Netlink Library (libnl)

Thomas Graf
<tgraf@suug.ch>
version 3.2, May 9 2011

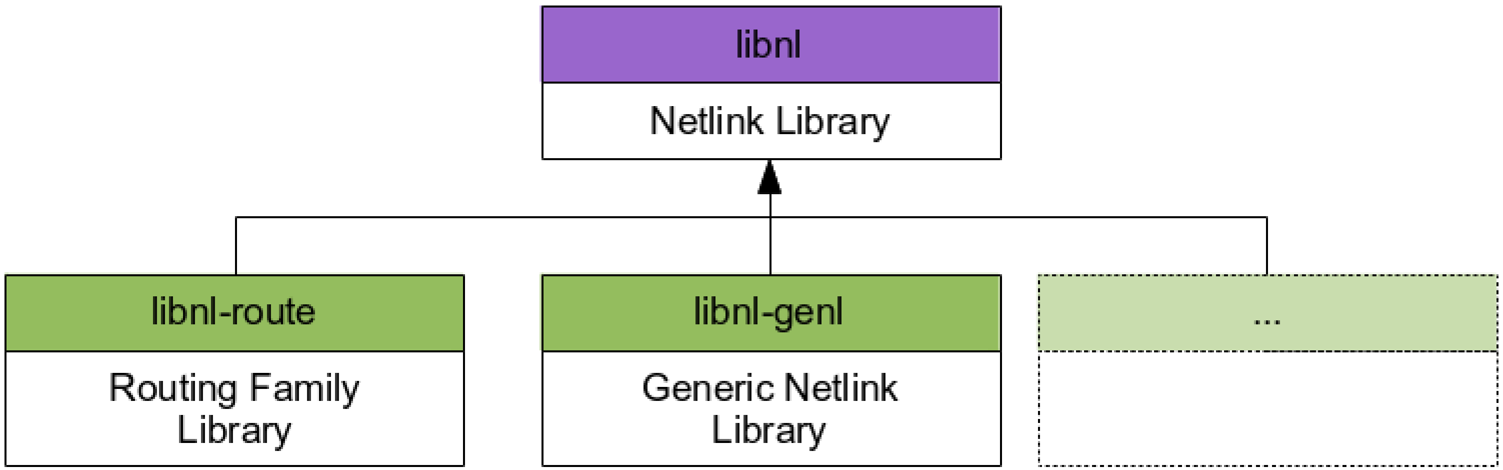
Table of Contents

1. Introduction

The core library contains the fundamentals required to communicate over netlink sockets. It deals with connecting and disconnectng of sockets, sending and receiving of data, construction and parsing of messages, provides a customizeable receiving state machine, and provides a abstract data type framework which eases the implementation of object based netlink protocols where objects are added, removed, or modified using a netlink based protocol.

Library Hierarchy

The suite is split into multiple libraries:



Netlink Library (libnl)
    Socket handling, sending and receiving, message construction and parsing, …

Routing Family Library (libnl-route)
: Adresses, links, neighbours, routing, traffic control, neighbour tables, …

Netfilter Library (libnl-nf)
: Connection tracking, logging, queueing

Generic Netlink Library (libnl-genl)
: Controller API, family and command registration

1.1. How To Read This Documentation

The libraries provide a broad set of APIs of which most applications only require a small subset of it. Depending on the type of application, some users may only be interested in the low level netlink messaging API while others wish to make heavy use of the high level API.

In any case it is recommended to get familiar with the netlink protocol first.

- Netlink Protocol Fundamentals

The low level APIs are described in:

- Netlink Sockets
- Sending and Receiving of Messages / Data

1.2. Linking to this Library

Checking the presence of the library using autoconf

Projects using autoconf may use `PKG_CHECK_MODULES()` to check if a specific version of libnl is available on the system. The example below also shows how to retrieve the `CFLAGS` and linking dependencies required to link against the library.

The following example shows how to check for a specific version of libnl. If found, it extends the `CFLAGS` and `LIBS` variable appropriately:

```
PKG_CHECK_MODULES(LIBNL3, libnl-3.0 >= 3.1, [have_libnl3=yes], [have_libnl3=no])

if (test "${have_libnl3}" = "yes"); then

        CFLAGS+="$LIBNL3_CFLAGS"

        LIBS+="$LIBNL3_LIBS"

fi
```

The pkgconfig file is named libnl-3.0.pc for historic reasons, it also covers library versions >= 3.1.

Header Files

The main header file is <netlink/netlink.h>. Additional headers may need to be included in your sources depending on the subsystems and components your program makes use of.

```
#include <netlink/netlink.h>

#include <netlink/cache.h>

#include <netlink/route/link.h>
```

Version Dependent Code

If your code wishes to be capable to link against multiple versions of libnl you may have direct the compiler to only include portions on the code depending on the version of libnl that it is compiled against.

```
#include <netlink/version.h>



#if LIBNL_VER_NUM >= LIBNL_VER(3,1)

        /* include code if compiled with libnl version >= 3.1 */

#endif
```

Linking

```
$ gcc myprogram.c -o myprogram $(pkgconfig --cflags --libs libnl-3.0)
```

1.3. Debugging

The library has been compiled with debugging statements enabled it will print debug information to `stderr` if the environment variable `NLDBG` is set to > 0.

```
$ NLDBG=2 ./myprogram
```

Table 1. Debugging Levels

Level	Description
0	Debugging disabled (default)
1	Warnings, important events and notifications
2	More or less important debugging messages
3	Repetitive events causing a flood of debugging messages
4	Even less important messages

Debugging the Netlink Protocol

It is often useful to peek into the stream of netlink messages exchanged with other sockets. Setting the environment variable `NLCB=debug` will cause the debugging message handlers to be used which in turn print the netlink messages exchanged in a human readable format to to `stderr`:

```
$ NLCB=debug ./myprogram

-- Debug: Sent Message:

--------------------------   BEGIN NETLINK MESSAGE ---------------------------

  [HEADER] 16 octets

    .nlmsg_len = 20

    .nlmsg_type = 18 <route/link::get>
```

```
    .nlmsg_flags = 773 <REQUEST,ACK,ROOT,MATCH>
    .nlmsg_seq = 1301410712
    .nlmsg_pid = 20014
  [PAYLOAD] 16 octets
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00       ................
--------------------------  END NETLINK MESSAGE   --------------------------
-- Debug: Received Message:
--------------------------   BEGIN NETLINK MESSAGE --------------------------
  [HEADER] 16 octets
    .nlmsg_len = 996
    .nlmsg_type = 16 <route/link::new>
    .nlmsg_flags = 2 <MULTI>
    .nlmsg_seq = 1301410712
    .nlmsg_pid = 20014
  [PAYLOAD] 16 octets
    00 00 04 03 01 00 00 00 49 00 01 00 00 00 00 00       ........I.......
  [ATTR 03] 3 octets
    6c 6f 00                                              lo.
  [PADDING] 1 octets
    00                                                    .
  [ATTR 13] 4 octets
    00 00 00 00                                           ....
  [ATTR 16] 1 octets
    00                                                    .
```

```
  [PADDING] 3 octets
    00 00 00                                                      ...
  [ATTR 17] 1 octets
    00                                                            .
  [...]
---------------------------  END NETLINK MESSAGE   ---------------------------
```

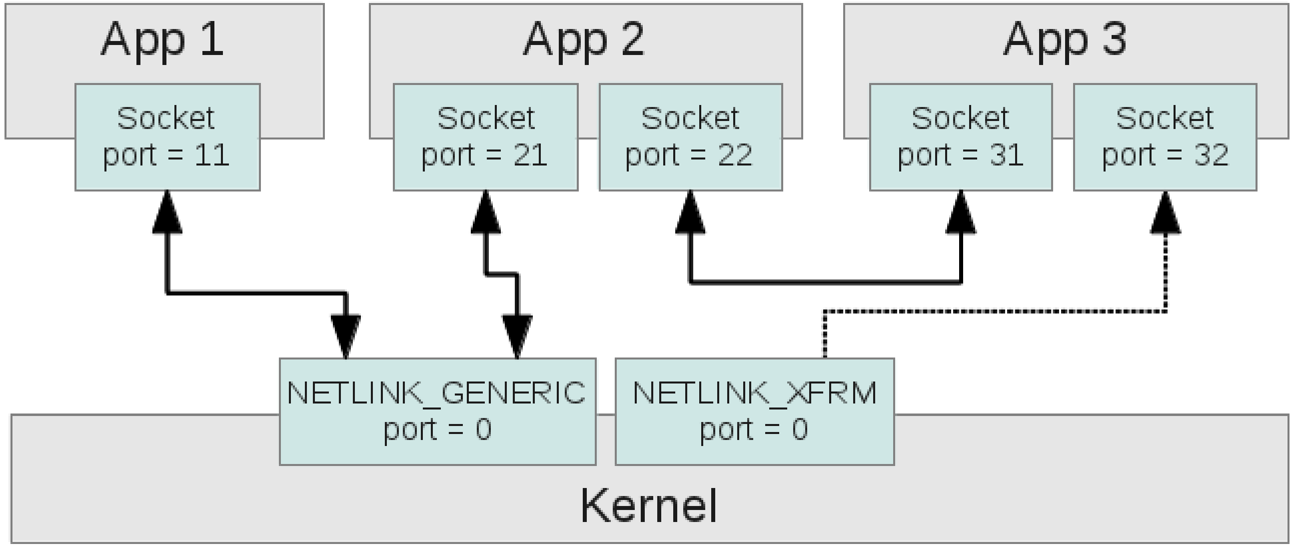
2. Netlink Protocol Fundamentals

The netlink protocol is a socket based IPC mechanism used for communication between userspace processes and the kernel or between userspace processes themselves. The netlink protocol is based on BSD sockets and uses the `AF_NETLINK` address family. Every netlink protocol uses its own protocol number (e.g. `NETLINK_ROUTE`, `NETLINK_NETFILTER`, etc). Its addressing schema is based on a 32 bit port number, formerly referred to as PID, which uniquely identifies each peer.

2.1. Addressing

The netlink address (port) consists of a 32bit integer. Port 0 (zero) is reserved for the kernel and refers to the kernel side socket of each netlink protocol family. Other port numbers usually refer to user space owned sockets, although this is not enforced.

> In the beginning, it was common practice to use the process identifier (PID) as the local port number. This became unpractical with the introduction of threaded netlink applications and applications requiring multiple sockets. Therefore libnl generates unique port numbers based on the process identifier and adds an offset to it allowing for multiple sockets to be used. The initial socket will still equal to the process identifier for backwards compatibility reasons.

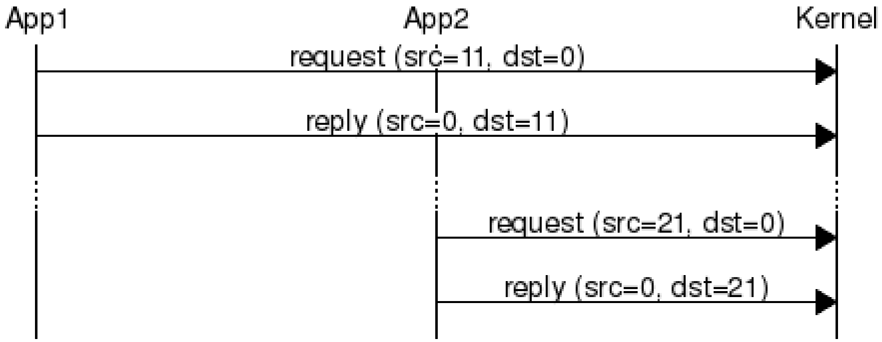


The above figure illustrates three applications and the kernel side exposing two kernel side sockets. It shows the common netlink use cases:

- User space to kernel
- User space to user space
- Listening to kernel multicast notifications

User Space to Kernel

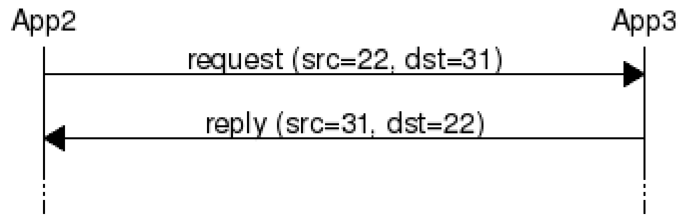
The most common form of netlink usage is for a user space application to send requests to the kernel and process the reply which is either an error message or a success notification.



User Space to User Space

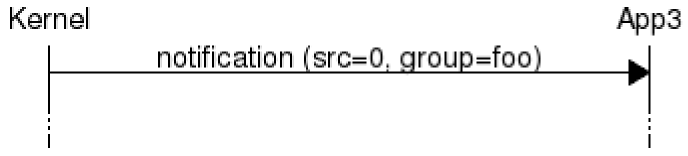
Netlink may also be used as an IPC mechanism to communicate between user space applications directly. Communication is not limited to two peers, any number of peers may communicate with each other and multicasting capabilities allow to reach multiple peers with a single message.

In order for the sockets to be visible to each other, both sockets must be created for the same netlink protocol family.



User space listening to kernel notifications

This form of netlink communication is typically found in user space daemons that need to act on certain kernel events. Such daemons will typically maintain a netlink socket subscribed to a multicast group that is used by the kernel to notify interested user space parties about specific events.

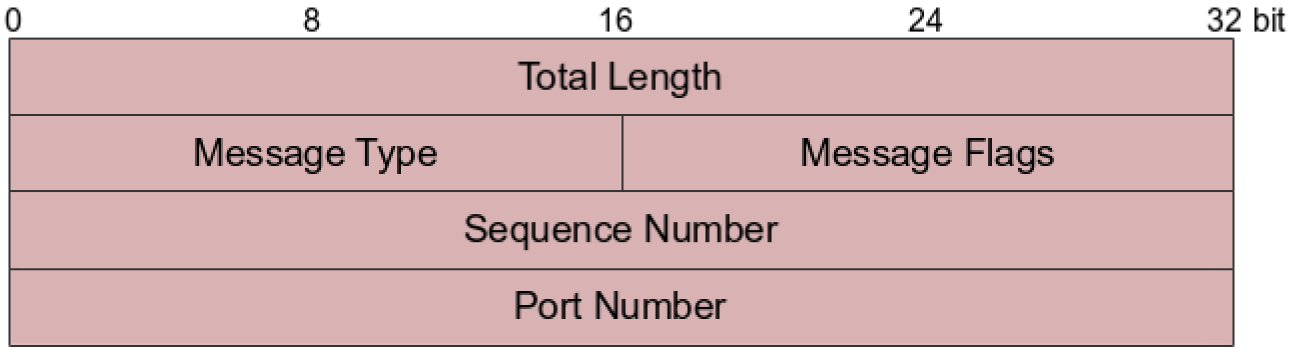


Use of multicasting is preferred over direct addressing due to the flexibility in exchanging the user space component at any time without the kernel noticing.

2.2. Message Format

A netlink protocol is typically based on messages and consists of the netlink message header (`struct nlmsghdr`) plus the payload attached to it. The payload can consist of arbitrary data but usually contains a fixed size protocol specific header followed by a stream of attributes.

Netlink message header (struct nlmsghdr)



Total Length (32bit)
: Total length of the message in bytes including the netlink message header.

Message Type (16bit)
: The message type specifies the type of payload the message is carrying. Several standard message types are defined by the netlink protocol. Additional message types may be defined by each protocol family. See Message Types for additional information.

Message Flags (16bit)
: The message flags may be used to modify the behaviour of a message type. See section Message Flags for a list of standard message flags.

Sequence Number (32bit)
: The sequence number is optional and may be used to allow referring to a previous message, e.g. an error message can refer to the original request causing the error.

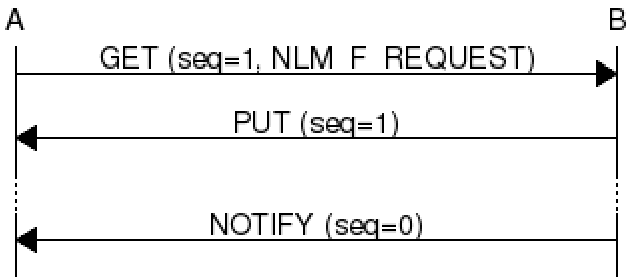
Port Number (32bit)
: The port number specifies the peer to which the message should be delivered to. If not specified, the message will be delivered to the first matching kernel side socket of the same protocol family.

2.3. Message Types

Netlink differs between requests, notifications, and replies. Requests are messages which have the `NLM_F_REQUEST` flag set and are meant to request an action from the receiver. A request is typically sent from a userspace process to the kernel. While not strictly enforced, requests should carry a sequence number incremented for each request sent.

Depending on the nature of the request, the receiver may reply to the request with another netlink message. The sequence number of a reply must match the sequence number of the request it relates to.

Notifications are of informal nature and no reply is expected, therefore the sequence number is typically set to 0.



The type of message is primarly identified by its 16 bit message type set in the message header. The following standard message types are defined:

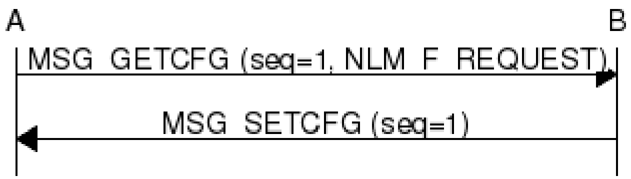
- `NLMSG_NOOP` - No operation, message must be discarded
- `NLMSG_ERROR` - Error message or ACK, see Error Message respectively ACKs
- `NLMSG_DONE` - End of multipart sequence, see Multipart Messages
- `NLMSG_OVERRUN` - Overrun notification (Error)

Every netlink protocol is free to define own message types. Note that message type values `< NLMSG_MIN_TYPE (0x10)` are reserved and may not be used.

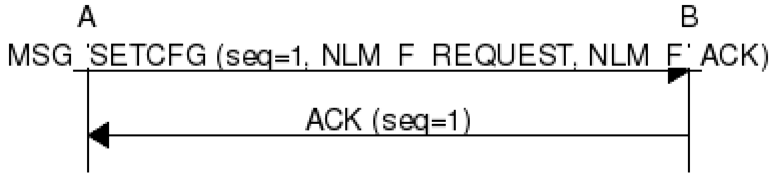
It is common practice to use own message types to implement RPC schemas. Suppose the goal of the netlink protocol you are implementing is allow configuration of a particular network device, therefore you want to provide read/write access to various configuration options. The typical "netlink way" of doing this would be to define two message types `MSG_SETCFG`, `MSG_GETCFG`:

```
#define MSG_SETCFG      0x11
#define MSG_GETCFG      0x12
```

Sending a `MSG_GETCFG` request message will typically trigger a reply with the message type `MSG_SETCFG` containing the current configuration. In object oriented terms one would describe this as "the kernel sets the local copy of the configuration in userspace".



The configuration may be changed by sending a `MSG_SETCFG` which will be responded to with either a ACK (see ACKs) or a error message (see Error Message).

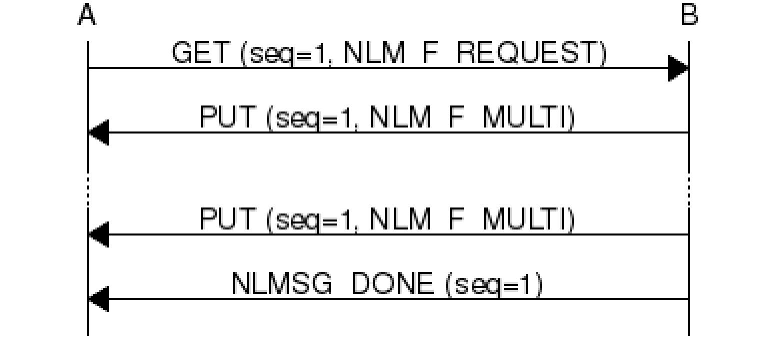


Optionally, the kernel may send out notifications for configuration changes allowing userspace to listen for changes instead of polling frequently. Notifications typically reuse an existing message type and rely on the application using a separate socket to differ between requests and notifications but you may also specify a separate message type.

