Epiphany

Text Projection

A companion to the Core Specification

Version 0.1.0 — The canonical s-expression form

Normative for the text form it defines

Contents

Chapter 1

About This Companion

The *Text Projection* is a companion to the Epiphany Core Specification. It fulfils the delegation the core specification makes in Chapter 8, Text Projection (`sec:format:textproj`), which declares that "the format admits a deterministic projection to a canonical s-expression text form" and that "the text projection is normative" — while leaving the form itself unwritten. The Binary Format companion likewise excludes it: "the canonical s-expression form — that is the *Text Projection* companion's".

This document supplies that form.

1.1 What This Document Covers

- The canonical text syntax: atoms, byte strings, text, and the layout that makes the projection deterministic.
- What is projected, and what is deliberately not.
- The projection and parse requirements, including the bidirectional round-trip the core specification demands.

It does *not* cover the binary encoding of anything — that is the Binary Format companion's — nor the semantics of any operation, which is the Operation Catalog's. The projection is a *re-presentation* of the canonical document, never a second definition of it. Wherever the two could disagree, the binary form is normative and the projection is wrong.

1.2 The Subject of the Projection

The projection's subject is the **canonical document**, not the file. Per the core specification's Chapter 8 Text Projection, the projection **MUST NOT** be required to preserve chunk offsets, compression choices, cache chunks (operation indexes, layout caches, integrity indexes), garbage bytes from prior commits, or superblock generation numbers, slot assignments, and CRCs.

Consequently two bundles that differ only in physical layout project to the same text, and a text re-serializes to *a* bundle rather than to *the* bundle it came from. That is the intent, not a limitation: the physical file is an encoding of the document, and the projection is of the document.

A bundle **MAY** cache its own projection in a `TextProjection` chunk named by `Manifest.text_projection_roo`
That chunk is a **non-canonical accelerator** (core specification Chapter 8, Schema Versioning):
a reader need not understand it, and a writer preserves it verbatim or discards it. A cached
projection that disagrees with the operations it claims to project is *stale*, not authoritative.

Chapter 2

The Canonical Text Form

2.1 Encoding and Character Set

Requirement

A text projection **MUST** be UTF-8, with no byte-order mark. Every text field it carries **MUST** be in Unicode NFC, matching the canonical-text rule of the core specification's Appendix D, Text and Unicode. A parser **MUST** reject non-NFC text rather than normalize it.

2.2 Atoms

An *atom* is a symbol, an integer, a byte string, or a text string.

Symbols are lowercase ASCII words, possibly hyphenated: `envelope`, `insert-event`, `strict-inverse`. They name constructors and enumeration cases. A symbol is never quoted.

Integers are written in base ten, with a leading `-` for negative values, no leading zeros, and no leading `+`. Zero is `0`, never `-0`.

Byte strings carry every identifier, hash, and opaque payload.

Requirement

A byte string **MUST** be written as `#x` followed by an even number of **lowercase** hexadecimal digits, one pair per byte, in the order the bytes appear in the canonical binary form. The empty byte string is `#x`. A parser **MUST** reject uppercase digits, an odd digit count, and any separator within the digits.

Rationale

One rule for every byte string. Hexadecimal has no alphabet variant and no padding to canonicalize, it is greppable, and a corrupted character is locally obvious. Identifiers are 16 or 32 bytes, so its expansion costs nothing where it is read; only an inlined snapshot pays. The core specification permits "base64 *or another canonical text form*"; a second

encoding would buy a quarter of the bytes of the one body nobody reads, at the price of pinning an alphabet, a padding rule, and a line-wrapping rule, and of choosing which encoding applies where. Ratified at 0.1.0.

Text strings are double-quoted. Inside a string, `\"` denotes a quotation mark, `\\` a backslash, `\n` a line feed, and `\t` a tab; no other escape exists.

Requirement

A text string **MUST** escape exactly the characters that require it: the quotation mark, the backslash, U+000A, and U+0009. Every other character **MUST** appear literally. A parser **MUST** reject an escape sequence outside this set, and **MUST** reject a literal character that the writer was required to escape.

Rationale

"Escape exactly" rather than "escape at least": the text is canonical, so two spellings of one string cannot both be valid. This is the same injectivity the binary form rests on (Binary Format, `req:binfmt:decode-vectors`), stated for text.

2.3 Layout

Requirement

A projection is a sequence of lines separated by a single U+000A, with a final U+000A and no other trailing whitespace. Each line is one complete s-expression. Tokens within a line are separated by exactly one space; there is no other whitespace, and no indentation. Each operation envelope **MUST** occupy exactly one line.

