ChatGPT

KPN ABI Extension Draft & Agent Prompt

Background and Expectations

KPN runtime & scheduling philosophy

The AMP project uses a **Kahn Process Network (KPN)** runtime where audio and control nodes communicate by passing tokens through channels. The development guidance emphasises several principles:

- **Baseline KPN with optional IPLS** – the runtime should continue to support pure KPN semantics while allowing an *Iterative Partial-Latency Schedule* (IPLS) that groups nodes by stage and type to maximise vectorised execution [1]. If an IPLS schedule is supplied, the runtime must honour its ordering.
- **Stage-aware archetypes** – when an IPLS is present, nodes that share a stage/type are fused into Eigen-powered kernels for efficient vectorised processing [2]. Kernels must still handle singleton inputs gracefully.
- **Delay-aware readiness** – no node should run until all required inputs have traversed their declared delays, so the scheduler must track per-edge delays and wait on gates where necessary [3].
- **Unified token arena & delayed storage** – channel caches live in one contiguous allocation with FIFO semantics. Tokens carry traversal metadata so consumers only see them after expiry [4].
- **Contract requirements for nodes** – every node must implement `forward()` and reversible `backward()` routines, operate on batched `(batch, channel, frame)` payloads, report processing delay and thermodynamic heat, and support slew-rate limited integrators [5].
- **Simulation fidelity** – scheduling and buffering must not assume a limited frequency band; multiband processing and high-fidelity oscillators should be supported without Python fallbacks [6].

Current runtime gaps and recent fixes

An audit of the native runtime identified several gaps:

- The existing scheduler executes nodes strictly in topological order and does not respect delays or synchrony gates; channels cache only the latest pointer and lack FIFO storage [7].
- The C API exposes only `amp_run_node` for forward execution; there is no backward/adjoint interface [8].
- Nodes cannot declare processing delays or oversampling ratios, and the descriptor JSON omits these fields [9].
- Parameter overrides are not shape-checked, allowing silent mismatches [10].
- No runtime facility surfaces the `thermo.heat` side-channel [11].

The `kpn_upgrade_action_plan.md` tracks progress:

- **Completed** – `kpn_contract.md` provides an authoritative node contract summarising invocation semantics, `(batch, channel, frame)` layout, state lifetimes and thermal reporting. The runtime now validates parameter shape and includes a `test_thermo_param.cpp` test [12].
- **Pending** – the ABI extension proposal (phase 4) and its implementation (phase 5) remain open [13]. CI and performance instrumentation are also planned.

General ABI concepts

An **application binary interface (ABI)** is the low-level interface exposed by compiled software; it defines machine-level aspects such as data type layout, calling conventions and name mangling [14]. In contrast to APIs, which describe source-level interfaces, ABIs must remain stable across compiler versions and hardware targets to allow libraries and programs to interoperate [15]. When extending an ABI, designers should:

1. **Ensure backwards compatibility** so existing binaries continue to run without recompilation. New entry points should not alter the signature of existing functions.
2. **Explicitly document new structures and enumerations** (e.g., mode fields) and align them naturally on common platforms.
3. **Include sufficient metadata** in plan blobs or descriptors so runtimes know when to use the new ABI.
4. **Provide a migration strategy** with tests and fallbacks for legacy nodes.

Agent Prompt for Drafting an ABI Extension Proposal

Use this prompt to instruct an agent (or yourself) to produce `docs/abi_extension_proposal.md` for the AMP KPN runtime.

> **Prompt:** You are tasked with drafting an **ABI extension proposal** for the AMP KPN runtime that builds on the existing `amp_run_node` contract. Follow these guidelines:
>
> 1. **Survey existing documentation** – read `docs/kpn_development_guidance.md`, `docs/kpn_contract.md`, `docs/kpn_audit_findings.md`, and `docs/kpn_upgrade_action_plan.md` to understand the current scheduler, node requirements, dataflow standards, and missing features. Pay particular attention to the need for declared delay metadata, oversampling hints, reversible execution and thermal reporting [16] [17].
> 2. **Define a new entry point** – propose `amp_run_node_v2(node_descriptor, state, inputs, outputs, mode)` where `mode` is an enum (`AMP_FORWARD`, `AMP_BACKWARD`) indicating whether to perform forward or adjoint evaluation. The function should return an error code and update `state` as in the current ABI. Explain the calling convention and how additional context (e.g., thermal side-channel pointer) is passed.
> 3. **Specify declared delays and oversample ratios** – extend the descriptor JSON (or plan blob) to include per-node `declared_delay` (integer frame count) and `oversample_ratio` (power-of-two integer). For edges, include `delay_frames`

and optional `synchrony_gate` arrays. Describe how the runtime stores these fields in the contiguous token arena and how the scheduler uses them for readiness checks [18].

