Package: zubin (via r-universe)

November 12, 2026

Title Read and Write Structured Binary Data

Version 0.1.0

Description Describe a fixed binary record once, as a layout, and read or write millions of them in one vectorised call: integers from 8 to 128 bits, the half, single and double precision floats of the IEEE 754 (Institute of Electrical and Electronics Engineers) standard, 'bfloat16' (brain floating point), complex numbers, booleans, fixed bytes and fixed strings, padding, arrays and 'C' struct alignment, in either byte order. Typed codecs convert whole vectors to and from bytes at every width and byte order, refusing rather than rounding a value that does not fit, and a growable byte builder with a hard cap appends without quadratic copying. The same buffer, cursor, typed reads and writes and layout kernels are a header-only 'C' library that other packages use through 'LinkingTo' alone.

License MIT + file LICENSE

URL https://pedrobtz.github.io/zubin/, https://github.com/pedrobtz/zubin

BugReports https://github.com/pedrobtz/zubin/issues

Depends R (>= 4.1)

LinkingTo zufast (>= 0.1.0)

Suggests bit64, knitr, rmarkdown, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-GB

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph]

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-11-11 22:51:39 UTC

RemoteUrl https://github.com/cran/zubin

RemoteRef HEAD

RemoteSha 46157489ce3c539356c3b214e63d09e510b191d2

Contents

zubin-package 2
bin_builder 3
bin_decode 4
bin_encode 5
bin_hash_object 6
bin_hexdump 7
bin_info 8
bin_layout 9
bin_pack 11
bin_put 12
bin_serialize 13
bin_size 15
bin_unpack 15
zubin-conditions 17

Index **19**

`zubin-package` *zubin: Read and Write Structured Binary Data*

Description

Describe a fixed binary record once, as a layout, and read or write millions of them in one vectorised call: integers from 8 to 128 bits, the half, single and double precision floats of the IEEE 754 (Institute of Electrical and Electronics Engineers) standard, 'bfloat16' (brain floating point), complex numbers, booleans, fixed bytes and fixed strings, padding, arrays and 'C' struct alignment, in either byte order. Typed codecs convert whole vectors to and from bytes at every width and byte order, refusing rather than rounding a value that does not fit, and a growable byte builder with a hard cap appends without quadratic copying. The same buffer, cursor, typed reads and writes and layout kernels are a header-only 'C' library that other packages use through 'LinkingTo' alone.

Author(s)

Maintainer: Pedro Baltazar <pedrobtz@gmail.com> [copyright holder]

Authors:

- Pedro Baltazar <pedrobtz@gmail.com> [copyright holder]

See Also

bin_layout() for the specification of a record and the type table; the vignette, vignette("zubin"), for a tour.

---

bin_builder *A growable byte buffer*

Description

bin_builder() creates a buffer that bytes are appended to with bin_put() and taken out of with bin_take(). It grows by doubling (by half once past 64 MiB), so a million appends cost a few dozen reallocations, not a million copies; and it never grows past max.

Usage

```
bin_builder(reserve = 0, max = Inf)

bin_reserve(b, n)

bin_reset(b)

bin_take(b)

## S3 method for class 'zubin_builder'
as.raw(x)
```

Arguments

reserve	Bytes to allocate at once: a whole number, at least 0.
max	The hard cap on the builder's size, in bytes: a positive whole number, or Inf for none. An append that would pass it is a zubin_limit_error and leaves the builder unchanged.
b, x	A builder.
n	Bytes to make room for.

Details

A builder is an external pointer: it is not copied on modification, every function below changes it in place, and it does not survive saveRDS() or a restored session; using a restored builder is a zubin_invalid_argument error. Its memory is freed when it is garbage collected.

bin_take(b) returns the bytes and empties the builder, which keeps its capacity and accepts new appends; as.raw(b) returns a copy and leaves the builder as it is. bin_reserve(b, n) makes room for n more bytes now, bin_reset(b) empties the builder without freeing its storage, and bin_size(b) is the number of bytes it holds.

