21 — Estimator Studio (Kalman filters)

TraxProbe & Traxcope User Manual · Version 2.2.0 · Updated 2026-11-02

The Estimator Studio runs **Extended Kalman Filters** over your live variables and publishes the estimated states back as ordinary channels. Use it to get a clean velocity out of a noisy position sensor, a drift-free angle out of a gyro that drifts, or a smoothed level whose responsiveness you can dial with one number — **without touching the firmware**.

Everything runs **on the host**. The device only has to stream the raw sensor variables it already streams; the model, the tuning, and the arithmetic live in Traxcope. When the filter is behaving, TraxAI can write the same filter into your firmware as plain C (§10).

> **License.** The Estimator Studio is part of the **DSP** feature set (**Professional** only). On Standard and Community the Toolbox button opens an upgrade page instead of the view, and Estimator docks in a restored workspace are silently skipped. The TraxAI estimator tools (§4, §10) need the same Professional license.

Why a filter and not a filter

Chapter 11 already gives you IIR/FIR pre-filters on any variable. Those are worth reaching for first: they are cheap, they need no model, and for "take the buzz off this reading" they are the right tool.

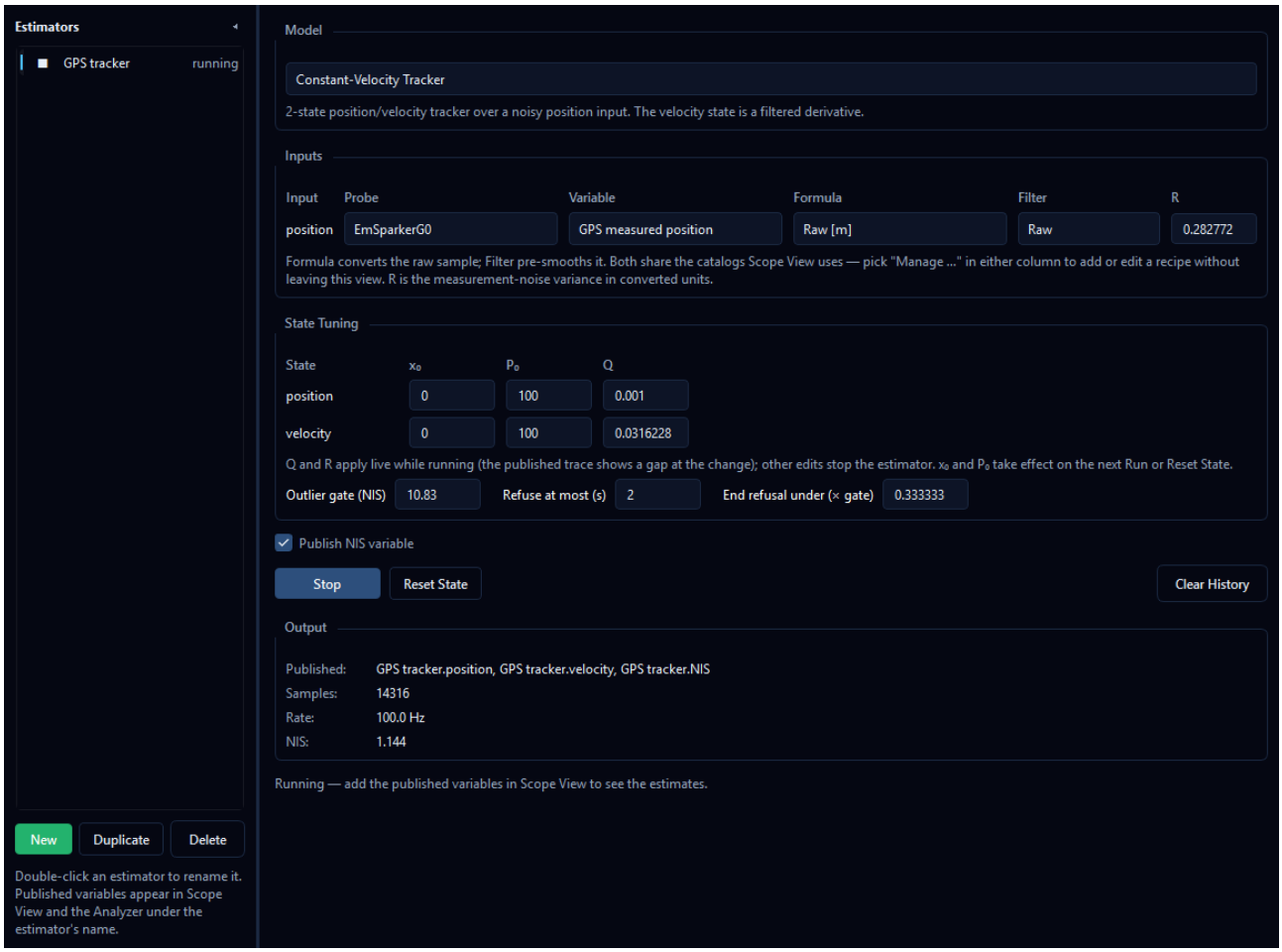
An estimator earns its place when a low-pass cannot answer the question:

You want	A low-pass gives you	The estimator gives you
Velocity from a noisy position sensor	A differentiated mess, or lag if you smooth first	A **velocity state** the model maintains, smoothed by construction
An angle from a drifting gyro	Drift, forever	Gyro fused with an absolute reference, plus an **online bias estimate**
Behavior across a 2-second dropout	A flat line, then a step	Prediction along the model, returning with **honestly inflated uncertainty**
Two sensors at two different rates	Nothing — you cannot low-pass two streams into one	Sequential fusion, each sample used the instant it arrives
"Is my noise model honest?"	No answer	**NIS**, a number you can read (§6)

The filter carries a *model of how the signal moves*. That is what lets it coast through a gap and fuse sensors that disagree.

1. The Estimator Studio

Open via Toolbox → **Estimator**. The layout follows the Trigger Studio pattern: instances on the left, one editor on the right.



Estimator Studio with an instance selected and the editor filled in

Screenshot: Estimator Studio — instance list on the left, Model / Inputs / State Tuning / Output panels on the right

Area	Purpose
Estimators list	Your configured filters. **New**, **Duplicate**, **Delete**; each row has its own Run/Stop button and shows *running* while active. Double-click a row to rename it. The ◂ button in its header folds the list down to a narrow rail (▸ brings it back, **+** adds an estimator without unfolding) when you want the whole dock width for the editor
Model	The preset model, with a one-line description of what it estimates
Inputs	One row per model input: source probe, variable, conversion formula, pre-filter, and that input's measurement-noise variance **R**
State Tuning	One row per state: initial estimate $\mathbf{x_0}$, initial covariance $\mathbf{P_0}$, process noise **Q**; below them the outlier gate row (§4)
Publish NIS variable	Publish the filter's consistency statistic as an extra channel (§6)
Run / Reset State / Clear History	Run control (§7)
Output	Live status: published variable names, samples processed, output rate, most recent NIS

There is deliberately **no plot in this view**. Estimates are published as regular variables, so you look at them in Scope View — with the full channel tree, T1/T2 cursors, measurements, and LOD — and in the Spectrum Analyzer. One less half-featured plot to learn.

Multiple Estimator Studio docks share **one** instance list: open a second dock to watch the status strip of one filter while you tune another.

Whether the list is folded is saved per dock, so a workspace laid out with the list collapsed comes back that way.

Instances are saved in the workspace and restored on the next launch, **stopped** (the probe data is not there yet). Press Run when your probe is streaming.

2. Models

The studio ships three presets. Each is a small, honest state-space model — not a black box:

Preset	States	Inputs	Use it for
Scalar Kalman Smoother	`level`	`measurement`	A tunable smoother: one knob (Q) trades responsiveness against noise rejection
Constant-Velocity Tracker	`position`, `velocity`	`position`	Clean derivative from a noisy position / encoder / level signal
Attitude + Gyro Bias Fusion	`angle`, `gyro_bias`, `rate`	`gyro_rate`, `accel_angle`	Drift-free angle from a drifting gyro plus a noisy absolute reference, with the bias estimated online

What the models assume

- **Scalar Kalman** — the level is a random walk; the measurement is the level plus white noise.
- **Constant-Velocity** — position advances by `velocity × dt`; velocity is a random walk. The measurement is position only, so velocity is inferred, never measured.
- **Attitude + Gyro Bias** — the angle integrates the true rate; bias and rate are random walks. The gyro measures `rate + bias`, the accel-derived input measures `angle` directly. The two inputs may arrive at completely different rates.

Selecting a preset rebuilds the Inputs and State Tuning grids and seeds every field with that model's defaults, so a fresh instance is runnable as soon as you pick its variables.