2.3.1. Multipart Messages

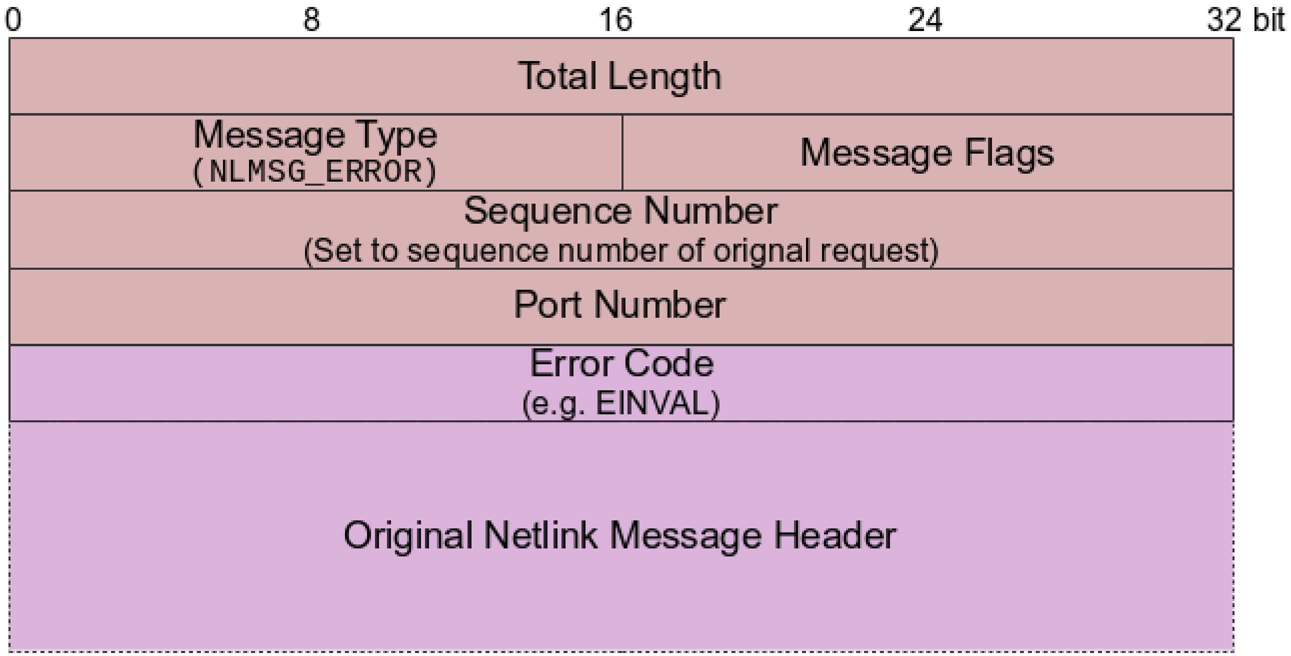
Although in theory a netlink message can be up to 4GiB in size. The socket buffers are very likely not large enough to hold message of such sizes. Therefore it is common to limit messages to one page size (PAGE_SIZE) and use the multipart mechanism to split large pieces of data into several messages. A multipart message has the flag `NLM_F_MULTI` set and the receiver is expected to continue receiving and parsing until the special message type `NLMSG_DONE` is received.

Multipart messages unlike fragmented ip packets must not be reassmbled even though it is perfectly legal to do so if the protocols wishes to work this way. Often multipart message are used to send lists or trees of objects were each multipart message simply carries multiple objects allow for each message to be parsed independently.

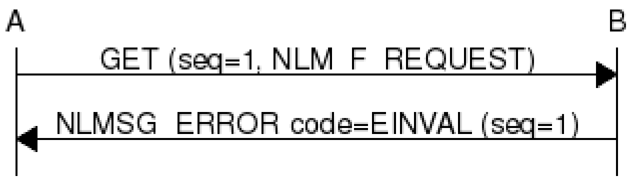


2.3.2. Error Message

Error messages can be sent in response to a request. Error messages must use the standard message type `NLMSG_ERROR`. The payload consists of a error code and the original netlink mesage header of the request.

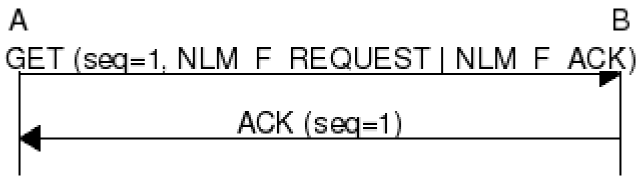


Error messages should set the sequence number to the sequence number of the request which caused the error.



2.3.3. ACKs

A sender can request an ACK message to be sent back for each request processed by setting the `NLM_F_ACK` flag in the request. This is typically used to allow the sender to synchronize further processing until the request has been processed by the receiver.



ACK messages also use the message type `NLMSG_ERROR` and payload format but the error code is set to 0.

2.3.4. Message Flags

The following standard flags are defined

```
#define NLM_F_REQUEST          1
#define NLM_F_MULTI            2
#define NLM_F_ACK              4
#define NLM_F_ECHO             8
```

- `NLM_F_REQUEST` - Message is a request, see Message Types.
- `NLM_F_MULTI` - Multipart message, see Multipart Messages
- `NLM_F_ACK` - ACK message requested, see ACKs.
- `NLM_F_ECHO` - Request to echo the request.

The flag `NLM_F_ECHO` is similar to the `NLM_F_ACK` flag. It can be used in combination with `NLM_F_REQUEST` and causes a notification which is sent as a result of a request to also be sent to the sender regardless of whether the sender has subscribed to the corresponding multicast group or not. See Multicast Groups

Additional universal message flags are defined which only apply for `GET` requests:

```
#define NLM_F_ROOT      0x100
#define NLM_F_MATCH     0x200
#define NLM_F_ATOMIC    0x400
```

```
#define NLM_F_DUMP      (NLM_F_ROOT|NLM_F_MATCH)
```

- `NLM_F_ROOT` - Return based on root of tree.
- `NLM_F_MATCH` - Return all matching entries.
- `NLM_F_ATOMIC` - Obsoleted, once used to request an atomic operation.
- `NLM_F_DUMP` - Return a list of all objects (`NLM_F_ROOT`|`NLM_F_MATCH`).

Use of these flags is completely optional and many netlink protocols only make use of the `NLM_F_DUMP` flag which typically requests the receiver to send a list of all objects in the context of the message type as a sequence of multipart messages (see Multipart Messages).

Another set of flags exist related to `NEW` or `SET` requests. These flags are mutually exclusive to the `GET` flags:

```
#define NLM_F_REPLACE   0x100

#define NLM_F_EXCL      0x200

#define NLM_F_CREATE    0x400

#define NLM_F_APPEND    0x800
```

- `NLM_F_REPLACE` - Replace an existing object if it exists.
- `NLM_F_EXCL` - Do not update object if it exists already.
- `NLM_F_CREATE` - Create object if it does not exist yet.
- `NLM_F_APPEND` - Add object at end of list.

Behaviour of these flags may differ slightly between different netlink protocols.

2.4. Sequence Numbers

Netlink allows the use of sequence numbers to help relate replies to requests. It should be noted that unlike in protocols such as TCP there is no strict enforcment of the sequence number. The sole purpose of sequence numbers is to assist a sender in relating replies to the corresponding requests. See Message Types for more information.

Sequence numbers are managed on a per socket basis, see Sequence Numbers for more information on how to use sequence numbers.

2.5. Multicast Groups

TODO

See Multicast Group Subscriptions

3. Netlink Sockets

In order to use the netlink protocol, a netlink socket is required. Each socket defines an independent context for sending and receiving of messages. An application may make use multiple sockets, e.g. a socket to send requests and receive the replies and another socket subscribed to a multicast group to receive notifications.

3.1. Socket structure (struct nl_sock)

The netlink socket and all related attributes including the actual file descriptor are represented by `struct nl_sock`.

```
#include <netlink/socket.h>


struct nl_sock *nl_socket_alloc(void)

void nl_socket_free(struct nl_sock *sk)
```

The application must allocate an instance of `struct nl_sock` for each netlink socket it wishes to use.

3.2. Sequence Numbers

The library will automatically take care of sequence number handling for the application. A sequence number counter is stored in the socket structure which is used and incremented automatically when a message needs to be sent which is expected to generate a reply such as an error or any other message type that needs to be related to the original message.

Alternatively, the counter can be used directly via the function nl_socket_use_seq(). It will return the current value of the counter and increment it by one afterwards.

```
#include <netlink/socket.h>


unsigned int nl_socket_use_seq(struct nl_sock *sk);
```

Most applications will not want to deal with sequence number handling themselves though. When using nl_send_auto() the sequence number is filled in automatically and matched again when a reply is received. See section Sending and Receiving of Messages / Data for more information.

This behaviour can and must be disabled if the netlink protocol implemented does not use a request/reply model, e.g. when a socket is used to receive notification messages.

```
#include <netlink/socket.h>


void nl_socket_disable_seq_check(struct nl_sock *sk);
```

For more information on the theory behind netlink sequence numbers, see section Sequence Numbers.

3.3. Multicast Group Subscriptions

Each socket can subscribe to any number of multicast groups of the netlink protocol it is connected to. The socket will then receive a copy of each message sent to any of the groups. Multicast groups are commonly used to implement event notifications.

Prior to kernel 2.6.14 the group subscription was performed using a bitmask which limited the number of groups per protocol family to 32. This outdated interface can still be accessed via the function nl_join_groups() even though it is not recommended for new code.

```
#include <netlink/socket.h>


void nl_join_groups(struct nl_sock *sk, int bitmask);
```

Starting with 2.6.14 a new method was introduced which supports subscribing to an almost infinite number of multicast groups.

```
#include <netlink/socket.h>

int nl_socket_add_memberships(struct nl_sock *sk, int group, ...);
int nl_socket_drop_memberships(struct nl_sock *sk, int group, ...);
```

3.3.1. Multicast Example

```
#include <netlink/netlink.h>
#include <netlink/socket.h>
#include <netlink/msg.h>

/*
 * This function will be called for each valid netlink message received
 * in nl_recvmsgs_default()
 */
static int my_func(struct nl_msg *msg, void *arg)
{
        return 0;
}

struct nl_sock *sk;

/* Allocate a new socket */
sk = nl_socket_alloc();
```

```
/*
 * Notifications do not use sequence numbers, disable sequence number
 * checking.
 */
nl_socket_disable_seq_check(sk);

/*
 * Define a callback function, which will be called for each notification
 * received
 */
nl_socket_modify_cb(sk, NL_CB_VALID, NL_CB_CUSTOM, my_func, NULL);

/* Connect to routing netlink protocol */
nl_connect(sk, NETLINK_ROUTE);

/* Subscribe to link notifications group */
nl_socket_add_memberships(sk, RTNLGRP_LINK, 0);

/*
 * Start receiving messages. The function nl_recvmsgs_default() will block
 * until one or more netlink messages (notification) are received which
 * will be passed on to my_func().
```

```
 */
while (1)
        nl_recvmsgs_default(sock);
```

3.4. Modifiying Socket Callback Configuration

See Callback Configurations for more information on callback hooks and overwriting capabilities.

Each socket is assigned a callback configuration which controls the behaviour of the socket. This is f.e. required to have a separate message receive function per socket. It is perfectly legal to share callback configurations between sockets though.

The following functions can be used to access and set the callback configuration of a socket:

```
#include <netlink/socket.h>


struct nl_cb *nl_socket_get_cb(const struct nl_sock *sk);
void nl_socket_set_cb(struct nl_sock *sk, struct nl_cb *cb);
```

Additionaly a shortcut exists to modify the callback configuration assigned to a socket directly:

```
#include <netlink/socket.h>


int nl_socket_modify_cb(struct nl_sock *sk, enum nl_cb_type type, enum nl_cb_kind kind,
                        nl_recvmsg_msg_cb_t func, void *arg);
```

Example:

```
#include <netlink/socket.h>
```

```
// Call my_input() for all valid messages received in socket sk
nl_socket_modify_cb(sk, NL_CB_VALID, NL_CB_CUSTOM, my_input, NULL);
```

3.5. Socket Attributes

Local Port

The local port number uniquely identifies the socket and is used to address it. A unique local port is generated automatically when the socket is allocated. It will consist of the Process ID (22 bits) and a random number (10 bits) thus allowing up to 1024 sockets per process.

```
#include <netlink/socket.h>


uint32_t nl_socket_get_local_port(const struct nl_sock *sk);
void nl_socket_set_local_port(struct nl_sock *sk, uint32_t port);
```

See section Addressing for more information on port numbers.

> Overwriting the local port is possible but you have to ensure that the provided value is unique and no other socket in any other application is using the same value.

Peer Port

A peer port can be assigned to the socket which will result in all unicast messages sent over the socket to be addresses to the peer. If no peer is specified, the message is sent to the kernel which will try to automatically bind the socket to a kernel side socket of the same netlink protocol family. It is common practice not to bind the socket to a peer port as typically only one kernel side socket exists per netlink protocol family.

```
#include <netlink/socket.h>
```

```
uint32_t nl_socket_get_peer_port(const struct nl_sock *sk);

void nl_socket_set_peer_port(struct nl_sock *sk, uint32_t port);
```

See section Addressing for more information on port numbers.

File Descriptor

Netlink uses the BSD socket interface, therefore a file descriptor is behind each socket and you may use it directly.

```
#include <netlink/socket.h>


int nl_socket_get_fd(const struct nl_sock *sk);
```

If a socket is used to only receive notifications it usually is best to put the socket in non-blocking mode and periodically poll for new notifications.

```
#include <netlink/socket.h>


int nl_socket_set_nonblocking(const struct nl_sock *sk);
```

Send/Receive Buffer Size

The socket buffer is used to queue netlink messages between sender and receiver. The size of these buffers specifies the maximum size you will be able to write() to a netlink socket, i.e. it will indirectly define the maximum message size. The default is 32KiB.

```
#include <netlink/socket.h>


int nl_socket_set_buffer_size(struct nl_sock *sk, int rx, int tx);
```

Enable/Disable Credentials

TODO

```
#include <netlink/socket.h>


int nl_socket_set_passcred(struct nl_sock *sk, int state);
```

Enable/Disable Auto-ACK Mode

The following functions allow to enable/disable Auto-ACK mode on a socket. See Auto-ACK Mode for more information on what implications that has. Auto-ACK mode is enabled by default.

```
#include <netlink/socket.h>


void nl_socket_enable_auto_ack(struct nl_sock *sk);

void nl_socket_disable_auto_ack(struct nl_sock *sk);
```

Enable/Disable Message Peeking

If enabled, message peeking causes nl_recv() to try and use MSG_PEEK to retrieve the size of the next message received and allocate a buffer of that size. Message peeking is enabled by default but can be disabled using the following function:

```
#include <netlink/socket.h>


void nl_socket_enable_msg_peek(struct nl_sock *sk);

void nl_socket_disable_msg_peek(struct nl_sock *sk);
```

Enable/Disable Receival of Packet Information

If enabled, each received netlink message from the kernel will include an additional struct nl_pktinfo in the control message. The following function can be used to enable/disable receival of packet information.

```
#include <netlink/socket.h>


int nl_socket_recv_pktinfo(struct nl_sock *sk, int state);
```

Processing of NETLINK_PKTINFO has not been implemented yet.

4. Sending and Receiving of Messages / Data

4.1. Sending Messages

The standard method of sending a netlink message over a netlink socket is to use the function nl_send_auto(). It will automatically complete the netlink message by filling the missing bits and pieces in the netlink message header and will deal with addressing based on the options and address set in the netlink socket. The message is then passed on to nl_send().

If the default sending semantics implemented by nl_send() do not suit the application, it may overwrite the sending function nl_send() by specifying an own implementation using the function nl_cb_overwrite_send().

```
    nl_send_auto(sk, msg)

          |

          |-----> nl_complete_msg(sk, msg)

          |

          |

          |              Own send function specified via nl_cb_overwrite_send()
```

```
      |- - - - - - - - - - - - - - - - - - - - -
      v                                       v
nl_send(sk, msg)                         send_func()
```

Using nl_send()

If you do not require any of the automatic message completion functionality you may use nl_send() directly but beware that any internal calls to nl_send_auto() by the library to send netlink messages will still use nl_send(). Therefore if you wish to use any higher level interfaces and the behaviour of nl_send() is to your dislike then you must overwrite the nl_send() function via nl_cb_overwrite_send()

The purpose of nl_send() is to embed the netlink message into a iovec structure and pass it on to nl_send_iovec().

```
nl_send(sk, msg)
      |
      v
nl_send_iovec(sk, msg, iov, iovlen)
```

Using nl_send_iovec()

nl_send_iovec() expects a finalized netlink message and fills out the struct msghdr used for addressing. It will first check if the struct nl_msg is addressed to a specific peer (see nlmsg_set_dst()). If not, it will try to fall back to the peer address specified in the socket (see nl_socket_set_peer_port(). Otherwise the message will be sent unaddressed and it is left to the kernel to find the correct peer.

nl_send_iovec() also adds credentials if present and enabled (see [core_sk_cred]).

The message is then passed on to nl_sendmsg().

```
nl_send_iovec(sk, msg, iov, iovlen)
      |
      v
nl_sendmsg(sk, msg, msghdr)
```

Using nl_sendmsg()

nl_sendmsg() expects a finalized netlink message and an optional struct msghdr containing the peer address. It will copy the local address as defined in the socket (see nl_socket_set_local_port()) into the netlink message header.

At this point, construction of the message finished and it is ready to be sent.

```
    nl_sendmsg(sk, msg, msghdr)

          |- - - - - - - - - - - - - - - - - - - - - v

          |                                 NL_CB_MSG_OUT()

          |<- - - - - - - - - - - - - - - - - - - - -+

          v

    sendmsg()
```

Before sending the application has one last chance to modify the message. It is passed to the NL_CB_MSG_OUT callback function which may inspect or modify the message and return an error code. If this error code is NL_OK the message is sent using sendmsg() resulting in the number of bytes written being returned. Otherwise the message sending process is aborted and the error code specified by the callback function is returned. See Modifiying Socket Callback Configuration for more information on how to set callbacks.

Sending Raw Data with nl_sendto()

If you wish to send raw data over a netlink socket, the following function will pass on any buffer provided to it directly to sendto():

```
#include <netlink/netlink.h>


int nl_sendto(struct nl_sock *sk, void *buf, size_t size);
```

Sending of Simple Messages

A special interface exists for sending of trivial messages. The function expects the netlink message type, optional netlink message flags, and an optional data buffer and data length.

```c
#include <netlink/netlink.h>

int nl_send_simple(struct nl_sock *sk, int type, int flags,
                   void *buf, size_t size);
```

The function will construct a netlink message header based on the message type and flags provided and append the data buffer as message payload. The newly constructed message is sent with nl_send_auto().

The following example will send a netlink request message causing the kernel to dump a list of all network links to userspace:

```c
#include <netlink/netlink.h>

struct nl_sock *sk;
struct rtgenmsg rt_hdr = {
        .rtgen_family = AF_UNSPEC,
};

sk = nl_socket_alloc();
nl_connect(sk, NETLINK_ROUTE);

nl_send_simple(sock, RTM_GETLINK, NLM_F_DUMP, &rt_hdr, sizeof(rt_hdr));
```

4.2. Receiving Messages

The easiest method to receive netlink messages is to call nl_recvmsgs_default(). It will receive messages based on the semantics defined in the socket. The application may customize these in detail although the default behaviour will probably suit most applications.

nl_recvmsgs_default() will also be called internally by the library whenever it needs to receive and parse a netlink message.

The function will fetch the callback configuration stored in the socket and call nl_recvmsgs():

```
nl_recvmsgs_default(sk)
      |
      | cb = nl_socket_get_cb(sk)
      v
nl_recvmsgs(sk, cb)
```

Using nl_recvmsgs()

nl_recvmsgs() implements the actual receiving loop, it blocks until a netlink message has been received unless the socket has been put into non-blocking mode.

