Here's a fast, practical way to ship your JUCE plugins as **CLAP** (with per-note modulation + voice info) and keep your realtime DSP pristine while giving the UI rich, per-voice visuals.

Why CLAP (in 1 minute)

- Open, MIT-licensed plugin standard; strong adoption and modern features like **polyphonic parameter modulation** and **voice-info** (per-voice IDs). [GitHub+2CLever Audio+2]
- For JUCE projects today, use the **community bridge**: clap-juce-extensions (JUCE 6/7/8) to emit .clap builds and opt into extended capabilities. [GitHub+1]

Minimal integration path for a JUCE plugin

1. **Add CLAP support**
    - Drop in **clap-juce-extensions**; follow its CMake/juce_add_plugin wiring to add the CLAP target alongside VST3/AU. Start with stock build, then enable extensions as needed. [GitHub]
2. **Enable modern CLAP features**
    - Implement parameter + modulation extensions via the helper interfaces in the repo (clap-juce-extensions.h) to expose fine-grained modulation and discover **voice IDs** from the host. [GitHub]
3. **Keep DSP authoritative; snapshot for UI**
    - In your processor: translate CLAP note/param events → (voice_id, param_id, value) tuples and push **compact summaries** into a **lock-free ring buffer** (audio thread only).
    - On the UI thread: consume snapshots at ~30–60 Hz to render **per-note lanes**, **voice meters**, and a **mod-matrix**—no heavy state leaves the DSP.
    - CLAP's C API examples + reference code help when validating your event flow. [GitHub]

4. **QA & validation**
    - Use **clap-validator** and **clap-info** in CI to sanity-check your artifacts (IDs, extensions, threading). [GitHub]

Sketch: adapter you can drop in

```
C++// Audio thread (DSP)
structVoiceMsg { int32_t voiceId; int32_t paramId; float delta; uint32_t frame; };
voidclap_event_to_voice_message(int32_tvoiceId, int32_tparamId, floatdelta,
uint32_tframe);

// UI thread
structVoiceSnapshot { int32_t voiceId; float lastValue; float env; /* etc. */ };
std::vector<VoiceSnapshot>pull_voice_snapshot(); // consume ring buffer
```

Back your VoiceMsg queue with a **lock-free SPSC ring** (no allocs/locks in processBlock). The UI builds derived state (per-voice meters, mod lanes), while DSP retains the ground truth.

Useful repos & docs

- **CLAP spec & reference** (extensions: params, modulation, voice-info) — primary source. [GitHub+1]
- **clap-juce-extensions** — JUCE bridge + helper interfaces; active maintenance. [GitHub]
- **Example CLAP plugins / host / tools** — templates, host samples, validator. [GitHub+1]
- **JUCE forum context** — ecosystem & adoption chatter (nice sanity read). [JUCE+1]

Drop-in CI idea (matrix build)

- Build VST3, AU, CLAP targets; run clap-validator on artifacts; print clap-info to logs for traceability. [GitHub]

If you want, I can draft a small JUCE example branch that:

- adds clap-juce-extensions,
- wires a **realtime SPSC ring**,
- exposes **voice-info** + **per-note modulation**,
- and includes a lightweight **ImGui/juce::Component** UI to visualize per-voice lanes.

You're absolutely right. This is directly relevant to **joshband/LocusQ**.

LocusQ is not just "another quad plugin." It's a spatial instrument. And CLAP gives you architectural primitives that match what you're already building in Praxis and the QA harness: deterministic DSP core, clear UI contract, and future-proof modulation routing.

Let's be blunt:

If LocusQ is about spatial voices in a room, CLAP's **voice info** + **polyphonic modulation** is a better conceptual fit than VST3/AU alone.

Why CLAP Specifically Matters for LocusQ

1. Per-Voice Spatial State

LocusQ is quadraphonic. Each event in the room can logically map to:

- voice_id
- (x, y) position
- depth / distance attenuation
- diffusion / room interaction
- envelope / trajectory

CLAP voice-info lets you explicitly bind modulation and parameter changes to a voice_id. That's not a hack. That's first-class.

That means:

- A moving source = per-voice modulation

- A rotating emitter = param modulation tied to voice
- Multiple objects in the room = independent modulation streams

VST3 can fake this.
CLAP is designed for it.

---

2. UI Visualization Becomes Truthful

Your LocusQ style guide emphasizes clarity and instrument-like control. Good.

With CLAP voice IDs:

- You can render spatial trails per voice
- Display independent per-voice meters
- Show modulation vectors in the wireframe room
- Avoid guessing which voice triggered what

You don't pollute `processBlock()`.
You snapshot compact voice telemetry into a lock-free ring (which aligns perfectly with your DSP_INVARIANTS and no-allocation rule).

---

3. Clean Separation of Concerns

For LocusQ architecture:

DSP thread:

- Processes voices
- Applies per-voice modulation
- Emits `(voice_id, param_id, delta, frame)` events

UI thread:

- Consumes snapshots
- Renders spatial overlays
- No direct DSP state mutation

That matches your QA harness philosophy: deterministic core, observable state.

