WORKING DRAFT · 2026-05-27

Cooldat Data Exchange Standard

CDX-1 — JSON wire format for UHF RFID temperature dataloggers

Reference:	QDatDroid → qdat.io contract
Tenant:	tapdpp.qdat.io (live)
Levels:	L0 identifier-only → L4 closed-loop downlink
Samples:	9 reference payloads under `samples/`

Cooldat Data Exchange Standard (CDX-1)

Status: Working draft — extracted from the live QDatDroid → qdat.io implementation on `tapdpp.qdat.io` (cluster `qdat-tapdpp`, namespace `qdat-production`), 2026-05-27. **Scope:** End-to-end JSON wire format for UHF RFID temperature dataloggers, with first-class support for Axzon OPUS-family chips ("Cooldat" dataloggers), and any UHF Gen-2 tag the publishing device can see. **Goal:** Define a single, vendor-neutral envelope that any acquisition app (mobile, fixed reader, gateway) can emit, and any back-end can ingest, without bespoke per-vendor adapters.

---

1. Why a standard

Cold-chain, pharma, and asset-tracking dataloggers from different vendors today speak a proprietary mix of:

- **Vendor JSON over MQTT** (e.g. Zebra IoT Connector, Impinj ItemSense),
- **Vendor CSV/XML exports** (Sensitech, ELPRO, Tempmate),
- **Raw EPCglobal LLRP** (with no application-level semantics for sensor values),
- **HTTPS callbacks** with one schema per device family.

A handful of facts are universal in practice across all of them, and a handful more are universal across the modern UHF-with-on-chip-sensor families (Axzon OPUS, Magnus-S2/S3, Impinj Monza-Sx temperature-aware parts, EM4325, IDS SL900A). CDX-1 codifies that intersection as a single JSON envelope with strict optionality semantics, so:

- A publisher only fills in what the device actually measured.
- An ingester treats absence as "unknown", never as a typed null.
- A vendor can extend with name-spaced fields without breaking conformant consumers.

The format already runs in production for `tapdpp.qdat.io` (Cooldat-branded Axzon OPUS dataloggers tracked via the QDatDroid Android app), and is the literal contract between the QDatDroid publisher and the qdat.io ingester. This document promotes that contract from "internal protocol" to **CDX-1**, an open profile any vendor can implement.

---

2. Conformance levels

Level	Capability	Required envelopes
L0 — Identifier-only	Publisher reads only the chip identifier (EPC/UID) and reports presence.	`SCAN` single
L1 — Inventory	Adds RSSI, GPS, optional chip family, optional state.	`SCAN` single + batch
L2 — Sensor snapshot	Adds live temperature / battery / on-chip RSSI / alarms (PC-word + USER-bank decode).	`SCAN` with sensor fields
L3 — Historical dump	Adds the device-side flash-memory dump (`READ` batches + `READ` complete).	`SCAN` + `READ` + `READ complete`
L4 — Closed loop	Adds the ingester's downlink `TAG_REGISTERED` ack, including DPP-rule redirect.	All of the above + `TAG_REGISTERED`

L0–L1 can be implemented by any reader that speaks Gen-2 inventory. L2 requires a sensor-aware tag. L3 requires the publisher to be able to read the tag's USER bank (or vendor-equivalent flash memory). L4 is for two-way deployments where the device wants to know whether the back-end accepted the scan and/or where to redirect the visitor.

3. Transport

CDX-1 does **not** prescribe a transport. The reference implementation uses MQTT with QoS 1, but the same JSON travels equally well over HTTPS POST, AMQP, or Kafka. The default topic shape used by QDatDroid is:

```
devices/<device_topic_id>/scan          # publisher → broker (SCAN, READ)
devices/<device_topic_id>/command       # broker → publisher (TAG_REGISTERED, etc.)
```

`device_topic_id` is a UUID provisioned by the back-end at device-registration time and stored in `reader.mqtt_topic_id` on the qdat.io side. Multi-tenant brokers can ACL on a `devices/<topic_id>/#` glob per device without leaking other tenants' scans.

