

Bovnar (BVNR) — A Practical Tutorial

Bovnar (BVNR) v1.0 documentation · Tutorial · 2026-06-01

Format version: 1.0 **Audience:** Developers already comfortable with JSON or similar text formats.

Why Bovnar Exists

Bovnar is **unit-safe serialization for scientific and industrial systems**. In those domains, the expensive failures are rarely bad syntax — they are unit confusion. A thrust value sent in pounds-force and read as newtons. An altitude in feet read as meters. The number parsed perfectly; the dimension was wrong, and nothing in the data said otherwise.

Most formats make a quiet trade-off: either they are easy for humans to read, or they carry precise machine-readable semantics — rarely both, and almost never including the *unit*. JSON tells you that `9.81` is a number; it does not tell you whether it is meters per second, volts, or a dimensionless ratio. Protobuf tells you the field type, but you need a `.proto` file and a compiler to interpret it, and units are outside its scope entirely. YAML is expressive but has notoriously sharp edges.

Bovnar makes a different choice: every value in a Bovnar document is **self-describing and unit-safe**. The type family (signed integer, float, string...), the bit-width, the numeric base, and the physical unit all travel *with* the value, in the same byte stream, without any external schema — and the unit is validated by the parser rather than assumed. Annotate a value as `m/s` and write a mismatched inline unit, and parsing fails. Hand a `.bvnr` file to anyone, and they have everything needed to interpret *and dimensionally trust* it: a 64-bit signed integer measured in kilopascals looks different from a 32-bit unsigned integer measured in mebibytes, and the format makes that difference explicit.

The format also supports raw binary embedding, multi-dimensional arrays with a clean row-separator syntax, and incremental streaming parsing — all in a text layer that is pleasant to read and write by hand.

The Absolute Minimum: A Valid Bovnar File

A Bovnar file is a sequence of **assignments**. Every assignment looks like this:

```
.key = value;
```

Three things to notice immediately:

1. The key always starts with a dot.
2. The value comes after `=`, optionally preceded by a type annotation.
3. Every assignment ends with a semicolon.

The simplest possible complete file:

```
.greeting = "hello";
.count    = 42;
.ratio     = 3.14;
```

Comments begin with `#` and run to end-of-line:

```
# This is a config file
.host      = "localhost";    # the server address
.port      = 5432;           # PostgreSQL default
```

The format is UTF-8 throughout. A UTF-8 BOM is accepted only at the very first byte of the stream.

Keys: Identifiers

A key is the bare word after the leading dot. Identifiers follow these rules:

- Must begin with a letter (`A–Z`, `a–z`), an underscore, or a UTF-8 leader byte in the range `0xC3–0xF4` (i.e., most non-ASCII letters).
- The body may also contain `+`, `-`, and digits.
- Byte `0xC2` (the UTF-8 leader for `U+0080–U+00BF`) is rejected at both the start and body of an identifier.
- Most ASCII punctuation is forbidden — specifically `! " # $ % & ' () * , . / : ; < = > ? @ [\] ^ { | } ~ ``. If you hit a parse error on what looks like a valid identifier, one of these characters is likely hiding inside it.
- An empty key (`.=`) is a hard error.

```
.simple      = 1;
.with_under  = 2;
.with-hyphen = 3;
.with+plus   = 4;
.camelCase   = 5;
.ALL_CAPS    = 6;
.v2          = 7;    # digit in body is fine; not at start
```

This would fail:

```
.123invalid = 1;    # error: starts with a digit
.bad,key    = 1;    # error: comma is forbidden in identifiers
```

Value Tokens: What Can Appear After `=`

There are ten categories of raw values:

Category	Example
Integer literal	<code>42</code> , <code>-7</code> , <code>"FF"</code> (base-16, quoted)
Float literal	<code>3.14</code> , <code>1e6</code> , <code>-.5</code>
Special number	<code>nan</code> , <code>inf</code> , <code>ninf</code>
Boolean	<code>true</code> , <code>false</code> , <code>on</code> , <code>off</code>
Quoted string	<code>"hello world"</code>
Symbol (bare word)	<code>ok</code> , <code>Monday</code> , <code>read_only</code>
Reference	<code>&.other.key</code>
Array	<code>[1, 2, 3]</code>
Struct	<code>{.x = 1; .y = 2;}</code>
Null	<code>;</code> (empty) or the <code>null</code> keyword

Each is covered in depth below, starting with numbers because that is where Bovnar's type system is most visible.

