Package 'raddr'

September 21, 2026

Type Package

Title Show What an IP Address Literal Means Under Every Standard

Version 0.1.2

Language en-US

Description Standards and implementations disagree about what an IP address literal means: the string ``0177.0.0.1'' is rejected by the dotted-quad grammar, read as 127.0.0.1 by browsers, and read as 177.0.0.1 by some 'inet_pton' implementations. Most libraries pick one reading and discard the rest. This package reports them all, alongside the reason codes that explain each one, and classifies parsed values against the IANA special-purpose address registries. It is pure R, performs no network access, and returns facts rather than allow or deny verdicts.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.0.0)

Imports rlang (>= 1.1.7), vctrs (>= 0.7.0)

URL https://gitlab.com/bart-turczynski/raddr

BugReports https://gitlab.com/bart-turczynski/raddr/-/issues

X-schema.org-keywords ip-address, ipv4, ipv6, ip-parser, inet-pton, inet-aton, whatwg-url, cidr, iana, special-purpose-registry, nat64, teredo, 6to4, r, rstats, r-stats, r-package

Suggests bignum, bit64, digest, hedgehog, knitr, rmarkdown, spelling, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Bart Turczynski [aut, cre] (ORCID: <https://orcid.org/0000-0002-8788-7980>), web-platform-tests contributors [cph] (bundled URL test corpus, BSD-3-Clause; see inst/COPYRIGHTS)

Maintainer Bart Turczynski <bartek@turczynski.pl>

Repository CRAN

Date/Publication 2026-09-21 21:40:02 UTC

Contents

`addr_address_space` *The bundled IANA address-space registries*

Description

Returns the IANA IPv4 and IPv6 Address Space registries as one data frame, exactly as vendored. This is the **fallback** layer: it answers what a range is for and who holds it, for every address, including the ranges the special-purpose registries never mention.

Usage

```
addr_address_space()
```

Value

A data frame of 276 rows: 256 IPv4 and 20 IPv6.

Why a second pair of registries

Each of these two registries is an **exact partition** of its address space – 256 IPv4 `/8`s and 20 IPv6 blocks that tile `::/0` with no gap and no overlap, asserted at build time. Together with `addr_registry()` that makes classification total: every address matches some row, so "globally reachable ordinary unicast" becomes a statement backed by a registry row rather than an inference from an absence.

Classifying from the special-purpose registries alone is a known CVE-producing pattern, and the clearest case is **multicast**: neither special-purpose registry contains `224.0.0.0/4` or `ff00::/8` at all. A table derived from them therefore has no multicast handling, which is how `ssrfcheck` shipped CVE-2025-8267.

Precedence, stated by the source

The special-purpose registries **outrank** these. That is not raddr's judgment: the address-space registries themselves carry "For authoritative registration, see [Special-Purpose Address Space]".

Because these two are exact partitions, *every* special-purpose block falls inside one of their rows, so precedence is not an edge case – it decides every lookup that matches both layers. Five prefixes appear in both pairs **identically** (`0.0.0.0/8`, `10.0.0.0/8`, `127.0.0.0/8`, `fc00::/7`, `fe80::/10`); for those the two layers agree and only the columns differ.

No policy columns, and none invented

The five IANA policy logicals do not exist in these registries, so they are absent here rather than filled in. The special-purpose registry answers policy; the address-space registry answers identity; neither invents the other's answer.

What these carry instead is `status` (IPv4: `ALLOCATED`, `LEGACY` or `RESERVED`; `NA` for IPv6, which has no such column), `date` (IPv4 only) and `notes` (IPv6 only – free prose, and the only record that `200::/7` and `fec0::/10` are deprecated).

`rfc` is `NA` for **every** IPv4 row, because that registry has no reference column. The RFCs behind its rows live in numeric footnotes whose text is on the registry page and not in the CSV, so raddr reports the marker in `footnotes` and does not transcribe the citation.

See Also

`addr_registry()` for the authoritative special-purpose layer, and `addr_address_space_version()` for this snapshot's own provenance.

Examples

```
space <- addr_address_space()
nrow(space)

# The multicast ranges that appear in no special-purpose registry
space[space$name == "Multicast", c("block", "status")]

# The IPv6 side is 20 rows that tile the whole space
space[space$space == "v6", c("block", "name")]
```

---

addr_address_space_version

Provenance of the bundled address-space snapshot

Description

Reports the date IANA itself records having last changed the vendored address-space registries. This is `addr_registry_version()`'s counterpart for `addr_address_space()`, and follows the same rules: IANA's page-level `Last Updated` field, the older of the two halves, and `NA` rather than a guess whenever the date cannot be read.

Usage

```
addr_address_space_version()
```

Value

A length-1 `character` `"YYYY-MM-DD"` date, or `NA_character_` when either half of the snapshot is undated.

Why this is stamped separately

The two pairs are different files from different registries, and one date across both would make each half assert something about a table it says nothing about – the same reason the transition overlay carries its own stamp (`addr_transition_version()`).

The separation is not theoretical, and this is the pair that proves it. These two registries carry editorial dates two weeks apart, `2025-10-10` and `2025-10-23`, so the "older of the two halves" rule does real work here while it is a no-op for the special-purpose pair.

This pair is why the stamp is editorial

raddr used to stamp from the `Last-Modified` header the CSVs are served with. For the special-purpose pair that header happens to match IANA's editorial date. For these two it does not: they are served with `2025-10-09` and `2025-10-11` against editorial dates of `2025-10-10` and `2025-10-23`, so the header approach reported this snapshot as a day older than IANA says it is, for reasons that have nothing to do with the data. Vendoring these two turned that from a caveat into a wrong number, and the stamp now comes from the registry page instead.

See Also

addr_address_space() for the data itself, and addr_registry_snapshot() for which bytes are installed.

Examples

```
addr_address_space_version()
```

---

addr_category *Read single fields of a classification*

Description

Each takes either a raddr_address, which it classifies, or a raddr_class that addr_classify() already produced.

Usage

```
addr_category(x)

addr_embedded_kind(x)

addr_embeddings(x)
```

Arguments

x	A raddr_address or raddr_class vector.

Value

addr_category() and addr_embedded_kind() return factors; addr_embeddings() returns a list_of<raddr_embedding> All are the same length as x.

category describes; it does not decide

addr_category() returns raddr's one-word vocabulary, and a policy layer must not enumerate it. Label vocabularies drift – ipaddr.js renamed deprecated to deprecatedOrchid – and a consumer that denies by named list turns every newly added level into a bypass. Policy belongs on the five IANA columns, the classify codes and the embeddings, all of which are three-valued and registry- or RFC-sourced (P8). See addr_category_map().

Embeddings are plural, and Teredo is why

`addr_embeddings()` always returns the typed list, never a single address. RFC 4380 section 4 puts two IPv4 addresses in a Teredo address – a server in the clear and a bitwise-complemented client – and section 5.2.6 makes the server a destination a host actually sends to, so neither is metadata for the other. There is deliberately no scalar accessor: reducing the pair to one address *is* the Teredo decision, and it is the consumer's to make.

Each row carries the `category` of the **extracted** address, not of the one it came out of. `addr_classify(addr_pton("::ffff:` therefore reports an IPv6 address in `::ffff:0:0/96` *and* an embedded `127.0.0.1` that is `loopback`, with neither fact collapsed into the other.

`embedded_kind` is affirmative only

`NA` from `addr_embedded_kind()` means no mechanism prefix matched. It does **not** mean the address is not NAT64: RFC 6052 permits a network-specific prefix at six lengths, and a prefix table cannot see one. A caller who knows their operator's prefix can read the address under it with `addr_nat64_embeddings()`, which is opt-in precisely because the prefix cannot come from the address.

See Also

`addr_classify()` for the whole record.

Examples

```
addr_category(addr_pton(c("127.0.0.1", "224.0.0.1", "8.8.8.8", "4000::1")))

# The mechanism is a separate fact from the category: this block is
# `protocol` AND `nat64_wk` AND globally reachable, all at once.
a <- addr_pton("64:ff9b::a9fe:a9fe")
addr_category(a)
addr_embedded_kind(a)

# `::` and `::1` are not IPv4-compatible addresses carrying an embedded
# 0.0.0.0 or 0.0.0.1, and are not reported as though they were
addr_embedded_kind(addr_pton(c("::", "::1", "::2")))
```

`addr_category_map` *raddr's block to category map*

Description

Returns raddr's own mapping from registry block to `category` level, as a data frame of `block` and `category`. This is **not** IANA data: it is raddr's vocabulary, hand-authored and stamped separately, so that neither stamp is evidence about the other.

Usage

```
addr_category_map()
```

Value

A data frame of 322 rows, with columns `block` and `category`. Fewer rows than the 327 vendored registry rows, because five blocks appear in both vendored pairs and are mapped once.

Why this is not a column of `addr_registry()`

`addr_registry()`'s promise is the IANA data *exactly as vendored*. A judgment column sitting beside the five IANA policy logicals would blur the one distinction this package exists to keep: what a registry says, versus what raddr concluded. P9 permits the map because it forbids hand-transcribing an upstream fact when an authoritative file exists, and `category` has no upstream value to preserve.

