Package: zuproto (via r-universe)

November 28, 2026

Title 'Protocol Buffers' Decoding and Encoding with 'upb'

Version 0.1.0

Description Decodes and encodes 'Protocol Buffers' messages from descriptor sets rather than generated code. One message becomes a named list; many messages become a data frame built column by column. Uses a bundled copy of the 'upb' runtime (<https://github.com/protocolbuffers/protobuf>), so no system 'protobuf' library is required. Schemas are compiled into descriptor sets beforehand, with 'protoc' or 'buf'.

License MIT + file LICENSE

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

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

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

Depends R (>= 4.1)

Suggests bit64, testthat (>= 3.0.0), withr, zujson

Config/testthat/edition 3

Encoding UTF-8

Language en-GB

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph], Google LLC [cph] (upb and utf8_range, bundled in src/vendor), Yibo Cai [cph] (utf8_range, bundled in src/vendor/utf8_range)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

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

Date/Publication 2026-11-28 21:07:26 UTC

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

RemoteRef HEAD

RemoteSha a20889cbf2743024282fa3d6f2f201b755a05780

Contents

proto_decode 2
proto_decode_df 4
proto_encode 6
proto_enums 8
proto_pool 9
proto_read_delimited 10
proto_to_json 12
proto_types 13
proto_unpack_any 15
zuproto-conditions 16
zuproto_info 17

Index **18**

`proto_decode` *Decode one Protocol Buffers message*

Description

Decodes the bytes of one message of type `type` into a named list, one element per field, by the field's name. upb decodes the whole message first; R values are built only from a message that decoded completely.

Usage

```
proto_decode(
  x,
  type,
  pool,
  int64 = c("double", "character", "integer64"),
  enums = c("factor", "integer", "character"),
  defaults = TRUE,
  unknown = c("drop", "keep"),
  bytes = c("raw", "base64"),
  max_size = 64 * 1024^2,
  max_depth = 100L
)
```

Arguments

`x`	The serialised message: a raw vector, a path to a file, or a connection. A file or connection is read up to `max_size` bytes.
`type`	The message type's full name, such as `"pkg.Message"`.
`pool`	A pool from `proto_pool()` holding `type`.

`int64` How 64-bit integer fields become R values: `"double"` (values beyond 2^53 lose precision, with a `zuproto_precision_loss` warning), `"character"` (exact decimal strings) or `"integer64"` (`bit64::integer64()` vectors, exact for signed values; needs bit64). Under `"integer64"`, `-2^63` (bit64's `NA`) and `uint64` values above `2^63 - 1` become `NA` with a `zuproto_precision_loss` warning.

`enums` How enum fields become R values: `"factor"`, `"integer"` (the numbers) or `"character"` (the names). A number the enum does not define is `NA` with a `zuproto_unknown_enum` warning, except as an integer.

`defaults` Whether unset fields without presence take their defaults (`TRUE`) or are left out (`FALSE`).

`unknown` `"drop"` to discard fields the type does not define, `"keep"` to keep their bytes in an attribute `proto_unknown` on the message (and on each nested message) holding them.

`bytes` How `bytes` fields become R values: `"raw"` or `"base64"` (strings, as protobuf JSON writes them).

`max_size` The largest input accepted, in bytes. upb's memory for the decode is also capped. The cap is a multiple of the input's size, set by the message type: the most a message of that type could need.

`max_depth` The deepest nesting accepted: how many levels of messages within messages, the top-level message being level 0. A message holding a message holding a message is depth 2. A map entry counts as a level too, as upb counts it, so a message reached through a map is two levels down. At most 1000, so that hostile input cannot exhaust the C stack.

Value

A named list. An `int32`-family value of `-2^31` is `NA`, with a `zuproto_precision_loss` warning, since R reserves it for `NA`.

