BOVNAR · SPEC 1.2

Bovnar — Rust Bindings

Bovnar (BVNR) spec 1.2 documentation · Rust Bindings · 2026-11-10

> **Spec version:** 1.2 **Status:** Reference — the Rust crate as implemented **Scope:** Installation, the high-level and streaming APIs, the DOM, `Quantity` and the exact numbers, JSON conversion, and the thread rules.

Rust bindings for the **Bovnar (BVNR)** typed serialisation library (library 2.1.0, format spec 1.2).

The crate is a thin wrapper around the C library: `build.rs` compiles the C sources into the crate, so there is no system library to install and nothing is looked up at run time. One module is not a wrapper: `bovnar::codec`, the incremental framing codec for live streams, is pure Rust (§4.8). The API stays close to the C library's own shapes: `parse` / `to_string` over a dynamic `Value`, `Reader` callbacks, `Writer` calls and a random-access DOM. There is no serde integration, no builder API for values and no typed deserialisation; the DOM has the C library's node builder (§12.5). Where Rust's types can say more at no cost, they do: the C enums are Rust enums, and an error carries the library's error kind.

Table of Contents

1. Requirements
2. Installation
3. Building the C library
   - 3.1 WebAssembly (`wasm32-wasip2`)
   - 3.2 Unit-profile features
   - 3.3 How the published crate is made
4. Quick-start
   - 4.1 High-level API (`parse` / `to_string`)
   - 4.2 SAX-style streaming reader
   - 4.3 Collecting events
   - 4.4 DOM (random-access) API
   - 4.5 Low-level writer
   - 4.6 Streaming / framing (`bovnar::stream`)
   - 4.7 Files and I/O streams (`std::io::Read` / `Write`)
   - 4.8 Live streams: the framing codec (`bovnar::codec`)
   - 4.9 Live documents through C (`bovnar::live`)

    - 4.10 Canonical form
5. Unit helpers
    - 5.1 The unit-profile notations (spec 1.2)
    - 5.2 Extended unit functions
    - 5.3 `UnitFlags`
    - 5.4 `Prefix`
    - 5.5 Inline unit suffix
    - 5.6 `UnitPolicy` — validation and conversion without a callback
    - 5.7 Building a `Unit`
    - 5.8 Validators and literal helpers
6. `Quantity`
    - 6.1 Construction
    - 6.2 Fields and methods
    - 6.3 Exact numbers (`Decimal`, `BigRational`, `BvnFloat`)
    - 6.4 `to_string()` integration
    - 6.5 Civil time and calendars
7. `write_array` helper
8. JSON conversion
9. Currency helpers
10. `Reader` reference
    - 10.1 Construction
    - 10.2 `read_mem`
    - 10.3 `ReadOptions`
    - 10.4 `EventData` and `EventPayload`
    - 10.5 `iter_mem`
    - 10.6 Error-state accessors
11. `Writer` reference
    - 11.1 Construction
    - 11.2 Output retrieval
    - 11.3 Scalar write methods
    - 11.4 Extended integer writers
    - 11.5 Struct helpers
    - 11.6 Version directive
    - 11.7 Array helpers
    - 11.8 Low-level `emit`
    - 11.9 Observing the output: `on_event`
    - 11.10 The C per-type helpers
12. DOM API
    - 12.1 `DomDoc::parse`

    - 12.2 `DomDoc`
    - 12.3 `DomNode`
    - 12.4 `DomType`
    - 12.5 Building a tree (`NewNode`)
13. Error handling
14. Threads
15. Running the test suite
16. Crate layout
17. Enums and `BaseUnit`
18. Spec 1.1 additions
19. License

- See also

---

1. Requirements

Requirement	Notes
Rust ≥ 1.88	The crate's `rust-version`; CI builds and tests on it
A C compiler	`build.rs` compiles the C library through the `cc` crate
Python 3 on `PATH` (only when building from a repository checkout)	In the repository, the build runs the upstream table generator scripts, exactly as CMake does. Set the `PYTHON` environment variable to name a different interpreter. The crate from crates.io carries the generated tables and needs no Python (§3.3)
`num-bigint`, `num-rational`, `num-integer`, `num-traits`	Pulled in by Cargo; they carry the exact numbers (arbitrary-width integers, rational numbers and the coefficient of a `Decimal`)
wasi-sdk	**Only** for the `wasm32-wasip2` target — see §3

The only optional features are the unit-profile notations, all on by default (§3.2). There are no optional dependencies and no integration with third-party unit or array libraries; see §16.

---

2. Installation

The crate carries the version of the C library it wraps (**2.1.0**: `version` in `Cargo.toml` is the `project(bovnar VERSION x.y.z)` of the top-level `CMakeLists.txt`, and CI fails when the two differ). It is published to crates.io with each release:

```toml
[dependencies]
bovnar = "2.1.0"
```

A repository checkout works too, as a path dependency or straight from Git, where Cargo finds the crate inside the repository. Built that way it needs Python 3 at build time (§1):

```toml
[dependencies]
bovnar = { path = "../bovnar/rust" }
# or
bovnar = { git = "https://github.com/sothis/bovnar" }
```

To work on the crate itself:

```
cd rust
cargo build
cargo test
```

---

3. Building the C library

The crate has no shared library to find: `build.rs` compiles the C library into the crate. The crate from crates.io carries the C sources and the generated unit tables in `c/` and compiles those; in a repository checkout `build.rs` runs the table generators and compiles `../src` and `../include` instead. Either way a downstream crate needs a C compiler at build time (plus Python 3 in the checkout case, §1) and nothing at run time.

`build.rs` re-runs when the C sources, the generators, the interpreter override, the enabled features or `WASI_SDK_PATH` change.

3.1 WebAssembly (`wasm32-wasip2`)

A WebAssembly build needs a C compiler for WASI, wasi-sdk. Point `WASI_SDK_PATH` at it and build for `wasm32-wasip2`:

```
rustup target add wasm32-wasip2
export WASI_SDK_PATH=/opt/wasi-sdk        # where wasi-sdk is unpacked
cargo build --release --target wasm32-wasip2
```

The same applies to a crate that depends on this one and is built for `wasm32-wasip2`: set `WASI_SDK_PATH` for its build.

The crate's own tests run there under wasmtime, which must see the checkout (the examples the tests read, and the build's output):

```
CARGO_TARGET_WASM32_WASIP2_RUNNER="wasmtime run --dir $(git rev-parse --show-toplevel)" \
    cargo test --target wasm32-wasip2 --lib --tests
```

A non-default `CARGO_TARGET_DIR` outside the checkout needs a `--dir` of its own. WASI has no threads and a panic aborts, so the one test of each is ignored there; the doctests run on native targets only.

3.2 Unit-profile features

Each unit-profile notation of doc/11 is a Cargo feature, and all seven are on by default:

Feature	Notation	C switch
`ucum`	`ucum:`	`BVNR_WITH_UCUM_PROFILE`
`unece`	`unece:`	`BVNR_WITH_UNECE_PROFILE`
`qudt`	`qudt:`	`BVNR_WITH_QUDT_PROFILE`
`qudt-qk`	`qudt-qk:`	`BVNR_WITH_QUDT_QK_PROFILE`
`udunits`	`udunits:`	`BVNR_WITH_UDUNITS_PROFILE`
`om`	`om:`	`BVNR_WITH_OM_PROFILE`
`cf`	`cf:`	`BVNR_WITH_CF_PROFILE`

Leaving a feature out drops that notation's tables from the library, as `-DBVNR_WITH_<NAME>_PROFILE=OFF` does for the CMake build, and the reader then refuses its codes. `THIRD_PARTY_NOTICES.md` says why a distributor may want to leave out `ucum` or `cf`:

```
[dependencies]
bovnar = { version = "2.1.0", default-features = false, features = ["unece", "qudt",
"qudt-qk", "udunits", "om"] }
```

3.3 How the published crate is made

`build.rs` reaches `../src`, which a package made from `rust/` would not contain, so `rust/Cargo.toml` keeps `publish = false` and the crate is published from a staged copy. `rust/package.py` builds that copy: the crate without `tests/` and `fuzz/` (they read `../examples` and `../src/gendata`), `LICENSE`, `THIRD_PARTY_NOTICES.md`, and in `c/` the headers, the C sources `build.rs` compiles and the generators' output. It then runs cargo on the copy:

```
python3 rust/package.py                          # cargo package: pack and verify
python3 rust/package.py -- publish --dry-run     # what CI runs on every push
```

CI's `rust_package` job runs that dry run on every push, and `publish_crate` publishes the same way on a `vX.Y.Z` tag through crates.io Trusted Publishing.

The crate carries `PROVENANCE.json`: the repository, the commit and, on a release, the tag it was staged from, and for every file in the crate its SHA-256 and the repository path it was copied from (`"generated"` for the tables in `c/generated/`). A program that vendors the C library from `c/` can tie what it compiles, and the notices it ships, to that one revision by comparing its copies against those hashes. Before cargo runs, `package.py` checks the

staged copy with `check_crate_provenance.py`, which stages the named commit again and requires the same files byte for byte. `rust_package` checks the packed `.crate` the same way, and a publish refuses uncommitted changes and a commit without its `vX.Y.Z` tag. To check a published crate against the repository:

```
python3 check_crate_provenance.py bovnar-X.Y.Z.crate --tag vX.Y.Z
```

---

4. Quick-start

4.1 High-level API (`parse` / `to_string`)

The dynamic document type is `Value`. `parse` reads BVNR bytes into a `Value` and `to_string` writes a `Value` out as BVNR bytes:

```rust
use bovnar::{Value, parse, to_string};

fn main() -> Result<(), bovnar::Error> {
    // Serialise a document to BVNR bytes
    let data = Value::Struct(vec![
        ("sensor_id".into(), Value::Int(42)),
        ("temperature".into(), Value::Float(-3.7)),
        ("label".into(), Value::String("outdoor".into())),
        ("active".into(), Value::Bool(true)),
        ("payload".into(), Value::Null),
    ]);
    let bvnr_bytes = to_string(&data, true)?;

    // Deserialise it back
    let recovered = parse(&bvnr_bytes)?;
    assert_eq!(recovered, data);
    let Value::Struct(fields) = recovered else { unreachable!() };
    assert_eq!(fields[0], ("sensor_id".to_string(), Value::Int(42)));
    Ok(())
}
```

`Value` is what `parse` returns and `to_string` takes:

Variant	Notes
`Null`	
`Bool(bool)`	
`Int(i128)`	Any integer an `i128` holds, every `uint64` and `sint64` included
`BigInt(num_bigint::BigInt)`	An integer outside the `i128` range (a `<uint:256>`, say); `to_string` writes it as `<uint:N>` / `<sint:N>`, `N` the smallest power of two from 128 that holds it
`Float(f64)`	`nan`, `inf` and `ninf` too, under an integer type as well (spec §6.4)
`String(String)`	Symbols and references read back as strings too
`Bytes(Vec<u8>)`	An octet stream

Variant	Notes
`Array(Vec<Value>)`	A multi-row array is an array of row arrays
`Struct(Vec<(String, Value)>)`	Keys in document order; a struct is a `Vec` of pairs, not a map
`Quantity(Box<Quantity>)`	A typed scalar with its literal, type and unit
`Decimal(Decimal)`	Written as an exact `float_dec:64` literal
`Fraction(BigRational)`	Written as an exact `float_dec:64` literal when it terminates

`parse_typed` is `parse` with typed values kept: they come back as `Value::Quantity` instead of being decoded to native scalars. That preserves the original text representation, exact type width, numeral base and physical unit for lossless round trips. What only the annotation carries comes back too: a quoted numeric literal under a default annotation (`<float:64,_10> "5.7"`), and a typed null, alone (`<float:64> ;`) or as an array element (`[<uint:32>, <sint:64>]`):

```rust
use bovnar::{Value, parse_typed, to_string};

fn main() -> Result<(), bovnar::Error> {
    let bvnr = b".pressure = <float:32,Pa> 101325.0;";
    let doc = parse_typed(bvnr)?;
    let Value::Struct(fields) = &doc else { unreachable!() };
    let Value::Quantity(q) = &fields[0].1 else { unreachable!() };
    assert_eq!(q.raw.as_deref(), Some("101325.0"));
    assert_eq!(q.unit_str(), "Pa");

    // to_string() accepts Quantity values — annotation and raw text are re-emitted as-is
    let out = to_string(&doc, true)?;
    assert_eq!(parse_typed(&out)?, doc);
    Ok(())
}
```

`to_string(value, pretty)` takes a `Value::Struct` — any other top-level value is an `Error::Argument`. Its write buffer starts at 4 MiB and doubles on overflow up to 256 MiB; there is no `cap` argument. It returns `Vec<u8>` rather than `String`, because an octet stream is written raw. `write_to(value, pretty, out)` does the same into any `std::io::Write`, and `parse_reader(input)` / `parse_typed_reader(input)` read from any `std::io::Read` (§4.7).

`parse` and its siblings read with the default `ReadOptions`; `parse_with_options(data, typed, &options)` and `parse_reader_with_options(input, typed, &options)` take any of them (§10.3) — a total cap, `strict_version`, `text_only`, the length and nesting caps — with `typed` choosing between `parse` and `parse_typed`:

```rust
use bovnar::{ErrorCode, ReadOptions};

fn main() {
    let untrusted = ReadOptions {
        max_file_size: 1 << 20,
        max_string_length: 4096,
        strict_version: true,
        ..ReadOptions::default()
    };
    let err = bovnar::parse_with_options(b"#!bovnar 9.0\n.a = 1;", false, &untrusted)
        .unwrap_err();
    assert_eq!(err.code(), Some(ErrorCode::UnsupportedSpecVersion));
}
```

`to_string()` writes what `parse()` reads — the streaming tier, not the document tier. `parse()` checks the grammar and each value's own type, unit and range and stops there; `to_string()` is its inverse and stops in the same place, so `to_string(&parse(x)?)` works for every `x` that `parse()` accepts. Three shapes are therefore written without complaint and then **refused by the document tier** (`DomDoc::parse`, `bovnar validate`), where an array is homogeneous and rectangular:

Rust value	Written as	Document tier says
`Array(vec![Int(1), String("a".into())])` — an array mixing kinds	`[1, "a"]`	`array_element_type_mismatch`
a ragged array of arrays — rows of 1 and 2 elements	`[[1], [2, 3]]`	`array_row_size_mismatch`
structs that do not share their keys	`[{.a = <uint:64> 1;}, {.b = <uint:64> 2;}]`	`struct_shape_mismatch`

An array mixing `Int` and `Float`, or holding `Null` beside values, is fine at both tiers. If the output has to pass `bovnar validate`, run it through `DomDoc::parse` before shipping it.

What `to_string()` does refuse, with `Error::Argument`, is what neither tier reads: an array whose elements carry different units, an integer wider than 64 bits in an array that cannot take the width annotation it needs (`[Int(1 << 70), Float(1.5)]`), and a `Decimal` past the range of `float_dec:64`. A key that is not a string is not on the list: a struct key is a `String`, so the type system has already refused it.

4.2 SAX-style streaming reader

The callback receives exactly **one** argument, the `EventData`. There is no userdata/context argument — capture external state in the closure instead. The closure returns `true` to go on and `false` to stop the read (which then ends in an `Error::Parse`).

```rust
use bovnar::{Event, EventData, ReadOptions, Reader};

fn main() -> Result<(), bovnar::Error> {
    let bvnr_bytes = b".a = 1; .b = \"two\";";
    let mut keys = Vec::new();
    let mut values = Vec::new();
    let mut on_event = |d: &EventData<'_>| {
        if d.event == Event::AssignmentStart {
            keys.push(d.text());
        } else if d.event == Event::Data {
            values.push(d.raw.to_vec());
        }
        true // returning false stops parsing (an Error::Parse)
    };

    let mut r = Reader::new()?;
    r.read_mem(bvnr_bytes, &ReadOptions::default(), Some(&mut on_event), None)?;
    assert_eq!(keys, ["a", "b"]);
    assert_eq!(values[0], b"1");
    Ok(())
}
```

`EventData` is the payload passed to a callback:

Field / method	Type	Description
`d.event`	`Event`	The event code
`d.token_type`	`Option<TokenType>`	How the token was written: number, string, symbol, ...
`d.value_type`	`ValueTypeSpec`	The type annotation (family, width, base)
`d.value_unit`	`Unit`	The unit the value carries (dimensionless when it has none)
`d.raw`	`&[u8]`	The raw token bytes
`text()`	`String`	The token bytes as text, invalid UTF-8 replaced
`d.frac`	`&[u8]`	Verbatim ISO sub-second digits of a `datetime` literal (spec 1.1); empty when there are none
`d.unit_source`	`&[u8]`	The unit exactly as the document spelled it (library 2.0); empty when the value carries no unit
`d.converted`	`bool`	`true` when a `want_unit` conversion or a unit policy was applied
`d.converted_text`	`&[u8]`	The exact converted value as a positional string; empty if the conversion does not terminate in its base
`d.converted_base`	`u32`	The base `converted_text` is rendered in
`d.converted_unit`	`Unit`	The unit the converted value is in
`d.converted_rational()`	`Option<(String, String)>`	The exact conversion as `(numerator, denominator)` decimal strings — they may exceed any machine integer

Field / method	Type	Description
`d.converted_in_base(base)`	`Result<Option<String>>`	The exact conversion re-rendered in `base`

The borrow ends with the callback. `EventData<'a>` borrows the reader's memory, so a field cannot be kept past the call; copy what you need (`text()`, `d.raw.to_vec()`). The owned copy that can outlive its callback is `EventPayload`, which `iter_mem` collects for you.

4.3 Collecting events

`Reader::iter_mem` returns the events as a `Vec` of owned `EventPayload` values (the whole stream, read to the end):

```rust
use bovnar::Reader;

fn main() -> Result<(), bovnar::Error> {
    let bvnr_bytes = b".a = 1; .b = 2;";
    let mut r = Reader::new()?;
    for payload in r.iter_mem(bvnr_bytes, true, 0)? {
        println!("{:?} {}", payload.event, payload.text());
    }
    Ok(())
}
```

4.4 DOM (random-access) API

```rust
use bovnar::{DomDoc, DomType};

fn main() -> Result<(), bovnar::Error> {
    let bvnr_bytes = br#"
        .sensor_id = 42;
        .config = { .host = "example.com"; };
        .values = [1.5, 2.5, 3.5];
    "#;
    let doc = DomDoc::parse(bvnr_bytes)?;

    // Top-level key lookup
    let node = doc.get("sensor_id").expect("present");
    assert_eq!(node.as_i64(), Some(42));
    assert_eq!(node.value_type().width, 64);

    // Struct traversal — by member, or by dotted path
    let cfg = doc.get("config").expect("present");
    assert_eq!(cfg.get("host").and_then(|n| n.as_str()), Some("example.com"));
    assert_eq!(doc.lookup("config.host").and_then(|n| n.as_str()), Some("example.com"));

    // Array access
    let arr = doc.get("values").expect("present");
    assert_eq!(arr.dom_type(), Some(DomType::Array));
    let sum: f64 = (0..arr.len()).filter_map(|i| arr.at(i)?.as_float()).sum();
    assert_eq!(sum, 7.5);

    // Convert the entire document to a plain Value (drops type/unit info)
    let value = doc.to_value()?;
    assert!(matches!(value, bovnar::Value::Struct(_)));
    Ok(())
}
```

A `DomNode` is a `Copy` view that borrows its `DomDoc`, so the compiler — not a convention — keeps the document alive for as long as a node is in use.

4.5 Low-level writer

```rust
use bovnar::{Base, Writer};

fn main() -> Result<(), bovnar::Error> {
    let mut w = Writer::to_mem(4096, true)?;
    w.write_float("velocity", 9.81, 64)?;
    w.write_uint("count", 1024, 32, Base::DECIMAL)?;
    w.finish()?;

    let output: &[u8] = w.get_output();
    assert_eq!(bovnar::parse(output)?, bovnar::parse(b".velocity = 9.81; .count =
<uint:32> 1024;")?);
    Ok(())
}
```

The scalar writers take **no unit arguments**: write a value with a unit through `write_quantity` or `write_array`, which take a `Unit`.

4.6 Streaming / framing (`bovnar::stream`)

Bindings for the streaming layer (see Streaming, Framing & Multiplexing for the full treatment). The functions work on in-memory buffers:

```rust
use bovnar::{Value, parse, stream, to_string};

fn main() -> Result<(), bovnar::Error> {
    let one = Value::Struct(vec![("id".into(), Value::Int(1))]);
    let two = Value::Struct(vec![("id".into(), Value::Int(2))]);

    // Multi-document record framing
    let blob = stream::dump_documents(&[one.clone(), two.clone()], true)?;
    let docs = stream::load_documents(&blob, false, false, 0)?;
    assert_eq!(docs, [Some(one.clone()), Some(two)]);
    // With continue_past_failed = true a document that fails is None and the rest are
read
    let docs = stream::load_documents(&blob, false, true, 0)?;
    assert_eq!(docs.len(), 2);

    // Octet multiplexing: (channel, payload) of any size, interleaved & reassembled
    let multiplexed = stream::mux_dump(&[(1, b"hello".as_slice()), (42,
b"world".as_slice())], "mux")?;
    let messages = stream::mux_load(&multiplexed, 0, None)?;
    assert_eq!(messages, [(1, b"hello".to_vec()), (42, b"world".to_vec())]);

    // Document-in-document
    let outer = stream::embed_document(&to_string(&one, true)?, "payload", true)?;
    let Value::Struct(fields) = parse(&outer)? else { unreachable!() };
    let Value::Bytes(inner) = &fields[0].1 else { unreachable!() };
    assert_eq!(stream::parse_embedded(inner, false)?, one);
    Ok(())
}
```

All arguments are positional:

Function	Arguments
`frame_documents(docs)`	already-serialised documents, `&[impl AsRef<[u8]>]`
`dump_documents(values, pretty)`	`&[Value]`
`load_documents(data, typed, continue_past_failed, max_document_size)`	`max_document_size` 0 is the default cap (`BVNR_DOC_DEFAULT_MAX`, 256 MiB)
`load_documents_results(data, typed, max_document_size, options)`	every frame read, a failed document `Err` with its error code
`mux_dump(messages, key)`	`(u64, payload)` pairs; the conventional key is `"mux"`
`mux_load(data, max_message, key)`	`max_message` 0 is the default cap (`BVNR_MUX_DEFAULT_MAX`, 64 MiB); `key` `None` accepts any
`embed_document(inner, key, pretty)`	the conventional key is `"payload"`
`parse_embedded(inner, typed)`	

`load_documents` with `continue_past_failed` gives `None` for a document that failed, and the error is gone. `load_documents_results` (and `load_documents_reader_results`) keep it: every frame is read, as with `continue_past_failed`, and the document at index i is `Ok(value)` or the `Err` it failed with — an `Error::Parse` with the document's error code and no position, which the frame reader does not report. The outer `Err` is the stream's own failure: the framing, a frame over the cap. `live::FrameParser` hands each document's index and result to `on_document` already (§4.9).

```
use bovnar::{ErrorCode, ReadOptions, Value, stream};

fn main() -> Result<(), bovnar::Error> {
    let frames = stream::frame_documents(&[&b".a = 1;"[..], b".a = @;", b".b = 2;"])?;
    let docs = stream::load_documents_results(&frames, false, 0,
&ReadOptions::default())?;
    assert_eq!(docs[0], Ok(Value::Struct(vec![("a".into(), Value::Int(1))])));
    assert_eq!(docs[1].as_ref().unwrap_err().code(),
Some(ErrorCode::UnexpectedInputByte));
    assert!(docs[2].is_ok());
    Ok(())
}
```

`parse_embedded` returns an `Error::Parse` carrying the inner document's own error code and, where the library reported one, its position. `mux_dump` channels are `u64`, so the channel range is checked by the type.

The readers read with the default `ReadOptions`. Each has a twin that takes them as its last argument — `load_documents_with_options`, `mux_load_with_options`, `parse_embedded_with_options`, and the same for the `*_reader` functions — and reads under them: `load_documents` each document, with `max_file_size` capping one document and `max_document_size` one frame; `mux_load` the document around the stream; `parse_embedded` the embedded document, as a document of its own, with `max_file_size` capping its bytes.

Each function has a twin for `std::io::Read` or `std::io::Write` (§4.7).

The `Reader` and `Writer` types of §10 and §11 are the building blocks underneath: use them directly when a document is produced or consumed event by event rather than as one buffer.

