

# Bovnar Specification

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

---

**Version:** 1.1 **Status:** Released (1.x line; additive over the frozen 1.0 baseline) **Last updated:** 2026-06-21

---

## Table of Contents

---

1. Overview
  2. File Format at a Glance
  3. Character Encoding & BOM
  4. Lexical Structure
  5. Type Annotations
  6. Value Tokens
  7. Arrays
  8. Structs (Scopes)
  9. Octet Streams (Binary Mode)
  10. Default Type Synthesis
  11. Units System
  12. Validation & Constraints
  13. Error Handling & Recovery
  14. Formal EBNF
  15. Complete Examples
  16. Reference API
  17. Versioning & Stability - Appendix A: Event Sequence Reference - Appendix B: Implementation Notes - Appendix C: Limits Summary
- 

## 1. Overview

---

**Bovnar (BVNR)** is **unit-safe serialization for scientific and industrial systems**: a **typed, self-describing, text-binary hybrid** format that carries a validated physical unit with every value. It combines a human-readable text layer with an efficient binary octet-stream escape mechanism and a rich, unit-aware type annotation system.

## Design Goals

- **Unit-safe** – every value may carry a physical unit that is validated against a built-in unit table; a unit that conflicts with the type annotation is a parse error (`error_unit_mismatch`)
- **Self-describing** – values carry explicit type metadata (family, bit-width, base, measurement unit)
- **Human-readable** for common cases (numbers, strings, symbols, nested structures)
- **Binary-friendly** via octet stream escape for opaque payloads
- **Schema-free** but type-safe – the type annotation is per-value, optional, and validated
- **Streamable** – parsed incrementally via a pull-based reader; events emitted to callbacks
- **Error-recoverable** – optional resync mode for parsing through minor corruption

## Architecture

The reference implementation is a **two-phase** parser:

Phase	Callback	Purpose
<b>Unverified</b> ( <code>on_unverified</code> )	Raw token events before semantic validation	Inspection, logging, partial consumption
<b>Verified</b> ( <code>on_verified</code> )	Fully validated events	Actual data consumption

Both phases receive the same stream of events (`bvnr_event_t`). The validator sits between them.

## Stream Model

```
stream_begin →
  (assignment)* →
  EOF
```

Each assignment is:

```
.identifier = value;
```

## 2. File Format at a Glance

```
# This is a comment

.my_first_value = 42;
.my_next_value = 3.14159;
.a_string = "hello world";
.a_typed_value = <uint:32> 1000;
.a_utf8_value = <utf8> "text";
.an_array = [1, 2, 3]/[4, 5, 6];
.a_struct = {
    .nested = true;
};
```

### Key Syntax Rules

Construct	Syntax	Example
Comment	# ... newline	# this is a comment
Assignment	.key = value ;	.foo = 42;
Type annotation	<type-spec> placed after =, before the value	.foo = <uint:32> 42;
Number	[ - ]digits[ .digits ][ e[+/-]digits ]	42, -3.14, 1e6
Special number	nan, inf, ninf	.x = inf;
Boolean	true / false / on / off	.b = true; , .b = off;
String	"..." with escapes ( \u{...} / \xHH in spec 1.1)	.s = "hello\nworld"; , .s = "caf\u{e9}";
Symbol	bare identifier (no quotes)	.s = ok; , .day = Monday;
Reference	&.path.to.key ; array indexing &.path[i][j] (spec 1.1)	.ref = &.config.host; , .c = &.m[0][1];
Array	[ ... ] rows separated by /	.a = [1,2,3]/[4,5,6];
Struct	{ ... }	.s = { .x = 1; .y = 2; };
Octet stream	\x00 ... binary ... \x00	binary escape
Null value	absent token or null keyword	.x = ; , .x = null; , [,1,]
Version directive (spec 1.1)	#!bovnar <major>.<minor> on the first line	#!bovnar 1.1
Datetime (spec 1.1)	<datetime:width,epoch> — signed epoch seconds	.t = <datetime:64,unix> 1750000000;
Unit (type annotation)	inside <...> as a type param	<float:64,m/s>
Unit (inline suffix)	after scalar value, before ;	.speed = 9.81 m/s;
Fixed-point type	<float_fix:[width],qN[,unit]>	<float_fix:32,q16> , <float_fix:q8>

Construct	Syntax	Example
Decimal float type	<code>&lt;float_dec:width[,unit]&gt;</code>	<code>&lt;float_dec:64,Pa&gt;</code>
Unit (compound)	<code>unit[*./unit...]</code>	<code>m/s<sup>2</sup></code> , <code>k~g·m/s<sup>2</sup></code> , <code>m*s</code>

## 3. Character Encoding & BOM

### 3.1 UTF-8

All text-layer bytes (outside octet-stream regions) must form **valid UTF-8**. The parser runs a parallel UTF-8 validator (`bvn_utf8_feed`) alongside the state machine. Violations produce `error_invalid_utf8_byte`.

Overlong sequences and surrogate halves (U+D800–U+DFFF) are rejected.

### 3.2 Byte Order Mark (BOM)

The UTF-8 BOM (`EF BB BF`) is only legal **at byte offset 0** of a stream.

The lexer runs a dedicated `first_comment_*` state machine over the first comment line to detect misplaced BOMs:

- A BOM (`EF BB BF`) detected **inside the first-line comment** produces `error_invalid_byte_order_mark`.
- A BOM byte (`EF`) seen **after** the first comment line (in the `first_bom` state, before the first `.` identifier) is unhandled and produces `error_unexpected_input_byte`.

```
# BOM at byte offset 0: valid
EF BB BF.foo = 1;

# BOM bytes inside the first comment → error_invalid_byte_order_mark
# comment text: # EF BB BF text

# BOM bytes after the first comment line → error_unexpected_input_byte
# comment line
EF BB BF.foo = 1;
```

A BOM appearing **inside** any later comment or string is **valid UTF-8** and accepted (only the `first_comment_*` states guard against BOM).

### 3.3 Byte Classes

Class	Bytes	Usage
Whitespace	<code>0x09</code> (HT), <code>0x0A</code> (LF), <code>0x0B</code> (VT), <code>0x0C</code> (FF), <code>0x0D</code> (CR), <code>0x20</code> (SP)	Token separators

Class	Bytes	Usage
Control (rejected)	0x00–0x08, 0x0E–0x1F, 0x7F (DEL)	Hard errors outside strings
Safe ASCII	0x20–0x7E except reserved punctuation	Identifier/symbol/reference bodies
High bytes	0x80–0xFF	UTF-8 multi-byte sequences

### 3.4 Version Directive (spec 1.1)

A document **may** declare the spec version it targets with a directive on its very first line:

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

**An unversioned document is spec 1.0.** A document with no directive is treated as targeting spec **1.0** — the frozen baseline grammar. The directive only ever opts in to a newer version; its absence is never ambiguous. Consequently, any syntax introduced after 1.0 (a future minor revision) is available only to a document that declares the matching version: a 1.1-only construct in an unversioned (i.e. 1.0) document is invalid, exactly as it would be to a 1.0 reader. `bvnr_reader_get_declared_version` returns `false` for such a document (there is nothing declared), and a consumer that wants a concrete number should default to 1.0.

**Form.** `# !bovnar` followed by one or more spaces/tabs, a `<major>.<minor>` decimal version, optional trailing whitespace, then end-of-line (LF/CR) or EOF. Each component is a decimal integer with no leading zero (a bare `0` excepted) and must fit in 16 bits.

**Backward compatible by construction.** Lexically the directive is a comment (`#...`), so a spec-1.0 reader skips it transparently — declaring a version never breaks an older parser. This is the one place where a comment carries meaning; everywhere else comments remain semantically inert.

#### Recognition rules.

- It is recognised only as the **very first comment**, after an optional BOM and leading whitespace. A `#!bovnar ...` appearing on any later line is an ordinary comment and is ignored.
- `#!bovnar` not followed by whitespace (e.g. `#!bovnarish`) is an ordinary comment, not a directive.
- A directive prefix followed by a malformed version (missing component, non-numeric, leading zero, or trailing junk) is `error_invalid_spec_version`.

**Enforcement.** A 1.1+ reader records the declared version and exposes it (`bvnr_reader_get_declared_version`). By default a version newer than the reader supports is

**accepted leniently** — recorded, but not rejected up front; the parse only fails if it later meets syntax the reader does not implement. A reader opened with `strict_version` instead rejects any unsupported version ( `error_unsupported_spec_version` ) at the directive. A reader supports spec `<major>.<minor>` when `major` equals its own and `minor` is  $\leq$  its own ( `BVNR_SPEC_VERSION_*` ).

```
#!bovnr 1.1      # current – accepted
#!bovnr 1.0      # older minor – accepted
#!bovnr 1.9      # newer minor – lenient: accepted; strict: error
#!bovnr 2.0      # newer major – lenient: accepted; strict: error
#!bovnr 1        # malformed (no minor) – error_invalid_spec_version
```

## 4. Lexical Structure

### 4.1 Whitespace & Comments

Whitespace and comments are freely interleaved between all tokens.

#### Whitespace characters:

```
HT (0x09), LF (0x0A), VT (0x0B), FF (0x0C), CR (0x0D), SP (0x20)
```

#### Comments:

```
# any bytes except 0x00–0x08, 0x0E–0x1F, 0x7F
# terminated by LF or CR (or EOF)
```

```
# A full-line comment
.foo = 42; # an inline comment
```

### 4.2 Identifiers (Keys)

Every assignment begins with `.` followed by an identifier (the key).

#### Syntax:

```
id-start = A–Z | a–z | "_" | UTF-8 leader bytes 0xC3–0xF4
id-body  = id-start | "+" | "-" | DIGIT | UTF-8 continuation bytes
```

#### Constraints:

- At least one `id-start` character is required after `.` (`.=` is `error_empty_identifier`)
- Byte `0xC2` is rejected at identifier start **and body** positions (U+0080–U+00BF are forbidden in identifiers everywhere)

- The following ASCII punctuation characters are **hard errors** inside identifier bodies:

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

### Valid identifiers:

```
.foo = 1;
.FooBar = 2;
._private = 3;
.my-key = 4;      # hyphen allowed
.my+key = 5;      # plus allowed
.user_defined = 6;
```

### Invalid identifiers:

```
. = 1;           # error_empty_identifier
.123 = 2;        # starts with digit
.foo,bar = 3;    # comma inside identifier – error
```

## 4.3 String Literals

### Syntax:

```
string-literal = ''' { safe-byte | escape-seq } '''
string         = string-literal { ws string-literal } # concatenation
```

**Safe bytes:** `0x09–0x0D` (HT, LF, VT, FF, CR), `0x20–0x7E` except `"` (0x22) and `\` (0x5C), and `0x80–0xFF`. `0x7F` (DEL) is rejected even inside strings.

**Control bytes** `0x00–0x08`, `0x0E–0x1F`, and `0x7F` (DEL) are hard errors inside strings. The whitespace bytes HT (0x09), LF (0x0A), VT (0x0B), FF (0x0C), and CR (0x0D) are accepted as raw string content.

### Escape Sequences

Escape	Meaning	Byte
<code>\t</code>	Horizontal Tab	<code>0x09</code>
<code>\n</code>	Line Feed	<code>0x0A</code>
<code>\v</code>	Vertical Tab	<code>0x0B</code>
<code>\f</code>	Form Feed	<code>0x0C</code>
<code>\r</code>	Carriage Return	<code>0x0D</code>
<code>\"</code>	Double Quote	<code>0x22</code>
<code>\\</code>	Backslash	<code>0x5C</code>

In a **spec 1.1** document (one declaring `#!bovnar 1.1` or newer — see §3.4) two further escapes are available:

Escape	Meaning
<code>\xHH</code>	the single byte <code>HH</code> (exactly two hex digits)
<code>\u{H...}</code>	the Unicode scalar value <code>U+H...</code> (1–6 hex digits), UTF-8 encoded

- `\u{...}` rejects surrogates ( `U+D800` – `U+DFFF` ) and values above `U+10FFFF` with `error_invalid_codepoint`; a missing/empty/over-long brace group or a non-hex digit is `error_illegal_escape_sequence`.
- `\x` writes a raw byte, but a string's contents must still be valid UTF-8 (the `utf8` family guarantee). So `"\xC3\xA9"` is `"é"`, whereas a lone `"\xFF"` is `error_invalid_utf8_byte`. Use an octet stream for arbitrary, non-textual bytes.
- **Gating.** `\x` and `\u` are 1.1-only. In a 1.0 or unversioned document (§3.4) the `x / u` after a backslash is not a recognised escape and yields `error_illegal_escape_sequence`, exactly as a 1.0 reader reports.

Any byte other than `t`, `n`, `v`, `f`, `r`, `"`, `\` (plus `x`, `u` in 1.1) after `\` causes `error_illegal_escape_sequence`.

**String concatenation:** Two or more adjacent string literals (separated only by whitespace/comments) are concatenated into a single token:

```
.long = "hello " "world";    # → "hello world"
```

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

## Examples

```
.simple = "hello";
.with_escapes = "tab:\there\nnewline";
.unicode = "café";           # UTF-8 bytes for é = 0xC3 0xA9
.empty = "";
```

## 4.4 Symbols

A **symbol** is an unquoted bare-word token that appears in value position. It starts with an `id-start` character and continues with `id-body` characters.

### Differences from identifier keys:

Context	<code>,</code>	<code>]</code>	<code>;</code>	<code>=</code>
Identifier body	hard error	hard error	hard error	transitions to value
Symbol body	terminates → new array element	terminates → close array	terminates → end value	hard error



```
.status = ok;           # symbol "ok"
.day = Monday;         # symbol "Monday"
.flags = [red, green, blue]; # symbols as array elements
```

**Reserved keywords.** Eight exact bare words are not symbols: `null`, `true`, `false`, `on`, `off`, `nan`, `inf`, and `ninf`. Lexically they are still symbol tokens, but the validator reserves the exact spellings and reclassifies them — `null` to a null value, `true` / `false` / `on` / `off` to a `bool` value (`on`  $\equiv$  `true`, `off`  $\equiv$  `false`), and the special floats `nan` / `inf` / `ninf` to numeric special values (§6.4). A bare boolean with no annotation synthesises a `<bool>` type (§10); an explicit `<bool>` annotation accepts only the four bool keywords. A longer word that merely begins with a keyword (`ontology`, `nullable`, `truthy`, `infinity`) remains an ordinary symbol.

```
.enabled = true;           # bool value (vt_bool), not a symbol
.debug   = off;           # bool false
.choice  = <bool> on;      # explicit; on == true
.label   = truthy;        # symbol "truthy" – not a keyword
```

## 4.5 References

A **reference** is a dotted path to another key, introduced by `&`.