Numbers and the Type Annotation

Bare Literals

Numbers are written as plain text. Positive integers default to `uint:64`, negative integers to `sint:64`, and anything with a `.` or `e` to `float:64`:

```
.a = 42;           # → uint:64
.b = -7;          # → sint:64
.c = 3.14;        # → float:64
.d = 1e6;         # → float:64 (1 000 000)
.e = -.5;         # → float:64 (leading-dot form, -0.5)
.f = 123.;        # → float:64 (trailing dot, valid)
.g = 007;         # → uint:64 (leading zeros are legal; value is 7)
```

This automatic type assignment is called **default type synthesis**. It is convenient for quick files and config work. For anything where precision matters, you add an explicit type annotation.

Type Annotation Syntax

The annotation is enclosed in angle brackets and placed **between `=` and the value** — not on the key, not after the value:

```
.port    = <uint:16> 443;
.temp    = <float:32> 23.5;
.offset  = <sint:64> -2147483648;
```

Placing the annotation anywhere else is a hard error:

```
.port<uint:16> = 443;      # WRONG: annotation must come after '='
```

Type Families

There are seven type families:

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

Parameters

After the family keyword you may add parameters separated by commas after a `:`. The three parameter classes are **width**, **base**, and **unit**. They are identified by their syntactic form and may appear in any order:

```
.a = <uint:32>          1000;    # width only
.b = <uint:32,_16>      "FF";    # width + base (hex)
.c = <uint:32,m>        500;     # width + unit (meters)
.d = <uint:32,_16,m>    "1F4";   # all three – 500 meters in hex
.e = <uint:_16,m,32>    "1F4";   # same thing; order doesn't matter
```

Width

Width is a plain decimal number of bits. `0` means "use the default," which is always 64:

```
.a8  = <uint:8>    255;
.a16 = <uint:16>   65535;
.a32 = <uint:32>   4294967295;
.a64 = <uint:64>   18446744073709551615;
.def = <uint:0>    99;           # same as <uint:64>
```

For `float`, valid widths are `0`, `16`, and any multiple of `32` up to `32768`. Width `8` or a non-multiple-of-32 (other than 16) is a hard error. For `float_fix` and `float_dec`, valid widths are `0`, `16`, `32`, `64`, `128`, and `256`.

```
.half = <float:16> 3.14; # half-precision
.single = <float:32> 3.14; # single-precision
.double = <float:64> 3.14; # double-precision
.quad = <float:128> 3.14; # quad-precision
```

For `sint` and `uint`, any positive integer is accepted as a width — `<uint:12>` is perfectly valid for a 12-bit ADC sample.

Base

The base parameter starts with `_` followed by a decimal number. For `uint` and `sint`, every base from `_2` through `_62` (consecutive) is allowed, plus `_64` and `_85`. It selects the numeral system used to interpret the value.

This is the biggest surprise for newcomers: **non-decimal values must be provided as quoted strings**. A bare `ff` in value position is lexed as a *symbol*, not a number, and will produce a type mismatch error. Always quote non-decimal digits:

```
.hex = <uint:8,_16> "FF"; # correct – 255
.hex = <uint:8,_16> FF; # WRONG – FF is a symbol
.bin = <uint:8,_2> "11010110"; # correct – 214
.oct = <uint:16,_8> "755"; # correct – Unix permission rwxr-xr-x
```

Signed values use a minus prefix inside the string:

```
.neg_hex = <sint:32,_16> "-7FFFFFFF";
.neg_bin = <sint:8,_2> "-10000000"; # -128 in binary
```

`float` accepts only `_10` (the default) and `_16`. `float_fix` and `float_dec` **do not accept a base parameter at all** — specifying one is a hard error.