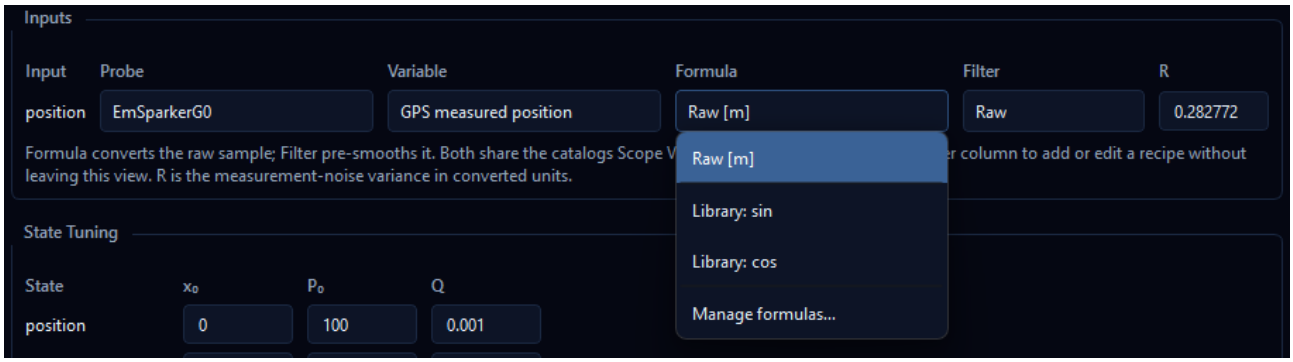
Measurement updates are **scalar and sequential**: a two-input model applies one update per input in timestamp order. That is numerically equivalent to a vector update with a diagonal R, needs no matrix inversion, and is exactly what makes the core embeddable (§10).

3. Inputs

Each input row binds one model input to one real variable.

Column	Meaning
Probe	Any registered source — a live probe or a loaded recording. Inputs may come from **different probes** (see the sync note below)
Variable	The variable feeding this input
Formula	Conversion applied to each raw sample before estimation (counts → lux, ADC → volts, …). `Raw` = none
Filter	Pre-filter applied after the formula. `Raw` = none
R	Measurement-noise **variance** for this input, in converted units

Which variables are offered. Only **stream variables** (`TRAX_STREAM_*`, Chapter 12) appear. Asynchronous `TRAX_VAR_SET*` variables are not eligible — the filter needs a regularly-sampled measurement timeline. Virtual variables are excluded too, so you cannot feed a math channel or another estimator's output back in. *(Recordings saved before stream schemas were persisted are a documented exception: every variable in such a file is offered.)*



Estimator Studio Inputs grid with the probe, variable, formula and filter pickers

Screenshot: Inputs group of a Constant-Velocity Tracker — the input row with its Probe / Variable / Formula / Filter pickers and R field; the Formula list open, showing the library's formulas (`Library: sin`, `Library: cos`) and "Manage formulas…"

Formula and filter share the Scope View catalogs. These are the same per-variable recipe lists you edit in Variable Properties (Chapter 11), including entries linked from the **Conversion Library** (Menu → Conversion Library), which appear prefixed `Library: …`. Pick **Manage formulas…** / **Manage filters…** at the bottom of either column to add or edit a recipe without leaving the Estimator Studio.

The binding stores an **explicit** choice, not "whatever is active". Changing a variable's active formula elsewhere in the app never silently changes what a configured estimator consumes.

Convert first, then tune R. The filter works in converted units, so R is a variance in those units — and it must be, or the tuning is meaningless. An inverse calibration (counts → lux, say) is the sharpest case: filtering the raw counts would give you an estimate that moves *opposite* to the channel you plot it against.

Mixing probes. Feeding two boards into one filter only makes sense if their clocks are related — set up multi-probe sync first (Chapter 18). The worker clamps negative `dt` to zero as a skew guard, so an unsynchronized pair degrades quietly rather than exploding, but the estimate is only as good as the shared timebase.

