lua_iptable reference

hertogp

Language bindings for iptable.c, which wraps radix.c

`lua_iptable.h`

#defines

`LUA_IPTABLE_ID`

Identity for the `table_t`-userdata.

`LUA_IPT_ITR_GC`

Identity for the `itr_gc_t`-userdata.

LIPTE errno's

0. LIPTE_NONE none
1. LIPTE_AF wrong or unknown address family
2. LIPTE_ARG wrong type of argument
3. LIPTE_BIN illegal binary key/mask
4. LIPTE_BINOP binary operation failed
5. LIPTE_BUF could not allocate memory
6. LIPTE_FAIL unspecified error
7. LIPTE_ITER internal iteration error
8. LIPTE_LIDX invalid Lua stack index
9. LIPTE_LVAL invalid Lua stack (up)value
10. LIPTE_MLEN invalid mask length
11. LIPTE_PFX invalid prefix string
12. LIPTE_RDX unhandled radix node type
13. LIPTE_SPLIT prefix already at max length
14. LIPTE_TOBIN error converting string to binary
15. LIPTE_TOSTR error converting binary to string

16. LIPTE_UNKNOWN unknown error number
17. LIPTE_ZMAX NULL (end of error string list)

lua_iptable.c

Lua bindings for iptable

Structures

itr_gc_t

The `itr_gc_t` type has 1 member: `table_t *t` and serves only to create a userdata to be supplied as an upvalue to each iterator function. That userdata has a metatable with a `__gc` garbage collector, (see `ipt_itr_gc`) which will remove radix nodes in `t` that are (still) flagged for deletion only when it is safe to do so (i.e. `t->itr_lock` has reached zero).

uint8_t IP4_MASK8[] = { 5, 255, 0, 0, 0}; */ *uint8_t IP4_MASK12[] = { 5, 255, 240, 0, 0};* / uint8_t IP4_MASK16[] = { 5, 255, 255, 0, 0}; */

Helper functions

_stridx

```
static int _str2idx(const char *, const char * const []);
```

Find the index of a string in a NULL-terminated list of strings. Used by functions like iptL_getaf, that take strings arguments that need to be converted to an index or number. Returns index on success, -1 on failure.

Lua library

luaopen_iptable

```
int luaopen_iptable(lua_State *);
```

```
-- lua
iptable = require "iptable"
```

Called by Lua upon requiring "iptable" to initialize the module. Returns `iptable`'s module-table with the module functions and constants.

error handlers

In general, functions that return values will return all nil's for these in case of an error and an extra string value describing the error is added to the values returned. In addition to this, an `iptable.error` is also set to indicate the error.

```
iptable.address("10.10.10.10/24")
--> 10.10.10.10  24  2, for ip, mlen and af. Whereas
iptable.address("10.10.10.10/33")
--> nil nil nil "invalid prefix string"

for host in iptable.hosts("10.10.10.0/33") do print(host) end
--> won't print anything, here's why:
iptable.error
--> stdin:1:12:error converting string to binary ('10.10.10.0/33' ?)
```

lipt_vferror

```
static int lipt_vferror(lua_State *L, int, int, const char *, va_list);
```

Helper function that clears the stack, sets iptable.error module level variable to a formatted error string `file:line:errno:description` and finally loads the stack with as many nils as arguments that would've been returned plus the `description` for the errno supplied. In Lua space, the user has access to both `iptable.error` as well as the last value returned by the function that errored out.

lipt_error

```
static int lipt_error(lua_State *L, int, int, const char *);
```

```
-- lua
iptable.error = nil -- clear any previous error
ip, mlen, af, err = iptable.address("10.10.10.0/33")
--> nil nil nil invalid prefix string
iptable.error
--> "file:line:errno:invalid prefix string"
```

A helper function to set iptable's global error number, clear the stack, push nil, and return 1. Is used by functions to bail out while setting an error indicator. When an interation is not possible due to some error, `iptable.error` is the only way to see what went wrong.