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:

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

Only these exact spellings are reserved: any other bare word (e.g. `infinity`, `nans`) is an ordinary symbol. They may be combined with any float type annotation and are valid in untyped context too. A special-number keyword takes no inline unit suffix — give its unit via the annotation (`<float:64,m/s> inf`).

Units

This is the feature most unique to Bovnar. Physical units are a first-class part of the type annotation, not a comment or a string field you hope someone reads.

Simple Units

The unit goes inside the angle brackets, after the other parameters:

```
.timeout = <float:64,s> 2.5; # seconds
.distance = <uint:64,m> 384400000; # meters (Earth-Moon distance)
.voltage = <float:32,V> 3.3; # volts
.temp = <float:32,°C> 23.5; # degrees Celsius (UTF-8 is fine)
```

SI Prefixes

SI prefixes are written before the base unit symbol with a **mandatory** `~` separator:

```
.k_ohm = <float:32,k~Ω> 4.7; # kilo-ohm
.milli = <float:32,m~V> 330.0; # millivolt
.micro = <float:32,μ~s> 50.0; # microsecond (μ = U+00B5)
.giga = <float:64,G~Hz> 2.4; # gigahertz
.kibi = <uint:64,Ki~B> 1024; # kibibytes (IEC binary prefix)
.mebi = <uint:64,Mi~B> 4096; # mebibytes
```

The `~` between prefix and unit is non-optional. `<float:32,kΩ>` (no `~`) would either fail to parse or be interpreted differently. Every prefix listed in the SI table uses the same pattern: `prefix~unit`.

IEC binary prefixes (`Ki`, `Mi`, `Gi`, `Ti`, `Pi`, `Ei`, `Zi`, `Yi`) follow the same rule:

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

Compound Units

Compound units are built from multiple components joined by `*` (or the middle dot `·`, U+00B7) for multiplication, and `/` for division:

```
.velocity = <float:64,m/s> 9.81; # meters per second
.acceleration = <float:64,m/s²> 9.81; # m·s⁻²
.force = <float:64,k~g·m/s²> 9.81; # Newton (kilogram × meter / second²)
.energy = <float:64,k~g·m²/s²> 1000; # Joule
.pressure = <float:64,k~g/(m·s²)> 101325; # Pascal
.momentum = <float:64,k~g·m/s> 0.5;
.field = <float:64,V/m> 150.0;
.moment = <float:64,m*s> 1.0; # asterisk = middle dot
```

The `/` separator is a **one-way switch**: once you write `/`, every subsequent component is in the denominator. Writing another `/` does not switch back to the numerator. If you need a component back in the numerator after a divisor, use a negative exponent:

```
.force_alt = <float:64,k~g~m~s^-2> 9.81;    # same as k~g~m/s^2
```

Exponents can be written as Unicode superscripts (`²`, `³`, `⁻¹`) or with an ASCII caret (`^2`, `^-2`). Both are valid:

```
.area1 = <float:64,m²>    100.0;
.area2 = <float:64,m^2>   100.0;    # same thing
.inv1  = <float:64,s⁻¹>   50.0;
.inv2  = <float:64,s^-1>  50.0;    # same thing
```

The maximum number of unit components in a compound unit is 8. More than that causes a parse error.

Explicitly Dimensionless: `no_unit`

When you want to be explicit that a value carries no physical dimension, use the keyword `no_unit`:

```
.ratio = <float:64,no_unit> 1.4142;
.count = <uint:32,no_unit> 42;
```

Explicitly writing `no_unit` yields `BVN_UNIT_NONE` (`num_components == 0`). Omitting the unit parameter entirely yields `BVN_UNIT_NO_PREFIX(bu_none)` (`num_components == 1`, `base == bu_none`). Both are semantically dimensionless and compare as compatible via `bvn_units_compatible`, and both serialise to `"no_unit"` via `bvn_unit_to_string`, but they are structurally distinct internal states. The explicit `no_unit` is useful for documentation: it tells a reader that the author actively chose dimensionless, not that they forgot.

