NAME

Net::IP − Perl extension for manipulating IPv4/IPv6 addresses

SYNOPSIS

```
  use Net::IP;

  my $ip = new Net::IP ('193.0.1/24') or die (Net::IP::Error());
  print ("IP  : ".$ip->ip()."\n");
  print ("Sho : ".$ip->short()."\n");
  print ("Bin : ".$ip->binip()."\n");
  print ("Int : ".$ip->intip()."\n");
  print ("Mask: ".$ip->mask()."\n");
  print ("Last: ".$ip->last_ip()."\n");
  print ("Len : ".$ip->prefixlen()."\n");
  print ("Size: ".$ip->size()."\n");
  print ("Type: ".$ip->iptype()."\n");
  print ("Rev:  ".$ip->reverse_ip()."\n");
```

DESCRIPTION

This module provides functions to deal with **IPv4/IPv6** addresses. The module can be used as a class, allowing the user to instantiate IP objects, which can be single IP addresses, prefixes, or ranges of addresses. There is also a procedural way of accessing most of the functions. Most subroutines can take either **IPv4** or **IPv6** addresses transparently.

OBJECT-ORIENTED INTERFACE

Object Creation

A Net::IP object can be created from a single IP address:

```
  $ip = new Net::IP ('193.0.1.46') || die ...
```

Or from a Classless Prefix (a /24 prefix is equivalent to a C class):

```
  $ip = new Net::IP ('195.114.80/24') || die ...
```

Or from a range of addresses:

```
  $ip = new Net::IP ('20.34.101.207 - 201.3.9.99') || die ...
```

Or from a address plus a number:

```
  $ip = new Net::IP ('20.34.10.0 + 255') || die ...
```

The *new()* function accepts IPv4 and IPv6 addresses:

```
  $ip = new Net::IP ('dead:beef::/32') || die ...
```

Optionally, the function can be passed the version of the IP. Otherwise, it tries to guess what the version is (see ***_is_ipv4()*** and ***_is_ipv6()***).

```
  $ip = new Net::IP ('195/8',4); # Class A
```

OBJECT METHODS

Most of these methods are front-ends for the real functions, which use a procedural interface. Most functions return undef on failure, and a true value on success. A detailed description of the procedural interface is provided below.

set

Set an IP address in an existing IP object. This method has the same functionality as the *new()* method, except that it reuses an existing object to store the new IP.

```
$ip->set('130.23.1/24',4);
```

Like *new()*, *set()* takes two arguments − a string used to build an IP address, prefix, or range, and optionally, the IP version of the considered address.

It returns an IP object on success, and undef on failure.

error

Return the current object error string. The error string is set whenever one of the methods produces an error. Also, a global, class-wide ***Error()*** function is available.

```
warn ($ip->error());
```

errno

Return the current object error number. The error number is set whenever one of the methods produces an error. Also, a global **$ERRNO** variable is set when an error is produced.

```
warn ($ip->errno());
```

ip

Return the IP address (or first IP of the prefix or range) in quad format, as a string.

```
print ($ip->ip());
```

binip

Return the IP address as a binary string of 0s and 1s.

```
print ($ip->binip());
```

prefixlen

Return the length in bits of the current prefix.

```
print ($ip->prefixlen());
```

version

Return the version of the current IP object (4 or 6).

```
print ($ip->version());
```

size

Return the number of IP addresses in the current prefix or range. Use of this function requires Math::BigInt.

```
print ($ip->size());
```

binmask

Return the binary mask of the current prefix, if applicable.

```
print ($ip->binmask());
```

mask

Return the mask in quad format of the current prefix.

```
print ($ip->mask());
```

prefix

Return the full prefix (ip+prefix length) in quad (standard) format.

```
print ($ip->prefix());
```

print

Print the IP object (IP/Prefix or First – Last)

```
print ($ip->print());
```

intip

Convert the IP in integer format and return it as a Math::BigInt object.

```
print ($ip->intip());
```

hexip

Return the IP in hex format

```
print ($ip->hexip());
```

hexmask

Return the mask in hex format

```
print ($ip->hexmask());
```

short

Return the IP in short format: IPv4 addresses: 194.5/16 IPv6 addresses: ab32:f000::

```
print ($ip->short());
```

iptype

Return the IP Type − this describes the type of an IP (Public, Private, Reserved, etc.) See procedural interface ip_iptype for more details.

```
print ($ip->iptype());
```

reverse_ip

Return the reverse IP for a given IP address (in.addr. format).

```
print ($ip->reserve_ip());
```

last_ip

Return the last IP of a prefix/range in quad format.

```
print ($ip->last_ip());
```

last_bin

Return the last IP of a prefix/range in binary format.

```
print ($ip->last_bin());
```

last_int

Return the last IP of a prefix/range in integer format.

```
print ($ip->last_int());
```

find_prefixes

This function finds all the prefixes that can be found between the two addresses of a range. The function returns a list of prefixes.

```
@list = $ip->find_prefixes($other_ip));
```

bincomp

Binary comparaison of two IP objects. The function takes an operation and an IP object as arguments. It returns a boolean value.

