# PyRIC v0.0.5: User Manual

Dale V. Patterson
wraith.wireless@yandex.com

May 8, 2016

## Contents

# 1 About PyRIC

PyRIC is a python port of (a subset of) iw and by extension a python port of Netlink[3] (w.r.t nl80211 functions). The goal of PyRIc is to provide a simple interface to the underlying nl80211 kernel support that handles the complex operations of Netlink seamlessy while maintaining a minimum of "code walking" to understand, modify and extend. PyRIC will only work on Linux with Netlink support.

While users can utilize libnl(.py) and libio(.py) to communicate directly with the kernel, the true utility of PyRIC is pyw(.py). Like iw, pyw provides an interface/buffer between the caller and the kernel, handling all message construction, parsing and transfer transparently and without requiring any Netlink knowledge or experience.

At this time, PyRIC (through pyw functions) can:

- enumerate interfaces and wireless interfaces,
- get/set hw address,
- identify a radio's chipset and driver,
- turn device on/off,
- get supported standards,
- get/set regulatory domain,
- get info on device,
- add/delete interfaces.

It also provides users with the ability, through libnl(.py) to extend the above functionality by creating additional commands.

Currently, PyRIC does handle multicast messages i.e. events or dumps although plans for future versions include adding the ability to process dumps.

## 1.1 Naming Conventions

The terms interface, device and radio are all used interchangeably throughout to refer to a network interface card (NIC) wireless or Ethernet. The following terms will always have one one meaning:

- **dev** - the device name i.e. wlan0 or eth0 of a NIC,
- **phy** - the physical index of a NIC i.e. the 0 in phy0,
- **ifindex** - the interface index of a NIC,
- **card** or **Card** - a NIC abstraction, an object used in pyw functions see the following section for a description.

## 1.2 Cards

A Card is merely a wrapper around a tuple t = (phy index,device name,ifindex). Since the underlying Netlink calls sometimes require the physical index, sometimes the device name, and sometimes the ifindex, pyw functions[1] take a Card object which doesn't require callers to know which identifier to use for each function. There are four primary methods to creating a Card:

1. **pyw.getcard** returns a Card object from a given dev,
2. **pyw.devinfo** returns the dict info where info['card'] is the Card object. This function will take either a card or a dev
3. **pyw.devadd** returns a new Card object,
4. **pyw.ifaces** returns a list of tuples t = (Card,mode) sharing the same phy as a given dev.

A side affect of using Cards is that many of the netlink calls require the ifindex. The ifindex is found through the use of ioctl, meaning two sockets have to be created and two messages have to be sent, received and parsed in order to execute the command. With Cards, the ifindex is requested for only once.

Keep in mind that any identifier (phy,dev,ifindex) can be invalidated outside of your control. Another program can rename your interface, that is change the dev without your knowledge. Depending on what functions are being used this may not be noticed right away as the phy will remain the same. Also, usb devices (if the usb is disconnected and reconnected) will have the same dev but the phy and ifindex will be different.

## 1.3 Hierarchy/Architecture

PyRIC's hierarchy is briefly discussed next.

1. **__init__.py**: Initialize PyRIC, defines the EUNDEF error code (PyRIC uses errno for all errorcodes adding EUNDEF) and PyRIC's common exception class, 'error' - all submodules use this class for any exceptions.
2. **README.md**: self-descriptive
3. **TODO**: lists any outstanding TODOs, ideas for future revisions
4. **RFI**: comments and observations about netlinks, nl80211 etc
5. **channels.py**: ISM and UNII frequencies and channels, with conversion functions
6. **device.py**: device and chipset utility functions
7. **pyw.py**: this is the interface, providing wireless interface manipulation functions
8. **lib**: lib subpackage
   (a) **__init__.py**: initialize lib subpackage
   (b) **libnl.py**: netlink API

[1] Not all functions accept a Card, the function pyw.devinfo() will accept either a Card or a dev and pyw.isinterface() only accepts a dev