### Syntax:

```
reference      = "&" ref-segment { ref-segment | index }
ref-segment    = "." id-start { ref-body-char }
ref-body-char  = same as symbol-body-char | "."
index          = "[" digit { digit } "]"          (* spec 1.1 *)
```

The stored text includes the leading dot and all intermediate dots:

```
.host = "localhost";
.port = 8080;

.connection = &.host;           # stored as ".host"
.full       = &.config.host;    # stored as ".config.host"
```

### Examples:

```
.config = { .host = "example.com"; }; # nested struct
.ref     = &.config.host;             # reference → ".config.host"
```

**Resolution.** A reference is stored **unresolved** — as the path string only. The parser and library never dereference it, so:

- the target **need not exist** at parse time; a reference to a missing key (`&.nope`), a forward reference (`&.x` before `.x` is defined), or a reference to a value outside the document is accepted and stored verbatim;

- **cycles are not detected** ( `.a = &.b; .b = &.a;` parses), and cannot hang the library, which never follows a reference ( `bvn_dom_lookup` navigates literal structure only, stopping at a reference rather than dereferencing it);
- **resolution is entirely the application's responsibility**, including how to treat dangling paths and cycles.

A reference path is bounded by `max_reference_length` ( `error_reference_too_long` otherwise). In an array, references are homogeneous by **kind** — an array of references is uniform regardless of what its targets resolve to (their target types are unknown to the parser).

**Array indexing (spec 1.1).** A reference path may address array elements with `[N]` index suffixes ( `&.matrix[0][1]` ). Like the rest of the path the index is captured **verbatim and unresolved** at the byte layer — the library never dereferences a reference, so `bvn_dom_lookup` on the reference node returns its stored path string, not the target. The index is interpreted only when an application itself resolves that path string against the tree by calling `bvn_dom_lookup(doc, ".matrix[0][1]")` directly (this is the same path-walker the CLI `query` command exposes — `bovnar query .matrix[0][1]` → the element). The index syntax is a 1.1 feature: in a 1.0/unversioned document a `[` in a reference is `error_unexpected_input_byte`, exactly as a 1.0 reader reports.

Resolution semantics follow the array model (§7): a flat `/`-row matrix ( `[10,20,30]/[40,50,60]` ) is addressed as `[row][col]` — `&.matrix[0][1]` → `20` — and a 1-D array as `[i]`; genuine nested arrays ( `[[1,2],[3,4]]` ) descend one index per level. A partial index of a flat matrix ( `&.matrix[0]` ), an out-of-range index, or indexing a non-array does not resolve (the application sees no node).

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

## 4.6 Numbers

### Bare Number Literals

```
number = ["-"] ( int-led | dot-led ) [ dec-exponent ]

int-led      = DIGIT { DIGIT } [ "." { DIGIT } ]
dot-led      = "." DIGIT { DIGIT }
dec-exponent = ("e" | "E") [ "+" | "-" ] DIGIT { DIGIT }
```

- Only `e/E` accepted as exponent marker in bare literals (base-16 float values in quoted string literals use `p/P` — see §6.3)
- Leading zeros are valid ( `007` is accepted)
- A trailing dot without fractional digits is valid ( `123.` )
- `.` alone is a hard error

```
.i = 42;
.neg = -17;
.float = 3.14;
.trailing = 123.;
.dot_led = .5;
.sci = 1e6;
.neg_sci = -2.5e-3;
```

## Special Number Literals

```
special-number = "nan" | "inf" | "ninf"
```

The special IEEE-754 values are **bare reserved keywords** — `nan`, `inf`, and `ninf` (negative infinity) — with no sigil. Like `null` / `true` / `false` / `on` / `off`, they are lexed as symbols and reclassified to numeric special values by the validator; a bare word that is not one of these exact spellings (e.g. `infinity`, `nans`) stays an ordinary symbol. The stored token text is the keyword itself: `nan`, `inf`, `ninf`. A special-number keyword takes **no inline unit suffix** — supply a unit through the type annotation instead ( `<float:64,m/s>inf` ).

```
.not_a_number = nan;
.infinite = inf;
.neg_infinite = ninf;
```

## 4.7 Null Values

A null value is the **absence** of a raw-value token, or the reserved keyword `null`. It occurs when:

- At assignment level: `.key = ;` (nothing between `=` and `;`), or `.key = null;`
- In array context: leading/trailing commas, or consecutive commas: `[,1,,2,]`; a bare `null` element is equivalent

```
.null = ;                # null value
.also = null;            # the null keyword – identical to the empty slot
.items = [,1, ,2,];      # null, 1, null, 2, null
```

When a type annotation precedes a null value, the null carries the annotated type:

```
.null_typed = <uint:32> ;
.nulls_in_array = [<uint:32>, <sint:64> , ];
```

## 5. Type Annotations

### 5.1 Syntax

```
type-annotation = "<" ws type-spec ws ">"
```

The type annotation **must** be placed in one of four positions:

1. Immediately after the `=` sign of an assignment, before the value: `.key = <uint:32> 42;`
2. Before the opening `[` of an array — a **whole-array annotation** that is inherited by every element that does not carry its own annotation: `.ports = <uint:16> [80, 443, 8080];`
3. After the opening `[` of an array, before the first element.
4. After a `,` inside an array, before the next element.

In all cases the annotation comes **before** the value it describes.

#### Correct placement:

```
.an_int = <uint:32> 42;
.a_float = <float:64> 3.14;
.arr = [<uint:8> 1, <sint:16> -2];

# Whole-array annotation – applies to all elements that lack their own annotation
.ports = <uint:16> [80, 443, 8080];

# Per-element annotations override the whole-array annotation
.mixed = <uint:8> [1, <sint:8> -1, 255];
```

#### Incorrect placement (hard error):

```
.key<uint:32> = 42;    # ERROR: type annotation must follow '=', not the key
```

Seven type families are recognized in spec 1.0, plus `datetime` in spec 1.1:

Family	Keyword	Parameter syntax	Default Width
Unsigned integer	<code>uint</code>	<code>:width,_base,unit</code>	64
Signed integer	<code>sint</code>	<code>:width,_base,unit</code>	64
Binary floating-point	<code>float</code>	<code>:width,_base,unit</code> (base <code>_10</code> or <code>_16</code> only)	64
Fixed-point (Q-format)	<code>float_fix</code>	<code>:width,qN,unit</code>	64
Decimal floating-point (IEEE 754-2008)	<code>float_dec</code>	<code>:width,unit</code>	64
UTF-8 string	<code>utf8</code>	none (any parameter → <code>error_illegal_value_type</code> )	—

Family	Keyword	Parameter syntax	Default Width
Boolean	<code>bool</code>	none (any parameter → <code>error_illegal_value_type</code> )	—
Timestamp (spec 1.1)	<code>datetime</code>	<code>:width,epoch</code> (no numeric base/unit)	64

**The `datetime` family (spec 1.1).** A `datetime` value is a **signed integer count of seconds** since a named epoch — a timestamp, as distinct from a duration (which is just a number with a time unit, e.g. `<float:64,s>`). The carrier is validated exactly like `sint` (signed, decimal, range per width; negative values denote instants before the epoch). Its one family-specific parameter is the **epoch name**, one of:

`unix` (default), `tai`, `gps`, `mjd`, `ntp`, `galileo`, `glonass`, `y2000`, `beidou`.

A numeric base, `q`, or physical unit parameter is `error_illegal_value_type`; a fractional or exponent numeric carrier (e.g. `1.5` or `1e3` — as opposed to the sub-second fraction of an ISO-8601 literal, covered below) is `error_type_value_mismatch`. As a 1.1 feature it requires a `#!bovnar 1.1` declaration (§3.4) — in a 1.0/unversioned document `datetime` is `error_illegal_value_type`. Recover the civil date/time with the `bvn_datetime.h` helpers (`bvn_datetime_epoch_mjd()` → the epoch, then `bvn_dt_epoch_seconds_to_datetime()`).

```
#!bovnar 1.1
.created = <datetime:64,unix> 1750000000;    # 2025-06-15T...Z
.tai_t   = <datetime:tai>      1400000000;
.before  = <datetime>          -100;          # 100 s before 1970-01-01Z
```

**ISO-8601 literals (spec 1.1).** Instead of a raw integer, a `datetime` value may be written as an ISO-8601 literal:

```
datetime-literal = YYYY-MM-DD [ "T" HH:MM:SS [ "." fraction ] [ zone ] ]
zone              = "Z" | ( "+" | "-" ) HH:MM
```

- `YYYY-MM-DD` — a calendar date (interpreted at `00:00:00Z`)
- `YYYY-MM-DDTHH:MM:SS` — a date and time (UTC when no zone is given)
- a trailing `Z` (UTC) or a numeric `±HH:MM` time-zone offset
- an optional `.fraction` (one or more digits) after the seconds

The literal is converted at parse time to the integer epoch-seconds carrier; that integer is what is stored and re-emitted, so a pretty-print round-trip is idempotent (`2026-06-15` becomes `<datetime:64> 1781481600`). A **bare** literal with no annotation infers `<datetime:64,unix>`, so `.t = 2026-06-15;` is a timestamp without any annotation. Fields are strictly validated (month `01–12`, a valid day-of-month, hour `00–23`, minute `00–59`, second `00–60`, offset `±HH:MM` with two-digit components); a malformed or out-of-range literal is `error_invalid_datetime_literal`. A second of `60` is a **UTC leap second** and is accepted; because the carrier is whole epoch-seconds it normalises onto the following

second (so `2016-12-31T23:59:60Z` and `2017-01-01T00:00:00Z` store the same `unix` value), which is the correct POSIX reading.

A `±HH:MM` **offset** shifts the written civil time to true UTC before the conversion (`12:00:00+02:00` is `10:00:00Z`); for `tai` the offset is applied before the leap-second lookup, so the atomic value stays correct. The `.`, `Z`, and `±HH:MM` parts are valid only after a full `HH:MM:SS` time.

**Fractional seconds** (`.` then one or more digits — ISO 8601 sets no upper bound on the digit count) are accepted. The integer carrier is still **whole seconds**, so the fraction takes no part in the value's arithmetic or comparison (`<datetime:64> 1781524800` and a literal flooring to that second compare equal at the carrier). But the verbatim digits are **preserved**: they are surfaced to consumers as a string (the streaming `bvnr_data_t.frac_data` / `.frac_length`, or `bvn_dom_get_datetime_fraction()` on a DOM node) and are re-emitted so the value **round-trips**. A datetime that carries a fraction is pretty-printed back as an ISO literal — `2026-06-15T12:00:00.5Z` canonicalises to `<datetime:64> 2026-06-15T12:00:00.5Z` (always normalised to UTC `Z`, with the annotation made explicit) and re-printing that is idempotent; a datetime written as a bare integer carrier still canonicalises to the integer. The fraction is informational only — for sub-second values that participate in computation, use a finer integer carrier (e.g. milliseconds since the epoch).

The UTC→epoch conversion is **leap-second correct**: the civil epochs (`unix`, `mjd`, `ntp`, `y2000`) use the uniform 86 400 s/day scale (so `unix` is ordinary POSIX time) and `tai` applies the IERS leap-second table. The atomic GNSS epochs (`gps`, `galileo`, `glonass`, `beidou`) reject a literal with `error_datetime_literal_unsupported_epoch` — those scales have no round-trippable civil seconds inverse in this implementation, so supply an integer epoch-seconds carrier for them. A literal carries no unit, and an ISO literal under a non-`datetime` annotation is `error_type_value_mismatch`.

```
#!/bovnr 1.1
.a = 2026-06-15;
.b = 2026-06-15T12:00:00Z;
.c = <datetime:64,tai> 2017-01-01T00:00:00Z;
.d = 2026-06-15T12:00:00+02:00;
.e = 2026-06-15T12:00:00.5Z;
literal

# bare -> <datetime:64,unix>
# date-time, UTC
# tai: leap-second correct
# offset -> 10:00:00Z
# fraction preserved; round-trips as a
```

## Parameter syntax:

```

type-spec = param-type [ ":" type-param-list ]

param-type = "uint" | "sint" | "float" | "float_fix" | "float_dec" | "utf8"
            | "bool" | "datetime"          (* datetime: spec 1.1 *)

type-param-list = type-param { ",", type-param }

type-param = width-param      # decimal digits only, e.g. 32
            | base-param      # "_" followed by digits, e.g. _16
                               # (forbidden for float_fix and float_dec)
            | q-param         # "q" followed by digits, e.g. q8, q16
                               # (only valid for float_fix)
            | unit-param      # unit string, e.g. m/s, k~g·m/s²
            | epoch-param     # datetime epoch name, e.g. unix, tai (spec 1.1)

```

**Lexer note on `float_fix` / `float_dec`:** The lexer keyword state machine recognises `float` as the type-family prefix and then accumulates the remaining annotation bytes (`_fix`, `_dec`, or the parameter separator `:` / closing `>`) via `copy_type_byte` into `type_data`. `bnv_parse_type_annotation` then distinguishes `float`, `float_fix`, and `float_dec` from the fully accumulated string.

## 5.2 Parameters

Parameter	Syntax	Valid Values	Applies To
Width	<code>N</code> (decimal digits)	<code>0</code> , <code>16</code> , or any multiple of <code>32</code> up to <code>32768</code> for <code>float</code> ; <code>0</code> , <code>16</code> , <code>32</code> , <code>64</code> , <code>128</code> , <code>256</code> for <code>float_fix</code> / <code>float_dec</code> ; <code>0</code> to <code>BNV_MAX_INT_WIDTH</code> (32768) for <code>uint</code> / <code>sint</code>	<code>uint</code> , <code>sint</code> , <code>float</code> , <code>float_fix</code> , <code>float_dec</code>
Base	<code>_N</code> (underscore + decimal digits)	<code>2–62</code> ( <code>uint/sint</code> ); <code>64</code> , <code>85</code> are <b>uint-only</b> (their Base64/Ascii85 alphabets use <code>+</code> / <code>-</code> as digits, so signed values are illegal — <code>error_illegal_value_type</code> ); <code>float</code> accepts only <code>10</code> or <code>16</code> ; <b>forbidden</b> for <code>float_fix</code> and <code>float_dec</code>	<code>uint</code> , <code>sint</code> , <code>float</code>
Q (fractional bits)	<code>qN</code> (lowercase <code>q</code> + decimal digits)	$0 \leq N < \text{effective\_width}$	<b>float_fix only</b>
Unit	unit-string	See <a href="#">Units System</a> — supports compound units	<code>uint</code> , <code>sint</code> , <code>float</code> , <code>float_fix</code> , <code>float_dec</code>

Width `0` means "default width" — `bnv_effective_width` returns `64`.