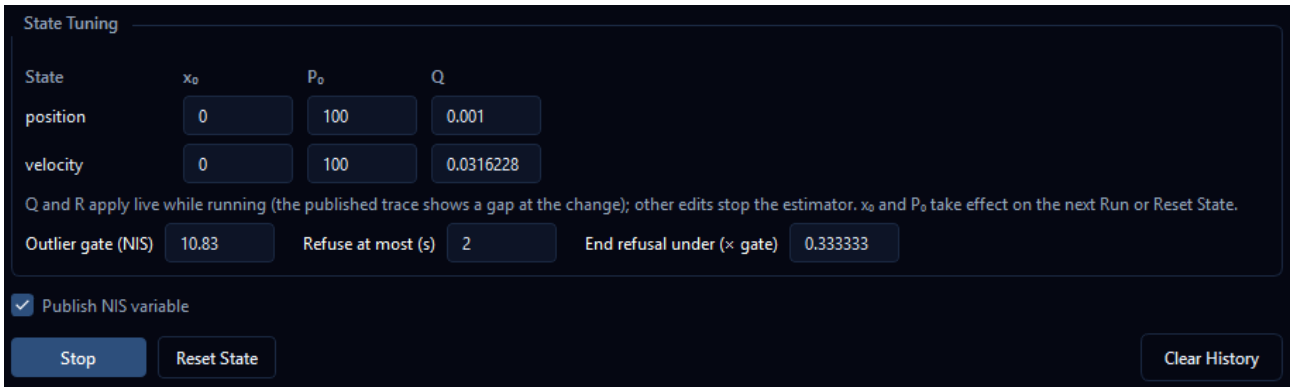
4. Tuning: Q, R, x_0, P_0

Field	Per	Meaning	Rule of thumb
R	input	Measurement-noise **variance** (not σ)	Park the sensor at a constant value, read σ off the Scope measurement panel, square it
Q	state	Continuous-time process-noise intensity — how much you believe the state can move on its own	Start with the preset default and move in factors of 10
x_0	state	Initial estimate	Leave 0 unless you know better. A state that the first measurement observes alone (the level, the position) starts at that measurement anyway
P_0	state	Initial variance of that guess	Large (100) = "I don't trust x_0, converge fast". Small = "x_0 is right, don't wander"

Q is a **continuous-time intensity**: the filter adds Q × `dt` at each prediction, so your tuning stays valid when the sample rate changes. That is why a Q tuned at 1 kHz still behaves at 100 Hz.

Q/R is the only ratio that matters. Scaling both by the same factor changes nothing. Read it as one question: *how much do I trust the model versus the sensor?*

- Q too small / R too large → sluggish, over-smoothed, laggy on real transients.
- Q too large / R too small → the estimate chases noise; you have written an expensive pass-through.



Estimator Studio State Tuning grid

Screenshot: State Tuning group — one row per state with x_0 / P_0 / Q fields, the outlier gate row, the Publish NIS checkbox, and the run row below (Stop while the estimator runs, Run when it is stopped), Reset State, Clear History

What applies when

Edit	Effect
Q or R while running	Applied **live**, no state reset. The published trace shows a deliberate gap at the change so you can see exactly where the tuning switched
x_0 or P_0	Takes effect at the next **Run** or **Reset State**
Preset, probe, variable, formula, filter, NIS checkbox	**Stops** the estimator — these change the published variable layout. The status line says so; press Run to restart
Outlier gate, Refuse at most, End refusal under	**Stops** the estimator. Press Run to restart with the new gate

Live Q/R retuning is the core workflow: run the filter, watch the estimate against the raw channel in Scope, and walk Q by decades until the trace tracks what you care about and ignores what you don't.

Validation on commit: R must be **positive**, Q and P_0 must be **non-negative**. The outlier gate and the refuse time must be **zero or more**. The refusal end must be **0, or a fraction below 1**. A malformed field is reported in the status line and the edit is rejected.

Outlier gate

A single wild sample — a GPS jump, an ADC glitch — can drag the estimate far off. The outlier gate refuses such samples. The row sits under the State Tuning grid:

Field	Default	Meaning
Outlier gate (NIS)	0 (off)	Refuse a measurement whose NIS (§6) is above this. 3.84 refuses about 5 % of honest measurements, 6.63 about 1 %, 10.83 about 0.1 %
Refuse at most (s)	3	After refusing for this long, the filter takes the next measurement anyway. A real, lasting change is then followed, not refused for ever
End refusal under (× gate)	0	Once refusing, only a measurement under this fraction of the gate ends the refusal. 0 = the gate itself. While refusing, the filter grows less sure, so later samples of the same jump start to pass the gate; 0.33 keeps them out. Whether it helps depends on the signal

Leave the gate at 0 until the filter is tuned and NIS sits near 1. Then set it from the table above.

Letting TraxAI tune it

TraxAI (Chapter 23) has three estimator tools. They need the Professional license, like the studio.

Tool	What it does
`tune_estimator`	Runs the studio's filter over stored data (the Scope selection, the visible range, a recording, or by default the last 60 s of a live probe) for each one-input model, a grid of Q and R, and outlier gates, and ranks the results — against a truth signal when you have one. Changes nothing
`apply_estimator`	Sets a studio estimator to a model, Q, R and gate (usually the best `tune_estimator` found) and runs it, after you approve
`export_estimator`	Writes the filter into your firmware's source as C (§10), after you approve the diff

Ask in plain words, for example: *"Find the best Kalman filter for* `encoder_pos` *over the selection and apply it."*

5. Running, and where the results go

Press **Run** (in the editor or on the list row). On the first Run the instance allocates its outputs and the **Output** panel fills in.

