ODNCA-STD-001 · SNS Resolution Specification

How on-chain names are claimed, transferred, and resolved

Status: v1.0 — Adopted

Normative reference implementations: ORDnet indexer V31 (claim validation, ownership tracking) and ORDnet SNS Resolver v1.1 (resolution, signed answers). This document describes deployed, running behaviour; where the two layers differ in detail, the difference is documented, not hidden (Annex B).

Companion documents: ODNCA-STD-002 (Protocol Parameters Registry), ODNCA-STD-003 (SNS-NAME-1, name format).

1. Scope and design position

This specification defines the full life of an SNS name: how it is **claimed** on-chain, how ownership **transfers**, and how any party turns a name into its current holder through **resolution** — with answers that are signed, time-bounded, and independently provable against the chain.

The design position throughout: **the chain is the root.** Validity and ownership are facts on a public blockchain, recomputable by anyone running the published rules. The resolver is a convenience layer that reads those facts and signs its reading; it is not a source of truth, and every answer it gives can be checked against the chain by the recipient. Trust is optional at every level (§9).

Out of scope: registrar sales policy, pricing, premium tiers, and service-level mail alias administration — all registrar-level concerns (see ODNCA-STD-003 §1 for the layer principle).

2. The on-chain record

2.1 Registration (claim)

A claim is a **1Sat ordinal inscription**: an inscription envelope carrying a content-type and a body, on an output of **exactly 1 satoshi**. The body of a valid SNS registration is a JSON(5) object:

```
{ "p": "sns", "op": "reg", "name": "alex.web3" }
```

The complete registration-validity rules — normalization, structure, character set, length — are defined in **ODNCA-STD-003 (SNS-NAME-1)** and are not repeated here. Two consequences matter for resolution:

- The **canonical name** (lowercased, head-stripped) is the index key and the unit of uniqueness.
- The **raw name** (original case) is retained for lookalike detection (§10).

Both spec-conformant script orders are recognised: locking-script-then-envelope, and envelope-then-locking-script.

2.2 Transfer: the name follows the sat

There is **no transfer operation in JSON**. A JSON body with `op: "transfer"` is not a registration and creates nothing. Transfer is a UTXO-level fact: **spending the output that carries the inscribed satoshi moves the name** to wherever that satoshi lands.

This is deliberate. Ownership transfer requires no protocol cooperation, no registry approval, and no special transaction format — any wallet that can move a satoshi can transfer a name.

2.3 First seen wins

Among valid claims of the same canonical name, the earliest on-chain claim owns the name; all later claims are void and create no rights. A **re-inscription on an already-tracked satoshi** is not a new asset and is ignored: first wins, on the name and on the sat.

3. Ownership tracking (sat-following)

A conformant indexer tracks each name as a triple **(txid, vout, sat_offset)** — not merely an output, but the exact satoshi position within it. This gives transfer tracking offset precision:

1. **Position math.** When a tracked output is spent, the sat's global position in the spending transaction = (sum of input values before the carrying input) + (offset within that input). That position is mapped onto the outputs in order.
2. **Landing classification.** The output where the sat lands is classified:

- **P2PKH** (bare, or combined with a 1Sat envelope in either order) → status `held`, with the holder address derived from the script.
- **OP_RETURN** → the sat is **burned**; the name is dead.
- Past the last output (into the fee) → **burned**.
- Any other script → status `contract`: owner unknown, script hash recorded, tracking continues. An unrecognised script is **never** treated as a burn.

1. **Current outpoint.** At every moment each living name has exactly one **current outpoint** — the (txid, vout) where its sat sits right now. This is the provable heart of resolution (§7): an outpoint that is unspent at any node is proof the answer is live.

2. **Reorg safety.** Every transfer is journaled. On a chain reorganisation, transfers above the fork height are rolled back and each affected name is restored to its last surviving state (or to its mint state if no transfer remains). Two conformant indexers that witness the same reorg converge on the same state.

---

4. Resolution: from address to holder

4.1 Address grammar

```
<address> := [ <mailbox> "@" ] <name> "." <tld>
```

Both `ordnet.web3` and `alexander@ordnet.web3` are resolvable. An empty mailbox (`@ordnet.web3`) means the domain itself.

4.2 Query normalization

Performed by the resolver on every incoming address, in this order:

1. Trim whitespace; strip a leading `@` or a `sns:` / `ordns:` scheme prefix.
2. More than one `@` → reject (`invalid_address`).
3. Split off the mailbox at the single `@`, if present.
4. Lowercase the **TLD** unconditionally.
5. Lowercase the **name** and **mailbox** only if they are pure ASCII. Non-ASCII input is matched on **exact UTF-8 bytes — no Unicode normalization, ever**. (Rationale in §10; interaction with registration-time canonicalization in Annex B.)

