goxpyriment — Timing Tests

christophe@pallier.org

2026-04-06

Contents

Timing-Tests: A Guide for Researchers

This document explains how to use the `Timing-Tests` program to characterise the timing behaviour of your computer before running a psychophysical experiment.

> **Quick reference** — command-line flags, equipment table, and one-liner descriptions of each test are in `tests/Timing-Tests/README.md`. This document explains the *why*.

Why timing matters

In a psychology experiment every stimulus has an intended onset and offset time. Whether you are presenting a word on screen for exactly 100 ms, playing a tone that should coincide with a visual flash, or measuring a participant's reaction time to the nearest millisecond, you are trusting the computer to do two things correctly:

1. **Present stimuli when you ask it to.** If you call `ShowTS` or `PresentStreamOfImages()` and ask for a 100 ms word, does the word actually appear on screen for 100 ms?
2. **Record timestamps accurately.** If you record the time of a key press, is that timestamp the time the key was physically depressed, or is it delayed by software polling?

Neither of these things is guaranteed. They depend on your operating system, graphics driver, audio driver, and hardware. The Timing-Tests suite lets you measure these properties on *your specific machine* so that you can report them in your methods section and, if needed, fix problems before data collection begins.

What the tests measure

The tests are organised in four tiers, from simple (no hardware needed) to comprehensive (full lab setup with oscilloscope and photodiode).

```
Tier 0 — check      sanity check: can I see a flash and hear two sounds?
Tier 1 — display    what is my true refresh rate and frame-interval stability?
         latency    what is my audio pipeline delay?
         stream     are my sequential (RSVP) stimuli timed accurately?
      vrr     can I present stimuli for arbitrary durations (not just multiples of the fram
Tier 2 — trigger    how precise is my DLP-IO8-G trigger device?
Tier 3 — frames     does the photodiode confirm what the software reports?
                (frames-on=1 tests single-frame / minimum-duration capability)
         tones      is my audio onset timing stable over a long session?
         av         what is the actual audio–visual delay?
Tier 4 — rt         how precise is my reaction-time measurement?
```

Run the tests in this order. Each tier builds on the previous one.

These tests are run from a Terminal. This document assumes that you have compiled `Timing-Tests/main.go`:

```
go build tests/Timing-Tests/main.go -o Timing-Tests
```

Understanding the display refresh cycle

The majority of display monitors refresh the image at a fixed rate — typically 60 Hz (one new frame every 16.67 ms), 120 Hz, or 144 Hz (Others, especially gaming monitors, have variable refresh rate displays, we discuss them below). Your graphics driver synchronises stimulus presentation to this cycle (VSYNC). This has two important consequences:

- **Stimulus durations are quantised to multiples of the frame duration.** At 60 Hz you can show a stimulus for 16.67 ms, 33.33 ms, 50 ms, etc., but not for 25 ms (there is no "half-frame" on an LCD). Plan your stimulus durations accordingly.

- **The exact onset time of a stimulus is not the moment you call `ShowTS`. It is the moment the next VSYNC occurs**, which could be anywhere between 0 and 16.67 ms later. goxpyriment's VSYNC-locked rendering keeps this offset constant and predictable once the pipeline is warmed up, but the warm-up phase (first few frames) should always be excluded from analysis — the `-warmup` flag does this automatically.

Tier 0 — Sanity check (`check`)

Before any measurement, verify that the basic hardware is responding:

```
Timing-Tests -test check
```

The program will: 1. Flash a bright white screen for one second (you should see it clearly). 2. Play a buzzer sound (you should hear it through speakers or headphones). 3. Play a ping sound one second later.

If you do not see the flash or do not hear either sound, stop and diagnose the problem before continuing. Common causes: the window opened on the wrong monitor (`-d` flag), the audio device is muted, or the default audio output is not your speakers.