4. Envelope (SCAN and READ)

Every CDX-1 payload is a JSON object at the root, with this skeleton:

```
{
  "command":     "SCAN" | "READ",
  "status":      "SUCCESS" | "ERROR",
  "command_id":  "<uuid>",
  "parameters":  { ... },        // present on SCAN
  "epc":         "<hex>",        // present on READ
  "data":        [...] | {...},  // shape varies — see §5, §6
  "batch_timestamp": <int>       // present only on SCAN batch
}
```

4.1 `command_id`

A v4 UUID generated per-publish by the publisher. It serves two purposes:

1. **Multi-batch correlation.** Every `READ` batch for the same flash-memory dump carries the same `command_id`; the back-end aggregates them by that key.
2. **Round-trip correlation.** When the back-end issued the command itself (e.g. a remote `inventory` via the `Reader` API), it can match the SCAN response to the originating row in its `command` table and surface a completion status to the operator.

`command_id` is mandatory and SHOULD be a v4 UUID. Implementations MAY use any 128-bit string that is unique per publisher per minute.

4.2 `parameters` (SCAN only)

Field	Type	Required	Description
`parser`	string	yes	`"opus"` for UHF Gen-2 SCANs (single or batch). Future profiles MAY register new parser names (`"ble"`, `"qr"`, …).
`tags_found`	int	yes	Length of the `data` array. Always 1 for single SCAN; ≥1 for batch SCAN.
`batch_interval_ms`	int	optional	Present iff the publisher is in batch mode. Equals the configured inter-batch interval in ms.

4.3 `data`

`SCAN` payloads carry `data` as an **array** of per-tag objects (see §5). `READ` payloads carry `data` as a **single object** per batch (see §6).

4.4 Absence vs. null

Implementations **MUST** omit a field rather than serialize a typed `null` whenever the source value is unknown / unset / zero-sentinel. The default reading of the field's reference table specifies the exact omission criterion. Consumers MUST treat absence as "unknown" — never as "zero" or "false". This rule is the single most important conformance requirement in CDX-1 because it lets a publisher cheaply express partial knowledge (e.g. an UHF reader that doesn't have a GPS fix) without inventing sentinel values.

---

5. Per-tag object — UHF Gen-2 (`parser="opus"`)

Field	Type	Required	Unit	Notes
`epc`	string (hex)	yes	—	MSB-first. Normalized so its length agrees with the PC-word's EPC-length field.
`tid`	string (hex)	yes	—	MSB-first, uppercase. Empty string when the publisher didn't read the TID. Typically 24 hex chars (12 bytes) for OPUS.
`tag_type`	string	yes	—	`"Opus"` for OPUS dataloggers (TID starts `E2C24500`), `"Generic"` for everything else.
`chip_manufacturer`	string	optional	—	Omitted when `tid` is empty. Falls back to the sentinel `"RFID"` when the TID's MDID is not in the local registry. See §7.1.
`cin_brand`	string	optional	—	Present only when the EPC carries a valid RAIN CIN+FFe encoding (PC bit 8 + AFI `0xAE` / `0xBC`).
`cin_decimal`	int	optional	—	Always co-emitted with `cin_brand`.
`cin_hex`	string	optional	—	Always co-emitted with `cin_brand`.
`rssi`	number	optional	dBm	Reader-side. Omitted when 0.
`state`	string	optional	—	One of `SLEEP`, `STANDBY`, `INTERNAL_02`, `ARMED`, `STARTED`, `BAP_MODE`, `LOGGING`, `FINISHED`. Decoded from PC bits 7–5; omitted when PC word == 0.
`alarms`	array<string>	optional	—	Ordered, deduplicated. See §7.4. Omitted when empty.
`temperature`	number	optional	°C	OPUS-only. Live snapshot from the inventory response.
`battery_voltage`	number	optional	V	OPUS-only. Range ~2.1–3.3 V.
`on_chip_rssi`	int	optional	(raw)	OPUS-only. Tag-side RSSI from the OPUS USER bank.
`samples`	int	optional	—	OPUS-only. Sample count currently stored in flash.
`gps`	object	optional	—	See §7.3.
`time_last_seen`	int	yes	Unix sec	Publisher's clock at publish time.