Lua stack functions

These are convencience functions to get/set values and parameters on the Lua stack. The include:

- `iptL`-functions get/set/push stack values.
- `iptT`-functions manipulate k,v-pairs of a table on top of L

iptL_gettable

```
static table_t * iptL_gettable(lua_State *, int);
```

Checks whether the stack value at the given index contains a userdata of type `LUA_IPTABLE_ID` and returns a `table_t` pointer. Errors out to Lua if the stack value has the wrong type.

iptL_getaf

```
static int iptL_getaf(lua_State *L, int idx, int *af);
```

Checks whether the stack value at the given index is a known af_family number or name and sets *af accordingly, otherwise it's set to `AF_UNSPEC`. If the stack value does not exist, the int pointer `*af` is not modified. Caller decides if a missing or unknown af_family is an error or not.

iptL_getbinkey

```
static int iptL_getbinkey(lua_State *L, int idx, uint8_t *buf, size_t * len);
```

Copy a binary key, either an address or a mask, at given 'idx' into given 'buf'. 'buf' size is assumed to be MAX_BINKEY. A binary key is a byte array [LEN | key bytes] and LEN is total length. However, radix masks may have set their LEN-byte equal to the nr of non-zero bytes in the array instead.

iptL_getpfxstr

```
static in iptL_getpfxstr(lua_State *L, int idx, const char **pfx, size_t *len);
```

Checks if `L[idx]` is a string and sets the const char ptr to it or NULL. Also sets len to the string length reported by Lua's stack manager. Lua owns the memory pointed to by the const char ptr (no free by caller needed). Returns 1 on success, 0 on failure.

iptL_refpcreate

```
static int iptL_refpcreate(lua_State *L);
```

Create a reference id for the lua_State given. This allocates an int pointer, gets a new ref_id, stores it in the allocated space and returns a pointer to it.

iptL_refpdelete

```
static void iptL_refpdelete(void *L, void **r);
```

Delete a value from `LUA_REGISTRYINDEX` indexed by **`r` (which is treated as an** int). Function signature is as per `purge_f_t` (see iptable.h) and acts as the table's purge function to release user data.

iptT_setbool

```
static void iptT_setbool(lua_State *L, const char *k, int v);
```

Needs a Lua table on top of the stack and sets key `k` to boolean for `v`.

iptT_setfstr

```
static void iptT_setfstr(lua_State *L, const char *k, const char *fmt,
const void *v)
```

Needs a Lua table on top of the stack and sets key `k` to the formatted string using `fmt` and `v`.

iptT_setint

```
static void iptT_setint(lua_State *L, const char *k, int v);
```

Needs a Lua table on top of the stack and sets key `k` to integer value `v`.

iptT_setkv

```
static void iptT_setkv(lua_State *L, struct radix_node *rn);
```

Needs a Lua table on top of the stack. It takes the radix node's key `rn->rn_key` and the value pointed to by the radix_node and sets that as a key,value-pair in the table on top.

The `rn` pointer actually points to the start of the user data structures built into the radix tree. Hence a cast to `(entry_t *)rn` gives access to the `rn`'s index

number, which is used to retrieve the prefix's (aka `rn->key`) value from the `LUA_REGISTRYINDEX`.

iptL_pushrnh

```
static int iptL_pushrnh(lua_State *L, struct radix_node_head *rnh)
```

