Yes. Here’s the “tell-it-like-it-is” version:

Speaker calibration is mostly physics + geometry (you can measure the room with a mic and fix delays/levels/phase, etc.).
Headphone “calibration” for binaural/ambisonics is mostly *psychoacoustics* + *personal anatomy* (HRTF/BRIR), and you generally **cannot measure it accurately with just a normal mic**. So you need a workflow that’s honest about what’s measurable vs what’s selectable.

Below is a pragmatic research-backed plan that fits your **LocusQ** architecture and the real constraints of AirPods/Sony/Dolby.

---

1) What “headphone calibration” actually means (in practice)

There are three separate problems people conflate:

A) Headphone frequency-response compensation (measurable-ish)

Goal: get the headphone closer to a known target (Harman, diffuse field, etc.) so your binaural cues aren’t being mangled by headphone coloration.

Open tools / ecosystems:

- **AutoEq** (open source) generates parametric EQ from large measurement databases.
- **REW (Room EQ Wizard)** is free and can measure audio devices, generate EQ, and analyze responses.
- **ASH Toolset** (open source) explicitly targets *headphone correction* + *binaural room simulation* and can output filters (WAV / SOFA) and integrate with Equalizer APO.

Hard truth: without a coupler (or in-ear mics), user-generated measurements will be noisy and inconsistent. So your best UX is: *ship curated profiles* (AutoEq/known sources), and allow “custom EQ” import.

B) HRTF / BRIR selection (often not measurable by end users)

Goal: pick an HRTF/BRIR that matches the listener enough to get stable externalization + elevation + front/back.

The standard data container is SOFA (AES69), which stores HRTFs and also BRIRs/DRIRs.
You already have the right instinct: make **SOFA** a first-class "device profile" input.

C) Head tracking alignment (measurable via sensors; alignment needs calibration)

Goal: "front" stays in front when the head rotates.

This is typically:

- a sensor feed (quaternion/yaw-pitch-roll)
- smoothing + dropout handling
- **a yaw offset** calibration step (user faces forward, press "Set Forward")

You already scoped this correctly as "companion app → IPC" territory (because plugin sandboxes + DAW hosts). Your own doc notes this constraint clearly for Apple's stack.

docs(review): expand Section 0b...

---

2) Open-source binaural + ambisonics toolchain (the ones worth building around)

Steam Audio (best "drop-in" binaural in a plugin)

Steam Audio's C API provides binaural rendering and supports loading **custom HRTFs from SOFA**.

docs(review): expand Section 0b...

o already calls this out as the best practical post-v1 binaural path.

SPARTA / SAF (ambisonics-heavy, very credible)

- **SPARTA** provides binaural ambisonic decoding, supports **SOFA HRIR import**, and even supports **head tracking via OSC**.
- **SAF** is the underlying framework (C/C++) for ambisonics encode/decode, HRIR processing, etc.

IEM Plugin Suitein JUCE)

Open-source ambisonics suite (JUCE-based) including decoders for headphones and loudspeaker layouts.

SOFA ecosystem + HRTF generation (if you want "personalization" later)

- SOFA conventions hub (standard + tooling pointers).
- **Mesh2HRTF** can generate SOFA HRTFs from 3D meshes (researchy, but real).

---

3) How this maps cleanly into LocusQ (you're already 70% there)

Your recent CALIBRATE work already has:

- **Topologies** including ambisonic, binaural_headphone, and Atmos-style layouts

  CALIBRATE: sync legacy config a...

- **Monitoring paths** including steam_binaural and virtual_binaural

  CALIBRATE: sync legacy config a...

- **Device profiles** including airpods_pro_2, sony_wh1000xm5, and custom_sofa

  CALIBRATE: sync legacy config a...

That's the correct product model. Now make it *do real work*:

Proposed CALIBRATE v2 "Headphones" flow (minimal, useful, shippable)

1. **Select Monitoring Path**
   - steam_binaural (plugin-internal binaural using Steam Audio)
   - virtual_binaural (OS/renderer externalization path; basically "trust the system")
2. **Select Device Profile**

- Generic / AirPods / Sony / Custom SOFA

3. **Headphone EQ (optional but high value)**
   - Built-in presets sourced from AutoEq datasets (mode

     CALIBRATE: sync legacy config a...

     c EQ or import correction IR (WAV)
4. **HRTF / BRIR**
   - Default HRTF (Ste

     CALIBRATE: sync legacy config a...

     SOFA (user-provided)
5. **Verification**
   - Quick perceptual test suite: front/back, e

     CALIBRATE: sync legacy config a...

     ion check, mono-compat check
   - Store: selected HRTF + yaw offset + headphone EQ profile per device/profile

This complements your existing **speaker+mic pulse** calibration: speakers get measured; headphones get *selected* + *compensated* + *verified*.

---

4) AirPods Pro / Sony WH-1000XM5 reality check (and how to still support them)

- **AirPods Spatial Audio / head tracking**: Apple's "best" pipeline is OS-level; you can't realistically embed PHASE inside a VST/AU processBlock, and you already documented that.

  docs(review): expand Section 0b...
- **Sony / Dolby Atmos**: similarly, the "Atmos binauralization" is typically handled by OS/renderer/device pipeline, not your plugin.

So your support strategy should be:

Mode A: “System binauralization” (fast path)

Let LocusQ output multichannel (quad/5.1/7.x.x) correctly, and instruct users to monitor through the OS/device spatial pipeline. This is what your doc already recommends as the v1 pragmatic approach.

docs(review): expand Section 0b...

Mode B: “LocusQ internal binaural” (serious path)

Steam Audio binaural render to stereo, works on any headphones, and becomes the backbone for calibration (SOFA + EQ + verification).

Mode C: “Head-tracked internal binaurads head pose (AirPods via Apple APIs, others via OSC/IMU), sends orientation to plugin, plugin rotates scene / ambisonics field and renders binaural. SPARTA proves the OSC head-tracking pattern is viable.

---

5) Strong recommendation for your next concrete step in LocusQ

Implement **one** clean, testable calibration deliverable:

“Binaural Monitoring (Steam Audio) + Custom SOFA +

docs(review): expand Section 0b...

tion tests”

Why:

- It’s plugin-native (no OS magic required).
- It’s compatible with your existing CALIBRATE topology/device/monitoring model.

  CALIBRATE: sync legacy config a...
- It gives you a calibration story that scales from “generic headphones” to “power users with SOFA files.”

If you do only one thing beyond that: add an OSC input for head pose (even before the AirPods companion exists). That de-risks the whole v2 head-tracking roadmap because the plugin side is done.

If you want this to move from research to implementation, the next step is mechanical: define a **HeadphoneCalibrationProfile** contract (device EQ + SOFA reference + yaw offset + monitoring path + last verification scores) and wire it into the CALIBRATE profile library you already built. That turns “headphone calibration” from a vague idea into a reproducible artifact that your QA harness can regression-test.

CALIBRATE: sync legacy config a...

Sources

Yes. The fastest way to make this real (and not a “calibration vibe”) is to turn it into **one concrete subsystem** inside LocusQ:

Headphone Calibration = (Headphone EQ compensation) + (HRTF/BRIR selection via SOFA) + (Head-tracking yaw offset) + (Verification tests) + (Preset/profile management).

Below is a practical implementation plan you can execute in LocusQ with minimal scope creep.

---

1) Define the deliverable (what “done” looks like)

v1 (ship this)

- A **Headphone Calibration panel** in CALIBRATE.
- A **profile object** saved/loaded with the plugin state.
- A **binaural monitoring path**:
  - steam_binaural (internal binaural to stereo)
- Inputs:
  - **Headphone EQ**: off / preset / custom param EQ / FIR IR file
  - **HRTF**: default / SOFA import

    - **Yaw offset**: “Set Forward” button
- A **verification suite** (audio test signals + user scoring) that saves results into the profile.

v2 (after v1 is stable)

- OSC head-tracking input (works with SPARTA-style workflows immediately).
- Companion app bridge (AirPods etc.) only after OSC path is proven.

---

2) Add one data model: HeadphoneCalibrationProfile

This is the keystone. Everything else plugs into it.

Fields you want:

- device_id (e.g., airpods_pro_2, sony_wh1000xm5, generic, custom)
- monitoring_path (steam_binaural vs system_spatial)
- eq:
    - type: none | peq | fir
    - peq_bands: list of biquad params
    - fir_path or embedded FIR coefficients (your choice; I’d store path + hash + optional embedded small FIR)
- hrtf:
    - type: default | sofa
    - sofa_path + metadata (sample rate, tap length, az/el coverage)
- head_tracking:
    - enabled
    - yaw_offset_deg (or quaternion)
    - smoothing params
- verification_results:

    - last run timestamp
    - scores: externalization, front/back, elevation, timbre, comfort
    - notes

This becomes part of your plugin state contract and can be regression-tested in audio-dsp-qa-harness.

---

3) Wire it into audio: the "monitoring chain"

For steam_binaural

Process should look like:

1. Your scene output is either:
    - Ambisonics (recommended internal format), OR
    - Multi-channel bed (quad/5.1/7.x)
2. Convert to what the binaural renderer expects (likely ambisonics or direct source rendering).
3. Apply **HRTF** rendering (Steam Audio).
4. Apply **Headphone EQ** last (compensate the playback system).
5. Output stereo.

Why EQ last: you want to compensate the headphone as a playback device after binauralization, not distort the spatial cues before HRTF.

For system_spatial

- You output multi-channel correctly and do **no binauralization**.
- Still allow **headphone EQ** optionally (but be careful: EQing a system binaural pipeline can help or harm; make it opt-in).

---

4) Implement SOFA import without boiling the ocean

You don't need a full SOFA ecosystem on day 1. You need:

- Load SOFA file
- Extract HRIRs (left/right per direction)
- Convert to the format required by your binaural renderer (or precompute your own convolution filters)

If Steam Audio can accept SOFA directly in your chosen integration path, use that. If not, build a thin converter.

Hard rule for sanity: constrain supported SOFA at first:

- HRIR only (not BRIR)
- one sample rate (or resample once at load)
- max tap length (e.g., 1024–4096)
- defined coordinate system checks

---

5) Verification suite (this makes calibration not fake)

Add a “VERIFY” button that runs a short guided test (60–90 seconds total):

- **Front/Back:** alternating bursts at 0° and 180° (randomized)
- **Elevation sanity:** a few elevated sources (if your HRTF supports it)
- **Externalization:** a speech or noise source at 1–2m distance
- **Mono compatibility:** sum-to-mono check and warn if it collapses weirdly
- **Timbre check:** sweep or pink noise; ask user if it’s harsh/hollow

Store scores into the profile. This gives you:

- a UX win
- a debug handle
- a QA target (“verification score should not regress across builds” for defaults)

---

6) What I can do for you immediately (pick the output you want)

I can produce any of these in one pass:

1. **Exact JSON schema + C++ struct mapping** for HeadphoneCalibrationProfile (matches your StateContract style)
2. **CALIBRATE UI spec** (controls, copy, state transitions)
3. **Implementation blueprint**:
   - new modules/classes (HrtfLoaderSofa, HeadphoneEq, BinauralMonitor, CalibrationVerifier)
   - where to hook into your existing audio graph
4. **QA harness plan**:
   - golden renders for default HRTF + EQ off/on
   - invariants (no allocations, stable latency, deterministic load)
5. **Steam Audio integration notes** tailored to JUCE/VST3/AU threading/RT safety

---

My recommendation (because focus wins)

Start with **(1) schema** + **(3) implementation blueprint** + **(4) QA plan**. UI can lag one iteration behind audio correctness.

To proceed, I'll assume your current LocusQ audio graph already has:

- a notion of "topology" (mono/stereo/quad/5.1/Atmos)
- a monitoring stage you can insert into
- the CALIBRATE profile store you referenced earlier

Next message from me can be the full **HeadphoneCalibrationProfile schema** + **C++ header** + **integration points** (ready to paste into the repo) unless you'd rather start from the UI spec.

Proceeding. I'm going to give you **paste-ready contracts** (schema + header skeleton) plus the **exact wiring plan** that matches how your harness wants state to look and how your CALIBRATE v2 spec already models (topology, monitoring path, device profile).

Documentation/plans/bl-026-cali...

---

1) The core decision: how this fits your existing state + QA contract

Your harness's canonical snapshot format only supports:

- **primitive parameters** (float/int/bool/choice)
- **blobs** (base64)
  ...and it enforces **lexicographic ordering, no unknown IDs, deterministic/idempotent migrations**.

  Add reusable qa_state module wi...

