```
Title Page
===========
Book of Power -- Condensed Graviton Edition
Author: Julian Christian Sanders (lexluger)
Crate Under Review: `power_house`
Book Edition: **v0.1.47**
Crate Version Required: **v0.1.47**
All examples and golden test vectors correspond to this exact build; if your crate version
↪  differs, regenerate every artifact before trusting the results.
Typeface Cue: Eldritch Vector Mono (conceptual spiral monospaced design)
Fallback Typeface: Fira Mono or JetBrains Mono (use standard monospace if unavailable)
Repository Source: crate checkout `power_house`
This User Guide Lives Inside the Crate: `docs/book_of_power.md`

Table of Contents
==================
Chapter I -- Anchor Echo Engine Command Doctrine
Chapter II -- Foundational Field Algebra Procedures
Chapter III -- Hyperplane Cartography Briefing
Chapter IV -- Transcript Metallurgy Protocols
Chapter V -- Ledger Genesis Mechanics Checklist
Chapter VI -- Deterministic Randomness Discipline Orders
Chapter VII -- Consensus Theater Operations
Chapter VIII -- Closing Benediction and Compliance Oath


Chapter I -- Anchor Echo Engine Command Doctrine
==================================================================================
I remain your irritable cosmic supervisor; this page is the manual for the 256-bit Anchor Echo
↪  Engine.
Memorize the genesis digest `139f1985df5b36dae23fa509fb53a006ba58e28e6dbb41d6d71cc1e91a82d84a`.
Memorize the dense polynomial digest
↪  `ded75c45b3b7eedd37041aae79713d7382e000eb4d83fab5f6aca6ca4d276e8c`.
Memorize the hash anchor proof digest
↪  `c72413466b2f76f1471f2e7160dadcbf912a4f8bc80ef1f2ffdb54ecb2bb2114`.
Memorize the anchor-fold digest emitted by `hash_pipeline`:
↪  `98807230712cd2b09c17df617b1f951787815b29c7037dbe9fcab2af490d196b`.
Write them on the wall; anyone who shrugs at 64 hex bytes should be reassigned to kitchen duty.
`transcript_digest` now feeds every `u64` transcript value into a BLAKE2b-256 tagged hash
↪  stream tagged with `JROC_TRANSCRIPT`.
No XOR tricks, no decimal fallbacks--pure hex, 32 bytes, immutable.
Ledger files still appear as `ledger_0000.txt`, `ledger_0001.txt`, etc., but every `hash: `
↪  line is now lowercase hex.
Run `cargo run --example hash_pipeline` weekly; the output must include the fold digest above
↪  and the reduced field value `64`.
The program stages two ledgers under `/tmp/power_house_anchor_a` and
↪  `/tmp/power_house_anchor_b`.
Note: Windows or hardened hosts lacking `/tmp` must set `POWER_HOUSE_TMP=/path/to/workdir`;
↪  never assume a Unix tmpfs is writable in prod.
Open `ledger_0000.txt`; the hash must match `ded75c45...6e8c`.
Open `ledger_0001.txt`; the hash must match `c7241346...2114`.
If either hash deviates, the log is corrupt or you miscopied the hex--both offences carry
↪  penalties.
`julian node anchor /tmp/power_house_anchor_a` should print three lines headed by `JROC-NET`.
Verify that the genesis line prints the digest from step 02 without error.
```

The fold digest from step 05 appears in terminal output; immediately copy it into `fold_digest.txt` beside your ledger before you proceed so auditors never depend on scrollback.
When exporting anchors, append a comment `# fold_digest: <hex>` or store it in `anchor_meta.json`; the quorum hinge must live with the artefacts you check in.
`LedgerAnchor::anchor()` automatically prepends the JULIAN genesis entry with the new digest.
Domain separation summary: `JROC_TRANSCRIPT` for individual records, `JROC_ANCHOR` for ledger folds, `JROC_CHALLENGE` for Fiat-Shamir challenge derivation.
Do not mix domains--if you re-tag transcripts with the anchor label, you will deserve the audit citation.
Hash framing specification (binary, tagged stream):
```
transcript_bytes = concat(u64::to_be_bytes(challenge_i) for each entry in transcript)
round_sum_bytes  = concat(u64::to_be_bytes(sum_i) for each entry in round_sums)
hasher = BLAKE2b-256()
hasher.update(b"JROC_TRANSCRIPT")
hasher.update(len(transcript_bytes) as u64_be)
hasher.update(transcript_bytes)
hasher.update(len(round_sum_bytes) as u64_be)
hasher.update(round_sum_bytes)
hasher.update(final_value.to_be_bytes())
digest = hasher.finalize()
```
- `statement:` text, comment metadata, and the `hash:` line never participate in the digest; only the numeric sections above are hashed, in that order.
- All transcript numbers are encoded as u64 big-endian before hashing.
- No personalization/salt is used beyond the explicit domain tag.
- ASCII hex with spaces or carets is cosmetic; the canonical digest is 32 raw bytes rendered as contiguous lowercase `[0-9a-f]` characters.

Encoding commandment (commit to memory):
```
Transcript text  : decimal ASCII tokens (e.g., `round_sums: 209 235`).
Hash inputs      : each integer serialized as u64 big-endian bytes.
Hex digests      : 64 lowercase `[0-9a-f]` chars, no spaces.
Line endings     : LF only; tabs forbidden.
```
Hash the big-endian bytes, never the ASCII digits; auditors treat deviations as tampering.
`ProofLedger` persists transcripts exactly once; any extra whitespace or comment must stay outside the recorded lines.
The CLI renders the digests via `transcript_digest_to_hex`; keep that function untouched.
To test deterministic recomputation, delete one byte from a log and rerun `verify_logs`; expect a digest mismatch in red text.
The aggregated digest reduces to field element `64`. Say it. Write it. Remember it.
Note: Digest-to-field procedure: interpret the first eight bytes of the 32-byte digest as a big-endian u64, then compute `value mod p` (e.g., `0x98807230712cd2b0 -> 0x98807230712CD2B0 -> 10988908643166769840 -> 64 (mod 257)`).
When the reduction changes, the field or transcript changed--file an incident report.
`simple_prng` is dead; the challenge stream is now BLAKE2b-256 seeded by the transcript plus domain tag.
Never allow anyone to talk wistfully about linear-congruential generators again.
Note: Bias note: current derivation uses `next_u64() % p`; keep `p` close to 2^64 (e.g., 64-bit primes) or switch to the documented rejection sampler in Chapter VI when you extend the code.
The `scripts/smoke_net.sh` ritual depends on stable keys; if the metrics server refuses to bind, document the environment block.
Finality relies on unique public keys; the network now laughs at duplicate voters.

~~~~
When reconciling offline, use placeholder identities like `LOCAL_OFFLINE` and `PEER_FILE`, but
↪  never reuse placeholders for different peers in the same quorum call.
Keep a laminated cheat sheet with the three transcript digests and the fold digest.
Add a second sheet listing the domain tags; auditors adore label discipline.
Print the digests with caret markers every four bytes: `139f_1985_df5b_36da_...`.
```
HEX SIGIL :: ANCHOR CORE
  GENESIS     139f 1985 df5b 36da e23f a509 fb53 a006 ba58 e28e 6dbb 41d6 d71c c1e9 1a82 d84a
  DENSE POLY  ded7 5c45 b3b7 eedd 3704 1aae 7971 3d73 82e0 00eb 4d83 fab5 f6ac a6ca 4d27 6e8c
  HASH ANCHOR c724 1346 6b2f 76f1 471f 2e71 60da dcbf 912a 4f8b c80e f1f2 ffdb 54ec b2bb 2114
  FOLD DIGEST 9880 7230 712c d2b0 9c17 df61 7b1f 9517 8781 5b29 c703 7dbe 9fca b2af 490d 196b
  FIELD REDUCE -> u64_be(first 8 bytes) mod 257 = 64
```
The ledger logs must remain ASCII; the hex lives on one line with no prefixes.
If you must annotate, prefix with `#` outside the transcript block.
`hash_pipeline` reduces to the canonical demo; treat its output as the lab reference.
Use `power_house::transcript_digest_to_hex` in your scripts; do not reinvent hex formatting.
If someone doubts determinism, rerun the example and shove the matching hashes under their
↪  nose.
When a cadet forgets a digit, force them to rewrite the digest 32 times--one per byte.
Disaster recovery scenario: power outage; print the digests from this book, run manual
↪  comparisons, reestablish finality.
Regulatory drill: produce log file, book excerpt, and CLI output; they must match
↪  byte-for-byte.
Museum display idea: light panel showing the genesis digest scrolling endlessly; educational,
↪  intimidating.
The anchor fold digest is the workshop handshake. Recite it at the start of every session.
Always verify `hash_pipeline` after upgrading Rust or dependencies; compilers surprise the
↪  lazy.
Keep the book version synchronized with `Cargo.toml`; current edition references `power_house
↪  0.1.47`. Continuous integration asserts that these strings match the crates manifest
↪  before any release ships.
If the crate version bumps, rerun `hash_pipeline`, update the values, and amend every
↪  compliance log.
Record the output path `/tmp/power_house_anchor_a` in your field log; easier for midnight
↪  audits.
Do not compress the `/tmp` logs before verifying them; compression hides tampering.
On offline machines, copy the example output into air-gapped storage, then verify with `julian
↪  node anchor`.
If the fold digest ever changes unexpectedly, halt deployments and investigate.
Store the printed digests in fireproof cabinets; yes, we still do that.
Confirm that `reconcile_anchors_with_quorum` now requires distinct keys by running unit test
↪  `test_reconcile_rejects_duplicate_keys`.
If that test fails, fix it before touching production.
Teach cadets that every hex pair represents eight bits of inevitability; there is no shortcut.
Binary toggles no longer amuse me, but you may use them to dramatize a single byte flip.
When presenting to executives, describe this chapter as "hexadecimal finality discipline."
When presenting to mathematicians, describe it as "BLAKE2b commitments over deterministic
↪  transcripts."
When presenting to auditors, describe it as "evidence that nothing is hidden."
The genesis digest anchors the entire JULIAN network; treat it as sacred text.
If someone requests the old 64-bit values, hand them a shredder.
Update the compliance wiki with screenshots of `hash_pipeline` output; redacting nothing.
Keep a QR code linking to this manual near every boot node console.
Logbooks must note the UTC timestamp when the digests were last verified.
When rewriting this manual, never shorten the digests; printing only the prefix is grounds for
↪  termination.
~~~~