For `float_fix`, the Q value (stored in `value_type_spec_t.base`) is the number of fractional bits. `bnv_effective_q` returns this value; `bnv_effective_base` always returns `10` for `float_fix` and `float_dec`.

For `float_dec`, the encoding is a custom binary-storage format (not DPD/BID wire format) parameterised by width:

Width	exp_bits	coeff_bits	bias	max decimal digits
16	6	9	24	2
32	8	23	101	7
64	10	53	398	16
128	14	113	6176	34
256	20	235	611867	70

For `float_fix`, the wire representation is a signed Q-format integer stored in `width` bits; the mathematical value is `raw_integer × 2(-Q)`.

### 5.3 Parameter Order

Parameters are **identified by class** — each class is recognised by its syntactic form — and at most one of each class may appear. They can appear in any order:

```
# All of these annotations are equivalent (parameter order is free):
.val = <uint:32,_10,no_unit> 42;
#      <uint:_10,no_unit,32>      - identical
#      <uint:no_unit,_10,32>      - identical
```

Because parameters are class-identified rather than positional, an empty positional slot is never needed, so the parameter list is parsed strictly: a comma must introduce a real parameter, and a `:` must be followed by at least one parameter. Empty, trailing, or doubled components — `<uint:8,>`, `<uint:8,,>`, `<uint:,_16>`, and the bare `<uint:>` — are `error_illegal_value_type`.



## 5.4 Examples

```
# Simple types
.a = <uint:32> 1000;
.b = <sint:16> -32768;
.c = <float:64> 3.14159;
.d = <utf8> "text";

# Width only
.g = <uint:8> 255;

# Base (requires quoted string for non-decimal)
.h = <uint:_16> "ff";
.i = <sint:_2> "101010";

# Unit
.j = <uint:32,no_unit> 42;           # explicitly dimensionless
.k = <float:64,m/s> 9.81;           # meters per second (compound)
.l = <uint:64,Ki~B> 1024;           # kibibytes
.m = <float:64,k~g·m/s²> 9.81;      # kilograms · meters per second squared
.n = <float:64,m*s> 1.0;           # meter-seconds (product)

# Type annotation with null value
.o = <uint:32> ;                   # null of type uint:32

# Fixed-point (Q-format): 16-bit, 8 fractional bits → resolution 2-8 ≈ 0.0039
.fx1 = <float_fix:16,q8> 3.14;

# Fixed-point: 32-bit, Q16 (16 fractional bits, 15 integer + 1 sign)
.fx2 = <float_fix:32,q16> -1.5;

# Fixed-point with unit
.fx3 = <float_fix:32,q8,m/s> 9.81;

# Fixed-point: width defaults to 64, Q=0 means pure integer fixed-point
.fx4 = <float_fix:64,q0> 42;

# Decimal float: 32-bit (7 significant decimal digits)
.df1 = <float_dec:32> 3.14;

# Decimal float: 64-bit (16 significant decimal digits) with unit
.df2 = <float_dec:64,Pa> 101325;

# Decimal float: 128-bit (34 significant decimal digits)
.df3 = <float_dec:128> 1.2345678901234567890123456789012345;

# float_fix and float_dec do NOT accept a base parameter:
# .bad = <float_fix:32,q8,_10> 1.0;    # ERROR: base forbidden for float_fix
# .bad = <float_dec:64,_10> 1.0;      # ERROR: base forbidden for float_dec
```

## 5.5 Non-decimal Base with Bare Numbers

A non-decimal base with a bare number literal is not caught by the validator (no error is raised). In practice, a bare non-decimal value such as `ff` is parsed as a symbol token, not a number, so type/value compatibility validation will flag the mismatch instead. Use a quoted string for non-decimal values:

```
# CORRECT: quoted string
.n = <uint:_16> "ff";    # hex value 255
```

## 6. Value Tokens

### 6.1 Type/Value Compatibility

Type Family	Accepts
<code>vt_plain</code> (default)	Any value
<code>vt_utf8</code>	String only
<code>vt_bool</code>	Boolean keyword only ( <code>true</code> / <code>false</code> / <code>on</code> / <code>off</code> )
<code>vt_uint</code>	Number or string (digits)
<code>vt_sint</code>	Number or string (digits, may be negative)
<code>vt_float</code>	Number or string (may have <code>.</code> , <code>e</code> / <code>E</code> ; base 16 strings use <code>p</code> / <code>P</code> ) — only base 10 or 16
<code>vt_float_fix</code>	Number or string (may have <code>.</code> , <code>e</code> / <code>E</code> ) — base 10 only
<code>vt_float_dec</code>	Number or string (may have <code>.</code> , <code>e</code> / <code>E</code> ) — base 10 only
<code>vt_datetime</code> (spec 1.1)	Number only — a decimal signed integer (epoch seconds); <code>.</code> / <code>e</code> and string carriers are rejected

### 6.2 Validation Rules per Numeric Type

#### uint (unsigned integer):

```
.valid = <uint:8> 255;
.valid = <uint:8> 0;

.invalid = <uint:8> -1;      # error_value_out_of_range (negative)
.invalid = <uint:8> 256;    # error_value_out_of_range (overflow)
```

#### sint (signed integer):

```
.valid = <sint:8> 127;
.valid = <sint:8> -128;

.invalid = <sint:8> 128;      # error_value_out_of_range
.invalid = <sint:8> -129;    # error_value_out_of_range
```

#### float (binary floating-point):

Valid widths: `0` (default 64), `16`, or any multiple of `32` up to `32768`. Base `10` (default) or `16` are accepted; all other bases are rejected.

```
.valid = <float:64> 3.14;
.valid = <float:64> 1e100;
.valid = <float:64> nan;
.valid = <float:16> 3.14;      # half-precision
.valid = <float:256> 3.14;    # 256-bit extended precision

.invalid = <float:8> 3.14;     # width 8 → error_illegal_value_type
.invalid = <float:12> 3.14;    # not 16 or a multiple of 32 → error_illegal_value_type
.invalid = <float:64,_2> 3.14; # base 2 → error_illegal_value_type
```

### float\_fix (fixed-point Q-format):

Valid widths: `0` (default 64), `16`, `32`, `64`, `128`, `256`. The Q parameter (`qN`) specifies fractional bits;  $0 \leq N < \text{effective\_width}$ . Base parameter (`_N`) is forbidden. The mathematical value of a fixed-point datum is  $\text{raw\_integer} \times 2^{(-Q)}$ .

**Range.** A `float_fix` value must lie within the declared format's signed range —  $\text{raw\_integer} = \text{round}(\text{value} \times 2^Q)$  must fit a signed `width`-bit field, i.e.  $\text{value} \in [-2^{(\text{width}-1-Q)}, 2^{(\text{width}-1-Q)} - 2^{(-Q)}]$ . A value outside that range is `error_value_out_of_range`, exactly as for `uint` / `sint` overflow (special numbers `nan` / `inf` / `ninf` are range-exempt, §6.4). The C encoders `bovn_float_to_fixNN` additionally **saturate** to the representable extreme on overflow rather than wrapping, so a fixed-point datum can never silently decode to an unrelated value.

```
.valid   = <float_fix:16,q8> 3.14;      # Q8 in 16-bit: range [-128, 127.99609375]
.valid   = <float_fix:16,q8> 127.99609375; # max in range
.valid   = <float_fix:16,q8> -128;      # min in range
.valid   = <float_fix:32,q16> -1.5;     # Q16 in 32-bit
.valid   = <float_fix:64,q0> 42;        # Q0 = pure integer, no fractional part
.valid   = <float_fix:32,q8,m/s> 9.81;

.invalid = <float_fix:16,q8> 128;       # exceeds Q8/16-bit range → error_value_out_of_range
.invalid = <float_fix:16,q0> 70000;     # exceeds signed-16-bit range →
error_value_out_of_range
.invalid = <float_fix:16,q16> 1.0;      # Q >= width → error_illegal_value_type
.invalid = <float_fix:8> 1.0;           # width 8 not in {0,16,32,64,128,256}
.invalid = <float_fix:32,q8,_10> 1.0;  # base param forbidden → error_illegal_value_type
```

### float\_dec (IEEE 754-2008 decimal floating-point):

Valid widths: `0` (default 64), `16`, `32`, `64`, `128`, `256`. Base parameter (`_N`) is forbidden (decimal base is implicit). Values are written as ordinary decimal literals or special numbers.

```
.valid   = <float_dec:32> 3.14;
.valid   = <float_dec:64,Pa> 101325;
.valid   = <float_dec:128> nan;

.invalid = <float_dec:8> 1.0;           # width 8 not valid → error_illegal_value_type
.invalid = <float_dec:64,_10> 1.0;     # base param forbidden → error_illegal_value_type
```

## 6.3 Digit Validation

Digits in values are checked against the declared base:

```
.value = <uint:_16> "ff";      # OK: f and f are valid in base 16
.value = <uint:_16> "fg";      # error_digit_not_in_base: 'g' > base 16
.value = <uint:_2> "1010";     # OK: valid binary
.value = <uint:_2> "210";      # error_digit_not_in_base
```

## Exponent Markers in Quoted String Literals

Bare number literals always use `e/E` as the exponent separator (§4.6). Quoted string number literals follow the same rule **except** for base-16 (`_16`) float values, where `p/P` must be used instead of `e/E`. This is required because `e` and `E` are valid hexadecimal digits: with base 16 you must write `"1.8p+2"`, not `"1.8e+2"` — the `e` would be read as a mantissa digit rather than an exponent marker, leaving the trailing `+2` invalid. Float values support a decimal base (the default) and base-16 (`_16`); the `p/P` exponent separator applies to the base-16 form.

```
.hex_float = <float:64,_16> "1.8p+2"; # OK: 1.816 × 22 = 6.0 (p = binary exponent)
.hex_mant  = <float:64,_16> "1.8e";   # OK: 'e' is a hex digit → mantissa 1.8e16
.dec_float = <float:64> 12.0;         # OK: default decimal base
```

The `p/P` exponent value is always interpreted as a decimal integer (the binary exponent bias), matching the C99 hexadecimal floating-point literal convention.

## 6.4 Special Number Semantics

`nan`, `inf`, `ninf` are accepted by any numeric type family (`uint`, `sint`, `float`, `float_fix`, `float_dec`) and in untyped context. `bvn_check_acc_range` explicitly returns `true` when `bvn_is_special_number_string` matches, bypassing all range validation regardless of the declared family. They are rejected only when the declared type is `utf8`, because the token type (`token_is_number`) is incompatible with a string-only family.

```
.okay_f64 = <float:64> inf;
.okay_f32 = <float:32> nan;
.okay_u8  = <uint:8>    nan;          # accepted – range check bypassed
.okay_s16 = <sint:16>   ninf;         # accepted – range check bypassed

# Fine in plain/untyped context too
.untyped = inf;                      # defaults to float:64
```

## 6.5 Inline Unit Suffix

A **scalar** number or string value may carry an optional unit suffix separated from the literal by **at least one whitespace character**. The suffix uses the same character set as the unit parameter inside a type annotation (`unit-param`).

Separator form	Example	Valid?
Space	<code>.a = 9.81 m/s;</code>	✓
No separator	<code>.b = 9.81m;</code>	✗ error

```
.distance = 100 m;      # plain integer with inline unit: meter
.speed    = 9.81 m/s;   # plain float with inline compound unit
.mass     = 70.0 k~g;    # with SI prefix
.storage  = 4 Gi~B;     # with IEC prefix
.ratio    = 3.14 no_unit; # explicitly dimensionless via inline suffix
```

The inline unit suffix is **forbidden inside arrays**; any character that would begin an inline unit — a letter, `_`, `$`, `%`, `(`, or a UTF-8 lead byte — following a value inside `[ ... ]` is a lexical error (`error_unexpected_input_byte`).

### Interaction with type annotations:

Situation	Outcome
No annotation, inline unit present	Inline unit is used as the effective unit
Annotation has no unit, inline unit present	Inline unit is used as the effective unit
Annotation unit and inline unit <b>match</b>	Valid; the common unit is used
Annotation unit and inline unit <b>differ</b>	<code>error_unit_mismatch</code>

```
# Annotation unit with no inline unit – normal
.dist = <float:64,m> 1.5;

# No annotation unit, inline unit – unit from suffix
.dist = <float:64> 1.5 m;

# Both present and identical – valid, redundant but allowed
.dist = <float:64,m> 1.5 m;

# Both present and different – error_unit_mismatch
.dist = <float:64,m> 1.5 s;      # ERROR: annotation says m, inline says s
```

An invalid inline unit string (unrecognised base unit, bad prefix, etc.) produces `error_unit_illegal`. The suffix may appear after `no_unit` checks as with any other unit string.

## 7. Arrays

### 7.1 Row Syntax

An array consists of one or more bracket-enclosed **rows** separated by `/`:

```
.array_2d = [1, 2, 3]/[4, 5, 6];
```

This produces the event sequence:

```
ev_array_row_start → 3× ev_data → ev_array_row_end
ev_array_dim_start → ev_array_row_start → 3× ev_data → ev_array_row_end
```

## 7.2 Null Elements

Leading/trailing commas, consecutive commas, and the bare keyword `null` all produce null elements:

```
.items = [,1,,2,]; # 5 elements: null, 1, null, 2, null
.one   = [null];  # 1 element: a single null
```

**Empty arrays.** A row with nothing between the brackets is **empty** — zero elements, no `ev_data` emitted. It is distinct from a one-null row:

```
.empty = [];      # 0 elements
.null1 = [null];  # 1 element (null)
```

So `bvn_dom_array_count([])` is `0` and `bvn_dom_array_count([null])` is `1`. The canonical serialiser writes a null array element as the explicit `null` keyword (e.g. `[, 1]` round-trips through `[null, 1]`), so an empty slot never collides with the empty array. Empty rows interact with row-size consistency (§7.3): all `/`-rows must share one width, and `0` is a valid width — `[]/[]` is two empty rows, but `[]/[1]` is `error_array_row_size_mismatch`.