The operation can be one of: lt: less than (smaller than) le: smaller or equal to gt: greater than ge: greater or equal to

```
if ($ip->bincomp('lt',$ip2) {...}
```

binadd

Binary addition of two IP objects. The value returned is an IP object.

```
my $sum = $ip->binadd($ip2);
```

aggregate

Aggregate 2 IPs − Append one range/prefix of IPs to another. The last address of the first range must be the one immediately preceding the first address of the second range. A new IP object is returned.

```
my $total = $ip->aggregate($ip2);
```

overlaps

Check if two IP ranges/prefixes overlap each other. The value returned by the function should be one of: `$IP_PARTIAL_OVERLAP` (ranges overlap) `$IP_NO_OVERLAP` (no overlap) `$IP_A_IN_B_OVERLAP` (range2 contains range1) `$IP_B_IN_A_OVERLAP` (range1 contains range2) `$IP_IDENTICAL` (ranges are identical) undef (problem)

```
if ($ip->overlaps($ip2)==$IP_A_IN_B_OVERLAP) {...};
```

looping

The + operator is overloaded in order to allow looping though a whole range of IP addresses:

```
  my $ip = new Net::IP ('195.45.6.7 - 195.45.6.19') || die;
  # Loop
  do {
      print $ip->ip(), "\n";
  } while (++$ip);
```

The ++ operator returns undef when the last address of the range is reached.

auth

Return IP authority information from the IP::Authority module

`$auth = ip−auth` ();>

Note: IPv4 only

PROCEDURAL INTERFACE

These functions do the real work in the module. Like the OO methods, most of these return undef on failure. In order to access error codes and strings, instead of using `$ip−>`*error()* and `$ip−>`*errno()*, use the global functions `Error()` and `Errno()`.

The functions of the procedural interface are not exported by default. In order to import these functions, you need to modify the use statement for the module:

`use Net::IP qw(:PROC);`

Error

Returns the error string corresponding to the last error generated in the module. This is also useful for the OO interface, as if the *new()* function fails, we cannot call `$ip−>`*error()* and so we have to use *Error()*.

warn *Error()*;

Errno

Returns a numeric error code corresponding to the error string returned by Error.

ip_iptobin

Transform an IP address into a bit string.

```
    Params  : IP address, IP version
    Returns : binary IP string on success, undef otherwise
```

`$binip = ip_iptobin ($ip,6);`

ip_bintoip

Transform a bit string into an IP address

```
    Params  : binary IP, IP version
    Returns : IP address on success, undef otherwise
```

`$ip = ip_bintoip ($binip,6);`

ip_bintoint

Transform a bit string into a BigInt.

```
    Params  : binary IP
    Returns : BigInt
```

`$bigint = new Math::BigInt (ip_bintoint($binip));`

ip_inttobin

Transform a BigInt into a bit string. *Warning*: sets warnings (`−w`) off. This is necessary because Math::BigInt is not compliant.

```
    Params  : BigInt, IP version
    Returns : binary IP
```

```
$binip = ip_inttobin ($bigint);
```

ip_get_version

Try to guess the IP version of an IP address.

```
    Params  : IP address
    Returns : 4, 6, undef(unable to determine)
```

```
$version = ip_get_version ($ip)
```

ip_is_ipv4

Check if an IP address is of type 4.

```
    Params  : IP address
    Returns : 1 (yes) or 0 (no)
```

```
ip_is_ipv4($ip) and print "$ip is IPv4";
```

ip_is_ipv6

Check if an IP address is of type 6.

```
    Params            : IP address
    Returns           : 1 (yes) or 0 (no)
```

```
ip_is_ipv6($ip) and print "$ip is IPv6";
```

ip_expand_address

Expand an IP address from compact notation.

```
    Params  : IP address, IP version
    Returns : expanded IP address or undef on failure
```

```
$ip = ip_expand_address ($ip,4);
```

ip_get_mask

Get IP mask from prefix length.

```
    Params  : Prefix length, IP version
    Returns : Binary Mask
```

```
$mask = ip_get_mask ($len,6);
```

ip_last_address_bin

Return the last binary address of a prefix.

```
    Params  : First binary IP, prefix length, IP version
    Returns : Binary IP
```

```
$lastbin = ip_last_address_bin ($ip,$len,6);
```