Descriptive, not a policy input

`category` describes; it does not decide. Do **not** build a deny-list of level names on it. `ipaddr.js` demonstrates both failure modes: label names drift across versions (`deprecated` became `deprecatedOrchid`), and consumers deny by named list, so a newly added label becomes a bypass.

raddr's answer is not a smaller vocabulary – that only postpones the bypass. Policy belongs on the five IANA columns, the classify codes, and the embeddings, all of which are three-valued and registry- or RFC-sourced. There is deliberately no "everything that is not `global`" helper: it would be wrong on `64:ff9b::a00:1`, a block IANA marks `Globally Reachable = True` that embeds `10.0.0.1`.

Keyed on the block, never on the name

A registry `Name` is a mutable display string – "DS-Lite [RFC6333]" became "IPv4 Service Continuity Prefix [RFC7335]" with no change of prefix – and a *new* row reusing an existing name would classify itself with nobody reading it. The block is the row's identity.

See Also

`addr_registry()` and `addr_address_space()` for the data being mapped, and `addr_category_version()` for this map's own stamp.

Examples

```
map <- addr_category_map()
table(map$category)

# The space IANA holds and has neither purposed nor delegated
map[map$category == "unallocated", ]
```

addr_category_version *Provenance of raddr's category map*

Description

Reports the version stamp of the hand-authored block to category map. This is raddr's own judgment and changes on raddr's schedule, so it is stamped separately from the vendored IANA snapshots (`addr_registry_version()`, `addr_address_space_version()`): one date across both would make each assert something about a table it says nothing about.

Usage

```
addr_category_version()
```

Details

There is deliberately no `addr_category_outdated()`. The map does not go stale on a clock – it goes stale when a registry row appears that it has no entry for, and that fails the build rather than aging quietly.

Value

A length-1 `character` `"YYYY-MM-DD"` date.

See Also

`addr_category_map()` for the map itself.

Examples

```
addr_category_version()
```

addr_classify *Classify addresses against the IANA registries*

Description

Reports what the IANA registries say about each address: the block that matched, its name and RFC, raddr's own one-word `category`, all five IANA policy columns, the transition mechanism the address belongs to, and the snapshot the answer came from.

Usage

```
addr_classify(x)
```

Arguments

x	A raddr_address vector.

Details

addr_classify() returns facts. It returns no verdict, no risk score and no allow-or-deny decision, and it ships no "everything that is not global" helper. Policy belongs to the consumer (P8).

Value

A raddr_class vector, one element per address. Every field is NA for a missing address.

Two registries, and which one answered

Classification is **total**: every non-missing address matches some row. That takes two layers, because the IANA special-purpose registries do not cover the whole address space – 224.0.0.0/4 and ff00::/8 appear in neither, which is how ssrfcheck shipped CVE-2025-8267.

addr_registry(), **the special-purpose pair** Answers **policy**: the five logical columns. Matched first, and it outranks the other layer on IANA's own instruction – the address-space registries carry "For authoritative registration, see [Special-Purpose Address Space]".

addr_address_space(), **the address-space pair** Answers **identity**: what a range is for and who holds it. Each half is an exact partition, which is what makes the lookup total.

registry says which one answered, and registry_version carries that layer's own stamp (P7). The distinction is load-bearing, because it is what keeps the two meanings of NA apart in the five policy columns:

registry = "special_purpose" NA is IANA's own N/A – a policy it specifically declined to state. Reading it as FALSE asserts something the registry withheld.

registry = "address_space" NA means the question was never asked: that registry has no policy columns at all. raddr leaves them absent rather than inventing them.

Every NA says why it is NA

Where raddr knows that two missing values mean different things, it reports what distinguishes them rather than leaving both blank. Four special-purpose blocks have a missing policy value, for three different reasons, and each reason is a column:

Deprecated: 192.88.99.0/24, 2001:10::/28 all five columns are NA and termination_date is set. IANA gives a withdrawn block no policy at all.

Withheld: 2001::/32 **(Teredo), footnote** [2] RFC 4380 section 5 makes relay advertisement voluntary and per-deployment, so no bits in the address answer reachability.

Withheld: 2002::/16 **(6to4), footnote** [3] a different reason entirely – reachability follows the *embedded* IPv4 address, which a prefix table cannot express.

`footnotes` does the same job for `rfc`, which is `NA` for every IPv4 address-space row. 42 of those 256 rows carry a footnote marker, meaning the citation exists and its text is on the registry page rather than in the CSV; the other 214 carry none. raddr reports that a caveat exists rather than inventing its wording, and `""` means the row carried no marker at all.

Footnote numbering is **per registry**, so a marker is only meaningful alongside `registry` and the address family.

What the fields are

`block`, `name`, `rfc`, `footnotes` The matched row, as vendored. `rfc` is `NA` for every IPv4 address-space row, because that registry has no reference column; `footnotes` is what says whether a citation nonetheless exists.

`category` raddr's own one-word vocabulary, 19 levels. It is **descriptive, not a policy input** – see `addr_category_map()`, and do not build a deny-list of level names on it.

`globally_reachable`, `forwardable`, `source`, `destination`, `reserved_by_protocol` IANA's five policy columns, per row and never collapsed. `globally_reachable` **is** the column, not a derived `is_global`.

`termination_date` Set on a deprecated block, and the reason its policy columns are empty. `NA` everywhere else, including for every address-space row.

`embedded_kind` The transition mechanism, when a mechanism prefix matched. `NA` means no prefix matched – it never means "not NAT64". A caller-supplied RFC 6052 network-specific prefix is invisible to a prefix table, so raddr states a NAT64 kind only affirmatively.

`embeddings` Zero or more extracted inner addresses, one `raddr_embedding` per element, each carrying the extracted address and **its own** `category`. Plural because a Teredo address carries **two** IPv4 addresses – a server in the clear and a bitwise-complemented client – and raddr does not choose between them.

`codes` Classify-layer reason codes, from the same vocabulary as `addr_codes_registry()` and graded by that registry's `strength` column. See below.

The codes are graded, and all of them are reported

`codes` carries what the RFCs say about an address that the registry row alone does not. Each one is graded in `addr_codes_registry()` by the force of the rule it reports, because reporting only the MUST rules would collapse a spectrum into a binary:

`nat64_wk_embedded_not_global` (`must`) `64:ff9b::/96` carries a non-global embedded IPv4 address, which RFC 6052 section 3.1 says translators MUST drop. The rule binds the well-known prefix **alone** – never a network-specific prefix, and RFC 8215 section 5 says in terms that it does not reach `64:ff9b:1::/48`.

`sixtofour_embedded_not_global` (`must`) the 6to4 `V4ADDR` is not a global unicast address, so RFC 3056 section 9 requires both encapsulators and decapsulators to discard the traffic silently.

`teredo_client_not_global` (`must`) a global Teredo address embeds a non-global client IPv4 (RFC 4380 section 4). The **server** is not graded: the RFC states no equivalent requirement on it.

`link_local_outside_fe80_64` (`must`) `febf::1` matches the `fe80::/10` registry row but is not a link-local address – RFC 4291 section 2.5.6 fixes the next 54 bits to zero. Both CPython's `ipaddress` and R's `ipaddress` report it as link-local with nothing attached to say otherwise.

`link_local_reserved_range` (`must`) `169.254.0.0/24` and `169.254.255.0/24` MUST NOT be selected by IPv4 autoconfiguration (RFC 3927 section 2.1). Neither has a registry row of its own.

`ipv4_compatible_low_tail` (`may`) the deprecated `::a.b.c.d` tail lands in `0.0.0.0/8`, so it is not a host address.

`nat64_local_layout_unspecified` (`unspecified`) RFC 8215 section 5 leaves the syntax under `64:ff9b:1::/48` deliberately undefined, so the RFC 6052 geometry raddr reads there is contested rather than implied.

`ula_l_bit_unset` (`unspecified`) `fc00::/8` is the L = 0 half of the ULA prefix, for which RFC 4193 section 3.1 defines nothing at all. Only `fd00::/8` is a specified ULA.

The first three are stated about the **embedded** address rather than the outer one, and all three are worded slightly differently – "non-global", "not in the format of a global unicast address", "a global scope unicast IPv4 address". raddr answers them with one predicate, and answers it affirmatively from both registry layers: an extracted address is global when IANA records `globally_reachable = TRUE` for it, or when it lies in space delegated to an RIR. Neither layer settles it alone.

A rule fires only where raddr actually extracted an address. A caller-supplied RFC 6052 network-specific prefix is invisible to a prefix table, so no code is emitted for one – silence here is not a clean bill of health, for the same reason `embedded_kind = NA` is not.

A string may never be classified

`addr_classify()` takes a parsed address and nothing else (P1). It also declines a `raddr_parse`, which holds four readings that may be four different addresses: choosing one is a decision, and raddr makes it by function name rather than silently. Pick a reading with `addr_reading()`, or parse with `addr_strict()`, `addr_whatwg()`, `addr_pton()` or `addr_aton()`.