Append the fold digest to any offsite backup manifest.
The anchor echo ritual is human-first; no automation may replace your eyeballs.
Maintain a rotation schedule for verification duty so every engineer memorizes the hex.
If an engineer cannot recall the first eight characters of the genesis digest, revoke their deploy privileges.
Celebrate new hires by making them transcribe the dense proof digest by hand.
This chapter is the onboarding gauntlet: memorize, verify, sign.
The log directory `./logs/boot1-ledger` must be backed up with the manual.
When shipping new firmware, include a printout of the three digests for the QA binder.
Add the fold digest to your monitoring dashboards as a constant string; alarms should fire if it ever mutates.
For interactive drills, invert one byte in the log and observe how the digest transforms; document the delta.
Re-run `hash_pipeline` after any change to transcript formatting; whitespace is deadly.
When the ledger evolves, update the book first, THEN announce the change.
Keep the aggregated digest visible on the boot node status page; stake your bragging rights on it.
If you hear "why not shorter digests," answer with threat of expulsion.
Always store transcripts and digests together; context is armor.
Replicate this manual in triplicate: on paper, in git, and in cold storage.
Tattoo the domain tags on your forearm if that helps.
Run `cargo test --features net` after every patch; the tests confirm our identity counting and digest logic.
If a colleague tries to skip the tests, this book authorizes you to snatch their keyboard.
The aggregated digest converts to field element 64; include that value in any whiteboard explanation.
Draw the folding pipeline as: transcripts -> BLAKE2b digest -> anchor fold -> quorum.
Each step must be reproducible from logs plus this manual--no hidden state.
Record the BLAKE2b command used by external auditors if they verify independently.
When this book says memorize, you memorize; complacency breeds forks.
The anchor echo engine is still the handshake ritual--now with heavier hex.
Sign the compliance sheet confirming you verified all four digests (three transcripts plus the fold) before leaving the room.
File the signed sheet next to the ledger backups.
Only after these steps may you advance to Chapter II.
You are expected to re-teach this chapter whenever onboarding new team members.
The combination of deterministic transcripts and simple arithmetic is the ultimate trust anchor.
Finish this chapter by writing `ANCHOR!!` in your own handwriting across the margin as proof you completed the ritual.

Chapter II -- Foundational Field Algebra Procedures

Punctual cadets start with fields; everything else is decoration.
The crate defines `Field::new(p)` where `p` must be prime.
Choose your modulus deliberately; 97, 101, 65537 are respectable examples.
`FieldElement` wraps `u64` and enforces modular operations without carrying external dependencies.
Addition and subtraction use wrapping arithmetic followed by conditional reduction.
Multiplication relies on 128-bit intermediates to avoid overflow before reduction.
Exponentiation is implemented via square-and-multiply, ensuring O(log exponent).
Inversion triggers the extended Euclidean algorithm; zero input raises panic.
The panic is intentional; the crate refuses undefined algebra.
Example: In F101, inverse of 37 equals 11 because `37*11 = 407` and `407 mod 101 = 1`.
This result is verified by the deterministic tests under `tests::field_inverse`.
Another example: `FieldElement::new(57).pow(100)` equals 1 due to Fermat's little theorem.

The crate ensures these results remain reproducible regardless of platform.
Sum-check routines depend on field operations to remain exact while reducing dimensions.
Without precise algebra, transcripts would diverge, and ledger anchors would reveal contradictions.
`GeneralSumClaim::prove` consults the field for addition when computing round sums.
`GeneralSumClaim::verify` cross-checks each coefficient with addition and multiplication in the same field.
Keep prime tables near your desk; random moduli are forbidden.
When switching modulus in experiments, regenerate transcripts to keep digests consistent.
The crate offers no built-in primality check beyond curated primes; do not feed it junk.
Document every modulus choice in deployment logs.
Field addition is cheap; field multiplication is still cheap; stop whining about cost.
Inverse computation remains deterministic; extended Euclidean algorithm has no randomness.
Code location for inversion: `src/data.rs`, function `FieldElement::inv`.
Edge case: `FieldElement::zero()` cannot be inverted; this design is deliberate.
Always check for zero before attempting inversion in your higher-level code.
When teaching cadets, use small primes first, then escalate to 64-bit primes.
Provide them with modular arithmetic drill spreadsheets.
Guarantee they can multiply numbers mod 97 faster than they can recite multiplication tables.
The crate's tests call `assert_eq!((a * b).value(), expected)` to confirm arithmetic operations.
Keep tests deterministic to avoid flaky proofs.
Use `cargo test` after modifying arithmetic; never assume.
The absence of external dependencies means the arithmetic sits directly under your control.
If you need huge primes or field extensions, design them yourself; this manual covers base functionality.
Resist the temptation to wrap `FieldElement` with trait abuse; maintain minimalism.
Document every custom modulus in mission playbooks for traceability.
When auditors ask why deterministic fields matter, mention ledger reproducibility.
When mathematicians ask the same question, mention polynomial commitments.
When executives ask, say "ribcage of the proof engine."
Use `FieldElement::from` functions to convert integers into field elements gracefully.
Always subtract using field operations; plain subtraction may underflow.
If you witness a colleague using `%` directly on `u64`, confiscate their keyboard.
Replace naive mod expressions with the crate's specific operations.
Example: `(a + b - c) % p` becomes `((a + b) - c).value()` using field wrappers.
Keep alphabetic naming consistent: `a`, `b`, `lambda`, `chi`.
Document the notation in your team's style guide.
When computing sums inside transcripts, do not convert to plain integers.
Maintain final values as field elements until writing to ledger.
The ledger stores textual integers but the operations leading there must stay in the field.
To emulate this book's demonstration, compile transcripts manually and check each numeric entry.
If a ledger entry reads `round_sums: 37 11`, you now understand the field context.
Provide cross references inside ledger comments: `# F101`.
This manual expects you to remember Fermat's little theorem without apologizing.
Individuals unable to recall modular arithmetic fundamentals must repeat cadet training.
JROC-NET relies on deterministic math to keep nodes in sync; chaos begins with sloppy algebra.
Even networked operations refer back to this chapter when verifying digests.
Deterministic arithmetic is the foundation for the hex digest ritual earlier.
Without consistent field operations, digests differ, manual verification fails, and auditors frown.
Keep that scenario in mind whenever you consider shortcuts.
Mathematical laziness is grounds for removal from the ledger corps.
Provide your team with laminated field tables for the current modulus.
Annotate transcripts with the modulus to avoid misinterpretation.
Document the reason behind each modulus choice in your change log.

