Phonon: Next Phase Implementation Plan

Overview

This plan covers the next major development phase for Phonon, focusing on: 1. Fundsp integration strategy (individual nodes, not generic wrapper) 2. Sidechain compression and advanced routing 3. Wet/dry effects architecture 4. Cross-modulation FM synthesis 5. Legacy code deprecation 6. Interactive tab completion enhancement

Timeline: 4-6 weeks of focused development **Priority Order**: Numbered 1-6 below

---

Phase 1: Sidechain & Advanced Routing (Priority 1)

Deliverables

1. **SidechainCompressorNode** - Duck one signal based on another
2. **SidechainNoiseGateNode** - Gate one signal based on another trigger
3. **Tests** - 8-10 tests per node (standard coverage)

Implementation Details

SidechainCompressorNode:

```rust
pub struct SidechainCompressorNode {
    main_input: NodeId,        // Audio to compress
    sidechain_input: NodeId,   // Audio controlling compression
    threshold: NodeId,
    ratio: NodeId,
    attack: NodeId,
    release: NodeId,
    state: CompressorState,
}
```

Use Cases: - EDM ducking (kick ducks bass) - Podcast voice-over music ducking - Mix clarity (prevent frequency masking) - Creative pumping effects

Files to Create: - `src/nodes/sidechain_compressor.rs` - `src/nodes/sidechain_noise_gate.rs` - `tests/test_sidechain_compressor.rs` - `tests/test_sidechain_noise_gate.rs`

Estimate: 8-10 hours total - SidechainCompressorNode: 5 hours (similar to ExpanderNode complexity) - SidechainNoiseGateNode: 3 hours - Testing/docs: 2 hours

---

Phase 2: Wet/Dry Effects Architecture (Priority 2)

Problem Statement

Many effects currently have hardcoded wet/dry mixing or no mixing at all. Need consistent pattern across all effects.

Analysis Required

Audit all effects nodes to determine wet/dry support:

```
# Check each effect in src/nodes/
- reverb.rs: Has mix parameter
- delay.rs: Has mix parameter
- chorus.rs: Has mix parameter
- distortion.rs: Has mix parameter
- flanger.rs: Check implementation
- phaser.rs: Check implementation
- vocoder.rs: No wet/dry - always 100% wet
- convolution.rs: Check implementation
- ... (audit all effects)
```

Deliverables

1. **Wet/Dry Pattern Decision**:
   - Option A: Add `mix` parameter to every effect (consistent with current pattern)
   - Option B: Create `WetDryNode` wrapper (more flexible, composable)
   - **Recommendation**: Option A (consistency, less node overhead)
2. **Implementation**:
   - Add `mix: NodeId` parameter to effects lacking it
   - Standardize mixing formula: `output = dry * (1.0 - mix) + wet * mix`
   - Update tests for all modified nodes
3. **Room Effects on Wet Signal**:

   ```
   // Enable processing wet signal through additional effects
   ~reverb_wet: signal # reverb 2.0 0.8 0.6  // mix=0.6 (60% wet)
   ~shaped_reverb: ~reverb_wet # lpf 8000 0.7  // Filter only the wet reverb
   ```

   Key Insight: Already possible! Just route reverb output through filters. The `mix` parameter controls how much wet signal comes through.

Files to Modify: - Audit: `src/nodes/*.rs` (all effects) - Update: Effects lacking `mix` parameter - Tests: Update affected test files

Estimate: 6-8 hours - Audit: 1 hour - Implementation: 3-4 hours (varies by effect count) - Testing: 2-3 hours

---

Phase 3: Cross-Modulation FM Synthesis (Priority 3)

Deliverable

FMCrossModNode - Use external audio as FM modulator

```
pub struct FMCrossModNode {
    /// Audio signal to modulate (carrier)
    carrier: NodeId,
    /// Audio signal doing the modulation (modulator)
    modulator: NodeId,
    /// Modulation depth/index
    mod_depth: NodeId,
}
```