Architectural Fit with Your Existing Repos

You already have:

- audio-dsp-qa-harness enforcing determinism
- Praxis enforcing adapter boundaries
- ADR docs around UI/DSP contracts

CLAP gives you:

- Host → DSP voice-level metadata
- Explicit modulation pipeline
- Modern host lifecycle model

That means:

- You can validate modulation determinism in QA
- You can build regression tests around voice mapping
- You can simulate polyphonic modulation events offline

That's extremely powerful for spatial DSP.

Strategic Take

If LocusQ is:

- A quad spatial effect
- Possibly evolving into an instrument-like spatial tool
- UI-driven but DSP-authoritative
- Designed with long-term architecture discipline

Then adding CLAP support is not feature creep.

It’s alignment.

Implementation Strategy for LocusQ (Practical, Not Romantic)

1. Add clap-juce-extensions as optional target.
2. Ship VST3/AU/CLAP in parallel.
3. Enable:
    - params
    - voice-info
    - modulation
4. Build a minimal lock-free SPSC ring for:
    - voice telemetry
    - spatial state
5. Expose a debug overlay mode in LocusQ UI:
    - Voice ID labels
    - Real-time modulation strength
    - Spatial motion vectors

That gives you:

- Developer introspection
- Marketing screenshots
- QA observability
- Future-proof architecture

Hard Truth

If you *don't* adopt CLAP, you're leaving expressive poly modulation and voice-level clarity on the table. Not fatal. But suboptimal for a spatial instrument.

And you're not building a toy. You're building infrastructure.

If you want, I can:

- Draft a LocusQ-specific CLAP integration plan
- Define a VoiceTelemetryRing struct aligned to your QA invariants
- Or map CLAP modulation → quad spatial transforms mathematically

Spatial DSP is a weird frontier. That's exactly why it's fun.

Good. Let's do this like an engineering decision document, not a vibe check.

You're building **LocusQ** as a quadraphonic spatial instrument/effect. The core already values determinism, no-alloc DSP, and observable state (per your QA harness discipline). CLAP isn't a checkbox format here. It's a structural advantage.

Below is a LocusQ-specific integration plan that respects your architecture and avoids format creep.

LocusQ – CLAP Integration Plan (v1)

0. Objective

Add CLAP support to LocusQ in a way that:

- Preserves deterministic DSP behavior
- Introduces per-voice spatial modulation
- Enables UI observability without breaking RT constraints
- Remains optional and parallel to VST3/AU builds
- Aligns with your QA harness regression model

No speculative abstraction layers. No premature complexity.

---

1. Build & Format Layer

1.1 Add CLAP Target

- Integrate clap-juce-extensions
- Add CLAP format in parallel with:
    - VST3
    - AU
- Keep identical processor class

Goal:
Single DSP core. Multiple format adapters.

Do **not** fork logic for CLAP. Format differences live at the edge.

---

2. Capability Scope (LocusQ-Specific)

Enable only what LocusQ actually benefits from:

Required CLAP Extensions

- clap.plugin.params
- clap.plugin.voice-info
- clap.plugin.note-ports
- clap.plugin.audio-ports

Optional (phase 2):

- clap.plugin.note-expression
- clap.plugin.latency

Skip everything else initially.

---

3. Voice Model for LocusQ

This is where CLAP becomes meaningful.

3.1 Voice Definition

For LocusQ, a “voice” = a spatial source instance.

Define:

```
C++structSpatialVoiceState
{
int32_t voiceId;
float x;          // -1..1
float y;          // -1..1
float depth;      // 0..1
float gain;
float azimuth;
float distance;
uint32_t startFrame;
};
```

This struct must:

- Be POD
- No heap allocations
- Fixed-size

3.2 Mapping CLAP Voice → Spatial Engine

CLAP provides:

- voice_id
- per-note modulation

LocusQ should:

- Maintain `voiceId -> SpatialVoiceState` mapping

- Store in a fixed-capacity array
- No dynamic resizing

Example:

```
C++constexprintkMaxVoices = 64;
SpatialVoiceStatevoices[kMaxVoices];
```

Voice allocation strategy:

- Deterministic reuse
- No unordered containers
- No std::map

---

4. Modulation Pipeline

This is the real reason you're doing this.

CLAP supports polyphonic modulation per parameter.

Define spatial parameters as modulate-able:

- Position X
- Position Y
- Depth
- Rotation
- Diffusion
- Room Size (optional per voice)

Inside processBlock:

1. Receive CLAP events
2. Convert to:
    - (voice_id, param_id, value, frame_offset)
3. Apply modulation at sample-accurate frame

Important:
Never mutate global parameter state.
Apply modulation only to voice-local state.

---

5. Real-Time Telemetry Channel (UI Bridge)

You already care about DSP/UI contracts. Do not cheat here.