Encodes a `radix_node_head` structure as a Lua table and adds some additional members to make processing on the Lua side easier. It will include nodes flagged for deletion with `IPTF_DELETE`, so the user can decide for themselves to ignore them or not. The radix_node_head table looks like:

```
{ _NAME_  = "RADIX_NODE_HEAD",
  _MEM_   =  "0x<rnh>",
  rh = { RADIX_HEAD },
  rnh_nodes = {
    _NAME_ = "RNH_NODES[3]",
    _MEM_  = "0x<rh>",
    [1]    = { RADIX_NODE },
    [2]    = { RADIX_NODE },
    [3]    = { RADIX_NODE }
   }
}
```

iptL_pushrn

```
static int iptL_pushrn(lua_State *L, struct radix_node *rn)
```

Encodes a `radix_node` structure as a Lua table and adds some additional members to make processing on the Lua side easier.

```
{ _NAME_     = "RADIX_NODE",
  _MEM_      = "0x<rn-ptr>",
  _NORMAL_   = 1, -- if NORMAL flag is set
  _ROOT_     = 1, -- if ROOT flag is set
  _ACTIVE_   = 1, -- if ACTIVE flag is set
  _DELETE_   = 1, -- if IPTF_DELETE flag is set (i.e. a deleted node)
  _LEAF_     = 1, -- if it is a LEAF node
  _INTERNAL_ = 1, -- if it is not a LEAF node

  rn_mklist  = "0x<ptr>",
  rn_parent  = "0x<ptr>",
  rn_bit     = rn->rn_bit,
  rn_bmask   = rn->rn_bmask,
  rn_flags   = rn->rn_flags,
```

```
  -- LEAF NODES only
  rn_key     = "prefix as a string"
  rn_mask    = "mask as a string"
  rn_dupedkey = "0x<ptr>",
  _rn_key_LEN = key-length, -- in bytes, may be -1
  _rn_mask_len = key-length, -- in bytes, may be -1
  _rn_mlen   = num-of-consecutive msb 1-bits,

  -- INTERNAL NODES only
  rn_offset  = rn->offset,
  rn_left    = rn->left,
  rn_right   = rn->right,
}
```

iptL_pushrh

```
static int iptL_pushrh(lua_State *L, struct radix_head *rh)
```

Encodes a `radix_head` structure as a Lua table and adds some additional members to make processing on the Lua side easier.

```
{ _NAME_      = "RADIX_HEAD",
  _MEM_       = "0x<ptr>",
  rnh_treetop = "0x<ptr>",
  rnh_masks   = "0x<ptr>",
}
```

iptL_pushrmh

```
static int iptL_pushrmh(lua_State *L, struct radix_mask_head *rmh)
```

Encodes a `radix_head` structure as a Lua table and adds some additional members to make processing on the Lua side easier.

```
{ _MEM_  = "0x<ptr>",
  _NAME_ = "RADIX_MASK_HEAD",
  head   = "0x<ptr>",
  mask_nodes = {
    _MEM_  = "0x<ptr>",
    _NAME_ = "MASK_NODES[3]",
    [1]    = { RADIX_NODE },
    [2]    = { RADIX_NODE },
    [3]    = { RADIX_NODE }
  }
}
```

iptL_pushrm

```
static int iptL_pushrm(lua_State *L, struct radix_mask *rm);
```

Encodes a `radix_mask` structure as a Lua table and adds some additional members to make processing on the Lua side easier.

```
{ _NAME_ = "RADIX_MASK",
  _MEM_  = "0x<ptr>",
  _NORMAL_ = 1, -- if NORMAL-flag is set
  _ROOT_   = 1, -- if ROOT-flag is set
  _ACTIVE_ = 1, -- if ACTIVE-flag is set
  _LEAF_   = 1, -- if it's a LEAF node
  _NORMAL_ = 1, -- if it's not a LEAF node
  rm_bit = rm->rm_bit,
  rm_unused = rm->rm_unused,
  rm_flags  = rm->rm_flags,

  -- LEAF NODE only
  rm_leaf   = "0x<ptr>"

  -- INTERNAL NODE only
  rm_mask   = "0x<ptr>"
```

Garbage collection

iptm_gc

```
static int iptm_gc(lua_State *L);
```

Garbage collector function (`__gc`) for `LUA_IPTABLE_ID` metatable, which is the metatable of `iptable`-table. Once a table instance is garbage collected this function gets called to fee up all its resources.