## 7.3 Row-Size Consistency

A single array's `/`-separated dimension rows (e.g. `[1,2,3]/[4,5,6]`) must all have the same element count. A mismatch produces `error_array_row_size_mismatch`; the lexer performs this check as each row closes, so an offending row is rejected at the earliest possible byte — the element that overshoots, or the `]` that falls short. This makes one `/`-array a clean rectangular N-dimensional block.

Comma-separated elements are distinct values, but since spec 1.0 they are no longer unconstrained: they must be **homogeneous** (§7.4).

## 7.4 Element Homogeneity

Since spec 1.0 the elements of an array must be **homogeneous**. The rule is "shape uniform, fields free", checked over the materialised value (above the lexer; it complements the streaming reader's per-value type and unit checks):

- **Kind.** Every non-null element shares the same kind — number, string, symbol, bool, reference, octet stream, array, or struct. `[1, "two"]` and `[1, {x=1;}]` are `error_array_element_type_mismatch`.

- **Dimension** (bare scalar arrays and matrices). Numeric elements must share the same physical dimension; the numeric encodings (`uint`, `sint`, `float`, `float_fix`, `float_dec`) may mix. `[<float:64,m> 1.0, <float:64,k~g> 2.0]` (length vs mass) is rejected; `[1, 2.5, 3]` (all dimensionless) is fine. Each currency is its own dimension, so a bare array may not mix `$USD` and `$EUR` values.
- **datetime (spec 1.1)**. A `datetime` is its own kind: it does **not** mix with the plain numeric encodings above (`[<datetime:64,unix> 1, <sint:64> 2]` is `error_array_element_type_mismatch`). And, exactly as for currencies, its **epoch is a dimension** — a bare array may not mix epochs (`[<datetime:64,unix> 1, <datetime:64,tai> 2]` is rejected); a homogeneous same-epoch datetime array is fine. (These are materialised-document/DOM-tier rules, like the rest of §7.4.)
- **Rectangular** (nested arrays). Sibling sub-arrays must have the same length and recursively-matching element shape: `[[1,2],[3,4]]` is valid, `[[1,2],[3,4,5]]` is `error_array_row_size_mismatch`.
- **Structs — same keys, fields free**. Sibling structs must share the same keys, in order, with the same per-field kinds and nesting. Differing keys are `error_struct_shape_mismatch`; a field that is a number in one record and a string in another is `error_array_element_type_mismatch`. But a scalar field may carry a **different unit** in each record, and a list field a **different length** — so a multi-currency ledger and per-record argument lists are valid.
- **Null is a hole**. `null` / empty array elements match any shape and never establish or break homogeneity, so sparse arrays like `[1, , 3]` remain valid.

```
.ok_scalars = [1, 2.5, 3];           # valid: same (no) dimension, encodings may mix
.ok_matrix  = [[1, 2], [3, 4]];      # valid: rectangular
.ok_records = [{.cur = USD; .bal = <float_dec:64,$USD> 1.0;},
               {.cur = EUR; .bal = <float_dec:64,$EUR> 2.0;}); # valid: same keys, fields
free
.ok_sparse  = [1, , 3];              # valid: null hole

.bad_kind   = [1, "two"];            # error_array_element_type_mismatch
.bad_dim    = [<float:64,m> 1.0, <float:64,k~g> 2.0]; # error_array_element_type_mismatch
.bad_ragged = [[1, 2], [3, 4, 5]];   # error_array_row_size_mismatch
.bad_keys   = [{.x = 1;}, {.y = 1;}]; # error_struct_shape_mismatch
```

A bare array of measurements is therefore uniform — a consumer may treat its elements identically — while records (structs) describe genuinely different things. **Heterogeneous data is modelled with a struct, not an array.** (This is a deliberate tightening over pre-1.0 drafts, which allowed ragged and mixed-type arrays; it also means a ragged or mixed-type JSON array has no bovnar representation and the `json → bvnr` converter rejects it rather than losing structure.)

## 7.5 Array Elements with Type Annotations

Individual elements may carry type annotations. The annotation appears **before** each element value, **after** the comma or opening bracket:

```
.mixed = [<uint:8> 1, <sint:8> -1, <float:64> 3.14];
.nulls = [<uint:32> , <sint:64> ];    # null, null with types
```

## 7.6 Constraints

Constraint	Limit
Array nesting depth	<code>max_array_nesting</code> — default 64, hard cap 255
Total array items	<code>max_array_items</code> (configurable; 0 → 2 147 483 647 internal default)

## 8. Structs (Scopes)

### 8.1 Syntax

```
struct = "{" ws { assignment } ws "}"
```

Structs group related assignments into a nested scope:

```
.person = {
  .name = "Alice";
  .age = 30;
  .address = {
    .street = "123 Main St";
    .city = "Springfield";
  };
};
```

**Key uniqueness.** Keys must be unique within a single scope — a struct, and the top-level document. A repeated key is `error_duplicate_struct_key`, so lookup, references, and iteration always agree on the value of a key:

```
.bad = {.x = 1; .x = 2;};    # error_duplicate_struct_key
.ok  = {.x = 1; .y = 2;};    # fine
```

The same key in different scopes is unrelated and always allowed (e.g. `.a = {.x = 1;}; .b = {.x = 2;};`). The rule is enforced over the materialised document, alongside array homogeneity (§7.4).

### 8.2 Nesting

Structs can be nested up to `max_struct_nesting` levels — default 64, hard cap 255 (the limit field is a `uint8_t`; setting it to 0 selects the default of 64). Exceeding the configured limit produces `error_struct_nesting_too_high`.



## 8.3 Empty Structs

```
.empty = {};
```

## 8.4 Structs as Array Elements

```
.people = [
    {.name = "Alice"; .age = 30;},
    {.name = "Bob"; .age = 25;}
];
```

## 8.5 Unmatched Braces

A `}` seen with `struct_nesting_level == 0` is `error_illegal_struct_close`.

# 9. Octet Streams (Binary Mode)

## 9.1 Overview

A NUL byte (`0x00`) where a value is expected switches from text mode to binary chunk mode. The parallel UTF-8 validator is suspended for the duration.

## 9.2 Wire Protocol

```
octet-stream = 0x00 { os-chunk } 0x00
os-chunk      = 0x01 os-length os-data
os-length     = uint16 (little-endian); 0x0000 encodes 65536 bytes
os-data       = exactly os-length bytes
```

Byte	Meaning
<code>0x00</code>	End-of-stream marker (at binary level)
<code>0x01</code>	Data chunk follows
<code>0x01</code> + <code>LL LL</code> + <code>data</code>	Chunk of <code>L</code> bytes ( <code>0x0000</code> = 65536)
Any other tag byte	<code>error_octet_stream_out_of_sync</code>

## 9.3 Events

```
ev_octet_stream_start → (emitted on the leading 0x00)
ev_data               → (emitted for each binary chunk)
ev_octet_stream_end   → (emitted on the trailing 0x00)
```

## 9.4 Example

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

This binary region encodes two chunks: `"hello"` (5 bytes) and `"bye"` (3 bytes), producing:

```
ev_octet_stream_start
ev_data (type=octet_stream, data="hello", length=5)
ev_data (type=octet_stream, data="bye", length=3)
ev_octet_stream_end
```

## 9.5 Constraints

- File size: `max_file_size` (0 → **unlimited / endless**, the default — no accumulated size limit; set to `16777216` for the recommended 16 MiB cap)
- Octet stream bytes contribute to the file size limit but not to `max_text_bytes`
- EOF inside an octet stream region preserves the current error code instead of overwriting with `error_got_incomplete_bvnr_stream`

# 10. Default Type Synthesis

When a number or string value carries **no explicit** type annotation (i.e. the validator's `value_type` is still `vt_plain`), the validator **synthesises** a default type annotation before emitting `ev_data`.

## 10.1 Rules

Value Form	Synthesised Type
Quoted string	<code>&lt;utf8&gt;</code>
Boolean keyword ( <code>true</code> / <code>false</code> / <code>on</code> / <code>off</code> )	<code>&lt;bool&gt;</code>
Special number ( <code>nan</code> , <code>inf</code> , <code>ninf</code> )	<code>&lt;float:64,_10,no_unit&gt;</code>
Number with <code>.</code> or <code>e</code> / <code>E</code> (float literal)	<code>&lt;float:64,_10,no_unit&gt;</code>
Negative integer	<code>&lt;sint:64,_10,no_unit&gt;</code>
Plain integer	<code>&lt;uint:64,_10,no_unit&gt;</code>
ISO-8601 datetime literal, no annotation (spec 1.1)	<code>&lt;datetime:64,unix&gt;</code>

**A bare integer inside a `datetime` array** inherits the array's `datetime` type (width and epoch) rather than synthesising `uint` / `sint`, so a datetime array's canonical form — annotation on the first element, bare integers after — stays homogeneous.

`float_fix` and `float_dec` are never auto-synthesised. `float_fix` requires a Q parameter that cannot be inferred from the value literal; `float_dec` requires an explicit choice of decimal encoding. Both families must be introduced by an explicit type annotation.

## 10.2 Event Sequence

The synthesised annotation produces the **same** event sequence as an explicit one. For numeric types (`uint`, `sint`, `float`) the sequence includes three parameter events; for `utf8` no parameter events are emitted:

```
# numeric (uint / sint / float)
ev_type_annotation_start
ev_type_annotation_type_family → "uint" / "sint" / "float"
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

# string
ev_type_annotation_start
ev_type_annotation_type_family → "utf8"
ev_type_annotation_end
ev_data
```

## 10.3 Examples

```
# No annotation → synthesised <uint:64,_10,no_unit>
.x = 42;

# No annotation → synthesised <float:64,_10,no_unit>
.y = 3.14;
.z = inf;

# No annotation → synthesised <sint:64,_10,no_unit>
.w = -7;

# No annotation → synthesised <utf8>
.s = "hello";
```

# 11. Units System

## 11.1 Base Units

The unit system supports **163 named base units** across SI, IEC-binary, Imperial/US customary, CGS electromagnetic, radiation, electrical-power, and other categories. The table below covers the SI base units, all 22 named SI-derived units (degree Celsius among them — it is listed under Non-SI units accepted for use with SI below, where it is conventionally grouped), and the other non-SI units accepted for use with SI. For the

complete reference — including Imperial/US customary, CGS, radiation, electrical-power, rotational, textile, surveying, volume, and other unit families — see [doc/2\\_bovnar\\_unit\\_system.md](#).

## SI base units and digital units

Symbol	Unit	Enum
<code>b</code>	bit	<code>bu_bit</code>
<code>B</code>	byte	<code>bu_byte</code>
<code>s</code>	second	<code>bu_second</code>
<code>m</code>	meter	<code>bu_meter</code>
<code>g</code>	gram	<code>bu_gram</code>
<code>A</code>	ampere	<code>bu_ampere</code>
<code>K</code>	kelvin	<code>bu_kelvin</code>
<code>mol</code>	mole	<code>bu_mol</code>
<code>cd</code>	candela	<code>bu_candela</code>

## Named SI-derived units

Symbol	Unit	Enum
<code>Hz</code>	hertz	<code>bu_hertz</code>
<code>N</code>	newton	<code>bu_newton</code>
<code>Pa</code>	pascal	<code>bu_pascal</code>
<code>J</code>	joule	<code>bu_joule</code>
<code>W</code>	watt	<code>bu_watt</code>
<code>V</code>	volt	<code>bu_volt</code>
<code>Ω</code>	ohm	<code>bu_ohm</code>
<code>F</code>	farad	<code>bu_farad</code>
<code>C</code>	coulomb	<code>bu_coulomb</code>
<code>S</code>	siemens	<code>bu_siemens</code>
<code>Wb</code>	weber	<code>bu_weber</code>
<code>T</code>	tesla	<code>bu_tesla</code>
<code>H</code>	henry	<code>bu_henry</code>
<code>lm</code>	lumen	<code>bu_lumen</code>
<code>lx</code>	lux	<code>bu_lux</code>
<code>Bq</code>	becquerel	<code>bu_becquerel</code>

Symbol	Unit	Enum
Gy	gray	bu_gray
Sv	sievert	bu_sievert
kat	katal	bu_katal
rad	radian	bu_radian
sr	steradian	bu_steradian

## Non-SI units accepted for use with SI

Symbol	Unit	Enum
L, l	liter	bu_liter
min	minute	bu_minute
h	hour	bu_hour
d	day	bu_day
wk	week	bu_week
yr	year	bu_year
°, deg, degr, degree, degrees	degree (angle)	bu_degree
°C, degC, degrC	degree Celsius	bu_celsius
t	tonne	bu_tonne
bar	bar	bu_bar
eV	electronvolt	bu_electronvolt
Da, dalton, amu, u	dalton	bu_dalton
au	astronomical unit	bu_astronomical_unit
ha	hectare	bu_hectare

For Imperial/US customary length (in, ft, yd, mi, nmi, Å, ly, pc, fur, fath, ch, rd, thou/mil), mass (lb, oz, gr, st, tn\_sh, tn\_l, oz\_t, ct, slug, dr, dwt), temperature (°F, Ra), pressure (atm, mmHg, Torr, psi, inHg, at), energy (cal, Btu, erg, thm, ft\_lb), power (hp, PS / CV), force (lbf, dyn, kip, kgf), speed/frequency/rotation (kn, rpm, rev), volume (US and UK gallons, pints, fluid ounces, and many more), area (ac, barn), angle (arcmin, arcsec, grad), CGS (P, St, G, Mx, Oe, sb, ph, Gal), radiation (Ci, R, rem), logarithmic (Np, dB), electrical power (var, VA), acceleration (gn), time (mo, fn), textile linear density (tex, den), and apothecary/dry volume (fl\_dr, minim, pk, bsh) — see the [Unit System Reference](#).