So: **don't try to store a rich nested JSON "profile" as a parameter value**. Instead:

- store *all "switches" and numbers* as parameters
- store *non-primitive payloads* (SOFA reference, FIR coefficients, PEQ preset metadata) as **blobs**

That keeps you compliant with your own rules and makes the QA harness happy.

---

2) Parameter ID contract (the minimal set to ship v1)

You already standardized these CALIBRATE profile axes in UI/plan:

- cal_topology_profile
- cal_monitoring_path
- cal_device_profile

  Documentation/plans/bl-026-cali...

Now add the headphone calibration controls (IDs are lowercase, stable, ASCII-safe):

Monitoring / calibration selection (CALIBRATE)

- cal_topology_profile — choice
- cal_monitoring_path — choice (speakers, stereo_downmix, steam_binaural, virtual_binaural)

  CALIBRATE: sync legacy config a…
- cal_device_profile — choice (generic, airpods_pro_2, sony_wh1000xm5, custom_sofa)

  CALIBRATE: sync legacy config a…

Headphone calibration (new)

- hp_enabled — bool (master enable)
- hp_yaw_offset_deg — float (Set Forward writes this)
- hp_tracking_enabled — bool (v1 can exist even if tracking source is “none”)
- hp_eq_mode — choice (off, peq, fir)
- hp_peq_band_count — int (0..8)
- hp_peq_b1_freq_hz, hp_peq_b1_gain_db, hp_peq_b1_q — float
- … repeat for bands 2..8
- hp_fir_taps — int
- hp_fir_latency_samples — int (explicit; don’t infer later)
- hp_hrtf_mode — choice (default, sofa_ref)
- hp_verify_last_score — float (0..1)
- hp_verify_frontback — float (0..1)
- hp_verify_elevation — float (0..1)
- hp_verify_externalization — float (0..1)

Blobs (new)

- hp_sofa_ref — base64(JSON) with { "uri": "...", "sha256": "...", "label": "..." }
- hp_fir_coeffs_f32 — base64(raw float32 little-endian taps)
- hp_peq_preset_meta — base64(JSON) optional, e.g. { "source": "AutoEq", "model": "Sony WH-1000XM5", "target": "Harman" }

This is intentionally **boring and testable**.

---

3) JSON schema fragment (plugin-owned) for the new IDs

Your harness schema (qa_state_schema_v1.json) stays unchanged.

Merge pull request #6 from josh...

What you update is your plugin's **migration catalog** + **known parameter list** + **goldens**.

Here's the **migration_rules.json** style fragment you'll add (same structure you already use).

state/migration_rules

```
JSON{
  "plugin_id": "com.joshband.locusq",
  "target_state_version": "2.1.0",
  "known_parameters": [
"bypass",
"delay_time_ms",
"master_gain_db",
"mod_mix",

"cal_topology_profile",
"cal_monitoring_path",
"cal_device_profile",

"hp_enabled",
"hp_tracking_enabled",
"hp_yaw_offset_deg",
"hp_eq_mode",
"hp_peq_band_count",

"hp_peq_b1_freq_hz", "hp_peq_b1_gain_db", "hp_peq_b1_q",
"hp_peq_b2_freq_hz", "hp_peq_b2_gain_db", "hp_peq_b2_q",
"hp_peq_b3_freq_hz", "hp_peq_b3_gain_db", "hp_peq_b3_q",
"hp_peq_b4_freq_hz", "hp_peq_b4_gain_db", "hp_peq_b4_q",
"hp_peq_b5_freq_hz", "hp_peq_b5_gain_db", "hp_peq_b5_q",
"hp_peq_b6_freq_hz", "hp_peq_b6_gain_db", "hp_peq_b6_q",
"hp_peq_b7_freq_hz", "hp_peq_b7_gain_db", "hp_peq_b7_q",
"hp_peq_b8_freq_hz", "hp_peq_b8_gain_db", "hp_peq_b8_q",

"hp_fir_taps",
"hp_fir_latency_samples",
"hp_hrtf_mode",

"hp_verify_last_score",
"hp_verify_frontback",
"hp_verify_elevation",
"hp_verify_externalization"
  ],
  "steps": [
    {
      "from_version": "2.0.0",
      "to_version": "2.1.0",
      "renames": [],
      "removed": [],
      "introduced": [
        { "id": "cal_topology_profile", "type": "choice", "default": 0 },
        { "id": "cal_monitoring_path", "type": "choice", "default": 0 },
        { "id": "cal_device_profile", "type": "choice", "default": 0 },

        { "id": "hp_enabled", "type": "bool", "default": 0 },
        { "id": "hp_tracking_enabled", "type": "bool", "default": 0 },
        { "id": "hp_yaw_offset_deg", "type": "float", "default": 0.0 },
        { "id": "hp_eq_mode", "type": "choice", "default": 0 },
        { "id": "hp_peq_band_count", "type": "int", "default": 0 },

        { "id": "hp_fir_taps", "type": "int", "default": 0 },
        { "id": "hp_fir_latency_samples", "type": "int", "default": 0 },
        { "id": "hp_hrtf_mode", "type": "choice", "default": 0 },

        { "id": "hp_verify_last_score", "type": "float", "default": 0.0 },
        { "id": "hp_verify_frontback", "type": "float", "default": 0.0 },
        { "id": "hp_verify_elevation", "type": "float", "default": 0.0 },
        { "id": "hp_verify_externalization", "type": "float", "default": 0.0 }
      ]
    }
  ]
}
```

Notes:

- choice defaults are integers (index into your string table).
- You can introduce PEQ band params later (or now) — they're deterministic either way.

---

4) Paste-ready C++ header skeleton: HeadphoneCalibrationProfile.h

This is designed to sit at your plugin boundary and be trivially adaptable to APVTS + qa_state.

```
C++#pragma once

#include<array>
#include<cstdint>
#include<optional>
#include<string>
#include<vector>

namespacelocusq {

enumclassHpEqMode : int {
Off = 0,
Peq = 1,
Fir = 2,
};

enumclassHpHrtfMode : int {
Default = 0,
SofaRef = 1,
};

structPeqBand {
```

```
float freqHz = 1000.0f;
float gainDb = 0.0f;
float q = 0.707f;
bool enabled = false;
};

structSofaRef {
// Stored as blob JSON in state: {"uri":"...","sha256":"...","label":"..."}
   std::string uri;
   std::string sha256;
   std::string label;
};

structFirFilter {
// Stored as blob float32 taps (little-endian) in state.
   std::vector<float> taps;
int latencySamples = 0;
};

structVerificationScores {
float lastScore = 0.0f;
float frontBack = 0.0f;
float elevation = 0.0f;
float externalization = 0.0f;
};

structHeadphoneCalibrationProfile {
bool enabled = false;
bool trackingEnabled = false;
float yawOffsetDeg = 0.0f;

HpEqMode eqMode = HpEqMode::Off;
int peqBandCount = 0;
   std::array<PeqBand, 8>peqBands{};

FirFilter fir;

HpHrtfMode hrtfMode = HpHrtfMode::Default;
   std::optional<SofaRef> sofaRef;

VerificationScores verify;
};
```

```
// ---------------------------------------------------------------------------
// State bridging contract: primitive parameters + blobs only.
// You'll implement these against your chosen state API:
// - APVTS for runtime
// - qa::state::StateSnapshot for QA harness regression
// ---------------------------------------------------------------------------

structStateParamWriter {
virtual~StateParamWriter() = default;
virtualvoid setBool(constchar* id, boolv) = 0;
virtualvoid setInt(constchar* id, intv) = 0;
virtualvoid setFloat(constchar* id, floatv) = 0;
virtualvoid setChoice(constchar* id, intindex) = 0;

// base64-encoded blob payload; caller provides canonical content.
virtualvoid setBlobBase64(constchar* id, const std::string& base64) = 0;
};

structStateParamReader {
virtual~StateParamReader() = default;
virtualbool getBool(constchar* id, boolfallback) const = 0;
virtualint getInt(constchar* id, intfallback) const = 0;
virtualfloat getFloat(constchar* id, floatfallback) const = 0;
virtualint getChoice(constchar* id, intfallbackIndex) const = 0;
virtual std::optional<std::string> getBlobBase64(constchar* id) const = 0;
};

classHeadphoneCalibrationCodec {
public:
staticvoid writeToState(constHeadphoneCalibrationProfile& hp, StateParamWriter& out);
staticHeadphoneCalibrationProfile readFromState(constStateParamReader& in);

// Helpers you implement in .cpp
static std::string encodeSofaRefBlobBase64(constSofaRef& ref);
static std::optional<SofaRef> decodeSofaRefBlobBase64(const std::string& b64);

static std::string encodeFirBlobBase64(const std::vector<float>& taps);
static std::vector<float> decodeFirBlobBase64(const std::string& b64);
};

} // namespace locusq
```

Why this shape:

- It’s **host-agnostic** like your harness boundary wants.

  Add reusable qa_state module wi…

- It enforces “profile = parameters + blobs” with deterministic ordering under the hood.

---

5) Where to hook this in LocusQ (specific to your existing CALIBRATE v2 architecture)

A) UI already has the selector IDs and labels

Your JS already lists topology IDs, monitoring path IDs, and device profile IDs.

CALIBRATE: sync legacy config a…

So the immediate wiring is:

- Create APVTS params for cal_topology_profile, cal_monitoring_path, cal_device_profile
- Mirror the string tables used in UI (same ordering), because choice is index-based

B) Scene snapshot diagnostics are already “expecting” headphone mode/profile signals

Your CALIBRATE status path computes requested/active headphone mode/profile (and shows validation chips).

CALIBRATE: sync legacy config a…

So the DSP/renderer side needs to publish:

- rendererHeadphoneModeRequested/Active
- rendererHeadphoneProfileRequested/Active
- Steam init diagnostics when binaural requested (your plan already says so).

  Documentation/plans/bl-026-cali…

C) Audio graph insertion point (the cleanest)

Implement a “monitoring stage” with this deterministic rule:

- If cal_monitoring_path == steam_binaural:
    a. Render spatial scene (your internal format)
    b. Binauralize to stereo
    c. Apply headphone EQ (PEQ/FIR)
    d. Output stereo
- If virtual_binaural:
    - Do **not** binauralize, but still publish requested/active + profile diagnostics (your UI wants it).

      Documentation/plans/bl-026-cali...

This aligns with your existing “requested/active/stage divergence” philosophy.

docs: sync backlog and validati...

---

6) QA harness integration plan (deterministic, regression-friendly)

You already have CI enforcing state migration goldens.

Add versioned plugin-state migr...

So do this:

1. **Bump plugin state_version** to 2.1.0.
2. Update state/migration_rules.json to add the introduced parameters (above).
3. Add **new golden preset** under state/goldens/2.1.0/ with:
    - cal_monitoring_path = steam_binaural
    - cal_device_profile = generic

    - hp_eq_mode = peq, one enabled band (e.g., gentle tilt)
    - hp_hrtf_mode = default
4. Add a migration test that asserts:
    - migrate(2.0.0 preset) produces all new IDs
    - second migration is identical (idempotence)

      Add reusable qa_state module wi...
    - canonical hash stable

That gives you “no silent drift” protection.

---

7) The one thing I’m *not* doing yet (on purpose)

I am **not** telling you to embed full SOFA IR data inside plugin state in v1. That’s a file-size and UX grenade.

v1: store a **SOFA reference blob** (uri + hash + label), and if missing at load time:

- fall back to default HRTF
- set a diagnostic flag so CALIBRATE shows “profile requested != active” (which you already built the UI to surface).

  docs: sync backlog and validati...

If you want the next step fully mechanical, I can now generate the **.cpp implementation** for HeadphoneCalibrationCodec (base64 packing, canonical JSON for blobs, band mapping, etc.) plus the **APVTS parameter registration snippet** that matches your UI choice tables exactly.

Sources

Good. We’re going to make this deterministic, RT-safe, and QA-friendly.

Below is:

1. **APVTS parameter registration snippet** (complete, consistent ordering)

2. **HeadphoneCalibrationCodec.cpp** (fully implemented: state ↔️ struct, base64 blobs, canonical JSON)
3. Notes on RT safety + migration hygiene

No fluff.

---

1) APVTS Parameter Registration

Assume you have something like:

```
C++juce::AudioProcessorValueTreeState::ParameterLayoutcreateParameterLayout();
```

Add this inside that layout builder.

