BOVNAR · V1.1

Bovnar (BVNR) — Frequently Asked Questions

Bovnar (BVNR) v1.1 documentation · FAQ · 2026-06-21

> **Applies to:** Bovnar specification v1.1

Table of Contents

1. General

What problem does Bovnar solve?

A bare `9.81` tells you nothing about whether the value is a float or an integer, 32-bit or 64-bit, and it carries no physical meaning. The type and the unit live outside the data — in an external schema, a naming convention, or documentation — and that gap is where unit-confusion failures hide.

Bovnar is self-describing at the per-value level: every value in a `.bvnr` stream carries its type family, bit-width, numeric base, and physical unit inside the same byte stream. No

external schema is required. You can hand a `.bvnr` file to anyone and they have everything they need to interpret it exactly as the author intended.

Is Bovnar a binary or a text format?

It is primarily a text format. The syntax is UTF-8 prose that is readable and writable by hand. The one binary escape mechanism — the octet stream (`0x00` framing) — lets you embed opaque binary payloads without Base64 overhead. Outside of octet-stream regions the parser enforces strict UTF-8 validity.

Does Bovnar perform unit conversion or dimensional analysis?

No. The unit annotation is parsed, validated, and delivered to the application through the event API, but Bovnar itself never converts between compatible units. That is deliberately left to the application. The Python bindings expose helper functions (`unit_convert_factor`, `convert_value`, `units_compatible`) that applications can use to implement dimensional checking and conversion on top of the parsed unit data.

Is there an official file extension?

Yes. The canonical extension is `.bvnr`.

Which version of the specification does the reference implementation target?

Spec 1.1 — the additive features documented here (the `#!bovnar` version directive, `\u`/`\x` escapes, the `datetime` family, and reference array indexing). Spec 1.0 remains the frozen, stable baseline: a document that declares no `#!bovnar` directive is treated as 1.0, and every 1.0 document parses unchanged. `bvnr_version_string()` reports the library version (`1.1.0`) and `bvnr_spec_version()` the highest spec it understands.

---

2. File Format and Syntax

What is the minimum valid Bovnar file?

An empty file (zero bytes) is valid. A file with a single assignment is the practical minimum:

```
.key = value;
```

Why do keys start with a dot?

The leading dot is a hard syntactic marker. It unambiguously separates key tokens from value tokens anywhere in the stream, including inside structs and arrays-of-structs. The parser uses it as the entry point for the identifier state machine.

Can I use Unicode characters in key names?

Yes, with restrictions. UTF-8 leader bytes in the range `0xC3–0xF4` are valid at both the start and body of an identifier, covering most non-ASCII letters. The one excluded range is `0xC2` (U+0080–U+00BF), which is explicitly rejected everywhere in identifiers. The following ASCII punctuation characters are also hard errors inside identifier bodies:

```
! " # $ % & ' ( ) * , . / : ; < = > ? @ [ \ ] ^ ` { | } ~
```

The characters `+`, `-`, and `_` are allowed in identifier bodies (but not as the first character, except for `_`).

Can a key start with a digit?

No. Identifiers must begin with a letter (`A–Z`, `a–z`), an underscore, or a valid UTF-8 leader byte. A leading digit is a hard parse error.

Are comments supported?

Yes. A `#` character starts a comment that runs to the end of the line. Comments are legal anywhere whitespace is legal — before the first assignment, between tokens, and at the end of a value line:

```
# full-line comment
.port = 5432;   # inline comment
```

Is whitespace significant?

No, except as a token separator and (required) separator between a scalar value and an inline unit suffix. Indentation and blank lines are purely cosmetic and are freely interleaved between all tokens.

Is a UTF-8 BOM accepted?

A UTF-8 BOM (`EF BB BF`) is accepted only at byte offset 0 of the stream. A BOM appearing later in a string is valid UTF-8 and passes through without error. For BOMs in other positions the error code depends on where the BOM appears:

- A BOM found **inside the first comment line** (the dedicated `first_comment_*` states) produces `error_invalid_byte_order_mark`.
- A BOM byte (`0xEF`) found **after the first comment line** but before the first `.` identifier (the `first_bom` state) is not handled in the state table and produces `error_unexpected_input_byte`.

A BOM appearing inside any subsequent comment is valid UTF-8 and accepted without error.

Can a document declare which bovnar version it uses?

Yes, since spec 1.1. Put a directive on the first line:

```
#!bovnar 1.1
.x = 42;
```

A document with **no** directive is treated as spec **1.0** — the frozen baseline grammar — so existing files need no change. The directive only opts in to a newer version.

It is recognised only as the very first comment (after an optional BOM and whitespace). A reader records the declared version (`bvnr_reader_get_declared_version`, `bovnar validate` reports it, and the Python `Reader.declared_version` / `bovnar.peek_version()` expose it), and a reader opened with `strict_version` rejects a version it does not support with `error_unsupported_spec_version`. A malformed directive is `error_invalid_spec_version`.

Wait — isn't a comment supposed to be meaningless? How can `#!bovnar …` matter?

The version directive is the single, deliberate exception. Everywhere else a comment is semantically inert. The directive is shaped like a comment on purpose: that is exactly what makes it backward compatible — a spec-1.0 reader (which knows nothing about versions) skips it as a plain comment and parses the rest of the document normally, while a 1.1+ reader additionally recognises the `#!bovnar` prefix on the first line and acts on it. Nothing else about comment semantics changes, and a `#!bovnar …` on any line other than the first is still just a comment.

---

3. Type System and Annotations

What are the seven type families?

Keyword	Description
`uint`	Unsigned integer
`sint`	Signed integer
`float`	IEEE 754 binary floating-point
`float_fix`	Q-format signed fixed-point
`float_dec`	IEEE 754-2008 decimal floating-point
`utf8`	UTF-8 string
`bool`	Boolean (`true` / `false`); takes no parameters

Spec 1.1 adds an eighth, `datetime` (see below).