Published variables. One virtual variable per state, named `<instance>.<state>`:

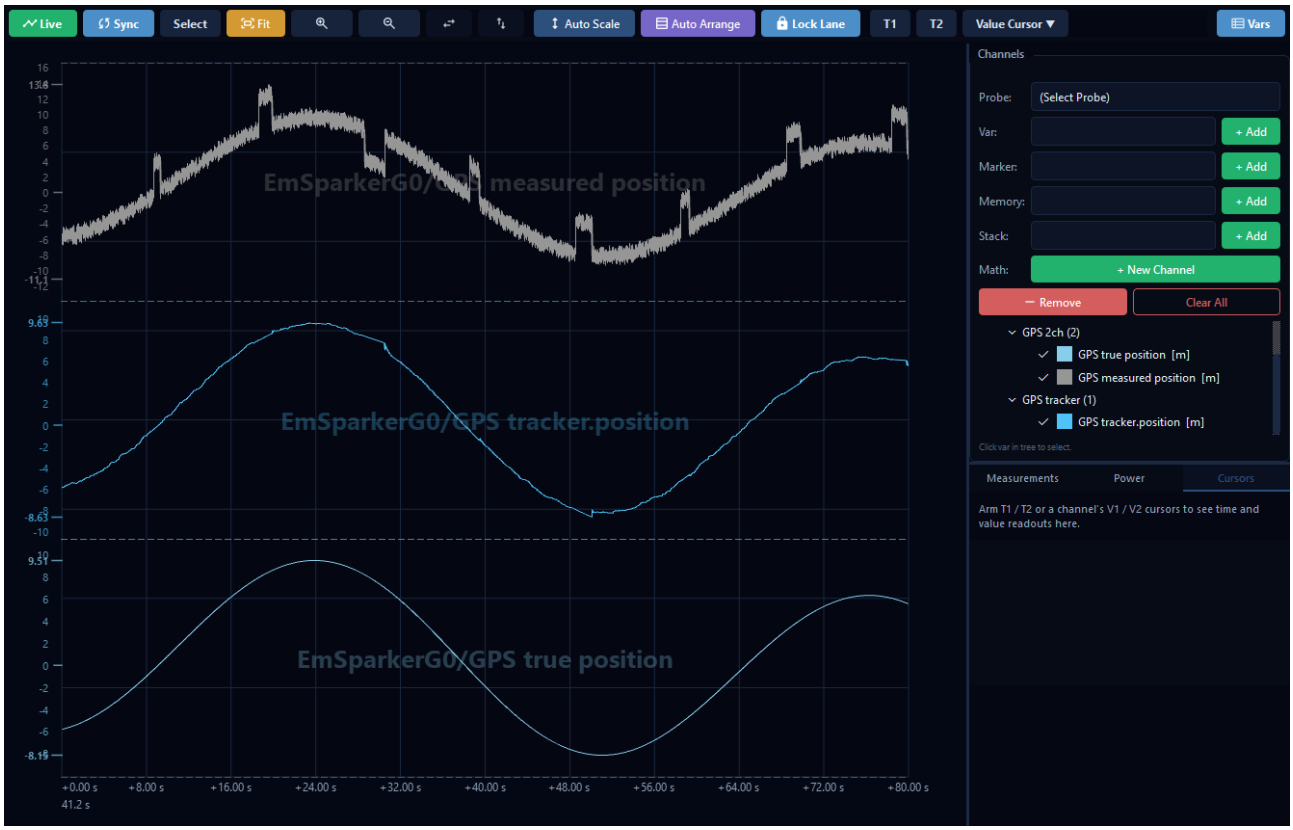
```
EKF 1.angle      EKF 1.gyro_bias      EKF 1.rate      EKF 1.NIS
```

They are published on the **primary input's probe** (input 1), as a virtual stream, at the measurement rate — one output sample per input measurement, no decimation. Add them in the Scope channel tree exactly like firmware variables; they appear grouped under the estimator's name, each with its own color.

Units and range are inherited from the primary input *through its picked formula*, so an estimate opens in Scope with the same units and a usable viewport as the source channel you will compare it against. States whose unit pattern is a rate (`velocity`, for example) get `<unit>/s`; states with no derivable unit (biases, NIS) carry none and open at ±1 — use Auto Scale.

Output panel

Row	Shows
Published	The variable names to look for in Scope (`— (assigned on first Run)` before that)
Samples	Measurements processed since the last (re)configure
Rate	Measurement throughput, Hz, over the last half-second window
NIS	Most recent normalized innovation squared (§6)



Scope View plotting an estimated state below its noisy source channel

Screenshot: Scope View on the T11_Estimator example — the noisy GPS measured position with its glitches, the tracker's published position, and the true position. With the outlier gate on (§4) the estimate follows the true path through every glitch. In the channel tree the estimator's variables are grouped under its name (GPS tracker)

Estimator outputs are never written into a recording. They depend on tuning parameters you are actively changing, so recording them would bake a transient decision into a permanent file. Record the raw inputs; the estimate is always reproducible from them.