Use Cases

```
-- Drums modulating bass
~kick: s "bd*4"
~bass: saw 55
~modulated_bass: fmcrossmod ~bass ~kick 0.5

-- LFO modulating synth with audio-rate modulation
~lfo: sine 8
~synth: saw 220
~vibrato: fmcrossmod ~synth ~lfo 10.0
```

Implementation Notes

- Uses audio signal to vary phase/frequency of carrier
- Different from FMOscillatorNode (which uses internal modulator oscillator)
- Enables complex timbral effects impossible with traditional FM

Files to Create: - `src/nodes/fm_crossmod.rs` - `tests/test_fm_crossmod.rs`

Estimate: 4-5 hours - Implementation: 2-3 hours - Testing: 2 hours

---

Phase 4: Fundsp Individual Node Integration (Priority 4)

Strategy: Individual Nodes, Not Generic Wrapper

Decision: Port specific fundsp units as dedicated nodes (OrganNode, etc.), NOT as FundspUnitNode generic wrapper.

Rationale: - Type safety (compiler catches errors) - Better documentation per node - Cleaner API - Consistent with existing architecture - Easier testing

Fundsp Units Inventory

Check for name collisions with existing nodes:

Fundsp Unit	Existing Node?	Action
organ_hz	□ No	**Port** as OrganNode
moog_hz	□ Yes (MoogLadderNode)	Skip - already have it
saw_hz, square_hz, triangle_hz	□ Yes (VCONode, PolyBLEPOscNode)	Skip - redundant
noise, pink_noise	□ Yes (NoiseNode, PinkNoiseNode)	Skip - redundant
chorus	□ Yes (ChorusNode)	Compare quality, keep better
reverb_stereo	⚠ Partial (ReverbNode is mono)	**Consider** for stereo
phaser	□ Yes (PhaserNode)	Compare quality
pulse (PWM)	□ Yes (PulseNode)	Skip - redundant
dlowpass_hz	□ No	**Consider** (nonlinear lowpass)
softsaw_hz	□ No	**Consider** (softer harmonics)

Deliverables

Recommended to port (unique value): 1. **OrganNode** - Additive synthesis organ (unique sound) 2. **StereoReverbNode** - True stereo reverb (current is mono) 3. **NonlinearLowpassNode** (dlowpass_hz) - Nonlinear filter character

Implementation Pattern:

```rust
// src/nodes/organ.rs
use fundsp::prelude::*;

struct OrganState {
    unit: Box<dyn AudioUnit>,  // Type-erased
    last_freq: f32,
}

impl OrganState {
    fn new(freq: f32, sample_rate: f64) -> Self {
        let mut unit = organ_hz(freq);
        unit.reset();
        unit.set_sample_rate(sample_rate);
        Self { unit: Box::new(unit), last_freq: freq }
    }

    fn tick(&mut self) -> f32 {
        self.unit.tick(&[]).0
    }
}

pub struct OrganNode {
    frequency: NodeId,
    state: OrganState,
}

impl AudioNode for OrganNode {
    fn process_block(&mut self, inputs: &[&[f32]], output: &mut [f32], sample_rate: f32, _context: &Pro
        let freq_buffer = inputs[0];

        for i in 0..output.len() {
            // Recreate unit if frequency changed significantly
            if (freq_buffer[i] - self.state.last_freq).abs() > 1.0 {
                self.state = OrganState::new(freq_buffer[i], sample_rate as f64);
            }

            // Sample-by-sample processing (fundsp is sample-rate)
            output[i] = self.state.tick();
        }
    }
}
```

Files to Create (per node): - `src/nodes/organ.rs` - `src/nodes/stereo_reverb.rs` - `src/nodes/nonlinear_lowpass.r` - `tests/test_organ.rs` - `tests/test_stereo_reverb.rs` - `tests/test_nonlinear_lowpass.rs`

Estimate: 12-15 hours total - OrganNode: 4 hours - StereoReverbNode: 5 hours (stereo handling) - NonlinearLowpassNode: 3 hours - Docs/examples: 1 hour each

