BOVNAR · V1.1

Bovnar — Unit Policy Reference

Bovnar (BVNR) v1.1 documentation · Unit Policy Reference · 2026-07-28

> **Spec version:** 1.1 **Status:** Reference — the unit policy carried by `bvnr_unit_policy_t`, on the reader and on the writer. **Scope:** What a policy declares, the order its parts are consulted in, what it validates, what it converts, and the errors it raises. Behaviour only; the C declarations live in the read/write API reference.

Companion to Unit & Currency Reference (the registry a policy names units from) and Read/Write API (the entry points that install one). Every conversion and every error shown here was produced by running the reference implementation built from this tree; the transcripts are output, not illustration.

A unit policy states, as **data**, what a document must contain and what unit the consumer wants values delivered in. It is the declarative form of the `want_unit` hook: because every unit is given as text, a binding can drive the whole feature without a per-value callback across an FFI boundary. Nothing about it weakens the exactness contract — the policy chooses targets, and the conversion itself is the same exact arbitrary-precision path `want_unit` uses.

Table of Contents

1. Overview

1.1 What a unit policy is

One `bvnr_unit_policy_t`, installed on a reader with `bvnr_reader_set_unit_policy` or on a writer with `bvnr_writer_set_unit_policy`. It carries two independent kinds of statement:

- **Validation** — what the document must contain. These reject a document and never change a value: `require_unit`, `require_dimension_of`, and the assertion half of a per-field rule.
- **Conversion** — what unit the consumer wants values in: `rules`, `targets`, `normalise`, with `base` and `on_inexact` shaping the result.

A policy describes the **consumer**, not the document. It may be installed before or after the reader or writer is opened, it survives re-opening on another document, and passing `NULL` clears it.

1.2 What the parser enforces without one

Without any policy the parser still enforces everything the format itself guarantees: a unit written on a value must exist in the registry, a unit parameter in a type annotation must agree with an inline unit on the same value, and an exponent must be in range. What it does not do is have an opinion about which units a particular consumer wanted. `.speed = 3 kg;` is a perfectly valid document; only the application knows it is nonsense.

A policy is where that application knowledge goes.

1.3 A policy and the want_unit hook

Both may be set. The hook is more specific and wins: for each value the reader asks `want_unit` first, falls back to `targets`, and falls back again to `normalise`. Normalising a document while hand-handling the one field that needs something else is the intended combination.

The two differ in their exactness contract. A `want_unit` hook is strictly all-or-nothing. A policy may be told to step over what it cannot deliver exactly — see 2.4.

---

2. The policy object

2.1 Per-field rules

A rule names **one field**, by the dotted key path a value sits at, leading dot included, exactly as the document writes its keys. It is the most specific thing a policy can say, and is consulted before everything else in it.

```
static const bvnr_unit_rule_t rules[] = {
    { ".inlet.temperature", "°C", 0, bvnr_rule_convert },
    { ".inlet.*",           "m",  0, bvnr_rule_require },
};
```

A path ending in `.*` matches every value below that point at any depth: `".inlet.*"` covers `".inlet.temperature"` and `".inlet.pump.rpm"`, but not `".inlet"` itself. A prefix matches only at a component boundary, so `".in.*"` does not claim `".inlet.a"`. Array elements sit at the path of the assignment holding them, so one rule covers every element of an array.

`bvnr_rule_convert` asserts and converts; `bvnr_rule_require` asserts only, delivering the value exactly as written — "this field is a length, in whatever length unit it chose".

A rule is an **assertion**, unlike a whole-document target: the caller named this field, so a value that cannot be applied to it is `error_unit_mismatch` rather than a value passed through quietly. Silence would defeat the point of naming it. A bare number fails a rule too — `.speed is m/s` is not satisfied by a value with no unit.

2.2 Conversion targets

`targets` applies to the whole document. Each numeric value is converted to the **first** target it can validly convert to, so order is significant: a list of `{"m", "k~m"}` never selects `k~m`. Each target may carry its own output base, `0` meaning the value keeps its own.

A value that matches no target is delivered untouched unless `normalise` catches it. A value already in the unit a target names is also left untouched — unless that target asks for an output base, which makes it a pure base conversion. The `converted` flag therefore means "the policy restated this value", not "the policy looked at it".

2.3 Normalisation

`bvnr_normalise_si` catches whatever the targets did not, delivering it in coherent SI base units with prefixes folded out. `base` sets the output base for those conversions.

Values with no coherent SI form are left alone rather than reinterpreted: currencies, and every dimensionless unit (`%`, `ppm`, `dB`, `pH`, `rad`, `°`).

2.4 Inexact results

`bvnr_inexact_error` (the default) aborts the parse with `error_unit_inexact`. That is the right behaviour when the caller named a target deliberately.