6. Per-batch object — historical dump (`command="READ"`)

A READ sequence dumps the device's flash memory in 1+ batches, terminated by a completion message:

```
SCAN{command_id=X}          → presence detected
READ{command_id=Y, batch=1, samples=[…128…]}     ┐
READ{command_id=Y, batch=2, samples=[…128…]}     ├─ historical dump
…                                                 │
READ{command_id=Y, batch=N, samples=[…N≤128…]}   ┘
READ{command_id=Y, batch=N+1, complete=true, samples=[]}  ← terminator
```

6.1 Envelope-level fields (READ)

- `epc` (string, hex, **required**) — the tag whose memory is being dumped. May be a left-truncation of the full registered EPC; ingesters MUST do prefix matching against their tag table.
- `data` is an **object** (not array) with the per-batch payload below.

6.2 `data` payload

Field	Type	Required	Notes
`batch`	int	yes	1-based batch counter inside this `command_id`.
`complete`	bool	yes	`true` only on the terminator message; the terminator carries an empty `samples` array.
`samples`	array	yes	Each entry `{ index, timestamp, temp }` — see §6.3. (Legacy publishers MAY emit the key `data` instead of `samples`.)
`config`	object	optional	The device-side flash-memory configuration that produced these samples. Treated as authoritative by the ingester — see §6.4.
`gps`	object	optional	See §7.3.

6.3 Sample

Field	Type	Required	Unit	Notes
`index`	int	yes	—	0-based sample slot in the device's flash.
`timestamp`	int	yes	Unix sec	The device's own clock at the time the sample was taken.
`temp`	number	yes	°C	The recorded temperature.

6.4 `config` (device-authoritative)

The `config` object is the device's own arm-time configuration. The reference ingester accepts three historically-emitted shapes and prefers the flat form below (it's what current QDatDroid emits):

Field	Type	Required	Unit	Notes
`arm_time`	int	optional	Unix sec	Device's clock at arming.
`delayed_start`	int	optional	sec	Delay between arming and first sample.
`lower_temp_limit`	number	optional	°C	Lower alarm threshold latched into the chip.
`upper_temp_limit`	number	optional	°C	Upper alarm threshold latched into the chip.
`logging_interval_value`	int	optional	—	Numeric part of the cadence.
`logging_interval_units`	string	optional	—	`"seconds"`, `"minutes"`, `"hours"`.
`log_interval_seconds`	int	optional	sec	Canonical cadence in seconds (publishers SHOULD also emit this for ease of consumption).
`number_of_samples`	int	optional	—	Sample buffer size on the device.
`total_samples`	int	optional	—	Samples already stored.
`led_enabled`	bool	optional	—	Device-side excursion indicator.
`led_mode`	string	optional	—	`"CONTINUOUS"`, `"ONDEMAND"`, etc.
`fingerSpot`, `antiTamper`	any	optional	—	OPUS-specific; passed through verbatim.

The ingester MUST overwrite the tag's typed `min_threshold` / `max_threshold` from `lower_temp_limit` / `upper_temp_limit` (the device is the authoritative source), and SHOULD merge the rest of the object into a free-form `tag.configuration` JSON column for display and audit.

6.5 ERROR semantics

A READ batch MAY arrive with `status="ERROR"` and an `error` object with `retrying=true`; this is a transient hop the publisher auto-recovers from. CDX-1 consumers MUST ignore ERROR-with-retrying messages (no state change, no log line beyond DEBUG).

7. Reference vocabularies

These tables are normative for any CDX-1 consumer that wants to map the wire values to typed domain concepts (manufacturer enums, alert types, etc.). The canonical JSON is shipped alongside this document under `samples/`, and the production manifest with every entry is published at `backend/app/data/qdat-io-tag-types.json` (60 UHF chip families, 27 RAIN CIN brands, the OPUS state + alarm enums, and the full payload reference).