Is there a date/time type?

Yes, since spec 1.1: the `datetime` family. A `datetime` value is a **signed integer count of seconds since a named epoch** — a timestamp:

```
#!bovnar 1.1
.created = <datetime:64,unix> 1750000000;
```

The epoch parameter is one of `unix` (default), `tai`, `gps`, `mjd`, `ntp`, `galileo`, `glonass`, `y2000`, `beidou`. The value validates like `sint` (negative = before the epoch); a numeric base/unit/`q` is `error_illegal_value_type` and a fractional value is `error_type_value_mismatch`. Being a 1.1 feature it requires a `#!bovnar 1.1` declaration. Convert to civil time with the `bvn_datetime.h` helpers.

You can also write the value as an **ISO-8601 literal** instead of a raw integer — `2026-06-15`, `2026-06-15T12:00:00`, a trailing `Z`, or a numeric `±HH:MM` offset, with an optional fractional second:

```
#!bovnar 1.1
.created = 2026-06-15T12:00:00Z;        # bare literal infers <datetime:64,unix>
.local   = 2026-06-15T12:00:00+02:00;   # offset folds to 10:00:00Z
.logline = 2026-06-15T12:00:00.123Z;    # fraction preserved as a string; round-trips
```

It is converted to the epoch-seconds carrier at parse time (the integer is what is stored, so round-trips are idempotent). A bare literal with no annotation infers `<datetime:64,unix>`. A `±HH:MM` offset shifts the time to true UTC before the conversion. A fractional part (any number of digits) does not change the whole-second carrier, but the digits are kept verbatim — consumers read them as a string (`bvnr_data_t.frac_data`, or

`bvn_dom_get_datetime_fraction()`), and a value carrying a fraction is pretty-printed back as an ISO literal so it round-trips. For sub-second values you compute on, use a finer integer carrier (e.g. milliseconds). The UTC→epoch conversion is leap-second correct for the civil epochs and `tai` (the offset is folded before `tai`'s leap-second lookup); the atomic GNSS epochs (`gps` / `galileo` / `glonass` / `beidou`) reject a literal (`error_datetime_literal_unsupported_epoch`) — give those an integer carrier. A malformed or out-of-range literal is `error_invalid_datetime_literal`.

Timestamp vs. duration — don't confuse them. A timestamp (an instant) is a `datetime`; a duration (an elapsed amount) is a plain number with a time unit, e.g. `<float:64,s> 2.5` (2.5 seconds) or `<uint:32,h> 8` (8 hours). The time units (`s`, `min`, `h`, `d`, …) measure spans; the `datetime` family names a point on a timeline.

What is default type synthesis?

When no explicit type annotation is present, the parser synthesises one based on the literal:

- Positive integer literal → `uint:64`
- Negative integer literal → `sint:64`
- Literal containing `.` or `e` → `float:64`
- Quoted string → `utf8`
- Boolean keyword (`true` / `false` / `on` / `off`) → `bool`
- ISO-8601 datetime literal (spec 1.1) → `datetime:64,unix`

This is called default type synthesis and is convenient for configuration files and quick hand-authored data. For precision-sensitive use cases, always write an explicit annotation.

Where exactly does the type annotation go?

Between `=` and the value — never on the key, and never after the value:

```
.port = <uint:16> 443;       # correct
.port<uint:16> = 443;        # WRONG — hard error
.port = 443 <uint:16>;       # WRONG — hard error
```

Can annotation parameters appear in any order?

Yes. Width, base, and unit are distinguished by their syntactic form, not by position. The following three are equivalent:

```
.x = <uint:32,_16,m> "1F4";
#    <uint:_16,m,32>     — identical (parameter order is free)
#    <uint:m,32,_16>     — identical
```

What widths are valid for each type family?

Family	Valid widths
`uint`, `sint`	Any positive integer up to `BVN_MAX_INT_WIDTH` (32768), e.g. `<uint:12>` is legal for a 12-bit ADC
`float`	`0` (→64), `16`, and any multiple of `32` up to `32768`
`float_fix`	`0` (→64), `16`, `32`, `64`, `128`, `256`
`float_dec`	`0` (→64), `16`, `32`, `64`, `128`, `256`
`datetime`	Same as `sint` — any positive integer up to `BVN_MAX_INT_WIDTH` (the carrier is a signed epoch-seconds integer); `0` →64

Width `0` always means "use the default," which is 64 bits for all families. Width `8` is legal for `uint`/`sint`/`datetime` but is a hard error for all float families. A non-multiple-of-32 (other than 16) width for `float` is `error_illegal_value_type`.

What is the `float_fix` (Q-format) type and how is it annotated?

`float_fix` encodes a value as a signed integer with a declared number of fractional bits. The mathematical value is `raw_integer × 2⁻Q`. The annotation requires a `qN` parameter specifying fractional bits:

```
.pid = <float_fix:32,q16> -1.5;    # 32-bit, 16 fractional bits
.adc = <float_fix:16,q8,°C> 25.0;  # 16-bit Q8 with a unit
```

Q must satisfy `0 ≤ Q < effective_width`. `<float_fix:16,q16>` is illegal. The base parameter (`_N`) is forbidden for `float_fix`.

What is `float_dec` and when should I use it?

`float_dec` is IEEE 754-2008 decimal floating-point. Binary floats cannot represent `0.1` exactly; decimal floats can. Use `float_dec` in financial, metrological, or billing contexts where exact decimal representation matters. The base parameter is forbidden for `float_dec` as well.

Can I annotate a string value with a type?

Yes, with `<utf8>`:

```
.label = <utf8> "hello";
```

This is redundant — a bare quoted literal is already synthesised as `utf8` — but it makes the type explicit, which is useful for documentation and tooling. `utf8` is a parameterless family: a width, base, q, or unit parameter inside a `<utf8>` annotation is `error_illegal_value_type`.