For the unlikely scenario that certain required receive characteristics can not be achieved by fine tuning the internal recvmsgs function using the callback configuration (see Modifiying Socket Callback Configuration) the application may provide a complete own implementation of it and overwrite all calls to nl_recvmsgs() with the function nl_cb_overwrite_recvmsgs().

```
nl_recvmsgs(sk, cb)
      |
      |     Own recvmsgs function specified via nl_cb_overwrite_recvmsgs()
      |- - - - - - - - - - - - - - - - - - - -
      v                                      v
internal_recvmsgs()                    my_recvmsgs()
```

Receive Characteristics

If the application does not provide its own recvmsgs() implementation with the function nl_cb_overwrite_recvmsgs() the following characteristics apply while receiving data from a netlink socket:

```
        internal_recvmsgs()
                |
+-------------->|     Own recv function specified with nl_cb_overwrite_recv()
|               |- - - - - - - - - - - - - - - -
|               v                               v
|           nl_recv()                       my_recv()
|               |<- - - - - - - - - - - - - - -+
|               |<-------------+
|               v              | More data to parse? (nlmsg_next())
|         Parse Message        |
|               |--------------+
|               v
+------- NLM_F_MULTI set?
                |
                v
            (SUCCESS)
```

The function nl_recv() is invoked first to receive data from the netlink socket. This function may be overwritten by the application by an own implementation using the function nl_cb_overwrite_recv(). This may be useful if the netlink byte stream is in fact not received from a socket directly but is read from a file or another source.

If data has been read, it will be attemped to parse the data. This will be done repeately until the parser returns NL_STOP, an error was returned or all data has been parsed.

In case the last message parsed successfully was a multipart message (see Multipart Messages) and the parser did not quit due to either an error or NL_STOP nl_recv() respectively the applications own implementation will be called again and the parser starts all over.

See [core_parse_character] for information on how to extract valid netlink messages from the parser and on how to control the behaviour of it.

Parsing Characteristics

The internal parser is invoked for each netlink message received from a netlink socket. It is typically fed by nl_recv() (see [core_recv_character]).

The parser will first ensure that the length of the data stream provided is sufficient to contain a netlink message header and that the message length as specified in the message header does not exceed it.

If this criteria is met, a new struct nl_msg is allocated and the message is passed on to the the callback function NL_CB_MSG_IN if one is set. Like any other callback function, it may return NL_SKIP to skip the current message but continue parsing the next message or NL_STOP to stop parsing completely.

The next step is to check the sequence number of the message against the currently expected sequence number. The application may provide its own sequence number checking algorithm by setting the callback function NL_CB_SEQ_CHECK to its own implementation. In fact, calling nl_socket_disable_seq_check() to disable sequence number checking will do nothing more than set the NL_CB_SEQ_CHECK hook to a function which always returns NL_OK.

Another callback hook NL_CB_SEND_ACK exists which is called if the message has the NLM_F_ACK flag set. Although I am not aware of any userspace netlink socket doing this, the application may want to send an ACK message back to the sender (see ACKs).

```
         parse()

           |

           v

       nlmsg_ok() --> Ignore

           |

           |- - - - - - - - - - - - - - - v

           |                         NL_CB_MSG_IN()

           |<- - - - - - - - - - - - - - -+

           |

           |- - - - - - - - - - - - - - - v

      Sequence Check                NL_CB_SEQ_CHECK()

           |<- - - - - - - - - - - - - - -+

           |

           |              Message has NLM_F_ACK set
```

```
        |- - - - - - - - - - - - - - - v
        |                      NL_CB_SEND_ACK()
        |<- - - - - - - - - - - - - - -+
        |
Handle Message Type
```

4.3. Auto-ACK Mode

TODO

5. Message Parsing & Construction

5.1. Message Format

See Netlink Protocol Fundamentals for an introduction to the netlink protocol and its message format.

Alignment

Most netlink protocols enforce a strict alignment policy for all boundries. The alignment value is defined by NLMSG_ALIGNTO and is fixed to 4 bytes. Therefore all netlink message headers, begin of payload sections, protocol specific headers, and attribute sections must start at an offset which is a multiple of NLMSG_ALIGNTO.

```
#include <netlink/msg.h>


int nlmsg_size(int payloadlen);

int nlmsg_total_size(int payloadlen);
```

The library provides a set of function to handle alignment requirements automatically. The function nlmsg_total_size() returns the total size of a netlink message including the padding to ensure the next message header is aligned correctly.

```
      <----------- nlmsg_total_size(len) ------------>

      <----------- nlmsg_size(len) ------------>

     +-------------------+- - -+- - - - - - - - +- - -+-------------------+- - -

     |  struct nlmsghdr   | Pad |     Payload     | Pad |  struct nlsmghdr  |

     +-------------------+- - -+- - - - - - - - +- - -+-------------------+- - -

      <---- NLMSG_HDRLEN -----> <- NLMSG_ALIGN(len) -> <---- NLMSG_HDRLEN ---
```

If you need to know if padding needs to be added at the end of a message, nlmsg_padlen() returns the number of padding bytes that need to be added for a specific payload length.

```
#include <netlink/msg.h>

int nlmsg_padlen(int payloadlen);
```

5.2. Parsing a Message

The library offers two different methods of parsing netlink messages. It offers a low level interface for applications which want to do all the parsing manually. This method is described below. Alternatively the library also offers an interface to implement a parser as part of a cache operations set which is especially useful when your protocol deals with objects of any sort such as network links, routes, etc. This high level interface is described in Cache System.

Splitting a byte stream into separate messages

What you receive from a netlink socket is typically a stream of messages. You will be given a buffer and its length, the buffer may contain any number of netlink messages.

The first message header starts at the beginning of message stream. Any subsequent message headers are access by calling nlmsg_next() on the previous header.

```
#include <netlink/msg.h>


struct nlmsghdr *nlmsg_next(struct nlmsghdr *hdr, int *remaining);
```

The function nlmsg_next() will automatically substract the size of the previous message from the remaining number of bytes.

Please note, there is no indication in the previous message whether another message follows or not. You must assume that more messages follow until all bytes of the message stream have been processed.

To simplify this, the function nlmsg_ok() exists which returns true if another message fits into the remaining number of bytes in the message stream. nlmsg_valid_hdr() is similar, it checks whether a specific netlink message contains at least a minimum of payload.

```
#include <netlink/msg.h>


int nlmsg_valid_hdr(const struct nlmsghdr *hdr, int payloadlen);

int nlmsg_ok(const struct nlmsghdr *hdr, int remaining);
```

A typical use of these functions looks like this:

```
#include <netlink/msg.h>


void my_parse(void *stream, int length)

{

        struct nlmsghdr *hdr = stream;


        while (nlmsg_ok(hdr, length)) {

                // Parse message here

                hdr = nlmsg_next(hdr, &length);
```

```
        }
}
```

nlmsg_ok() only returns true if the **complete** message including the message payload fits into the remaining buffer length. It will return false if only a part of it fits.

The above can also be written using the iterator nlmsg_for_each():

```
#include <netlink/msg.h>

struct nlmsghdr *hdr;

nlmsg_for_each(hdr, stream, length) {
        /* do something with message */
}
```

Message Payload

The message payload is appended to the message header and is guranteed to start at a multiple of `NLMSG_ALIGNTO`. Padding at the end of the message header is added if necessary to ensure this. The function nlmsg_data() will calculate the necessary offset based on the message and returns a pointer to the start of the message payload.

```
#include <netlink/msg.h>

void *nlmsg_data(const struct nlmsghdr *nlh);
void *nlmsg_tail(const struct nlmsghdr *nlh);
```

```
int nlmsg_datalen(const struct nlmsghdr *nlh);
```

The length of the message payload is returned by nlmsg_datalen().

```
                               <--- nlmsg_datalen(nlh) --->
     +-------------------+- - -+----------------------------+- - -+
     |  struct nlmsghdr  | Pad |           Payload          | Pad |
     +-------------------+- - -+----------------------------+- - -+
nlmsg_data(nlh) ---------------^                                  ^
nlmsg_tail(nlh) --------------------------------------------------^
```

The payload may consist of arbitary data but may have strict alignment and formatting rules depening on the actual netlink protocol.

Message Attributes

Most netlink protocols use netlink attributes. It not only makes the protocol self documenting but also gives flexibility in expanding the protocol at a later point. New attributes can be added at any time and older attributes can be obsoleted by newer ones without breaking binary compatibility of the protocol.

```
                               <---------------------- payload ------------------------->
                               <----- hdrlen ---->       <- nlmsg_attrlen(nlh, hdrlen) ->
     +-------------------+- - -+-----  ------------+- - -+--------------------------------+- - -+
     |  struct nlmsghdr  | Pad |  Protocol Header  | Pad |           Attributes           | Pad |
     +-------------------+- - -+-------------------+- - -+--------------------------------+- - -+
nlmsg_attrdata(nlh, hdrlen) -----------------------------^
```

The function nlmsg_attrdata() returns a pointer to the begin of the attributes section. The length of the attributes section is returned by the function nlmsg_attrlen().

```
#include <netlink/msg.h>


struct nlattr *nlmsg_attrdata(const struct nlmsghdr *hdr, int hdrlen);

int nlmsg_attrlen(const struct nlmsghdr *hdr, int hdrlen);
```

See Attributes for more information on how to use netlink attributes.

Parsing a Message the Easy Way

The function nlmsg_parse() validate a complete netlink message in one step. If `hdrlen > 0` it will first call nlmsg_valid_hdr() to check if the protocol header fits into the message. If there is more payload to parse, it will assume it to be attributes and parse the payload accordingly. The function behaves exactly like nla_parse() when parsing attributes, see [core_attr_parse_easy].

```
int nlmsg_parse(struct nlmsghdr *hdr, int hdrlen, struct nlattr **attrs,

                int maxtype, struct nla_policy *policy);
```

The function nlmsg_validate() is based on nla_validate() and behaves exactly the same as nlmsg_parse() except that it only validates and will not fill a array with pointers to each attribute.

```
int nlmsg_validate(struct nlmsghdr *hdr, int hdrlen, intmaxtype,

                   struct nla_policy *policy);
```

See [core_attr_parse_easy] for an example and more information on attribute parsing.

5.3. Construction of a Message

See Message Format for information on the netlink message format and alignment requirements.

Message construction is based on struct nl_msg which uses an internal buffer to store the actual netlink message. struct nl_msg `does not` point to the netlink message header. Use nlmsg_hdr() to retrieve a pointer to the netlink message header.

At allocation time, a maximum message size is specified. It defaults to a page (PAGE_SIZE). The application constructing the message will reserve space out of this maximum message size repeatedly for each header or attribute added. This allows construction of messages across various layers of code where lower layers do not need to know about the space requirements of upper layers.

`Why is setting the maximum message size necessary?` This question is often raised in combination with the proposed solution of reallocating the message payload buffer on the fly using realloc(). While it is possible to reallocate the buffer during construction using nlmsg_expand() it will make all pointers into the message buffer become stale. This breaks usage of nlmsg_hdr(), nla_nest_start(), and nla_nest_end() and is therefore not acceptable as default behaviour.

Allocating struct nl_msg

The first step in constructing a new netlink message it to allocate a `struct nl_msg` to hold the message header and payload. Several functions exist to simplify various tasks.

```
#include <netlink/msg.h>


struct nl_msg *nlmsg_alloc(void);

void nlmsg_free(struct nl_msg *msg);
```

The function nlmsg_alloc() is the default message allocation function. It allocates a new message using the default maximum message size which equals to one page (PAGE_SIZE). The application can change the default size for messages by calling nlmsg_set_default_size():

```
void      nlmsg_set_default_size(size_t);
```

Calling nlmsg_set_default_size() does not change the maximum message size of already allocated messages.

```
struct nl_msg *nlmsg_alloc_size(size_t max);
```

Instead of changing the default message size, the function nlmsg_alloc_size() can be used to allocate a message with a individual maximum message size.

If the netlink message header is already known at allocation time, the application may sue nlmsg_inherit(). It will allocate a message using the default maximum message size and copy the header into the message. Calling nlmsg_inherit with `set` to NULL is equivalent to calling nlmsg_alloc().

```
struct nl_msg *nlmsg_inherit(struct nlmsghdr *hdr);
```

Alternatively nlmsg_alloc_simple() takes a netlink message type and netlink message flags. It is equivalent to nlmsg_inherit() except that it takes the two common header fields as arguments instead of a complete header.

```
#include <netlink/msg.h>


struct nl_msg *nlmsg_alloc_simple(int nlmsg_type, int flags);
```

Appending the netlink message header

After allocating struct nl_msg, the netlink message header needs to be added unless one of the function nlmsg_alloc_simple() or nlmsg_inherit() have been used for allocation in which case this step will replace the netlink message header already in place.

```
#include <netlink/msg.h>


struct nlmsghdr *nlmsg_put(struct nl_msg *msg, uint32_t port, uint32_t seqnr,
                           int nlmsg_type, int payload, int nlmsg_flags);
```

The function nlmsg_put() will build a netlink message header out of `nlmsg_type`, `nlmsg_flags`, `seqnr`, and `port` and copy it into the netlink message. `seqnr` can be set to `NL_AUTO_SEQ` to indiciate that the next possible sequence number should be used automatically. To use this feature, the message must be sent using the function nl_send_auto(). Like `port`, the argument `seqnr` can be set to `NL_AUTO_PORT` indicating that the local port assigned to the socket should be used as source port. This is generally a good idea unless you are replying to a request. See Netlink Protocol Fundamentals for more information on how to fill the header.

The argument `payload` can be used by the application to reserve room for additional data after the header. A value of > 0 is equivalent to calling `nlmsg_reserve(msg, payload, NLMSG_ALIGNTO)`. See [core_msg_reserve] for more information on reserving room for data.

Example

```
#include <netlink/msg.h>

struct nlmsghdr *hdr;
struct nl_msg *msg;
struct myhdr {
        uint32_t foo1, foo2;
} hdr = { 10, 20 };

/* Allocate a message with the default maximum message size */
msg = nlmsg_alloc();

/*
 * Add header with message type MY_MSGTYPE, the flag NLM_F_CREATE,
 * let library fill port and sequence number, and reserve room for
 * struct myhdr
 */
hdr = nlmsg_put(msg, NL_AUTO_PORT, NL_AUTO_SEQ, MY_MSGTYPE, sizeof(hdr), NLM_F_CREATE);

/* Copy own header into newly reserved payload section */
memcpy(nlmsg_data(hdr), &hdr, sizeof(hdr));
```

```
/*
 * The message will now look like this:
 *     +-------------------+- - -+----------------+- - -+
 *     |  struct nlmsghdr  | Pad |  struct myhdr  | Pad |
 *     +-------------------+-----+----------------+- - -+
 * nlh -^                        /                \
 *                              +--------+---------+
 *                              |  foo1  |  foo2   |
 *                              +--------+---------+
 */
```

Reserving room at the end of the message

Most functions described later on will automatically take care of reserving room for the data that is added to the end of the netlink message. In some situations it may be requried for the application to reserve room directly though.

```
#include <netlink/msg.h>

void *nlmsg_reserve(struct nl_msg *msg, size_t len, int pad);
```

The function nlmsg_reserve() reserves `len` bytes at the end of the netlink message and returns a pointer to the start of the reserved area. The `pad` argument can be used to request `len` to be aligned to any number of bytes prior to reservation.

The following example requests to reserve a 17 bytes area at the end of message aligned to 4 bytes. Therefore a total of 20 bytes will be reserved.

```
#include <netlink/msg.h>
```

```
void *buf = nlmsg_reserve(msg, 17, 4);
```

`nlmsg_reserve()` will **not** align the start of the buffer. Any alignment requirements must be provided by the owner of the previous message section.

Appending data at the end of the message

The function `nlmsg_append()` appends `len` bytes at the end of the message, padding it if requested and necessary.

```
#include <netlink/msg.h>


int nlmsg_append(struct nl_msg *msg, void *data, size_t len, int pad);
```

It is equivalent to calling `nlmsg_reserve()` and `memcpy()`ing the data into the freshly reserved data section.

`nlmsg_append()` will **not** align the start of the data. Any alignment requirements must be provided by the owner of the previous message section.

Adding attribtues to a message

Construction of attributes and addition of attribtues to the message is covereted in section Attributes.

6. Attributes

Any form of payload should be encoded as netlink attributes whenever possible. Use of attributes allows to extend any netlink protocol in the future without breaking binary compatibility. F.e. Suppose your device may currently be using 32 bit counters for statistics but years later the device switches to maintaining 64 bit counters to account for faster network hardware. If your protocol is using attributes the move to 64 bit counters is trivial and only involves in sending an additional attribute containing the 64 bit variants while still providing the old legacy 32 bit counters. If your protocol is not using attributes you will not be able to switch data types without breaking all existing users of the protocol.

The concept of nested attributes also allows for subsystems of your protocol to implement and maintain their own attribute schemas. Suppose a new generation of network device is introduced which requires a completely new set of configuration settings which was unthinkable when the netlink protocol was initially designed. Using attributes the new generation of devices may define a new attribute and fill it with its own new structure of attributes which extend or even obsolete the old attributes.

Therefore, *always* use attributes even if you are almost certain that the message format will never ever change in the future.

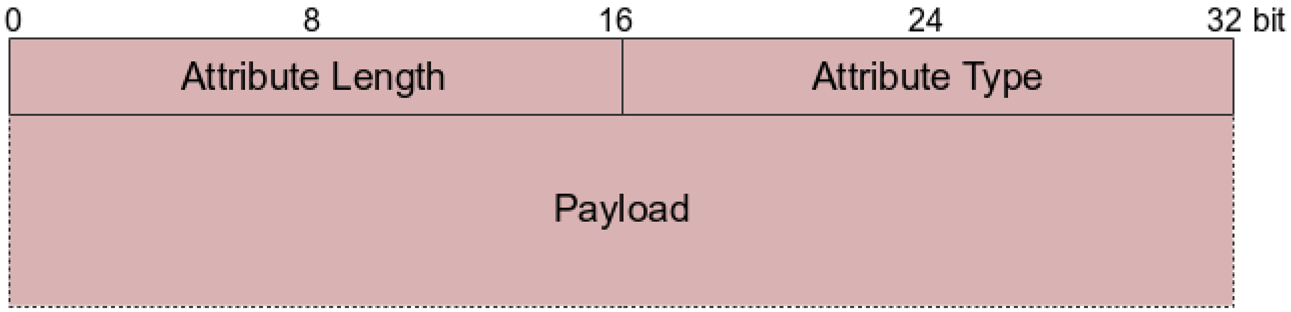
6.1. Attribute Format

Netlink attributes allow for any number of data chunks of arbitary length to be attached to a netlink message. See [core_msg_attr] for more information on where attributes are stored in the message.

The format of the attributes data returned by nlmsg_attrdata() is as follows:

```
     <----------- nla_total_size(payload) ----------->
     <---------- nla_size(payload) ----------->
    +-----------------+- - -+- - - - - - - - - +- - -+-----------------+- - -
    |  struct nlattr  | Pad |     Payload      | Pad |  struct nlattr  |
    +-----------------+- - -+- - - - - - - - - +- - -+-----------------+- - -
     <---- NLA_HDRLEN -----> <--- NLA_ALIGN(len) ---> <---- NLA_HDRLEN ---
```

Every attribute must start at an offset which is a multiple of `NLA_ALIGNTO` (4 bytes). If you need to know whether an attribute needs to be padded at the end, the function nla_padlen() returns the number of padding bytes that will or need to be added.



Every attribute is encoded with a type and length field, both 16 bits, stored in the attribute header (struct nlattr) preceding the attribute payload. The length of an attribute is used to calculate the offset to the next attribute.

6.2. Parsing Attributes

Splitting an Attributes Stream into Attributes

Although most applications will use one of the functions from the nlmsg_parse() family (See [core_attr_parse_easy]) an interface exists to split the attributes stream manually.

As described in Attribute Format the attributes section contains a infinite sequence or stream of attributes. The pointer returned by nlmsg_attrdata() (See [core_msg_attr]) points to the first attribute header. Any subsequent attribute is accessed with the function nla_next() based on the previous header.

```
#include <netlink/attr.h>


struct nlattr *nla_next(const struct nlattr *attr, int *remaining);
```

The semantics are equivalent to nlmsg_next() and thus nla_next() will also subtract the size of the previous attribute from the remaining number of bytes in the attributes stream.

