Package: zucbor (via r-universe)

October 21, 2026

Title Deterministic and Secure CBOR Encoding and Decoding

Version 0.1.0

Description Encodes R values as Concise Binary Object Representation (CBOR, RFC 8949; Bormann and Hoffman (2020) <doi:10.17487/RFC8949>) and decodes CBOR into ordinary R vectors and lists. Decoding checks the whole input against configurable depth, size and item limits before any R object is built, so untrusted input from network peers and devices cannot drive large allocations. Encoding is deterministic: identical R objects always produce identical bytes. Uses a bundled copy of the 'TinyCBOR' library (<https://github.com/intel/tinycbor>), so no system library is required.

License MIT + file LICENSE

Copyright See inst/COPYRIGHTS and src/vendor/PROVENANCE.

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

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

Depends R (>= 4.1)

Suggests 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], Thiago Macieira [aut] (Author of the bundled TinyCBOR library), Intel Corporation [cph] (TinyCBOR, bundled in src/vendor/tinycbor)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

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

Date/Publication 2026-10-21 14:30:08 UTC

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

RemoteRef HEAD

RemoteSha 83e436a97a860daae66c9e91b047bbacb7aca2ff

Contents

as_cbor 2
cbor-values 3
cbor_annotate 5
cbor_decode 6
cbor_decode_prefix 9
cbor_diagnose 11
cbor_encode 12
cbor_read 15
cbor_validate 16
zucbor-conditions 18
zucbor_info 19

Index **20**

`as_cbor` *Teach* `cbor_encode()` *a class*

Description

`cbor_encode()` calls `as_cbor()` on every object whose class it does not know, so a package or a script can say how its own class is written by adding a method. A method returns something `cbor_encode()` does know, typically a `cbor_tag()` holding a raw vector, a string or a number. `cbor_decode()`'s `tag_handlers` argument is the other half: it turns the tag back into the class.

Usage

```
as_cbor(x, ...)

## Default S3 method:
as_cbor(x, ...)
```

Arguments

`x`	An object of a class `cbor_encode()` does not know.
`...`	For methods; `cbor_encode()` passes nothing.

Details

The default method returns `x` unchanged, and `cbor_encode()` then writes it as its underlying type, as it does for any class without a method.

`as_cbor()` is called only for a class `cbor_encode()` does not handle itself: not for `POSIXct`, `Date`, `factor`, data frames, `POSIXlt` or zucbor's own classes, nor for a value whose only class is `AsIs`. It is called once per value: the result is written as it is, even if its class is one `cbor_encode()` does not know, though each of its elements is converted in turn. A result of the same class as `x`, other than `x` itself, is refused with `zucbor_unsupported_type`, since it is most likely a method that forgot to convert.

Encoding stays deterministic only if the methods are: a method should depend on nothing but `x`.

Value

A value `cbor_encode()` can write.

See Also

`cbor_decode()` for `tag_handlers`, and the examples article for recipes for UUIDs, IP addresses and decimal fractions.

Examples

```
# RFC 9562 UUIDs as tag 37, from a class of our own.
uuid <- function(hex) structure(list(hex = hex), class = "my_uuid")
as_cbor.my_uuid <- function(x, ...) {
  h <- gsub("-", "", x$hex)
  bytes <- as.raw(strtoi(substring(h, seq(1, 31, 2), seq(2, 32, 2)), 16L))
  cbor_tag(37, bytes)
}
# Methods defined in a script are found from the global environment;
# a package registers them with S3method(zucbor::as_cbor, my_uuid).
id <- uuid("f81d4fae-7dec-11d0-a765-00a0c91e6bf6")
bytes <- cbor_encode(list(id = id))
cbor_diagnose(bytes)

back <- cbor_decode(bytes, tag_handlers = list("37" = function(value) {
  h <- paste(format(as.hexmode(as.integer(value)), width = 2), collapse = "")
  uuid(paste(substring(h, c(1, 9, 13, 17, 21), c(8, 12, 16, 20, 32)), collapse = "-"))
}))
identical(back$id, id)
```

`cbor-values` *CBOR values without a native R type*

Description

Constructors for the four values `cbor_decode()` returns when R has no faithful type of its own. Encoding accepts them too, to write those values back.

Usage

```
cbor_map(keys = list(), values = list())

cbor_tag(tag, value)

cbor_simple(value)

cbor_bigint(x)
```

Arguments

keys, values	Lists (or vectors, taken element by element) of equal length.
tag	A whole number from 0 to 2^53.
value	For cbor_tag(), the tagged content; for cbor_simple(), integer simple values.
x	Decimal strings or whole numbers.

