BOVNAR · V1.1

Bovnar — Read & Write API

Bovnar (BVNR) v1.1 documentation · Read & Write API · 2026-06-21

> **Spec version:** 1.1

This document covers every function you need to read and write Bovnar streams, in the order you call them. Nothing else is included.

The writer uses the same event/data model as the reader — `bvnr_event_t` and `bvnr_data_t` — so the two APIs are deliberately symmetric. Learn one, the other follows.

Contents

12. `bvnr_write_bvni` / `bvnr_write_bvni_unit`
13. `BVN_TYPE_FLOAT_BASE`

Shared

1. `bvnr_write_type_annotation`
2. `bvn_parse_unit`
3. `bvn_unit_to_string`
4. `bvn_error_to_string`

Reader

---

1. `bvnr_reader_create` / `bvnr_reader_destroy`

```
bvnr_reader_t *bvnr_reader_create(void);
void           bvnr_reader_destroy(bvnr_reader_t *r);
```

Allocate and free a reader on the heap. `bvnr_reader_create` returns `NULL` if allocation fails. Always pair with `bvnr_reader_destroy` when done.

```
bvnr_reader_t *r = bvnr_reader_create();
if (!r) { perror("alloc"); exit(1); }

/* ... open, read ... */

bvnr_reader_destroy(r);
```

---

2. `bvnr_source_from_fd`

```
void bvnr_source_from_fd(bvnr_source_t *s, int fd);
```

Initialise the source `s` to read from an open, readable POSIX file descriptor. The caller retains ownership of `fd`; the library will not close it.

- `s` — pointer to a caller-allocated `bvnr_source_t` (stack is fine).
- `fd` — any readable descriptor: a file, a pipe, a socket.

```
bvnr_source_t src;
int fd = open("config.bvnr", O_RDONLY);
if (fd < 0) { perror("open"); exit(1); }

bvnr_source_from_fd(&src, fd);
/* pass &src to bvnr_open_read_source */
```

---

3. `bvnr_source_from_mem`

```
void bvnr_source_from_mem(bvnr_source_t *s, const void *buf, uint64_t len);
```

Initialise the source `s` to read from an in-memory buffer. The buffer must remain valid for the duration of `bvnr_read`. No copy is made.

- `buf` — pointer to the Bovnar data.
- `len` — number of bytes in the buffer.

```
static const char payload[] =
    ".host = \"localhost\";\n"
    ".port = <uint:16> 8080;\n";

bvnr_source_t src;
bvnr_source_from_mem(&src, payload, sizeof(payload) - 1);
```

---

4. `bvnr_open_read_source`

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

Attach the source to the reader and configure it. Must be called before `bvnr_read`. Returns `false` on invalid arguments (`error_invalid_argument`).

- `r` — reader obtained from `bvnr_reader_create`.
- `src` — source initialised with one of the `bvnr_source_from_*` functions.
- `src_mirror` — optional sink that mirrors the raw input bytes as they are consumed (useful for debugging; pass `NULL` in production).
- `options` — configuration struct. Zero-initialise to get all defaults. The most important fields are:

```
typedef struct bvnr_read_flags_s {
    void  *userdata;
    bool (*on_unverified)(void *userdata, bvnr_event_t, bvnr_data_t *);
    bool (*on_verified)  (void *userdata, bvnr_event_t, bvnr_data_t *);
    bool   continue_on_error;
    bvnr_on_error_fn on_error;
    uint64_t max_file_size;        /* 0 → unlimited / endless (default); set positive to
cap */
    uint8_t  max_struct_nesting;   /* 0 → 64 internal default; hard cap 255 */
    uint8_t  max_array_nesting;    /* 0 → 64 internal default; hard cap 255 */
    /* ... other size limits ... */
} bvnr_read_flags_t;
```

`on_verified` is the callback you will implement almost always. `on_unverified` fires before semantic validation — use it only for diagnostics or partial inspection. Both callbacks must return `true` to continue parsing, `false` to abort (sets `error_scanner_callback_failed`).

The `options` pointer is not stored; the struct is read during `bvnr_open_read_source` only, so it may live on the stack.

Reader default limits. When a `bvnr_read_flags_t` field is set to `0`, the reader substitutes an internal default. For the nesting fields, the default is **64** (not 255); the hard maximum is 255. For `max_array_items` and `max_text_bytes`, the default is **2 147 483 647** — permissive but finite. **`max_file_size` differs: `0` means unlimited / endless** (no byte-count cap accumulated), which is the default so endless streaming works out of the box. Setting `max_file_size` explicitly to `16777216` (16 MiB) is recommended for production.

```
static bool on_event(void *ud, bvnr_event_t ev, bvnr_data_t *d)
{
    /* handle ev / d ... */
    return true;
}

bvnr_read_flags_t opts = {
    .on_verified   = on_event,
    .userdata      = &my_ctx,
    .max_file_size = 16777216,  /* 16 MiB cap */
    /* max_array_nesting: 0 → 64 internal default; hard cap 255 */
};

if (!bvnr_open_read_source(r, &src, NULL, &opts)) {
    fprintf(stderr, "open failed\n");
    return -1;
}
```

---

5. `bvnr_open_read_mem`

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

Convenience wrapper that constructs a memory source (and optionally a memory mirror sink) internally. Equivalent to calling `bvnr_source_from_mem` followed by `bvnr_open_read_source`. Pass `NULL` / `0` for `mirror_buf` / `mirror_cap` to skip mirroring.

```
bvnr_read_flags_t opts = { .on_verified = on_event, .userdata = &ctx };

if (!bvnr_open_read_mem(r, payload, payload_len, NULL, 0, &opts))
    return -1;
```

---

6. `bvnr_read`

```
bool bvnr_read(bvnr_reader_t *r);
```

Drive the parser until EOF or a fatal error. Fires the registered callbacks for every event. Returns `true` on clean completion, `false` on any error.

