

Package: zufast (via r-universe)

October 21, 2026

Title Fast Portable 'C' Primitives for Parsing and Formatting Data

Version 0.1.0

Description A header-only library of small, fast, portable 'C99' primitives for converting external text and bytes into native values and back: correctly rounded number parsing (from 'fast_float', through its 'C' port 'ffc.h'), shortest round-trip number formatting (from 'Ryu'), ISO 8601 dates and timestamps, booleans, universally unique identifiers (UUIDs), hexadecimal and Base64, 'XXH3' hashing (from 'xxHash'), half-precision floats, endian helpers and UTF-8 validation. Other packages use it through 'LinkingTo' alone; the 'R' functions exist to test, benchmark and demonstrate the 'C' layer.

License MIT + file LICENSE

Copyright See inst/COPYRIGHTS.

URL <https://pedrobtz.github.io/zufast/>,
<https://github.com/pedrobtz/zufast>

BugReports <https://github.com/pedrobtz/zufast/issues>

Depends R (>= 4.1)

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph], Bjoern Hoehrmann [cph] (UTF-8 decoder state table in inst/include/zufast/utf8.h), Daniel Lemire [cph] (fast_float, ported to C as ffc.h), João Paulo Magalhaes [cph] (fast_float, ported to C as ffc.h), The fast_float authors [cph] (fast_float, ported to C as ffc.h), K Coleman Nix [cph] (ffc.h, the C port of fast_float), Ulf Adams [cph] (Ryu), Yann Collet [cph] (xxHash)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>
Repository <https://cran.r-universe.dev>
Date/Publication 2026-10-21 14:30:02 UTC
RemoteUrl <https://github.com/cran/zufast>
RemoteRef HEAD
RemoteSha 640ceec97f0ad7186b60578638684decf244e0bd

Contents

fast_base64_encode	2
fast_format_datetime	3
fast_format_double	4
fast_hash	5
fast_hex_encode	5
fast_info	6
fast_parse_date	7
fast_parse_double	8
fast_utf8_valid	9

Index	10
--------------	-----------

fast_base64_encode	<i>Base64 encoding and decoding</i>
--------------------	-------------------------------------

Description

Standard (RFC 4648 section 4) or URL-safe (section 5) Base64. Decoding is strict: no whitespace, nothing outside the selected alphabet, padding either complete or absent, and the unused trailing bits zero, so that every accepted input has exactly one encoding.

Usage

```
fast_base64_encode(x, url = FALSE, pad = TRUE)

fast_base64_decode(x, url = FALSE)
```

Arguments

x	For encoding, a raw vector (encoded as one string) or a character vector (the bytes of each element encoded separately, NA kept). For decoding, a character vector.
url	Use the URL-safe alphabet (- and _ instead of + and /).
pad	Write = padding.

Value

fast_base64_encode() returns a character vector. fast_base64_decode() returns a list of raw vectors, one per element of x, with NULL where the element is NA or not valid Base64.

Examples

```
fast_base64_encode(charToRaw("hello"))
fast_base64_encode("hello", url = TRUE, pad = FALSE)
rawToChar(fast_base64_decode("aGVsbG8=")[[1]])
```

fast_format_datetime *Format dates and timestamps as RFC 3339*

Description

Writes a POSIXct as YYYY-MM-DDTHH:MM:SS[.fff]Z in UTC, or a Date as YYYY-MM-DD. Years outside 0000 to 9999 are written in the ISO 8601 expanded form with a sign.

Usage

```
fast_format_datetime(x, digits = NULL)
```

Arguments

x	A POSIXct or Date vector.
digits	For POSIXct, the number of fraction digits, 0 to 9. NULL writes the shortest of 0, 3 or 6 digits that represents the time at microsecond resolution, which is about the precision a POSIXct double holds for present-day times.

Value

A character vector, NA where x is NA, not finite, or outside the calendar's range (a day count beyond the int32_t range, about plus or minus 5.8 million years).

Examples

```
fast_format_datetime(as.POSIXct("2024-03-01 12:30:00.25", tz = "UTC"))
fast_format_datetime(Sys.time(), digits = 0)
fast_format_datetime(as.Date("2024-03-01"))
```

fast_format_double	<i>Format doubles with the shortest round-trip digits</i>
--------------------	---

Description

Writes each value with the fewest significant digits that parse back to exactly the same double, in the notation of ECMAScript's `Number.prototype.toString()`: positional for decimal exponents from -6 to 20 (0.000001, 0.1, 123.5, 100), d.ddde+x otherwise (1e+21, 1e-7).