Details

- cbor_map() is a map whose keys are not all non-empty, unique text: COSE and CWT use small integers as keys, for example. keys and values are lists of equal length, in order.
- cbor_tag() is a tagged item: tag is the tag number, value its content. cbor_encode() refuses content its tag number does not allow, such as a number under tag 0.
- cbor_simple() is a simple value other than false, true, null and undefined: 0 to 19 or 32 to 255.
- cbor_bigint() is an integer outside what a double holds exactly, stored as canonical decimal text. It accepts decimal strings, or whole numbers within 2^53.

Value

An object of class cbor_map, cbor_tag, cbor_simple or cbor_bigint.

Examples

```
cbor_map(list(1L, 3L), list(-7L, "ES256"))
cbor_tag(32, "https://example.com")
cbor_simple(16)
cbor_bigint("18446744073709551616")
```

cbor_annotate *Annotated hex dump of CBOR*

Description

Shows where each byte of CBOR goes: one line per head, with its offset, its bytes in hex, indented by nesting, and what it is. A string's content follows its head, 16 bytes a line, and the first of those lines previews it. Every byte of the input appears exactly once in the hex column. This is the view needed to review a signed message byte by byte; `cbor_diagnose()` shows what the message means instead.

Usage

```
cbor_annotate(
  x,
  sequence = FALSE,
  deterministic = FALSE,
  duplicate_keys = FALSE,
  max_depth = 256L,
  max_size = 64 * 1024^2,
  max_items = 1e+06
)
```

Arguments

`x`	A raw vector.
`sequence`	If `TRUE`, `x` is an RFC 8742 CBOR sequence of zero or more items; if `FALSE`, it must be exactly one item.
`deterministic`	If `TRUE`, also require RFC 8949 section 4.2.1 core deterministic encoding: shortest integers, lengths and floats, definite lengths, map keys in bytewise order, and bignums in preferred form.
`duplicate_keys`	If `FALSE` (the default), a map with the same key twice is invalid.
`max_depth`	Deepest nesting allowed, counting arrays, maps and tags. At most `zucbor_info()$max_depth_cap`.
`max_size`	Largest input allowed, in bytes, or `Inf`.
`max_items`	Most data items allowed, counting every tag and every chunk of an indefinite-length string, or `Inf`.

Details

The input is checked first, exactly as `cbor_validate()` checks it. The output is bounded by a constant multiple of the input's size: previews stop at 32 bytes and indentation at 16 levels.

Offsets are 0-based, in decimal, as in zucbor's error conditions. Integers show their value (`negative(-501)`), floats their width and value (`float16 1.5`), and indefinite-length items (`*`) and a `break`.

Value

A character vector of class `cbor_annotation`, one element per line, which prints as the lines.

See Also

`cbor_diagnose()` for diagnostic notation.

Examples

```
cbor_annotate(as.raw(c(0xa2, 0x61, 0x61, 0x01, 0x61, 0x62, 0x82, 0x02, 0x03)))

# A COSE_Sign1 message's protected header, {1: -7}, inside a byte string.
cbor_annotate(cbor_encode(list(cbor_encode(cbor_map(list(1L), list(-7))), "payload")))
```

---

cbor_decode *Decode CBOR*

Description

Turns CBOR bytes into ordinary R values. The whole input is checked first – well-formedness, validity, duplicate keys and the limits, exactly as `cbor_validate()` does – and nothing is built until that check passes, so a length header cannot make R allocate for data the input does not hold.

Usage

```
cbor_decode(
  x,
  simplify = c("preserve", "none"),
  map_keys = c("auto", "map", "string"),
  tags = c("convert", "keep"),
  big_integers = c("bigint", "double", "error"),
  duplicate_keys = FALSE,
  deterministic = FALSE,
  max_depth = 256L,
  max_size = 64 * 1024^2,
  max_items = 1e+06,
  tag_handlers = NULL,
  data_frame = FALSE,
  max_cells = 1e+07
)

cbor_decode_seq(
  x,
  simplify = c("preserve", "none"),
  map_keys = c("auto", "map", "string"),
  tags = c("convert", "keep"),
  big_integers = c("bigint", "double", "error"),
```

```
  duplicate_keys = FALSE,
  deterministic = FALSE,
  max_depth = 256L,
  max_size = 64 * 1024^2,
  max_items = 1e+06,
  tag_handlers = NULL,
  data_frame = FALSE,
  max_cells = 1e+07
)
```