---

4. Units

What is the purpose of the unit annotation?

Physical units are a first-class part of the type system, not a comment field. Attaching a unit to a value makes the data self-documenting and enables the consuming application to perform dimensional verification and conversion. The parser fully validates every unit expression and exposes the structured result through the `value_unit_t` field of the `ev_data` event.

How do I write a simple unit?

Place the unit symbol inside the angle brackets after the other type parameters:

```
.distance  = <uint:64,m>      384400000;
.voltage   = <float:32,V>     3.3;
.frequency = <float:64,Hz>    2400000000;
```

How do SI prefixes work, and why is the `~` separator mandatory?

An SI prefix is written before the unit symbol with a mandatory `~` separator: `k~m` (kilometer), `m~V` (millivolt), `G~Hz` (gigahertz). The `~` is required because without it there is no general way to distinguish a two-character unit symbol from a one-character prefix followed by a one-character unit — `mV` would be ambiguous. The `~` resolves all such cases unambiguously:

```
.k_ohm = <float:32,k~Ω> 4.7;
.micro = <float:32,µ~s> 50.0;
.giga  = <float:64,G~Hz> 2.4;
```

IEC binary prefixes (`Ki`, `Mi`, `Gi`, `Ti`, …) follow the same rule:

```
.ram  = <uint:64,Gi~B> 8;
.disk = <uint:64,Ti~B> 2;
```

What is the difference between `M~B` (megabytes) and `Mi~B` (mebibytes)?

`M~` is the SI decimal prefix (mega = 10^6). `Mi~` is the IEC binary prefix (mebi = 2^{20} = 1,048,576). The difference matters for storage:

```
.link_rate = <uint:32,M~b>  1000;   # 1 000 × 10⁶ bits
.cache     = <uint:64,Mi~B> 512;    # 512 × 2²⁰ bytes
```

How do I write compound units like m/s² or kg·m/s²?

Use `*` or the middle-dot `·` (U+00B7) for multiplication and `/` for division:

```
.velocity     = <float:64,m/s>          9.81;
.acceleration = <float:64,m/s²>         9.81;
.force        = <float:64,k~g·m/s²>     9.81;
.energy       = <float:64,k~g·m²/s²>    1000.0;
```

The `/` separator is a one-way switch: once written, every subsequent component is in the denominator. To place a component back in the numerator after a divisor, use a negative exponent instead:

```
.force_alt = <float:64,k~g·m·s⁻²> 9.81;   # identical to k~g·m/s²
```

The maximum number of unit components in a compound unit is 8; exceeding that is `error_unit_illegal`.

Can exponents be written in ASCII instead of Unicode superscripts?

Yes. Both forms are accepted and produce identical internal representations:

```
.area1 = <float:64,m²>   100.0;   # Unicode superscript
.area2 = <float:64,m^2>  100.0;   # ASCII caret form — same result
.inv1  = <float:64,s⁻¹>  50.0;
.inv2  = <float:64,s^-1> 50.0;
```

Only a single ASCII digit is supported after `^`; multi-digit exponents are a parse error.

What does `no_unit` mean, and how does it differ from omitting the unit?

Both mean "dimensionless," but they produce distinct internal states:

- Omitting the unit parameter → `BVN_UNIT_NO_PREFIX(bu_none)`: `num_components == 1`, `base == bu_none`.
- Writing `no_unit` explicitly → `BVN_UNIT_NONE`: `num_components == 0`.

Both compare as compatible via `bvn_units_compatible` and both serialise to `"no_unit"` via `bvn_unit_to_string`. The distinction is expressive: explicit `no_unit` signals that the author actively chose dimensionless, whereas an omitted unit might mean the author simply did not think about units. For documentation-grade data, prefer the explicit form.

Can I write a unit as an inline suffix instead of inside the annotation?

Yes, for scalar values in non-array context. Write the unit directly after the value literal, separated by at least one whitespace character:

```
.speed   = 9.81 m/s;       # no annotation; inline unit adopted
.mass    = 70.0 k~g;
.storage = 65536 B;
```

The inline unit is forbidden inside array elements — any letter or underscore following an array-element value is `error_unexpected_input_byte`. If both an annotation unit and an inline unit are present, they must be identical; a mismatch is `error_unit_mismatch`.

How many base units does Bovnar support?

163 named base units across the following categories:

- **7 SI base units** — second, meter, gram, ampere, kelvin, mole, candela.
- **21 named SI-derived units** — hertz through katal (Hz, N, Pa, J, W, V, Ω, F, C, S, Wb, T, H, lm, lx, Bq, Gy, Sv, kat, rad, sr).
- **14 non-SI units accepted for use with SI** — liter, minute, hour, day, week, year, degree (angle), degree Celsius, tonne, bar, electronvolt, dalton, astronomical unit, hectare.
- **2 digital units** — bit (`b`), byte (`B`), with IEC binary prefixes (kibi through quebi).
- **13 Imperial/US customary length units** — inch, foot, yard, mile, nautical mile, ångström, light-year, parsec, furlong, fathom, **chain** (`ch`), **rod** (`rd`), and **thou** (thousandth of an inch, alias `mil`).
- **11 Imperial/US customary mass units** — pound, ounce, grain, stone, short ton, long ton, troy ounce, carat, **slug**, **dram** (`dr`), **pennyweight** (`dwt`).
- **6 temperature scales** — degree Fahrenheit (affine), **degree Rankine** (linear, absolute), and the historical **Delisle** (`°De`), **Newton** (`°N`), **Réaumur** (`°Re`), and **Rømer** (`°Ro`) scales (all affine). Degree Celsius is counted among the non-SI accepted units above.
- **6 pressure units** — atmosphere, mmHg, Torr, psi, **inch of mercury** (`inHg`), and **atmosphere technical** (`at`, = 1 kgf/cm² = 98 066.5 Pa).
- **5 energy units** — calorie, BTU, erg, therm, **foot-pound** (`ft_lb`).