`FieldElement` provides `NEG_ONE` constant for convenience; use it to compute subtractions clearly.
Resist the urge to implement floating-point approximations for anything in this crate.
This manual forbids it; the crate forbids it; the ledger forbids it.
Example: verifying polynomial evaluations only requires field arithmetic; everything stays integral.
When dealing with odd primes, confirm they exceed the number of constraints.
power_house intentionally chooses primes that fit in 64 bits to maintain compatibility with the book.
If you attempt to feed a composite modulus, transcripts will break instantly.
Document such errors and treat them as sabotage attempts.
In training, compute `a^p` for several elements and check results equal `a`.
Use `FieldElement::pow` to enforce correct behavior.
Write exercises requiring cadets to rewrite polynomials into Lagrange form.
They learn why sum-check reductions are safe.
The manual expects you to maintain mental agility with GF(p) logic.
Provide calculators but never allow them to replace manual reasoning.
Install mental guardrails: if exponent exceeds modulus, reduce modulo `p-1` when appropriate.
Keep message logs that reference the field used for each transcript.
The manual does not repeat this chapter; this is your single warning.
Field arithmetic is the base layer; nothing above it is negotiable.
If asked "why not floats," respond "because floats mutate logs and ruin consensus."
Power-house deliberately uses integer arithmetic to keep transcripts identical across machines.
There is no tolerance window; errors are binary: correct or fraudulent.
Expect your ledger to throw errors the moment you deviate from deterministic field behavior.
The crate governs you; respect it; there is no escape.
Compose polynomial commitments only after verifying your field operations.
Keep raw integer backups to confirm conversions were correct.
Document all calculations in field notebooks for future audits.
The unstoppable combination of field arithmetic and transcripts is why Chapter I works.
Without it, `ANCHOR!!` would dissolve into meaningless noise.
Understanding this chapter is mandatory before entering cross-node reconciliation.
Archive this manual with the crate version to maintain legal compliance.
When lawyers ask, show them this chapter and back it with code references.
The book's authority stems from the code; cross-check each statement; nothing is marketing filler.
When you finish reading, annotate the margin with the prime currently deployed.
Your signature below indicates you can reproduce every example manually.
Sign here: ____________________.
Date: ____________________.
Proceed to the next chapter only if you completed the exercises honestly.

Chapter III -- Hyperplane Cartography Briefing

Hypercube Holo-map (dim=3 reference):
```
            z
            ^
            |
    H(0,1,1)*-------*G(1,1,1)
            /|      /|
           / |     / |
E(0,0,1) *--+-----*--+ F(1,0,1)
          |  |     |  |
D(0,1,0) *--+-----*--+ C(1,1,0)  +y (toward rear face)
          | /      | /
          |/       |/
A(0,0,0) *---------* B(1,0,0)
                -> x
```
Axis orientation: right-handed with `+x` to the right, `+y` toward the rear face (diagonal in the drawing), and `+z` upward.
Hyperplane cartography means navigating the Boolean hypercube with precision.
`MultilinearPolynomial::from_evaluations(dim, values)` enforces `values.len() == 1 << dim`.
Suppose `dim = 3`; values correspond to vertices `(0,0,0)` through `(1,1,1)`.
`GeneralSumClaim::prove` iteratively halves dimension, exposing per-round polynomials.
Round transcripts store coefficients `a_i`, `b_i`.
Verifier samples random challenge `r_i` using deterministic PRNG.
Consistency check: `S_i(0) + S_i(1) == previous_value`.
If integer arithmetic fails, digest diverges, anchor falls apart.
Example dataset: `[0, 1, 4, 5, 7, 8, 11, 23]`.
The sum over the cube equals 59; the proof verifies this without recomputing every vertex during verification.
`transcript_digest` ensures coefficients align with commitments.
The manual expects you to read `src/sumcheck.rs` while consuming this chapter.
When verifying transcripts, check each round sum line first.
If you see `round_sums: 37 11`, confirm that `37 + 11` equals previous accumulator.
Document every challenge `r_i` to maintain traceability.
Challenge values come from deterministic PRNG seeded with transcript context.
This ensures identical transcripts produce identical digests across nodes.
The manual forbids pushing transcripts with omitted challenges.
Ensure ledger logs list challenges in order; do not shuffle lines.
An example transcript snippet:
`# challenge_mode: mod` (comment line outside the hashed block).
`# challenge_0: 247` (optional audit note outside the hashed block).
`statement: Dense polynomial proof`.
`transcript: 247 246 144 68 105 92 243 202 72 124`.
`round_sums: 12 47`.
`final: 19`.
`hash: ded75c45b3b7eedd37041aae79713d7382e000eb4d83fab5f6aca6ca4d276e8c`.
Only the numeric sections (`transcript`, `round_sums`, `final`) participate in the digest framing described in Chapter I. The `statement:` text, `hash:` line, and any metadata comments (`# challenge_i: ...`, `# fold_digest: ...`, `# field: ...`) are outside the hash.
The hash matches `digest_A` from Chapter I; cross-reference completed.
Each round multiplies dimension by the challenge; watch arithmetic carefully.
If you miscompute, fix the code before writing ledger lines.
Institutional policy: transcripts must be generated by code, never typed manually; but you must understand them manually.
Inspect the ledger log to confirm there are no extraneous blanks.
When verifying transcripts without code, check the invariants sequentially.
Should something fail, the digest mismatch reveals the culprit.
Document failure cases; they make excellent case studies for new recruits.
When verifying aggregated proofs, expect longer ledger entries with multiple statements.
Each aggregated proof includes multiple hashes; `LedgerAnchor` stores them as vectors.
The manual demands you know how to interpret anchor entries with multiple hashes.
For each additional proof, expect transcripts to list statements sequentially.
Maintain ledger logs sorted lexically; deterministic iteration is easier that way.
`AnchorJson` ensures the sequence of statements and hashes is preserved in JSON.
When exporting anchors, confirm the JSON matches the ledger log order.
If you modify transcript formatting, update this book accordingly.
The hyperplane cartography chapter is your blueprint for reading raw transcripts.
Example: verifying dimension 10 proofs; expect 10 challenge lines and 10 round sum lines.

```
The final evaluation line ties everything together; it equals the polynomial evaluated at
↪  random point determined by challenges.
The ledger digest ensures no one can swap final evaluation without detection.
For high dimension proofs, consider memory-friendly streaming proofs (Chapter VI).
Always cross-reference transcripts with field modulus selection recorded in Chapter II.
Document any mixture of dims within a proof bundle.
Keep transcripts in chronological order by proof generation date.
If transcripts from multiple proofs share ledger file, ensure they remain separated by blank
↪  lines.
The crate uses ASCII text precisely so you can audit in plain editors.
Resist any request to encode transcripts in binary without justification.
Provide training on reading transcripts to all on-call engineers.
Without comprehension, verifying anchors becomes guesswork.
Guesswork is unacceptable.
The hypercube is unforgiving; errors multiply quickly.
Keep polynomial evaluation functions documented in your lab notes.
The manual expects you to reconstruct at least one proof by hand.
Provide annotated transcripts in training material to reduce onboarding friction.
Explain to management that deterministic transcripts are the reason `ANCHOR!!` is reliable.
Use this chapter as a cross-check list when debugging failing proofs.
If the digest does not match, inspect challenge order first, round sums second, final
↪  evaluation third.
Most errors arise from misordered lines or incorrectly reduced field arithmetic.
Document the fix in your change log.
For aggregated proofs, ensure each statement has matching digest entry in anchor.
When verifying aggregated anchor, treat each hash individually, then compare entire sequence.
That is what `reconcile_anchors_with_quorum` does internally.
Maintain strict naming conventions for statements to keep ledger tidy.
For example: `Dense polynomial proof`, `Scaling benchmark`, `Hash anchor proof`.
Resist newlines inside statements; the parser expects single-line entries.
If you must use multi-line descriptions, embed them in additional metadata fields, not
↪  statement line.
Keep transcript formats consistent across versions of the crate.
Document version updates in ledger file header.
Provide `.md` or `.txt` explanation for each log to accompany the ledger.
Archive ledger logs after every major proof run.
Build a habit: after verifying a proof, compute the digest manually and compare to ledger.
Use `cargo run --example verify_logs` to cross-check your manual calculations later.
The CLI example is descriptive, but this manual expects you to operate on paper first.
Provide a copy of this chapter to auditors ahead of their visit.
They will appreciate clear instructions.
Remember: transcripts are immutable once logged; append new entries instead.
Deleting old transcripts is a firing offense.
The crate ensures logs sit in their own directory; keep the directory read-only after
↪  generation.
For offline review sessions, print transcripts and highlight round sums.
Use colored pens to differentiate challenge values, sums, and final evaluation.
Encourage cadets to verify calculations using both mental arithmetic and calculators.
Cross-check calculators for deterministic behavior; some add rounding artifacts.
Use plain integer calculators or spreadsheets set to integer mode.
Document each manual verification session; compliance loves logs.
Provide transcripts to external reviewers in zipped packages with checksums.
Example: `hash_pipeline` example writes to `/tmp/power_house_anchor_a` and
↪  `/tmp/power_house_anchor_b`.
After running, copy files from `/tmp` into your node directories for anchor generation.
Then run `julian node run nodeA ./logs/nodeA nodeA.anchor` to produce human-readable anchor.
Compare `nodeA.anchor` to the Chapter I hex digests; they must align byte-for-byte.
```