Tier 1 — Self-contained measurements

`display` — frame timing and refresh rate

```
Timing-Tests -test display -duration-s 30
```

This test flips a gray screen continuously for the specified duration and records the wall-clock interval between consecutive `RenderPresent` calls (each of which blocks until the next VSYNC). It answers:

- **What is my monitor's true refresh rate?** The nominal rate may be 60 Hz but the actual measured rate could be 59.94 Hz. Use the measured value with the `-hz` flag in subsequent tests so that "N frames" targets are exact.
- **How variable is my frame timing?** On a well-configured Linux system with no desktop compositor (X11 + direct rendering), the standard deviation of frame intervals is typically below 0.1 ms. On macOS or Windows with an active compositor, it is typically 1–3 ms. A high standard deviation or a large fraction of frames more than 0.5 ms from the mean indicates CPU scheduling interference; consider running the experiment under `chrt -r 99` or isolating a CPU core.

What to look for:

```
Estimated refresh rate: 59.940 Hz  (use -hz 59.94 for stream targets)
── Frame intervals ──────────────────────────────────────────────────
  n       : 1796
  target  : 16.683 ms  (measured mean)
  mean    : 16.683 ms
```

```
  SD      :  0.085 ms
  min/max : 16.402 / 18.209 ms
  >0.5 ms :  2 (0.1 %)
  >1.0 ms :  0 (0.0 %)
```

An SD below 0.1 ms and fewer than 1 % of frames more than 0.5 ms from the mean indicate that the display is suitable for frame-accurate stimulus presentation. An SD above 0.5 ms or many outlier frames indicate problems.

Output file: one row per frame: `frame, t_before_ms, t_after_ms, interval_ms`.

`latency` — audio pipeline delay

```
Timing-Tests -test latency
```

When you call `tone.Play()` in goxpyriment, the PCM audio data is placed into an SDL audio stream buffer. The hardware DAC then reads from this buffer and converts the data to an analogue signal. There is a delay — the *pipeline latency* — between the moment `Play()` returns and the moment you would actually hear the first sample.

This test measures that delay *from the software side*, without a microphone or oscilloscope, by doing the following: for each of several nominal tone durations (25, 50, 100, 200, 500 ms), it plays the tone, then polls the stream's `Queued()` count every 0.5 ms until the buffer reports zero bytes remaining. The elapsed time from `Play()` to drain-complete is `drain_ms`. The pipeline latency is `mean(drain_ms) − nominal_ms`.

```
── Drain time for 50 ms tone (latency = mean − target) ──────────
  n       : 10
  target  : 50.000 ms
  mean    : 61.8 ms
  SD      :  0.4 ms
  pipeline latency ≈ 11.8 ms
```

The theoretical minimum latency for a 512-sample buffer at 44 100 Hz is `512 / 44100 × 1000 ≈ 11.6 ms`. If the measured latency is much larger, PulseAudio or PipeWire is adding a mixing buffer on top. You can reduce latency by: - Reducing the `-audio-frames` flag (e.g. `-audio-frames 256`), which sets the SDL hardware buffer to 256 samples (~5.8 ms at 44 100 Hz). Run the test again with this flag to check whether the smaller buffer is stable. - On Linux, bypassing the desktop audio server by setting `SDL_AUDIODRIVER=alsa` before running the program (note: this may conflict with other applications using the sound card).

> **Important caveat:** `drain_ms` measures when the last byte leaves the *SDL software buffer*, not when the last sample exits the speaker cone. The DAC and amplifier add a further constant delay (typically sub- millisecond to a few milliseconds) that is not captured here. To measure the full software-to-acoustic latency you need an oscilloscope and the `av` test.

Output file: `duration_ms, rep, drain_ms, overhead_ms`.

`stream` — RSVP sequential-stimulus timing

```
Timing-Tests -test stream -cycles 120 -frames-on 3 -frames-off 3
```

Many paradigms in cognitive psychology use *rapid serial visual presentation* (RSVP): stimuli presented one after another at a fixed rate, each for a small number of frames. The `display` test tells you whether raw frame flips are timed correctly; the `stream` test tells you whether a *sequence* of stimuli presented in a trial loop is timed correctly.