This is the only call needed after `bvnr_open_read_*`. The reader does not allocate during this call; all buffering is internal to the reader struct.

```
if (!bvnr_read(r)) {
    fprintf(stderr, "parse error: %s at line %" PRIu64 " col %" PRIu64 "\n",
            bvn_error_to_string(bvnr_reader_get_error(r)),
            bvnr_reader_get_error_line(r),
            bvnr_reader_get_error_column(r));
    bvnr_reader_destroy(r);
    return -1;
}
```

---

7. `bvnr_reader_get_error` and friends

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

All five error/location getters above (everything except `bvnr_reader_get_recovery_count`) are only meaningful when `bvnr_read` returned `false` (or after a recoverable error when `continue_on_error` is set). `bvnr_reader_get_error_byte` returns the raw byte value that triggered the error. `bvnr_reader_get_recovery_count` is the exception: it returns how many errors triggered entry into resync mode in `continue_on_error` mode (and so is meaningful even when `bvnr_read` ultimately returned `true`). This count is incremented at error entry, not when resync completes at `";".

```
if (!bvnr_read(r)) {
    error_code_t ec = bvnr_reader_get_error(r);
    fprintf(stderr,
            "%s at line %" PRIu64 ", col %" PRIu64
            ", offset %" PRIu64 ", byte 0x%02X\n",
            bvn_error_to_string(ec),
            bvnr_reader_get_error_line(r),
            bvnr_reader_get_error_column(r),
            bvnr_reader_get_error_offset(r),
            bvnr_reader_get_error_byte(r));
}
```

7a. Version directive (spec 1.1)

```
bool        bvnr_reader_get_declared_version(
                const bvnr_reader_t *r, uint16_t *major, uint16_t *minor);
bool        bvnr_peek_version(
                const void *buf, uint64_t len, uint16_t *major, uint16_t *minor);
uint32_t    bvnr_version(void);
const char *bvnr_version_string(void);
void        bvnr_spec_version(uint16_t *major, uint16_t *minor);
```

A document may begin with a `#!bovnar <major>.<minor>` directive (see spec §3.4). After `bvnr_read`, `bvnr_reader_get_declared_version` returns `true` and fills `major`/`minor` when the document carried one (either out pointer may be NULL). Set `bvnr_read_flags_t.strict_version` to reject a version newer than this build supports with `error_unsupported_spec_version`; by default such a version is recorded but accepted. A malformed directive is always `error_invalid_spec_version`.

`bvnr_peek_version` scans a raw buffer for the directive without a full parse (handy before opening a writer to round-trip it). `bvnr_version` / `bvnr_version_string` return the library version; `bvnr_spec_version` returns the highest spec version this build understands (`BVNR_SPEC_VERSION_*`).

```
uint16_t maj, min;
if (bvnr_reader_get_declared_version(r, &maj, &min))
    printf("document declares bovnar %u.%u\n", maj, min);
```

To emit a directive, call `bvnr_write_version` right after opening the writer (see §13), or set `bvnr_write_flags_t.emit_version` to stamp the current spec version automatically.

7b. Datetime family (spec 1.1)

```
const char *bvnr_datetime_epoch_name (value_type_spec_t vt);
int32_t     bvnr_datetime_epoch_mjd  (value_type_spec_t vt);
int32_t     bvnr_datetime_epoch_index(const char *name);
bool        bvnr_write_datetime(bvnr_writer_t *w, const char *key,
                                uint32_t width, const char *epoch, int64_t value);
```

A `<datetime:width,epoch>` value (family `vt_datetime`) is a **signed integer count of seconds since a named epoch** — a timestamp, distinct from a duration (a number with a time unit, e.g. `<float:64,s>`). The carrier is validated like `sint`; the epoch is stored as a small dense index in `value_type_spec_t.base` (not a numeric base — the carrier is always decimal). These two helpers recover the epoch from a spec: `bvnr_datetime_epoch_name` returns its lowercase name (`"unix"` — the default — `"tai"`, `"gps"`, `"mjd"`, `"ntp"`, `"galileo"`, `"glonass"`, `"y2000"`, `"beidou"`), and `bvnr_datetime_epoch_mjd` returns its Modified Julian Day (the `bvn_epoch_t` value from `bvn_datetime.h`). Pass that to `bvn_dt_epoch_seconds_to_datetime()` to convert to a civil date/time.

```c
/* on a datetime data event: */
int64_t secs;  bvn_parse_int64((const char *)d->data, d->value_type, &secs);
bvn_datetime_t civil;
bvn_dt_epoch_seconds_to_datetime(&civil,
    (bvn_epoch_t)bvnr_datetime_epoch_mjd(d->value_type), secs);
```

To **write** a datetime, use the typed helper `bvnr_write_datetime` — `epoch` is an epoch name (or `NULL` for unix; an unknown name is `error_invalid_argument`), `value` is signed seconds. `bvnr_datetime_epoch_index` maps a name to the index stored in `value_type_spec_t.base` (the inverse of `bvnr_datetime_epoch_name`), for building a spec by hand. The document must declare `#!bovnar 1.1` to be re-read (emit the directive with `bvnr_write_version`).

```c
bvnr_write_version(w, 1, 1);
bvnr_write_datetime(w, "created", 64, "gps", 1750000000);  /* <datetime:64,gps> */
```

The family is spec 1.1: it requires a `#!bovnar 1.1` declaration, and in a 1.0/unversioned document a `datetime` annotation is `error_illegal_value_type`.

ISO-8601 literals and fractional seconds. A datetime may be written as an ISO-8601 literal (`2026-06-15T12:00:00.5Z`) instead of a raw integer; the reader converts it to the whole-second carrier you receive in `d->data`. When the literal carries a fractional second, the verbatim digits (no leading `.`) are delivered alongside the carrier in two `bvnr_data_t` fields, `frac_data` and `frac_length` — `NULL`/`0` for every other value. The carrier is unchanged (the value floors to the written second), so the fraction is informational, but it lets a consumer see sub-second precision the integer cannot hold, and the writer re-emits it: a datetime data event whose `frac_data` is set is serialised back as an ISO literal so the value round-trips. Like `d->data`, `frac_data` is **not NUL-terminated** — bound the read by `frac_length`.

```c
/* on a datetime data event: */
if (d->frac_data && d->frac_length) {
    /* d->frac_data[0 .. frac_length) are the sub-second digits, e.g. "5" */
}
```

---

8. `bvn_parse_uint64` / `bvn_parse_int64` / `bvn_parse_double`

```c
bool bvn_parse_uint64(const char *s, value_type_spec_t vt, uint64_t *out);
bool bvn_parse_int64 (const char *s, value_type_spec_t vt, int64_t  *out);
bool bvn_parse_double(const char *s, value_type_spec_t vt, double   *out);
```

Convert the raw token string received in `ev_data` into a C numeric type. The `vt` argument — taken directly from `d->value_type` — supplies the base and bit-width for range checking.