See Also

`addr_category()` and `addr_embeddings()` for single fields, `addr_registry()` and `addr_address_space()` for the data behind the answer.

Examples

```
addr_classify(addr_pton(c("127.0.0.1", "8.8.8.8", "224.0.0.1", "4000::1")))

# All four facts about the NAT64 well-known prefix, none collapsed
cl <- addr_classify(addr_pton("64:ff9b::a9fe:a9fe"))
as.data.frame(cl)[c("block", "category", "globally_reachable")]

# The carve-out that forces longest-prefix matching
as.data.frame(addr_classify(addr_pton(c("192.0.0.9", "192.0.0.100"))))
```

`addr_codes_registry` *The reason-code registry*

Description

Every reason code raddr can attach to a reading, with its layer, its provenance and the version it was introduced in. `addr_codes()` returns codes from this vocabulary; this is where you look one up.

Usage

```
addr_codes_registry()
```

Details

The registry is the vocabulary's single definition. raddr derives the set of valid codes from it rather than keeping a second list, and a test asserts that every code has at least one input in the corpus that produces it, so a code that nothing can emit fails the build.

Value

A data frame with one row per code and the columns `code`, `layer`, `rfc`, `summary`, `strength` and `since`.

A versioned vocabulary, not an alias

The codes are meant to be read by other packages, so adding one is an addition to raddr's API and removing one is a breaking change – which is what the `since` column records.

They are **not** a vocabulary another package echoes verbatim. `ssrfr` owns its own reason codes and its own result model, and the relationship between the two vocabularies is many-to-one and conditional rather than an alias: raddr states facts, a policy layer interprets them into a refusal reason. A raddr code may travel in a detailed result as evidence without being that package's public reason. This is the same separation drawn between raddr's `category` and a policy verdict, and for the same purpose – a policy layer must not enumerate a descriptive classifier's output as its deny list.

Layers

`parse` Why a dialect declined to read a literal as an address. Reported by `addr_codes()`.

`classify` Facts about a parsed address noted during classification. Reported in the `codes` field of `addr_classify()`.

Why every rule is reported, not only the MUSTs

`strength` records how much force the rule a code reports actually carries: `"must"`, `"should"`, `"may"` or `"unspecified"`. Reporting only the MUST rules would collapse a spectrum into a binary, which is the move raddr exists to refuse – so a rule stated in weaker language is still reported, and the grade is what says not to act on it as though it were a MUST.

It is NA for every parse code, and that is the honest value rather than a filler: those codes describe what a parser **did** with a literal, not what a specification mandates about an address.

The grade follows the rule's substance, not the presence of an RFC 2119 keyword, because the sources do not agree about keywords: RFC 4291 and RFC 8215 invoke RFC 2119 nowhere and state their rules in lowercase or as a format diagram, while RFC 3056, 3927, 4193, 4380 and 6052 all invoke it. Each summary says which case it is, so the grading can be checked rather than taken on trust.

"should" has no member yet. The level is kept anyway, so that a consumer does not read must and may as the whole scale.

Examples

```
registry <- addr_codes_registry()
registry$code

registry[registry$code == "out_of_range", ]

# The classify layer, graded by normative force
classify <- registry[registry$layer == "classify", ]
classify[c("code", "rfc", "strength")]
```

---

addr_family *Read the family of an address*

Description

Read the family of an address

Usage

```
addr_family(x)
```

Arguments

x	A raddr_address vector.

Value

A factor with levels "v4", "v6" and "v6_4in6", NA for missing addresses.

Examples

```
addr_family(raddr_address(0L, 0L, 0L, 1L, "v4"))
```

`addr_format` *Render addresses as text*

Description

`addr_format()` renders an address in its **canonical** form: RFC 5952 for IPv6, dotted-quad for IPv4. Every address has exactly one canonical spelling, so two addresses that compare equal always format identically. This is what `format()` and `as.character()` emit.

Usage

```
addr_format(x)

addr_expand(x)
```

Arguments

`x`	A `raddr_address` vector.

Details

`addr_expand()` renders the **fully expanded** form instead: eight four-digit hextets, nothing compressed, nothing abbreviated. Reach for it when addresses have to line up in a column, sort as text, or be matched by a prefix – none of which the canonical form supports, because it is variable-width by design.

Value

A character vector the same length as `x`, `NA` for missing addresses.

What RFC 5952 asks for

4.1 Leading zeros in a field are suppressed: `2001:0db8` is `2001:db8`, and an all-zero field is `0`.

4.2.1 The `::` is used wherever it can be.

4.2.2 But never for a *single* zero field – `2001:db8:0:1::1`, not `2001:db8::1::1`, and `2001:db8:0:1:1:1:1:1` keeps its `0`.

4.2.3 The longest run of zero fields is the one compressed, and the **first** of two equally long runs wins.

4.3 Hex digits are lowercase.

5 An address with an embedded IPv4 address is rendered in the mixed form: `::ffff:192.0.2.1`.

RFC 5952 publishes no test vectors; `tests/testthat/test-format.R` carries the ones raddr authored against its text, section by section.

The 4-in-6 form

The mixed form is emitted for the `v6_4in6` family, which is decided by the bits rather than by the spelling (see `raddr_address()`). That is what makes `parse(format(x)) == x` hold for `::ffff:192.0.2.1`: it round-trips back into its own family rather than collapsing onto the bare IPv4 address.

The zone

A zone ID is appended as `%zone` by both renderers, and neither reads it back into the address bits. A missing address renders as `NA`.

Examples

```
a <- addr_strict(c("2001:db8::1", "::ffff:192.0.2.1", "192.0.2.1"))
addr_format(a)
addr_expand(a)

# The canonical form round-trips
addr_strict(addr_format(a)) == a
```

---

addr_global_reachability

Whether an address is in globally reachable space

Description

The two-layer positive fact raddr already uses internally to decide the antecedent of RFC 6052 section 3.1, RFC 3056 section 9 and RFC 4380 section 4. It is a **fact, not a verdict**: it reports what the two vendored registry layers say, and says nothing about whether a caller should permit the address.

Usage

```
addr_global_reachability(x)
```

Arguments

`x`	A `raddr_address`, `raddr_class` or `raddr_embedding` vector. An embedding is classified by its extracted address, so the answer is the same one the classify layer used when grading that embedding.

Value

A logical vector the same length as `x`: `TRUE`, `FALSE` or `NA`.

The two layers, and why both

Neither layer answers alone, and each fixes what the other gets wrong:

special-purpose IANA's own `globally_reachable` column, unmodified. Without it the five blocks IANA marks globally reachable – PCP and TURN anycast, AS112 twice, AMT – read as non-global.

address space that layer has no policy column at all, and the question it does answer is whether the space is delegated to an RIR, which is `category = "global"`. Without it `8.8.8.8` reads as non-global and `224.0.0.0/4` reads as nothing – CVE-2025-8267's shape.

This is **not** the `category` deny-list `addr_category()` warns against. It reads one *positive* level, only in the layer that has no other column, and a level added later changes no answer that layer gives today. Reaching this fact through this function rather than rebuilding it from `category` is the whole reason it is exported.

`NA` is a third answer, not a missing one

`NA` means the registries leave the question open, and it must not be read as `FALSE`. Today exactly one block is in that tier: `192.88.99.0/24`, which IANA withdrew and gave no policy at all, together with its 6to4 image `2002:c058:6301::`. Asserting a MUST-drop there would be reading IANA's `N/A` as a `FALSE` one level down.

Handle it explicitly. R propagates `NA` rather than resolving it: `any()` returns `NA` instead of `FALSE`, `which()` drops the element entirely, and `if` raises an error on it. A policy layer must branch on all three values rather than let the third fall through to either side. raddr reports it; deciding what it costs is the caller's.

See Also

`addr_classify()` for the whole record, including the `globally_reachable` column this reads. `addr_category()` for why the descriptive vocabulary is not a policy input.

Examples

```
# The address-space layer answers for an ordinary host, the special-purpose
# layer for a carve-out, and neither for the withdrawn anycast prefix.
addr_global_reachability(addr_pton(c("8.8.8.8", "192.0.0.9", "192.88.99.1")))

# Both families, one rule
addr_global_reachability(addr_pton(c("2001:4860:4860::8888", "fe80::1")))

# The outer address and what it embeds are separate questions: this block is
# globally reachable and the IPv4 inside it is not
a <- addr_pton("64:ff9b::a9fe:a9fe")
addr_global_reachability(a)
addr_global_reachability(addr_embeddings(a)[[1]])
```

addr_nat64_embeddings *Read the IPv4 address a caller-supplied NAT64 prefix embeds*

Description

addr_classify() names NAT64 only from the two prefixes that are written down – the RFC 6052 Well-Known Prefix 64:ff9b::/96 and the RFC 8215 local-use prefix 64:ff9b:1::/48. A Network-Specific Prefix is invisible to a prefix table, so raddr states a NAT64 kind only affirmatively and addr_embedded_kind() returns NA under an operator's own prefix. This function is the opt-in for a caller who knows the prefix and wants the address read under it.