Value

`bin_builder()` returns a zubin_builder. `bin_take()` and `as.raw()` return a raw vector; `bin_size()` returns a double; `bin_reserve()` and `bin_reset()` return `b`, invisibly.

See Also

`bin_put()` to append.

Examples

```
b <- bin_builder()
bin_put(b, as.raw(c(0x89, 0x50, 0x4e, 0x47)))
bin_put(b, "IHDR", type = "s4")
bin_size(b)
bin_take(b)
bin_size(b)
```

---

bin_decode *Read a vector of one type from bytes*

Description

`bin_decode()` reads `n` consecutive values of one type from the raw vector `x`, starting at the 0-based byte `offset`: `bin_decode(x, "u32", offset = 4)` is a header field, `bin_decode(x, "f32", offset = 64, n = 1e6)` is a column. It is `readBin()` generalised to every width and both byte orders, exact or an error; `bin_encode()` is its inverse. Offsets are 0-based.

Usage

```
bin_decode(
  x,
  type,
  offset = 0,
  n = NULL,
  endian = c("little", "big", "native"),
  int64 = c("double", "integer64"),
  na = c("error", "allow")
)
```

Arguments

`x`	A raw vector.
`type`	Any type token of `bin_layout()` except padding (`"u8"`, `"i128"`, `"f16"`, `"c64"`, `"bool"`, `"b16"`, `"s8"`, ...), optionally with a `.le` or `.be` suffix.
`offset`	The 0-based byte position of the first record.
`n`	The number of values, or `NULL` for every whole value that fits.

endian	The byte order: "little", "big", or "native".
int64	How i64, u64, i128 and u128 fields are returned: "double", exact or an error; or "integer64", a double vector carrying the 64-bit pattern with class integer64, as the bit64 package defines it.
na	"error", or "allow" to read -2^31 in an i32 field as NA_integer_ (and -2^63 in integer64 mode as NA).

Value

A vector of the type's R type (see bin_layout()): integer, double, integer64, complex, logical, character, or a list of raw vectors for b<n>.

Examples

```
x <- as.raw(c(0x01, 0x00, 0x02, 0x00, 0xff, 0xff))
bin_decode(x, "u16")
bin_decode(x, "i16")
bin_decode(x, "u16", endian = "big", offset = 2, n = 1)
```

---

bin_encode *Write a vector of one type to bytes*

Description

bin_encode() writes every element of x as type, in endian order (a matrix or array in column-major order, as its elements are stored): writeBin() generalised to every width and both byte orders, rounding to f16, bf16 and f32 to nearest even, and refusing, never truncating, an integer that does not fit.

Usage

```
bin_encode(
  x,
  type,
  endian = c("little", "big", "native"),
  na = c("error", "allow")
)
```

Arguments

x	A vector: numeric, logical, integer64, complex for c64 and c128, character for s<n>, or a list of raw vectors for b<n>.
type	Any type token of bin_layout() except padding ("u8", "i128", "f16", "c64", "bool", "b16", "s8", ...), optionally with a .le or .be suffix.
endian	The byte order: "little", "big", or "native".
na	"error", or "allow" to write NA into i32 as -2^31 (and an integer64 NA into i64 as -2^63).

Details

`bin_encode(bin_decode(b, t), t)` is `b` when `b` is the canonical encoding of its values, which every encoding of a number is, with three exceptions. A `bool` byte other than 00 or 01 reads as `TRUE` and writes back as 01. An `s<n>` reads up to its first NUL, so the bytes after it write back as NULs. And an `i64`, `u64`, `i128` or `u128` beyond 2^53 does not decode as a double at all: ask for `int64 = "integer64"` (which holds 64 bits). A NaN keeps its payload through `f64` and `c128`, but the narrower floats may not.

Value

A raw vector of `length(x)` times the type's width.