Protobuf to R

Protobuf	R
`double`, `float`	double
`int32`, `sint32`, `sfixed32`	integer
`uint32`, `fixed32`	double, exact
`int64`, `uint64`, `sint64`, `fixed64`, `sfixed64`	per `int64`
`bool`	logical
`string`	character, UTF-8
`bytes`	raw, or per `bytes`
enum	factor with the enum's value names as levels, or per `enums`
message	named list
`repeated` scalar	vector
`repeated` message	list of lists
`map<K, V>`	named vector or named list, sorted by key
`oneof`	the member that is set
extension	element named by its full name in brackets, `[pkg.ext]`

Well-known types: `Timestamp` becomes `POSIXct` in UTC, rounded to the microsecond; `Duration` a `difftime` in seconds; a wrapper (`Int32Value` and the rest) its value; `Struct` a named list, `Value` its value (`NULL` for `null`) and `ListValue` a list, as in protobuf JSON; `Any` a list of `type_url` and `value` (the packed bytes); `FieldMask` a character vector of paths; `Empty` an empty list. Repeated `Timestamp`, `Duration` and wrapper fields become vectors.

A field that tracks presence (a message, a oneof member, a proto2 or proto3 `optional` field) is left out when it is not set. A field that does not track presence cannot be told apart from its default on the wire. With `defaults = TRUE` it takes its default (`0`, `""`, `FALSE`, the enum's first value, an empty vector), as other protobuf runtimes do. With `defaults = FALSE` it is left out when it holds its default.

Required fields are not checked: a proto2 `required` field missing from the input decodes like any unset field.

Concatenated messages merge, as protobuf specifies: decoding `c(raw1, raw2)` gives the message `raw1` holds with `raw2` merged into it. A singular field takes the last value, a repeated field appends, a map adds entries (a later key replaces an earlier one) and a message field merges. This is protobuf's `MergeFromString()`. A message's encoded size is `length(proto_encode(x, type, pool))`.

See Also

zuproto-conditions for the errors it raises.

Examples

```
pool <- proto_pool()
# A Timestamp of 1.5 seconds after the epoch: field 1 = 1, field 2 = 5e8.
bytes <- as.raw(c(0x08, 0x01, 0x10, 0x80, 0xca, 0xb5, 0xee, 0x01))
proto_decode(bytes, "google.protobuf.Timestamp", pool)
proto_decode(as.raw(c(0x08, 0x2a)), "google.protobuf.Int32Value", pool)
```

---

proto_decode_df *Decode many messages into a data frame*

Description

Decodes many messages of one type into a data frame, one row per message, built column by column: every message is decoded first, then each column is filled across all of them. A message that fails stops the whole call, and the condition's `index` says which one.

Usage

```
proto_decode_df(
  x,
  type,
  pool,
```

```
  field = NULL,
  int64 = c("double", "character", "integer64"),
  enums = c("factor", "integer", "character"),
  defaults = TRUE,
  bytes = c("raw", "base64"),
  max_size = 64 * 1024^2,
  max_depth = 100L,
  max_rows = 1e+07
)
```

Arguments

x	For `proto_decode_df()`, a list of raw vectors, one message each; with `field`, one message as a raw vector, a path or a connection. For `proto_read_delimited()`, a raw vector, a path or a connection holding a length-delimited stream.
type	The message type's full name, such as `"pkg.Message"`.
pool	A pool from `proto_pool()` holding `type`.
field	The name of a repeated message field of `type`, or `NULL`.
int64	How 64-bit integer fields become R values: `"double"` (values beyond 2^53 lose precision, with a `zuproto_precision_loss` warning), `"character"` (exact decimal strings) or `"integer64"` (`bit64::integer64()` vectors, exact for signed values; needs bit64). Under `"integer64"`, `-2^63` (bit64's `NA`) and `uint64` values above `2^63 - 1` become `NA` with a `zuproto_precision_loss` warning.
enums	How enum fields become R values: `"factor"`, `"integer"` (the numbers) or `"character"` (the names). A number the enum does not define is `NA` with a `zuproto_unknown_enum` warning, except as an integer.
defaults	Whether unset fields without presence take their defaults (`TRUE`) or are left out (`FALSE`).
bytes	How `bytes` fields become R values: `"raw"` or `"base64"` (strings, as protobuf JSON writes them).
max_size	The largest input accepted, in bytes. upb's memory for the decode is also capped. The cap is a multiple of the input's size, set by the message type: the most a message of that type could need.
max_depth	The deepest nesting accepted: how many levels of messages within messages, the top-level message being level 0. A message holding a message holding a message is depth 2. A map entry counts as a level too, as upb counts it, so a message reached through a map is two levels down. At most 1000, so that hostile input cannot exhaust the C stack.
max_rows	The largest number of rows accepted.

