ZOOBC (ZBC) · ANNEX TO THE SIGNING SPECIFICATION

Answers to the Integration Questions

Prepared for hardware-wallet host and firmware engineers.

REVISION
Annex 1.1

DATE
10 September 2026

SIGNATURE SCHEME
Ed25519 over SHA3-256

DERIVATION PATH
m/44'/883'/account'

SLIP-44 COIN TYPE
883 | ZBC

VERIFIED AGAINST
ZooBC node v0.4.0 · 7ad161a4

Every statement in this document was checked against ZooBC node v0.4.0 and against transaction vectors a live chain has accepted. Source files are cited so a reviewer can re-check any line.

Contents

0 Answers at a glance

#	YOUR QUESTION	ONE-LINE ANSWER	SPEC
H1	Account based or UTXO-based	**Account-based.** No nonce, no sequence number	§5.4, §A1
H2	Balance API	`GET /api/v1/accounts/<address>` → `spendable_balance`	§9.1, §A2
H3	History API	`GET /api/v1/accounts/<a>/history` plus `/movements`, most recent N via `limit`	§9.1, §A3
H4	Fee API	`GET /api/v1/blockchain/estimate-fee` per transaction; `/fee-params` for constants	§5.1, §A4
H5	Unsigned tx creation	**No endpoint. The host serialises it.** Layout fully specified and vector-pinned	§2, §A5
H6	Broadcast API	`POST /api/v1/transactions` → returns `transaction_hash`. Takes body + named fields, **not the signed blob**	§9.2, §A6
F1	Derivation	BIP-39 → **SLIP-0010 Ed25519**, `m/44'/883'/account'`, 3 hardened levels, base32 + 3-byte SHA3 checksum	§1.3, §1.4, §A7
F2	Byte layout	11 fields, **fixed-width little-endian** with `u32le` length prefixes. No varint, no protobuf	§2, §A8
F3	Signing	`Ed25519.sign(sk, SHA3-256("ZBC-TX" ‖ genesis_hash ‖ unsigned_bytes))`	§3, §A9

0.1 Read these four first

Four points that shape an implementation more than the rest.

1. **There is no nonce, and there is no unsigned-transaction endpoint.** Account-based does not imply either one here. See §A1 and §A5.
2. **The signature covers a tagged prefix**, `"ZBC-TX"` plus the chain's genesis hash, not the transaction bytes alone. Six ASCII bytes `5a 42 43 2d 54 58`, no terminator, no length prefix. Get this wrong and every signature fails with a generic error. See §A9.
3. **The broadcast endpoint does not take the signed transaction blob.** It takes the transaction *body* bytes plus the envelope as named JSON keys and rebuilds the envelope itself. See §A6.
4. **History is returned newest-first, bounded by `limit`.** Two endpoints together give a complete picture, and the amount is type-dependent rather than an envelope field. See §A3.

A1 "Is zoobc account based (nonce/sequence) or UTXO-based"

Account-based. An address holds a balance. There are no unspent outputs to select, no change outputs and no coin control.

The qualification that matters: ZooBC is account-based **without a nonce or sequence number**. The signed bytes contain no counter; §2 lists all eleven fields and none of them is a nonce. Three other mechanisms bound a signature instead.

- **Chain binding.** The chain's genesis block hash is inside the signing digest, so a signature is valid on exactly one chain (§A9). This is new in node v0.4.0.
- **Hash uniqueness.** A transaction is admitted at most once. Re-posting identical bytes is a no-op and answers `duplicate: true` rather than an error.
- **A 3600-second inclusion window.** A block may include a transaction only while `block.timestamp ≤ tx.timestamp + 3600` (`src/core/block_validator.cpp:456`). The mempool drops it after 60 minutes.

So a signature is usable **once**, on **one chain**, and **within an hour**.