Examples

```
bin_encode(c(1, 2, 65535), "u16")
bin_encode(1:3, "i32", endian = "big")
bin_encode(c(0.1, 1e5), "f16")
try(bin_encode(300, "u8"))

# bytes that are not the canonical encoding do not survive a round trip
bin_encode(bin_decode(as.raw(2), "bool"), "bool")
```

`bin_hash_object` *A content fingerprint of an R object*

Description

`bin_hash_object()` is XXH3 of the body of R's XDR serialisation of `x`, format version 2 or 3: everything `serialize()` writes after its header. It is streamed through the hasher, so the serialised bytes are never allocated. Skipping the header, which records the R version that wrote it (and, for `version = 3`, the native encoding), keeps the digest from changing with those; it stays the same for as long as R writes the same body for `x`, which R does not promise across its versions (see `serialize()`), least of all for closures, environments and ALTREP objects. Objects with the same serialisation have the same digest; the converse holds only up to the chance of a collision.

Usage

```
bin_hash_object(x, algo = c("xxh3_64", "xxh3_128"), version = 2L, seed = 0)
```

Arguments

`x`	Any R object.
`algo`	`"xxh3_64"` for a 64-bit digest, `"xxh3_128"` for 128 bits.
`version`	The serialisation format version, 2 or 3.
`seed`	A whole number from 0 to 2^53, the XXH3 seed.

Details

It is a fingerprint, not a cryptographic digest and not a durable content address: fast, and well distributed, but not a defence against someone choosing inputs to collide.

The digest is of the bytes R writes, not of the values they stand for, so `identical(x, y)` does not imply equal digests:

- A double is its 64 bits. `0` and `-0` differ, as do NaNs of different sign or payload, and `NA` once arithmetic has set its quiet bit (`c(NA, 1) + 0`), though `identical()` treats each pair as the same. How NaN and `NA` propagate through arithmetic depends on the platform (see NA), so the same computation may hash differently on two machines; and on 32-bit x86 even a double `NA` written as a literal has different bits.
- A string is its bytes and its encoding mark, so the same non-ASCII text hashes differently when it is marked UTF-8, latin1 or not at all; and unmarked text is in the session's native encoding, which depends on the locale. Apply `enc2utf8()` to the strings first for a digest that does not depend on either.

Apart from these, the digest of an object is the same on every platform R runs on.

`version = 2`, the default, writes compact sequences such as `1:10` as the vectors they are; `version = 3` writes their compact form, so the same values may then hash differently.

Value

A string of 16 or 32 lower-case hexadecimal digits.

Examples

```
bin_hash_object(mtcars)
bin_hash_object(mtcars, "xxh3_128")
identical(bin_hash_object(1:3 + 0L), bin_hash_object(c(1L, 2L, 3L)))
```

---

bin_hexdump *Show bytes as a hex dump, and find where two raw vectors differ*

Description

`bin_hexdump()` formats `n` bytes of `x` from the 0-based byte `offset` the way `xxd` does: each line is the offset of its first byte in hex, the bytes in hex in groups of two, and the bytes as ASCII, with `.` for anything not printable. Offsets are 0-based and absolute, so a dump from `offset = 64` starts at `00000040`.

Usage

```
bin_hexdump(x, offset = 0, n = NULL, width = 16L)

bin_diff(a, b, n = 10L)
```

Arguments

x, a, b	Raw vectors.
offset	The 0-based position of the first byte to show; at most length(x), or the call is a zubin_bounds_error.
n	For bin_hexdump(), the number of bytes to show, or NULL for the rest; for bin_diff(), the most differences to list.
width	Bytes per line, 1 to 256.

Details

bin_diff() lists the first n positions, 0-based, at which a and b hold different bytes, comparing over the shorter length; the lengths themselves are attributes, so a prefix shows as no rows and two different lengths.

Both are presentation: they take any raw vector, and n past the end of x shows what there is.

