17 — Triggers (Trigger Studio)

> **TraxProbe & Traxcope User Manual** · Version 1.0.0 · Updated 2026-08-25
 · https://embedya.com

Triggers bring oscilloscope-style capture discipline to trace data: define a condition (or a logic combination of conditions), arm it, and when it fires the timeline freezes centered on the event. Triggers are evaluated **entirely on the host** — the firmware needs no extra code; anything it already traces can be a trigger source.

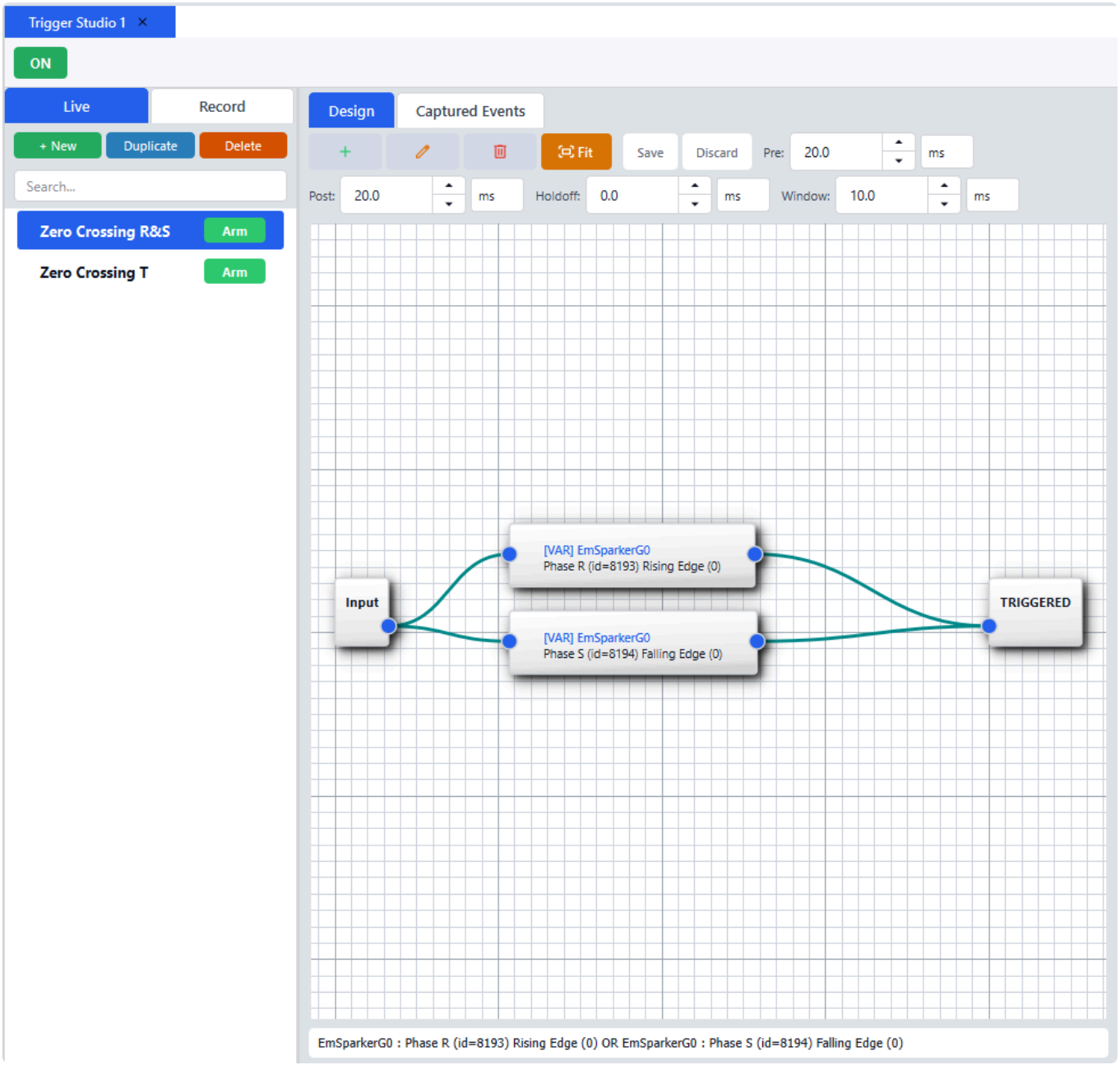
Triggers work in two contexts:

- **Live** — watch the incoming stream and fire in real time ("freeze when the error state is entered while the ISR is active").
- **Review (record scan)** — scan a loaded recording for every place the condition was true ("find all occurrences in last night's 8-hour capture").