Two consequences for the host, both of which follow from the missing nonce rather than from anything exotic:

- **You cannot queue or pipeline transactions from one account.** There is no ordering primitive, so two transactions signed in the same second race. Sign, broadcast and confirm before signing the next one. Batch and retry logic has to be serial.
- **Resubmission is safe but expires.** Re-posting the identical bytes is a no-op; after the hour the transaction must be rebuilt with a new timestamp and re-signed.

Balances are read at tip − 3 (`api_server.cpp:1454-1456`), so a freshly confirmed transfer can take roughly 45 seconds to appear. Do not treat a stale balance as a failed send; poll the status endpoint (§A6).

A2 "Balance API: endpoint/SDK call that returns balance for a given address"

```
GET /api/v1/accounts/<address>
```

The path accepts either form: the `ZBC_…` display address (66 characters, separators optional, any case) or the 72-character hex account. Both resolve to the same account.

FIELD	MEANING
`spendable_balance`	what a transaction may spend now. **This is the one for a send screen.**
`balance`	total, including amounts locked in escrow or node registration
`locked_balance`	the locked portion
`total_balance`	balance plus unclaimed revenue
`pop_revenue`	proof-of-participation revenue
`transact_policy`	the account's 9-bit opt-out mask

All values are **atomic units**, u64. 1 ZBC = 100 000 000 atomic, 8 decimals. Divide by 1e8 for display and never round-trip through a float: §10 vector 6 carries a fee above 2^53 precisely to catch a JSON parser that does.

Token balances are a separate call, because a ZBC balance and a token balance are different units:

```
GET /api/v1/accounts/<a>/tokens      -> tokens[] each with its own decimals
GET /api/v1/tokens/<id>              -> symbol, decimals, supply
```

Always read `decimals` per token. They differ, ZUSD 2, ZBTC 8, ZETH 9 as configured, and assuming 8 misprices the screen by orders of magnitude. The device cannot verify them either, which is why §7.2 has it label host-supplied decimals as reported by the app.

A3 "History API: given an address, returns paginated send/receive transactions, each with: direction, from address, to address, tx hash, fee, amount, timestamp/block height"

Two endpoints, and you want both:

```
GET /api/v1/accounts/<a>/history?limit=<n>      transactions this account sent or received
GET /api/v1/accounts/<a>/movements?limit=<n>    every value movement, including non-transaction credits
```

`history` is the transaction list. `movements` also carries credits that arrive without a transaction naming this account, such as block rewards and escrow releases. A wallet that shows only `history` will display a balance that does not reconcile with the list above it.

Taking your fields in order:

YOU ASKED FOR	FIELD	NOTE
direction	derived	compare `sender_account_address` with the queried account
from address	`sender_account_address`	72-hex; render per §1.4 or §1.5
to address	`recipient_account_address`	may be the empty marker `02000000`, meaning no recipient
tx hash	`transaction_hash`	32-byte SHA3-256, hex. **This is the lookup key, not the numeric id**
fee	`fee`	atomic u64
amount	**in the body**	see below
timestamp	`timestamp`	Unix seconds
block height	`block_height`	present once confirmed

Three things to flag, because two of them will change your implementation.

Amount is type-dependent and is not an envelope field. It lives in the body, whose layout varies per transaction type (§6). For SendZBC the body is the 8-byte amount; for TransferToken it is a token id followed by an amount, in that token's own decimals. A history view must switch on `transaction_type` rather than read a single amount column, and it must not add a token amount to a ZBC fee.

Look transactions up by hash, not by id. `tx_id` is `int64le(tx_hash[0..8])`, a display convenience. `GET /api/v1/transactions/<hash-hex>` is the retrieval path.

`limit` bounds the result, newest first. Size it to the view you are rendering. For a wallet that shows a transaction list this is what you want; `limit` plus the confirmed-read lag in §A1 is the whole contract.

Note that `history` is served by a node or gateway. **An archival endpoint does not carry this route**, so route reads archival → gateway → node and fall back for this path specifically.

