Summary:

libcidr is a library to make it easier to handle IP addresses and blocks, and manipulate them in various ways.

The core of the library is a pair of functions that take a human readable string and turn it into our internal representation of a CIDR address block (**cidr_from_str()**), and one to take that internal representation and turn it into a human-readable string (**cidr_to_str()**). There are a large number of options for how to format that string, as well.

Additionally, there are functions to compare different CIDR blocks, to determine if they're equal, or if one is contained within the other. This functionality can be useful for writing access-control code, or client-dependant configuration, or similar things. There are functions to manipulate address blocks and determine attributes of them, like network/broadcast addresses, the range of host addresses, the number of available host addresses, etc. There are functions to split a CIDR block into the two smaller blocks it contains, or to derive the parent block that it is itself contained within. And there are functions to translate to and from in_addr-type structures, which the operating system commonly uses to represent addresses for handle socket connections and so forth.

In short, just about anything you might do in a program with IP addressing, whether referring to individual hosts, or to any sized subnets, libcidr is designed to simplify handling. It's not a DNS library, nor is it a socket abstraction layer. It's just a set of functions for manipulating raw IP addresses in various ways.

The functions generally follow standard C conventions. They tend to return 0 or a pointer when acting properly, and -1 or NULL when something went wrong (unless the function usage suggests other returns, of course, as in **cidr_get_pflen()**). They set errno when returning an error; the error codes each function can return are documented with the function.

libcidr doesn't use any threading itself. It should, however, be safe to use in any threaded program if used sensibly. Only a very few functions use static strings, and those that do (**cidr_version()** and **cidr_numaddr()** and its related functions being the only ones I can think of) tend to be constant strings as well, so they wouldn't be changing. Of course, you don't want to **cidr_free()** a **CIDR** in one thread while you're still using it in another, but if you do, it's not libcidr's fault.

For the current version or any extra information, see the libcidr project homepage, at <http://www.over-yonder.net/~fullermd/projects/libcidr>.

This reference manual is build using the codelibrary SGML DTD, which is specifically designed for documenting libraries. See the codelibrary homepage at <http://www.over-yonder.net/~fullermd/projects/sgml/codelibrary> for more details on it.

Contents

Data structures:

Functions:

Data structures:

- CIDR: A single CIDR-format IP block

*** This datatype is for internal use only ***

Note:

Use the **cidr_free()** function to free the memory associated with this datatype, and the **cidr_alloc()** function to allocate and initialize the structure.

Members:

- int version: The structure version. This is reserved for future use, and put in to hold its place at the start of the array.

- uint8_t addr[16]: The 16 octets that make up an IP address. For v6 addresses, all are used. For v4 addresses, only the last 4 are really used. The prior 2 octets are filled in with all-ones, so that the internal representation matches the v4-compat IPv6 addressing block. This is useful when, for instance, using **cidr_to_in6addr()**, in that it gives you the expected result.

- uint8_t mask[16]: The 16 octets that make up an IP netmask. For v4 addresses, only the last 4 are really used; the rest are intialized to all-bits-one however, which is correct in spirit.

- int proto: The protocol the address described is. Currently possible values are CIDR_IPV4 and CIDR_IPV6. I think that's pretty self-explanatory.

Functions:

- cidr_addr_broadcast(): Find the broadcast address

Summary:

Generate a **CIDR** structure describing the broadcast address of the passed-in netblock.

Note that using this with an IPv6 netblock is technically asking for something that doesn't exist, since IPv6 doesn't have subnet broadcast addresses. This function will still return the all-1's address though, on the assumption that if you're asking the question, it's the answer you want.

An additional somewhat specialized case is that of an IPv4 /31 or IPv6 /127. Depending on your interpretation and usage, this is either a useless subnet since it only contains network and broadcast and no hosts, or a subnet that holds 2 hosts with neither network or broadcast address (commonly used for point-to-point links). As a result, **cidr_addr_broadcast()** and **cidr_addr_network()** will give answers as if it were a host-less subnet, while **cidr_addr_hostmax()** and **cidr_addr_hostmin()** will answer as though it were a 2-host subnet. It can be seen as a little strange for **cidr_addr_broadcast()** and **cidr_addr_hostmax()** to give the same answer, but as above, libcidr assumes that if you ask the question, you want the answer that makes sense in that context.

Similar caveats apply to calling these on a v4 /32 or v6 /128. Ask a silly question, get a silly answer :)

The returned structure should be cleaned up using **cidr_free()**.