The `data` pointer inside `bvnr_data_t` is **not NUL-terminated**, and for null/empty values `d->length` may be `0`. Always guard the copy by `d->length` and NUL-terminate manually.

- Returns `true` and writes `*out` on success.

- Returns `false` if the string is not representable in the declared type.

```
static bool on_event(void *ud, bvnr_event_t ev, bvnr_data_t *d)
{
    if (ev != ev_data) return true;

    char buf[256];
    if (d->length >= sizeof(buf)) return false;
    if (d->length) memcpy(buf, d->data, d->length);
    buf[d->length] = '\0';

    switch (d->value_type.family) {
    case vt_uint: {
        uint64_t v;
        if (bvn_parse_uint64(buf, d->value_type, &v))
            printf("uint = %" PRIu64 "\n", v);
        break;
    }
    case vt_sint: {
        int64_t v;
        if (bvn_parse_int64(buf, d->value_type, &v))
            printf("sint = %" PRId64 "\n", v);
        break;
    }
    case vt_float: {
        double v;
        if (bvn_parse_double(buf, d->value_type, &v))
            printf("float = %g\n", v);
        break;
    }
    default:
        break;
    }
    return true;
}
```

Writer

---

9. `bvnr_writer_create` / `bvnr_writer_destroy`

```
bvnr_writer_t *bvnr_writer_create(void);
void           bvnr_writer_destroy(bvnr_writer_t *w);
```

Allocate and free a writer on the heap. Mirrors the reader lifecycle exactly. Returns `NULL` on allocation failure.

```
bvnr_writer_t *w = bvnr_writer_create();
if (!w) { perror("alloc"); exit(1); }

/* ... open, write events, finish ... */

bvnr_writer_destroy(w);
```

---

10. `bvnr_sink_to_fd`

```
void bvnr_sink_to_fd(bvnr_sink_t *s, int fd);
```

Initialise sink `s` to write serialised bytes to an open, writable POSIX file descriptor. The caller retains ownership of `fd`.

```
bvnr_sink_t sink;
bvnr_sink_to_fd(&sink, STDOUT_FILENO);
/* or: int fd = open("out.bvnr", O_WRONLY|O_CREAT|O_TRUNC, 0644); */
```

---

11. `bvnr_sink_to_mem`

```
void bvnr_sink_to_mem(bvnr_sink_t *s, void *buf, uint64_t cap);
```

Initialise sink `s` to write into a caller-provided memory buffer of `cap` bytes. Writing beyond `cap` produces `error_sink_buffer_exhausted`. Use `bvnr_sink_bytes_written` to query how many bytes were actually written.

```
char out[4096];
bvnr_sink_t sink;
bvnr_sink_to_mem(&sink, out, sizeof(out));
```

---

12. `bvnr_sink_bytes_written`

```
uint64_t bvnr_sink_bytes_written(const bvnr_sink_t *s);
```

Return the number of bytes pushed into a memory sink so far. Only meaningful for sinks created with `bvnr_sink_to_mem`. Useful after `bvnr_write_finish` to know the exact output length.

```
bvnr_write_finish(w);
uint64_t n = bvnr_sink_bytes_written(&sink);
fwrite(out, 1, n, stdout);
```

---

13. `bvnr_open_write_sink`

```
bool bvnr_open_write_sink(bvnr_writer_t        *w,
                          const bvnr_sink_t    *sink,
                          bool                  pretty,
                          bvnr_write_flags_t   *options);
```

Attach `sink` to the writer and configure it. Must be called before any `bvnr_write_event`. Returns `false` on invalid arguments.

- `pretty` — when `true`, the serialiser emits newlines and indentation. When `false`, output is compact (single line per assignment, no extra whitespace).
- `options` — configuration struct. Zero-initialise for defaults.

```
typedef struct bvnr_write_flags_s {
    /* ── Writer enforces these ──────────────────────────────── */
    uint8_t  max_struct_nesting;      /* 0 → 64 internal default; hard cap 255 */
    uint8_t  max_array_nesting;       /* 0 → 64 internal default; hard cap 255 */
    void    *userdata;
    bool   (*on_event)(void *userdata, bvnr_event_t, bvnr_data_t *);
    bvn_unit_flags_t unit_flags;      /* controls unit annotation format */
    bool     emit_version;            /* emit a leading "#!bovnar M.N" on open */

    /* ── Present for API symmetry with bvnr_read_flags_t;          ─
       the writer does not read or enforce these fields. Set to 0. */
    uint16_t max_identifier_length;
    uint16_t max_string_length;
    uint16_t max_number_length;
    uint16_t max_symbol_length;
    uint16_t max_reference_length;
    uint64_t max_array_items;
    uint64_t max_text_bytes;
    uint64_t max_file_size;
    bool     continue_on_error;       /* no-op in the writer */
    bvnr_on_error_fn on_error;        /* no-op in the writer */
} bvnr_write_flags_t;
```

> **Writer limits.** Only `max_struct_nesting`, `max_array_nesting`, `on_event`, `userdata`, and `unit_flags` have any effect on the writer. All other fields are present solely to keep `bvnr_write_flags_t` structurally parallel to `bvnr_read_flags_t`; they are silently ignored. In particular, `continue_on_error`, `on_error`, and all per-token-length fields have no effect. The writer never internally limits array items, text bytes, or file size.

`on_event` in the write flags fires for each event as it is serialised — useful for logging or auditing. Pass `NULL` if not needed.

