12 — Stream Variables & Spectrum Analysis

> **TraxProbe & Traxcope User Manual** · Version 1.0.0 · Updated 2026-08-25

Variable **streams** are for fixed-rate sampled data: ADC scans, software signal generators, SDR/FPGA baseband. Instead of timestamping every sample (as async VARs do), a stream is timestamped **once at start**; every data frame then carries only a sequence index and raw samples. Traxcope reconstructs each sample's time as `start_time + index × sample_period`. The result: multi-channel kHz–MHz data at a fraction of the wire cost, with exact, jitter-free timing — and the data quality needed for FFT analysis in the Spectrum Analyzer.

On the device

1. Declare capacity

```
/* trax_config.h — must cover the number of stream TIDs you define */
#define TRAX_CFG_STREAM_CNT  2
```

2. Define channels and the stream group

Each stream channel is a regular VAR (its `TRAX_VAR_DEFINE` data type fixes the per-sample byte size). The group definition binds channels to the stream and declares the sample rate. Example — six analog sensors (current shunt, bus voltage, pressure, setpoint pot, NTC, LDR) scanned by one ADC sequencer at 1 kHz, timer-triggered and streamed via DMA:

```
#define SAMPLE_RATE_HZ    1000u
#define ADC_CLK_HZ        16000000u

/* Per-channel physical sampling skew: the sequencer converts the channels
 * back-to-back, 25 ADC clocks each, so channel N is sampled
 * N × 25 / ADC_CLK_HZ seconds after channel 0 (1.5625 µs per step). */
#define CH_OFFSET_S(_n)   ((double)(_n) * 25.0 / (double)ADC_CLK_HZ)

/* Per-channel VAR metadata + engineering conversion + selectable host filter
*/
TRAX_VAR_DEFINE(VAR_TID_ADC_CURRENT, "A0 Current", TRAX_DATA_TYPE_UINT16, 0,
    4095, "u", TRAX_COLOR_ORANGE, TRAX_PLOT_LINE);
```

```
TRAX_VAR_FORMULA(VAR_TID_ADC_CURRENT, "ADC to Volt", "V", "(x * 3300.0) /
4095.0");
TRAX_VAR_FILTER (VAR_TID_ADC_CURRENT, "LPF 100 Hz", 2,
                 (0.24524f, 0.24524f), 1, (-0.50952f));
/* ... same pattern for the other five channels ... */

/* The stream group: TID, description, rate (num/denom Hz), byte order,
channels */
TRAX_STREAM_ADC_GROUP_DEFINE(STREAM_TID_ADC, "ADC 1 kHz 6-ch",
    SAMPLE_RATE_HZ, 1, TRAX_BYTE_ORDER_LITTLE,
    TRAX_STREAM_VAR(VAR_TID_ADC_CURRENT,  CH_OFFSET_S(0)),  /* reference   */
    TRAX_STREAM_VAR(VAR_TID_ADC_VBUS,     CH_OFFSET_S(1)),  /* +1.5625 µs  */
    TRAX_STREAM_VAR(VAR_TID_ADC_PRESSURE, CH_OFFSET_S(2)),  /* +3.1250 µs  */
    TRAX_STREAM_VAR(VAR_TID_ADC_SETPOINT, CH_OFFSET_S(3)),  /* +4.6875 µs  */
    TRAX_STREAM_VAR(VAR_TID_ADC_NTC,      CH_OFFSET_S(4)),  /* +6.2500 µs  */
    TRAX_STREAM_VAR(VAR_TID_ADC_LDR,      CH_OFFSET_S(5))   /* +7.8125 µs  */
);
```

Notes:

- **Sample rate** is a rational number (`freq_num / freq_denom`) so non-integer rates are exact.
- Data on the wire is **interleaved by sequence**: `[ch0, ch1, ch2, ch0, ch1, ch2, ...]`. One "sequence" = one sample of every channel.
- Besides the generic ADC group there is a specialized **rotor stream** kind, `TRAX_STREAM_ROTOR_DEFINE` : a single-channel position stream that additionally declares the encoder geometry (`counts_per_rev`). It uses the same START/UPDATE/SEND runtime and is the required input of the Rotary Motor Monitor (Chapter 19).

> **Why a rational and not a float?** The precision problem isn't representing the frequency — a 64-bit double holds 10 GHz with ~16 significant digits. The problem is reconstructing the timestamp of sample *k* as `k / f` : in floating point, that division carries a relative error of ~10^{-16} of the *total elapsed time*, so the placement error grows with capture length — after an hour it is already ~0.4 ns, several sample periods at 10 GS/s. The rational form gives Traxcope the exact identity " `freq_num` samples = `freq_denom` seconds", so the integer-seconds part of every timestamp is computed in exact 64-bit arithmetic and only the sub-second remainder uses floating point. The error stays below femtoseconds and **never accumulates**, from kHz audio streams to multi-GHz declared rates. It also preserves intent for non-decimal rates: a 25 MHz clock divided by 768 is exactly `25000000, 768` — as a float you'd type a pre-rounded `32552.0833…` and the true ratio would be lost.

Channel time offsets

`TRAX_STREAM_VAR`'s second argument declares **when each channel is physically sampled**, in seconds, relative to the sequence's nominal timestamp (`start_time + k × sample_period`). All samples of one sequence travel together on the wire, but in reality they are rarely taken at the same instant — the offset lets Traxcope put every channel back at its true position on the timeline.

How it works:

- **Each offset is absolute from the sequence timestamp** — not a delta from the previous channel. Channel 0 is usually `0.0`, but it may be non-zero too: the stream start timestamp is captured when the hardware is armed (`TRAX_STREAM_START`), and a real ADC needs trigger latency plus sample-and-hold and conversion time before the first result exists. Give channel 0 that acquisition-start latency and the other channels the same latency plus their scan skew.
- The offset is a **pure metadata property**: the firmware buffer is untouched, no per-channel arithmetic runs on the device. Traxcope shifts each channel's timeline at display time, so channel *n*'s sample with sequence index *k* is plotted at `start_time + k × sample_period + offset_n`.
- The offset is **independent of the stream sample rate**. It is fixed by the acquisition hardware (ADC conversion sequence, multiplexer settling, FPGA pipeline) or by the physics of the signal — so it stays correct even if you later retune the stream rate.
- It must be a **constant expression** evaluable at file scope (literals and macro arithmetic are fine), because it is consumed by the static group descriptor.

Typical uses:

- **Sequential ADC scan** — the example above: one ADC converts the channels back-to-back, so each channel lags channel 0 by its position in the scan (`CH_OFFSET_S(N)`). If the scan itself starts a known latency after the trigger (S/H settling, startup delay), fold that into every offset: `#define CH_OFFSET_S(N) (T_START_S + (double)(N) * 25.0 / ADC_CLK_HZ)`.
- **Known phase relationship** — a 3-phase grid generator can store the *same* reference sine in all three channels and declare the 120°/240° electrical lag purely in metadata (`PHASE_OFFSET(_deg) = (_deg) / 360.0 / 50.0` → +6.667 ms / +13.333 ms); the Scope shows a correct three-phase picture with zero per-phase computation on the MCU.
- **Simultaneously-sampled sources** (dual-ADC I/Q, simultaneous-sampling ADCs) with negligible start latency — simply give every channel offset `0.0`.