Arguments:

- const **CIDR** * addr: A **CIDR** structure describing an arbitrary netblock.

Return value:

CIDR *

Returns a pointer to a **CIDR** structure describing the broadcast address on success. Returns NULL on failure.

Error codes:

- [EFAULT]
  Given NULL

Note:

cidr_addr_broadcast() can also fail and set errno for any of the reasons listed for **cidr_alloc()**.

- cidr_addr_hostmax(): Find the highest host address

Summary:

Generate a **CIDR** structure describing the highest-numbered address available for a host IP in the given netblock.

See the discussion under **cidr_addr_broadcast()** concerning near- amd maximal-prefix-length blocks for edge case details.

The returned structure should be cleaned up using **cidr_free()**.

Arguments:

- const **CIDR** * addr: A **CIDR** structure describing an arbitrary netblock.

Return value:

CIDR *

Returns a pointer to a **CIDR** structure describing the max host address on success. Returns NULL on failure.

Error codes:

Note:

cidr_addr_hostmax() can fail and set errno for any of the reasons listed for **cidr_addr_broadcast()**.

- cidr_addr_hostmin(): Find the lowest host address

Summary:

Generate a **CIDR** structure describing the lowest-numbered address available for a host IP in the given netblock.

See the discussion under **cidr_addr_broadcast()** concerning near- amd maximal-prefix-length blocks for edge case details.

The returned structure should be cleaned up using **cidr_free()**.

Arguments:

- const **CIDR** * addr: A **CIDR** structure describing an arbitrary netblock.

Return value:

CIDR *

Returns a pointer to a **CIDR** structure describing the min host address on success. Returns NULL on failure.

Error codes:

Note:

> **cidr_addr_hostmin()** can fail and set errno for any of the reasons listed for **cidr_addr_network()**.

- cidr_addr_network(): Find the network address

Summary:

Generate a **CIDR** structure describing the network address of the passed-in netblock.

See the discussion under **cidr_addr_broadcast()** concerning near- amd maximal-prefix-length blocks for edge case details.

The returned structure should be cleaned up using **cidr_free()**.

Arguments:

- const **CIDR** * addr: A **CIDR** structure describing an arbitrary netblock.

Return value:

CIDR *

> Returns a pointer to a **CIDR** structure describing the network address on success. Returns NULL on failure.

Error codes:

- [EFAULT]
  Given NULL

Note:

> **cidr_addr_network()** can also fail and set errno for any of the reasons listed for **cidr_alloc()**.

- cidr_alloc(): Create a **CIDR**

Summary:

Allocate memory for a **CIDR** structure and initialize the necessary pieces.

The returned structure should be cleaned up using **cidr_free()**.

Note that you should probably never need to call this function yourself; you'll generally get your **CIDR** structures as a return from a function like **cidr_from_str()** or **cidr_from_inaddr()**.

Arguments:

None.

Return value:

CIDR *

Returns a pointer to an initialized **CIDR** structure on success. Returns NULL on failure.

Error codes:

- [ENOMEM]
  malloc() failed

- cidr_contains(): Compare netblocks

Summary:

This function is passed two **CIDR** structures describing a pair of netblocks. It then determines if the latter is wholly contained within the former.

A common use-case of this will generally involve the second "block" actually being a host (/32 or /128) address, as when you're implementing ACL's. But that's really just a specific case of the second block being any other size; there's nothing special or magical about it. As far as libcidr is concerned, they're just two netblocks.

Arguments:

- const **CIDR** * big: The netblock which may (or may not) contain the second arg.
- const **CIDR** * little: The netblock which may (or may not) be contained within the first arg.

Return value:

int

Returns 0 if little is wholly contained within big. Returns -1 if it's not, or if an error occured.

Error codes:

- [0]
  No error (little not in big)
- [EFAULT]
  Passed NULL
- [EINVAL]
  Invalid argument
- [ENOENT]
  Internal error (shouldn't happen)
- [EPROTO]
  Protocols don't match

- cidr_dup(): Duplicate a netblock

Summary:

Allocate a **CIDR**, and fill it in with a duplicate of the information given. The returned structure should be cleaned up using **cidr_free()**.

Arguments:

- const **CIDR** * src: A **CIDR** structure to be copied.

Return value:

CIDR *

Returns a **CIDR** struct containing a copy of src. Returns NULL on failure.

Error codes:

Note:

> **cidr_dup()** can fail and set errno for any of the reasons listed for **cidr_alloc()**.