Renaming a running instance keeps the old published names until the next Run — the names are baked into the schema at Run time.

6. NIS — is the filter telling the truth?

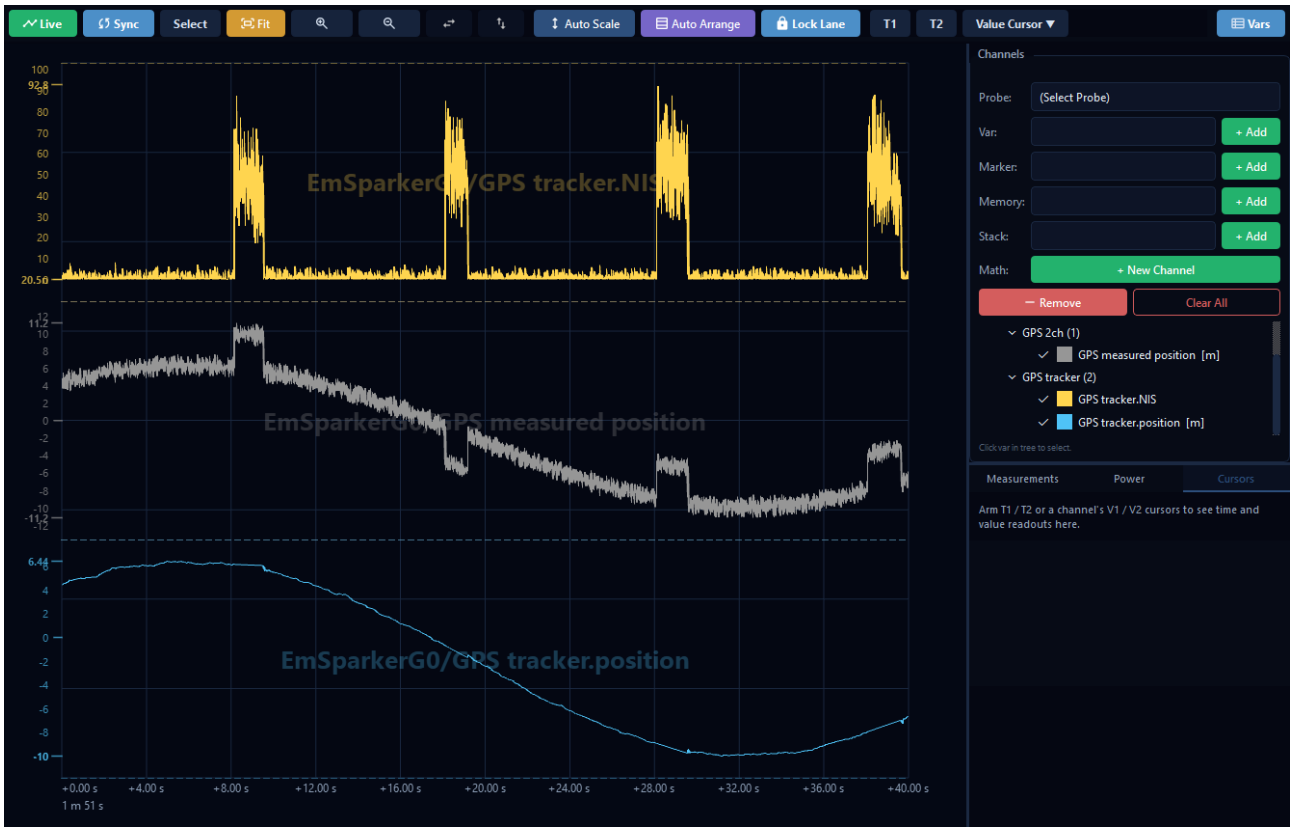
Tick **Publish NIS variable** and the filter publishes its **normalized innovation squared**: how surprised it was by each measurement, divided by how surprised it expected to be.

For a scalar update, NIS is chi-square with one degree of freedom, so:

Sustained NIS	Reading	Fix
≈ 1	Honest. Q and R are consistent with the data	Nothing — you are done
≫ 1	**Over-confident.** The filter is repeatedly startled: it claims a precision it does not have	Raise R (the sensor is noisier than you said) or raise Q (the state moves more than the model allows)
≪ 1	**Over-cautious.** Innovations are far smaller than predicted — you are throwing away good data and lagging	Lower R, or lower Q

The outlier gate (§4) uses the same number: a measurement whose NIS is above the gate is refused.