Value

bin_hexdump() returns a character vector of class zubin_hexdump, one line per element, which prints as the dump. bin_diff() returns a data frame with columns offset (double), a and b (raw), and attributes length_a and length_b.

Examples

```
x <- charToRaw("Structured binary data, one layout at a time.")
bin_hexdump(x)
bin_hexdump(x, offset = 16, n = 16, width = 8)

y <- x
y[c(3, 20)] <- as.raw(0)
bin_diff(x, y)
```

---

bin_info *Information about the compiled zubin headers*

Description

Reports, from compiled code, the version of the zubin headers the package's own shared object was built with, the version of the zufast headers they were compiled against, the host's byte order, and the compiler and build flags. Packages that use LinkingTo: zubin, zufast carry their own copy of the code; this describes zubin's copy only.

Usage

```
bin_info()
```

Value

A named list with elements `version` (a string such as `"0.1.0"`), `version_major`, `version_minor`, `version_patch` (integers), `zufast` (the zufast header version, a string), `endian` (`"little"` or `"big"`, what `endian = "native"` means on this host), `compiler` (a string) and `build` (a named character vector: `c_standard`, the value of `__STDC_VERSION__`; `optimized` and `ndebug`, each `"true"` or `"false"`; and `fortify_source`, the `_FORTIFY_SOURCE` level or `"0"`).

Examples

```
bin_info()$version
bin_info()$endian
```

---

bin_layout *Describe a binary record*

Description

`bin_layout()` parses a specification of a fixed-size binary record into a `zubin_layout`: the type, element count, size, 0-based byte offset and byte order of each field, the record's size, and its alignment. Offsets are 0-based, as in every file format specification.

Usage

```
bin_layout(spec, endian = c("little", "big", "native"), align = FALSE)
```

Arguments

`spec`	A specification string; or a named character vector of type tokens, whose names are the field names (`c(id = "u32", "x3")`); or a `zubin_layout`, which is returned unchanged.
`endian`	The byte order of fields the specification does not order itself: `"little"`, `"big"`, or `"native"`, the host's (see `bin_info()`).
`align`	`FALSE` for packed fields, `TRUE` for C struct alignment.

Value

A `zubin_layout`: a list with `spec`, the normalised specification; `fields`, a data frame with columns `name`, `type`, `count` (array elements, 1 for a scalar), `size`, `offset` and `endian` (`NA` for one-byte types); `size`, the record size in bytes; and `align`, the record's alignment (1 when packed). `length()` and `names()` describe the fields that carry values, which excludes padding; `as.data.frame()` returns `fields`; `bin_size()` returns `size`.

The specification

One string of fields separated by whitespace or a comma:

```
layout  := [ endian ] field { sep field }
endian  := "<" | ">" | "="          little, big, host; default `endian`
field   := [ name ":" ] type [ "[" count "]" ] [ "." order ]
         | "x" width                padding: never read, written as zeros
type    := u8 i8 u16 i16 u32 i32 u64 i64 u128 i128
         | f16 bf16 f32 f64 c64 c128 bool
         | "b" width                fixed bytes
         | "s" width                fixed string
order   := "le" | "be"              multi-byte numeric types only
name    := [A-Za-z_] [A-Za-z0-9_.]*
count, width := 1 .. 2^31 - 1
```

For example `"<magic:u32 version:u16 flags:u16 count:u64 name:s32"`, or `"<ts:i64 price:f64 qty:i32 side:u8 x3 crc:u32.be"`. An endian prefix overrides `endian`, and a field's `.le` or `.be` suffix overrides both. Names are unique; an unnamed field is called `V1`, `V2`, ... by its position among the fields that are not padding. `t[k]` is an array of `k` elements, for the numeric types and `bool` only (write `b24`, not `b8[3]`). Every malformed specification is a `zubin_spec_error` carrying `position`, the 0-based byte position of the problem in the string.