The test presents `-cycles` elements. Each element is a bright rectangle (`-frames-on` frames at luminance `-level-b`) followed by a dark inter-stimulus interval (`-frames-off` frames at luminance `-level-a`). The target stimulus onset asynchrony (SOA) is:

```
SOA = (frames-on + frames-off) × frame_duration
```

For 3 on + 3 off frames at 60 Hz this is 6 × 16.67 ms = 100 ms.

If a DLP-IO8-G is connected, a trigger pulse is fired at the onset of each bright phase so that the software timestamps can be validated against a photodiode on the oscilloscope.

Two statistics are reported:

- **Duration error** (actual on-duration − target on-duration): a non-zero mean indicates that the on-phase is consistently one frame too long or short, typically due to driver double-buffering; high SD indicates occasional frame drops.
- **SOA error** (actual onset-to-onset interval − target SOA): this is the quantity that matters most for RSVP experiments. If the mean SOA error is zero but the SD is 2 ms, successive words drift in and out of phase with any auditory rhythm you may be synchronising to.

```
── Duration error (target 50.00 ms) ──────────────────────────────
  n       : 110
  mean    :  0.012 ms
  SD      :  0.088 ms
  >0.5 ms :  0 (0.0 %)

── SOA error (target 100.00 ms) ──────────────────────────────────
  n       : 109
  mean    : -0.003 ms
  SD      :  0.094 ms
  >0.5 ms :  0 (0.0 %)
```

A mean close to zero and an SD below 0.1 ms indicate excellent RSVP timing.

Output file: `element, t_onset_ms, t_offset_ms, onset_ns, offset_ns, duration_ms, duration_error_ms, interval_ms, interval_error_ms, trigger`.

`vrr` — Variable Refresh Rate duration sweep

```
Timing-Tests -test vrr -vrr-max-ms 50 -cycles 5
```

Background: why fixed refresh is a problem On a standard monitor running at 60 Hz, every stimulus duration must be a multiple of 16.67 ms. You can show a word for 16.67 ms, 33.33 ms, or 50 ms, but not for 20 ms or 25 ms — the display simply cannot change faster than once per frame. At 120 Hz the quantum shrinks to 8.33 ms, which is better but still significant for subliminal stimulation research where you may want 10 ms, 12 ms, or 15 ms durations.

Variable Refresh Rate (VRR) technology — marketed as AMD FreeSync, NVIDIA G-Sync, or the underlying VESA Adaptive-Sync standard — breaks this quantisation. With VRR, the display holds the current frame for exactly as long as you ask and then refreshes when you tell it to. The result: stimulus durations are controlled by your software timer, not the display clock.

How goxpyriment uses VRR goxpyriment normally calls `SDL_RenderPresent` with VSync enabled (vsync=1), which blocks until the next fixed VSYNC edge. The `vrr` test switches the renderer to vsync=0 before the timing loop:

```
exp.Screen.SetVSync(0)   // Present() returns immediately; no VSYNC block
```

With vsync=0 on a VRR display: 1. `Present()` returns as soon as the GPU accepts the frame — typically within a fraction of a millisecond. 2. The display holds that frame until the *next* `Present()` call. 3. The gap between two `Present()` calls is therefore the displayed stimulus duration.

The test controls that gap with the `sleepUntil()` busy-wait helper, which spins for the last 500 µs to keep duration error below 0.5 ms (one standard deviation). For each target duration D (1 ms, 2 ms, … up to `-vrr-max-ms`), it repeats `-cycles` trials:

```
draw bright screen → Present() → record onsetNS
sleepUntil(now + D)
draw blank screen  → Present() → record offsetNS
actual_ms = (offsetNS − onsetNS) / 1e6
```

`onsetNS` and `offsetNS` are both `sdl.TicksNS()` timestamps captured immediately after each `Present()` returns, so their difference directly measures the software-controlled duration.