(c) **libio.py**: ioctl API

9. **net**: net subpackage
   (a) **__init__.py**: initialize net subpackage
   (b) **if_h.py**: inet/ifreq definition
   (c) **sockios_h.py**: socket-level I/O control call flags
   (d) **netlink_h.py**: port of netlink.h
   (e) **genetlink_h.py**: port of genetlink.h
   (f) **policy.py**: defines attribute datatypes
   (g) **wireless**: wireless subpackage
      i. **__init__.py**: initialize lib subpackage
      ii. **nl80211_h.py**: port of nl80211 (commands & attributes)
      iii. **nl80211_c.py**: nl80211 attribute datatypes/policies
10. **docs**: docs subpackage
11. **__init__.py**: initialize docs subpackage
12. **nlhelp.py**: functions to search display nl80211 constants
13. **commands.help**: nl80211 commands help data (json)
14. **attributes.help**: nl80211 attributes help data (json)
15. **PyRIC.pdf**: this file
16. **res**:resource subpackage
   (a) **PyRIC.tex**: LaTeX for user guide
   (b) **PyRIC.bib**: bibliography for user guide

## 2 Installing PyRIC

PyRIC is self-sufficient in that there are no third-party/external dependencies and can be run out of the box. To install, from a terminal type 'pip install –pre PyRIC' as root. This is my first attempt at packaging and it has not been currently tested. The latest PyRIC release can be downloaded as a tarball on PyPI at https://pypi.python.org/pypi?name=PyRIC or github at https://github.com/wraith-wireless/pyric/releases/latest. If you decide to download the tarball and run PyRIC outside of it's directory or from another program, you should put it on your Python path. Use Google for directions on doing so. Note: put /yourpath/pyric/pyric in the .pth file, not the outer pyric directory.

## 3 Using PyRIC

As stated previously, PyRIC provides a set of functions to interact with your system's radio(s) and the ability to interact directly with the kernel through netlink and ioctl sockets.

It is expected that the reader has a basic knowledge of netlinks. For a review see Graf[2].

### 3.1 Interacting with the Wireless Core and Wireless NICs: pyw.py

If you can use iw, you can use pyw. The easist way to explain how to use pyw is with an example. Imagine your wireless network on ch 6 has been experiencing difficulites lately and you want to capture some traffic to analyse it. Listing 1 shows how to set up a wireless pentest environment.

```
 1: import pyric                      # pyric error (and ecode EUNDEF)
 2: from pyric import pyw             # for iw functionality
 3: from pyric import device          # for chipset/driver
 4: from pyric.channels import rf2ch # rf to channel conversion
 5:
 6: # list interfaces, wireless interfaces
 7: print "Interfaces: ", pyw.interfaces()
 8: print "Wireless Interfaces: ", pyw.winterfaces()
 9:
10: # get a Card & info for alfa0 & print a description
11: info = pyw.devinfo('alfa0')
12: card = info['card']
13: driver = device.ifdriver(card.dev)
14: chipset = device.ifchipset(driver)
15: msg = "{Using {0} in mode: {1}".format(card,info['mode'])
16: if info['mode'] == 'managed':
17:     msg += " currently on channel {0} width {1}".format(rf2ch(info['RF']),
18:                                                         info['CHW'])
19: msg += " driver: {0}, chipset: {1}".format(driver,chipset)
20: print msg
21:
22: # prepare a virtual interface named pent0 in monitor mode
23: pdev = 'pent0'
24: pcard = pyw.devadd(card,pdev,'monitor')
25: for iface in pyw.ifaces(card):
26:     if iface[0].dev != pdev:
27:         msg = "deleting {0} in mode {1}".format(iface[0],iface[1])
28:         print msg
29:         pyw.devdel(iface[0])
30: pyw.macset(pcard,'00:03:93:57:54:46')
31: pyw.up(pcard)
32: pyw.chset(pcard,6,None)
33: msg = "Virtual interface {0} in monitor mode on ch 6".format(pcard)
34: print msg + ", using hwaddr: {0}".format(pyw.macget(pcard)
35:
36: # DO stuff here
37:
38: # restore original
39: print "deleting ", pcard
40: pyw.devdel(pcard)
41: card = pyw.devadd(card,card.dev,info['mode'])
42: pyw.up(card)
43: print "card ", card, " restored"
```