Details

With `field`, the input is one message instead, and the rows are the elements of its repeated message field `field`. That is how a feed that wraps its records in one message becomes a table. A GTFS Realtime `FeedMessage` is one, with its records in the `entity` field.

Value

A data frame.

Columns

Field	Column
scalar, enum, scalar-like well-known type	atomic vector, as in `proto_decode()`
`bytes`	list of raw vectors
message	data frame (a column of the frame)
`repeated` scalar	list of vectors
`repeated` message	list of data frames
`map`	list of named vectors or lists
`Struct`, `Value`, `ListValue`, `Any`	list, as in `proto_decode()`
`oneof`	a column per member, plus `<oneof>_case` with the member set
extension	a column named `[pkg.ext]` per extension the pool knows

A field a row does not set is `NA` (`NULL` in a list column) when the field tracks presence. Otherwise it takes its default, or is `NA` with `defaults = FALSE`. The `<oneof>_case` column is named `<oneof>_case_` instead when a field has that name. A message field that no row sets is an all-`NA` logical column rather than a data frame, which is also where the columns of a recursive type end. A message type with no fields, such as `Empty`, is a list column: `list()` where set, `NULL` where not. A nested message that is set but has every field unset reads as unset, since its row is all `NA`.

See Also

`proto_read_delimited()` for length-delimited streams.

Examples

```
pool <- proto_pool()
# Three FieldDescriptorProto messages, one per row.
type <- "google.protobuf.FieldDescriptorProto"
msgs <- lapply(c("id", "name", "when"), function(name) {
  proto_encode(list(name = name, number = 1L), type, pool)
})
df <- proto_decode_df(msgs, type, pool)
df[c("name", "number")]
```

`proto_encode` *Encode one Protocol Buffers message*

Description

Encodes a named list as one message of type `type`, the inverse of `proto_decode()`: every value `proto_decode()` returns encodes back to the same message.

Usage

```
proto_encode(
  x,
  type,
  pool,
  deterministic = TRUE,
  unknown = TRUE,
  check_required = TRUE,
  extra = c("error", "ignore")
)
```

Arguments

x	A named list, one element per field. For a well-known type with an R scalar form (Timestamp, Duration, wrappers), that scalar.
type	The message type's full name, such as "pkg.Message".
pool	A pool from proto_pool() holding type.
deterministic	Whether to encode deterministically: map entries sorted by key, so equal values always give equal bytes.
unknown	Whether to write back unknown fields kept by proto_decode(unknown = "keep") in a proto_unknown attribute.
check_required	Whether a missing proto2 required field is an error.
extra	"error" to refuse names the type does not define, "ignore" to skip them.

Details

Each element is a field, by its name, or an extension, by its full name in brackets ("[pkg.ext]"). An element that is NULL or NA leaves its field unset, as does a field the list leaves out. Values take the R forms proto_decode() gives them, and also:

- integer kinds: integers, or doubles that are whole and in range; 64-bit kinds also decimal strings, which are exact;
- enums: factors, value names, or numbers (a closed enum accepts only the numbers it defines);
- Timestamp: POSIXct or seconds since the epoch (not a Date, whose number is days: convert it with as.POSIXct()); Duration: difftime or seconds; a wrapper: its value;
- strings: valid UTF-8, or in an encoding R translates to it;
- Value: NULL, a logical, number or string, a named list (a Struct), or an unnamed list or vector (a ListValue).

bytes fields take raw vectors (a list of them when repeated).

Value

A raw vector.

See Also

zuproto-conditions for the errors it raises; each one about a value carries `path`, the field at fault, such as `"points[2].x"`.

Examples

```
pool <- proto_pool()
proto_encode(42L, "google.protobuf.Int32Value", pool)
when <- as.POSIXct("2026-10-25 12:00:00", tz = "UTC")
raw <- proto_encode(when, "google.protobuf.Timestamp", pool)
proto_decode(raw, "google.protobuf.Timestamp", pool)
```

`proto_enums` *Enum values*

Description

Lists the values of one enum type in a pool: the levels a decoded enum field takes as a factor, with their numbers.