iptL_pushitrgc

```
static void iptL_pushitrgc(lua_State *L, table_t *t);
```

Each iterator factory function MUST call this function in order to push a new userdata (an iterator guard) as an upvalue for its actual iterator function. Once that iterator is done, Lua will garbage collect this userdata (the iterator's upvalues are being garbage collected, includeing this iterator guard userdata). The sole purpose of this iterator guard userdata is to check for radix nodes flagged for deletion and actually delete them, unless there are still iterators active for this table. So with nested iterators, only the last userdata gets to actually delete radix nodes from the tree. Note, the actual iterator functions donot use

this userdata at all, its only there to get garbage collected at some point after the iterator function is finished and its upvalues are garbage collected.

Hence, deletions (in Lua) during tree iteration are safe, since deletion operations also check the table for any active iterators. If there are none, deletion is immediate. Otherwise, the node(s) only get flagged for deletion which makes them 'inactive'. All tree operations treat radix nodes thus flagged as not being there at all. Although inactive, they can still be used to navigate around the tree by iterators.

This function:

- creates a new userdatum
- sets its metatable to the `LUA_IPT_ITR_GC` metatable
- points this userdata to this table (`t`-member)
- increments this table active iterator count `itr_lock`

and leaves it on the top of the stack so it can be closed over by the iteration (closure) function when created by its factory.

`ipt_itr_gc`

```
static int ipt_itr_gc(lua_State *L);
```

The garbage collector function of the LUA_IPT_ITR_GC metatable, used on the iterator guard userdata pushed as an upvalue to all table iterator functions. Once an iterator is finished this userdata, since it is an upvalue of the iterator function, is garbage collected and this function gets called, which will:

- decrease this table's active iterator count
- if it reaches zero, it'll delete all radix nodes flagged for deletion

Iterators

The functions in this section are two helper functions and the actual iterator functions used by the iterator factory functions to setup some form of iteration. These are collected here whereas the factory functions are listed in both the `modules functions` and `instance methods` sections since the relate to functions callable from Lua. Since the actual iterator functions donot, they are listed here.

`iter_error`

```
static int iter_error(lua_State *L, int errno, const char *fmt, ...);
```

Helper function to bail out of an iteration factory function. An iteration factory function MUST return a valid iterator function (`iter_f`) and optionally an

`invariant` and `ctl_var`. By returning `iter_fail_f` as `iter_f` we ensure the iteration in question will yield no results in Lua.

iter_fail_f

```
static int iter_fail_f(lua_State *L);
```

An iteration function that immediately terminates any iteration.

iter_hosts_f

```
static int iter_hosts_f(lua_State *L);
```

The actual iterator function for iptable.hosts(pfx), yields the next host ip until its stop value is reached. Ignores the stack: it uses upvalues for next, stop

iter_interval_f

```
static int iter_interval_f(lua_State *L)
```

The actual iterator function for `iter_interval`. It calculates the prefixes that, together make up the address space interval, by starting out with the maximum size possible for given start address and stop address. Usually ends with smaller prefixes. On errors, it won't iterate anything in which case `iptable.error` should provide some information.

iter_subnets_f

```
static int iter_subnets_f(lua_State *L)
```

The actual iterator function for `iter_subnets`. Uses upvalues: `start`, `stop`, `mask`, `mlen` and a `sentinal` which is used to signal address space wrap around.

`stop` represents the broadcast address of the last prefix to return. This might actually be the max address possible in the AF's address space and increasing beyond using `key_incr` would fail. So if the current prefix, which will be returned in this iteration, has a broadcast address equal to `stop` the sentinal upvalue is set to 0 so iteration stops next time around and no `key_incr` is done to arrive at the next 'network'-address of the next prefix since we're done already..