## 11.2 SI Prefixes

Prefix	Symbol	Factor	Enum
quetta	Q	$10^{30}$	si_quetta
ronna	R	$10^{27}$	si_ronna
yotta	Y	$10^{24}$	si_yotta
zetta	Z	$10^{21}$	si_zetta
exa	E	$10^{18}$	si_exa
peta	P	$10^{15}$	si_peta
tera	T	$10^{12}$	si_tera
giga	G	$10^9$	si_giga
mega	M	$10^6$	si_mega
kilo	k	$10^3$	si_kilo
hecto	h	$10^2$	si_hecto
deca	da	$10^1$	si_deca
deci	d	$10^{-1}$	si_deci
centi	c	$10^{-2}$	si_cent
milli	m	$10^{-3}$	si_milli
micro	μ (or u )	$10^{-6}$	si_micro
nano	n	$10^{-9}$	si_nano
pico	p	$10^{-12}$	si_pico
femto	f	$10^{-15}$	si_femto
atto	a	$10^{-18}$	si_atto
zepto	z	$10^{-21}$	si_zepto
yocto	y	$10^{-24}$	si_yocto
ronto	r	$10^{-27}$	si_ronto
quecto	q	$10^{-30}$	si_quecto

**Note:** μ = U+00B5 (MICRO SIGN), encoded as 0xC2 0xB5 in UTF-8. ASCII u is accepted as an input-only alias for the micro prefix (u~m = μ~m); the canonical output is always μ.

## 11.3 IEC Binary Prefixes

Prefix	Symbol	Factor	Enum
kibi	Ki	$2^{10}$	iec_kibi

Prefix	Symbol	Factor	Enum
mebi	Mi	2 <sup>20</sup>	iec_mebi
gibi	Gi	2 <sup>30</sup>	iec_gibi
tebi	Ti	2 <sup>40</sup>	iec_tebi
pebi	Pi	2 <sup>50</sup>	iec_pebi
exbi	Ei	2 <sup>60</sup>	iec_exbi
zebi	Zi	2 <sup>70</sup>	iec_zebi
yobi	Yi	2 <sup>80</sup>	iec_yobi
robi	Ri	2 <sup>90</sup>	iec_robi
quebi	Qi	2 <sup>100</sup>	iec_quebi

## 11.4 Unit Notation

The unit system supports **compound units** composed of multiple base-unit terms combined with product and division separators.

```

compound-unit = "no_unit" | unit-expr
unit-expr     = unit-factor { unit-sep unit-factor }
unit-factor   = unit-component | "(" unit-expr ")"

unit-sep      = "*" | "/" | "."      (* "." = U+00B7 MIDDLE DOT *)
unit-component = [ prefix "~" ] base-unit [ unit-exponent ]

```

### Separators:

Separator	Code Point	Meaning
*	U+002A	Product (multiplication)
·	U+00B7	Product (multiplication) — visually preferred
/	U+002F	Division — subsequent components are in the denominator

The **·** (middle dot, U+00B7, encoded as **0xC2 0xB7**) and **\*** (asterisk) are semantically equivalent; both indicate multiplication of the adjacent unit components.

The **/** separator divides the preceding components by the following ones. The first **/** switches all subsequent components into the denominator; additional **/** separators do not toggle back to the numerator. Every component after the first **/** is always in the denominator.

**Parenthesised grouping.** A **(...)** group is a sub-expression evaluated independently; like any factor it obeys the latching denominator, so a **/** before a group negates the group's net exponents as a whole. Thus **k~g/(m·s<sup>2</sup>)** parses to **kg·m<sup>-1</sup>·s<sup>-2</sup>** (identical to **k~g/m·s<sup>2</sup>**), while **(k~g/m)·s<sup>2</sup>** is **kg·m<sup>-1</sup>·s<sup>2</sup>**. An explicit separator is required before a group (**m·(s)**), not

`m(s)`); a group is not followed by its own exponent (`(m·s)2` is rejected); parentheses must balance and nest no deeper than 16. The writer emits the canonical, parenless form.

#### Within each `unit-component`:

- When a prefix is present, the separator `~` between the prefix and the base unit is **mandatory**.
- A bare base unit with no prefix requires no separator.

```
# Simple (single-component) units – same as before
.time = <float:64,s> 2.5;           # seconds
.speed = <float:64,k~m> 1.5;        # kilometers (kilo-meter)

# Compound units
.velocity = <float:64,m/s> 9.81;    # meters per second
.accel = <float:64,m/s²> 9.81;      # meters per second squared
.force = <float:64,k~g·m/s²> 9.81;  # kilogram-meters per second squared
.energy = <float:64,k~g·m²/s²> 1000; # kilogram-square-meters per second squared
.moment = <float:64,m*s> 1.0;       # meter-seconds
.area_density = <float:64,k~g/m²> 5.0; # kilograms per square meter
.three_term = <float:64,k~g·m·s⁻²> 9.81; # equivalent to k~g·m/s²
.pressure = <float:64,k~g/(m·s²)> 101325; # grouped denominator (= k~g/m·s²)

# Explicitly dimensionless
.no_unit_float = <float:64,no_unit> 3.14;
```

## 11.5 Unit Exponents

Exponents can be written in two forms:

Form	Example	Meaning
Unicode superscript	<code>m<sup>2</sup></code> , <code>m<sup>-3</sup></code>	Using U+00B2/U00B3 etc.
ASCII caret	<code>m^2</code> , <code>m^-3</code> , <code>m^+2</code>	Using <code>^[+-]?[1-9]</code> (single digit; <code>^0</code> is not a valid exponent)

#### Superscript mapping:

Glyph	Code Point	Exponent
<code>¹</code>	U+00B9	1
<code>²</code>	U+00B2	2
<code>³</code>	U+00B3	3
<code>⁴</code>	U+2074	4
<code>⁵</code>	U+2075	5
<code>⁶</code>	U+2076	6
<code>⁷</code>	U+2077	7
<code>⁸</code>	U+2078	8



Glyph	Code Point	Exponent
9	U+2079	9
+	U+207A	positive sign (no-op)
-	U+207B	negate exponent

## 11.6 Examples

```
.distance = <float:64,k~m> 1.5;           # kilometers
.mass = <float:64,g> 500;                   # grams
.velocity = <float:64,m/s> 9.81;             # meters per second
.acceleration = <float:64,m/s²> 9.81;        # meters per second squared
.pressure = <float:64,Pa> 101325;            # pascals (= N/m² = k~g/(m·s²))
.energy = <float:64,k~J> 1000;               # kilojoules
.storage = <uint:64,Ti~B> 2;                 # tebibytes
.frequency = <float:64,k~Hz> 2.4;            # kilohertz
.force = <float:64,k~g·m/s²> 9.81;           # kilogram-meters per second squared
.momentum = <float:64,k~g·m/s> 0.5;         # kilogram-meters per second
.density = <float:64,k~g/m³> 7800;           # kilograms per cubic meter
```

## 11.7 Compound Unit Constraints

Constraint	Limit
Maximum components per compound unit	8 ( <code>BVNR_MAX_UNIT_COMPONENTS</code> )

If a compound unit string contains more than `BVNR_MAX_UNIT_COMPONENTS` components after parsing, the validator raises `error_unit_illegal`.

Empty components between separators (e.g., `m//s`, `m*·s`) produce `error_unit_illegal`.

## 11.8 The `no_unit` Keyword

The literal string `no_unit` in the unit parameter position means "explicitly dimensionless":

```
.dimensionless = <uint:32,no_unit> 42;
```

Omitting the unit parameter also defaults to dimensionless and produces a **different** internal representation: `BVN_UNIT_NO_PREFIX(bu_none)` with `num_components == 1` and `base == bu_none`.

An explicit `no_unit` parameter yields `BVN_UNIT_NONE` with `num_components == 0`. Both forms are semantically equivalent — they compare as compatible via `bvn_units_compatible` and both serialize to `"no_unit"` via `bvn_unit_to_string` — but they are structurally distinct internal states.

## 12. Validation & Constraints

### 12.1 UTF-8 Validation

Constraint	Error
Non-UTF-8 byte sequence	<code>error_invalid_utf8_byte</code>
Overlong encoding	Rejected (by UTF-8 rules)
Surrogate halves (U+D800-U+DFFF)	Rejected

### 12.2 Size Limits

Quantity	Configurable	Default	Overflow Error
Identifier length	Yes ( <code>max_identifier_length</code> )	255	<code>error_identifier_too_long</code>
String length	Yes ( <code>max_string_length</code> )	65535	<code>error_string_too_long</code>
Number length	Yes ( <code>max_number_length</code> )	65535	<code>error_number_too_long</code>
Symbol length	Yes ( <code>max_symbol_length</code> )	255	<code>error_symbol_too_long</code>
Reference length	Yes ( <code>max_reference_length</code> )	65535	<code>error_reference_too_long</code>
Array items	Yes ( <code>max_array_items</code> )	2 147 483 647	<code>error_too_many_array_items</code>
Text bytes	Yes ( <code>max_text_bytes</code> )	2 147 483 647	<code>error_text_data_too_long</code>
File size	Yes ( <code>max_file_size</code> )	0 (→ unlimited / endless)	<code>error_file_too_long</code>
Struct nesting	Yes ( <code>max_struct_nesting</code> )	0 (→64 internal)	<code>error_struct_nesting_too_high</code>
Array nesting	Yes ( <code>max_array_nesting</code> )	0 (→64 internal, hard cap 255)	<code>error_array_nesting_too_high</code>

Setting most fields to `0` in `bvnr_read_flags_t` substitutes an internal default — **64** for both nesting depths, and **2 147 483 647** ( $2^{31} - 1$ ) for `max_array_items` and `max_text_bytes`. **`max_file_size` is the exception: `0` means unlimited / endless** (no byte-count cap accumulated), so endless streams are the default; set a positive value to cap. These defaults apply to both the reader and the writer. The writer does not internally limit array items, text bytes, or file size.

## 12.3 Value Validation

Check	Error
Number in base-N string contains out-of-base digit	<code>error_digit_not_in_base</code>
Integer value exceeds declared width	<code>error_value_out_of_range</code>
<code>float_fix</code> value outside the declared Q-format range (§6.2)	<code>error_value_out_of_range</code>
Negative number with <code>uint</code> type	<code>error_value_out_of_range</code>
Mismatched type family for value token	<code>error_type_value_mismatch</code>
Dot or exponent in integer-typed value	<code>error_type_value_mismatch</code>
Malformed or out-of-range ISO-8601 datetime literal (spec 1.1)	<code>error_invalid_datetime_literal</code>
ISO-8601 literal for an atomic GNSS epoch — <code>gps</code> / <code>galileo</code> / <code>glonass</code> / <code>beidou</code> (spec 1.1)	<code>error_datetime_literal_unsupported_epoch</code>
Invalid unit string	<code>error_unit_illegal</code>
Empty parameter component — <code>&lt;uint:&gt;</code> , <code>&lt;uint:8,&gt;</code> , <code>&lt;uint:8,,&gt;</code> , <code>&lt;uint:,_16&gt;</code> (§5.3)	<code>error_illegal_value_type</code>
Wholly empty annotation <code>&lt;&gt;</code> / <code>&lt; &gt;</code> (no family keyword)	<code>error_unexpected_input_byte</code>
<code>utf8</code> (string) type given a number value	<code>error_type_value_mismatch</code>
Non-decimal base for float type	<code>error_illegal_value_type</code>
Base <code>_N</code> ( $N \neq 10$ , $N \neq 16$ ) for <code>float</code>	<code>error_illegal_value_type</code>
Base param ( <code>_N</code> ) used with <code>float_fix</code> or <code>float_dec</code>	<code>error_illegal_value_type</code>
Q param ( <code>qN</code> ) used with family other than <code>float_fix</code>	<code>error_illegal_value_type</code>
Q value $\geq$ effective width for <code>float_fix</code>	<code>error_illegal_value_type</code>
Invalid float width (not 0/16/multiple-of-32) for <code>float</code>	<code>error_illegal_value_type</code>
Invalid <code>float_fix</code> / <code>float_dec</code> width (not 0/16/32/64/128/256)	<code>error_illegal_value_type</code>

## 12.4 Array Validation

Check	Error	Tier
Element count mismatch across the <code>/</code> -dimension rows of a single array	<code>error_array_row_size_mismatch</code>	Streaming
Ragged sibling sub-arrays — differing lengths between siblings (§7.4)	<code>error_array_row_size_mismatch</code>	DOM

Check	Error	Tier
Array elements of mixed kind or dimension, or a struct field that differs in kind across record elements (§7.4)	<code>error_array_element_type_mismatch</code>	DOM
Array nesting overflow (exceeds <code>max_array_nesting</code> )	<code>error_array_nesting_too_high</code>	Streaming
Comma outside array context	<code>error_unexpected_input_byte</code>	Streaming

## 12.5 Struct Validation

Check	Error	Tier
Unmatched <code>}</code>	<code>error_illegal_struct_close</code>	Streaming
A key repeated within one scope (struct or top-level document) (§8.1)	<code>error_duplicate_struct_key</code>	DOM
Struct array elements with differing key sets (§7.4)	<code>error_struct_shape_mismatch</code>	DOM
Nesting depth exceeded	<code>error_struct_nesting_too_high</code>	Streaming

**Validation tier.** Streaming checks are raised by the pull-based reader (`bvnr_read`) during the `on_verified` event stream. DOM checks — the spec-1.0 homogeneity (§7.4), struct-shape, and duplicate-key (§8.1) rules — are enforced only when a document is **materialised** into a tree (`bvn_dom_parse`), because they require comparing sibling elements that the streaming reader sees one at a time. A streaming-only consumer therefore accepts a heterogeneous array, a ragged sibling sub-array, a shape-mismatched record set, or a duplicate key without error; a full spec-1.0 implementation must apply these four checks (`error_array_row_size_mismatch` for ragged siblings, `error_array_element_type_mismatch`, `error_struct_shape_mismatch`, `error_duplicate_struct_key`) in its document/tree API. See the conformance tool's "Validation tiers" (§3 of `doc/7_bovnr_conformance.md`).