4.3 TLD gate

The registrable and retired TLD sets are parameters, published in ODNCA-STD-002 and served live by the registry (single source — the resolver does not hardcode them). Currently:

- Registrable: `web3, bitcoin, crypto, blockchain, ordnet`
- Retired: `bsv, bitcoinsv` — **existing names still resolve**; new registration is closed (`retired_tld` on registration paths only).
- Unknown TLD → `unknown_tld`.

4.4 Lookup

The canonical name is looked up in the index state (§3). Found → build the answer (§6) with the current outpoint and holder. Not found → `not_registered`. Found but holder momentarily undetermined (a fresh registration still propagating registry → inscription → indexer, up to ~10 minutes) → `no_holder`, retry shortly.

---

5. Mailbox semantics

The rule is one sentence: **the holder of a mailbox is, by definition, the holder of the domain.** The resolver therefore resolves the domain and pays its holder — one path, no failure case.

If the queried mailbox is unknown (nonexistent, or a typo), the answer still succeeds but carries `fallback: true`: payment goes to the domain holder, and the wallet MUST disclose this inline before sending ("mailbox unknown — payment goes to the holder of *name.tld*"). Fallback never blocks payment; it informs it.

Mailbox/alias *administration* — creating aliases, deactivating them when a domain changes hands, premium gating — is service behaviour at the registrar layer and is out of scope for this standard. The chain-level invariant stands regardless: whoever holds the sat holds every mailbox on the name.

---

6. The signed answer

```
{
  "ok": true,
  "v": 1,
  "input": "alexander@ordnet.web3",
  "name": "ordnet.web3",
  "mailbox": "alexander",
  "source": "sns",
  "fallback": false,
  "holder_address": "1...",
  "holder_script": "76a914...88ac",
  "origin":  { "txid": "...", "vout": 0 },
  "current": { "txid": "...", "vout": 0 },
  "as_of_height": 960102,
  "expires": 1785410386,
  "sig": "3045...",
  "signer": "03088f..."
}
```

Field semantics:

- `current` — the outpoint carrying the name right now. Verify it is unspent (any node, any indexer) and the answer is *proven* live, not trusted.
- `origin` — the outpoint of the first inscription; the name's birth certificate.
- `holder_script` — the script to pay. It always pays the rightful owner as administered; for names held in registry custody the script on the `current` outpoint may differ from `holder_script` — the outpoint check proves freshness, not script identity. (A signed `custody` field is planned for v2.)
- `fallback` — see §5. Signed, so a relay cannot strip the disclosure.
- `expires` — Unix time; answers are valid for a short TTL (300 s). Signed, so an old answer cannot be replayed as fresh.
- `as_of_height` — index height at answer time; `0` means the tip was unavailable (skip the reorg comparison, rely on the outpoint check).
- `source` — informational in v1 (not signed); v2 signs it when additional protocols join the endpoint.

---

7. Signature scheme

Deterministic, JSON-free serialization so that every language produces bit-identical bytes:

```
fields    = v, name, mailbox, holder_script,
            origin.txid, origin.vout, current.txid, current.vout,
            as_of_height, fallback, expires
canonical = fields joined with 0x1f      (booleans as "true"/"false",
                                          numbers in decimal, UTF-8)
sighash   = SHA256( SHA256( "ORDNS-RESOLVE" || 0x1f || canonical ) )
sig       = ECDSA(secp256k1) over sighash, DER-encoded, hex
```

Notes:

- **Signed set is exhaustive as listed.** `input`, `source`, `holder_address`, and `signer` are *not* signed. `holder_address` is derivable from `holder_script`: verify the script, pay the script.
- **Missing mailbox serializes as the empty string.**

7.1 Conformance test vector

An implementation is correct when this input reproduces this sighash exactly:

```
v=1  name=ordnet.web3  mailbox=alexander
holder_script=76a914e8e5f64b0c7943b93e58b24e3f82d533e70b3db188ac
origin =367a0a1d553002f0f3427168a10f86835e2741c111df43262d35fb475400e3ee:0
current=dc54c20af97682eebf99dc8392c21b904908398d543aae6fabffe09a9b7780ac:0
as_of_height=959941  fallback=true  expires=1785312000

sighash=28a4252e92fdcdb70d6fd287cdb602cda504d288963e106b47a6d8d19420ec6b
```

7.2 Key management and rotation

The resolver's signing key is published at `GET /pubkey`, in the operator listing, and MUST NOT be treated by clients as an unverifiable constant: **pin on first use, treat `/pubkey` as the live authority.** Rotation procedure: the new key is announced **signed by the old key**, published at `/pubkey`, on the operator's site, and as an on-chain inscription. A client that sees an unknown signer fetches `/pubkey`, verifies the rotation announcement against its pinned key, then updates the pin.