The Trigger Studio

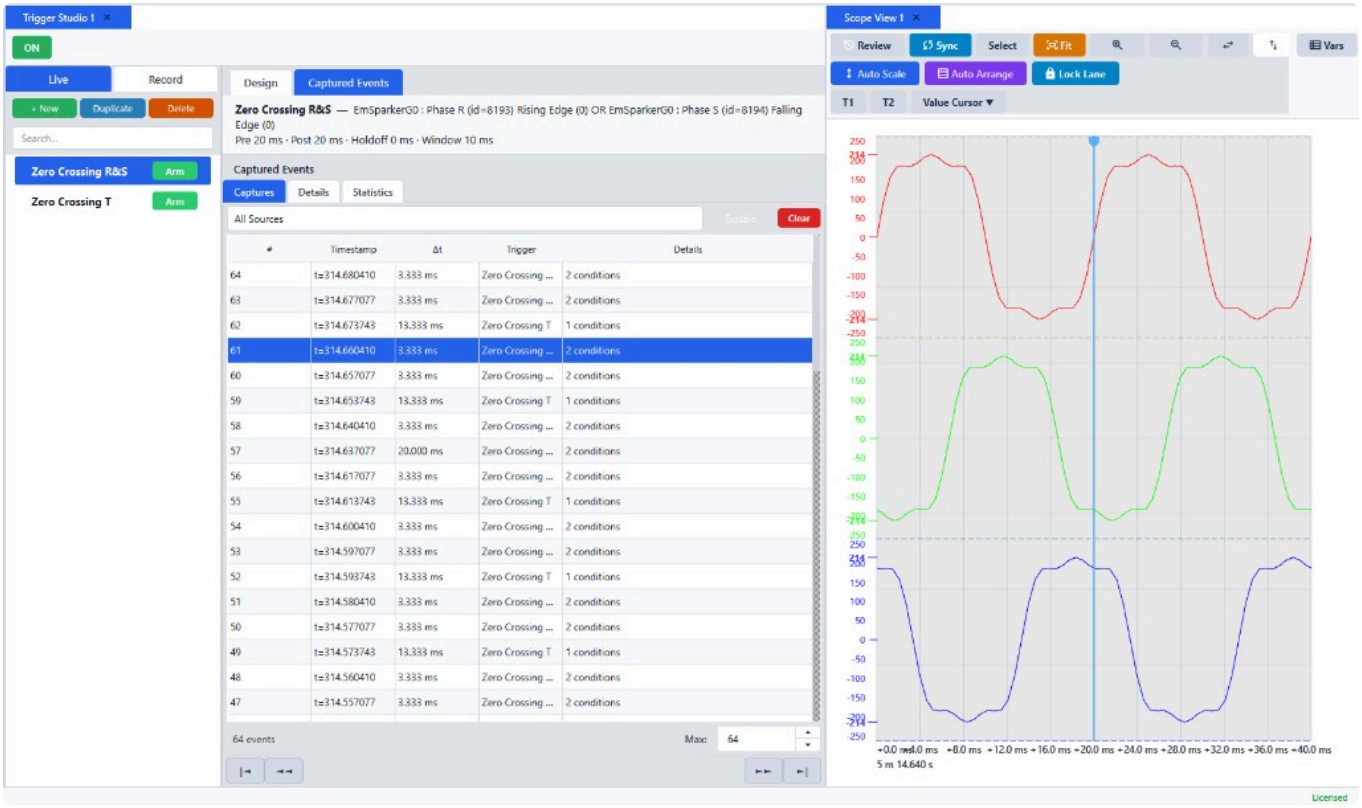
Open via Toolbox → **Triggers**. The trigger list sits on the left; the center area is a two-tab workspace that follows the workflow:

Area	Purpose
Trigger list	Your trigger configurations: create, duplicate, delete, select the active one
Design tab	Build the trigger: condition nodes wired in series (AND) / parallel (OR) between Input and TRIGGERED
Captured Events tab	History of firings — each entry records when it fired and which conditions were active



Screenshot: Trigger Studio — list on the left, Design tab with the node-graph editor filling the center

The tabs switch automatically with the workflow: arming a trigger (or starting a Record-mode scan) jumps to **Captured Events**, and selecting or creating a trigger returns to **Design**. Events that fire while you are on the Design tab don't interrupt you — they accumulate as a badge count on the Captured Events tab. A summary strip above the events table keeps the armed trigger's formula and Pre/Post/Holdoff/Window timing visible while the editor is hidden.



