until target offset (returns immediately
```

> **Note:** Prefer `exp.Wait(ms)` over `clock.Wait(ms)` in experiment code — `exp.Wait` pumps SDL events and detects ESC.
>
> **Clock domains:** `GetTimeNS()` and `NowNanos()` use the Go monotonic clock (`time.Since`). SDL event timestamps from `Screen.FlipTS`, `GetKeyEventTS`, `GetPressEventTS`, and `WaitAnyEventTS` use `sdl.TicksNS()`. The two clocks have different origins and **must not be subtracted from each other** for reaction-time computation. Use the SDL-based functions exclusively for RT measurement.

Package `geometry`

Import: `github.com/chrplr/goxpyriment/geometry`

```
geometry.GetDistance(p1, p2 sdl.FPoint) float32
geometry.CartesianToPolar(x, y float32) (radius, angleDeg float32)
geometry.PolarToCartesian(radius, angleDeg float32) (x, y float32)
geometry.DegreeToRadian(deg float32) float64
```

Package `triggers`

Import: `github.com/chrplr/goxpyriment/triggers`

Provides hardware TTL signal output (EEG/MEG trigger codes) and TTL input (response pads wired over serial). Lines are **0-indexed (0–7)** throughout; bit N of a bitmask corresponds to line N.

Interfaces

```
// OutputTTLDevice — send trigger codes to recording equipment.
type OutputTTLDevice interface {
    Send(mask byte) error                      // set all 8 lines from bitmask
    SetHigh(line int) error                    // drive line HIGH (0-indexed)
    SetLow(line int) error                     // drive line LOW  (0-indexed)
    Pulse(line int, d time.Duration) error     // HIGH for d, then LOW (blocks)
    AllLow() error                             // all lines LOW
    Close() error                              // AllLow + release port
}

// InputTTLDevice — read TTL inputs from response hardware.
type InputTTLDevice interface {
    ReadAll() (byte, error)                                          // bitmask of
```

```
    ReadLine(line int) (byte, error)                                  // 0 or 1 (0-i
    WaitForInput(ctx context.Context) (mask byte, rt time.Duration, err error)
    DrainInputs(ctx context.Context) error
    Close() error
}
```

DLPIO8 (DLP-IO8-G, USB-CDC serial)

Implements both `OutputTTLDevice` and `InputTTLDevice`.

```
// Auto-detect (recommended): returns NullOutputTTLDevice if not found, no error.
out, portName, err := triggers.AutoDetectDLPIO8()
defer out.Close()
out.Pulse(0, 10*time.Millisecond)   // 10 ms pulse on line 0

// Manual:
d, err := triggers.NewDLPIO8("/dev/ttyUSB0")
defer d.Close()
d.Send(0b00000101)                  // lines 0 and 2 HIGH
mask, err := d.ReadAll()            // bitmask of all 8 input lines
mask, rt, err := d.WaitForInput(ctx)
```

MEGTTLBox (NeuroSpin Arduino Mega TTL box)

Implements both `OutputTTLDevice` and `InputTTLDevice`. Provides 8 TTL output lines (D30–D37) and 8 TTL input lines for a FORP response pad (D22–D29).

```
box, err := triggers.NewMEGTTLBox("/dev/ttyACM0",
    triggers.WithResetDelay(2*time.Second),   // wait for Arduino boot (default 2 s)
    triggers.WithPollInterval(5*time.Millisecond),
)
defer box.Close()

// Output
box.Pulse(0, 5*time.Millisecond)    // pulse line 0
box.PulseMask(0b00000011, 5*time.Millisecond)  // pulse lines 0 and 1
box.Send(0b00000001)                // set line 0 HIGH, all others LOW

// Input (FORP response pad)
_ = box.DrainInputs(ctx)            // clear latched presses from previous trial
mask, rt, err := box.WaitForInput(ctx)
buttons := triggers.DecodeMask(mask)  // []FORPButton
```

`FORPButton` constants (also serve as 0-indexed line numbers for bitmask operations):

```
triggers.FORPLeftBlue    // 0
triggers.FORPLeftYellow  // 1
triggers.FORPLeftGreen   // 2
triggers.FORPLeftRed     // 3
```

```
triggers.FORPRightBlue   // 4
triggers.FORPRightYellow // 5
triggers.FORPRightGreen  // 6
triggers.FORPRightRed    // 7
```

ParallelPort (Linux LPT)

Implements `OutputTTLDevice`.

