BOVNAR · V1.1

Bovnar Streaming, Framing & Multiplexing

Bovnar (BVNR) v1.1 documentation · Streaming, Framing & Multiplexing · 2026-06-21

This document describes four capabilities layered **on top of** the Bovnar lexer/validator event API, beside the DOM. None of them change the frozen 1.0 grammar or wire format (see doc/1_bovnar_spec.md); they are applications built on the SAX-style event pipe and the octet-stream value form.

#	Capability	Where it lives	Header
1	Endless streaming	core policy knob	`bovnar.h`
2	Multi-document framing	transport beside the lexer	`bovnar_stream.h`
3	Octet multiplexing	convention inside one octet stream	`bovnar_stream.h`
4	Document-in-document	a document embedded as octet payload	`bovnar_stream.h`

The mental model: the **reader** turns bytes into a `bvnr_event_t` stream and the **writer** turns events back into bytes. The DOM is one consumer of that pipe; everything here is another. Octet streams (spec §9) are the binary substrate — opaque, chunked, UTF-8-validation-suspended byte regions — and three of the four capabilities are sub-protocols carried in or around them.

---

1. Endless streaming ($2^{64}-1$)

Every byte/offset/line/column counter in the core is 64-bit, so a single never-ending document — or a single octet stream of unbounded length — is limited only by one policy knob: `max_file_size` on `bvnr_read_flags_t` / `bvnr_write_flags_t`.

```
#define BVNR_FILESIZE_UNLIMITED  0   /* in bovnar.h */
```

Semantics — `0` **is unlimited and the default**:

`max_file_size` value	Effect
`0` (= `BVNR_FILESIZE_UNLIMITED`, the default)	**No cap** — endless, no accumulated size limit
any positive N	Cap at N bytes (`error_file_too_long` beyond)

```
bvnr_read_flags_t fl = {
    .on_verified   = my_cb,
    .max_file_size = BVNR_FILESIZE_UNLIMITED,   /* endless */
};
```

A long-lived document is processed incrementally: process each value in the `on_verified`/`on_unverified` callback and retain nothing, and memory stays flat regardless of stream length. (The DOM builder, by contrast, retains a tree and is therefore not the tool for an endless stream.)

---

2. Multi-document record framing

The grammar has no in-band document separator: a reader ends one document at EOF. To carry a **sequence of independent documents** over one byte stream (file, socket, pipe) without closing it, wrap each in a length-prefixed frame. The framing lives beside the lexer; each payload is a complete, standalone document parsed by an ordinary reader.

Wire format

```
frame   = "BVF1"            ; 4-byte magic
          payload_len       ; uint64, little-endian
          payload[payload_len]
stream  = { frame }         ; terminated by EOF on a frame boundary
```

A clean EOF before any header byte ends the sequence successfully; an EOF part-way through a header or payload is a truncation error.

API

```
/* Producer */
bool bvnr_frame_write(bvnr_sink_t* sink, const void* doc, uint64_t doc_len);

/* Consumer */
typedef struct {
    bvnr_read_flags_t* flags;            /* per-document reader callbacks/limits */
    void*              userdata;
    bool (*on_document)(void* ud, uint64_t index, bool ok, error_code_t err);
    bool               continue_past_failed; /* keep going after a failed document */
    uint64_t           max_document_size;    /* 0 => BVNR_DOC_DEFAULT_MAX (256 MiB) */
} bvnr_doc_stream_opts_t;

bool bvnr_doc_stream_read(bvnr_source_t* src,
                          const bvnr_doc_stream_opts_t* opts,
                          uint64_t* out_count);
```

`bvnr_doc_stream_read` reads frames until EOF, parsing each payload with a fresh reader (reusing one handle internally) and invoking `on_document` after each. It returns false on a framing/IO error, or on the first per-document parse failure unless `continue_past_failed` is set. Each frame payload is buffered in RAM before parsing (the buffer grows with the bytes

that actually arrive, so a crafted oversized length cannot force a large up-front allocation), bounded by `max_document_size`. Unlike `max_file_size`, this framing guard stays finite when `0` (it then selects `BVNR_DOC_DEFAULT_MAX`, 256 MiB); set an explicit huge cap (e.g. `UINT64_MAX`) to lift it.