ip_splitprefix

Split a prefix into IP and prefix length. If it was passed a simple IP, it just returns it.

```
    Params  : Prefix
    Returns : IP, optionally length of prefix
```

```
($ip,$len) = ip_splitprefix ($prefix)
```

ip_prefix_to_range

Get a range of IPs from a prefix.

```
    Params  : Prefix, IP version
    Returns : First IP, last IP
```

```
($ip1,$ip2) = ip_prefix_to_range ($prefix,6);
```

ip_bincomp

Compare binary Ips with <, >, <=, >=.
 Operators are lt(<), le(<=), gt(>), and ge(>=)

```
        Params  : First binary IP, operator, Last binary IP
        Returns : 1 (yes), 0 (no), or undef (problem)
```

```
    ip_bincomp ($ip1,'lt',$ip2) == 1 or do {}
```

ip_binadd

Add two binary IPs.

```
        Params  : First binary IP, Last binary IP
        Returns : Binary sum or undef (problem)
```

```
    $binip = ip_binadd ($bin1,$bin2);
```

ip_get_prefix_length

Get the prefix length for a given range of 2 IPs.

```
        Params  : First binary IP, Last binary IP
        Returns : Length of prefix or undef (problem)
```

```
    $len = ip_get_prefix_length ($ip1,$ip2);
```

ip_range_to_prefix

Return all prefixes between two IPs.

```
        Params  : First IP (binary format), Last IP (binary format), IP version
        Returns : List of Prefixes or undef (problem)
```

The prefixes returned have the form q.q.q.q/nn.

```
    @prefix = ip_range_to_prefix ($ip1,$ip2,6);
```

ip_compress_v4_prefix

Compress an IPv4 Prefix.

```
        Params  : IP, Prefix length
        Returns : Compressed Prefix
```

```
    $ip = ip_compress_v4_prefix ($ip, $len);
```

ip_compress_address

Compress an IPv6 address. Just returns the IP if it is an IPv4.

```
        Params  : IP, IP version
        Returns : Compressed IP or undef (problem)
```

```
    $ip = ip_compress_adress ($ip, $version);
```

ip_is_overlap

Check if two ranges of IPs overlap.

```
        Params  : Four binary IPs (begin of range 1,end1,begin2,end2), IP version
            $IP_PARTIAL_OVERLAP (ranges overlap)
            $IP_NO_OVERLAP      (no overlap)
            $IP_A_IN_B_OVERLAP  (range2 contains range1)
            $IP_B_IN_A_OVERLAP  (range1 contains range2)
            $IP_IDENTICAL       (ranges are identical)
            undef               (problem)
```

```
    (ip_is_overlap($rb1,$re1,$rb2,$re2,4) eq $IP_A_IN_B_OVERLAP) and do {};
```

ip_get_embedded_ipv4

Get an IPv4 embedded in an IPv6 address

```
        Params  : IPv6
        Returns : IPv4 string or undef (not found)
```

```
    $ip4 = ip_get_embedded($ip6);
```

ip_check_mask

Check the validity of a binary IP mask

```
    Params  : Mask
    Returns : 1 or undef (invalid)
```

```
ip_check_mask($binmask) or do {};
```

Checks if mask has only 1s followed by 0s.

ip_aggregate

Aggregate 2 ranges of binary IPs

```
    Params  : 1st range (1st IP, Last IP), last range (1st IP, last IP), IP versi
    Returns : prefix or undef (invalid)
```

```
$prefix = ip_aggregate ($bip1,$eip1,$bip2,$eip2) || die ...
```

ip_iptypev4

Return the type of an IPv4 address.

```
    Params:  binary IP
    Returns: type as of the following table or undef (invalid ip)
```

See RFC 5735 and RFC 6598

AddressBlockPresentUseReference
——
0.0.0.0/8“This”NetworkRFC1122PRIVATE 10.0.0.0/8Private-UseNetworksRFC1918PRIVATE
100.64.0.0/10CGNSharedAddressSpaceRFC6598SHARED 127.0.0.0/8LoopbackRFC1122LOOPBACK
169.254.0.0/16LinkLocalRFC3927LINK-LOCAL 172.16.0.0/12Private-UseNetworksRFC1918PRIVATE
192.0.0.0/24IETFProtocolAssignmentsRFC5736RESERVED 192.0.2.0/24TEST−NET−1RFC5737TEST-NET
192.88.99.0/246to4RelayAnycastRFC30686TO4−RELAY 192.168.0.0/16Private-
UseNetworksRFC1918PRIVATE 198.18.0.0/15NetworkInterconnect
DeviceBenchmarkTestingRFC2544RESERVED 198.51.100.0/24TEST−NET−2RFC5737TEST-NET
203.0.113.0/24TEST−NET−3RFC5737TEST-NET 224.0.0.0/4MulticastRFC3171MULTICAST
240.0.0.0/4ReservedforFutureUseRFC1112RESERVED
255.255.255.255/32LimitedBroadcastRFC919BROADCAST RFC922

ip_iptypev6

Return the type of an IPv6 address.

```
    Params:  binary ip
    Returns: type as of the following table or undef (invalid)
```