5.1 Lock-Free SPSC Ring

Audio thread → UI thread

Define:

```
C++structVoiceTelemetryEvent
{
int32_t voiceId;
float x;
float y;
float gain;
uint32_t frame;
};
```

Ring buffer:

- SPSC
- Fixed capacity
- No heap
- Overwrite oldest on overflow

Audio thread:

- Push compact telemetry snapshot
- No branching based on UI state

UI thread:

- Pull at 30–60Hz
- Build interpolated render model

---

6. UI Integration (LocusQ Style Guide Alignment)

CLAP allows you to visually differentiate:

- Active voices
- Per-voice modulation strength
- Independent motion vectors

Add Debug Mode:

- Show voice IDs in wireframe room
- Show modulation magnitude as radial line
- Fade inactive voices

Do not tie UI lifecycle to voice lifetime.
UI is a consumer, not authority.

---

7. QA Harness Integration

This is where you gain leverage.

7.1 Deterministic Voice Tests

Add QA tests for:

- Voice creation order
- Modulation determinism
- Identical output given identical event stream
- Stable voice reuse policy

Use synthetic CLAP event sequences:

- Simulate poly modulation bursts
- Validate spatial output consistency

Golden reference WAV per:

- 1 voice
- 4 voices
- 16 voices
- Modulated rotation sweep

---

8. Latency & Block Boundaries

CLAP supports sample-accurate events.

You must:

- Respect frame offsets inside blocks
- Avoid assuming block start alignment
- Validate sample-accurate modulation inside QA

Spatial transforms must be continuous across block boundaries.

---

9. Failure Modes to Avoid

Be ruthless here.

Do NOT:

- Allocate per voice on note-on
- Use std::vector resize in audio thread
- Leak CLAP-specific branching into core spatial engine
- Mix global parameter state with per-voice state
- Depend on UI pull timing for DSP state stability

---

10. Phased Rollout

Phase 1

- Basic CLAP format
- Per-voice position modulation
- Telemetry ring
- QA determinism tests

Phase 2

- Note-expression support
- Per-voice rotation
- Host automation stress testing

Phase 3

- CLAP-specific advanced modulation routing UI
- Per-voice spatial envelopes

Strategic Payoff

This turns LocusQ into:

- A spatially polyphonic instrument
- A modulation-aware spatial processor
- A host-integrated modern plugin
- A regression-testable polyphonic engine

Not hype. Structural capability.

If you want next-level rigor, we can:

- Define the exact memory layout and cache-line alignment strategy

- Draft the modulation-to-quad panning math with per-voice interpolation
- Or design a CLAP-specific ADR for LocusQ's repo

Spatial DSP isn't crowded territory. Which means the architectural decisions matter more than the marketing copy.

Good. We're going to treat this like infrastructure, not UI glitter.

You've been consistent about DSP invariants:

- No allocations in processBlock
- No locks
- Deterministic behavior
- Explicit DSP → UI contract
- Bounded memory
- Reproducible offline rendering

So the VoiceTelemetryRing must:

- Be fixed-capacity
- Be POD-only
- Use SPSC (single-producer audio thread → single-consumer UI thread)
- Avoid dynamic memory
- Avoid undefined behavior under overflow
- Be trivially resettable for deterministic test harness runs

Let's design it properly.

Design Goals

1. **Audio thread writes only**
2. **UI thread reads only**
3. No heap

4. No mutex
5. No atomics with expensive ordering
6. Deterministic overflow behavior
7. Fixed memory footprint
8. Snapshot-safe

We'll use a classic SPSC ring with:

- std::atomic<uint32_t> write index
- std::atomic<uint32_t> read index
- Power-of-two capacity
- Mask-based wrapping

Acquire/Release ordering only. No seq_cst.

Telemetry Payload (Strictly Minimal)

Keep it small. This runs in the audio thread.

```
C++structalignas(32) VoiceTelemetryEvent
{
int32_tvoiceId;
uint32_tframeOffset;   // within block
floatx;
floaty;
floatgain;
floatazimuth;
floatdistance;
};
```

Why this shape?

- No std:: types
- No padding surprises (aligned to 32 bytes for cache friendliness)
- Pure POD

- No pointers
- No ownership semantics
- Small enough to push frequently

VoiceTelemetryRing

Power-of-two capacity required.

```
C++template<size_tCapacityPow2>
classVoiceTelemetryRing
{
static_assert((CapacityPow2& (CapacityPow2-1)) ==0,
"Capacity must be power of two");

public:
VoiceTelemetryRing() noexcept
      : writeIndex(0), readIndex(0)
   {
clear();
   }

// Audio thread only
bool push(constVoiceTelemetryEvent& event) noexcept
   {
constuint32_tw = writeIndex.load(std::memory_order_relaxed);
constuint32_tr = readIndex.load(std::memory_order_acquire);

if ((w-r) >=CapacityPow2)
      {
// Overflow policy: drop oldest deterministically
readIndex.store(r+1, std::memory_order_release);
      }

buffer[w&mask] = event;
writeIndex.store(w+1, std::memory_order_release);
returntrue;
   }

// UI thread only
```

```
bool pop(VoiceTelemetryEvent& out) noexcept
   {
constuint32_tr = readIndex.load(std::memory_order_relaxed);
constuint32_tw = writeIndex.load(std::memory_order_acquire);

if (r==w)
returnfalse;

out = buffer[r&mask];
readIndex.store(r+1, std::memory_order_release);
returntrue;
   }

void clear() noexcept
   {
writeIndex.store(0, std::memory_order_relaxed);
readIndex.store(0, std::memory_order_relaxed);
   }

constexprsize_t capacity() constnoexcept
   {
returnCapacityPow2;
   }

private:
staticconstexpruint32_t mask = CapacityPow2-1;

alignas(64) std::atomic<uint32_t>writeIndex;
alignas(64) std::atomic<uint32_t>readIndex;

alignas(64) VoiceTelemetryEvent buffer[CapacityPow2];
};
```

Why This Aligns With Your QA Invariants

1. No allocations

All memory fixed at compile time.