Types

The type model, in both directions (`bin_unpack()` and `bin_decode()` read; `bin_pack()`, `bin_encode()` and `bin_put()` write):

Spec	Bytes	Read as
`u8 i8 u16 i16`	1–2	integer
`i32`	4	integer; -2^31 is an error unless `na = "allow"`
`u32`	4	double
`i64 u64`	8	double, an error above 2^53 in magnitude; or `integer64` (`int64 = "integer64"`), an error for
`i128 u128`	16	as `i64`: double, an error above 2^53 in magnitude; or `integer64`, an error outside its range
`f16 bf16`	2	double
`f32 f64`	4, 8	double
`c64 c128`	8, 16	complex: two `f32` or two `f64`, real part first
`bool`	1	logical; any non-zero byte is `TRUE`
`b<n>`	n	a list of raw vectors
`s<n>`	n	character: the bytes before the first NUL, as UTF-8
`x<n>`	n	not returned
`t[k]`	k x width	a matrix of k columns (`name.1` ... `name.k` in a data frame)

A value that does not fit is a `zubin_range_error`, and an `NA` with no bytes a `zubin_na_error`; nothing is truncated, wrapped or rounded into an integer. See zubin-conditions.

Alignment

With `align = FALSE` fields are packed at consecutive offsets. With `align = TRUE` each field starts at the next multiple of its element width (1 for `bool`, `b`, `s` and `x`; a complex type aligns as its part, `c64` to 4 and `c128` to 8; `i128` and `u128` to 16) and the record size is rounded up to the largest of them: the rule C compilers apply to a `struct` on 64-bit targets, so a struct written with `fwrite()` reads back with `align = TRUE`. (32-bit x86 aligns 8-byte members to 4 inside a struct; such a struct needs explicit `x` padding.)

Examples

```
wav <- bin_layout("<riff:s4 size:u32 wave:s4")
wav
bin_size(wav)
wav$fields

# The same, built programmatically
bin_layout(c(riff = "s4", size = "u32", wave = "s4"))

# A C struct { uint8_t a; uint32_t b; uint16_t c; }
bin_layout("a:u8 b:u32 c:u16", endian = "native", align = TRUE)$fields
```

---

bin_pack *Write records to bytes*

Description

`bin_pack()` writes one record per row of its columns, laid out as `layout` describes, into a raw vector of `n * bin_size(layout)` bytes. Padding and alignment gaps are zeros, so the output is deterministic and never holds stray memory.

Usage

```
bin_pack(layout, ..., na = c("error", "allow"))
```

Arguments

`layout`	A `bin_layout()`, or a specification string for one.
`...`	Named columns, or one data frame or list of them.
`na`	`"error"`, or `"allow"` to write `NA` into an `i32` field as -2^31 (and an `integer64` `NA` into `i64` as -2^63).

Details

The columns are named vectors in `...`, one per field that carries a value (padding takes none), or a single data frame or list with those names. A data frame from `bin_unpack()` works as it is: array fields may be given as a matrix with one row per record, or as the columns `name.1`, ..., `name.k`, and a single record's array as a vector of length `k`. Columns are recycled to the longest; a length that

does not divide it is a zubin_invalid_argument error, never a warning. A field named layout or na cannot be given by name in ..., where R matches it to bin_pack()'s own argument: give the columns as a data frame or list instead, bin_pack(hdr, list(layout = 2, size = 8)).

Each field accepts the R types of the table in bin_layout(): integer, whole double and logical for the small integer types; integer and whole double for i32 and u32; integer, whole double and integer64 for i64, u64, i128 and u128; numeric for the floating types (rounded to nearest even); complex, or numeric as a real part, for c64 and c128; logical for bool; character for s<n> (UTF-8, at most n bytes); and a list of raw vectors of exactly n bytes for b<n>. A string marked "bytes" is written as its bytes, unchecked, so that bin_unpack(x, 1, encoding = "bytes") packs back to x; any other string must be valid UTF-8 once translated to it, or the call is a zubin_encoding_error. Nothing is truncated, wrapped or rounded into an integer type: a value that does not fit is a zubin_range_error, and an NA or a NaN a zubin_na_error, each carrying field and the 0-based record index of the first one. NA has bytes only in i32 (with na = "allow"), f64, and the other floating types, where it becomes a NaN.