- **2 power units** — horsepower (`hp`), **metric horsepower** (`PS`, also `CV`; = 735.49875 W).
- **4 force units** — pound-force, dyne, kip, **kilogram-force** (`kgf`).
- **1 acceleration unit** — **standard gravity** (`gn`, = 9.80665 m/s², exact).
- **3 speed/frequency/rotation units** — knot, **revolutions per minute** (`rpm`), **revolution** (`rev`, full angular turn = 2π rad).
- **15 US and UK volume units** — gallon, gallon (UK), quart, pint, cup, fluid ounce, tablespoon, teaspoon, barrel, **US gill** (`gi`), **imperial gill** (`gi_uk`), **fluid dram** (`fl_dr`), **minim**, **peck** (`pk`), **bushel** (`bsh`).
- **3 UK imperial volume units** — **pint_uk**, **fluid_ounce_uk**, **quart_uk**.
- **2 area units** — acre, barn.
- **3 angle units** — arcminute, arcsecond, gradian; plus **revolution** (see rotation above).
- **8 CGS units** — poise, stokes, gauss, maxwell, oersted, stilb, phot, galileo.
- **3 radiation units** — curie, röntgen, rem.
- **2 logarithmic units** — neper, **decibel** (`dB`).
- **2 electrical power units** — **var** (reactive power), **volt-ampere** (`VA`, apparent power).
- **2 time extensions** — **month** (`mo`, Julian month = 2 629 800 s), **fortnight** (`fn` = 14 d = 1 209 600 s).
- **2 textile linear density units** — **tex** (1 g/km = 10^{-6} kg/m, ISO 1144), **denier** (`den`, 1 g/9 000 m; 9 den = 1 tex).
- **13 Old German units** — **Pfund** (`Pfd`), **Zentner** (`Ztr`), **Doppelzentner**, **Lot** (mass); **Prussian line, Zoll, Fuß, Elle, Rute**, **Klafter**, and **German (geographical) mile** (length); **Morgen** (area); **Scheffel** (volume). None accept an SI or IEC prefix.
- **7 surveying & signalling units** — **US survey foot**, **league**, **cable length**, **hand** (length); **quintal**, **scruple** (mass); **baud** (`Bd`, signalling rate).
- **6 ratio / proportion units** — **percent** (`%`), **per mille** (`‰`), **per myriad** (`‱`), **per cent mille** (`pcm`), **ppm**, **ppb** — dimensionless scaling factors that take no prefix.

The `bu_gram` base unit is used for mass so that the `k~` prefix can carry the kilo: `k~g` = kilogram. The Rankine symbol is `Ra` (not `R`, which is reserved for röntgen). Thou accepts `mil` as an alternative spelling. `var` and `VA` share the same SI dimensional signature as watt but are kept distinct for physical clarity. `rpm` has the same SI dimension as `Hz` (s^{-1}) but a distinct base unit for semantic clarity in rotational contexts. `at` (atmosphere technical) must not be confused with `atm` (standard atmosphere): 1 at = 98 066.5 Pa; 1 atm = 101 325 Pa.

---

5. Numbers and Special Values

Do I need to quote non-decimal numbers?

Yes. Bovnar's lexer distinguishes bare words (symbols) from numbers by context. A bare `FF` in value position is a symbol, not a hex number, and will produce `error_type_value_mismatch`. Any non-decimal value must be a quoted string:

```
.hex = <uint:8,_16>  "FF";       # correct — 255
.hex = <uint:8,_16>  FF;         # WRONG — FF is a symbol
.bin = <uint:8,_2>   "11010110"; # correct
```

For signed non-decimal values, the minus sign goes inside the string:

```
.neg = <sint:32,_16> "-7FFFFFFF";
```

What base systems are supported?

For `uint` and `sint`: every base from `_2` through `_62` (consecutive). Bases `_64` (standard Base64) and `_85` (Ascii85) are also supported, but **for `uint` only** — their alphabets use `+`/`-` as digits, leaving no sign character, so a signed value in these bases is a hard error (`error_illegal_value_type`). For `float`: `_10` (decimal, the default) and `_16` (hexadecimal) only. `float_fix` and `float_dec` do not accept a base parameter at all — specifying one is a hard error.

What are the special number literals?

Three special values bypass normal numeric parsing. They are bare reserved keywords — no sigil — and, like `null`/`true`/`false`, are reclassified out of the symbol space by the validator. Only these exact spellings are reserved; any other bare word (e.g. `infinity`, `nans`) is an ordinary symbol:

```
.nan = nan;
.inf = inf;
.neg = ninf;     # negative infinity
```

They are valid in both typed and untyped context and may be combined with any float type annotation. A special-number keyword takes no inline unit suffix — give its unit via the annotation (`<float:64,m/s> inf`).

Are leading zeros legal in decimal integers?

Yes. `007` parses as the integer `7`. There is no octal interpretation — the leading zeros are ignored.

Are there shorthand float forms like `-.5` or `123.`?

Yes. Both leading-dot (`.5` or `-.5`) and trailing-dot (`123.`) forms are valid for `float:64`:

```
.a = -.5;     # -0.5 as float:64
.b = 123.;    # 123.0 as float:64
```

---

6. Strings

What escape sequences are defined?

Exactly seven:

Escape	Meaning
`\t`	Horizontal tab
`\n`	Line feed
`\v`	Vertical tab
`\f`	Form feed
`\r`	Carriage return
`\"`	Double quote
`\\`	Backslash

In **spec 1.0** that is all — any other character after a backslash is `error_illegal_escape_sequence`.