2. Deterministic overflow

When full:

- Oldest event is dropped.
- Always exactly one slot reclaimed.
- No branching chaos.
- Reproducible behavior offline.

If you want “drop newest” instead, that’s one line change — but dropping oldest preserves recency in UI visualization.

3. Thread discipline enforced by design

- Audio thread = push only
- UI thread = pop only

If someone violates that contract, your QA harness should catch it.

4. Sample-accurate friendly

`frameOffset` allows interpolation on UI side without forcing UI to track block timing.

5. Cache-conscious

- Indices separated and cache-line aligned
- Buffer aligned
- No false sharing between producer/consumer

6. Offline render friendly

For QA:

- Call `clear()` before render.
- Inject deterministic modulation stream.
- Consume entire ring after render.
- Compare against expected telemetry snapshot.

That means you can regression-test spatial trajectories — not just audio output.

Capacity Guidance for LocusQ

Let's reason this out.

Worst case:

- 64 voices
- 128 sample block
- 2–4 telemetry events per voice per block

That's roughly:
64 * 4 = 256 events per block

At 60 FPS UI pull rate, you might accumulate ~10 blocks between pulls.

Safe capacity:

C++VoiceTelemetryRing<4096>ring;

That gives you breathing room without blowing L1 cache.

Things You Must Not Do

- Do not store std::vector in events.
- Do not store pointers.
- Do not read ring from audio thread.
- Do not let UI block waiting for pop.
- Do not add dynamic resizing "just in case".

If it overflows, that's a signal your UI is too slow — not a DSP failure.

If You Want to Go Even Harder

We can:

- Add compile-time static_assert on sizeof(VoiceTelemetryEvent)
- Add high-watermark debug counter (non-RT)
- Add telemetry checksum for QA determinism validation
- Integrate ring state snapshot into your QA harness golden logs

Spatial DSP isn't just math. It's observability discipline.

And this ring is the spinal cord between your deterministic engine and your instrument UI.

Let's get straight into the math that's practical for quad spatialization. You've got four speakers arranged in a square: front-left (FL), front-right (FR), rear-left (RL), rear-right (RR). We'll assume a normalized coordinate space where the listener is at (0,0) and the speakers are at (-1,1), (1,1), (-1,-1), (1,-1).

1. CLAP Modulation Input

CLAP modulation can be per-voice and per-parameter. We'll focus on three core parameters that will shape spatial positioning:

- **X Position**: Modulated per voice (horizontal position, -1 to 1)
- **Y Position**: Modulated per voice (front-back position, -1 to 1)
- **Gain/Amplitude**: Modulated per voice (scaling factor, 0 to 1 or beyond if you allow boost)

You'll apply these modulated parameters per voice. Every block, you'll receive updated modulations. Your job is to compute speaker gains (FL, FR, RL, RR) per voice and sum them.

2. Coordinate-Based Panning

We map the modulated (X, Y) coordinates of a voice to quad speaker gains using a bilinear interpolation approach—also known as vector-based amplitude panning for a quad setup.

- Imagine the coordinate plane (X, Y) as a square where:

- FL speaker is at (-1, 1)
- FR speaker is at (1, 1)
- RL speaker is at (-1, -1)
- RR speaker is at (1, -1)

Given a voice's modulated (X, Y) position, the speaker gains are derived from how close the voice is to each speaker.

3. Bilinear Weighting

For a voice at (x, y) with x in [-1, 1] and y in [-1, 1], the weight for each speaker is calculated from proximity to that speaker.

Let's define weights:

- $w_{FL} = (1 - x) \times (1 + y) / 4$
- $w_{FR} = (1 + x) \times (1 + y) / 4$
- $w_{RL} = (1 - x) \times (1 - y) / 4$
- $w_{RR} = (1 + x) \times (1 - y) / 4$

This ensures each weight is between 0 and 1. When the voice is exactly on top of a speaker, that speaker's weight is 1, and others are 0. At (0, 0), all weights are 0.25, evenly distributed.

4. Apply Gain Modulation

Once you've got the spatial weights, apply the per-voice modulated gain (let's call it G_{mod}):

- $G_{FL} = w_{FL} \times G_{mod}$
- $G_{FR} = w_{FR} \times G_{mod}$
- $G_{RL} = w_{RL} \times G_{mod}$
- $G_{RR} = w_{RR} \times G_{mod}$

This gives you per-voice gain values for each speaker.