Usage

```
addr_nat64_embeddings(x, prefix)
```

Arguments

x	A raddr_address vector.
prefix	A single CIDR block, as a string: an IPv6 prefix at one of the six lengths RFC 6052 section 2.2 permits (/32, /40, /48, /56, /64, /96). One call reads one prefix.

Value

A list_of<raddr_embedding> the same length as x, shaped exactly as addr_embeddings() is: one row for each address that lies under prefix, and zero rows for each that does not.

The reading is yours, and it is labeled that way

Nothing in an IPv6 address says it is NAT64 under some prefix, so supplying one is an assertion, not a discovery – and a wrong assertion produces a plausible wrong IPv4 address rather than an error. Every row this returns therefore carries kind = "nat64_nsp", a level addr_classify() can never emit, so a configured reading stays distinguishable from one raddr reached from the address alone.

addr_classify() is unchanged by this call. There is no way to register a prefix so that classification starts seeing it: that would make the same address classify differently depending on state held elsewhere.

The u-byte split is why this is not a one-liner

RFC 6052 section 2.2 reserves bits 64-71, so at /40, /48 and /56 the embedded address is **not contiguous** – it resumes after the reserved octet, and reading 32 bits from the prefix boundary yields a wrong address that looks right. Under a /48, 192.0.2.33 read that way comes back as 192.0.0.2. This function reads the segments from the same geometry table the fixed prefixes use, published as addr_transition_registry("embeddings").

No RFC 2119 rule attaches to the result

RFC 6052 section 3.1's MUST-drop is written about the Well-Known Prefix alone: "translators MUST NOT translate packets in which an address is composed of the Well-Known Prefix and a non-global IPv4 address". It states no equivalent requirement for a Network-Specific Prefix, and RFC 8215 section 5 says in terms that it does not reach the local-use prefix either. So no classify code is emitted here even when the embedded address is not global – see `addr_global_reachability()` for the fact, which is a fact and not a permission.

This function emits **no** codes at all, including `nat64_u_byte_nonzero`: a code is a property of a classification, and a caller-supplied prefix produces a reading rather than a classification. RFC 6052 section 2.2's reserved octet sits at bits 64-71 whatever the prefix length, so a caller who wants that check under their own prefix can make it directly. `addr_classify()` reports it for the prefixes raddr names.

See Also

`addr_embeddings()` for the mechanisms raddr names on its own, and `addr_transition_registry()` for the geometry this reads.

Examples

```
a <- addr_pton(c("2001:db8:122:344::c000:221", "2001:db8::1", "8.8.8.8"))

# Under the operator's own /96: one reading, two addresses it does not cover
addr_nat64_embeddings(a, "2001:db8:122:344::/96")

# Classification still says nothing about it, and that is not a disagreement
addr_embedded_kind(a)

# RFC 6052 section 2.4's own worked example, at the /48 where the reserved
# u-byte splits the octets: both of these embed 192.0.2.33
under <- addr_pton(c("2001:db8:122:c000:2:2100::", "2001:db8:c000:221::"))
addr_nat64_embeddings(under[1], "2001:db8:122::/48")
addr_nat64_embeddings(under[2], "2001:db8::/32")
```

---

addr_parse *Read an address literal under every dialect at once*

Description

`addr_parse()` is raddr's primary answer. It reads each literal under all four dialect primitives and reports every reading, the outcome of each, and the reason codes behind each rejection – rather than picking one reading and discarding the rest.

Usage

```
addr_parse(x)
```

Arguments

x	A character vector of address literals.

Value

A raddr_parse vector with one element per input, carrying input, the four per-dialect readings, the per-dialect outcome and codes, and the derived status.

Why the outcome is per-dialect

Because a single one cannot be written down honestly. "4294967296" is accepted by aton as 0.0.0.0, rejected by whatwg as out of range, and rejected by strict as not a dotted quad – simultaneously, on one machine. A record with one status has to choose which of those three to report, and every choice is a lie about the other two.

The derived status

addr_status() does collapse the four outcomes to one value, as a **convenience and never as truth**:

ok Every primitive with a say accepts, and they yield the same address.

divergent They are not unanimous – on the value, or on whether to accept at all. "4294967296" is divergent.

not_an_address No primitive treats the input as an attempt at an address. "example.com" is not_an_address.

malformed At least one primitive treats it as an attempt, and none accepts.

"With a say" is doing work in the first of those. inet_aton is AF_INET by signature, so it has no reading of "::1" to withhold and its silence there is not dissent – otherwise every IPv6 address on earth would be divergent. A dialect that *has* the grammar and still declines is a different matter: "1.2.3.4 junk" is divergent, because aton finds an address in it that the other three do not, which is the class where curl reaches a host a browser will not dial.

Whenever the answer matters, read the per-dialect outcome instead.

Printing

The print method is quiet when the dialects agree and loud when they do not: a vector of ordinary addresses prints as a column of addresses, and a divergent row is expanded underneath to show what each dialect made of it. There is no mode to select and nothing to force – see dialects for why the dialect is a function name rather than an argument.

See Also

addr_reading() to pull one dialect's reading back out, addr_codes_registry() for the reason-code vocabulary, and dialects for the single-dialect shortcuts.

Examples

```
# One string, four readings
addr_parse("0177.0.0.1")

# Accepted by one dialect, rejected by three, for two different reasons
p <- addr_parse("4294967296")
addr_status(p)
addr_codes(p)

# Agreement prints quietly
addr_parse(c("127.0.0.1", "::1"))
```

`addr_registry` *The bundled IANA special-purpose address registries*

Description

Returns the two IANA special-purpose address registries as one data frame, exactly as vendored. This is the table `addr_classify()` will match against. (Not a link: `addr_classify()` arrives with the classification layer.)

Usage

```
addr_registry()
```

Value

A data frame of 51 rows. See the sections above for the columns.

All five policy columns, never collapsed

IANA records five independent properties per block, and raddr surfaces all five rather than reducing them to a single "is it private" flag. `globally_reachable` **is** IANA's `Globally Reachable` column, per row, with an RFC citation – not a value raddr derives.

Each is `logical`, and `NA` is a real answer with a real meaning: IANA declined to give one. Two cases produce it.

The block is deprecated `192.88.99.0/24` and `2001:10::/28` carry a `termination_date` and no policy values at all.

The answer depends on something the table cannot express Teredo (`2001::/32`) and 6to4 (`2002::/16`) are both recorded as `N/A` for `globally_reachable` – but for **two different reasons**, carrying two different IANA footnotes. Do not merge them.

The two `N/A` reasons, kept apart because a reader following either footnote must find the reason raddr states:

6to4 (`2002::/16`)**, footnote** `[3]`**, RFC 3056** reachability follows the *embedded* IPv4 address, which a prefix table cannot express.

Teredo (`2001::/32`)**, footnote** `[2]`**, RFC 4380 section 5** a different thing entirely: relay advertisement is **voluntary and per-deployment**, so whether any given Teredo address is reachable depends on what its operator chose to advertise – not on the embedded client address.

Reading either as `FALSE` would assert a policy IANA specifically withheld, so raddr keeps them `NA`.

Blocks, not rows

One row here is one prefix. That is not always one row upstream: the CSV names two prefixes in a single record (`192.0.0.170/32, 192.0.0.171/32`), and three records wrap across lines because they cite more than one RFC. The vendored 25 + 25 records become **51 blocks**.

Longest prefix, not first match

The registry contains deliberate carve-outs – `192.0.0.9/32` and `192.0.0.10/32` are globally reachable inside a `192.0.0.0/24` that is not. Any lookup over this table must be longest-prefix-match.

Footnotes

`footnotes` records which upstream footnote markers a row carried, as a space-separated string, and is `""` when it carried none. The footnote **text** is not in the CSV – it lives on the registry page – so raddr reports that a caveat exists rather than inventing its wording.

See Also

`addr_registry_version()` for the snapshot's provenance.

Examples

```
reg <- addr_registry()
nrow(reg)

# The carve-out that forces longest-prefix matching
reg[startsWith(reg$block, "192.0.0."), c("block", "globally_reachable")]

# The blocks whose reachability IANA declined to state
reg[is.na(reg$globally_reachable), c("block", "name", "termination_date")]
```

---

addr_registry_snapshot

Content-addressed identity of the bundled registry snapshot

Description

Returns one string identifying exactly which vendored IANA bytes are installed: a `"sha256:..."` digest over all four registry files. This is the value to quote in a bug report, because it pins the data a result came from without depending on the package version.

Usage

```
addr_registry_snapshot()
```

Value

A length-1 `character` of the form `"sha256:"` followed by 64 hex digits, or `NA_character_` if the installed snapshot records no id.

What it answers, and what it does not

It answers **which bytes**. Two installations reporting the same id have the same four files, byte for byte.

It does **not** answer which of two snapshots is newer. A hash has no order. Currency is what `addr_registry_version()` and `addr_address_space_version()` report, and those two are deliberately separate because they make claims about separate tables.