Value

A raw vector.

See Also

bin_unpack(), the inverse; bin_put() to append to a builder.

Examples

```
hdr <- bin_layout(">magic:u32 version:u16 count:u32")
bin_pack(hdr, magic = 0xCAFEBABE, version = 1, count = 3)

rec <- bin_layout("<id:u16 x:f32 flag:bool x1")
df <- data.frame(id = 1:3, x = c(0.5, 1.5, 2.5), flag = c(TRUE, FALSE, TRUE))
bytes <- bin_pack(rec, df)
bin_unpack(bytes, rec)
```

---

bin_put *Append bytes or strings to a builder*

Description

Appends x to the end of the builder b. A raw vector is appended as it is (type = NULL). Any other vector needs a type, so that a double is never silently eight bytes: a numeric, logical or integer64 vector is encoded as bin_encode() would encode it ("u32", "f64", "bool", ...), in endian order, written straight into the builder; a list of raw vectors as "b<n>". A character vector is appended with type = "z", each string's UTF-8 bytes followed by a NUL, or with type = "s<n>", each string's UTF-8 bytes padded with NULs to exactly n bytes, the fixed-string field of bin_layout(). A string marked "bytes" (as bin_unpack() returns them with encoding = "bytes") is appended as its bytes, unchecked; any other must be valid UTF-8 once translated to it. Nothing is appended unless every element can be: an NA is a zubin_na_error, a value that does not fit its type a zubin_range_error, and a string that is not valid UTF-8 a zubin_encoding_error, each carrying the 0-based index of the first one.

Usage

```
bin_put(b, x, type = NULL, endian = c("little", "big", "native"))
```

Arguments

b	A builder from bin_builder().
x	A raw vector; or a vector to encode as type.
type	NULL for raw input; one type token of bin_layout() otherwise; or "z" for NUL-terminated strings.
endian	Byte order of a typed append: "little", "big" or "native".

Details

Appending past the builder's max is a zubin_limit_error carrying size, the size the builder would have reached, and max; the builder is unchanged. Long appends check for an interrupt every 64 MiB, and an interrupted append leaves the builder unchanged too.

Value

b, invisibly, so appends can be chained.

Examples

```
b <- bin_builder()
bin_put(b, c("RIFF", "WAVE"), type = "s4")
bin_put(b, c(1, 65535), type = "u16", endian = "big")
bin_put(b, "a C string", type = "z")
bin_take(b)
```

---

bin_serialize *Serialise R objects into a builder, and back from bytes at an offset*

Description

bin_serialize() appends the serialisation of x to the builder b, exactly as serialize() would write it, without first allocating the whole stream as a raw vector. bin_unserialize() reads one serialised object from the raw vector x, starting at the 0-based byte offset, so a stream stored inside a larger file or record is read where it lies. Offsets are 0-based.

Usage

```
bin_serialize(x, b, version = 3L, xdr = TRUE, refhook = NULL)

bin_unserialize(x, offset = 0, refhook = NULL)
```

Arguments

x	For bin_serialize(), any R object; for bin_unserialize(), a raw vector holding a serialisation stream at offset.
b	A builder from bin_builder().
version	The serialisation format version, 2 or 3, as in serialize().
xdr	TRUE for the portable big-endian (XDR) format, FALSE for the host's native binary format, as in serialize().
refhook	NULL, or a function handling reference objects, as in serialize() and unserialize().
offset	The 0-based position of the stream in x.