4.7 Files and I/O streams (`std::io::Read` / `Write`)

Every entry point that reads or writes a whole document has a twin that takes any `std::io::Read` or `std::io::Write`: a `File`, a pipe, a socket, a `Cursor`. There is no separate path-based or file-descriptor form; open the `File` and pass it in.

```rust
use std::io::Cursor;
use bovnar::{Base, DomDoc, ReadOptions, Reader, Writer};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let path = std::env::temp_dir().join(format!("bovnar-doc-{}.bvnr",
std::process::id()));

    // Write: the document streams into the file, the rest on finish()
    let mut w = Writer::to_writer(std::fs::File::create(&path)?, false)?;
    w.write_uint("port", 443, 16, Base::DECIMAL)?;
    w.finish()?;
    drop(w);

    // Random access over the file
    let dom = DomDoc::parse_reader(std::fs::File::open(&path)?)?;
    assert_eq!(dom.lookup("port").and_then(|n| n.as_u64()), Some(443));
    std::fs::remove_file(&path)?;

    // Streaming over any reader
    let mut keys = Vec::new();
    let mut on_event = |d: &bovnar::EventData<'_>| {
        if d.event == bovnar::Event::AssignmentStart {
            keys.push(String::from_utf8_lossy(d.raw).into_owned());
        }
        true
    };
    Reader::new()?.read_reader(Cursor::new(b"a = 1;"), &ReadOptions::default(),
Some(&mut on_event), None)?;
    assert_eq!(keys, ["a"]);
    Ok(())
}
```

The plain-value and `stream` functions have twins too:

```rust
use std::io::Cursor;
use bovnar::{parse_reader, stream, write_to};

fn main() -> Result<(), bovnar::Error> {
    let value = parse_reader(Cursor::new(b"a = 1;"))?;

    // Serialise into any Write; here a Vec<u8>
    let mut out = Vec::new();
    write_to(&value, false, &mut out)?;
    assert_eq!(bovnar::parse(&out)?, value);

    // Framed documents round-trip through a Write and a Read
    let mut framed = Vec::new();
    stream::dump_documents_to(&[value.clone(), value.clone()], false, &mut framed)?;
    let docs = stream::load_documents_reader(Cursor::new(framed), false, false, 0)?;
    assert_eq!(docs, [Some(value.clone()), Some(value)]);
    Ok(())
}
```

In-memory entry point	Reader or writer twin
`parse(data)`	`parse_reader(input)`
`parse_typed(data)`	`parse_typed_reader(input)`
`parse_with_options(data, typed, &options)`	`parse_reader_with_options(input, typed, &options)`
`to_string(value, pretty)`	`write_to(value, pretty, out)`

In-memory entry point	Reader or writer twin
`Reader::read_mem(data, …)`	`Reader::read_reader(input, &options, on_verified, on_unverified)`
`Reader::read_mem_want_unit(data, …)`	`Reader::read_reader_want_unit(input, &options, want_unit, on_verified)`
`Reader::read_mem_with(data, &options, handlers)`	`Reader::read_reader_with(input, &options, handlers)`
`Reader::read_mem_mirror(data, &options, handlers, mirror)`	`Reader::read_reader_mirror(input, &options, handlers, mirror)`
`canonicalize(data, pretty, &options)`	`canonicalize_reader(input, out, pretty, &options)`
`DomDoc::parse(data)`	`DomDoc::parse_reader(input)`
`DomDoc::parse_with_policy(data, &policy)`	`DomDoc::parse_reader_with_policy(input, &policy)`
—	`DomDoc::parse_reader_capped(input, max_bytes)`, a lower cap on what is read (§12.2)
`Writer::to_mem(cap, pretty)`	`Writer::to_writer(out, pretty)`
`Writer::to_mem_with(cap, pretty, unit_flags, emit_version)`	`Writer::to_writer_with(out, cap, pretty, unit_flags, emit_version)`
`Writer::to_mem_with_options(cap, options)`	`Writer::to_writer_with_options(out, cap, options)`
`stream::frame_documents(docs)`	`stream::frame_documents_to(docs, out)`
`stream::dump_documents(values, pretty)`	`stream::dump_documents_to(values, pretty, out)`
`stream::load_documents(data, typed, continue_past_failed, max_document_size)`	`stream::load_documents_reader(input, typed, continue_past_failed, max_document_size)`
`stream::load_documents_with_options(data, typed, continue_past_failed, max_document_size, &options)`	`stream::load_documents_reader_with_options(input, typed, continue_past_failed, max_document_size, &options)`
`stream::load_documents_results(data, typed, max_document_size, &options)`	`stream::load_documents_reader_results(input, typed, max_document_size, &options)`
`stream::mux_dump(messages, key)`	`stream::mux_dump_to(messages, key, out)`
`stream::mux_load(data, max_message, key)`	`stream::mux_load_reader(input, max_message, key)`
`stream::mux_load_with_options(data, max_message, key, &options)`	`stream::mux_load_reader_with_options(input, max_message, key, &options)`
`stream::embed_document(inner, key, pretty)`	`stream::embed_document_to(inner, key, pretty, out)`
`stream::parse_embedded(inner, typed)`	`stream::parse_embedded_reader(input, typed)`
`stream::parse_embedded_with_options(inner, typed, &options)`	`stream::parse_embedded_reader_with_options(input, typed, &options)`

They stream. A reader twin hands its `Read` to the C library as a callback source (`bvnr_source_from_callback`): the library pulls from it as it parses, so the events of what has

arrived are delivered while the rest is still to come, and the input is never held whole (`Reader::read_reader` keeps memory flat however long the input is; the twins that return values hold those values). The results are those of the in-memory twin for the same bytes, `file_too_long` at the same offset for a document over `ReadOptions::max_file_size` and the `max_document_size` and `max_message` of `load_documents_reader` and `mux_load_reader` included. The input is read until it reports its end (`Ok(0)`), so these suit a file, a pipe or a blocking socket; for input that is handed over when it arrives, use the fed readers of §4.9. Only the DOM reads its input into memory first, up to `DOM_READ_MAX_BYTES` (256 MiB, the cap of `bvn_dom_parse_fd`), because it builds the whole document anyway.

A writer twin, and `Writer::to_writer`, push the document into `out` through a callback sink (`bvnr_sink_to_callback`) as it is serialised: the C writer batches its output as it does for an fd, so `out` receives 64 KiB blocks as they fill and the rest on `finish()`, which then flushes `out` once; a `to_writer` writer also hands over what it holds whenever `flush()` is called (§11.5). The bytes are those of the in-memory twin. Since output leaves as it is made, a value that cannot be serialised part-way through leaves the blocks before it in `out` (a value that is not a struct at all leaves nothing), and a `to_writer` writer dropped unfinished drops what it still holds. `to_writer` has no cap on the document's size; `to_writer_with(out, cap, …)` keeps `cap` as one, and output past it fails with `sink_buffer_exhausted`, as with `Writer::to_mem` -- from the call that pushes it into `out` (the write that fills a block, or `flush()` or `finish()`), not necessarily the write that made it. `Writer::get_output()` is empty for such a writer. `out` must be `Send + 'static`; the writer stays `Send` (§14).

Short reads and writes are continued and `Interrupted` is retried. A failing `input` or `out` is an `Error::Io` (§13) with the code the C library gives an unreadable fd source or unwritable fd sink (`reading_from_source_fd`, `writing_to_sink`), after the events of what had arrived, and a `to_writer` writer reports it from every later call too. A panic inside `input` or `out` reaches the caller, as one in an event handler does (§13); it never unwinds through the C library. No file descriptor is involved, so these work the same on Windows.

4.8 Live streams: the framing codec (`bovnar::codec`)

The `stream` functions of §4.6 work on whole buffers. `bovnar::codec` does the same two wire conventions — octet multiplexing and `BVF1` frames (Streaming §2 and §3) — on a live stream: the writer hands out each message's bytes as it is sent, and the reader is fed bytes in pieces of any size, as they arrive from a socket or a pipe, and hands out each message as soon as its last byte is in. The module is pure Rust (`#![forbid(unsafe_code)]`): no message goes through the C library, and nothing is allocated per message once the reader's buffers have grown to the largest message.

```rust
use bovnar::codec::{MuxReader, MuxWriter};

fn main() -> Result<(), bovnar::Error> {
    // A stream under the key "media" that opens with ordinary values
    let mux = MuxWriter::with_preamble(b".meta = { .codec = \"av1\"; .rate = 30; };", "me-
dia")?;
    let mut wire = Vec::new(); // stands in for a socket
    wire.extend_from_slice(mux.opener());
    let mut head = Vec::new();
    for (channel, frame) in [(1, &b"key frame"[..]), (2, b"audio"), (1, b"delta")] {
        // Only the chunk headers are written, into `head`; the frame is
        // borrowed, ready for a vectored write
        for part in mux.encode(&mut head, channel, frame) {
            wire.extend_from_slice(part);
        }
    }
    wire.extend_from_slice(mux.terminator());

    // The C library reads the same bytes
    assert_eq!(bovnar::stream::mux_load(&wire, 0, Some("media"))?.len(), 3);

    // Fed in pieces, as from a socket
    let mut reader = MuxReader::with_options(0, Some("media"))?;
    let mut received = Vec::new();
    for piece in wire.chunks(7) {
        let mut input = piece;
        while let Some(m) = reader.read(&mut input)? {
            received.push((m.channel, m.payload.to_vec()));
        }
    }
    reader.finish()?; // the input ended where a stream may end
    assert_eq!(received[1], (2, b"audio".to_vec()));
    let meta = reader.preamble_value(false)?.expect("the stream is open");
    assert_eq!(meta, bovnar::parse(b".meta = { .codec = \"av1\"; .rate = 30; };")?);
    Ok(())
}
```

Writing. `MuxWriter::new(key)` or `MuxWriter::with_preamble(preamble, key)` validates the key (1 to 255 bytes, the identifier rules of spec §4.2: what the C writer accepts and a reader at its defaults reads) and the preamble, once. A stream is `opener()`, the messages, then `terminator()`; `begin(&mut vec)` and `finish(&mut vec)` append them to a `Vec`. Each message is one of:

Call	What it writes
`encode(&mut head, channel, payload)`	only the chunk headers, into `head` (cleared, then reused); returns `Parts`, an iterator over the message's wire bytes in order — header, payload slice, header, … — for a vectored write
`write_message(&mut vec, channel, payload)`	the whole message, appended to a `Vec`
`write_message_to(&mut out, channel, payload)`	the whole message to any `std::io::Write`, one vectored write per chunk, nothing allocated; `out` is not flushed, and must block (a failure part-way, `WouldBlock` included, leaves the stream broken; for a non-blocking socket write `encode`'s parts yourself)

A message is cut into chunks exactly as `bvnr_mux_send` cuts it (a message of at most `stream::BVNR_MUX_MAX_MESSAGE` bytes is one chunk), so the bytes are those of `stream::mux_dump` for the same key and messages.

Reading. `MuxReader::read(&mut input)` takes `input: &mut &[u8]`, the bytes not yet read, reads from its front and returns the next complete message as `Message { channel, payload }`. `None` means every byte of `input` was taken (an unfinished message is kept inside the reader), so the buffer can be refilled; after a message, call again with the same slice until `None`. A message that lies whole in `input` is borrowed from it; one that spans pieces or chunks is gathered in a buffer per channel that is reused. The slice is valid until the next call. `finish()` checks that the input ended after the document's `;` (or before its first byte); `reset()` readies the reader for another stream, keeping the buffers it has grown. The reader reads every valid octet region, not only what `bvnr_mux_send` writes: several messages packed into one chunk, channels interleaved chunk by chunk, a length varint split across chunks.

Option	Default
`MuxReader::with_options(max_message, key)`	`max_message` 0 is 64 MiB, as for `mux_load`; with `key`, a stream under another key is an `Error::Argument` naming both (`bvnr_demux_set_key` would ignore it, but this reader reads one stream)
`.with_max_preamble(bytes)`	64 KiB of text before the octet stream, preamble and opening assignment together; more is `text_data_too_long`

Peek, skip, forward. For a router or relay, `next_item(&mut input, n)` also stops at the start of each message with `Item::Peek`, holding its channel, its declared length and its first `n` bytes; nothing else of the message has been read. Drop the peek to receive the message as `Item::Message`, call `skip()` to have its bytes passed over as they arrive, or `forward()` to receive it as `Item::Wire`: whole chunks carrying that message alone, which can be written into any other multiplexed stream between two messages. When those chunks lie consecutive in the input, `Wire::bytes` is the input itself, not copied; otherwise the reader gathers the message and writes it out again as `MuxWriter` would.

```rust
use bovnar::codec::{Item, MuxReader, MuxWriter};

fn main() -> Result<(), bovnar::Error> {
    let mux = MuxWriter::new("mux")?;
    let mut inbound = Vec::new();
    mux.begin(&mut inbound);
    for (channel, msg) in [(1, &b"for the relay"[..]), (2, b"local only"), (1, b"again")]
{
        mux.write_message(&mut inbound, channel, msg);
    }
    mux.finish(&mut inbound);

    // Forward channel 1, drop the rest, decided on the peek alone
    let mut reader = MuxReader::new();
    let mut outbound = mux.opener().to_vec();
    let mut input = &inbound[..];
    while let Some(item) = reader.next_item(&mut input, 0)? {
        match item {
            Item::Peek(p) if p.channel == 1 => p.forward(),
            Item::Peek(p) => p.skip(),
            Item::Wire(w) => outbound.extend_from_slice(w.bytes),
            Item::Message(_) => unreachable!("every message was decided"),
        }
    }
    outbound.extend_from_slice(mux.terminator());
    let relayed = bovnar::stream::mux_load(&outbound, 0, None)?;
    assert_eq!(relayed, [(1, b"for the relay".to_vec()), (1, b"again".to_vec())]);
    Ok(())
}
```

The preamble. Text before the stream's assignment — complete statements, a `#!bovnar` directive first if needed — lets a stream carry ordinary values about itself in the same document. `with_preamble` checks it once with the full parser; it must be text, so an octet stream in it is `octet_stream_forbidden` (a reader takes the document's first `0x00` for the stream's opener). The reader gathers the text up to that `0x00`, requires it to end with the assignment `.KEY =`, checks the preamble with the full parser (once per stream; a bare `.KEY=` or `.KEY =` skips the check), and then reads the octet stream in pure Rust. `read_preamble(&mut input)` reads only up to the opener and returns the preamble, so it can be acted on before any message; `preamble()` returns it afterwards and `preamble_value(typed)` parses it into a `Value`. A C-written document with values before `bvnr_mux_begin` reads the same way.

What the reader reads. A preamble (possibly empty), one octet stream as the value of the document's last assignment (`.KEY =`, with no comment between the `=` and the stream), `;`, then only whitespace and comments, which may also come before the `;`. A statement after the stream, or any other byte after it, is `unexpected_input_byte`: the codec reads multiplexed streams, not the grammar. Parse a general document with `stream::mux_load` or a `Reader`.

Errors are `Error::Parse` with the code the C demultiplexer gives (`mux_load` gives the same code for a desync, an oversized message, the 4097th channel and a cut-off stream; what only the codec refuses -- bytes after the stream, a preamble over its cap, a key other than the expected one -- has a code of its own), and they are latched:

Condition	Code
A chunk tag other than `0x01` / `0x00`, a channel varint not whole in its chunk, a varint over 10 bytes or 64 bits, a 4097th distinct channel	`octet_stream_out_of_sync`
A declared length over `max_message`	`value_out_of_range`
The octet stream closes inside a message or its length	`octet_stream_truncated`
`finish()` inside the octet stream	`read_complete_chunk_failed`
`finish()` before the stream opened, or before its `;`	`got_incomplete_bvnr_stream`
A comment after the `;` that is not UTF-8, or that the input ends inside a character of (`finish()`)	`invalid_utf8_byte`

Frames. `codec::frame_header(len)`, `write_frame(&mut vec, doc)` and `write_frame_to(&mut out, doc)` (one vectored write, no copy) write what `bvnr_frame_write` writes. `FrameReader::read(&mut input)` returns each complete document's bytes, borrowed when whole in the input, gathered otherwise; parse each with `parse`, a `Reader` or `DomDoc`. `FrameReader::with_max_document_size(n)` caps a document (0 is 256 MiB, as in `bvnr_doc_stream_read`; the buffer only grows with bytes that arrived). A bad magic is `unexpected_input_byte`, a length over the cap `value_out_of_range`, and `finish()` inside a frame `got_incomplete_bvnr_stream`.

```
use bovnar::codec::{FrameReader, write_frame};

fn main() -> Result<(), bovnar::Error> {
    let mut wire = Vec::new();
    write_frame(&mut wire, b".id = 1;");
    write_frame(&mut wire, b".id = 2;");
    assert_eq!(wire, bovnar::stream::frame_documents(&[".id = 1;", ".id = 2;"])?);

    let mut reader = FrameReader::new();
    let mut ids = Vec::new();
    for piece in wire.chunks(5) {
        let mut input = piece;
        while let Some(doc) = reader.read(&mut input)? {
            ids.push(bovnar::parse(doc)?);
        }
    }
    reader.finish()?;
    assert_eq!(ids.len(), 2);
    Ok(())
}
```

Cost per message, from `cargo bench --bench codec`: a release build pinned to one core (`taskset -c 3`) of an Intel Core i7-2700K (2011), whose clock scales with load. Each figure is the median of seven runs (each run the median of nine passes over at least 64 MiB of messages), and in brackets the fastest run. The stream is a few MiB held in the cache, as the bytes of a socket read are when the reader sees them; it is opened once and fed the same messages every pass, as a live stream is fed, so opening it is not counted. A 400 KB message spans seven chunks.

Message	write	write + copy	peek + skip	forward	read	`mux_dump`	`mux_load`
150 B	8.4 ns (8.3)	26 ns (25)	12 ns (12)	13 ns (13)	8.2 ns (8.1)	244 ns (242)	123 ns (122)
30 KB	8.6 ns (8.6)	1.5 µs (1.5)	12 ns (12)	14 ns (14)	8.8 ns (8.8)	8.4 µs (8.2)	2.7 µs (2.5)
400 KB	37 ns (37)	22 µs (21)	55 ns (55)	56 ns (56)	25 µs (23)	110 µs (110)	76 µs (70)

write is `encode` (the headers only, for a vectored write); write + copy is `write_message` into a reused `Vec`; peek + skip and forward are `next_item`, a look at the channel and the first payload byte, and the decision, the forwarded chunks borrowed from the input; read is `read`, which gathers (copies) a message that spans chunks, so the 400 KB one; mux_dump and mux_load are the batch functions through the C library, for comparison (`mux_load` copies every message into the `Vec` it returns, which the C demux alone does not: doc/10 §8.4). Counted in instructions, which the clock does not move, a 150-byte message is about 145 to write, 120 to read and 140 to peek at and skip, the benchmark's loop included. From a stream much larger than the cache, every chunk header a 400 KB message has is a cache miss, which puts peek and forward at about 130 ns for it, and small messages are read as fast as memory delivers them: about 24 ns to read a 150-byte one and 29 ns to peek at and skip it. The other figures move little.

4.9 Live documents through C (`bovnar::live`)

The codec of §4.8 reads the framing; it does not parse documents. `bovnar::live` wraps the C library's push-fed readers (Streaming §8) for that: they are handed bytes in pieces of any size, as they come from a non-blocking socket, an event loop, a frame or another document's octet stream, and are told separately when the input is over. Each keeps one C object for its whole life, so an endless stream costs one setup, and whatever the cuts, the events, values and errors are exactly those the batch functions give for the whole input (the frame readers' framing errors aside, below).

Type	Over	What comes out
`DocumentFeed`	`bvnr_open_read_feed`	the events of `Reader::read_mem`, from each `feed` and `finish`
`ValueFeed`	the same	each top-level assignment as a `Value` once it is complete
`ValueBuilder`	—	the same from events you hand it (those of a `Demux`, say)
`EmbeddedFeed`	`DocumentFeed`	the documents embedded under a key of another document, as their chunks arrive
`FrameParser`	`bvnr_frame_reader_create`	the events and the result of every framed document

Type	Over	What comes out
`FramePull`	`bvnr_frame_reader_create_pull`	each framed document's bytes
`Demux`	`bvnr_demux_open_feed`, `bvnr_demux_next`	a multiplexed stream's messages, and the document's own events in order with them

Documents. `DocumentFeed::new(&options)` takes the `ReadOptions` of a `Reader` (and `set_unit_policy` its policy). `feed(bytes, handler)` delivers the events those bytes complete before it returns, so the buffer can be reused at once; running out of input is never an error there. `finish(handler)` declares the end: a document cut off fails there, with the code and position `read_mem` reports. `reset()` starts the next document on the same reader. An error ends the document, and every later `feed` or `finish` returns it until `reset()`. `ValueFeed` does the same and appends to a `Vec` each assignment the bytes complete, those before an error included: a scalar, a struct or an octet stream with its last event, an array (which a `/` may still continue) with the event after it. A key assigned twice comes twice, and nothing is kept once it is handed out, so an endless document is read in constant memory. Its text is capped by the library's default of 2 147 483 647 bytes per document (`error_text_data_too_long`, doc/10 §1), and each array at as many items: set `ReadOptions::max_text_bytes` (and `max_array_items`) to `u64::MAX` for a document that does not end, and `max_file_size` where a cap still matters.

```
use bovnar::live::ValueFeed;
use bovnar::{ReadOptions, Value};

fn main() -> Result<(), bovnar::Error> {
    // An endless document arriving from a socket, cut anywhere
    let pieces: [&[u8]; 4] = [b".temp = 21", b".5; .unit = \"C\"; .te", b"mp = 22.0;",
b" .tag = [1, 2];"];
    let mut feed = ValueFeed::new(false, &ReadOptions::default())?;
    let mut got = Vec::new();
    for piece in pieces {
        feed.feed(piece, &mut got)?;
    }
    assert_eq!(got[0], ("temp".to_string(), Value::Float(21.5)));
    assert_eq!(got.len(), 3, "the array may still go on");
    feed.finish(&mut got)?; // the end of the document completes it
    assert_eq!(got[3].0, "tag");
    Ok(())
}
```

Every handler. `feed` and `finish` take the event handler alone. `feed_with(bytes, handlers)` and `finish_with(handlers)` take a `ReadHandlers` (§10.2), as `Reader::read_mem_with` does: `on_verified`, the raw lexer events of `on_unverified`, the `want_unit` hook and `on_error` — which sees each error a `continue_on_error` document recovers from — called inside the call that parses the bytes they concern. `ValueFeed` takes them through its own `feed_with(bytes, out, handlers)` and `finish_with(out, handlers)`, its `on_verified` seeing each event after the value builder has. The C reader is given its callbacks when a document starts, so the first call of a document decides whether `on_unverified` and `want_unit` are called for the whole of it: a later call that needs one the document was not started with is an `Error::Argument` (`reset()` starts the next document, decided afresh),

and a later call may pass fewer. A panic in a handler reaches the caller of the call, and the document has failed.

```
use bovnar::live::DocumentFeed;
use bovnar::{EventData, ReadHandlers, ReadOptions, Unit};

fn main() -> Result<(), bovnar::Error> {
    let mut feed = DocumentFeed::new(&ReadOptions::default())?;
    let (mut tokens, mut metres) = (0, Vec::new());
    for piece in [&b".d = <uint:64,k~m"[..], b"> 3; .e = 4;"] {
        let mut on_raw = |_: &EventData<'_>| { tokens += 1; true };
        let mut want = |d: &EventData<'_>| {
            (!d.value_unit.is_dimensionless()).then(|| (Unit::parse("m").unwrap(), 0))
        };
        let mut on_value = |d: &EventData<'_>| {
            if d.converted {
metres.push(String::from_utf8_lossy(d.converted_text).into_owned()); }
            true
        };
        feed.feed_with(piece, ReadHandlers {
            on_verified: Some(&mut on_value),
            on_unverified: Some(&mut on_raw),
            want_unit: Some(&mut want),
            on_error: None,
        })?;
    }
    feed.finish(&mut |_| true)?;
    assert!(tokens > 0);
    assert_eq!(metres, ["3000"]);
    Ok(())
}
```