Two **orthogonal** error controls — because each document occupies its own length-delimited frame, the two are independent:

Control	Scope	Effect
`flags->continue_on_error`	within a document	resync/recover from errors inside one document
`continue_past_failed`	across documents	report a failed document and advance to the next frame instead of aborting

So you can parse each document **strictly** (`continue_on_error = false`) yet **resiliently** process every frame (`continue_past_failed = true`), reporting the bad ones via `on_document` — the natural mode for a log of independent records.

CLI

```
bovnar frames pack a.bvnr b.bvnr c.bvnr > docs.bvf   # wrap each file in a frame
bovnar frames list docs.bvf                          # list documents + status
bovnar frames list -                                 # read the frame stream from stdin
```

---

3. Octet multiplexing (out-of-band channels, cross-chunk)

Many logical channels interleaved inside a **single octet-stream value**. The channel id and message framing are a convention interpreted above `ev_data`, so to the lexer these are ordinary `0x01` data chunks and `bvnr_data_t` needs no channel field. The 1.0 wire format is untouched.

Wire convention (two layers)

```
per octet chunk:    varint(channel)  fragment_bytes
per channel:        concat(fragments in order) = the channel's logical stream,
                    a sequence of length-prefixed messages:
                        varint(msg_len)  msg_bytes
```

- `varint` is unsigned LEB128 (1 byte for channels/lengths < 128; up to 10 bytes).
- A message is **length-prefixed**, so it may span any number of chunks, and channels may interleave freely at chunk granularity — the demultiplexer reassembles each channel independently. Both the message data **and** its length varint may straddle chunk boundaries (the demux accumulates a partial length varint per channel).

- `BVNR_MUX_MAX_MESSAGE` is a conservative **single-chunk guarantee**, not an enforced cap: any message of at most that many bytes is emitted by `bvnr_mux_send` in one chunk regardless of channel id or length magnitude (it subtracts the maximum 10+10 varint bytes from the 64 KiB chunk). Larger messages are simply split; nothing rejects them.
- **The one wire requirement:** the per-chunk **channel** varint — the routing prefix the demux needs to attribute a chunk — is whole within its chunk. `bvnr_mux_send` always satisfies this (the channel id is ≤ 10 bytes and leads every chunk).

API

```
/* Producer */
bool bvnr_mux_begin(bvnr_writer_t* w, const char* key);
bool bvnr_mux_send (bvnr_writer_t* w, uint64_t channel,
                    const void* data, uint64_t len);   /* split as needed */
bool bvnr_mux_end  (bvnr_writer_t* w);

/* Consumer */
typedef bool (*bvnr_mux_on_msg_fn)(void* ud, uint64_t channel,
                                   const uint8_t* data, uint64_t len);

bvnr_demux_t* bvnr_demux_create(bvnr_mux_on_msg_fn on_message,
                                void* userdata, uint64_t max_message);
void          bvnr_demux_destroy(bvnr_demux_t* dm);
bool          bvnr_demux_set_key (bvnr_demux_t* dm, const char* key);
bool          bvnr_demux_on_event(void* demux, bvnr_event_t ev, bvnr_data_t* d);
error_code_t  bvnr_demux_error(const bvnr_demux_t* dm);
```

Install `bvnr_demux_on_event` as `bvnr_read_flags_t.on_verified` with `userdata` set to the `bvnr_demux_t*`. It peels the channel varint off each chunk, reassembles per-channel messages (owning a buffer per active channel), and invokes `on_message` for each complete message. A malformed varint, an oversized declared length (> `max_message`, default 64 MiB), or an allocation failure latches `bvnr_demux_error` and aborts the read. If `on_message` returns false the read is aborted and `bvnr_demux_error` reports `error_scanner_callback_failed` (distinct from the desync code, so a deliberate stop is distinguishable from a framing error).

Key scoping. By default the demux treats every octet stream in the document as multiplexed. Call `bvnr_demux_set_key(dm, "mux")` so it only demuxes streams opened under that key — a document can then mix one mux stream with ordinary binary octet payloads (those under other keys are ignored, not misread as framing). Pass `NULL` to clear the filter.