`unit_flags` controls how unit annotations are serialised by the writer. The valid flags are:

Flag	Value	Effect
`BVN_UNIT_FLAGS_NONE`	`0`	Default: Unicode superscript exponents, no reduction
`BVN_UNIT_REDUCE`	`1 << 0`	Reduce compound units to canonical form before serialising
`BVN_UNIT_ASCII_EXP`	`1 << 1`	Use `^N` ASCII caret notation instead of Unicode superscripts

These flags can be OR-combined: `BVN_UNIT_REDUCE | BVN_UNIT_ASCII_EXP`. The flags are fixed at open time. To change serialisation behaviour, destroy the writer and open a new one with the updated `unit_flags`. The getter `bvnr_writer_unit_flags(w)` is used internally by the Python FFI layer to retrieve the live flags before each unit serialisation call; there is no public setter.

```c
bvnr_sink_t sink;
bvnr_sink_to_fd(&sink, fd);

bvnr_write_flags_t opts = { 0 };  /* all defaults */
if (!bvnr_open_write_sink(w, &sink, /*pretty=*/true, &opts))
    return -1;
```

---

14. `bvnr_open_write_mem`

```c
bool bvnr_open_write_mem(bvnr_writer_t      *w,
                         void               *buf,
                         uint64_t            cap,
                         bool                pretty,
                         bvnr_write_flags_t *options);
```

Convenience wrapper that constructs a memory sink internally and calls `bvnr_open_write_sink`. Equivalent to `bvnr_sink_to_mem` + `bvnr_open_write_sink`. To retrieve the written byte count after finishing, call `bvnr_writer_bytes_written`.

```c
char out[4096];
bvnr_write_flags_t opts = { 0 };
if (!bvnr_open_write_mem(w, out, sizeof(out), false, &opts))
    return -1;
```

---

15. `bvnr_write_event`

```c
bool bvnr_write_event(bvnr_writer_t *w, bvnr_event_t ev, bvnr_data_t *data);
```

Emit one event to the writer. This is the only function used to produce output. It serialises the event and the data it describes directly into the configured sink.

Returns `true` on success, `false` on any serialisation error.

The event sequence you must emit for every assignment mirrors exactly what the reader delivers to `on_verified`. At minimum, for a typed scalar value:

```
ev_assignment_start          (data->data = key, data->length = key length)
ev_type_annotation_start     (data->data = annotation text, or NULL for no annotation)
ev_type_annotation_type_family
ev_type_annotation_type_family_parameter   (for each parameter)
ev_type_annotation_end
ev_data                      (data->data = value string, data->value_type/value_unit set)
```

The `BVN_TYPE_*` macros build `value_type_spec_t` literals conveniently:

```
#define BVN_TYPE_PLAIN          ((value_type_spec_t){ .family = vt_plain })
#define BVN_TYPE_UTF8           ((value_type_spec_t){ .family = vt_utf8  })
#define BVN_TYPE_BOOL           ((value_type_spec_t){ .family = vt_bool  })
#define BVN_TYPE_UINT(w)        ((value_type_spec_t){ .family = vt_uint,      .width =
(w) })
#define BVN_TYPE_SINT(w)        ((value_type_spec_t){ .family = vt_sint,      .width =
(w) })
#define BVN_TYPE_FLOAT(w)       ((value_type_spec_t){ .family = vt_float,     .width =
(w) })
/* float_fix: .base is repurposed to store Q (fractional bits). */
#define BVN_TYPE_FLOAT_FIX(w,q) ((value_type_spec_t){ .family = vt_float_fix, .width =
(w), .base = (q) })
/* float_dec: base field is unused (always 0).                    */
#define BVN_TYPE_FLOAT_DEC(w)   ((value_type_spec_t){ .family = vt_float_dec, .width =
(w) })
/* float with explicit numeral base (for base-16 output):         */
#define BVN_TYPE_FLOAT_BASE(w,b) ((value_type_spec_t){ .family = vt_float, .width =
(w), .base = (b) })
/* uint/sint with explicit numeral base:                          */
#define BVN_TYPE_UINT_BASE(w,b) ((value_type_spec_t){ .family = vt_uint, .width =
(w), .base = (b) })
#define BVN_TYPE_SINT_BASE(w,b) ((value_type_spec_t){ .family = vt_sint, .width =
(w), .base = (b) })
```

The maximum bit-width accepted for `uint` and `sint` is `BVN_MAX_INT_WIDTH` (defined as `32768u` in `bovnar.h`). The validator and writer reject any declared `uint`/`sint` width exceeding this limit with `error_illegal_value_type`.

> **Critical:** The writer dispatches `ev_type_annotation_type_family_parameter` events on `d->type`, not on `d->value_type`. For each parameter event, `d.type` must be set to the appropriate `token_type_t` value: `token_is_type_width` for the width parameter, `token_is_type_base` for the base parameter, `token_is_type_q` for the Q (fractional bits) parameter of `float_fix`, and `token_is_unit` for the unit parameter. An unrecognised `d.type` causes the writer to emit nothing for that event — the annotation will be silently incomplete. **Use `bvnr_write_type_annotation` (see §22) to avoid this complexity entirely.**

Example: write `.port = <uint:16> 8080;`

```c
/* Helper: emit a fully-typed uint16 assignment */
static bool write_uint16(bvnr_writer_t *w,
                         const char *key, uint16_t value)
{
    char valbuf[16];
    snprintf(valbuf, sizeof(valbuf), "%" PRIu16, value);

    value_type_spec_t vt = BVN_TYPE_UINT(16);
    value_unit_t      vu = BVN_UNIT_NONE;

    bvnr_data_t d;

    /* 1. Assignment start — key */
    d = (bvnr_data_t){ .data = (void*)key, .length = (uint32_t)strlen(key) };
    if (!bvnr_write_event(w, ev_assignment_start, &d)) return false;

    /* 2. Type annotation — use bvnr_write_type_annotation for the full sequence */
    if (!bvnr_write_type_annotation(w, vt, vu)) return false;

    /* 3. Value */
    d = (bvnr_data_t){
        .type       = token_is_number,
        .value_type = vt,
        .value_unit = vu,
        .data       = valbuf,
        .length     = (uint32_t)strlen(valbuf),
    };
    return bvnr_write_event(w, ev_data, &d);
}
```