Usage

```
proto_enums(pool, type)
```

Arguments

`pool`	A pool from `proto_pool()`.
`type`	An enum type's full name, such as `"google.protobuf.NullValue"`. `proto_fields()` gives it as a field's `type_name`.

Value

A data frame with one row per value, in definition order, and columns `name` and `number`. An alias (`allow_alias`) is a row of its own with the number it shares. R has no integer for -2^31, so that number is `NA`, with a `zuproto_precision_loss` warning.

Examples

```
pool <- proto_pool()
proto_enums(pool, "google.protobuf.Field.Kind")
```

proto_pool *Descriptor pools*

Description

A pool holds the message types zuproto can decode and encode. It is built from serialised `FileDescriptorSets`, which `protoc` and `buf` write:

Usage

```
proto_pool(x = NULL)

proto_pool_add(pool, x)
```

Arguments

`x`	A serialised `FileDescriptorSet`: a raw vector, a path to a file, or a connection. `NULL` for the well-known types only.
`pool`	A pool from `proto_pool()`.

Details

```
protoc --include_imports --descriptor_set_out=schema.binpb schema.proto
buf build -o schema.binpb
```

Every pool starts with the well-known types (`google/protobuf/*.proto`, including `descriptor.proto`), so a set need not include them.

Descriptor sets are trusted input: upb checks them, but less thoroughly than `protoc` does. Load sets you or your tools produced, not ones from an untrusted source. Message bytes, by contrast, are never trusted.

Files are added in dependency order, whatever order the set lists them in. A file already in the pool is skipped if its contents are identical. It is a `zuproto_pool_conflict` if they are not, and the same holds for a file listed twice. The exception is a bundled well-known type's file, which another `protoc` release writes differently: the pool keeps its own copy and skips the one in the set. A file importing one that is neither in the set nor in the pool is a `zuproto_pool_error` naming both, and nothing from the set is added. Extensions are loaded with the files that declare them.

Value

`proto_pool()` returns a pool. `proto_pool_add()` returns `pool`, invisibly, after adding `x` to it.

See Also

`proto_types()` and `proto_fields()` to see what a pool holds.

Examples

```
pool <- proto_pool()
pool
"google.protobuf.Timestamp" %in% proto_types(pool)$name
```

`proto_read_delimited` *Length-delimited streams*

Description

In a length-delimited stream, each message is preceded by its length as a varint. Java's `writeDelimitedTo()` writes such streams, as do Python's `_VarintBytes` helpers and many `.pb` log files.

Usage

```
proto_read_delimited(
  x,
  type,
  pool,
  each = NULL,
  chunk_rows = 10000,
  int64 = c("double", "character", "integer64"),
  enums = c("factor", "integer", "character"),
  defaults = TRUE,
  bytes = c("raw", "base64"),
  max_size = 64 * 1024^2,
  max_depth = 100L,
  max_rows = 1e+07
)

proto_write_delimited(x, type, pool, path = NULL, ...)
```

Arguments

`x`	For `proto_decode_df()`, a list of raw vectors, one message each; with `field`, one message as a raw vector, a path or a connection. For `proto_read_delimited()`, a raw vector, a path or a connection holding a length-delimited stream.
`type`	The message type's full name, such as `"pkg.Message"`.
`pool`	A pool from `proto_pool()` holding `type`.
`each`	`NULL`, or a function called with each chunk's data frame.
`chunk_rows`	The number of messages in each chunk given to `each`.
`int64`	How 64-bit integer fields become R values: `"double"` (values beyond 2^53 lose precision, with a `zuproto_precision_loss` warning), `"character"` (exact decimal strings) or `"integer64"` (`bit64::integer64()` vectors, exact for signed values; needs bit64). Under `"integer64"`, `-2^63` (bit64's `NA`) and `uint64` values above `2^63 - 1` become `NA` with a `zuproto_precision_loss` warning.

enums	How enum fields become R values: "factor", "integer" (the numbers) or "character" (the names). A number the enum does not define is NA with a zuproto_unknown_enum warning, except as an integer.
defaults	Whether unset fields without presence take their defaults (TRUE) or are left out (FALSE).
bytes	How bytes fields become R values: "raw" or "base64" (strings, as protobuf JSON writes them).
max_size	The largest input accepted, in bytes. upb's memory for the decode is also capped. The cap is a multiple of the input's size, set by the message type: the most a message of that type could need.
max_depth	The deepest nesting accepted: how many levels of messages within messages, the top-level message being level 0. A message holding a message holding a message is depth 2. A map entry counts as a level too, as upb counts it, so a message reached through a map is two levels down. At most 1000, so that hostile input cannot exhaust the C stack.
max_rows	The largest number of rows accepted. With each, it bounds each chunk, not the whole stream, which is read a chunk at a time and never held whole.
path	NULL to return the stream, or a path to write it to.
...	Passed to proto_encode() for each message.

