TensorCash Asset Protocol: TLV, Governance, KYC, and Distribution on a Bitcoin Fork

Imosuke Takakuni[1]

2 May 2026

ABSTRACT

This paper describes the TensorCash asset protocol — a faithful, consensus-enforced extension to a Bitcoin Core fork that allows a single chain to issue, govern, and transfer arbitrary fungible assets alongside its native coin. The protocol is built around three load-bearing ideas. **First**, every output may carry a single typed binary record (TLV) which is committed to by the transaction's sighash, so that asset state cannot be rebound after a signature is produced. **Second**, every asset has an *Issuer Control Unit* (ICU): a UTXO-shaped on-chain credential that the issuer must rotate on every authorized spend, and that is bonded by a posted native-coin stake released only when a fee threshold has accumulated. **Third**, regulated assets carry zero-knowledge proofs of holder eligibility (KYC) using on-chain Groth16 verification keys, on-chain rolling compliance roots, and per-spend Taproot input-key binding. The result is a system where a token is a first-class UTXO — composable with Bitcoin's existing script families, taproot covenants, and PSBT tooling — but where issuance, transferability, compliance, and the consensus-governed subset of issuer governance are enforced at consensus, not at an off-chain layer.

I. Introduction

Bitcoin's UTXO model is the most battle-tested ledger primitive in cryptocurrency. It is also, deliberately, the most minimal: each transaction output is a tuple of an amount in satoshis and a script. Anything beyond — tokens, registries, identity, governance — historically had to be bolted on as a separate protocol layer (Omni, RGB, Liquid, OP_RETURN-based standards) or as a separate chain entirely.

TensorCash takes the opposite stance. Because the chain is already a fork of Bitcoin Core to support inference-anchored Proof-of-Work [2], the same fork is the right place to teach the consensus layer about assets. We do this by adding *exactly one* optional field to the transaction output (`vExt`), gated by a single bit in the transaction marker, and we make that field part of the sighash. From there, asset metadata for registries, mint/burn, governance ballots, KYC proofs, and contract delivery is carried as TLV records inside `vExt` and validated by the same node that validates a Bitcoin transaction. Higher-level derivative covenants remain normal script/RPC constructions that preserve those TLVs.

This paper is a technical reference for that protocol. It is meant to be read by implementers and auditors, not by traders. Where we make a normative claim, we cite the relevant source path so the reader can verify it against the reference implementation.

1. "Imosuke Takakuni" is a pseudonym. Contact: takakuni@tensorcash.org

2. See *Inference-Based Decentralised Value: TensorCash* (Takakuni, 1 April 2025) for the TensorCash proof-of-work scheme that secures this chain.

TABLE OF CONTENTS

II. Design Goals

The asset protocol is designed against five hard constraints. We state them up front because every consensus rule that follows is a direct consequence of one or more of them.

1. **Self-contained.** A node validating a TensorCash block must not need to consult any off-chain oracle, registry, or identity provider. Asset state is a function of the chain alone.
2. **No double-meaning UTXOs.** A coin either carries an asset balance or it doesn't, and which it is must be visible from the UTXO record without consulting transaction history. This is what allows the UTXO database to be queried, compressed, and undone in the usual Bitcoin way.
3. **Signatures must commit to asset effects.** If a spender signs a transaction, the asset deltas that transaction produces must be inside the sighash. Otherwise an adversary who relays the transaction can graft outputs that move tokens the signer never intended to move.
4. **Issuer authority is a UTXO, not a key.** Issuer privileges (mint, burn, ticker, KYC root) live on a single rotating output — the *Issuer Control Unit* — rather than at a fixed scriptPubKey. This makes issuer key rotation, multi-sig issuer governance, and stake-bonded issuance a single mechanism.
5. **Compliance without identity leakage.** Where regulation requires it, holders prove eligibility under a Groth16 statement; the chain enforces the proof; the proof publishes nothing about who the holder is.

The rest of this document explains how each consensus rule serves one of these goals.

III. The vExt Wire Format

Every Bitcoin transaction output has historically been the pair `(nValue, scriptPubKey)`. TensorCash adds an optional third field, `vExt`, of type `std::vector<unsigned char>` (`src/primitives/transaction.h:151–157`). It is an opaque byte string from the wire's point of view; only the parsing layer interprets it.

`vExt` is gated by a single bit in the transaction's flags byte, `TXFLAG_OUTEXT = 0x02` (`src/primitives/transaction.h:217`). When the flag is clear, every output of the transaction serializes exactly as in stock Bitcoin — there is no extra byte on the wire and old wallets remain readable. When the flag is set, every output appends a `CompactSize` length followed by `vExt.size()` raw bytes; an empty `vExt` is encoded as a single zero byte.

Two consensus caps protect node memory:

- `MAX_OUEXT_SIZE_PER_OUTPUT = 100 KiB` per `CTxOut`,
- `MAX_OUEXT_BYTES_PER_TX = 128 KiB` summed across the transaction.

A transaction that overruns either cap is invalid (`src/primitives/transaction.h:221–222`, `src/primitives/transaction.h:321–328`). Coinbase transactions are forbidden from using `vExt` to carry `AssetTag` balances (`src/validation.cpp:3685–3695`), so no asset units can be conjured into circulation by mining alone. Consensus does not reject an `IssuerReg` in coinbase, but every mint still must spend a pre-existing ICU.

i. Activation gating

Asset support activates at a configurable block height controlled by the consensus parameter exposed as `-assetsheight` (`src/kernel/chainparams.cpp:235–236`). Below that height, both transaction input checks (`src/consensus/tx_verify.cpp:215–221`) and block connection (`src/validation.cpp:3658–3663`) reject any non-empty `vExt`. This makes the deployment behaviour identical to a soft fork: pre-activation blocks are bit-for-bit Bitcoin, post-activation blocks may carry asset traffic, and a node that never sees a non-empty `vExt` never executes any of the asset code paths.

ii. Sighash binding

The single most important property of `vExt` is that it is part of the transaction's signature digest. Both BIP-143 (legacy/segwit) and BIP-341 (Taproot) sighash routines hash outputs through `SerTxOutWithExt`, so that the bytes appearing in `vExt` are inside the message a signer commits to (`src/script/interpreter.cpp:1485–1496`). If even one byte of any output's `vExt` changes after a signature is computed, the signature is invalid.

This property is what turns `vExt` from a glorified comment field into a consensus-enforced asset ledger. A signer who authorises *"send 1000 units of asset X to Alice"* is committing not only to the recipient's `scriptPubKey` and `nValue`, but to the `AssetTag` TLV that *encodes* the *"1000 units of asset X"* claim.

iii. Weight and fee accounting

Asset transactions do not get a separate accounting lane. The `vExt` bytes are part of transaction serialization whenever the output-extension flag is set, so they are included in transaction weight and therefore in virtual size (`vsize = ceil(weight / 4)`). The standard fee path charges the native-coin fee against that vsize (`src/consensus/validation.h:130–138`, `src/policy/policy.cpp:600–607`).

The practical consequence is simple: every `AssetTag`, `IssuerReg`, `ICU_TEXT_CHUNK`, `ZK_PROOF_PAYLOAD`, or keywrap byte consumes block weight and must be paid for at the transaction's sat/vB fee rate. The asset protocol caps large payloads for DoS resistance, but those caps do not create free block space. Asset units themselves never pay miner fees; the transaction's native-coin inputs pay the fee for the full weighted transaction, including the asset TLVs.

iv. Single-TLV-per-output rule

A `vExt` payload, if present, must parse as exactly one TLV (`src/primitives/transaction.h:237–273`). The format is:

$$ \text{vExt} \;=\; \underbrace{T}_{1\,\text{byte}} \;\Vert\; \underbrace{L}_{\text{CompactSize}} \;\Vert\; \underbrace{V}_{L\,\text{bytes}} $$

where T is one of the type bytes in the `OutExtType` enum (`src/assets/asset.h:16–24`), L is a `CompactSize` length, and V is the raw payload. A single TLV per output simplifies undo records, parser fuzzing, and human inspection. Where richer composition is needed — e.g. an `AssetTag` that also carries an ICU keywrap — the inner record is itself a sub-TLV stream within V, but the outer container is always one record.

IV. The TLV Catalog

TensorCash defines seven `OutExtType` codes. Each has a fixed type byte and a deterministic byte layout. The following list is the *complete* set of TLVs the consensus layer will accept; an unrecognised type byte is a hard error (`src/consensus/tx_verify.cpp:342–345`, `src/validation.cpp:3667–3679`).