Like messages, attributes do not contain an indicator whether another attribute follows or not. The only indication is the number of bytes left in the attribute stream. The function nla_ok() exists to determine whether another attribute fits into the remaining number of bytes or not.

```
#include <netlink/attr.h>
```

```
int nla_ok(const struct nlattr *attr, int remaining);
```

A typical use of nla_ok() and nla_next() looks like this:

nla_ok()/nla_next() usage

```
#include <netlink/msg.h>

#include <netlink/attr.h>


struct nlattr *hdr = nlmsg_attrdata(msg, 0);

int remaining = nlmsg_attrlen(msg, 0);


while (nla_ok(hdr, remaining)) {

        /* parse attribute here */

        hdr = nla_next(hdr, &remaining);

};
```

nla_ok() only returns true if the **complete** attributes including the attribute payload fits into the remaining number of bytes.

Accessing Attribute Header and Payload

Once the individual attributes have been sorted out by either splitting the attributes stream or using another interface the attribute header and payload can be accessed.

```
                              <- nla_len(hdr) ->

     +-----------------+- - -+- - - - - - - - - +- - -+
```

```
    |  struct nlattr  | Pad |     Payload      | Pad |
    +-----------------+- - -+- - - - - - - - - +- - -+
nla_data(hdr) ---------------^
```

The functions nla_len() and nla_type() can be used to access the attribute header. nla_len() will return the length of the payload not including eventual padding bytes. nla_type returns the attribute type.

```
#include <netlink/attr.h>


int nla_len(const struct nlattr *hdr);
int nla_type(const struct nlattr *hdr);
```

The function nla_data() will return a pointer to the attribute payload. Please note that due to `NLA_ALIGNTO` being 4 bytes it may not be safe to cast and dereference the pointer for any datatype larger than 32 bit depending on the architecture the application is run on.

```
#include <netlink/attr.h>


void *nla_data(const struct nlattr *hdr);
```

> Never rely on the size of a payload being what you expect it to be. *Always* verify the payload size and make sure that it matches your expectations. See [core_attr_validation]

Attribute Validation

When receiving netlink attributes, the receiver has certain expections on how the attributes should look like. These expectations must be defined to make sure the sending side meets our expecations. For this purpose, a attribute validation interface exists which must be used prior to accessing any payload.

All functions providing attribute validation functionality are based on struct nla_policy:

```
struct nla_policy {
        uint16_t        type;
        uint16_t        minlen;
        uint16_t        maxlen;
};
```

The `type` member specifies the datatype of the attribute, e.g. `NLA_U32`, `NLA_STRING`, `NLA_FLAG`. The default is `NLA_UNSPEC`. The `minlen` member defines the minmum payload length of an attribute to be considered a valid attribute. The value for `minlen` is implicit for most basic datatypes such as integers or flags. The `maxlen` member can be used to define a maximum payload length for an attribute to still be considered valid.

> Specyfing a maximum payload length is not recommended when encoding structures in an attribute as it will prevent any extension of the structure in the future. Something that is frequently done in netlink protocols and does not break backwards compatibility.

One of the functions which use struct nla_policy is nla_validate(). The function expects an array of struct nla_policy and will access the array using the attribute type as index. If an attribute type is out of bounds the attribute is assumed to be valid. This is intentional behaviour to allow older applications not yet aware of recently introduced attributes to continue functioning.

```
#include <netlink/attr.h>


int nla_validate(struct nlattr *head, int len, int maxtype, struct nla_policy *policy);
```

The function nla_validate() returns 0 if all attributes are valid, otherwise a validation failure specific error code is returned.

Most applications will rarely use nla_validate() directly but use nla_parse() instead which takes care of validation in the same way but also parses the the attributes in the same step. See [core_attr_parse_easy] for an example and more information.

The validation process in detail:

1. If attribute type is 0 or exceeds `maxtype` attribute is considered valid, 0 is returned.
2. If payload length is < `minlen`, `-NLE_ERANGE` is returned.
3. If `maxlen` is defined and payload exceeds it, `-NLE_ERANGE` is returned.
4. Datatype specific requirements rules, see Attribute Data Types
5. If all is ok, 0 is returned.

Parsing Attributes the Easy Way

Most applications will not want to deal with splitting attribute streams themselves as described in [core_attr_parse_split] A much easier method is to use nla_parse().

```
#include <netlink/attr.h>


int nla_parse(struct nlattr **attrs, int maxtype, struct nlattr *head,

              int len, struct nla_policy *policy);
```

The function nla_parse() will iterate over a stream of attributes, validate each attribute as described in [core_attr_validation] If the validation of all attributes succeeds, a pointer to each attribute is stored in the `attrs` array at `attrs[nla_type(attr)]`.

As an alernative to nla_parse() the function nlmsg_parse() can be used to parse the message and its attributes in one step. See [core_attr_parse_easy] for information on how to use these functions.

Example:

The following example demonstrates how to parse a netlink message sent over a netlink protocol which does not use protocol headers. The example does enforce a attribute policy however, the attribute MY_ATTR_FOO must be a 32 bit integer, and the attribute MY_ATTR_BAR must be a string with a maximum length of 16 characters.

```
#include <netlink/msg.h>

#include <netlink/attr.h>


enum {

        MY_ATTR_FOO = 1,
```

```
        MY_ATTR_BAR,
        __MY_ATTR_MAX,
};

#define MY_ATTR_MAX (__MY_ATTR_MAX - 1)

static struct nla_policy my_policy[MY_ATTR_MAX+1] = {
        [MY_ATTR_FOO] = { .type = NLA_U32 },
        [MY_ATTR_BAR] = { .type = NLA_STRING,
                          .maxlen = 16 },
};

void parse_msg(struct nlmsghdr *nlh)
{
        struct nlattr *attrs[MY_ATTR_MAX+1];

        if (nlmsg_parse(nlh, 0, attrs, MY_ATTR_MAX, my_policy) < 0)
                /* error */

        if (attrs[MY_ATTR_FOO]) {
                /* MY_ATTR_FOO is present in message */
                printf("value: %u\n", nla_get_u32(attrs[MY_ATTR_FOO]));
        }
```

```
}
```

Locating a Single Attribute

An application only interested in a single attribute can use one of the functions nla_find() or nlmsg_find_attr(). These function will iterate over all attributes, search for a matching attribute and return a pointer to the corresponding attribute header.

```
#include <netlink/attr.h>


struct nlattr *nla_find(struct nlattr *head, int len, int attrtype);
```

```
#include <netlink/msg.h>


struct nlattr *nlmsg_find_attr(struct nlmsghdr *hdr, int hdrlen, int attrtype);
```

`nla_find()` and `nlmsg_find_attr()` will **not** search in nested attributes recursively, see Nested Attributes.

6.2.1. Iterating over a Stream of Attributes

In some situations it does not make sense to assign a unique attribute type to each attribute in the attribute stream. For example a list may be transferd using a stream of attributes and even if the attribute type is incremented for each attribute it may not make sense to use the nlmsg_parse() or nla_parse() function to fill an array.

Therefore methods exist to iterate over a stream of attributes:

```
#include <netlink/attr.h>
```

```
nla_for_each_attr(attr, head, len, remaining)
```

nla_for_each_attr() is a macro which can be used in front of a code block:

```
#include <netlink/attr.h>


struct nalttr *nla;

int rem;


nla_for_each_attr(nla, attrstream, streamlen, rem) {

        /* validate & parse attribute */

}


if (rem > 0)

        /* unparsed attribute data */
```

6.3. Attribute Construction

The interface to add attributes to a netlink message is based on the regular message construction interface. It assumes that the message header and an eventual protocol header has been added to the message already.

```
struct nlattr *nla_reserve(struct nl_msg *msg, int attrtype, int len);
```

The function nla_reserve() adds an attribute header at the end of the message and reserves room for `len` bytes of payload. The function returns a pointer to the attribute payload section inside the message. Padding is added at the end of the attribute to ensure the next attribute is properly aligned.

```
int nla_put(struct nl_msg *msg, int attrtype, int attrlen, const void *data);
```

The function nla_put() is base don nla_reserve() but takes an additional pointer `data` pointing to a buffer containing the attribute payload. It will copy the buffer into the message automatically.

Example:

```
struct my_attr_struct {
        uint32_t a;
        uint32_t b;
};

int my_put(struct nl_msg *msg)
{
        struct my_attr_struct obj = {
                .a = 10,
                .b = 20,
        };

        return nla_put(msg, ATTR_MY_STRUCT, sizeof(obj), &obj);
}
```

See Attribute Data Types for datatype specific attribute construction functions.

Exception Based Attribute Construction

Like in the kernel API an exception based construction interface is provided. The behaviour of the macros is identical to their regular function counterparts except that in case of an error, the target `nla_put_failure` is jumped.

Example:

```
#include <netlink/msg.h>

#include <netlink/attr.h>


void construct_attrs(struct nl_msg *msg)

{

        NLA_PUT_STRING(msg, MY_ATTR_FOO1, "some text");

        NLA_PUT_U32(msg, MY_ATTR_FOO1, 0x1010);

        NLA_PUT_FLAG(msg, MY_ATTR_FOO3, 1);


        return 0;


nla_put_failure:

        /* NLA_PUT* macros jump here in case of an error */

        return -EMSGSIZE;

}
```

See Attribute Data Types for more information on the datatype specific exception based variants.

6.4. Attribute Data Types

A number of basic data types have been defined to simplify access and validation of attributes. The datatype is not encoded in the attribute, therefore bthe sender and receiver are required to use the same definition on what attribute is of what type.

Type	Description

Type	Description
NLA_UNSPEC	Unspecified attribute
NLA_U{8|16|32}	Integers
NLA_STRING	String
NLA_FLAG	Flag
NLA_NESTED	Nested attribute

Besides simplified access to the payload of such datatypes, the major advantage is the automatic validation of each attribute based on a policy. The validation ensures safe access to the payload by checking for minimal payload size and can also be used to enforce maximum payload size for some datatypes.

6.4.1. Integer Attributes

The most frequently used datatypes are integers. Integers come in four different sizes:

NLA_U8	8bit integer
NLA_U16	16bit integer
NLA_U32	32bit integer
NLA_U64	64bit integer

Note that due to the alignment requirements of attributes the integer attribtue `NLA_u8` and `NLA_U16` will not result in space savings in the netlink message. Their use is intended to limit the range of values.

Parsing Integer Attributes

```
#include <netlink/attr.h>


uint8_t  nla_get_u8(struct nlattr *hdr);
uint16_t nla_get_u16(struct nlattr *hdr);
```

```
uint32_t nla_get_u32(struct nlattr *hdr);

uint64_t nla_get_u64(struct nlattr *hdr);
```

Example:

```
if (attrs[MY_ATTR_FOO])

        uint32_t val = nla_get_u32(attrs[MY_ATTR_FOO]);
```

Constructing Integer Attributes

```
#include <netlink/attr.h>


int nla_put_u8(struct nl_msg *msg, int attrtype, uint8_t value);

int nla_put_u16(struct nl_msg *msg, int attrtype, uint16_t value);

int nla_put_u32(struct nl_msg *msg, int attrtype, uint32_t value);

int nla_put_u64(struct nl_msg *msg, int attrtype, uint64_t value);
```

Exception based:

```
NLA_PUT_U8(msg, attrtype, value)

NLA_PUT_U16(msg, attrtype, value)

NLA_PUT_U32(msg, attrtype, value)

NLA_PUT_U64(msg, attrtype, value)
```

Validation

Use NLA_U8, NLA_U16, NLA_U32, or NLA_U64 to define the type of integer when filling out a struct nla_policy array. It will automatically enforce the correct minimum payload length policy.

Validation does not differ between signed and unsigned integers, only the size matters. If the appliaction wishes to enforce particular value ranges it must do so itself.

```
static struct nla_policy my_policy[ATTR_MAX+1] = {
        [ATTR_FOO] = { .type = NLA_U32 },
        [ATTR_BAR] = { .type = NLA_U8 },
};
```

The above is equivalent to:

```
static struct nla_policy my_policy[ATTR_MAX+1] = {
        [ATTR_FOO] = { .minlen = sizeof(uint32_t) },
        [ATTR_BAR] = { .minlen = sizeof(uint8_t) },
};
```

6.4.2. String Attributes

The string datatype represents a NUL termianted character string of variable length. It is not intended for binary data streams.

The payload of string attributes can be accessed with the function nla_get_string(). nla_strdup() calls strdup() on the payload and returns the newly allocated string.

```
#include <netlink/attr.h>


char *nla_get_string(struct nlattr *hdr);
char *nla_strdup(struct nlattr *hdr);
```

String attributes are constructed with the function `nla_put_string()` respectively `NLA_PUT_STRING()`. The length of the payload will be strlen()+1, the trailing NUL byte is included.

```
int nla_put_string(struct nl_msg *msg, int attrtype, const char *data);


NLA_PUT_STRING(msg, attrtype, data)
```

For validation purposes the type `NLA_STRING` can be used in `struct nla_policy` definitions. It implies a minimum payload length of 1 byte and checks for a trailing NUL byte. Optionally the `maxlen` member defines the maximum length of a character string (including the trailing NUL byte).

```
static struct nla_policy my_policy[] = {
        [ATTR_FOO] = { .type = NLA_STRING,
                       .maxlen = IFNAMSIZ },
};
```

6.4.3. Flag Attributes

The flag attribute represents a boolean datatype. The presence of the attribute implies a value of `true`, the absence of the attribute implies the value `false`. Therefore the payload length of flag attributes is always 0.

```
int nla_get_flag(struct nlattr *hdr);
int nla_put_flag(struct nl_msg *msg, int attrtype);
```

The type `NLA_FLAG` is used for validation purposes. It implies a `maxlen` value of 0 and thus enforces a maximum payload length of 0.

Example:

```
/* nla_put_flag() appends a zero sized attribute to the message. */
nla_put_flag(msg, ATTR_FLAG);
```

```
/* There is no need for a receival function, the presence is the value. */
if (attrs[ATTR_FLAG])
        /* flag is present */
```

6.4.4. Nested Attributes

As described in Attributes, attributes can be nested allowing for complex tree structures of attributes. It is commonly used to delegate the responsibility of a subsection of the message to a subsystem. Nested attributes are also commonly used for transmitting list of objects.

When nesting attributes, the nested attributes are included as payload of a container attribute.

> **Note:** When validating the attributes using nlmsg_validate(), nlmsg_parse(), nla_validate(), or nla_parse() only the attributes on the first level are being validated. None of these functions will validate attributes recursively. Therefore you must explicitely call nla_validate() or use nla_parse_nested() for each level of nested attributes.

The type `NLA_NESTED` should be used when defining nested attributes in a struct nla_policy definition. It will not enforce any minimum payload length unless `minlen` is specified explicitely. This is because some netlink protocols implicitely allow empty container attributes.

```
static struct nla_policy my_policy[] = {
        [ATTR_OPTS] = { .type = NLA_NESTED },
};
```

Parsing of Nested Attributes

The function nla_parse_nested() is used to parse nested attributes. Its behaviour is identical to nla_parse() except that it takes a struct nlattr as argument and will use the payload as stream of attributes.

```
if (attrs[ATTR_OPTS]) {
        struct nlattr *nested[NESTED_MAX+1];
```

```
        struct nla_policy nested_policy[] = {
                [NESTED_FOO] = { .type = NLA_U32 },
        };

        if (nla_parse_nested(nested, NESTED_MAX, attrs[ATTR_OPTS], nested_policy) < 0)
                /* error */

        if (nested[NESTED_FOO])
                uint32_t val = nla_get_u32(nested[NESTED_FOO]);
}
```

Construction of Nested Attributes

Attributes are nested by surrounding them with calls to nla_nest_start() and nla_nest_end(). nla_nest_start() will add a attribute header to the message but no actual payload. All data added to the message from this point on will be part of the container attribute until nla_nest_end() is called which "closes" the attribute, correcting its payload length to include all data length.

```
int put_opts(struct nl_msg *msg)
{
        struct nlattr *opts;

        if (!(opts = nla_nest_start(msg, ATTR_OPTS)))
                goto nla_put_failure;

        NLA_PUT_U32(msg, NESTED_FOO, 123);
        NLA_PUT_STRING(msg, NESTED_BAR, "some text");
```

```
        nla_nest_end(msg, opts);
        return 0;

nla_put_failure:
        nla_nest_cancel(msg, opts);
        return -EMSGSIZE;
}
```

6.4.5. Unspecified Attribute

This is the default attribute type and used when none of the basic datatypes is suitable. It represents data of arbitary type and length.

See Address Allocation for a more information on a special interface allowing the allocation of abstract address object based on netlink attributes which carry some form of network address.

See Abstract Data Allocation for more information on how to allocate abstract data objects based on netlink attributes.

Use the function nla_get() and nla_put() to access the payload and construct attributes. See Attribute Construction for an example.

6.5. Examples

6.5.1. Constructing a Netlink Message with Attributes

```
struct nl_msg *build_msg(int ifindex, struct nl_addr *lladdr, int mtu)
{
        struct nl_msg *msg;
        struct nlattr *info, *vlan;
        struct ifinfomsg ifi = {
                .ifi_family = AF_INET,
```

```
        .ifi_index = ifindex,
};

/* Allocate a default sized netlink message */
if (!(msg = nlmsg_alloc_simple(RTM_SETLINK, 0)))
        return NULL;

/* Append the protocol specific header (struct ifinfomsg)*/
if (nlmsg_append(msg, &ifi, sizeof(ifi), NLMSG_ALIGNTO) < 0)
        goto nla_put_failure

/* Append a 32 bit integer attribute to carry the MTU */
NLA_PUT_U32(msg, IFLA_MTU, mtu);

/* Append a unspecific attribute to carry the link layer address */
NLA_PUT_ADDR(msg, IFLA_ADDRESS, lladdr);

/* Append a container for nested attributes to carry link information */
if (!(info = nla_nest_start(msg, IFLA_LINKINFO)))
        goto nla_put_failure;

/* Put a string attribute into the container */
NLA_PUT_STRING(msg, IFLA_INFO_KIND, "vlan");
```

```
	/*
	 * Append another container inside the open container to carry
	 * vlan specific attributes
	 */
	if (!(vlan = nla_nest_start(msg, IFLA_INFO_DATA)))
		goto nla_put_failure;

	/* add vlan specific info attributes here... */

	/* Finish nesting the vlan attributes and close the second container. */
	nla_nest_end(msg, vlan);

	/* Finish nesting the link info attribute and close the first container. */
	nla_nest_end(msg, info);

	return msg;

nla_put_failure:
	nlmsg_free(msg);
	return NULL;
}
```

6.5.2. Parsing a Netlink Message with Attributes

```c
int parse_message(struct nlmsghdr *hdr)
{
        /*
         * The policy defines two attributes: a 32 bit integer and a container
         * for nested attributes.
         */
        struct nla_policy attr_policy[] = {
                [ATTR_FOO] = { .type = NLA_U32 },
                [ATTR_BAR] = { .type = NLA_NESTED },
        };
        struct nlattr *attrs[ATTR_MAX+1];
        int err;

        /*
         * The nlmsg_parse() function will make sure that the message contains
         * enough payload to hold the header (struct my_hdr), validates any
         * attributes attached to the messages and stores a pointer to each
         * attribute in the attrs[] array accessable by attribute type.
         */
        if ((err = nlmsg_parse(hdr, sizeof(struct my_hdr), attrs, ATTR_MAX,
                               attr_policy)) < 0)
                goto errout;

        if (attrs[ATTR_FOO]) {
```

```
		/*
		 * It is safe to directly access the attribute payload without
		 * any further checks since nlmsg_parse() enforced the policy.
		 */
		uint32_t foo = nla_get_u32(attrs[ATTR_FOO]);
	}

	if (attrs[ATTR_BAR]) {
		struct *nested[NESTED_MAX+1];

		/*
		 * Attributes nested in a container can be parsed the same way
		 * as top level attributes.
		 */
		err = nla_parse_nested(nested, NESTED_MAX, attrs[ATTR_BAR],
		                       nested_policy);
		if (err < 0)
			goto errout;

		// Process nested attributes here.
	}

	err = 0;
```

```
errout:
        return err;
}
```

7. Callback Configurations

Callback hooks and overwriting capabilities are provided in various places inside library to control the behaviour of several functions. All the callback and overwrite functions are packed together in struct nl_cb which is attached to a netlink socket or passed on to functions directly.

