# The Sibuna Discussion Process and Engineering Standards

STATE
published
STATUS
Published
TYPE
Process Memo
AUTHORS
Sibuna Contributors <team@sibuna.local>
CREATED
2026-09-07
UPDATED
2026-10-15
SUMMARY
Process document establishing RFC/RFD design lifecycle, TigerStyle engineering standards, and Elm-style error reporting

DISCUSSION
No external discussion URL assigned

LABELS
documentation, process, standards

**Record status**

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

## Contents

1. Revision note ..... 1
2. Abstract ..... 2
3. Introduction ..... 2
4. The SID Lifecycle ..... 2
5. Tooling and Numbering Workflow ..... 2
6. Engineering Principles and Code Standards ..... 3
7. Elm-Style Diagnostic and Error Reporting ..... 4
8. Compilation Targets and Verification ..... 4
9. Conclusion ..... 4
10. References ..... 4

## 1. Revision note

2026-10-15: Ported from the Typst original in the Zig repository. The record describes the Zig implementation as it was decided; SID 0010 maps its packages to the Rust crates of the port. The catalog is now managed with `guidedog dsc` (`new`, `promote`, `state`, `list`, `check`, `build`) in reStructuredText, in place of the Typst records, the `zig build sid-*` steps and `registry.typ` described below. The structural limits now apply to Rust code and are checked by `cargo xtask check`: a function has at most 70 logical lines, a file at most 1,408 lines, and a line at most 99 characters.

## 2. Abstract

Process document establishing RFC/RFD design lifecycle, TigerStyle engineering standards, and Elm-style error reporting.

The `sibuna` monorepo requires a durable decision-record process for architectural, cryptographic, protocol, security, and operational changes across the high-performance Web AI Firewall and anti-crawler daemon. The project adopts Sibuna Discussions, or SID, as RFC/RFD-style Typst documents that support structured technical reasoning, long-lived references, high-quality PDF archival output, and an HTML discussion website.

This memo defines the lifecycle of an SID, the placeholder numbering workflow, the enforced coding guidelines and structural limits, the TigerStyle engineering priorities, and the project's Elm-style diagnostic error reporting standard.

## 3. Introduction

A Sibuna Discussion is a Typst document stored in git under `docs/sid/records`. Each discussion is part design memo, part review artifact, and part historical record. The structure intentionally follows IETF RFCs and Oxide RFDs because those formats force explicit scope, status, rationale, trade-offs, and operational constraints instead of relying on ephemeral chat messages or implicit assumptions.

SID is used for topics such as:

- Reverse proxy architecture and zero-allocation socket I/O loops
- Cryptographic proof-of-work algorithms (HashX, Argon2id, SHA-256 SIMD)
- Multi-pattern bot detection, SIMD Aho-Corasick, and Radix CIDR routing
- Session token formats, Ed25519 signing, and cookie binding
- State caching, lockless decay maps, and distributed store integrations
- Performance benchmarking methodology and reproducible comparison gates

## 4. The SID Lifecycle

Every SID progresses through explicit lifecycle states:

1. **Prediscussion** (`prediscussion`): A draft with the placeholder number `XXXXX`. The problem statement, design overview, and open questions are being formulated.
2. **Discussion** (`discussion`): The draft has been promoted to a four-digit number (`NNNN`), registered in `registry.typ`, and opened for review by the working group.
3. **Accepted** (`accepted`): The working group agrees on the architectural direction and technical commitments.
4. **Committed** (`committed`): Implementation has completed and merged to main, satisfying the defined verification and benchmark gates.
5. **Published** (`published`): Normative design or process record that reflects active system guarantees.
6. **Abandoned** (`abandoned`): The proposal was superseded or rejected.

## 5. Tooling and Numbering Workflow

To prevent merge conflicts and premature number squatting across concurrent branches, drafts use the placeholder prefix `XXXXX-<slug>.typ`.

### 5.1. Creating a Draft

To create a new draft:

```
zig build sid-new -- <slug>
```

This copies `docs/sid/template/rfc-template.typ` to `docs/sid/records/XXXXX-<slug>.typ`, stamps the creation date, and provides a target for local compilation.

### 5.2. Promoting a Draft

When a draft is ready for working group discussion:

```
zig build sid-promote -- <slug>
```

The tool:

1. Determines the next available four-digit sequence number `NNNN`.
2. Renames `XXXXX-<slug>.typ` to `NNNN-<slug>.typ`.
3. Sets state to `discussion` and status to `Open for Discussion`.
4. Updates `docs/sid/registry.typ` with the new entry metadata.
5. Updates `docs/sid/bundle.typ` for HTML and PDF generation.

### 5.3. Listing Discussions

```
zig build sid-list
```

Displays registered discussions, active placeholder drafts, and detects consistency warnings between the filesystem and `registry.typ`.

## 6. Engineering Principles and Code Standards

Sibuna inherits and strictly enforces the engineering priorities established across `insan.ai` and `paxos-zig`: **Safety** → **Performance** → **Developer Experience**, in that exact order.