Type	Name	Purpose
0x01	`ASSET_TAG`	Carries an asset balance on a normal-looking UTXO.
0x10	`ISSUER_REG`	Issuer Control Unit: declares or rotates an asset's policy and ticker.
0x20	`ZK_PARAMS_CHUNK`	Chunked Groth16 verification key transport.
0x21	`TFR_ANCHOR`	Transferability anchor for compliance tooling.
0x22	`ZK_PROOF_PAYLOAD`	Groth16 proof + public inputs for a regulated spend.
0x30	`ICU_TEXT_CHUNK`	Encrypted, opaque payload chunk attached to an ICU.
0x31	`ICU_KEYWRAP`	Per-recipient wrapping of the ICU symmetric key.

i. AssetTag (0x01)

The `AssetTag` is the workhorse of the system: any output that carries asset value must hold one. Layout (`src/assets/asset.cpp:101–208`):

- `asset_id` — 32 bytes, the asset's unique identifier; must be non-zero.
- `amount` — 8 bytes little-endian, the unit count; must be non-zero.
- `flags` — 4 bytes little-endian in canonical tags, currently consensus-required to be zero. Some parser paths tolerate omission before epoch/keywrap sub-TLVs, but proposal-hash tags should include the zero flags word.
- Optional sub-TLV `0x02` (`epoch`, 1 byte) — used for governance epoching.
- Optional sub-TLV `0x03` (`ICU_KEYWRAP`) — embeds a recipient-bound symmetric key wrap; see ICU Encrypted Payloads.
- Optional sub-TLV `0x04` (`proposal_hash`, 32 bytes) — used to bind a holder vote to a particular governance proposal; see Governance.

The `AssetTag` must reside on a normal `scriptPubKey` from one of the issuer's allowed script families. From the spender's wallet point of view it is a regular UTXO that happens to carry token units alongside its native-coin balance. Consensus enforces that token units are conserved per `asset_id`; the native `nValue` remains ordinary fee/accounting value and can affect miner fees unless wallet policy preserves or funds it separately.

ii. IssuerReg (0x10)

The `IssuerReg` is the *Issuer Control Unit* serialized form. It is parsed by a strict v1 parser (`src/assets/asset_parser_v1.cpp:42–186`) which accepts a fixed 254–265-byte canonical layout. A simplified outline:

Section	Field	Bytes	Notes
Header	`asset_id`	32	
	`policy_bits`	4 LE	mint/burn/KYC/TFR flags
	`allowed_spk_families`	2 LE	bitmask over the five script families
	`format_version`	1	must be `0x01`
Optional	`ticker_len` and ticker	1+0..11	uppercase A-Z 0-9, first char A-Z

Section	Field	Bytes	Notes
Fixed	`decimals`	1	0-18 or 0xFF
	`unlock_fees_sats`	8 LE	bond-release threshold
ZK	`kyc_flags`	4	
	`vk_commitment`	32	Groth16 VK hash; must be non-zero when KYC is enabled
	`max_root_age`	4	freshness window in blocks
	`tfr_flags`	4	
	`compliance_root_commit`	32	legacy 28+4 or HDv1 32 MiMC
ICU	`icu_flags`	4	WRAP_REQUIRED, ICU_COMPRESSED
	`issuance_cap_units`	8	
	`icu_ctxt_commit`	32	SHA256 of encrypted payload
	`icu_plain_commit`	32	
	`kdf_salt`	16	
	`icu_version`	1	
	`icu_visibility`	1	0=public, 1=holder-only
	`core_policy_commit`	32	freezes the immutable mask once supply has been issued
	`policy_epoch`	1	
	`policy_quorum_bps`	2 LE	0 means immutable after issuance; otherwise holder quorum bps

The strict layout makes the parser's job trivial and makes the on-disk registry deterministic across nodes.

iii. ZK_PARAMS_CHUNK (0x20), TFR_ANCHOR (0x21), ZK_PROOF_PAYLOAD (0x22)

These three TLVs are the compliance toolkit and are described in detail in KYC. In short:

- `ZK_PARAMS_CHUNK` carries one fragment of a Groth16 verification key. A VK can be split across at most `MAX_ZK_CHUNKS = 8` chunks of `MAX_ZK_CHUNK_SIZE = 512` bytes each, totalling 4 KiB.
- `TFR_ANCHOR` declares transferability anchors (`tfr_commit` plus an opaque off-chain `locator` of at most 128 bytes). Consensus requires the anchor count to match asset outputs; current proof binding checks the first anchor.
- `ZK_PROOF_PAYLOAD` carries a single 192-byte Groth16 proof (compressed `A‖B‖C`) and its public-input vector, sized for either the legacy 4-input circuit or the HDv1 6-input circuit.

iv. ICU_TEXT_CHUNK (0x30) and ICU_KEYWRAP (0x31)

`ICU_TEXT_CHUNK` carries the issuer's encrypted off-chain *contract text* (terms, prospectus, custody disclosure) attached to an ICU. The chain treats it as opaque bytes up to `MAX_ICU_PAYLOAD_BYTES = 100 KiB` and only checks SHA-256 against the `icu_ctxt_commit` field declared in the `IssuerReg`. `ICU_KEYWRAP` carries the per-recipient wrapping of the symmetric key under which `ICU_TEXT_CHUNK` was encrypted, with optional `wrap_commit` and `kc_tag` for verifiable wrap-on-spend protocols. See ICU Encrypted Payloads for the rationale.

V. Issuer Control Units and the Bond Lifecycle

The Issuer Control Unit (ICU) is the central abstraction of the asset protocol. An ICU is a single UTXO that carries:

- a non-zero satoshi balance (the **bond**), and
- an `IssuerReg` TLV (the **policy**),

and that exists at exactly one outpoint at any time. The combination of "single UTXO" plus "must be spent on every authorised action" is what gives the ICU its useful properties: it is impossible to clone, it is trivial to track in the UTXO set, and rotating it is a normal Bitcoin transaction.

i. Birth: `registerasset`

A new asset is usually created by a wallet transaction that spends native-coin funding inputs and produces an output bearing an `IssuerReg` TLV. Consensus also accepts an `IssuerReg` in coinbase, but never a coinbase `AssetTag`. The bond floor is the hardwired consensus constant `Consensus::Params::AssetMinIcuBond` (5 BTC; `src/consensus/params.h`). The issuer chooses:

- the policy bits that become immutable once supply has been issued (described below),
- the allowed scriptPubKey families,
- the ticker (3–11 uppercase ASCII, immutable once set),
- the optional ZK / TFR commitments,
- and the **unlock threshold** `unlock_fees_sats`, which must be at least the bond.

Once mined, the (txid, vout) of this `IssuerReg` output is the asset's *current ICU outpoint*. The chainstate stores it in a LevelDB-backed registry keyed by `asset_id` (`src/txdb.cpp:161–213`).

ii. Rotation: every authorised spend rotates the ICU

This is the crucial consensus rule. Whenever a transaction spends a coin under issuer authority — which always means an `IssuerReg` input — that transaction must produce **exactly one** new `IssuerReg` output for the same `asset_id` (`src/validation.cpp:4219–4227`). The new ICU may have updated mutable fields (e.g. an updated `compliance_root_commit`, an updated `icu_ctxt_commit`). Once issued supply exists, the *immutable* fields captured by `core_policy_commit` must be unchanged.

The rotated ICU's bond must satisfy a floor: at least 95% of the inception bond until the asset is *unlocked*, then dust afterwards (`src/validation.cpp:4365–4374`). The 95% slack lets the issuer pay BTC fees out of the bond without immediately falling below the floor; the dust floor after unlock keeps the rotation cheap during normal operations.

iii. Unlock: the fee accumulator

Each rotation credits the native-coin fees consumed by the carrying transaction to the asset's `fees_accum_sats` counter (`src/validation.cpp:3950–3961`). The bond is not automatically repaid at registration time, mint time, or first rotation. It remains trapped in the current ICU until cumulative paid fees reach the issuer's threshold: `fees_accum_sats >= unlock_fees_sats`.

Once the inequality is true, the asset is permanently *unlocked*. From that point on, the rotated ICU only has to retain a dust-level native-coin value, so the issuer can recover the rest of the original bond by routing it to an ordinary output in the same rotation. Before unlock, every ICU spend must preserve the inception floor (`rotation_min_sats`, normally 95% of the posted bond), so a series of rotations cannot gradually drain the stake.

This is also how the protocol discourages governance churn. A mutable governance change is not just a metadata edit: it is an on-chain ICU rotation, it consumes block space, it pays the full sat/vB fee for the transaction, and when the ICU payload or visibility changes the implementation requires a `UNLOCK_BUMP_MIN_DEFAULT = 50,000,000` sat increase to the unlock target (`src/assets/asset.h:203`). Churning governance therefore both pays immediate transaction fees and pushes the bond-release target further away. Other rotations must still set `unlock_fees_sats` at least as high as the rotated bond, but there is no general consensus rule that the threshold increases on every rotation.

iv. SIGHASH discipline on ICU inputs

The ICU is the only authority that can mint or burn. So the chain demands that every signature on an ICU input commits to the full output set: `SIGHASH_ALL` (or Taproot `SIGHASH_DEFAULT`) is required, and `SIGHASH_ANYONECANPAY` is forbidden (`src/consensus/tx_verify.cpp:305–327`). The error code is `icu-invalid-sighash`. Without this rule, an issuer who signed *"mint 1M units to Alice"* could have their signature spliced into a transaction that mints those units to an attacker's address.