A **spec 1.1** document (declaring `#!bovnar 1.1`, see §3.4) adds two more:

Escape	Meaning
`\xHH`	the single byte `HH` (two hex digits)
`\u{H…}`	Unicode scalar `U+H…` (1–6 hex digits), UTF-8 encoded

`\u{…}` rejects surrogates and values above `U+10FFFF` (`error_invalid_codepoint`). `\x` writes a raw byte but the string must stay valid UTF-8, so `"\xC3\xA9"` is `"é"` while a lone `"\xFF"` is `error_invalid_utf8_byte` — for arbitrary binary data, still use an octet stream. Both `\x` and `\u` are **1.1-only**: in a 1.0/unversioned document they remain `error_illegal_escape_sequence`, so existing documents are unaffected.

Can strings span multiple lines?

Yes. Raw whitespace bytes (HT, LF, VT, FF, CR) are accepted unescaped inside string literals:

```
.poem = "roses are red
violets are blue";
```

How does string concatenation work?

Adjacent string literals separated only by whitespace or comments are concatenated at lex time. This is the idiomatic way to split a long string across lines:

```
.url  = "https://" "api.example.com" "/v1";
.guid = <uint:128,_16> "deadbeef"
                       "cafebabe"
                       "01234567"
                       "89abcdef";
```

The combined byte length must not exceed `max_string_length` (default 65535).

Is the `<utf8>` annotation ever required?

No. A bare quoted literal is always synthesised as `utf8`. The annotation is purely optional and expressive.

---

7. Arrays

What is the basic array syntax?

Comma-separated values inside square brackets:

```
.primes = [2, 3, 5, 7, 11];
```

How are multi-dimensional arrays written?

Rows are separated by `/`:

```
.matrix = [1, 2, 3]/[4, 5, 6]/[7, 8, 9];
```

Each `/[…]` is a new row dimension. The parser emits `ev_array_row_start` and `ev_array_row_end` events for each row and `ev_array_dim_start` for each `/` separator.

Can array elements have different types?

Array elements must be **homogeneous** (spec 1.0): every non-null element shares the same kind, and a bare array of numbers shares the same physical dimension. The numeric encodings may still mix, since they all denote the same dimension:

```
.mixed = [<uint:8> 1, <sint:8> -1, <float:32> 3.14];   # valid: all dimensionless
```

But genuinely different kinds (`[1, "two"]`) or dimensions (`[<float:64,m> 1.0, <float:64,k~g> 2.0]`) are rejected with `error_array_element_type_mismatch`. Heterogeneous data goes in a **struct**, not an array.

Homogeneity is a **materialised-document** rule: it is checked over the whole value once it is assembled, so it is enforced by the DOM parser (`bvn_dom_parse` / `bovnar.dom_parse`), not by the streaming SAX `Reader` (nor by `loads`, which is built on it) — the streaming layer surfaces per-value type/unit errors as the bytes arrive but cannot see sibling elements. Parse through the DOM tier (or the `bovnar` CLI's `convert`) if you need these rules enforced. A whole-array annotation (placed before `[`) is the default type for elements that do not carry their own annotation:

```
.ports = <uint:16> [80, 443, 8080];
```

How do null elements work inside arrays?

Leading, trailing, or consecutive commas produce null elements:

```
.sparse = [, 1, , 2, ];   # null, 1, null, 2, null — five elements
```

Can I nest arrays?

Yes, inner arrays are just element values:

```
.nested = [[1, 2], [3, 4]];    # valid; rectangular sub-arrays
.ragged = [[1, 2], [3, 4, 5]];  # error_array_row_size_mismatch — sibling sub-arrays
differ in length
```

Note: since spec 1.0 sibling sub-arrays must match in length (and element shape), so bare arrays and matrices are rectangular — `[[1, 2], [3, 4, 5]]` is rejected. `error_array_row_size_mismatch` fires both when the `/`-rows of one array disagree (`[1,2,3]/[4,5]`) and when ragged sibling sub-arrays disagree (`[[1,2],[3,4,5]]`).

Are inline unit suffixes allowed inside arrays?

No. The inline unit suffix is forbidden at the lexer level for all array elements. Any letter or underscore following an array-element value token is `error_unexpected_input_byte`.

8. Structs

What is the struct syntax?

A struct groups zero or more assignments inside `{…}`. Every field inside is a complete `.key = value;` assignment. The struct value at the parent level ends with `};`:

```
.server = {
    .host = "db.internal";
    .port = <uint:16> 5432;
};
```

How deeply can structs nest?

Up to `max_struct_nesting` levels — the reference reader defaults to 64 and the field (a `uint8_t`) has a hard cap of 255. An empty struct (`{}`) is valid. An unmatched closing brace is `error_illegal_struct_close`.

Are structs valid as array elements?

Yes:

```
.sensors = [
    {.id = <uint:8> 0; .value = <float:32,°C> 27.3;},
    {.id = <uint:8> 1; .value = <float:32,°C> 31.7;}
];
```

9. Null, Symbols, and References

What is a null value?

An explicit absence of value — nothing between `=` and `;`, or the reserved keyword `null`:

```
.nothing    = ;              # null, no type annotation
.also_null  = null;          # the null keyword — identical to the empty slot
.typed_null = <uint:32> ;    # null with an explicit type
```

Does Bovnar have booleans?

Yes. `true`, `false`, `on`, and `off` are reserved keywords carrying the `bool` type family (`on` ≡ `true`, `off` ≡ `false`). A bare boolean synthesises a `<bool>` annotation; `<bool>` takes no

parameters and accepts only those four keywords. They serialize canonically as `true` / `false`:

```
.enabled = true;
.debug   = off;          # == false
.typed   = <bool> on;    # explicit; on == true
```

What is a symbol?