- cidr_equals(): Compare two blocks for equality

Summary:

This function is passed two **CIDR** structures describing a pair of netblocks. It checks to see if they happen to describe the same netblock.

Arguments:

- const **CIDR** * one: One netblock.
- const **CIDR** * two: Another netblock.

Return value:

int

Returns 0 if the two **CIDR** structs describe the same netblock. Returns -1 otherwise.

- cidr_free(): Free a **CIDR** structure.

Summary:

Takes a **CIDR** structure and **free()**'s all its component parts.

Arguments:

- **CIDR** * tofree: A single **CIDR** structure which has outlived its usefulness.

Return value:

void

- cidr_from_inaddr(): Parse a struct in_addr

Summary:

Takes a populated struct in_addr, as you'd get from **accept()** or **getaddrinfo()** or similar functions. Parses it out and generates a **CIDR** structure based on it. Note that an in_addr only contains a host address, so the netmask is initialized to all-1's (/32).

Arguments:

- const struct in_addr * uaddr: A populated struct in_addr, from whatever source obtained.

Return value:

CIDR *

Returns a pointer to a populated **CIDR** containing the address in the passed-in struct in_addr. The netmask is initialized to all-1's, and the protocol to IPv4. Use **cidr_free()** to free the structure when you're finished with it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL

Note:

> **cidr_from_inaddr()** can also fail and set errno for any of the reasons listed for **cidr_alloc()**.

- cidr_from_in6addr(): Parse a struct in6_addr

Summary:

Takes a populated struct in6_addr, as you'd get from **accept()** or **getaddrinfo()** or similar functions. Parses it out and generates a **CIDR** structure based on it. Note that a in6_addr only contains a host address, so the netmask is initialized to all-1's (/128).

Arguments:

- const struct in6_addr * uaddr: A populated struct in6_addr, from whatever source obtained.

Return value:

CIDR *

Returns a pointer to a populated **CIDR** containing the address in the passed-in struct in6_addr. The netmask is initialized to all-1's, and the protocol to IPv6 (though it may contain an IPv4-mapped address). Use **cidr_free()** to free the structure when you're finished with it. Returns NULL on error.

Error codes:

- [EFAULT]
    Passed NULL

Note:

> **cidr_from_inaddr()** can also fail and set errno for any of the reasons listed for **cidr_alloc()**.

- cidr_from_str(): Parse a human-readable string

Summary:

Takes in a netblock description as a human-readable string, and creates a **CIDR** structure from it.

This is probably the most intricate function in the library. It accepts addresses in "address/mask" format. 'address' is an IP address in valid written form. For IPv4, it's 1 through 4 period-separated pieces, expressed in octal, hex, or decimal, with the last octet being treated as an 8, 16, 24, or 32-bit quantity depending on whether there are 4, 3, 2, or 1 pieces given (respectively). Of course, you're nuts for using that flexibility. For IPv6, it's nice and simple; 8 colon-separated double-octets, excepting that the last 4 octets can be expressed as a 4-piece dotted-decimal, like an IPv4 address (the full flexibility of the IPv4 parsing engine is not available, however; intentionally, though that may change if necessary). 'mask' can be either a prefix length (/0-/32 for IPv4, /0-/128 for IPv6), or a netmask written in the standard form for the address family.

IPv6 addresses can be specified in fully expanded form, or with ::-style contraction. IPv4-mapped IPv6 addresses (::ffff:a.b.c.d), will be treated as IPv6 addresses. The mask can be left off, in which case addresses are treated as host addresses (/32 or /128, depending on address family).

Also, **cidr_from_str()** will parse DNS PTR-record-style address formats. That is, representations like "4.3.2.1.in-addr.arpa" for IPv4, and an extremely long and annoying form ending in .ip6.arpa for IPv6. **cidr_from_str()** also understands the deprecated RFC1886 form of IPv6 PTR records, which ends in .ip6.int, though **cidr_to_str()** will only generate the current RFC3152-style .ip6.arpa version. Note also that while **cidr_to_str()** treats all addresses as host addresses when building the PTR string (ignoring the netmask), **cidr_from_str()** will fill in the netmask bits as appropriate for the string given; any octets (or half-octets, in the IPv6 form) that are left off the beginning will have their netmask bits set to 0.