VI. Asset Distribution: Mint, Transfer, Burn

The protocol exposes three primary operations and a small family of helpers. All of them produce ordinary Bitcoin transactions; the asset semantics are entirely in the TLVs and in the consensus check that runs over them.

i. Conservation

The base law is that for every transaction and every `asset_id`,

$$\sum_{\text{inputs}} \text{units(asset\_id)} = \sum_{\text{outputs}} \text{units(asset\_id)}$$

unless the transaction includes an ICU input for that asset and the policy bits permit a non-zero delta. The check is implemented in `src/consensus/tx_verify.cpp:265–276` over a 128-bit accumulator to make overflow attacks unrepresentable. Asset units *never* contribute to miner fees: a transaction that "burns" tokens without an ICU input is invalid; only the native-coin component of `nValue` ever counts.

ii. Mint (`mintasset`)

A mint transaction spends the current ICU and creates one or more `AssetTag` outputs whose `amount` totals the desired mint quantity. It must:

1. include the current ICU input,
2. produce exactly one rotated ICU output,
3. have `policy_bits & MINT_ALLOWED` set on the issuer's policy,
4. obey the `allowed_spk_families` mask on every recipient script,
5. carry a SIGHASH_ALL signature on the ICU input,
6. respect `issuance_cap_units` if set in the ICU.

iii. Transfer (`sendasset`)

A transfer is the most common case. It is a regular Bitcoin transaction that spends asset-bearing UTXOs and produces new asset-bearing UTXOs to the same total per `asset_id`. It does *not* require an ICU input. Where the asset is regulated (`KYC_REQUIRED`, `TFR_ANCHOR_REQUIRED`), the transfer must additionally carry the corresponding ZK and anchor TLVs (see KYC).

The wallet RPC `sendasset` automates the full pipeline (`src/wallet/rpc/assets.cpp:541–1550`): selection of asset UTXOs, generation of `AssetTag` TLVs, optional generation of `ZK_PROOF_PAYLOAD` and `ICU_KEYWRAP`, BTC fee funding from non-asset coins, and re-application of `vExt` to outputs after `FundTransaction` strips them. A critical option is `return_skeleton=true`, which causes `sendasset` to return an *unsigned* transaction plus metadata (deliver index, change indices, locked inputs); higher-level covenant builders graft additional inputs and outputs onto this skeleton before signing. This is the common path for atomic swaps, repos, and forwards.

iv. Burn (`burnasset`)

Burn consumes asset units without producing replacements. Like mint, it requires the ICU input and `policy_bits & BURN_ALLOWED`. Two refinements are available:

- `BURN_REQUIRE_ICU` (set by default for any asset where the issuer wants exclusive burn rights): the holder cannot burn alone; only an issuer-authorised transaction can.
- `BURN_JOINT_REQUIRED`: the spending script for the burned asset units must come from `SPK_HOLDER_ONLY = SPK_P2PKH | SPK_P2WPKH`, i.e. a single-signer script. This is the regulated-asset case: burns happen only when both the issuer (via ICU) and the holder (via their own signature) cooperate.

v. Raw-transaction helpers

For tooling and testing, the node exposes:

- `rawtxaddoutext(hex, vout, tlv_hex)` — attach any TLV to an output;
- `rawtxattachissuerreg(...)` — build a canonical `IssuerReg`;
- `rawtxattachassettag(...)` — build a canonical `AssetTag`, optionally with an embedded keywrap.

`decoderawtransaction` is extended to expose `vout[].outext` so any third-party tool can inspect TLVs without re-implementing the parser.

VII. Governance: Immutable Bits, Quorum Rotation, Ballot Binding

The protocol distinguishes sharply between *spend semantics* (which can never change after issuance) and *operational parameters* (which can change under a quorum vote).

i. The immutable mask

The following bits are frozen after supply has been issued:

```
MINT_ALLOWED        = 0x0001   // Issuer may create units
BURN_ALLOWED        = 0x0002   // Issuer may destroy units
BURN_REQUIRE_ICU    = 0x0004   // Burn must include ICU input
BURN_JOINT_REQUIRED = 0x0008   // Burn requires holder signature
KYC_REQUIRED        = 0x0010   // ZK proofs required on every transfer
TFR_ANCHOR_REQUIRED = 0x0020   // Transfers must carry a TFR anchor
```

Together with `allowed_spk_families`, `kyc_flags` and `tfr_flags`, they are hashed into a single 32-byte `core_policy_commit`:

```
core_policy_commit = ComputeCorePolicyCommit(
    allowed_spk_families,
    policy_bits & POLICY_BITS_IMMUTABLE_MASK,
    kyc_flags,
    tfr_flags)
```

Once issued supply exists, the chain enforces on every ICU rotation that this commitment is byte-identical to the previous ICU's. An attempt to flip an immutable bit after issuance results in a hard rejection (`src/assets/asset.cpp:452–471`). The intent is clear: a holder who acquired *"a non-burnable asset"* will never wake up to find that, by issuer governance, their tokens have become burnable.

ii. The mutable surface

The implementation currently has two mutable paths. ZK/compliance parameters can rotate without holder quorum when they are not part of the immutable core policy, while ICU governance fields are quorum-gated once issued supply exists.

- `compliance_root_commit` — rotated whenever the issuer onboards a new cohort of holders;
- `max_root_age` — the freshness window for ZK proofs;
- `zk_vk_commitment` — the verification-key commitment, where allowed by policy;
- `icu_ctxt_commit` and `icu_plain_commit` — the issuer can publish updated contract text by uploading a fresh ciphertext;
- `icu_visibility` — public ⇄ holder-only flip;
- `policy_epoch` — a wallet-advisory counter; RPC increments it and registry history snapshots are taken when it increases;
- `policy_quorum_bps` — the quorum requirement, expressed in basis points (0–10000).

When `policy_quorum_bps == 0`, the ICU policy is immutable after issuance. When it is non-zero, quorum-gated rotations are valid only if enough holders have signed ballot transactions referencing the proposal.

iii. Ballot binding via `proposal_hash`

A holder casts a vote by spending one of their `AssetTag`-bearing UTXOs in a transaction whose new `AssetTag` carries the optional `proposal_hash` sub-TLV (`src/assets/asset.cpp:242–248`). The function `BuildAssetTagWithProposal()` constructs such a tag (`src/assets/asset.h:242–248`). Because `proposal_hash` is inside `vExt` and `vExt` is inside the sighash, the holder's signature is a cryptographic commitment to the specific ICU rotation proposal. The chain validates matched ballot self-bounces and counts the paired asset units bound to the proposal when deciding whether the rotation has met quorum.

This produces a clean, on-chain governance audit trail: anybody can re-run the consensus tally from the ballot transactions, even if proposal discovery and ballot transport used an off-chain bulletin board.

iv. Worked example: mutating ICU text under quorum

Assume an asset has 10,000,000 settled units and `policy_quorum_bps = 6000`. The current ICU says the issuer contract is public, carries `policy_epoch = 4`, and has accumulated 1.2 BTC of fees toward a 5.1 BTC unlock target. The issuer wants to publish revised public terms and raise the issuance cap. This is a governance mutation, not a unilateral metadata write.

1. **Issuer prepares the rotation.** The issuer wallet calls `prepare_rotation(asset_id, options)`. The template transaction has input 0 equal to the current ICU outpoint and output 0 equal to the proposed replacement ICU. Output 0 carries the new `IssuerReg`: same immutable core policy, incremented `policy_epoch`, updated cap, updated `icu_plain_commit` / `icu_ctxt_commit`, and an unlock target bumped by at least 0.5 BTC when the ICU payload changes. The transaction version sets the rotation flag. The proposal hash is:

   ```
   SHA256("ROTATION/PROPOSAL" || current_icu_outpoint || new_issuer_reg_vExt)
   ```

   This binds the vote to the exact ICU being consumed and to the exact replacement policy (`src/primitives/transaction.cpp:229–241`).
2. **Holders vote with self-bounces.** Each holder chooses one or more asset UTXOs and calls `ballot(template_psbt, asset_utxos)`. For every voted UTXO, the wallet inserts a matching input and a same-index output returning the same asset id, same units, same native value, and same script back to the holder. The output's `AssetTag` adds `proposal_hash`. The holder signs that input with `SIGHASH_ANYONECANPAY | SIGHASH_SINGLE`, so parallel ballots can be merged while each voter still signs the paired self-bounce containing the proposal hash.
3. **Issuer merges ballots.** The issuer calls `validate_ballot` and `merge_rotation`. The merge step rejects duplicate ballot inputs, mismatched proposal hashes, non-self-bounces, mixed asset ids, and any non-zero asset delta. For this example, quorum is `ceil(6000 * 10,000,000 / 10000) = 6,000,000` units, so the merged PSBT must contain at least 6,000,000 units worth of valid ballot self-bounces.
4. **Issuer finalizes the on-chain mutation.** The issuer calls `finalize_rotation`. The wallet adds native-coin fee inputs, signs the ICU input with `SIGHASH_ALL`, optionally appends the `ICU_TEXT_CHUNK`, and broadcasts. A valid mined transaction consumes the old ICU, creates exactly one new ICU, returns each ballot UTXO to its holder, pays the full vbyte fee, increments the asset fee accumulator by the paid native-coin fee, stores the new ICU registry entry, and journals the previous entry for reorg undo.
5. **Consensus enforces the result.** During block connection, validators require exactly one replacement ICU for the same asset, preserve the locked bond floor unless the fee accumulator has reached the unlock threshold, reject any immutable-core change, require zero asset delta for the governance transaction, and count paired ballot units against the previous quorum. If `policy_quorum_bps == 0`, the post-issuance mutable governance path is closed: the asset is immutable except for non-governed operational rotations allowed by policy.