Arguments

x	A raw vector holding exactly one CBOR data item (cbor_decode()), or an RFC 8742 sequence of zero or more (cbor_decode_seq()).
simplify	"preserve" simplifies arrays whose elements agree to atomic vectors; "none" makes every array a list.
map_keys	"auto" gives a named list when every key is a non-empty, unique text string, and a cbor_map otherwise. "map" always gives a cbor_map. "string" always gives a named list, naming each entry by its text key, or by the key's diagnostic notation when it is not text; it is lossy, and refuses a map whose keys collide once stringified.
tags	"convert" turns the tags in the table into R values and keeps the rest as cbor_tag; "keep" makes every tag a cbor_tag.
big_integers	What to do with an integer beyond 2^53, which a double cannot hold exactly: "bigint" returns a cbor_bigint, "double" the nearest double, and "error" refuses the input.
duplicate_keys	If FALSE (the default), a map with the same key twice is invalid.
deterministic	If TRUE, also require RFC 8949 section 4.2.1 core deterministic encoding: shortest integers, lengths and floats, definite lengths, map keys in bytewise order, and bignums in preferred form.
max_depth	Deepest nesting allowed, counting arrays, maps and tags. At most zucbor_info()$max_depth_cap.
max_size	Largest input allowed, in bytes, or Inf.
max_items	Most data items allowed, counting every tag and every chunk of an indefinite-length string, or Inf.
tag_handlers	NULL, or a list of functions of one argument, named by tag number, such as list("37" = function(value) ...). See "Tag handlers".
data_frame	If TRUE, an array whose elements are all maps with text keys becomes a data frame. See "Data frames".
max_cells	With data_frame = TRUE, the most cells (rows times columns) one data frame may have, or Inf. Checked before the frame is allocated: rows that share no keys make a frame quadratic in the input.

Value

The decoded value; for cbor_decode_seq(), a list with one element per item.

CBOR to R

CBOR	R
integer	`integer` if it fits, else `double` up to 2^53, else by `big_integers`
float (half, single, double)	`double`, exactly
`false` / `true`	`logical`
`null`, `undefined`	`NULL`, or `NA` inside an atomic vector
byte string	`raw`
text string	`character`, UTF-8
other simple value	`cbor_simple`
array	atomic vector when the elements agree, else `list`
map with non-empty, unique text keys	named `list`
any other map	`cbor_map` (by `map_keys`)
array of maps with non-empty, unique text keys	`data.frame`, with `data_frame = TRUE`
tag 0 (date/time text), tag 1 (epoch)	`POSIXct`, UTC
tag 2, 3 (bignum)	the integer it holds, as for an integer; payloads over 128 bytes stay `cbor_ta`
tag 100 (days), tag 1004 (full date)	`Date`
tag 55799 (self-describe)	its content
tags 64-82, 84-86 (RFC 8746 typed arrays)	`integer` (8- and 16-bit, signed 32-bit) or `double`; 64-bit beyond 2^53 by `bi`
tags 1040, 40 (multi-dimensional arrays)	`matrix` or `array`, in R's column-major order
any other tag	`cbor_tag`

An array simplifies to an atomic vector only when its elements agree: integers and floats combine to the wider; booleans stay logical, and text stays text; wide integers and integer-valued numbers combine to `cbor_bigint`; `POSIXct` and `Date` stay their class. `null` joins any of them as `NA`. Anything else – raw vectors, nested arrays and maps, tags, booleans with numbers, a mixture – is a list. `[]` is `logical(0)`.

A one-element array that simplifies is marked with `I()`, so that `cbor_encode()` writes it back as an array rather than a single value: decoding and then encoding gives the same bytes for input in deterministic form, apart from the lossy conversions listed in `cbor_encode()`.

`null` and `undefined` both decode as missing: R has one missing value.

Tag handlers

`tag_handlers` gives meaning to tags zucbor does not convert, or replaces a conversion it does make. Each handler is called with the tag's content, decoded with the same options, and its result takes the tag's place:

```
cbor_decode(x, tag_handlers = list(
  "37" = function(value) my_uuid(value),            # RFC 9562 UUID
  "0"  = function(value) value                      # keep the text
))
```

A handler runs only once the whole input has been checked, so it never sees bytes that are malformed, invalid or over a limit. Its result never joins an array's simplification: an array holding one is a list. A handler applies whatever `tags` says, and a tag without one follows `tags`. An error in a

handler becomes zucbor_handler_error, with the tag number as tag and the original condition as parent. A handler that calls cbor_decode() again, for CBOR embedded in a byte string, passes that call its own limits. as_cbor() is the encoding half.