It's not the intention of the author that this function necessarily be able to decipher any possible address format. However, the capabilities given should parse any rational address specification, and many irrational ones (like hex/oct and collapsed v4 addresses). The intention is rather to support the ways the addresses and netmasks are commonly written and read, so that a human-readable form can quickly be transformed into a format that libcidr can then use in its various ways, whether through comparing addresses with functions like **cidr_contains**(), or generating references and stats about a netblock with functions like **cidr_addr_broadcast()** and **cidr_numhost()**, or simply spitting it out in different human-readable forms with **cidr_to_str()**.

Arguments:

- const char * addr: A string containing some human-readable IP block.

Return value:

CIDR *

Returns a pointer to a populated **CIDR** describing (hopefully) the block you talked about in the string. Use **cidr_free()** to free the structure when you're finished with it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL
- [EINVAL]
  Can't parse the input string
- [ENOENT]
  Internal error (shouldn't happen)

Note:

cidr_from_str() can also fail and set errno for any of the reasons listed for **cidr_alloc()** or **cidr_get_pflen()**.

- cidr_get_addr(): Return address bits

Summary:

Return the address bits which compose the address. This should be used in preference to simply referencing inside the **CIDR** manually in external code, since the structure might change on you.

Generally, if you think you need to call this, you should probably rethink what you're doing. Most of the time, one of the formatted outputs from **cidr_to_str()** or one of the manipulation functions like **cidr_addr_hostmin()** is what you want. Still, there are times when you're interesting in manipulating the address by yourself as a bunch of binary bits (the cidrcalc example program does this), so this function should be used instead of groping around in the structure manually.

Arguments:

- const **CIDR** * addr: An arbitrary netblock.

Return value:

uint8_t *

Returns a pointer to an 16-element array of uint8_t's representing the address. This array must be **free()**'d when you're through with it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL
- [ENOMEM]
  malloc() failed

- cidr_get_mask(): Return netmask bits

Summary:

Return the netmask bits which of the given netblock. This should be used in preference to simply referencing inside the **CIDR** manually in external code, since the structure might change on you.

See further notes about the desirability of using this function above in the notes for **cidr_get_addr()**.

Arguments:

- const **CIDR** * addr: An arbitrary netblock.

Return value:

uint8_t *

Returns a pointer to an 16-element array of uint8_t's representing the netmask. This array must be **free()**'d when you're through with it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL
- [ENOMEM]
  malloc() failed

- cidr_get_pflen(): Network bits in the netmask

Summary:

Poke around the netmask of the passed-in **CIDR** structure and determine how many bits there are in the netmask, as appropriate to the address family.

Arguments:

- const **CIDR** * block: An arbitrary netblock.

Return value:

int

Returns the number of network bits in the netmask (0-32 for IPv4, 0-128 for IPv6). Returns -1 on error.

Error codes:

- [EFAULT]
  Passed NULL
- [EINVAL]
  Invalid (non-contiguous) netmask
- [ENOENT]
  Internal error (shouldn't happen)

- cidr_get_proto(): Find a netblock's protocol family

Summary:

Returns the protocol family of an address using one of the defined constants. The current choices are CIDR_IPV4 and CIDR_IPV6.

Arguments:

- const **CIDR** * addr: An arbitrary netblock.

Return value:

int

Returns the address family of the given netblock.

Error codes:

- [EFAULT]
  Passed NULL

- cidr_is_v4mapped(): Is address IPv4-mapped IPv6 address?

Summary:

An IPv6 address may be in the network range reserved for IPv4-mapped addresses. This function will tell you whether it is or not. Note that an IPv4 **CIDR** is NOT considered an IPv4-mapped address, and so will return failure.

Arguments:

- const **CIDR** * addr: An arbitrary netblock.

Return value:

int

Returns 0 if the address is an IPv4-mapped IPv6 address. Returns -1 otherwise.

- cidr_net_subnets(): Divide a netblock

Summary:

Take in a netblock, and derive the two netblocks which it divides up into. Return them in an array.

Arguments:

- const **CIDR** * addr: The netblock to subdivide.

Return value:

CIDR **

Returns a 2-element array of **CIDR** structs, containing the two subnets of addr. Each of the elements should be cleaned up with **cidr_free()**, and the array itself then cleaned up with **free()**. Returns NULL on failure.

Error codes:

- [0]
  No error (already a /32 or /128)
- [EFAULT]
  Passed NULL argument
- [ENOMEM]
  malloc() failed

Note:

cidr_net_subnets() can also fail and set errno for any of the reasons listed for **cidr_addr_network()** or **cidr_dup()**.

- cidr_net_supernet(): Undivide a netblock

Summary:

Take in a netblock, and derive the parent netblock in which it fits.