The mirror. `set_mirror(w)` gives the feed any `std::io::Write + Send` that receives every byte the reader consumes (`src_mirror`, a debugging aid): each piece fed, once the reader has parsed it, and a document that fails has mirrored the input up to the byte the error is at. It starts the next document, as `reset()` does, and stays for every document after; `take_mirror::<W>()` hands it back as the type it was given (`None`, and the mirror kept, for another type), and the rest of a document in progress is not mirrored. A failing mirror fails the document with an `Error::Io` with `writing_to_sink`; a panic in it reaches the caller. `ValueFeed` has the same two methods. For a `Reader`, `read_mem_mirror` and `read_reader_mirror` take a `&mut dyn std::io::Write` for the one read (§10.2). The frame readers and the demux take no mirror, as the C objects behind them do not: their caller holds the bytes it feeds.

```
use bovnar::ReadOptions;
use bovnar::live::DocumentFeed;

fn main() -> Result<(), bovnar::Error> {
    let mut feed = DocumentFeed::new(&ReadOptions::default())?;
    feed.set_mirror(Vec::<u8>::new())?;
    for piece in [&b".a = 1;"[..], b"  # note\n.b = 2;"] {
        feed.feed(piece, &mut |_| true)?;
    }
    feed.finish(&mut |_| true)?;
    assert_eq!(feed.take_mirror::<Vec<u8>>().unwrap(), b".a = 1;  # note\n.b = 2;");
    Ok(())
}
```

Frames. `FramePull` hands out each framed document's bytes, to be parsed by a reused `DocumentFeed` or `ValueFeed` (or `parse`); `codec::FrameReader` does the same in pure Rust. `FrameParser::new(&options, continue_past_failed, max_document_size)` parses every document as `stream::load_documents` does, streaming each frame's payload into one reused reader so that a frame is never held whole: `feed(bytes, on_event, on_document)` calls `on_document(index, result)` for each document it completes — `Ok(())` or the `Err` with the document's error code — and its returning false ends the stream with `scanner_callback_failed`. `feed_with(bytes, handlers, on_document)` takes a `ReadHandlers` for the documents, as `DocumentFeed::feed_with` does; the frame reader is given its callbacks when the stream starts, so the first feed decides whether `on_unverified` and `want_unit` are called. A bad magic is `unexpected_input_byte`, a frame over the cap `value_out_of_range`, a stream that does not end on a frame boundary `got_incomplete_bvnr_stream` at `finish`.

```
use bovnar::live::{FramePull, ValueFeed};
use bovnar::{ReadOptions, stream};

fn main() -> Result<(), bovnar::Error> {
    let wire = stream::frame_documents(&[&b".id = 1;"[..], b".id = 2;"])?;
    let mut frames = FramePull::new(0)?;
    let mut values = ValueFeed::new(false, &ReadOptions::default())?;
    let mut ids = Vec::new();
    for piece in wire.chunks(5) {
        let mut input = piece;
        while let Some(doc) = frames.next(&mut input)? {
            values.reset()?;
            values.feed(doc, &mut ids)?;
            values.finish(&mut ids)?;
        }
    }
    frames.finish()?;
    assert_eq!(ids.len(), 2);
    Ok(())
}
```

Multiplexed streams. `Demux::new(&DemuxOptions { max_message, key, deliver_events, read })` reads the document that carries the stream, with the key scope, `max_message`, the 4096-channel limit and every error of `stream::mux_load`. Each `next` returns a `DemuxItem::Message { channel, payload }` or, with `deliver_events`, a `DemuxItem::Event` for every event but the stream's chunks, in document order: the metadata before the stream comes out before its first message, and a `ValueBuilder` makes values of it (the demux withholds the stream's chunks but not its start and end, so the builder also hands out the stream's own key, as an empty `Value::Bytes`, once the stream ends; Python's `DemuxFeed(values=True)` leaves it out). After `finish()`, `next` on an empty slice hands out the closing events (more input is refused with an `Error::Argument`); `reset()` starts the next document: the channel buffers are kept, the channel ids (and with them the 4096-channel count) start afresh, as `codec::MuxReader::reset` does. `recovery_count()`, `skipped_bytes()` and `declared_version()` are those of the demux's reader, as on `DocumentFeed`. `next_with(input, handlers)` and `finish_with(handlers)` take a `ReadHandlers` for that reader, which parses inside `next` and `finish`: its `on_verified` sees every verified event, the stream's chunks included, before the demux does, and `on_unverified`, `want_unit` and

`on_error` are a `Reader`'s. The first call of a document decides which of `on_verified`, `on_unverified` and `want_unit` its reader calls, as for `DocumentFeed`.

```rust
use bovnar::live::{Demux, DemuxItem, DemuxOptions, ValueBuilder};
use bovnar::Writer;

fn main() -> Result<(), bovnar::Error> {
    // Metadata, the stream, and a value after it, as a C writer writes them
    let mut w = Writer::to_mem(1 << 16, false)?;
    w.begin_struct("meta")?;
    w.write_string("codec", "av1")?;
    w.end_struct()?;
    w.mux_begin("media")?;
    w.mux_send(1, b"key frame")?;
    w.mux_send(2, b"audio")?;
    w.mux_end()?;
    w.write_bool("complete", true)?;
    w.finish()?;
    let wire = w.get_output().to_vec();

    let options = DemuxOptions { key: Some("media".into()), deliver_events: true, ..De-
muxOptions::default() };
    let mut demux = Demux::new(&options)?;
    let mut meta = ValueBuilder::new(false);
    let mut messages = Vec::new();
    for piece in wire.chunks(6) {
        let mut input = piece;
        while let Some(item) = demux.next(&mut input)? {
            match item {
                DemuxItem::Event(e) => meta.push(&e.data()),
                DemuxItem::Message { channel, payload } => {
                    if messages.is_empty() {
                        // Everything before the stream is here already
                        assert_eq!(meta.drain().next().map(|(k, _)| k).as_deref(),
Some("meta"));
                    }
                    messages.push((channel, payload.to_vec()));
                }
            }
        }
    }
    demux.finish()?;
    while let Some(item) = demux.next(&mut &[][..])? {
        if let DemuxItem::Event(e) = item {
            meta.push(&e.data());
        }
    }
    assert_eq!(messages, [(1, b"key frame".to_vec()), (2, b"audio".to_vec())]);
    assert!(meta.drain().any(|(k, _)| k == "complete"));
    Ok(())
}
```

Draining. `FramePull::next` and `Demux::next` take `&mut &[u8]` and return one item per call; unlike the codec's readers, they do not advance the slice past each item. An item that lies whole in the input is borrowed from it without a copy; one that spans pieces or chunks is gathered in a buffer the reader keeps and reuses. Either way it borrows the reader until the next call, which the borrow checker enforces. The C object keeps a pointer into the bytes until they are drained, so pass the same slice to every call until one returns `None` (which empties it): the slice is not shortened meanwhile, another one is refused with an

`Error::Argument`, and so is `finish()` while bytes are left. A warm `Demux` or `FramePull` allocates nothing per message or document.

Embedded documents. `Writer::embed_document(key, doc)` puts a document into an open writer (§11.5), one after another into a `to_writer` stream if you like. To read them while the outer document arrives, hand its events to an `EmbeddedFeed`: it feeds each embedded document's chunks to one reused `DocumentFeed` as they come and returns the document's value, or its error, when its octet stream closes; `stream::parse_embedded` gathers the bytes and starts a reader for every document instead.

```
use bovnar::live::{DocumentFeed, EmbeddedFeed};
use bovnar::{EventData, ReadOptions, Value, Writer};

fn main() -> Result<(), bovnar::Error> {
    let mut w = Writer::to_mem(4096, false)?;
    for n in 1..=3 {
        w.embed_document("payload", format!(".n = {n};").as_bytes())?;
    }
    w.finish()?;
    let outer = w.get_output().to_vec();

    let mut embedded = EmbeddedFeed::new("payload", false)?;
    let mut docs = Vec::new();
    let mut on_event = |d: &EventData<'_>| {
        docs.extend(embedded.push(d));
        true
    };
    let mut feed = DocumentFeed::new(&ReadOptions::default())?;
    for piece in outer.chunks(4) {
        feed.feed(piece, &mut on_event)?;
    }
    feed.finish(&mut on_event)?;
    assert_eq!(docs.len(), 3);
    assert_eq!(docs[2], Ok(Value::Struct(vec![("n".into(), Value::Int(3))])));
    Ok(())
}
```

`live::Demux` or `codec::MuxReader`? Both read a multiplexed stream as it arrives, with the C demultiplexer's limits and error codes.

	`codec::MuxReader`	`live::Demux`
Implementation	pure Rust, no C call per message	the C parser over the whole document
Document shape	one octet stream, after an optional text preamble, ending the document	any: values before and after the stream, other octet streams beside it (`key`)
The values around the stream	the preamble's bytes, parsed on request	every event, in order with the messages
Limits and policy	`max_message`, the preamble cap	`max_message` and every `ReadOptions` limit
Extras	peek, skip and forward a message without gathering it	—

	`codec::MuxReader`	`live::Demux`
Use it for	a dedicated stream, a router or a relay	documents of any other shape, or when the values around the stream matter

Errors are those of the batch functions for the same document or mux stream, as `Error::Parse`: with a position when the document failed, the demultiplexer's code alone (`octet_stream_out_of_sync`, `value_out_of_range`, `octet_stream_truncated`) when the stream did, as `mux_load` reports them. A frame reader's errors carry no position, as `load_documents`' do not, and its framing errors are `load_documents`' too: a bad magic is `unexpected_input_byte`, a frame over the cap `value_out_of_range` and a stream cut inside a frame `got_incomplete_bvnr_stream`. A panic in a handler reaches the caller, as for `Reader` (§13).

What it costs, from `cargo bench --bench live`: a release build pinned to one core (`taskset -c 4`) of the same i7-2700K as §4.8, load average under 2; the median of three runs, each the median of nine passes. The input is a few MiB held in the cache; pieces feeds it in 1460-byte pieces, a TCP segment's payload, so most messages and documents straddle a feed. The `mux_load` and `MuxReader` columns were measured in the same run as the rest of this table, not taken from §4.8's, which is a separate run; compare figures within one table. Per message:

Message	`Demux` whole	`Demux` pieces	with `deliver_events`	`mux_load`	`MuxReader` whole	`MuxReader` pieces
150 B	100 ns	109 ns	100 ns	123 ns	8 ns	14 ns
30 KB	150 ns	2.4 µs	182 ns	2.9 µs	11 ns	1.8 µs
400 KB	26 µs	50 µs	26 µs	68 µs	23 µs	28 µs

A message whole in the fed buffer is borrowed, so its cost does not grow with its size; one that straddles a feed or spans chunks is gathered, and copying it is most of the cost, in the codec as in the demux. `mux_load` copies every message into a `Vec`. The codec is about ten times cheaper per small message: no C call and no grammar. Per document (a struct of integers and strings per line, an octet stream every 512 lines), with the throughput in brackets:

Document	`DocumentFeed` whole	`DocumentFeed` pieces	`Reader::read_mem`	`ValueFeed`	`parse`
196 B	16 µs (12 MB/s)	16 µs (12 MB/s)	16 µs (12 MB/s)	20 µs (10 MB/s)	27 µs (7 MB/s)
32 KB	2.0 ms (16 MB/s)	1.9 ms (16 MB/s)	2.0 ms (16 MB/s)	2.4 ms (13 MB/s)	2.5 ms (13 MB/s)
402 KB	24 ms (17 MB/s)	24 ms (17 MB/s)	24 ms (17 MB/s)	29 ms (14 MB/s)	30 ms (13 MB/s)

Feeding costs what reading the whole buffer costs, wherever the pieces are cut: the time is the parser's. `ValueFeed` hands each assignment out as it completes. `parse` builds the whole document, and past 32 entries keeps an index from key to entry instead of scanning the keys already there — which took its 402 KB column from 113 ms to 30 ms.

4.10 Canonical form

`bovnar::canonicalize(data, pretty, &options)` re-serialises a document from its verified events through the C library's canonicalising observer (`bvnr_canon_observer_*`, doc/08 §3.6) and returns the bytes; `canonicalize_reader(input, out, pretty, &options)` reads any `std::io::Read` as it parses and writes to any `std::io::Write` as the output is made, in 64 KiB pieces, so neither is held whole.

```
use bovnar::{ReadOptions, canonicalize};

fn main() -> Result<(), bovnar::Error> {
    let options = ReadOptions::default();
    let canon = canonicalize(b".a = 1 ;  # one\n.b = { .c = \"x\"; };", false, &options)?;
    assert_eq!(canon, b".a=<uint:64,_10,no_unit>1;.b={.c=<utf8>\"x\";};");
    assert_eq!(canonicalize(&canon, false, &options)?, canon); // idempotent
    assert_eq!(canonicalize(b".a=1;.b={.c=\"x\";};", false, &options)?, canon);
    let mut out = Vec::new();
    bovnar::canonicalize_reader(&b"#!bovnar 1.1\n.a = 1;"[..], &mut out, true, &options)?;
    assert!(out.starts_with(b"#!bovnar 1.1\n"));
    Ok(())
}
```

It is the form `bovnar events -d` prints (`-p` for `pretty`): the compact one is a single line with no whitespace outside literals, each value written with the annotation the reader gives it (`1` as `<uint:64,_10,no_unit>1`). Two documents that differ only in layout, comments and the spelling of separators canonicalise to the same bytes, and canonicalising the output again gives it back unchanged; the order of the keys is kept. Units are written as the document spelled them: this is not `bovnar pretty-print --canonical`, which writes through a writer under `UnitFlags::REDUCE | UnitFlags::NUM_CANONICAL` and folds them to one form. A leading `#!bovnar M.N` directive is kept — from an `std::io::Read` it is found in the first bytes read, before the reader sees them.

The document is read under `options`, and one the reader refuses is its `Error::Parse`. Under `continue_on_error` a document the reader had to recover from is refused too, with the first error it recovered from (code and position): the observer writes what it is fed, and the statement a recovery leaves without its value would make the output unreadable. A failing `input` is an `Error::Io` with `reading_from_source_fd`, a failing `out` one with `writing_to_sink`, and a panic in either reaches the caller; after an error `out` may hold the start of the output. The observer's C function is the reader's `on_verified` itself, with the observer as its userdata, so no Rust runs per event.

---

5. Unit helpers

The unit functions live in the `bovnar::units` module; `Unit` itself is at the crate root.

```rust
use bovnar::Unit;
use bovnar::units::{unit_factor, unit_prefix_factor, unit_to_str};

fn main() -> Result<(), bovnar::Error> {
    // Parse a unit string into a Unit
    let u = Unit::parse("k~g·m/s²")?;

    // Convert a Unit back to its canonical string
    assert_eq!(unit_to_str(&u)?, "k~g·m/s²");
    assert_eq!(u.to_string(), "k~g·m/s²"); // Display is the canonical spelling too

    // Scalar SI/IEC PREFIX factor of a unit — the prefix only, never the
    // base unit's own factor.
    assert_eq!(unit_prefix_factor(&Unit::parse("M~Hz")?), 1_000_000.0);
    assert_eq!(unit_prefix_factor(&Unit::parse("in")?),
1.0); // NOT 0.0254 — the inch has no prefix
    assert_eq!(unit_prefix_factor(&Unit::parse("h")?), 1.0);  // NOT 3600.0

    // unit_factor() takes the unit as text and is the same prefix factor
    assert_eq!(unit_factor("M~Hz")?, 1_000_000.0);
    assert_eq!(unit_factor("in")?, 1.0);
    assert_eq!(unit_factor("h")?, 1.0);

    // The FULL factor to coherent SI is Unit::factor
    assert_eq!(Unit::parse("in")?.factor()?, 0.0254);
    assert_eq!(Unit::parse("h")?.factor()?, 3600.0);
    Ok(())
}
```

`unit_factor` is the prefix scale, not the full factor. It is `unit_prefix_factor` on a unit given as text, and reports only what the prefixes contribute: `1000000.0` for `M~Hz`, `1.0` for `in` and `h`. The full factor to coherent SI is `Unit::factor` on the parsed unit (or `unit_to_si_factor`); it fails for an affine unit (`°C`) or one with no SI factor.

`Unit` is the one unit type; every function that takes a unit takes a `&Unit`.

5.1 The unit-profile notations (spec 1.2)

These notations are spec 1.2: in a document they need a `#!bovnar 1.2` directive, while the API entry points below have no document and take them unconditionally. `Unit::parse` takes a profile notation as readily as the native one, and returns the same `Unit` either way — so everything else in this chapter works on the result unchanged. **Seven** namespaces are accepted — `ucum:`, `unece:`, `qudt:`, `qudt-qk:`, `udunits:`, `om:` and `cf:` — but each is a compile-time switch of the C library, so which of them a given build carries is a question to ask rather than assume; see `available_profiles()` below.

```rust
use bovnar::{ErrorCode, Unit};
use bovnar::units::{
    available_profiles, units_compatible, unit_error_code, unit_is_profile_only,
unit_to_profile,
    unit_to_str, unit_to_ucum,
};

fn main() -> Result<(), bovnar::Error> {
    let vu = Unit::parse("ucum:mm[Hg]")?;
    assert_eq!(unit_to_str(&vu)?, "mmHg");              // the native canonical form
    assert_eq!(unit_to_ucum(&vu)?, "mm[Hg]");           // back to a UCUM code
    assert!(units_compatible(&vu, &Unit::parse("k~Pa")?));

    // Which profiles THIS build carries. Unit::parse fails with the same
    // Error::Argument for an absent namespace as for a bad unit;
    // unit_error_code() is what tells the two apart (49 against 32).
    assert_eq!(available_profiles(), ["ucum", "unece", "qudt", "qudt-qk", "udunits",
"om", "cf"]);

    // The same unit in other vocabularies — all one Unit
    let kg = Unit::parse("k~g")?;
    assert_eq!(unit_to_profile("ucum", &kg)?, "kg");
    assert_eq!(unit_to_profile("unece", &kg)?, "KGM");
    assert_eq!(unit_to_profile("qudt", &kg)?, "KiloGM");
    assert_eq!(unit_to_profile("udunits", &kg)?, "kg");
    assert_eq!(kg.to_profile("unece")?, "KGM");         // the method form

    let iu = Unit::parse("ucum:[IU]/mL")?;
    assert!(unit_is_profile_only(&iu));                 // no native spelling exists
    assert_eq!(unit_to_str(&iu)?, "ucum:[IU].mL-1");

    let packaging = Unit::parse("unece:XBX")?;
    assert!(packaging.is_profile_only());             // a countable package
    assert_eq!(unit_to_str(&packaging)?, "unece:XBX");  // in its OWN namespace

    assert_eq!(unit_error_code("ucum:B[SPL]"), ErrorCode::UnitProfileUnsupported as
i32); // 50
    assert_eq!(unit_error_code("udunits:days since 1970-01-01"), 50); // valid UDUNITS,
not carryable
    assert_eq!(unit_error_code("nosuch:m"), ErrorCode::UnitProfileUnknown as
i32);        // 49
    assert_eq!(unit_error_code("m/s"), 0);              // it parses
    Ok(())
}
```

`unit_to_profile` and `unit_to_ucum` fail rather than inventing a code, in three situations. A native unit outside that vocabulary's transliteration table has no form in it — the Old German units, the water-hardness degrees, the turbidity kinds and every currency. An opaque unit belonging to a different profile has none either: `[IU]` is UCUM's, so `unit_to_profile("unece", …)` refuses it. And a **flat** vocabulary (`unece`, `qudt`, `qudt-qk`, `om`) can spell only a single component — no product, quotient or exponent, though a prefixed one is fine (`k~g` is `unece:KGM`) — because a flat code is one whole token with no operator to build it out of — so `k~m/h` has no UNECE form even though `unece:KMH` parses to exactly it.

In a **document** the notation additionally requires a `#!bovnar 1.2` directive; without it `parse` fails with `unit_illegal`. `Unit::parse` has no document and therefore no declared version, so it accepts the notation unconditionally.

See Unit Profiles for the transliteration tables and the codes that have no representation.

5.2 Extended unit functions

The following functions operate on `Unit` values and live in `bovnar::units`.

```rust
use bovnar::{BaseUnit, Exponent, IECPrefix, Prefix, Unit};
use bovnar::units::{
    convert_value, exponent_to_int, int_to_exponent, prefix_unit_valid, unit_con-
vert_factor,
    unit_dimension_vector, unit_prefix_exponent, unit_prefix_factor, unit_reduce,
    unit_to_si_factor, unit_to_str_ex, unit_valid, units_compatible, UnitFlags,
};

fn main() -> Result<(), bovnar::Error> {
    let km = Unit::parse("k~m")?;
    let gib = Unit::parse("Gi~B")?;

    // Validate a Unit
    assert!(unit_valid(&km));

    // Prefix scale factor (SI or IEC) for a Unit
    assert_eq!(unit_prefix_factor(&km), 1000.0);
    assert_eq!(unit_prefix_factor(&gib), 2f64.powi(30));

    // Prefix exponent (base-10 for SI, base-2 for IEC)
    assert_eq!(unit_prefix_exponent(&km), 3);
    assert_eq!(unit_prefix_exponent(&gib), 30);

    // Validate a prefix for a base unit (IEC prefixes are only valid on bit/byte)
    let p = Prefix::Iec(IECPrefix::Gibi);
    assert!(prefix_unit_valid(p, BaseUnit::Byte));
    assert!(!prefix_unit_valid(p, BaseUnit::Meter));

    // Full SI conversion including affine terms (e.g. Celsius → Kelvin)
    let c = unit_to_si_factor(&Unit::parse("°C")?)?;
    assert!(c.is_affine);
    assert_eq!(c.affine_offset, 273.15);

    // Check convertibility: same SI dimension vector AND same quantity kinds
    assert!(units_compatible(&km, &Unit::parse("m")?));
    assert!(!units_compatible(&Unit::parse("b")?, &Unit::parse("B")?)); // the dimensions
agree, the kinds do not

    // Conversion factor between two compatible units
    let conv = unit_convert_factor(&km, &Unit::parse("m")?)?;
    assert_eq!((conv.factor, conv.requires_affine, conv.ok()), (1000.0, false, true));

    // 7-element SI dimension exponent vector [m, kg, s, A, K, mol, cd]
    assert_eq!(unit_dimension_vector(&Unit::parse("m/s")?)?, [1, 0, -1, 0, 0, 0, 0]);

    // Fold every prefix out and combine repeated bases. NOT the named-SI collapse:
    // the reduced unit of k~g·m/s² is "m·g/s²", and it is
    // unit_to_str_ex(.., UnitFlags::REDUCE) that gives "N". See the warning in §5.3
    // before applying r.scale to a value you are also going to serialise.
    let r = unit_reduce(&Unit::parse("k~g·m/s²")?)?;
    assert_eq!(r.unit.to_string(), "m·g/s²");
    assert_eq!(r.scale, 1000.0);

    // Convert a scalar value between units (handles affine conversions)
    let kelvin = convert_value(25.0, &Unit::parse("°C")?, &Unit::parse("K")?)?;
    assert!((kelvin - 298.15).abs() < 1e-9);

    // ... or EXACTLY, as the reader's want_unit does: (BigRational, exact),
    // exact false for an irrational factor (° to rad)
    let v = num_rational::BigRational::from_integer(42.into());
    let (ms, exact) =
        bovnar::units::convert_rational(&v, &Unit::parse("k~m/h")?, &Unit::parse("m/
s")?)?;
    assert_eq!(ms, num_rational::BigRational::new(35.into(), 3.into()));
    assert!(exact);
```

```
    // Serialise with formatting options (see UnitFlags below)
    let force = Unit::parse("k~g·m/s²")?;
    assert_eq!(unit_to_str_ex(&Unit::parse("m/s²")?, UnitFlags::ASCII_EXP)?, "m/s^2"); //
^N instead of superscripts
    assert_eq!(unit_to_str_ex(&force, UnitFlags::REDUCE)?, "N");                       //
reduce first — but see the warning below
    assert_eq!(unit_to_str_ex(&force, UnitFlags::REDUCE | UnitFlags::ASCII_EXP)?, "N");

    // Exponent ↔ integer conversions
    assert_eq!(exponent_to_int(Exponent::NEG_SQUARE), -2);
    assert_eq!(int_to_exponent(-2), Exponent::NEG_SQUARE);
    Ok(())
}
```

`SI_DIM_NAMES` is the array `["m", "kg", "s", "A", "K", "mol", "cd"]` — the index positions used by `unit_dimension_vector`. Both `SIConversion` (`factor`, `is_affine`, `affine_offset`) and `UnitConversion` (`factor`, `requires_affine`, and the method `ok()`, §13) are small `Copy` structs, and `ReducedUnit` is `{ unit, scale }`.