7.1 UHF chip-manufacturer registry

Derived from the TID's 9-bit MDID + 12-bit TMN. Sample entries:

ID	Family	TID prefix(es)
`Axzon OPUS`	Axzon temperature datalogger	`E2C24500`
`Axzon Magnus-S3`	Axzon RH+temperature	`E2C24022`, `E282403D`
`Axzon Magnus-S2`	Axzon RH	`E2C24020`
`Impinj M730`	Impinj Monza R6	`E2801191`
`Impinj M4QT`	Impinj Monza 4QT	`E2801105`
…	(58 total)	…
`RFID` (fallback)	Catch-all for Class-1 Gen-2 tags with an unknown MDID — TID starts `0xE2` but the MDID is not in the table.	—

Publishers MUST emit a known ID when the TID's MDID matches; MUST emit `"RFID"` when it doesn't; and MUST omit the field when the TID is empty.

7.2 GPS sub-object

```
{
  "latitude":  46.3725245,
  "longitude": -72.6014715,
  "altitude":  32.0,
  "override":  true
}
```

Field	Type	Required	Notes
`latitude`	number	yes	WGS-84 decimal degrees.
`longitude`	number	yes	WGS-84 decimal degrees.
`altitude`	number	optional	Metres above WGS-84 ellipsoid. Omitted when 0.0.
`override`	bool	optional	`true` when the publisher used an operator-pinned position instead of a live GNSS fix; absent when live.

The `gps` object is attached to a per-tag object iff GPS is enabled on the publisher AND (override is set OR the live fix is non-zero). Absent otherwise.

7.3 Alarm vocabulary

The `alarms` array on UHF SCAN per-tag objects is an ordered, deduplicated set of names. Reference values:

Name	Source	Maps to qdat.io alert type
`TEMPERATURE`	PC word bit 0 (generic temperature alarm bit)	`OVER_TEMPERATURE`
`HIGH_TEMPERATURE`	PC word bit 1 or TID 0x10 bit 0 (latched)	`OVER_TEMPERATURE`
`LOW_TEMPERATURE`	PC word bit 2 or TID 0x10 bit 1 (latched)	`UNDER_TEMPERATURE`
`LOW_BATTERY`	USER bank flag	`LOW_BATTERY`
`INITIAL_LOW_BATTERY`	XPC bit 11 (battery was low at arming)	`LOW_BATTERY`
`TAMPER`	USER bank flag	`TAMPER`

A consumer that doesn't need vendor-specific resolution MAY treat any unknown alarm name as "generic alarm" without breaking conformance.

7.4 State vocabulary

The `state` field on UHF SCAN per-tag objects is decoded from PC bits 7–5 (3 bits, 8 values). Reference values: `SLEEP`, `STANDBY`, `INTERNAL_02`, `ARMED`, `STARTED`, `BAP_MODE`, `LOGGING`, `FINISHED`. The qdat.io ingester additionally promotes `ALERT` when an alarm is active.

---

8. The qdat.io tag-template binding (informative)

The same JSON, when received by qdat.io, is bound to a `tagtemplate` row that turns the raw chip identifier into a domain decision (alert thresholds, alarm-routing flags, auto-discovery naming). Binding is tiered:

1. Exact `chip_manufacturer` match (e.g. `"Axzon OPUS"`) wins,
2. otherwise `tag_type` match (e.g. `"Opus"`, `"Generic"`),
3. otherwise the org's `is_default` template,
4. otherwise auto-discovered ("Auto-discovered tag …" + EPC).

A "downgrade guard" prevents a payload carrying the generic `"Opus"` from clobbering a tag previously bound to the more specific `"Axzon OPUS"` — important when QDatDroid couldn't read the TID on a particular scan.

The shipped sample `07-tag-template.json` is the live `Cooltag-Opus` row from the `tapdpp.qdat.io` tenant; the alert-routing flags are precisely the ones the ingester gates each alarm by.