A bare, unquoted word in value position — for application-defined enums and the like. The reserved keywords `null`, `true`, `false`, `on`, `off` and the special floats `nan`, `inf`, `ninf` are not symbols (a word that merely starts with one, like `Monday`, `ontology`, or `infinity`, is):

```
.status  = ok;
.mode    = read_only;
.day     = Monday;
```

Symbol bodies follow the same character rules as identifier bodies. The key distinction from an identifier is context: identifiers appear after the leading dot, symbols appear as values.

What is a reference?

A dotted path to another key in the document, introduced by `&`. The parser stores the path as a string token; it does not resolve it — resolution is left entirely to the application:

```
.host       = "db.internal";
.conn_host  = &.host;             # stored as ".host"
.deep       = &.config.db.host;   # multi-segment path
```

Can a reference index into an array?

Yes, since spec 1.1: a reference path may carry `[N]` index suffixes.

```
#!bovnar 1.1
.matrix  = [10, 20, 30]/[40, 50, 60];
.row0c1  = &.matrix[0][1];        # stored as ".matrix[0][1]"
```

As with the rest of a reference, the index is stored **verbatim and unresolved**; it is interpreted only when the application resolves the path against the materialised tree (`bvn_dom_lookup`, also used by the CLI `query` command), where `&.matrix[0][1]` resolves to `20`. A flat `/`-row matrix is addressed `[row][col]`, a 1-D array `[i]`, and nested arrays descend one index per level; a partial/out-of-range index simply doesn't resolve. Being a

1.1 feature, a `[` in a reference is `error_unexpected_input_byte` in a 1.0/unversioned document.

---

10. Octet Streams

When should I use an octet stream?

When you need to embed raw binary data without Base64 overhead. A NUL byte (`0x00`) in value position switches the parser into binary chunk mode. The UTF-8 validator is suspended for the duration of the binary region.

In practice you will not hand-author octet streams. Use the writer API, which generates the correct framing automatically:

```
0x00                      ← stream open
  0x01 <LL> <LL> <data>   ← chunk: tag 0x01, length as little-endian uint16, data
0x00                      ← stream close
```

A length value of `0x0000` means exactly 65536 bytes. Any tag byte other than `0x01` (chunk) or `0x00` (close) inside a stream is `error_octet_stream_out_of_sync`.

---

11. Error Handling and Debugging

What information does a parse error carry?

Every error includes:

- `error_code_t` — a named code (e.g. `error_value_out_of_range`)
- Line number and column number in the input stream
- Byte offset from stream start
- The raw byte value that triggered the error

Retrieve these after a failed `bvnr_read`:

```
fprintf(stderr, "%s at line %" PRIu64 " col %" PRIu64
        " offset %" PRIu64 " byte 0x%02X\n",
        bvn_error_to_string(bvnr_reader_get_error(r)),
        bvnr_reader_get_error_line(r),
        bvnr_reader_get_error_column(r),
        bvnr_reader_get_error_offset(r),
        bvnr_reader_get_error_byte(r));
```

What is `continue_on_error` mode and when should I use it?

When `continue_on_error` is set in `bvnr_read_flags_t`, a parse error invokes the `on_error` callback and enters a resync state machine that skips bytes while tracking bracket and brace nesting. Once the resync state machine finds a `;` at the original nesting depth, normal parsing resumes. The recovery count (accessible via `bvnr_reader_get_recovery_count`) is incremented at error entry, not on resync completion.

Use this mode for log streams and unreliable transports where a single malformed assignment should not discard the rest of the file. For configuration and data serialization, disable it and fail fast.

What does `error_type_value_mismatch` mean?

The value token is incompatible with the declared type annotation. The most common cause is a non-decimal number written as a bare word (symbol) instead of a quoted string:

```
.x = <uint:8,_16> FF;   # FF is a symbol → error_type_value_mismatch
.x = <uint:8,_16> "FF"; # correct
```

What causes `error_unit_mismatch`?

An inline unit suffix and a type-annotation unit are both present but resolve to different `value_unit_t` values:

```
.bad = <float:64,m> 1.0 s;   # annotation says m, inline says s → error
```

Semantically equivalent notations that parse to the same internal representation (e.g. `m/s` and `m·s⁻¹`) compare as equal and do not trigger this error.

What are the most common mistakes?

Mistake	Error produced
Annotation on the key side of `=`	`error_unexpected_input_byte`
Non-decimal value not quoted	`error_type_value_mismatch`
Signed value in an unsigned type	`error_value_out_of_range`
Integer overflow (e.g. `<uint:8> 256`)	`error_value_out_of_range`
`<float:8>` or a non-16, non-multiple-of-32 width for `float`	`error_illegal_value_type`
Base parameter on `float_fix` or `float_dec`	`error_illegal_value_type`
`q ≥ width` in `float_fix`	`error_illegal_value_type`
Empty unit component (`m//s`)	`error_unit_illegal`

Mistake	Error produced
More than 8 unit components	`error_unit_illegal`
Inline unit suffix inside an array	`error_unexpected_input_byte`
Unknown escape sequence (`\0`, `\uXXXX`; `\x` / `\u{}` outside spec 1.1)	`error_illegal_escape_sequence`
`\u{…}` surrogate or `> U+10FFFF` (spec 1.1)	`error_invalid_codepoint`
Unmatched `}`	`error_illegal_struct_close`
Empty key (`.= value;`)	`error_empty_identifier`

---

12. C API

What is the overall reader lifecycle?

```
bvnr_reader_create()
  → bvnr_source_from_fd() or bvnr_source_from_mem()
  → bvnr_open_read_source() or bvnr_open_read_mem()
  → bvnr_read()
  → bvnr_reader_destroy()
```

The reader does not allocate during `bvnr_read`; all buffering is internal to the reader struct. The `bvnr_read_flags_t` struct is read only during `bvnr_open_read_source` and may live on the stack.

Which callback should I implement — `on_verified` or `on_unverified`?