5.3 `UnitFlags`

```
use bovnar::units::UnitFlags;

fn main() {
    assert_eq!(UnitFlags::NONE, 0);           // no special formatting
    assert_eq!(UnitFlags::REDUCE, 1);         // reduce to a canonical named SI unit
before serialising
    assert_eq!(UnitFlags::ASCII_EXP, 2);      // use ^N exponent notation instead of
Unicode superscripts
    assert_eq!(UnitFlags::NUM_CANONICAL, 4);  // writer only: a float REDUCE rescaled
keeps float shape
}
```

`UnitFlags` is a namespace of `u32` constants, not an `IntFlag` type: OR them together and pass the `u32`. The same flags configure a **writer**: `Writer::to_mem_with(cap, pretty, UnitFlags::REDUCE, false)` rewrites each unit as it is written and rescales the value with it — `2.5 k~g` goes out as `<float:64,g> 2500`. `NUM_CANONICAL` is meaningful only there: it appends `.0` to a rescaled float that came out as an integer string, so the canonical document mode lands on one spelling.

> **`UnitFlags::REDUCE` returns the reduced unit, not a rescaled value.** Reduction folds every prefix out, so `unit_to_str_ex(&k~g, UnitFlags::REDUCE)` is `"g"` — a string denoting a quantity 1000× smaller than the unit you passed. Moving the value with it is your job. (The writer does it for you; nothing else does.)
>
> **Do not use `unit_reduce(&u)?.scale` for that.** It is the scale to the fully reduced unit, and this function does not always emit the fully reduced unit: where the reduction lands on a named SI unit the formatter re-attaches the prefix. `k~N` comes back `"k~N"` with nothing to rescale while `unit_reduce` still reports `1000.0`, and `k~g` comes back `"g"` where the `1000.0` must be applied. The two are indistinguishable from outside — both a lone unit carrying a kilo prefix — so take the scale from the unit that is actually emitted:

```rust use bovnar::Unit; use bovnar::units::{convert_value, unit_to_str_ex, UnitFlags}; fn main() -> Result<(), bovnar::Error> { let vu = Unit::parse("k~g")?; let mut value = 2.5; let emitted = Unit::parse(&unit_to_str_ex(&vu, UnitFlags::REDUCE)?)?; value *= convert_value(1.0, &vu, &emitted)?; // 1.0 for k~N, 1000.0 for k~g assert_eq!((emitted.to_string().as_str(), value), ("g", 2500.0)); Ok(()) } ```

The collapse never substitutes one named unit for another, so a `Sv` stays a `Sv` rather than becoming a `Gy`. A lone base at an exponent other than 1 does still collapse (`s⁻¹` → `"Hz"`, `m~s⁻¹` → `"k~Hz"`), and a reduction whose summed exponent leaves the `±100` range (`EXPONENT_MIN` / `EXPONENT_MAX`) fails with `Error::Argument` instead of returning a unit that lost a dimension — `m⁵⁰·m⁵⁰` reduces to `m¹⁰⁰`, `m⁵⁰·m⁵¹` fails.

5.4 `Prefix`

`Prefix` is one value for what the C struct `value_unit_prefix_t` splits into a system and an id: an enum with an SI and an IEC variant. It is what `UnitComponent::prefix` holds:

```rust
use bovnar::{IECPrefix, Prefix, SIPrefix, Unit};

fn main() -> Result<(), bovnar::Error> {
    let p_si = Prefix::Si(SIPrefix::Kilo);
    let p_iec = Prefix::Iec(IECPrefix::Gibi);
    assert!(!p_si.is_none() && !p_iec.is_none());
    assert_eq!(Prefix::default(), Prefix::Si(SIPrefix::None));

    let vu = Unit::parse("Gi~B")?;
    let comp = vu.components()[0];
    assert_eq!(comp.prefix, p_iec);             // the Prefix of a component
    assert_eq!(comp.prefix, IECPrefix::Gibi);   // compares equal to the bare IEC prefix
as well
    Ok(())
}
```

`Prefix::default()` is no prefix (`SIPrefix::None`), and an unprefixed component equals whatever its prefix system.

5.5 Inline unit suffix

In addition to the unit embedded in a type annotation (`<float:64,m/s>`), the Bovnar format supports an **inline unit suffix** placed directly after a scalar value, before the terminating `;`:

```
.speed = 9.81 m/s;           # inline suffix, no annotation
.mass  = <float:64> 70.5 k~g;# annotation without unit, inline adopted
.dist  = <float:64,m> 1.5 m; # annotation and inline both say 'm' — valid
```

From the Rust layer, inline and annotation units are **identical**: both reach the application as `d.value_unit` in the `Event::Data` payload. No extra code is required to consume inline units.
```

The validator fails with `ErrorCode::UnitMismatch` (38, an `Error::Parse`) when an inline suffix is present and a type-annotation unit is also present but the two do not resolve to the same unit. Inline unit suffixes inside array elements always fail with `ErrorCode::UnexpectedInputByte`.

5.6 `UnitPolicy` — validation and conversion without a callback

`Reader::set_unit_policy` states what the document must contain and what unit values should arrive in. Everything is unit **text**, so unlike `want_unit` it costs no per-value trampoline across the FFI boundary — the C side parses the strings once, when the policy is set.

```rust
use bovnar::{EventData, ReadOptions, Reader, UnitPolicy};

fn main() -> Result<(), bovnar::Error> {
    let mut r = Reader::new()?;
    r.set_unit_policy(Some(&UnitPolicy {
        targets: vec![("m/s".into(), 0), ("°C".into(), 0)], // convert to the first one
that fits
        normalise_si: true,    // ...and everything else to SI base units
        leave_inexact: true,   // step over what cannot be exact
        require_unit: true,    // reject any bare numeric value
        ..UnitPolicy::default()
    }))?;

    let mut delivered = Vec::new();
    let mut handler = |d: &EventData<'_>| {
        if d.converted {
            delivered.push(String::from_utf8_lossy(d.converted_text).into_owned());
        }
        true
    };
    r.read_mem(b".v = <float:64,k~m/h> 36.0;", &ReadOptions::default(), Some(&mut
handler), None)?;
    assert_eq!(delivered, ["10"]);
    Ok(())
}
```

A policy survives re-reading the same reader on another document — it describes the consumer, not the document — and `set_unit_policy(None)` clears it. Every unit string is parsed by the call, so a typo fails with `Error::Argument` before a byte is read, and a rejected policy leaves the previous one in force. The struct is plain data: `UnitPolicy { rules, targets, normalise_si, base, leave_inexact, require_unit, require_dimension_of }`, so `..UnitPolicy::default()` fills what a policy leaves out. At most `MAX_UNIT_TARGETS` (8) targets, `MAX_UNIT_RULES` (8) rules, and `MAX_UNIT_PATH` (96) bytes of path are accepted.

Conversion. Each numeric value takes the first target it can validly convert to, so order is significant: `["m", "k~m"]` never selects `k~m`. A value that matches nothing, or that is already in the unit a target names, arrives untouched — `d.converted` is what tells the two apart. Each target carries its own output base as the second tuple element: `("m".into(), 16)`; `0` is decimal.

`normalise_si` catches whatever the targets did not, delivering it in coherent SI base units with prefixes folded out. Currencies and every **dimensionless** unit (`%`, `ppm`, `dB`, `pH`, `rad`, `°`) are left as written rather than reinterpreted.

A value with no unit only ever matches a target that is itself `no_unit`. A bare number is dimensionally compatible with `%` and `ppm`, so without that fence a policy naming `"%"` would deliver `0.25` as `25`.

Exactness. Nothing approximate is ever delivered. `leave_inexact: true` hands over a value the conversion cannot deliver exactly — an irrational factor, a non-terminating expansion (`42 km/h` is `35/3 m/s`), an exponent past the work limit — in its native unit instead of failing. It applies only to policy-chosen targets; a `want_unit` hook keeps its strict all-or-nothing contract.

Validation. `require_unit` rejects a document containing any bare numeric value; `require_dimension_of: vec!["m".into()]` requires every numeric value to be validly convertible to at least one listed unit. Both are evaluated on the unit the **document** wrote, before any conversion, and both fail with `ErrorCode::UnitMismatch`.

Per-field rules name one field by its key path, and are consulted before everything else:

```rust
use bovnar::{UnitPolicy, UnitRule};

fn main() {
    let policy = UnitPolicy {
        rules: vec![
            UnitRule::new(".inlet.temperature", "°C"), // convert this field
            UnitRule { convert: false, ..UnitRule::new(".inlet.*", "m") }, // assert the
subtree
        ],
        ..UnitPolicy::default()
    };
    assert_eq!(policy.rules.len(), 2);
}
```

A path ending in `".*"` names a subtree at any depth, and a prefix only matches at a component boundary (`".in.*"` does not claim `".inlet.a"`). Unlike a whole-document target, a rule is an **assertion**: a value it cannot be applied to fails with `UnitMismatch`, because silence would defeat the point of naming the field. `convert: false` asserts without converting.

The DOM takes the same policy, so a random-access consumer gets both halves:

```rust
use bovnar::{DomDoc, UnitPolicy, UnitRule};

fn main() -> Result<(), bovnar::Error> {
    let text = ".inlet = { .temperature = <float:64,°F> 212.0; };".as_bytes();
    let policy = UnitPolicy {
        rules: vec![UnitRule::new(".inlet.temperature", "°C")],
        ..UnitPolicy::default()
    };
    let doc = DomDoc::parse_with_policy(text, &policy)?;
    let t = doc.lookup("inlet.temperature").expect("present");
    assert_eq!(t.as_float(), Some(100.0)); // from a document in °F
    Ok(())
}
```

A value the policy converted is STORED converted — digits, unit and base — and an integer that converts to a fraction (5 g in kilograms is 0.005) is stored as a float. A policy the library refuses is an `Error::Argument` — the same error, naming the same entry, as `Reader::set_unit_policy` — before anything is parsed, rather than a parse error, so a mistake in the policy is never mistaken for a fault in the document.

The writer takes the same policy, validation half only. A reader can only reject a document somebody already wrote; `Writer::set_unit_policy` is what stops a bare number reaching the file:

```rust
use bovnar::{BaseUnit, Exponent, Quantity, SIPrefix, Unit, UnitPolicy, ValueTypeFamily,
ValueTypeSpec, Writer};

fn main() -> Result<(), bovnar::Error> {
    let mut w = Writer::to_mem(4096, true)?;
    w.set_unit_policy(Some(&UnitPolicy {
        require_unit: true,
        require_dimension_of: vec!["m".into()],
        ..UnitPolicy::default()
    }))?;

    let float64 = ValueTypeSpec { family: ValueTypeFamily::Float as i32, width: 64, base:
0 };
    let inches = Unit::parse("in")?;
    w.write_quantity("tank_level", &Quantity::new("12.0", float64, inches))?; // a length
— written
    let err = w.write_quantity("spare", &Quantity::new("0.25", float64,
Unit::default())).unwrap_err();
    assert!(matches!(err, bovnar::Error::Write { .. })); // code UnitMismatch
    let _ = (BaseUnit::Meter, Exponent::LINEAR, SIPrefix::Kilo); // see §5.7 for building
units
    Ok(())
}
```

A policy carrying `targets`, `normalise_si`, `base` or `leave_inexact` fails with `Error::Argument` there rather than being half-honoured: write-side unit rewriting is `UnitFlags::REDUCE`'s job — open the writer with `Writer::to_mem_with(cap, pretty, UnitFlags::REDUCE, false)` — and two rewriting modes with different rules about exactness is how a document ends up meaning two things.

Module-level helpers back the same decisions outside the reader:

```rust
use bovnar::Unit;
use bovnar::units::{unit_si_normal_form, units_compatible, units_convertible};

fn main() -> Result<(), bovnar::Error> {
    let (kusd, usd) = (Unit::parse("k~$USD")?, Unit::parse("$USD")?);
    // units_compatible is the wrong screen for a conversion target: a currency
    // carries no dimension, so it is reported incompatible with itself.
    assert!(!units_compatible(&kusd, &usd));
    assert!(units_convertible(&kusd, &usd));

    assert_eq!(unit_si_normal_form(&Unit::parse("in")?), Some(Unit::parse("m")?));
    assert_eq!(unit_si_normal_form(&Unit::parse("g")?),
Some(Unit::parse("k~g")?)); // the SI base unit for mass
    assert_eq!(unit_si_normal_form(&Unit::parse("%")?), None);                    // di-
mensionless
    assert_eq!(unit_si_normal_form(&Unit::parse("lm")?), None);                   //
carries the steradian's kind
    Ok(())
}
```

Three questions, in decreasing strictness. `units_equal` asks whether the two are the same unit, `units_compatible` whether their dimensions and quantity kinds line up, `units_convertible` whether a conversion can actually be performed. Only the last is the right screen for a conversion target.

```rust
use bovnar::Unit;
use bovnar::units::units_equal;

fn main() -> Result<(), bovnar::Error> {
    let p = Unit::parse;
    assert!(units_equal(&p("m")?, &p("m")?));
    assert!(!units_equal(&p("m")?, &p("k~m")?));             // convertible, not equal
    assert!(!units_equal(&p("N")?, &p("k~g·m/s²")?));        // same SI form
    assert!(units_equal(&p("mmHg")?, &p("ucum:mm[Hg]")?));
    Ok(())
}
```

`units_equal` is also what `==` means between two `Unit`s, with one addition: "no unit at all" (a typed value's empty unit) and a dimensionless unit (a bare literal's) are both the absence of one and compare, and hash, equal, as the Python binding's do. `Unit` is `Eq` and `Hash` with the two agreeing — equal units are one `HashMap` key regardless of the order their components were written in. The hash follows the equality in the one place the two could part: a component with **no prefix** is the same component whether its prefix system says SI or IEC, so `Unit::parse("B")` and `Unit::iec(BaseUnit::Byte, IECPrefix::None, Exponent::LINEAR)` are equal and one key.

A `Unit` also compares equal to a **string** that parses to it (`unit == "k~m"`, via `PartialEq<str>` and `PartialEq<&str>`); a string that does not parse is simply unequal. That is a convenience for comparison only and is **not hash-compatible** — `"km"`, `"k~m"` and `"ucum:km"` are three strings with three hashes and one unit. Key a map or set on `Unit` values (or on `unit.to_string()`), never on a mix of the two.

`Unit::convert_factor(&other)` and `Unit::factor()` return a bare number, so they **fail** with `Error::Argument` where there is no such number: an affine pair (`°C` → `K`) has no multiplic-

ative factor and an affine unit no SI factor. Use `convert_value()` there, or `unit_to_si_factor()` for the scale and the offset.

5.7 Building a `Unit`

`Unit::parse` builds a unit from text. To build one programmatically — without going through the notation — use the constructors on `Unit`:

```rust
use bovnar::{BaseUnit, Exponent, IECPrefix, SIPrefix, Unit, UnitComponent};

fn main() -> Result<(), bovnar::Error> {
    let kg = Unit::si(BaseUnit::Gram, SIPrefix::Kilo, Exponent::LINEAR);
    assert_eq!(kg.to_string(), "k~g");
    let m2 = Unit::si(BaseUnit::Meter, SIPrefix::None, Exponent::SQUARE);
    assert_eq!(m2.to_string(), "m²");
    let gib = Unit::iec(BaseUnit::Byte, IECPrefix::Gibi, Exponent::LINEAR);
    assert_eq!(gib.to_string(), "Gi~B");

    let newton = Unit::compound(&[
        UnitComponent::si(BaseUnit::Gram, SIPrefix::Kilo, Exponent::LINEAR),
        UnitComponent::si(BaseUnit::Meter, SIPrefix::None, Exponent::LINEAR),
        UnitComponent::si(BaseUnit::Second, SIPrefix::None, Exponent::NEG_SQUARE),
    ])?;
    assert_eq!(newton.to_string(), "k~g·m/s²");
    Ok(())
}
```

The C constructors `make_unit_si`, `make_unit_iec` and `make_unit_compound` are `Unit::si`, `Unit::iec` and `Unit::compound`; the prefix and exponent are always passed (`SIPrefix::None`, `Exponent::LINEAR` for a plain base unit), and `Unit::compound` fails with `Error::Argument` beyond `MAX_UNIT_COMPONENTS` (32) components.

`Unit::default()` and `Unit::dimensionless()` are not the same thing, even though both render as `no_unit`. The first (the C `make_unit_none()`) is a unit with no components at all — a value carrying no unit. The second (the C `make_unit_dimensionless()`) is one component whose base is `BaseUnit::None` — a value carrying `no_unit` explicitly. That is exactly the pair `UnitPolicy { require_unit: true, .. }` refuses together (§5.6):

```rust
use bovnar::Unit;
use bovnar::units::unit_to_str;

fn main() -> Result<(), bovnar::Error> {
    let (none, dimensionless) = (Unit::default(), Unit::dimensionless());
    assert_eq!(unit_to_str(&none)?, "no_unit");
    assert_eq!(unit_to_str(&dimensionless)?, "no_unit");
    assert_eq!((none.num_components(), dimensionless.num_components()), (0, 1));
    assert!(Unit::is_dimensionless(&none) && Unit::is_dimensionless(&dimensionless));
    // Display, unlike unit_to_str, spells a unit-less value as nothing at all
    assert_eq!(none.to_string(), "");
    Ok(())
}
```

`Unit::is_dimensionless()` is true for both of them and for any unit made only of `none` components; `%` and `ppm` are real units and answer false.

5.8 Validators and literal helpers

`bovnar::literals` binds the C library's checks and literal helpers (`bovnar.h`): the tests the writer applies to keys, symbols, references and numbers, the integer and float parsers and formatters behind it, and the parser of a type annotation. Each function is the C function of the same name without its `bvn_` prefix (`escape_repl_table` is `bvn_get_escape_repl_table`), and answers as it does. A string holding NUL is none of the things they check: a validator answers `false` for it, a parser fails — except `validate_string`, which takes bytes and checks only that they are UTF-8, so NUL passes.

Type annotations. `parse_type_annotation(text)` parses the text between the angle brackets, as the reader does, into a `TypeAnnotation` of `value_type` (a `ValueTypeSpec`), `unit` (a `Unit`, `Unit::default()` when there is no unit or it is spelled `no_unit`) and `unit_source` (the unit parameter as written, `""` when there is none). The struct is `#[non_exhaustive]`, so that a field can be added to it without breaking anyone.

```
use bovnar::literals::parse_type_annotation;
use bovnar::{ErrorCode, ValueTypeFamily, ValueTypeSpec};

fn main() -> Result<(), bovnar::Error> {
    let a = parse_type_annotation("uint:32,_16,k~g")?;
    let spec = ValueTypeSpec { family: ValueTypeFamily::Uint as i32, width: 32, base:
16 };
    assert_eq!(a.value_type, spec);
    assert_eq!(a.unit.to_string(), "k~g");
    assert_eq!(a.unit_source, "k~g");
    let ml = parse_type_annotation("float:64,ucum:mL{total}")?;
    assert_eq!((ml.unit.to_string(), ml.unit_source.as_str()), ("m~L".into(),
"ucum:mL{total}"));

    // A base given twice; a unit parameter over 255 bytes.
    let e = parse_type_annotation("uint:8,_16,_16").unwrap_err();
    assert_eq!(e.code(), Some(ErrorCode::IllegalValueType));
    let e = parse_type_annotation(&format!("uint:8,{}", "m".repeat(256))).unwrap_err();
    assert_eq!(e.code(), Some(ErrorCode::UnitTooLong));
    Ok(())
}
```

A refusal is an `Error::Parse` (with no position) whose code is the one the reader gives the same annotation in a document, so it tells the C function's three flags apart:

Code	C flags	Cause
`IllegalValueType`	`type_ok` false	An unknown family, an empty or repeated parameter, a parameter the family does not take (a base on `float_fix`, a width on `bool`), a width or base out of range
`UnitTooLong`	`unit_ok` false, `unit_too_long` true	A unit parameter over 255 bytes

Code	C flags	Cause
`UnitIllegal`, `UnitProfileUnknown`, `UnitProfileUnsupported`	`unit_ok` false	A unit parameter that is not a unit this build knows; the code is the one `units::unit_error_code` gives it

The reader's version gates are not applied: `datetime` (spec 1.1) and a profile unit (spec 1.2) parse here whatever version a document declares. A `datetime`'s epoch is no unit: its index is in `value_type.base` (`datetime_epoch_index`, §18), as a `float_fix`'s q is.

Validators. Each returns a `bool`:

Function	`true` for
`validate_identifier(s)`	A key: an ASCII letter, `_` or a character from U+00C0 on, then no space, control character, U+0080 to U+00BF, or punctuation spec §4.2 forbids (`-` and `+` are allowed). No length limit: the reader's `max_identifier_length` (255) is separate
`validate_symbol(s)`	A symbol, which is spelled as a key is
`validate_reference(s)`	A path such as `.a.b[0]`, without the `&`: identifier segments, each with optional `[digits]` indexes
`validate_number(s)`	A decimal number: an optional `-`, digits with at most one `.`, an optional `e` / `E` exponent. No `+`, and not `inf`
`is_special_number_string(s)`	Exactly `inf`, `ninf` or `nan`
`validate_digits_for_base(s, base)`	An integer literal: an optional `-` and digits of `base` (no sign at all in bases 64 and 85, where `+` and `-` are digits)
`validate_number_in_base(s, base)`	A quoted float literal: either sign, digits of `base` with at most one `.`, and an exponent marked `e` (bases up to 14) or `p` (the power-of-two bases up to 16)
`validate_uint_range(s, width, base)` / `validate_sint_range(s, width, base)`	A literal that fits `width` bits (0: any width); `inf`, `ninf` and `nan` pass
`validate_string(bytes)`	Bytes that are well-formed UTF-8

Two of them are narrower than their names. `validate_digits_for_base` does not check the base against the ones a type may declare: in base 1 the one digit is `0`. And `validate_string` is the same test as `std::str::from_utf8`; NUL and the other control bytes the writer refuses in a string pass it. The range checks are meant for text the digit check has passed: they refuse an empty literal, and in base 10 anything but digits after an optional sign (`validate_uint_range(" 5", 8, 10)` is `false`), but they take a leading `+`.

Digits. `char_to_digit(ch, base)` is the value of one digit, `None` when it is no digit of `base`: up to base 36 a letter's case does not matter, above it `A` to `Z` are 36 to 61, and bases 64 and 85 use the Base64 and Ascii85 alphabets. `min_digits_for_type(vt)` is how many digits

of the type's base the largest value of an integer type of up to 64 bits takes; 0 for a width of 0 or above 64, and a float type is answered as if it were an unsigned integer of its width.

Parsing and formatting. These are the functions the writer and the DOM use. Each returns `Err(Error::Argument)` where the C one returns `false` or -1:

Function	Result
`parse_int64(s, vt)` / `parse_uint64(s, vt)`	The integer literal in the type's base; refused when it does not fit the type's width (64 when it declares none), and for a `+` or a space before it
`parse_double(s, vt)` / `parse_double_in_base(s, base)`	The float literal (base 10 or 16; `inf`, `ninf` and `nan`, not `-inf`) as the nearest `f64`, rounded once from its exact value
`format_uint64(value, base, min_digits)` / `format_int64(...)`	The digits in base 2 to 62, 64 or 85, padded with zero digits to `min_digits` (which the C function caps at 64). A negative value has no form in bases 64 and 85
`format_double(value, vt)`	The literal the writer emits under `vt`: the shortest digits that read back to the same `f64`, in base 10 or 16 (any other base fails). A `float:16` or `float:32` value is first rounded to IEEE binary16 or binary32 (`2049.0` is `2.048e+3` under `float:16`) and printed with the digits of a 16- or 32-bit significand, so it reads back as that value without rounding to the format (`65504.0` is `6.5504e+4`); a finite value the format cannot hold fails
`escape_repl_table()`	A `&'static [u8; 256]`: for each byte, the byte the escape `\` + that byte stands for, 0 for none. The escapes are `t`, `n`, `v`, `f`, `r`, `"` and `\`; the `\x` and `\u` escapes of spec 1.1 take digits and are not in it

```rust
use bovnar::literals;
use bovnar::{ValueTypeFamily, ValueTypeSpec};

fn main() -> Result<(), bovnar::Error> {
    assert!(literals::validate_identifier("max-speed"));
    assert!(!literals::validate_identifier("max.speed"));
    assert!(literals::validate_reference(".axis[2].unit"));
    assert!(!literals::validate_number("+1"));
    assert!(!literals::validate_uint_range("256", 8, 10));
    assert_eq!(literals::char_to_digit('Z', 62), Some(61));

    let u8t = ValueTypeSpec { family: ValueTypeFamily::Uint as i32, width: 8, base: 0 };
    let f64t = ValueTypeSpec { family: ValueTypeFamily::Float as i32, width: 64, base:
0 };
    assert_eq!(literals::parse_uint64("255", u8t)?, 255);
    assert!(literals::parse_uint64("256", u8t).is_err());
    assert_eq!(literals::format_double(0.1, f64t)?, "1e-1");
    assert_eq!(literals::parse_double("1e-1", f64t)?, 0.1);
    assert_eq!(literals::format_uint64(5, 2, 8)?, "00000101");
    assert_eq!(literals::escape_repl_table()[usize::from(b't')], b'\t');
    Ok(())
}
```

> **No `0x` in base 16.** The integer parsers take what the reader takes, so `parse_uint64("0x10", …)` under `uint:64,_16` fails, though `strtoull`, which they read with, would skip the prefix.