---

9. Storage model on the consumer side (informative)

For implementers who want their consumer to behave like qdat.io:

Persisted as	Source field	Lifecycle
`tag.epcid`	SCAN/READ `epc`	Unique per organisation.
`tag.tid`	SCAN `tid`	Normalised uppercase hex.
`tag.state`, `tag.battery_status`	SCAN `state`, `battery_voltage`	Updated on every SCAN; `battery_status` stored in millivolts (`round(V × 1000)`).
`tag.last_lat`, `tag.last_lon`, `tag.last_altitude`, `tag.last_seen_at`	SCAN `gps`, `time_last_seen`	Per scan.
`tag.alert_active`, `tag.alert_type`, `tag.alert_triggered_at`, `tag.alert_value`	SCAN `alarms`	Gated on template flags; auto-resolves when alarms clear.
`tag.min_threshold`, `tag.max_threshold`	READ `config.{lower,upper}_temp_limit`	Device-authoritative (overwrites).
`tag.configuration`	READ `config` (merged)	Free-form JSON audit.
TagEvent row	One per per-tag object on SCAN	Carries the raw per-tag SCAN object verbatim under `data`. Live temperatures live here.
TagReading row	One per `samples[]` entry on READ	TimescaleDB hypertable, PK `(tag_id, timestamp)`, `ON CONFLICT DO NOTHING` for idempotent re-dumps.
TagAlert row	One per new SCAN alarm not already active	Resolves when alarm clears in subsequent SCAN.

Live SCAN temperatures are deliberately stored in `TagEvent.data` and **not** as a `TagReading` — only flash-memory dumps produce TagReadings, so the time-series view is monotonic with respect to device-recorded sample cadence.

10. The downlink (`TAG_REGISTERED`) — closing the loop

After a SCAN batch commits, the back-end MAY publish a `TAG_REGISTERED` message back to the originating device on `devices/<topic_id>/command`. This is what enables L4 conformance:

```
{
  "command":    "TAG_REGISTERED",
  "command_id": "<uuid>",
  "data": {
    "tag_id":           "<uuid>",
    "epc":              "<hex>",
    "url":              "<resolved URL>",
    "tag_type":         "<enum>",
    "chip_type":        "<bound chip identifier>",
    "state":            "<enum>",
    "organisation_id":  "<uuid>",
    "newly_created":    bool,
    "scanned_at":       "<ISO-8601>",
    "redirect_url":     "<url>",        // present only when a DPP rule fired
    "redirect_reason":  "<rule name>"   // present only when a DPP rule fired
  }
}
```

The device uses `url` to navigate the visitor (Cooldat's mobile UI shows a result page after each scan); when a DPP rule fires (geofence, time-elapsed, spatiotemporal AND), the back-end overwrites `url` with the rule's destination so older devices that ignore `redirect_url` still do the right thing.

---

11. Path to standardisation

The current state of the spec covers everything QDatDroid emits today and qdat.io ingests today, but five things would make it credible as an open data-exchange standard:

1. **Schema artefacts.** Publish a JSON-Schema 2020-12 file for each envelope (`scan.json` , `read.json` , `tag-registered.json` , `tag-template.json`). The shipped `qdat-io-tag-types.json` is already the precursor; a `$schema_version` field is in place.
2. **Conformance test suite.** Reuse the 331 stress payloads under `dist/qdat-io-stress-payloads.json` as the L1–L2 conformance corpus, and add the equivalent READ corpus for L3. Any consumer that round-trips all 331 without dropping a row passes.
3. **Vendor neutrality of the registries.** Carve the manufacturer registries (UHF chip MDID/TMN, RAIN CIN brand) out of QDatDroid's source tree into a separate, vendor-neutral repository governed by a steward (GS1, RAIN Alliance, or a new working group). Today they live in `TagTypeIdentifier.java` inside the Android app.
4. **Field-name namespacing for extensions.** Reserve the unprefixed top-level field names for the standard; require vendor extensions to use a `x-<vendor>-<field>` prefix (e.g. `x-axzon-`

`fingerSpot`). This is how qdat.io can keep `fingerSpot` and `antiTamper` flowing into `tag.configuration` without polluting the CDX-1 surface.

5. **A reference open-source consumer.** The qdat.io ingester (`backend/app/services/mqtt_consumer.py`) already is the reference consumer — extract the SCAN / READ / TAG_REGISTERED handlers into a stand-alone Python library (`cdx1-ingest`) with a TimescaleDB schema migration, releasable under Apache-2.0. This drops the implementation barrier for any cold-chain ISV to support CDX-1.

Once those five exist, CDX-1 is in a position to be proposed to the RAIN Alliance Sensor Tag working group (or to GS1's EPCIS 2.0 sensor-data successor) as a profile.