Inline Unit Suffix

For untyped or partially-typed values, you can append a unit suffix directly after the value literal, separated by at least one whitespace character:

```
.speed = 9.81 m/s;          # valid
.mass  = 70.0 k~g;          # result: float:64, unit = k~g
.heap  = 65536 B;           # result: uint:64, unit = B (bytes)
```

This is forbidden:

```
.bad = 9.81m;               # ERROR: no separator at all
```

If a type annotation also specifies a unit and you additionally write an inline suffix, they must match exactly. A mismatch is a hard error:

```
.ok = <float:64,m/s> 9.81 m/s;    # redundant but valid: both say m/s
.bad = <float:64,m> 9.81 s;      # ERROR: annotation says m, inline says s
```

The inline unit suffix is completely forbidden inside array elements. This is a lexer-level restriction, not a semantic one — any letter or underscore following a value token inside `[...]` is an immediate hard error.

Fixed-Point Numbers: `float_fix`

Fixed-point (Q-format) numbers store a value as a signed integer, with the binary point shifted by a declared number of fractional bits. The wire representation is an integer; the mathematical value is `raw_integer × 2-Q`.

The annotation requires a `qN` parameter specifying fractional bits, in addition to the width:

```
.adc = <float_fix:16,q8> 3.14;    # 16-bit, 8 fractional bits
                                   # range: [-128, 127.996], resolution ≈ 0.0039
.pid = <float_fix:32,q16> -1.5;   # 32-bit Q16: 15 integer bits + sign + 16 fractional
.vel = <float_fix:32,q8,m/s> 9.81; # with a unit
.cnt = <float_fix:64,q0> 4096;    # Q0 = no fractional bits (pure integer shell)
```

Constraints:

- `Q` must be in the range `0 ≤ Q < effective_width`. `<float_fix:16,q16>` is illegal because `Q` must be strictly less than the width.
- The base parameter (`_N`) is forbidden. `<float_fix:32,q8,_10>` is an error.
- Valid widths: `0` (→64), `16`, `32`, `64`, `128`, `256`. Width `8` is not valid.

Decimal Floating-Point: `float_dec`

`float_dec` is IEEE 754-2008 decimal floating-point — useful in financial and metrological contexts where exact decimal representation matters (binary floats cannot represent `0.1` exactly; decimal floats can):

```
.price  = <float_dec:32> 12.99;    # 7 significant decimal digits
.voltage = <float_dec:64,Pa> 101325.0; # 16 significant decimal digits
.pi_big = <float_dec:128> 3.14159265358979323846264338327950288; # 34 digits
```

Rules are similar to `float_fix`: no base parameter, same set of valid widths.

Strings

Strings are enclosed in double quotes. The `<utf8>` annotation is optional — a bare quoted literal is automatically synthesised as `utf8`:

```
.greeting = "hello";           # → <utf8>
.typed    = <utf8> "world";    # explicit, same result
.empty    = "";
```

Escape Sequences

Only seven escape sequences are defined:

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

Any other character after a backslash is immediately a hard error — there is no `\uXXXX`, no `\xNN`, no `\0`. If you need to embed arbitrary bytes in a Bovnar document, use an octet stream (see below).

Raw whitespace bytes (HT, LF, VT, FF, CR) are accepted unescaped inside string literals, so a multi-line string is valid:

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

String Concatenation

Adjacent string literals separated only by whitespace or comments are concatenated at lex time:

```
.url  = "https://" "api.example.com" "/v1";
.long = "first part "
       "second part "  # a comment is fine here
       "third part";
```

This is the idiomatic way to write a long string across multiple lines or to construct a non-decimal value that happens to be split for readability:

```
.guid = <uint:128,_16> "deadbeef"
                        "cafebabe"
                        "01234567"
                        "89abcdef";
```

Symbols

A symbol is a bare word in value position — no quotes. It is its own token type, distinct from strings:

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

Symbol bodies follow the same character rules as identifier bodies: letters, digits, `+`, `-`, `_`, and valid UTF-8 high bytes. The key distinction from an identifier is context: identifiers appear as keys (after the opening dot), symbols appear as values.

Inside an array, `[]` and `,` terminate a symbol. At assignment level, `;` terminates it.