## 12.6 Identifier Validation

Check	Error
Empty key ( <code>.=</code> or <code>.</code> + non-identifier char)	<code>error_empty_identifier</code>
Invalid character in identifier	<code>error_unexpected_input_byte</code>

## 12.7 String Validation

Check	Error
Unknown escape sequence (incl. <code>\x</code> / <code>\u</code> outside spec 1.1)	<code>error_illegal_escape_sequence</code>
<code>\u{...}</code> surrogate or value <code>&gt; U+10FFFF</code> (spec 1.1)	<code>error_invalid_codepoint</code>

Check	Error
<code>\x</code> byte(s) that break the string's UTF-8 validity (spec 1.1)	<code>error_invalid_utf8_byte</code>
Control byte in string	<code>error_unexpected_input_byte</code>
String length exceeded	<code>error_string_too_long</code>

## 12.8 Unit Validation

Check	Error
Invalid unit string	<code>error_unit_illegal</code>
Compound unit exceeds <code>BVNR_MAX_UNIT_COMPONENTS</code>	<code>error_unit_illegal</code>
Empty component between separators	<code>error_unit_illegal</code>
Unit string too long	<code>error_unit_too_long</code>
Inline unit suffix differs from type-annotation unit	<code>error_unit_mismatch</code>
Inline unit suffix inside an array element	<code>error_unexpected_input_byte</code>

## 12.9 Octet Stream Validation

Check	Error
Unknown tag byte in binary mode	<code>error_octet_stream_out_of_sync</code>
Incomplete chunk read	<code>error_read_complete_chunk_failed</code>
EOF in binary mode	Preserves current error

# 13. Error Handling & Recovery

## 13.1 Error Model

Errors are reported through the `on_error` callback:

```
typedef void (*bvnr_on_error_fn)(
    void* userdata, error_code_t err,
    uint64_t line, uint64_t column,
    uint32_t byte, uint64_t offset);
```

## 13.2 Recovery Mode

When `bvnr_read_flags_t.continue_on_error` is `true`, the parser enters **resync mode** after any error:

1. The `on_error` callback is invoked with error details

2. A **resync state machine** ( `state_t: resync, resync_string, resync_string_escape, resync_comment` ) skips bytes
3. Tracking bracket `[]` and brace `{}` nesting
4. On encountering `;` at the saved nesting depth, parsing resumes
5. `recovery_count` (accessible via `bvnr_reader_get_recovery_count` ) is incremented immediately when an error triggers entry into resync mode

### State machine behavior during resync:

Byte(s)	Action
<code>0x00–0xFF</code> (most)	Skip (consume and continue)
<code>"</code>	Enter <code>resync_string</code> : skip until matching <code>"</code>
<code>#</code>	Enter <code>resync_comment</code> : skip until newline
<code>[</code> , <code>{</code>	Increment <code>resync_depth</code>
<code>]</code> , <code>}</code>	Decrement <code>resync_depth</code> ; if 0, emit array/struct close
<code>;</code> at depth 0	Reset state, resume normal parsing

## 13.3 EOF in Resync

If EOF is reached while in any resync state, `error_got_incomplete_bvnr_stream` is fired as a **second** `on_error` notification in addition to the original error that triggered resync entry. Both error codes are delivered to the caller's `on_error` callback in order: the original error first, then `error_got_incomplete_bvnr_stream` when EOF is detected.

## 14. Formal EBNF

The complete grammar is maintained as a standalone file:

`doc/5_bovnar.ebnf`

The grammar uses ISO/IEC 14977:1996 notation and is derived from and verified against the reference implementation. It covers:

- Top-level stream and assignment structure
- Type annotations (seven core families: `uint`, `sint`, `float`, `float_fix`, `float_dec`, `utf8`, `bool`; plus `datetime` in spec 1.1)
- Value forms: numbers, special numbers, booleans, strings, symbols, references, arrays, structs, octet streams, inline unit suffixes
- Lexical primitives and UTF-8 byte class definitions
- Unit sub-grammar (SI/IEC prefixes, base units, compound units, exponents)

- Constraints not expressible in context-free EBNF (UTF-8 validity, BOM placement, nesting limits, type/value compatibility, error recovery behaviour)

## 15. Complete Examples

### 15.1 Simple Configuration

```
# Application configuration
.app_name = "Bovnar Demo";
.version = 1;
.debug = false;
.max_connections = 100;
.timeout_s = 30;
```

### 15.2 Typed Scientific Data

```
# Physical measurements with units
.measurements = [
  { .name = "temperature";
    .value = <float:32,°C> 23.5;
    .precision = <float:32,°C> 0.1; },
  { .name = "pressure";
    .value = <float:64,Pa> 101325;
    .precision = <float:64,Pa> 100; },
  { .name = "humidity";
    .value = <float:32> 0.45;
    .precision = <float:32> 0.01; },
  { .name = "wind_speed";
    .value = <float:32,m/s> 5.2;
    .precision = <float:32,m/s> 0.1; }
];

.calibration = <float:64,no_unit> 1.00042;
.density = <float:64,k~g/m³> 7800;
.accel = <float:64,m/s²> 9.81;
```

### 15.3 Inline Unit Suffix

The unit may be written directly after the value literal instead of — or redundantly alongside — the type annotation:

```
# No type annotation: inline unit supplies both type default and unit
.distance = 1500 m;           # uint:64, no_unit → unit overridden to m
.speed    = 9.81 m/s;        # float:64, unit = m/s
.mass     = 70.5 k~g;        # float:64, unit = k~g

# Type annotation without unit: inline suffix supplies the unit
.dist     = <float:32> 1.5 k~m;

# Annotation and inline unit match: valid (redundant)
.pressure = <float:64,Pa> 101325 Pa;

# Annotation and inline unit differ: error_unit_mismatch
# .bad     = <float:64,m> 1.5 s;    # ERROR
```

## 15.4 Binary Data with Octet Stream

```
# An image file embedded as binary
.image = \x00
        \x01\x10\x00\xff\xD8\xff\xE0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\x01\x00\x00
        \x01\x00\x00\x00\xff\xD9
        \x00;

# A checksum alongside
.checksum = <uint:_16> "abcd";
```

## 15.5 Arrays with Mixed Dimensions

```
# 2D matrix (uniform rows)
.matrix = [1, 2, 3]/[4, 5, 6];

# Uniform nested arrays (all inner arrays same size)
.uniform_nested = [[1,2],[3,4]];

# Multi-dimensional uniform array
.cube = [[1,2],[3,4]],[[5,6],[7,8]];

# Array with typed nulls
.nullable = [<sint:16> 1, <sint:16> , <sint:16> 3];

# Sibling sub-arrays must match in length (homogeneity, §7.4):
.rect_nested = [[1,2],[3,4]];

# These produce error_array_row_size_mismatch:
# .bad1 = [1,2,3]/[4,5];           # /-row sizes differ: 3 vs 2
# .bad2 = [[1,2]/[3,4,5]];         # inner /-array's own rows differ: 2 vs 3
# .bad3 = [[1,2],[3,4,5]];         # ragged sibling sub-arrays differ: 2 vs 3
```



## 15.6 Deeply Nested Struct

```
.api_response = {  
    .status = ok;  
    .code = 200;  
    .data = {  
        .users = [  
            { .id = 1; .name = "Alice"; .roles = ["admin", "user"]; },  
            { .id = 2; .name = "Bob"; .roles = ["user"]; }  
        ];  
        .pagination = {  
            .page = 1;  
            .per_page = 50;  
            .total = 2;  
        };  
    };  
};
```

## 15.7 References and Symbols

```
# Configuration with references  
.config = {  
    .host = "api.example.com";  
    .port = 443;  
    .tls = true;  
};  
  
.endpoint_defaults = {  
    .host = &.config.host;  
    .port = &.config.port;  
    .tls = &.config.tls;  
};  
  
# Using symbols as enum-like values  
.status = ok;  
.mode = readonly;  
.direction = north;
```

## 15.8 Compound Unit Examples

```
# Velocity
.velocity = <float:64,m/s> 9.81;

# Acceleration
.acceleration = <float:64,m/s²> 9.81;

# Force (Newton = kg·m/s²)
.force = <float:64,k~g·m/s²> 9.81;

# Energy (Joule = kg·m²/s²)
.energy = <float:64,k~g·m²/s²> 100;

# Momentum (kg·m/s)
.momentum = <float:64,k~g·m/s> 5.0;

# Pressure (Pa = kg/(m·s²))
.pressure = <float:64,k~g/(m·s²)> 101325;

# Area density (kg/m²)
.area_density = <float:64,k~g/m²> 5.0;

# Electric field (V/m)
.electric_field = <float:64,V/m> 150;

# Magnetic flux density (T = kg/(A·s²))
.mag_flux_density = <float:64,k~g/(A·s²)> 0.5;

# Product form with asterisk
.moment = <float:64,m*s> 1.0;

# Alternative superscript notation for compound units
.force_alt = <float:64,k~g·m·s⁻²> 9.81;
```

## 15.9 Fixed-Point and Decimal Float Examples

```
# — float_fix: fixed-point Q-format —————

# 16-bit Q8: 8 fractional bits, resolution 2^-8 ≈ 0.00390625
# range: [-128.0, +127.99609375]
.adc_reading    = <float_fix:16,q8> 3.14;

# 32-bit Q16: 15 integer bits + sign + 16 fractional bits
.fine_angle     = <float_fix:32,q16> -1.5;

# 64-bit Q0: no fractional bits – pure integer in fixed-point shell
.sample_count   = <float_fix:64,q0> 4096;

# 32-bit Q8 with unit (meters/second at 1/256 resolution)
.velocity_fx    = <float_fix:32,q8,m/s> 9.81;

# null of a fixed-point type
.missing_fx     = <float_fix:16,q8> ;

# inline unit suffix also works with float_fix
.temperature    = <float_fix:32,q8> 23.5 °C;

# — float_dec: IEEE 754-2008 decimal floating-point —————

# 32-bit decimal float (7 significant decimal digits)
.price          = <float_dec:32> 12.99;

# 64-bit decimal float with unit (16 significant decimal digits)
.pressure_dec   = <float_dec:64,Pa> 101325.0;

# 128-bit decimal float (34 significant decimal digits)
.pi_dec        = <float_dec:128> 3.14159265358979323846264338327950288;

# 256-bit decimal float (70 significant decimal digits)
.big_constant   = <float_dec:256>
1.4142135623730950488016887242096980785696718753769480731766797;

# Special values are accepted
.nan_dec        = <float_dec:64> nan;
.inf_dec        = <float_dec:32> inf;

# null of a decimal float type
.missing_dec    = <float_dec:64> ;

# — Contrast with binary float —————

# Binary IEEE float (existing)
.val_bin        = <float:64> 3.14;

# Decimal float – same text representation, different wire encoding
.val_dec        = <float_dec:64> 3.14;

# Fixed-point – same text representation, Q-format wire encoding
.val_fix        = <float_fix:32,q16> 3.14;
```

## 15.10 Error Examples

```
# These will produce parse errors:

# Empty identifier
. = 42;                                # error_empty_identifier

# Type violation (type annotation on the identifier, not the value)
.x<utf8> = 42;                          # error (annotation must be after '=')

# Correct: .x = <utf8> "text";

# Type violation – value doesn't match type
.x = <utf8> 42;                          # error_type_value_mismatch

# Value out of range
.y = <uint:8> 300;                       # error_value_out_of_range

# Negative unsigned
.z = <uint:8> -1;                        # error_value_out_of_range

# Unmatched struct close – a '}' at the top level (struct nesting 0)
.stray_close = 1;}                      # error_illegal_struct_close

# Unknown escape
.string = "\x";                          # error_illegal_escape_sequence

# error_array_row_size_mismatch – /-dimension rows of one array, and (since 1.0)
# ragged sibling sub-arrays, must match in length:
# .bad1 = [1,2,3]/[4,5];                  # /-row sizes differ: 3 vs 2
# .bad2 = [[1,2]/[3,4,5]];                # inner /-array's own rows differ: 2 vs 3
# .bad3 = [[1,2],[3,4,5]];                # ragged sibling sub-arrays differ: 2 vs 3
# Uniform /-rows and rectangular sibling sub-arrays are valid:
.ok1 = [1,2,3]/[4,5,6];                  # valid – both dimension rows have 3 elements
.ok2 = [[1,2],[3,4]];                    # valid – rectangular sub-arrays

# Comma outside array
.comma_outside = 42;;                   # error_unexpected_input_byte

# Non-decimal base – bare token is parsed as a symbol, causing type mismatch
.hex = <uint:16> ff;                    # error: symbol value for numeric type

# Empty component in compound unit
.x = <float:64,m//s> 1.0;                # error_unit_illegal

# Too many components (> 8)
.y = <float:64,m*s*g*A*K*mol*cd*b*V> 1.0; # error_unit_illegal (9 components)

# float_fix: Q >= effective width
.bad_q = <float_fix:16,q16> 1.0;          # Q=16 >= width=16 → error_illegal_value_type

# float_fix: invalid width
.bad_fw = <float_fix:8,q4> 1.0;           # width 8 not in {0,16,32,64,128,256}

# float_fix: base param forbidden
.bad_fb = <float_fix:32,q8,_10> 1.0;      # error_illegal_value_type

# float_dec: invalid width
.bad_dw = <float_dec:24> 1.0;             # width 24 not in {0,16,32,64,128,256}

# float_dec: base param forbidden
.bad_db = <float_dec:64,_10> 1.0;         # error_illegal_value_type

# q~param on non-float_fix type
.bad_qu = <float:64,q8> 1.0;              # q~param only valid for float_fix
```



## 16. Reference API

---

### 16.1 Core Types

```

typedef enum bvnr_event_e {
    ev_stream_start,
    ev_assignment_start,
    ev_octet_stream_start,
    ev_octet_stream_end,
    ev_struct_start,
    ev_struct_end,
    ev_array_row_start,
    ev_array_row_end,
    ev_array_dim_start,
    ev_data,
    ev_type_annotation_start,
    ev_type_annotation_end,
    ev_type_annotation_type_family,
    ev_type_annotation_type_family_parameter,
    ev_stream_end
} bvnr_event_t;

typedef enum value_type_family_e {
    vt_plain,
    vt_utf8,
    vt_sint,
    vt_uint,
    vt_float,
    vt_float_fix, /* fixed-point binary, Q-format; Q stored in value_type_spec_t.base */
    vt_float_dec, /* IEEE 754-2008 decimal floating-point
*/
    vt_bool, /* boolean (true/false/on/off); see §4.4 and §6.1
*/
    vt_datetime, /* spec 1.1 – timestamp: signed epoch-seconds; see §5
*/
    vt_illegal
} value_type_family_t;

typedef struct value_type_spec_s {
    value_type_family_t family;
    uint32_t width; /* bit-width; 0 = default (64) */
    uint32_t base; /* for uint/sint/float: numeral base; 0 = default (10) */
    /* for float_fix: Q (fractional bits) */
    /* for float_dec: unused (always 0) */
} value_type_spec_t;

#define BVNR_MAX_UNIT_COMPONENTS 8

typedef struct value_unit_prefix_s {
    prefix_system_t system;
    union {
        si_prefix_id_t si;
        iec_prefix_id_t iec;
    } id;
} value_unit_prefix_t;

typedef struct value_unit_component_s {
    value_base_unit_t base;
    unit_exponent_t exponent;
    value_unit_prefix_t prefix;
} value_unit_component_t;

typedef struct value_unit_s {
    uint32_t num_components;
    value_unit_component_t components[BVNR_MAX_UNIT_COMPONENTS];
} value_unit_t;

typedef enum token_type_e {
    token_is_identifier,
    token_is_string,