Example: write a string assignment `.host = "localhost";`

```c
static bool write_string(bvnr_writer_t *w,
                         const char *key, const char *value)
{
    value_type_spec_t vt = BVN_TYPE_UTF8;
    value_unit_t      vu = BVN_UNIT_NONE;
    bvnr_data_t d;

    d = (bvnr_data_t){ .data = (void*)key, .length = (uint32_t)strlen(key) };
    if (!bvnr_write_event(w, ev_assignment_start, &d)) return false;

    d = (bvnr_data_t){ .value_type = vt };
    if (!bvnr_write_event(w, ev_type_annotation_start, &d)) return false;
    if (!bvnr_write_event(w, ev_type_annotation_type_family, &d)) return false;
    if (!bvnr_write_event(w, ev_type_annotation_end, &d)) return false;

    d = (bvnr_data_t){
        .value_type = vt,
        .data       = (void*)value,
        .length     = (uint32_t)strlen(value),
    };
    return bvnr_write_event(w, ev_data, &d);
}
```

Example: open and close a struct

```c
static bool write_struct_start(bvnr_writer_t *w, const char *key)
{
    bvnr_data_t d = { .data = (void*)key, .length = (uint32_t)strlen(key) };
    if (!bvnr_write_event(w, ev_assignment_start, &d)) return false;
    return bvnr_write_event(w, ev_struct_start, &(bvnr_data_t){0});
}

static bool write_struct_end(bvnr_writer_t *w)
{
    return bvnr_write_event(w, ev_struct_end, &(bvnr_data_t){0});
}
```

15a. `bvnr_write_version`

```c
bool bvnr_write_version(bvnr_writer_t *w, uint16_t major, uint16_t minor);
```

Emit a leading `#!bovnar <major>.<minor>` version directive (spec §3.4). Must be called immediately after `bvnr_open_write_*` and before any value; calling it once output has begun is `error_invalid_argument`. Use it to round-trip a directive read from a source document, or pass `BVNR_SPEC_VERSION_MAJOR` / `BVNR_SPEC_VERSION_MINOR` to stamp the current spec version. Setting `bvnr_write_flags_t.emit_version` is equivalent to calling it with the current spec version right after open.

```c
bvnr_open_write_sink(w, &sink, true, NULL);
bvnr_write_version(w, 1, 1);          /* "#!bovnar 1.1\n" */
bvnr_write_uint(w, "port", 16, 443);
bvnr_write_finish(w);
```

---

16. `bvnr_write_finish`

```c
bool bvnr_write_finish(bvnr_writer_t *w);
```

Flush any buffered output and finalise the stream. Must be called after all `bvnr_write_event` calls and before `bvnr_writer_destroy`. Returns `false` if any struct is still open (`error_got_incomplete_bvnr_stream`), if writing the trailing semicolon fails, or if flushing the write buffer to the sink fails.

64 KiB write buffer. The writer accumulates output into an internal 64 KiB buffer. Individual `bvnr_write_event` calls do not push bytes to the sink immediately; instead bytes accumulate in the buffer and are forwarded to the sink only when the buffer is full or when `bvnr_write_finish` is called. This means the sink receives large contiguous writes rather than one tiny push per token, which is important for fd-based sinks. `bvnr_writer_bytes_written` always reflects the true total of bytes handed off to the sink plus bytes still in the buffer, so the count is accurate at any point during writing.

```c
if (!bvnr_write_finish(w)) {
    fprintf(stderr, "write finish failed: %s\n",
            bvn_error_to_string(bvnr_writer_get_error(w)));
}
bvnr_writer_destroy(w);
```

---

17. `bvnr_writer_get_error` and friends

```c
error_code_t     bvnr_writer_get_error       (const bvnr_writer_t *w);
uint64_t         bvnr_writer_get_error_offset(const bvnr_writer_t *w);
uint64_t         bvnr_writer_bytes_written   (const bvnr_writer_t *w);
bvn_unit_flags_t bvnr_writer_unit_flags      (const bvnr_writer_t *w);
```

The writer error API is smaller than the reader's: there are **no** `bvnr_writer_get_error_line` or `bvnr_writer_get_error_column` functions. The writer has no lexer and therefore cannot track source positions. Use `bvnr_writer_get_error_offset` (byte count into the output stream) and `bvnr_writer_get_error` (error code) for diagnostics. `bvnr_writer_bytes_written` returns the total bytes emitted to the sink so far — available at any point, not only after errors.

`bvnr_writer_unit_flags` returns the `bvn_unit_flags_t` bitmask currently stored in the writer object (set via `bvnr_write_flags_t.unit_flags` at open time). The writer uses these flags whenever it serialises a unit annotation string (via `bvn_unit_to_string_ex`). This function is primarily used by the Python bindings FFI layer to retrieve the live flags before each unit serialisation call.

```c
if (!bvnr_write_event(w, ev_data, &d)) {
    fprintf(stderr, "write error: %s at offset %" PRIu64 "\n",
            bvn_error_to_string(bvnr_writer_get_error(w)),
            bvnr_writer_get_error_offset(w));
    return -1;
}
```

---

18. `bvn_format_uint64` / `bvn_format_int64` / `bvn_format_double`

```c
int32_t bvn_format_uint64(char *buf, size_t bufsize,
                          uint64_t value, uint32_t base, uint32_t min_digits);

int32_t bvn_format_int64(char *buf, size_t bufsize,
                         int64_t value, uint32_t base, uint32_t min_digits);

int32_t bvn_format_double(char *buf, size_t bufsize,
                          double value, value_type_spec_t vt);
```

Produce the value string that goes into `bvnr_data_t.data` when writing numeric values. All three return the number of bytes written (excluding NUL terminator), or `-1` on buffer overflow.

- `base` — numeric base (2–62, 64, 85). Use `10` for the common case.
- `min_digits` — zero-pad to at least this many digits. Pass `0` for no padding.
- For `bvn_format_double`, the type spec `vt` controls the output precision according to `vt.width`. Because the input is a C `double`, the effective precision is capped at the 64-bit format (a wider `vt.width` yields no extra digits); render the full precision of a 128/256-bit value with the arbitrary-precision writer `bvnr_write_bvnf_base` instead.

```
char buf[32];

/* Format 255 as a base-16 value, minimum 2 digits → "ff" */
int32_t n = bvn_format_uint64(buf, sizeof(buf), 255, 16, 2);
/* buf = "ff", n = 2 */

/* Format 9.81 as a 64-bit float */
value_type_spec_t vt = BVN_TYPE_FLOAT(64);
n = bvn_format_double(buf, sizeof(buf), 9.81, vt);
/* buf = "9.81", n = 4 */
```

These strings are then placed into `bvnr_data_t.data` / `.length` before calling `bvnr_write_event(w, ev_data, &d)`.