Almost always `on_verified`. This callback receives fully validated events after semantic checking. `on_unverified` fires before validation and receives raw token events; use it only for diagnostics or partial pre-inspection.

There is an important asymmetry for `ev_type_annotation_start`: - `on_unverified` receives it with raw annotation bytes but no `value_type` or `value_unit` populated. - `on_verified` receives it with `value_type` and `value_unit` filled in.

All other type-annotation events and `ev_data` are emitted to both callbacks simultaneously.

Do callbacks need to return anything?

Yes. Both `on_verified` and `on_unverified` must return `bool`. Return `true` to continue parsing; return `false` to abort — the parser stops and `bvnr_read` returns `false` with `error_scanner_callback_failed`.

How do I read a value from the `ev_data` event?

The `bvnr_data_t` struct provides the raw bytes in `data` / `length`. Use the `bvn_parse_*` helpers to convert them to native C types:

```
uint64_t v;
bvn_parse_uint64(vbuf, d->value_type, &v);

double f;
bvn_parse_double(vbuf, d->value_type, &f);
```

The `d->value_type` contains `family`, `width`, and `base_or_q`. Use `bvn_effective_width(d->value_type)` to get the resolved width (handles the `width == 0` → 64 default).

How is the writer API structured?

It is symmetric with the reader. You construct events and push them through `bvnr_write_event`, then call `bvnr_write_finish`:

```
bvnr_writer_create()
  → bvnr_sink_to_fd() or bvnr_sink_to_mem()
  → bvnr_open_write_sink() or bvnr_open_write_mem()
  → bvnr_write_event() × N
  → bvnr_write_finish()
  → bvnr_writer_destroy()
```

Use `bvnr_write_type_annotation(w, vt, vu)` as a convenience to emit the full annotation event sequence in one call. Use `bvnr_write_bvnf_base` and `bvnr_write_bvni` for the convenience integer/float helpers that generate the scalar key+value pair in a single call.

Does the library close file descriptors for me?

No. Both `bvnr_source_from_fd` and `bvnr_sink_to_fd` leave ownership of the file descriptor with the caller. Close it yourself after the reader or writer is destroyed.

What is the `max_array_nesting` limit and why is it capped at 255?

The lexer stores per-nesting-level state in a fixed array of 256 entries. `max_array_nesting` is a `uint8_t` field, so values above 255 cannot be represented; no runtime rejection is performed. Zero-initialising `bvnr_read_flags_t` is safe: a zero value causes the reader to substitute the internal default of **64**. The hard maximum is 255. The same default and cap apply to `max_struct_nesting`.

---

13. Python Bindings

Do the Python bindings require a compiled extension module?

No. The bindings are pure `ctypes` and load `libbvnr.so` at import time via `ctypes.CDLL`. No compilation step beyond building the C library is needed.

How does the library discovery order work?

1. `LIBBOVNAR_PATH` environment variable — absolute path to the `.so`.
2. `LIBBOVNAR_DIR` environment variable — directory containing the `.so`.
3. `ctypes.util.find_library('bovnar')` — standard `ldconfig` / `LD_LIBRARY_PATH` search.
4. In-tree build paths relative to `_ffi.py` (`../../build/`, etc.).

If none of these resolves the library, `BovnarLibraryNotFound` is raised with the list of searched paths.

What is the difference between `loads` / `dumps`, the SAX reader, and the DOM?

Interface	When to use
`bovnar.loads` / `bovnar.dumps`	Simple dict serialization — loses type and unit metadata
`Reader` (SAX)	Streaming; low memory; full access to type and unit data via callbacks
`bovnar.dom_parse` / `DomDoc`	Random-access queries on a fully parsed in-memory tree

---

How do I capture state in a SAX callback?

The Python `on_verified` callback receives exactly two positional arguments — the event code and the data payload — with no userdata parameter. Use a closure to capture external state:

```
state = {"current_key": ""}

def on_event(ev, data):
    if ev == Event.ASSIGNMENT_START:
        state["current_key"] = data.raw_str()
    elif ev == Event.DATA:
        print(state["current_key"], "→", data.raw_bytes())
    return True
```

What happens when a callback raises a Python exception?

The exception is captured, the C callback returns `False` to stop parsing, the C call returns, and then the original Python exception is re-raised from `read_mem` / `read_fd`. The exception is not wrapped; it propagates as-is.

How do I access unit information in Python?

The `data.value_unit` field of an `EVENT.DATA` payload is a `ValueUnit` ctypes struct. Use the helper functions to work with it:

```
import bovnar

vu = bovnar.parse_unit("k~g·m/s²")          # parse string → ValueUnit
s  = bovnar.unit_to_str(vu)                   # → "k~g·m/s²"
ok = bovnar.units_compatible(vu_a, vu_b)      # dimensional compatibility
c  = bovnar.unit_convert_factor(vu_from, vu_to)
kelvin = bovnar.convert_value(25.0, vu_celsius, vu_kelvin)
```

`unit_dimension_vector` returns a 7-element list of SI dimension exponents in the order `[m, kg, s, A, K, mol, cd]`.

What exception classes exist?

Exception	Trigger
`BovnarLibraryNotFound`	`libbvnr.so` not found at import
`BovnarParseError`	Parse error (carries `code`, `line`, `column`, `offset`, `byte`)
`BovnarWriteError`	Write error (carries `code`, `offset`)
`BovnarArgumentError`	Invalid argument to a helper function

All are subclasses of `BovnarError`.

What Python version is required?

Python ≥ 3.10. The bindings use `dataclasses`, `enum.IntEnum`, and union-type annotations (`X | Y`).

---

14. Limits and Performance

What are the default size limits and what should I set in production?

All limits are configurable via `bvnr_read_flags_t`. Defaults are permissive.