```

```

    token_is_number,
    token_is_symbol,
    token_is_reference,
    token_is_array_number,
    token_is_array_string,
    token_is_type,
    token_is_octet_stream,
    token_is_null_value,
    token_is_structure,
    token_is_unit,
    token_is_type_width,
    token_is_type_base,
    token_is_type_q,
    token_is_bool,
    token_is_unknown
} token_type_t;

typedef struct bvnr_data_s {
    token_type_t      type;
    value_type_spec_t value_type;
    value_unit_t      value_unit;
    const void*       data;
    uint32_t          length;
    const void*       frac_data; /* spec 1.1 – ISO datetime sub-second digits, else
NULL */
    uint32_t          frac_length; /* spec 1.1 – length of frac_data, else 0 */
} bvnr_data_t;

```

## 16.2 Type Construction Macros

```

/* Type-spec convenience constructors (from bovnar.h) */
#define BVN_TYPE_PLAIN          ((value_type_spec_t){ .family = vt_plain, .width =
0, .base = 0 })
#define BVN_TYPE_UTF8          ((value_type_spec_t){ .family = vt_utf8, .width =
0, .base = 0 })
#define BVN_TYPE_BOOL          ((value_type_spec_t){ .family = vt_bool, .width =
0, .base = 0 })
#define BVN_TYPE_UINT(w)       ((value_type_spec_t){ .family = vt_uint, .width =
(w) })
#define BVN_TYPE_SINT(w)       ((value_type_spec_t){ .family = vt_sint, .width =
(w) })
#define BVN_TYPE_FLOAT(w)      ((value_type_spec_t){ .family = vt_float, .width =
(w) })
/* float_fix: .base is repurposed to store Q (fractional bits). */
#define BVN_TYPE_FLOAT_FIX(w,q) ((value_type_spec_t){ .family = vt_float_fix, .width =
(w), .base = (q) })
/* float_dec: base field is unused (always 0). */
#define BVN_TYPE_FLOAT_DEC(w)  ((value_type_spec_t){ .family = vt_float_dec, .width =
(w) })
/* With explicit numeral base (uint/sint only): */
#define BVN_TYPE_UINT_BASE(w,b) ((value_type_spec_t){ .family = vt_uint, .width =
(w), .base = (b) })
#define BVN_TYPE_SINT_BASE(w,b) ((value_type_spec_t){ .family = vt_sint, .width =
(w), .base = (b) })

```



## 16.3 Unit Macros

```

#define BVN_UNIT_NO_PREFIX(b) \
    ((value_unit_t){ \
        .num_components = 1, \
        .components = {{ \
            .base = (b), .exponent = exp_linear, \
            .prefix.system = prefix_si, .prefix.id.si = si_none \
        }} \
    })

#define BVN_UNIT_SI(b, p) \
    ((value_unit_t){ \
        .num_components = 1, \
        .components = {{ \
            .base = (b), .exponent = exp_linear, \
            .prefix.system = prefix_si, .prefix.id.si = (p) \
        }} \
    })

#define BVN_UNIT_IEC(b, p) \
    ((value_unit_t){ \
        .num_components = 1, \
        .components = {{ \
            .base = (b), .exponent = exp_linear, \
            .prefix.system = prefix_iec, .prefix.id.iec = (p) \
        }} \
    })

#define BVN_UNIT_SI_EXP(b, p, e) \
    ((value_unit_t){ \
        .num_components = 1, \
        .components = {{ \
            .base = (b), .exponent = (e), \
            .prefix.system = prefix_si, .prefix.id.si = (p) \
        }} \
    })

#define BVN_UNIT_NONE \
    ((value_unit_t){ .num_components = 0 })

/* Compound-unit helper: two SI-prefixed components */
#define BVN_UNIT_COMPOUND2(b1, p1, e1, b2, p2, e2) \
    ((value_unit_t){ \
        .num_components = 2, \
        .components = { \
            { .base = (b1), .exponent = (e1), \
              .prefix.system = prefix_si, .prefix.id.si = (p1) }, \
            { .base = (b2), .exponent = (e2), \
              .prefix.system = prefix_si, .prefix.id.si = (p2) } \
        } \
    })

```

## 16.4 Reader Setup

```
typedef struct bvnr_read_flags_s {
    uint16_t max_identifier_length; // default 255
    uint16_t max_string_length;    // default 65535
    uint16_t max_number_length;    // default 65535
    uint16_t max_symbol_length;    // default 255
    uint16_t max_reference_length; // default 65535
    uint64_t max_array_items;      // 0 → 2 147 483 647 internal default
    uint64_t max_text_bytes;       // 0 → 2 147 483 647 internal default
    uint64_t max_file_size;        // 0 → unlimited / endless (default); 16 777 216
    recommended for a cap
    uint8_t max_struct_nesting;    // 0 → 64 internal default; hard cap 255
    uint8_t max_array_nesting;     // 0 → 64 internal default; hard cap 255
    void* userdata;
    bool (*on_unverified)(void*, bvnr_event_t, bvnr_data_t*);
    bool (*on_verified)(void*, bvnr_event_t, bvnr_data_t*);
    bool continue_on_error;
    bvnr_on_error_fn on_error;
} bvnr_read_flags_t;
```

## 16.5 Source/Sink Creation

```
void bvnr_source_from_fd(bvnr_source_t* s, int fd);
void bvnr_source_from_mem(bvnr_source_t* s, const void* buf, uint64_t len);
void bvnr_sink_to_fd(bvnr_sink_t* s, int fd);
void bvnr_sink_to_mem(bvnr_sink_t* s, void* buf, uint64_t cap);
uint64_t bvnr_sink_bytes_written(const bvnr_sink_t* s);
```

`bvnr_sink_bytes_written` queries the total bytes written to a memory sink created with `bvnr_sink_to_mem`. For the writer-encapsulated variant, see `bvnr_writer_bytes_written` in §16.7.

## 16.6 Reading

```
bool bvnr_open_read_source(bvnr_reader_t* r, const bvnr_source_t* src,
                           const bvnr_sink_t* src_mirror,
                           bvnr_read_flags_t* options);

bool bvnr_open_read_mem(bvnr_reader_t* r, const void* buf, uint64_t len,
                        void* mirror_buf, uint64_t mirror_cap,
                        bvnr_read_flags_t* options);

bool bvnr_read(bvnr_reader_t* r);
```

## 16.7 Error Queries

```
error_code_t bvnr_reader_get_error(const bvnr_reader_t* r);
uint64_t bvnr_reader_get_error_line (const bvnr_reader_t* r);
uint64_t bvnr_reader_get_error_column(const bvnr_reader_t* r);
uint32_t bvnr_reader_get_error_byte (const bvnr_reader_t* r);
uint64_t bvnr_reader_get_error_offset(const bvnr_reader_t* r);
uint64_t bvnr_reader_get_recovery_count(const bvnr_reader_t* r);
const char* bvnr_error_to_string(error_code_t code);
```

## 16.8 Utility Functions

```

bool bvn_validate_identifier(const char* id);
bool bvn_validate_symbol(const char* surr);
bool bvn_validate_reference(const char* link);
bool bvn_validate_number(const char* s);
bool bvn_validate_string(const uint8_t* data, size_t len);

bool bvn_is_special_number_string(const char* s);
bool bvn_validate_digits_for_base(const char* s, uint32_t base);
bool bvn_validate_number_in_base(const char* s, uint32_t base);

bool bvn_validate_uint_range(const char* s, uint32_t w, uint32_t base);
bool bvn_validate_sint_range(const char* s, uint32_t w, uint32_t base);

uint32_t bvn_char_to_digit(uint32_t c, uint32_t base);
uint32_t bvn_min_digits_for_type(value_type_spec_t vt);

int32_t bvn_format_uint64(char* buf, size_t bufsize,
                        uint64_t value, uint32_t base, uint32_t min_digits);
int32_t bvn_format_int64(char* buf, size_t bufsize,
                        int64_t value, uint32_t base, uint32_t min_digits);
int32_t bvn_format_double(char* buf, size_t bufsize,
                        double value, value_type_spec_t vt);

bool bvn_parse_int64(const char* s, value_type_spec_t vt, int64_t* out);
bool bvn_parse_uint64(const char* s, value_type_spec_t vt, uint64_t* out);
bool bvn_parse_double(const char* s, value_type_spec_t vt, double* out);
bool bvn_parse_double_in_base(const char* s, uint32_t base, double* out);
bool bvn_looks_like_double(const char* s);

/* Parse a NUL-terminated unit string into a value_unit_t.
   Sets *ok to false on error. */
value_unit_t bvn_parse_unit(const uint8_t* unit, bool* ok);

/* Length-bounded variant; does not require a NUL terminator. */
value_unit_t bvn_parse_unit_n(const uint8_t* unit, uint32_t len, bool* ok);

/* Serialize a value_unit_t (possibly compound) back to a string.
   Numerator components are joined by ".", followed by "/" and
   denominator components joined by "...". Returns bytes written,
   or -1 on buffer overflow. */
int32_t bvn_unit_to_string(value_unit_t u, char* buf, size_t bufsize);

/* Extended variant accepting bvn_unit_flags_t:
   BVN_UNIT_FLAGS_NONE (0) - Unicode superscript exponents, no reduction
   BVN_UNIT_ASCII_EXP (1 << 1) - use ^N caret notation for exponents
   BVN_UNIT_REDUCE (1 << 0) - reduce compound unit before serialising
   Flags may be OR-combined. Returns bytes written, or -1 on overflow. */
int32_t bvn_unit_to_string_ex(value_unit_t u, char* buf, size_t bufsize,
                            bvn_unit_flags_t flags);

/* Returns true if every component in u has a valid exponent (not
   exp_invalid), a known base unit, and a prefix legal for that base
   unit per bvn_prefix_unit_valid. Both serialisation functions call
   this predicate internally before writing. */
bool bvn_unit_valid(value_unit_t u);

/* Structural equality of two units: same num_components and the same set
   of components (matching base, exponent, and prefix). The comparison is
   ORDER-INSENSITIVE - unit multiplication is commutative, so components are
   matched as multisets and "N·m" equals "m·N". This is the comparison the
   validator uses to match an inline unit suffix against a type-annotation
   unit (error_unit_mismatch on disagreement). For dimensional equivalence
   - units that measure
   the same physical quantity but differ in spelling or factoring (e.g.
   W vs VA, or the two no_unit forms) - use bvn_units_compatible from
   bovnar_si_units.h (§11.8) instead. */

```

```

bool bvn_unit_equal(value_unit_t a, value_unit_t b);

/* Compute the combined prefix factor across all components (ignoring
   base-unit conversion factors). Each component's prefix factor is
   raised to |exponent| and multiplied together; denominator components
   are inverted. */
double bvn_unit_prefix_factor(value_unit_t u);

/* Compute the combined prefix exponent (sum of prefix_base_exponent ×
   |unit_exponent| across all components, negated for denominator
   components). */
int32_t bvn_unit_prefix_exponent(value_unit_t u);

const uint8_t* bvn_get_escape_repl_table(void);

```

## 16.9 Typed Write Helpers

Convenience functions that emit a type annotation + value in one call. All functions return `false` on serialisation error.

```

/* — Plain scalar writers — */
bool bvnr_write_string(bvnr_writer_t* w, const char* key, const char* value);
bool bvnr_write_plain (bvnr_writer_t* w, const char* key, const char* value);
bool bvnr_write_null  (bvnr_writer_t* w, const char* key);
bool bvnr_write_bool  (bvnr_writer_t* w, const char* key, bool value);

/* — Integer writers — */
bool bvnr_write_uint(bvnr_writer_t* w, const char* key,
                    uint32_t width, uint64_t value);
bool bvnr_write_sint(bvnr_writer_t* w, const char* key,
                    uint32_t width, int64_t value);

/* — Binary float writers — */
bool bvnr_write_float(bvnr_writer_t* w, const char* key,
                    uint32_t width, double value);

/* — Fixed-point writers (float_fix) — */
/* q = number of fractional bits (Q parameter). */
bool bvnr_write_float_fix(bvnr_writer_t* w, const char* key,
                        uint32_t width, uint32_t q, double value);

/* — Decimal float writers (float_dec) — */
bool bvnr_write_float_dec(bvnr_writer_t* w, const char* key,
                        uint32_t width, double value);

/* — Writers with explicit unit — */
bool bvnr_write_uint_unit (bvnr_writer_t* w, const char* key,
                        uint32_t width, uint64_t value, value_unit_t unit);
bool bvnr_write_sint_unit (bvnr_writer_t* w, const char* key,
                        uint32_t width, int64_t value, value_unit_t unit);
bool bvnr_write_float_unit (bvnr_writer_t* w, const char* key,
                        uint32_t width, double value, value_unit_t unit);
bool bvnr_write_float_fix_unit(bvnr_writer_t* w, const char* key,
                        uint32_t width, uint32_t q,
                        double value, value_unit_t unit);
bool bvnr_write_float_dec_unit(bvnr_writer_t* w, const char* key,
                        uint32_t width, double value, value_unit_t unit);

/* — Wide-precision float writers (bvn_float_t) — */
/* vt.width must match f->prec or be 0 (auto). */
bool bvnr_write_bvnf (bvnr_writer_t* w, const char* key,
                    const bvn_float_t* f, uint32_t width);
bool bvnr_write_bvnf_unit(bvnr_writer_t* w, const char* key,
                    const bvn_float_t* f, uint32_t width, value_unit_t unit);

/* — Struct helpers — */
bool bvnr_write_struct_start(bvnr_writer_t* w, const char* key);
bool bvnr_write_struct_end (bvnr_writer_t* w);