VIII. KYC: Zero-Knowledge Compliance

Some assets need to enforce "only verified holders may transfer". TensorCash supports this without ever publishing a holder's identity, by carrying Groth16 proofs in `vExt` and binding them to the spend at consensus.

The KYC layer is the most novel piece of the protocol, and the property that makes it usable in practice is hierarchical: a holder undergoes KYC **once**, enrolling a single master public key into the issuer's compliance Merkle tree, and from that one enrolment can derive an unbounded number of fresh Taproot addresses, each of which can transact independently against the same compliance commitment. The chain never has to be told about new addresses, the issuer never has to re-onboard the holder, and an external observer cannot link two of the holder's addresses back to the same enrolled identity. We call this the *register-once, trade-many* property, and the rest of this section is mostly about how the circuit and the consensus layer cooperate to deliver it.

i. What the issuer commits to

A KYC-gated asset's `IssuerReg` carries:

- `kyc_flags` — the policy bitfield (currently the gate is `KYC_REQUIRED = 0x10`);
- `zk_vk_commitment` — the SHA-256 of the Groth16 verification key bytes; zero is only valid for non-KYC assets, and KYC-gated registration or transfer rejects a missing VK commitment;
- `max_root_age` — the maximum number of blocks between a compliance root being committed by the issuer and a proof being mined;
- `compliance_root_commit` — the root of the issuer's KYC Merkle tree of approved master identities.

The verification key itself is too big to fit in a single TLV (typically ~3.5 KiB), so it is published as a series of `ZK_PARAMS_CHUNK` TLVs, each at most 512 bytes, totalling at most 4 KiB. The chain reassembles the chunks, hashes them, and verifies that the result equals `zk_vk_commitment`.

The
leaves
of the Merkle tree are deliberately minimal. Each leaf is

$$\text{leaf} = \text{MiMC}(P_x,\ P_y)$$

where (P_x, P_y) is the holder's enrolled secp256k1 master public key, encoded as two field elements (both coordinates are bound to disambiguate P from $-P$). What the leaf does *not* contain is just as important: there is no country code, no age, no risk score, no sanctions flag, no document type. Earlier iterations of the circuit had embedded such predicates as in-circuit constraints (`country == 840`, `age >= 18`, etc.); the current circuit removed them on purpose. Encoding a jurisdictional rule into the proving system makes it impossible to operate the same circuit across regulators with different rules; pushing it out into *tree-inclusion policy* — the issuer accepts or rejects an enrolee off-chain on whatever criteria they like, and only adds approved leaves — keeps the circuit universal across all issuers and jurisdictions while leaving every off-chain compliance decision firmly with the issuer (and that issuer's regulator).

ii. What the holder proves

For every transfer of a KYC-gated asset, the holder includes one `ZK_PROOF_PAYLOAD` TLV per `asset_id`:

- 32 bytes of `asset_id` (binds the proof to *this* asset, defeating cross-asset reuse);
- 192 bytes of compressed BLS12-381 Groth16 proof, structured as `A‖B‖C` of sizes 48+96+48;
- a public-input vector of either 4 or 6 field elements.

The legacy 4-input layout is

Index	Meaning
0	chain separator (prevents cross-chain replay)
1	`asset_id` (big-endian)
2	`compliance_root ∥ height` (28 + 4 packed bytes)
3	`tfr_commit` (or zero)

The HDv1 6-input layout adds, after the four legacy fields:

Index	Meaning
4	`child_key_high` — upper 128 bits of the spent Taproot prevout's *child* x-only key, left-padded to 32 bytes
5	`child_key_low` — lower 128 bits of that child key, left-padded

The split into two halves is structural rather than cryptographic: a single secp256k1 x-coordinate is 256 bits, but the BLS12-381 scalar field Fr is only ~255 bits, so about 26% of x-coordinates would not fit in one field element. Splitting at the byte boundary into two 128-bit halves embeds any x-coordinate cleanly into two field elements with zero ambiguity and zero modular reduction.

What is bound here is the **child** key, not the master. The proof's secret witness contains the master public key, the Merkle path proving its leaf is in the cohort root, and the derivation parameters (path, salt, derivation commitment) that link the child to the master; the proof's public output is just the two halves of the child x-only key (plus the asset, chain, and root context). Consensus, in turn, reads the child x-only key directly out of the prevout's `scriptPubKey` (Taproot witness program bytes 2..34) and compares it byte-for-byte against `public_inputs[4]‖[5]` (`src/consensus/tx_verify.cpp:844–890`). Crucially this comparison is a raw byte equality check — there is no MiMC at consensus, no `dlopen`, no circuit code on the validation hot path; the heavy hash lives in the prover and the validator just confirms the prover committed to the right key.

The output-key binding closes the obvious splice attack — a Groth16 proof by itself says only *"someone in the cohort authorised a spend"*, so without binding a relay could detach the proof and re-attach it to a transaction with a different recipient. By forcing the proof to commit to the spent prevout's child key, *and* by forcing the spending input to be a Taproot v1 prevout whose witness program *is* that child key, the chain ensures a verified proof can only land in the transaction it was minted for. For multi-input same-asset spends the consensus layer additionally requires *every* input for that asset to share the same child key — a single proof covers a single child, and mixing children of the same master into one input set requires one proof per child.

The proof rides in the TLV layer; the witness stack stays standard (signature + pubkey). The output-binding sighash already commits to `vExt`, so the proof is ratified by the same signature that authorises the spend.

iii. Register once, trade many: HD enrolment

The mechanism that turns a single enrolment into an unbounded address set is a Merkle-friendly variant of BIP-32-style hierarchical derivation. We work it out from scratch because the on-chain footprint and the privacy properties only make sense if the reader has the full picture.

Enrolment (off-chain, once per holder). The holder generates a secp256k1 master keypair (s, P) where $P = s \cdot G$. They submit only the public key $P = (P_x, P_y)$ to the issuer. The issuer (or an issuer-trusted KYC oracle such as a regulated KYC vendor) runs whatever off-chain checks the issuer's compliance regime requires — sanctions screening, document verification, jurisdiction filters, age gating — and on approval

inserts a new leaf $\mathrm{MiMC}(P_x, P_y)$ into the cohort Merkle tree. The Merkle root is then committed on-chain by an ICU rotation that updates `compliance_root_commit`. Nothing about the holder's identity, jurisdiction, or attribute set is on-chain or in the proof; the *fact* of inclusion is the only thing the chain ever sees.

Derivation (off-chain, per address). When the holder needs a fresh address, they pick a derivation path — a triple `(account, change, index)` analogous to BIP-32 — and a per-address salt, and compute

$h = \mathrm{MiMC}(\text{"KYCHDV1"} \,\|\, P_x \,\|\, \text{path} \,\|\, \text{salt}) \pmod{n}$

where n is the secp256k1 group order. The child keypair is the standard scalar-additive derivation:

$Q = P + h \cdot G, \qquad q = s + h \pmod{n}$

so $Q = q \cdot G$ — the holder controls Q via q, the same way they would control any other secp256k1 key. The Taproot scriptPubKey for the new address is the trivial `OP_1 OP_PUSHBYTES_32 <Q_x>` (a `rawtr(<x-only>)` descriptor in Bitcoin Core's wallet language), so the chain sees nothing more exotic than a P2TR output.

Spending proof (per spend). When the holder spends an asset UTXO sitting on Q, they generate a Groth16 proof whose public inputs are the chain separator, the asset id, the Merkle root they want to prove against, the TFR anchor, and the two halves of Q_x. The private witness is (P_x, P_y), the eight Merkle siblings, the `(account, change, index)` triple, the salt, and the derivation commitment; the circuit verifies internally that

1. $\mathrm{MiMC}(P_x, P_y)$ is in the tree under the public root,
2. the derivation commitment is consistent with $\mathrm{MiMC}(\text{"KYCHDV1"} \,\|\, P_x \,\|\, \text{path} \,\|\, \text{salt})$,
3. the resulting child x-coordinate equals the public-input pair $(\text{key\_high}, \text{key\_low})$.

Note that the master *secret* s never enters the prover. The circuit proves derivation of a public key from a public key, not knowledge of any private key; ownership of Q is enforced by the ordinary Taproot `OP_CHECKSIG` on the spending input, not by the proof. This is the property that makes server-side proving safe: a prover-as-a-service can mint compliance proofs for any spend a holder wants, with no master-key custody risk, because there is no master key in the request.

Consensus check (per spend). The validator does a fixed-cost Groth16 pairing over the proof and verification key, then byte-compares `public_inputs[4]‖[5]` against the prevout's witness program (and `public_inputs[0..3]` against the chain separator, asset id, an entry of `compliance_root_history` no older than `max_root_age` blocks, and the on-chain TFR anchor commit). All of that fits comfortably inside the per-transaction proof cap of `MAX_ZK_PROOFS_PER_TX = 2`.