What the test tells you **On a VRR-capable display:** duration errors should be small (well below 0.5 ms) across the entire sweep. You can present stimuli for 1 ms, 7 ms, 17 ms, or any other duration that falls within the panel's supported VRR range.

On a non-VRR display: with vsync=0 the display still refreshes at its fixed rate and produces tearing. Duration errors cluster at multiples of the frame period (e.g. ±8 ms at 60 Hz). The test self-diagnoses this: if you see large, periodic duration errors, your monitor does not support VRR or VRR is not enabled in the OS display settings.

VRR range limits: every VRR panel has a supported refresh-rate range, for example 48–144 Hz (frame durations 6.9–20.8 ms). Requesting a duration shorter than 1000/max_Hz or longer than 1000/min_Hz takes the panel outside its VRR window. Outside this range the display reverts to fixed-rate behaviour and duration errors increase sharply. The boundary is clearly visible in a plot of duration error vs. target duration — it is a direct, empirical characterisation of your specific panel's VRR window.

Typical output on a VRR-capable 144 Hz monitor (FreeSync range 48–144 Hz):

```
  1 ms  rep 0:  actual=  1.021 ms  error= +0.021 ms
  1 ms  rep 1:  actual=  1.018 ms  error= +0.018 ms
── 1 ms: mean=+0.019 ms  SD=0.003 ms
  ...
  7 ms  rep 0:  actual=  7.022 ms  error= +0.022 ms
── 7 ms: mean=+0.021 ms  SD=0.004 ms
  ...
 21 ms  rep 0:  actual= 20.854 ms  error= -0.146 ms   ← just outside VRR range
 21 ms  rep 1:  actual= 33.321 ms  error=+12.321 ms   ← snapped to 2 frames
── 21 ms: mean=+6.09 ms  SD=6.21 ms                  ← VRR window ends here
```

The sharp increase in error at 21 ms (one frame below the 48 Hz lower limit of 20.8 ms) pinpoints the VRR boundary exactly.

Important caveats **Software vs. photon timing.** `onsetNS` / `offsetNS` are captured right after `Present()` returns, not at photon emission. The full software-to-photon latency (GPU pipeline + panel scan-out + pixel rise time) is a constant offset that can be measured independently with the `frames` test and a photodiode. Because this latency is constant across trials, it does not affect duration *accuracy* — only the absolute onset *time*.

Tearing on non-VRR displays. With vsync=0 on a non-VRR display, the GPU writes to the framebuffer while the display is reading it, producing a horizontal tear line. This is harmless for the measurement (the photodiode still sees the luminance change) but should not be used in actual experiments on non-VRR monitors.

VRR must be enabled in the OS. On Linux, enable FreeSync in the DRM/KMS layer or in your compositor settings. On Windows, enable "Variable refresh rate" in Display Settings → Advanced display. The application does not need to do anything special beyond disabling vsync — the OS/driver handles the rest.

Enable VRR in the OS before running this test. goxpyriment itself only needs `SetVSync(0)`; the VRR handshake between GPU, driver, and panel is entirely managed by the OS. If the test reports large duration errors despite a VRR-capable monitor, check that VRR is enabled in your system settings.

```
# Check if your display supports VRR (look for "vrr_capable: 1")
cat /sys/class/drm/card*/card*-*/vrr_capable
```

```
# Enable VRR for a connector (e.g. HDMI-1) via sysfs (requires root):
echo 1 | sudo tee /sys/class/drm/card*/card*-HDMI-A-1/vrr_enabled

# Or enable it permanently via KMS option in /etc/modprobe.d/ (amdgpu):
echo "options amdgpu freesync_video=1" | sudo tee /etc/modprobe.d/amdgpu.conf
```

Enabling and verifying VRR on Linux On X11 with an AMD GPU, FreeSync can also be enabled per-screen in `xorg.conf`:

```
Option "VariableRefresh" "true"
```

Output file: `target_ms, rep, actual_ms, duration_error_ms, onset_ns, offset_ns, trigger`.