Choice string tables (must match UI ordering)

```
C++staticconst juce::StringArraykCalTopologyChoices {
"mono",
"stereo",
"quad",
"5_1",
"7_1_2",
"ambisonic"
};

staticconst juce::StringArray kCalMonitoringChoices {
"speakers",
"stereo_downmix",
"steam_binaural",
"virtual_binaural"
};

staticconst juce::StringArray kCalDeviceChoices {
"generic",
"airpods_pro_2",
"sony_wh1000xm5",
"custom_sofa"
};

staticconst juce::StringArray kHpEqModeChoices {
"off",
"peq",
"fir"
};

staticconst juce::StringArray kHpHrtfModeChoices {
"default",
"sofa_ref"
};
```

Parameter registration

```
C++juce::AudioProcessorValueTreeState::ParameterLayoutcreateParameterLayout()
{
    std::vector<std::unique_ptr<juce::RangedAudioParameter>>params;

// --- CALIBRATE axis ---
params.push_back(std::make_unique<juce::AudioParameterChoice>(
"cal_topology_profile", "CAL Topology",
kCalTopologyChoices, 0));

params.push_back(std::make_unique<juce::AudioParameterChoice>(
"cal_monitoring_path", "CAL Monitoring Path",
kCalMonitoringChoices, 0));

params.push_back(std::make_unique<juce::AudioParameterChoice>(
"cal_device_profile", "CAL Device Profile",
kCalDeviceChoices, 0));

// --- Headphone master ---
params.push_back(std::make_unique<juce::AudioParameterBool>(
"hp_enabled", "HP Enabled", false));

params.push_back(std::make_unique<juce::AudioParameterBool>(
"hp_tracking_enabled", "HP Tracking Enabled", false));

params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_yaw_offset_deg", "HP Yaw Offset",
        juce::NormalisableRange<float>(-180.f, 180.f, 0.01f), 0.f));

params.push_back(std::make_unique<juce::AudioParameterChoice>(
"hp_eq_mode", "HP EQ Mode",
kHpEqModeChoices, 0));
```

```
params.push_back(std::make_unique<juce::AudioParameterChoice>(
"hp_hrtf_mode", "HP HRTF Mode",
kHpHrtfModeChoices, 0));

params.push_back(std::make_unique<juce::AudioParameterInt>(
"hp_peq_band_count", "HP PEQ Band Count", 0, 8, 0));

// --- 8 PEQ bands ---
for (inti = 1; i<=8; ++i)
   {
autosuffix = juce::String(i);

params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_peq_b"+suffix+"_freq_hz",
"HP PEQ B"+suffix+" Freq",
          juce::NormalisableRange<float>(20.f, 20000.f, 0.01f, 0.5f),
1000.f));

params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_peq_b"+suffix+"_gain_db",
"HP PEQ B"+suffix+" Gain",
          juce::NormalisableRange<float>(-24.f, 24.f, 0.01f),
0.f));

params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_peq_b"+suffix+"_q",
"HP PEQ B"+suffix+" Q",
          juce::NormalisableRange<float>(0.1f, 10.f, 0.001f, 0.5f),
0.707f));
   }

// --- FIR metadata (coeffs stored as blob externally) ---
params.push_back(std::make_unique<juce::AudioParameterInt>(
"hp_fir_taps", "HP FIR Taps", 0, 8192, 0));

params.push_back(std::make_unique<juce::AudioParameterInt>(
"hp_fir_latency_samples", "HP FIR Latency Samples", 0, 8192, 0));

// --- Verification ---
params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_verify_last_score", "HP Verify Score",
        juce::NormalisableRange<float>(0.f, 1.f), 0.f));
```

```
params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_verify_frontback", "HP Verify FrontBack",
        juce::NormalisableRange<float>(0.f, 1.f), 0.f));

params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_verify_elevation", "HP Verify Elevation",
        juce::NormalisableRange<float>(0.f, 1.f), 0.f));

params.push_back(std::make_unique<juce::AudioParameterFloat>(
"hp_verify_externalization", "HP Verify Externalization",
        juce::NormalisableRange<float>(0.f, 1.f), 0.f));

return { params.begin(), params.end() };
}
```

Blobs (hp_sofa_ref, hp_fir_coeffs_f32) live in your **ValueTree state** directly, not as AudioParameters.

---

2) HeadphoneCalibrationCodec.cpp

Paste-ready implementation. Assumes:

- You have JUCE available
- You use JUCE base64 utilities
- JSON via juce::var + juce::JSON

```
C++#include"HeadphoneCalibrationProfile.h"
#include<juce_core/juce_core.h>

namespacelocusq {

//==========================================================
// Helpers
//==========================================================

static juce::StringtoB64(const juce::MemoryBlock& mb)
```

```
{
return juce::Base64::toBase64(mb.getData(), mb.getSize());
}

static juce::MemoryBlockfromB64(const juce::String& b64)
{
   juce::MemoryBlockmb;
   juce::Base64::convertFromBase64(mb, b64);
returnmb;
}

//==========================================================
// SOFA REF
//==========================================================

std::string HeadphoneCalibrationCodec::encodeSofaRefBlobBase64(constSofaRef& ref)
{
   juce::DynamicObject::Ptrobj = new juce::DynamicObject();
obj->setProperty("uri", ref.uri);
obj->setProperty("sha256", ref.sha256);
obj->setProperty("label", ref.label);

   juce::Stringjson = juce::JSON::toString(juce::var(obj), true);

   juce::MemoryBlockmb(json.toRawUTF8(), json.getNumBytesAsUTF8());
returntoB64(mb).toStdString();
}

std::optional<SofaRef>
HeadphoneCalibrationCodec::decodeSofaRefBlobBase64(const std::string& b64)
{
   juce::MemoryBlockmb = fromB64(b64);

autojsonText = juce::String::fromUTF8(
static_cast<constchar*>(mb.getData()),
      (int)mb.getSize());

autovar = juce::JSON::parse(jsonText);
if (!var.isObject())
return std::nullopt;

auto* obj = var.getDynamicObject();
```

```
SofaRefref;
ref.uri = obj->getProperty("uri").toString().toStdString();
ref.sha256 = obj->getProperty("sha256").toString().toStdString();
ref.label = obj->getProperty("label").toString().toStdString();
returnref;
}

//=====================================================
// FIR
//=====================================================

std::string HeadphoneCalibrationCodec::encodeFirBlobBase64(
const std::vector<float>& taps)
{
   juce::MemoryBlockmb(taps.data(), taps.size() *sizeof(float));
returntoB64(mb).toStdString();
}

std::vector<float>
HeadphoneCalibrationCodec::decodeFirBlobBase64(const std::string& b64)
{
   juce::MemoryBlockmb = fromB64(b64);
size_tcount = mb.getSize() /sizeof(float);

   std::vector<float>taps(count);
   std::memcpy(taps.data(), mb.getData(), count*sizeof(float));
returntaps;
}

//=====================================================
// STATE WRITE
//=====================================================

void HeadphoneCalibrationCodec::writeToState(
constHeadphoneCalibrationProfile& hp,
StateParamWriter& out)
{
out.setBool("hp_enabled", hp.enabled);
out.setBool("hp_tracking_enabled", hp.trackingEnabled);
out.setFloat("hp_yaw_offset_deg", hp.yawOffsetDeg);

out.setChoice("hp_eq_mode", (int)hp.eqMode);
out.setChoice("hp_hrtf_mode", (int)hp.hrtfMode);

out.setInt("hp_peq_band_count", hp.peqBandCount);

for (inti = 0; i<8; ++i)
    {
constauto& b = hp.peqBands[i];
const std::stringidx = std::to_string(i+1);

out.setFloat(("hp_peq_b"+idx+"_freq_hz").c_str(), b.freqHz);
out.setFloat(("hp_peq_b"+idx+"_gain_db").c_str(), b.gainDb);
out.setFloat(("hp_peq_b"+idx+"_q").c_str(), b.q);
    }

out.setInt("hp_fir_taps", (int)hp.fir.taps.size());
out.setInt("hp_fir_latency_samples", hp.fir.latencySamples);

if (!hp.fir.taps.empty())
out.setBlobBase64("hp_fir_coeffs_f32",
encodeFirBlobBase64(hp.fir.taps));

if (hp.sofaRef.has_value())
out.setBlobBase64("hp_sofa_ref",
encodeSofaRefBlobBase64(*hp.sofaRef));

out.setFloat("hp_verify_last_score", hp.verify.lastScore);
out.setFloat("hp_verify_frontback", hp.verify.frontBack);
out.setFloat("hp_verify_elevation", hp.verify.elevation);
out.setFloat("hp_verify_externalization", hp.verify.externalization);
}

//==========================================================
// STATE READ
//==========================================================

HeadphoneCalibrationProfile
HeadphoneCalibrationCodec::readFromState(constStateParamReader& in)
{
HeadphoneCalibrationProfilehp;

hp.enabled = in.getBool("hp_enabled", false);
hp.trackingEnabled = in.getBool("hp_tracking_enabled", false);
hp.yawOffsetDeg = in.getFloat("hp_yaw_offset_deg", 0.f);

hp.eqMode = (HpEqMode)in.getChoice("hp_eq_mode", 0);
hp.hrtfMode = (HpHrtfMode)in.getChoice("hp_hrtf_mode", 0);

hp.peqBandCount = in.getInt("hp_peq_band_count", 0);

for (inti = 0; i<8; ++i)
    {
auto& b = hp.peqBands[i];
const std::stringidx = std::to_string(i+1);

b.freqHz = in.getFloat(("hp_peq_b"+idx+"_freq_hz").c_str(), 1000.f);
b.gainDb = in.getFloat(("hp_peq_b"+idx+"_gain_db").c_str(), 0.f);
b.q = in.getFloat(("hp_peq_b"+idx+"_q").c_str(), 0.707f);
    }

hp.fir.latencySamples = in.getInt("hp_fir_latency_samples", 0);

if (autoblob = in.getBlobBase64("hp_fir_coeffs_f32"))
hp.fir.taps = decodeFirBlobBase64(*blob);

if (autoblob = in.getBlobBase64("hp_sofa_ref"))
hp.sofaRef = decodeSofaRefBlobBase64(*blob);

hp.verify.lastScore = in.getFloat("hp_verify_last_score", 0.f);
hp.verify.frontBack = in.getFloat("hp_verify_frontback", 0.f);
hp.verify.elevation = in.getFloat("hp_verify_elevation", 0.f);
hp.verify.externalization =
in.getFloat("hp_verify_externalization", 0.f);

returnhp;
}

} // namespace locusq
```

3) Important Engineering Notes (Read This)

1) No allocations in processBlock

- Decode SOFA / FIR **only on state change**, never in audio thread.
- Pre-build FIR convolution object offline.
- Swap atomically (lock-free pointer exchange).

2) FIR latency must be explicit

You already track hp_fir_latency_samples.
When FIR changes:

- call setLatencySamples()
- notify host

No hidden latency. Ever.

3) Canonical JSON

encodeSofaRefBlobBase64 must produce stable field ordering.
JUCE DynamicObject preserves insertion order. Keep insertion order fixed.

4) Migration discipline

After adding these:

- bump state_version
- update migration rules
- regenerate goldens
- assert idempotence

Your harness explicitly enforces that.

Add reusable qa_state module wi…

If you want the next layer, I'll give you:

- A **Steam Audio binaural wrapper class**

- A **PEQ implementation (biquad cascade, RT-safe)**
- A **minimal FIR convolution engine selection strategy** (partitioned vs direct, based on taps)

That’s the part that turns this from “state plumbing” into actual spatial credibility.