`mlen` is stored as a convenience and represents the prefix length of the binary `mask` and alleviates the need to calculate it on every iteration.

iter_kv_f

```
static int iter_kv_f(lua_State *L)
```

The actual iteration function for `iter_kv` yields all k,v pairs in the tree(s). When an iterator is active, deletion of entries are flagged and only carried out when no iterators are active. So it should be safe to delete entries while iterating.

Note: upvalue(1) is the leaf to process.

iter_more_f

```
static int iter_more_f(lua_State *L);
```

The actual iteration function for `iter_more`.

- `top` points to subtree where possible matches are located
- `rn` is the current leaf under consideration

iter_less_f

```
static int iter_less_f(lua_State *L);
```

The actual iteration function for `iter_less`. It basically works by decreasing the prefix length which needs to match for the given search prefix.

iter_masks_f

```
static int iter_masks_f(lua_State *L);
```

The actual iteration function for `iter_masks`. Masks are read-only: the Lua bindings donot (need to) interact directly with a radix mask tree.

iter_supernets_f

```
static int iter_supernets_f(lua_State *L);
```

The actual iteration function for `iter_supernets`.

The next node stored as upvalue before returning, must be the first node of the next group.

iter_radix

```
static int iter_radix(lua_State *L);
```

The actual iterator function for `iter_radixes` which traverses a prefix-tree (and possible its associated mask-tree) and yields all nodes one at the time.

Notes:

- returns current node as a Lua table or nil to stop iteration
- saves next (type, node) as upvalues (1) resp. (2)

module functions

iptable.new

```
static int ipt_new(lua_State *L);
```

Creates a new userdata, sets its `iptable` metatable and returns it to Lua. It also sets the purge function for the table to *`iptL_refpdelete`* which frees any memory held by the user's data once a prefix is deleted from the radix tree.

iptable.tobin

```
static int iptable.tobin(lua_State *L);
```

```
-- lua
 binkey, mlen, af, err = iptable.tobin("10.10.10.10/24")
```

Convert the prefix string to a binary key and return it, together with its prefix length & AF for a given prefix. Note: a byte array is length encoded [LEN | key-bytes], where `LEN` is the total length of the byte array. In case of errors, returns nil's and an error description.

iptable.tostr

```
static int ipt_tostr(lua_state *L);
```

```
-- lua
pfx, err = iptable.tostr(binkey)
```

Returns a string representation for given binary key or nil & an error msg.

iptable.toredo

```
static int ipt_tostr(lua_state *L);
```

```
-- lua
t, err = iptable.toredo(ipv6)
t, err = iptable.toredo(tserver, tclient, udp, flags)
```

Either compose an ipv6 address using 4 arguments or decompose an ipv6 string into a table with fields: `ipv6`, the toredo ip6 address `server`, toredo server's ipv4 address `client`, toredo client's ipv4 address `flags`, flags `udp`, udp port

iptable.masklen

```
static int ipt_masklen(lua_State *L);
```

```
-- lua
mlen, err = iptable.masklen(binkey)
```

Return the number of consecutive msb 1-bits, nil & error msg on errors. Note: stops at the first zero bit, without checking remaining bits since non-contiguous masks are not supported.

iptable.size

```
static int ipt_size(lua_State *L);
```

```
-- lua
num, err = iptable.size("10.10.10.0/24") -- 256.0
```

Returns the number of hosts in a given prefix, nil & an error msg on errors. Note: uses Lua's arithmatic (2^hostbits), since ipv6 can get large, so numbers show up as floating points.

iptable.address

```
static int ipt_address(lua_State *L);
```

```
-- lua
addr, mlen, af, err = iptable.address("10.10.10.10/24")
```

Return host address, masklen and af_family for pfx; nil's & an error msg on errors.