Details

Both are R's own serialisation: unserialize(bin_take(b)) after bin_serialize(x, b) is identical to x, and bin_unserialize() reads what serialize() writes. Errors that R raises while writing or reading the stream (an object it cannot serialise, a malformed stream, an error in refhook) arrive as R raised them. A stream that ends before the object does is a zubin_bounds_error carrying offset and length. An append that would pass the builder's max is a zubin_limit_error; in every failure, an interrupt included, the builder is left as it was. refhook must not use b: a refhook that changes it is a zubin_invalid_argument error, and one that removed bytes from b leaves it holding what the refhook left.

Only read serialised data from a source you trust: a serialisation stream can hold any R object, functions and environments included.

Value

bin_serialize() returns b, invisibly. bin_unserialize() returns the object.

See Also

bin_hash_object() for a digest of the same stream.

Examples

```
b <- bin_builder()
bin_put(b, as.raw(c(0xde, 0xad)))          # a 2-byte header of our own
bin_serialize(list(a = 1:3, b = "zubin"), b)
bytes <- bin_take(b)
bin_unserialize(bytes, offset = 2)
```

bin_size *The size of a builder or a layout, in bytes*

Description

For a builder, the number of bytes it holds; for a layout, the size of one record.

Usage

```
bin_size(x, ...)

## S3 method for class 'zubin_builder'
bin_size(x, ...)

## S3 method for class 'zubin_layout'
bin_size(x, ...)
```

Arguments

x	A zubin_builder or a zubin_layout.
...	Unused.

Value

A double for a builder; an integer for a layout.

Examples

```
b <- bin_builder()
bin_put(b, as.raw(1:3))
bin_size(b)
```

bin_unpack *Read records from bytes*

Description

bin_unpack() reads n records described by layout from the raw vector x, the first at the 0-based byte offset and each next one stride bytes further on, into one column per field. Offsets are 0-based.

Usage

```
bin_unpack(
  x,
  layout,
  offset = 0,
  n = NULL,
  stride = NULL,
  as = c("data.frame", "list"),
  int64 = c("double", "integer64"),
  na = c("error", "allow"),
  encoding = c("UTF-8", "latin1", "bytes")
)
```

Arguments

x	A raw vector.
layout	A bin_layout(), or a specification string for one.
offset	The 0-based byte position of the first record.
n	The number of records, or NULL for every whole record that fits; trailing bytes that do not make a whole record are left alone. An explicit n that runs past the end of x is a zubin_bounds_error carrying offset and length.
stride	Bytes from the start of one record to the next, at least the record size; NULL for the record size. A larger stride reads the head of a larger record.
as	"data.frame" for a data frame, whose array fields expand to columns name.1, name.2, ... and whose b<n> fields are list columns wrapped in I(); or "list" for the columns as they are, array fields as matrices.
int64	How i64, u64, i128 and u128 fields are returned: "double", exact or an error; or "integer64", a double vector carrying the 64-bit pattern with class integer64, as the bit64 package defines it.
na	"error", or "allow" to read -2^31 in an i32 field as NA_integer_ (and -2^63 in integer64 mode as NA).
encoding	How s<n> bytes are marked: "UTF-8" (validated), "latin1" or "bytes" (any bytes).

Details

Fields are read field by field, each in one pass over the records, so a million records cost one loop per field. Nothing is rounded or wrapped: a value that has no exact R representation is a zubin_range_error carrying field and the 0-based record index of the first one, namely an i32 of -2^31 (R's NA_integer_; allowed with na = "allow"), an i64, u64, i128 or u128 above 2^53 in magnitude when int64 = "double", and, when int64 = "integer64", a value outside the signed 64-bit range or an i64 of -2^63 (bit64's NA). c64 and c128 fields read as complex vectors. An s<n> field holds the bytes up to its first NUL; with encoding = "UTF-8" they must be valid UTF-8, or the call is a zubin_encoding_error carrying field and index.

A single header is bin_unpack(x, hdr, n = 1): a one-row result. Give n = 1 whenever x holds more than the header, as a whole file does: with n = NULL the layout is read again at every multiple

of its size, and the rows after the first are the bytes that follow the header. A layout of padding alone has no columns, and so returns a data frame of no columns and no rows.