Listing 1: Setting up a Wireless Pentest Environment

Listing 1 attempts to show most of the available pyw functions in use and is the basic shell used in another project, Wraith[4], to instantiate a wireless (802.11) sensor. Lines 1 and 2 should always be included as they import the pyric error and pyw functions. Lines 3 and 4 import ifchipset, ifdriver and rf2ch functions. In lines 10 through 20, a Card object for 'alfa0' is created and details about

the interface are printed. On line 24, a device named 'pent0' is created in monitor mode and in lines 25 through 29, all interfaces on the same phy as the new device are deleted (we have found that it is better to delete all interfaces on the same phy ensuring that processes like NetworkManager for example, don't interfere with the new device). Then, the mac address is spoofed and the device is set to listen on channel 6 NOHT. Starting on line 40, the system is restored by deleting the monitor card and restoring the original device, 'alfa0'. Note, only 'alfa0' is restored, and not any devices that may have been deleted on line 29. Additionally, bringing the device up (line 43) will usually result in the device reconnecting although in some situations, a manual reconnect to the AP is required. Future versions of pyw will include "connect" functionality to rectify this.

### 3.1.1 One-time vs Persistent Sockets

The example Listing 1 uses one-time sockets (netlink and ioctl). When using iw, there are several things that occur prior to the actual command or request being submitted. First, iw creates a netlink socket. Then, iw will request the family id for nl80211. The relative time spent doing this is neglible but, it is redundant and it may become noticeable in programs that repeatedly use the Netlink service. Once comlete, iw closes the socket. PyRIC eliminates these redundancies by using a global variable in pyw that stores the family id after the first time it is requested and by providing callers the option to use persistent sockets.

- **One-time Sockets** Similar to iw. The command, creates the netlink socket (or ioctl socket), composes the message, sends the message and receives the response, parses the results, closes the socket and returns the results to the caller. At no time does the caller need to be aware of any underlying Netlink processes or structures.
- **Persistent Sockets** Communication and parsing only. The onus of socket creation and deletion is on the caller which allows them to create one (or more) socket(s). The pyw functions will only handle message construction, message sending and receiving and message parsing.

The caller needs to be cognizant of whether the function requires a netlink or ioctl socket. Passing the wrong type will result in an error.

NOTE: One must remember that there is an upper limit to the number of open netlink sockets. It is advised to use one-time functions as much as possible and save the use of persistent sockets for use in code that repeatedly makes use of netlink.

The latest version of pyw (v 0.1.*) implements this functionality through the use of what I call templates[2], Listing 2 and stubs Listing 3.

```
def fcttemplate(arg0,arg1,..,argn,*argv):
    # put parameter validation (if any) here
    try:
        nlsock = argv[0]
    except IndexError:
        return _nlstub_(fcttemplate,arg0,arg1,...,argn)

    # command execution
    ...
    return results
```

[2]I use templates and stubs for the lack of any better naming convention

Listing 2: A Basic Netlink Function Template

The template function in Listing 2 checks if argv has a netlink socket[3] at index 0. If so, it proceeds to execution. If there is no socket, the stub is executed which creates one. If something other than a netlink socket is at argv[0], an error will be raised during execution.

```
def _nlstub_(fct,*argv):
    nlsock = None
    try:
        nlsock = nlsock = nl.nl_socket_alloc()
        argv = list(argv) + [nlsock]
        return fct(*argv)
    except pyric.error:
        raise # catch & release
    finally:
        if nlsock: nl.nl_socket_free(nlsock)
```