---

Phase 5: Legacy Code Deprecation (Priority 5)

Current State

unified_graph.rs: 14,146 lines - the old sample-by-sample architecture

Status: Still in use! The system currently runs BOTH architectures: - Old: `SignalNode` enum in unified_graph.rs (sample-by-sample) - New: `AudioNode` trait in src/nodes/*.rs (buffer-based, 10-100x faster)

Analysis Required

Determine Usage:

```
# Check if old architecture is still called
grep -r "eval_signal\|SignalNode" src/main.rs src/compositional_compiler.rs
grep -r "UnifiedGraph::new\|add_node" src/main.rs
```

Two Scenarios:

Scenario A: Old Architecture Still Active **Action**: Defer deprecation until after buffer architecture is fully integrated into compiler **Reason**: Can't remove code that's still being used **Timeline**: Revisit after Phase 1-4 complete

Scenario B: Old Architecture Unused (Dead Code) **Action**: Remove immediately **Steps**: 1. Verify no callers: `grep -r "SignalNode" src/main.rs src/*compiler*.rs` 2. Check for tests depending on it: `grep -r "SignalNode" tests/` 3. Remove `src/unified_graph.rs` (14K lines!) 4. Remove related imports/dependencies 5. Run full test suite to confirm nothing broke

Deliverables

If deprecated: - Remove `src/unified_graph.rs` - Update `src/lib.rs` to remove exports - Clean up any dead imports - **Result**: -14,000 lines of code!

If deferred: - Document current usage - Create migration plan for remaining SignalNode users - Add deprecation warnings to old architecture

Estimate: 2-3 hours (if ready to remove)

Phase 6: Interactive Tab Completion Enhancement (Priority 6)

Current State

Completion system exists: - `src/modal_editor/completion/function_metadata.rs` (1,915 lines) - Already has function metadata, parameter info, defaults - **Problem**: Manually maintained, not auto-generated from source

Goal: Auto-Generated Completion from Source Code

Generate completion metadata directly from: 1. Node struct doc comments 2. Parameter names in `new()` constructors 3. Default values in implementation

Implementation Strategy

Step 1: Doc Comment Parser (Build-Time)

```
// build.rs or proc_macro

/// Parse doc comments from src/nodes/*.rs
/// Extract:
/// - Function name
/// - Description (first line of doc)
/// - Parameters (from `pub fn new(...)` signature)
/// - Defaults (from implementation or doc comments)

// Example extraction:
// From: src/nodes/limiter.rs
/// Limiter - brick-wall dynamics limiter
/// ...
pub fn new(input: NodeId, threshold: NodeId, ceiling: NodeId) -> Self

// Generates:
FunctionMetadata {
    name: "limiter",
    description: "Brick-wall dynamics limiter",
    params: vec![
        ParamMetadata { name: "input", type: "NodeId", optional: false, ... },
        ParamMetadata { name: "threshold", type: "Pattern", optional: false, ... },
        ParamMetadata { name: "ceiling", type: "Pattern", optional: false, ... },
    ],
}
```

Step 2: Parameter Completion on Tab

```
// When user types: lpf <TAB><TAB>
// Show: :cutoff Pattern :resonance Pattern=0.7

// When user types: limiter <TAB><TAB>
// Show: :input Signal :threshold Pattern :ceiling Pattern
```

Step 3: Template Insertion

```
// When user types: limiter <TAB> and selects template
// Insert: limiter :input ___ :threshold ___ :ceiling ___
// With cursor at first ___ placeholder
```

Deliverables

1. **Doc Comment Parser**:
    - `build.rs` script or proc_macro
    - Parses `src/nodes/*.rs` doc comments
    - Extracts parameters from `new()` signatures
    - Generates `function_metadata.rs` automatically
2. **Enhanced Completion UI**:
    - Parameter completion after function name
    - Show parameter types and defaults
    - Template insertion with placeholders