Field	Default	Suggested production cap
`max_identifier_length`	255	255
`max_string_length`	65535	application-defined
`max_number_length`	65535	65535
`max_symbol_length`	255	255
`max_reference_length`	65535	65535
`max_array_items`	0 (→ 2 147 483 647 internal default)	application-defined
`max_text_bytes`	0 (→ 2 147 483 647 internal default)	application-defined
`max_file_size`	0 (→ unlimited / endless)	`16777216` (16 MiB)
`max_array_nesting`	0 (→ 64 internal default; hard cap 255)	32 or less
`max_struct_nesting`	0 (→ 64 internal default; hard cap 255)	32 or less

Setting a field to `0` causes the reader to substitute a finite internal default — **2 147 483 647** for `max_array_items` / `max_text_bytes` and **64** for both nesting depths. **`max_file_size` is the exception: `0` means unlimited / endless** (no byte-count cap), which is the default so endless streams work out of the box. Production deployments should set `max_file_size` explicitly at minimum. For untrusted input, set `max_array_items`, `max_text_bytes`, and the nesting limits as well.

Does `bvnr_read` allocate heap memory?

No. The reader does not allocate during `bvnr_read`. All buffering is internal to the `bvnr_reader_t` struct, which is allocated once by `bvnr_reader_create`. Event data pointers (`d->data`) point into the reader's internal buffer and are valid only for the duration of the callback invocation.

What throughput can I expect from the parser?

Performance depends heavily on the input profile (scalars, typed values, structs, arrays, units) and hardware. The CLI's benchmark subcommand (`bovnar bench`) can measure MB/s, assignments/s, and events/s across configurable profiles and payload sizes. Run `bovnar bench --profile all --size 1024,65536 --iterations 200` for a representative baseline. Use `--min-overhead` to isolate raw lexer throughput by skipping the `on_verified` callback.

How many unit components can a compound unit have?

A maximum of 8 (`BVNR_MAX_UNIT_COMPONENTS`). Exceeding this limit during parsing is `error_unit_illegal`.

---

15. Why C99 and Not C23?

What is the stated language standard for the reference implementation?

Strict C99 (`-std=c99 -pedantic`). No C11, C17, or C23 extensions are used intentionally.

Why not C23, given that it was finalised in 2024?

Several compilers that are still in active use on embedded and cross-compilation targets do not yet ship a complete C23 implementation, and some may never receive one for cost or lifecycle reasons:

Toolchain / target	C23 status (as of 2026)
GCC < 14	Partial; `_BitInt`, `typeof`, `nullptr` missing or gated behind `-std=c2x`
Clang < 17	Partial; several C23 headers and attributes absent
IAR Embedded Workbench	Trails the standard by several years; C11 support only on recent versions
Arm Compiler 5 (armcc)	Frozen at C99/C11 subset; no C23 roadmap
Green Hills MULTI	Embedded-safety toolchain; C23 not yet qualified
SDCC (small-device C compiler)	C11 subset; no C23
Keil MDK (ARMCC legacy)	C99 only

C99 has essentially universal compiler support across every architecture Bovnar is expected to run on — x86-64, Arm Cortex-M/R/A, RISC-V, MIPS, PowerPC, and various DSP families. C23 does not.

Which specific C23 features would have been attractive but were ruled out?

Feature	C23 addition	Why skipped
`bool`, `true`, `false` keywords (no `<stdbool.h>`)	C23	Requires C23-capable front-end; `<stdbool.h>` works everywhere
`nullptr`	C23	Cosmetic; `NULL` or `(void*)0` is unambiguous
`typeof` / `typeof_unqual`	C23	Useful for generic macros; GCC/Clang extensions exist but are non-standard pre-C23
`#embed`	C23	Attractive for embedding binary tables at compile time; workaround via `xxd -i` is portable

Feature	C23 addition	Why skipped
`[[attributes]]`	C23	`__attribute__((…))` is already used where needed via compiler-specific guards
`_BitInt(N)`	C23	Would help for arbitrary-width integer types; current code uses `uint64_t` + masking
Improved `constexpr`	C23	Not applicable to a runtime library

None of these are blockers. Every C23 feature Bovnar would have used has a correct, readable C99 equivalent.

What about C11? It has `_Atomic` and `_Static_assert`.

C11 adds atomics, `_Generic`, `_Static_assert`, and `<threads.h>`. `_Static_assert` is useful and is approximated in the codebase with a macro. The atomics and threading additions are irrelevant: the reader and writer objects are not thread-safe by design — the caller is expected to guard them with their own synchronisation primitive. Pulling in C11 just for `_Static_assert` is not worth fragmenting compiler compatibility.

Does C99 impose any meaningful limitations on the implementation?

In practice, no. The limitations that do exist are handled cleanly:

- **Variable-length arrays (VLAs):** Not used. All fixed-size buffers are declared with compile-time constants. VLAs are optional in C11 and later and are dangerous on stack-limited embedded targets regardless of standard version.
- **`<stdint.h>` and `<stdbool.h>`:** Both are C99 additions and are available everywhere the library targets.
- **`//` comments:** C99. No issue.
- **Designated initialisers:** C99. Used extensively for `bvnr_data_t` and flag struct setup.
- **Flexible array members:** C99. Not currently used.
- **`restrict`:** C99. Used on internal buffer pointers where beneficial.

The one area where C99 imposes a real discipline is the absence of `_Static_assert` — the implementation works around this with the classic `typedef char static_assert_failed[condition ? 1 : -1]` pattern.

Is this likely to change in the future?

The minimum standard may be raised to C11 if a compelling use of `_Generic` or `_Static_assert` arises and the embedded toolchain matrix shifts accordingly. C23 as a

baseline is not on the roadmap until the IAR, Keil, and Green Hills ecosystems ship production-qualified C23 front-ends, which historically lags the standard publication by three to five years.

End of Bovnar FAQ — Specification (v1.1)