Details

proto_read_delimited() reads a stream into a data frame like proto_decode_df() builds. With each, it reads the stream in chunks of chunk_rows messages and calls each(df) on each chunk instead, so a stream larger than memory can be processed. proto_write_delimited() writes a stream.

A stream that ends inside a length or a message is a zuproto_parse_error.

proto_write_delimited() takes a data frame whose columns follow the column table of proto_decode_df(), one message per row, or a list of messages as proto_encode() takes them. NA values leave fields unset, and <oneof>_case columns are ignored. A NULL cell leaves its field unset too, so a Value that held JSON null is not written back: a data frame cannot tell it from an unset one.

Value

proto_read_delimited(): a data frame, or with each, NULL, invisibly.

proto_write_delimited(): the stream as a raw vector, or with path, the path, invisibly.

Examples

```
pool <- proto_pool()
rows <- data.frame(seconds = c(1, 2, 3), nanos = 0L)
stream <- proto_write_delimited(rows, "google.protobuf.Duration", pool)
proto_read_delimited(stream, "google.protobuf.Duration", pool)
```

proto_to_json *Protobuf JSON*

Description

Converts a binary message to protobuf's canonical JSON mapping and back, through upb. JSON field names are the `json_name`s (lowerCamelCase) by default, 64-bit integers are strings, bytes are base64, and well-known types take their special JSON forms (`Timestamp` as an RFC 3339 string, `Struct` as an object, and so on).

Usage

```
proto_to_json(
  x,
  type,
  pool,
  emit_defaults = FALSE,
  proto_names = FALSE,
  enums = c("name", "integer"),
  parse = FALSE,
  max_size = 64 * 1024^2,
  max_depth = 100L
)

proto_from_json(
  json,
  type,
  pool,
  ignore_unknown = FALSE,
  check_required = TRUE,
  max_size = 64 * 1024^2
)
```

Arguments

`x`	For `proto_to_json()`, the binary message: a raw vector, a path or a connection.
`type`	The message type's full name, such as `"pkg.Message"`.
`pool`	A pool from `proto_pool()` holding `type`.
`emit_defaults`	Whether to write fields that hold their defaults.
`proto_names`	Whether to use the `.proto` field names rather than their `json_name`s.
`enums`	`"name"` or `"integer"`: how enum values are written.
`parse`	Whether to return the JSON parsed into an R value by `zujson::json_parse()` rather than as text.
`max_size`	The largest input accepted, in bytes. upb's memory for the decode is also capped. The cap is a multiple of the input's size, set by the message type: the most a message of that type could need.

max_depth	The deepest nesting accepted: how many levels of messages within messages, the top-level message being level 0. A message holding a message holding a message is depth 2. A map entry counts as a level too, as upb counts it, so a message reached through a map is two levels down. At most 1000, so that hostile input cannot exhaust the C stack.
json	Protobuf JSON for one message: a single string of JSON text, or an R value to write as JSON with zujson::json_write().
ignore_unknown	Whether to skip JSON fields the type does not define, rather than fail.
check_required	Whether a missing proto2 required field is an error.

Details

Any messages convert only when the type they pack is in the pool.

The entries of a map field, and so the keys of a Struct, are written in no fixed order: it can differ between R sessions. With parse = TRUE they are sorted by key, as proto_decode() sorts maps, so the parsed value is the same in every session. Compare JSON as parsed values, not as text.

JSON meets R values through zujson, which must be installed for it. proto_to_json(parse = TRUE) returns zujson::json_parse()'s R value. proto_from_json() takes an R value as well as JSON text, and writes it with zujson::json_write(). There, a length-1 vector is a JSON scalar, so give a repeated field of one element as I(x) or list(x).