---

12. Mapping to existing standards

CDX-1 concept	Closest existing standard	Gap
`epc` (UHF)	EPCglobal Tag Data Standard 1.13	None — CDX-1 carries the same hex string, MSB-first.
`cin_brand` / `cin_decimal`	RAIN CIN+FFe encoding	None — CDX-1 just decodes it.
`alarms[]`	GS1 EPCIS 2.0 sensor element (`sensorReport.exception`)	Stricter vocabulary; CDX-1 enumerates 6 names.
`temperature`, `battery_voltage`	EPCIS 2.0 `sensorReport.type` + `value` + `uom`	CDX-1 collapses to direct fields + implicit unit; EPCIS is more flexible but verbose.
READ batch	No equivalent	EPCIS expresses one report per event; CDX-1's flash-dump idiom is novel and required by the realities of intermittently-connected dataloggers.
`TAG_REGISTERED` downlink	No equivalent	Vendor-specific in practice (Sensitech SmartLink, ELPRO LIBERO Connect).

CDX-1 deliberately stays simpler than EPCIS 2.0 — it's an on-the-wire format, not a query model — but a bidirectional adapter is a small piece of work. An EPCIS publisher implementing CDX-1 would emit one `ObjectEvent` per SCAN-array entry with a `sensorReport` per OPUS field, and one `ObjectEvent` per READ sample.

---

13. Reference samples

The `samples/` directory beside this document contains a runnable corpus of every envelope in CDX-1:

File	Envelope	Notes
`01-scan-single-opus.json`	SCAN single	Live Axzon OPUS datalogger from tapdpp, EPC `5201F25100009140`.
`02-scan-batch-mixed.json`	SCAN batch	Three tags, mixed chip families — covers omission rules.
`03-scan-with-alarms.json`	SCAN with alarms	Multi-alarm scan triggering OVER_TEMPERATURE + LOW_BATTERY + TAMPER.
`04-read-historical-batch.json`	READ batch	Flash-memory dump with embedded device `config`.
`05-read-complete.json`	READ complete	Terminator.
`06-tag-template.json`	Tag-template binding	Live `Cooltag-Opus` from tapdpp.
`07-tag-registered-ack.json`	TAG_REGISTERED downlink	Successful registration ack.
`08-tagreading-stored.json`	qdat.io REST `TagReading` row	Storage form of one historical sample.
`09-tagevent-stored.json`	qdat.io REST `TagEvent` row	Storage form of one SCAN hit.

Two additional bulk artefacts already in this repository round out the picture:

- `dist/qdat-io-tag-types.json` — the manifest of every registry value any CDX-1 publisher may emit.
- `dist/qdat-io-stress-payloads.json` — 331 payloads covering every identifier value plus documented fallbacks, suitable as a conformance test corpus.

14. Versioning

CDX-1 envelopes do not carry a version field — the field set is the version. Future revisions ship under a new top-level qualifier:

```
{
  "command":  "SCAN",
  "cdx":      "2",        // present only on CDX-2+ payloads
  …
}
```

A consumer that doesn't recognise `cdx` MUST reject the payload. A consumer that recognises `cdx="1"` (or its absence) MUST accept it. Within CDX-1 itself, additions are backwards-compatible by the absence-vs-null rule; renames or removals would be a new major version.

---

15. Editor's note

This document was assembled from:

- The live `tapdpp.qdat.io` production database (cluster `gke_qc2-prod_northamerica-northeast1-a_qdat-tapdpp`, `qdat-production` namespace) for shape verification.
- The QDatDroid → qdat.io contract artefacts under `dist/` and `backend/app/data/qdat-io-tag-types.json`.
- The reference ingester implementation in `backend/app/services/mqtt_consumer.py` (`_process_inventory_result_inner`, `_process_qdat_read_result`, `_apply_qdat_config_to_tag`).
- The Cooltag template definition in `cooltag.json`.

Implementations have been validated end-to-end on Cooldat (Axzon OPUS) temperature dataloggers in cold-chain pilots, and the same envelopes have been observed to work unchanged with non-sensor Impinj and EM Microelectronic UHF tags as L0/L1 publishers.