iptable.dnsptr

```
static int ipt_dnsptr(lua_State *L);
```

```
-- lua
ptr, mlen, af, err = iptable.dnsptr("10.10.10.10/24")
--> 10.10.10.10.in-addr.arpa.
```

```
ptr, mlen, af, err = iptable.dnsptr("10.10.10.10/24", true)
--> 10.10.10.in-addr.arpa.

ptr, mlen, af = iptable.dnsptr("acdc:1979:/32")
--> 0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.9.7.9.1.c.d.c.a.ip6.arpa.
--> 32 10 nil
ptr, mlen, af = iptable.dnsptr("acdc:1979:/32", true)
--> 9.7.9.1.c.d.c.a.ip6.arpa. 32 10
```

Return a reverse dns name for the address part of the prefix, along with its mask length and address family. If the optional 2nd argument is true and the prefix has a mask larger than zero bits, skip as many leading labels as the mask allows. Note: this works on a byte resp. nibble boundary since as represented by the dns labels for ipv4 resp. ipv6 reverse names. A prefix length of zero would be non-sensical since at least 1 label needs to be present left of in-addr.arpa or ip6.arpa. In case of errors, returns all nils plus an error message.

iptable.longhand

```
static int ipt_longhand(lua_State *L);

-- lua
addr, mlen, af, err = iptable.longhand("2001::/120")
--> 2001:0000:0000:0000:0000:0000:0000:0000  120  10
```

Return unabbreviated address, masklen and af_family for pfx; nil's & an error msg on errors. Only has a real effect on ipv6 prefixes.

iptable.network

```
static int ipt_network(lua_State *L);

-- lua
netw, mlen, af, err = iptable.network("10.10.10.10/24")
```

Return network address, masklen and af_family for pfx; nil's & an error msg on errors.

iptable.neighbor

```
static int ipt_neighbor(lua_State *L);

-- lua
nei, mlen, af, err = iptable.neighbor("10.10.10.10/24")
--> 10.10.11.0  24  2
```

Given a prefix, return the neighbor prefix, masklen and af_family, such that both can be combined into a supernet whose prefix length is 1 bit shorter. Note that a /0 will not have a neighbor prefix whith which it could be combined. Returns nil's and an error msg on errors.

iptable.invert

```
static int ipt_invert(lua_State *L);
```

```
-- lua
inv, mlen, af, err = iptable.invert("255.255.255.0")
--> 0.255.255.255  -1  2
```

Return inverted address, masklen and af_family. Returns nil's and an error message on errors. Note: the mask is NOT applied before invering the address.

iptable.properties

```
static int ipt_properties(lua_State *L);
```

```
-- lua
props, err = iptable.properties("192.168.1.1/24")
--> { address="192.168.1.1",
      masklength = 24,
      private = true,
      size = 256
      class = C
    }
```

Returns a table with (some) properties for given prefix. Returns nil's and an error message on errors.

iptable.offset

```
static int ipt_offset(lua_State *L);
```

```
-- lua
ip, mlen, af, err = iptable.offset("10.10.10.0/24", 1)
--> 10.10.10.1  24  2
ip, mlen, af, err = iptable.offset("10.10.10.0/24", -1)
--> 10.10.9.255 24  2
```

Return a new ip address, masklen and af_family by adding an offset to a prefix. If no offset is given, increments the key by 1. Returns nil and an error msg on errors such as trying to offset beyond the AF's valid address space (i.e. it won't wrap around). Note: any mask is ignored during offsetting.

iptable.reverse

```
static int ipt_reverse(lua_State *L);
```

```
-- lua
rev, mlen, af, err = iptable.reverse("1.2.3.4/24")
-> 4.3.2.1  24  2
```

Return a reversed address, masklen and af_family. Returns nil's and an error msg on errors. Note: the mask is NOT applied before reversing the address.

iptable.split

```
static int ipt_split(lua_State *L);
```

```
-- lua
addr1, addr2, mlen, af, err = iptable.split("10.10.10.10/24")
--> 10.10.10.0  10.10.10.128  25  2  nil
```