7.1. Callback Hooks

Callback hooks are spread across the library to provide entry points for message processing and to take action upon certain events.

Callback functions may return the following return codes:

Return Code	Description
`NL_OK`	Proceed.
`NL_SKIP`	Skip message currently being processed and continue parsing the receive buffer.
`NL_STOP`	Stop parsing and discard all remaining data in the receive buffer.

Default Callback Implementations

The library provides three sets of default callback implementations: * `NL_CB_DEFAULT` This is the default set. It implets the default behaviour. See the table below for more information on the return codes of each function. * `NL_CB_VERBOSE` This set is based on the default set but will cause an error message to be printed to stderr for error messages, invalid messages, message overruns and unhandled valid messages. The `arg` pointer in nl_cb_set() and nl_cb_err() can be used to provide a FILE * which overwrites stderr. * `NL_CB_DEBUG` This set is intended for debugging purposes. It is based on the verbose set but will decode and dump each message sent or received to the console.

Table 2. nl_sendmsg() callback hooks:

Callback ID	Description	Default Return Value
NL_CB_MSG_OUT	*Each message sent*	NL_OK

Any function called by NL_CB_MSG_OUT may return a negative error code to prevent the message from being sent and the error code being returned.

nl_recvmsgs() callback hooks (ordered by priority):

Callback ID	Description	Default Return Value
NL_CB_MSG_IN	*Each message received*	NL_OK
NL_CB_SEQ_CHECK	*May overwrite sequence check algo*	NL_OK
NL_CB_INVALID	*Invalid messages*	NL_STOP
NL_CB_SEND_ACK	*Messages with NLM_F_ACK flag set*	NL_OK
NL_CB_FINISH	*Messages of type NLMSG_DONE*	NL_STOP
NL_CB_SKIPPED	*Messages of type NLMSG_NOOP*	NL_SKIP
NL_CB_OVERRUN	*Messages of type NLMSG_OVERRUN*	NL_STOP
NL_CB_ACK	*ACK Messages*	NL_STOP
NL_CB_VALID	*Each valid message*	NL_OK

Any of these functions may return NL_OK, NL_SKIP, or NL_STOP.

Message processing callback functions are set with nl_cb_set():

```
#include <netlink/handlers.h>
```

```
int nl_cb_set(struct nl_cb *cb, enum nl_cb_type type, enum nl_cb_kind kind,
              nl_recvmsg_msg_cb_t func, void *arg);

typedef int (*nl_recvmsg_msg_cb_t)(struct nl_msg *msg, void *arg);
```

Callback for Error Messages

A special function prototype is used for the error message callback hook:

```
#include <netlink/handlers.h>

int nl_cb_err(struct nl_cb *cb, enum nl_cb_kind kind, nl_recvmsg_err_cb_t func, void *arg);

typedef int(* nl_recvmsg_err_cb_t)(struct sockaddr_nl *nla, struct nlmsgerr *nlerr, void *arg);
```

Example: Setting up a callback set

```
#include <netlink/handlers.h>

/* Allocate a callback set and initialize it to the verbose default set */
struct nl_cb *cb = nl_cb_alloc(NL_CB_VERBOSE);

/* Modify the set to call my_func() for all valid messages */
nl_cb_set(cb, NL_CB_VALID, NL_CB_CUSTOM, my_func, NULL);

/*
```

```
 * Set the error message handler to the verbose default implementation
 * and direct it to print all errors to the given file descriptor.
 */
FILE *file = fopen(...);
nl_cb_err(cb, NL_CB_VERBOSE, NULL, file);
```

7.2. Overwriting of Internal Functions

When the library needs to send or receive netlink messages in high level interfaces it does so by calling its own low level API. In the case the default characteristics are not sufficient for the application, it may overwrite several internal function calls with own implementations.

Overwriting recvmsgs()

See Receiving Netlink Messages for more information on how and when recvmsgs() is called internally.

```
#include <netlink/handlers.h>


void nl_cb_overwrite_recvmsgs(struct nl_cb *cb,
                              int (*func)(struct nl_sock *sk, struct nl_cb *cb));
```

The following criteras must be met if a recvmsgs() implementation is supposed to work with high level interfaces:

- MUST respect the callback configuration `cb`, therefore:
- MUST call `NL_CB_VALID` for all valid messages, passing on
- MUST call `NL_CB_ACK` for all ACK messages
- MUST correctly handle multipart messages, calling NL_CB_VALID for each message until a NLMSG_DONE message is received.
- MUST report error code if a NLMSG_ERROR or NLMSG_OVERRUN mesasge is received.

Overwriting nl_recv()

Often it is sufficient to overwrite `nl_recv()` which is responsible from receiving the actual data from the socket instead of replacing the complete `recvmsgs()` logic.

See Receive Characteristics for more information on how and when `nl_recv()` is called internally.

```
#include <netlink/handlers.h>


void nl_cb_overwrite_recv(struct nl_cb *cb,
                          int (*func)(struct nl_sock * sk,
                                      struct sockaddr_nl *addr,
                                      unsigned char **buf,
                                      struct ucred **cred));
```

The following criteras must be met for an own `nl_recv()` implementation:

- **MUST** return the number of bytes read or a negative error code if an error occured. The function may also return 0 to indicate that no data has been read.
- **MUST** set `*buf` to a buffer containing the data read. It must be safe for the caller to access the number of bytes read returned as return code.
- **MAY** fill out `*addr` with the netlink address of the peer the data has been received from.
- **MAY** set `*cred` to a newly allocated struct ucred containg credentials.

Overwriting nl_send()

See Sending Netlink Messages for more information on how and when nl_send() is called internally.

```
#include <netlink/handlers.h>


void nl_cb_overwrite_send(struct nl_cb *cb, int (*func)(struct nl_sock *sk,
                                                        struct nl_msg *msg));
```

Own implementations must send the netlink message and return 0 on success or a negative error code.

8. Cache System

8.1. Allocation of Caches

Almost all subsystem provide a function to allocate a new cache of some form. The function usually looks like this:

```
struct nl_cache *<object name>_alloc_cache(struct nl_sock *sk);
```

These functions allocate a new cache for the own object type, initializes it properly and updates it to represent the current state of their master, e.g. a link cache would include all links currently configured in the kernel.

Some of the allocation functions may take additional arguments to further specify what will be part of the cache.

All such functions return a newly allocated cache or NULL in case of an error.

8.2. Cache Manager

The purpose of a cache manager is to keep track of caches and automatically receive event notifications to keep the caches up to date with the kernel state. Each manager has exactly one netlink socket assigned which limits the scope of each manager to exactly one netlink family. Therefore all caches committed to a manager must be part of the same netlink family. Due to the nature of a manager, it is not possible to have a cache maintain two instances of the same cache type. The socket is subscribed to the event notification group of each cache and also put into non-blocking mode. Functions exist to poll() on the socket to wait for new events to be received.

```
 App       libnl                        Kernel

        |                            |

            +-----------------+       [ notification, link change ]

        |   |  Cache Manager  |      | [   (IFF_UP | IFF_RUNNING)  ]

            |                 |                |

        |   |   +------------+|      |         |  [ notification, new addr ]
```

```
     <-------|---| route/link |<-------(async)--+  [  10.0.1.1/32 dev eth1  ]
         |   |   +------------+|      |                      |
             |   +------------+|                             |
     <---|---|---| route/addr |<------|-(async)--------------+
             |   +------------+|
         |   |   +------------+|      |
     <-------|---| ...        ||
         |   |   +------------+|      |
             +-----------------+
         |                            |
```

Creating a new cache manager

```
struct nl_cache_mngr *mngr;


// Allocate a new cache manager for RTNETLINK and automatically

// provide the caches added to the manager.

mngr = nl_cache_mngr_alloc(NETLINK_ROUTE, NL_AUTO_PROVIDE);
```

Keep track of a cache

```
struct nl_cache *cache;


// Create a new cache for links/interfaces and ask the manager to

// keep it up to date for us. This will trigger a full dump request
```

```
// to initially fill the cache.
cache = nl_cache_mngr_add(mngr, "route/link");
```

Make the manager receive updates

```
// Give the manager the ability to receive updates, will call poll()
// with a timeout of 5 seconds.
if (nl_cache_mngr_poll(mngr, 5000) > 0) {
        // Manager received at least one update, dump cache?
        nl_cache_dump(cache, ...);
}
```

Release cache manager

```
nl_cache_mngr_free(mngr);
```

9. Abstract Data Types

A few high level abstract data types which are used by a majority netlink protocols are implemented in the core library. More may be added in the future if the need arises.

9.1. Abstract Address

Most netlink protocols deal with networking related topics and thus dealing with network addresses is a common task.

Currently the following address families are supported:

- AF_INET
- AF_INET6
- AF_LLC

- `AF_DECnet`
- `AF_UNSPEC`

Address Allocation

The function nl_addr_alloc() allocates a new empty address. The `maxsize` argument defines the maximum length of an address in bytes. The size of an address is address family specific. If the address family and address data are known at allocation time the function nl_addr_build() can be used alternatively. You may also clone an address by calling nl_addr_clone()

```
#include <netlink/addr.h>


struct nl_addr *nl_addr_alloc(size_t maxsize);

struct nl_addr *nl_addr_clone(struct nl_addr *addr);

struct nl_addr *nl_addr_build(int family, void *addr, size_t size);
```

If the address is transported in a netlink attribute, the function nl_addr_alloc_attr() allocates a new address based on the payload of the attribute provided. The `family` argument is used to specify the address family of the address, set to `AF_UNSPEC` if unknown.

```
#include <netlink/addr.h>


struct nl_addr *nl_addr_alloc_attr(struct nlattr *attr, int family);
```

If the address is provided by a user, it is usually stored in a human readable format. The function nl_addr_parse() parses a character string representing an address and allocates a new address based on it.

```
#include <netlink/addr.h>


int nl_addr_parse(const char *addr, int hint, struct nl_addr **result);
```

If parsing succeeds the function returns 0 and the allocated address is stored in `*result`.

> Make sure to return the reference to an address using `nl_addr_put()` after usage to allow memory being freed.

Example: Transform character string to abstract address

```
struct nl_addr *a = nl_addr_parse("::1", AF_UNSPEC);
printf("Address family: %s\n", nl_af2str(nl_addr_get_family(a)));
nl_addr_put(a);
a = nl_addr_parse("11:22:33:44:55:66", AF_UNSPEC);
printf("Address family: %s\n", nl_af2str(nl_addr_get_family(a)));
nl_addr_put(a);
```

Address References

Abstract addresses use reference counting to account for all users of a particular address. After the last user has returned the reference the address is freed.

If you pass on a address object to another function and you are not sure how long it will be used, make sure to call nl_addr_get() to acquire an additional reference and have that function or code path call nl_addr_put() as soon as it has finished using the address.

```
#include <netlink/addr.h>


struct nl_addr *nl_addr_get(struct nl_addr *addr);
void nl_addr_put(struct nl_addr *addr);
int nl_addr_shared(struct nl_addr *addr);
```

You may call nl_addr_shared() at any time to check if you are the only user of an address.

Address Attributes

The address is usually set at allocation time. If it was unknown at that time it can be specified later by calling nl_addr_set_family() and is accessed with the function nl_addr_get_family().

```
#include <netlink/addr.h>


void nl_addr_set_family(struct nl_addr *addr, int family);

int nl_addr_get_family(struct nl_addr *addr);
```

The same is true for the actual address data. It is typically present at allocation time. For exceptions it can be specified later or overwritten with the function `nl_addr_set_binary_addr()`. Beware that the length of the address may not exceed `maxlen` specified at allocation time. The address data is returned by the function `nl_addr_get_binary_addr()` and its length by the function `nl_addr_get_len()`.

```
#include <netlink/addr.h>


int nl_addr_set_binary_addr(struct nl_addr *addr, void *data, size_t size);

void *nl_addr_get_binary_addr(struct nl_addr *addr);

unsigned int nl_addr_get_len(struct nl_addr *addr);
```

If you only want to check if the address data consists of all zeros the function `nl_addr_iszero()` is a shortcut to that.

```
#include <netlink/addr.h>


int nl_addr_iszero(struct nl_addr *addr);
```

9.1.1. Address Prefix Length

Although this functionality is somewhat specific to routing it has been implemented here. Addresses can have a prefix length assigned which implies that only the first n bits are of importance. This is f.e. used to implement subnets.

Use set functions `nl_addr_set_prefixlen()` and `nl_addr_get_prefixlen()` to work with prefix lengths.

```
#include <netlink/addr.h>


void nl_addr_set_prefixlen(struct nl_addr *addr, int n);

unsigned int nl_addr_get_prefixlen(struct nl_addr *addr);
```

The default prefix length is set to (address length * 8)

Address Helpers

Several functions exist to help when dealing with addresses. The function `nl_addr_cmp()` compares two addresses and returns an integer less than, equal to or greater than zero without considering the prefix length at all. If you want to consider the prefix length, use the function `nl_addr_cmp_prefix()`.

```
#include <netlink/addr.h>


int nl_addr_cmp(struct nl_addr *addr, struct nl_addr *addr);

int nl_addr_cmp_prefix(struct nl_addr *addr, struct nl_addr *addr);
```

If an abstract address needs to presented to the user it should be done in a human readable format which differs depending on the address family. The function `nl_addr2str()` takes care of this by calling the appropriate conversion functions internaly. It expects a `buf` of length `size` to write the character string into and returns a pointer to `buf` for easy `printf()` usage.

```
#include <netlink/addr.h>


char *nl_addr2str(struct nl_addr *addr, char *buf, size_t size);
```

If the address family is unknown, the address data will be printed in hexadecimal format `AA:BB:CC:DD:...`

Often the only way to figure out the address family is by looking at the length of the address. The function `nl_addr_guess_family()` does just this and returns the address family guessed based on the address size.

```
#include <netlink/addr.h>


int nl_addr_guess_family(struct nl_addr *addr);
```

Before allocating an address you may want to check if the character string actually represents a valid address of the address family you are expecting. The function `nl_addr_valid()` can be used for that, it returns 1 if the supplised `addr` is a valid address in the context of `family`. See `inet_pton(3)`, `dnet_pton(3)` for more information on valid adddress formats.

```
#include <netlink/addr.h>


int nl_addr_valid(char *addr, int family);
```

9.2. Abstract Data

The abstract data type is a trivial datatype with the primary purpose to simplify usage of netlink attributes of arbitary length.

Allocation of a Data Object

The function `nl_data_alloc()` alloctes a new abstract data object and fill it with the provided data. `nl_data_alloc_attr()` does the same but bases the data on the payload of a netlink attribute. New data objects can also be allocated by cloning existing ones by using `nl_data_clone()`.

```c
struct nl_data *nl_data_alloc(void *buf, size_t size);

struct nl_data *nl_data_alloc_attr(struct nlattr *attr);

struct nl_data *nl_data_clone(struct nl_data *data);

void nl_data_free(struct nl_data *data);
```

Access to Data

The function `nl_data_get()` returns a pointer to the data, the size of data is returned by `nl_data_get_size()`.

```c
void *nl_data_get(struct nl_data *data);

size_t nl_data_get_size(struct nl_data *data);
```

Data Helpers

The function nl_data_append() reallocates the internal data buffers and appends the specified `buf` to the existing data.

```c
int nl_data_append(struct nl_data *data, void *buf, size_t size);
```

> Any call to `nl_data_append()` invalidates all pointers returned by `nl_data_get()` of the same data object.

```c
int nl_data_cmp(struct nl_data *data, struct nl_data *data);
```

Version 3.2
Last updated 2014-01-21 20:42:44 CET

OLEG KUTKOV PERSONAL BLOG

Programming, electronics and diy projects

Linux Kernel, Linux System Development, Networking, Software

Getting Linux routing table using netlink

Oleg Kutkov / March 24, 2019

In the previous article, we discussed the monitoring of the network interfaces using Netlink. Now it's time to do something more complex and interesting.
Let's discover how to get and print the system routing table like "ip route" command.

The routing table is a runtime in-memory data structure that stores the routes (and in some cases, metrics associated with those routes) to particular network destinations. This is very important with TCP/IP. Using this table network stack decides where and how to put packets for a specified network.
Linux kernel supports multiple routing tables. Beyond the two commonly used routing tables (the local and main routing tables), the kernel supports 252 additional routing tables.
The multiple routing table system provides a flexible infrastructure on top of which to implement policy routing. By allowing multiple traditional routing tables (keyed primarily to destination address) to be combined with the routing policy database (RPDB) (keyed primarily to source address), the kernel supports a well-known and well-understood interface while simultaneously expanding and extending its routing capabilities.

To get Linux main routing table, we can use commands "**route -n**", "**netstat -rn**" and "**ip route**":

> $route -n

The first utility is used the classic **ioctl** interface to get information from the kernel. This way is limited and became deprecated now.
Instead of **ioctl** "**ip route**" is based on the Netlink sockets, and now we discover how it works.

Like in the monitor, everything starts with the creation of the Netlink socket and binding. Binding here is essential. This allows us to execute this program as a normal user.

```
struct sockaddr_nl saddr;

/* Open raw socket for the NETLINK_ROUTE protocol */
int nl_sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

if (nl_sock < 0) {
    perror("Failed to open netlink socket");
    return -1;
}

memset(&saddr, 0, sizeof(saddr));

saddr.nl_family = AF_NETLINK;
saddr.nl_pid = getpid();

/* Bind current process to the netlink socket */
if (bind(nl_sock, (struct sockaddr *)&saddr, sizeof(saddr)) < 0) {
    perror("Failed to bind to netlink socket");
    close(nl_sock);
    return -1;
}
```

Now it's time to send the request to the kernel.

```
/* Request struct */
struct {
    struct nlmsghdr nlh;  /* Netlink header */
    struct rtmsg rtm;     /* Payload - route message */
} nl_request;

nl_request.nlh.nlmsg_type = RTM_GETROUTE;  /* We wish to get routes */
nl_request.nlh.nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
nl_request.nlh.nlmsg_len = sizeof(nl_request);
```

CATEGORIES

- Allsky camera (5)
- Astro tools (9)
- Automotive (1)
- Electronics (30)
- Firmware (3)
- hardware (8)
- Linux kernel (8)
- Linux system development (11)
- Networking (9)
- Radio & antennas (14)
- Radioastronomy (5)
- Reverse engineering (5)
- Software (17)
- Uncategorized (1)

RECENT POSTS

- Data logger for UNI-T UT800 multimeters
- How to add Ethernet port to the Gen 2 Starlink router
- EQMOD adapter for telescopes. Version 2
- Initial analysis of the Starlink router gen2
- Reverse engineering of the Starlink Ethernet adapter

RECENT COMMENTS

- Andrii on Reverse engineering of the Starlink Ethernet adapter
- Oleg Kutkov on C++ in Linux kernel
- Johannes on C++ in Linux kernel
- Oleg Kutkov on C++ in Linux kernel
- Johannes on C++ in Linux kernel

OLEG KUTKOV PERSONAL BLOG

- Home

```
nl_request.nlh.nlmsg_seq = time(NULL);
nl_request.rtm.rtm_family = AF_INET;

ssize_t sent = send(sock, &nl_request, sizeof(nl_request), 0);

if (sent < 0) {
    perror("Failed to perfom request");
    close(nl_sock);
    return -1;
}
```

We need to declare a request structure that describes the Netlink packet with a header and some payload – actual message.
In the header, we specify what we need with RTM_GETROUTE as a message type that can return the main routing table.
Additional flags "NLM_F_REQUEST | NLM_F_DUMP" telling the kernel that this is a dump request.
As rtm_family we can specify AF_INET if we want to get the table for IPv4 protocol and AF_INET6 for IPv6.

Getting kernel response is more complex.
We need to execute vectored reading using already known **recvmsg** and struct **iovec**.
For simplification reasons, this code is split into the 3 functions.

On the lowest level is a simple wrapper around **recvmsg**, which more robust and can handle "busy" states.

```
int rtnl_receive(int fd, struct msghdr *msg, int flags)
{
    int len;

    /* Try to read the message in case of busy or interrupted call */
    do {
        len = recvmsg(fd, msg, flags);
    } while (len < 0 && (errno == EINTR || errno == EAGAIN));

    if (len < 0) {
        perror("Netlink receive failed");
        return -errno;
    }

    if (len == 0) {
        perror("EOF on netlink");
        return -ENODATA;
    }

    return len;
}
```

Receive is called from the rtnl_recvmsg function, which reads the message size first, then allocating buffer using size info and reading the actual response message.