The end result is the property the section's title claims: one KYC enrolment, one Merkle leaf, one entry in `compliance_root_history`, and after that, an arbitrary number of independently spendable Taproot addresses each producing its own compliance proof against the same root.

iv. Privacy properties

Because $h = \mathrm{MiMC}(\ldots)$ is unknown to anyone but the holder, the relationship $Q = P + h \cdot G$ is one-way under the discrete-log assumption: given Q, an external observer cannot recover P. The issuer is no exception. After the enrolment moment the issuer knows P and the off-chain attributes; after derivation the issuer sees the on-chain Q only as an arbitrary Taproot address indistinguishable from any other Taproot address. Concretely:

- **The chain learns:** the child key Q_x (because it is the spend's witness program), the asset id, the cohort root, the proof and verifier (so anyone can check the proof). Nothing about P, the path, the salt, or the leaf hash is on-chain.

- **The issuer learns at enrolment:** P, the off-chain attributes the issuer's regime requires (jurisdiction, attestation provider, AML score, etc.). Nothing about future Q values.
- **The issuer cannot link** two on-chain Q values to the same enrolment: under DLP, the map $(P, \text{path}, \text{salt}) \mapsto Q$ is one-way, and the issuer knows neither path nor salt.
- **Two children of the same master are unlinkable** to each other on-chain. They share no public x-coordinate algebraic relation that an outsider can detect.
- **Forward privacy.** Even if the master secret leaks at some later date, past spends remain unlinkable from on-chain data alone; recovering the linkage would still require knowing the per-address h, which is not published.

The trade-off is that the holder's wallet (or whatever service holds their salt/path bookkeeping) is the single point that ties their addresses together; that bookkeeping is by construction off-chain and is the holder's own concern, not the chain's.

v. Cohort rotation and hand-off

Cohorts are not static. New holders are enrolled, existing holders are revoked, the issuer's regulator updates a sanctions list, a holder's KYC document expires. The protocol accommodates all of this through ICU rotation of `compliance_root_commit` (see Issuer Control Units). On every such rotation, the new root is appended to the on-chain `compliance_root_history`. The validator accepts a proof that references *any* historical root within `max_root_age` of the spending block height; this matters for two reasons:

1. **Proofs in flight survive rotation.** A holder who generated a proof against root R_t and broadcast it just as the issuer rotated to R_{t+1} is not invalidated; their proof continues to verify until $\text{tip} - \text{activation\_height}(R_t) > \text{max\_root\_age}$
2. **Revocation has a finite latency.** A holder whose enrolment is removed from the latest cohort can still spend until the older roots that included them age out of the freshness window. This is a deliberate operational property: regulators that require revocation-on-event accept it through `max_root_age` being short (a day, an hour) rather than through immediate-on-chain key burn, which would be cryptographically expensive and operationally fragile.

In practice an issuer publishes a new root whenever its cohort changes, and `max_root_age` is the parameter that trades grace-period length against revocation responsiveness. Both are immutable per asset only in their *commitment structure* — the value of `max_root_age` itself is part of the mutable governance surface, so a regulator-driven tightening of the revocation window is a normal ICU rotation, not a fork.

vi. Trust model and what stays off-chain

Three actors are involved in any KYC-gated asset, and it is worth being explicit about which trust assumption sits where.

- **The issuer** holds the ICU and signs every rotation. They commit to the cohort root, the verification key, and the freshness window. They never hold the holder's master secret.
- **The KYC oracle** (which may be the issuer themselves, a third-party KYC vendor, or an operator-run gateway) runs the off-chain attribute checks that decide who may be enrolled. The protocol does not name or trust any specific oracle; the issuer is free to be their own oracle, to outsource it, or to operate a multi-oracle setup. What the chain cares about is the cohort root, not how it was assembled.
- **The prover** transforms a holder's intent-to-spend into a 192-byte Groth16 proof. Because the witness is pubkey-only, the prover sees no secret material, and a hosted prover (running over standard TLS,

with a ~266 MB proving key in memory) is safe by default. Privacy-conscious holders can run the prover locally; the same artefact is produced either way.

The clean separation between enrolment (oracle), commitment (issuer ICU), derivation (holder wallet), proving (anywhere), and verification (every consensus node) is what makes the system regulator-friendly without making it issuer-custodial. Each role can be operated by a different party, audited independently, and rotated independently of the others.

vii. Freshness

Compliance roots rotate as cohorts change. Two enforcement modes coexist:

- **Legacy** (4-input circuit): the root is packed with its capture height, and the policy verifier enforces freshness against `max_root_age`.
- **HDv1** (6-input circuit): the public input carries a pure 32-byte MiMC root, and the chain maintains a `compliance_root_history` of the last (up to 32) roots with their activation heights. A proof verifies if it references any historical root within `max_root_age` of the spending height. This lets the issuer rotate roots without invalidating proofs in flight.

viii. Caps

To bound verification cost:

- `MAX_ZK_PROOFS_PER_TX = 2`
- `MAX_ZK_PROOFS_PER_BLOCK = 400` currently caps per-block VK installs, not aggregate transfer proofs.

A single transaction may transfer more than two assets, but only two of them may be KYC-gated under the current per-transaction proof count.

IX. ICU Encrypted Payloads and Holder-Only Visibility

So far the protocol has dealt only with arithmetic: how many tokens exist, who can mint them, who is allowed to receive them. But a real-world security — a corporate share, a bond, a regulated stablecoin, a tokenised real-estate fraction — is not just a number. It is a number

plus a contract

: a body of legal text that explains what the number entitles its holder to, under which jurisdiction, against which obligor, and with which restrictions.

The ICU's text payload is where TensorCash carries that contract. Every ICU output may be associated with an off-chain document, attached to it through an `ICU_TEXT_CHUNK` TLV (type 0x30) on a transaction output and committed to inside the `IssuerReg` itself. The chain enforces that the published bytes hash to the issuer's commitment; everything else — content, language, format, signatures — is left to the issuer and to standards from the off-chain world.

This section explains what that payload is for, how it is structured to preserve the legal weight of its content, and how it accommodates Qualified Electronic Signatures (QES, in the sense of EU regulation 910/2014

eIDAS

) and other forensically robust signing modes without having to put a megabyte of certificate chains on-chain.

i. What the ICU text represents

Concretely, the canonical text is meant to carry whatever document an issuer would otherwise have published as a PDF appendix to a token launch, plus whatever holder-specific addenda that token's regulatory regime requires. Typical examples:

- **Issuance prospectus** — the offering memorandum: identity of the issuer, governing law (e.g. *"laws of England and Wales"*, *"Codice Civile italiano, art. 2421 ss."*), describe-and-name of the underlying right, valuation methodology, redemption rules, and any regulator-required disclosures (MiCA, Reg D, Regulation S, MAR, etc.).
- **Articles of association / company bylaws** — for an asset that tokenises an equity stake, the relevant company-law instrument and the board resolution authorising the issuance (*"Resolution of the Board of Directors of TensorCash Holdings S.r.l. dated 12 March 2026 authorising the issuance of up to 10,000,000 Class B Tokens..."*).
- **Custody and transfer restrictions** — lock-up schedules, vesting cliffs, accredited-investor warranties, transfer-restriction legends (*"This security has not been registered under the Securities Act of 1933..."*), tag-along/drag-along clauses.
- **Voting and economic rights** — dividend / coupon mechanics, governance weight (which proposal classes the token can vote on), waterfall treatment in liquidation, conversion features.
- **Regulatory disclosures** — KYC/AML policy text, sanctions-screening commitments, the legal entity identifier (LEI) of the obligor, the address and competent supervisory authority, the procedure for investor complaints.
- **Holder-specific addenda** — for a holder-only payload, a per-holder side letter: subscription agreement, custody agreement with the holder's prime broker, withholding-tax election, beneficial-owner declaration.

The protocol is deliberately content-agnostic. The chain does not parse a prospectus; it commits to its hash. What this gives an issuer is not legal advice — it is a

cryptographic anchor

for the legal document, so that two years later, in a courtroom, the issuer can demonstrate that the document the holder claims to have agreed to is bit-for-bit the document committed to at the moment of issuance, signed by the same authority, and visible on a public ledger.

ii. The canonical / witness split

The RPC-layer canonical structure is `CanonicalIcuPayload` (`src/assets/icu_payload.h:56–131`):

```
struct CanonicalIcuPayload {
    uint8_t version;                                // = 1
    uint8_t compression;                            // 0 = none, 1 = zstd
    uint8_t encryption_mode;                        // 0 = plain, 1 = ChaCha20-Poly1305, ...
    uint8_t visibility;                             // 0 = public, 1 = holder_only
    std::vector<unsigned char> canonical_text;      // UTF-8, normalised
    std::vector<unsigned char> witness_bundle;      // arbitrary signature evidence
    std::vector<unsigned char> metadata;            // reserved
};
```

Two independent hashes are exposed:

$$\text{canonical\_hash} = \text{SHA256}(\text{canonical\_text}) \qquad \text{witness\_hash} = \text{SHA256}(\text{witness\_bundle})$$

RPC builders set the `IssuerReg` field `icu_plain_commit` to this `canonical_hash`. Consensus stores that value and checks the raw ICU ciphertext hash, but it does not parse opaque payload bytes to prove that `icu_plain_commit` equals a decoded canonical text hash.