`bvn_looks_like_double` is not bound. It answers whether the text contains one of `.`, `e`, `E`, `p` and `P`, so `deadbeef` looks like a float to it and `inf` does not: a shortcut for the writer's own output, not a test of text from anywhere else.

---

6. `Quantity`

`Quantity` is a typed, unit-annotated scalar value that preserves the original text representation, type width, numeral base, and physical unit across a `parse_typed` / `to_string` round trip.

6.1 Construction

`Quantity` is normally produced by `parse_typed` rather than constructed by hand, but direct construction is supported. Its fields are public. The type is a `ValueTypeSpec` — a plain struct of the family (a `ValueTypeFamily` as its C integer), the bit width and the numeral base:

```rust
use bovnar::{Quantity, Unit, ValueTypeFamily, ValueTypeSpec};

fn main() -> Result<(), bovnar::Error> {
    let vt = ValueTypeSpec { family: ValueTypeFamily::Float as i32, width: 32, base: 0 };
    let q = Quantity::new("101325.0", vt, Unit::default());       // dimensionless
    let q2 = Quantity::new("9.81", vt, Unit::parse("m/s²")?);     // with a unit
    assert_eq!(q.unit_str(), "");
    assert_eq!(q2.unit_str(), "m/s²");
    Ok(())
}
```

For exact numeric input there is also `Quantity::from_number`, which stores the value as an exact decimal literal (so writing it with `to_string` is lossless to the format's precision):

```rust
use bovnar::{Decimal, Quantity, Unit, ValueTypeFamily};
use num_rational::BigRational;

fn main() -> Result<(), bovnar::Error> {
    let price: Decimal = "19.99".parse()?;
    Quantity::from_number(price, ValueTypeFamily::FloatDec, 64, None, Unit::default())?;

    let one_point_one: Decimal = "1.1".parse()?;
    Quantity::from_number(one_point_one, ValueTypeFamily::Float, 128, None,
Unit::default())?;

    // A float_fix takes its fractional bits: 837/256 is 3.26953125
    let ratio = BigRational::new(837.into(), 256.into());
    Quantity::from_number(ratio, ValueTypeFamily::FloatFix, 32, Some(8),
Unit::default())?;
    Ok(())
}
```

It takes an `impl Into<ExactNumber>`: a literal (`&str` or `String`, used verbatim), a `Decimal`, a terminating `BigRational`, a `BigInt`, or an `f64` (only as precise as the double). It validates the width for the chosen family. For `float_dec` it also refuses a finite value past the width's range — `1e400` does not fit `float_dec:64` and would materialise to infinity — and a non-finite `f64` is stored under the format's own spelling (`inf`, `ninf`, `nan`). A `float_fix` needs its `frac` (`Some(q)`).

6.2 Fields and methods

`Quantity` is a plain struct: the properties that merely hold a value are public fields and the others are methods:

Name	Type	Description
`q.raw`	`Option<String>`	Original text token as it appeared in the BVNR stream; `None` for a typed null
`q.vtype`	`ValueTypeSpec`	Type family, bit width, and numeral base
`q.unit`	`Unit`	Physical unit; `q.unit.to_string()` is its canonical text
`q.token_type`	`TokenType`	How the literal was written: number, quoted string, symbol, ...
`q.frac`	`Option<String>`	For a `datetime` written as a literal with a fractional second, the verbatim sub-second digits (spec 1.1)
`q.unit_source`	`Option<String>`	The unit exactly as the document spelled it (library 2.0) — `ucum:mL{total}` where `q.unit` is `m~L`. Set by `parse_typed`; `to_string()` writes it back, and refuses one that does not parse to `q.unit` — including a spelling on a `Quantity` that has no unit at all. `None` writes the canonical form
`value()`	`Value`	Decode `raw` to the closest plain value (`Int`, `Float`, `String`, `Bool`) — **lossy** for `float_dec` / `float_fix` / `float:128`+ (goes through a C `double`)

Name	Type	Description
`q.family()`	`Option<ValueTypeFamily>`	The type family, when it is one this binding knows
`unit_str()`	`String`	Canonical unit string (e.g. `"m/s²"`), or `""` when dimensionless
`decimal()`	`Result<Option<Decimal>>`	**Exact** value as a `Decimal` from the verbatim literal — lossless at any width; an error for non-numeric families, `None` for a typed null
`fraction()`	`Result<Option<BigRational>>`	Exact value as a rational (for `float_fix`, the exact `mantissa / 2**frac`). An error for `nan`/`inf`, and for a literal whose decimal exponent is past `bovnar::FRACTION_EXPONENT_CAP` (§6.3)
`fixed_point()`	`Result<(BigInt, u32)>`	`(mantissa, frac_bits)` of a `float_fix` value; the value is `mantissa / 2**frac_bits`. An error for a typed null and for `nan`/`inf`/`ninf`, which a fixed-point datum has no encoding for
`stored_value()`	`Result<Option<Decimal>>`	The value materialised into the declared IEEE/fixed format (round-to-nearest-even) — differs from `decimal` only when the literal carries more precision than the format holds. `None` for a typed null, as `decimal` is
`ieee_bits()`	`Result<Vec<u8>>`	IEEE-754 interchange bytes (binary16…256 for `float`, decimal16…256 for `float_dec`), little-endian word order
`epoch_name()`	`Option<String>`	For a `datetime`, the epoch name (`"unix"`, `"tai"`, …); `None` otherwise
`epoch_mjd()`	`Option<i32>`	For a `datetime`, the epoch's Modified Julian Day; `None` otherwise
`to_civil()`	`Result<CivilDateTime>`	For a `datetime`, its UTC date and time (§6.5); an error otherwise
`type_key()`	`(i32, u32, u32)`	`(family, width, base)` with the defaults filled in: width 0 is 64, base 0 is 10; a `float_fix` Q and a `datetime` epoch as stored

Two `Quantity`s are equal when they carry the same literal, sub-second digits, effective type (`type_key()`) and unit: `<float:0> 2.5`, `<float:64,_10> 2.5` and a bare `2.5` are one value, and the token form and unit spelling are presentation — the same rule as the Python binding's `Quantity.__eq__`.

6.3 Exact numbers (`Decimal`, `BigRational`, `BvnFloat`)

`value()` decodes through a C `double`, which loses precision for the decimal-float, fixed-point, and wide binary-float families. The accessors above instead use the verbatim literal text (and bovnar's arbitrary-precision `bvn_float`), so the full precision the format carries is reachable from Rust.

An integer is `i128` in a `Value` (`Value::BigInt` past it) and `num_bigint::BigInt` where a number has no width limit, a rational is `num_rational::BigRational`, and a decimal is **`bovnar::Decimal`**: an exact decimal — sign, coefficient (`BigUint`) and exponent, or an infinity or a NaN — that compares numerically and prints in the General Decimal Arithmetic scientific-string form (`1.50` stays `1.50`). `Decimal` is the exact value only; it does no arithmetic.

```
use bovnar::{Decimal, Value, parse_typed};

fn main() -> Result<(), bovnar::Error> {
    let doc = parse_typed(b".p=<float_dec:64> 3.14159265358979323846264338327950 3;")?;
    let Value::Struct(fields) = doc else { unreachable!() };
    let Value::Quantity(p) = &fields[0].1 else { unreachable!() };

    // Quantity::value is lossy: a C double
    assert_eq!(p.value(), Value::Float(3.141592653589793));
    // the literal, every digit of it ...
    let literal: Decimal = "3.14159265358979323846264338327950 3".parse()?;
    assert_eq!(p.decimal()?, Some(literal));
    // ... and the value decimal64 stores: 16 digits, rounded half to even
    assert_eq!(p.stored_value()?.map(|d| d.to_string()),
Some("3.141592653589793".into()));
    assert_eq!(p.ieee_bits()?.len(), 8);

    let doc = parse_typed(b".x=<float_fix:32,q8> 3.27;")?;
    let Value::Struct(fields) = doc else { unreachable!() };
    let Value::Quantity(f) = &fields[0].1 else { unreachable!() };
    assert_eq!(f.fraction()?, Some(num_rational::BigRational::new(837.into(),
256.into())));
    assert_eq!(f.fixed_point()?, (837.into(), 8));
    Ok(())
}
```

These materialise over the **full** representable range, including exponents the C parser cannot otherwise reach (beyond ~1e9865). Overflow and underflow past the format's range are decided from the decimal exponent, so a 28-byte `<float:64> 1e-99999999` is `0.0` at once rather than an exact rational with a hundred-million-digit denominator. `fraction()` has no format to round into: it refuses a literal whose decimal exponent, of its magnitude or of its last digit, is past **`bovnar::FRACTION_EXPONENT_CAP`** (100 000; `float_dec:256`, which reaches $10^{-611867}$, gets 611 937), the same cap as the Python binding's. `decimal()` is exact at any exponent. `decimal` / `fraction` work at every binary width bovnar allows (16, or a multiple of 32 up to 32768); the bit-exact `stored_value` / `ieee_bits` apply to the IEEE encodings (`float:16/32/64/128/256`), which are also exposed directly as `bovnar::BvnFloat`.

`BvnFloat` and the `bvnfloat` module reach the library's arbitrary-precision float: the IEEE binary and decimal interchange encodings and the fixed-point datum, plus the exact materialisers used past the C parser's exponent cap.

```rust
use bovnar::BvnFloat;

fn main() -> Result<(), bovnar::Error> {
    let f = BvnFloat::from_str("0.1", 10, 256)?;     // text, base, precision in bits
    assert!(!f.is_nan() && !f.is_inf() && !f.is_zero());
    assert_eq!(f.to_double(), 0.1);
    let bits = f.to_binary_bits(32)?;                // binary32, little-endian
    assert_eq!(bits.len(), 4);
    assert_eq!(BvnFloat::from_binary_bits(32, &bits)?.to_double(), 0.1f32 as f64);
    Ok(())
}
```

Exactness. The `stored_value` of a `float_fix` is exact, never rounded to a working precision, however wide the datum; the same goes for the literal of a terminating `Fraction`.

6.4 `to_string()` integration

`to_string` dispatches on `Value::Quantity` before the plain `Int` / `Float` path, so any value that came from `parse_typed` can be passed directly to `to_string` and will produce identical output:

```rust
use bovnar::{parse_typed, to_string};

fn main() -> Result<(), bovnar::Error> {
    let bvnr = b".speed = <float:32,m/s> 9.81;";
    let doc = parse_typed(bvnr)?;
    let out = to_string(&doc, false)?;
    assert_eq!(out, b".speed=<float:32,m/s>9.81;");
    assert_eq!(parse_typed(&out)?, doc);
    Ok(())
}
```

The annotation is left out where it says nothing the literal does not — i.e. when the type is `float:64` with no unit and no non-decimal base (matching the C library's own default-annotation omission rules).

`to_string` also accepts a bare `Value::Decimal` (written as an exact `float_dec:64` literal) and a terminating `Value::Fraction` — a non-terminating fraction (e.g. `1/3`) fails with `Error::Argument`, as does a `Decimal` outside the range of `float_dec:64`. An array whose elements are all `Decimal` / `Fraction` is written under one `<float_dec:64>` annotation, so the elements mean what the scalars mean:

```rust
use bovnar::{Value, to_string};
use num_rational::BigRational;

fn main() -> Result<(), bovnar::Error> {
    let half = BigRational::new(1.into(), 2.into());
    let doc = Value::Struct(vec![("h".into(), Value::Fraction(half))]);
    assert_eq!(to_string(&doc, false)?, b".h=<float_dec:64>0.5;");

    let third = BigRational::new(1.into(), 3.into());
    let bad = Value::Struct(vec![("t".into(), Value::Fraction(third))]);
    assert!(matches!(to_string(&bad, false), Err(bovnar::Error::Argument { .. })));
    Ok(())
}
```

Arrays of `Quantity`. An array has one annotation. Elements that agree on type and unit are written under it. Elements that agree on the unit but not the type — or that carry no unit and differ in type, `[<uint:8> 1, <float:64> 2.5]` — are each written under their own annotation, as they were read. Elements that carry different units have no valid form and fail.

Octet streams. `parse()` returns an octet stream as `Value::Bytes`, and `to_string()` writes `Value::Bytes` back as one, so a document holding one round-trips.

What "can be passed directly to `to_string()`" does not promise is that the result passes the document tier: `to_string()` writes what `parse()` reads, and `parse()` reads heterogeneous and ragged arrays that `DomDoc::parse` and `bovnar validate` refuse. See the note in §4.1.

6.5 Civil time and calendars

A `datetime` value is a count of seconds since its epoch (§18). `bovnar::timekeeping` turns such counts into civil dates and times and back, and carries the rest of the C library's calendar: `bvn_datetime.h` and `bvn_gregorian_date.h`. Every result is computed by the C functions, so the crate gives the answers the reader and writer use when they read and print an ISO literal. Nothing is read from the host's time zone or clock, so the module behaves the same on every target, `wasm32-wasip2` included.

```
use bovnar::timekeeping::{self, CivilDateTime, Epoch, GregorianDate};
use bovnar::{Value, parse_typed};

fn main() -> Result<(), bovnar::Error> {
    let doc = parse_typed(b"#!bovnar 1.1\n.t = <datetime:64,tai> 1861920036;")?;
    let Value::Struct(fields) = doc else { unreachable!() };
    let Value::Quantity(t) = &fields[0].1 else { unreachable!() };
    assert_eq!(t.to_civil()?.to_string(), "2016-12-31T23:59:60");

    let leap = CivilDateTime::new(2016, 12, 31, 23, 59, 60)?;
    assert_eq!(timekeeping::utc_to_tai_seconds(&leap)?, 1_861_920_036);
    let next = timekeeping::tai_seconds_to_utc(1_861_920_037)?;
    assert_eq!(next.to_string(), "2017-01-01T00:00:00");
    let unix = timekeeping::epoch_seconds_to_utc(1_750_000_000, Epoch::Unix)?;
    assert_eq!(unix.to_string(), "2025-06-15T15:06:40");

    // Germany: UTC+1, one hour more in summer
    let t = timekeeping::utc_to_tai_seconds(&CivilDateTime::new(2024, 7, 1, 12, 0, 0)?)?;
    let dst = timekeeping::tai_second_is_dls_in_europe(t);
    let local = timekeeping::tai_seconds_to_local_time(t, 3600, dst)?;
    assert_eq!(local.to_string(), "2024-07-01T14:00:00");

    assert_eq!(GregorianDate::new(2000, 1, 1)?.to_mjd(), 51544);
    assert!(GregorianDate::new(2024, 2, 30).is_err());
    assert_eq!(GregorianDate::normalize(2024, 2, 30)?.to_string(), "2024-03-01");
    assert_eq!(timekeeping::easter_sunday(2025)?.to_string(), "2025-04-20");
    let good_friday = GregorianDate::new(2024, 3, 29)?;
    assert!(good_friday.is_federal_holiday_in_germany());
    let march = GregorianDate::new(2024, 3, 5)?;
    assert_eq!(march.last_banking_day_in_month()?.to_string(), "2024-03-28");
    Ok(())
}
```

The types. `GregorianDate` is a date of the proleptic Gregorian calendar (`bvn_gregorian_date_t`), with astronomical years (1 BC is year 0) from `YEAR_MIN` to `YEAR_MAX`, about ±2.5·10¹⁶. `GregorianDate::new(year, month, day)` takes only a real date; `GregorianDate::normalize(year, month, day)` takes any `i64` fields and carries an out-of-range month or day into the next field, as `mktime` does. Its accessors are `year()`, `month()`, `day()`, `leap()`, `wday()` (0 = Sunday) and `yday()` (0 = 1 January), the last three from the library. `CivilDateTime::new(year, month, day, hour, minute, second)` adds a time of day (`bvn_datetime_t`) and checks hour 0–23, minute 0–59 and second 0–60; `date()` is its `GregorianDate` and `mjd()` the date's Modified Julian Day. Both are `Copy`, `Eq` (through the C equality functions), `Hash` and `Ord` (chronological), and `Display` as ISO 8601 (`2016-12-31T23:59:60`). `Epoch` is `bvn_epoch_t`: each discriminant is its epoch's MJD (`Epoch::mjd()`), `Epoch::from_mjd` of a datetime `Quantity`'s `epoch_mjd()` is its epoch, and `Epoch::from_name` resolves the library's names (`"unix"`, `"y2000"`, …).

Seconds and epochs. The functions are those of the Python binding (doc/09 §6.5), under the same names and with the arguments in the same order. Two things differ: an epoch is an `Epoch` here, where Python also takes its MJD or its name, and `is_last_banking_day_in_quarter` / `is_first_banking_day_in_quarter` return a `Result<bool>`, an `Error::Argument` for a quarter outside 1–4, where Python raises:

Function	C function	Result
`epoch_seconds_to_datetime(seconds, epoch)`	`bvn_dt_epoch_seconds_to_datetime`	The civil time `s` after `epoch` on uniform scale (8 day, no leap sec every `i64`, so r `ult`
`datetime_to_epoch_seconds(&dt, epoch)`	`bvn_dt_datetime_to_epoch_seconds`	Its inverse; `sec` `60` counts as th minute's first se
`convert_epoch_seconds(seconds, from, to)`	`bvn_dt_convert_epoch_seconds`	The same count belled to anothe the time scale d change
`utc_to_tai_seconds(&dt)`	`bvn_dt_utc_to_tai_seconds`	UTC to TAI secon the leap-second (10 s before 197 since 2017)
`tai_seconds_to_utc(tai_seconds)`	`bvn_dt_tai_seconds_to_utc`	Its inverse; an in leap second is `23:59:60`

Function	C function	Result
`tai_seconds_to_local_time(tai_seconds, utc_offset_seconds, dls_active)`	`bvn_dt_tai_seconds_to_local_time`	Wall-clock time a offset, one hour when `dls_acti` offset must be w minutes
`tai_second_is_dls_in_europe(tai_second)`	`bvn_dt_tai_second_is_dls_in_europe`	Whether Germa European summ is in force (none 1980)
`epoch_seconds_from_gps_time(time_of_week_ms, week, epoch)`, `tai_seconds_from_gps_time(time_of_week_ms, week)`	`bvn_dt_epoch_seconds_from_gps_time`, `bvn_dt_tai_seconds_from_gps_time`	GPS week + tim week (ms, round the second) as s GPS time except `Epoch::Tai`, w 19 s are added
`epoch_seconds_from_galileo_time(…)`, `tai_seconds_from_galileo_time(…)`	`bvn_dt_epoch_seconds_from_galileo_time`, `bvn_dt_tai_seconds_from_galileo_time`	The same for Ga System Time
`epoch_seconds_to_utc(seconds, epoch)`, `utc_to_epoch_seconds(&dt, epoch)`	—	A datetime value time and back, a reader and write an ISO literal: th second table on uniform scale or mjd, ntp and y2 GNSS epochs ar ror
`Quantity::to_civil()`	—	`epoch_seconds` of a `datetime` the sub-second stay in `frac`
`a == b`, `a.is_date_equal(&b)`, `a.is_time_equal(&b)`	`bvn_dt_is_equal`, `bvn_dt_is_date_equal`, `bvn_dt_is_time_equal`	Field-wise compa two `CivilDate`

GPS and Galileo seconds stay in their own scale: for UTC, take `tai_seconds_from_gps_time` to `tai_seconds_to_utc`. A datetime on a GNSS epoch has no civil form in the library, which is why `epoch_seconds_to_utc` and `to_civil()` refuse it.

The calendar. `GregorianDate::from_mjd` / `to_mjd` / `normalize` and `==` are `bvn_gregorian_date_from_mjd` / `_to_mjd` / `_normalize` / `_is_equal`; `mjd_to_wday`, `mjd_to_yday` and `mjd_in_leap_year` take a bare MJD; `easter_sunday_mjd(year)` and `easter_sunday`, `good_friday`, `easter_monday`, `ascension_day`, `whit_monday` give Easter and the feasts that hang off it. The German calendar is methods of `GregorianDate`: `is_federal_holiday_in_germany`, `is_banking_day_in_germany`, `is_business_day_in_germany`; `last_banking_day_in_month`, `penultimate_…`, `mid_…`, `first_…`, `last_sunday_in_month`, `last_saturday_in_month`; `last_bank-`

`ing_day_in_quarter(q)`, `first_banking_day_in_quarter(q)`; the `is_…` form of each banking-day getter; and `is_daylight_saving_switch_in_germany`, `is_day_before_daylight_saving_switch_in_germany`. The federal holidays are the nine that hold in all of Germany, by the rules in force that year (3 October from 1990, 17 June from 1954 to 1989, 1 May from 1933). A banking day is Monday to Friday, not a holiday and not 24 or 31 December; a business day is Monday to Saturday and not a holiday; the mid banking day is the first one on or after the 15th.

Errors. Where a C function signals failure with a sentinel, a `false` or an untouched output — an overflow, an MJD or year outside the supported range, a quarter outside 1–4 — the crate returns `Error::Argument`. Not bound: `bvn_gregorian_date_to_mjd_fast` (undefined behaviour out of range; `to_mjd()` is the checked form) and `bvn_dt_dump` / `bvn_dt_print`, which print to stdout (`Display` and `Debug` take their place).

---

7. `write_array` helper

`write_array` is a free function at the crate root. It handles both flat and multi-dimensional arrays, and takes the `Writer` to write into:

```rust
use bovnar::{Value, ValueTypeFamily, ValueTypeSpec, Writer, write_array};

fn main() -> Result<(), bovnar::Error> {
    let ints = |xs: &[i128]| xs.iter().map(|&x| Value::Int(x)).collect::<Vec<_>>();
    let mut w = Writer::to_mem(4096, true)?;

    // Single-row array
    write_array(&mut w, "primes", &ints(&[2, 3, 5, 7]), None, None, None)?;

    // Multi-row array (rows separated by /)
    let matrix = [Value::Array(ints(&[1, 2, 3])), Value::Array(ints(&[4, 5, 6]))];
    write_array(&mut w, "matrix", &matrix, None, None, None)?;

    // Typed array: whole-array type annotation
    let vt = ValueTypeSpec { family: ValueTypeFamily::Uint as i32, width: 16, base: 0 };
    write_array(&mut w, "ports", &ints(&[80, 443, 8080]), Some(vt), None, None)?;

    w.finish()?;
    let Value::Struct(fields) = bovnar::parse(w.get_output())? else { unreachable!() };
    assert_eq!(fields.len(), 3);
    Ok(())
}
```

The signature is `write_array(w, key, rows, vt, unit, unit_source)`: `vt` is the whole-array annotation (`Option<ValueTypeSpec>`), `unit` an `Option<&Unit>`, and `unit_source` the unit's spelling if it is to be kept.

Element types accepted per element: `Int`, `Float`, `String`, `Bool`, `Null`, `Bytes` (an octet stream), `Struct` (written as a struct), `Quantity`, or a nested `Array` (written as a nested array). When rows is a flat slice it is treated as a single-row array; when all top-level elements are themselves `Array`s it is treated as a multi-row array.