3. Start the stream, ship the blocks

The CPU never fills the buffer — the DMA writes samples in the background; the firmware only *registers* each completed block (UPDATE) and *ships* it (SEND). Continuing the same example:

the BSP configures timer-triggered ADC scans with double-buffered DMA (2 × 128 sequences here), then a single call arms the hardware *and* anchors the stream timeline:

```
static uint16_t p_buff[NUM_BLOCKS][BLOCK_SAMPLES];   /* DMA double buffer */
static TaskHandle_t stream_send_task_handle;

int stream_adc_init(void)
{
    struct bsp_adc_stream_cfg_t cfg;
    cfg.sample_freq_hz           = SAMPLE_RATE_HZ;
    cfg.p_stream_buffer          = p_buff;
    cfg.stream_buffer_size_bytes = sizeof(p_buff);
    cfg.stream_callback          = adc_stream_callback;   /* DMA HT/TC events
*/

    if (bsp_adc_stream_set_config(BSP_ADC_STREAM_0, &cfg) !=
BSP_ADC_STREAM_OK) {
        return -1;
    }

    xTaskCreate(stream_send_task, "adc_send", STACK_SIZE, NULL,
                SEND_TASK_PRIO, &stream_send_task_handle);

    /* Anchor the timeline and start sampling in ONE atomic step: the 2nd
     * argument is the hardware-start call itself, evaluated inside the same
     * critical section that captures the start timestamp. t = 0 is therefore
     * exactly the instant the timer+ADC+DMA chain begins — no race between
     * "timestamp taken" and "first conversion triggered". Software
     * generators with no hardware to arm pass NULL instead. */
    TRAX_STREAM_START(STREAM_TID_ADC,
        bsp_adc_stream_start(BSP_ADC_STREAM_0));

    return 0;
}

/* DMA half-transfer / transfer-complete callback (IRQ context): keep the ISR
 * minimal — register the just-completed half (UPDATE is ISR-safe and must run
 * at block completion so the sequence index stays honest), then just wake the
 * sender task. */
static void adc_stream_callback(
    const struct bsp_adc_stream_complete_event_t *p_event)
{
    BaseType_t woken = pdFALSE;

    TRAX_STREAM_UPDATE(STREAM_TID_ADC, p_event->p_block, p_event->block_size);
    vTaskNotifyGiveFromISR(stream_send_task_handle, &woken);
```

```
    portYIELD_FROM_ISR(woken);
}

/* Sender task: ships the pending block at task priority, keeping the
 * transport work (and any blocking inside it) out of interrupt context. */
static void stream_send_task(void *arg)
{
    for (;;) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        TRAX_STREAM_SEND_ALL(STREAM_TID_ADC);
    }
}
```

Deferring SEND to a task puts the timing budget in **your** hands: the completed half must be shipped before the DMA wraps and overwrites it. From block completion you have one full block period — here 128 sequences at 1 kHz = 128 ms — to get SEND executed, so size `BLOCK_SEQS` to comfortably cover your worst-case notification and scheduling latency. Both macros are also ISR-safe (critical section + deterministic copy into the TraxProbe ring), so on bare-metal systems, or when the block period is tight, calling `SEND_ALL` directly in the callback is a perfectly valid simpler variant.

Macro	Role
`TRAX_STREAM_START(tid, hw_cb)`	Atomically arm hardware + capture the start timestamp; emits the STREAM_START control frame
`TRAX_STREAM_UPDATE(tid, p, bytes)`	Register a block: stores the pointer and advances the sequence index by `bytes / sequence_size`
`TRAX_STREAM_SEND_ALL(tid)`	Ship the whole pending block (no-op if nothing pending; ISR-race-safe)
`TRAX_STREAM_SEND(tid, max_bytes)`	Ship at most `max_bytes` from the head of the block (floored to whole sequences)
`TRAX_STREAM_STOP(tid, hw_cb)`	Atomically stop hardware + deactivate; emits STREAM_STOP

For software-generated signals (no hardware to arm), pass `NULL` to `TRAX_STREAM_START` and call UPDATE+SEND from a periodic task instead.

4. Missed blocks are honest, not silent

The single contract that keeps the timeline truthful is: **the sequence index must always advance by the number of sequences the hardware actually captured** — whether or not the data is sent. UPDATE does exactly that (it advances the index by `bytes /`

`sequence_size`), so as long as your DMA/capture callback calls UPDATE for every completed block, every sample that *does* reach the host lands at its true time `start_time + index × sample_period`. Sending is a separate, optional step.

This has two powerful consequences:

- **Overruns are honest.** If the producer overruns and a block is never sent, the next UPDATE advances the index past the gap — the host plots a **NaN gap** at the correct place instead of silently compressing time. Your timeline never lies.
- **Bandwidth is your knob, not your limit.** The declared sample rate can be arbitrarily high — 190 GS/s is fine — because nothing obliges you to ship all of it. Keep incrementing the index at the true capture rate, and choose *how often* you send (the send interval) and *how much* you send each time (the send window). Traxcope renders the un-shipped stretches as NaN gaps and runs its processing on each **coherent segment** — a contiguous run of samples with no gap. The one sizing rule: make each shipped window long enough for the DSP you want, e.g. at least one FFT size of contiguous sequences for the Spectrum Analyzer (a 4096-point FFT needs 4096 contiguous samples per channel per window).

So a slow UART link can still carry a stream declared at MHz–GHz rates: you get periodic, fully time-accurate bursts — each one FFT-able — separated by honest gaps, instead of a downsampled or time-compressed lie. Section 5 shows the firmware pattern for this.

5. The sliced pattern — when the source outruns the link

A front-end (FPGA, fast ADC) may capture far more than the transport can carry. Declare the **conceptual** block size in UPDATE, then ship only an affordable slice:

```
/* FPGA "captured" BLOCK_BYTES this period (huge, conceptual);
 * the MCU ships only the first SLICE_BYTES (e.g. 4 KB) of it. */
TRAX_STREAM_UPDATE(STREAM_TID_IQ, p_ring[slot], BLOCK_BYTES);   /* uint64 size
OK */
TRAX_STREAM_SEND  (STREAM_TID_IQ, SLICE_BYTES);
```

The host timeline advances at the true sample rate (a sliced I/Q stream can declare GHz rates this way); the un-shipped remainder renders as a NaN gap until the next slice. **Never use `SEND_ALL` in this mode** — it would read past the real buffer.

In Traxcope

Scope View

Stream channels appear in the Scope channel tree like any variable, but render as continuous, evenly-sampled waveforms. Everything from Chapter 11 applies (cursors, measurements,

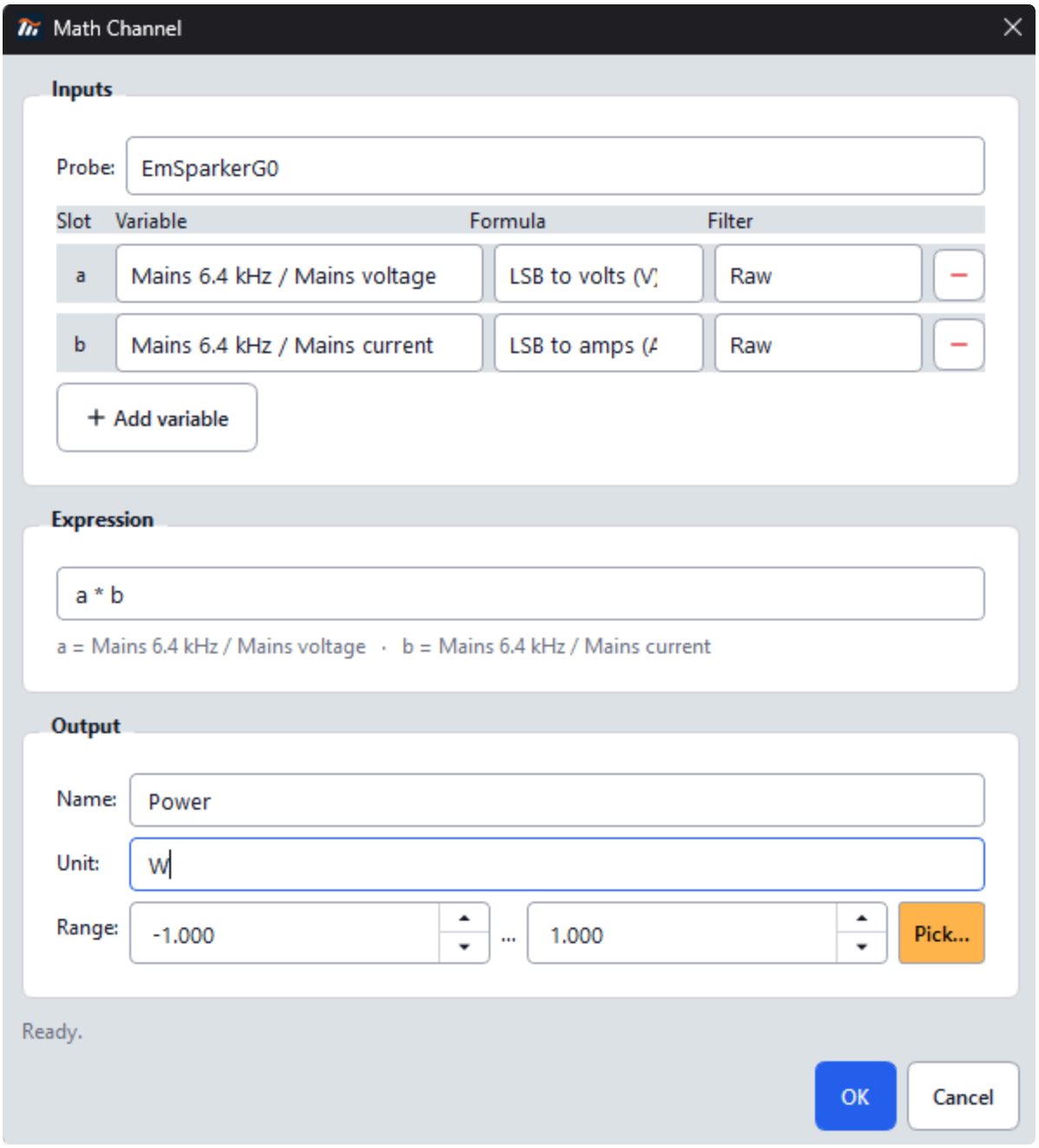
lanes, sync, math channels, power analysis).