---

19. `bvnr_write_bvnf_base` / `bvnr_write_bvnf_base_unit`

```
bool bvnr_write_bvnf_base(bvnr_writer_t *w, const char *key,
                          const bvn_float_t *f,
                          uint32_t width, uint32_t base);

bool bvnr_write_bvnf_base_unit(bvnr_writer_t *w, const char *key,
                               const bvn_float_t *f,
                               uint32_t width, uint32_t base,
                               value_unit_t unit);
```

Write an arbitrary-precision `bvn_float_t` value at any valid `float` width (0, 16, or any multiple of 32 up to 32768) and in either base 10 or base 16. The float string is generated internally by `bvn_float_to_str` and then validated by the writer before being sent to the sink — the call fails and sets an error if the generated string does not satisfy the writer's type constraints.

`bvnr_write_bvnf_base` is the no-unit variant; `bvnr_write_bvnf_base_unit` attaches a physical unit to the type annotation.

Base 10 — the value string uses the same decimal format as `bvn_format_double` and is emitted as a bare number token (`token_is_number`). The type annotation is `<float:W>` or `<float:W,_10>`.

Base 16 — the value string uses a binary-exponent hexadecimal format: `[-]D.DDDDp[+|-]EEE` where `D.DDDD` are lowercase hex mantissa digits and `EEE` is the decimal binary exponent. The string is emitted as a quoted string token (`token_is_string`) because non-decimal literals must be quoted in BVNR. The type annotation is `<float:W,_16>`.

```
#include "bvn_float.h"

bvn_float_t *f = bvn_float_alloc(1024u);
bvn_float_from_str(f, "3.14159265358979323846", 10);

/* .pi_dec = <float:1024> 3.14... ; */
bvnr_write_bvnf_base(w, "pi_dec", f, 1024u, 10u);

/* .pi_hex = <float:256,_16> "1.921fb54442d18p+1"; */
bvn_float_t *g = bvn_float_alloc(256u);
bvn_float_from_double(g, 3.14159265358979323846);
bvnr_write_bvnf_base(w, "pi_hex", g, 256u, 16u);

bvn_float_free(f);
bvn_float_free(g);
```

`width` 0 uses the precision stored in the `bvn_float_t` itself. An invalid width (not 0, not 16, not a positive multiple of 32, or greater than `BVN_FLOAT_MAX_PREC`) causes the call to return `false` with `error_illegal_value_type`.

These functions supersede calling `bvnr_write_bvnf` / `bvnr_write_bvnf_unit` when base 16 output or widths greater than 128 are needed. `bvnr_write_bvnf` and `bvnr_write_bvnf_unit` remain available as convenience wrappers that always use base 10.

---

20. `bvnr_write_bvni` / `bvnr_write_bvni_unit`

```
bool bvnr_write_bvni(bvnr_writer_t *w, const char *key,
                     const bvn_int_t *n,
                     uint32_t width, uint32_t base);

bool bvnr_write_bvni_unit(bvnr_writer_t *w, const char *key,
                          const bvn_int_t *n,
                          uint32_t width, uint32_t base,
                          value_unit_t unit);
```

Write an arbitrary bit-width integer. The type family (`uint` or `sint`) is determined by the `negative` flag of the `bvn_int_t`: negative values produce `<sint:W,...>`, non-negative values produce `<uint:W,...>`. Any numeric base supported by the writer (2–62, 64, 85) may be specified.

The integer string is generated by `bvn_int_to_str` and then validated before reaching the sink. For non-decimal bases the string is emitted as a quoted token (`token_is_string`); for base 10 it is emitted as a bare number token (`token_is_number`).

```c
#include "bvn_int.h"

bvn_int_t *n = bvn_int_alloc();

/* 128-bit unsigned max in decimal */
bvn_int_from_str(n, "340282366920938463463374607431768211455", 10);
bvnr_write_bvni(w, "u128max", n, 128u, 10u);

/* 256-bit value in hex — emitted as <uint:256,_16> "deadbeef..."; */
bvn_int_from_str(n, "deadbeefcafebabe0011223344556677"
                    "8899aabbccddeeff0011223344556677", 16);
bvnr_write_bvni(w, "wide_hex", n, 256u, 16u);

/* signed negative in hex */
bvn_int_from_str(n, "-7fffffff", 16);
bvnr_write_bvni(w, "neg_hex", n, 32u, 16u);

bvn_int_free(n);
```

`width` 0 defaults to 64 bits for range validation. A `width` that cannot hold the value causes the call to return `false` with `error_value_out_of_range`.

---

21. `BVN_TYPE_FLOAT_BASE`

```c
#define BVN_TYPE_FLOAT_BASE(w, b)  /* value_type_spec_t */
```

Convenience macro that constructs a `value_type_spec_t` for a `float` with explicit width `w` and base `b`. The existing `BVN_TYPE_FLOAT(w)` macro always yields base 0 (equivalent to base 10); `BVN_TYPE_FLOAT_BASE` is needed when base 16 must be encoded in the type spec before being passed to `bvnr_write_type_annotation` or `bvnr_write_event` directly.

```c
value_type_spec_t vt16 = BVN_TYPE_FLOAT_BASE(256u, 16u);
/* → { .family = vt_float, .width = 256, .base = 16 } */

value_type_spec_t vt10 = BVN_TYPE_FLOAT_BASE(64u, 10u);
/* equivalent to BVN_TYPE_FLOAT(64) */
```

Shared

---

22. `bvnr_write_type_annotation`

```c
bool bvnr_write_type_annotation(bvnr_writer_t *w,
                                value_type_spec_t vt,
                                value_unit_t vu);
```

Emit a complete type-annotation event sequence (`ev_type_annotation_start`, `ev_type_annotation_type_family`, zero or more `ev_type_annotation_type_family_parameter` events, `ev_type_annotation_end`) in a single call. Returns `false` on any serialisation error.