Sources

```
C++// SteamAudioBinaural.h
// Minimal, RT-safe wrapper for Steam Audio binaural monitoring in a JUCE plugin.
//
// This version targets *Ambisonics -> Stereo (binaural)* using
// IPLAmbisonicsDecodeEffect with binaural=true, plus optional listener rotation.
// API refs: Ambisonics Decode Effect, HRTF settings. :contentReference[oaicite:0]
{index=0}

#pragma once

#include<atomic>
#include<memory>
#include<mutex>
#include<string>
#include<vector>

#include<phonon.h>// Steam Audio C API

namespacelocusq {

//---------------------------------------------
// Utility: RAII holders for Steam Audio handles
//---------------------------------------------
structIplHrtfHandle {
IPLHRTF h = nullptr;
IplHrtfHandle() = default;
explicitIplHrtfHandle(IPLHRTFhh) : h(hh) {}
~IplHrtfHandle() { reset(); }
IplHrtfHandle(constIplHrtfHandle&) = delete;
IplHrtfHandle& operator=(constIplHrtfHandle&) = delete;
IplHrtfHandle(IplHrtfHandle&& o) noexcept : h(o.h) { o.h = nullptr; }
IplHrtfHandle& operator=(IplHrtfHandle&& o) noexcept {
if (this!= &o) { reset(); h = o.h; o.h = nullptr; }
return *this;
    }
void reset() {
if (h) iplHRTFRelease(&h);
h = nullptr;
    }
IPLHRTF get() const { returnh; }
};

structIplAmbiDecodeHandle {
IPLAmbisonicsDecodeEffect e = nullptr;
~IplAmbiDecodeHandle() { reset(); }
IplAmbiDecodeHandle(constIplAmbiDecodeHandle&) = delete;
IplAmbiDecodeHandle& operator=(constIplAmbiDecodeHandle&) = delete;
IplAmbiDecodeHandle(IplAmbiDecodeHandle&& o) noexcept : e(o.e) { o.e = nullptr; }
IplAmbiDecodeHandle& operator=(IplAmbiDecodeHandle&& o) noexcept {
if (this!= &o) { reset(); e = o.e; o.e = nullptr; }
return *this;
    }
void reset() {
if (e) iplAmbisonicsDecodeEffectRelease(&e);
e = nullptr;
    }
IPLAmbisonicsDecodeEffect get() const { returne; }
};

//---------------------------------------------
// SteamAudioBinaural
//---------------------------------------------
classSteamAudioBinauralfinal {
public:
structSettings {
int sampleRate = 48000;
int framesPerBuffer = 512;     // your plugin block size
int maxAmbisonicOrder = 3;     // e.g., 1->4ch, 2->9ch, 3->16ch
// Optional: normalization for custom HRTFs
IPLHRTFNormType normType = IPL_HRTFNORMTYPE_RMS;
float hrtfVolume = 1.0f;
    };

SteamAudioBinaural() = default;
~SteamAudioBinaural() { shutdown(); }
```

```
SteamAudioBinaural(constSteamAudioBinaural&) = delete;
SteamAudioBinaural& operator=(constSteamAudioBinaural&) = delete;

// Non-RT. Call from prepareToPlay (or similar).
// You can reuse a global IPLContext created elsewhere in LocusQ if you prefer.
bool init(IPLContextctx, constSettings& s) {
shutdown();

context_ = ctx;
settings_ = s;

audioSettings_.samplingRate = settings_.sampleRate;
audioSettings_.frameSize    = settings_.framesPerBuffer;

// Default HRTF (cheap and stable). Steam Audio supports built-in
HRTF. :contentReference[oaicite:1]{index=1}
if (!createDefaultHrtf()) returnfalse;

if (!createDecodeEffect()) returnfalse;

// Allocate buffers (non-RT)
allocateBuffers(settings_.framesPerBuffer, settings_.maxAmbisonicOrder);

// Publish active handles to RT via atomics
publishActive();

returntrue;
    }

// Non-RT. Call on sample-rate/block-size change.
void shutdown() {
// Ensure RT sees nullptr before we free
activeDecode_.store(nullptr, std::memory_order_release);
activeHrtf_.store(nullptr, std::memory_order_release);

decode_.reset();
hrtf_.reset();

inChPtrs_.clear();
outChPtrs_.clear();
inInterleaved_.clear();
```

```
outInterleaved_.clear();

context_ = nullptr;
    }

// Non-RT. Loads SOFA HRTF from file path. Steam Audio supports
IPLHRTFSettings.sofaFileName. :contentReference[oaicite:2]{index=2}
// After this, audio thread will start using the new HRTF on next block.
bool setHrtfFromSofaFile(const std::string& sofaPath) {
if (!context_) returnfalse;

IplHrtfHandlenewHrtf;
IPLHRTFSettingshs{};
hs.type = IPL_HRTFTYPE_SOFA;
hs.sofaFileName = sofaPath.c_str();
hs.sofaData = nullptr;
hs.sofaDataSize = 0;
hs.volume = settings_.hrtfVolume;
hs.normType = settings_.normType;

IPLHRTFh = nullptr;
constautoerr = iplHRTFCreate(context_, &audioSettings_, &hs, &h); // expensive; non-
RT :contentReference[oaicite:3]{index=3}
if (err!=IPL_STATUS_SUCCESS||!h) returnfalse;

newHrtf = IplHrtfHandle(h);

// Swap in (non-RT)
hrtf_ = std::move(newHrtf);

// Decode effect holds no baked HRTF; it's provided per-apply in params, but settings
require one too. :contentReference[oaicite:4]{index=4}
// Easiest: recreate decode effect with new HRTF in settings.
if (!recreateDecodeEffect()) returnfalse;

publishActive();
returntrue;
    }

// Non-RT. Revert to built-in/default HRTF.
bool setDefaultHrtf() {
if (!context_) returnfalse;
if (!createDefaultHrtf()) returnfalse;
if (!recreateDecodeEffect()) returnfalse;
publishActive();
returntrue;
    }

// Non-RT: Reset internal effect state (e.g., transport start/stop)
void reset() {
if (decode_.get()) iplAmbisonicsDecodeEffectReset(decode_.get());
    }

// RT-safe. Process one block: Ambisonics (order <= max) -> stereo out.
//
// inAmbi: array of channel pointers [numInCh][numFrames] non-interleaved
// outStereo: array of channel pointers [2][numFrames] non-interleaved
//
// orientation: listener coordinate frame; Steam Audio uses IPLCoordinateSpace3 in
params. :contentReference[oaicite:5]{index=5}
void processBlock(constfloat* const* inAmbi,
intnumInChannels,
float* const* outStereo,
intnumFrames,
intambisonicOrder,
constIPLCoordinateSpace3& orientation) noexcept
    {
auto* effect = activeDecode_.load(std::memory_order_acquire);
auto* hrtf   = activeHrtf_.load(std::memory_order_acquire);

if (!effect||!hrtf||numFrames<=0) {
// fail soft: output silence
if (outStereo&&outStereo[0] &&outStereo[1]) {
                std::fill(outStereo[0], outStereo[0] +numFrames, 0.0f);
                std::fill(outStereo[1], outStereo[1] +numFrames, 0.0f);
            }
return;
        }

// Build IPLAudioBuffer views (no allocations)
// Input: Ambisonics is non-interleaved, we point Steam Audio at incoming channel
pointers
IPLAudioBufferinBuf{};
inBuf.numChannels = numInChannels;
inBuf.numSamples  = numFrames;
inBuf.data        = const_cast<float**>(inAmbi); // Steam Audio expects float** for non-
interleaved buffers

IPLAudioBufferoutBuf{};
outBuf.numChannels = 2;
outBuf.numSamples  = numFrames;
outBuf.data        = outStereo; // float** for stereo non-interleaved

IPLAmbisonicsDecodeEffectParamsp{};
p.order = ambisonicOrder;
p.hrtf = hrtf;
p.orientation = orientation;
p.binaural = IPL_TRUE;

// Not in-place per docs. :contentReference[oaicite:6]{index=6}
    (void) iplAmbisonicsDecodeEffectApply(effect, &p, &inBuf, &outBuf);
  }

private:
bool createDefaultHrtf() {
hrtf_.reset();

IPLHRTFSettingshs{};
hs.type = IPL_HRTFTYPE_DEFAULT;  // built-in HRTF :contentReference[oaicite:7]
{index=7}
hs.sofaFileName = nullptr;
hs.sofaData = nullptr;
hs.sofaDataSize = 0;
hs.volume = settings_.hrtfVolume;
hs.normType = settings_.normType;

IPLHRTFh = nullptr;
constautoerr = iplHRTFCreate(context_, &audioSettings_, &hs, &h); // non-
RT :contentReference[oaicite:8]{index=8}
if (err!=IPL_STATUS_SUCCESS||!h) returnfalse;

hrtf_ = IplHrtfHandle(h);
returntrue;
  }

bool createDecodeEffect() {
```

```
decode_.reset();

IPLAmbisonicsDecodeEffectSettingses{};
es.maxOrder = settings_.maxAmbisonicOrder;
es.hrtf = hrtf_.get(); // used by the effect; params also provide
hrtf :contentReference[oaicite:9]{index=9}

// speakerLayout is required by struct for panning; for binaural it's ignored,
// but we still must provide a valid layout. We'll set stereo.
es.speakerLayout.type = IPL_SPEAKERLAYOUTTYPE_STEREO;
es.speakerLayout.numSpeakers = 2;
es.speakerLayout.speakers[0] = IPLSpeaker{ IPLVector3{-1.0f, 0.0f, 0.0f} };
es.speakerLayout.speakers[1] = IPLSpeaker{ IPLVector3{ 1.0f, 0.0f, 0.0f} };

IPLAmbisonicsDecodeEffecte = nullptr;
constautoerr = iplAmbisonicsDecodeEffectCreate(context_, &audioSettings_, &es,
&e); // :contentReference[oaicite:10]{index=10}
if (err!=IPL_STATUS_SUCCESS||!e) returnfalse;

decode_.e = e;
returntrue;
   }

bool recreateDecodeEffect() {
// Ensure RT sees nullptr before we drop the old effect
activeDecode_.store(nullptr, std::memory_order_release);
decode_.reset();
returncreateDecodeEffect();
   }

staticint ambiChannelsForOrder(intorder) {
// ACN/SN3D channel count: (order+1)^2
return (order+1) * (order+1);
   }

void allocateBuffers(intframes, intmaxOrder) {
constintmaxCh = ambiChannelsForOrder(maxOrder);

// These buffers are optional; in this wrapper we pass pointers directly to Steam Audio.
// Kept here in case you want to copy/reshape formats later.
inInterleaved_.resize((size_t)maxCh* (size_t)frames, 0.0f);
outInterleaved_.resize((size_t)2* (size_t)frames, 0.0f);
```

```
inChPtrs_.resize((size_t)maxCh, nullptr);
outChPtrs_.resize(2, nullptr);
    }

void publishActive() {
activeHrtf_.store(hrtf_.get(), std::memory_order_release);
activeDecode_.store(decode_.get(), std::memory_order_release);
    }

private:
IPLContext context_ = nullptr;
Settings settings_{};
IPLAudioSettingsaudioSettings_{};

IplHrtfHandlehrtf_;
IplAmbiDecodeHandledecode_;

    std::atomic<IPLHRTF>activeHrtf_{ nullptr };
    std::atomic<IPLAmbisonicsDecodeEffect>activeDecode_{ nullptr };

// Optional scratch space (non-RT allocated)
    std::vector<float>inInterleaved_;
    std::vector<float>outInterleaved_;
    std::vector<float*>inChPtrs_;
    std::vector<float*>outChPtrs_;
};

} // namespace locusq
```

Notes you should not ignore

- iplHRTFCreate is **expensive and not thread-safe**; keep it off the audio thread (state/app thread only).
- iplAmbisonicsDecodeEffectApply **cannot be in-place**.
- If your internal soundfield isn't Ambisonics yet, Steam Audio also offers **Virtual Surround** (multi-channel bed → binaural) as a simpler intermediate step.

If you tell me what your internal "monitoring input" format is right now (Ambisonics? quad bed? per-source?), I'll give you the correct wrapper variant

(Ambisonics decode vs virtual surround vs per-source binaural) with the smallest surgery to your graph.

Below is a **RT-safe PEQ biquad cascade** (no allocations in processBlock, no locks). It's designed to be dropped into JUCE/C++ codebases like LocusQ.