Data frames

With data_frame = TRUE, an array whose elements are all maps that decode to named lists – every key a non-empty text string, none twice – becomes a data frame, one row per map, as cbor_encode() writes one. Its columns are the union of the maps' keys, in the order they are first seen; a key a row lacks is missing there, as null is. Each column simplifies as an array of its cells would, so a column of numbers is numeric and a column of mixed values is a list (in which a missing cell is NULL). An empty array stays logical(0), and an array holding anything other than such maps decodes as usual.

Rows that share no keys make a frame with as many columns as rows, so the cell count is checked against max_cells before the frame is allocated, and a frame over it is zucbor_cell_limit.

```
# SenML (RFC 8428): a pack of records.
cbor_decode(x, data_frame = TRUE)
```

See Also

cbor_encode(), as_cbor(), cbor_validate(), cbor_read(), cbor-values, zucbor-conditions.

Examples

```
cbor_decode(as.raw(c(0x83, 0x01, 0x02, 0x03)))        # [1, 2, 3]
cbor_decode(as.raw(c(0xa1, 0x61, 0x61, 0xf5)))        # {"a": true}

# A COSE header: integer keys, so a cbor_map.
hdr <- cbor_decode(as.raw(c(0xa1, 0x01, 0x26)))       # {1: -7}
hdr

cbor_decode_seq(as.raw(c(0x01, 0x61, 0x61)))          # 1, then "a"
```

---

cbor_decode_prefix *Decode the CBOR item at the start of a raw vector*

Description

Decodes the one CBOR data item that x starts with and reports how many bytes it used, for CBOR inside binary framing: a COSE key in the middle of WebAuthn authData, or a payload after a fixed header. The item is checked and decoded exactly as by cbor_decode(), with the same arguments; what follows it is not read at all.

Usage

```
cbor_decode_prefix(
  x,
  simplify = c("preserve", "none"),
  map_keys = c("auto", "map", "string"),
  tags = c("convert", "keep"),
  big_integers = c("bigint", "double", "error"),
  duplicate_keys = FALSE,
  deterministic = FALSE,
  max_depth = 256L,
  max_size = 64 * 1024^2,
  max_items = 1e+06,
  tag_handlers = NULL,
  data_frame = FALSE,
  max_cells = 1e+07
)
```

Arguments

`x`	A raw vector starting with a CBOR data item.
`simplify`	`"preserve"` simplifies arrays whose elements agree to atomic vectors; `"none"` makes every array a list.
`map_keys`	`"auto"` gives a named list when every key is a non-empty, unique text string, and a `cbor_map` otherwise. `"map"` always gives a `cbor_map`. `"string"` always gives a named list, naming each entry by its text key, or by the key's diagnostic notation when it is not text; it is lossy, and refuses a map whose keys collide once stringified.
`tags`	`"convert"` turns the tags in the table into R values and keeps the rest as `cbor_tag`; `"keep"` makes every tag a `cbor_tag`.
`big_integers`	What to do with an integer beyond 2^53, which a double cannot hold exactly: `"bigint"` returns a `cbor_bigint`, `"double"` the nearest double, and `"error"` refuses the input.
`duplicate_keys`	If `FALSE` (the default), a map with the same key twice is invalid.
`deterministic`	If `TRUE`, also require RFC 8949 section 4.2.1 core deterministic encoding: shortest integers, lengths and floats, definite lengths, map keys in bytewise order, and bignums in preferred form.
`max_depth`	Deepest nesting allowed, counting arrays, maps and tags. At most `zucbor_info()$max_depth_cap`.
`max_size`	Largest input allowed, in bytes, or `Inf`.
`max_items`	Most data items allowed, counting every tag and every chunk of an indefinite-length string, or `Inf`.
`tag_handlers`	`NULL`, or a list of functions of one argument, named by tag number, such as `list("37" = function(value) ...)`. See "Tag handlers".
`data_frame`	If `TRUE`, an array whose elements are all maps with text keys becomes a data frame. See "Data frames".

`max_cells` With `data_frame = TRUE`, the most cells (rows times columns) one data frame may have, or `Inf`. Checked before the frame is allocated: rows that share no keys make a frame quadratic in the input.

Details

That also means nothing about the rest is known: it may be more CBOR, or the next field of the framing, or garbage. It is the caller's to make sense of, as `x[-seq_len(consumed)]`.