```

## 16.10 Error Codes

```

typedef enum error_code_e {
    error_none = 0,
    error_unknown_token_type = 1,
    error_array_row_size_mismatch = 2,
    error_identifier_too_long = 3,
    error_empty_identifier = 4,
    error_struct_nesting_too_high = 5,
    error_array_nesting_too_high = 6,
    error_illegal_struct_close = 7,
    error_string_too_long = 8,
    error_illegal_escape_sequence = 9,
    error_number_too_long = 10,
    error_symbol_too_long = 11,
    error_reference_too_long = 12,
    error_read_complete_chunk_failed = 13,
    error_octet_stream_out_of_sync = 14,
    error_unexpected_input_byte = 15,
    error_text_data_too_long = 16,
    error_reading_from_source_fd = 17,
    error_got_incomplete_bvnr_stream = 18,
    error_invalid_utf8_byte = 19,
    error_invalid_byte_order_mark = 20,
    error_type_too_long = 21,
    error_unit_too_long = 22,
    error_expected_string_in_array = 23, /* reserved; never set by the library */
    error_expected_number_in_array = 24, /* reserved; never set by the library */
    error_illegal_value_type = 25,
    error_scanner_callback_failed = 26,
    error_file_too_long = 27,
    error_invalid_argument = 28,
    error_too_many_array_items = 29,
    error_writing_to_sink = 30,
    error_sink_buffer_exhausted = 31,
    error_unit_illegal = 32,
    error_base_requires_string_literal = 33,
    error_type_value_mismatch = 34,
    error_value_out_of_range = 35,
    error_digit_not_in_base = 36,
    error_recovered = 37, /* reserved; never set by the library */
    error_unit_mismatch = 38,
    /* Array element homogeneity (spec 1.0): every non-null element of an array
     * must share the same kind and physical dimension; sibling sub-arrays must
     * match in length and element shape (recursively); sibling structs must
     * share the same keys with recursively-matching fields. */
    error_array_element_type_mismatch = 39,
    error_struct_shape_mismatch = 40,
    /* A struct (or the top-level document) repeats a key. Keys must be unique
     * within one scope so lookup, references and iteration always agree. */
    error_duplicate_struct_key = 41,
    /* spec 1.1 – a leading "#!bovnr ..." directive is present but malformed. */
    error_invalid_spec_version = 42,
    /* spec 1.1 – the declared version exceeds what the reader supports and
     * strict_version was set. */
    error_unsupported_spec_version = 43,
    /* spec 1.1 – a \u{...} escape names a non-scalar value (a surrogate, or a
     * code point above U+10FFFF). */
    error_invalid_codepoint = 44,
    /* spec 1.1 – an ISO-8601 datetime literal is malformed or has an
     * out-of-range field (bad width, separator, month/day/time component). */
    error_invalid_datetime_literal = 45,
    /* spec 1.1 – an ISO-8601 literal was given for an atomic GNSS epoch
     * (gps/galileo/glonass/beidou), which has no round-trippable inverse. */
    error_datetime_literal_unsupported_epoch = 46,
} error_code_t;

```



## 17. Versioning & Stability

Bovnar follows semantic versioning of the **format**, independent of any implementation's version.

**Implementation version.** The reference implementation exposes its own version, which tracks the format version but may advance independently for implementation-only fixes. The C header defines `BVNR_VERSION_MAJOR`, `BVNR_VERSION_MINOR`, `BVNR_VERSION_PATCH`, the comparable integer `BVNR_VERSION` (`major*10000 + minor*100 + patch`), so `#if BVNR_VERSION >= 10100` tests " $\geq 1.1.0$ ", and `BVNR_VERSION_STRING` (`"1.1.0"`). The Python package mirrors this as `bovnar.__version__`. Separately, `BVNR_SPEC_VERSION_MAJOR` / `BVNR_SPEC_VERSION_MINOR` name the highest **spec** version the build understands (`1.1`); `bovnr_version()`, `bovnr_version_string()` and `bovnr_spec_version()` expose both at runtime.

**Declaring a document's version.** Since 1.1 a document may state which spec version it targets with a leading `#!/bovnar <major>.<minor>` directive (§3.4). Because the directive is lexically a comment, this is fully backward compatible: a 1.0 reader ignores it. A 1.1+ reader records it (`bovnr_reader_get_declared_version`) and, in `strict_version` mode, rejects a version it does not support.

**What 1.0 freezes.** The grammar is stable. A `.bovnr` document that is valid under spec 1.0 will remain valid, and will decode to the same values, under every 1.x revision. This covers the lexical structure, the type families and their annotations, arrays (including the homogeneity rules of §7.4), structs, octet streams, references, and the error-code values in §16.10. Conforming archives may rely on this for long-term storage.

**What may still grow in 1.x (additive only).** The following may be extended without breaking existing documents, and such extensions ship as minor (1.x) revisions:

- the **unit and currency tables** — new physical units, prefixes, and ISO 4217 / crypto currency codes may be added (a document never depends on a code being absent);
- **new error codes** appended after the current maximum (existing numeric values never change) — 1.1 appends `error_invalid_spec_version` (42), `error_unsupported_spec_version` (43), `error_invalid_codepoint` (44), `error_invalid_datetime_literal` (45), and `error_datetime_literal_unsupported_epoch` (46);
- the **optional version directive** (§3.4), added in 1.1: it is an ordinary comment to any 1.0 reader, so adding one never invalidates a document;
- new optional reader/writer flags and limits whose defaults preserve current behaviour.

A reader from an older 1.x point release may not recognise a unit or currency added in a newer one; that is the expected direction of forward compatibility and is not a break of the 1.0 promise.

**What requires a 2.0.** Any change that could render a valid 1.x document invalid, change how it decodes, renumber an error code, or alter the grammar is a breaking change and is reserved for a major (2.0) revision. The changes that motivated the 1.0 freeze — the **mandatory \$ currency sigil** (§10.4 of the unit-system reference), **array element homogeneity** (§7.4), and **float\_fix value-range validation** (§6.2, rejecting a value the declared Q-format cannot represent) — were exactly such breaks, so they were made before 1.0 and cannot be reconsidered within 1.x.

## Appendix A: Event Sequence Reference

### A.1 Simple Assignment (Untyped)

Input: `.foo = 42;`

```
ev_stream_start
ev_assignment_start      data="foo"
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                  data="42"
ev_stream_end
```

Every stream is bracketed by `ev_stream_start` ... `ev_stream_end`; the reader emits `ev_stream_end` once after the final assignment (it is omitted from the remaining appendix examples for brevity).

### A.2 Typed Assignment

Input: `.bar = <float:32,m/s> 9.81;`

```
ev_assignment_start      data="bar"
ev_type_annotation_start data="float:32,m/s"
ev_type_annotation_type_family "float"
ev_type_annotation_type_family_parameter (width:32)
ev_type_annotation_type_family_parameter (unit:m/s)
ev_type_annotation_end
ev_data                  data="9.81"
```

### A.3 Compound Unit Assignment

Input: `.force = <float:64,k~g·m/s²> 9.81;`

```

ev_assignment_start          data="force"
ev_type_annotation_start     data="float:64,k~g·m/s²"
ev_type_annotation_type_family "float"
ev_type_annotation_type_family_parameter (width:64)
ev_type_annotation_type_family_parameter (unit:k~g·m/s²)
  → value_unit = {
    num_components = 3,
    components = [
      { base=bu_gram,   exponent=exp_linear,   prefix={prefix_si, si_kilo} },
      { base=bu_meter,  exponent=exp_linear,   prefix={prefix_si, si_none} },
      { base=bu_second, exponent=exp_neg_square, prefix={prefix_si, si_none} }
    ]
  }
ev_type_annotation_end
ev_data                      data="9.81"

```

## A.4 Array

Input: `.arr = [1, 2]/[3, 4];`

```

ev_assignment_start          data="arr"
ev_array_row_start
ev_type_annotation_start     (synthesised for 1)
ev_type_annotation_type_family "uint"
...params...
ev_type_annotation_end
ev_data                      data="1"
ev_type_annotation_start     (synthesised for 2)
...params...
ev_type_annotation_end
ev_data                      data="2"
ev_array_row_end
ev_array_dim_start
ev_array_row_start
ev_type_annotation_start     (synthesised for 3)
...params...
ev_type_annotation_end
ev_data                      data="3"
... (4) ...
ev_array_row_end

```

## A.5 Struct

Input: `.s = {.x = 1; .y = 2;;};`

```

ev_assignment_start          data="s"
ev_struct_start
ev_assignment_start          data="x"
...ev_data for 1...
ev_assignment_start          data="y"
...ev_data for 2...
ev_struct_end

```

## A.6 Octet Stream

Input: `.bin = \x00\x01\x03\x00abc\x00;`

```

ev_assignment_start      data="bin"
ev_octet_stream_start
ev_data (octet_stream)    data="abc", length=3
ev_octet_stream_end

```

## Appendix B: Implementation Notes

### B.1 Keyword State Machine

Type family keywords are recognised through a dedicated state machine in the lexer. The lexer fires `ACT_tf_float_done` after the shared `f→l→o→a→t` path, storing `"float"` in `type_data` and transitioning to `type_body_outro`. If the next bytes are `_fix` or `_dec`, they are accumulated via `copy_type_byte`, so the final string is `"float_fix"` or `"float_dec"`. `bvn_parse_type_annotation` then dispatches on the full accumulated string.

```

u → i → n → t      → keyword "uint"
u → t → f → 8      → keyword "utf8"
s → i → n → t      → keyword "sint"
f → l → o → a → t  → ACT_tf_float_done → type_body_outro
                                     → accumulate "_fix" → "float_fix"
                                     → accumulate "_dec" → "float_dec"
                                     → (nothing)         → "float"

```

### B.2 Special Number Keywords

The special floats are bare reserved keywords — `nan`, `inf`, and `ninf` (negative infinity) — with no sigil. The lexer reads them as ordinary symbols; the validator then reclassifies a symbol whose text is exactly one of these three spellings into a numeric special value (`token_is_number`), the same way it reclassifies `null` / `true` / `false` / `on` / `off`:

```

symbol "nan"  → reclassify → special number "nan"  (3 bytes)
symbol "inf"  → reclassify → special number "inf"  (3 bytes)
symbol "ninf" → reclassify → special number "ninf" (4 bytes)

```

Any other bare word (e.g. `infinity`, `nans`) stays an ordinary symbol. The stored token text is the keyword itself: `nan`, `inf`, `ninf`. A special-number keyword takes no inline unit suffix; a unit is supplied through the type annotation (`<float:64,m/s> inf`).

## B.3 Default Width, Base, and Q

```
static inline uint32_t bvn_effective_width(value_type_spec_t s) {
    return s.width ? s.width : 64u;
}

/*
 * For float_fix and float_dec the .base field has a different meaning
 * (Q for float_fix, unused for float_dec); always report base 10 for those.
 */
static inline uint32_t bvn_effective_base(value_type_spec_t s) {
    if (s.family == vt_float_fix || s.family == vt_float_dec)
        return 10u;
    return s.base ? s.base : 10u;
}

/*
 * Returns the Q (fractional bits) for float_fix, 0 for all other families.
 * Q is stored in the .base field of value_type_spec_t.
 */
static inline uint32_t bvn_effective_q(value_type_spec_t s) {
    return (s.family == vt_float_fix) ? s.base : 0u;
}
```

## B.4 Type Equality

```
static inline bool bvn_type_spec_eq(value_type_spec_t a, value_type_spec_t b) {
    return a.family == b.family && a.width == b.width && a.base == b.base;
}
```

For `float_fix`, `.base` holds Q, so two `float_fix` specs are equal only if they share the same width **and** the same Q.

## B.5 Numeric Type Check

```
static inline bool bvn_type_is_numeric(value_type_spec_t s) {
    return s.family == vt_sint || s.family == vt_uint ||
           s.family == vt_float ||
           s.family == vt_float_fix || s.family == vt_float_dec;
}
```

## B.6 Unit Component Access

When iterating compound units, always check `num_components`:

```
for (uint32_t i = 0; i < u.num_components && i < BVNR_MAX_UNIT_COMPONENTS; i++) {
    value_unit_component_t* c = &u.components[i];
    /* c->base, c->exponent, c->prefix.system, c->prefix.id */
}
```

## B.7 Fixed-point and Decimal Float Wire Representations

`float_fix` and `float_dec` are both serialised as ordinary decimal number literals in the Bovnar text layer. The type annotation is the sole indicator of wire encoding. At the C API level, the conversion path is:

```
float_fix (width ≤ 64):
    text literal → bvn_float_t → bvn_float_to_fixNN(f, frac_bits) → signed integer return
    value

float_fix (width = 128, 256):
    text literal → bvn_float_t → bvn_float_to_fixNN(f, frac_bits, out) → wire bits in
    out[]

float_dec:
    text literal → bvn_float_t → bvn_float_to_decNN(f, out) → wire bits in *out / out[]
```

The `bvn_float_t` intermediate representation is MPFR-layout-compatible (see `bvn_float.h`) and provides exact round-trip fidelity up to the declared precision.

## Appendix C: Limits Summary

Constant	Value	Description
reader default struct nesting	64	Default applied by the reader when <code>max_struct_nesting</code> is 0; hard cap is 255
reader default array nesting	64	Default applied by the reader when <code>max_array_nesting</code> is 0; hard cap is 255
writer default struct nesting	64	Default applied by the writer when <code>max_struct_nesting</code> is 0; hard cap is 255
writer default array nesting	64	Default applied by the writer when <code>max_array_nesting</code> is 0; hard cap is 255
reader default max_array_items	2 147 483 647	Default applied by the reader when <code>max_array_items</code> is 0
reader default max_text_bytes	2 147 483 647	Default applied by the reader when <code>max_text_bytes</code> is 0
reader default max_file_size	0 (unlimited / endless)	A <code>max_file_size</code> of 0 imposes no byte-count cap; set to 16 777 216 (16 MiB) in production
recommended file size cap	16 777 216	Suggested explicit value for <code>max_file_size</code> (16 MiB)
<code>BVNR_MAX_UNIT_COMPONENTS</code>	8	Maximum number of unit components in a compound unit
<code>BVN_MAX_INT_WIDTH</code>	32768	Maximum bit-width for <code>uint</code> and <code>sint</code> types. The validator and writer reject any declared width exceeding this value with <code>error_illegal_value_type</code> .

---

End of Bovnar Specification (v1.1)