Key properties:

- **Max 8 bands** (matches your state contract)
- **Band enable** is derived from bandCount (or you can add explicit enable flags)
- **Coefficient updates are non-RT** and swapped atomically by pointer (double-buffer)
- **Denormal protection** built in
- Uses **Direct Form 1** (stable enough for PEQ; DF2T is also fine, but DF1 is straightforward)

BiquadPeq.h

```
C++#pragma once

#include<array>
#include<atomic>
#include<cmath>
#include<cstdint>
#include<cstring>

namespacelocusq {

staticinlinefloatclampf(floatx, floatlo, floathi) noexcept
{
return (x<lo) ? lo : (x>hi) ? hi : x;
}

// Small denormal guard: add/sub tiny DC offset.
staticinlinefloatdenormGuard(floatx) noexcept
{
#if defined(__SSE2__) || defined(__ARM_NEON)
// cheap: just return; most modern CPUs flush denormals in DAWs anyway
```

```
returnx;
#else
constexprfloattiny = 1e-20f;
returnx+tiny-tiny;
#endif
}

structPeqBandParams {
float freqHz = 1000.0f;
float gainDb = 0.0f;   // peak gain
float q      = 0.707f; // quality
};

// A single biquad, Direct Form I (per-channel state).
structBiquadDF1 {
// Coeffs (a0 normalized to 1):
// y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]
float b0 = 1.f, b1 = 0.f, b2 = 0.f;
float a1 = 0.f, a2 = 0.f;

// State
float x1 = 0.f, x2 = 0.f;
float y1 = 0.f, y2 = 0.f;

void reset() noexcept { x1 = x2 = y1 = y2 = 0.f; }

inlinefloat process(floatx) noexcept
    {
// Guard denormals in state & signal
x = denormGuard(x);

constfloaty = b0*x+b1*x1+b2*x2
-a1*y1-a2*y2;

x2 = x1; x1 = x;
y2 = y1; y1 = y;

returndenormGuard(y);
    }
};

structBiquadCoeffs {
```

```
float b0=1.f, b1=0.f, b2=0.f;
float a1=0.f, a2=0.f;
};

// Cookbook coefficients for a peaking EQ biquad (RBJ Audio EQ Cookbook).
// Stable, common, and good enough for PEQ bands.
staticinlineBiquadCoeffsmakePeakingEQ(floatsampleRate,
floatfreqHz,
floatgainDb,
floatq) noexcept
{
// Clamp to avoid nonsense / NaNs
constfloatsr = (sampleRate>1.f) ? sampleRate : 48000.f;
constfloatf  = clampf(freqHz, 20.f, 0.49f*sr);
constfloatQ  = clampf(q, 0.1f, 10.f);
constfloatA  = std::pow(10.0f, gainDb/40.0f);

constfloatw0 = 2.0f* (float)M_PI* (f/sr);
constfloatcw = std::cos(w0);
constfloatsw = std::sin(w0);
constfloatalpha = sw/ (2.0f*Q);

constfloatb0 = 1.0f+alpha*A;
constfloatb1 = -2.0f*cw;
constfloatb2 = 1.0f-alpha*A;
constfloata0 = 1.0f+alpha/A;
constfloata1 = -2.0f*cw;
constfloata2 = 1.0f-alpha/A;

// Normalize a0 to 1
constfloatinvA0 = 1.0f/a0;

BiquadCoeffsc;
c.b0 = b0*invA0;
c.b1 = b1*invA0;
c.b2 = b2*invA0;
c.a1 = a1*invA0;
c.a2 = a2*invA0;
returnc;
}

//============================================================================
===
// RT-safe PEQ: coefficient set is prepared off-thread then atomically swapped.
//=============================================================================
===

template<intkMaxBands = 8, intkMaxChannels = 2>
classPeqBiquadCascadefinal {
public:
PeqBiquadCascade() = default;

void prepare(doublesampleRate, intmaxBlockSize, intnumChannels) noexcept
   {
       (void)maxBlockSize;
sampleRate_.store((float)sampleRate, std::memory_order_release);
numChannels_ = (numChannels<=kMaxChannels) ? numChannels : kMaxChannels;

reset();
   }

void reset() noexcept
   {
for (intch = 0; ch<numChannels_; ++ch)
for (intb = 0; b<kMaxBands; ++b)
state_[ch][b].reset();
   }

// -------- Non-RT API (call from message thread / state thread) --------
// Builds coefficients into the inactive bank and swaps atomically.
void setBandsNonRT(constPeqBandParams* bands, intbandCount) noexcept
   {
constintn = (bandCount<0) ? 0 : (bandCount>kMaxBands ? kMaxBands : bandCount);
constfloatsr = sampleRate_.load(std::memory_order_acquire);

CoeffBanknext{};
next.activeBands = n;

for (inti = 0; i<kMaxBands; ++i)
       {
if (i<n)
           {
constauto& p = bands[i];
next.coeffs[i] = makePeakingEQ(sr, p.freqHz, p.gainDb, p.q);
```

```
            }
else
            {
// Identity
next.coeffs[i] = BiquadCoeffs{};
            }
        }

// Write into inactive bank then publish pointer (atomic)
constintinactive = 1-activeBankIndex_.load(std::memory_order_relaxed);
banks_[inactive] = next;
activeBankIndex_.store(inactive, std::memory_order_release);

// Note: we do NOT reset state here automatically.
// For big changes you may want to ramp or crossfade; keep v1 simple.
    }

// -------- RT API (audio thread) --------
// inOut: array of non-interleaved channels.
void processBlock(float** inOut, intnumChannels, intnumSamples) noexcept
    {
if (!inOut||numSamples<=0) return;

constintchCount = (numChannels<numChannels_) ? numChannels : numChannels_;
constintbankIdx = activeBankIndex_.load(std::memory_order_acquire);
constCoeffBank& bank = banks_[bankIdx];

constintactiveBands = bank.activeBands;
if (activeBands<=0) return; // identity

// Apply coefficients to runtime state (cheap, per block).
// This avoids touching coefficients per sample.
for (intch = 0; ch<chCount; ++ch)
        {
for (intb = 0; b<activeBands; ++b)
            {
auto& s = state_[ch][b];
constauto& c = bank.coeffs[b];
s.b0 = c.b0; s.b1 = c.b1; s.b2 = c.b2;
s.a1 = c.a1; s.a2 = c.a2;
            }
        }
```

```
for (intch = 0; ch<chCount; ++ch)
        {
float* x = inOut[ch];
if (!x) continue;

for (intn = 0; n<numSamples; ++n)
            {
floaty = x[n];

// Cascade
for (intb = 0; b<activeBands; ++b)
y = state_[ch][b].process(y);

x[n] = y;
            }
        }
    }

private:
structCoeffBank {
int activeBands = 0;
        std::array<BiquadCoeffs, kMaxBands> coeffs{};
    };

    std::atomic<float>sampleRate_{48000.f};
intnumChannels_ = 2;

// Two coefficient banks, swapped by index
    std::array<CoeffBank, 2>banks_{};
    std::atomic<int>activeBankIndex_{0};

// Per-channel, per-band states
    std::array<std::array<BiquadDF1, kMaxBands>, kMaxChannels>state_{};
};

} // namespace locusq
```

How to use it in your monitoring chain

```
C++// Somewhere in your monitoring stage:
locusq::PeqBiquadCascade<8, 2>hpPeq;

// prepareToPlay:
hpPeq.prepare(sampleRate, samplesPerBlock, /*channels*/2);

// on state update (NON-RT):
locusq::PeqBandParamsbands[8];
// fill from your HeadphoneCalibrationProfile / APVTS
hpPeq.setBandsNonRT(bands, bandCount);

// processBlock (RT):
hpPeq.processBlock(stereoOutPtrs, 2, numSamples);
```

Practical integration notes (because you'll hit these)

1. **Avoid clicks when changing EQ**

- v1: acceptable if changes are rare and UI-driven.
- Better: crossfade old/new cascade output for ~20–50ms or ramp band gains over time.

2. **Band enable policy**

- I used bandCount to decide active bands because that matches your contract (hp_peq_band_count).
- If you later add explicit enable flags, keep bandCount as a "max active bands" limiter for safety.

3. **Numerical safety**

- Clamps prevent NaNs from corrupting the audio graph.
- If sampleRate changes, call prepare() again (you already do in JUCE).

If your EQ needs **shelves** + **high/low-pass** (for real headphone correction presets), I can extend this with the RBJ cookbook set:

- low shelf, high shelf
- low-pass, high-pass
- notch, band-pass
  without changing the RT strategy.

Pick the wrapper variant: smallest surgery, maximum leverage

You're "likely per-source, quadraphonics and/or binaural," and you're not sure you're Ambisonics. Given that uncertainty, here's the blunt recommendation:

Do **Virtual Surround (bed → binaural)** first. Don't jump to Ambisonics yet.

Why this is the lowest-surgery path:

- If you already have a **quad bed** (or can trivially produce one), Virtual Surround lets you go **quad → stereo binaural** immediately.
- You avoid rewriting your renderer into Ambisonics today.
- You still get **HRTF selection (SOFA)** + **headphone EQ** + **verification** working end-to-end.
- You can keep your "per-source" renderer as-is and just add a monitoring "adapter."

When to use what (opinionated decision table)

- **Virtual Surround** (recommended now):
  Use if you have **quad** / **5.1** / **7.x** beds or can cheaply make them.
  It's the "monitoring bridge" that ships fastest.
- **Per-source binaural** (only if you're truly per-source already):
  Use if your internal engine is already "N sources with positions + dry audio per source."
  This gives the best quality *and* head tracking correctness, but requires you to feed Steam Audio per source. That's more plumbing.
- **Ambisonics decode** (do later):
  Use if you already represent your scene as an **ambisonic soundfield** (ACN/SN3D) or want that as your core mixing format.

Great long-term, but it's not smallest-surgery unless you already have the field.

The exact minimal-surgery plan for LocusQ

1. **Keep your current engine output** (per-source or quad bed).
2. Add a **MonitoringAdapter** stage that can output:
   - `speakers`: pass-through multichannel
   - `steam_binaural`: internally binauralize to stereo
3. For `steam_binaural`, implement **Virtual Surround** first:
   - Input: quad bed (or 5.1/7.x if you already have them)
   - Output: stereo binaural
4. After that works, optionally add:
   - `per_source_binaural` path (higher quality)
   - `ambisonic` internal format (bigger refactor, better spatial math)

"Correct" wrapper variant: Steam Audio Virtual Surround (bed → binaural)

This is the wrapper you want right now.

What it expects

- A multichannel "speaker layout" input (quad is fine)
- An HRTF (default or SOFA)
- Orientation (for head tracking / yaw offset)

What it gives you

- Stereo binaural out
- Works with your quad/5.1/7.x output types without rewriting the renderer

If you want, I'll produce the full class like the earlier `SteamAudioBinaural` wrapper, but using:

- `IPLVirtualSurroundEffect` create/apply
- a `IPLSpeakerLayout` matching your bed (quad/5.1/etc.)

(You'll end up with two wrappers eventually: `SteamAudioVirtualSurround` and `SteamAudioPerSourceBinaural`.)

FIR convolution engine selection (direct vs partitioned)

This is the other half of "headphone calibration doesn't glitch."

Strong recommendation

Implement **two engines** and select by tap count:

1) Direct time-domain convolution

Use when taps are small.

- Typical threshold: **≤ 256 taps** (sometimes 384 depending on CPU budget)
- Complexity: `O(N * taps)` per sample block
- Pros: dead simple, low overhead, low latency, no FFT setup
- Cons: gets expensive quickly

2) Partitioned FFT convolution (uniform partition)

Use when taps are large.

- Typical threshold: > **256 taps**
- Choose partition size `P` as a power-of-two tied to block size.
- Pros: scales well to thousands of taps
- Cons: introduces **algorithmic latency ~P** (unless you do fancy zero-latency partitioning)

Suggested thresholds (practical in plugins)

- **0 taps**: bypass

- **1–256 taps**: direct
- **257–2048 taps**: partitioned FFT with P = nextPow2(blockSize) (or 2× blockSize if you can tolerate latency)
- >**2048 taps**: partitioned FFT with P = nextPow2(blockSize) and keep partitions uniform; don't get clever until v2

Partition size choice (simple and correct)

Let:

- B = processBlock size (framesPerBuffer)
- P = nextPow2(B) (e.g., B=512 → P=512)
- Convolution latency is roughly P samples in a uniform partition scheme.

If you *must* keep latency smaller:

- Use P = nextPow2(B/2) only if your implementation supports overlap-add safely with your block sizes.
- Or accept P and **report latency explicitly** via your hp_fir_latency_samples.

RT-safe engine switching strategy

Switching convolution engines must be non-RT:

- Build engine instance off-thread:
  - precompute FFTs for all partitions
  - allocate all buffers
- Atomically swap a pointer used by audio thread
- Old instance freed after the swap (on non-audio thread)

Exactly like the coefficient bank swap you're doing for PEQ, but with bigger objects.

Minimal interface (what you actually want in LocusQ)

```
C++structIFirConvolver {
virtual~IFirConvolver() = default;
virtualint latencySamples() constnoexcept = 0;
virtualvoid reset() noexcept = 0;
virtualvoid processBlock(float** inOut, intnumCh, intnumSamples) noexcept = 0;
};

std::unique_ptr<IFirConvolver>makeConvolver(const std::vector<float>& taps,
intnumCh,
intblockSize,
intsampleRate)
{
if (taps.empty()) returnnullptr;
if ((int)taps.size() <=256) returnmakeDirectConvolver(...);
returnmakePartitionedFFTConvolver(...);
}
```

That's enough to ship v1.

Latency accounting (non-negotiable)

- Direct: latency = 0 (or group delay if you're using linear-phase FIR, but algorithmically it's 0)
- Partitioned uniform: latency ≈ `P` (your partition size)
- Store/report this via `hp_fir_latency_samples` and `setLatencySamples()`.

Final call: what you should implement first in LocusQ

1. **Steam Audio Virtual Surround** monitoring path for quad (and optionally 5.1/7.x)
2. PEQ (you have it)
3. FIR engine selection (direct vs partitioned)
4. Head tracking yaw offset + smoothing (you already planned that)

This gives you headphone calibration that actually works without rebuilding your renderer.