---

8. Errors

Errors are always readable JSON with a machine code — never an empty reply:

CODE	MEANING
`invalid_address`	More than one `@`, empty name, no dot
`unknown_tld`	TLD is not in the registrable or retired sets
`not_registered`	The name does not exist
`retired_tld`	Registration closed on this TLD; existing names still resolve
`no_holder`	Holder temporarily undetermined (fresh registration propagating); retry
`rate_limited`	Client exceeded the rate limit (default 120 req/min per IP)

---

9. Client verification levels

Integration is graded; each level is a strict superset of the previous. Skipping a level is the client's choice, never the protocol's requirement:

1. **Trust** — call `GET /resolve/<address>`, read JSON, pay `holder_script`.
2. **Verify** — additionally check `sig` against the known resolver key (§7). Makes caching safe and every answer provable after the fact.
3. **Prove** — additionally check that `current.txid:vout` is **unspent** via your own node or any independent indexer. At this level you trust nobody: the answer is a claim you have verified against the chain itself.

Reference wallet flow:

```
1. normalize the address (§4.2)
2. GET /resolve/<address>
3. verify sig against the pinned resolver pubkey
4. check expires > now
5. check current.txid:vout is unspent
      yes -> live holder confirmed
      no  -> stale answer, re-query
6. fallback == true -> disclose inline (§5)
7. pay holder_script
```

---

10. Security considerations

- **Replay.** `expires` is inside the signed fields; an expired answer fails verification as fresh.
- **Sale mid-payment.** If the name is transferred between answer and payment, the outpoint check (level 3) fails and the wallet re-queries. No funds reach the previous owner.
- **Reorgs.** Compare `as_of_height` with the chain tip; if it lags more than a few blocks, re-query. Indexer-side reorg handling is defined in §3.4.
- **Caching.** Cache freely within `expires`; the outpoint check turns a stale cached answer into a harmless re-query instead of a misdirected payment.
- **Lookalike names.** `ordnet.web3` and `0rdnet.web3` are different names; with non-ASCII input the space of confusables grows (mixed scripts, zero-width characters). The resolver deliberately performs **no Unicode normalization** — it returns exactly what is on chain, because silent normalization is itself a spoofing vector. Display layers SHOULD warn inline and offer a codepoint/hex rendering for non-ASCII or mixed-script names; the retained raw name (§2.1) supports detection.
- **Hardened input handling.** Control-byte and encoding attacks on the query path are rejected at normalization (v1.1 hardening).

---

11. Conformance

Two independent implementations conform when, given the same chain, they produce (a) the same set of valid names with the same canonical keys (ODNCA-STD-003 vectors), (b) the same current outpoint and holder classification for every living name (§3), and (c) bit-identical sighashes for the answers they sign (§7.1). Conformance is technical and testable; no permission is involved. Run the vectors, match the answers.

Annex A — Bridge decisions (normative for the bsvalias bridge)

The compatibility bridge (bsvalias v1.0.1) lets legacy paymail wallets pay SNS names today. Two conventions are fixed as protocol parameters (ODNCA-STD-002):

- **D1 — default TLD.** A bare handle arriving via the bridge (`alexander@ordnet.io`) maps to the `.web3` TLD: `alexander.web3`, unless an explicit name is encoded.
- **D2 — mailbox separator.** The bridge convention for addressing a mailbox on a name through a single legacy local-part, preserving §5 semantics (mailbox holder = domain holder, fallback disclosure included).

The bridge is an on-ramp, not a second root: every bridge resolution terminates in the same signed-answer path defined by this specification.

Annex B — Implementation note: case handling across layers

Registration-time canonicalization (indexer, ODNCA-STD-003 §4) applies **Unicode** lowercasing; query-time normalization (resolver, §4.2) lowercases **ASCII only** and matches non-ASCII on exact bytes. Consequence: a name inscribed with non-ASCII uppercase (`CAFÉ.web3`) is indexed as `café.web3`, and a query repeating the uppercase form will miss. Queries in the canonical (lowercase) form always match. This asymmetry is documented as deployed behaviour; a future revision may align query normalization with registration canonicalization. It affects no pure-ASCII name.