Value

A data frame with one row per record, or a named list of columns.

See Also

`bin_decode()` for one field read as a vector; `bin_pack()` for the inverse.

Examples

```
hdr <- bin_layout(">magic:u32 minor:u16 major:u16")
x <- as.raw(c(0xca, 0xfe, 0xba, 0xbe, 0x00, 0x00, 0x00, 0x41))
bin_unpack(x, hdr, n = 1)

# records with an array field
pts <- bin_layout("<id:u16 xy:i8[2]")
y <- as.raw(c(1, 0, 5, 250, 2, 0, 7, 9))
bin_unpack(y, pts)
bin_unpack(y, pts, as = "list")$xy
```

---

zubin-conditions *Conditions raised by zubin*

Description

Every error zubin raises inherits `zubin_error`, and its class says what went wrong, so code catches it by kind and reads the details from its fields; messages may be reworded. Every offset and index is 0-based. The call is reduced to the function's name, so that a large raw vector is never printed with an error.

Details

`zubin_invalid_argument` An argument is unusable: the wrong type, a negative or fractional size, an unknown type token, a column with no field, or a builder that was saved and restored. Field: `arg`, the argument's name, when there is one.

`zubin_spec_error` A layout specification is malformed. Field: `position`, the byte in the specification where the problem is.

`zubin_bounds_error` The records asked for run past the end of the input. Fields: `offset` and `length`.

`zubin_range_error` A value does not fit: 300 into `u8`, 1.5 into an integer type, an `i64` above 2^53 read as a double, a string longer than its field. Fields: `field` (`NA` for a codec) and `index`, the record of the first such value.

`zubin_na_error` An `NA` has no bytes in its field. Fields: `field` and `index`.

`zubin_encoding_error` An s<n> field read is not valid UTF-8 (read it with `encoding = "latin1"` or `"bytes"`), or a string to be written is not (mark it `"bytes"` to write its bytes as they are). Fields: `field` and `index`.

`zubin_limit_error` An append would pass a builder's `max`, which leaves it unchanged. Fields: `size`, what the builder would have held, and `max`.

`zubin_memory_error` An allocation zubin makes itself failed (a builder's storage), or a size would not fit in memory. A result that R allocates, such as the columns of `bin_unpack()`, fails as R's own allocations do, with R's `cannot allocate vector` error.

Examples

```
tryCatch(
  bin_encode(c(1, 300), "u8"),
  zubin_range_error = function(e) e$index
)
tryCatch(
  bin_layout("id:u32 name:s0"),
  zubin_spec_error = function(e) e$position
)
```

Index

∗ **package**
    zubin-package, 2

as.raw.zubin_builder *(*bin_builder*)*, 3

bin_builder, 3
bin_builder(), *13, 14*
bin_decode, 4
bin_decode(), *10*, *17*
bin_diff *(*bin_hexdump*)*, 7
bin_encode, 5
bin_encode(), *4*, *10*, *12*
bin_hash_object, 6
bin_hash_object(), *14*
bin_hexdump, 7
bin_info, 8
bin_info(), *9*
bin_layout, 9
bin_layout(), *3–5*, *11–13*, *16*
bin_pack, 11
bin_pack(), *10*, *17*
bin_put, 12
bin_put(), *3, 4*, *10*, *12*
bin_reserve *(*bin_builder*)*, 3
bin_reset *(*bin_builder*)*, 3
bin_serialize, 13
bin_size, 15
bin_take *(*bin_builder*)*, 3
bin_take(), *3*
bin_unpack, 15
bin_unpack(), *10–12*, *18*
bin_unserialize *(*bin_serialize*)*, 13

enc2utf8(), *7*

I(), *16*

NA, *7*

readBin(), *4*

saveRDS(), *3*
serialize(), *6*, *13, 14*

unserialize(), *14*

writeBin(), *5*

zubin *(*zubin-package*)*, 2
zubin-conditions, *10*, 17
zubin-package, 2