See IANA Internet Protocol Version 6 Address Space <http://www.iana.org/assignments/ipv6-address-space/ipv6-address-space.txt> and IANA IPv6 Special Purpose Address Registry <http://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.txt>

PrefixAllocationReference
————————————————————————————————————
0000::/8ReservedbyIETF[RFC4291]RESERVED 0100::/8ReservedbyIETF[RFC4291]RESERVED
0200::/7ReservedbyIETF[RFC4048]RESERVED 0400::/6ReservedbyIETF[RFC4291]RESERVED
0800::/5ReservedbyIETF[RFC4291]RESERVED 1000::/4ReservedbyIETF[RFC4291]RESERVED
2000::/3GlobalUnicast[RFC4291]GLOBAL-UNICAST 4000::/3ReservedbyIETF[RFC4291]RESERVED
6000::/3ReservedbyIETF[RFC4291]RESERVED 8000::/3ReservedbyIETF[RFC4291]RESERVED
A000::/3ReservedbyIETF[RFC4291]RESERVED C000::/3ReservedbyIETF[RFC4291]RESERVED
E000::/4ReservedbyIETF[RFC4291]RESERVED F000::/5ReservedbyIETF[RFC4291]RESERVED
F800::/6ReservedbyIETF[RFC4291]RESERVED FC00::/7UniqueLocalUnicast[RFC4193]UNIQUE-LOCAL-
UNICAST FE00::/9ReservedbyIETF[RFC4291]RESERVED FE80::/10LinkLocalUnicast[RFC4291]LINK-
LOCAL-UNICAST FEC0::/10ReservedbyIETF[RFC3879]RESERVED
FF00::/8Multicast[RFC4291]MULTICAST

PrefixAssignmentReference
———
::1/128LoopbackAddress[RFC4291]UNSPECIFIED ::/128UnspecifiedAddress[RFC4291]LOOPBACK
::FFFF:0:0/96IPv4−mappedAddress[RFC4291]IPV4MAP 0100::/64Discard-OnlyPrefix[RFC6666]DISCARD
2001:0000::/32TEREDO[RFC4380]TEREDO 2001:0002::/48BMWG[RFC5180]BMWG
2001:db8::/32DocumentationPrefix[RFC3849]DOCUMENTATION 2001:10::/28ORCHID[RFC4843]ORCHID
2002::/166to4[RFC3056]6TO4 FC00::/7Unique-Local[RFC4193]UNIQUE-LOCAL-UNICAST
FE80::/10Linked-ScopedUnicast[RFC4291]LINK-LOCAL-UNICAST
FF00::/8Multicast[RFC4291]MULTICAST

ip_iptype

Return the type of an IP (Public, Private, Reserved)

```
    Params  : Binary IP to test, IP version (defaults to 6)
    Returns : type (see ip_iptypev4 and ip_iptypev6 for details) or undef (invali
```

```
$type = ip_iptype ($ip);
```

ip_check_prefix

Check the validity of a prefix

```
    Params  : binary IP, length of prefix, IP version
    Returns : 1 or undef (invalid)
```

Checks if the variant part of a prefix only has 0s, and the length is correct.

```
ip_check_prefix ($ip,$len,$ipv) or do {};
```

ip_reverse

Get a reverse name from a prefix

```
    Params  : IP, length of prefix, IP version
    Returns : Reverse name or undef (error)
```

```
$reverse = ip_reverse ($ip);
```

ip_normalize

Normalize data to a range/prefix of IP addresses

```
    Params  : Data String (Single IP, Range, Prefix)
    Returns : ip1, ip2 (if range/prefix) or undef (error)
```

```
($ip1,$ip2) = ip_normalize ($data);
```

ip_auth

Return IP authority information from the IP::Authority module

```
    Params  : IP, version
    Returns : Auth info (RI for RIPE, AR for ARIN, etc)
```

```
$auth = ip_auth ($ip,4);
```

Note: IPv4 only

BUGS

The Math::BigInt library is needed for functions that use integers. These are ip_inttobin, ip_bintoint, and the size method. In a next version, Math::BigInt will become optional.

AUTHORS

Manuel Valente <manuel.valente@gmail.com>.

Original IPv4 code by Monica Cortes Sack <mcortes@ripe.net>.

Original IPv6 code by Lee Wilmot <lee@ripe.net>.

BASED ON

ipv4pack.pm, iplib.pm, iplibncc.pm.

SEE ALSO

perl(1), IP::Authority