4. **Define modulation & thermal parameters** – reserve a canonical parameter name `thermo.heat` for passing accumulated heat and specify shape/units. Suggest adding optional side-channel structures (e.g., `amp_node_metrics_t`) that can be passed to `amp_run_node_v2` to retrieve measured delay and heat.
5. **Backward/adjoint semantics** – detail how `AMP_BACKWARD` mode traverses inputs/outputs, referencing dual channels, and how state reversal is handled. Indicate that default implementations may return `AMP_E_UNSUPPORTED` until nodes are upgraded.
6. **Migration strategy** – emphasise that existing binaries using `amp_run_node` will continue to work. Nodes can opt into the new ABI by implementing `amp_run_node_v2` and advertising their capabilities. Describe version negotiation (e.g., function pointer check or descriptor flag).
7. **Testing and validation** – outline a test plan: update unit tests to verify delay propagation, oversampling, and backward mode; add microbenchmarks for oversampled oscillators; ensure CI runs both `kpn_unit_test` and `test_thermo_param` through the native pathway [19].
8. **Document examples** – include example JSON snippets showing oversample and declared delay fields inside `params_json`, and illustrate how a node descriptor would declare `amp_run_node_v2`. Use the `(batch, channel, frame)` layout consistently [5].

Produce the proposal in clear, versioned Markdown, referencing the existing contract and emphasising the **no-Python-fallback** policy. Use bullet points and diagrams where appropriate.

Proposed ABI Extension (High-Level Summary)

If you opt to craft the proposal directly, the following elements should be included:

1. **Function signature**

```
typedef enum {
    AMP_FORWARD = 0,
    AMP_BACKWARD = 1
} amp_execution_mode_t;

int amp_run_node_v2(const amp_node_descriptor_t* desc,
                    amp_state_t** state,
                    const amp_buffer_t* inputs,
                    amp_buffer_t* outputs,
```

```
                    amp_execution_mode_t mode,
                    amp_node_metrics_t* metrics /* optional */);
```

- `desc` points to a descriptor containing declared delay and oversample metadata.
- `state` behaves as in `amp_run_node` (allocated on first call, released via `amp_release_state`).
- `inputs` / `outputs` are arrays of `(batch, channel, frame)` tensors.
- `mode` selects forward or backward evaluation; unknown modes return `AMP_E_INVALID_ARG`.
- `metrics` is optional and, when non-null, receives measured delay in frames and accumulated heat.

- **Descriptor extensions**

In the JSON schema for each node:

```
{
  "name": "GainNode",
  "oversample_ratio": 4,
  "declared_delay": 64,
  "params": {
    "gain": 0.5,
    "thermo.heat": 0.0
  },
  "supports_v2": true
}
```

- `oversample_ratio` hints to the planner that this node operates at 4× the base sample rate.
- `declared_delay` declares a 64-frame latency; the scheduler should not consume this node's output until 64 frames after invocation.
- `supports_v2` flags that `amp_run_node_v2` is available for this node; absent or `false` means `amp_run_node` should be used instead.

For edges, extend the plan blob to include `delay_frames` and optional `synchrony_gate_id` so the scheduler can check readiness before dispatching downstream nodes.

1. **Backward/adjoint rules**

2. In `AMP_BACKWARD` mode, `inputs` correspond to gradients of the node's outputs, and `outputs` hold gradients of the node's inputs. The `metrics` structure may accumulate time and thermal contributions for gradient propagation.

3. Nodes that do not implement backward logic should return `AMP_E_UNSUPPORTED`; the runtime will propagate zeros for their gradients.

4. **Migration and compatibility**

5. Existing binaries and nodes continue to use `amp_run_node`.
6. The runtime loader can detect `supports_v2` and dispatch to `amp_run_node_v2` when available.
7. New nodes implementing the extended ABI must still provide `amp_run_node` for backwards compatibility during rollout.
8. **Testing and CI**
9. Extend unit tests to verify that declared delays postpone consumption of outputs and that oversample ratios are honoured by the scheduler.
10. Add tests for `AMP_BACKWARD` that check gradient flow through reversible nodes.
11. Update CI workflows (`.github/workflows/native-kpn-ci.yml`) to run the new tests on multiple platforms, verifying that no Python fallback occurs and that the new API functions correctly [20].

Conclusion

The AMP project's design brief calls for a more expressive and discoverable KPN runtime capable of explicit delays, oversampling, and reversible execution. A carefully designed ABI extension centered around `amp_run_node_v2`, additional descriptor metadata, and a clear migration path will allow the team to honour these goals while preserving compatibility with existing binaries. The agent prompt above guides contributors through drafting the necessary specification using the project's documentation and general ABI design principles [14].

---

[1] [2] [3] [4] [5] [6] [16] [18] kpn_development_guidance.md
https://github.com/sangderenard/amp/blob/HEAD/docs/kpn_development_guidance.md

[7] kpn_runtime_action_plan.md
https://github.com/sangderenard/amp/blob/HEAD/docs/kpn_runtime_action_plan.md

[8] [9] [10] [11] [17] kpn_audit_findings.md
https://github.com/sangderenard/amp/blob/HEAD/docs/kpn_audit_findings.md

[12] [13] [19] [20] kpn_upgrade_action_plan.md
https://github.com/sangderenard/amp/blob/HEAD/docs/kpn_upgrade_action_plan.md

[14] [15] Application binary interface - Wikipedia
https://en.wikipedia.org/wiki/Application_binary_interface