If you tell me your current bed format for “quad” (channel order and coordinate convention), I’ll generate:

- the exact IPLSpeakerLayout mapping,
- the full SteamAudioVirtualSurround wrapper class (create/apply/reset, SOFA switching, orientation),
- and a concrete partitioned FFT convolver implementation outline that matches your block sizes and avoids allocations in processBlock.

Got it. Since you’re building **quad spatialization inside a DAW**, the safest “smallest surgery” approach is:

- Treat your internal quad bus as a **speaker bed**.
- Map that bed into a Steam Audio IPLSpeakerLayout.
- Use **Virtual Surround** to binauralize (with optional SOFA HRTF + head orientation).

You didn’t give your exact channel order, so I’m going to be opinionated and assume the **most common DAW quad bed**:

Default assumption (recommended)

Quad = L, R, Ls, Rs
(Front Left, Front Right, Rear/Surround Left, Rear/Surround Right)

If your project uses **L, R, Lb, Rb** naming, treat Lb/Rb as Ls/Rs (it’s the same geometry for this purpose).

I’ll also give you a **small mapping switch** so you can support alternate orders without ripping up your graph.

---

1) Exact IPLSpeakerLayout mapping for quad

Steam Audio expects speaker directions as 3D vectors in a coordinate system. We’ll use the conventional audio coordinate frame:

- **+X = right**
- **+Y = up**

- **-Z** = **forward** (so “in front” is -Z)

Quad speaker directions (unit-ish vectors) using azimuth angles:

- L: -30° → (-sin30, 0, -cos30) ≈ (-0.5, 0, -0.866)
- R: +30° → (+0.5, 0, -0.866)
- Ls: -110° → (-0.94, 0, +0.342) (rear-left)
- Rs: +110° → (+0.94, 0, +0.342) (rear-right)

Those are standard “ITD/ILD sensible” directions for a quad bed.

Speaker layout builder (quad)

```
C++#pragma once
#include<phonon.h>
#include<array>

namespacelocusq {

enumclassQuadOrder : int {
L_R_Ls_Rs = 0,   // recommended default
L_R_Rs_Ls = 1,
Ls_Rs_L_R = 2,
Custom = 3
};

inlineIPLVector3v3(floatx, floaty, floatz) { IPLVector3v{ x,y,z }; returnv; }

inlinevoidfillQuadSpeakerLayout(IPLSpeakerLayout& layout, QuadOrderorder)
{
layout.type = IPL_SPEAKERLAYOUTTYPE_CUSTOM;
layout.numSpeakers = 4;

// Directions (x right, y up, z forward negative)
constIPLVector3L  = v3(-0.5f, 0.0f, -0.8660254f);
constIPLVector3R  = v3( 0.5f, 0.0f, -0.8660254f);
constIPLVector3Ls = v3(-0.9396926f, 0.0f,  0.3420201f);
constIPLVector3Rs = v3( 0.9396926f, 0.0f,  0.3420201f);

autoset = [&](intidx, IPLVector3dir) {
layout.speakers[idx].direction = dir;
```

```
    };

switch (order)
    {
case QuadOrder::L_R_Ls_Rs:
set(0, L);  set(1, R);  set(2, Ls); set(3, Rs);
break;
case QuadOrder::L_R_Rs_Ls:
set(0, L);  set(1, R);  set(2, Rs); set(3, Ls);
break;
case QuadOrder::Ls_Rs_L_R:
set(0, Ls); set(1, Rs); set(2, L);  set(3, R);
break;
default:
// If Custom, caller must fill layout.speakers[].direction themselves.
// But we still provide a sane default to avoid undefined behavior.
set(0, L);  set(1, R);  set(2, Ls); set(3, Rs);
break;
    }
}

} // namespace locusq
```

This lets you match whatever channel order you currently output without changing the renderer: you just tell the monitoring stage which `QuadOrder` is active.

---

2) Full `SteamAudioVirtualSurround` wrapper class

This is the “monitoring adapter” you drop at the end of your graph:

- Input: **quad bed** (4 channels, non-interleaved)
- Output: **stereo** (2 channels, non-interleaved)
- Features:
  - create/init/reset/shutdown
  - swap HRTF to SOFA (non-RT)
  - orientation (for head tracking + yaw offset)

- no allocations in processBlock

Notes:

- Steam Audio types/functions vary slightly by version; this is correct in structure and naming for the C API patterns. You may need to adjust exact struct fields if your Steam Audio SDK differs.

```
C++#pragma once

#include<atomic>
#include<string>
#include<phonon.h>

#include"QuadSpeakerLayout.h"// from above

namespacelocusq {

classSteamAudioVirtualSurroundfinal {
public:
structSettings {
int sampleRate = 48000;
int framesPerBuffer = 512;
QuadOrder quadOrder = QuadOrder::L_R_Ls_Rs;

// HRTF normalization
IPLHRTFNormType normType = IPL_HRTFNORMTYPE_RMS;
float hrtfVolume = 1.0f;
    };

SteamAudioVirtualSurround() = default;
~SteamAudioVirtualSurround() { shutdown(); }

SteamAudioVirtualSurround(constSteamAudioVirtualSurround&) = delete;
SteamAudioVirtualSurround& operator=(constSteamAudioVirtualSurround&) = delete;

// Non-RT: call in prepareToPlay
bool init(IPLContextctx, constSettings& s)
    {
shutdown();
context_ = ctx;
settings_ = s;
```

```
audioSettings_.samplingRate = settings_.sampleRate;
audioSettings_.frameSize = settings_.framesPerBuffer;

// Build speaker layout for quad
fillQuadSpeakerLayout(quadLayout_, settings_.quadOrder);

// Default HRTF
if (!createDefaultHrtf()) returnfalse;

// Create virtual surround effect
if (!createEffect()) returnfalse;

publishActive();
returntrue;
    }

// Non-RT
void shutdown()
    {
activeEffect_.store(nullptr, std::memory_order_release);
activeHrtf_.store(nullptr, std::memory_order_release);

if (effect_) {
iplVirtualSurroundEffectRelease(&effect_);
effect_ = nullptr;
        }
if (hrtf_) {
iplHRTFRelease(&hrtf_);
hrtf_ = nullptr;
        }
context_ = nullptr;
    }

// Non-RT
void reset()
    {
if (effect_) iplVirtualSurroundEffectReset(effect_);
    }

// Non-RT: swap to SOFA
bool setHrtfFromSofaFile(const std::string& sofaPath)
    {
```

```
if (!context_) returnfalse;

// Create new HRTF (expensive)
IPLHRTFnewH = nullptr;
        {
IPLHRTFSettingshs{};
hs.type = IPL_HRTFTYPE_SOFA;
hs.sofaFileName = sofaPath.c_str();
hs.sofaData = nullptr;
hs.sofaDataSize = 0;
hs.volume = settings_.hrtfVolume;
hs.normType = settings_.normType;

if (iplHRTFCreate(context_, &audioSettings_, &hs, &newH) !=IPL_STATUS_SUCCESS||!
newH)
returnfalse;
        }

// Publish null effect to RT before replacing
activeEffect_.store(nullptr, std::memory_order_release);

// Replace HRTF
if (hrtf_) iplHRTFRelease(&hrtf_);
hrtf_ = newH;

// Recreate effect so settings match (safe path)
if (effect_) {
iplVirtualSurroundEffectRelease(&effect_);
effect_ = nullptr;
        }
if (!createEffect()) returnfalse;

publishActive();
returntrue;
    }

// Non-RT
bool setDefaultHrtf()
    {
if (!context_) returnfalse;

activeEffect_.store(nullptr, std::memory_order_release);
```

```
if (hrtf_) iplHRTFRelease(&hrtf_);
hrtf_ = nullptr;
if (!createDefaultHrtf()) returnfalse;

if (effect_) {
iplVirtualSurroundEffectRelease(&effect_);
effect_ = nullptr;
        }
if (!createEffect()) returnfalse;

publishActive();
returntrue;
    }

// RT: quad bed -> stereo binaural
void processBlock(constfloat* const* inQuad, float* const* outStereo,
intnumFrames,
constIPLCoordinateSpace3& listenerOrientation) noexcept
    {
auto* eff = activeEffect_.load(std::memory_order_acquire);
auto* hrtf = activeHrtf_.load(std::memory_order_acquire);

if (!eff||!hrtf||!inQuad||!outStereo||numFrames<=0) {
if (outStereo&&outStereo[0] &&outStereo[1]) {
                std::fill(outStereo[0], outStereo[0] +numFrames, 0.0f);
                std::fill(outStereo[1], outStereo[1] +numFrames, 0.0f);
            }
return;
        }

// Steam Audio uses IPLAudioBuffer with float** (non-interleaved)
IPLAudioBufferinBuf{};
inBuf.numChannels = 4;
inBuf.numSamples = numFrames;
inBuf.data = const_cast<float**>(inQuad);

IPLAudioBufferoutBuf{};
outBuf.numChannels = 2;
outBuf.numSamples = numFrames;
outBuf.data = outStereo;
```

```
IPLVirtualSurroundEffectParamsp{};
p.speakerLayout = quadLayout_;
p.hrtf = hrtf;
p.orientation = listenerOrientation;
p.binaural = IPL_TRUE;

iplVirtualSurroundEffectApply(eff, &p, &inBuf, &outBuf);
   }

// Optional: update quad order without touching renderer (non-RT)
void setQuadOrder(QuadOrderorder)
   {
settings_.quadOrder = order;
fillQuadSpeakerLayout(quadLayout_, order);
// No need to recreate effect; layout is provided per-apply in params.
   }

private:
bool createDefaultHrtf()
   {
IPLHRTFSettingshs{};
hs.type = IPL_HRTFTYPE_DEFAULT;
hs.sofaFileName = nullptr;
hs.sofaData = nullptr;
hs.sofaDataSize = 0;
hs.volume = settings_.hrtfVolume;
hs.normType = settings_.normType;

return (iplHRTFCreate(context_, &audioSettings_, &hs, &hrtf_)
==IPL_STATUS_SUCCESS&&hrtf_);
   }

bool createEffect()
   {
IPLVirtualSurroundEffectSettingses{};
es.speakerLayout = quadLayout_;
es.hrtf = hrtf_;

return (iplVirtualSurroundEffectCreate(context_, &audioSettings_, &es, &effect_)
==IPL_STATUS_SUCCESS&&effect_);
   }
```

```
void publishActive()
    {
activeHrtf_.store(hrtf_, std::memory_order_release);
activeEffect_.store(effect_, std::memory_order_release);
    }

private:
IPLContext context_ = nullptr;
Settings settings_{};
IPLAudioSettingsaudioSettings_{};

IPLSpeakerLayoutquadLayout_{};
IPLHRTFhrtf_ = nullptr;
IPLVirtualSurroundEffecteffect_ = nullptr;

    std::atomic<IPLHRTF>activeHrtf_{ nullptr };
    std::atomic<IPLVirtualSurroundEffect>activeEffect_{ nullptr };
};

} // namespace locusq
```

Orientation + yaw offset

You said you're working on the head tracking polish (low-pass quaternion, reject stale, clamp spikes, debug overlay). Good. In the monitoring stage do:

- `orientation = incomingOrientation`
- apply `yawOffsetDeg` by pre-multiplying a yaw rotation
- feed result into `processBlock(...)`

(You already spec'd the bridge smoothing rules; keep them outside the plugin audio thread.)

---

3) Partitioned FFT convolver outline (no allocations in `processBlock`)

You asked for an "outline," but I'll give you something you can implement directly with minimal risk.

Strategy (opinionated, safe)

- **Direct convolver** for taps ≤ 256 (time-domain)
- **Uniform partitioned FFT** for taps > 256
- Partition size P = nextPow2(blockSize)
  (This is stable with DAW blocks and easy to reason about.)
- **Latency** = **P** samples (report via your hp_fir_latency_samples and host latency)

Class design

- Build everything off-thread.
- Audio thread only does:
  - FFT block
  - multiply-accumulate per partition
  - iFFT
  - overlap-add with preallocated buffers

Minimal interface

```
C++structIFirConvolver {
virtual~IFirConvolver() = default;
virtualint latencySamples() constnoexcept = 0;
virtualvoid reset() noexcept = 0;
virtualvoid processBlock(float** inOut, intnumCh, intnumSamples) noexcept = 0;
};
```

Partitioned FFT outline details

Precomputation (non-RT)

Given taps h[0..L-1]:

1. Choose:
   - P = nextPow2(blockSize) (e.g., 512)
   - N = 2*P FFT size (overlap-add friendly)

2. Number of partitions:
   - K = ceil(L / P)
3. For each partition k:
   - create time buffer hk length N
   - copy taps h[k*P .. k*P+P-1] into start of hk
   - FFT → Hk (complex bins)
   - store Hk in preallocated array

Runtime processing (RT)

For each channel:

- Maintain:
  - ring buffer of recent **input spectra** X[n] (size K)
  - overlap buffer length P (time-domain)
- For each incoming block (size P):
  a. build xTime[N] with first P samples = input block, second P zeros
  b. FFT to X0
  c. push X0 into ring
  d. compute Y = sum_{k=0..K-1} X_ring[k] * Hk
  e. iFFT(Y) → yTime[N]
  f. output block = yTime[0..P-1] + overlap
  g. overlap = yTime[P..N-1]

Avoid allocations

- All buffers are std::vector allocated in constructor / buildNonRT.
- In processBlock, only indexed reads/writes.