One id covers all four files for that same reason inverted. A *date* spanning both pairs would make each half assert currency for a table it says nothing about; a content hash asserts only what is installed, which is a property of the payload as a whole.

How it is computed

A sha256 over a canonical manifest: one `"<key> sha256:<hex>"` line per source, each terminated by a newline, in the fixed order `v4`, `v6`, `v4_space`, `v6_space`, hashed as UTF-8 bytes.

The order and the spelling are part of the definition rather than formatting, which is what makes the id reproducible outside R – the manifest is a byte string anything can build from the installed CSVs and hash. The manifest is also stored beside the id, and `data-raw/build-registry.R --check` verifies both steps: that the manifest still describes the files on disk, and that the id still follows from the manifest. Being content rather than dates, both belong in that guard, which never compares dates.

See Also

`addr_registry_version()` and `addr_address_space_version()` for currency rather than identity.

Examples

```
addr_registry_snapshot()
```

addr_registry_version *Provenance of the bundled special-purpose registry snapshot*

Description

addr_registry_version() reports the date IANA itself records having last changed the vendored **special-purpose** registries. addr_registry_outdated() says whether that is longer ago than max_age days.

Usage

```
addr_registry_version()

addr_registry_outdated(max_age = 365)
```

Arguments

max_age	Maximum acceptable age in days. Default 365.

Details

These two answer for addr_registry() only. The address-space pair is vendored from different files and stamped separately; see addr_address_space_version(). For *which bytes* are installed rather than how current they are, see addr_registry_snapshot().

Value

addr_registry_version() returns a length-1 character "YYYY-MM-DD" date, or NA_character_ when the snapshot is undated. addr_registry_outdated() returns a length-1 logical, TRUE when the snapshot is older than max_age days **or** undated.

What the stamp is, and is not

The IANA CSVs carry no version field. The stamp is the page-level Last Updated field from IANA's own registry page – its editorial date – read at build time and stored as ISO text.

It is deliberately **not** the Last-Modified header the CSV is served with, which earlier versions of raddr used. That header is a site **deploy** timestamp: unrelated CSVs across different IANA registries are served with the same timestamp to the second, and at least one IANA registry has been edited months after the Last-Modified its own export still carries. For this pair the two happen to agree; for the address-space pair they do not, which is what settled the question. The served header is still recorded in the package's internal metadata, because it is a fact about the fetch – it is simply not an answer to "when did IANA last change this".

Content identity is tracked separately and exactly, by a sha256 per file and by the single snapshot id addr_registry_snapshot() returns. data-raw/build-registry.R --check compares content and never dates, so a stamp that drifts for deploy reasons cannot make the staleness guard pass or fail.

Unknown is not fresh

When either half has no date, the snapshot has no date: `addr_registry_version()` returns `NA` and `addr_registry_outdated()` returns `TRUE`.

That asymmetry is deliberate. A snapshot of unknown age is one you have no evidence about, and treating no evidence as evidence of freshness is the one failure mode a staleness check exists to prevent.

That extends to the source of the date. Scraping a field out of upstream markup can fail in several ways – the field renamed, duplicated, emptied, or reformatted – and every one of them yields `NA` here rather than a guess. That is what makes reading the editorial date acceptable at all: the mode it fails in is the safe one.

The stamp is also the **older** of the two halves, because a snapshot is only as current as its stalest part. Both special-purpose registries currently record the same editorial date, so the rule does no work for this pair; it does for `addr_address_space_version()`.

No refresh

There is no `addr_registry_refresh()`. raddr performs no network access at all: the registries change on a multi-year cadence and the whole vendored payload is 29 KB, so shipping it outright is a cleaner claim than network code that defaults to off. A stale snapshot is fixed by upgrading the package.

See Also

`addr_registry()` for the data itself.

Examples

```
addr_registry_version()
addr_registry_outdated()

# An undated or overly old snapshot is reported, never assumed fresh
addr_registry_outdated(max_age = 0)
```

`addr_reverse_pointer` *Reverse DNS pointer name*

Description

The name that holds an address's `PTR` record: `in-addr.arpa` for IPv4 (RFC 1035 §3.5), `ip6.arpa` for IPv6 (RFC 3596 §2.5).

Usage

```
addr_reverse_pointer(x)
```

Arguments

x	A raddr_address vector.

Value

A character vector the same length as x, NA for missing addresses.

How the name is built

IPv4 reverses whole **octets** and IPv6 reverses 4-bit **nibbles**, each least significant first, and the two are not interchangeable – reversing an IPv6 address by octet yields a plausible-looking name that points somewhere else. RFC 1035 §3.5 gives the reason for the reversal: it "allows zones to be delegated which are exactly one network of address space".

```
10.2.0.52    ->  52.0.2.10.in-addr.arpa.
2001:db8::1  ->  1.0.0. ... .0.8.b.d.0.1.0.0.2.ip6.arpa.
```

An ip6.arpa name is always 32 labels

No ::, no suppressed leading zeros, no mixed 4-in-6 spelling. ::1 has 32 labels, 31 of them 0. This is the opposite of the text form – see addr_format() – and it is why the name is built from the bits rather than from the rendered address. An in-addr.arpa name is always 4 labels, in decimal, with leading zeros omitted (RFC 1035 §3.5: "leading zeros omitted except in the case of a zero octet which is represented by a single zero").

The trailing dot, and the case

The name is emitted **fully qualified**, with the trailing dot that stands for the root label (RFC 1035 §3.1). 1.2.0.192.in-addr.arpa and 1.2.0.192.in-addr.arpa. denote the same name but are not the same string, so raddr picks the unambiguous one.

Hex labels are lowercase. Comparison in the DNS is case-insensitive (RFC 1035 §3.1: "Name servers and resolvers must compare labels in a case-insensitive manner"), so B.A.9 and b.a.9 are the same name; the lowercase choice follows RFC 5952 §4.3.

The 4-in-6 form gets the mechanical answer

No RFC says whether ::ffff:192.0.2.1 should map into ip6.arpa or into 1.2.0.192.in-addr.arpa. raddr returns the ip6.arpa name, because the address is an IPv6 address and that is the mechanical reading of RFC 3596 §2.5. The *useful* name is often the in-addr.arpa one, because that is where the data actually lives – ask for it by naming the embedded address directly, which is the same choice addr_to_bytes() makes about width.

The zone is not part of the name

A zone ID is dropped, silently and by design. It is strictly local to a node (RFC 4007 §6), so it has no meaning in a DNS name, and there is nowhere in the ip6.arpa grammar to put it. Note that a zoned address is *rendered*, not rejected: Python's IPv6Address.reverse_pointer raises on one, which is a bug in its renderer rather than a rule about zones.

What this function is not

- **Not a resolver.** raddr is offline. RFC 8501 §1.2 notes that pre-populating an IPv6 reverse zone is impractical – "2^80 possible addresses could be configured in a single /48 zone alone" – and §2.1 records `NXDOMAIN` as a legitimate answer. A correct name is not a resolvable name.
- **Not reversible here.** There is no pointer-to-address function, because a pointer name does not have to name an address: `10.in-addr.arpa.` is a /8 (RFC 1035 §3.5's own gateway example), so the general answer is a prefix, and raddr has no prefix type yet.
- **Not** `ip6.int`**.** Deprecated by RFC 3152 §2 and retired by RFC 4159 ("the DNS domain 'ip6.int' should no longer be used"). raddr never emits it.
- **Not a bitstring label.** RFC 2673 and RFC 2874 were reclassified Experimental by RFC 3363, whose §3 concluded that the hexadecimal text form "appears to be capable of expressing all of the delegation schemes that we expect to be used".
- **Not an RFC 2317 name.** Classless `in-addr.arpa` delegation is an operator convention – the block boundary and even the separator character are choices (RFC 2317 §4) – so those names are generate-only and cannot claim to round-trip.

Examples

```
a <- addr_pton(c("10.2.0.52", "2001:db8::1", "::ffff:192.0.2.1"))
addr_reverse_pointer(a)

# RFC 1035 section 3.5's own example
addr_reverse_pointer(addr_pton("10.2.0.52"))

# Always 32 labels for IPv6, however short the text form is
lengths(strsplit(addr_reverse_pointer(addr_pton("::1")), ".", fixed = TRUE))

# The zone is not part of a DNS name
addr_reverse_pointer(addr_pton(c("fe80::1", "fe80::1%eth0")))
```

---

addr_to_bytes *Encode and decode addresses as bytes, hex or binary*

Description

Three symmetric pairs. Each `addr_to_*()` turns addresses into an encoding, and each `*_to_addr()` turns that encoding back into addresses.

Usage

```
addr_to_bytes(x)

addr_to_hex(x)

addr_to_binary(x)
```

```
bytes_to_addr(x)

hex_to_addr(x)

binary_to_addr(x)
```

Arguments

x	For `addr_to_*()`, a `raddr_address` vector. For `bytes_to_addr()`, a list of `raw` vectors of length 4 or 16. For `hex_to_addr()` and `binary_to_addr()`, a character vector.