Stream vars also have one capability regular VARs don't: **filters**. A discrete-time IIR/FIR filter is only meaningful on fixed-rate data, so the filter dropdown (Chapter 11 §Filters) is exclusive to stream channels — pick a firmware-shipped `TRAX_VAR_FILTER` or one defined in the host-side editor and A/B raw vs filtered live, without reflashing. The same per-channel filters apply in the Spectrum Analyzer below, which is designed around stream vars from the ground up.

Math channels on a stream

Because a stream is interleaved — one sequence = one sample of every channel, reconstructed to the **same timestamp** (plus the optional per-channel offset) — it is the natural input of a Scope **math channel**. Traxcope pairs samples index-by-index and evaluates an expression in slots `a` … `d`. The MCU never computes the product.

The usual example is a voltage/current pair on one stream. Bind each channel with its engineering-unit formula (volts and amps — not raw ADC counts) and write `a * b`:



*Screenshot: same-stream pairing — `a` = Mains voltage (LSB→V), `b` = Mains current (LSB→A), expression `a b`, output **Power** [W]*

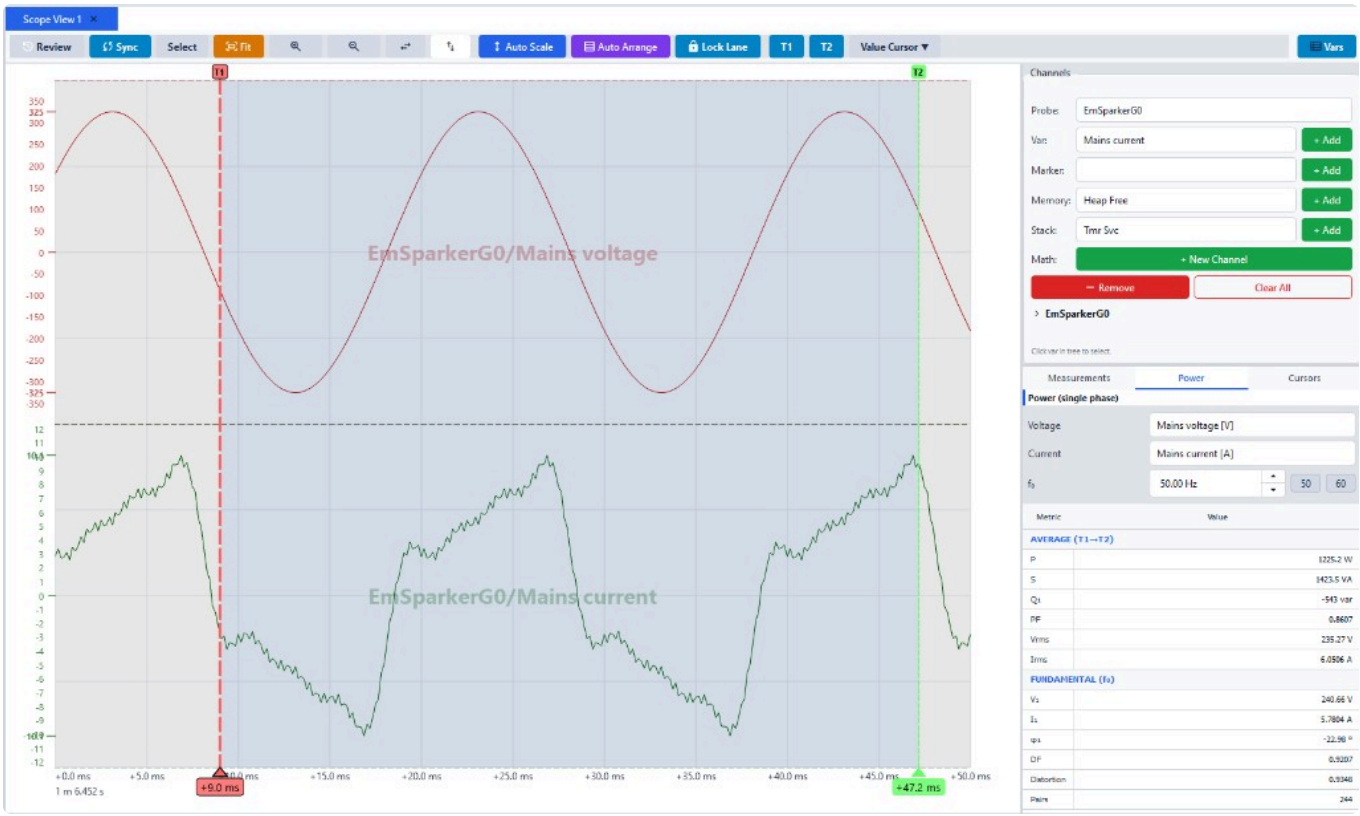
Slot	Stream channel	After its formula
`a`	`Mains voltage`	volts
`b`	`Mains current`	amps
—	`a * b`	instantaneous power [W]

The result is a virtual stream channel: it plots next to the sources, takes T1/T2, and can go into the Spectrum Analyzer (spectrum of instantaneous power, ripple, …). Full recipe, license, and constraints: Chapter 11 — Math channels. Inputs must be channels of **this** stream; mixing two streams is rejected because their samples are not taken together.

A math channel is the **waveform**. For the windowed IEEE-style *numbers* — mean P, apparent S, power factor, and the displacement factor *cos φ₁* — use the Scope sidebar **Power** tab on the same V/I pair. You do not have to build `a * b` first.

Power analysis (V/I stream pair)

Open the Scope's **Power** tab, pick the voltage and current channels of the stream, set **f₀** (50 / 60 Hz shortcuts), switch to Review and arm T1/T2 around at least one line cycle. Traxcope pairs V and I by timestamp (same-stream samples lock 1:1) and reports:



Screenshot: 6.4 kHz mains stream — clean voltage, distorted current, T1→T2 over ~2 cycles at 50 Hz. P / S / PF in the average group; V_1 / I_1 / φ_1 / DF in the fundamental group. ***Math*** *→* ***+ New Channel*** *(top of the sidebar) is the same V/I pair turned into an instantaneous-power waveform.*

- **Average** — P = mean(*v* · *i*), S = Vrms · Irms, **PF** = P/S, plus Vrms / Irms. No FFT.
- **Fundamental** — V_1, I_1, φ_1, **DF** = cos(φ_1) (displacement factor), Q_1, and Distortion = PF/DF, from a single-bin Fourier coefficient at the declared f_0 — not the Analyzer FFT.