Value

proto_to_json(): a single string, or with parse = TRUE an R value. proto_from_json(): the message as a raw vector, encoded deterministically.

Examples

```
pool <- proto_pool()
raw <- proto_encode(
  as.POSIXct("2026-10-25 12:00:00", tz = "UTC"),
  "google.protobuf.Timestamp", pool
)
json <- proto_to_json(raw, "google.protobuf.Timestamp", pool)
json
identical(proto_from_json(json, "google.protobuf.Timestamp", pool), raw)
```

---

proto_types *Message types and their fields*

Description

proto_types() lists every message type in a pool, nested types included, as a data frame. proto_fields() lists the fields of one type, followed by the extensions the pool knows for it.

Usage

```
proto_types(pool)

proto_fields(pool, type)
```

Arguments

pool	A pool from proto_pool().
type	A message type's full name, such as "google.protobuf.Timestamp".

Details

An extension is named by its full name in brackets, as in protobuf's JSON mapping, for example "[transit_realtime.trip_descriptor_ext]". Decoded messages use the same names.

Value

proto_types(): a data frame with columns name (full name), file (the .proto file declaring it) and fields (the number of fields), sorted by name. Map entry types are left out.

proto_fields(): a data frame with one row per field and columns

- name: the field's name, or [full.name] for an extension;
- number: the field number;
- type: the protobuf type, such as "int32", "string", "message" or "enum";
- label: "optional", "required" or "repeated";
- type_name: the message or enum type's full name, NA for scalars;
- oneof: the oneof the field belongs to, or NA;
- json_name: the name in protobuf JSON;
- presence: whether the field tracks whether it is set;
- packed: whether repeated values are packed on the wire;
- map: whether the field is a map;
- extension: whether the field is an extension.

Examples

```
pool <- proto_pool()
head(proto_types(pool))
proto_fields(pool, "google.protobuf.Timestamp")
```

`proto_unpack_any` *Pack and unpack an Any*

Description

`proto_unpack_any()` decodes the message a `google.protobuf.Any` packs, as `proto_decode()` decodes any message. The type is the part of `type_url` after its last `/`, and must be in the pool. `proto_pack_any()` is the inverse: it encodes a message with `proto_encode()` and names its type, giving a value to set an `Any` field to.

Usage

```
proto_unpack_any(x, pool, ...)

proto_pack_any(x, type, pool, prefix = "type.googleapis.com/", ...)
```

Arguments

`x`	For `proto_unpack_any()`, an `Any` as `proto_decode()` returns it: a list of `type_url` and `value` (the packed bytes). For `proto_pack_any()`, the message, as `proto_encode()` takes it.
`pool`	A pool from `proto_pool()` holding the packed type.
`...`	Passed to `proto_decode()` or `proto_encode()`.
`type`	The message type's full name.
`prefix`	The `type_url` before the type name, ending in `/`. Protobuf's JSON mapping and most runtimes expect the default.

Value

`proto_unpack_any()`: the packed message, as `proto_decode()` returns it. `proto_pack_any()`: an `Any`, a list of `type_url` and `value`.

Examples

```
pool <- proto_pool()
any <- proto_pack_any(42L, "google.protobuf.Int32Value", pool)
any$type_url
proto_unpack_any(any, pool)
```

`zuproto-conditions` *Conditions raised by zuproto*

Description

Every error zuproto 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 `zuproto_error`.

Details

`zuproto_invalid_argument` An argument was unusable, such as a pool that is not one, or a type name that is not a single string. Carries `arg`, the argument at fault.

`zuproto_io_error` A file could not be opened or read, or a connection could not be read from.

`zuproto_pool_error` A descriptor set could not be added to a pool: the bytes are not a `FileDescriptorSet`, a file imports one that is neither in the set nor in the pool, or upb rejected a file. Carries `file`, the file at fault, when there is one.

`zuproto_pool_conflict` A subclass of `zuproto_pool_error`: a file of the same name is already in the pool, or earlier in the set, with different contents. Carries `file`.

`zuproto_unknown_type` A message type is not in the pool. Carries `type`.

`zuproto_parse_error` The input is not a valid message of the type: it ends early, has a bad tag, a length beyond the input, or the wrong wire type. Carries `status`.

`zuproto_invalid_utf8` A subclass of `zuproto_parse_error`: a `string` field is not valid UTF-8, in proto2 as in proto3.