Listing 3: Function _nlstub_

The stub function, Listing 3 allocates a netlink socket, executes the original (now with a netlink socket) and then destroys the netlink socket.

```
 1: import pyric                      # pyric error (and ecode EUNDEF)
 2: from pyric import pyw             # for iw functionality
 3: from pyric import device          # for chipset/driver
 4: from pyric.channels import rf2ch  # rf to channel conversion
 5: from pyric.lib import libnl as nl # for netlink sockets
 6: from pyric.lib import libio as io # for ioctl sockets
 7:
 8: # create our sockets
 9: nlsock = nl.nl_socket_alloc(timeout=1)
10: iosock = io.io_socket_alloc()
11:
12: # list interfaces, wireless interfaces
13: print "Interfaces: ", pyw.interfaces()
14: print "Wireless Interfaces: ", pyw.winterfaces(iosock):
15:
16: # get a Card & info for alfa0 & print a description
17: info = pyw.devinfo('alfa0',nlsock)
18: card = info['card']
19: driver = device.ifdriver(card.dev)
20: chipset = device.ifchipset(driver)
21: msg = "{Using {0} in mode: {1}".format(card,info['mode'])
22: if info['mode'] == 'managed':
23:    msg += " currently on channel {0} width {1}".format(rf2ch(info['RF']),
24:                                                        info['CHW'])
25: msg += " driver: {0}, chipset: {1}".format(driver,chipset)
26: print msg
27:
28: # prepare a virtual interface named pent0 in monitor mode
29: pdev = 'pent0'
30: pcard = pyw.devadd(card,pdev,'monitor',nlsock)
```

[3]ioctl calls operate in the same manner

```
31: for iface in pyw.ifaces(card,nlsock):
32:     if iface[0].dev != pdev:
33:         msg = "deleting {0} in mode {1}".format(iface[0],iface[1])
34:         print msg
35:         pyw.devdel(iface[0],nlsock)
36: pyw.macset(pcard,'00:03:93:57:54:46',iosock)
37: pyw.up(pcard,ioctl)
38: pyw.chset(pcard,6,None)
39: msg = "Virtual interface {0} in monitor mode on ch 6".format(pcard)
40: msg += ", using hwaddr: {0}".format(pyw.macget(pcard,iosock)
41:
42: # DO stuff here
43:
44: # restore original
45: print "deleting ", pcard
46: pyw.devdel(pcard,nlsock)
47: card = pyw.devadd(card,card.dev,info['mode'],nlsock)
48: pyw.up(card,ioctl)
49: print "card ", card, " restored"
50:
51: # release the sockets
52: nl.nl_socket_free(nlsock)
53: io.io_socket_alloc(iosock)
```

Listing 4: Using Persistent Sockets

In Listing 4, the wireless pentesting environment from Listing 1 is repeated, this time using persistent sockets. Note the imports at lines 5 and 6 for libnl and libio, socket creation at lines 9 and 10 and socket destructions at lines 52 and 53. Also, note the lack of a socket used in the pyw.interfaces() call. This is due to the fact the function interfaces() reads from the file system and does not use netlink or ioctl.

Use Python's built in help features on pyw functions or see Appendex A to determine what type of socket is needed.

### 3.2 Interacting with the Kernel: libnl.py and libio.py

The kernel interfaces, libnl.py and libio.py are located in the lib directory. They handle socket creation/deletion, message creation/parsing and kernel communication. Aside from creating and deleting persistent sockets, there is little need to access their functions unless you plan on extending pyw functionality. As such, a further discussion of libnl.py and libio.py can be in the next section.

## 4 Extending PyRIC

You may find that pyw does not offer some of the functionality you need. Using libnl.py and/or libnl.io, additional functionality can be added to your program.

### 4.1 Porting C

All Python ports of C header files can be found in the net directory. C Enums and #defines are ported using constants. C structs are ported using three Python structures and the Python struct package:

1. a format string for packing and unpacking the struct
2. a constant specifying the size of the struct in bytes
3. a function taking the attributes of the struct as arguments and returning a packed string

Listing 5 shows the C definition of the nlmsghdr found in netlink.h.

```
struct nlmsghdr {
    __u32 nlmsg_len;
    __u16 nlmsg_type;
    __u16 nlmsg_flags;
    __u32 nlmsg_seq;
    __u32 nlmsg_pid;
};
```