Eight bare words are **reserved keywords** and are *not* symbols: `null`, `true`, `false`, `on`, `off`, and the special floats `nan`, `inf`, `ninf` (see the special-number section). A longer word that merely starts with one — `ontology`, `nullable`, `truthy`, `infinity` — is still an ordinary symbol.

Booleans

`true`, `false`, `on`, and `off` are reserved keywords carrying the `bool` type family. `on` is an alias for `true` and `off` for `false`; all four collapse to a boolean value and serialize canonically as `true` / `false`:

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

A bare boolean synthesises a `<bool>` annotation automatically. The `<bool>` family takes no parameters (`<bool:8>` is a hard error), and only the four keywords are valid `<bool>` values.

Null Values

A null is the explicit absence of a value. It appears as an empty slot — nothing between `=` and `;` — or, equivalently, the reserved keyword `null`; extra commas inside an array also produce nulls:

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

Inside arrays, null elements arise from leading, trailing, or consecutive commas:

```
.sparse = [, 1, , 2, ];      # null, 1, null, 2, null – five elements
.holes  = [<sint:16> , <sint:16> 0, <sint:16> ]; # typed nulls
```

References

A reference is 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 entirely up to the application:

```
.host      = "db.internal";
.port      = 5432;

.conn_host = &.host;      # stored as ".host"
.conn_port = &.port;      # stored as ".port"
```

Multi-segment paths:

```
.server = {
  .address = "10.0.0.1";
  .tls     = { .cert = "/etc/ssl/server.pem"; };
};

.proxy_addr = &.server.address; # ".server.address"
.cert_path  = &.server.tls.cert; # ".server.tls.cert"
```

References are valid in arrays and structs:

```
.nodes = [&.master, &.replica-a, &.replica-b];

.worker = {
  .retries = &.defaults.retries;
  .delay   = &.defaults.delay_s;
};
```

A reference is not a null. If a field is optional and currently has no referent, represent the absence with an explicit null:

```
.override = ;           # null – no reference, use internal default
# .override = &.config.x; # or, non-null – follow this reference
```

Arrays

An array is enclosed in square brackets. A single row looks like JSON:

```
.primes = [2, 3, 5, 7, 11, 13];
.names  = ["Alice", "Bob", "Carol"];
```

Multi-Dimensional Rows

Multiple rows are separated by `/`. Each bracketed group is one row:

```
.matrix = [1, 2, 3]/[4, 5, 6];    # 2 rows × 3 columns
```

The event stream reflects this: `ev_array_row_start`, elements, `ev_array_row_end`, then `ev_array_dim_start` before the second `ev_array_row_start`. Row-separator `/` and array nesting are independent concepts — do not confuse them. The `/` between `]` and `[` creates a new dimension in the *same* array, not a nested one.

The `/`-separated dimension rows of a single array must have a **consistent element count**, and (since spec 1.0) array elements are **homogeneous**: sibling sub-arrays must also match in length, so bare arrays and matrices are rectangular:

```
.ok1 = [1, 2, 3]/[4, 5, 6];    # valid: both /-rows of one array have 3 elements
.ok2 = [[1, 2], [3, 4]];      # valid: rectangular sub-arrays
# .ok3 = [[1, 2], [3, 4, 5]];  # error_array_row_size_mismatch: sibling sub-arrays differ
                                (2 vs 3)

.bad1 = [1, 2, 3]/[4, 5];      # error_array_row_size_mismatch: 3 vs 2 (/ -rows of one
                                array)
.bad2 = [[1, 2]/[3, 4, 5]];    # error_array_row_size_mismatch: 2 vs 3 (one /-array's own
                                rows)
```

Per-Element Type Annotations

Each element may carry its own annotation, placed immediately after the comma (or the opening bracket for the first element):

```
.mixed = [<uint:8> 255, <sint:8> -128, <float:64> 3.14];
```

A single annotation before `[` applies to all elements — it is a whole-array annotation:

```
.ports = <uint:16> [80, 443, 8080, 8443];
.temps = <float:32, °C> [23.5, 24.1, 22.8];
```

Per-element annotations can still appear inside a whole-array-annotated array.