`zuproto_unrepresentable` The input is valid but holds a value R cannot: a string containing a NUL byte, or longer than an R string may be. Carries `field`, the field's full name.

`zuproto_type_mismatch` `proto_encode()` was given a value of the wrong form for its field: the wrong R type, a number out of range or not whole, a name that is not a value of the enum, `NA` inside a repeated field, a string that is not valid UTF-8 text, a date or time where a number is expected (or a `Date` for a `Timestamp`), a second member of a oneof, or a field or map key given twice. Carries `path`, the field at fault, such as `"points[2].x"`.

`zuproto_unknown_field` `proto_encode()` was given a name the message type does not define. Carries `path`.

`zuproto_missing_required` `proto_encode()` with `check_required = TRUE`: a proto2 `required` field is not set.

`zuproto_json_error` `proto_from_json()` was given JSON that is not valid, or not the type's, or `proto_to_json()` could not write a message (an `Any` whose packed type is not in the pool). Carries `detail`, upb's description.

`zuproto_limit_error` A limit was reached, and a subclass names which. `zuproto_size_limit`: the input, or the memory decoding it needs, is too large for `max_size`. `zuproto_depth_limit`: messages nest deeper than `max_depth`. `zuproto_row_limit`: there are more rows than `max_rows`. It carries `limit`, the argument's name, and `limit_value`, its value. A decode can need more memory than any message of its type and size could. Then `limit` is `"memory"` and `limit_value` is `NA`, since no argument raises that ceiling. `proto_encode()` raises `zuproto_depth_limit` for messages nested deeper than 1000 levels, the most `max_depth` lets `proto_decode()` accept.

`zuproto_internal_error` Memory ran out, or upb reported a status zuproto does not expect. Carries `code`.

An error about one message of `proto_decode_df()`, `proto_read_delimited()` or `proto_write_delimited()` also carries `index`, the message's position in the batch or stream. It is a double, since a stream can hold more than 2^31 - 1 messages.

Warnings inherit `zuproto_warning`:

`zuproto_precision_loss` Values could not be represented exactly: 64-bit integers beyond 2^53 decoded as doubles, or a 32-bit `-2^31`, which R reserves for `NA`. Carries `n`, how many.

`zuproto_unknown_enum` Enum numbers the enum does not define became `NA`. Carries `n`.

Examples

```
pool <- proto_pool()
tryCatch(
  proto_fields(pool, "no.such.Message"),
  zuproto_unknown_type = function(e) e$type
)
```

---

zuproto_info *Build information*

Description

Reports the zuproto version and the Protocol Buffers release whose upb runtime is bundled.

Usage

```
zuproto_info()
```

Value

A list of class `zuproto_info` with elements:

- `version`: zuproto's version.
- `protobuf`: the protobuf release tag upb was vendored from, such as `"v36.2"`.
- `protobuf_commit`: that tag's commit.
- `limits`: the default `max_size` (bytes) and `max_depth` of `proto_decode()`.

Examples

```
zuproto_info()
```

Index

as.POSIXct(), *7*

bit64::integer64(), *3*, *5*, *10*

proto_decode, 2
proto_decode(), *6*, *13*, *15–17*
proto_decode_df, 4
proto_decode_df(), *11*, *17*
proto_encode, 6
proto_encode(), *11*, *15, 16*
proto_enums, 8
proto_fields *(*proto_types*)*, 13
proto_fields(), *8, 9*
proto_from_json *(*proto_to_json*)*, 12
proto_from_json(), *16*
proto_pack_any *(*proto_unpack_any*)*, 15
proto_pool, 9
proto_pool(), *2*, *5*, *7, 8*, *10*, *12*, *14, 15*
proto_pool_add *(*proto_pool*)*, 9
proto_read_delimited, 10
proto_read_delimited(), *6*, *17*
proto_to_json, 12
proto_to_json(), *16*
proto_types, 13
proto_types(), *9*
proto_unpack_any, 15
proto_write_delimited
        *(*proto_read_delimited*)*, 10
proto_write_delimited(), *17*

zujson::json_parse(), *12, 13*
zujson::json_write(), *13*
zuproto-conditions, *4*, *8*, 16
zuproto_info, 17