Usage

```
fast_format_double(x, scientific = FALSE, trailing_zero = FALSE)
```

Arguments

x	A double vector.
scientific	Always use d.ddde+x notation.
trailing_zero	Write integral values in positional notation with a trailing .0, as in "1.0", so that they read back as non-integers in languages that distinguish the two.

Details

This differs from `as.character()`, which writes 15 significant digits: `as.character(0.1 + 0.2)` is "0.3", while `fast_format_double(0.1 + 0.2)` is "0.30000000000000004", because those are different doubles. `fast_parse_double(fast_format_double(x))` is identical to x for every double, including -0, which is written as "-0".

Value

A character vector; NA stays NA, NaN, Inf and -Inf are written as R writes them.

Examples

```
fast_format_double(c(1, 0.1, 0.1 + 0.2, 1e21, 1e-7, -0, NA, Inf))
fast_format_double(123.456, scientific = TRUE)
fast_format_double(c(1, 2.5), trailing_zero = TRUE)
```

fast_hash	<i>XXH3 hashes</i>
-----------	--------------------

Description

Computes the 64- or 128-bit XXH3 hash of raw bytes or of the bytes of each string, as lower-case hexadecimal in xxHash's canonical (big-endian) form. The values are identical to those of the reference implementation, so they can be compared with hashes computed elsewhere.

Usage

```
fast_hash(x, bits = 64, seed = 0)
```

Arguments

x	A raw vector (hashed as one input) or a character vector (each element hashed separately; the encoding is not converted, the bytes are hashed as stored).
bits	64 or 128.
seed	A whole number from 0 to 2^{53} .

Details

XXH3 is not a cryptographic hash. Use it for caches, deduplication, fingerprints and change detection, never where an adversary could choose the input to produce a collision; that needs a cryptographic digest.

Value

A character vector of 16 (64-bit) or 32 (128-bit) hex digits, NA where x is NA.

Examples

```
fast_hash(charToRaw("hello"))
fast_hash(c("a", "b", NA), bits = 128)
fast_hash("hello", seed = 42)
```

fast_hex_encode	<i>Hexadecimal encoding and decoding</i>
-----------------	--

Description

fast_hex_encode() writes two hex digits per byte. fast_hex_decode() accepts digits in either case and rejects an odd length or any other byte.

Usage

```
fast_hex_encode(x, upper = FALSE)
```

```
fast_hex_decode(x)
```

Arguments

x For encoding, a raw vector (encoded as one string) or a character vector (the bytes of each element encoded separately, NA kept). For decoding, a character vector.

upper Use upper-case digits.

Value

`fast_hex_encode()` returns a character vector. `fast_hex_decode()` returns a list of raw vectors, one per element of `x`, with `NULL` where the element is NA or not valid hex.

Examples

```
fast_hex_encode(as.raw(c(0xde, 0xad, 0xbe, 0xef)))
fast_hex_decode(c("DEADbeef", "abc", NA))
```

fast_info

Information about the compiled zufast headers

Description

Reports, from compiled code, the version of the zufast headers the package's own shared object was built with, the versions of the vendored libraries compiled into it, and the compiler and build flags that built it. Consumers that use `LinkingTo: zufast` carry their own copy of the code; this describes zufast's copy only.

Usage

```
fast_info()
```

Value

A named list with elements `version` (a string such as "0.1.0"), `version_major`, `version_minor`, `version_patch` (integers), `compiler` (a string), `build` (a named character vector: `c_standard`, the value of `__STDC_VERSION__`; `optimized`, `ndebug` and `int128`, each "true" or "false"; `fortify_source`, the `_FORTIFY_SOURCE` level or "0"; `endian`; and `simd`, the widest x86 or Arm vector extension the compiler targeted) and `vendored` (a named character vector of upstream versions).

Examples

```
fast_info()$version
```

fast_parse_date

*Parse ISO 8601 / RFC 3339 dates and timestamps***Description**

`fast_parse_date()` parses YYYY-MM-DD. `fast_parse_datetime()` parses a date, optionally followed by T, t or a space and HH:MM, HH:MM:SS or HH:MM:SS.fff (1 to 9 fraction digits; further digits are discarded), optionally followed by Z, z, +HH:MM, +HHMM or +HH (or -). `fast_datetime_fields()` returns the parsed fields themselves, for callers that apply a time zone of their own.