5. Summing All Voices

If you have multiple voices active, simply sum their speaker contributions:

For each speaker:

- $G_{FL}^{total} = \sum_{voices} G_{FL}(voice)$
- $G_{FR}^{total} = \sum_{voices} G_{FR}(voice)$
- $G_{RL}^{total} = \sum_{voices} G_{RL}(voice)$
- $G_{RR}^{total} = \sum_{voices} G_{RR}(voice)$

This summation happens in your processBlock and gives you the total signal scaling for each speaker per block.

6. Normalize (Optional, but Wise)

Depending on how you handle scaling in your DSP, you might want to normalize to avoid clipping. If the sum of weights across voices exceeds 1 per speaker, you can normalize the total gain by the maximum value:

- $\maxGain = \max(G_{FL}^{total}, G_{FR}^{total}, G_{RL}^{total}, G_{RR}^{total})$

If $\maxGain > 1$, scale each speaker's gain by $1 / \maxGain$.

7. Spatial Rotation Modulation (Optional Extension)

If you want to allow modulation of rotation (e.g., spinning the room around the listener), introduce an angle parameter θ (modulated per voice).

Given an angle θ, rotate the coordinates before calculating speaker weights:

- $x' = x\cos(\theta) - y\sin(\theta)$
- $y' = x\sin(\theta) + y\cos(\theta)$

Then apply the same bilinear weighting as above using (x', y').

This would allow modulation to "rotate" the perceived space around the listener.

Summary

1. Use CLAP modulation to dynamically update (X, Y) position and gain for each voice.
2. Compute speaker weights via bilinear interpolation based on the (X, Y) coordinates.
3. Apply the modulated gain to each speaker's weight.
4. Sum all voices into final speaker gains per block.
5. Optionally normalize if total gain exceeds 1.

This keeps the math efficient, sample-accurate, and fully tied to your per-voice CLAP modulation parameters. It's deterministic, which aligns nicely with your QA harness and real-time constraints.

Short answer: yes. It needs fleshing out — but not in the way marketing implies.

The math we outlined is fine as a *core panner*. What it does not yet cover is:

- Host capability variability
- Polyphonic modulation semantics
- Sample-accurate event timing
- Energy normalization
- MIDI 2.0 / note-expression mapping
- QA determinism under modulation stress
- CPU scaling under voice bursts

CLAP gives you expressive infrastructure. LocusQ needs a disciplined mapping layer that respects your invariants.

Let's break this down cleanly.

---

1. The Panning Math Is Correct — But Not Perceptually Neutral

The bilinear quad weights:

$$w_{FL} = (1 - x)(1 + y)/4$$

etc.

They are geometrically correct for square interpolation. But they are **not constant power**.

At the center (0,0), you get 0.25 per speaker.
Sum of gains = 1.
Sum of squared gains = 0.25.

At a corner (1,1), you get one speaker at 1.
Sum of squared gains = 1.

That means the center is perceptually quieter.

If LocusQ is an instrument — not just a utility — this matters.

You likely want constant-power normalization:

$$\text{normalize} = \sqrt{w_{FL}^2 + w_{FR}^2 + w_{RL}^2 + w_{RR}^2}$$

Then:

$$w_i = \frac{w_i}{\text{normalize}}$$

That preserves perceived energy across the field.

Without this, modulation-driven motion will “breathe” in volume.

---

2. CLAP Modulation Is Sample-Accurate — Your Mapping Must Be

CLAP delivers per-voice modulation events with frame offsets.

If you compute panning once per block, you lose:

- Rapid LFO sweeps
- Transient positional modulation
- Polyphonic envelopes

For LocusQ, the correct implementation is:

Inside process block:

For each sample frame:

1. Apply modulation deltas scheduled for that frame
2. Update voice-local x,y,gain
3. Compute weights
4. Accumulate output

Block-level panning is architecturally wrong for CLAP poly modulation.

---

3. Voice Lifecycle Must Be Explicit

CLAP gives:

- note on
- note off
- voice id reuse

You must define:

- What happens when a voice dies mid-block?
- Does spatial interpolation continue through release?
- Is position frozen at note-off?

For determinism, you likely want:

Voice state persists until envelope reaches zero.
Then slot becomes reusable.

Never reuse voiceId until release stage fully ends.

Otherwise QA regressions become nightmares.

---

4. MIDI 2.0 / Note Expression Mapping

CLAP supports per-note expression beyond MPE.

You should explicitly map:

- Note pitch → optional spatial radius?
- Note velocity → depth?
- Per-note pressure → rotation speed?
- Timbre → diffusion?

Don’t let this become implicit magic.

Define a clear modulation matrix:

CodeCLAP param → spatial transform

Example:

- Poly mod param 1 → X offset
- Poly mod param 2 → Y offset
- Poly mod param 3 → depth
- Poly mod param 4 → azimuth rotation

Code
Deterministic. Documented. Testable.

---

5. Host Capability Variability

Not all hosts support all extensions.

For LocusQ:

You must gracefully degrade:

If host:
- Has no poly modulation → fall back to global params
- Has no note-expression → ignore voice-specific transforms
- Has unstable voice-info → treat as monophonic

Never assume Bitwig-level support everywhere.

---

6. Multicore & Scaling

CLAP is multicore-friendly.

But quad spatial math is cheap.