A4 "Fee API: endpoint for current network fee / fee rate"

Two endpoints, for two jobs:

```
GET /api/v1/blockchain/fee-params
GET /api/v1/blockchain/estimate-fee?
type=&body_length=&message_length=&escrow=&escrow_instruction_length=&escrow_lock_seconds=
```

`fee-params` returns the constants, `min_fee`, `fee_scale`, `effective_min_fee`, `one_zbc` and the storage rent parameters. Cache it.

`estimate-fee` returns the exact `minimum_fee` for a proposed transaction, and it is the one to call before every send, because there is no flat fee rate. The fee is

```
minimum = 2 500 000 × fee_scale / 1 000 000
        + persistence_rent(body + message bytes)
        + escrow_rent(escrow bytes × lock duration)
```

so a longer message or a longer escrow lock genuinely costs more. On testnet today a plain SendZBC needs 2 500 003 atomic, 0.02500003 ZBC.

Three practical points:

- The response also carries `timestamp`, **the node's own clock**. Use it as the transaction timestamp. The device has no clock, and a timestamp outside the window costs a resubmission. Getting the fee and the clock in one call is why this endpoint sits where it does in §8.2.
- **Overpayment is refunded** for most transaction types under the honest-fee rule, so a small buffer over the quoted minimum is safe.
- Type 0, the Empty message carrier, must still declare a fee greater than zero.

A5 "Unsigned tx creation: if account-based: endpoint/SDK to build an unsigned transaction. If UTXO-based: endpoint to list all UTXOs for an address"

There is no endpoint, and none is needed. ZooBC is account-based, so the UTXO branch of your question does not apply, and the host serialises the transaction itself from the layout in §2, which is fixed, fully documented and pinned by seven chain-validated vectors in §10.

This looks like a gap and is not. For a hardware wallet it is the stronger arrangement: the device receives the bytes, parses every field, re-derives the sender from its own key and refuses if it does not match, and computes the digest itself. Nothing it displays is taken on trust from the host (§7.1). An endpoint returning a pre-built transaction would add a party to trust without removing any work, since the device would still have to parse and verify what came back.

What the host does need from the chain before serialising is three reads, all covered above: the sender's `spendable_balance` (§A2), the token `decimals` if a token is involved (§A2), and the fee plus node clock from `estimate-fee` (§A4). Plus `genesis_hash` and `signing_version` from `/api/v1/node/info`, which is new and is covered in §A9. §8.2 draws the full sequence.

For a serialiser to copy, §11 is the whole algorithm in about 30 lines of Python, standard library plus `cryptography`, reproducing vector 0 end to end. There is a second, byte-identical implementation in JavaScript at `zoobc-signer/extension/lib/zoobc-crypto.js` (`buildTx`), runnable on Node 18 with no build step, whose `npm test` checks all seven vectors. Either works as an oracle.

A6 "Broadcast API: endpoint that accepts a signed tx and returns the resulting tx hash"

```
POST /api/v1/transactions        Content-Type: application/json
```

Yes, it returns the hash, as `transaction_hash`. One clarification on the input: it does **not take a signed transaction blob**. It accepts the transaction *body* bytes together with the envelope fields as named JSON keys, and rebuilds and re-hashes the envelope server-side, prepending its own genesis hash and the tag (`api_server.cpp:764`, fields from `:856`). Every field posted must be identical to what was signed or verification fails.

```
{
  "version": 1,
  "timestamp": 1700000000,
  "transaction_type": 1,
  "fee": 5000000,
  "sender_account_address":    "000000005e8eb28dccaa94a992f7169fa67966207d96aa55df7d24d2d6ec1f538c9f2152",
  "recipient_account_address": "000000005e8eb28dccaa94a992f7169fa67966207d96aa55df7d24d2d6ec1f538c9f2152",
  "transaction_body_bytes": "00e1f50500000000",
  "signature": "9a3b044dca7c2f351e9d2b37dd6924e4260536c496207de87f0e1f1a31dd9f01cd6359e202e13964580a561a89
e26688749fe9cdbf12056ed05eb1fbcd03bb07"
}
```

`transaction_body_bytes` is the body **only**, hex, not the whole transaction. `fee`, `commission` and `timeout` are JSON numbers (int64). Omit optional keys rather than sending them empty. A message goes in `message_hex`, or as text in `message`. An escrow goes in an `escrow` object with `approver_address`, `commission`, `timeout` and `instruction`. The recipient for "no recipient" is the hex string `02000000`.