`bvnr_inexact_leave` delivers the value untouched instead, with `converted == false`. It exists for `bvnr_normalise_si`, where every value is a conversion candidate and a single 5/18 factor would otherwise reject an ordinary document. A consumer using it must read `converted` to know which values it happened to.

It applies only to a result that is exact as a rational but has no terminating expansion in the output base. A genuinely irrational factor still aborts — there is nothing exact to hand over either way.

2.5 Requiring a unit

`require_unit` demands that every numeric value carry one. A bare number fails, whether it was written without a unit parameter or with an explicit `no_unit`.

2.6 Requiring a dimension

`require_dimension_of` demands that every numeric value be validly convertible to at least one of the listed units: "this document is lengths and temperatures, in whatever unit it chose to write them". It never changes a value.

A value with **no** unit satisfies this only if one of the listed units is itself `no_unit`, for the reason in 4.2.

2.7 Limits and lifetime

Limit	Value	Applies to
`BVNR_MAX_UNIT_TARGETS`	8	`targets`, `require_dimension_of`
`BVNR_MAX_UNIT_RULES`	8	`rules`
`BVNR_MAX_UNIT_PATH`	96	a rule's `path`, NUL excluded

Every unit string is parsed by the setter, so the strings themselves need not outlive the call.

---

3. Resolution order

3.1 The ladder

For each numeric value, in order, stopping at the first that applies:

1. a **per-field rule** whose path matches;
2. the `want_unit` hook, if one is set;
3. the first **target** the value can validly convert to;
4. **normalisation**, if `bvnr_normalise_si` is set;
5. otherwise the value is delivered as written.

Validation is separate and does not participate in this ladder: it runs regardless of which rung delivered the value.

3.2 Validation runs on the written unit

`require_unit` and `require_dimension_of` are evaluated on the unit the **document** wrote, before any conversion. Validate what you were sent, convert for the consumer. This means they say the same thing whether or not a conversion was also requested — a document is not made to pass by the policy that was going to rewrite it anyway.

3.3 First match wins

Both `rules` and `targets` are first-match-wins, so order is part of the policy. Put `".a.b"` before `".a.*"`, or the wildcard swallows the specific case.

---

4. Conversion semantics

4.1 Compatible is not convertible

Dimensional compatibility is not the test for whether a conversion can be performed. Currencies carry no dimension vector and therefore fail a compatibility test even against themselves, yet prefix conversion between them is exact and supported:

```
5 k~$USD  ->  $USD      =  5000
```

Affine conversions have no single multiplicative factor, so a screen on "is there a conversion factor" would drop every temperature in the format. Both of these work:

```
212 °F  ->  °C          =  100
 25 °C  ->  K           =  298.15
```

4.2 A bare number matches only no_unit

A value with no unit only ever matches a target that is itself `no_unit`.

A bare number is dimensionally compatible with `%` and with `ppm`, so without that fence a policy naming `"%"` would deliver `0.25` as `25` — the silent factor-of-a-hundred the format exists to prevent, arrived at through the machinery meant to prevent it. Under a policy with `targets = ["%"]`, a bare `0.25` is therefore delivered untouched, with `converted == false`.

The fence is not "never convert dimensionless things": `ppm -> %` is a real and wanted conversion. It is specifically about values that carry no unit at all.

4.3 Exact, or not at all

Nothing approximate is ever delivered. An exact **rational** and a terminating **positional expansion** are different things, and only the second can be handed over as digits:

```
100 mph   ->  m/s   base 10    44.704
100 km/h  ->  m/s   base 10    error_unit_inexact      (250/9)
212 °F    ->  °C    base 10    100
100 °F    ->  °C    base 10    error_unit_inexact      (340/9)
 12 in    ->  m     base 10    0.3048
  1 m     ->  k~m   base 10    0.001
```

Three things follow. Metric-to-metric is where the trouble is, not customary-to-metric: `km/h -> m/s` carries a factor of 5/18 and does not terminate, while `mph -> m/s` is exactly `44.704` and does. Whether a conversion terminates depends on the **value** as well as the units — `212 °F` converts cleanly and `100 °F` does not, through the same 5/9 slope. And the output base matters independently: `1 m -> k~m` is `0.001` in base 10 and non-terminating in base 2.

This is what `on_inexact` exists to shape; see 2.4.

4.4 Quantity kinds keep the dimensionless units apart

Dimensionless does not collapse into one bucket. The library carries a quantity-kind vector alongside the dimension vector, so decibels, the pH scale, angles, turbidity and practical salinity are each their own kind and do not convert into one another:

```
dB   ->  %          refused
pH   ->  no_unit    refused
B/s  ->  Hz         refused
rad  ->  °          allowed
```

Normalisation needs no hand-maintained skip list to keep these apart. The hazards are narrower than the registry suggests: currencies (4.1) and unitless values (4.2), not the logarithmic scales.