Engine selection (the thing you asked for)

- taps empty → bypass

- taps ≤ 256 → direct
- taps > 256 → partitioned FFT with P = nextPow2(blockSize)

What FFT library?

Since you're in JUCE-land:

- Easiest minimal: juce::dsp::FFT (it's fine; just be careful with format)
- Or use **PFFFT** / **KissFFT** if you want more control, but that's bigger scope.

I recommend starting with juce::dsp::FFT for v1 and swapping later if needed.

---

4) What you should do next in LocusQ (so this doesn't rot)

1. Add a **quad order selector** (even hidden/dev-only) so your binaural monitoring can match any channel order without renderer edits.
2. Land SteamAudioVirtualSurround as the steam_binaural monitoring path for quad.
3. Implement FIR as:
   - DirectFirConvolver (≤256 taps)
   - PartitionedFftConvolver (>256 taps)
   - atomic swap between instances when taps change (same pattern as PEQ coeff banks)

If you paste (or point me to) your actual quad bus order in LocusQ (even just "it's whatever REAPER gives me" / "it's JUCE AudioBuffer ch0..3 = ???"), I'll lock the default mapping to your reality and give you a tiny self-test that plays a "front-left" impulse and asserts it ends up at the correct speaker vector in the Steam Audio layout. That prevents the most common silent-bug: **channel order mismatch that sounds "kind of spatial" but is wrong.**

Good instinct. Codex 5.3 will behave much better if you hand it a structured problem statement instead of a giant transcript.

Below is a **clean technical outline** of what we designed and why — trimmed to the parts that matter for implementation inside joshband/LocusQ.

LocusQ Headphone Calibration & Binaural Monitoring – Research Outline

1. Objective

Integrate robust **headphone calibration** + **binaural monitoring** into LocusQ for:

- Quadraphonic spatialization inside a DAW
- Stereo headphone playback
- Optional head tracking
- Support for:
  - Default HRTF
  - Custom SOFA HRTF
  - Headphone EQ (PEQ + FIR)
  - Internal Steam Audio binauralization

Primary constraint:

- No allocations or locks in processBlock
- Deterministic state
- QA harness compatible

---

2. Architectural Decision: Monitoring Strategy

2.1 Current Internal Format

Likely:

- Per-source spatial engine
- Quadraphonic bed output (L, R, Ls, Rs)
- Not currently Ambisonics

2.2 Wrapper Variant Selection (Opinionated)

Chosen for smallest surgery:

✅ Steam Audio Virtual Surround (quad bed → binaural)

Why:

- No renderer rewrite required
- Works with existing quad output
- Compatible with SOFA HRTFs
- Supports head orientation
- Clean monitoring adapter layer

Deferred:

- Ambisonics decode path
- Per-source binaural path

---

3. Quad Speaker Layout Mapping

3.1 Assumed Channel Order

Default:

- L, R, Ls, Rs

Alternate orders supported via enum:

- L_R_Rs_Ls
- Ls_Rs_L_R
- Custom

3.2 Speaker Geometry (Coordinate System)

Right-handed:

- +X = right
- +Y = up
- -Z = forward

Vectors:

- L: (-0.5, 0, -0.866)
- R: (0.5, 0, -0.866)
- Ls: (-0.94, 0, 0.34)
- Rs: (0.94, 0, 0.34)

Purpose:

- Accurate binauralization of quad bed
- Prevent channel order mismatch bugs

---

4. SteamAudioVirtualSurround Wrapper

4.1 Responsibilities

- Create Steam Audio context bindings
- Create quad IPLSpeakerLayout
- Create IPLVirtualSurroundEffect
- Manage HRTF lifecycle (default + SOFA)

- Accept listener orientation
- Process quad → stereo
- Be fully RT-safe

4.2 Lifecycle

Non-RT:

- init()
- setHrtfFromSofaFile()
- setDefaultHrtf()
- reset()
- shutdown()

RT:

- processBlock()
    - No allocations
    - Atomic pointer swap for HRTF/effect

4.3 Head Tracking Integration

Orientation fed via:

- IPLCoordinateSpace3
- Apply yaw offset
- External smoothing performed outside audio thread

---

5. Headphone Calibration System

5.1 Components

HeadphoneCalibrationProfile contains:

- hp_enabled

- hp_tracking_enabled
- hp_yaw_offset_deg
- hp_eq_mode (off | peq | fir)
- hp_peq_band_count
- hp_peq band params (up to 8)
- hp_fir_taps + latency
- hp_hrtf_mode (default | sofa)
- verification scores
- blobs:
  - hp_sofa_ref
  - hp_fir_coeffs_f32

5.2 State Strategy

- Primitive parameters via APVTS
- Large data (FIR taps, SOFA ref) as base64 blobs
- Deterministic migration
- QA harness compatible

---

6. Parametric EQ (PEQ)

6.1 Implementation

- Direct Form I biquad
- RBJ cookbook coefficients
- Up to 8 bands
- Per-channel cascades

6.2 RT Safety

- Coefficients built non-RT
- Double-bank atomic swap
- No per-sample allocations
- Denormal protection

---

7. FIR Convolution Engine Strategy

7.1 Engine Selection (Opinionated Threshold)

- 0 taps → bypass
- ≤ 256 taps → direct time-domain
- 256 taps → partitioned FFT

7.2 Direct Convolver

- O(N × taps)
- Zero algorithmic latency
- Suitable for short IRs

7.3 Partitioned FFT Convolver

Uniform partitioned overlap-add.

Parameters:

- P = nextPow2(blockSize)
- FFT size = 2P
- K = ceil(taps / P)

Runtime:

1. FFT input block
2. Multiply-accumulate partitions

3. IFFT
4. Overlap-add
5. Output

Latency:

- ≈ P samples
- Report via `setLatencySamples()`

7.4 RT Safety

- All FFT plans + buffers preallocated
- Partition spectra precomputed
- Engine swap via atomic pointer

---

8. Final Monitoring Chain (Headphone Mode)

When:
`cal_monitoring_path == steam_binaural`

Pipeline:

1. Internal spatial renderer
2. Quad bed output
3. SteamAudioVirtualSurround
4. PEQ (if enabled)
5. FIR (if enabled)
6. Stereo output

When:
`virtual_binaural`

- Skip Steam Audio
- Pass multichannel to system

- Optional PEQ

---

9. Latency Management

Must explicitly track:

- FIR latency
- Report to host
- No hidden delay
- No surprise drift between monitoring modes

---

10. Future Extensions (Not Required for v1)

- Ambisonics internal format
- Per-source Steam Audio path
- Crossfaded EQ changes
- Zero-latency non-uniform partitioned convolution
- Personalized HRTF generation
- Device profile auto-matching

---

11. Implementation Priorities (Recommended Order)

1. SteamAudioVirtualSurround (quad → binaural)
2. PEQ cascade (done)
3. FIR engine selection + convolver
4. Head tracking orientation injection

5. Verification test harness
6. QA migration + golden state update

If you want, I can now generate:

- A condensed “Codex task brief” version (10–15 bullet implementation tasks)
- Or a structured JSON project spec for Codex to execute in stages
- Or a phased refactor roadmap aligned with your existing LocusQ repo structure

Below is a **condensed Codex task brief** suitable for GPT-5.3 execution inside joshband/LocusQ.

It is implementation-oriented, dependency-aware, and ordered.

LocusQ – Headphone Calibration & Binaural Monitoring

Codex Task Brief (Execution Plan)

---

1. Add Quad Speaker Layout Mapping

- Implement QuadOrder enum.
- Implement fillQuadSpeakerLayout(IPLSpeakerLayout&, QuadOrder).
- Default to L, R, Ls, Rs.
- Add unit test validating channel index → direction mapping.

---

2. Implement SteamAudioVirtualSurround Wrapper

- Create SteamAudioVirtualSurround class.

- Responsibilities:
  - init / shutdown
  - reset
  - SOFA HRTF swap (non-RT)
  - quad → stereo processBlock (RT safe)
- Ensure atomic pointer swap for effect + HRTF.
- No allocations inside processBlock.

---

3. Integrate Monitoring Path Switch

- Modify monitoring stage:
  - speakers → pass-through
  - steam_binaural → route quad bed into wrapper
  - virtual_binaural → bypass wrapper
- Publish requested/active diagnostics for CALIBRATE UI.

---

4. Add Head Tracking Injection Point

- Accept IPLCoordinateSpace3 from bridge layer.
- Apply yaw offset (hp_yaw_offset_deg) before passing to Steam Audio.
- Do not perform smoothing inside audio thread.
- Add defensive null/orientation fallback.

---

5. Finalize PEQ Cascade Integration

- Use existing PeqBiquadCascade.
- Bind to hp_eq_mode + band parameters.
- Ensure coefficient updates happen off audio thread.

- Insert PEQ after binaural stage.

---

6. Implement Direct FIR Convolver

- Time-domain convolution.
- No heap allocations in processBlock.
- Suitable for ≤ 256 taps.
- Latency = 0.

---

7. Implement Partitioned FFT FIR Convolver

- Uniform partitioned overlap-add.
- Partition size P = nextPow2(blockSize).
- Precompute partition spectra off-thread.
- Maintain ring buffer of input spectra.
- Latency = P samples.
- Report latency via host API.

---

8. Add FIR Engine Selection Logic

- 0 taps → bypass
- ≤ 256 taps → direct convolver
- 256 taps → partitioned FFT
- Swap engine instances atomically.
- Store/report hp_fir_latency_samples.

---

9. Integrate FIR Stage into Monitoring Chain

Final chain when steam_binaural:

1. Quad bed
2. SteamAudioVirtualSurround
3. PEQ (if enabled)
4. FIR (if enabled)
5. Stereo out

Ensure deterministic stage order.

---

10. Add State Version Migration

- Bump state_version to next minor.
- Add new headphone params to migration rules.
- Regenerate golden snapshots.
- Add idempotence migration test.

---

11. Add Basic Verification Test Mode

- Implement internal test tone routing:
  - front impulse
  - rear impulse
- Validate channel routing and orientation behavior.
- Save verification scores to state.

---

12. Add Latency Reporting & Validation

- Call setLatencySamples() when FIR changes.
- Add QA test verifying reported latency matches engine.
- Ensure latency resets properly on bypass.

---

13. Add Runtime Safety Guards

- Assert no allocations in processBlock (debug instrumentation).
- Add null-effect fallback (silence output if misconfigured).
- Clamp extreme yaw offsets.
- Protect against invalid SOFA load.

---

14. Add Quad Channel Order Diagnostic

- Temporary debug toggle to solo each quad channel.
- Verify expected spatial position in binaural output.
- Prevent silent channel order bugs.

---

15. Document Monitoring Architecture

- Add MONITORING_ARCHITECTURE.md:
    - Quad → Binaural path
    - Head tracking flow
    - FIR strategy
    - Latency rules
- Keep aligned with QA harness documentation.

Execution Order (Recommended)

1 → 2 → 3 → 5 → 6 → 7 → 8 → 9 → 4 → 10 → 12 → 11 → 13 → 14 → 15

If you want, I can next generate:

- A structured **multi-phase Codex execution prompt** that instructs GPT-5.3 to implement this incrementally without breaking builds, or

- A **risk analysis section** identifying the 5 most likely failure points in this architecture.

Below is a **structured JSON project spec** designed for Codex (GPT-5.3) staged execution inside joshband/LocusQ.