Screenshot: Captured Events tab — summary strip on top, capture history with per-event details, synced scope view frozen on a capture

A **master enable** toggle arms/disarms the whole trigger system.

Conditions — what can fire a trigger

A trigger is built from one or more conditions, each tied to a source (and optionally a specific probe):

Source	Condition types
Variable	Rising / falling / either edge through a threshold; level above/below; inside/outside a window
Task	A specific kernel event (e.g. switch-in), or "task is running" as a continuous state
ISR	ISR enter/exit event, or "ISR is active" as a state
Log	Message matches a regex pattern, with a minimum severity
State machine	State entered / exited, a specific from→to transition (catch illegal edges!), or "in state X" as a state
Marker	Interval opened / closed; "inside the interval" as a state; duration above or below a threshold (µs) — see below
Timing	No activity for longer than a timeout (watchdog-style)

Source	Condition types
DSP	Frequency-domain conditions on a stream variable: magnitude at a frequency bin, band energy, dominant-frequency shift, spectral mask violation, THD, Nth-harmonic level

Conditions come in two natures, which you can mix freely:

- **Events** — momentary (an edge, a log match, a marker start/stop, a duration verdict). Latched for one evaluation cycle.
- **States** — continuous booleans (level above threshold, task running, in state X, **in marker**).

Marker conditions

A **Marker** source watches the START/STOP intervals from Chapter 15. The firmware already emits those frames; Trigger Studio just evaluates them on the host, so a deadline overrun can freeze the timeline the instant it happens — you do not have to hunt for it later in Marker Stats.

Pick a specific marker or **Any marker**, then a type:

Type	Nature	Fires when
Marker Start	Event	The interval opens (`TRAX_MARKER_START`)
Marker Stop	Event	The interval closes (`TRAX_MARKER_STOP`)
In Marker	State	True for the whole time between a start and its matching stop
Duration Above	Event	The matching stop arrives and elapsed time is **longer than** the threshold
Duration Below	Event	The matching stop arrives and elapsed time is **faster than** the threshold

Duration is measured in **microseconds**, start-to-matching-stop, on the same marker and core. Nested uses of the same TID pair with a stack (same rule as Marker Stats). The duration field is pre-filled from the marker's configured deadline (`TRAX_MARKER_DEFINE` / the requirements editor) so the common "did this overrun its budget?" trigger needs no typing; you can still override the number. **Any marker** leaves the field empty — there is no single deadline to copy.

A stop with no matching start (the capture began mid-interval, or the start was lost) has no measurable duration, so Duration Above / Below do not fire on it. The evaluator keeps pairing start/stop even while the trigger is disarmed, so an interval that opened before you armed still yields a duration when it closes.

Typical recipes:

```
ControlLoop duration > 600 µs
In ControlLoop  AND  motor_current > 10 A
ControlLoop stop  AND  log matches "overrun"
```

The first is a live deadline police. The second is the compound-bug form: "only while that interval is open". Marker Stats still answers the statistical question ("does it *always* finish in time?"); the trigger answers the forensic one ("freeze the first time it does not").

Combining conditions — the node graph

Conditions are nodes wired between the **Input** and **TRIGGERED** endpoints, and the AND/OR logic comes from the wiring topology itself — like contacts in a relay ladder:

- **Series** — conditions chained one after another along a path form an **AND**: every condition on the path must be true.
- **Parallel** — separate paths from Input to TRIGGERED form an **OR**: any complete path fires the trigger.

There are no explicit NOT/NAND/NOR gates. Inverse logic is expressed *inside* the condition node by picking the opposite condition type: greater vs. smaller than a threshold, inside vs. outside a window, rising vs. falling edge, state entered vs. exited, and so on.