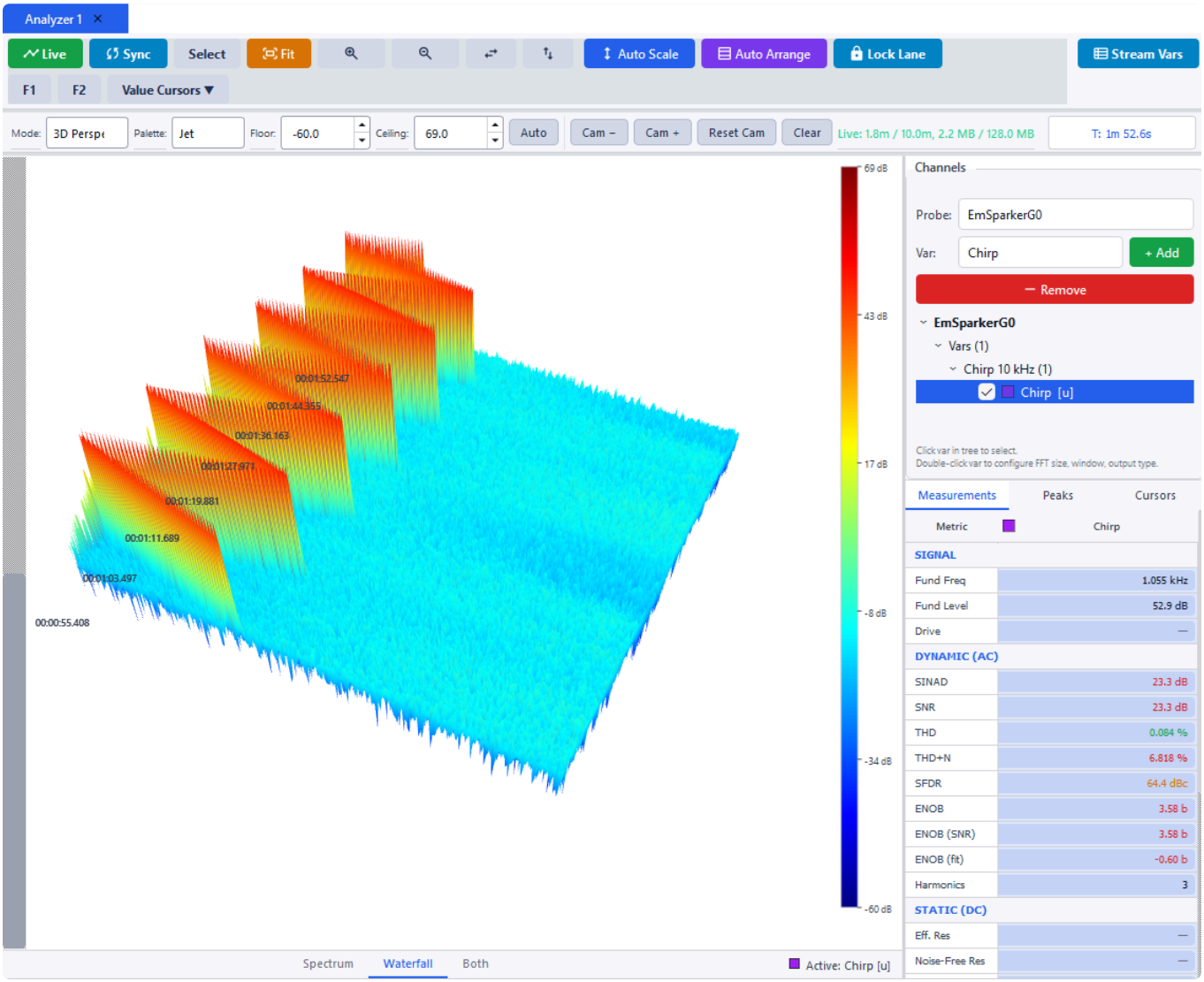
PF falls for both phase shift *and* harmonics; DF is only the fundamental phase shift. On a clean sine they agree; on a rectifier they diverge (in the capture above, PF 0.86 vs DF 0.92). Details, pairing rules, and the license note: Chapter 11 — Power analysis.

Spectrum Analyzer

Open via Toolbox → **Analyzer**. This is the frequency-domain companion to the Scope, designed around stream channels (a meaningful FFT needs a fixed sample rate).

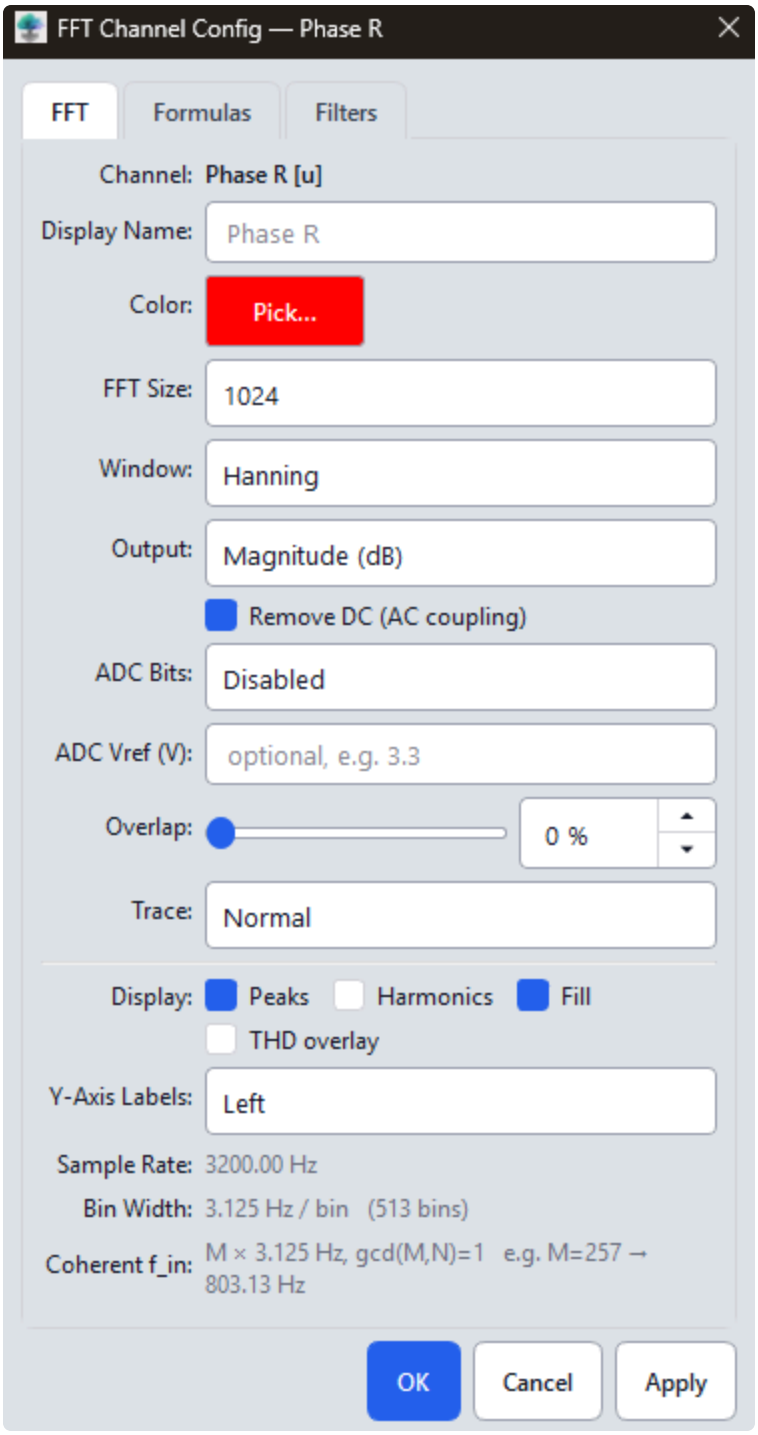
Screenshot: Analyzer with live spectrum on top and scrolling waterfall below

- **Spectrum plot** — live FFT magnitude per channel.
- **Waterfall** — time-frequency history with selectable palette and mode, plus an optional 3D perspective rendering. A swept chirp draws the classic diagonal line; harmonics show as stable verticals.



Screenshot: Waterfall in 3D Perspective mode — a 10 kHz swept chirp rendered as a moving ridge over the noise floor, with the Measurements sidebar (SINAD, SNR, THD, ENOB, …) computed live on the same stream

- **Per-channel FFT configuration** — window, FFT size and related parameters per channel via the FFT config dialog.