Arguments:

- const **CIDR** * addr: The netblock to find the parent of.

Return value:

CIDR *

Returns a **CIDR** struct defining the parent network of addr. Clean this up with **cidr_free()** when you're finished with it. Returns NULL on failure.

Error codes:

- [0]
  No error (already a /0)
- [EFAULT]
  Passed NULL argument

Note:

cidr_net_supernet() can also fail and set errno for any of the reasons listed for **cidr_dup()**.

- cidr_numaddr(): Addresses in a netblock

Summary:

Determine the total number of addresses in a netblock (including the network and broadcast addresses).

This function returns a pointer to a pre-formatted string because we're potentially returning a value up to 2**128. I don't feel like trying to portably do 128-bit arithmetic. Do you?

Arguments:

- const **CIDR** * addr: An arbitrary netblock.

Return value:

const char *

Returns a pointer to a string containing the number of addresses in the netblock. Note that this is a static string; it should not be overwritten, and doesn't need to be **free()**'d. Make a copy if you want to manipulate it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL

Note:

> **cidr_numaddr()** can also also fail and set errno for any of the reasons listed for **cidr_numaddr_pflen()**.

- cidr_numaddr_pflen(): Addresses in a prefix length

Summary:

Determine the total number of addresses in a netblock with the given prefix length (including the network and broadcast addresses).

Note that this takes an IPv6 prefix length; that is, 0-128. If you're interested in an IPv4 address with a given prefix length, add 96 to it when you call this function.

See the note in **cidr_numaddr()** for why we're returning a string and not a number.

Arguments:

- int pflen: A prefix length (0-128).

Return value:

const char *

> Returns a pointer to a string containing the number of addresses in the netblock. Note that this is a static string; it should not be overwritten, and doesn't need to be **free()**'d. Make a copy if you want to manipulate it. Returns NULL on error.

Error codes:

- [EINVAL]
  Invalid prefix length

- cidr_numhost(): Host addresses in a netblock

Summary:

Determine the total number of host addresses in a netblock (excluding the network and broadcast addresses).

See the note in **cidr_numaddr()** for why we're returning a string and not a number.

Arguments:

- const **CIDR** * addr: An arbitrary netblock.

Return value:

const char *

Returns a pointer to a string containing the number of host addresses in the netblock. Note that this is a static string; it should not be overwritten, and doesn't need to be **free()**'d. Make a copy if you want to manipulate it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL

Note:

> **cidr_numhost()** can also also fail and set errno for any of the reasons listed for **cidr_numhost_pflen()**.

- cidr_numhost_pflen(): Host addresses in a prefix length

Summary:

Determine the total number of host addresses in a netblock with the given prefix length (excluding the network and broadcast addresses).

Note that this takes an IPv6 prefix length; that is, 0-128. If you're interested in an IPv4 address with a given prefix length, add 96 to it when you call this function.

See the note in **cidr_numaddr()** for why we're returning a string and not a number.

Arguments:

- int pflen: A prefix length (0-128).

Return value:

const char *

Returns a pointer to a string containing the number of host addresses in the netblock. Note that this is a static string; it should not be overwritten, and doesn't need to be **free()**'d. Make a copy if you want to manipulate it. Returns NULL on error.

Error codes:

- [EINVAL]
  Invalid prefix length

- cidr_to_inaddr(): Create a struct in_addr

Summary:

Takes in a **CIDR** and creates a struct in_addr from it. This struct can then be used in **connect()** or similar network-related functions. If the users passes in a struct in_addr, it will be filled in. Otherwise, one will be allocated and returned.

Arguments:

- const **CIDR** * addr: A **CIDR** structure describing the host to be translated into a struct in_addr. Note that the netmask is irrelevant and will be ignored. **cidr_to_inaddr()** supports only IPv4 addresses, as the underlying structure only does.
- struct in_addr * uptr: A pointer to a pre-allocated struct in_addr, or NULL. If non-NULL, the pointed-at structure will be filled in. If NULL, a new structure will be allocated, filled in, and returned.

Return value:

struct in_addr *

Returns a pointer to the filled-in struct in_addr. If the user passed one in, this will just point to the same place and can profitably be ignored. If the user passed in NULL, this will point to the struct in_addr we allocated, which should be **free()**'d by the user when they're finished with it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL
- [ENOMEM]
  malloc() failed
- [EPROTOTYPE]
  Bad protocol type (must be IPv4)

- cidr_to_in6addr(): Create a struct in6_addr

Summary:

Takes in a **CIDR** and creates a struct in6_addr from it. This struct can then be used in **connect()** or similar network-related functions. If the users passes in a struct in6_addr, it will be filled in. Otherwise, one will be allocated and returned.