Split a prefix into its two constituent parts. Returns nils on errors plus an error message.

iptable.broadcast

```
static int ipt_broadcast(lua_State *L);
```

```
-- lua
bcast, mlen, af, err = iptable.broadcast("10.10.10.10/24")
--> 10.10.10.255  24  2
```

Return broadcast address, masklen and af_family for pfx; nil's and an error message on errors.

iptable.mask

```
static int ipt_mask(lua_State *L);
```

```
-- lua
mask = iptable.mask(iptable.AF_INET, 30)
--> 255.255.255.252
mask = iptable.mask(iptable.AF_INET, 30, true)
--> 0.0.0.3
```

Given an address family `af` and prefix length `mlen`, returns the mask or or nil and an error message on errors. If the third argument evaluates to true, also inverts the mask.

iptable.hosts

```
static int iter_hosts(lua_State *L);
```

```
-- lua
for host in iptable.hosts("10.10.10.0/30") do print(host) end
--> 10.10.10.1
--> 10.10.10.2
for host in iptable.hosts("10.10.10.0/30", true) do print(host) end
--> 10.10.10.0
--> 10.10.10.1
--> 10.10.10.2
--> 10.10.10.4
```

Iterate across host adresses in a given prefix. An optional second argument defaults to false, but when given & true, the network and broadcast address will be included in the iteration results. If prefix has no mask, the af's max mask is used. In which case, unless the second argument is true, it won't iterate anything. In case of errors (it won't iterate), check out `iptable.error` for clues.

iptable.interval

```
static int ipt_interval(lua_State *L);
```

```
-- lua
  for subnet in iptable.interval("10.10.10.0", "10.10.10.9") do
    print(subnet)
  end
  --> 10.10.10.0/29
  --> 10.10.10.8/31
```

Iterate across the prefixes that, combined, cover the address space between the `start` & `stop` addresses given (inclusive). Any masks in either `start` or `stop` are ignored and both need to belong to the same AF_family. In case of errors it won't iterate anything, in that case `iptable.error` will show the last error seen.

iptable.subnets

```
static int ipt_subnets(lua_State *L);
```

```
-- lua
  for subnet in iptable.subnets("10.10.10.0/24", 26) do
    print(subnet)
  end
  --> 10.10.10.0/26
  --> 10.10.10.64/26
```

```
--> 10.10.10.128/26
--> 10.10.10.192/26
```

Iterate across the smaller prefixes given a start prefix and a larger network mask, which is optional and defaults to being 1 bit longer than that of the given prefix (similar to split). In case of errors, it won't iterate anything in which case `iptable.error` might shed some light on the error encountered.

instance methods

iptm_newindex

```
static int iptm_newindex(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
ipt["acdc:1979::"] = "Jailbreak"
```

Implements `t[k] = v`

If v is nil, t[k] is deleted. Otherwise, v is stored in the lua_registry and its ref-value is stored in the radix tree. If k is not a valid ipv4/6 prefix, cleanup and ignore the request.

iptm_index

```
static int iptm_index(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
ipt["10.10.10.0/25"] = "lower half"
ipt["10.10.10.1"]    --> lower half
ipt["10.1.10.128"]   --> nil
ipt["10.10.10.0/24"] --> nil
```

Given an index `k`:

- do longest prefix search if `k` is a prefix without mask,
- do an exact match if `k` is a prefix with a mask
- otherwise, do metatable lookup for property named by `k`.

iptm_len

```
static int iptm_len(lua_State *L);
```

```
-- lua
t = require"iptable".new()
```

```
t["1.2.3.4/24"] = "boring"
t["acdc:1979::/120"] = "touch too much"
#t --> 2
```

Return the sum of the number of entries in the ipv4 and ipv6 radix trees as the 'length' of the table.

iptm_tostring

```
static int iptm_tostring(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
ipt -- iptable{#ipv4=0, #ipv6=0}  --- an empty ip table
```