  - Context-aware suggestions
3. **Standardized Doc Comments**:
   - Update all `src/nodes/*.rs` with standard format:

```rust
/// NodeName - Short description
///
/// Longer description...
///
/// # Parameters
/// - `input`: Audio signal to process
/// - `threshold`: Threshold in dB (default: -20.0)
/// - `ratio`: Compression ratio (default: 4.0)
///
/// # Example
/// ```phonon
/// ~compressed: signal # compressor -20 4.0 0.01 0.1
/// ```
pub fn new(input: NodeId, threshold: NodeId, ratio: NodeId, ...) -> Self
```

4. **Completion Testing**:
   - Integration tests for completion system
   - Verify all nodes have metadata
   - Verify parameter completions work

Files to Create/Modify: - `build.rs` or `proc_macro` crate for parsing - `src/modal_editor/completion/codegen.rs` (generated file) - Update all `src/nodes/*.rs` with standardized docs - `src/modal_editor/completion/parameter.rs` (parameter completion logic)

Estimate: 15-20 hours - Doc comment parser: 6-8 hours - Parameter completion UI: 4-5 hours - Standardize node docs: 4-5 hours (many files) - Testing: 2 hours

---

Implementation Order & Timeline

Week 1-2: Core Features

- **Phase 1**: Sidechain nodes (8-10 hours)
- **Phase 2**: Wet/dry audit & fixes (6-8 hours)

Week 3: Advanced Synthesis

- **Phase 3**: FM cross-modulation (4-5 hours)
- **Phase 4 Start**: Begin fundsp individual nodes (6 hours)

Week 4-5: Fundsp & Cleanup

- **Phase 4 Continue**: Complete fundsp nodes (6-9 hours remaining)
- **Phase 5**: Legacy code deprecation (2-3 hours)

Week 6: Developer Experience

- **Phase 6**: Tab completion enhancement (15-20 hours)

Total Estimate: 55-68 hours (~1.5 months at 10 hours/week)

Success Criteria

Phase 1: Sidechain

☐ SidechainCompressorNode passes 10+ tests
☐ SidechainNoiseGateNode passes 8+ tests
☐ Example tracks demonstrate ducking effect

Phase 2: Wet/Dry

☐ All effects have consistent `mix` parameter
☐ Documentation shows wet signal routing patterns
☐ Tests verify mix parameter behavior

Phase 3: FM Cross-Mod

☐ FMCrossModNode passes 8+ tests
☐ Example shows drums modulating bass
☐ Audio analysis confirms FM artifacts

Phase 4: Fundsp

☐ OrganNode, StereoReverbNode, NonlinearLowpassNode implemented
☐ Each has 8-10 tests
☐ No name collisions with existing nodes
☐ Performance testing shows acceptable overhead

Phase 5: Legacy Deprecation

☐ Old architecture removed OR migration plan documented
☐ All tests pass after removal
☐ Code size reduced significantly

Phase 6: Tab Completion

☐ Completion metadata auto-generated from source
☐ Parameter completion works for all nodes
☐ Template insertion with placeholders
☐ All nodes have standardized doc comments

Risk Mitigation

Risk 1: Fundsp Sample-by-Sample Overhead

Mitigation: Performance testing during Phase 4. If overhead too high, skip fundsp integration.

Risk 2: Legacy Code Still in Use

Mitigation: Audit before Phase 5. If still needed, defer deprecation.

Risk 3: Tab Completion Parser Complexity

Mitigation: Start with manual metadata, gradually automate. Partial automation still valuable.

Risk 4: Wet/Dry Breaks Existing Patches

Mitigation: Add `mix` parameter with default=1.0 (100% wet) to maintain backward compatibility.

Next Steps

1. **Immediate**: Start Phase 1 (sidechain compression)
2. **After review**: Adjust priorities based on user needs
3. **Continuous**: Update this plan as implementation reveals new requirements

Questions to Resolve

1. **Phase 2**: Wet/dry via parameter or wrapper node?
2. **Phase 4**: Which fundsp units provide most value?
3. **Phase 5**: Is old architecture truly unused?
4. **Phase 6**: Build-time or runtime doc parsing?

This plan is a living document. Update as you progress through phases.