The real cost is:

- Per-sample per-voice modulation
- Constant-power normalization
- Rotation math (sin/cos)

Optimization strategy:

- Precompute cos/sin per voice per block if rotation is slow-varying
- Avoid sqrt per sample unless energy normalization required
- Use fast approximations if necessary (but verify QA drift)

---

7. QA Harness Implications

You now must regression-test:

1. Modulated motion paths
2. Energy invariance across field
3. Voice reuse determinism
4. Host-like poly modulation bursts

Add golden tests like:

- 16 voices orbiting center
- Poly LFO on X only
- Rapid note on/off storms

Compare rendered quad waveforms and telemetry ring snapshots.

If CLAP integration breaks determinism, you will know.

---

8. Rotation Is Not Enough

Simple 2D rotation:

$$
x' = x\cos\theta - y\sin\theta
$$

is mathematically correct.

But spatial perception is not Euclidean.

If LocusQ aspires beyond “quad panner”:

You may later want:

- Distance attenuation curves
- Diffuse field modeling
- Non-linear warping of space
- Room boundary reflections

CLAP modulation makes that feasible.
But don’t overbuild in phase one.

Strategic Take

CLAP isn’t magic. It’s infrastructure.

For LocusQ to truly benefit, you must:

- Make panning constant-power
- Make modulation sample-accurate
- Define explicit voice lifecycle
- Lock modulation-to-spatial mapping
- Harden determinism under stress

If you do that, LocusQ stops being “quad effect” and becomes a modulation-aware spatial

instrument.

If you don’t, you’re just shipping another panner with a new format badge.

And that would be a waste of a very modern API.

You don’t “address” that by repeating it in a README.

You address it architecturally.

CLAP’s stable ABI + extension model is not marketing copy. It means one thing for LocusQ:

You must treat CLAP as a **capability-negotiated runtime**, not a guaranteed feature surface.

If you don’t, you’ll ship something that works beautifully in Bitwig and falls apart elsewhere.

Let’s convert that wall of prose into engineering moves.

---

1. Stable ABI → Strict Boundary Layer

CLAP’s stable ABI means your binary must remain compatible across hosts and versions.

For LocusQ, that implies:

- A thin CLAP adapter layer.
- Zero CLAP headers leaking into your core spatial engine.
- No conditional logic inside DSP based on host identity.

Design:

```
Code[LocusQ Spatial Engine] ← format-agnostic
      ↑
[CLAP Adapter Layer]
      ↑
[Host Extensions Negotiation]
```

Your spatial engine never sees `clap_host` structs. It sees clean events:

CodeVoiceStart
VoiceModulation
VoiceEnd
GlobalParamChange

That keeps ABI stability from infecting your DSP core.

---

2. Extension Model → Capability Negotiation Table

CLAP extensions are optional.

Hosts may support:

- CLAP_EXT_PARAMS
- CLAP_EXT_NOTE_PORTS
- CLAP_EXT_NOTE_EXPRESSION
- CLAP_EXT_VOICE_INFO
- None of the above

So LocusQ must implement:

A runtime capability table at plugin init.

Example internal struct:

```
Codestruct ClapCapabilities
{
   bool hasParams;
   bool hasNotePorts;
   bool hasVoiceInfo;
   bool hasNoteExpression;
   bool supportsPolyMod;
};
```

At activation:

- Query each extension.
- Store result.
- Never branch on host name.

- Only branch on capability flags.

If hasVoiceInfo == false:
→ LocusQ operates in mono-voice mode.

If hasNoteExpression == false:
→ Per-voice modulation disabled.
→ Fall back to global parameters.

That is how you survive host variability.

---

3. clapdb.tech → Test Matrix Strategy

The compatibility matrix is not trivia. It is your test plan.

You need:

Tier A Hosts (full poly mod support)

- Bitwig
- MultitrackStudio

Tier B Hosts (partial support)

Tier C Hosts (basic CLAP only)

For each tier, define expected behavior:

Tier A:

- Per-voice spatial modulation
- Sample-accurate note expression
- Voice lifecycle correctness

Tier B:

- Voice IDs but no poly mod

Tier C:

- Global spatial control only

Then your QA harness can simulate:

- Poly mod bursts
- Missing extension environment
- Voice info disabled mode

You do not want to discover this at user launch.

---

4. Real Hosts Using Poly Mod → What That Means for LocusQ

When MultitrackStudio or Bitwig drives per-note LFOs:

You will receive:

- Voice-specific parameter modulation events
- Possibly high-frequency per-voice deltas
- Concurrent modulation on multiple parameters

If LocusQ:

- Updates spatial position only at block boundaries
- Does not apply frame offsets correctly
- Reuses voice IDs prematurely

It will glitch.

So you must implement:

- Sample-accurate modulation queue
- Per-voice state containers
- Deterministic voice allocation policy
- Proper release-stage handling

Otherwise, poly modulation becomes undefined behavior.

---

5. MIDI 2.0 / Extended Expression → Explicit Mapping Layer

CLAP may deliver:

- Per-note pitch
- Per-note pressure
- Per-note timbre
- Per-note brightness
- 32-bit resolution values

LocusQ must not implicitly treat these as spatial data.