Listing 5: C Struct nlmsghdr

And Listing 6 shows the ported version in Python.

```
nl_nlmsghdr = "IHHII"
NLMSGHDRLEN = struct.calcsize(nl_nlmsghdr)
def nlmsghdr(mlen,nltype,flags,seq,pid):
    return struct.pack(nl_nlmsghdr,NLMSGHDRLEN+mlen,nltype,flags,seq,pid)
```

Listing 6: Corresponding Python Definition

When using pyw, dealing with these structures is handled transparently by libnl.py and libio.py. When extending or customizing pyw, a basic understanding of the definitions in netlink_h.py, genetlink_h.py and if_h.py.

### 4.2 Input/Output Control (ioctl)

PyRIC provides more than just iw-related functions, it also implements functions from ifconfig and iwconfig. These command line tools still use ioctl (or the proc directory). For example, interfaces() reads from '/proc/net/dev' to retrieve all system interfaces and winterfaces() use ioctl to check if a device is wireless. Input/Output control calls have only been used when there was no viable alternative and, it should not be necessary to have to add any further ioctl commands. If you find that you need an ioctl related command, search through if_h.py for the appropriate structure and add it's definitions to ifreq.

### 4.3 Netlink and nl80211

Documentation on Netlink, and nl80211 in particular, is so minimal as to be neglible. The clusterfuck of code and lack of comments in the iw source tree make it impossible to use as any sort of roadmap. Fortunately Thomas Graf's site[2] has excellent coverage of the libnl, the Netlink library. Using this as a reference, a simple Netlink parser was put together. With strace and the parser, Netlink messages could be dissected and analyzed.

# Appendix A API: pyw.py

## A.1 Constants

- **_FAM80211ID_**: Global netlink family id of nl80211. Do not touch
- **IFTYPES**: redefined (from nl80211_h.py) interface modes
- **MNTRFLAGS**: redefined (from nl80211_h.py) monitor mode flags

## A.2 Objects/Classes

**Card** A wrapper around a tuple `t = (physical index,device name,interface index)` which exposes the following properties through '.':

- **phy**: physical index
- **dev**: device name
- **idx**: interface index (ifindex)

Because the underlying Netlink calls will sometimes require the physical index, sometimes the device name, and sometimes the ifindex, pyw functions accept a Card, object. This allows callers to use pyw functions without having to remember which identifier the function requires. However, in some cases the function requires a dev or accepts both. See the next section on functions.

While callers could create their own Cards, it is recommend to use one of the following

- **pyw.getcard** returns a Card object from a given dev
- **pyw.devinfo** returns the dict info where info['card'] is the Card object. This function will take either a card or a dev
- **pyw.devadd** returns a new Card object
- **pyw.ifaces** returns a list of tuples t = (Card,mode) sharing the same phy as a given dev to do so. It is also recommended to periodically validate the Card. On some cheaper usb wireless nics, there are periodic disconnects which results in a new phy and ifindex.

## A.3 Functions

- interfaces(): (ifconfig), type: filesystem, returns list of all network dev
- isinterface(dev): (ifconfig <dev>) type: filesystem, check dev is an interface
- winterfaces([iosock]): (iwconfig), type: ioctl, list wireless interfaces
- iswireless(dev,[iosock]): (iwconfig <dev>), type: ioctl, check dev is a wireless interface
- regget([nlsock]: (iw reg get), type: netlink, get regulatory domain
- regset(rd,[nlsock]): (iw reg set <rd>), type: netlink, set regulatory domain to rd
- getcard(dev,[nlsock]) (N/A), type: hybrid netlink and ioctl: get a Card object for dev