Screenshot: FFT channel configuration dialog with window/size settings

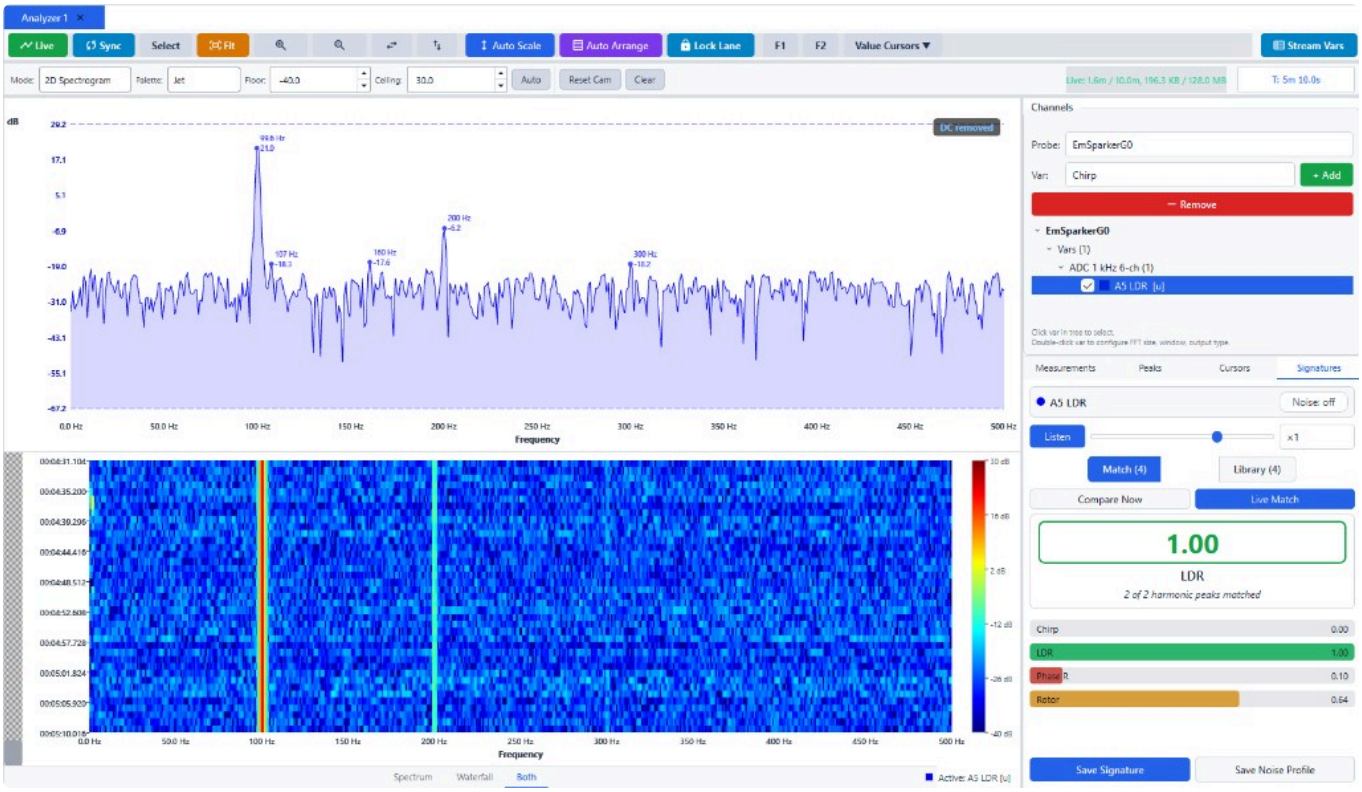
- **Toolbar parity with the Scope** — Live/Review, Fit, zoom, **F1/F2 frequency cursors**, value cursors, lane lock, plus sidebar tabs for **Measurements**, **Peaks**, **Cursors**, and **Signatures** (sound/vibration recognition — see below); cursors here measure **frequency** positions and deltas (e.g. read a harmonic's exact frequency).
- **Sync** — participates in view sync, so the analyzer's review position follows the same timeline as your Scope/Trace views.

Practical example: a 3-phase grid stream that carries 3rd/5th harmonics shows them as 150/250 Hz lines in the spectrum; selecting a firmware-shipped 80 Hz low-pass in the channel

properties rolls them off live.

Signatures — Sound & Vibration Recognition

The Analyzer's **Signatures** sidebar tab turns the live spectrum into a recognition engine. A **spectral signature** is a saved fingerprint of what a stream *sounds like* — a fan, a bearing, a motor winding, a pump. Once a signature is saved, Traxcope can look at live (or recorded) stream data and tell you, with a 0–1 confidence score, **which saved signature the current sound/vibration matches**.



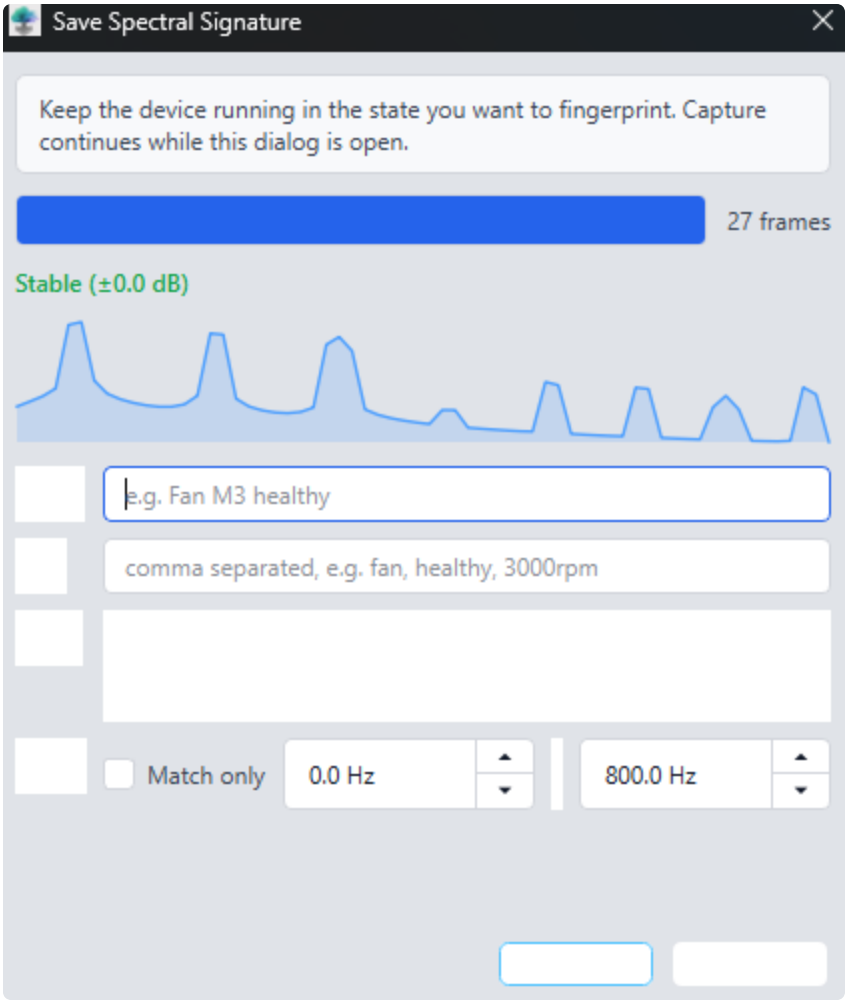
Screenshot: the Analyzer with the Signatures tab open — spectrum + waterfall on the left, and on the right the live match card reporting a 1.00 match to "LDR" with the ranked library scores below

The whole engine is deterministic C++ — there is no black-box model and no training data. Every number in a result can be traced back to the spectrum, so **the score is always explainable**. The rest of this section is the exact method, written so you can walk a customer through how a match is calculated end to end.

> **The one-sentence version:** a signature stores the **mean and variance of a machine's spectrum** measured over many FFT frames. Matching a new sound is a **statistical hypothesis test** ("could this spectrum have come from that machine?") combined with a **spectral-shape** comparison and a **harmonic-peak** check, fused into a single 0–1 score.

What a signature actually stores

A signature is **not** a single snapshot of the spectrum. A snapshot is noisy and would only ever match itself. Instead we build a small *statistical model* of the source, accumulated over many FFT frames while the device runs in the state you want to capture (e.g. "Fan M3, healthy, 3000 rpm"). Press **Save Signature** and keep the device in that state while the enrollment dialog captures frames.