This is the **preferred** way to write type annotations. Using `bvnr_write_event` directly for parameter events requires setting `d.type` to the appropriate `token_type_t` constant for each parameter; `bvnr_write_type_annotation` handles this correctly and automatically.

The function emits parameters as follows:

- **Width** — emitted for numeric families when `vt.width != 0`. A width of `0` is **not** written to the stream (the absence implies the default width of 64 on the reader side via `bvn_effective_width`). Also emitted for `utf8` when `vt.width != 0`, though width on `utf8` has no defined semantics.
- **Base** — emitted for `float` when `vt.base` is non-zero and not `10`; emitted for `uint` / `sint` when `vt.base` is non-zero and not `10`.
- **Q** — emitted for `float_fix` when `vt.base` (which stores Q) is non-zero. A Q value of `0` is therefore not written explicitly.
- **Unit** — emitted when `vu.num_components > 0`. `BVN_UNIT_NONE` (num_components == 0) produces no unit parameter; `BVN_UNIT_NO_PREFIX(bu_none)` (num_components == 1) produces `no_unit` in the annotation.

```
value_type_spec_t vt = BVN_TYPE_FLOAT(64);
value_unit_t      vu = BVN_UNIT_COMPOUND2(
                           bu_meter,  si_none, exp_linear,
                           bu_second, si_none, exp_neg_square);

/* Emits: <float:64,m/s²> */
if (!bvnr_write_type_annotation(w, vt, vu)) return false;
```

---

23. `bvn_parse_unit` / `bvn_parse_unit_n`

```
value_unit_t bvn_parse_unit  (const uint8_t *unit, bool *ok);
value_unit_t bvn_parse_unit_n(const uint8_t *unit, uint32_t len, bool *ok);
```

Parse a compound unit string (e.g. `"k~g·m/s²"`) into a `value_unit_t`. Both set `*ok` to `false` and return a zeroed unit on any error.

`bvn_parse_unit` requires a NUL-terminated string. `bvn_parse_unit_n` accepts a length `len` and does **not** require a NUL terminator — use this variant when the unit string is a substring of a larger buffer (as is the case inside the parser itself).

This is useful when reading: after `ev_type_annotation_type_family_parameter`, the unit is already parsed for you in `d->value_unit`. You only need `bvn_parse_unit` / `bvn_parse_unit_n` if you are constructing a unit from a string yourself (e.g. from a config or CLI argument).

The validator also calls `bvn_parse_unit_n` internally when processing an **inline unit suffix** (the optional unit token that may follow a scalar value before its terminating `;`). You do not need to call either function yourself to consume inline units; the parsed result is automatically placed in `d->value_unit` of the `ev_data` event, exactly as for annotation-specified units.

```
bool ok;
value_unit_t u = bvn_parse_unit((const uint8_t *)"k~g·m/s²", &ok);
if (!ok) {
    fprintf(stderr, "bad unit\n");
    return;
}
/* u now holds { num_components=3, [{bu_gram,exp_linear,si_kilo},
                                    {bu_meter,exp_linear,si_none},
                                    {bu_second,exp_neg_square,si_none}] } */

/* Length-bounded variant — no NUL needed */
const uint8_t *annotation = (const uint8_t *)"float:64,m/s";
value_unit_t u2 = bvn_parse_unit_n(annotation + 9, 3, &ok); /* "m/s" */
```

---

24. `bvn_unit_to_string` / `bvn_unit_to_string_ex`

```
int32_t bvn_unit_to_string(value_unit_t u, char *buf, size_t bufsize);

int32_t bvn_unit_to_string_ex(value_unit_t u, char *buf, size_t bufsize,
                              bvn_unit_flags_t flags);
```

Serialise a `value_unit_t` back into its canonical string form. Numerator components are joined by `·`, followed by `/` and denominator components joined by `·`. Returns bytes written (excluding NUL), or `-1` on buffer overflow.

`bvn_unit_to_string` is equivalent to calling `bvn_unit_to_string_ex` with `flags = BVN_UNIT_FLAGS_NONE`.

`bvn_unit_to_string_ex` accepts a `bvn_unit_flags_t` bitmask that controls output format:

Flag	Effect
`BVN_UNIT_FLAGS_NONE`	Default: Unicode superscript exponents, no reduction
`BVN_UNIT_REDUCE`	Reduce compound unit to canonical named SI unit before serialising
`BVN_UNIT_ASCII_EXP`	Use `^N` ASCII caret notation instead of Unicode superscripts

These flags can be OR-combined. The writer uses `bvn_unit_to_string_ex` internally, passing the flags from `bvnr_writer_unit_flags(w)`.

> **Note on writer usage:** When driving the writer manually via `bvnr_write_event`, do **not** pass a unit string in `bvnr_data_t.data` for the `ev_type_annotation_start` event — the serialiser ignores that field and derives the annotation from `data->value_type` and

> the subsequent parameter events. Use `bvnr_write_type_annotation` (§22) to emit a complete, correct type annotation in one call.

```
value_unit_t u = BVN_UNIT_COMPOUND2(
    bu_gram,   si_kilo,  exp_linear,
    bu_second, si_none,  exp_neg_square);

char buf[64];
int32_t n = bvn_unit_to_string(u, buf, sizeof(buf));
/* buf = "k~g/s²", n = 7 */

n = bvn_unit_to_string_ex(u, buf, sizeof(buf), BVN_UNIT_ASCII_EXP);
/* buf = "k~g/s^2", n = 7 */
```

---

25. `bvn_error_to_string`

```
const char *bvn_error_to_string(error_code_t code);
```

Return a short, static, human-readable description of an error code. The returned pointer is valid for the lifetime of the program; do not free it.

```
fprintf(stderr, "error: %s\n", bvn_error_to_string(bvnr_reader_get_error(r)));
/* e.g. "error: value_out_of_range" */
```

Unit-related error codes (for reference):

Code	Value	String	Trigger
`error_unit_illegal`	32	`"unit_illegal"`	Unparseable unit string (unknown base, bad prefix, empty component, >8 components)
`error_unit_too_long`	22	`"unit_too_long"`	Unit string exceeds internal buffer
`error_unit_mismatch`	38	`"unit_mismatch"`	Inline unit suffix present, type-annotation unit also present, and the two differ