With `vt`, every element is held to it. An `Int` under an integer annotation is checked against the width and written in the annotation's base — quoted where the base requires it. An element the annotation cannot describe — `80800` under `<uint:16>`, a `Float` or a `String` under an integer type, a `Struct` under any — fails with `Error::Argument` instead of producing a document nothing reads. `Null` is a null hole and is always allowed; a `Quantity` element carries its own literal and is written as it is.

---

8. JSON conversion

The `bovnar::json` module mirrors the CLI's `bovnar convert` in both directions:

Function	Description
`json::from_json(&str) -> Result<Vec<u8>>`	JSON text to a BVNR document. The root must be an object. An integer wider than 64 bits is written as `<uint:N>` / `<sint:N>` (`N` the smallest power of two from 128 that holds it), byte for byte as `bovnar convert` writes it, and refused inside an array, whose elements share one type
`json::to_json(&[u8]) -> Result<String>`	A BVNR document to pretty-printed JSON, as `bovnar convert` prints it
`json::detect(&[u8]) -> Format`	Sniffs the format from the first significant byte: `Format::Json`, `Format::Bovnar` or `Format::Unknown`. The CLI goes by file extension; this is the content-based fallback for data without a name

```rust
use bovnar::json::{self, Format};

fn main() -> Result<(), bovnar::Error> {
    let text = r#"{"name": "example", "ports": [8080, 443]}"#;
    assert_eq!(json::detect(text.as_bytes()), Format::Json);

    let bvnr = json::from_json(text)?;
    assert_eq!(json::detect(&bvnr), Format::Bovnar);
    let back = json::to_json(&bvnr)?;
    assert_eq!(json::from_json(&back)?, bvnr);
    Ok(())
}
```

JSON arrays become a single bracket row with nested arrays as nested rows (never `/`-separated), as in the CLI. Nesting is limited to 255 levels — the writer's and the reader's limit — and a document deeper than any reader takes is refused as JSON rather than by the writer half-way through. NaN and the infinities have no JSON spelling and are written as `null`.

Two documented round-trip edge cases. JSON has one number type, so two values change on the way back, exactly as with the CLI:

- a float with an integral value becomes an integer — `1e+6` is written `1000000` and reads back as `uint`;
- `-0.0` is written `-0`, which reads back as `0`.

```rust
use bovnar::json::{from_json, to_json};

fn main() -> Result<(), bovnar::Error> {
    let first = to_json(&bovnar::to_string(
        &bovnar::Value::Struct(vec![("big".into(), bovnar::Value::Float(1e6))]),
        true,
    )?)?;
    assert!(first.contains("1000000"));           // not 1e+6
    let back = from_json(&first)?;                // ... and it reads back as an integer
    assert_eq!(to_json(&back)?, first);           // from here on the JSON is stable
    Ok(())
}
```

From the second trip on, the JSON is stable: the first `to_json` / `from_json` pair normalises, and nothing after it changes.

---

9. Currency helpers

The `bovnar::currency` module provides metadata for the ISO 4217 fiat and cryptocurrency `BaseUnit` members. The catalogue is read from the C library, so there is no table to keep in step.

```rust
use bovnar::BaseUnit;
use bovnar::currency::{
    all_crypto, all_fiat, currency_code, currency_info, from_code, is_crypto,
is_currency, is_fiat,
    minor_unit,
};

fn main() -> Result<(), bovnar::Error> {
    assert!(is_currency(BaseUnit::Usd));
    assert!(is_fiat(BaseUnit::Usd));
    assert!(is_crypto(BaseUnit::Btc));

    let info = currency_info(BaseUnit::Usd)?;
    assert_eq!(info.code, "USD");
    assert_eq!(info.numeric_code, 840);   // ISO 4217 numeric; 0 for crypto
    assert_eq!(info.minor_unit, 2);       // decimal places: 1 major = 10^minor minor
units
    assert_eq!(info.name, "US Dollar");

    assert_eq!(minor_unit(BaseUnit::Jpy)?, 0);
    assert_eq!(currency_code(BaseUnit::Eur)?, "EUR");
    assert_eq!(from_code("gbp")?, BaseUnit::Gbp);   // surrounding space and case are ig-
nored

    assert!(all_fiat().any(|ci| ci.code == "CHF")); // all_crypto() / all_currencies()
also exist
    assert!(all_crypto().all(|ci| ci.numeric_code == 0));
    Ok(())
}
```

`CurrencyInfo` is a plain struct (`base`, `code`, `numeric_code`, `minor_unit`, `name`) with an `is_fiat()` method; its `Display` is the code. `currency_info`, `minor_unit`, `currency_name`, `currency_code` and `from_code` fail with `Error::Argument` for non-currency bases or

unknown codes. `CURRENCY_FIRST`, `CURRENCY_LAST` and `CURRENCY_COUNT` are re-exported from the module.

---

10. `Reader` reference

10.1 Construction

```rust
use bovnar::Reader;

fn main() -> Result<(), bovnar::Error> {
    let r = Reader::new()?;
    drop(r); // the C object is released when the Reader is dropped
    Ok(())
}
```

`Reader::new` calls `bvnr_reader_create` immediately. There is no context manager and no `close()`: the C object is released when the `Reader` is dropped, and a dropped reader cannot be used again because the compiler has moved it.

10.2 `read_mem`

```rust,ignore read_mem(data: &[u8], options: &ReadOptions, on_verified: Option<&mut dyn FnMut(&EventData) -> bool>, on_unverified: Option<&mut dyn FnMut(&EventData) -> bool>) -> Result<()>

read_mem_want_unit(data, options, want_unit: &mut dyn FnMut(&EventData) -> Option<(Unit, u32)>, on_verified) -> Result<()>

read_mem_with(data, options, handlers: ReadHandlers) -> Result<()>

read_mem_mirror(data, options, handlers: ReadHandlers, mirror: &mut dyn std::io::Write) -> Result<()>
```

```
Parse BVNR from a byte slice. The options are the `ReadOptions` struct, and the
`want_unit` hook is the separate method `read_mem_want_unit`. `read_mem_with`
(and `read_reader_with` over a `std::io::Read`) takes every handler at once, as
a `ReadHandlers` whose fields are all optional: `on_verified`, `on_unverified`,
`want_unit` — so a hook can sit beside an `on_unverified` callback — and
`on_error`.

| Parameter | Type | Description |
|---|---|---|
| `data` | `&[u8]` | Input buffer |
| `options` | `&ReadOptions` | See [§10.3](#103-readoptions) |
| `on_verified` | `Option<&mut dyn FnMut(&EventData) -> bool>` | Callback for validated
events |
| `on_unverified` | `Option<&mut dyn FnMut(&EventData) -> bool>` | Callback for raw pre-
validation events |
| `want_unit` | `&mut dyn FnMut(&EventData) -> Option<(Unit, u32)>` | Read-time lossless
unit/base conversion hook. Called per numeric value; return `Some((unit, base))` to re-
quest a conversion (the exact result arrives on the payload as `converted` and `conver-
ted_text`, with `converted_rational()` / `converted_in_base()` beside it — see §4.2), or
`None` to leave the value untouched. An inexact conversion aborts the read (`er-
ror_unit_inexact`) |
| `on_error` | `&mut dyn FnMut(&Error)` (in `ReadHandlers`) | Called for every error the
reader reports, as an `Error::Parse` with its position: under `continue_on_error`, each
error it recovers from — reported nowhere else — and the error that ends a read, before
the read returns it. A panic in it is parked: the read stops at the next call of an
`on_verified`, `on_unverified` or `want_unit` handler, or, with none installed, runs to
the end; then the panic resumes in the caller |

```rust
use bovnar::{Error, ReadHandlers, ReadOptions, Reader};

fn main() -> Result<(), Error> {
    let mut recovered = Vec::new();
    let mut on_error = |e: &Error| recovered.push(e.position().unwrap().offset);
    let options = ReadOptions { continue_on_error: true, ..ReadOptions::default() };
    let _ = Reader::new()?.read_mem_with(
        b".a = 1; .b = @; .c = 2;",
        &options,
        ReadHandlers { on_error: Some(&mut on_error), ..ReadHandlers::default() },
    );
    assert_eq!(recovered.first(), Some(&13)); // the '@'
    Ok(())
}
```

`read_mem_mirror` (and `read_reader_mirror` over a `std::io::Read`) is `read_mem_with` with `mirror` receiving every byte the reader consumes — the input mirror of `bvnr_open_read_source`, a debugging aid — comments and octet-stream payloads included, so after a read that succeeded it holds the input. The bytes reach it a piece at a time, each once the reader has parsed it: a buffer pulled (up to 64 KiB), a piece fed, an octet-stream chunk. A read that fails, or that a handler stops, has mirrored the input up to the byte it stopped at. A failing `mirror` ends the read with an `Error::Io` with `writing_to_sink`; a panic in it reaches the caller. A `live::DocumentFeed` takes one with `set_mirror` (§4.9).
```

```rust
use bovnar::{ReadHandlers, ReadOptions, Reader};

fn main() -> Result<(), bovnar::Error> {
    let mut seen = Vec::new();
    Reader::new()?.read_mem_mirror(b".a = 1;  # note\n", &ReadOptions::default(),
                                   ReadHandlers::default(), &mut seen)?;
    assert_eq!(seen, b".a = 1;  # note\n");
    Ok(())
}
```

A reader opens at the 255-level struct and array nesting cap, in both directions, like the CLI and the DOM, unless `ReadOptions` lowers it.

The file-descriptor entry points `read_fd` and `read_file` are `read_reader`, `read_reader_want_unit`, `read_reader_with` and `read_reader_mirror`, over any `std::io::Read` — see §4.7.

10.3 `ReadOptions`

`ReadOptions` is plain data with a `Default`; fill what you need and leave the rest with `..ReadOptions::default()`.

Field	Type	Default	Description
`max_file_size`	`u64`	`0` (unlimited)	Hard limit on bytes consumed; past it the read fails with `file_too_long`
`max_text_bytes`	`u64`	`0` (default 2 147 483 647)	Most bytes of text in a document (octet-stream payloads not counted); past it the read fails with `text_data_too_long`. `u64::MAX` lifts it, for an endless text document
`max_array_items`	`u64`	`0` (default 2 147 483 647)	Most items in one array; past it the read fails with `too_many_array_items`. `u64::MAX` lifts it
`continue_on_error`	`bool`	`false`	Enable resync mode
`strict_version`	`bool`	`false`	Reject a declared spec version newer than this build (`error_unsupported_spec_version`)
`text_only`	`bool`	`false`	Refuse a document that contains an octet stream (`error_octet_stream_forbidden`), for a channel that is not byte-transparent
`want_unit_allow_nonterminating`	`bool`	`false`	Deliver an exact-rational-but-non-terminating conversion (e.g. km/h→m/s) as a rational — `converted_text` is empty, `converted_rational()` returns the exact pair — instead of aborting with `error_unit_inexact`

Field	Type	Default	Description
`max_conversion_length`	`u32`	`0` (default 1024)	Longest converted text a conversion may produce, in characters; a longer exact result aborts with `error_value_out_of_range`
`max_identifier_length`	`u16`	`0` (default 255)	Longest key, in bytes (`identifier_too_long`)
`max_string_length`	`u16`	`0` (default 65 535)	Longest string value, in bytes (`string_too_long`)
`max_number_length`	`u16`	`0` (default 65 535)	Longest number literal (`number_too_long`)
`max_symbol_length`	`u16`	`0` (default 255)	Longest symbol (`symbol_too_long`)
`max_reference_length`	`u16`	`0` (default 65 535)	Longest reference (`reference_too_long`)
`max_struct_nesting`	`u8`	`0` (255)	Deepest struct nesting (`struct_nesting_too_high`). `0` is the format's limit of 255, what the DOM and the CLI read at — not the C reader's own default of 64
`max_array_nesting`	`u8`	`0` (255)	Deepest array nesting, as `max_struct_nesting` (`array_nesting_too_high`)

Every field of `bvnr_read_flags_t` that is a limit or a switch is here, and every reading entry point takes the whole struct: `Reader`, the `live` feeds, `parse_with_options` / `parse_reader_with_options` (§4.1), and the `*_with_options` twins of the `stream` readers (§4.6). The five length caps and the two nesting caps are for untrusted input: lower them to what the consumer can take.

`max_file_size` is 0 by default, which means no cap; set it where a cap matters. `max_text_bytes` and `max_array_items` are 0 by default too, but 0 selects the library's finite default there, as in C (`bvnr_read_flags_t`), so an untrusted document stays bounded unless a caller lifts the cap.

10.4 `EventData` and `EventPayload`

`EventData<'a>` — the table in §4.2 — is the borrowed view a callback receives. `EventPayload` is the owned copy: the same fields without the borrow, with `raw` a `Vec<u8>`, `conver-`

`ted_text` an `Option<String>` (`None` when the conversion does not terminate in `converted_base`), and a `text()` method.

Field	Type	Description
`event`	`Event`	The event code
`token_type`	`Option<TokenType>`	How the token was written
`value_type`	`ValueTypeSpec`	Type annotation if present
`value_unit`	`Unit`	Unit if present
`raw`	`Vec<u8>`	Raw token bytes
`text()`	`String`	`raw` decoded as UTF-8, invalid bytes replaced
`converted`	`bool`	`true` when a `want_unit` conversion was applied to this value
`converted_text`	`Option<String>`	Exact converted value as a positional string; `None` if the conversion does not terminate in `converted_base`
`converted_base`	`u32`	Base the converted text is rendered in

10.5 `iter_mem`

`iter_mem(data, verified_only, max_file_size) -> Result<Vec<EventPayload>>` reads `data` and returns all its events, in order. It is a `Vec`, not a lazy iterator: the read has finished when it returns.

Parameter	Description
`verified_only`	When `false`, both callbacks fire and events may appear twice
`max_file_size`	Forwarded to `read_mem`; `0` is unlimited

For any other option, call `read_mem` with a collecting handler.

10.6 Error-state accessors

These query the C reader object after a failed parse. They describe the error that **ended** the read; the `Error::Parse` that `read_mem` returned carries the same code and position. An error recovered under `continue_on_error: true` does not appear here: after a read that succeeded, `error_code()` is `Some(ErrorCode::None)` and the line is `0`, however many errors were recovered, and `recovery_count()` is how to tell that any were (spec §13.4).

Method	Type	Description
`error_code()`	`Option<ErrorCode>`	The error that ended the read; `ErrorCode::None` after a successful one

Method	Type	Description
`error_line()`	`u64`	1-based line number of the error
`error_position()`	`Position`	Line, character column, byte offset and offending byte of the error, all 0 after a clean read
`recovery_count()`	`u64`	Number of times resync was entered (incremented at error entry, not at resync completion)
`skipped_bytes()`	`u64`	How many input bytes a `continue_on_error` read skipped
`declared_version()`	`Option<(u16, u16)>`	The `#!bovnar M.N` directive of the last read, or `None` when the document carried none

The same position is on the `Error` itself — see §13 — as `position()`. To see the errors a `continue_on_error` read recovered from, pass an `on_error` handler (§10.2).

---

11. `Writer` reference

11.1 Construction

Method	Description
`Writer::to_mem(cap, pretty)`	Write to an in-process buffer of `cap` bytes; `pretty` indents the output. Writing past `cap` fails with `sink_buffer_exhausted` (`DEFAULT_MEM_CAP` is 4 MiB)
`Writer::to_mem_with(cap, pretty, unit_flags, emit_version)`	`to_mem` with two more settings: `unit_flags` (`UnitFlags`) rewrite units on output, `emit_version` leads the document with `#!bovnar <major>.<minor>`
`Writer::to_writer(out, pretty)`	Write to any `std::io::Write`: the document streams into `out` in 64 KiB blocks as it is written, the rest on `flush()` or `finish()`, which flush `out`; no cap on its size
`Writer::to_writer_with(out, cap, pretty, unit_flags, emit_version)`	`to_writer` for a document of at most `cap` bytes (`sink_buffer_exhausted` past it, from the call that pushes the output over it), plus `unit_flags` and `emit_version` as in `to_mem_with`
`Writer::to_mem_with_options(cap, options: WriteOptions)`	`to_mem` with every write option, below
`Writer::to_writer_with_options(out, cap, options: WriteOptions)`	`to_writer_with` with every write option; `cap` `usize::MAX` is no cap

See §4.7 for the I/O writer. By default every writer accepts structs and arrays nested up to **255** deep, the same limit `Reader` and `parse()` read at. `unit_flags` takes `UnitFlags` — `RE-`

`DUCE` rewrites each unit and rescales its value, `ASCII_EXP` writes `^N` exponents, `NUM_CANONICAL` keeps a rescaled float in float shape. `emit_version: true` writes a leading `#!bovnar M.N` naming the spec version this build implements, as `write_version()` would.

`WriteOptions` is the whole of `bvnr_write_flags_t` that the C writer consults, with a `Default`: `pretty`, `unit_flags`, `emit_version`, the nesting caps `max_struct_nesting` and `max_array_nesting` (`0` is 255, as for `ReadOptions`; a struct or an array one level deeper fails the write with `struct_nesting_too_high` or `array_nesting_too_high`), and `on_event`, the observer of §11.9. It holds a boxed closure, so it is passed by value and is not `Clone`.

```
use bovnar::{ErrorCode, WriteOptions, Writer};

fn main() -> Result<(), bovnar::Error> {
    let options = WriteOptions { max_struct_nesting: 2, ..WriteOptions::default() };
    let mut w = Writer::to_mem_with_options(4096, options)?;
    w.begin_struct("a")?;
    w.begin_struct("b")?;
    let err = w.begin_struct("c").unwrap_err();
    assert_eq!(err.code(), Some(ErrorCode::StructNestingTooHigh));
    Ok(())
}
```

The high-level `to_string` needs no buffer size: it retries on a writer twice the size until the output fits (up to 256 MiB).

11.2 Output retrieval

Method	Description
`w.get_output() -> &[u8]`	The document written so far; empty for a `to_writer` writer, whose output has gone to its `std::io::Write`
`Writer::bytes_written -> u64`	Number of bytes written
`w.flush()`	Hand over what is written so far without ending the document (`bvnr_write_flush`): a `to_writer` writer pushes what it holds to its `std::io::Write` and flushes it once; a `to_mem` writer does nothing. See §11.5
`w.finish()`	Flush and seal the output. Fails with `Error::Argument` if any struct is still open. For a `to_writer` writer it also pushes what the writer still holds to its `std::io::Write` and flushes it

There is no `destroy()`: dropping the `Writer` releases the C object.

11.3 Scalar write methods

Every scalar writer takes the key first and returns `Result<()>`. They take no unit parameters: a value that carries a unit is written with `write_quantity` (or inside `write_array`), which take a `Unit`.

`write_uint(key, value: u64, width: u32, base: Base)`

Write an unsigned integer. `base` selects the numeral system (`Base::DECIMAL` or `Base::HEX` for inline values; any Bovnar-supported base for `write_bvni`).

`write_sint(key, value: i64, width: u32, base: Base)`

Write a signed integer. `base` 64 and 85 are **unsigned** — `+` and `-` are digits in those alphabets — and fail with `Error::Argument` here and for a signed `write_bvni`.

`write_float(key, value: f64, width: u32)`

Write an IEEE 754 binary float.

`write_float_fix(key, value: f64, width: u32, q: u32)`

Write a Q-format fixed-point value. `q` is the number of fractional bits (`0 ≤ q < width`).

`write_float_dec(key, value: f64, width: u32)`

Write an IEEE 754-2008 decimal float.

`write_string(key, value: &str)`

Write a bare quoted UTF-8 string with no type annotation:

```
.host = "localhost";
```

To emit an explicit `<utf8>` annotation, use the low-level `emit` API with `Event::TypeAnnotationStart` / `TypeAnnotationEnd` before `Event::Data`.

`write_bool(key, value: bool)`

Write a `bool` value — serialized as the bare keyword `true` or `false`. On read-back it decodes to `Value::Bool`.

`write_null(key)`

Write a null value (empty slot).

`write_octets(key, value: &[u8])`

Write an octet stream, chunked. It reads back as `Value::Bytes`.

`write_quantity(key, q: &Quantity)`

Write a typed scalar as `parse_typed` read it: type, unit, unit spelling and datetime fraction are kept. The annotation is left out where it says nothing the literal does not.

```rust
use bovnar::{Quantity, Unit, ValueTypeFamily, ValueTypeSpec, Writer};

fn main() -> Result<(), bovnar::Error> {
    let float64 = ValueTypeSpec { family: ValueTypeFamily::Float as i32, width: 64, base:
0 };
    let mut w = Writer::to_mem(4096, false)?;
    w.write_quantity("velocity", &Quantity::new("9.81", float64, Unit::parse("m/s")?))?;
    w.finish()?;
    assert_eq!(w.get_output(), b".velocity=<float:64,m/s>9.81;");
    Ok(())
}
```

11.4 Extended integer writers

`write_bvni(key, value: i128, width: u32, base: Base, signed: Signedness)`

Arbitrary-width integer writer that supports all Bovnar numeral bases (**2–62, 64, and 85**), for an integer of any width up to 128 bits. Non-decimal values are formatted with bovnar's own big-integer arithmetic and emitted as quoted strings. `signed` is a `Signedness`: `Signed`, `Unsigned`, or `Auto` — `sint` for a negative value, `uint` otherwise (the default). `format_bigint(value, base)` renders an integer in any base on its own.

`write_big_int(key, value: &BigInt, width, base, signed)`

`write_bvni` for an integer of any size: `value` is a `num_bigint::BigInt`, as `Value::BigInt` carries one past `i128`, and `width`, `base` (`Base`) and `signed` (`Signedness`, `Auto` writing a negative value as `sint`) are those of `write_bvni`. `writer::format_big_int(value, base)` renders it on its own. A signed value in base 64 or 85 fails with `Error::Argument`, as for `write_bvni`.

`write_bvnf_base(key, literal: &str, width: u32, base: Base)`

Write a float from a pre-formatted string in base 10 or 16. `base` must be `Base::DECIMAL` or `Base::HEX`; any other value fails with `Error::Argument`.

`Base`

`Base` is a checked newtype over the numeral bases bovnar has an alphabet for — 2 to 62, 64 and 85 — rather than an enum of 63 variants. The common ones are named: `BINARY`, `OCTAL`, `DECIMAL`, `HEX`, `BASE36`, `BASE62`, `BASE64`, `BASE85`. `Base::new(n)` is `Some` for a base bovnar has an alphabet for and `None` otherwise; `get()` returns the number.

```rust
use bovnar::{Base, Signedness, Writer};

fn main() -> Result<(), bovnar::Error> {
    assert_eq!(Base::new(16), Some(Base::HEX));
    assert_eq!(Base::new(63), None);

    let mut w = Writer::to_mem(4096, false)?;
    w.write_bvni("big", 1i128 << 100, 128, Base::DECIMAL, Signedness::Auto)?;
    w.write_bvni("neg", -255, 16, Base::HEX, Signedness::Auto)?;
    w.finish()?;
    assert_eq!(bovnar::parse(w.get_output())?, bovnar::parse(
        b".big = <uint:128> 126765060022822940149670320537 6; .neg = <sint:16,_16>
\"-ff\";",
    )?);
    Ok(())
}
```

11.5 Struct helpers

```rust
use bovnar::Writer;

fn main() -> Result<(), bovnar::Error> {
    let mut w = Writer::to_mem(4096, true)?;
    w.begin_struct("server")?;        // emit AssignmentStart + StructStart, increment
depth
    w.write_string("host", "example.com")?;
    w.end_struct()?;                  // emit StructEnd, decrement depth
    w.finish()?;
    Ok(())
}
```

`finish()` verifies that the struct depth is zero; an unclosed struct fails with `Error::Argument` before the C library is asked to finish.

Two more kinds of value go into an open writer, whatever it writes to:

Method	Writes
`w.embed_document(key, doc)`	`.key = <octets>;` holding the document `doc` (`bvnr_embed_document`), for `stream::parse_embedded` or `live::EmbeddedFeed` to read back
`w.mux_begin(key)`, `w.mux_send(channel, payload)`, `w.mux_end()`	a multiplexed octet stream under `key` (`bvnr_mux_begin` / `_send` / `_end`), the bytes of `stream::mux_dump`; values written before it are read back before its first message, and the document may go on after it

On a `to_writer` writer a message reaches `out` when the writer's 64 KiB block fills, when the document is finished, or at `w.flush()` (`bvnr_write_flush`), which writes out what the block holds and then calls `out.flush()`, once per call. A consumer that needs each message as it is sent — a `live::Demux` at the other end of a socket — gets it by a `flush()` after each `mux_send`. It is legal between any two writer calls, inside an open struct, array or mux stream too, and it changes only when the bytes leave, never which: a document flushed after every call is byte for byte the one written without a flush. It allocates nothing.