Usage

```
fast_parse_date(x)
```

```
fast_parse_datetime(x)
```

```
fast_datetime_fields(x)
```

Arguments

x A character vector.

Details

Validation is complete: year 0000 to 9999, month 1 to 12, a day that exists in the proleptic Gregorian calendar, hour 0 to 23, minute 0 to 59, second 0 to 60 (a leap second counts as the first second of the next minute), an offset within 23:59. Anything else, including a string with trailing bytes, is NA without a warning.

Value

`fast_parse_date()` returns a Date vector. `fast_parse_datetime()` returns a POSIXct vector in UTC: an offset is applied, and a timestamp without one is taken to be UTC. `fast_datetime_fields()` returns a data frame with integer columns year, month, day, hour, minute, second, nanosecond, offset (seconds east of UTC, NA when none was given) and logical columns has_time and has_offset.

Examples

```
fast_parse_date(c("2024-02-29", "2023-02-29", NA))
fast_parse_datetime(c("2024-03-01T12:30:00Z", "2024-03-01 12:30:00.25+01:00"))
fast_datetime_fields("2024-03-01T12:30:00-05:00")
```

fast_parse_double	<i>Parse numbers</i>
-------------------	----------------------

Description

`fast_parse_double()` parses decimal and scientific notation, Inf, -Inf, Infinity and NaN in any case, with a leading - or +, after trimming ASCII whitespace. The result is correctly rounded: the nearest double to the decimal value, ties to even, on every platform.

Usage

```
fast_parse_double(x)
```

```
fast_parse_integer(x)
```

Arguments

x	A character vector.
---	---------------------

Details

This is not exactly `as.numeric()`: R accumulates digits in long double and is not correctly rounded, so the two differ in the last bit for roughly one ordinary decimal in five thousand ("0.799012" is one), and R's result depends on whether it was built with long double. Hexadecimal input, which `as.numeric()` accepts, is not accepted.

`fast_parse_integer()` parses a decimal integer and returns NA for anything that is not one or does not fit an R integer, rather than going through a double as `as.integer()` does: "1.5", "1e3" and "3000000000" are all NA.

Invalid input is NA with no warning.

Value

A double vector (`fast_parse_double()`) or an integer vector (`fast_parse_integer()`) of the same length as x.

Examples

```
fast_parse_double(c("1.5", "-2e-3", "+inf", "0x10", "abc", NA))
fast_parse_integer(c("42", "+7", "2147483647", "2147483648", "1.0"))
```

fast_utf8_valid	<i>Validate UTF-8</i>
-----------------	-----------------------

Description

Checks that bytes are well-formed UTF-8: no overlong forms, no surrogates (U+D800 to U+DFFF), nothing above U+10FFFF. The result agrees with `validUTF8()`; the declared encoding of a string is ignored, only its bytes are examined.

Usage

```
fast_utf8_valid(x)
```

Arguments

x	A character vector, or a raw vector holding one byte sequence.
---	--

Value

For a character vector, a logical vector of the same length, NA where x is NA. For a raw vector, a single TRUE or FALSE.

Examples

```
fast_utf8_valid(c("plain", rawToChar(as.raw(c(0x63, 0x61, 0x66, 0xc3, 0xa9)))), NA))
fast_utf8_valid(as.raw(c(0xc3, 0x28))) # invalid continuation byte
```

Index

`as.character()`, [4](#)
`as.numeric()`, [8](#)

`fast_base64_decode`
 (`fast_base64_encode`), [2](#)
`fast_base64_encode`, [2](#)
`fast_datetime_fields` (`fast_parse_date`),
 [7](#)
`fast_format_datetime`, [3](#)
`fast_format_double`, [4](#)
`fast_hash`, [5](#)
`fast_hex_decode` (`fast_hex_encode`), [5](#)
`fast_hex_encode`, [5](#)
`fast_info`, [6](#)
`fast_parse_date`, [7](#)
`fast_parse_datetime` (`fast_parse_date`), [7](#)
`fast_parse_double`, [8](#)
`fast_parse_integer` (`fast_parse_double`),
 [8](#)
`fast_utf8_valid`, [9](#)

`validUTF8()`, [9](#)