Define a modulation matrix:

CodeExpression Source → Spatial Target

Example:

- Note pressure → radial distance
- Timbre → diffusion coefficient
- Brightness → elevation (if later expanded to 3D)
- Pitch bend → rotational velocity

Make this user-configurable or at least documented.

If you leave it implicit, users will not understand spatial motion.

---

6. Non-Destructive Modulation → Internal Param Architecture

CLAP modulation is layered on top of base parameter values.

So in LocusQ:

Do not mutate base parameter state when modulation arrives.

Instead:

CodeeffectiveValue = baseParam + modulationDelta

Per voice.

That means your internal param model must separate:

- Base param storage
- Modulation accumulators
- Voice-local overrides

This aligns beautifully with your QA harness because you can test base vs modulated output separately.

---

7. Multicore Threading → Don't Overreact

CLAP supports multicore-friendly designs.

But LocusQ's spatial math is cheap.

Do not prematurely shard per voice across threads.

Instead:

- Keep per-voice spatial math in the audio thread.
- Only parallelize if profiling proves need.

Stability > theoretical parallelism.

---

8. What You Actually Need to Add to LocusQ

Concrete actions:

1. Capability negotiation layer.
2. Explicit modulation matrix.
3. Sample-accurate event application.
4. Per-voice lifecycle state machine.
5. Deterministic voice reuse policy.
6. Host-tier test plan.

7. Fallback mode for missing extensions.

If you do that, CLAP becomes a structural advantage.

If you skip that and just “enable poly mod,” you are relying on ideal host behavior.

The Real Question

Does LocusQ want to be:

A quad panner that happens to compile as CLAP?

Or

A modulation-aware spatial instrument that fully exploits per-voice expressivity?

If it’s the second, then you must formalize:

- Capability negotiation
- Deterministic per-voice mapping
- Sample-accurate modulation
- Explicit expression routing

Otherwise, CLAP’s extension model becomes noise.

And noise is the one thing we are trying to control.

Below is a formalization you can drop into **LocusQ** as an internal contract + implementation blueprint. It’s written as “how it must behave” plus concrete data structures and algorithms that keep you inside your QA invariants (no allocs, no locks, deterministic).

---

1) Capability negotiation

Goal

At runtime, detect which CLAP extensions the host actually supports, then lock LocusQ into a predictable “mode.” No host-name hacks. No mid-session feature flipping.

Contract

1. Negotiation happens once at plugin init/activate.
2. Results are stored in an immutable `Capabilities` struct.
3. The DSP core never queries CLAP directly; it reads `Capabilities`.
4. If an extension is missing, LocusQ must degrade to a defined fallback mode.

Modes

Define 3 explicit runtime modes:

- **Mode A: Poly/Voice**
  Host provides reliable voice IDs and per-voice modulation (or note-expression).
  LocusQ uses per-voice state and per-voice modulation.
- **Mode B: Voice IDs only**
  Host provides voice IDs but not per-voice modulation/expression.
  LocusQ uses per-voice state, but only global params apply.
- **Mode C: Global only**
  No voice ID support.
  LocusQ runs as a single spatial “source,” global params only.

Implementation shape

```
C++structCapabilities
{
bool hasParams          = false;
bool hasNotePorts       = false;
bool hasVoiceInfo       = false;
bool hasNoteExpression    = false;
bool hasPolyMod         = false;   // if supported in your adapter
// Add only what you use
};

enumclassRuntimeMode : uint8_t
{
PolyVoice,
VoiceOnly,
GlobalOnly
};
```

```
structNegotiationResult
{
Capabilities caps;
RuntimeMode  mode = RuntimeMode::GlobalOnly;
};
```

Deterministic decision rule

```
Plain textif hasVoiceInfo && (hasNoteExpression || hasPolyMod) -> PolyVoice
else if hasVoiceInfo -> VoiceOnly
else -> GlobalOnly
```

This makes behavior predictable and testable.

---

2) Deterministic per-voice mapping

Goal

Map incoming CLAP “voice identity” to a stable LocusQ `VoiceSlot` without allocations, without non-deterministic containers, and with deterministic reuse.

Contract

1. Fixed maximum voices: `kMaxVoices`.
2. No heap allocations on note-on/off.
3. Given the same event sequence, voice slot assignment must be identical.
4. Voice IDs are treated as opaque host-provided integers. You map them to slots.
5. A slot is not reusable until the voice has fully released (envelope ended).

Data model

Use a fixed array of voice slots + a fixed mapping table.

```
C++constexprintkMaxVoices = 64;

enumclassVoiceStage : uint8_t { Off, Active, Release };

structVoiceSlot
{
```

```
VoiceStage stage = VoiceStage::Off;
int32_t    clapVoiceId = -1;   // host voice id
uint32_t   age = 0;            // monotonic counter for deterministic stealing
float      x = 0.f;
float      y = 0.f;
float      gain = 1.f;
// envelope state, etc
};
```

Mapping strategy (deterministic)

- Primary lookup: linear scan for matching `clapVoiceId` where `stage != Off`.
- Allocation: first free `Off` slot.
- If none free: steal the oldest `Release` slot (or oldest `Active` if you must, but document it).