### 6.1. Hard Structural Limits

All source code files (`*.zig`) across `build.zig`, `apps/`, `libs/`, and `tools/` must adhere to the following non-negotiable structural constraints, automated via `tools/check-style.awk` and verified on every `zig build fmt`:

1. **Function Length** ($\leq 70$ **Lines of Code**): Functions must not exceed 70 lines of code. Blank lines, comment-only lines, and docstrings are explicitly excluded from the line count calculation. Long functions must be decomposed into focused, cohesive helper functions placed immediately adjacent to their caller.
2. **Line Length** ($\leq 99$ **Characters for Code**): Lines in code files must not exceed 99 characters. Wrapping must prioritize readability and clear syntactic boundaries.
3. **Documentation Line-Length Exclusion:** The 99-character line length limit is *explicitly not applied* to `docs/` (Typst documents, diagrams, and markdown documentation). Prose, formatted tables, and mathematical formulas must not be artificially broken by code line limits.
4. **File Length** ($\leq 1408$ **Lines of Code**): Each source code file must not exceed 1408 lines of code (excluding comments and blank lines). When a module approaches this boundary, it must be split across logical sub-packages.
5. **Forbidden Tab Characters:** Tab characters (`\t`) are forbidden in all source code files; standard 4-space indentation is required.

### 6.2. Core TigerStyle Priorities

- **Zero-Allocation Hot Path:** The firewall's primary classification and challenge verification path must never allocate dynamic memory. Memory pools, arena buffers, and socket connection rings are allocated statically at startup or scoped to thread-local stack frames.
- **Explicit Control Flow:** Control flow must remain flat and shallow. Return early when preconditions fail to keep the happy path unnested. Avoid hidden side effects, callbacks, or complex meta-programming in performance-critical paths.
- **State Invariants Positively:** Use assertions (`std.debug.assert(...)`) to verify preconditions, postconditions, and loop invariants. State invariants positively in code comments before implementing their state transitions.
- **Handle Every Error Explicitly:** Never swallow, ignore, or discard errors. Errors must be handled at the boundary where they occur or propagated up with domain-specific diagnostic context.
- **Explain the "Why" in Comments:** Comments must not narrate syntax or restate what code does; they must explain why design invariants, memory bounds, or protocol ordering rules exist.

## 7. Elm-Style Diagnostic and Error Reporting

Sibuna adopts Elm-style error reporting (Compiler and Operator Errors for Humans) across all daemon subsystems, CLI commands, and HTTP client interactions. When an error occurs, the system must not emit an opaque numeric code or bare error name; it must provide a structured, operator-oriented explanation with actionable recovery directions.

### 7.1. The Three Elements of an Elm-Style Error

Every diagnostic consists of three mandatory components:

1. **A Distinct Visual Boundary:** A prominent uppercase title framed with ASCII hyphens, establishing an unambiguous header (e.g., `-- INVALID NONCE ----------`).
2. **A Plain-English Explanation:** A concise description explaining what failed within the state machine and why the current operation was rejected.
3. **An Actionable Hint / Alternative:** A line prefixed with `Hint:` providing concrete direction on what the operator or client can do instead to resolve the error.

### 7.2. Concrete Diagnostic Example

```
-- CHALLENGE EXPIRED -------------------------------------------------------

The issued proof-of-work challenge has passed its 30-minute validity window.

Hint: Request a new challenge from /challenge/make and re-run the solver worker.
```

### 7.3. Implementation Architecture

Elm-style error handling is implemented as a core architectural service in:

- `libs/core/src/diagnostic.zig`: Exposes `Diagnostic` struct and `diagnostic.write(writer, title, message, hint)` to render formatted error blocks to any output stream.
- `libs/core/src/errors.zig`: Exposes `explainError(err: anyerror) []const u8`, mapping domain errors across challenge, token, network, policy, and store subsystems into structured explanations.

### 7.4. Recoverable Errors vs Invariant Assertions

The system strictly distinguishes between two failure classes:

- **Recoverable Operating Errors:** Expected events in untrusted network environments (expired tokens, invalid nonces, unrecognized headers, full rate-limit buckets). These are returned via Zig error sets and accompanied by human-friendly Elm-style explanations and hints.
- **Invariant Violations:** Internal programming bugs or impossible states (corrupted ring buffer pointers, unreachable switch arms). These halt execution immediately via `std.debug.assert(...)` in safe builds.

## 8. Compilation Targets and Verification

`build.zig` drives Typst compilation and verification:

- `zig build sid`: Builds PDFs for all registered records into `docs/build/`.
- `zig build sid -Dshd=<number_or_slug>`: Compiles a single record.
- `zig build sid-index`: Compiles the registry index PDF.
- `zig build sid-site`: Generates the HTML bundle into `docs/build/sid-site/`.
- `zig build fmt`: Runs `zig fmt --check` and `tools/check-style.sh` to enforce all structural limits.
- `zig build test`: Runs all library unit tests, including diagnostic formatting and error explanation checks.

## 9. Conclusion

By combining version-controlled Typst design records (SID), rigorous structural limits, TigerStyle engineering discipline, and Elm-style human-friendly diagnostics, Sibuna guarantees long-term codebase maintainability, extreme operational reliability, and sub-microsecond anti-crawler defense.

## 10. References

- IETF Request for Comments (RFC) series, the model for the structure of these records.

- Oxide Computer Company Requests for Discussion (RFD), the second model for the structure of these records.
- TigerStyle, the engineering priorities inherited from `insan.ai` and `paxos-zig`.
- Elm-style error reporting (Compiler and Operator Errors for Humans), the model for Sibuna diagnostics.

### *Lifecycle event*

```
2026-10-15: prediscussion -> discussion. Ported from the Zig repository's
Typst record
```

### *Lifecycle event*

```
2026-10-15: discussion -> published. State carried over from the Zig
repository's registry
```