Here struct **iovec** is passed from the top-level function get_route_dump_response.

```
int get_route_dump_response(int sock)
{
    struct sockaddr_nl nladdr;
    struct iovec iov;
    struct msghdr msg = {
        .msg_name = &nladdr,
        .msg_namelen = sizeof(nladdr),
        .msg_iov = &iov,
        .msg_iovlen = 1,
    };

    char *buf;
    int dump_intr = 0;

    /* Get the message */
    int status = rtnl_recvmsg(sock, &msg, &buf);

    /* Pointer to the messages head */
    struct nlmsghdr *h = (struct nlmsghdr *)buf;
    int msglen = status;

    printf("Main routing table IPv4\n");

    /* Iterate through all messages in buffer */
    while (NLMSG_OK(h, msglen)) {
        if (h->nlmsg_flags & NLM_F_DUMP_INTR) {
            perror("Dump was interrupted\n");
            free(buf);
            return -1;
        }

        if (nladdr.nl_pid != 0) {
```


Starlink repairs archive

About me

```
            continue;
        }

        if (h->nlmsg_type == NLMSG_ERROR) {
            perror("netlink reported error");
            free(buf);
        }

        /* Decode and print single message */
        print_route(h);

        h = NLMSG_NEXT(h, msglen);
    }

    free(buf);

    return status;
}
```

Netlink messages can be split into parts, so this function is trying to read all those parts.
After successfully receiving the message, we can call the printer function print_route.

```
void parse_rtattr(struct rtattr *tb[], int max, struct rtattr *rta, int len)
{
    memset(tb, 0, sizeof(struct rtattr *) * (max + 1));

    while (RTA_OK(rta, len)) {
        if (rta->rta_type <= max) {
            tb[rta->rta_type] = rta;
        }

        rta = RTA_NEXT(rta,len);
    }
}

static inline int rtm_get_table(struct rtmsg *r, struct rtattr **tb)
{
    __u32 table = r->rtm_table;

    if (tb[RTA_TABLE]) {
        table = *(__u32 *)RTA_DATA(tb[RTA_TABLE]);
    }

    return table;
}
```

The printer function is using two auxiliary functions for parsing the message. One of these functions is already known from the network monitor.
Both these functions perform simple iteration on the memory, some conversion of the types, and alignment.
Finally, we ready to print the route.

```
void print_route(struct nlmsghdr* nl_header_answer)
{
    struct rtmsg* r = NLMSG_DATA(nl_header_answer);
    int len = nl_header_answer->nlmsg_len;
    struct rtattr* tb[RTA_MAX+1];
    int table;
    char buf[256];

    len -= NLMSG_LENGTH(sizeof(*r));

    if (len < 0) {
        perror("Wrong message length");
        return;
    }

    /* Parse message */
    parse_rtattr(tb, RTA_MAX, RTM_RTA(r), len);

    table = rtm_get_table(r, tb);

    if (r->rtm_family != AF_INET && table != RT_TABLE_MAIN) {
        return;
    }

    /* Read destination address from the tb at RTA_DST index */
    if (tb[RTA_DST]) {
        if ((r->rtm_dst_len != 24) && (r->rtm_dst_len != 16)) {
            return;
        }
```

```
        /* Print readable address using inet_ntop */
        printf("%s/%u ", inet_ntop(r->rtm_family, RTA_DATA(tb[RTA_DST]), buf, sizeof(buf)), r->r

    } else if (r->rtm_dst_len) {
        printf("0/%u ", r->rtm_dst_len);
    } else {
        printf("default ");
    }

    /* Do the same thing for rest of the fields */
    if (tb[RTA_GATEWAY]) {
        printf("via %s", inet_ntop(r->rtm_family, RTA_DATA(tb[RTA_GATEWAY]), buf, sizeof(buf)));
    }

    if (tb[RTA_OIF]) {
        char if_nam_buf[IF_NAMESIZE];
        int ifidx = *(__u32 *)RTA_DATA(tb[RTA_OIF]);

        printf(" dev %s", if_indextoname(ifidx, if_nam_buf));
    }

    if (tb[RTA_SRC]) {
        printf("src %s", inet_ntop(r->rtm_family, RTA_DATA(tb[RTA_SRC]), buf, sizeof(buf)));
    }

    printf("\n");
}
```

In the beginning, this function performing parsing of the message and some checks.
Then we can easily access different parts of the routing message using array and indices with readable defines.

To get the IPv4 address in a human-readable text form is used standard inet_ntop.
The network interface is presented as numeric indexes and converted to the readable form (like “eth0”) using if_indextoname.
This function required a pre-allocated buffer with IF_NAMESIZE size.

Now all together:

```
/*
 *
 */

#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/rtnetlink.h>

int rtnl_receive(int fd, struct msghdr *msg, int flags)
{
    int len;

    do {
        len = recvmsg(fd, msg, flags);
    } while (len < 0 && (errno == EINTR || errno == EAGAIN));

    if (len < 0) {
        perror("Netlink receive failed");
        return -errno;
    }

    if (len == 0) {
        perror("EOF on netlink");
        return -ENODATA;
    }

    return len;
}

static int rtnl_recvmsg(int fd, struct msghdr *msg, char **answer)
{
    struct iovec *iov = msg->msg_iov;
    char *buf;
    int len;

    iov->iov_base = NULL;
```

```
    iov->iov_len = 0;

    len = rtnl_receive(fd, msg, MSG_PEEK | MSG_TRUNC);

    if (len < 0) {
        return len;
    }

    buf = malloc(len);

    if (!buf) {
        perror("malloc failed");
        return -ENOMEM;
    }

    iov->iov_base = buf;
    iov->iov_len = len;

    len = rtnl_receive(fd, msg, 0);

    if (len < 0) {
        free(buf);
        return len;
    }

    *answer = buf;

    return len;
}

void parse_rtattr(struct rtattr *tb[], int max, struct rtattr *rta, int len)
{
    memset(tb, 0, sizeof(struct rtattr *) * (max + 1));

    while (RTA_OK(rta, len)) {
        if (rta->rta_type <= max) {
            tb[rta->rta_type] = rta;
        }

        rta = RTA_NEXT(rta,len);
    }
}

static inline int rtm_get_table(struct rtmsg *r, struct rtattr **tb)
{
    __u32 table = r->rtm_table;

    if (tb[RTA_TABLE]) {
        table = *(__u32 *)RTA_DATA(tb[RTA_TABLE]);
    }

    return table;
}

void print_route(struct nlmsghdr* nl_header_answer)
{
    struct rtmsg* r = NLMSG_DATA(nl_header_answer);
    int len = nl_header_answer->nlmsg_len;
    struct rtattr* tb[RTA_MAX+1];
    int table;
    char buf[256];

    len -= NLMSG_LENGTH(sizeof(*r));

    if (len < 0) {
        perror("Wrong message length");
        return;
    }

    parse_rtattr(tb, RTA_MAX, RTM_RTA(r), len);

    table = rtm_get_table(r, tb);

    if (r->rtm_family != AF_INET && table != RT_TABLE_MAIN) {
        return;
    }

    if (tb[RTA_DST]) {
        if ((r->rtm_dst_len != 24) && (r->rtm_dst_len != 16)) {
            return;
```

```
        }

        printf("%s/%u ", inet_ntop(r->rtm_family, RTA_DATA(tb[RTA_DST]), buf, sizeof(buf)), r->r

    } else if (r->rtm_dst_len) {
        printf("0/%u ", r->rtm_dst_len);
    } else {
        printf("default ");
    }

    if (tb[RTA_GATEWAY]) {
        printf("via %s", inet_ntop(r->rtm_family, RTA_DATA(tb[RTA_GATEWAY]), buf, sizeof(buf)));
    }

    if (tb[RTA_OIF]) {
        char if_nam_buf[IF_NAMESIZE];
        int ifidx = *(__u32 *)RTA_DATA(tb[RTA_OIF]);

        printf(" dev %s", if_indextoname(ifidx, if_nam_buf));
    }

    if (tb[RTA_SRC]) {
        printf("src %s", inet_ntop(r->rtm_family, RTA_DATA(tb[RTA_SRC]), buf, sizeof(buf)));
    }

    printf("\n");
}

int open_netlink()
{
    struct sockaddr_nl saddr;

    int sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

    if (sock < 0) {
        perror("Failed to open netlink socket");
        return -1;
    }

    memset(&saddr, 0, sizeof(saddr));

    saddr.nl_family = AF_NETLINK;
    saddr.nl_pid = getpid();

    if (bind(sock, (struct sockaddr *)&saddr, sizeof(saddr)) < 0) {
        perror("Failed to bind to netlink socket");
        close(sock);
        return -1;
    }

    return sock;
}

int do_route_dump_requst(int sock)
{
    struct {
        struct nlmsghdr nlh;
        struct rtmsg rtm;
    } nl_request;

    nl_request.nlh.nlmsg_type = RTM_GETROUTE;
    nl_request.nlh.nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
    nl_request.nlh.nlmsg_len = sizeof(nl_request);
    nl_request.nlh.nlmsg_seq = time(NULL);
    nl_request.rtm.rtm_family = AF_INET;

    return send(sock, &nl_request, sizeof(nl_request), 0);
}

int get_route_dump_response(int sock)
{
    struct sockaddr_nl nladdr;
    struct iovec iov;
    struct msghdr msg = {
        .msg_name = &nladdr,
        .msg_namelen = sizeof(nladdr),
        .msg_iov = &iov,
        .msg_iovlen = 1,
    };
```

```
    char *buf;
    int dump_intr = 0;

    int status = rtnl_recvmsg(sock, &msg, &buf);

    struct nlmsghdr *h = (struct nlmsghdr *)buf;
    int msglen = status;

    printf("Main routing table IPv4\n");

    while (NLMSG_OK(h, msglen)) {
        if (h->nlmsg_flags & NLM_F_DUMP_INTR) {
            fprintf(stderr, "Dump was interrupted\n");
            free(buf);
            return -1;
        }

        if (nladdr.nl_pid != 0) {
            continue;
        }

        if (h->nlmsg_type == NLMSG_ERROR) {
            perror("netlink reported error");
            free(buf);
        }

        print_route(h);

        h = NLMSG_NEXT(h, msglen);
    }

    free(buf);

    return status;
}

int main()
{
    int nl_sock = open_netlink();

    if (do_route_dump_requst(nl_sock) < 0) {
        perror("Failed to perfom request");
        close(nl_sock);
        return -1;
    }

    get_route_dump_response(nl_sock);

    close (nl_sock);

    return 0;
}
```

Compilation and execution:

> $ gcc -g -ggdb routing.c -o routing
> $ ./routing
> Main routing table IPv4
> default via 192.168.8.1 dev eth0
> 169.254.0.0/16 dev eth0
> 192.168.8.0/24 dev eth0

That's it.
In the next article, I will show how to delete and add new routes.

Share this:

Related

Modifying Linux network routes using netlink
August 29, 2019
In "Linux kernel"

Monitoring Linux networking state using netlink
February 14, 2018
In "Linux kernel"

Writing a PCI device driver for Linux
January 7, 2021
In "Linux kernel"

Tagged kernel, linux, netlink, netwo, routing

Related Posts

Data logger for UNI-T UT800 multimeters

July 18, 2022

How to add Ethernet port to the Gen 2 Starlink router

April 30, 2022

Initial analysis of the Starlink router gen2

April 10, 2022

About Oleg Kutkov

View all posts by Oleg Kutkov →

Dish antenna for the amateur radioastronomy

Simple logger with STDOUT, Files and syslog support for C projects in Linux

9 thoughts on “Getting Linux routing table using netlink”

Mike B says:

June 2, 2019 at 11:14 pm

very useful, waiting for next articles about netlink

Loading...

Reply

Oleg Kutkov says:

June 3, 2019 at 10:59 am

Thanks!
New article is upcoming. Stay tuned.

Loading...

Reply

Pingback: Modifying Linux network routes using netlink - Oleg Kutkov personal blog

Savva says:

August 18, 2020 at 5:38 am

Hey Oleg, thanks for the post! Very useful. I’m using this code on an ad-hoc network (with olsrd). I am seeing at least a 20 second delay between an ip address appearing among the results of the “route -n” command and the same ip address appearing among the outputs of the get_route_dump_response() function. Any insights as to why this might be happening?

Loading...

Reply

Oleg Kutkov says:

August 18, 2020 at 11:05 pm

Hello!
Can you repeat your tests with command **ip r** ?
This command is also based on netlink. Command “**route -n**” uses old-style ioctl requests.

Loading...

Reply

Digvijay says:

September 23, 2020 at 3:24 pm

Very useful. Nicely coded.
I think there should be a return statement in the following block otherwise it can crash:

```
if (h->nlmsg_type == NLMSG_ERROR) {
perror("netlink reported error");
free(buf);
}
```

★ Loading...

Reply

Sandy says:

November 9, 2020 at 10:57 pm

This code is not working for IPV6 if I change family to AF_INET6 as specified above.
nl_request.rtm.rtm_family = AF_INET6

Please help

★ Loading...

Reply

Oleg Kutkov says:

November 9, 2020 at 11:31 pm

Hello. It's no problem.
This code is just adopted for the IPv4, you can easily switch to IPv6.
Just replace everywhere AF_INET to AF_INET6 + remove the address length check at line 119:

```
if ((r->rtm_dst_len != 24) && (r->rtm_dst_len != 16)) {
return;
}
```

This code just skips your IPv6 addresses due to IPv6 rtm_dst_len == 128.
Without this check IPv6 table can be printed without any problems:

```
$ ./print_route
Main routing table IPv6
::1/128 dev lo
fe80::/64 dev enp4s0
::1/128 dev lo
fe80::d448:77d5:68fa:bf6b/128 dev enp4s0
ff00::/8 dev enp4s0
```

You can completely remove this check or better adopt it to the 128 bit address lengths.

★ Loading...

Reply

Ort says:

January 26, 2022 at 10:06 am

Hi Oleg,
Thanks for the useful information!

Is there any way to know if a specific route exists in the routing table? For example to use RTM_GETROUTE but also to specify the ip address of the route we're searching? I don't want all the entries in the routing table, only a specific entry.
Thanks!

★ Loading...

Reply

Leave a Reply

Enter your comment here...

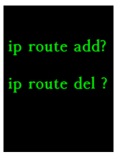
OLEG KUTKOV PERSONAL BLOG

Programming, electronics and diy projects

Linux Kernel, Linux System Development, Networking, Software

Modifying Linux network routes using netlink

Oleg Kutkov / August 29, 2019



Last time we talked about getting a Linux routing table with a simple Netlink code.
Now it's time to do more interesting stuff. Let's add and delete some routes using the power of the Netlink!

At the end of this article, we will create a command-line utility with syntax similar to **ip route** command, which can add and delete custom routes.

Like in previous examples, everything starts with a Netlink socket.

```
/* Open netlink socket */
int open_netlink()
{
    int sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

    if (sock < 0) {
        perror("Failed to open netlink socket");
        return -1;
    }

    return sock;
}
```

And that's it!
We don't need to bind to the socket or do some other things. All we have to do is to build a special message and send it to the Netlink socket.

Let's describe the message structure.

```
struct {
    struct nlmsghdr n;
    struct rtmsg r;
    char buf[4096];
} nl_request;
```

Now we need to configure some fields of **nlmsghdr** and **rtmsg**.
Some of them are basic and used both in "add" and "delete" requests, but some contain actual command of what to do.

Basic initialization:

```
nl_request.n.nlmsg_len = NLMSG_LENGTH(sizeof(struct rtmsg));
nl_request.r.rtm_table = RT_TABLE_MAIN;
nl_request.r.rtm_scope = RT_SCOPE_NOWHERE;
nl_request.n.nlmsg_flags = 0;
```

Let's specify what we want to do – add or remove the route.
To add a new route, specify **nlmsg_type** as **RTM_NEWROUTE**

```
nl_request.n.nlmsg_type = RTM_NEWROUTE;
```

And **RTM_DELROUTE** in case of deleting

```
nl_request.n.nlmsg_type = RTM_DELROUTE;
```

Additionally, we can specify flags for the "add" operation, combining with **NLM_F_REQUEST** flag.

> NLM_F_REPLACE Replace existing matching object.
> NLM_F_EXCL Don't replace if the object already exists.

OLEG KUTKOV PERSONAL BLOG

> NLM_F_CREATE Create object if it doesn't already exist.
> NLM_F_APPEND Add to the end of the object list.

Create a new routing table entry and don't replace already existing record:

```
nl_request.n.nlmsg_flags = NLM_F_REQUEST | NLM_F_CREATE | NLM_F_EXCL
```

Also, we need to specify the route type in case of adding a new one.

> RTN_UNSPEC unknown route
> RTN_UNICAST a gateway or direct route
> RTN_LOCAL a local interface route
> RTN_BROADCAST a local broadcast route (sent as a broadcast)
> RTN_ANYCAST a local broadcast route (sent as a unicast)
> RTN_MULTICAST a multicast route
> RTN_BLACKHOLE a packet dropping route
> RTN_UNREACHABLE an unreachable destination
> RTN_PROHIBIT a packet rejection route
> RTN_THROW continue routing lookup in another table
> RTN_NAT a network address translation rule

In simple cases, we can use **RTN_UNICAST**:

```
if (nl_request.n.nlmsg_type != RTM_DELROUTE) {
    nl_request.r.rtm_type = RTN_UNICAST;
}
```

Now the most interesting part – adding route details. It may vary depending on what we want. We can specify the target network, gateway, network interface, or just gateway.

According to these details – protocol family, scope, and address length should be set.

Let's describe a simple case with the IPv4 route.

```
nl_request.r.rtm_family = AF_INET;
nl_request.r.rtm_scope = RT_SCOPE_LINK;
```

If we add a route to some network, not a default gateway – we also need to specify destination address length in Bits. It's simply 32 for IPv4 and 128 for IPv6.

```
nl_request.r.rtm_dst_len = 32;
```

Typically IP addresses are represented in human-readable text forms, but Netlink accepts only binary format.

To deal with this, we can use **inet_pton** function from **arpa/inet.h.**
This function supports converting both IPv4 and IPv6 into binary form.

Conversion of the **AF_INET** (IPv4) address 192.168.1.0 into binary form and put it to **data** buffer:

```
#include <arpa/inet.h>

unsigned char data[sizeof(struct in6_addr)];

inet_pton(AF_INET, "192.168.1.0", data);
```

In some cases, we also need to specify the outgoing network interface.
User-friendly names like "eth0" should also be converted to numeric indexes. Here we can use **if_nametoindex** from **net/if.h.**

```
#include <net/if.h>

int if_idx = if_nametoindex("eth0");
```

To add IP addresses data and interface index to our Netlink request, we need to use a special function that actually acts as a reverse of the **parse_rtattr** from the previous articles.

```
/* Add new data to rtattr */
int rtattr_add(struct nlmsghdr *n, int maxlen, int type, const void *data, int alen)
{
    int len = RTA_LENGTH(alen);
    struct rtattr *rta;

    if (NLMSG_ALIGN(n->nlmsg_len) + RTA_ALIGN(len) > maxlen) {
        fprintf(stderr, "rtattr_add error: message exceeded bound of %d\n", maxlen);
        return -1;
    }

    rta = NLMSG_TAIL(n);
```

Starlink repairs archive
About me

SOCIAL

SITE SEARCH

Search …
Search

Support author

You can send donations on PayPal using my email contact@olegkutkov.me

Thank you for your support!

```
    rta->rta_type = type;
    rta->rta_len = len;

    if (alen) {
        memcpy(RTA_DATA(rta), data, alen);
    }

    n->nlmsg_len = NLMSG_ALIGN(n->nlmsg_len) + RTA_ALIGN(len);

    return 0;
}
```

And here is how to use this function and add network interface-id to our **nl_request** structure:

```
rtattr_add(&nl_request.n, sizeof(nl_request), RTA_OIF, &if_idx, sizeof(int));
```

Add gateway:

```
rtattr_add(&nl_request.n, sizeof(nl_request), RTA_GATEWAY, gw_bin_data, 16);
```

gw_bin_data is IP address binary data acquired with inet_pton, and 16 is IPv4 address length in bytes (not a bit in this case), for IPv6 use 16.