Arguments:

- const **CIDR** * addr: A **CIDR** structure describing the host to be translated into a struct in6_addr. Note that the netmask is irrelevant and will be ignored. **cidr_to_in6addr()** supports both IPv4 and IPv6 addresses, as the underlying structure does as well. IPv4 addresses are treated as v4-mapped IPv6 addresses.

- struct in6_addr * uptr: A pointer to a pre-allocated struct in6_addr, or NULL. If non-NULL, the pointed-at structure will be filled in. If NULL, a new structure will be allocated, filled in, and returned.

Return value:

struct in6_addr *

Returns a pointer to the filled-in struct in6_addr. If the user passed one in, this will just point to the same place and can profitably be ignored. If the user passed in NULL, this will point to the struct in6_addr we allocated, which should be **free**()'d by the user when they're finished with it. Returns NULL on error.

Error codes:

- [EFAULT]
  Passed NULL
- [ENOMEM]
  malloc() failed
- [EPROTOTYPE]
  Bad protocol type (must be IPv4 or IPv6)

- cidr_to_str(): Create a human-readable netblock description

Summary:

Takes in a **CIDR** structure, and generates up a human-readable string describing the netblock. This function has a lot of flexibility, depending on the flags passed to it. The default output is "address/pflen" form, with the address in a reasonably compact form, and the prefix length given numerically. Flags alter the output in various ways, and are set as bitmasks, so they can be combined however you wish. They can be used in any combination that makes sense, and a large number of combinations that don't.

The current flags are:

CIDR_NOFLAGS: A stand-in for when you just want the default output
CIDR_NOCOMPACT: Don't do ::-style IPv6 compaction
CIDR_VERBOSE: Show leading 0's in octets [v6 only]
CIDR_USEV6: Use IPv4-mapped address form for IPv4 addresses (::ffff:a.b.c.d)
CIDR_USEV4COMPAT: Use IPv4-compat form (::a.b.c.d) instead of IPv4-mapped form (only meaningful in combination with CIDR_USEV6)
CIDR_NETMASK: Return a netmask in standard form after the slash, instead of the prefix length. Note that the form of the netmask can thus be altered by the various flags that alter how the address is displayed.
CIDR_ONLYADDR: Show only the address, without the prefix/netmask
CIDR_ONLYPFLEN: Show only the prefix length (or netmask, when combined with CIDR_NETMASK), without the address.
CIDR_WILDCARD: Show a Cisco-style wildcard mask instead of the netmask (only meaningful in combination with CIDR_NETMASK)
CIDR_FORCEV6: Forces treating the **CIDR** as an IPv6 address, no matter what it really is. This doesn't do any conversion or translation; just treats the raw data as if it were IPv6.

CIDR_FORCEV4: Forces treating the **CIDR** as an IPv4 address, no matter what it really is. This doesn't do any conversion or translation; just treats the raw data as if it were IPv4.
CIDR_REVERSE: Generates a .in-addr.arpa or .ip6.arpa-style PTR record name for the given block. Note that this always treats it solely as an address; the netmask is ignored. See some notes in **cidr_from_str()** for details of the asymmetric treatment of this form of address representation relating to the netmask.

Many combinations can give somewhat surprising results, but they should allow any of a host of manipulations to output just the data you might be interested in. The "mkstr" test program in the source tree is extremely useful for manual testing of the various flags to see visually what they do, and is a lot quicker than trying to code them all to test it out. Use it to your advantage.

Arguments:

- const **CIDR** * block: The **CIDR** structure to generate a string form of. The address family will be autodetected.
- int flags: A bitmask of the various possible flags the function accepts.

Return value:

char *

Returns a pointer to a string containing the representation of the network. Be sure to **free()** it when you're finished.

Error codes:

- [EINVAL]
  Invalid argument (bad block or flags)
- [ENOENT]
  Internal error (shouldn't happen)
- [ENOMEM]
  malloc() failed

Note:

> **cidr_to_str()** can also fail and set errno for any of the reasons listed for **cidr_alloc()** or **cidr_get_pflen()**.

- cidr_version(): Library version

Summary:

Returns a static string describing the library release version.

Arguments:

None.

Return value:

const char *

Returns a pointer to a static string describing the library version number. It shouldn’t be overwritten or **free()**’d.