STATUS	MEANING
202, and treat 200 as success too	`{"status":"success","duplicate":false,"transaction_hash":"…"}`
`duplicate: true`	the node already holds it. Do not resend
400	`{"success":false,"error":"…","code":400}` for signature, fee floor or structure problems
503	load shedding, for example `mempool backpressure`. Retry

Cross-check the hash. Compare the returned `transaction_hash` against the `tx_hash` the device computed. They must be identical; if they differ, the host altered the bytes after signing and the transaction must not be treated as sent.

Then poll:

```
GET /api/v1/transactions/<hash-hex>/status   -> staging | mempool | confirmed (+ block_height) | 404
not_found
```

Broadcast to a gateway or a node, **never to an archival endpoint**.

Two notes that will save time. The staging pool checks the signature first (`transaction_staging_pool.cpp:147`), so a transaction from an unfunded key that comes back `Sender account does not exist` has already passed signature verification. That is a clean way to validate your whole signing path before funding anything. And a rejected signature currently gives one generic message for every cause, so §9.3 of the specification lists the causes in the order worth checking.

A7 "Mnemonic to private key to public key to address: exact derivation (BIP-39/32/44? custom path? curve: ed25519/secp256k1? address encoding/checksum)?"

Taking your sub-questions in order.

Curve: Ed25519 (RFC 8032, pure). Not secp256k1. No prehash variant, no context string.

Standards: BIP-39 for the mnemonic, then SLIP-0010 for Ed25519. BIP-32 does not apply, because Ed25519 has no public child derivation. The path is BIP-44 shaped but shorter than usual.

Path: `m/44'/883'/account'`, three hardened levels, and that is the whole path. There are no change or address-index levels. Account 0 is `m/44'/883'/0'`, account 1 is `m/44'/883'/1'`. Coin type 883 is registered as `883 | ZBC | ZooBC`.

The derivation, exactly:

1. `seed = PBKDF2-HMAC-SHA512(password = NFKD(mnemonic), salt = NFKD("mnemonic" ‖ passphrase), 2048 iterations, 64 bytes)`
2. `I = HMAC-SHA512(key = "ed25519 seed", data = seed)`; `k = I[0..32]`, `c = I[32..64]`
3. For each index in `[44, 883, account]`, all hardened: `data = 0x00 ‖ k ‖ u32be(index + 0x80000000)`; `I = HMAC-SHA512(c, data)`; `k = I[0..32]`, `c = I[32..64]`
4. The final `k` is the 32-byte Ed25519 seed; `pubkey = Ed25519.publicKeyFromSeed(k)`

The child indexes are **big-endian**, per SLIP-0010, while everything on the ZooBC wire is little-endian. That is the only byte-order exception in the whole integration.

Derivation is **not** affected by chain binding. One key set serves every ZooBC chain.

Address encoding and checksum. Two forms exist and you need both.

The *account*, which goes on the wire and into the JSON API, is `u32le(account_type) ‖ payload`. For a ZooBC key that is `00 00 00 00 ‖ pubkey(32)`, 36 bytes, 72 hex characters.

The *display address* is base32 with a 3-byte SHA3 checksum:

1. `buf[35] = pubkey(32) ‖ "ZBC"`, three ASCII bytes
2. `h = SHA3-256(buf)`; overwrite `buf[32..35] = h[0..3]`
3. `s = base32(buf)`, RFC 4648 alphabet `A-Z2-7`, no padding, exactly 56 characters
4. output `"ZBC" + "_" + s[0..8] + "_" + s[8..16] + …`, seven groups, canonical length 66

Decoding accepts underscores, dashes, spaces or no separators in any case, but the body must be exactly 56 base32 characters and the checksum must be recomputed and compared.