- macget(card,[iosock]): (ifconfig card.<dev>), type: ioctl get card's hw address
- macset(card,mac,[iosock]): (ifconfig card.<dev> hw ether <mac>), type: ioctl, set card's hw address to mac
- txget(card,[iosock]): (iwconfig card.<dev> | grep Tx-Power card), type: ioctl, get card's transmission power
- up(card,[iosock]) (ifconfig card.<dev> up), type: ioctl, bring card up
- down(card,[iosock]): (ifconfig card.<dev> down), type: ioctl, bring card down
- devstds(card,[iosock]): (iwconfig card.<dev> | grep IEEE), type: ioctl, get list of card's 802.11 supported standards
- validcard(card,[nlsock]): (N/A), type: (hyrbrid netlink and ioctl), verify card is still valid
- devinfo(card,[nlsock]): (iw dev card.<dev> info), type: netlink, get info for dev
- phyinfo(card,[nlsock]): (iw phy card.<phy> info), type: netlink, get info for phy
- ifaces(card,[nlsock]): (APX iw card.dev | grep phy#), type: netlink, get all cards (w/ modes) of interfaces sharing the same phy as card
- chget(card,[nlsock]): (iw dev <card.dev> info | grep channelS), type: netlink, get card's current channel (only works for cards in mode managed)
- chset(card,ch,chw,[nlsock]): iw phy <card.phy> set channel <ch> <chw>), type: netlink, set card's current channel to ch with width chw
- devmodes(card,[iosock]): (iw phy card.<phy>), type: netlink, get modes supported by card
- devadd(card,vnic,mode,[flags],[nlsock]): (iw phy card.<phy> interface add <vnic> type <mode> flags <flags>), type: netlink, creates a new virtual interface on card's phy with dev vdev, in mode and using flags. Note: flags are only supported in when creating a monitor mode
- devdel(card,[nlsock]): (iw card.<dev> del), type: netlink, deletes dev
  - _hex2mac_(v): returns a ':' separated mac address from byte stream v
  - _issetf_(flags,flag): determines if flag is set in flags
  - _setf_(flags,flag): set flag in flags to on
  - _unsetf_(flags,flag): set flag in flags to off
  - _familyid_(nlsock): returns and sets the Netlink family id for nl80211, only called once per module import
  - _ifindex_(dev,[iosock]): returns dev's ifindex
  - _flagsget_(dev,[iosock]): get's the dev's interface flags
  - _flagsset_(dev,flags,[iosock]): set's the dev's interface flags
  - _iostub_(fct,*argv): ioctl stub function, calls fct with parameter list argv and an allocated ioctl socket
  - _nlstub_(fct,*argv): netlink stub function, calls fct with parameter list argv and an allocated netlink socket

# Appendix B API: libnl.py

Providing libnl similar functionality, libnl.py provides the interface between pyw and the underlying nl80211 core. It relates similarily to libnl by providing functions handling netlink messages and sockets and where possible uses similarly named functions as those libnl to ease any transitions from C to PyRIC. However, several liberties have been taken as libnl.py handles only nl80211 generic netlink messages.

## B.1 Constants

- **BUFSZ** default rx and tx buffer size

## B.2 Classes/Objects

The two classes in libnl.py, NLSocket and GENLMsg, discussed in the following sections subclass Python's builtin dict. This has been done IOT to take advantage of dict's already existing functions and primarily their mutability and Python's 'pass by name' i.e. modifications in a function will be reflected in the caller. This makes the classes very similar to the use C pointers to structs in libnl.

### B.2.1 NLSocket

NLSocket is a wrapper around a netlink socket which exposes the following properties through '.':

- **sock**: the actual socket
- **fd**: the socket's file descriptor (deprecated)
- **tx**: size of the send buffer
- **rx**: size of the receive buffer
- **pid**: port id
- **grpm**: group mask
- **seq**: sequence number
- **timeout**: socket timeout

and has the following methods:

- incr(): increment sequence number
- send(pkt): sends pkt returning bytes sent
- recv(): returns received message (will block unless timeout is set)
- close(): close the socket

NLSockets are created with nl_socket_alloc and must be freed with nl_socket_free. See Section B.3.