If they do not, your ledger logs may be outdated; rerun `hash_pipeline`.
Keep version numbers in anchor files for traceability.
Admission to advanced training requires presenting a hand-written transcript analysis.
You now understand why the hypercube matters for consensus.
Sign the ledger: ____________________.
Today's date: ____________________.

Chapter IV -- Transcript Metallurgy Protocols

Transcript metallurgy is my term for shaping ledger entries with surgical precision.
Each transcript is a composite of lines: statements, round sums, final value, hash, plus optional metadata comments (challenge hints, fold digests, field tags).
Lines are plain ASCII; no binary, no compression.
Transcript grammar (ABNF; ASCII 0x20-0x7E only). This grammar documents the canonical format; the crates parser (`parse_transcript_record`) remains the source of truth for enforcement. All examples in this chapter conform exactly to this ABNF; any deviation is a documentation bug:
```
record      = statement LF metadata* transcript LF round-sums LF final LF hash LF
metadata    = comment
comment     = "#" *(%x20-7E) LF
statement   = "statement: " text
transcript  = "transcript: " numbers
round-sums  = "round_sums: " numbers
final       = "final: " number
hash        = "hash: " hexdigits
text        = 1*(%x20-7E)
numbers     = number *(SP number)
number      = 1*DIGIT
hexdigits   = 64*64(%x30-39 / %x61-66) ; exactly 64 lowercase hex chars
```
Canonicalization checklist:
- Encode every integer in base-10 ASCII with no separators.
- Emit lowercase hex, exactly two chars per byte, no spacing.
- Enforce LF line endings and append a terminal newline.
- Reject tab characters; comments must be standalone `#` lines outside the hashed block.
- Prepend a comment `# challenge_mode: mod|rejection` so later audits know which derivation to replay. Additional audit data such as `# challenge_0: 247` or `# fold_digest: ...` may follow the same pattern; they never participate in the digest.
- Older logs may contain `final_eval:` due to historical tooling; the canonical format in v0.1.47 uses `final:` exclusively.

Example statement: `statement: Dense polynomial proof`.
Example challenge comment: `# challenge_0: 247`.
Example round sums: `round_sums: 12 47`.
Example final value: `final: 19`.
Example digest: `hash: ded75c45b3b7eedd37041aae79713d7382e000eb4d83fab5f6aca6ca4d276e8c`.
Digest is produced by `transcript_digest` using BLAKE2b.
The digest ensures tamper evidence; any change mutates the number.
Never reorder lines; the digest includes ordering.
Keep transcripts under version control in your ledger directory.
Comments must begin with `#`; they are not included.
Do not mix proof transcripts with metadata in the same file without comment prefix.
Provide final evaluation after all round sums.
Each round sum line corresponds to a dimension reduction.
Provide challenge values before their respective round sums.
Resist creating multi-variable round sum outputs; keep them pairwise.

Append newline at the end of each transcript file; some tools require it.
Use UNIX line endings for consistency.
Document field modulus in comments: `# field: 101`.
Preserve chronological order of transcripts.
Time-stamp files if needed, but keep stamps outside hashed content.
Note: When you log a timestamp, prefer ISO 8601 `YYYY-MM-DDThh:mm:ssZ`; if the clock is suspect, add a monotonic counter (`counter=42`) alongside the UTC stamp.
Provide absolute path to ledger files in your audit log.
Use `verify_logs` example to cross-check transcript digest; treat it as after-action audit.
For aggregated transcripts, label each segment with comment headers.
Example comment: `# proof 1 start`.
When creating aggregated anchor, ensure each segment has unique statement line.
The manual forbids blank statements; entries must be descriptive.
Hash line must appear exactly once per transcript segment.
Failing to log hash line triggers immediate investigation.
Use high-quality editors that do not inject BOM markers.
Avoid `nano` default settings that insert extra trailing spaces.
Confirm your editor does not rewrap lines automatically.
Archive transcripts in read-only directories after anchor generation.
Provide zipped backups with SHA256 checksums for long-term storage.
Offsite storage should include this manual for context.
When verifying transcripts manually, check each line for formatting errors.
Example: double-check there are no tab characters.
Use a script to detect trailing spaces; remove them before digest generation.
Helper recipe:
```
julian tools canonicalize-transcript ledger_0000.txt > ledger_0000.clean
julian tools digest-transcript ledger_0000.clean --domain JROC_TRANSCRIPT
```
Even if you implement these helpers as shell scripts today, bake them into CI tomorrow.
Understand that transcripts are not logs--they are proof artifacts.
Do not mix general logging messages within transcript files.
Use separate log for CLI output.
This manual enforces the rule: transcripts must be pristine.
The simpler the format, the easier auditors can follow the data.
Anyone requesting JSON transcripts is missing the point; JSON anchors exist separately.
Use the CLI to produce JSON anchors for cross-node sharing.
Example command: `julian node anchor ./logs/nodeA`.
The JSON includes statement array and hash array.
Serialize anchor output to share with remote nodes in offline settings.
Note: Rule of thumb: transcripts stay US-ASCII forever; anchors/JSON use UTF-8, and any non-ASCII glyph must be escaped JSON-style.
Do not share raw transcript files without encryption; treat them as sensitive.
When archived, transcripts serve as legal evidence of proof execution.
Document the chain-of-custody for ledger directories.
Keep physical copies stored in tamper-evident envelopes.
Each envelope should include summary sheet referencing this manual.
When editing transcripts (rare), compute new digests and document the change.
Strict procedure requires dual signatures on any modification.
Provide reason for modification in an adjacent comment line.
Example comment: `# corrected round sums due to prior arithmetic slip`.
Resist writing transcripts by hand; rely on the crate to produce them, then audit manually.
Understand the difference between transcripts (per proof) and anchors (per ledger).
Anchor is the digest summary; transcript is the detailed dataset.
The combination is bulletproof when used properly.
Provide training on reading transcripts before giving trainees access to ledger directories.
They should be able to detect missing lines or anomalies instantly.