`max_size` applies to `x` as a whole, since all of it is in memory already. An empty `x` is `zucbor_parse_error`, as for `cbor_decode()`; so is an item cut short by the end of `x`.

Value

A list: `value`, the decoded item, and `consumed`, the number of bytes it took, a double.

See Also

`cbor_decode()` for one item and nothing else, and `cbor_decode_seq()` for items all the way to the end.

Examples

```
# [1, 2] followed by four bytes of something else.
x <- as.raw(c(0x82, 0x01, 0x02, 0xde, 0xad, 0xbe, 0xef))
r <- cbor_decode_prefix(x)
r$value
x[-seq_len(r$consumed)]
```

`cbor_diagnose` *CBOR diagnostic notation*

Description

Shows CBOR as RFC 8949 section 8 diagnostic notation: the human-readable form CBOR specifications use for examples. The input is checked first, exactly as `cbor_validate()` checks it, so nothing hostile reaches the printer, and the output is bounded by the input's size.

Usage

```
cbor_diagnose(
  x,
  sequence = FALSE,
  deterministic = FALSE,
  duplicate_keys = FALSE,
  max_depth = 256L,
  max_size = 64 * 1024^2,
  max_items = 1e+06
)
```

Arguments

`x`	A raw vector.
`sequence`	If `TRUE`, `x` is an RFC 8742 CBOR sequence of zero or more items; if `FALSE`, it must be exactly one item.
`deterministic`	If `TRUE`, also require RFC 8949 section 4.2.1 core deterministic encoding: shortest integers, lengths and floats, definite lengths, map keys in bytewise order, and bignums in preferred form.
`duplicate_keys`	If `FALSE` (the default), a map with the same key twice is invalid.
`max_depth`	Deepest nesting allowed, counting arrays, maps and tags. At most `zucbor_info()$max_depth_cap`.
`max_size`	Largest input allowed, in bytes, or `Inf`.
`max_items`	Most data items allowed, counting every tag and every chunk of an indefinite-length string, or `Inf`.

Details

The notation matches RFC 8949's own Appendix A. Numbers print as JSON and JavaScript print them, with `.0` added to a float that would otherwise read as an integer (`1.0`, `1.0e+300`). Text is ASCII: characters beyond it appear as `\u` escapes, as Appendix A writes them. An indefinite-length item shows `_` after its opening bracket, and a chunked string its chunks as `(_ "a", "b")`. A sequence's items are separated by commas (RFC 8742 section 4.2).

Value

A single string.

See Also

`cbor_decode()` to turn CBOR into R values.

Examples

```
cbor_diagnose(as.raw(c(0xa2, 0x61, 0x61, 0x01, 0x61, 0x62, 0x82, 0x02, 0x03)))
cbor_diagnose(cbor_encode(list(when = Sys.Date(), pi = pi, bytes = as.raw(1:4))))
cbor_diagnose(as.raw(c(0x01, 0xf9, 0x3e, 0x00)), sequence = TRUE)
```

`cbor_encode` *Encode R values as CBOR*

Description

Turns an R value into CBOR bytes. The encoding is always RFC 8949 section 4.2.1 core deterministic encoding, so identical R objects give identical bytes on every platform: the shortest form of every integer, length and tag number; definite lengths; the shortest float that holds each value exactly; and map entries sorted by their encoded keys.

Usage

```
cbor_encode(
  x,
  auto_unbox = TRUE,
  self_describe = FALSE,
  max_depth = 256L,
  typed_arrays = FALSE
)

cbor_encode_seq(
  x,
  auto_unbox = TRUE,
  self_describe = FALSE,
  max_depth = 256L,
  typed_arrays = FALSE
)
```

Arguments

`x`	An R value. For `cbor_encode_seq()`, a list whose elements are encoded one after another as an RFC 8742 CBOR sequence.
`auto_unbox`	If `TRUE`, a length-one atomic vector is a single value; if `FALSE`, it is an array of one. Names used as map keys are always single text strings.
`self_describe`	If `TRUE`, prefix each item with the self-describe tag 55799 (`d9 d9 f7`), which marks bytes as CBOR.
`max_depth`	Deepest nesting to write, counting arrays, maps and tags as `cbor_decode()` does, so its output always decodes at the same `max_depth`.
`typed_arrays`	If `TRUE`, write numeric vectors, matrices and arrays as RFC 8746 typed arrays: see "Typed arrays". Off by default, since many decoders do not read them.

Value

A raw vector.