The editor renders the graph and, below it, the equivalent readable formula. Examples:

```
ERROR_state_entered AND motor_current > 10 A
(log matches "overrun" AND motor_current > 10 A) OR queue_send_failed
THD(grid_R) > 5%  AND  acquisition_task_running
ControlLoop duration > 600 µs
In ControlLoop AND ISR_ADC is active
```

This is the tool for compound bugs — "only when X happens *while* Y is true" — that are nearly impossible to catch by staring at live data.

Timing parameters — Pre, Post, Holdoff, Window

The Design toolbar carries four timing values (each with a selectable ns/µs/ms/s unit):

Parameter	Meaning
Pre	How much data *before* the trigger instant is kept in the capture. This is the "what led up to it" window — the trigger fires at t, the capture spans back to t − Pre.

Parameter	Meaning
Post	How much data *after* the trigger instant is captured before the capture is finalized. The full capture window is t − Pre … t + Post.
Holdoff	Minimum quiet time after a firing before the trigger re-arms. Use it to suppress bursts: a bouncing condition fires once per holdoff period instead of flooding the event list. 0 = re-arm immediately.
Window	The AND **correlation window** — see below.

The Window parameter

Series (AND) wiring raises a question the graph alone can't answer: what does it mean for two *momentary* events to happen "at the same time"? An edge on Phase R and an edge on Phase S never land on exactly the same timestamp — they come from different sources, are sampled at different instants, and arrive in different packets.

Window is the answer: all conditions feeding an AND must occur within this much time *of each other* to count as simultaneous. With `Window = 10 ms`, an R rising edge at t = 100.000 s and an S falling edge at t = 100.004 s satisfy the AND (4 ms apart); the same pair 15 ms apart does not.

Two practical rules:

- **Never leave Window at 0 for edge/event ANDs.** Zero means "perfectly simultaneous", which momentary events essentially never are — the trigger will look armed but never fire. A small non-zero value (e.g. 10 ms) is the right default.
- **Window is irrelevant for pure state ANDs.** Continuous states (level above threshold, task running, in state X, **in marker**) overlap in time naturally, so the AND is satisfied whenever they are all true at once — the correlation window only matters when at least one input is a momentary event.

The firing cycle

An armed trigger runs continuously: **Idle** → **Armed** → **Triggered** → **(holdoff)** → **Armed...**

Each firing produces a capture and the trigger re-arms itself automatically — with a non-zero **Holdoff** enforcing the dead-time between consecutive firings. Captures accumulate in the Captured Events tab (its **Max** field caps how many rows are kept); if the capture-history storage fills up, all triggers disarm automatically. Click **Disarm** on the trigger to stop it manually at any time.

What happens on a firing

- Every firing is appended to the **Captured Events tab** with its timestamp, the time between consecutive firings (Δt), and the set of conditions that were true. Live views keep streaming — nothing freezes or interrupts your monitoring.
- **Double-click a captured event** to navigate every synced view (Scope, Trace, Event, …) to that event's time range (t − Pre … t + Post). Enter on a selected row does the same, and the first/prev/next/last buttons step through the captures in order.

Scanning recordings (Review context)

Switch the studio to **Review** to run the same trigger over recorded data:

1. Set the scan **start/end time** (or scan the whole recording).
2. Click **Scan** — the engine replays the recording through the condition evaluator; a progress bar tracks it, **Pause/Resume/Stop** are available.
3. Every match lands in the Captured Events tab; click through them to inspect each occurrence with all synced views.

This turns an overnight capture into a query-able dataset: "show me every time the queue overflowed while ACQ was in PROCESS".

Tips

- Scope conditions to a specific probe in multi-probe setups, or leave "any probe" deliberately for fleet-wide conditions.
- For noisy analog signals prefer **window** conditions or DSP band-energy over a raw edge — fewer false firings.
- The SM **from→to transition** condition is the cheapest way to permanently police illegal state transitions during development: arm it and forget it — it re-arms after every firing.
- A marker **Duration Above** pre-filled from the deadline is the same idea for timing budgets: arm it, and the first overrun lands in Captured Events with Pre/Post context. Pair **In Marker** with a variable edge when the bug only happens *inside* that interval.
- Trigger configurations are part of the workspace — your "catch the glitch" setup survives restarts.