RIFTLang Token Specification: Classical and Quantum Governance

1. Token Architecture Overview

I present the formal specification for the RIFTLang token triplet architecture. This fundamental structure powers our governance-first language design across both classical and quantum execution contexts.

Every token in RIFTLang follows the triplet model:

```
token = (token_type, token_value, token_memory)
```

Our architecture prioritizes memory alignment before type enforcement, reversing traditional language design principles to create a more flexible yet governable system.

2. Token Component Specifications

2.1 Token Memory Specification

Memory represents the foundation of our token architecture and must be declared before type or value assignment:

```rift
// Memory declaration must precede type and value assignment
align span<row> {
  direction: right -> left,
  bytes: 4096,  // Classical mode fixed alignment
  type: continuous,
  open: true  // Mutable until policy enforcement
}
```

2.2 Token Type Specification

Types define the semantic classification of tokens:

```rift
// Type declaration follows memory alignment
type INT = {
  bit_width: 32,
  signed: true,
  memory: aligned(4)
}

// Quantum type with superposition support
type QINT = {
  bit_width: 32,
  signed: true,
  memory: aligned(8),
  superposition: enabled
}
```

2.3 Token Value Specification

Values contain the actual data or symbolic representation:

```rift
// Classical value assignment
x := 42  // Inferred as INT

// Quantum value with potential states
y =: superpose(1, 2, 3)  // Must resolve via DAG or collapse()
```

3. Classical vs. Quantum Mode Specification

The following two-way table defines the enforceable policies and behaviors across execution modes:

Feature	Classical Mode	Quantum Mode
Memory Alignment	Fixed 4096-bit alignment	Dynamic 8-qubit alignment
Memory Declaration	`align span<fixed>`	`align span<superposed>`
Type Declaration	`type T = {...}`	`type QT = {..., superposition: enabled}`
Value Assignment	`:=` (immediate binding)	`=:` (deferred binding)
Resolution Mechanism	Immediate, type-checked	DAG traversal or explicit `collapse()`
Concurrency Model	Locked, emulated via context switching	Phase-locked, true parallel execution
Policy Enforcement	Immediate after assignment	Deferred until entropy threshold met
Operator Behavior	Deterministic, bitwise	Probabilistic, interference-based
Binding Syntax	`x := value`	`x =: entangled(value)`
Error Handling	Compile-time type errors	Runtime entropy resolution errors

4. Token Governance Directives

4.1 Classical Mode Governance

I define the following governance directives for classical mode execution:

```rift
// Classical mode governance
!govern classic {
  token_memory: {
    alignment: fixed(4096),
    access: [CREATE, READ, UPDATE, DELETE],
    phase: deterministic
  },

  token_type: {
    inference: static,
    checking: eager,
    casting: explicit
  },

  token_value: {
    binding: immediate,
    resolution: eager,
    validation: static
  },

  policy_enforcement: {
    timing: immediate,
    violation: error,
    recovery: none
  }
}
```

4.2 Quantum Mode Governance

For quantum mode, governance directives enable flexible, context-aware execution:

rift

```
// Quantum mode governance
!govern quantum {
  token_memory: {
    alignment: dynamic(8),
    access: [CREATE, READ, UPDATE, DELETE, SUPERPOSE, ENTANGLE],
    phase: probabilistic
  },

  token_type: {
    inference: dynamic,
    checking: lazy,
    casting: implicit
  },

  token_value: {
    binding: deferred,
    resolution: context_dependent,
    validation: entropy_threshold(0.25)  // Default, can be overridden
  },

  policy_enforcement: {
    timing: deferred,
    violation: warning,
    recovery: auto_collapse
  }
}
```

5. Token Lifecycle Semantics

5.1 Classical Token Lifecycle

rift

```
// Classical token lifecycle
token INT x := 42;  // Type, memory, value all bound immediately

// Operations enforce immediate type checking
x := x + 1;  // Type checked, memory validated

// Policy enforcement occurs at assignment
policy_enforce(x);  // Immediate validation
```

5.2 Quantum Token Lifecycle

```rift
// Quantum token lifecycle
token QINT y =: superpose(1, 2, 3);  // Type, memory entangled, value deferred

// Operations maintain entanglement until observation
y =: y + 1;  // Type and memory remain in superposition

// Policy enforcement occurs at observation or explicit collapse
observe(y);  // Triggers policy enforcement
y.collapse();  // Explicit collapse forcing policy check
```

6. Memory Governance Policy

Memory governance in both modes is defined through policy functions:

```rift
// Classical memory policy
policy_fn on memory_space<T> {
  default_access: [READ, WRITE],
  reassert_lock: after every operation
}

// Quantum memory policy
policy_fn on q_memory_space<T> {
  default_access: [READ, WRITE, SUPERPOSE],
  reassert_lock: when entropy < 0.25 OR after span<clone> expires
}
```

7. Extension Mechanism

The token architecture is designed for extensibility through feature directives:

```rift
// Adding a new feature to the token architecture
!extend token_memory {
  feature: persistent_state,
  modes: [classic, quantum],
  default: disabled,
  governance: {
    policy_fn: require_explicit_opt_in,
    enforcement: immediate
  }
}
```

New token types can be created through the type extension mechanism:

```rift
// Creating a new token type
!extend token_type {
  name: TENSOR,
  parent: matrix<T>,
  properties: {
    dimensions: [2, 3, 4],
    element_type: float32
  },
  governance: {
    memory_alignment: 512,
    access_control: [READ, TENSOROP]
  }
}
```

8. Entanglement Model

The three-layer entanglement model is formalized as:

```rift
// Shadow type - Type metadata only
shadow_type ST := type_of(x);

// Shadow copy - Structure without values
shadow_copy SC := clone_structure(x);

// Real type - Complete type and value
real_type RT := x;
```

Entanglement operations are defined through explicit directives:

```rift
// Entangle two tokens
entangle(x, y) {
  mode: bidirectional,
  collapse: synchronized,
  entropy_threshold: 0.1
}

// Create superposition
superpose(x, [1, 2, 3]) {
  weights: [0.2, 0.5, 0.3],
  collapse: on_observation
}
```

9. Implementation Requirements

I define the following requirements for compliant implementations:

1. Token memory must be aligned before type or value assignment
2. Policy enforcement must follow the mode-specific directives
3. Entropy threshold validation must be implemented for quantum mode
4. Extension mechanisms must preserve backward compatibility
5. Classical and quantum modes must be switchable at compile time
6. Token lifecycle events must trigger appropriate policy checks
7. Entanglement models must support all three layers

This specification serves as the foundation for implementing the RIFTLang token architecture across both execution modes while ensuring strong governance through policy enforcement.