Plot `<instance>.NIS` in Scope next to the estimate and use its mean over a T1→T2 span as the tuning target. Single samples bounce around a lot; the average is the statistic.



NIS trace in Scope next to the estimated position

Screenshot: Scope View — the published NIS channel stays near 1 while the model fits the data and spikes at each GPS glitch, where the measured position (middle) jumps away from the prediction and the outlier gate refuses it; the estimated position below stays on track

This is the difference between a filter that *looks* smooth and one that is *correct*. A trace can look beautiful with NIS at 30 — the smoothness is the filter ignoring measurements it should be listening to.

7. Run control

Action	Effect
Run	Allocates and registers outputs (first Run), then starts the worker
Stop	Halts the worker. Published variables and all their data **remain** — the traces stay in Scope and stay inspectable
Reset State	Re-seeds $x = x_0$, $P = P_0$ on the next tick. Published history is kept; the trace shows a gap at the reset
Clear History	Discards every published sample of this instance. The Scope trace empties; a running estimator continues from now
Delete	Stops the instance, unregisters its variables, and clears their data
Duplicate	Copies the recipe as `<name> (copy)`. The copy gets its **own** output variables and starts stopped

Each instance runs on its **own worker thread**, ticked at 20 Hz, polling up to 4096 new samples per input per tick. Run as many as you have signals worth estimating.

8. Gaps, resets, and multi-rate fusion

The behavior around discontinuities is where an estimator differs most from a filter, so it is worth knowing exactly what happens.

Sample gaps — a store eviction, a block boundary, a dropped chunk — clear the pre-filters' feedback memory and break the plotted output trace. They do **not** reset the filter by themselves.

Elapsed time decides. The single rule is the interval between consecutive measurements:

- **≤ 5 s** — the filter *predicts across the outage*. It returns tracking the signal, with a covariance honestly inflated by the time it flew blind. This is the whole point: a low-pass would have flat-lined and then stepped.
- **> 5 s** — too long to trust the model. The filter resets to x_0/P_0 and starts a new block.

Every discontinuity is visible. A gap marker breaks the plotted trace whenever the filter (re)starts: after a configuration change, after **Reset State**, after a live Q/R retune, and after an input gap. If you did not expect a break in the trace, something happened — go and find out what.

Multi-rate fusion. Each tick polls every input, merges all new samples into one chronological stream, then walks it: predict to the sample's timestamp, update with that sample's input. A 1 kHz gyro and a 50 Hz accelerometer need no resampling and no alignment — each measurement is applied at the instant it was taken. Samples sharing a timestamp are applied in input order.

Non-finite values are skipped, not fatal. A formula that goes out of domain (`log` of a negative, say) drops that sample without breaking the trace.

9. Worked example — drift-free angle from a drifting gyro

The firmware streams two variables: `gyro_z` (rad/s, fast, drifting) and `accel_angle` (rad, slow, noisy, but absolute).

1. Toolbox → **Estimator** → **New**. Double-click the row and name it `Attitude`.
2. **Model** → *Attitude + Gyro Bias Fusion*.
3. **Inputs**:
   - `gyro_rate` → probe → `gyro_z`, Formula `Raw`, Filter `Raw`.
   - `accel_angle` → probe → `accel_angle`.
4. **R**: leave the gyro at `0.01` and the accel at `0.5` for the first run — the defaults already encode "the gyro is precise, the accel is noisy".
5. **Run**. The Output panel reads `Attitude.angle, Attitude.gyro_bias, Attitude.rate`.
6. In Scope View, add `Attitude.angle` next to the raw `accel_angle`, and `Attitude.gyro_bias` on its own axis.
7. Hold the board still. `gyro_bias` should converge to a small non-zero constant within a few seconds — that is the offset that was quietly integrating into your angle.
8. Rotate and hold. `Attitude.angle` should track the motion crisply (from the gyro) and settle without drifting away (from the accel). If it lags the real motion, raise Q on `rate`; if it jitters with the accel noise, raise R on `accel_angle`.
9. Tick **Publish NIS variable** (this stops the estimator — press Run again) and check that the mean NIS over a quiet span sits near 1.

Each Q/R edit lands live and marks a gap in the trace, so a single Scope screenshot shows every tuning attempt in sequence.

10. Taking the filter to the target

The filter you tuned can run in your firmware. The firmware version is **`trax_kf.h` / `trax_kf.c`**: plain C99 (also valid C++), no heap, no libm, fixed-size storage. It is the same filter the studio runs — the same three models, the same Joseph-form update and the same outlier gate — so a setting tuned on recorded data behaves the same on the target.