Rationale

The core specification's stated use case is that "merge conflicts surface at the operation-envelope level, which is the meaningful level for collaborative editing". One envelope per line makes a line-based three-way merge conflict *exactly* an envelope conflict — never a conflict inside an envelope, which could otherwise produce a syntactically valid operation that neither side wrote. It also disposes of indentation: there is no whitespace to canonicalize, so the "identical semantics project to identical text" requirement below has nothing to hide in.

The lines are long. Readability is a *tooling* concern, and a pretty-printer is free to reformat for display; what it must not do is write the reformatted text back and call it a projection. Ratified at 0.1.0.

Chapter 3

What Is Projected

3.1 Document Structure

A projection is, in order:

1. a `(text-projection <version>)` header line, naming the version of *this companion* the text conforms to;
2. a `(document #x<document-id>)` line, and a `(lineage #x<lineage-id>)` line if the manifest declares one;
3. zero or more `(profile ...)` lines, in canonical order;
4. zero or more `(extension ...)` lines, in canonical order;
5. at most one `(canonical-base ...)` line;
6. zero or more `(envelope ...)` lines, in canonical operation order (core specification Appendix D).

Every sequence is written in the normative order its binary counterpart uses. The projection introduces no ordering of its own.

3.2 Reduced State

Requirement

The projection **MUST** preserve canonical reduced state *by preserving the operations that determine it*. It **MUST NOT** carry a second, literal copy of the reduced state.

Rationale

Reduced state is a deterministic function of the operation set and the canonical base (core specification Chapter 6, Design Principles). A text that carried both would have two sources of truth for one fact, and nothing could stop them disagreeing — a projection with an internally contradictory document is worse than no projection. This reading is what the core specification's own round-trip clause already implies: text that "parses to identical canonical document semantics" must re-serialize to bundles with identical semantics, and reduced state is a function of semantics.

Read the core specification's "all canonical reduced state" as *determines*, not *contains*. Ratified at 0.1.0.

3.3 The Canonical Base Snapshot

Requirement

If the manifest declares a canonical base, the projection **MUST** carry a `(canonical-base ...)` line bearing the snapshot's identity, its causal frontier, its reduction-algorithm version, its profile, *and its payload inline* as a single byte-string atom.

Rationale

A canonical base exists precisely so that the operations before its frontier need not be retained. Where they have been pruned, the snapshot is *not* derivable from anything else in the document, and a projection that carried only a reference would be **lossy** — the text would no longer determine the document, which is the one thing it is for. The core specification permits the payload to be "encoded compactly or referenced externally"; inline and compact is the choice that keeps the text self-contained.

The snapshot diffs as one opaque atom. That is acceptable: merges happen among operations, and a base snapshot changes only when the document is compacted, at which point the whole line changes anyway. A later revision **MAY** project the snapshot structurally; doing so does not break the round trip, because the document it denotes is unchanged. Ratified at 0.1.0.

3.4 Extensions

An `(extension ...)` line carries the extension's identity, its required-or-optional flag, its edit barriers, and its preserved chunk roots. An extension payload is a byte string (Requirement 2.2); the projection never interprets it.

Chapter 4

Requirements

4.1 Canonicality

Requirement

Two bundles whose canonical document semantics are identical **MUST** project to **byte-identical** text. A projector **MUST NOT** have any freedom the document does not determine: no optional whitespace, no alternative spelling of an atom, no ordering choice.

4.2 Round Trip

Requirement

Parsing a projection and re-serializing it to binary **MUST** yield a bundle whose canonical document semantics are identical to the original's. The bundle's physical layout, chunking, and compression **MAY** differ.
Equivalently, and more usefully to an implementer: for every bundle B,

$$\text{semantics}(\text{parse}(\text{project}(B))) = \text{semantics}(B),$$

and for every valid projection T,

$$\text{project}(\text{serialize}(\text{parse}(T))) = T.$$

The second equation is the text's own injectivity: it is *stronger* than the first, and it is the one a conformance test can check with byte equality.

4.3 Strict Parsing

Requirement

A parser **MUST** reject any text that is not the canonical projection of the document it denotes. It **MUST NOT** normalize: not whitespace, not letter case in a byte string, not an escape sequence, not an out-of-order sequence, not a duplicate in a set-typed field. Accepting non-canonical text and normalizing it *is* accepting it, and does not satisfy this requirement.

Rationale

This is the same discipline the binary decoders carry, and it exists for the same reason: a lenient parser makes two texts denote one document, and the projection's contract is that a text *determines* its document. The Binary Format companion learned this concretely — a whole-value re-encode guard catches the fields a decoder normalizes and is blind to order-preserving sequences, and a guard on an outer value can *mask* a lenient inner codec rather than fix it (Binary Format, The Decode Vector Corpus). A text parser inherits both hazards, and the cheapest total defence is the same one: re-project the parsed document and compare, *and* check per-site the orders that re-projection would restore.