Tier 2 — Trigger device characterisation

`trigger` — DLP-IO8-G precision

```
Timing-Tests -test trigger -period-ms 100 -duty 50 -duration-s 30 -trigger-pin 1
```

The DLP-IO8-G communicates over a USB-CDC virtual serial port at 115 200 baud. Each `SetHigh` / `SetLow` command is a single byte written to the port. The round-trip latency — the time between writing the byte and the pin actually changing state — is determined by the USB host controller's polling interval (nominally 1 ms on full-speed USB).

Before trusting trigger pulses in `frames`, `tones`, `av`, or `rt`, use this test to characterise how precise your DLP-IO8-G is in isolation. The test drives a square wave on pin `-trigger-pin` for `-duration-s` seconds and records the jitter of each rising and falling edge against the ideal target time.

Typical result on Linux with `latency_timer` reduced to 1 ms:

```
── Rising-edge jitter (ms from target) ──────────────────────────
  mean    :  0.831 ms
  SD      :  0.294 ms
  min/max :  0.201 / 1.842 ms
```

If the mean jitter exceeds 3 ms or the SD exceeds 1 ms, reduce the USB latency timer:

```
echo 1 | sudo tee /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
```

Replace `ttyUSB0` with your actual port. The change reverts on device unplug; add a udev rule to make it permanent.

Output file: `cycle, edge, target_ms, actual_ms, jitter_ms`.

Tier 3 — Stimulus timing validation

The tests in this tier require a photodiode taped to your screen and an oscilloscope. They answer the question: *does the monitor actually show what the software reports?*

`frames` — visual onset and phase duration (alias: `flash`)

```
# Multi-frame alternation (e.g. 2 bright + 2 dark at 60 Hz → 33.3 ms phases):
Timing-Tests -test frames -frames-on 2 -frames-off 2 -cycles 120

# Single-frame flashes (minimum-duration test, formerly "flash"):
Timing-Tests -test frames -frames-on 1 -frames-off 60 -cycles 60
```

This test alternates between a bright phase (`-frames-on` frames at luminance `-level-b`) and a dark phase (`-frames-off` frames at luminance `-level-a`). A trigger pulse is sent at the onset of every bright phase.

Durations are specified in **frames**, not milliseconds, so no `-hz` estimate is needed. The measured mean is used as the reference for deviation statistics.

Two measurements are reported per run:

- **Bright-phase duration** — time from the first bright flip to the first dark flip. Should equal `frames-on × frame_interval`. This is what a photodiode measures as pulse width.
- **Period** — time from one bright onset to the next. Should equal `(frames-on + frames-off) × frame_interval`.

On the oscilloscope: - Channel 1 (photodiode): square wave. Rising-edge width = bright-phase duration; full period = frames-on + frames-off frame intervals. - Channel 2 (trigger): should align with the rising edge of the photodiode. The gap between them is the **display input lag** — time from software flip to first photon. Typically one to two frames on LCD monitors without “gaming” mode.

Single-frame capability (`-frames-on 1`): the photodiode pulse width should equal one frame duration (~16.67 ms at 60 Hz). A pulse that is two frames wide (~33 ms) indicates triple-buffering or compositor interference; single-frame stimuli are then not achievable without driver changes.

From the CSV alone (no oscilloscope): check that `bright_duration_ms` and `period_ms` match the expected frame multiples and that their SD is low.

Output file: `cycle, t_before_ms, t_after_ms, bright_duration_ms, period_ms`.