Value

`addr_to_bytes()` returns a `list_of<raw>`, with `NULL` for a missing address. `addr_to_hex()` and `addr_to_binary()` return character vectors, `NA` for a missing address. The three decoders return a `raddr_address` vector, missing wherever the input could not be decoded – they signal no error and no warning, exactly as the single-dialect parsers in dialects do.

The width carries the family

An IPv4 address encodes to **4** octets, 8 hex digits or 32 bits; an IPv6 address to **16** octets, 32 hex digits or 128 bits. The width is decided by the family and never by the bits, which is what keeps `::ffff:192.0.2.1` apart from `192.0.2.1`: the two share their low 32 bits (RFC 4291 §2.5.5) and the length is the only thing that tells them apart.

The 4-in-6 form therefore encodes to the **full 16 octets**, not to the 4 of the address it embeds. Use `addr_embeddings()` when the embedded address is what you want.

Leading zeros

Every output is fixed width and zero padded. `::1` is 32 hex digits, 31 of them `0`. This is the opposite of RFC 5952 §4.1, which suppresses leading zeros – that rule is about *text* form, and these are not text forms.

Decoding is exact about it: a string of any width other than the two the family fixes decodes to `NA`, and is never padded to the nearest one. Seven hex digits could be an IPv4 address missing a zero or an IPv6 address missing twenty-five, and raddr will not guess.

What survives a round trip, and what does not

`bytes_to_addr(addr_to_bytes(x))` **equals** `x`, and likewise for the other two pairs. Equality is over the 128 bits and the family (see `raddr_address()`), and all three encodings preserve both.

The **zone does not survive**. `fe80::1%eth0` and `fe80::1%eth1` encode to identical octets, and the decoders return an address with no zone at all – RFC 4007 §6 explains why it cannot be recovered, since zone indices are strictly local to a node. The round trip still satisfies `==` because the zone does not participate in equality, but `addr_zone()` on the result is `NA`. Carry it separately if you need it.

A **prefix length** does not survive either, for the simpler reason that an address does not carry one. Four octets are `192.0.2.0`, full stop.

Case, prefixes and grouping on input

Hex output is lowercase, following RFC 5952 §4.3. Uppercase input is accepted, because RFC 3596 §2.5 and RFC 2874 §2.2.1 both print their examples in uppercase.

`hex_to_addr()` accepts an optional `0x` or `0X` prefix and never emits one. No RFC defines a `0x`-prefixed address encoding; it is a presentation convention, so raddr reads it and does not write it.

Whitespace grouping – `c000 0201`, or a binary string spaced per octet – is stripped on input by both string decoders. Nothing else is normalized away.

Examples

```
a <- addr_pton(c("192.0.2.1", "2001:db8::1", "::ffff:192.0.2.1"))

addr_to_hex(a)
addr_to_bytes(a)

# Every pair round-trips
hex_to_addr(addr_to_hex(a)) == a
bytes_to_addr(addr_to_bytes(a)) == a
binary_to_addr(addr_to_binary(a)) == a

# The width is the family: the same low 32 bits, two different encodings
addr_to_hex(addr_pton(c("192.0.2.1", "::ffff:192.0.2.1")))

# Uppercase and a 0x prefix are read; neither is written back
addr_to_hex(hex_to_addr("0xC0000201"))

# A width the family does not fix is not guessed at
hex_to_addr("c000201")
```

---

addr_to_integer *Encode and decode addresses as unsigned integers*

Description

`addr_to_integer()` is the numeric value of an address: the 4 octets of an IPv4 address read big-endian (RFC 4632 §3.1), or the 16 octets of an IPv6 address (RFC 4291 §2). `integer_to_addr()` reads one back.

Usage

```
addr_to_integer(x, output = c("character", "double", "bignum"))

integer_to_addr(x, family)
```

Arguments

x	For addr_to_integer(), a raddr_address vector. For integer_to_addr(), a character vector of decimal digits, a numeric vector, or anything whose as.character() is decimal digits – a bignum::biginteger(), for instance. Not a raw vector; see below.
output	One of "character" (the default), "double" or "bignum".
family	The family the number is to be read as: "v4", "v6" or "v6_4in6", length 1 or length(x). Required.

Value

addr_to_integer() returns a character, double or biginteger vector as output asks, NA for a missing address. integer_to_addr() returns a raddr_address vector.

R has no unsigned integer, which decides the default

R's integer is **signed 32-bit**, so it holds barely half the IPv4 space – as.integer(4294967295) is NA – and R's double is exact only to 2^53, which is comfortable for IPv4 and twenty-five orders of magnitude short for IPv6. The carrier that always works is a decimal **string**, so that is what output = "character", the default, returns.

"character" Decimal digits, no padding, no separators. Always available, exact for both families.

"double" Exact for IPv4 and NA **for IPv6**, including the 4-in-6 family, whose value is 128 bits like any other address. A double cannot carry an IPv6 address, so raddr returns nothing rather than something close.

"bignum" A bignum::biginteger(). The only output that needs an installed package, and the only one that can fail – see below.

The bignum dependency is optional, and actually optional

bignum is in Suggests, and the two default-reachable outputs never touch it. The comparison worth stating: ipaddress::ip_to_integer() calls check_installed("bignum") before doing anything, so without that package it errors – including for IPv4, where no arbitrary-precision arithmetic is needed at all (verified 2026-07-28, ipaddress 1.0.3). raddr does its own arithmetic in base 10^6 over the four 32-bit words, so you can encode and decode every address of either family with nothing installed.

output = "bignum" does require the package, and **errors** when it is missing rather than quietly handing back the character vector. The digits would be right and the answers would not: character ordering is lexicographic, so max() of c("9", "16777216") is "9" and sort() puts 10 before 9. A caller who asked for numbers and silently received text gets a wrong answer out of the first thing they do with it. The error names the install command and the "character" alternative.

What bignum shows you is not what it stores

bignum displays 7 significant figures by default, and its as.character() and format() follow the display – so a biginteger holding 42540766411282592856903984951653826561 prints, formats, coerces and write.csv()s as "4.254077e+37". The stored value is exact and arithmetic on it is exact; only the rendering rounds. Use format(x, notation = "dec"), or raise options(bignum.sigfig),

to see all of it. `integer_to_addr()` reads a `biginteger` by its decimal notation for this reason, so the round trip is unaffected.

The family does not travel in the number, so you must pass it

`integer_to_addr()` requires `family`, and has no default. One integer names three different objects: `3221225985` is `192.0.2.1` as IPv4, and as a 128-bit value the deprecated IPv4-compatible `::192.0.2.1` (RFC 4291 §2.5.5), while `::ffff:192.0.2.1` is a fourth thing again. Nothing in the digits says which, so raddr does not guess – `ipaddress::integer_to_ip()` takes `is_ipv6 = NULL` and infers one.

`family` accepts `"v4"`, `"v6"` and `"v6_4in6"`, scalar or one per element, and takes the factor from `addr_family()` directly. `"v6"` and `"v6_4in6"` both mean 128 bits: which of the two families comes back is decided by the bits, exactly as it is when parsing a literal.

Out of range is `NA`, and so is anything that is not a number

A value of 2^32 or more with `family = "v4"`, 2^128 or more with an IPv6 family, a negative number, a sign, an exponent, a decimal point, or empty text all decode to `NA`. So does a `double` above 2^53, because such a double has already lost the value it was meant to carry – raddr will not decode the nearest representable number instead. Leading zeros and surrounding whitespace are accepted, being unambiguous in a decimal integer.

Like the other decoders in `addr_to_bytes()`, `integer_to_addr()` signals nothing about a *value* it cannot read: the answer is a missing address. A wrong *type* is a different matter and errors, as it does everywhere else in raddr. A `raw` vector is the case worth naming, because its `as.character()` is hexadecimal – reading `as.raw(16)` as a number would silently yield `0.0.0.10`. Bytes go to `bytes_to_addr()`, which knows they are bytes.

See Also

`addr_to_bytes()` for the byte, hex and binary pairs.

Examples

```
a <- addr_pton(c("192.0.2.1", "2001:db8::1", "::ffff:192.0.2.1"))
addr_to_integer(a)

# Exact in a double for IPv4, and NA rather than lossy for IPv6
addr_to_integer(a, output = "double")

# The family has to be carried alongside the number
integer_to_addr(3221225985, family = "v4")
integer_to_addr(3221225985, family = "v6")

# Which makes the round trip this
integer_to_addr(addr_to_integer(a), addr_family(a)) == a

# The largest address of each family
addr_to_integer(addr_pton("ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"))
```

`addr_transition_registry` *The transition-prefix overlay*

Description

The prefixes whose classification needs more granularity than the IANA special-purpose registries provide, and the bit geometry of the IPv4 addresses embedded in them.

Usage

```
addr_transition_registry(what = c("prefixes", "embeddings"))

addr_transition_version()
```

Arguments

`what`	Which table to return: `"prefixes"` (default) or `"embeddings"`.