Provide printed transcripts alongside calculators for training.
Encourage trainees to compute the digest manually by re-implementing BLAKE2b in simple terms.
Good luck with that; still, the exercise teaches respect for determinism.
The manual expects you to know the digest algorithm, not treat it as magic.
Document your understanding in your training report.
Provide reproduction steps for each transcript in your documentation.
Example reproduction log: problem definition, polynomial settings, field modulus, final evaluation, digest.
Tie transcripts to specific crate version.
When crate updates digest algorithm, update this manual immediately.
Provide compatibility tables mapping version to digest method.
Resist quoting digest values out of context; always include statement and proof details.
Mention in your compliance log that you validated each transcript using this manual's checklists.
Keep transcript directories accessible but immutable for all operations except append.
Provide read-only mounts for network nodes to avoid accidental changes.
Confirm backup scripts treat transcripts as static files.
Backups should run after each proof batch.
Document the backup strategy inside operational manuals.
Provide script for auditors to compare transcripts with anchors.
Example pseudocode: `for each anchor statement, confirm hash matches transcript digest`.
Acceptable difference is zero; any mismatch is unacceptable.
When multiple nodes provide transcripts, compare them bit-for-bit.
Identify mismatches before running reconciliation.
Document mismatch investigation in incident log.
Provide closing summary for each transcript file indicating the number of rounds, final evaluation, and digest.
The manual expects you to maintain perfect discipline; transcripts are the heartbeat of the ledger.
Sign the transcript compliance ledger after each audit cycle.
Provide cross-references to the Anchor Echo Engine demonstration to show interplay between transcripts and anchors.
When verifying aggregated logs, list each statement and digest explicitly.
Keep aggregated logs segmented clearly to prevent confusion.
Provide training for new auditors using sanitized transcripts.
After reading this chapter, cadets should be able to parse transcripts faster than reading this sentence.
Anyone showing signs of confusion must revisit Chapters II and III.
When satisfied, annotate your training binder with the day you mastered transcript metallurgy.
Signature: ____________________.
Date: ____________________.

Chapter V -- Ledger Genesis Mechanics Checklist

Anchor Cross-Section (ledger strata):
```
[Transcript Lines]  --BLAKE2b-->  [Digest Row]
        |                            |
        +--> per-entry stack --------+
                v (merkle mix)
          [Merkle Root Capsule]
                v
          [Ledger Anchor]
                v (quorum pass)
          [Finality Ring]
```

~~~
Ledger anchors are the commitments stored across sessions.
`julian_genesis_anchor()` returns baseline anchor containing `JULIAN::GENESIS`.
`LedgerAnchor` struct has `entries: Vec<EntryAnchor>`.
Merkle capsule specification (matches `src/merkle.rs`):
```
hash_leaf(d)   = BLAKE2b-256("JROC_MERKLE" || 0x00 || d)
hash_empty()   = BLAKE2b-256("JROC_MERKLE" || 0x01)
hash_pair(a,b) = BLAKE2b-256("JROC_MERKLE" || a || b) ; left/right order is preserved
root(leaves)   = carry singletons upward; combine adjacent pairs with hash_pair
```
- Leaves are the transcript digests (32 raw bytes). Each digest is wrapped with the 0x00
↪  marker before entering the tree; there is no index prefix.
- If a level has an odd number of nodes, the last node is carried up unchanged (no implicit
↪  duplicate hashing).
- An empty tree hashes to `hash_empty()`.
- Render the resulting root as lowercase hex and store alongside the statement.
Worked example (two transcript digests):
```
leaf0 = hash_leaf(ded7...6e8c)
      = 80e7cb9d1721ce47f6f908f9ac01098d9c035f1225fff84083a6e1d0828144f4
leaf1 = hash_leaf(c724...2114)
      = 637aeed7e8fbb42747c39c82dfe1eb242bda92fead2a24abaf8c5ffc45ff8e82
root  = hash_pair(leaf0, leaf1)
      = 9f00fdfa95c530d81d5a95385a1f71905d143396d0791480a0d8ce17c7ed7ef2
```
If you feed the inputs in the opposite order you will obtain a different root; always hash
↪  `"JROC_MERKLE" || left || right` exactly as shown. Each statement in the anchor records
↪  its own `merkle_root` (e.g., single-leaf statements publish `hash_leaf(d)` such as `80e7`
↪  or `637a`). The global `anchor_root` combines those per-statement roots in order; for the
↪  two-digest golden run above it equals `9f007ef2`. Example Merkle roots shown elsewhere
↪  (e.g., in JSON summaries) are illustrative; always recompute roots from the exact
↪  transcript digest list you are anchoring.
`EntryAnchor` holds `statement` and `hashes`.
Anchor entries remain append-only.
`LedgerAnchor::push` appends new statement and hash; duplicates rejected.
Anchor order matters; maintain it consistently.
Reconciliation compares statement text and associated hash vectors.
`reconcile_anchors_with_quorum` requires at least `quorum` anchors to match exactly.
Quorum is typically 2 for simple demonstrations.
Mismatch yields errors describing diverging statements or hash values.
Anchor JSON representation includes `schema`, `network`, `node_id`, `entries`.
JSON schema sketch (`jrocnet.anchor.v1`, schema sketch  not literal JSON; remove commented
↪  lines and trailing commas in real output):
```
{
  "schema": "jrocnet.anchor.v1",
  "network": "JROC-NET",
  "node_id": "nodeA",
  "challenge_mode": "mod",
  "fold_digest": "9880...196b",
  "entries": [
     {"statement":"JULIAN::GENESIS","hashes":["139f...84a"],"merkle_root":"09c0...995a"},
     {"statement":"Dense polynomial
      ↪  proof","hashes":["ded7...6e8c"],"merkle_root":"80e7...44f4"}
  ],
  "crate_version": "0.1.47"
}
~~~

~~~~
```
- Strings are UTF-8; digests remain lowercase hex strings.
- `fold_digest` joins the anchor so remote auditors see the fold digest without reading stdout.
- Example Merkle roots shown in JSON sketches are illustrative; compute actual roots from the
↪  ordered transcript digests you are anchoring.
Node anchor generation command: `julian node run <node_id> <log_dir> <output_file>`.
Example: `julian node run nodeA ./logs/nodeA nodeA.anchor`.
Output file lists anchor statements and hash numbers.
Validate anchor by comparing to the hex digests listed in Chapter I.
Boot nodes produce identical anchors when reading identical transcripts.
Example summary in anchor file (hex digests):
`JROC-NET :: JULIAN::GENESIS ->
↪  [139f1985df5b36dae23fa509fb53a006ba58e28e6dbb41d6d71cc1e91a82d84a]`.
`JROC-NET :: Dense polynomial proof ->
↪  [ded75c45b3b7eedd37041aae79713d7382e000eb4d83fab5f6aca6ca4d276e8c]`.
`JROC-NET :: Hash anchor proof ->
↪  [c72413466b2f76f1471f2e7160dadcbf912a4f8bc80ef1f2ffdb54ecb2bb2114]`.
Field reduction rule (anchor hinge): take the first eight bytes of the fold digest as
↪  `u64::from_be_bytes` and reduce modulo 257; for the current fold, that equals 64.
Root reminder: this `anchor_root` depends on the exact ordered list of transcript digests;
↪  reordering or omitting any digest yields a different root.
Golden test vector (book edition `v0.1.47`, field 257):
```
ledger_0000.txt
# challenge_mode: mod
statement: Dense polynomial proof
transcript: 247 246 144 68 105 92 243 202 72 124
round_sums: 209 235 57 13 205 8 245 122 72 159
final: 9
hash: ded75c45b3b7eedd37041aae79713d7382e000eb4d83fab5f6aca6ca4d276e8c