This separation is not cosmetic; it is the load-bearing design choice for everything legal that follows.

The `canonical_text` is the part that requires a deterministic, byte-stable representation. RPC tooling normalises it before hashing (`NormalizeCanonicalText`, `src/assets/icu_payload.h:206`): UTF-8 is validated, control and non-character code points are rejected, lone `LF` line endings are promoted to canonical `CRLF`, and trailing ASCII spaces and tabs are trimmed. This is the same kind of normalisation a registry of deeds applies to a document before notarisation — what you sign is what is in the cabinet, byte for byte.

The `witness_bundle`, by contrast, is *deliberately* opaque to the chain. It is a JSON or binary container that may carry any number of signature artefacts, each of which references `canonical_hash`. RPC and audit tooling can call `VerifyWitnessLinkage` (`src/assets/icu_payload.cpp:191–215`) to check that the bundle contains the `canonical_hash` somewhere — as a JSON field `"canonical_hash"`, or as an embedded hex string — so that any reader can verify, without parsing the witness's internal format, that the witness in fact certifies *this* contract and not some other one. That linkage helper is not part of block consensus.

The witness bundle is an open extension point. Today, real-world deployments populate it with combinations of:

- **DocuSign** envelope IDs, completion certificates, and full audit trails;
- **OpenPGP** detached signatures from issuer keys with on-chain key-server pinning;
- **Schnorr** signatures over `canonical_hash` from the same Taproot keys that hold the ICU bond;
- **CAdES** / **PAdES** / **XAdES** envelopes for QES-grade signatures, including their qualified certificate chain, OCSP responses, CRLs, and qualified timestamp tokens;
- **Aruba**, **Infocert**, **Namirial**, and other qualified-trust-service-provider (QTSP) signature receipts, where the QTSP itself attests that a natural person signed under their qualified certificate;
- **eIDAS Qualified Timestamp Tokens** (RFC 3161, with QTSP-anchored trust) as standalone evidence of the moment at which the canonical text existed;
- multi-party CCP-style co-signatures, where a clearing-house attests that *both* counterparties signed the same `canonical_hash`.

iii. Why the split makes QES and similar signing modes cheap

A Qualified Electronic Signature under eIDAS is legally equivalent to a handwritten signature; it shifts the burden of proof to the alleged signatory. To produce one in the EU, the signer uses a qualified signature creation device (QSCD) under a qualified certificate issued by a QTSP. The output is, structurally, a CMS or PDF envelope that contains:

1. the **document hash** that was signed,
2. the **qualified certificate** of the signer (and its issuing chain up to the EU Trusted List),
3. **revocation evidence** (CRLs and/or OCSP responses) at the moment of signing,
4. a **qualified timestamp token** anchoring the signature in time,
5. and the **signature value** itself.

All of that is bulky — typical QES envelopes are tens of kilobytes once the certificate chain and timestamp evidence are embedded. None of it is the contract text. All of it references the contract only through a hash.

If the protocol had committed to a single monolithic blob — one `icu_commit` over *"contract + signatures"* — every QES signing event would force a republication of the contract text, would invalidate previous signatures, and would multiply storage. Worse, no holder could be added later: the document a new holder signs is, by construction, a *different* blob, with a *different* hash, and would not match the issuer's earlier on-chain commitment.

The canonical / witness split sidesteps all of that:

1. **The contract can be signed once and reused.** The off-chain/RPC convention keeps `canonical_text` stable; its hash, `canonical_hash = icu_plain_commit`, is committed inside the `IssuerReg`. Consensus anchors the raw ciphertext hash and stored plain commitment, while quorum-gated ICU rotations can publish new payload commitments when policy allows.
2. **Signatures accumulate.** Each new signer (issuer's authorised director, regulator's approving officer, each individual holder, a QTSP attesting to a holder's identity) computes their QES envelope over `canonical_hash`. The envelope is dropped into `witness_bundle`. RPC/storage records the resulting `witness_hash` when payload metadata exposes it; consensus accepts rotations that satisfy the ICU hash, bond, and quorum rules.
3. **Heterogeneous evidence coexists.** The same `witness_bundle` can carry a Taproot Schnorr signature, an OpenPGP signature, a DocuSign certificate, *and* a CAdES QES envelope — all four certifying the same `canonical_hash`. A counterparty's compliance team can satisfy itself with whichever of those evidence types its internal procedures recognise as authoritative.
4. **The QES envelope stays self-contained.** Because the QES standard envelopes already carry their own hash of the document, their own certificate chain, their own timestamp, and their own legal weight, the chain does not need to understand them. Auditor and RPC tooling can use `VerifyWitnessLinkage` to check that they reference the right `canonical_hash`. The forensic value of the QES is preserved end-to-end: the same bytes a court would inspect off-chain are the bytes embedded on-chain.
5. **Adding a signer never invalidates earlier signatures.** This is the actual invariant the split delivers, and it is more subtle than caching. Because `canonical_hash = SHA256(canonical\_text)` is computed only over `canonical_text` and is *not* a function of `witness_bundle`, every QES envelope that was made over `canonical_hash` continues to verify when the issuer adds a new witness — the document hash those envelopes signed is unchanged. What does change is the encrypted blob itself: the implementation serialises the full `CanonicalIcuPayload` (`canonical_text` *and* `witness_bundle`) and encrypts that as one buffer (`src/assets/icu_payload.cpp:289–342`), so any change to the witness bundle changes the ciphertext and therefore `icu_ctxt_commit`. Adding a witness requires publishing a fresh `ICU_TEXT_CHUNK` and rotating the ICU with a new `icu_ctxt_commit`; what it does *not* require is re-signing, re-notarising, or re-asserting the contract text. The legal anchor — the thing earlier signers committed to — is stable across the entire signature-accumulation process.

In short: `canonical_text` is what the law says; `witness_bundle` is who agrees to it. The chain anchors the ciphertext hash on every rotation, the canonical hash inside the `IssuerReg` stays put across signer additions, and the legal weight of every QES (or other witness) is preserved end-to-end while the on-chain footprint stays minimal.

iv. Visibility and encryption

Two visibility modes are supported through the `icu_visibility` field of the `IssuerReg`:

- **Public (`visibility = 0`).** The KDF salt is zeroed, and the published `ICU_TEXT_CHUNK` is plaintext or compressed plaintext under the RPC builder's public mode. Any explorer, any regulator, any holder can read the contract text and replay the witness verification. This is the right mode for prospectuses of public offerings, for stablecoin terms of service, and for tokens that are required by their regulator to publish their full legal package.
- **Holder-only (`visibility = 1`).** The KDF salt is 16 bytes derived deterministically from the data-encryption-key and `icu_plain_commit` (so that the same contract under the same key produces the same ciphertext, which is important to keep `icu_ctxt_commit` stable across re-publications), and the symmetric key is delivered to each individual holder out-of-band, wrapped under the holder's own key material via the `ICU_KEYWRAP` sub-TLV. A non-holder node sees only the ciphertext; it can verify that `SHA256(\text{ciphertext}) = \text{icu\_ctxt\_commit}`, but it cannot read the contract. This is the right mode for per-holder side letters, custody agreements, qualified-investor declarations, and any document whose contents the issuer would not put in a public press release.

The default symmetric primitive is **ChaCha20-Poly1305** AEAD (`src/assets/icu_payload.cpp:458–516`); XChaCha20-Poly1305 is reserved as `encryption_mode = 2`. Because nonce material is derived deterministically from the DEK and `icu_plain_commit`, identical canonical payloads under the same DEK reproduce the same ciphertext and therefore the same `icu_ctxt_commit`. Witness-only changes are exposed through payload metadata and off-chain tooling; consensus still sees the rotated raw payload commitment.

v. The keywrap TLV

For holder-only payloads, the symmetric key is delivered through the `ICU_KEYWRAP` sub-TLV (type 0x31), which carries:

- `asset_id`, `ctxt_hash`, `spk_hash32` — binding the key to *this* asset, *this* ciphertext, and *this* recipient;
- `wrapped_key` — a UTF-8 armoured wrapping (≤ 512 bytes) of the symmetric key under the recipient's public material;
- `suite_id` — the wrap suite identifier;
- `extras_mask` — a bitfield enabling optional fields:
  - `ICU_KEYWRAP_EXTRA_WRAP_COMMIT (0x01)` adds a 32-byte `wrap_commit` allowing third-party verification of correct wrapping;
  - `ICU_KEYWRAP_EXTRA_KC_TAG (0x02)` adds a 16-byte authenticated tag, enabling key-confirmation protocols where the issuer can prove to a regulator that the key wrapped to a specific holder is in fact the key the regulator expects.