`tones` — audio onset jitter over a long session

```
Timing-Tests -test tones -cycles 300 -freq-hz 1000 -tone-ms 50 -iti-ms 450
```

This test plays a long sequence of identical sine tones and measures, for each tone, the error between the actual and target onset time. Run it for at least 300 tones (~2.5

minutes at 500 ms SOA) to reveal cumulative drift and scheduling outliers.

Onset error (`actual_onset − target_onset`): the target is `i × SOA`, where `SOA = tone_ms + iti_ms`. A growing onset error indicates that the audio clock is drifting relative to the system wall clock. A drift of a few milliseconds per minute is typical and acceptable for most purposes; a drift of tens of milliseconds per minute indicates a problem.

Inter-onset interval (IOI) (`actual_onset[i] − actual_onset[i-1]`): this should equal the SOA with low variance. High IOI variance indicates OS audio scheduling issues.

If a DLP-IO8-G is connected, a trigger is fired just before each `Play()` call. Connect the trigger to oscilloscope channel 2 and the audio line-out to channel 1: the gap between the trigger edge and the acoustic onset is the **software-to-acoustic latency** for that tone. The mean of this gap across 300 trials is the audio latency; the SD is the trial-to-trial jitter.

> `actual_onset_ms` is when `Play()` was called — when PCM data entered the SDL buffer — not when sound left the speaker. The oscilloscope measures the true acoustic onset. The two differ by the pipeline latency measured in the `latency` test.

Output file: `tone_num, target_onset_ms, actual_onset_ms, onset_error_ms, actual_offset_ms, ioi_ms, ioi_error_ms, trigger_sent`.

---

av — audio–visual synchrony

```
Timing-Tests -test av -soa-ms 0 -freq-hz 1000 -tone-ms 50 -iti-ms 1000 -cycles 30
```

This test presents pairs of audio and visual stimuli with a controlled *stimulus onset asynchrony* (SOA). Set `-soa-ms 0` to present them simultaneously (minimum software SOA); positive values delay the audio; negative values delay the visual.

Software-only (no oscilloscope): the program records `t_visual_after_ms` (when `RenderPresent` returned) and `t_audio_queued_ms` (when `Play()` was called). The actual software SOA = `t_audio_queued_ms − t_visual_after_ms`. This tells you what the software *thinks* it did, not what the hardware actually produced.

With oscilloscope (photodiode on channel 1, audio line-out on channel 2): the actual acoustic-visual delay = `t_audio_channel − t_photodiode_channel`. This is the quantity that matters for perceptual experiments. The difference between the software SOA and the oscilloscope-measured SOA is dominated by the audio pipeline latency (measured separately by the `latency` test).

To achieve a specific *perceptual* SOA (e.g. simultaneous percept), set `-soa-ms` to compensate for the difference: if the audio arrives 12 ms after the light even when `soa-ms=0`, set `soa-ms=-12` to delay the visual by 12 ms so that the light and sound arrive at the ear/eye simultaneously.

Output file: `trial, t_visual_before_ms, t_visual_after_ms, t_audio_queued_ms, soa_intended_ms, soa_actual_ms`.

Tier 4 — Response timing

`rt` — reaction-time timestamp precision

```
Timing-Tests -test rt -cycles 60 -iti-ms 1000
```

This test measures the precision of reaction-time measurement itself. Each trial: a white screen flashes for one frame; the participant presses any key as quickly as possible. The key RT is computed as:

`RT = key_event_timestamp_ns − screen_flip_timestamp_ns`

Both timestamps come from the same SDL3 nanosecond hardware clock (`SDL_GetTicksNS`). The screen flip timestamp is captured immediately after `SDL_RenderPresent` returns; the key event timestamp is the hardware interrupt time recorded by the OS keyboard driver. Because both are on the same clock, no polling latency affects the RT on the response side.

The remaining sources of error are: - **Display input lag** (measured by `frames`): the time from the software flip to the first photon. This adds a constant offset to all RTs. - **OS keyboard scheduling latency** (typically < 2 ms on Linux): the time from the physical key depression to the SDL event arriving in the queue.