The 3-byte SHA3 checksum above is the definition the code and the chain use. Implement from it, and from the reference implementation in §11 of the specification, which reproduces the addresses below.

Test values. BIP-39 test phrase `abandon` × 11 + `about`, no passphrase:

```
account 0   m/44'/883'/0'
seed        51bae95d58f32304a5c9d894819989a4fb04da6115d9a43612a026e7a5dd5d96
pubkey      5e8eb28dccaa94a992f7169fa67966207d96aa55df7d24d2d6ec1f538c9f2152
address     ZBC_L2HLFDOM_VKKKTEXX_C2P2M6LG_EB6ZNKSV_356SJUWW_5QPVHDE7_EFJA3PEX

account 1   m/44'/883'/1'
seed        2c0f8a5722d5701a7a4e7d61219c5235a9b74ba46fdbaff9f7978ec7bc14f85d
pubkey      fd4472bbdec43b919aa37b9b3366a3318f2902a753a15f5fd6d43c44b5121b67
address     ZBC_7VCHFO66_YQ5ZDGVD_PONTGZVD_GGHSSAVH_KOQV6X6W_2Q6EJNIS_DNTVIFT4
```

A8 "Unsigned transaction structure: byte-by-byte layout (field order, sizes, encoding: fixed-width/varint/protobuf/etc.)?"

Encoding: fixed-width little-endian, with `u32le` length prefixes on the variable parts. No varint, no protobuf, no RLP, no ASN.1. Byte order is little-endian everywhere on the wire; the sole exception in the integration is the SLIP-0010 child index (§A7).

Field order and sizes (`src/util/transaction_util.cpp:144-270`, `GetTransactionBytes`):

#	SIZE	FIELD	NOTES
1	4	`transaction_type` u32le	type id = group + 256 × subtype
2	1	`version`	always `0x01`; the node rejects anything else
3	8	`timestamp` u64le	Unix seconds, greater than 0
4	36	`sender`	`00000000` ‖ pubkey(32) for a ZBC signer
5	4, or 4+n	`recipient`	36 bytes for ZBC; `u32le(type)` ‖ payload otherwise; **`02 00 00 00` alone** when there is no recipient
6	8	`fee` u64le	atomic units
7	4	`body_length` u32le	
8	n	`body`	per-type layout
9	4, or var	`escrow`	**no escrow:** the 4 bytes `02 00 00 00`. **With escrow:** the block below
10	4	`message_length` u32le	
11	m	`message`	plain text, ≤ 256 bytes for ordinary transactions

Note that `version` stays `0x01`. It was not bumped for chain binding and it is not a version discriminator; see §A9.

The escrow block, when field 9 carries one (`transaction_util.cpp:961-1000`):

SIZE	FIELD
36	`approver` account
8	`commission` u64le, atomic ZBC paid to the approver
8	`timeout` u64le, an **absolute** Unix timestamp, greater than the tx timestamp
4	`instruction_length` u32le
var	`instruction`, UTF-8
1	`multi_party` flag, `0x00` for a normal escrow
only if `multi_party = 1`	`cosig_len` u32le ‖ cosig ‖ `escrow_request_id` i64le

The trailing `multi_party` byte is mandatory. Omitting it, or writing the no-escrow marker with a different value, yields nothing more diagnostic than `Invalid transaction signature`. §10 vector 2 pins that byte.

A worked example, §10 vector 0: SendZBC of 1 ZBC to self, fee 0.05 ZBC, timestamp 1700000000, no escrow, no message:

```
01000000                type = 1 (SendZBC)
01                      version
00f1536500000000        timestamp 1700000000
00000000 5e8eb28d…2152  sender    (type 0 + pubkey)
00000000 5e8eb28d…2152  recipient (type 0 + pubkey)
404b4c0000000000        fee 5 000 000
08000000                body length 8
00e1f50500000000        body: amount 100 000 000
02000000                no-escrow marker
00000000                message length 0
```

That is 113 bytes.