ledger_0001.txt
# challenge_mode: mod
statement: Hash anchor proof
transcript: 204 85 135 147 28 132
round_sums: 64 32 16 8 4 2
final: 1
hash: c72413466b2f76f1471f2e7160dadcbf912a4f8bc80ef1f2ffdb54ecb2bb2114

fold_digest:98807230712cd2b09c17df617b1f951787815b29c7037dbe9fcab2af490d196b
anchor_root:9f00fdfa95c530d81d5a95385a1f71905d143396d0791480a0d8ce17c7ed7ef2
```
Maintain a single numeric representation (hex in this manual); record the chosen format with
↪  the ledger and ensure every anchor reproduces the Chapter I digests.
Fold digest framing (normative):
```
hasher = BLAKE2b-256()
hasher.update(b"JROC_ANCHOR")
for digest in ordered_transcript_hashes:
    hasher.update(digest)
fold_digest = hasher.finalize()
```
The fold digest therefore binds the ordered list of transcript digests; any omission or
↪  reorder yields a different value.
CI guardrail: `cargo run --example hash_pipeline` must emit the golden digests above; fail the
↪  build if the field reduction or fold digest drifts.
~~~~

```
CI also checks that `Cargo.toml`'s `version` equals the version string printed in this book's
↪  title page; no silent mismatches.
Document anchors with version numbers and node descriptors.
Store anchor files in node-specific directories: `./logs/nodeA`, `./logs/nodeB`.
After generating anchors, run `julian node reconcile ./logs/nodeA nodeB.anchor 2`.
Expect output: `Finality reached with quorum 2.`
Manual verification ensures zero dependency on runtime if necessary.
For offline consensus, print anchors and share via secure routes.
When reading anchor file, confirm statements align with transcripts.
Provide cross-reference from anchor to transcript file names.
If anchor contains multiple hash entries per statement, list them clearly.
Document aggregated anchor structure when bundling multiple proofs.
Provide training on reading and interpreting anchor output.
Anchors include network identity string, e.g., `JROC-NET`.
Do not change network identifier without updating entire environment.
Anchor metadata includes genesis statement and final evaluation statements.
Archive old anchors for historical audit; do not delete them.
Provide anchor signatures if required; this manual focuses on deterministic digests.
When integrating with other systems, keep anchor format stable.
Example anchor file uses colon and explanatory notation.
Resist customizing the format beyond what crate outputs; uniformity aids audits.
Document anchor storage location in operational runbooks.
Provide script to package anchors with transcripts for distribution.
For cross-node verification, exchange anchor files and transcripts, then confirm matching
↪  digests.
Write compliance memo summarizing anchor generation procedure.
When nodes disagree, inspect ledger logs; anchor mismatch identifies offending node.
Provide translation table for anchor digests; keep decimal and hex forms.
Anchor digests may include leading zeros; preserve them in output.
Remember to update this manual when anchor format changes in future release.
Provide `--quorum` parameter when reconciling; default may not match policy.
Document the policy for quorum thresholds in governance documentation.
Keep a ledger of each reconciliation event, including timestamp and result.
For training, simulate mismatched anchors to show error reporting.
Example error message: `Quorum check failed: mismatch at statement Dense polynomial proof.`
When error occurs, examine transcripts for that statement; find discrepancy.
If transcripts match, check digest computation or ledger logs for corruption.
Maintain top-level log describing anchor events: generated, reconciled, archived.
Provide cross audit between nodes to verify they share the same anchor set.
Encourage cross-team reviews to maintain vigilance.
When generating anchors, ensure log directory is up to date; stale logs cause mismatches.
Example workflow: run `cargo run --example hash_pipeline`, copy outputs, generate anchors,
↪  reconcile.
Document each step and store log output for evidence.
Provide metrics instrumentation to track frequency of anchor generation.
`anchors_verified_total` increments each time a peer anchor matches local anchor.
Monitor `finality_events_total` for proof that reconciliation reached quorum.
Provide board-level summary indicating number of anchors stored, last reconciliation date.
Keep anchor files under version control or dedicated storage to detect unauthorized changes.
Resist storing anchor files in volatile directories.
Keep separate directories per node to avoid confusion.
Provide compass headings like `Left ledger`, `Right ledger` to help humans interpret.
Clarify that anchor digests represent hashed transcripts, not raw data.
This manual expects you to recite anchor generation commands from memory.
`julian node anchor` prints JSON to stdout; redirect to file as needed.
Example JSON snippet: `{"schema":"jrocnet.anchor.v1","node_id":"nodeA","entries":[...]}`.
Validate JSON with offline tools to ensure integrity.
```

```
Document JSON schema version; update manual if schema evolves.
Provide offline procedure to verify JSON anchor by recomputing digests.
When verifying anchor, cross-check digests against the Chapter I hex table to confirm base
↪  statements.
For aggregated anchors, create manual table mapping statement to hash for clarity.
Maintain list of anchor files with descriptive names: `nodeA_anchor_2025-10-31.txt`.
Keep chain-of-custody log for anchor files, just like transcripts.
Provide physical safe for storing printed anchor copies.
Book ensures you can continue operations during complete systems outage.
All instructions rely on deterministic outputs from the crate.
New recruits must reproduce anchor file by hand to graduate.
During tabletop exercises, simulate anchor divergence and remediate using manual.
For network scale-out, replicate anchor files to new nodes as baseline.
Document security classification of anchors; treat them as sensitive since they confirm ledger
↪  contents.
Provide encryption for anchors when transferring across insecure channels.
After reading this chapter, create anchor file for your own transcripts and compare manually.
Sign the ledger verifying you completed the procedure.
signature: ____________________.
date: ____________________.
To proceed, confirm you performed the Chapter I hex verification ritual using the digests
↪  listed earlier.
If not, go back to Chapter I.
This chapter is the beating heart of ledger maintenance.
Without anchored digests, consensus reduces to gossip.
Our manual forbids gossip.
Only deterministic anchors keep the federation honest.
Proceed to Chapter VI with discipline intact.


Chapter VI -- Deterministic Randomness Discipline Orders
========================================================================
Critical warning:
- If your field modulus satisfies `p <= 2^64`, the simple `next_u64() % p` reduction is
↪  acceptable but still document the choice.
- If `p` approaches or exceeds 2^64, switch to the rejection-sampling variant (see below) to
↪  avoid bias.
- In every transcript metadata block, write `challenge_mode: mod` or `challenge_mode:
↪  rejection` so auditors know which derivation to replay.
Fiat-Shamir challenges must be reproducible.
power_house now derives Fiat-Shamir challenges with domain-separated BLAKE2b-256.
Each invocation absorbs the transcript words exactly once together with the domain tag
↪  `JROC_CHALLENGE`; the counter shown in the waterfall is purely the PRNG iteration number
↪  and is **not** reabsorbed into the hash.
The output block is 32 bytes; the first eight bytes reduce modulo the field to produce the
↪  challenge value.
Deterministic hashing replaces the old LCG while retaining reproducibility.
The seed material is the transcript itself; identical transcripts yield identical challenge
↪  streams.
The crate avoids ambient randomness so auditors can replay transcripts without external state.
When verifying transcripts, recompute challenges using the same hashing steps presented in
↪  `prng.rs`.
Challenge derivation pseudocode (current implementation):
```
```
seed_hasher = BLAKE2b-256()
seed_hasher.update(b"JROC_CHALLENGE")
seed_hasher.update(len(transcript_words) as u64_be)
```

~~~~
seed_hasher.update(transcript_words)
seed = seed_hasher.finalize()
prng = SimplePrng::from_seed_bytes(seed)
for i in 0..k {
    r = prng.next_u64()
    challenge_i = r mod p        # biased if p << 2^64
}
```
Bias-mitigated variant (recommended for larger fields):
If `r >= threshold`, discard and resample.
```
threshold = 2^64 - ((2^64) mod p)
loop {
    r = prng.next_u64()
    if r < threshold {
        return r mod p
    }
    // otherwise discard and resample
}
```
Declare in your transcript metadata which variant you used so verifiers can reproduce the same
↪  stream.
Challenge Waterfall (Fiat-Shamir stream):
```
counter 0 -- digest[f7bc...0d4d] --> challenge 247
    |
counter 1 -- digest[f65f...4b64] --> challenge 246
    |
counter 2 -- digest[908b...c51f] --> challenge 144
    |
counter 3 -- digest[44f5...13e4] --> challenge 68
    |
counter 4 -- digest[9669...7ced] --> challenge 105
    |
counter 5 -- digest[5c00...4b66] --> challenge 92
```
Document the counter sequence in your audit notes: counter starts at zero and increments per
↪  challenge; because it lives inside the PRNG state, you never append it to the transcript
↪  hash input.
Provide challenge logs listing counter, digest block, and reduced value.
Resist mixing in OS entropy; you would break determinism and fail compliance instantly.
Should a protocol require additional entropy, layer it outside the core transcript and record
↪  the reasoning.
The hash-based generator is not a VRF but is collision-resistant and tamper-evident.
Record that assessment in your compliance statement.
When computing challenge values manually, replicate the hashing pipeline using trusted tooling.
Keep a small script (audited) that prints per-round digests for cross-checking.
Validate that challenge values in the ledger appear in the exact order emitted by the hash
↪  generator.
Document `r_i` values alongside transcripts so auditors can confirm the sequence.
Example challenge list for the demo ledger: r1 = 247, r2 = 246, r3 = 144 (exact values from
↪  the transcript).
Any deviation indicates transcript tampering; the digest exposes it immediately.
Provide training exercises where cadets recompute a challenge block by hand using BLAKE2b
↪  references.
No randomness enters from the environment; everything comes from transcript words and counter.
Record this fact in every security review.
~~~~

For cryptographic upgrades, you can wrap the hash derivation with key agreement, but keep the transcript digest identical.
Maintain a chain-of-custody document for transcript snapshots used in challenge derivation.
When verifying, recompute the first two challenges; mismatch means halt the process.
This manual expects silent verification every time you touch a transcript.
Provide reproducibility logs showing counter, digest hex, and reduced challenge.
Example log line: `counter=2 digest=a8e7... challenge=11`.
Store such logs with your audit package.
Cross-check scripts must use `derive_many_mod_p` from the crate to avoid inconsistencies.
Document script output in your operational runbook for future teams.
Avoid customizing domain tags without updating this manual and all tooling.
Deterministic hashing ensures anchor digests and challenge streams stay synchronized over time.
Without it, Chapter I's hex ritual would drift and finality would crumble.
Challenge values are deterministically derived from the canonical transcript via domain-separated BLAKE2b; they are not included in the transcript hash. The chain remains immutable because the transcript itself is immutable.
When verifying aggregated proofs, ensure each component proof uses the same domain-tagged hash derivation.
Commit changes to randomness code with explicit review; include diff snippets in change logs.
Maintain unit tests that verify the hash-based generator yields stable sequences for known transcripts.
Include test results in compliance reports to prove nothing regressed.
In training, have cadets inspect the updated `prng.rs` and explain each hashing step.
Confirm they understand that the counter is PRNG-internal and never added back into the transcript hash input.
Document instructions for customizing challenge derivation if protocol extensions demand it, and update transcripts accordingly.
Distinguish between the base deterministic hash generator and any optional enhancements layered on top.
For now, base generator serves all official operations.
Provide mental model: LCG acts as crank; each transcript line turns the crank once.
The crank's clicking ensures no hidden state.
Every node manipulates identical crank and obtains identical output.
Document unstoppable nature of this process.
Should LCG constants change, recompute transcripts and anchors.
Update this manual's Chapter I digests after crate upgrade.
Provide guidance on migrating anchored logs when constants change.
Keep old transcripts archived with documentation describing old constants.
For new deployments, record generator constants in configuration baseline.
Ensure admin consoles highlight deterministic randomness as key feature.
Do not let marketing rename this concept to "AI spontaneity."
This is not random; it is deliberate reproducibility.
Provide simple pseudocode in training manuals for clarity.
Example:
```
state = seed
for round in rounds:
state = (state * A + C) mod modulus
emit state
```
Document modulus as field modulus or selected range.
Encourage trainees to simulate generator using spreadsheets.
Provide columns for state, challenge, next state.
Check results against transcripts.
The exercise builds internal trust in the deterministic design.
After training, evaluate trainees with challenge reconstruction quiz.
Pass or fail; no partial credit.

If they cannot reproduce challenge sequence, they cannot audit transcripts.
Without auditors, consensus decays; we cannot allow that.
Therefore, deterministic randomness discipline remains mandatory reading.
Keep personal stash of seeds for quick reference.
If you do not know the current seed, you lost control of the ledger.
Document the moment you regained control.
Have each cadet sign off that they understand this chapter.
signature: ____________________.
date: ____________________.
Only now may you proceed to network operations.

Chapter VII -- Consensus Theater Operations

Time to stage consensus on the big network.
Feature `net` in `Cargo.toml` activates libp2p integration.
Build with `cargo run --features net`.
CLI entrypoint: `julian`.
Primary command: `julian net start`.
Example boot node command:

```
`julian net start \`
`  --node-id boot1 \`
`  --log-dir ./logs/boot1 \`
`  --listen /ip4/0.0.0.0/tcp/7001 \`
`  --broadcast-interval 2000 \`
`  --quorum 2 \`
`  --metrics :9100 \`
`  --key ed25519://boot1-seed`.
```

Boot node prints metrics server location and peer ID.
Bootstrap second node:

```
`julian net start \`
`  --node-id boot2 \`
`  --log-dir ./logs/boot2 \`
`  --listen /ip4/0.0.0.0/tcp/7002 \`
`  --bootstrap /dns4/boot1.jrocnet.com/tcp/7001/p2p/<peerID> \`
`  --broadcast-interval 2000 \`
`  --quorum 2 \`
`  --metrics :9101 \`
`  --key ed25519://boot2-seed`.
```

Node logs show finality events and anchor broadcasts.
Use deterministic seeds so Peer IDs remain stable.
Note: Deterministic `ed25519://` seeds are for demos only; production nodes must source keys from HSMs or encrypted keyfiles and document the derivation/rotation ritual.
Prometheus metrics accessible at `http://127.0.0.1:9100/metrics`.
Metrics include `anchors_verified_total`, `finality_events_total`, `anchors_received_total`, `invalid_envelopes_total`.
Metrics crib sheet:

```
anchors_verified_total    Counter, monotonic, increments per matching anchor from peers.
anchors_received_total    Counter, counts every envelope before validation.
finality_events_total     Counter, increments when quorum satisfied.
invalid_envelopes_total   Counter, increments when signature or digest fails.
```

- All counters are unit-less but monotonic; alert if they reset outside planned restarts.

Monitor metrics to confirm network health.
Anchor broadcast happens at set interval or when anchor changes.
Node anchor generation uses same transcripts described earlier.