```
ports := triggers.AvailableParallelPorts()      // scans /dev/parport0..3
pp := triggers.NewParallelPort("/dev/parport0")
if err := pp.Open(); err != nil { log.Fatal(err) }
defer pp.Close()
pp.Send(0b00000111)                             // lines 0,1,2 HIGH
pp.Pulse(0, 10*time.Millisecond)
status, _ := pp.ReadStatus()                    // Linux only: status register
```

Prerequisites: `sudo modprobe ppdev`; user in the `lp` group.

Null devices

`NullOutputTTLDevice` and `NullInputTTLDevice` are silent no-ops, safe to call without hardware. `AutoDetectDLPIO8` returns `NullOutputTTLDevice` when no device is found.

Package `assets_embed`

Import: `github.com/chrplr/goxpyriment/assets_embed`

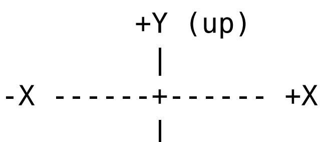
Provides embedded default assets as `[]byte` slices, ready for `FontFromMemory` or `PlaySoundFromMemory`:

```
assets_embed.InconsolataFont  []byte  // default monospace TTF font
assets_embed.BuzzerWav        []byte  // error/incorrect feedback sound
assets_embed.CorrectWav       []byte  // correct/reward feedback sound
```

Coordinate System

All stimulus positions are in **screen-center coordinates**:

- `(0, 0)` = screen center
- Positive X = right; positive Y = **up** (not down)
- `screen.CenterToSDL(x, y)` converts to SDL's top-left origin


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

```
        -Y (down)
```

Typical Experiment Structure

```go
package main

import (
    "log"
    "github.com/chrplr/goxpyriment/control"
    "github.com/chrplr/goxpyriment/design"
    "github.com/chrplr/goxpyriment/stimuli"
)

func main() {
    exp := control.NewExperimentFromFlags("My Experiment", control.Black, control.White
    defer exp.End()

    exp.AddDataVariableNames([]string{"block", "trial", "condition", "key", "rt_ms"})

    // Build design
    block := design.NewBlock("main")
    for _, cond := range []string{"left", "right"} {
        t := design.NewTrial()
        t.SetFactor("condition", cond)
        block.AddTrial(t, 10, false)
    }
    block.ShuffleTrials()
    exp.AddBlock(block, 1)

    err := exp.Run(func() error {
        exp.ShowInstructions("Press F for left, J for right.\n\nPress SPACE to start.")

        for bi, blk := range exp.Design.Blocks {
            for ti, trial := range blk.Trials {
                cond := trial.GetFactor("condition").(string)

                exp.Show(stimuli.NewFixCross(20, 2, control.White))
                exp.Wait(500)

                stim := stimuli.NewTextLine(cond, 0, 0, control.White)
                exp.Show(stim)

                key, rt, err := exp.Keyboard.WaitKeysRT(
                    []control.Keycode{control.K_F, control.K_J}, 3000,
                )
                if control.IsEndLoop(err) {
```

```
                    return control.EndLoop
                }

                exp.Data.Add(bi, ti, cond, key, rt)
                exp.Blank(500)
            }
        }
        return control.EndLoop
    })
    if err != nil && !control.IsEndLoop(err) {
        log.Fatalf("experiment error: %v", err)
    }
}
```