---

Complete Read Example

```c
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "bovnar.h"

typedef struct { char key[256]; } ctx_t;

static bool on_verified(void *ud, bvnr_event_t ev, bvnr_data_t *d)
{
    ctx_t *ctx = ud;

    if (ev == ev_assignment_start) {
        size_t n = d->length < 255 ? d->length : 255;
        memcpy(ctx->key, d->data, n);
        ctx->key[n] = '\0';
        return true;
    }

    if (ev != ev_data || d->length == 0) return true;

    char vbuf[256];
    size_t n = d->length < 255 ? d->length : 255;
    memcpy(vbuf, d->data, n);
    vbuf[n] = '\0';

    switch (d->value_type.family) {
    case vt_uint: {
        uint64_t v;
        bvn_parse_uint64(vbuf, d->value_type, &v);
        printf(".%-20s = %" PRIu64 "  (uint%u)\n",
               ctx->key, v, bvn_effective_width(d->value_type));
        break;
    }
    case vt_float: {
        double v;
        bvn_parse_double(vbuf, d->value_type, &v);
        printf(".%-20s = %g  (float%u)\n",
               ctx->key, v, bvn_effective_width(d->value_type));
        break;
    }
    case vt_utf8:
        printf(".%-20s = \"%s\"\n", ctx->key, vbuf);
        break;
    default:
        printf(".%-20s = %s\n", ctx->key, vbuf);
        break;
    }
    return true;
}

int main(int argc, char **argv)
{
    if (argc < 2) { fprintf(stderr, "usage: %s <file>\n", argv[0]); return 1; }

    int fd = open(argv[1], O_RDONLY);
    if (fd < 0) { perror(argv[1]); return 1; }

    bvnr_reader_t *r = bvnr_reader_create();
    ctx_t ctx = {0};

    bvnr_source_t src;
    bvnr_source_from_fd(&src, fd);

    bvnr_read_flags_t opts = {
```

```
        .on_verified   = on_verified,
        .userdata      = &ctx,
        .max_file_size = 16777216,
    };

    if (!bvnr_open_read_source(r, &src, NULL, &opts)) {
        fputs("failed to open reader\n", stderr);
        bvnr_reader_destroy(r);
        close(fd);
        return 1;
    }

    int ret = 0;
    if (!bvnr_read(r)) {
        fprintf(stderr, "%s at line %" PRIu64 " col %" PRIu64 "\n",
                bvn_error_to_string(bvnr_reader_get_error(r)),
                bvnr_reader_get_error_line(r),
                bvnr_reader_get_error_column(r));
        ret = 1;
    }

    bvnr_reader_destroy(r);
    close(fd);
    return ret;
}
```

Inline Unit Suffix — Reading

A scalar value may carry an **inline unit suffix** directly after the literal, before its terminating `;`:

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

From the application's perspective, inline units and annotation units are transparent: both end up in `d->value_unit` of the `ev_data` event. No special handling is needed.

The only behavioral difference occurs when **both** are present and **disagree**: the validator raises `error_unit_mismatch` (38) and parsing fails:

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

Inline unit suffixes are **illegal inside array elements**. The lexer rejects them with `error_unexpected_input_byte`.

Reading inline unit values

```c
/* The callback below works identically whether the unit came from a
 * type annotation or from an inline suffix — no change needed.        */
static bool on_verified(void *ud, bvnr_event_t ev, bvnr_data_t *d)
{
    if (ev != ev_data) return true;

    char unit_str[128] = "no_unit";
    /* dimensionless: num_components==0 (BVN_UNIT_NONE) or
       num_components==1 with base==bu_none (BVN_UNIT_NO_PREFIX(bu_none)) */
    bool is_dim = (d->value_unit.num_components == 0) ||
                  (d->value_unit.num_components == 1 &&
                   d->value_unit.components[0].base == bu_none);
    if (!is_dim)
        bvn_unit_to_string(d->value_unit, unit_str, sizeof(unit_str));

    char val[256];
    size_t n = d->length < 255 ? d->length : 255;
    memcpy(val, d->data, n);
    val[n] = '\0';

    printf("value=%s  unit=%s\n", val, unit_str);
    return true;
}
```

Complete Write Example

```c
#include <inttypes.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "bovnar.h"

/* Emit: .velocity = <float:64,m/s²> 9.81; */
static bool write_velocity(bvnr_writer_t *w)
{
    value_type_spec_t vt = BVN_TYPE_FLOAT(64);
    value_unit_t      vu = BVN_UNIT_COMPOUND2(
                               bu_meter,  si_none, exp_linear,
                               bu_second, si_none, exp_neg_square);

    char valbuf[32];
    bvn_format_double(valbuf, sizeof(valbuf), 9.81, vt);

    bvnr_data_t d;

    d = (bvnr_data_t){ .data = "velocity", .length = 8 };
    if (!bvnr_write_event(w, ev_assignment_start, &d)) return false;

    if (!bvnr_write_type_annotation(w, vt, vu)) return false;

    d = (bvnr_data_t){
        .type       = token_is_number,
        .value_type = vt,
        .value_unit = vu,
        .data       = valbuf,
        .length     = (uint32_t)strlen(valbuf),
    };
    return bvnr_write_event(w, ev_data, &d);
}

int main(void)
{
    bvnr_writer_t *w = bvnr_writer_create();

    bvnr_sink_t sink;
    bvnr_sink_to_fd(&sink, STDOUT_FILENO);

    bvnr_write_flags_t opts = { 0 };
    if (!bvnr_open_write_sink(w, &sink, /*pretty=*/true, &opts)) {
        fputs("failed to open writer\n", stderr);
        bvnr_writer_destroy(w);
        return 1;
    }

    int ret = 0;
    if (!write_velocity(w) || !bvnr_write_finish(w)) {
        fprintf(stderr, "write error: %s\n",
                bvn_error_to_string(bvnr_writer_get_error(w)));
        ret = 1;
    }

    bvnr_writer_destroy(w);
    return ret;
}
/* Output: .velocity = <float:64,m/s²> 9.81; */
```

End of Bovnar Read & Write API Reference (v1.1)