Memory. The demux keeps one reassembly buffer per channel it has seen; each grows to that channel's largest message and is retained for reuse until `bvnr_demux_destroy`, so steady-state memory is the sum of per-channel high-water marks (bounded by 4096 channels). Channels are not reclaimed individually.

```
bvnr_demux_t* dm = bvnr_demux_create(on_message, ctx, 0 /* default cap */);
bvnr_read_flags_t fl = { .on_verified = bvnr_demux_on_event, .userdata = dm };
/* ... open_read_* + bvnr_read ... */
if (bvnr_demux_error(dm) != error_none) { /* desync */ }
bvnr_demux_destroy(dm);
```

CLI

```
bovnar mux pack 1:req.bvnr 42:log.bvnr > mux.bvnr   # files onto channels 1 and 42
bovnar mux list mux.bvnr                            # channel <n>: <len> bytes
bovnar mux pack 7:a.bvnr | bovnar mux list -        # pipe producer to consumer
```

---

4. Document-in-document

Embed a complete Bovnar document as the octet-stream payload of a key. The outer parser treats the inner document as opaque octets; only an application that knows the key holds a nested document re-feeds those bytes to a sub-reader. This is the natural recursion primitive — no envelope type, just the octet-stream value form.

```
/* Producer: emits  .<key> = <octets of doc>;  (64 KiB chunks) */
bool bvnr_embed_document(bvnr_writer_t* w, const char* key,
                         const void* doc, uint64_t doc_len);

/* Consumer: parse the embedded bytes with an ordinary reader (recursion). */
bool bvnr_parse_embedded(const void* doc, uint64_t doc_len,
                         bvnr_read_flags_t* flags);
```

To recurse while reading the outer document, accumulate the octet chunks for the known key between `ev_octet_stream_start` and `ev_octet_stream_end`, then call `bvnr_parse_embedded` on the accumulated bytes with the nested callbacks:

```
case ev_octet_stream_end:
    if (in_payload)
        bvnr_parse_embedded(buf, len, &inner_flags);   /* parses the nested doc */
    break;
```

The inner callbacks may themselves embed/extract further documents, so nesting is arbitrarily deep (bounded by your own recursion budget).

Python bindings

All four capabilities are exposed through the `bovnar.stream` module (a thin ctypes layer over the C functions, so framing, varint reassembly and the strict-vs-resilient document policy behave identically):

```python
import bovnar
from bovnar import stream

# Multi-document framing
blob = stream.dump_documents([{"id": 1}, {"id": 2}])     # list[dict] -> bytes
docs = stream.load_documents(blob)                        # bytes -> list[dict]
docs = stream.load_documents(blob, continue_past_failed=True)  # bad docs -> None

# Octet multiplexing (channel, payload) — any size, interleaved
m   = stream.mux_dump([(1, b"hello"), (42, b"world"), (1, b"x" * 200000)])
out = stream.mux_load(m)                                  # [(1, b"hello"), ...]

# Document-in-document
outer = stream.embed_document(bovnar.dumps({"v": 1}), key="payload")
inner = stream.parse_embedded(bovnar.loads(outer)["payload"])   # {"v": 1}
```

`stream.BVNR_FILESIZE_UNLIMITED` (`== 0`) selects endless mode for **`max_file_size`** only. Note it does **not** lift the framing/mux guards: for `max_document_size` and `max_message`, `0` selects their finite defaults (`BVNR_DOC_DEFAULT_MAX` / `BVNR_MUX_DEFAULT_MAX`), so pass an explicit large value there to raise them. See Python Bindings and `python/tests/test_stream.py`.

Composition

The four compose cleanly because they live at different layers:

- A **frame stream** (2) can carry documents that each **embed** sub-documents (4).
- A **multiplexed** octet stream (3) is itself just one value in a document, which can be one frame in a frame stream (2).
- Any of the above is **endless** (1) by default (`max_file_size` 0 == `BVNR_FILESIZE_UNLIMITED`); set a positive `max_file_size` to impose a cap.

Because each is a thin application over the public event API, you can also write your own consumer/producer at the same seam without touching the core.

See also

- Specification §9 — Octet Streams
- Read & Write API — the underlying event interface
- `include/bovnar_stream.h` — full declarations and per-function contracts
- `tests/bovnar_stream_test.c` — round-trip tests for all four capabilities
- `python/bovnar/stream.py` — Python bindings; `python/tests/test_stream.py`