The keywrap is the technical mechanism behind a perhaps-surprising property:
the chain itself need not know who the holders of a holder-only asset are

. The issuer wraps each holder's key off-chain, publishes the wrapped material on-chain (so that key delivery is auditable and resistant to censorship), and the holder decrypts at home. From the chain's point of view, every holder-only ICU is just a series of ciphertexts and wrapped keys whose hashes match commitments; the holder's identity is what the ZK compliance proof attests to, not what the ICU encrypts.

vi. Where the ICU text layer meets the KYC layer

It is worth being explicit about something the previous two sections leave implicit, because the relationship between the ICU text/witness layer and the ZK KYC layer is the part of the design most often misread.

At the consensus layer, they do not meet. The Groth16 verifier described in KYC takes its inputs from `vk_commitment`, `compliance_root_commit`, `tfr_commit`, the `asset_id`, the chain separator, and (for HDv1) the recipient Taproot key. It never reads `icu_plain_commit`, `icu_ctxt_commit`, or any byte of `canonical_text` or `witness_bundle`. Conversely, `VerifyWitnessLinkage` (`src/assets/icu_payload.cpp:191–215`), which is the only validator that ever inspects a witness bundle, lives in the RPC/storage layer, runs only on demand, and never reads the compliance root. The two are deliberately decoupled at the bottom of the stack: the ZK layer enforces *who is allowed to spend* in machine-checkable, identity-preserving fashion, and the ICU layer carries *the legal terms under which that spending takes place* in human-readable, court-admissible fashion. Each is simple in isolation; coupling them at consensus would compound the verification cost and tie the two failure modes together.

At the operational layer, they fit together cleanly. A regulated asset typically uses both, in complementary roles:

- The `compliance_root_commit` is the issuer's *operational* attestation: it answers *"is this holder currently in our approved set?"* with a 32-byte root and a 192-byte proof, with no off-chain document required at spend time.
- The ICU's `canonical_text` is the issuer's *contractual* attestation: it answers *"what regime did the issuer declare governs this set?"* — the AML/KYC policy text, the regulator the issuer is supervised by, the standards under which holders were screened, the obligor's identity, the holder rights and remedies.
- The ICU's `witness_bundle` is where the *legal-grade evidence* lives: an issuer-side QES envelope over the AML policy text certifies, with eIDAS legal weight, that the document the chain references is what a named director of the issuer signed off on. For per-holder side letters carried as a holder-only ICU, the `witness_bundle` is the natural home for the holder's own subscription-agreement signature, their beneficial-owner declaration, and any QTSP attestation of the holder's identity.

So the practical pattern is: the ZK proof gates the spend at consensus; the ICU text and its witness bundle make it forensically inspectable. A regulator auditing such an asset has, on a single chain, both the cryptographic record that every transfer was made by an approved holder and the QES-anchored legal paper trail that documents who those approved holders were and under which framework they were approved. Neither piece can substitute for the other, and the design keeps them independent precisely so that each can evolve on its own schedule — the compliance root rotates as cohorts change without forcing a republication of the policy text, and the policy text is amended (with a new QES envelope and an ICU rotation) without invalidating in-flight ZK proofs.

X. Derivatives: Atomic Swaps, Repos, Forwards

Because asset-bearing UTXOs are normal UTXOs from the script's point of view, all of Bitcoin's existing covenant primitives compose with them. TensorCash ships three covenant families on top:

i. Atomic spot swaps

Each side of the swap calls `sendasset(..., return_skeleton=true)` to obtain an unsigned skeleton — a delivery output, change outputs, locked inputs — for their own asset leg. The two skeletons are merged into a single PSBT, both parties sign, the transaction broadcasts atomically. Because every leg is built by its owning wallet, full TLV metadata (KYC proofs, keywraps, anchors) is preserved on each side without either wallet having to know the counterparty's compliance state.

ii. Repos

The repo flow (`src/wallet/rpc/contracts.cpp`) opens an Initial Margin (IM) vault for the borrower's collateral, a separate output for the principal, and uses skeleton-based asset legs wherever the principal or collateral is a token rather than BTC. `repo.build_open` opens the position; `repo.build_repay_release` redeems it; `repo.build_default_sweep` lets the lender claim the collateral after the maturity height has passed.

iii. Forwards

The forward contract is the most elaborate covenant. Each party deposits an IM vault whose taproot tree contains four leaves:

```
Internal Key: NUMS (unspendable)
├─ Leaf 0: Cooperative Close (physical settlement)
|  └─ MuSig2(Long, Short) + OP_OUTPUTMATCH on 4 outputs:
|     [delivery_long→short, delivery_short→long, margin_long, margin_short]
├─ Leaf 1: Reserved for cooperative cash settlement (planned)
├─ Leaf 2: Escrow Refund (timeout)
└─ Leaf 3: IM Vault Timeout Sweep
```

The cooperative leaf uses `OP_OUTPUTMATCH` to force the four output commitments to match the negotiated structure. Each party builds its own delivery leg through `BuildForwardAssetSkeleton`, which delegates to `sendasset`'s skeleton path; the two skeletons are merged with the vault inputs, the adaptor signing ceremony (`adaptor.prepare` / `partial` / `complete`) produces both partials, and the final transaction settles physically with full asset-tag fidelity. Pre-expiry self-delivery and post-expiry escrow claims work analogously through dedicated leaves.

The advantage of this design is that the asset protocol *contributes nothing new* to the covenant layer: every covenant remains pure Bitcoin Script, and the `sendasset` skeleton is just a way to outsource the bookkeeping for one party's leg.

XI. Mempool and Reorg Behaviour

Two operational concerns deserve specific mention.

i. Policy and consensus knobs

The node exposes (`src/init.cpp:642–657`):

Flag	Default	Purpose
`-assetsheight`	configured	consensus activation block height
`-policymaxassetspertx`	64	

Flag	Default	Purpose
		maximum AssetTag outputs per transaction
`-policymaxassetoutsize`	160 bytes	maximum `vExt` size on a single output
`-assetmindustbtc`	family-aware	minimum BTC value on an AssetTag-carrying output
`-assetminmultitouchfee`	0	minimum fee floor when a tx touches ≥ 2 assets

`-policymaxassetspertx`, `-policymaxassetoutsize`, `-assetmindustbtc`, and `-assetminmultitouchfee` are mempool policy: they restrict what the node accepts into its mempool, but they are not block-validity rules. Consensus-critical values are NOT configurable: the minimum bond on an initial `IssuerReg` is the hardwired constant `Consensus::Params::AssetMinIcuBond` (5 BTC, enforced in block validation and mirrored by mempool pre-policy; the former `-assetminicubond` option was removed), and the asset/delegation activation heights are hardwired to genesis on the main and test chains (`-assetsheight` / `-assetsdelegationheight` are honored on regtest only). Consensus also enforces conservation, ICU rotation, sighash, KYC, ticker, and asset activation rules.

ii. Reorg safety

Every consensus mutation of the asset registry — ICU rotations, ticker bindings, fee accumulator increments, `compliance_root_history` appends — is journalled in a per-block undo record (`src/undo.h:33–80`). On a reorg, the registry is rolled back exactly as the UTXO set is, and the post-reorg state is byte-identical to what it would have been had the orphaned chain never existed. The functional test suite (`feature_assets_deep_reorg.py`, `feature_assets_ticker_reorg*.py`, `feature_assets_network_partition.py`) exercises this against multi-block reorgs and partition healing.

XII. Security Model and Attack Surface

We close with a deliberate enumeration of the attacks the protocol is designed to resist.

Splice attacks. Asset state is in the sighash; a signed transaction cannot be edited to redirect tokens.

Amount inflation. The conservation check uses 128-bit accumulators; overflow returns `asset-arith-overflow`. AssetTag amounts of zero are rejected at parse time.

Cross-asset proof reuse. Every `ZK_PROOF_PAYLOAD` includes the `asset_id` as a public input; a proof minted for asset *X* cannot satisfy the verifier for asset *Y*.

Cross-spend proof reuse. HDv1 binds the proof to the Taproot x-only key of the spent asset prevout, so a proof cannot be replayed against a different holder input set.

Cross-chain proof reuse. A chain separator is the first public input.

ICU theft. `SIGHASH_ANYONECANPAY` is forbidden on ICU inputs; the issuer's signature commits to the entire output set, so it cannot be lifted to a different mint.

Coinbase laundering. Coinbase outputs are forbidden from carrying an `AssetTag`; tokens cannot enter circulation through mining. Coinbase `IssuerReg` registration is allowed, but it creates authority, not asset units.

Issuer policy flip. Once issued supply exists, immutable bits are committed by `core_policy_commit`; rotation rejects any change to them.

Bond evaporation. The rotation floor stays at 95% of the inception bond until unlock; the dust floor only applies after the issuer has paid in enough fees. ICU payload or visibility changes must also raise `unlock_fees_sats` by the configured minimum bump.

Ticker squatting and reorg replay. Tickers are immutable once bound; bindings are journalled and undo cleanly under reorg.

DoS via verification. Per-transaction transfer proof cap (`MAX_ZK_PROOFS_PER_TX = 2`) and per-block VK-install cap (`MAX_ZK_PROOFS_PER_BLOCK = 400`); 128 KiB aggregate `vExt` cap per tx; 100 KiB cap per output; 4 KiB cap on a single VK; 100 KiB cap on an ICU payload.