Value

A data frame. For `"prefixes"`: `block`, `kind`, `rfc`, `note`. For `"embeddings"`: `kind`, `role`, `prefix_len`, `offset`, `length`, `complement`, where `offset` and `length` are bit positions counted from the most significant bit of the 128-bit address.

Why an overlay exists at all

IANA records `Globally Reachable` as `N/A` for Teredo (`2001::/32`) and 6to4 (`2002::/16`) – see `addr_registry()`. That is not an omission: reachability follows the **embedded** IPv4 address, which no prefix table can express. IANA is marking the point where table lookup stops being sufficient, and this overlay is what raddr uses past that point.

Separately stamped

This table has its own version, independent of `addr_registry_version()`. The two change for unrelated reasons – one when IANA republishes, the other when a maintainer transcribes another RFC – so neither stamp is evidence about the other. There is no `addr_transition_outdated()`: the RFCs this is drawn from do not expire.

Prefixes and embeddings are different shapes

`what = "prefixes"` gives fixed prefixes with a `kind`. `what = "embeddings"` gives one row per **contiguous segment** of an embedded IPv4 address, which is not always one row per form:

Teredo carries two addresses a server, in the clear, and a client stored bitwise-complemented so a NAT will not rewrite it (`complement = TRUE`). The client is the only complemented *address*, not the only complemented *field* – RFC 4380 section 4 also stores the mapped UDP port at bits 80-95 as XOR `0xFFFF`. This table reports addresses, so the port does not appear in it.

NAT64 geometry follows the prefix length, not a prefix RFC 6052 permits six lengths, and a network-specific prefix may be any prefix of one of them – so those rows carry a `prefix_len` and no block. At /40, /48 and /56 the embedded address **straddles the reserved u-byte** at bits 64-71 and arrives in two segments, most significant first.

ISATAP has an embedding but no prefix it is an interface-identifier pattern (`0000:5efe` or `0200:5efe`) that can sit under any `/64`.

See Also

`addr_registry()` for the IANA table this overlays.

Examples

```
addr_transition_registry()

# The two forms IANA declines to answer for
reg <- addr_registry()
reg[is.na(reg$globally_reachable) & is.na(reg$termination_date), "block"]

# RFC 6052's split geometry: two segments at /40, /48 and /56
emb <- addr_transition_registry("embeddings")
emb[emb$kind == "nat64", c("prefix_len", "offset", "length")]
```

---

addr_within_any *Is an address inside a block?*

Description

`addr_within()` tests each address against the block in the same position; `addr_within_any()` tests each address against **every** block and answers whether any of them contains it. The second is the denylist question.

Usage

```
addr_within_any(x, blocks)

addr_within(x, blocks)
```

Arguments

`x`	A `raddr_address` vector.
`blocks`	A character vector of CIDR blocks, `"10.0.0.0/8"` or `"2001:db8::/32"`. The address part is read by the RFC grammar, as `addr_strict()` reads one, so that a block means the same thing on every platform. For `addr_within()` it is recycled against `x`.

Value

A logical vector the length of `x`, `NA` where the address is missing.

The family decides the space, and 4-in-6 is IPv6

An IPv4 address is never inside an IPv6 block and an IPv6 address is never inside an IPv4 one, so those pairs are `FALSE` rather than an error – a mixed denylist is an ordinary thing to hold. A `v6_4in6` address such as `::ffff:192.0.2.1` searches the **IPv6** space: it is inside `::ffff:0:0/96` and it is *not* inside `192.0.2.0/24`, because it is a 128-bit address that happens to embed an IPv4 one. That embedding is a separate fact, reported by `addr_embeddings()`; testing the address it contains means naming that address. This is the width rule of `addr_to_bytes()` in its containment form.

A block is the question, so a bad block is an error

The decoders elsewhere in raddr return a missing value for input they cannot read. Blocks are the exception, because a block is not data being read – it is the question being asked. A denylist entry that silently matched nothing would be a hole in the denylist that the caller has no way to see. So a malformed block, a missing one, a prefix length outside `0:32` or `0:128`, and a block with **host bits set** all error, and the message names the fix.
`192.168.1.1/24` is refused rather than masked to `192.168.1.0/24`, because it is equally likely to be a host someone meant to write `/32`. A missing *address* is still a missing answer: `NA`, never `FALSE`.

See Also

`addr_classify()` for what the IANA registry says about an address, which is the question to ask when the blocks would have come from there.

Examples

```
a <- addr_pton(c("10.1.2.3", "192.0.2.1", "2001:db8::1", "::ffff:10.0.0.1"))

addr_within_any(a, c("10.0.0.0/8", "2001:db8::/32"))

# Recycled, one block per address
addr_within(a, "10.0.0.0/8")

# A 4-in-6 address is an IPv6 address: it is in the mapped block and not in
# the IPv4 block whose address it embeds
addr_within_any(addr_pton("::ffff:10.0.0.1"), "10.0.0.0/8")
addr_within_any(addr_pton("::ffff:10.0.0.1"), "::ffff:0:0/96")

# A /0 covers its own space and nothing else
addr_within_any(a, "0.0.0.0/0")
```

addr_zone *Read the zone ID of an address*

Description

The RFC 4007 zone ID is stored alongside the address bits, never inside them, and it does not participate in equality. Two addresses that differ only by zone compare equal; this is how you tell them apart.

Usage

```
addr_zone(x)
```

Arguments

x	A `raddr_address` vector.

Value

A character vector, `NA` where the address carries no zone.

Examples

```
a <- raddr_address(-25165824L, 0L, 0L, 1L, "v6", zone = "lo0")
b <- raddr_address(-25165824L, 0L, 0L, 1L, "v6", zone = "en0")
a == b
addr_zone(a) == addr_zone(b)
```

dialects *Read an address literal under one dialect*

Description

Six functions, one per dialect. They exist because standards and implementations disagree about what an IP address literal means, and raddr's answer is to show you all of the readings rather than pick one.

Usage

```
addr_strict(x)

addr_whatwg(x)

addr_pton(x)
```

```
addr_aton(x)

addr_getaddrinfo(x)

addr_curl(x)
```

Arguments

x	A character vector of address literals.

Details

The dialect is chosen by calling a named function. There is deliberately no `strict = FALSE` argument and no dialect knob buried in `...`: a named function is harder to helpfully default away than an argument is.

Value

A `raddr_address` vector, `NA` where the dialect rejects the input.

On paper

`addr_strict()` The RFC dotted-quad grammar: exactly four decimal octets, no leading zeros, no hex, no octal, no short form. This is what Python's `ipaddress`, Go and Rust accept. It is **not** what `inet_pton()` accepts, although the two are often conflated.

`addr_whatwg()` The WHATWG URL host parser – what browsers do. Hex and octal parts, one to four parts with the last filling the remainder, and one trailing dot dropped, so `1.2.3.` is `1.2.0.3`. Values above 2^32 - 1 are rejected rather than wrapped.

In reality

`addr_pton()` Apple `inet_pton()`. Four decimal parts, leading zeros allowed and ignored, so `0177.0.0.1` is **177.0.0.1** and not `127.0.0.1`.

`addr_aton()` Apple `inet_aton()`. Hex, octal and short forms, and three quirks worth knowing: a whole-host number is truncated to 32 bits rather than rejected, so `4294967296` is `0.0.0.0`; parsing stops at the first whitespace character and ignores the rest, so `1.2.3.4 junk` is an address; and a digitless `0x` is tolerated in any part but the last. `inet_aton()` is `AF_INET` by signature, so it rejects **every** IPv6 literal.

Both say *Apple* rather than POSIX or BSD, and that is load-bearing. Measured across Apple, glibc and musl on 2026-07-29, there is no reality-side reading the three libcs agree on: glibc and musl `inet_pton()` **reject** every leading zero above, and their `inet_aton()` rejects every overflow rather than wrapping it. raddr models Apple on all four reality-side readings, as a dialect that varied with the host would not be a function – and reports that choice here rather than implying a standard it does not have.

IPv6

The shape of the disagreement inverts. The two paper dialects agree about IPv6 on every measured input, and all of the divergence is on the reality side:

Leading zeros A hextet is four hex digits on paper. Apple `inet_pton()` counts only the *significant* four and lets the zeros run as wide as they like, so `0000000000001::` is `1::` where `strict` and `whatwg` reject. The dotted-quad tail splits the same way.

The zone ID The paper dialects have none: RFC 4291's grammar does not admit one and the WHATWG parser rejects `%`. The reality dialects accept a zone on any address and resolve nothing, so `%bogus0` parses. The zone is stored beside the bits and read with `addr_zone()`; it never enters the address and never affects equality.

`fe80::/10` `addr_getaddrinfo()` lifts the second hextet of a link-local address out into the zone and clears it, zone ID or not, so `fe80:abcd::1` is `fe80::1` with zone `43981` – while `addr_pton()` leaves it alone. One string, one machine, two different hosts. `addr_curl()` goes with `getaddrinfo`, because that is the entry point curl reaches.

Apple `inet_pton()` also does the reverse, writing a resolved interface index *into* the second hextet. raddr deliberately does not reproduce that: the index comes from the host's interface table, so it is not a function of the input, and raddr is pure and offline.