We can use attribute type RTA_DST or RTA_NEWDST on the newest Linux kernels to add a destination network. Just check what's available on your system.

```
rtattr_add(&nl_request.n, sizeof(nl_request), /*RTA_NEWDST*/ RTA_DST, dst_net_bin_data, 16);
```

Please note that there are some rules with a combination of these attributes. For the default gateway, we DON'T need to specify the destination network and even network interface id. The kernel can figure it out by itself.

Now send this message to the socket.

```
send(sock, &nl_request, sizeof(nl_request), 0);
```

A complete example is below.
I decided to write a program with quite a complex command-line interface that can act as the **ip route** tool. All arguments parsing is implemented withing **main()** function, parser requires strict order of the params.
This program might be buggy and imperfect, but this is just an example that can do the job 🙂

```
/*
 *
 */

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/rtnetlink.h>

/* Open netlink socket */
int open_netlink()
{
    struct sockaddr_nl saddr;

    int sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

    if (sock < 0) {
        perror("Failed to open netlink socket");
        return -1;
    }

    memset(&saddr, 0, sizeof(saddr));

    return sock;
}

/* Helper structure for ip address data and attributes */
typedef struct {
    char family;
    char bitlen;
    unsigned char data[sizeof(struct in6_addr)];
} _inet_addr;

/* */
```

```
#define NLMSG_TAIL(nmsg) \
    ((struct rtattr *) (((void *) (nmsg)) + NLMSG_ALIGN((nmsg)->nlmsg_len)))

/* Add new data to rtattr */
int rtattr_add(struct nlmsghdr *n, int maxlen, int type, const void *data, int alen)
{
    int len = RTA_LENGTH(alen);
    struct rtattr *rta;

    if (NLMSG_ALIGN(n->nlmsg_len) + RTA_ALIGN(len) > maxlen) {
        fprintf(stderr, "rtattr_add error: message exceeded bound of %d\n", maxlen);
        return -1;
    }

    rta = NLMSG_TAIL(n);
    rta->rta_type = type;
    rta->rta_len = len;

    if (alen) {
        memcpy(RTA_DATA(rta), data, alen);
    }

    n->nlmsg_len = NLMSG_ALIGN(n->nlmsg_len) + RTA_ALIGN(len);

    return 0;
}

int do_route(int sock, int cmd, int flags, _inet_addr *dst, _inet_addr *gw, int def_gw, int if_i
{
    struct {
        struct nlmsghdr n;
        struct rtmsg r;
        char buf[4096];
    } nl_request;

    /* Initialize request structure */
    nl_request.n.nlmsg_len = NLMSG_LENGTH(sizeof(struct rtmsg));
    nl_request.n.nlmsg_flags = NLM_F_REQUEST | flags;
    nl_request.n.nlmsg_type = cmd;
    nl_request.r.rtm_family = dst->family;
    nl_request.r.rtm_table = RT_TABLE_MAIN;
    nl_request.r.rtm_scope = RT_SCOPE_NOWHERE;

    /* Set additional flags if NOT deleting route */
    if (cmd != RTM_DELROUTE) {
        nl_request.r.rtm_protocol = RTPROT_BOOT;
        nl_request.r.rtm_type = RTN_UNICAST;
    }

    nl_request.r.rtm_family = dst->family;
    nl_request.r.rtm_dst_len = dst->bitlen;

    /* Select scope, for simplicity we supports here only IPv6 and IPv4 */
    if (nl_request.r.rtm_family == AF_INET6) {
        nl_request.r.rtm_scope = RT_SCOPE_UNIVERSE;
    } else {
        nl_request.r.rtm_scope = RT_SCOPE_LINK;
    }

    /* Set gateway */
    if (gw->bitlen != 0) {
        rtattr_add(&nl_request.n, sizeof(nl_request), RTA_GATEWAY, &gw->data, gw->bitlen / 8);
        nl_request.r.rtm_scope = 0;
        nl_request.r.rtm_family = gw->family;
    }

    /* Don't set destination and interface in case of default gateways */
    if (!def_gw) {
        /* Set destination network */
        rtattr_add(&nl_request.n, sizeof(nl_request), /*RTA_NEWDST*/ RTA_DST, &dst->data, dst->b

        /* Set interface */
        rtattr_add(&nl_request.n, sizeof(nl_request), RTA_OIF, &if_idx, sizeof(int));
    }

    /* Send message to the netlink */
    return send(sock, &nl_request, sizeof(nl_request), 0);
}

/* Simple parser of the string IP address
```

```c
 */
int read_addr(char *addr, _inet_addr *res)
{
    if (strchr(addr, ':')) {
        res->family = AF_INET6;
        res->bitlen = 128;
    } else {
        res->family = AF_INET;
        res->bitlen = 32;
    }

    return inet_pton(res->family, addr, res->data);
}

#define NEXT_CMD_ARG() do { argv++; if (--argc <= 0) exit(-1); } while(0)

int main(int argc, char **argv)
{
    int default_gw = 0;
    int if_idx = 0;
    int nl_sock;
    _inet_addr to_addr = { 0 };
    _inet_addr gw_addr = { 0 };

    int nl_cmd;
    int nl_flags;

    /* Parse command line arguments */
    while (argc > 0) {
        if (strcmp(*argv, "add") == 0) {
            nl_cmd = RTM_NEWROUTE;
            nl_flags = NLM_F_CREATE | NLM_F_EXCL;

        } else if (strcmp(*argv, "del") == 0) {
            nl_cmd = RTM_DELROUTE;
            nl_flags = 0;

        } else if (strcmp(*argv, "to") == 0) {
            NEXT_CMD_ARG(); /* skip "to" and jump to the actual destination addr */

            if (read_addr(*argv, &to_addr) != 1) {
                fprintf(stderr, "Failed to parse destination network %s\n", *argv);
                exit(-1);
            }

        } else if (strcmp(*argv, "dev") == 0) {
            NEXT_CMD_ARG(); /* skip "dev" */

            if_idx = if_nametoindex(*argv);

        } else if (strcmp(*argv, "via") == 0) {
            NEXT_CMD_ARG(); /* skip "via"*/

            /* Instead of gw address user can set here keyword "default" */
            /* Try to read this keyword and jump to the actual gateway addr */
            if (strcmp(*argv, "default") == 0) {
                default_gw = 1;
                NEXT_CMD_ARG();
            }

            if (read_addr(*argv, &gw_addr) != 1) {
                fprintf(stderr, "Failed to parse gateway address %s\n", *argv);
                exit(-1);
            }
        }

        argc--; argv++;
    }

    nl_sock = open_netlink();

    if (nl_sock < 0) {
        exit(-1);
    }

    do_route(nl_sock, nl_cmd, nl_flags, &to_addr, &gw_addr, default_gw, if_idx);

    close (nl_sock);
```

```
    return 0;
}
```

Before testing, let's print the current routing table:

> $ ip route
> default via 192.168.8.1 dev eth0
> 192.168.8.0/24 dev eth0 proto kernel scope link src 192.168.8.2

Now compile our program.

> gcc set_route.c -o set_route

Add a new route to network 192.168.1.0 via eth0:

> $ sudo ./set_route add to 192.168.1.0 dev eth0
>
> $ ip route
> default via 192.168.8.1 dev eth0
> 192.168.1.0 dev eth0 proto none scope link
> 192.168.8.0/24 dev eth0 proto kernel scope link src 192.168.8.2

Works!

Delete this route:

> $ sudo ./set_route del to 192.168.1.0 dev eth0
>
> $ ip route
> default via 192.168.8.1 dev eth0
> 192.168.8.0/24 dev eth0 proto kernel scope link src 192.168.8.2

More examples.

Add route to 192.168.1.0 using eth0 and 192.168.8.1 gateway: **sudo ./set_route add to 192.168.8.0 dev eth0 via 192.168.8.1**
Delete this route: **sudo ./set_route del to 192.168.8.0 dev eth0 via 192.168.8.1**

To add default gateway via 192.168.8.1 just: **sudo ./set_route add via default 192.168.8.1**

Thanks for reading!

Related

Getting Linux routing table using netlink
March 24, 2019
In "Linux kernel"

Monitoring Linux networking state using netlink
February 14, 2018
In "Linux kernel"

Linux block device driver
February 10, 2020
In "Linux kernel"

Tagged linux, netlink, network, routes

Related Posts

Data logger for UNI-T UT800 multimeters
July 18, 2022

How to add Ethernet port to the Gen 2 Starlink router
April 30, 2022

Initial analysis of the Starlink router gen2
April 10, 2022

About Oleg Kutkov

View all posts by Oleg Kutkov →

Hello HTTPS!

Printing sk_buff data

2 thoughts on “Modifying Linux network routes using netlink”

Abhishek Sagar says:

November 24, 2019 at 1:46 am

Delete route is not working.

I have the following entry in Routing table :
192.168.8.7 192.168.117.2 255.255.255.255 UGH 0 0 0 ens33

sudo ./set_route del to 192.168.8.7 dev ens33
The above entry continue to exist.

Even addition case, sometimes it dont work.

★ Loading...

Reply

Venkateswaran says:

May 5, 2021 at 11:35 am

It’s source specific routing

★ Loading...

Reply

Leave a Reply

Enter your comment here...

This site uses Akismet to reduce spam. Learn how your comment data is processed.

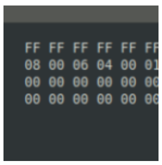
OLEG KUTKOV PERSONAL BLOG

Programming, electronics and diy projects

Linux Kernel, Linux System Development, Networking, Software

Printing sk_buff data

Oleg Kutkov / October 17, 2019



Sometimes when working with network packets inside the Linux kernel, it might be very useful to print packet contents to see what is actually going on.
Here I'm describing how to print packet from **sk_buff** structure and analyze this data with Wireshark.
In this short note, I will not describe capturing the packets inside the kernel but only show how to print the **sk_buff**.

Struct **sk_buff** is a famous Linux kernel structure that holds network packets (with all headers) during travel through the Linux network stack.

As you probably know, **sk_buff** contains a few pointers representing different regions in the one memory that contains all data of the packet.
Pointers '**data**' and '**tail**' may be changed on different layers of the network stack.

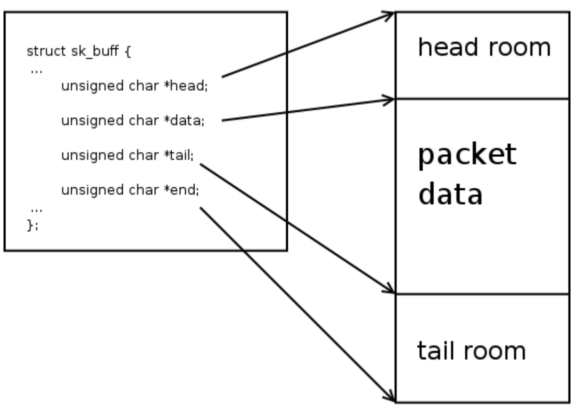


Image credits: kernel.org

Let's describe the reception of the packet. The initial state '**data**' points directly to the beginning of the packet on the Ethernet header.
It's the L2 layer of the network stack. The L3 layer 'data' pointer is incremented by the Ethernet header's size and points on the IP header. And so on.
But we still can access the Start of the packet and Ethernet header because data is still here.
Linux kernel provides a set of functions to access the different layers headers and **sk_buff** pointers manipulation. It's highly recommended to use these functions instead of direct pointers access. Please refer ***include/linux/skbuff.h***

If we want to print a full packet with a network header, we need to reach the mac header's pointer.
Function **skb_mac_header()** can help us.
Let's check how this function is implemented.

```
static inline unsigned char *skb_mac_header(const struct sk_buff *skb)
{
    return skb->head + skb->mac_header;
}
```

As you can see, this function is straightforward. The result is offset from the **sk_buff** memory start (head) by some value from **the mac_header** variable. This variable initialized during packet reception in a driver (or in the stack during packet generation and transmission).

You can see function **skb_reset_mac_header()** which set mac_header to the position of the 'data' pointer. This might be useful during the initial construction of the packet inside **sk_buff**.

Also, you may know function **eth_hdr()**
This function is just a simple wrapper around **skb_mac_header()** with typecasting.

CATEGORIES

Allsky camera (5)
Astro tools (9)
Automotive (1)
Electronics (30)
Firmware (3)
hardware (8)
Linux kernel (8)
Linux system development (11)
Networking (9)
Radio & antennas (14)
Radioastronomy (5)
Reverse engineering (5)
Software (17)
Uncategorized (1)

RECENT POSTS

Data logger for UNI-T UT800 multimeters
How to add Ethernet port to the Gen 2 Starlink router
EQMOD adapter for telescopes. Version 2
Initial analysis of the Starlink router gen2
Reverse engineering of the Starlink Ethernet adapter

RECENT COMMENTS

Andrii on Reverse engineering of the Starlink Ethernet adapter
Oleg Kutkov on C++ in Linux kernel
Johannes on C++ in Linux kernel
Oleg Kutkov on C++ in Linux kernel
Johannes on C++ in Linux kernel

OLEG KUTKOV PERSONAL BLOG

Home

```
static inline struct ethhdr *eth_hdr(const struct sk_buff *skb)
{
    return (struct ethhdr *)skb_mac_header(skb);
}
```

Now we can get a pointer to the whole packet, so it's time to print some data.

We can print an Ethernet header with source/destination addresses and protocol numbers if it's required.

```
struct ethhdr *ether = eth_hdr(skb);

printk("Source: %x:%x:%x:%x:%x:%x\n", ether->h_source[0], ether->h_source[1], ether->h_source[2]
printk("Destination: %x:%x:%x:%x:%x:%x\n", ether->h_dest[0], ether->h_dest[1], ether->h_dest[2],
printk("Protocol: %d\n", ether->h_proto);
```

Please note that the protocol number is in network byte order.

Typically network packets are printed as hex string by 16 bytes in one line and with line numbering.
Something like this:

```
000000 FF FF FF FF FF FF 20 CF 30 38 56 A1 08 06 00 01
000010 08 00 06 04 00 01 20 CF 30 38 56 A1 C0 A8 01 02
000020 00 00 00 00 00 00 C0 A8 01 05 00 00 00 00 00 00
000030 00 00 00 00 00 00 00 00 00 00 00 00
```

To get such output, we can write a simple function.

```
void pkt_hex_dump(struct sk_buff *skb)
{
    size_t len;
    int rowsize = 16;
    int i, l, linelen, remaining;
    int li = 0;
    uint8_t *data, ch;

    printk("Packet hex dump:\n");
    data = (uint8_t *) skb_mac_header(skb);

    if (skb_is_nonlinear(skb)) {
        len = skb->data_len;
    } else {
        len = skb->len;
    }

    remaining = len;
    for (i = 0; i < len; i += rowsize) {
        printk("%06d\t", li);

        linelen = min(remaining, rowsize);
        remaining -= rowsize;

        for (l = 0; l < linelen; l++) {
            ch = data[l];
            printk(KERN_CONT "%02X ", (uint32_t) ch);
        }

        data += linelen;
        li += 10;

        printk(KERN_CONT "\n");
    }
}
```

KERN_CONT in printk allows us to add data to the message buffer without flushing and without printing module name (and other information) at the beginning of every string. Except for time 🙂

After executing this function, you can find in dmesg something like this:

```
[ 1869.042384] 000000   FF FF FF FF FF FF 20 CF 30 38 56 A1 08 06 00 01
[ 1869.042424] 000010   08 00 06 04 00 01 20 CF 30 38 56 A1 C0 A8 01 02
[ 1869.042463] 000020   00 00 00 00 00 00 C0 A8 01 05 00 00 00 00 00 00
[ 1869.042502] 000030   00 00 00 00 00 00 00 00 00 00 00 00
```

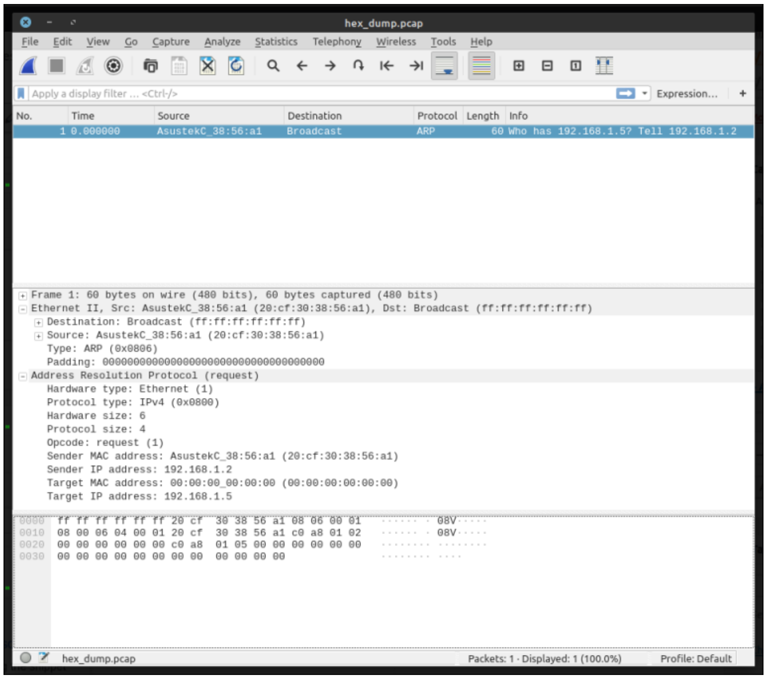
To analyze this data with Wireshark (which is very handy), we need to copy this text in some text files (for example, packet_dump.txt), remove timestamps, and convert this text into binary pcap format.

We need text2pcap utility, which can be found in most Linux distros.

Run the following command:

```
cat packet_dump.txt | awk '{$1=""; print $0}' | text2pcap - hex_dump.pcap
```

Now open the resulting **hex_dump.pcap** with Wireshark.



I can recommend using this method to print and analyze only small and medium-size packets. Large packets may hang or slow down your system during printing.

Thanks for reading!

Share this:

Related

Monitoring Linux networking state using netlink
February 14, 2018
In "Linux kernel"

Simple Linux character device driver
March 14, 2018
In "Linux system development"

Getting Linux routing table using netlink
March 24, 2019
In "Linux kernel"

Tagged kernel, linux, network, sk_buff

Related Posts

Data logger for UNI-T UT800 multimeters
July 18, 2022

How to add Ethernet port to the Gen 2 Starlink router
April 30, 2022

Initial analysis of the Starlink router gen2
April 10, 2022

About Oleg Kutkov

View all posts by Oleg Kutkov →

Modifying Linux network routes using netlink

RS-485 practice and theory

7 thoughts on “Printing sk_buff data”

anuj says:

June 15, 2020 at 2:01 pm

This was very helpful, thanks 😀

★ Loading...

Reply

gregoireg says:

August 12, 2020 at 7:12 pm

Very very useful page. Thank you for writing this.

★ Loading...

Reply

tomkcook says:

April 20, 2021 at 11:27 pm

Might I humbly suggest the following which is somewhat simpler and has much the same effect:

```
void dump_skb(struct sk_buff *skb) {
size_t len;
printk("Packet hex dump:\n");
uint8_t *data = (uint8_t *) skb_mac_header(skb);


if (skb_is_nonlinear(skb)) {
len = skb->data_len;
} else {
len = skb->len;
}


for (size_t ii = 0; ii < len; ++ii) {
printk("%06x\t%02x ", ii, data[ii++]);


for (; ii < len && (ii % 16 != 0); ++ii) {
printk(KERN_CONT "%02x ", (uint32_t)(data[ii]));
}
printk(KERN_CONT "\n");
}
}
```

In particular, using `%06d` as a format specifier and then adding 10 to `li` every time you print a row of 16 bytes is pretty horrible…

★ Loading...

Reply

1. **tomkcook** says:

   April 21, 2021 at 12:02 pm

   Though of course `printk("%06x\t%02x ", ii, data[ii++])` is undefined – you need to move the increment to a separate statement.

   ★ Loading...