Parser ambiguity. All TLV parsers are deterministic, single-pass, and covered by fuzz targets (`test/fuzz/asset_tlv_parser.cpp`, `asset_transaction.cpp`, `asset_registry.cpp`, `asset_mempool.cpp`, `asset_fee_accumulator.cpp`).

Reorg-induced double-spend. The asset registry uses the same write-undo discipline as the UTXO set; deep-reorg tests cover this path.

The overall security argument is structural: the asset protocol does not introduce a new authority or a new authoritative state, it overlays consensus-enforced records on top of the existing UTXO and signature primitives. Every rule we add is a rule about something the consensus layer already validated; the surface for novel attacks is therefore strictly the surface for novel parsing or novel verification, both of which we have aggressively bounded.

Appendix: Constants, Limits, and Activation

Constant	Value	Source
`TXFLAG_OUTEXT`	0x02	`src/primitives/transaction.h:217`
`MAX_OUEXT_SIZE_PER_OUTPUT`	100 KiB	`src/primitives/transaction.h:221`
`MAX_OUEXT_BYTES_PER_TX`	128 KiB	`src/primitives/transaction.h:222`
`OutExtType::ASSET_TAG`	0x01	`src/assets/asset.h:17`
`OutExtType::ISSUER_REG`	0x10	`src/assets/asset.h:18`
`OutExtType::ZK_PARAMS_CHUNK`	0x20	`src/assets/asset.h:19`
`OutExtType::TFR_ANCHOR`	0x21	`src/assets/asset.h:20`
`OutExtType::ZK_PROOF_PAYLOAD`	0x22	`src/assets/asset.h:21`
`OutExtType::ICU_TEXT_CHUNK`	0x30	`src/assets/asset.h:22`
`OutExtType::ICU_KEYWRAP`	0x31	`src/assets/asset.h:23`
`MINT_ALLOWED`	0x0001	`src/assets/asset.h:218`
`BURN_ALLOWED`	0x0002	`src/assets/asset.h:219`
`BURN_REQUIRE_ICU`	0x0004	`src/assets/asset.h:220`
`BURN_JOINT_REQUIRED`	0x0008	`src/assets/asset.h:221`
`KYC_REQUIRED`	0x0010	`src/assets/asset.h:222`
`TFR_ANCHOR_REQUIRED`	0x0020	`src/assets/asset.h:223`
`WRAP_REQUIRED` (icu_flags)	0x0001	`src/assets/asset.h:231`
`ICU_COMPRESSED` (icu_flags)	0x0002	`src/assets/asset.h:232`
`SPK_DEFAULT_ALLOWED`	P2WPKH ∪ P2WSH ∪ P2TR	`src/assets/asset.h:215`

Constant	Value	Source
SPK_HOLDER_ONLY	P2PKH ∪ P2WPKH	src/assets/asset.h:214
MAX_ZK_CHUNKS	8	src/assets/asset.h:174
MAX_ZK_CHUNK_SIZE	512	src/assets/asset.h:175
MAX_VK_PAYLOAD_SIZE	4 KiB	src/assets/asset.h:176
MAX_TFR_LOCATOR_SIZE	128	src/assets/asset.h:177
MAX_ZK_PROOFS_PER_TX	2	src/assets/asset.h:178
MAX_ZK_PROOFS_PER_BLOCK	400	src/assets/asset.h:179
GROTH16_PROOF_SIZE	192	src/assets/asset.h:182
GROTH16_FR_SIZE	32	src/assets/asset.h:183
GROTH16_MIN_PUBLIC_INPUTS	4	src/assets/asset.h:184
GROTH16_MAX_PUBLIC_INPUTS_SIZE	256	src/assets/asset.h:185
GROTH16_HDV1_PUBLIC_INPUTS	6	src/assets/asset.h:186
MAX_ICU_PAYLOAD_BYTES	100 KiB	src/assets/asset.h:189
MAX_ICU_KEYWRAP_WRAPPED_KEY_BYTES	512	src/assets/asset.h:192
MAX_ICU_KEYWRAP_VEXT_BYTES	700	src/assets/asset.h:193
ICU_KEYWRAP_KC_TAG_SIZE	16	src/assets/asset.h:196
UNLOCK_BUMP_MIN_DEFAULT	50,000,000 sats	src/assets/asset.h:203
ISSUER_REG_FORMAT_V1	1	src/assets/asset.h:202

Despite its name, the current implementation applies `MAX_ZK_PROOFS_PER_BLOCK` to per-block VK installation, while transfer proof verification is bounded by `MAX_ZK_PROOFS_PER_TX`.

The activation height is set per-network via `ChainParams::AssetsHeight`. Below it, `vExt` is forbidden by both consensus and policy. Above it, the protocol is live and a node that has validated the chain has, by induction, validated every asset registry mutation, every mint, every transfer, every burn, every ZK proof, every ICU rotation, and every governance ballot it has seen.

Appendix: Qt Wallet and RPC Surface

TensorCash Qt is not a separate protocol implementation. It is a wallet front-end over the same RPC surface used by scripts, tests, and external tooling. The UI exposes the issuer lifecycle, the holder lifecycle, and the lower-level inspection tools needed to audit what was built.

i. Issuer workflow

The **Asset Issuer** view in the Treasury page ships tabs for registration, minting, burning, ICU inspection, compliance, distribution, and governance. Registration builds the canonical `IssuerReg` through `registerasset`, including ticker, decimals, policy bits, allowed script families, ICU bond value, `unlock_fees_sats`, optional ICU text, holder-only visibility, KYC parameters, issuance cap, and quorum. If KYC is enabled, the UI uses the embedded verification key, can generate an initial compliance root in pre-registration mode, and passes the relevant commitments into the registration RPC.

Ongoing issuer operations map directly to wallet RPCs:

Qt surface	Primary RPCs	Purpose
Register Asset		

Qt surface	Primary RPCs	Purpose
	`registerasset`, `buildcanonicalicupayload`	Create the initial ICU, optional ICU text payload, ticker/decimals, KYC commitments, issuance cap, quorum, and bond threshold.
Mint	`mintasset`, `getassetpolicy`	Spend the current ICU, rotate it, and create new `AssetTag` outputs within policy and cap limits.
Burn	`burnasset`, `listassetutxos`, `getassetpolicy`	Coordinate issuer-authorised burns and rotate the ICU when policy requires issuer participation.
ICU Dashboard	`getassetpolicy`, `geticupayload`, `geticupayload_prior`, `decrypticupayload`, `rotateicu`	Inspect current/prior ICU policy, view public text, decrypt holder-only text with a DEK, and perform direct ICU rotations where quorum is not required.
Compliance	`updatecompliancerоot`, `getassetpolicy`	Build and publish new compliance roots as holder cohorts change.
Distribution	`distributeasset`, `listassetutxos`, `getassetpolicy`	Snapshot asset holders and distribute native coin or another asset according to holder balances.
Governance	`prepare_rotation`, `merge_rotation`, `validate_ballot`, `finalize_rotation` plus bulletin-board RPCs	Prepare quorum-gated rotations, publish or privately share proposal PSBTs, collect signed ballots, merge them, and finalize the on-chain mutation.

ii. Holder workflow

The default Treasury view is **Asset Holder**. It shows the wallet's registered assets, balances, pending units, locked units, policy flags, text visibility, script-family constraints, compliance status, and TFR requirements. The overview page includes an asset-balance widget, and double-clicking an asset opens the holder asset view. The Send dialog includes an asset selector and routes asset sends through `sendasset`, which handles asset UTXO selection, asset change, native-coin fee funding, keywrap preservation, and regulated-transfer proof/keywrap handling where configured.

Holder operations map as follows:

Qt surface	Primary RPCs	Purpose
Asset Balances / My Assets	`listassets`, `getassetbalance`, `listassetutxos`, `getassetpolicy`, `geticupayload`	Inspect balances, policy, issuer text, and current ICU metadata.
Send	`sendasset`	Transfer asset units while paying native-coin fees based on full transaction vsize.
Compliance	`generatezkproof`, `getassetpolicy`	Generate a KYC master public key for issuer enrollment and build HDv1 spend proofs for regulated transfers.
Governance	`ballot`, `listassetutxos`, bulletin-board RPCs	Discover public or private proposals, verify template hashes/attestations, select voting UTXOs, sign proposal-bound self-bounce ballots, and send ballots back to the issuer.
ICU Text Access	`geticupayload`, `decrypticupayload`	Read public ICU text or decrypt holder-only payloads when the holder has the relevant DEK/keywrap.

The raw-transaction RPCs remain available for auditors and external tooling: `rawtxaddoutext`, `rawtxattachissuerreg`, `rawtxattachassettag`, `rawtxattachzkchunk`, `decodeoutext`, `decoderawtransaction`, `decodeassettransaction`, and `validateassetconservation`. The intended separation is that Qt covers ordinary issuer and holder operations, while raw RPCs expose the full TLV surface for testing, recovery, hardware-wallet workflows, and independent review.