R to CBOR

R	CBOR
`NULL`, `NA` of any type	`null`
`TRUE`, `FALSE`	`true`, `false`
`integer`	integer
`double`, whole, within CBOR's integer range, not `-0`	integer
any other `double`, including `NaN`, `Inf` and `-0`	float, shortest exact width
`character`	text string, UTF-8
`raw`	one byte string, whatever its length
`factor`	its labels, as text
`POSIXct`	tag 1: whole seconds as an integer, else a float
`Date`	tag 1004 `"YYYY-MM-DD"`; tag 100 (days) outside years 0000-9999

`cbor_bigint`	integer, or tag 2 or 3 beyond CBOR's integer range
`cbor_map`, `cbor_tag`, `cbor_simple`	a map, a tagged item, a simple value
unnamed list or vector	array
fully named list or vector	map with text keys
data frame	array of maps, one per row, keyed by column name

A length-one atomic vector is a single value, not an array, unless it is wrapped in `I()` or `auto_unbox = FALSE`. A matrix is a flat array in column-major order. An object whose class zucbor does not know goes through `as_cbor()` first, so a method can say how to write it; without one it is written as its underlying type.

Whole doubles become integers because R has no integer literal: `-7` written in R is a double, and as a CBOR float it would be a different value from the integer a COSE algorithm identifier needs. So `1L` and `1` encode identically.

These have no CBOR form and raise `zucbor_unsupported_type`: complex numbers, functions, environments, external pointers, S4 objects, `POSIXlt` (convert with `as.POSIXct()`), and data frame columns that are matrices, data frames or raw vectors. Names that are partly missing, `NA` or empty are `zucbor_invalid_argument`, and two keys that encode identically are `zucbor_duplicate_key`.

A `cbor_tag()` is written as given, but its content must be what `cbor_validate()` accepts under that tag number: text under tag 0 must be an RFC 3339 date/time, a typed array a whole number of elements, and so on; anything else is `zucbor_invalid_argument`. Under a tag whose content cannot be an array, a length-one vector is one value even with `auto_unbox = FALSE`. Tags 2 and 3 are written in preferred form, without leading zero bytes, and as a plain integer when the value fits in 64 bits.

Data frames

A data frame is written row by row, as `zujson` writes one: an array of maps, each keyed by the column names, so `data.frame(a = 1:2)` is `[{"a": 1}, {"a": 2}]`. A cell is written as the same element of its column would be in a vector, `NA` as `null`, and a list column's cell as a value of its own. Row names are dropped. A column of a class zucbor does not know goes through `as_cbor()` whole, once. `cbor_decode(x, data_frame = TRUE)` reads such an array back as a data frame.

Typed arrays

With `typed_arrays = TRUE`, an `integer` or `double` vector that would be written as an array (not a single value) is one tagged byte string instead of one item per element: tag 78 (32-bit signed integers) or tag 86 (64-bit floats), little-endian on every platform. Every bit is kept: `NA_integer_` is written as the smallest 32-bit integer, which decodes to `NA_integer_` again, and `NA_real_`, `NaN` and `-0` keep their bits. A whole double stays a float, unlike in an array. A matrix or array, of any type, is tag 1040: its dimensions and its elements in R's column-major order, the elements a typed array when they are numeric. Vectors with a class, such as factors and dates, are written as without the option.

For 10^6 doubles, a typed array is 8 MB against 9 MB as an array, and is written and read several times faster.

Conversions that do not round-trip

Decoding what `cbor_encode()` wrote gives back the value, except that: `NA` comes back as `NULL`, or `NA` of the vector's type; a whole double comes back as an integer; `list(1L)` and `1L` both encode as `1`; `NaN` payloads are not kept; fractional days of a `Date` are dropped; and a data frame comes back, with `data_frame = TRUE`, without its row names, with factors as text, and with its columns in the encoded key order; a `POSIXct` comes back in UTC, whatever its `tzone`; attributes other than names, `dim` and class are dropped; `cbor_tag(55799, v)` comes back as `v`; and a `cbor_bigint`, or a tag 2 or 3, whose value fits in 64 bits comes back as an integer or a double.

See Also

`cbor_decode()`, `as_cbor()`, cbor-values.

Examples

```
cbor_encode(list(a = 1, b = c(2, 3)))
cbor_encode(c(1.5, NaN, Inf))

# A COSE header: integer keys need cbor_map().
cbor_encode(cbor_map(list(1L), list(-7)))

# Deterministic: map entries are sorted by their encoded keys.
identical(cbor_encode(list(b = 1, a = 2)), cbor_encode(list(a = 2, b = 1)))

cbor_encode_seq(list(1, "a", TRUE))
```

---

cbor_read — *Read CBOR from a file or connection*

Description

Reads the input and decodes it as `cbor_decode()` or `cbor_decode_seq()` would. At most `max_size + 1` bytes are ever read, so an oversized file or an endless connection fails with `zucbor_size_limit` rather than exhausting memory.