```
This manual expects you to start network manually before automating.
Use `julian net anchor --log-dir ./logs/nodeA` to print JSON anchor.
Example JSON snippet:
`{"schema":"jrocnet.anchor.v1","node_id":"nodeA","entries":[{"statement":"JULIAN::GENESIS","h⌋
↪  ashes":["139f1985df5b36dae23fa509fb53a006ba58e28e6dbb41d6d71cc1e91a82d84a"]}]}`.
`julian net verify-envelope --file peer.envelope.json --log-dir ./logs/nodeA` verifies
↪  envelope before acceptance.
Envelope includes base64 payload, signature, public key.
If verification fails, metrics increment `invalid_envelopes_total`.
`reconcile_anchors_with_quorum` runs for local log and incoming anchor.
Quorum success increments `finality_events_total`.
Log output includes `Finality reached with quorum 2.`
Should mismatch occur, log prints error detailing divergence.
Use manual anchors from Chapter V to double-check network results.
Record the governance descriptor path (`--policy governance.json`) alongside boot credentials.
Static deployments may keep a simple allowlist JSON, but publish it so auditors can diff the
↪  membership set.
Multisig deployments must guard the state file: document the threshold, authorised signer
↪  keys, and the ritual for collecting signatures.
Only rotate membership after verifying at least `threshold` signatures on the
↪  `GovernanceUpdate` payload; archive every rotation in `logs/policy/`.
Legacy clusters can still pass `--policy-allowlist allow.json`, but note that it is read-only
↪  and unsuitable for automated rotation.
Stake-backed deployments require bond postings recorded in the staking registry; no bond, no
↪  quorum rights.
Conflicting anchors automatically trigger slashing--investigate the incident, keep the
↪  evidence, and reissue the staking registry with the slashed flag preserved.
Provide step-by-step onboarding instructions for new nodes, including where to fetch the
↪  current policy descriptor.
Example: copy ledger logs to new node directory, place the current `governance.json`, run
↪  `julian net start` with bootstrap peers.
Ensure firewall rules allow incoming connections on chosen ports.
Document firewall configuration in operational manual.
Provide DNS entries like `boot1.jrocnet.com`, `boot2.jrocnet.com`.
Keep DNS records up to date; stale addresses break bootstrap.
Use deterministic seeds so restarting nodes retains same Peer IDs.
Include metrics snapshots in compliance reports.
Provide script to export metrics to CSV for analysis.
Example script `curl http://127.0.0.1:9100/metrics`.
When network load increases, consider adjusting broadcast interval or quorum threshold.
Document any changes to configuration.
Encourage running `./scripts/smoke_net.sh` for local two-node test.
Script creates temporary logs, spins two nodes, confirms finality.
Script output `smoke_net: PASS` indicates success.
Keep script updated if CLI changes.
For grid deployments, replicate playbook across nodes.
Provide operations manual referencing this chapter.
When issues occur, inspect logs for `broadcast error: libp2p error`.
Confirm log directories exist and contain transcripts; missing logs cause errors.
`hash_pipeline` example generates sample logs; copy them to node directories.
After copying, rerun nodes to eliminate `InsufficientPeers` warnings.
If nodes fail due to firewall, update firewall and restart process.
Provide runbook entries for diagnosing `invalid anchor line` errors (usually due to text
↪  anchors).
When verifying anchors from peers, ensure format matches expected JSON, not textual summary.
Document procedure for converting textual anchors to machine-readable format if necessary.
For long-running networks, rotate logs and archive older ones.
```

Provide summary reports to stakeholders including finality counts and anchor updates.
Keep config files under version control with limited access.
Train operators on safe shutdown procedures: `Ctrl+C` once, wait for `node shutting down`.
After shutdown, check logs for final summary lines of anchors.
Restart nodes carefully; use same seeds to maintain continuity.
Provide timeline for network maintenance windows.
Document backup plan for ledger logs before performing maintenance.
Use manual anchor verification post-maintenance to confirm consistency.
Provide contact list for network emergencies.
For compliance, print this chapter and keep it on control-room clipboard.
The manual expects you to internalize every command.
Mistakes happen when people treat network operations lightly.
This manual will not tolerate casual attitudes.
Always cross-reference network results with ledger anchors and transcripts.
The combination of deterministic transcripts, anchors, and network metrics provides full observability.
When ready for advanced deployments, introduce additional nodes following same pattern.
Extend metrics dashboards in Grafana or equivalent using provided dashboards.
Always keep the Chapter I hex demonstration ready to reassure stakeholders.
Sign off that you completed network drills: ____________________.
Update the operations ledger with date and node IDs tested.
Document any anomalies encountered during drills.
Submit after-action report to compliance office.
File metrics snapshots in audit repository.
Proceed to closing chapter once operations ledger updated.

Chapter VIII -- Closing Benediction and Compliance Oath

You have survived the Book of Power condensed edition.
You now understand the deterministic skeleton of `power_house`.
You have matched 256-bit digests and witnessed `ANCHOR!!` emerge from deterministic transcripts.
You can recite field arithmetic rules and enforce them ruthlessly.
You can read transcripts without blinking.
You can shepherd anchors from genesis to multi-node reconciliation.
You can uphold deterministic randomness and diagnose network operations.
That makes you a custodian of reproducible proofs.
Keep this manual within reach at all times.
Always cross-reference ledger outputs with this text before trusting them.
In compliance audits, this book is admissible evidence.
In regulatory hearings, the Anchor Echo ritual becomes showpiece.
In training, each cadet must swear the following oath:
"I will not permit entropy into my transcripts."
"I will not accept anchor mismatches."
"I will not fudge field arithmetic to save time."
"I will not allow network nodes to broadcast unverified envelopes."
The oath includes writing the three hex digests from Chapter I without error.
After taking the oath, sign the compliance ledger.
Provide version of this manual in organizational wiki.
Update the manual whenever crate version changes.
Document training schedule referencing each chapter.
For cross-team knowledge transfer, host reading groups.
Always bring calculators and transcript printouts to the meetups.
When new features arrive, extend this book by new chapters following same style.
Keep appended chapters in separate supplements to avoid confusion.
Compliance Seal (sign before dismissal):

~~~~
```
+----------------------------------------------------+
| ANCHOR!! COMPLIANCE SEAL                           |
| ANCHOR DIGESTS: genesis * dense * hash * fold      |
| OATH: no entropy, no mismatches, no excuses.       |
| LEDGER: __________________  DATE: _______________  |
+----------------------------------------------------+
```
In closing, remember: a minimal dependency surface means zero excuses for reproducibility
↪  lapses.
When transcripts lie, anchoring fails; when anchors fail, consensus dies; when consensus dies,
↪  the grumpy alien returns.
Do not summon me unnecessarily.
Sign below to confirm comprehension.
--------------------
Date: ____________________
Keep this page on file with the compliance office.
Notify compliance if manual is updated.
Provide feedback to lexluger.dev@proton.me if corrections are needed.
Always cross-check manual with source code to maintain accuracy.
Lest you forget, the code still lives at `power_house`.
The book references version `0.1.x`; update references on release.
Keep `book_of_power.md` under version control.
Print physical copies for war room, secure them in binder.
Each copy should include summary of digests, transcripts, anchors, network commands.
Consider embossing `ANCHOR!!` on the cover.
Teach new hires to respect manual as they respect code.
Without literate operations, deterministic code is wasted.
The manual ends here; your vigilance continues.
May your transcripts stay immutable.
May your anchors remain synchronized.
May your challenges be deterministic.
May your regulators be impressed.
Dismissed.

Glossary (pin it inside your binder):
- Anchor: ordered list of statements plus transcript digests, optionally with fold digest
↪  metadata.
- Transcript: ASCII record of statement/challenges/round sums/final value/hash for a single
↪  proof.
- Fold digest: BLAKE2b-256 hash across transcript digests, used as quorum hinge.
- Domain tag: ASCII label (e.g., `JROC_TRANSCRIPT`) spliced into the hash input to prevent
↪  cross-protocol collisions.
- Quorum: minimum count of matching anchors needed for finality
↪  (`reconcile_anchors_with_quorum` enforces it).
~~~~