Screenshot: the guided enrollment dialog — capture progress, stability indicator, live spectrum preview, and the name/tags/notes/range fields

The enrollment pipeline runs on every incoming FFT frame:

1. **Normalize to linear power.** The Spectrum Analyzer can output magnitude, dB, or power. We convert whatever it is into **linear power** (magnitude²) so the maths downstream is consistent regardless of the channel's display setting.
2. **Optional noise subtraction.** If a **noise profile** is active (a signature captured with the target source *off* — just the background), it is subtracted from every frame first:

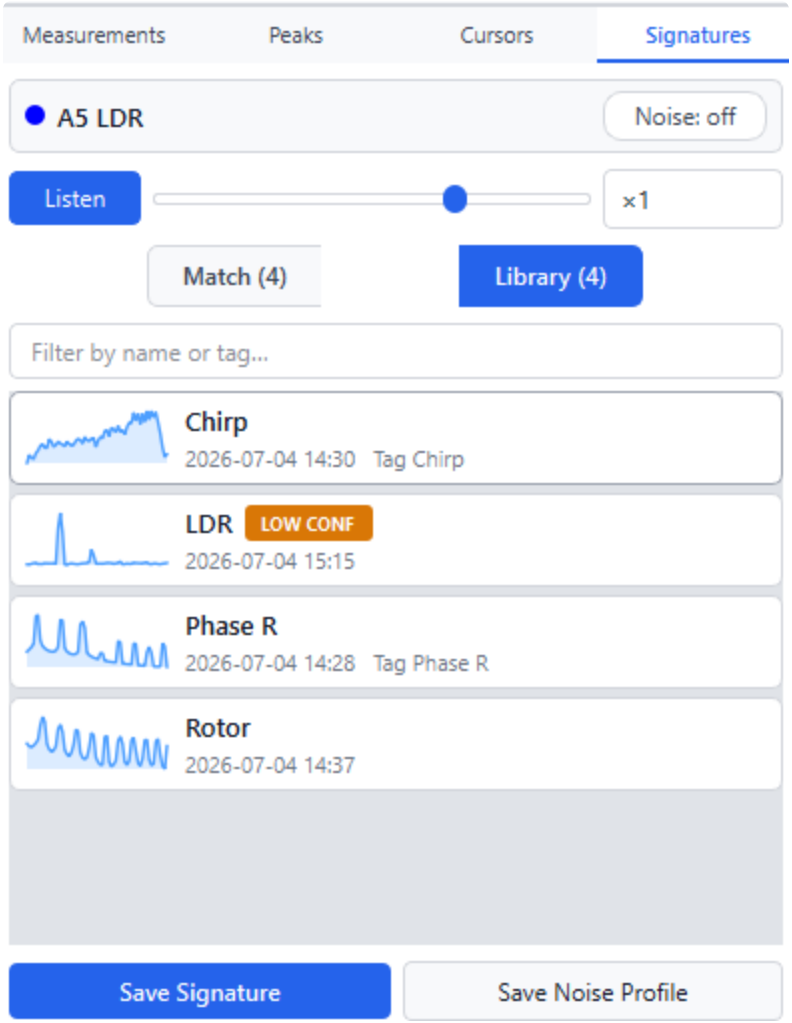
```
clean_power[i] = max( power[i] − α · noise[i],  β · power[i] )
```

with `α = 1.5` (slight over-subtraction, to be safe) and a spectral floor `β = 0.02` (never drive a bin fully to zero — that creates artifacts). This removes the room/rig background so the fingerprint describes the *machine*, not the environment.

3. **Mel filterbank → band energies.** Raw FFT bins are too fine and too many. We collapse them into a **mel-spaced triangular filterbank** — the perceptual frequency spacing used in speech/audio recognition (fine at low frequencies, coarser at high), up to 64 bands. Crucially, each band's energy is **divided by its bandwidth in Hz** (turning it into a power *density*) and stored in dB. This one detail is what makes a signature captured at, say, a 1024-point FFT still match a candidate computed at 4096 points — the dB values become **FFT-size invariant**.
4. **Running mean and variance (per band).** Across all frames we keep a running **mean** (`bandMeanDb`) and **variance** (`bandVarDb2`) for every band (Welford's algorithm). The variance is the important part: a machine's spectrum wobbles differently in different bands; some are rock-steady, others breathe. Storing the variance lets matching later **automatically trust the steady bands more and forgive the noisy ones** — no manual tuning.
5. **Scalar descriptors + consensus peaks.** From the averaged spectrum we also extract human-readable descriptors — **spectral centroid** (brightness), **flatness** (tonal vs noise-like), **95% roll-off** frequency, **crest** (peakiness) — and the dominant **harmonic peaks** (up to 10). A peak only counts if it rises at least **10 dB above the median noise floor**, so random noise bumps never enroll as fake peaks. Peak frequencies are refined to sub-bin accuracy by parabolic interpolation.

Confidence gating: below **4 frames** enrollment is refused; below **20 frames** the signature is saved but flagged **low-confidence** in the UI.

Saved signatures live in the **Library** (searchable by name or tag). Each card shows a spectrum thumbnail, the capture time, tags, and a **LOW CONF** badge where applicable; right-click a card to rename, edit tags/range, export/import, activate as a noise profile, or delete.



Screenshot: the Library tab — saved signatures (Chirp, LDR flagged LOW CONF, Phase R, Rotor) each with a spectrum thumbnail and metadata, plus the Save Signature / Save Noise Profile actions

How a candidate is matched against a signature

At match time we build a **candidate** the exact same way — using the *same* code path — from a short rolling window of recent frames. Running enrollment and matching through one shared pipeline means there is no "train/test skew": both sides are measured identically. Matching a candidate against one signature produces three complementary sub-scores, each 0–1.

- **Grid alignment first.** If the candidate and signature were captured at different FFT sizes, the candidate's stored average spectrum is **re-banded onto the signature's exact band grid** before comparison (interpolating already-collapsed bands would smear sharp tones and corrupt the statistics). This is why cross-FFT-size matching works.
- **Band weighting.** Bands **outside an optional user frequency range** get weight 0 (e.g. "only match 200–800 Hz"). If a noise profile is present, each band is weighted by its **signal-to-noise ratio**: full weight only where the signature sits **≥ 6 dB above the noise**; bands buried in the background are down-weighted toward 0. The decision is made only where the machine actually rises above the environment.

Sub-score 1 — Statistical match (weight 0.5). The core: a **diagonal Mahalanobis distance** — the statistically-correct way to ask *"how many standard deviations is this candidate from the model?"*

```
d² = Σ  weight[b] · ( candidate[b] − mean[b] )²  /  variance[b]
```

Each band's deviation is scaled by **that band's own variance** — a big swing in a naturally-noisy band barely counts, a small swing in a rock-steady band counts a lot. We drop the worst 10% of bands (robust trimming, so one freak band or a transient can't sink an otherwise perfect match), then convert the distance to a 0–1 probability-like score via the **chi-square survival function** (Wilson-Hilferty approximation). Near 1 = "highly consistent with that machine"; near 0 = "statistically implausible".

Sub-score 2 — Spectral shape (weight 0.3). A **cosine similarity** of the two band vectors after subtracting each one's mean. Because the mean is removed, this compares the *shape* of the spectrum independent of overall loudness — robust to gain differences (mic distance, drive level).

Sub-score 3 — Harmonic peaks (weight 0.2). The fraction of the signature's stored peaks that reappear in the candidate within a **±1.5%** frequency tolerance — the "same harmonic family" check, the tell-tale for rotating machinery where fault signatures live at specific harmonic/sideband frequencies.

Fusion into one score:

```
fused = 0.5 · statistical  +  0.3 · shape  +  0.2 · peaks
```

(If the signature has no usable peaks, the peak weight is redistributed onto the other two.) The whole library is scored against the candidate and ranked best-first. The UI colour-codes the headline score:

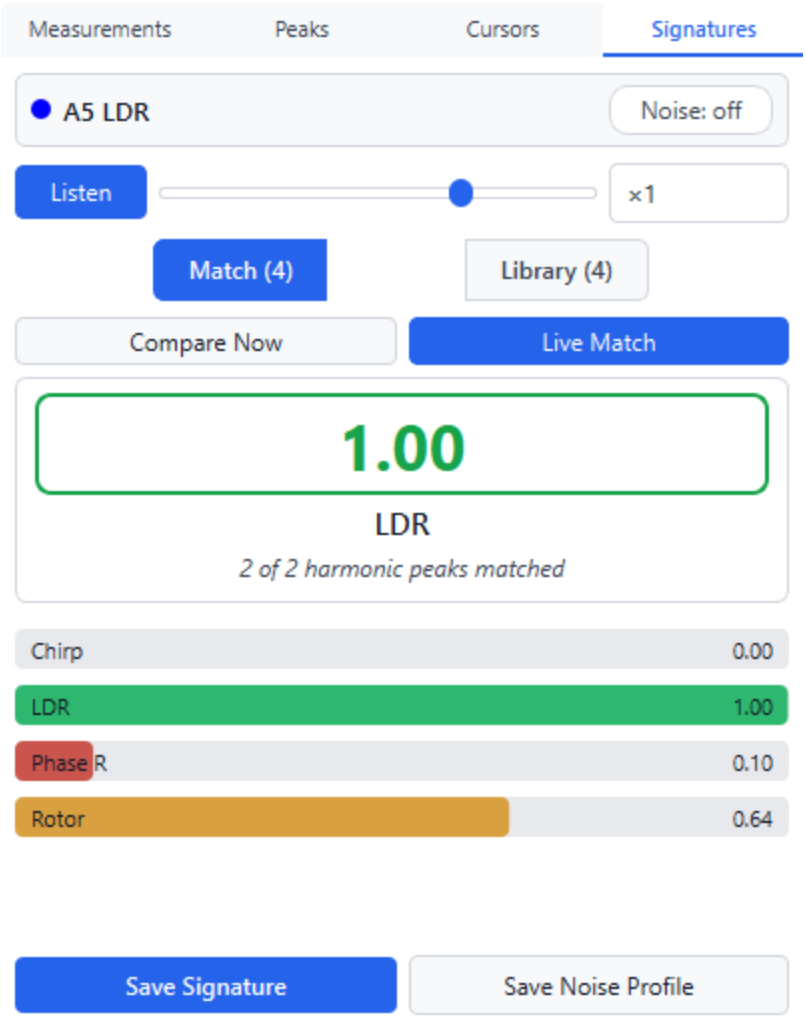
Fused score	Meaning	Colour
≥ 0.85	Strong match	green
0.60 – 0.85	Possible match	amber
< 0.60	Weak / no match	red

Every result also carries its three sub-scores, the number of matched peaks, and the fraction of bands used — so you can always explain *why* a score is what it is.

Live (background) matching

Recognition runs continuously without touching UI responsiveness. A dedicated **worker thread** holds an immutable snapshot of the library, keeps a rolling window of the newest frames (**12 frames**; older frames are dropped, so a fast FFT rate can never flood it), and roughly **every 500 ms** builds a candidate and re-scores the whole library, emitting a ranked result list back to the panel. Nothing heavy ever runs on the GUI thread. You can also **Listen** to the channel (audio playback with pitch/volume) and A/B what a match sounds like.

Toggle **Live Match** to run continuously, or press **Compare Now** for a one-shot score of the current window. The match card shows the best match's headline score (colour-coded) with its evidence ("2 of 2 harmonic peaks matched"), and every library entry gets a ranked bar below.



Screenshot: the live match card — a 1.00 green match to "LDR" with peak evidence, and the ranked bars below (LDR 1.00 green, Rotor 0.64 amber, Phase R 0.10 red, Chirp 0.00)

Detecting multiple sources at once (Components mode)

Everything above answers *"which single source is this?"*. But real channels often carry **several sources at the same time** — two machines and a pump on one microphone. Feed such a mixture to the Identify matcher and every individual signature scores mediocre: each one is penalized for the *other* sources' energy. That is not a bug, it is the wrong question. The Match

page's **Identify / Components** switch asks the right one: *"which enrolled signatures are **contained** in this spectrum?"*

The physics that makes this answerable: **uncorrelated sources add in linear power**. The mixture's power spectrum is (approximately) the sum of the sources' power spectra, each scaled by how loud it currently is:

```
mix[b]  ≈  a·A[b] + b·B[b] + c·C[b] + n·Noise[b] ,      a, b, c, n ≥ 0
```

Every signature already stores its average linear-power spectrum, so the library doubles as a **dictionary**: Components mode solves this equation for the gains with **non-negative least squares** (NNLS — gains can't be negative because a machine can't emit anti-sound). The active noise profile plus a flat "white" column are included so background energy is never blamed on a signature. Note the dB-domain scoring used everywhere else is exactly why Identify *can't* do this — addition only works in linear power.

Each signature then gets a 0–1 **presence** score fused from three pieces of evidence, every one of them explainable:

- **Own-band energy (weight 0.40)** — inside the bands where that signature actually lives (within 20 dB of its strongest band), what fraction of the mixture's energy does its fitted term explain?
- **Leave-one-out fit (weight 0.35)** — remove the signature's column and re-solve: if the fit gets much worse, the mixture genuinely needs this source. Two near-identical signatures naturally split this evidence — the honest answer for an ambiguous pair is "one of these, can't say which", and that is what you'll see (both amber, neither confidently detected).
- **Harmonic peaks (weight 0.25)** — the signature's stored peaks found in the mixture. Peaks survive mixing (a tone doesn't move because another machine turned on), which makes them strong containment evidence.

Presence ≥ 0.5 reports the source as **detected** (green bar with a check mark); the card summarizes "2 of 4 sources detected" with the detected names. The card also shows the **unexplained energy** — the share of the mixture no dictionary column accounts for. When it climbs past ~25% something is playing that was never enrolled: an *unknown-source* alarm you get for free from the residual.

Hovering any row shows the full evidence: presence, fitted level relative to the enrolled loudness (in dB — a source can be detected quieter or louder than it was enrolled), own-band energy explained, leave-one-out degradation, and peak matches. **Compare Now** works in Components mode too (one-shot decomposition of the current window), and Live Match re-solves every 500 ms on the same background worker — the NNLS problem is at most 64 bands × library size, microseconds of work.

To see the difference on a mixture: enroll each source from a clean recording, open a channel that plays several of them at once, and switch to Components — the detected set should follow which sources are actually present. Flip back to Identify on the same mix to see why the single-source question fails.

The Match page remembers the Identify/Components choice across sessions and starts in **Components** on a fresh install — field spectra are usually mixes, and Components degrades gracefully to the single-source case.

Enrolling when sources can't be isolated

In the field you rarely get each machine alone in a silent room. Two tools cover the two contamination cases:

- **Enroll Residual… (peel-off enrollment)** — visible on the Match page in Components mode. Use it when a source is running that you *can't* switch off alone but everything else in the mix is already enrolled. The panel fits the current spectrum against the library once (freezing the fitted levels), then runs a normal enrollment where that fitted reconstruction — every known signature at its current loudness, plus the active noise profile — is subtracted from **every** captured frame. What gets enrolled is exactly the part of the mix the library could not explain: the unknown source, with genuine per-frame variance. The result is tagged `residual`. Two conditions matter: keep the known sources at **steady levels** during the capture (the fitted gains are frozen at the moment you click), and check the card's *unexplained energy* first — if it's near zero there is nothing left to enroll.
- **Derive by Subtraction… (signature arithmetic)** — right-click any library entry that stores its average spectrum. Use it *after the fact*, when a signature was enrolled with a contaminant running (a sibling machine, a hum) and you also have that contaminant enrolled alone: `A with B` − `B` = clean A. The subtraction happens in linear power (energies of uncorrelated sources add, so they subtract the same way) and the residual spectrum is re-run through the standard enrollment pipeline, so band model, descriptors and consensus peaks stay consistent. Per-frame variance can't be reconstructed from two averages, so the derived entry carries the LOW CONF badge and leans on shape + peak evidence — fine as a Components dictionary column, weaker as a precision Identify reference. The result is tagged `derived` and its notes record the formula.

Rule of thumb: prefer a **noise profile** when the background is on during enrollment (it also drives SNR band weighting), **Enroll Residual** when the mixture is live and the rest of it is known, and **Derive by Subtraction** to repair recordings you already have.

Analyzing recordings (Review mode)

Everything above also works on **recorded** data — you don't need the machine running to identify what was on a channel last Tuesday. Switch the Analyzer to **Review**, fetch the range

you care about (the *Re-run review…* dialog), and three tools become available:

> Adding a recording channel auto-fetches a **preview of the recording's last 10 minutes** so a spectrogram appears immediately — it deliberately does *not* process the whole file (recordings can span months). Use **Re-run review…** to fetch any other window, at any frame interval, when you decide what part of the recording matters.