Bodies for the three recommended types. They are small and fixed, which is what makes a standard-edition firmware practical:

ID	NAME	BODY
1	SendZBC	`amount u64le`, 8 bytes
11	TransferToken	`token_id i64le` (8) ‖ `amount i64le` (8) ‖ optional `fee_in_token u8` (17th byte, `0x01` = pay the fee in this token)
4	ApprovalEscrow	`approval u32le` (0 = approve, 1 = reject) ‖ `escrowed_transaction_hash` (32) = **36 bytes**

ApprovalEscrow changed in v0.4.0, from `approval` ‖ 8-byte id (12 bytes) to the 36-byte form above. The parser refuses any other length. The reason is a hardware-wallet one: a signer that saw only an 8-byte id could not check what it was releasing, and an 8-byte id is findable by a 2^64 search against a fake escrow shown to the device. With the full hash the device can verify what it approves; §7.3 gives the four-step check, and the hash to put in the body comes from `GET /api/v1/accounts/<a>/escrows` as `transaction_hash`.

Fifty-six transaction types exist. §6 catalogues all of them in three tiers, with a recommendation to refuse Tier 3 by default behind an explicit expert setting, showing `SHA3-256(body)` as a fingerprint, which is how other chains' apps handle unknown contract calls.

Size limits: body ≤ 65 536 bytes, plain message ≤ 256, dataset value ≤ 4096. The escrow instruction sits outside the body and is bounded only by the node's request size, so §5.3 recommends firmware caps of 2 KB for the whole transaction, 256 B message and 512 B escrow instruction. Tier 1 transactions are under 300 bytes.

A9 "Signing: what exactly gets hashed/signed, signature algorithm, how the signature is attached to form the final signed tx which broadcasting endpoint expects?"

Signature algorithm: Ed25519, RFC 8032, pure, verified by libsodium `crypto_sign_ed25519_verify_detached` (`src/crypto/signature.cpp:68`). Signature is 64 bytes, `R ‖ S`.

What gets hashed and signed:

```
digest    = SHA3-256("ZBC-TX" ‖ genesis_block_hash(32) ‖ unsigned_bytes)
signature = Ed25519.sign(sk, digest)          the 32-byte digest IS the Ed25519 message
```

Three parts of that need spelling out, because each is a place an implementation goes wrong silently.

The tag. `"ZBC-TX"` is six ASCII bytes, `5a 42 43 2d 54 58`. **No NUL terminator and no length prefix.** A firmware that terminates the string, or writes seven bytes, produces a valid-looking signature that fails on chain with a generic error and no further diagnostic. Test this against §10 vector 0 before anything else.

The genesis hash is the raw 32 bytes of the chain's genesis block hash, exactly as `GET /api/v1/node/info` returns `genesis_hash`, hex-decoded, same byte order, no reversal, no re-hashing. A node prints lower-case hex and an archival API upper-case, so decode case-insensitively.

The hash function is SHA3-256, FIPS 202. Substituting Keccak-256, the Ethereum variant, changes the padding and every signature fails with the same bare `Invalid transaction signature`. Along with the tag, this is the most likely source of a silent integration failure.

Why the genesis hash is in there. It binds the signature to one chain, so a signature made for one ZooBC chain cannot be replayed on another. It is new in node v0.4.0. Note that the transaction `version` byte was **not** bumped and is not a discriminator: a v0.4.0 node has one rule and verifies everything against the chain-bound digest. The discriminator a host needs is in `GET /api/v1/node/info` as `signing_version: 2` and `signing_tag: "ZBC-TX"`. Read those from the endpoint you will broadcast to, since that is the chain that judges the signature.

The genesis hash travels with the transaction, host to device. The device does not look it up, so your signing code is chain-agnostic: one code path for testnet, devnet, mainnet and any private chain, with no chain table compiled in for correctness. A firmware table of known hashes is still worth having, but only to display a trustworthy network name, falling back to "UNKNOWN CHAIN". Do not refuse unrecognised chains by default; devnet's hash changes at every relaunch.