4.4 Conformance

Requirement

An implementation claiming Text Projection conformance **MUST** implement both directions. A projector alone does not conform: the round trip (Requirement 4.2) is the requirement, and half of it is not checkable.

Chapter 5

Grammar

```
projection   ::= header document lineage? profile* extension*
                 canonical-base? envelope*

header       ::= "(text-projection " version ")" LF
version      ::= integer "." integer "." integer

document     ::= "(document " bytes ")" LF
lineage      ::= "(lineage " bytes ")" LF

profile      ::= "(profile " symbol " " version " " constraints ")" LF
extension    ::= "(extension " bytes " " ("required"|"optional")
                 " (barriers " barrier* ") (chunks " bytes* "))" LF

canonical-base
             ::= "(canonical-base " bytes " (frontier " bytes ")"
                 " (reduction " integer ") (profile " symbol ")"
                 " (payload " bytes "))" LF

envelope     ::= "(envelope " bytes " (author " bytes ")"
                 " (stamp " integer " " integer " " bytes ")"
                 " (causal " vector " " dots ")"
                 " " transaction " " payload ")" LF

transaction  ::= "()" | "(transaction " bytes ")"
payload      ::= "(primitive " kind ")"
               | "(resolve-conflict " bytes " " action ")"
               | "(undo " bytes " " policy ")"
               | "(resolve-equivocation " bytes " " bytes ")"

vector       ::= "(" ("(" bytes " " integer ")")* ")"   ; replica, counter
dots         ::= "(" bytes* ")"                        ; operation ids

bytes        ::= "#x" hexdigit*        ; even count, lowercase
integer      ::= "-"? digit+           ; no leading zeros, no "-0"
symbol       ::= [a-z] [a-z0-9-]*
string       ::= '"' schar* '"'
```

What this grammar is, and is not. The atom productions (`bytes`, `integer`, `symbol`, `string`)

and the line shapes above are **normative**. The productions left unexpanded — `kind`, `action`, `policy`, `constraints`, `barrier` — are *derived* rather than invented here: there is exactly one per corresponding binary discriminant, named by the Operation Catalog's section name in lowercase hyphenated form (`insert-event`, `transpose-interval`, ...), with its fields in the payload schema's declaration order. A future revision **SHOULD** spell them out; until it does, the Operation Catalog and the Binary Format companion's wire table jointly determine them, and an implementation disagreeing with those disagrees with this document.

That is a real gap, stated rather than papered over. It is the difference between a design gate and a finished companion.

Chapter 6

A Worked Example

Non-normative. The byte strings below are illustrative. The conformance vectors that pin real bytes are a deliverable of the implementation, not of this gate; elisions are marked ….

A document of one operation — a transposition of two pitches up a perfect fifth — projects to four lines:

```
(text-projection 0.1.0)
(document #x0505050505050505050505050505050505)
(canonical-base #x1f8b... (frontier #x00) (reduction 1) (profile full) (payload #
    x0000...))
(envelope #x00000000000000070000000000000001 (author #
    x00000000000000000000000011223344) (stamp 42 7 #
    x00000000000000070000000000000001) (causal ((#x0000000000000001 3)) (#
    x00000000000000020000000000000009)) (transaction #
    x00000000000000070000000000000005) (primitive (transpose-interval (targets #
    x00000000000000070000000000000001000000000000000700000000000000002) (interval 4
     7))))
```

The envelope's targets are a *set*: strictly increasing, no duplicates (Operation Catalog, `req:opcat:transpose-int` Binary Format seq⇑). A parser **MUST** reject a duplicate rather than absorb it, exactly as the binary decoder does.

Chapter 7

Revision History

Date	Section	Change
July 9, 2026	All	0.1.0 — Initial companion. Supplies the canonical s-expression form the core specification's Chapter 8 Text Projection declares normative and leaves unwritten, and which the Binary Format companion excludes as "the *Text Projection* companion's". Ratified: reduced state is preserved by *determining* it, never by a second literal copy (`req:textproj:reduced-state-derived`); a canonical base snapshot is inlined as one opaque byte string, because a pruned document's base is derivable from nothing and a reference-only projection would be lossy (`req:textproj:base-snapshot-inline`); lowercase hex is the single byte-string encoding (`req:textproj:hex`); one envelope per line, so a line-based merge conflict is exactly an envelope conflict (`req:textproj:envelope-per-line`). Parsing is strict-canonical (`req:textproj:strict-parse`) — normalizing non-canonical text is accepting it — and conformance requires *both* directions (`req:textproj:conformance`). No implementation yet; this document is the design gate.