- **Analyze a selected span.** Turn on the toolbar **Select** toggle and drag a vertical band across the waterfall. The Signatures panel immediately analyzes exactly that time slice: the match card shows a *Recorded span* badge with the range and frame count, and the current mode (Identify or Components) scores it on the spot. Drag a different band and the scores follow; **Compare Now** re-runs the same span on demand. Live Match is disabled while a span is selected — the data is frozen, there is nothing to match "live". Everything that consumes the capture window consumes the selection instead, so you can also **Save Signature**, **Save Noise Profile** or **Enroll Residual…** straight from a recorded segment — enrolling a machine from an archived recording works exactly like enrolling it live.
- **Listen to the selection.** With a span selected, the **Listen** toggle plays that recorded segment through the PC speakers **on a loop** (volume and the ×1/×4/×16 pitch shift work as in live mode). Selecting a new span while listening switches the audio to it. Ideal for A/B-ing a suspect segment against what a healthy signature *sounds* like.

  While the segment plays, a green **playhead line** sweeps across the selected band on the waterfall showing exactly which moment is coming out of the speakers (the position is compensated for the audio output buffer, so it tracks what you *hear*, not what has been queued). Two things follow the line: the **spectrum panel** re-computes the FFT at the audible moment (the same one-shot compute a waterfall row click triggers, refreshed several times a second), and the **match card** re-scores the analysis window under the playhead — same 12-frame window Live Match and Scan Recording use — so in Components mode you can watch sources appear and disappear from the detection list as the audible content changes, with the chip switching to "Playing hh:mm:ss". When you stop listening (or the selection goes away), the line disappears and the card returns to the whole-span result.
- **Scan Recording…** (Match page, Review mode only). Instead of asking about one span, sweep the **whole reviewed range**: a sliding window (12 frames, 50% hop — the same window Live Match uses, so scores are comparable) runs the Components decomposition at every step and the result opens as a **component timeline**. The activity map draws one row per enrolled signature — green blocks where it was detected (brighter = stronger presence), amber where it hovered below the detection threshold — plus an **Unknown** row marking windows where the unexplained-energy residual crossed the alarm level. Below it, a **segments table** merges consecutive detections into start / end / duration / average / peak-presence rows ("Pump B ran 00:02:10 – 00:07:42"), and **Export CSV…** writes the raw

per-window presence matrix (one column per signature plus the residual) for offline analysis.

Together these close the loop: record a session in the field, review it in the office, identify which sources were active and when, enroll anything unknown from the very segment where it appeared — and listen to it while you do.

A worked example (healthy vs bearing fault)

To make the scoring concrete, here is a fully-computed match on a simplified **6-band** model (the real engine uses up to 64 — the arithmetic is identical, just longer). All weights are 1 (no noise profile, no range restriction).

The saved signature — "Fan M3, healthy":

Band center	**100 Hz**	**200 Hz**	**400 Hz**	**800 Hz**	**1600 Hz**	**3200 Hz**
Mean level (dB)	40	55	42	30	25	20
Variance (dB²)	4	2	6	9	5	8
Stored peaks		200 Hz	400 Hz			

The 200 Hz band is loud **and** steady (variance 2) — the fan's fundamental. The high bands are quieter and naturally wobblier. Two live candidates arrive:

Band	**100**	**200**	**400**	**800**	**1600**	**3200**
A — same healthy fan	41	54	43	31	24	22
B — bearing fault (energy climbs in the highs)	40	52	48	45	40	38

Statistical — `d² = Σ (candidate − mean)² / variance`:

```
A: 0.25 + 0.50 + 0.17 + 0.11 + 0.20 + 0.50   =  d² ≈ 1.73  (6 dof)  →
statistical ≈ 0.94
B: 0    + 4.5  + 6.0  + 25.0 + 45.0 + 40.5    =  d² ≈ 121   (6 dof)  →
statistical ≈ 0.00
```

1.73 is far below the expected value of 6 → high score; 121 is astronomically beyond → rejected. The variance did the work automatically: B's big deviations landed in the low-variance bands (800/1600 Hz), exactly where they hurt most.

Shape (mean-subtracted cosine): **A ≈ 0.99** (near-identical silhouette); **B ≈ 0.84** (low end still lines up, but raised highs tilt the shape — shape alone is forgiving, which is why it's only 30%).

Peaks (±1.5%): **A** → **1.0** (both 200 & 400 Hz reappear); **B** → **0.5** (fault smears 400 Hz; only 200 Hz survives).

Fusion:

```
A:  0.5·0.94 + 0.3·0.99 + 0.2·1.0  =  0.97   → GREEN  (strong match)
B:  0.5·0.00 + 0.3·0.84 + 0.2·0.5  =  0.35   → RED    (weak / no match)
```

The takeaway for a customer: the **statistical term is what separates them** (0.94 vs 0.00). Shape alone would have called B a 0.84 "maybe" — the variance-aware statistical test is what confidently rejects the fault, and you can point at the exact bands (800/1600 Hz) that drove the rejection.

Why this method (talking points)

- **Statistically principled, not ad-hoc.** A Mahalanobis distance calibrated to a real probability via chi-square — the textbook answer to "does this sample belong to this distribution?".
- **Self-calibrating.** Per-band variance means the system learns which parts of each machine's spectrum are trustworthy. No per-machine threshold tuning.
- **Robust by construction.** Mean-subtracted shape handles loudness/gain; robust trimming handles transients; SNR weighting + noise-profile subtraction handle the environment; mel-band-density normalization handles FFT-size differences.
- **Explainable.** Three named sub-scores + peak evidence + bands-used — you can defend any verdict. No opaque neural network, nothing to "retrain".
- **Cheap and deterministic.** Pure C++, runs live on a background thread, identical results every time for the same input.

Key parameters at a glance

Parameter	Value	Role
Max bands	64	Mel filterbank resolution ceiling
Min frames (hard / recommended)	4 / 20	Enrollment gating
Peak prominence	10 dB above median	Rejects noise bumps as fake peaks
Peak match tolerance	±1.5%	Harmonic alignment window
Variance floor	1 dB²	Stops a too-steady band from dominating

Parameter	Value	Role
Robust trim	worst 10%	Transient / edge-case rejection
Noise over-subtraction α / floor β	1.5 / 0.02	Background removal
SNR full-weight margin	6 dB	Band must beat background by this
Fusion weights (stat / shape / peaks)	0.5 / 0.3 / 0.2	Final score blend
Live window / cadence	12 frames / 500 ms	Background matcher
Components: own-band window	20 dB below signature max	Bands counted as "its own"
Components: presence weights (energy / leave-one-out / peaks)	0.40 / 0.35 / 0.25	Presence blend
Components: detection threshold	presence ≥ 0.5	Reported as detected
Components: unknown-source residual	≥ 25% unexplained	Flags un-enrolled energy

Troubleshooting

Symptom	Likely cause
Compile error "Stream TID index >= TRAX_CFG_STREAM_CNT"	Increase `TRAX_CFG_STREAM_CNT` in `trax_config.h`
Stream plots but timing drifts vs other events	`TRAX_CFG_TIMER_FREQ_HZ` / sample rate metadata doesn't match the real hardware clocks
Periodic NaN gaps	Producer overruns (block not sent before next UPDATE) or intentional sliced mode; check task priorities and transport bandwidth
Garbled multi-channel data	Buffer not interleaved `[ch0, ch1, ..., ch0, ch1, ...]`, or channel data types don't match the buffer layout
Math channel rejected / timestamps don't pair	Inputs are not channels of the same stream, or the ADC scan offset is larger than half a sample period (retune the offset metadata, or drop the slower channel from the recipe)
Power tab finds no V/I pairs	Both traces need data in T1→T2 and shared timestamps — use two channels of this stream, Review mode, cursors armed; zoom to RAW if a lane shows `[≈]`
No spectrum	Channel is an async VAR, not a stream — the analyzer needs fixed-rate data

Symptom	Likely cause
Signature saved but flagged "low-confidence"	Fewer than 20 frames captured at enrollment — keep the device in the target state longer while the dialog captures
Everything scores low, even the same source	An FFT channel change or the wrong noise profile is active; recapture, or turn the noise profile Off in the Signatures header
A different machine scores high	Signatures are too similar in that band range — set a **Match only** frequency range on the signature to focus on its discriminating band(s)
Identify scores drop when several sources play together	Expected — Identify answers "is this ONE source?". Switch the Match page to **Components** to detect each contained source separately
Components shows two similar signatures both amber, neither detected	The pair is spectrally ambiguous inside a mixture (the fit can't tell them apart) — re-enroll one with a **Match only** range on its discriminating bands