---

5. Writer-side policy

5.1 Validation only

The same policy object drives `bvnr_writer_set_unit_policy`, but only `require_unit` and `require_dimension_of` are accepted. A policy carrying `targets`, `normalise`, `base` or `on_inexact` is rejected outright rather than half-honoured.

`bvnr_write_event` fails with `error_unit_mismatch`, and the writer latches it like any other write error — query it with `bvnr_writer_get_error`.

The unit checked is the one the value will carry in the output, whether it was given inline or as a parameter of its type annotation.

5.2 Why the producing side carries the promise

A reader can only reject a document somebody has already written. Only the writer can stop a bare number reaching a file in the first place, which is what "hand the file to anyone and they have everything required to interpret it" actually depends on.

5.3 Relationship to BVN_UNIT_REDUCE

The writer already has a value-rewriting mode: `BVN_UNIT_REDUCE` in `bvnr_write_flags_t.unit_flags`, which folds prefixes out and rescales the value exactly. The writer-side policy deliberately does not add a second one. Two rewriting paths arriving through different doors, with different rules about exactness, is how two features end up disagreeing about what a document says.

6. Errors

6.1 Rejected before the parse

The setter parses every unit string, so a malformed unit, an out-of-range count or an unusable base is reported by a `false` return **before** the parse begins, rather than as a mid-document error in somebody else's file. A rejected policy leaves the previous one in force.

6.2 Raised during the parse

Error	Raised by
`error_unit_mismatch`	`require_unit` on a bare value; `require_dimension_of` matching nothing; a value at a path a rule names that is not convertible to the rule's unit
`error_unit_inexact`	a conversion that cannot be delivered exactly, under `bvnr_inexact_error`

7. Using a policy

7.1 C

```c
static const bvnr_unit_target_t targets[] = { { "m/s", 0 }, { "°C", 0 } };
bvnr_unit_policy_t p = {0};
p.targets     = targets;
p.num_targets = 2;
p.normalise   = bvnr_normalise_si;
p.on_inexact  = bvnr_inexact_leave;
p.require_unit = true;

if (!bvnr_reader_set_unit_policy(r, &p)) { /* a unit string was malformed */ }
```

7.2 Python

```python
from bovnar import Reader, UnitPolicy

with Reader() as r:
    r.set_unit_policy(UnitPolicy(
        targets=["m/s", "°C"],
        normalise_si=True,
        leave_inexact=True,
        require_unit=True,
    ))
    r.read_file("sensors.bvnr", on_verified=handler)
```

A malformed unit raises `BovnarArgumentError` before a byte is read.

7.3 Command line

`validate`, `events` and `query` take the validation options; `pretty-print` and `convert` take the conversion options as well.

Flag	Field
`--require-unit`	`require_unit`
`--require-dimension <unit>`	`require_dimension_of` (repeatable)
`--require-field <path>=<unit>`	a `bvnr_rule_require` rule
`--unit <unit>`	a target (repeatable, first match wins)
`--field <path>=<unit>`	a `bvnr_rule_convert` rule
`--si`	`bvnr_normalise_si`
`--base <N>`	`base`
`--leave-inexact`	`bvnr_inexact_leave`

---

8. Worked examples

The document used throughout:

```
.inlet = {
    .temperature = 212 °F;
    .speed       = 100 mph;
};
.spare = 0.25;
```

8.1 Deliver a document in SI

```
$ bovnar query --unit m/s .inlet.speed sensors.bvnr
44.704
$ bovnar query --unit °C .inlet.temperature sensors.bvnr
100
```

`.spare` carries no unit, so no target claims it and it is delivered as written.

8.2 Assert a schema without converting

```
$ bovnar validate --require-field .inlet.speed=m/s sensors.bvnr
sensors.bvnr: OK

$ bovnar validate --require-unit sensors.bvnr
Validation failed: unit_mismatch at line 5, col 14
```

The first asserts that `.inlet.speed` is a speed without touching the value — `mph` is convertible to `m/s`, so the assertion holds. The second fails on `.spare`, which is the bare number.

8.3 Refuse to write a bare number

Set the same `require_unit` policy on a writer and the bare value never reaches a file:

```
bvnr_unit_policy_t p = {0};
p.require_unit = true;
bvnr_writer_set_unit_policy(w, &p);
/* bvnr_write_event for a value with no unit now fails with
 * error_unit_mismatch, latched on the writer. */
```

See also

- Unit & Currency Reference — the unit registry and the notation grammar
- Unit Ambiguities — how a unit token is resolved, and the pairs that look interchangeable
- Read/Write API — the C declarations, the `want_unit` hook, and the read flags
- Python Bindings — the `UnitPolicy` dataclass and `Reader.set_unit_policy`

End of Bovnar — Unit Policy Reference (Bovnar spec 1.1).