How the signature is attached. There are two different answers here and conflating them is the trap in your question:

- **For hashing:** `signed_bytes = unsigned_bytes ‖ signature(64)`, and `tx_hash = SHA3-256(signed_bytes)`, `tx_id = int64le(tx_hash[0..8])`. The tag and genesis hash do **not** enter `tx_hash`.
- **For broadcasting:** those `signed_bytes` are **not** what you post. The endpoint takes the body bytes plus the named envelope fields, with the signature as a separate hex string, and reconstructs the envelope itself. See §A6.

So the concatenation exists to define the transaction hash, not the broadcast payload. The device should return both the signature and its own computed `tx_hash`, so the host can compare that against the hash the node returns after broadcast.

One firmware rule to carry into your API design. Some ZooBC flows sign bytes without the SHA3 prehash, plain `Ed25519.sign(k, bytes)`. None is a hardware-wallet flow, but if the firmware exposes a raw message-signing command it must **never sign an input of exactly 32 bytes**, because a 32-byte raw input could be the digest of a transaction the user never saw, which turns the message command into blind transaction signing. Chain binding does not change this: the tag and genesis live inside the SHA3 preimage, not inside what Ed25519 sees, so the Ed25519 message is still exactly 32 bytes.

Reference values. §11 has the whole algorithm in about 30 lines of Python. For reference chain R, whose genesis hash is `090ab3c7e0ce0630f9b6a3d9694dc4d2d5e1ccd018352ef27de6cd8de0a61878`, vector 0 gives:

```
unsigned   0100000001 00f15365 00000000 … 0200000000000000           113 bytes
preimage   5a42432d5458 ‖ 090ab3c7…a61878 ‖ unsigned                 151 bytes
digest     f37e53402bd62fd5b7f2fe39eea7129eaedb929c0038a132aeefeaad774644fc
signature  9a3b044dca7c2f351e9d2b37dd6924e4260536c496207de87f0e1f1a31dd9f01
           cd6359e202e13964580a561a89e26688749fe9cdbf12056ed05eb1fbcd03bb07
tx_hash    f00a6692eccb249e0e1cf70f1215502dcc82df6b4d47799efe3dc13d53ce78b0
```

A live v0.4.0 node returned HTTP 202 and that exact `transaction_hash`. If your implementation reproduces this from the `abandon … about` mnemonic, the derivation, serialisation, tag, genesis prefix, hash and signature are all correct. §10 has six more vectors, including one with an escrow, one from account 1 and one with a fee above 2^53.

B Endpoints and what to build against

CHAIN	GATEWAY (HTTPS)	NODE API	SIGNING	NOTE
testnet	`https://zoobc.network`	behind the gateway	v0.3.2 legacy, **moving to v0.4.0 imminently**	the integration target once it flips
devnet	`https://socialconnect.network`	`http://193.181.213.106:8080`	v0.4.0	11 nodes, faucet at `/faucet`; genesis hash changes at every relaunch
mainnet	—	—	v0.4.0 or later	not launched

Read the genesis hash at runtime rather than hard-coding it. Devnet's changes at each relaunch, and testnet's will be whatever its move to v0.4.0 fixes it at. Because the hash is supplied per transaction (§A9), a firmware needs no chain table to sign correctly; a table only puts a name on the screen, and an unrecognised hash shows "UNKNOWN CHAIN" while signing continues to work.

Route reads archival → gateway → node, with the `history` exception in §A3. Broadcast to a gateway or a node, never to an archival endpoint.

`GET /api/v1/node/info` returns `genesis_hash`, `signing_version`, `signing_tag`, `height`, `version`, `synced`, `archival_node` and `wedge_reason`. A gateway answers that route from the archival service, which passes the two signing fields through from the node it mirrors on build `7ad161a4` and later. Read them from a node endpoint, or from a gateway on that build or later.