Return a string representation of the iptable instance the respective ipv4 and ipv6 counters. Note that the ipv6 counter may overflow at some point...

iptm_counts

```
static int iptm_counts(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
ip4c, ip6c = ipt:counts()
--> 0  0
```

Return the number of entries in both the ipv4 and ipv6 radix tree.

iter_kv

```
static int iter_kv(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
ipt["1.1.1.1"] = 1
ipt["2001:abab::"] = 2

for k,v in pairs(ipt) do print(k,v) end
--> 1.1.1.1       1
--> 2001:abab::   2
```

Iterate across key,value-pairs in the table. The first ipv4 or ipv6 leaf node is pushed. The `iter_kv_f` iterator uses nextleaf to go through all leafs of the tree. It will traverse ipv6 as well if we started with the ipv4 tree first. If an errors occurs, it won't iterate anything and `iptable.error` should provide some information.

iter_more

```
static int iter_more(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
for prefix in ipt:more("10.10.10.0/24") do ... end
for prefix in ipt:more("10.10.10.0/24", true) do ... end
```

Iterate across more specific prefixes. AF_family is deduced from the prefix given. Note this traverses the entire tree if pfx has a /0 mask.

The second optional argument may be used to include the search prefix in the search results should it exist in tree.

iter_less

```
static int iter_less(lua_State *L);
```

```
-- lua
ipt = require"iptable".new()
for prefix in ipt:less("10.10.10.10/26") do ... end
for prefix in ipt:less("10.10.10.10/26", true) do ... end
```

Iterate across prefixes in the tree that are less specific than pfx. The optional second argumnet, when true, causes the search prefix to be included in the search results should it be present in the table itself.

iter_masks

```
static int iter_masks(lua_State *L);
```

```
-- lua
iptable = require"iptable"
ipt = iptable.new()
for mask in ipt:masks(iptable.AF_INET4) do ... end
for mask in ipt:masks(iptable.AF_INET6) do ... end
```

Iterate across all masks used in the given `af`-family's radix tree. Notes:

- a mask tree stores masks in rn_key fields.
- a mask's KEYLEN is the nr of non-zero mask bytes, not a real LEN byte
- a /0 (zeromask) is never stored in the radix mask tree (!).
- a /0 (zeromask) is stored in ROOT(0.0.0.0)'s *last* dupedkey-chain leaf.
- the AF_family cannot be deduced from the mask in rn_key.

So, the iteration function needs the AF to properly format masks for the given family (an ipv6/24 mask would otherwise come out as 255.255.255.0) and an explicit flag needs to be passed to iter_f that a zeromask entry is present.

iter_supernets

```
static int iter_supernets(lua_State *L);
```

```
-- lua
iptable = require"iptable"
ipt = iptable.new()
for super, group in ipt:supernets(iptable.AF_INET) do ... end
for super, group in ipt:supernets(iptable.AF_INET6) do ... end
```

Iterate across pairs of pfx's that may be combined into their parental supernet whose prefix length is 1 less than that of its subnets. The iterator returns the supernet prefix (string) and a regular Lua table that contains either 2 of 3 prefix,value-pairs. In case the supernet itself is also present in the table, it's included the group table as well.

iter_radixes

```
static int iter_radixes(lua_State *L);
```

```
-- lua
iptable = require"iptable"
ipt = iptable.new()
for rdx in ipt:radixes(iptable.AF_INET) do ... end
for rdx in ipt:radixes(iptable.AF_INET6) do ... end
```

Iterate across all nodes of all types in the given af's radix tree. By default it will only iterate across the prefix-tree. To include the radix nodes of the associated mask-tree include a second argument that evaluates to `true`.

The iterator will return the C-structs as Lua tables and is used to graph the radix trees. Useful for debugging iptable's code and for getting the hang of the radix structures built by *FreeBSD*'s `radix.c` code.