### B.2.2 GENLMsg

GENLMsg is a wrapper around a dict with the following key->value pairs:

- **len**: total message length including the header
- **nltype**: netlink type
- **flags**: message flags
- **seq**: seq. #
- **pid**: port id
- **cmd**: generic netlink command
- **attrs**: list of message attributes. Each attribute is a tuple t = (attribute,value,datatype) where:
  - **attribute**: netlink attribute type i.e. CTRL_ATTR_FAMILY_ID
  - **value**: the unpacked attribute value
  - **datatype**: datatype of the attribute as defined in nelink_h i.e. NLA_U8

NOTE: as discussed below, on sending, the seq. # and port id are overridden with values of the netlink socket.

GENLMsg exposes the following properties:

- **len**: length of the message (get only)
- **vers**: returns 1 (default version) (get only)
- **nltype**: message content i.e. generic or nl80211 (get or set)
- **flags**: message flags (get or set)
- **seq**: current sequence # (get or set)
- **pid**: port id (get or set)
- **cmd**: netlink command (get or set)
- **attrs**: attribute list (get only)
- **numattrs**: number of attributes (get only)

GENLMsg has the following methods:

- __repr__(): returns a string representation useful for debugging
- tostream(): returns a packed netlink message
- nla_put(v,a,t): appends the attribute a, with value v and datatype t to the attribute list
- nla_put_<DATATYPE>(v,a): eight specialize functions that append attribute a with value v and type <DATATYPE> to the attribute list

- nla_putat(i,v,a,d): puts attribute a, with value v and datatype d at index i in the attribute list.
- nla_pop(i): removes the attribute tuple at index i, returning the popped tuple
- nla_find(a,value=True): returns the first attribute a. If value returns only the value otherwise returns the attribute tuple
- nla_get(i,value=True): returns the attribute at index i. If value returns only the value otherwise returns the attribute tuple
- _attrpack(a,v,d): (private) packs the attribute tuple

There are two methods of creating a GENLMsg. Create a new message (to send) with nlmsg_new and create a message from a received packet with nlmsg_fromstream. These are discussed below.

### B.3 Functions

- **Netlink Socket Related**
  - nl_socket_alloc(pid,grps,seq,rx,tx,timeout): creates a netlink socket with port id = pid, group mask = grps, initial seq. # = seq, send and receive buffer size = tx and rx respectively and blocking timeout = timeout
  - nl_socket_free(sock): closes the socket
  - nl_socket_pid(sock): (deprecated for NLSocket.pid) returns the port id
  - nl_socket_grpmask(sock): (deprecated for NLSocket.grpmask) returns the group mask
  - nl_sendmsg(sock,msg,override=False): sends the netlink msg over socket. NOTE: NLSockets will automatically set the port id and seq. # regardless of their value in the message. If override is True, the message's pid and seq. # will be used instead.
  - nl_recvmsg(sock): returns a GENLMsg or blocks unless the socket's timeout is set. Should only be called once per every nl_sendmsg.
- **Netlink Message Related**
  - nlmsg_new(nltype=None,cmd=None,pid=None,flags=None,attrs=None): creates a new GENLMsg with zero or more attributes defined.
  - nlmsg_fromstream(stream): parses the message in stream returning the corresponding GENLMsg
  - nla_parse(msg,l,mtype,stream,idx): parses the attributes in stream appending them to the attribute list of message where msg = the GENLMsg, l = the total length of the message, mtype = the message content (i.e. netlink type) stream is the original byte stream and idx is the index of the start of the attribute list
  - nla_parse_nested(nested): returns the list of packed nested attributes extracted from the stream nested. Callers must unpack and parse the returned attributes themselves
  - _nla_strip(v): (private) strips padding bytes from the end of v
  - _maxbufsz_(): returns the maximum allowable socket buffer size

## Appendix C Copyright and License

PYRIC: Python Radio Interface Controller v0.0.5



## References

[1] Gnu general public license, June 2007.

[2] Graf, T. Netlink library (libnl), May 2011.

[3] Pablo Neira Ayuso, Rafael M. Gasca, L. L. Communicating between the kernel and user-space in linux using netlink sockets. *Software - Practice And Experience 40* (August 2010), 797–810.

[4] Patterson, D. V. Wireless reconnaissance and intelligent target harvesting, April 2016.