Structs as Array Elements

Arrays of structs are the standard idiom for a list of records:

```
.users = [
  { .id = <uint:32> 1; .name = "Alice"; .role = admin; },
  { .id = <uint:32> 2; .name = "Bob"; .role = user; }
];
```

Structs

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

```
.person = {
  .name = "Alice";
  .age  = 30;
  .active = true;
};
```

Structs nest deeply — the limit is `max_struct_nesting`, which defaults to 64 and has a hard cap of 255:

```
.config = {
  .database = {
    .primary = {
      .host = "db1.internal";
      .port = <uint:16> 5432;
      .tls = {
        .enabled = true;
        .cert    = "/etc/ssl/db.pem";
      };
    };
  };
};
```

An empty struct is valid:

```
.placeholder = {};
```

An unmatched closing brace — a `}` with no matching `{` — is `error_illegal_struct_close`.

Octet Streams: Inline Binary

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 wire protocol for an octet stream is:

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

In practice, you will not hand-author octet streams — the writer API does this for you. In a `.bvnr` file shown for documentation purposes it appears as an escaped binary sequence:

```
.binary = \x00\x01\x05\x00hello\x01\x03\x00bye\x00;
```

This produces two data events with payloads `"hello"` (5 bytes) and `"bye"` (3 bytes). The UTF-8 validator is suspended for the duration of a binary region.

Putting It Together: Realistic Examples

Hardware Configuration

```
# Sensor node configuration
.node_id      = <uint:8> 7;
.firmware_ver = "2.4.1";

.radio = {
  .frequency = <float:64,M~Hz> 868.1;
  .tx_power  = <sint:8,dB> 14;          # dB (power level); 'dBm' is not a bovnar
unit
  .bandwidth = <float:32,k~Hz> 125.0;
  .sf        = <uint:8> 7;            # spreading factor
};

.sampling = {
  .interval   = <uint:32,s> 60;
  .burst_size = <uint:16> 8;
  .adc_bits   = <uint:8> 12;
};

.calibration = {
  .offset     = <float_fix:32,q8,°C> -0.5;
  .gain       = <float_fix:32,q16> 1.002;
};
```

Scientific Measurement Batch

```
.experiment = "thermal_runaway_001";
.timestamp = <uint:64,s> 1715000000;

.measurements = [
    {.channel = <uint:8> 0;
     .value   = <float:32,°C> 27.3;
     .valid   = true;},
    {.channel = <uint:8> 1;
     .value   = <float:32,°C> 31.7;
     .valid   = true;},
    {.channel = <uint:8> 2;
     .value   = <float:32,°C> 112.4;
     .valid   = true;}
];

.threshold = <float:32,°C> 80.0;
.alarm      = triggered;
```

Multi-Dimensional Matrix

```
# 3x3 rotation matrix in float64
.R = <float:64> [1.0, 0.0, 0.0]
               /[0.0, 0.866, -0.5]
               /[0.0, 0.5,  0.866];
```

Mixed Binary and Text

```
.packet = {
    .type      = data_frame;
    .sequence  = <uint:32> 42;
    .timestamp = <uint:64,μs> 1715000000123456;
    .payload   = \x00\x01\x08\x00\xDE\xAD\xBE\xEF\xCA\xFE\xBA\xBE\x00;
    .crc32     = <uint:32,_16> "A1B2C3D4";
};
```

Common Mistakes and the Errors They Produce

Understanding the error codes is essential for debugging. The parser reports line, column, the offending byte, and a byte-offset into the stream.

Annotation on the wrong side of `=`:

```
.x<uint:32> = 42;    # error_unexpected_input_byte (< after identifier)
```

Non-decimal value not quoted:

```
.x = <uint:_16> FF;  # FF is parsed as a symbol → error_type_value_mismatch
```

Signed value in unsigned type:

```
.x = <uint:8> -1;    # error_value_out_of_range
```

Overflow:

```
.x = <uint:8> 256;    # error_value_out_of_range (max is 255)
```

Float with an unsupported width:

```
.x = <float:8> 1.0; # error_illegal_value_type
.x = <float:12> 1.0; # error_illegal_value_type (not 16 or multiple of 32)
```