How to get it. The studio has no export button. Ask TraxAI to export the estimator; its `export_estimator` tool does it (Professional, see §4). It needs the probe's **Sources** folders set. It writes:

- `trax_kf.h` and `trax_kf.c`, once, shared by every filter. By default `trax_kf.c` goes next to `main.c` and `trax_kf.h` into the `Inc` folder beside it.
- One configuration per filter: a `kf_<name>.h` header with a `static const trax_kf_config KF_<NAME>`, or the configuration inserted into a user region of a file you name.

You see all of it as one diff and approve it before anything is written. Re-export to change a filter; edits to the generated header are overwritten.

Using it:

```
#include "kf_encoder.h"            /* static const trax_kf_config KF_ENCODER */

static trax_kf kf_encoder;

trax_kf_init(&kf_encoder, &KF_ENCODER);          /* once, at start-up */

/* each sample: */
trax_kf_predict(&kf_encoder, dt_s);              /* seconds since the last sample */
trax_kf_update(&kf_encoder, position, 0);        /* measurement, input index */
float pos = kf_encoder.x[0];                     /* the estimate: x[0] position, x[1] velocity */
```

`trax_kf_update()` returns the update's NIS, also when the gate refused the sample, or a negative value for an invalid input or a non-finite measurement. `trax_kf_reset()` re-seeds the filter to x_0 / P_0. A two-input model (Attitude + Gyro Bias) takes one `trax_kf_update()` per input: input 0 is `gyro_rate`, input 1 is `accel_angle`.

The numbers transfer directly, because Q is a continuous-time intensity (rate-independent) and the measurement updates are scalar (no matrix inversion, no dynamic sizing).

Precision	`float` by default. Ask for `double` at export, or define `TRAX_KF_REAL` as `double` when compiling `trax_kf.c` and everything that includes the header
Limits	3 states (`TRAX_KF_MAX_STATES`), 2 inputs (`TRAX_KF_MAX_INPUTS`) — enough for the three studio models
Models	`TRAX_KF_SCALAR`, `TRAX_KF_CONSTANT_VELOCITY`, `TRAX_KF_ATTITUDE_BIAS` — the studio's presets. Custom models are not supported

11. Troubleshooting

Symptom	Cause	Fix
Toolbox → Estimator opens an upgrade page	DSP not licensed on this tier	Professional license required (Menu → Manage License)
Variable dropdown is empty	The probe exposes no **stream** variables	The estimator needs `TRAX_STREAM_*` data — async `TRAX_VAR_SET` variables are not eligible (Chapter 12)
The variable you want is missing	It is a virtual variable (math channel or another estimate)	Feedback loops are blocked by design — feed the estimator raw probe data
"Input … has no variable selected."	An input row is unbound	Pick a probe and a variable in every row
"Could not allocate virtual variables…"	No free virtual varIds / stream TIDs on the output probe	Delete unused estimators or math channels on that probe
Run does nothing and reports no error	The license check inside the worker failed	Check the log; re-validate the license
Samples climb, Rate reads 0	The probe stopped streaming	The rate decays to zero on purpose so a dead stream is visible
Estimate is offset from, or moves opposite to, the plotted source	Formula mismatch — raw counts filtered against a converted display channel	Pick the same formula in the input row that the Scope channel uses
Trace breaks constantly	Input gaps, or the store is evicting samples faster than the estimator polls	Reduce the number of live streams, or raise the RAM budget
Estimate flat-lines then jumps	A gap longer than 5 s reset the filter	Expected. Shorter outages are predicted across
NIS ≫ 1 all the time	Over-confident tuning	Raise R, or raise Q on the state that actually moves (§6)
Estimate lags visibly	Q too small relative to R	Raise Q by a decade and watch the trace
Estimate does not follow a real step at first, then jumps to it	The outlier gate refused the step until **Refuse at most (s)** ran out	Expected for a short hold. Shorten the hold, or raise the gate
Estimate looks like the raw signal	Q too large — the filter trusts the sensor completely	Lower Q by a decade
Estimator is stopped after reopening the workspace	Instances always restore **stopped**	Press Run once the probe is streaming
The estimate is not in the recording	Outputs are never recorded, by design	Record the raw inputs and re-run the estimator over the recording

See also

- Chapter 11 — Variables & Scope — conversion formulas, pre-filters, math channels, plotting the results
- Chapter 12 — Stream Variables & Spectrum Analysis — producing the stream variables the estimator consumes
- Chapter 17 — Triggers — same studio pattern; trigger on an estimated state like any other variable
- Chapter 18 — Multi-Probe Synchronization — required before fusing inputs from two boards
- Chapter 22 — Gyro View — bind an estimated attitude to the 3-D viewer
- Chapter 23 — TraxAI — connecting the model that runs `tune_estimator`, `apply_estimator` and `export_estimator`