For the most accurate RT measurement: 1. Use the `frames` test to measure your display's input lag. 2. Subtract that lag from all RTs in your experiment. 3. If you use a USB response box, its internal timestamping may be more precise than the OS keyboard driver; compare it to the SDL event timestamp to characterise the pipeline delay.

Output file: `trial, onset_ns, event_ts_ns, rt_ns, rt_ms`.

Interpreting results: what is "good"?

Metric	Excellent	Acceptable	Problematic
Frame-interval SD (`display`)	< 0.1 ms	< 0.5 ms	> 1 ms
Frames > 0.5 ms late (`display`)	< 1 %	< 5 %	> 10 %
Audio pipeline latency (`latency`)	< 15 ms	< 30 ms	> 50 ms

Metric	Excellent	Acceptable	Problematic
Audio onset SD (`tones`)	< 0.5 ms	< 2 ms	> 5 ms
RSVP SOA SD (`stream`)	< 0.2 ms	< 1 ms	> 2 ms
VRR duration error SD (`vrr`)	< 0.1 ms	< 0.5 ms	> 1 ms (or periodic → no VRR)
Trigger jitter SD (`trigger`)	< 0.3 ms	< 1 ms	> 2 ms
RT SD (`rt`)	< 3 ms	< 10 ms	> 20 ms

Loading data in Python

All tests write a `.csv` file to `~/goxpy_data/` with #-prefixed metadata headers. Load any run in Python with:

```
import pandas as pd

df = pd.read_csv("~/goxpy_data/Timing-Tests_000_*.csv", comment="#")
print(df.head())
```

For the `stream` test, onset jitter and SOA error are directly in the `duration_error_ms` and `interval_error_ms` columns. For `tones`, see `onset_error_ms` and `ioi_ms`. For `rt`, see `rt_ms`.

Improving timing on your system

Linux - Disable the desktop compositor: use a plain window manager (i3, openbox) or start the experiment from a virtual terminal (`Ctrl+Alt+F2`) and disable the window system (`systemctl stop gdm`) . - Run the experiment process with real-time scheduling: `bash sudo chrt -r 99 Timing-Tests -test display -duration-s 30` - Reduce USB trigger latency (see DLP-IO8-G section above). - Disable CPU frequency scaling: `cpupower frequency-set -g performance`.

macOS: - The macOS WindowServer compositor is always active; expect 1–3 ms frame jitter and one frame of fixed display latency. - Always run in fullscreen mode.

Windows: - Run in fullscreen exclusive mode to bypass DWM composition. - Disable “Hardware-Accelerated GPU Scheduling” in Display Settings if you observe high frame jitter.

Audio buffer size

The hardware audio buffer size is controlled by the SDL hint `SDL_AUDIO_DEVICE_SAMPLE_FRAMES`, exposed via the `-audio-frames` flag. It must be set before the audio device opens.

```
# Default (platform-dependent, often 512–2048 samples):
Timing-Tests -test latency

# Aggressive low-latency (~5.8 ms at 44100 Hz):
Timing-Tests -test latency -audio-frames 256 -drain-reps 20

# Conservative (~46 ms, stable on any system):
Timing-Tests -test latency -audio-frames 2048
```

On startup the program prints the actual device format:

```
audio: 44100 Hz  1 ch  256 sample frames (~5.8 ms latency)
```

Use the `latency` test at several buffer sizes to find the smallest value that gives a stable drain time (low SD). Then set that buffer size for all subsequent tests and for your actual experiment.

DLP-IO8

See https://github.com/chrplr/dlp-io8-g

Parallel port alternative

If you have a parallel port (LPT) at `/dev/parport0`, use `triggers.NewParallelPort("/dev/parport0"` in your experiment code. The `Send(byte)` method sets all 8 data lines simultaneously.
Linux Prerequisites: `sudo modprobe ppdev` and membership in the `lp` group