A failing `out` is an `Error::Io` with `writing_to_sink`, from the `flush()` and from every call after it; on a `to_mem` writer `flush()` does nothing.

```rust
use std::io::Read;
use bovnar::live::{Demux, DemuxItem, DemuxOptions};
use bovnar::Writer;

fn main() -> Result<(), bovnar::Error> {
    let (mut rx, tx) = std::io::pipe().expect("pipe");
    let mut w = Writer::to_writer(tx, false)?;
    w.begin_struct("meta")?;
    w.write_string("codec", "av1")?;
    w.end_struct()?;
    w.mux_begin("media")?;
    w.mux_send(1, b"key frame")?;
    w.flush()?; // the metadata and the first message leave now

    // The other end has the message while the document is still open
    let options = DemuxOptions { key: Some("media".into()), ..DemuxOptions::default() };
    let mut demux = Demux::new(&options)?;
    let mut buf = [0u8; 4096];
    let mut first = None;
    while first.is_none() {
        let n = rx.read(&mut buf).expect("read");
        let mut input = &buf[..n];
        while let Some(item) = demux.next(&mut input)? {
            if let DemuxItem::Message { channel, payload } = item {
                first = Some((channel, payload.to_vec()));
            }
        }
    }
    assert_eq!(first, Some((1, b"key frame".to_vec())));
    w.mux_end()?;
    w.finish()
}
```

Two writers produce a multiplexed stream message by message, byte for byte the same for the same messages:

	`Writer` with `mux_send` and `flush`	`codec::MuxWriter` (§4.8)
Implementation	the C writer: a C call per send and per flush, each message copied into the writer's 64 KiB buffer	pure Rust, no C call; the chunk headers can be handed out next to the borrowed payload for a vectored write
Document shape	any: the stream is one value, with values of every kind before and after it, each one checked as the writer checks every value	a text preamble, the stream, and nothing after it
Destination	a `to_writer` writer's `std::io::Write`, flushed by `flush()`	a `Vec` or any `std::io::Write`, which the caller flushes
Use it for	a stream inside a larger document, or when the writer already writes the rest	a dedicated stream, a router or a relay

11.6 Version directive

```rust
use bovnar::Writer;

fn main() -> Result<(), bovnar::Error> {
    let mut w = Writer::to_mem(4096, true)?;
    w.write_version(1, 1)?;           // emit a leading "#!bovnar 1.1" directive
    w.write_uint("n", 1, 8, bovnar::Base::DECIMAL)?;
    w.finish()?;
    assert!(w.get_output().starts_with(b"#!bovnar 1.1"));
    Ok(())
}
```

Emits a leading `#!bovnar <major>.<minor>` directive. Must be called before any value is written; calling it after output has begun fails with `Error::Write` (`InvalidArgument`). Use it to self-declare the spec version a document targets (spec-1.1 constructs such as datetime literals need it to re-read).

11.7 Array helpers

```rust
use bovnar::{EmitSpec, Event, ValueTypeSpec, Writer};

fn main() -> Result<(), bovnar::Error> {
    let plain = ValueTypeSpec::default();
    let mut w = Writer::to_mem(4096, false)?;
    w.emit(Event::AssignmentStart, EmitSpec::key("m"))?;
    for (i, row) in [["1", "2"], ["3", "4"]].iter().enumerate() {
        if i > 0 {
            w.new_array_dim()?;     // emit ArrayDimStart (the / separator between rows)
        }
        w.begin_array_row()?;       // emit ArrayRowStart
        for n in row {
            w.emit(Event::Data, EmitSpec::value(n, plain))?;
        }
        w.end_array_row()?;         // emit ArrayRowEnd
    }
    w.finish()?;
    assert_eq!(bovnar::parse(w.get_output())?, bovnar::parse(b".m = [1, 2] / [3, 4];")?);
    Ok(())
}
```

These are the raw events; `write_array` is the helper that drives them for you.

11.8 Low-level `emit`

```rust,ignore w.emit(event: Event, spec: EmitSpec<'_>) -> Result<()>
```

```
Send an arbitrary event to the C writer. What goes with it — a key, a value and a type
annotation — is an `EmitSpec`; it carries no unit. When both a `vt` and a `value` are sup-
plied the token type is
inferred: a string token for `ValueTypeFamily::Utf8`, a number otherwise.

| Purpose | Call |
|---|---|
| Start an assignment with a key | `w.emit(Event::AssignmentStart, EmitSpec::key("x"))` |
| Write a typed value | `w.emit(Event::Data, EmitSpec::value("42", spec))` |
| End a struct (nothing attached) | `w.emit(Event::StructEnd, EmitSpec::NONE)` |

```rust
use bovnar::{EmitSpec, Event, ValueTypeFamily, ValueTypeSpec, Writer};

fn main() -> Result<(), bovnar::Error> {
    let spec = ValueTypeSpec { family: ValueTypeFamily::Uint as i32, width: 16, base: 0 };
    let mut w = Writer::to_mem(4096, false)?;
    w.emit(Event::AssignmentStart, EmitSpec::key("x"))?;
    w.emit(Event::Data, EmitSpec::value("42", spec))?;
    w.finish()?;
    assert_eq!(bovnar::parse(w.get_output())?, bovnar::parse(b".x = <uint:16> 42;")?);
    Ok(())
}
```

Two calls go a level lower, for what an `EmitSpec` cannot say:

```rust,ignore w.event(event: Event, token: TokenType, vt: ValueTypeSpec, data: &[u8]) -> Result<()> w.annotation(vt: ValueTypeSpec, unit: &Unit, unit_source: Option<&str>) -> Result<()>

````
`event` sends one event with the token type given explicitly
(`bvnr_write_event`): a plain symbol is `Event::Data` with `TokenType::Symbol`,
an array item `TokenType::ArrayNumber` or `ArrayString`. `data` longer than
4 GiB fails with `Error::Argument`. `annotation` writes a whole type
annotation, `<family:width[,_base|,qN|,epoch][,unit]>`, as the events from
`TypeAnnotationStart` to `TypeAnnotationEnd`: the width of a sized family when
it is not 0, the base of an integer or float that is not 10, the `q` of a
fixed-point value, the epoch of a datetime and the unit when it has components
(left out for `datetime` and `utf8`). `unit_source` is the unit's spelling to
write instead of its canonical one, checked by the C writer as for
`bvnr_writer_set_unit_source`; a unit the library cannot serialise fails with
`Error::Argument`. It goes after the key and before the value, or before an
array's first row ([§11.10](#1110-the-c-per-type-helpers) uses both).

11.9 Observing the output: `on_event`

`WriteOptions::on_event` is a `Box<dyn FnMut(&EventData<'_>) -> bool + Send>`
(`WriteObserver`), called with each event once it has been serialised
(`bvnr_write_flags_t.on_event`), with what reached the output: under
`UnitFlags::REDUCE` a rescaled value's new digits and unit, not the ones it was
written with. Every event the writer serialises reaches it — those of the
`write_*` methods, `emit`, the struct and array helpers, the mux and
embedding calls. It is owned by the writer, so it shares state with the code
around it through an `Arc` (or a channel); it cannot reach the writer itself.

```rust
use bovnar::units::UnitFlags;
use bovnar::{Event, Quantity, Unit, ValueTypeFamily, ValueTypeSpec, WriteOptions, Writer};
use std::sync::{Arc, Mutex};

fn main() -> Result<(), bovnar::Error> {
    let seen = Arc::new(Mutex::new(Vec::new()));
    let log = Arc::clone(&seen);
    let options = WriteOptions {
        unit_flags: UnitFlags::REDUCE,
        on_event: Some(Box::new(move |d| {
            if d.event == Event::Data {
                log.lock().unwrap().push((d.text(), d.value_unit.to_string()));
            }
            true
        })),
        ..WriteOptions::default()
    };
    let mut w = Writer::to_mem_with_options(4096, options)?;
    let uint = ValueTypeSpec { family: ValueTypeFamily::Uint as i32, width: 64, base: 0 };
    w.write_quantity("d", &Quantity::new("5", uint, Unit::parse("k~m")?))?;
    w.finish()?;
    assert_eq!(w.get_output(), b".d=<uint:64,m>5000;");
    assert_eq!(*seen.lock().unwrap(), [("5000".to_string(), "m".to_string())]);
    Ok(())
}
```
````

Returning false fails the write, and the writer, with `scanner_callback_failed`: a veto, say of a key that must not leave the process. A panic in the observer fails it the same way and reaches the caller of the write call that ran it; a later call returns the `Error::Write`.

11.10 The C per-type helpers

The bindings write every value through `bvnr_write_event`; each of the C per-type helpers has its counterpart, so none is bound itself:

C helper	Rust
`bvnr_write_string`, `_null`, `_bool`	`write_string`, `write_null`, `write_bool`
`bvnr_write_uint`, `_sint`, `_float`, `_float_fix`, `_float_dec`	the methods of §11.3
their `_unit` forms, `bvnr_writer_set_unit_source`	`write_quantity`, whose `Quantity` carries the unit and its spelling (`unit_source`)
`bvnr_write_datetime`	`write_quantity` with a `datetime` type (the epoch in its `base`, `datetime_epoch_index`)
`bvnr_write_plain`	`event(Event::Data, TokenType::Symbol, …)` after the key, or `write_quantity` of a symbol
`bvnr_write_type_annotation`	`annotation(vt, &unit, unit_source)`, before a value or an array's first row
`bvnr_write_bvnf*`, `bvnr_write_bvni*`	`write_bvnf_base`, `write_bvni`, `write_big_int` (§11.4)
`bvnr_write_struct_start`, `_end`	`begin_struct`, `end_struct` (§11.5)

```rust
use bovnar::{
    EmitSpec, Event, Quantity, TokenType, Unit, ValueTypeFamily, ValueTypeSpec, Writer,
    datetime_epoch_index,
};

fn main() -> Result<(), bovnar::Error> {
    let mut w = Writer::to_mem_with(4096, false, 0, true)?; // datetime is spec 1.1
    w.emit(Event::AssignmentStart, EmitSpec::key("mode"))?;
    w.event(Event::Data, TokenType::Symbol, ValueTypeSpec::default(), b"auto")?;
    let tai = datetime_epoch_index("tai").unwrap();
    let datetime = ValueTypeSpec { family: ValueTypeFamily::Datetime as i32, width: 64,
base: tai };
    w.write_quantity("t", &Quantity::new("1750000000", datetime, Unit::default()))?;
    let u8_ = ValueTypeSpec { family: ValueTypeFamily::Uint as i32, width: 8, base: 0 };
    w.emit(Event::AssignmentStart, EmitSpec::key("v"))?;
    w.annotation(u8_, &Unit::parse("m")?, None)?;
    w.begin_array_row()?;
    w.event(Event::Data, TokenType::ArrayNumber, u8_, b"1")?;
    w.event(Event::Data, TokenType::ArrayNumber, u8_, b"2")?;
    w.end_array_row()?;
    w.finish()?;
    let out = w.get_output();
    let body = &out[out.iter().position(|&b| b == b'\n').unwrap() + 1..];
    assert_eq!(body, b".mode=auto;.t=<datetime:64,tai>1750000000;.v=<uint:8,m>[1,2];");
    Ok(())
}
```

12. DOM API

The DOM API parses a complete BVNR document into an in-memory tree for random-access queries without writing a SAX callback.

12.1 `DomDoc::parse`

```rust
use bovnar::DomDoc;

fn main() -> Result<(), bovnar::Error> {
    let doc = DomDoc::parse(b".a = 1;")?;
    assert_eq!(doc.len(), 1);
    Ok(())
}
```

A malformed document is an `Error::Parse` carrying the C error code. **It has no position:** the C DOM reports only the code (`bvn_dom_doc_get_parse_error`), so `Error::position()` is `None`. To locate the fault, read the same document with a `Reader`, which does report where.

12.2 `DomDoc`

Owning wrapper around `bvn_dom_doc_t`. Dropping it frees the entire tree; the `DomNode`s derived from it borrow it, so none can outlive it.

Method	Description
`DomDoc::parse(data)`	Parse a byte slice
`DomDoc::parse_reader(input)`	`parse` over any `std::io::Read`; see the note below
`DomDoc::parse_reader_with_policy(input, &policy)`	`parse_reader` under a policy
`DomDoc::parse_reader_capped(input, max_bytes)` / `parse_reader_capped_with_policy(input, max_bytes, &policy)`	`parse_reader` with a lower cap, as `bvn_dom_parse_fd_ex` takes one; see the note below
`DomDoc::parse_with_policy(data, &policy)`	Parse under a `UnitPolicy`: values arrive converted, and a document breaking the policy fails with `unit_mismatch`
`doc.get(key)`	The top-level value assigned to `key`, as `Option<DomNode>`
`doc.lookup(path)`	Dot-separated path lookup, e.g. `"server.tls.cert"`, with indexed elements (`"matrix[1][0]"`). `None` when absent
`doc.len()` / `doc.is_empty()`	Number of top-level entries
`doc.entries()`	All top-level `(String, DomNode)` pairs, in document order
`doc.to_value()`	The whole document as a `Value::Struct`, dropping type and unit info. Fails where a node cannot be read as its kind — see `node.to_value()`
`DomDoc::create()`	An empty document to build in (§12.5)

Method	Description
`doc.add(key, node)`	Add a `NewNode` as a top-level entry (§12.5)

> **File-size limits:** `DomDoc::parse_reader` and `parse_reader_with_policy` take any `std::io::Read` and apply the same ceiling, `DOM_READ_MAX_BYTES` (256 MiB, `BVN_DOM_FD_MAX_BYTES`). A longer input is an `Error::Parse` with `file_too_long`, where the C function returns no document. Input over 4 GiB is refused as well (the C API's length is a `uint32_t`). `parse_reader_capped` lowers the ceiling to `max_bytes`; 0, or anything above 256 MiB, keeps it — there is no unlimited mode, as in C.
>
> **The DOM takes no read options.** The C DOM parse functions read with a fixed set (the 255-level nesting, the reader's defaults for everything else), so `ReadOptions` cannot reach them; read with a `Reader` or `parse_with_options` for those limits.
>
> **Building a tree.** `DomDoc::create()` and `doc.add(key, node)` are the C DOM's builder, for a tree made by hand from `NewNode`s (§12.5). Such a tree cannot be written out; build a document as a `Value` or with the `Writer`.

Membership is `get(key).is_some()`, and iteration is `entries()`.

12.3 `DomNode`

A `Copy` view into a `bvn_dom_node_t` that borrows its `DomDoc`.

Method	Description
`dom_type()`	`Option<DomType>` — `None` for a kind this binding does not know
`node.value_type()`	`ValueTypeSpec`
`node.unit()`	`Unit`; dimensionless when the value has none
`unit_str()`	Unit as a canonical string via `bvn_dom_get_unit_string`, or `""` when the value carries no unit — the same contract as `unit_str()` (the C formatter's `no_unit` spelling is never returned)
`node.is_null()`	`true` for null values
`node.value_in_base_units()`	Numeric value scaled to SI base units (`f64`); an affine unit gets its offset applied (`25 °C` → `298.15`). Returns `0.0` for any failure — non-numeric node, currency, affine-in-compound — which is indistinguishable from a genuine zero, and it rounds to `double`. Use `units::unit_to_si_factor()` (which reports the failure) or the `want_unit` hook when either matters
`node.as_i64()` / `as_u64()`	Signed / unsigned 64-bit integer, as an `Option`
`node.as_i32()` / `as_u32()`	Signed / unsigned 32-bit integer
`node.as_i16()` / `as_u16()`	Signed / unsigned 16-bit integer
`node.as_i8()` / `as_u8()`	Signed / unsigned 8-bit integer

Method	Description
`node.as_float()`	`Option<f64>`
`node.as_bool()`	`Option<bool>` for `Bool` nodes
`node.as_str()`	`Option<&str>` for `String`, `Symbol`, or `Reference` nodes
`node.as_bytes()`	`Option<&[u8]>` for `OctetStream` nodes
`node.as_int_str(base)`	Any integer node, however wide, as digits in the given base
`datetime_fraction()`	`Option<&str>` of the verbatim ISO sub-second digits for a `date-time` written as a literal with a fraction (spec 1.1)
`node.unit_source()`	`Option<&str>` of the unit exactly as the document spelled it (library 2.0), where that differs from the canonical form — `ucum:mL{total}` for a unit of `m~L`; `None` for a node with no unit, one already written canonically, or one a unit policy converted
`node.get(key)`	Child `DomNode` by string key (struct)
`node.at(index)`	Child `DomNode` by flat index (array)
`node.len()` / `is_empty()`	Element count for struct or array nodes, `0` otherwise
`node.entries()`	`(String, DomNode)` pairs (struct nodes)
`node.array_dims()`	Number of `/`-separated dimensions (array nodes only)
`node.to_value()`	Recursively convert to a plain `Value` (drops type/unit info). A multi-row array comes back as an array of rows, empty rows included (`[]/[]` → `[[], []]`), as `parse()` gives it

The `as_*` accessors return `Option`. A node that is not that kind gives `None` rather than a zero or a panic, and `as_i64` on a `float` node is `None` rather than a truncated number.

`to_value` **fails rather than inventing a value.** Text or a key that is not UTF-8, a float or bool node without its value, or an unknown node kind is an `Error`, rather than an invented value: two distinct keys must not collapse into one.

12.4 `DomType`

```
Null=0  Int=1  Float=2  String=3  Symbol=4
Reference=5  Struct=6  Array=7  OctetStream=8  Bool=9
```

`DomType::from_raw(i32)` converts a raw C value.

12.5 Building a tree (`NewNode`)

`DomDoc::create()` makes an empty document and `NewNode`, `NewStruct` and `NewArray` the nodes to put in it: the C DOM's builder (`bvn_dom_doc_create`, `bvn_dom_node_alloc`, `bvn_dom_node_from_*`, `bvn_dom_struct_add`, `bvn_dom_array_append`, `bvn_dom_doc_add`, doc/08 §4.5). It is there for code that consumes a DOM, a test fixture say, and it has the C builder's limits: it makes structs, arrays, nulls and integers only, and a built tree **cannot be**

written out, since the C library has no DOM serialiser. To produce a document, use a `Value` or the `Writer`.

```rust
use bovnar::{DomDoc, NewArray, NewNode, NewStruct, Signedness, Unit, Value};
use num_bigint::BigInt;

fn main() -> Result<(), bovnar::Error> {
    let mut server = NewStruct::new()?;
    server.add("port", NewNode::from_u16(8080)?)?;
    server.add("tls", NewNode::null()?)?;
    let mut ports = NewArray::new()?;
    ports.push(NewNode::from_u16(80)?)?;
    ports.push(NewNode::from_u16(443)?)?;
    server.add("ports", ports)?;

    let mut doc = DomDoc::create()?;
    doc.add("server", server)?;
    let m = Unit::parse("m")?;
    let big = BigInt::from(1) << 100;
    doc.add("big", NewNode::from_bigint(&big, 128, Signedness::Auto, Some(&m))?)?;

    assert_eq!(doc.lookup("server.port").and_then(|n| n.as_u16()), Some(8080));
    assert_eq!(doc.get("big").map(|n| n.unit_str()), Some("m".to_string()));
    assert_eq!(
        doc.lookup("server.ports").map(|n| n.to_value()).transpose()?,
        Some(Value::Array(vec![Value::Int(80), Value::Int(443)]))
    );
    Ok(())
}
```

Constructor / method	Description
`DomDoc::create()`	An empty document
`doc.add(key, node)`	Add `node` (anything `Into<NewNode>`) as a top-level entry. A parsed document takes one too
`NewNode::null()`	A null node (`bvn_dom_node_alloc`)
`NewNode::from_i8(v)` … `from_u64(v)`	An integer node of type `sint:8` … `uint:64`, with no unit (`bvn_dom_node_from_i8` … `_u64`)
`NewNode::from_bigint(&value, width, signed, unit)`	An integer node of 65 to 32768 bits, with an `Option<&Unit>` (`bvn_dom_node_from_bigint`); `Signedness::Auto` is `sint` for a negative value and `uint` otherwise, as for `write_bvni`. A value that does not fit the type, another width, or a unit that is not structurally valid is `Error::Argument`
`NewStruct::new()` / `s.add(key, node)`	A struct node, and a member added to it (`bvn_dom_struct_add`)
`NewArray::new()` / `a.push(node)`	An array node, and an element appended to it (`bvn_dom_array_append`)
`len()` / `is_empty()` / `dom_type()`	Members or elements of a `NewStruct` / `NewArray`; a `NewNode`'s kind

Every constructor returns a `Result`, `Error::OutOfMemory` when the C library cannot allocate, as `Reader::new` and the `Writer` constructors do.

Ownership is a move. The C attach functions take the node they are given on every path: on success the container owns it, and on failure they destroy it. `add` and `push` take their node by value, so the type system makes that hand-over the last use: a node cannot be added twice, nor a struct filled after it was attached (a tree is built from its leaves up), and neither compiles. A node dropped before it is attached frees its subtree. All node types are `Send`.

```rust,compile_fail use bovnar::{DomDoc, NewNode};

fn main() -> Result<(), bovnar::Error> { let mut doc = DomDoc::create()?; let n = NewNode::from_i32(1)?; doc.add("a", n)?; doc.add("b", n)?; // error[E0382]: use of moved value: `n` Ok(()) }
```

````
`add` fails with `Error::Argument` for a key that is not an identifier (§5.8;
the C functions refuse only an empty one, but the path lookups could not reach
another), and with the code `DuplicateStructKey` for a key the struct or
document already has; the node is dropped then. The kinds are types, so there is
no appending to a struct or adding to an array. Only an allocation failure
inside the C call is `Error::OutOfMemory`.

**Reading it back.** A built tree reads through `DomDoc` and `DomNode` like a
parsed one, with two differences that come from the C builder: an integer node
of up to 64 bits carries no unit, and a built array records no row shape, so its
`array_dims()` is 0 and `to_value()` gives one flat array. `doc.add` takes
`&mut self`, so no `DomNode` is borrowed from the document while it grows.

Not bound: `bvn_dom_node_alloc` of the other kinds (a float, string, symbol,
reference, bool or octet-stream node it makes holds no value and cannot be given
one), `bvn_dom_strdup` (a C allocation helper; the attach functions copy the key
themselves), and `bvn_dom_get_bigint` (a borrowed `bvn_int_t`; `as_int_str()`
and `to_value()` already read an integer of any width).

---

## 13. Error handling

Every fallible call returns `bovnar::Result<T>`, whose error is `bovnar::Error`.
`Error` has one variant per failure class:

| Variant | When raised |
|---|---|
| `Error::Parse { code, position }` | Parse error in `Reader` or `DomDoc` (carries the
code and, where the library reported one, the `Position`) |
| `Error::Write { code, offset }` | Write error in `Writer` |
| `Error::Argument { code, message }` | Invalid argument passed to a helper (e.g. bad unit
string, refused unit policy) |
| `Error::OutOfMemory(call)` | The library could not allocate; names the call that failed
|
| `Error::Io { code, io_kind, message }` | The `std::io::Read` or `Write` behind
`read_reader`, `parse_reader`, `write_to`, `to_writer` or a `stream` twin, a reader's
`mirror` ([§10.2](#102-read_mem)) or the `input` or `out` of `canonicalize_reader`
([§4.10](#410-canonical-form)) failed. `code` is what the C library reports for an unread-
able fd source or unwritable fd sink (`reading_from_source_fd`, `writing_to_sink`);
`io_kind` and `message` keep what the I/O error said |

The library is part of the crate, so there is no library-not-found error. `Error`
implements `std::error::Error` and `Display`, so it works with `?` into
`Box<dyn Error>` and `anyhow`.

`kind()` gives the library's error name as an `ErrorKind` (an alias of
`ErrorCode`) wherever the library reported one — an unparsable unit or a refused
policy included — `name()` its C spelling, `code_raw()` the raw integer, and
`position()` the line, column and byte offset where the input was refused:

```rust
use bovnar::{Error, ErrorKind, Unit};

fn main() {
    let err = bovnar::parse(b".a = 1;\n.b = @;").unwrap_err();
    assert_eq!(err.kind(), Some(ErrorKind::UnexpectedInputByte));
    assert_eq!(err.name(), Some("unexpected_input_byte"));
    let pos = err.position().expect("a parse error from the reader has a position");
    assert_eq!(pos.line, 2);
    assert!(pos.column > 0 && pos.offset > 0);
    assert_eq!(pos.byte, b'@' as u32);              // the byte the error was raised on

    match Unit::parse("furlong per fortnight") {
        Err(Error::Argument { code, message }) => {
            assert!(code.is_some(), "the library said why: {message}");
````
````

```
        }
        other => panic!("expected an argument error, got {other:?}"),
    }
}
```

`position()` is `None` for errors the library locates no better than by code: the DOM parser, and the stream helpers' framing errors. `kind()` is `None` for an argument error caught before the library and for an allocation failure. `ErrorCode::name()` is the library's text for a code on its own (`bvn_error_to_string`): `ErrorCode::StringTooLong.name()` is `"string_too_long"`.