Steal rule must be deterministic:

- "Oldest" means smallest `age` or largest, but define one and stick to it.
- `age` increments globally per voice-start.

Pseudo:

```
Plain textfindSlot(voiceId):
  for i in 0..kMaxVoices-1:
    if slots[i].stage != Off && slots[i].clapVoiceId == voiceId: return i
  return -1

allocSlot(voiceId):
  for i in 0..kMaxVoices-1:
    if slots[i].stage == Off: init and return i
  // no Off slots:
  pick oldest Release slot (deterministic tie-break: lowest index)
  re-init and return i
```

Tie-breaker always lowest index to avoid randomness.

---

3) Sample-accurate modulation

Goal

Apply modulation at the exact sample frame offset within a block, per voice, without allocations and without per-sample event scanning overhead exploding.

Contract

1. Modulation events include frameOffset within block.
2. Events must be applied in non-decreasing frameOffset order.
3. A modulation event modifies **voice-local modulation accumulators**, not global base params.
4. Effective value = base + accumulated modulation (+ optional clamp).
5. Same event stream → identical output (determinism).

Practical algorithm: “frame cursor”

Process audio in segments between modulation frames.

Data structures:

- A fixed-capacity event queue per block, filled during event parsing.
- Each event: {frame, voiceSlot, paramId, delta}

```
C++structModEvent
{
uint16_t frame;      // 0..blockSize-1
uint8_t  slot;       // 0..kMaxVoices-1
uint16_t paramId;    // your internal param id
float    delta;      // modulation amount (or absolute, your choice)
};
```

Queue constraints:

- Capacity bounded, e.g. kMaxModEvents = 2048.
- Overflow behavior deterministic: drop oldest or newest, but document it and test it.

Processing:

```
Plain textsort events by frame (stable).  // ideally host already ordered; if not, do a fixed-cost stable bucket pass by frame.
cursor = 0
for each event in order:
  render [cursor, event.frame) with current effective params
  apply event delta to that slot's mod accumulator
  cursor = event.frame
render [cursor, blockSize) remaining
```

Sorting without heap

If you need ordering but want no sort cost:

- Use bucket bins by frame (0..blockSize-1) with fixed arrays of head indices (a stable linked list in arrays).
  That gives deterministic ordering without allocations.

---

4) Explicit expression routing

Goal

Turn CLAP "expression sources" (note expression, poly mod, MIDI 2.0-ish controllers) into **named spatial targets** with an explicit matrix. No hidden magic.

Contract

1. Routing is a mapping from **source → target** with:
    - depth (amount)
    - curve
    - optional smoothing
    - optional clamp
2. Routing happens at the adapter boundary:
    - source events become modulation events targeting internal paramIds
3. Routing is deterministic and serializable (so presets and QA agree).
4. Targets are clearly defined parameters in LocusQ.

Define targets (LocusQ spatial core)

Start minimal:

- X
- Y
- Gain
- Rotation (optional)
- Depth/Distance (optional)

Define sources (CLAP side)

Sources you might see:

- Poly modulation by parameter id
- Note expression types (pressure, timbre, brightness, pitch bend)

Routing table

Fixed-size table is easiest to keep RT safe:

```
C++enumclassExprSource : uint16_t
{
PolyParam0,
PolyParam1,
NotePressure,
NoteTimbre,
NoteBrightness,
PitchBend,
// ...
};

enumclassSpatialTarget : uint16_t
{
X,
Y,
Gain,
Rotation,
Distance
};
```

```
structRoute
{
ExprSource    src;
SpatialTarget dst;
float         amount;     // scale
float         offset;     // bias if needed
uint8_t       curve;      // enum for linear, exp, etc
uint8_t       smoothingMs;// optional
};
```

Your adapter translates SpatialTarget → internal paramId used by modulation processing.

Routing application

When an expression arrives:

1. Resolve voiceId → slot.
2. Find applicable routes for that source.
3. Compute delta (or absolute) from expression value:
    - mapped = curve(value) * amount + offset
4. Emit ModEvent { frameOffset, slot, paramId(dst), mapped }

This makes the mapping explicit and testable.

What this looks like in LocusQ docs

Create a short internal contract doc (or ADR) with sections:

- Capability negotiation and runtime modes
- Voice mapping invariants (slot allocation, reuse policy)
- Modulation semantics (base vs modulated, absolute vs delta, ordering)
- Expression routing matrix (sources, targets, curves)

This becomes the “DSP/UI/Host contract” for CLAP.

Non-negotiables for QA

To keep your harness happy:

- Every fallback mode must have a golden test.
- Voice allocation must be reproducible under voice storms.
- Modulation ordering must be deterministic even if host event order is weird.
- Overflow policies must be defined and covered by tests.

If you want the next step, I'll write this as an **ADR** + **a DSP_UI_CONTRACT_CLAP.md** style doc with normative language ("MUST/SHOULD/MAY"), plus a minimal C++ header (LocusQClapContract.h) that encodes the structs and enums so the implementation can't drift.

Locus Q Clap Contract + Adr

Copy

Edit

Download