The lift is Apple's own, measured 2026-07-29. glibc and musl do not perform it – `fe80:abcd::1` stays `fe80:abcd::1` there – and their `inet_pton()` takes no zone ID at all, so the fold cannot arise either. That makes `addr_getaddrinfo()` and `addr_curl()` Apple readings across the whole of `fe80::/10` rather than at its edges. Outside that block the platforms agree.

Compositions

The last two are precedence orderings over the reality primitives, not parsers in their own right:

`addr_getaddrinfo()` `pton`, falling back to `aton`. Whitespace is the one place the composition leaks: `getaddrinfo()` rejects an input containing whitespace outright, where bare `aton` would accept it.

`addr_curl()` `aton`, falling back to `addr_getaddrinfo()` – the opposite precedence, which is the whole reason `192.0.048.1` reaches a host under curl that a browser refuses to dial.

The asymmetry in that second fallback is not a slip. curl's URL layer normalizes a numeric host itself, `aton`-style, and hands the resolver whatever is left, so the fallback is the resolver entry point rather than the bare parser under it. For IPv4 the distinction is invisible, because `getaddrinfo()` reduces to `pton` once `aton` has rejected. For IPv6 it is the whole composition: `aton` rejects every IPv6 literal, so `fe80:abcd::1` is `fe80::1` with zone `43981` under `addr_curl()`, exactly as under `addr_getaddrinfo()`.

What these do not give you

These are shortcuts for a caller who has already chosen a dialect. They return a bare address, so a rejected input comes back as `NA` with no reason attached. The total, outcome-bearing form – every reading at once, with the reason codes – is `addr_parse()`, and its result is what `addr_reading()` reads a single dialect back out of.

Provenance

The reality dialects and `addr_getaddrinfo()` were measured against Apple libc on macOS Darwin 25.4.0 arm64 on 2026-07-26. `addr_curl()`'s precedence was measured against real curl 8.20.0 on 2026-07-28 – until then it was derived from the other two rather than run, and the IPv6 half of it

was wrong. data-raw/oracle-ipv4.py and data-raw/oracle-tools.R regenerate the measurements; tests/testthat/test-ipv4.R holds them as the divergence table.

The same corpus was run under glibc 2.36 and musl 1.2.5 on 2026-07-29 by data-raw/oracle-libc-linux.sh, which is what fixes these functions to Apple rather than to a standard. Those fixtures are recorded, never modeled; tests/testthat/test-libc.R asserts the divergence set so a libc upgrade shows up as a changed file.

Examples

```
# One string, one machine, three different hosts
addr_strict("0177.0.0.1")
addr_whatwg("0177.0.0.1")
addr_pton("0177.0.0.1")

# curl reaches a host a browser refuses to dial
addr_whatwg("192.0.048.1")
addr_curl("192.0.048.1")

# inet_aton truncates a whole-host number instead of rejecting it
addr_aton("4294967296")

# Two libc entry points, one machine, two different IPv6 hosts -- and curl
# reaches the one that lifts the scope
addr_pton("fe80:abcd::1")
addr_getaddrinfo("fe80:abcd::1")
addr_curl("fe80:abcd::1")

# The zone travels beside the bits, so it does not affect equality
addr_pton("fe80::1%lo0") == addr_pton("fe80::1%en0")
addr_zone(addr_pton("fe80::1%lo0"))
```

---

is_raddr_address *Test whether an object is a* raddr_address

Description

Test whether an object is a raddr_address

Usage

```
is_raddr_address(x)
```

Arguments

x	An object.

Value

A single TRUE or FALSE.

Examples

```
is_raddr_address(raddr_address(0L, 0L, 0L, 1L, "v4"))
is_raddr_address("127.0.0.1")
```

---

is_raddr_class *Test whether an object is a* raddr_class

Description

Test whether an object is a raddr_class

Usage

```
is_raddr_class(x)
```

Arguments

x	An object.

Value

A single TRUE or FALSE.

Examples

```
is_raddr_class(addr_classify(addr_pton("127.0.0.1")))
is_raddr_class("127.0.0.1")
```

---

is_raddr_embedding *Test whether an object is a* raddr_embedding

---

Description

The elements of the embeddings column of a addr_classify() result. There is no public constructor: these are produced by classification, not built by hand.

Usage

```
is_raddr_embedding(x)
```

Arguments

x	An object.

Value

A single TRUE or FALSE.

Examples

```
is_raddr_embedding(addr_embeddings(addr_pton("64:ff9b::a9fe:a9fe"))[[1]])
is_raddr_embedding("169.254.169.254")
```

`is_raddr_parse` *Test whether an object is a* `raddr_parse`

Description

Test whether an object is a raddr_parse

Usage

```
is_raddr_parse(x)
```

Arguments

x	An object.

Value

A single TRUE or FALSE.

Examples

```
is_raddr_parse(addr_parse("127.0.0.1"))
is_raddr_parse("127.0.0.1")
```

`parse-accessors` *Pull one dialect's reading, outcome or reason codes back out*

Description

Accessors on a addr_parse() result. All of them admit the two compositions as well as the four primitives: the record stores only the primitives, and getaddrinfo and curl are resolved from those on request (section 3.2).

Usage

```
addr_reading(x, dialect)

addr_outcome(x, dialect)

addr_codes(x, dialect = NULL)

addr_status(x)

addr_is_divergent(x)

addr_input(x)
```

Arguments

x	A raddr_parse vector.
dialect	One of "strict", "whatwg", "pton", "aton", "getaddrinfo" or "curl". For addr_codes(), NULL unions every dialect's codes.

Details

dialect is a **view selector on output, not a leniency knob on input**. The parsing already happened, under every dialect, and choosing one here only chooses which of the finished readings to look at. That is why there is a dialect argument on these and not on addr_parse().

Value

addr_reading() a raddr_address; addr_outcome() a factor with levels "ok", "rejected" and "not_an_address"; addr_codes() a list of character vectors; addr_status() a factor with levels "ok", "divergent", "not_an_address" and "malformed"; addr_is_divergent() a logical vector.

Examples

```
p <- addr_parse(c("0177.0.0.1", "127.0.0.1", "example.com"))

addr_reading(p, "whatwg")
addr_reading(p, "curl")
addr_outcome(p, "strict")
addr_codes(p, "strict")
addr_status(p)
addr_is_divergent(p)
```

raddr_address *An IP address vector*

Description

`raddr_address()` builds a vector of IP addresses from raw 32-bit words. It is a low-level constructor: it does no parsing and accepts whatever bits it is given. Parsing text into addresses is the job of `addr_parse()` and the single-dialect shortcuts in dialects.

Usage

```
raddr_address(
  w1 = integer(),
  w2 = integer(),
  w3 = integer(),
  w4 = integer(),
  family = character(),
  zone = NA_character_
)
```

Arguments

`w1`, `w2`, `w3`, `w4`	Integer vectors of 32-bit words, most significant first. Interpreted as raw bit patterns: `NA_integer_` means `0x80000000`.
`family`	A character or factor vector of `"v4"`, `"v6"` or `"v6_4in6"`. `NA` marks a missing address.
`zone`	A character vector of RFC 4007 zone IDs, `NA` where absent.

Value

A `raddr_address` vector.

Storage

A `vctrs::new_rcrd()` with six fields:

`w1`-`w4` Four 32-bit words as `integer`, most significant first (big-endian). An IPv4 address occupies `w4` and leaves `w1`-`w3` zero.

`family` A factor with levels `"v4"`, `"v6"` and `"v6_4in6"`. `NA` marks a **missing address** – see below.

`zone` The RFC 4007 zone ID, or `NA` when the address carries none. Never stored in the address bits.

Missingness lives in `family`

R reserves the bit pattern `0x80000000` as `NA_integer_`, so a word cannot use `NA` to mean "absent" without also losing the one address that has that pattern. raddr therefore reads a word as raw bits and nothing else: `NA_integer_` in `w1`-`w4` means the pattern `0x80000000`, not missingness.

A row is a missing address if and only if its `family` is `NA`, and `is.na()` reports exactly that.

The consequence a caller can see is that `128.0.0.0` compares equal to itself, which is not true of every R package that stores addresses this way.

Equality and ordering

Equality is over the 128 bits and the family, and nothing else. The **zone does not participate**: `fe80::1%lo0` equals `fe80::1%en0`, because they are the same address named on two interfaces. Query `addr_zone()` when the interface matters.

Ordering is **total**, so `sort()` and `order()` work on a vector mixing families: IPv4 sorts before IPv6, and the 4-in-6 form sorts with IPv6 by its full 128 bits. Ordering agrees with equality – `x == y` implies `vctrs::vec_compare(x, y)` is `0`.

Examples

```
# 127.0.0.1 lives in w4
raddr_address(0L, 0L, 0L, 2130706433L, "v4")

# The zone travels alongside the bits, not inside them
addr_zone(raddr_address(-25165824L, 0L, 0L, 1L, "v6", zone = "lo0"))
```

Index