Base parameter on `float_fix` or `float_dec`:

```
.x = <float_fix:32,q8,_10> 1.0; # error_illegal_value_type
.x = <float_dec:64,_10> 1.0; # error_illegal_value_type
```

Q value too large for `float_fix`:

```
.x = <float_fix:16,q16> 1.0; # error_illegal_value_type (Q must be < width)
```

Empty unit component:

```
.x = <float:64,m//s> 1.0;    # error_unit_illegal (empty component between //)
```

Too many unit components:

```
.x = <float:64,m*s*k~g*A*K*mol*cd*b*V> 1.0; # error_unit_illegal (9 > 8 max)
```

Inline unit suffix inside array:

```
.x = [9.81 m/s, 3.14 m]; # error_unexpected_input_byte (units forbidden in arrays)
```

Comma outside array:

```
.x = 42,; # error_unexpected_input_byte
```

Unknown escape sequence:

```
.x = "\x41"; # error_illegal_escape_sequence (\x is not defined)
```

Unmatched struct close:

```
.x = 1;} # error_illegal_struct_close (a stray '}' at the top level)
```

A `/`-array's dimension rows must match:


```
.ok = [[1, 2]/[3, 4]];      # valid – the /-array's two rows both have 2 elements
.bad = [[1, 2]/[3, 4, 5]];  # error_array_row_size_mismatch – 2 vs 3 (/rows)
# note: [[1,2],[3,4,5]] is also rejected – sibling sub-arrays must match in length (1.0
homogeneity)
```

Error Recovery Mode

The parser supports an optional `continue_on_error` mode. When enabled, a parse error invokes the `on_error` callback and then enters a resync state machine that skips bytes while tracking bracket and brace nesting. The `recovery_count` counter (accessible via `bvnr_reader_get_recovery_count`) is incremented immediately when an error triggers entry into resync mode — not when the resync eventually completes. When the resync state machine reaches a `;` at the original nesting depth, it resumes normal parsing.

This mode is intended for log streams and unreliable transports — situations where one malformed assignment should not discard the rest of the file. For configuration and data serialization, disable it: fail fast and reject corrupt input entirely.

Size Limits

All limits are configurable via `bvnr_read_flags_t`. The defaults are intentionally permissive — 2 147 483 647 for array items, text bytes, and file size. Production deployments should set explicit caps:

Field	Default	Suggested cap
<code>max_identifier_length</code>	255	255
<code>max_string_length</code>	65535	application-defined
<code>max_number_length</code>	65535	65535
<code>max_symbol_length</code>	255	255
<code>max_reference_length</code>	65535	65535
<code>max_array_items</code>	2 147 483 647	application-defined
<code>max_text_bytes</code>	2 147 483 647	application-defined
<code>max_file_size</code>	2 147 483 647	<code>16777216</code> (16 MiB)
<code>max_array_nesting</code>	0 (→64 internal)	32 or less for most applications
<code>max_struct_nesting</code>	0 (→64 internal)	32 or less for most applications

Setting any field to `0` in `bvnr_read_flags_t` causes the reader to substitute an internal default — **64** for both nesting depths, and **2 147 483 647** for the three byte/item counters. These are permissive but finite. Explicitly setting `max_file_size = 16777216` (16 MiB) is the recommended practice for production deployments.

Event Model (C API Overview)

The C reader is a two-phase, callback-driven SAX-style parser. You supply two optional callbacks: `on_unverified` receives events before semantic validation, `on_verified` receives them after. For normal consumption, `on_verified` is sufficient.

The event sequence for `.port = <uint:16> 443;` is:

```
ev_stream_start
ev_assignment_start      → key = "port"
ev_type_annotation_start  → raw = "uint:16"
ev_type_annotation_type_family → "uint"
ev_type_annotation_type_family_parameter → width: 16
ev_type_annotation_end
ev_data                  → value = "443", type = {uint, 16}
```