Usage

```
cbor_read(file, ..., max_size = 64 * 1024^2)

cbor_read_seq(file, ..., each = NULL, max_size = 64 * 1024^2)
```

Arguments

`file`	A file path, a URL, or a connection.
`...`	Arguments passed on to `cbor_decode()` or `cbor_decode_seq()`.
`max_size`	The largest input, in bytes, read before giving up with `zucbor_size_limit`; with `each`, the largest item.
`each`	`NULL`, or a function of one argument, called with each item of the sequence in turn. See "Reading a sequence item by item".

Value

As cbor_decode() or cbor_decode_seq(); with each, the number of items read, a double.

Reading a sequence item by item

With each, cbor_read_seq() reads a sequence of any length in memory bounded by its largest item: each item is decoded and passed to each as soon as its last byte has been read, nothing is kept, and the call returns the number of items read. A long telemetry log or a socket that stays open is read this way.

Each item is still checked whole before anything is built from it, as cbor_decode() does, and the limits then apply to each item: max_size bounds the bytes of one item, and max_items the data items inside one. A faulty item stops the read with its condition, after each has been called for every item before it; the condition's offset counts from the start of the read, not of the item. So does input that ends inside an item. An error in each propagates unchanged and stops the read too.

A string with an http, https, ftp, ftps or file scheme is read with url(), any other string as a file path. A connection that is not open is opened in "rb" mode and closed afterwards; an open one must be binary, is read from its current position, and is left open.

Examples

```
path <- tempfile(fileext = ".cbor")
writeBin(as.raw(c(0xa1, 0x61, 0x61, 0x01)), path)
cbor_read(path)

# A sequence of three items, one at a time.
writeBin(cbor_encode_seq(list(1L, "two", list(3L))), path)
cbor_read_seq(path, each = function(item) str(item))
unlink(path)
```

---

cbor_validate *Validate CBOR*

Description

Checks that x holds well-formed, valid CBOR within the given limits, without building any R value. This is exactly the check cbor_decode() runs before it builds anything, so the two agree about the bytes; they disagree only where the bytes are valid but R cannot hold the value, such as a text string containing U+0000.

Usage

```
cbor_validate(
  x,
  sequence = FALSE,
  deterministic = FALSE,
  duplicate_keys = FALSE,
  max_depth = 256L,
```

```
  max_size = 64 * 1024^2,
  max_items = 1e+06,
  error = FALSE
)
```

Arguments

x	A raw vector.
sequence	If TRUE, x is an RFC 8742 CBOR sequence of zero or more items; if FALSE, it must be exactly one item.
deterministic	If TRUE, also require RFC 8949 section 4.2.1 core deterministic encoding: shortest integers, lengths and floats, definite lengths, map keys in bytewise order, and bignums in preferred form.
duplicate_keys	If FALSE (the default), a map with the same key twice is invalid.
max_depth	Deepest nesting allowed, counting arrays, maps and tags. At most zucbor_info()$max_depth_cap.
max_size	Largest input allowed, in bytes, or Inf.
max_items	Most data items allowed, counting every tag and every chunk of an indefinite-length string, or Inf.
error	If TRUE, raise the classed condition describing the fault instead of returning FALSE; see zucbor-conditions.

Details

The check covers well-formedness (RFC 8949 section 3), UTF-8 in text strings, the content of the tags RFC 8949, RFC 8943 and RFC 8746 restrict (their content type, the date text of tags 0 and 1004, the length of a typed array, the shape of a multi-dimensional array), duplicate map keys (compared by value, so 1 encoded in one byte and in two is the same key), and the limits.

Value

TRUE or FALSE; with error = TRUE, TRUE invisibly or an error.

Examples

```
cbor_validate(as.raw(c(0x83, 0x01, 0x02, 0x03)))   # [1, 2, 3]
cbor_validate(as.raw(c(0x83, 0x01, 0x02)))         # truncated
cbor_validate(as.raw(c(0xa2, 0x01, 0x00, 0x01, 0x00)))  # {1: 0, 1: 0}
cbor_validate(as.raw(c(0x01, 0x02)), sequence = TRUE)
```

zucbor-conditions *Conditions raised by zucbor*

Description

Every error zucbor raises carries a condition class, so it can be caught by kind rather than by matching the message, which may change. Every class below inherits from `zucbor_error`.