It is:

- Stage-driven
- Dependency-aware
- Build-safe (incremental)
- Explicit about RT constraints
- Explicit about QA + state migration

You can paste this directly into Codex as the project execution plan.

```
JSON{
  "project": {
    "name": "LocusQ Headphone Calibration & Binaural Monitoring",
    "repo": "joshband/LocusQ",
    "primary_goal": "Implement quad-bed to binaural monitoring with Steam Audio, PEQ,
FIR calibration, and RT-safe architecture.",
    "constraints": {
      "real_time_safe": true,
      "no_allocations_in_processBlock": true,
      "no_locks_in_processBlock": true,
      "state_migration_required": true,
      "qa_harness_compatible": true
    }
  },

  "architecture": {
    "monitoring_chain": [
"Spatial Renderer (quad bed)",
"SteamAudioVirtualSurround (optional)",
"PEQ (optional)",
"FIR Convolver (optional)",
"Stereo Output"
    ],
    "monitoring_modes": [
"speakers",
"steam_binaural",
"virtual_binaural"
    ],
    "quad_channel_order_default": "L_R_Ls_Rs"
  },

  "stages": [

    {
      "id": "S1",
      "name": "Quad Speaker Layout Mapping",
      "objective": "Implement IPLSpeakerLayout mapping for quad bed.",
      "tasks": [
"Create QuadOrder enum.",
"Implement fillQuadSpeakerLayout().",
"Default to L_R_Ls_Rs.",
"Add debug validation for channel order."
      ],
      "deliverables": [
"QuadSpeakerLayout.h",
"Unit test for channel mapping"
      ],
      "build_should_pass": true
    },

    {
      "id": "S2",
      "name": "SteamAudioVirtualSurround Wrapper",
      "objective": "Implement quad → stereo binaural wrapper.",
      "dependencies": ["S1"],
      "tasks": [
"Create SteamAudioVirtualSurround class.",
"Implement init(), shutdown(), reset().",
"Implement processBlock() (RT-safe).",
"Add atomic pointer swap for effect + HRTF."
      ],
      "constraints": [
"No allocations in processBlock.",
"No locks in processBlock."
      ],
      "deliverables": [
"SteamAudioVirtualSurround.h/.cpp"
      ],
      "build_should_pass": true
    },

    {
      "id": "S3",
      "name": "Monitoring Path Integration",
      "objective": "Integrate wrapper into monitoring switch.",
      "dependencies": ["S2"],
      "tasks": [
"Add steam_binaural path.",
"Route quad bed into wrapper.",
"Ensure speakers path remains unchanged.",
"Expose diagnostics (requested vs active mode)."
      ],
      "deliverables": [
"Updated monitoring stage implementation"
      ],
      "build_should_pass": true
    },

    {
      "id": "S4",
      "name": "Head Tracking Injection",
      "objective": "Inject orientation into Steam Audio.",
      "dependencies": ["S3"],
      "tasks": [
"Accept IPLCoordinateSpace3 input.",
"Apply yaw offset parameter.",
"Ensure orientation fallback if unavailable.",
"Do not smooth inside audio thread."
      ],
      "deliverables": [
"Orientation injection in monitoring stage"
      ],
      "build_should_pass": true
    },

    {
      "id": "S5",
      "name": "PEQ Integration",
      "objective": "Insert PEQ cascade after binaural stage.",
      "tasks": [
"Integrate PeqBiquadCascade.",
"Bind parameters from state.",
"Ensure coefficient updates off-thread.",
"Confirm no RT allocations."
      ],
      "deliverables": [
"PEQ stage in monitoring chain"
      ],
      "build_should_pass": true
    },

    {
      "id": "S6",
      "name": "Direct FIR Convolver",
      "objective": "Implement time-domain FIR engine.",
      "tasks": [
"Implement DirectFirConvolver.",
"Support multi-channel stereo.",
"Zero allocations in processBlock.",
"Latency = 0."
      ],
      "deliverables": [
"DirectFirConvolver.h/.cpp"
      ],
      "build_should_pass": true
    },

    {
      "id": "S7",
      "name": "Partitioned FFT FIR Convolver",
      "objective": "Implement uniform partitioned overlap-add engine.",
      "dependencies": ["S6"],
      "tasks": [
"Choose partition size P = nextPow2(blockSize).",
"Precompute partition spectra.",
"Implement ring buffer for input spectra.",
"Implement overlap-add.",
"Expose latencySamples()."
      ],
      "constraints": [
```

"All FFT buffers allocated off-thread.",
"No dynamic memory in processBlock."
      ],
      "deliverables": [
"PartitionedFftConvolver.h/.cpp"
      ],
      "build_should_pass": true
    },

    {
      "id": "S8",
      "name": "FIR Engine Selection",
      "objective": "Switch engine based on tap count.",
      "dependencies": ["S6", "S7"],
      "tasks": [
"Implement engine factory.",
"≤256 taps → Direct.",
">256 taps → Partitioned.",
"Swap engines atomically.",
"Report latency to host."
      ],
      "deliverables": [
"FIR engine manager class"
      ],
      "build_should_pass": true
    },

    {
      "id": "S9",
      "name": "State Migration Update",
      "objective": "Add headphone parameters to state version.",
      "tasks": [
"Bump state_version.",
"Update migration_rules.json.",
"Add golden snapshots.",
"Add idempotence migration test."
      ],
      "deliverables": [
"Updated migration files",
"New golden state files"
      ],
      "build_should_pass": true
    },

```
    {
      "id": "S10",
      "name": "Latency Validation",
      "objective": "Ensure correct latency reporting.",
      "dependencies": ["S8"],
      "tasks": [
"Call setLatencySamples() on FIR change.",
"Add QA test verifying latency.",
"Ensure bypass resets latency."
      ],
      "deliverables": [
"Latency test cases"
      ],
      "build_should_pass": true
    },

    {
      "id": "S11",
      "name": "Verification Test Mode",
      "objective": "Add basic internal spatial verification.",
      "tasks": [
"Implement front impulse test.",
"Implement rear impulse test.",
"Store verification score in state."
      ],
      "deliverables": [
"Verification utility code"
      ],
      "build_should_pass": true
    },

    {
      "id": "S12",
      "name": "Documentation",
      "objective": "Document monitoring architecture.",
      "tasks": [
"Create MONITORING_ARCHITECTURE.md.",
"Document quad mapping.",
"Document FIR latency strategy.",
"Document RT constraints."
      ],
      "deliverables": [
"Documentation file"
      ],
      "build_should_pass": true
    }
  ],

  "success_criteria": [
"Quad bed renders correctly to binaural.",
"SOFA HRTF loads without RT allocation.",
"PEQ and FIR operate without audio glitches.",
"Latency correctly reported and tested.",
"QA harness passes migration + golden tests."
  ]
}
```

If you'd like, I can now generate:

- A **parallel execution variant** optimized for multi-agent Codex sessions, or
- A **risk and failure-mode matrix** tailored to LocusQ's DSP + state architecture.

Excellent. Now we shift from "feature add" to "surgical refactor planning."
This is about not destabilizing LocusQ while introducing a new monitoring spine.

Below are two deliverables:

1. **Phased Refactor Roadmap aligned to a typical LocusQ JUCE + QA harness repo layout**
2. **Parallel multi-agent Codex execution plan (optimized for GPT-5.3 sessions)**

PART I — Phased Refactor Roadmap

(Aligned to LocusQ structure: dsp/, monitoring/, state/, qa/, ui/)

I'll assume a conventional structure similar to what you've described in earlier LocusQ work:

```
Code/Source
  /dsp
  /monitoring
  /state
  /calibrate
  /ui
  PluginProcessor.*
  PluginEditor.*

/state
  migration_rules.json
  goldens/

/qa
  audio-dsp-qa-harness submodule
```

If your exact folder names differ, the conceptual boundaries still apply.

PHASE 0 — Architectural Freeze (No Behavior Change)

Goal:

Prepare repo for monitoring refactor without changing audio output.

Actions:

- Introduce /monitoring folder.
- Extract current monitoring logic from PluginProcessor into:
  - MonitoringStage.h/.cpp
- Introduce enum:
  - MonitoringMode { Speakers, SteamBinaural, VirtualBinaural }
- Wire pass-through only.

Why:

We isolate monitoring before injecting Steam Audio.

Build must behave identically to current main branch.

PHASE 1 — Quad Layout Canonicalization

Goal:

Define quad bed once and forever.

Add:

Code/monitoring/QuadSpeakerLayout.h

Includes:

- QuadOrder enum
- fillQuadSpeakerLayout()
- Debug assertion utility

Also:

- Add temporary debug test to confirm channel ordering.

Risk mitigated:

Silent channel-order mismatch.

PHASE 2 — Steam Audio Virtual Surround Integration

Add:

Code/monitoring/SteamAudioVirtualSurround.h/.cpp

Responsibilities:

- init / shutdown
- default HRTF
- SOFA HRTF swap

- processBlock(quad → stereo)
- atomic pointer swap

Modify:

- MonitoringStage
    - If mode == SteamBinaural, route quad bed here.

RT constraints:

- No allocations in processBlock
- No locks
- No file I/O

Build checkpoint:

Steam Audio path compiles but may not yet be enabled by UI.

PHASE 3 — Head Tracking Injection

Modify:

- /monitoring/MonitoringStage
- Add OrientationProvider interface

Inject:

- Quaternion → IPLCoordinateSpace3
- Yaw offset application

Keep smoothing in bridge layer (not audio thread).

PHASE 4 — PEQ Refactor Integration

Add:

Code/dsp/PeqBiquadCascade.h

Insert stage in MonitoringStage:

Codeif (hp_eq_mode == PEQ)
    peq.processBlock()

Coefficient updates:

- Off-thread only
- Double-buffer swap

Checkpoint:
Audio identical when disabled.

PHASE 5 — FIR Infrastructure

Split cleanly:

Code/dsp/fir/IFirConvolver.h
/dsp/fir/DirectFirConvolver.h
/dsp/fir/PartitionedFftConvolver.h
/dsp/fir/FirEngineManager.h

FirEngineManager:

- Chooses engine by tap count
- Owns atomic pointer
- Reports latency

Modify:

- MonitoringStage
    - Insert FIR after PEQ

Checkpoint:
Direct FIR only first.
Then partitioned.

PHASE 6 — State & Migration Update

Modify:

Code/state/migration_rules.json
/state/goldens/

Add:

- hp_enabled
- hp_eq_mode
- hp_hrtf_mode
- hp_fir_taps
- etc.

Run:

- Golden snapshot regeneration
- Idempotence migration test

Checkpoint:
QA harness passes.

PHASE 7 — Latency Contract Enforcement

Add:

- Host latency reporting hook
- QA test verifying latency changes
- Reset latency on bypass

Critical:
No hidden delay drift between monitoring modes.

PHASE 8 — Verification Utilities

Add:

Code/monitoring/SpatialVerification.h

Implements:

- Front impulse
- Rear impulse
- Mono collapse test

Store scores in state.

PHASE 9 — Documentation & Stability Sweep

Add:

Code/docs/MONITORING_ARCHITECTURE.md

Include:

- Quad mapping diagram
- Monitoring chain diagram
- FIR decision tree
- Latency rules
- RT safety guarantees

PART II — Parallel Multi-Agent Codex Execution Plan

This is how to avoid stepping on your own feet.

We divide into four parallel tracks:

Track A — Monitoring Core (Agent A)

Scope

- Quad mapping
- SteamAudioVirtualSurround
- MonitoringStage refactor

Constraints

- No state changes
- No FIR
- No PEQ

Deliverables

- Fully functional quad → binaural wrapper
- Clean monitoring stage

Agent A must not touch:

- state/
- dsp/fir/
- qa/

Track B — FIR System (Agent B)

Scope

- IFirConvolver
- DirectFirConvolver
- PartitionedFftConvolver

- FirEngineManager

Constraints

- Standalone unit test harness
- No dependency on monitoring yet

Deliverable:

- FIR module compiles independently.

Agent B must not modify:

- MonitoringStage
- SteamAudioVirtualSurround

Track C — PEQ & Calibration (Agent C)

Scope

- PeqBiquadCascade
- HeadphoneCalibrationProfile codec
- Parameter wiring

Constraints

- No Steam Audio changes
- No FIR engine integration yet

Deliverable:

- PEQ stage compiles and processes stereo buffers.

Track D — State & QA (Agent D)

Scope

- Migration update
- Golden regeneration
- Latency QA tests

Constraints

- No DSP changes
- No Steam Audio edits

Deliverable:

- State tests pass on CI.

Integration Phase (Sequential)

After A/B/C/D are green:

1. Integrate PEQ into MonitoringStage.
2. Integrate FIR engine.
3. Connect state params to monitoring.
4. Enable steam_binaural path.
5. Final latency validation.

Why This Parallel Strategy Works

- Monitoring and FIR are isolated.
- State changes do not block DSP implementation.
- QA harness work does not interfere with audio thread work.
- Each agent works in a bounded domain.

Critical Coordination Rules for Codex

- Each stage must:
    - Compile cleanly
    - Not introduce allocations in processBlock
    - Not introduce new warnings
- No agent edits files outside its declared scope.
- Integration occurs only after all tracks are green.

If you want next-level rigor, I can generate:

- A **Git branch strategy** (exact branch names + merge order),
- Or a **risk matrix identifying the 7 highest failure modes** in this architecture and how to guard against them.