The `ev_type_annotation_type_family_parameter` event fires once per **present** parameter. Because the explicit annotation `<uint:16>` carries neither a base nor a unit parameter, only the width event is emitted. Contrast this with a bare `42;` (no annotation), where the validator **synthesises** `<uint:64, _10, no_unit>` and therefore emits three parameter events:

```
ev_stream_start
ev_assignment_start      → key = "count"
ev_type_annotation_start  → (synthesised)
ev_type_annotation_type_family → "uint"
ev_type_annotation_type_family_parameter → width: 64
ev_type_annotation_type_family_parameter → base: _10
ev_type_annotation_type_family_parameter → unit: no_unit
ev_type_annotation_end
ev_data                  → value = "42", type = {uint, 64}
```

The synthesised annotation always produces all three parameter events; explicit annotations only produce events for the parameters that are actually present.

ev_type_annotation_start asymmetry: This event is emitted to `on_unverified` first (with raw annotation bytes in `data` / `length` but no `value_type` or `value_unit` populated), and then separately to `on_verified` (with `value_type` and `value_unit` filled in). All other type-annotation events and `ev_data` are emitted to both callbacks simultaneously via the same call.

The `bvnr_data_t` structure passed with `ev_data` carries:

- `type` — the token type (`token_is_number`, `token_is_string`, `token_is_symbol`, `token_is_reference`, `token_is_array_number`, `token_is_array_string`, `token_is_null_value`, `token_is_octet_stream`)
- `value_type` — the `value_type_spec_t` (family, width, base/Q)
- `value_unit` — the `value_unit_t` (up to 8 components)
- `data` — pointer to the raw value bytes
- `length` — byte count

For an octet stream, `ev_octet_stream_start` and `ev_octet_stream_end` bracket one or more `ev_data` events with `type == token_is_octet_stream`.

For arrays, `ev_array_row_start` opens each row, `ev_array_row_end` closes it, and `ev_array_dim_start` separates dimensions (the `/` rows).

For structs, `ev_struct_start` and `ev_struct_end` bracket the nested assignments. No `ev_data` event is emitted for the struct value itself.

Quick-Reference: Syntax Cheat Sheet

```

# — Assignments —————
.key = value;
.key = <type-family:width,_base,unit> value;

# — Type families —————
<uint:32>          # unsigned 32-bit integer
<sint:64>          # signed 64-bit integer
<float:32>         # IEEE 754 single-precision
<float:128>        # IEEE 754 quad-precision (or any multiple of 32)
<float_fix:32,q16> # Q-format fixed-point, 32-bit, 16 fractional bits
<float_dec:64>     # IEEE 754-2008 decimal, 64-bit
<utf8>            # UTF-8 string
<bool>            # boolean (true/false); no parameters

# — Units —————
<float:64,s>       # seconds
<float:64,k~m>     # kilometers (SI prefix~unit, `~` mandatory)
<float:64,m/s>     # meters per second
<float:64,m/s^2>   # meters per second squared
<float:64,k~g~m/s^2> # kilogram-meters per second squared
<uint:64,Ki~B>     # kibibytes
<float:64,no_unit> # explicitly dimensionless

# — Special values —————
nan inf ninf

# — Booleans and null (reserved keywords) —————
.b = true;   .b = false;   .b = on;   .b = off;   # on==true, off==false
.n = null;   .n = ;        # null keyword == empty

# — Inline unit suffix (scalar context only) —————
.x = 9.81 m/s;

# — Non-decimal (must be a quoted string) —————
.x = <uint:_16> "DEADBEEF";
.x = <sint:_2>  "-100000000";

# — Arrays —————
.a = [1, 2, 3];
.a = [1, 2, 3]/[4, 5, 6];    # 2D: two rows
.a = <float:32> [1.0, 2.0];  # whole-array annotation
.a = [<uint:8> 1, <sint:8> -1]; # per-element annotations
.a = [, 1, , 2, ];          # null, 1, null, 2, null

# — Structs —————
.s = {.x = 1; .y = 2;};
.s = {};

# — Symbols —————
.state = running;
.ok    = true;

# — References —————
.ref  = &.other_key;
.deep = &.config.db.host;

# — Null —————
.x = ;
.x = <uint:32> ;

# — Strings —————
.s = "hello\nworld";
.s = "part one " "part two";  # concatenation

```

Bovnar Specification (v1.0) — format by the Bovnar project.