Details

`zucbor_invalid_argument` An argument was unusable: an input that is not a raw vector, a flag that is not `TRUE` or `FALSE`, a limit that is not a positive whole number within its range, or a value `cbor_encode()` cannot write as given, such as partial names or a string that is not valid UTF-8.

`zucbor_parse_error` The input is not well-formed CBOR: it ends early, uses a reserved encoding, has a misplaced "break", or has bytes after the item.

`zucbor_invalid_error` The input is well-formed but not valid: a text string that is not UTF-8, a tag whose content has the wrong type, a typed array whose length is not a whole number of elements, or a multi-dimensional array whose shape does not match its elements.

`zucbor_deterministic_error` `deterministic = TRUE` and the input is not in RFC 8949 core deterministic encoding.

`zucbor_duplicate_key` A map has the same key twice and `duplicate_keys = FALSE`.

`zucbor_unrepresentable` The input is valid CBOR but holds a value R cannot: a text string containing U+0000 or longer than an R string, a tag number beyond 2^53, an integer beyond 2^53 with `big_integers = "error"`, or an array dimension beyond R's limit.

`zucbor_unsupported_type` `cbor_encode()` was given an R value with no CBOR form, such as a function, an environment, `POSIXlt` or a data frame.

`zucbor_handler_error` A function in `cbor_decode()`'s `tag_handlers` raised an error. The condition carries `tag`, the tag number, and `parent`, the error the handler raised.

`zucbor_io_error` `cbor_read()` or `cbor_read_seq()` could not read from its connection. The message includes R's own.

`zucbor_limit_error` A limit was reached. The subclasses `zucbor_depth_limit`, `zucbor_size_limit`, `zucbor_item_limit` and `zucbor_cell_limit` name which one.

A condition raised while checking input carries `offset`, the 0-based byte offset of the item at fault, or `NA` when the validator that found the fault reports no position; and `status`, the name of the underlying status, such as `"CborErrorUnexpectedEOF"`. It is there for diagnostics: branch on the class, not on it. A limit error also carries `limit`, the argument's name, such as `"max_depth"`, and `limit_value`. A `zucbor_invalid_argument` condition carries `arg`, the argument at fault.

Examples

```
tryCatch(
  cbor_validate(as.raw(c(0x82, 0x01)), error = TRUE),
  zucbor_parse_error = function(e) e$offset
)
```

zucbor_info *Build information*

Description

Reports the bundled TinyCBOR version, the compile-time ceiling on nesting depth, and the default decoding limits.

Usage

```
zucbor_info()
```

Value

A list of class `zucbor_info` with elements:

- `tinycbor_version`: version of the bundled TinyCBOR, e.g. `"7.0.0"`.
- `max_depth_cap`: the largest `max_depth` any decoder accepts, set by TinyCBOR's compile-time recursion limit.
- `limits`: the default `max_depth`, `max_size` (bytes) and `max_items`.
- `smoke_ok`: `TRUE` if the bundled validator accepts a known-good item.

Examples

```
zucbor_info()
```

Index

as.POSIXct(), *14*
as_cbor, 2
as_cbor(), *9*, *14*, *15*

cbor-values, 3, *9*, *15*
cbor_annotate, 5
cbor_bigint *(*cbor-values*)*, 3
cbor_decode, 6
cbor_decode(), *2*, *3*, *9*, *11–13*, *15*, *16*, *18*
cbor_decode_prefix, 9
cbor_decode_seq *(*cbor_decode*)*, 6
cbor_decode_seq(), *11*, *15*, *16*
cbor_diagnose, 11
cbor_diagnose(), *5*, *6*
cbor_encode, 12
cbor_encode(), *2–4*, *8*, *9*, *18*
cbor_encode_seq *(*cbor_encode*)*, 12
cbor_map *(*cbor-values*)*, 3
cbor_read, 15
cbor_read(), *9*, *18*
cbor_read_seq *(*cbor_read*)*, 15
cbor_read_seq(), *18*
cbor_simple *(*cbor-values*)*, 3
cbor_tag *(*cbor-values*)*, 3
cbor_tag(), *2*, *14*
cbor_validate, 16
cbor_validate(), *5*, *6*, *9*, *11*, *14*

I(), *8*, *14*

url(), *16*

zucbor-conditions, *9*, *17*, 18
zucbor_info, 19