`unit_convert_factor` error semantics. The C function uses the `(ok, requires_affine)` pair to signal four distinct outcomes:

ok	requires_affine	Meaning
`true`	`false`	Multiplicative conversion; `factor` is ready to use
`true`	`true`	Both sides are affine and share their zero (e.g. °C ↔ °Re, Réaumur), so `factor` alone is correct
`false`	`true`	Affine conversion the offsets do not cancel (e.g. °C ↔ K). **`factor` is `0.0`** — not a usable multiplier
`false`	`false`	Units are dimensionally incompatible → `Error::Argument`

`unit_convert_factor` fails with `Error::Argument` **only** when both `ok=false` and `requires_affine=false`; for the other three the result's `ok()` method gives `ok`, so check it before multiplying by `factor`. It is a method rather than a field because a new public field would break code that builds or destructures `UnitConversion`; it is exact all the same, since the C function never succeeds with a factor of 0.0. When it is `false`, call `convert_value`, which handles the two-step affine path, or call `unit_to_si_factor` on both units and convert by hand. The `Unit` method `convert_factor()` returns the bare number and therefore fails for the `0.0` row instead of returning it.

Callbacks returning `false`. When an `on_verified` or `on_unverified` callback returns `false`, the C parser stops and `read_mem` returns `Error::Parse` with whatever error code the reader recorded.

Callbacks that panic. If the callback panics, the panic is caught at the FFI boundary, the C callback returns `false` to stop the parse, and the original panic is resumed from `read_mem` after the C call returns — a Rust panic never unwinds through C frames.

---

14. Threads

`Reader`, `Writer` and `DomDoc` are `Send` but not `Sync`. As `bovnar.h` (THREADING) allows, each can be moved to another thread, and distinct ones can be used on distinct threads,

but one object must not be shared: a single reader or writer holds the whole parse or serialise state of one stream.

```rust
use bovnar::{DomDoc, Writer};

fn main() -> Result<(), bovnar::Error> {
    let doc = DomDoc::parse(b".n = 7;")?;
    // A DomDoc moves to another thread; a DomNode borrows it, so it stays with its docu-
ment.
    let n = std::thread::spawn(move || doc.get("n").and_then(|n| n.as_i64())).join().un-
wrap();
    assert_eq!(n, Some(7));

    // Distinct writers on distinct threads need no locking
    let outputs: Vec<Vec<u8>> = (0..4u64)
        .map(|i| {
            std::thread::spawn(move || {
                let mut w = Writer::to_mem(4096, false).unwrap();
                w.write_uint("i", i, 8, bovnar::Base::DECIMAL).unwrap();
                w.finish().unwrap();
                w.get_output().to_vec()
            })
        })
        .collect::<Vec<_>>()
        .into_iter()
        .map(|h| h.join().unwrap())
        .collect();
    assert_eq!(outputs.len(), 4);
    Ok(())
}
```

The `live` readers (§4.9) are `Send` and not `Sync` too: each owns one C object for its stream.

`parse`, `parse_typed`, `to_string`, their `_reader` / `write_to` twins and the `units`, `currency` and `json` functions are free functions that build their own reader, writer or DOM for each call, so no two calls share an object and they can run on several threads at once.

---

15. Running the test suite

```sh
cd rust

# Run everything
cargo test

# One test file
cargo test --test test_units

# The documentation examples of rust/README.md
cargo test --doc
```

`cargo test` runs the unit tests in `src/`, the integration tests in `tests/` (one file per topic — `test_units.rs`, `test_dom.rs`, `test_stream.rs` and so on — plus `smoke.rs`) and the documentation examples. The ABI check is three unit tests in `src/ffi.rs`, because the C

structs it compares against are private. `test_codec_alloc.rs` installs a counting global allocator to show that the codec allocates nothing per message, so it is a test binary of its own; `test_live_alloc.rs` does the same for the `live` readers and the `std::io` twins.

`test_writers_agree` compares `to_string` with the C CLI's `pretty-print`, so it is `#[ignore]`d unless the CLI is built at `<repo>/build/bovnar`:

```
cmake -B build . && cmake --build build --target bovnar
cargo test --test test_writers_agree -- --ignored
```

The C library's own suite runs unchanged through CMake: `cmake -B build . && cmake --build build && ctest --test-dir build`. That suite also parses every `bovnar` code block of this document (`ctest -R bvnr_doc`).

`rust/fuzz/` holds cargo-fuzz targets (nightly toolchain): `parse` (the streaming parsers and a write-back round trip), `dom` (parse and walk every node), `json_roundtrip` (`from_json` and `to_json` until the JSON is stable), `codec` (`MuxReader` and `FrameReader` fed in pieces, against themselves and against the C library's `mux_load` and `load_documents`) and `live` (the fed readers of `bovnar::live` in pieces, against `Reader::read_mem`, `mux_load`, `codec::FrameReader` and `load_documents`). Instrument the C library too by building it with clang:

```
cd rust/fuzz
CC=clang CFLAGS="-fsanitize=fuzzer-no-link,address" \
    cargo +nightly fuzz run -O parse -- -max_total_time=600
```

---

16. Crate layout

```
rust/
├── Cargo.toml        # version = the C library's version; publish = false (see pack-
age.py)
├── build.rs          # compiles the bundled c/, or in the repository ../src after running
the generators
├── package.py        # stages the crate with the C sources bundled, for crates.io
├── src/
│   ├── lib.rs        # parse() / parse_typed() / to_string() / write_array() — public API
│   ├── ffi.rs        # the C declarations; the only `unsafe` code
│   ├── bvnfloat.rs   # BvnFloat: arbitrary-precision float + IEEE binary/decimal/fixed
encoders
│   ├── codec.rs      # MuxWriter, MuxReader, FrameReader — the pure-Rust framing codec
│   ├── currency.rs   # ISO 4217 / crypto currency metadata (read from the C catalogue)
│   ├── decimal.rs    # Decimal: an exact decimal number (sign, coefficient, exponent)
│   ├── dom.rs        # DomDoc, DomNode, DomType, NewNode — random-access DOM and its
builder
│   ├── enums.rs      # Rust enums mirroring the C enums
│   ├── error.rs      # Error, ErrorKind, Position
│   ├── io.rs         # a caller's std::io::Read as a callback source for read_reader /
parse_reader
│   ├── json.rs       # from_json / to_json / detect (the CLI's `bovnar convert`)
│   ├── literals.rs   # the C validators, literal parsers/formatters,
parse_type_annotation
│   ├── live.rs       # DocumentFeed, ValueFeed, EmbeddedFeed, FrameParser, FramePull, De-
mux — fed reading through C
│   ├── quantity.rs   # Quantity — typed, unit-annotated scalar for lossless round-trips
│   ├── reader.rs     # Reader, ReadOptions, ReadHandlers, EventData, EventPayload, Unit-
Policy, UnitRule, canonicalize
│   ├── stream.rs     # framing, octet multiplexing, document-in-document
│   ├── timekeeping.rs # civil time and calendars: GregorianDate, CivilDateTime, Epoch
│   ├── unit.rs       # Unit, UnitComponent — a unit with equality, hashing and its opera-
tions
│   ├── units.rs      # unit_to_si_factor, unit_convert_factor, etc.
│   └── writer.rs     # Writer, WriteOptions, WriteObserver, Base, Signedness, EmitSpec
├── tests/            # one integration-test file per topic, plus smoke.rs
├── benches/          # codec.rs, live.rs: per-message figures of the codec and the fed
readers (std only)
└── fuzz/             # cargo-fuzz targets
```

Module	Contents
`src/ffi.rs`	the C declarations and the mirrored C structs
`src/reader.rs`	`Reader`, `ReadOptions`, `ReadHandlers`, `EventData`, `EventPayload`, `UnitPolicy`, `UnitRule`, `canonicalize`, `canonicalize_reader`
`src/writer.rs`	`Writer`, `WriteOptions`, `WriteObserver`, `Base`, `Signedness`, `EmitSpec`
`src/dom.rs`	`DomDoc`, `DomNode`, `DomType`, and the builder's `NewNode`, `NewStruct`, `NewArray`
`src/literals.rs`	the C validators and literal helpers, `parse_type_annotation`, `TypeAnnotation`
`src/enums.rs`	the Rust enums behind the C enums, and `Exponent`
`src/error.rs`	`Error`, `ErrorKind`, `Position`
`src/unit.rs`	`Unit`, `UnitComponent`
`src/units.rs`	the unit functions: conversion, dimension vectors, profile notations

Module	Contents
`src/quantity.rs`	`Quantity`
`src/bvnfloat.rs`	`BvnFloat` and the IEEE / fixed-point encoders
`src/decimal.rs`	`Decimal`, the exact decimal value
`src/currency.rs`	currency metadata
`src/stream.rs`	framing, multiplexing and document-in-document, with their `Read` / `Write` twins
`src/timekeeping.rs`	civil time and calendars: `GregorianDate`, `CivilDateTime`, `Epoch` and the functions of `bvn_datetime.h` / `bvn_gregorian_date.h`
`src/codec.rs`	the incremental framing codec in pure Rust: `MuxWriter`, `MuxReader`, `FrameReader`, the frame writers
`src/live.rs`	fed reading through the C library: `DocumentFeed`, `ValueFeed`, `ValueBuilder`, `EmbeddedFeed`, `FrameParser`, `FramePull`, `Demux`
`src/io.rs`	the `std::io::Read` helpers behind the `_reader` entry points (the sinks of the `_to` ones are in `ffi.rs` and `writer.rs`)
`src/json.rs`	JSON ⇄ BVNR, as the CLI's `bovnar convert` does it
`src/lib.rs`	`parse`, `parse_typed`, `to_string`, the reader and writer twins, `write_array` and the crate-root re-exports

Out of scope. The crate has no bridges to third-party unit or array libraries. It takes no raw file descriptors or paths either: the entry points that read or write a whole document are `std::io::Read` / `std::io::Write` ones (§4.7), and a file is an opened `std::fs::File` passed to them.

---

17. Enums and `BaseUnit`

The C enums are Rust enums with CamelCase variants (the C member `METER` is `BaseUnit::Meter`): `Event`, `TokenType`, `ValueTypeFamily`, `ErrorCode` (also `ErrorKind`), `BaseUnit`, `PrefixSystem`, `SIPrefix` and `IECPrefix`. Raw C values convert with `TryFrom<i32>`, which refuses an unknown value with an `UnknownValue` error, and back with `i32::from` (or `as i32`). Each also has `from_raw`, which returns an `Option`.

`Exponent` and `Base` are open ranges, so they are checked newtypes with the common values named (`Exponent::LINEAR`, `Exponent::NEG_SQUARE`, …). `Exponent` runs from `EXPONENT_MIN` (−100) to `EXPONENT_MAX` (100).

```rust
use bovnar::{BaseUnit, Exponent, SIPrefix, Unit};

fn main() -> Result<(), bovnar::Error> {
    let km = Unit::si(BaseUnit::Meter, SIPrefix::Kilo, Exponent::LINEAR);
    assert_eq!(km.to_string(), "k~m");
    assert_eq!(Unit::parse("k~m")?, km);
    let c = km.components()[0];
    assert_eq!((c.base, c.prefix), (BaseUnit::Meter, SIPrefix::Kilo.into()));
    assert_eq!(BaseUnit::try_from(i32::from(BaseUnit::Meter)), Ok(BaseUnit::Meter));
    assert_eq!(BaseUnit::Meter.name(), "METER");
    Ok(())
}
```

The `BaseUnit` enum mirrors the C `value_base_unit_e`. See doc/05 §11.1 for the block layout — an id's leading two digits name the vocabulary it comes from. The native units are 100000–100263 (`UNIT_NATIVE_FIRST` … `UNIT_NATIVE_LAST`) and the currencies 900000–900215 (`CURRENCY_FIRST` … `CURRENCY_LAST`), with the CamelCase spelling of each member (`POUND_FORCE` is `PoundForce`, `GERMAN_MILE` is `GermanMile`):

Range	Members
0	`None`
100000–100001	`Bit`, `Byte`
100002–100008	SI base units (`Second` … `Candela`)
100009–100027	Named SI derived units (`Hertz` … `Katal`)
100028–100043	Non-SI accepted units (`Liter` … `Year`)
100044–100053	Imperial/US length (`Inch` … `Fathom`)
100054–100061	Imperial/US mass (`Pound` … `Carat`)
100062	`Fahrenheit`
100063–100066	Pressure (`Atmosphere` … `Psi`)
100067–100070	Energy (`Calorie` … `Therm`)
100071	`Horsepower`
100072–100074	Force (`PoundForce`, `Dyne`, `Kip`)
100075	`Knot`
100076–100084	US volume (`Gallon` … `Barrel`)
100085–100086	Area (`Acre`, `Barn`)
100087–100089	Angle (`Arcminute`, `Arcsecond`, `Grad`)
100090–100097	CGS units (`Poise` … `Galileo`)
100098–100100	Radiation (`Curie`, `Roentgen`, `Rem`)
100101–100102	Logarithmic (`Neper`, `Decibel`)
100103	`Rankine`
100104	`Slug`

Range	Members
100105	`Thou`
100106–100108	UK imperial volume (`PintUk`, `FluidOunceUk`, `QuartUk`)
100109–100110	Electrical power (`Var`, `VoltAmpere`)
100111	`KilogramForce`
100112	`InchHg`
100113	`Rpm`
100114	`FootPound`
100115–100116	Mass additional (`Dram`, `Pennyweight`)
100117–100118	Length additional (`Chain`, `Rod`)
100119–100120	Volume additional (`Gill`, `GillUk`)
100121	`StandardGravity`
100122	`MetricHorsepower`
100123	`Revolution`
100124–100125	Time additional (`Month`, `Fortnight`)
100126	`AtmosphereTechnical`
100127–100128	Textile linear density (`Tex`, `Denier`)
100129–100132	Apothecary/dry volume (`FluidDram`, `Minim`, `Peck`, `Bushel`)
100133–100145	Historical German units (`Pfund`, `Zentner`, `Doppelzentner`, `Lot`, Prussian line/zoll/fuss/elle/rute, `Klafter`, `GermanMile`, `Morgen`, `Scheffel`)
100146–100152	Additional physical units (`SurveyFoot`, `League`, `Cable`, `Hand`, `Quintal`, `Scruple`, `Baud`)
100153–100156	Temperature scales (`Delisle`, `NewtonTemp`, `Reaumur`, `Romer`)
100157–100162	Ratio/proportion units (`Percent`, `PerMille`, `PerMyriad`, `PerCentMille`, `Ppm`, `Ppb`)
100163	`PhScale`
100164–100165	Named speed units (`MilePerHour`, `KilometerPerHour`)
100166–100170	Water hardness (`GermanHardness`, `EnglishHardness`, `FrenchHardness`, `RussianHardness`, `AmericanHardness`)
100171–100172	Concentration (`Val`, `GrainsPerGallon`)
100173–100179	Turbidity, salinity and conductivity (`TurbidityNtu`, `TurbidityFnu`, `PracticalSalinity`, `ConductivityFactor`, `TurbidityFtu`, `TurbidityFau`, `TurbidityJtu`)
100180–100185	Temperature differences (`DeltaKelvin`, `DeltaFahrenheit`, `DeltaDelisle`, `DeltaNewtonTemp`, `DeltaReaumur`, `DeltaRomer`) — `Δ°C` shares `DeltaKelvin` and `Δ°Ra` shares `DeltaFahrenheit`

Range	Members
100186–100191	The six units the profiles needed (`MeterWater`, `CalorieIt`, `BtuTh`, `TroyPound`, `ApothecaryDram`, `LongHundredweight`)
100192–100200	US survey lengths (`SurveyInch` … `SurveyAcre`)
100201–100203	Typographic lengths (`Point`, `Pica`, `Line`)
100204–100209	US dry volumes and the trade measures (`DryGallon`, `DryQuart`, `DryPint`, `BoardFoot`, `Cord`, `SurveyAcreFoot`)
100210–100214	`Darcy`, `ThermEc`, `RefrigerationTon`, `Dobson`, `Shake`
100215–100216	The ratios below ppb (`Ppt`, `Ppq`) — spelled `pptr` and `ppq` in documents
100217–100225	Nine units two or three vocabularies define (`PrintersPoint`, `PrintersPica`, `HorsepowerElectric`, `HorsepowerBoiler`, `Abvolt`, `AssayTon`, `BushelUk`, `Clo`, `Debye`)
100226–100236	Calendar and sidereal time (`YearTropical`, `YearGregorian`, `YearSidereal`, `YearCommon`, `MonthSynodal`, `MonthTropical`, `MonthSidereal`, `DaySidereal`, `HourSidereal`, `MinuteSidereal`, `SecondSidereal`)
100237–100243	CGS electrostatic (`Statvolt`, `Statampere`, `Statcoulomb`, `Statfarad`, `Statohm`, `Stathenry`, `Statsiemens`)
100244–100246	Luminance (`Lambert`, `Apostilb`, `Footlambert`)
100247–100249	Water-vapour permeance (`Perm0c`, `Perm23c`, `PermMetric`)
100250–100255	Heat conventions (`CalorieMean`, `Calorie15c`, `Calorie20c`, `Btu59f`, `Btu60f`, `BtuMean`)
100256–100261	`ElementaryCharge`, `Spere`, `CircularMil`, `UnitPole`, `HorsepowerWater`, `MonthGregorian`
100262–100263	CGS electrostatic magnetics (`Statweber`, `Stattesla`)
900000–900165	ISO 4217 fiat currencies (`Aed` … `Zwl`), block 90 — starts at `CURRENCY_FIRST`
900166–900215	Cryptocurrencies (`Btc` … `Rune`), the rest of block 90 — ends at `CURRENCY_LAST`

There is **no sentinel member** one past the highest id: the id space is blocked and sparse, so such a number would only invite a bounds check it cannot support. Use `UNIT_NATIVE_FIRST` / `UNIT_NATIVE_LAST` / `CURRENCY_FIRST` / `CURRENCY_LAST`, or ask `currency::is_currency`.

> **The profiles' opaque units are in this enum.** The C `value_base_unit_t` has a block per profile for the profile-only units, and as shipped two of them have members: UCUM's arbitrary atoms (200000–200039, `UcumIu` for `ucum:[IU]`) and UN/ECE's package and count codes (300000–300024, `UneceOne` … `UneceVial`; see doc/11 §7.1). They have no native spelling, so a unit carrying one renders in its profile notation: `unit_to_str` returns `"unece:XBX"`. `BaseUnit::try_from` accepts the raw base of a profile-only unit.

> **Note on `Cup_`:** the Cuban-Peso currency is exposed as `BaseUnit::Cup_` (trailing underscore), not `BaseUnit::Cup`. The plain name `Cup` is the US-cup volume unit (enum value 100080); since enum variant names must be unique, the currency at value 900033 takes the suffixed name. This affects the variant name only — the `.bvnr` wire token is still the bare uppercase `CUP`, and case stays load-bearing (`cup` = volume, `CUP` = currency).

---

18. Spec 1.1 additions

These bindings target the **Bovnar spec 1.2**; spec 1.0 remains the frozen baseline, and the crate's own version (2.1.0) is a separate number (`bovnar::version()` against `bovnar::spec_version()`). Spec 1.2's one addition, the unit-profile notation, is covered in §5.1: it is gated on a `#!bovnar 1.2` directive, which `to_string()` prepends when a kept spelling or a profile-only unit needs it. The 1.1 features (all gated on a `#!bovnar 1.1` directive — an unversioned document is treated as 1.0) are exposed as:

- **Version:** `bovnar::version()` (library version string), `bovnar::version_number()` (the same as one number, `major * 10000 + minor * 100 + patch`: 20100 for 2.1.0), `bovnar::spec_version()` → `(major, minor)` of the highest spec understood, and `bovnar::peek_version(data)` → the `(major, minor)` a document declares, or `None`. `Reader::declared_version()` gives the same after a read; `ReadOptions { strict_version: true, .. }` rejects a too-new version (`ErrorCode::UnsupportedSpecVersion`), from a `Reader`, a feed, or `parse_with_options`.
- **Richer escapes:** `\u{…}` and `\xHH` in string literals are decoded by the C reader, so `parse` returns the resulting text transparently. A surrogate / out-of-range `\u` is `ErrorCode::InvalidCodepoint`.
- **Datetime family:** `parse` decodes a `<datetime:width,epoch>` value to a plain `Value::Int` (signed epoch seconds). With `parse_typed` it is a `Quantity` whose `epoch_name()` (`"unix"`, `"tai"`, …) and `epoch_mjd()` recover the epoch, and `to_civil()` its UTC date and time (§6.5); `bovnar::datetime_epoch_index(name)` goes the other way, to the index a datetime `ValueTypeSpec` holds in `base`, `None` for an unknown name. `to_string()` emits the `<datetime:…>` annotation and prepends `#!bovnar 1.1` automatically when the value contains a datetime. With `parse_typed` the sub-second fraction of an ISO-8601 literal (spec 1.1) is preserved on the `Quantity` (`q.frac`) and re-emitted, so a `parse_typed` → `to_string()` round trip is lossless (`…T12:00:00.5Z` survives verbatim). The plain (non-typed) value is only the whole-second integer carrier, so a non-typed `parse` → `to_string()` drops the fraction; to read it explicitly use the DOM tier (`datetime_fraction()`) or the streaming reader (`d.frac` in a callback). `ValueTypeFamily::Datetime` is the family enum member.
- **Reference array indexing:** `&.matrix[0][1]` paths are stored verbatim and resolved by `DomDoc::lookup("matrix[0][1]")` at the DOM layer.

```rust
use bovnar::{Value, parse_typed, peek_version, spec_version, to_string};

fn main() -> Result<(), bovnar::Error> {
    assert_eq!(spec_version(), (1, 2));
    let bvnr = b"#!bovnar 1.1\n.t = <datetime:64,unix> 2026-01-01T12:00:00.5Z;";
    assert_eq!(peek_version(bvnr), Some((1, 1)));

    let doc = parse_typed(bvnr)?;
    let Value::Struct(fields) = &doc else { unreachable!() };
    let Value::Quantity(q) = &fields[0].1 else { unreachable!() };
    assert_eq!(q.epoch_name().as_deref(), Some("unix"));
    assert_eq!(q.frac.as_deref(), Some("5"));
    // the fraction survives a typed round trip
    assert_eq!(parse_typed(&to_string(&doc, true)?)?, doc);
    Ok(())
}
```

`ErrorCode` adds `InvalidSpecVersion` (42), `UnsupportedSpecVersion` (43), `InvalidCodepoint` (44), `InvalidDatetimeLiteral` (45), and `DatetimeLiteralUnsupportedEpoch` (46).

---

19. License

MIT, like the C library — see LICENSE.

The crate compiles `libbvnr` into your binary, and that embeds Bovnar's unit-profile tables, which carry identifier strings from UCUM, QUDT, OM 2, UDUNITS-2, the CF standard name table and UN/ECE Recommendations 20 and 21. Those identifiers belong to their publishers and the MIT grant does not extend to them; the notices are at `THIRD_PARTY_NOTICES.md`.

See also

- Read & Write API — the C reader, writer, and DOM these bindings wrap
- Streaming, Framing & Multiplexing — the framing, multiplexing and embedding behind `bovnar::stream`
- Specification — the format the crate reads and writes
- Unit & Currency Reference — the unit model behind `Unit` and `Quantity`
- Tutorial — the document syntax, by example

End of Bovnar — Rust Bindings (Bovnar spec 1.2).