   Reply

Joseph Cheng says:

April 21, 2021 at 3:29 am

Is it possible to print unfragmented ping message , " ping -l 8972 -f 192.168.0.108 –t" , sent from another computer ?

Thanks.

★ Loading...

Reply

Pingback: Netfilter Kernel module doesn't get the ftp packets' data – MVR

Ethelene Xiong says:

September 26, 2022 at 5:54 am

why the len is not the “len = skb->tail – skb->mac_header;”?

Loading...

Reply

Leave a Reply

Enter your comment here...

This site uses Akismet to reduce spam. Learn how your comment data is processed.

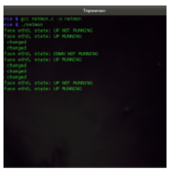
OLEG KUTKOV PERSONAL BLOG

Programming, electronics and diy projects

Linux Kernel, Linux System Development, Networking

Monitoring Linux networking state using netlink

Oleg Kutkov / February 14, 2018



Once in my work, I needed to monitor all changes in the Linux networking subsystem: adding or deleting IP addresses, routes, etc.
Maybe the best way to do this is to use socket-based Netlink technology. Using Netlink, we can "subscribe" to some network-related notifications from the kernel. It's also possible to send commands to the network stack and change the routing table, interface configurations, and packet filtering. For example, popular utilities like "iproute2" are also using Netlink to do their job.
The easiest way to access Netlink sockets from the userspace is to use a libnetlink library, which provides many macros, defines, and functions.
The worst part of this library and whole Netlink technology is a lack of good examples.
In this case, a good solution is using iproute2 source code to discover things you interesting in. This article is also may be used as a good startup point.

Introduction in Netlink

The Netlink is a socket-based Linux kernel interface used for inter-process communication (IPC) between both the kernel and userspace processes and between different userspace processes, in a way similar to the Unix domain sockets.

Like the Unix domain sockets, unlike INET sockets, Netlink communication cannot traverse host boundaries.
However, while the Unix domain sockets use the file system namespace, Netlink processes are addressed by process identifiers (PIDs).

Communication with Netlink is made using a separate socket's family – **AF_NETLINK**.
Every Netlink message contains a header, represented with **nlmsghdr** structure. After the header may be attached some payload: some special structure or RAW data.
Netlink can split big messages into multiple parts. In such a case, every "partial" package is marked with **NLM_F_MULTI** flag, and the last package is marked with **NLMSG_DONE** flag.

There are a lot of useful macros that can help us to parse Netlink messages.
Everything is defined in Netlink.h and rtnetlink.h header files.

Creating of Netlink socket is pretty standard.

```
socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE)
```

where:
AF_NETLINK — netlink domain
SOCK_RAW — raw socket
NETLINK_ROUTE — required protocol.

In particular, **NETLINK_ROUTE** is used for routing and link information.

All available protocols can be found in the documentation. Here is a list of the most interesting:

- **NETLINK_ROUTE —** routing and link information, monitoring and configuration routines
- **NETLINK_FIREWALL** — transfer packets to userspace from the firewall
- **NETLINK_INET_DIAG** — information about sockets of various protocol families
- **NETLINK_NFLOG** — Netfilter/iptables ULOG
- **NETLINK_SELINUX** — SELinux event notifications
- **NETLINK_NETFILTER** — communications with Netfilter subsystem
- **NETLINK_KOBJECT_UEVENT** — get kernel messages
- **NETLINK_USERSOCK** — reserved for user-defined protocols

Communication

All communications through the Netlink socket is made with two well-known structures: **msghdr** and **iovec**.

```
struct iovec
{
    void *iov_base; // data buff
    __kernel_size_t iov_len; // size of the data
};
```

CATEGORIES

Allsky camera (5)
Astro tools (9)
Automotive (1)
Electronics (30)
Firmware (3)
hardware (8)
Linux kernel (8)
Linux system development (11)
Networking (9)
Radio & antennas (14)
Radioastronomy (5)
Reverse engineering (5)
Software (17)
Uncategorized (1)

RECENT POSTS

Data logger for UNI-T UT800 multimeters
How to add Ethernet port to the Gen 2 Starlink router
EQMOD adapter for telescopes. Version 2
Initial analysis of the Starlink router gen2
Reverse engineering of the Starlink Ethernet adapter

RECENT COMMENTS

Andrii on Reverse engineering of the Starlink Ethernet adapter
Oleg Kutkov on C++ in Linux kernel
Johannes on C++ in Linux kernel
Oleg Kutkov on C++ in Linux kernel
Johannes on C++ in Linux kernel

OLEG KUTKOV PERSONAL BLOG

Home

This structure contains a link to the actual message buffer with some data and its size.

```
struct msghdr {
    void *msg_name; // client addr (socket name)
    int msg_namelen; // length of the client addr
    struct iovec *msg_iov; // pointer to the iovec structure with message data
    __kernel_size_t msg_iovlen; // count of the data blocks
    void *msg_control; // points to a buffer for other protocol control-related messages or misc
    __kernel_size_t msg_controllen; // length of the msg_control
    unsigned  msg_flags; // flags on received message
};
```

struct **msghdr** can be directly passed to socket's **recvmsg** and **sendmsg** and used to minimize the number of directly supplied arguments.

This structure is defined in *<sys/socket.h>*

See recvmsg and sendmsg for details.

A Netlink message stored in **iovec** typically contains a Netlink message header (**struct nlmsghdr**) and the payload attached. The payload can consist of arbitrary data but usually contains a fixed size protocol-specific header followed by a stream of attributes.

```
struct nlmsghdr
{
    __u32 nlmsg_len; // message size, include this header
    __u16 nlmsg_type; // message type (see below)
    __u16 nlmsg_flags; // message flags (see below)
    __u32 nlmsg_seq; // sequence number
    __u32 nlmsg_pid; // sender identifier (typically - process id)
};
```

The following standard message types are defined:

- **NLMSG_NOOP** – No operation, a message must be discarded
- **NLMSG_ERROR** – Error message or ACK, see Error Message respectively ACKs
- **NLMSG_DONE** – End of multipart sequence, see Multipart Messages
- **NLMSG_OVERRUN** – Overrun notification (Error)

> Every netlink protocol is free to define own message types. Note that message type values `< NLMSG_MIN_TYPE (0x10)` are reserved and may not be used.

The following standard flags are defined:

- **NLM_F_REQUEST** — Request message
- **NLM_F_MULTI** — Part of the multipart message
- **NLM_F_ACK** — Acknowledge requested
- **NLM_F_ECHO** — Request to echo this request; typical direction is from kernel to user
- **NLM_F_ROOT** — Return based on the root of the tree
- **NLM_F_MATCH** — Return all matching entries
- **NLM_F_ATOMIC** — Is obsolete now, used to request an atomic operation
- **NLM_F_DUMP** — Same as **NLM_F_ROOT|NLM_F_MATCH**

The client's identifications (user and kernel spaces) are made with structure **sockaddr_nl.**

```
struct sockaddr_nl
{
    sa_family_t nl_family; // always AF_NETLINK
    unsigned short nl_pad; // typically filled with zeros
    pid_t nl_pid; // client identifier (process id)
    __u32 nl_groups; // mask for senders/recivers group
};
```

nl_pid – unique socket identifier, for the kernel sockets, this value is always zero. On the userspace, typically used current process id. This may cause problems in multithreading applications if multiple threads are trying to create and use Netlink sockets.
To work around this, we can initialize every nl_pid with this construction:

```
pthread_self() << 16 | getpid()
```

nl_groups — is a special bitmask of Netlink groups. This value is used after calling bind() on the Netlink socket to "subscribe" to specified groups' events.
This is what we gonna use in our current task – network monitoring.

The definition of all groups can be found in the Netlink header file.
Here is some of them, which we can use in the current situation:

- **RTMGRP_LINK** — notifications about changes in network interface (up/down/added/removed)
- **RTMGRP_IPV4_IFADDR** — notifications about changes in IPv4 addresses (address was added or removed)
- **RTMGRP_IPV6_IFADDR** — same for IPv6
- **RTMGRP_IPV4_ROUTE** — notifications about changes in IPv4 routing table
- **RTMGRP_IPV6_ROUTE** — same for IPv6

Netlink message payload

As I already said – after the header, we can found some payload, which may be split into parts. Libnetlink contains several macros that are extremely helpful in accessing and checking message payload.

Some most useful:

- **NLMSG_DATA** — Get pointer to the message payload
- **NLMSG_PAYLOAD** — Get the actual size of the message payload
- **NLMSG_ALIGN** — Rounds the message size to the nearest aligned value
- **NLMSG_LENGTH** — Get the size of the payload and returns a correct aligned value
- **NLMSG_SPACE** — Get the actual size of the data in the Netlink packet
- **NLMSG_NEXT** — Get the next part of the multipart message. When using these macros, it's important to check for **NLMSG_DONE** message flag to avoid buffer overruns.
- **NLMSG_OK** — Returns true if the message is correct and was successfully parsed

Practical usage of Netlink

Okay, I think that it's enough of boring theory 🙂
Time to write some code and testing of the application.

Here is the full source code:

```
#include <errno.h>
#include <stdio.h>
#include <memory.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/rtnetlink.h>

// little helper to parsing message using netlink macroses
void parseRtattr(struct rtattr *tb[], int max, struct rtattr *rta, int len)
{
    memset(tb, 0, sizeof(struct rtattr *) * (max + 1));

    while (RTA_OK(rta, len)) {  // while not end of the message
        if (rta->rta_type <= max) {
            tb[rta->rta_type] = rta; // read attr
        }
        rta = RTA_NEXT(rta,len);    // get next attr
    }
}


int main()
{
    int fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);   // create netlink socket

    if (fd < 0) {
        printf("Failed to create netlink socket: %s\n", (char*)strerror(errno));
        return 1;
    }

    struct sockaddr_nl  local;  // local addr struct
    char buf[8192];             // message buffer
    struct iovec iov;           // message structure
    iov.iov_base = buf;         // set message buffer as io
    iov.iov_len = sizeof(buf);  // set size

    memset(&local, 0, sizeof(local));

    local.nl_family = AF_NETLINK;       // set protocol family
    local.nl_groups =   RTMGRP_LINK | RTMGRP_IPV4_IFADDR | RTMGRP_IPV4_ROUTE;   // set groups we
    local.nl_pid = getpid();    // set out id using current process id

    // initialize protocol message header
    struct msghdr msg;
    {
        msg.msg_name = &local;                  // local address
        msg.msg_namelen = sizeof(local);        // address size
        msg.msg_iov = &iov;                     // io vector
        msg.msg_iovlen = 1;                     // io size
    }

    if (bind(fd, (struct sockaddr*)&local, sizeof(local)) < 0) {     // bind socket
        printf("Failed to bind netlink socket: %s\n", (char*)strerror(errno));
        close(fd);
        return 1;
    }

    // read and parse all messages from the
    while (1) {
        ssize_t status = recvmsg(fd, &msg, MSG_DONTWAIT);
```

```
        //  check status
        if (status < 0) {
            if (errno == EINTR || errno == EAGAIN)
            {
                usleep(250000);
                continue;
            }

            printf("Failed to read netlink: %s", (char*)strerror(errno));
            continue;
        }

        if (msg.msg_namelen != sizeof(local)) { // check message length, just in case
            printf("Invalid length of the sender address struct\n");
            continue;
        }

        // message parser
        struct nlmsghdr *h;

        for (h = (struct nlmsghdr*)buf; status >= (ssize_t)sizeof(*h); ) {   // read all message
            int len = h->nlmsg_len;
            int l = len - sizeof(*h);
            char *ifName;

            if ((l < 0) || (len > status)) {
                printf("Invalid message length: %i\n", len);
                continue;
            }

            // now we can check message type
            if ((h->nlmsg_type == RTM_NEWROUTE) || (h->nlmsg_type == RTM_DELROUTE)) { // some ch
                printf("Routing table was changed\n");
            } else {    // in other case we need to go deeper
                char *ifUpp;
                char *ifRunn;
                struct ifinfomsg *ifi;  // structure for network interface info
                struct rtattr *tb[IFLA_MAX + 1];

                ifi = (struct ifinfomsg*) NLMSG_DATA(h);    // get information about changed net

                parseRtattr(tb, IFLA_MAX, IFLA_RTA(ifi), h->nlmsg_len);  // get attributes

                if (tb[IFLA_IFNAME]) {  // validation
                    ifName = (char*)RTA_DATA(tb[IFLA_IFNAME]); // get network interface name
                }

                if (ifi->ifi_flags & IFF_UP) { // get UP flag of the network interface
                    ifUpp = (char*)"UP";
                } else {
                    ifUpp = (char*)"DOWN";
                }

                if (ifi->ifi_flags & IFF_RUNNING) { // get RUNNING flag of the network interface
                    ifRunn = (char*)"RUNNING";
                } else {
                    ifRunn = (char*)"NOT RUNNING";
                }

                char ifAddress[256];    // network addr
                struct ifaddrmsg *ifa; // structure for network interface data
                struct rtattr *tba[IFA_MAX+1];

                ifa = (struct ifaddrmsg*)NLMSG_DATA(h); // get data from the network interface

                parseRtattr(tba, IFA_MAX, IFA_RTA(ifa), h->nlmsg_len);

                if (tba[IFA_LOCAL]) {
                    inet_ntop(AF_INET, RTA_DATA(tba[IFA_LOCAL]), ifAddress, sizeof(ifAddress));
                }

                switch (h->nlmsg_type) { // what is actually happenned?
                    case RTM_DELADDR:
                        printf("Interface %s: address was removed\n", ifName);
                        break;

                    case RTM_DELLINK:
                        printf("Network interface %s was removed\n", ifName);
                        break;
```

```
                    case RTM_NEWLINK:
                        printf("New network interface %s, state: %s %s\n", ifName, ifUpp, ifRunn
                        break;

                    case RTM_NEWADDR:
                        printf("Interface %s: new address was assigned: %s\n", ifName, ifAddress
                        break;
                }
            }

            status -= NLMSG_ALIGN(len); // align offsets by the message length, this is importan

            h = (struct nlmsghdr*)((char*)h + NLMSG_ALIGN(len));    // get next message
        }

        usleep(250000); // sleep for a while
    }

    close(fd);  // close socket

    return 0;
}
```

The compilation is straightforward, nothing additional:

```
gcc netmon.c -o netmon
```

And run:

```
./netmon
```

Now you can try to play with your network interfaces – unplug and plug back of the Ethernet cable, reconnect WiFi, and so on.

You will get something like this:

It's alive! 🙂

Data processing

In this example, you can find some new structures:

```
struct ifinfomsg
{
    unsigned char  ifi_family;  // interface family
    unsigned short ifi_type;    // device type
    int            ifi_index;   // interface index
    unsigned int   ifi_flags;   // device flags
    unsigned int   ifi_change;  // reserved, currently always 0xFFFFFFFF
};
```

struct ifinfomsg represents a network device and contains some useful fields, like device flags and index.

```
struct ifaddrmsg
{
    unsigned char  ifa_family;    // Adress type (AF_INET or AF_INET6)
    unsigned char  ifa_prefixlen; // Length of the network mask
    unsigned char  ifa_flags;     // Address flags
    unsigned char  ifa_scope;     // Address scope
    int            ifa_index;     // Interface index, same as in struct ifinfomsg
};
```

struct ifaddrmsg represents the network address assigned to the device

```
struct rtattr
{
    unsigned short rta_len; // Length of the option
    unsigned short rta_type; // Type of the option
    /* data */
}
```

struct rtattr is a helper structure used to store some parameters of the address or network link

After the successful creation of the Netlink socket, we initializing **sockaddr_nl** structure by setting a mask of the groups which messages we want to receive:
RTMGRP_LINK, RTMGRP_IPV4_IFADDR and **RTMGRP_IPV4_ROUTE**.
Also, at this point, we are allocating message structure and data buffer with a length of 8192 bytes.
After all of this, we can call **bind()** on a socket, subscribing to group events.
We get new messages from the socket in the infinity cycle and then parsing this message using Netlink macro.
Checking **nlmsg_type** field, we can detect the type of the received message. In the case of some interface/address event, we are digging deeper and getting all the interesting data.
All information is stored as an array of attributes with **struct rtattr**.
Using the little helper function parseRattr we can parse all attributes and extract readable information from this array.

```
struct ifinfomsg *ifi = (struct ifinfomsg*) NLMSG_DATA(h); // where h is netlink message header
parseRtattr(tb, IFLA_MAX, IFLA_RTA(ifi), h->nlmsg_len);
char* ifName = (char*)RTA_DATA(tb[IFLA_IFNAME]); // readable interface name, eth0 for example
```

You can check rtnetlink manual page to get more information about **rtattr** arrays and possible attributes indexes.

I believe that all other code in this example is pretty obvious and didn't require detailed explanations.
But if you have some questions – please ask in the comments.

I hope this article will be helpful.

Additional materials:

1. tools.ietf.org/html/rfc3549
2. http://man7.org/linux/man-pages/man7/netlink.7.html
3. http://man7.org/linux/man-pages/man7/rtnetlink.7.html
4. http://linuxjournal.com/article/7356

Related

Modifying Linux network routes using netlink
August 29, 2019
In "Linux kernel"

Getting Linux routing table using netlink
March 24, 2019
In "Linux kernel"

Printing sk_buff data
October 17, 2019
In "Linux kernel"

Tagged linux, netlink, network, socket

Related Posts

Data logger for UNI-T UT800 multimeters
July 18, 2022

How to add Ethernet port to the Gen 2 Starlink router
April 30, 2022

Initial analysis of the Starlink router gen2
April 10, 2022

About Oleg Kutkov

View all posts by Oleg Kutkov →

Listening to aircrafts and receiving images from the satellites

Isolated eqmod adapter for the telescope control

10 thoughts on "Monitoring Linux networking state using netlink"

io says:

March 3, 2019 at 8:53 pm

Thanks for the article!
There are a few weird not documented macros directly from linux kernel – how did you figure out what they are supposed to do? Seems like the only way to work with netlink without libs like libnl is to debug and pull things from iproute2…

Loading...

Reply

Oleg Kutkov says:

March 3, 2019 at 9:48 pm

Hello! Yep, I spent a lot of the time trying to figure out how it supposed to work.
I digged into the kernel and iproute2 sources, debugging and experimenting.

Loading...

Reply

Pingback: Getting Linux routing table using netlink. - Oleg Kutkov personal blog

dmytro says:

April 10, 2019 at 4:53 pm

Hello Oleg,
I would be grateful if you checked this SO question
https://stackoverflow.com/questions/55614270/how-to-asynchronously-check-if-an-ipv6-netwrok-interface-changes-state-from-tent

and if possible provided answers/ideas, etc.
Thanks in advance!!!

Loading...

Reply

Oleg Kutkov says:

April 11, 2019 at 1:00 am

Hello.
Please check out my answer on the Stack Overflow.

Loading...

Reply

Puneet Sharma says:

July 16, 2020 at 4:53 pm

Nice article.
Very useful to jumpstart understanding netlink sockets.
Thank you.

Loading...

Reply

shahriar says:

November 5, 2020 at 12:06 pm

NLMSG_DATA(h) is first casted to ifinfomsg and then ifaddrmsg in your code. Can you explain how it works? I thought we have ifinfomsg in case of NEW_LINK,DEL_LINK and ifaddrmsg in case of NEW_ADDR and DEL_ADDR

Loading...

Reply

Shahriar Basiri says:

November 7, 2020 at 9:30 am

Hello Oleg,
Another question dear Oleg:). In function parseRtattr(), “h->nlmsg_len” is passed as “len” where it is the size of whole netlink message (nlmsghdr+ifinfomsg/ifaddrmsg+rtattrs). Then, this len is checked in RTA_OK and updated in RTA_NEXT macros. I think this size should be just size of rtattrs so that RTA_OK be valid.

★ Loading...

Reply

Venkateswaran says:

May 7, 2021 at 12:35 am

RTM_DELLINK event is not triggered in any case. Do you know why ? I thought it ll be triggered when I remove my cable but not

★ Loading...

Reply

Pragya Nand says:

September 7, 2022 at 4:31 pm

Hi Oleg,
Thanks for such an informative blogpost on netlink.
Is there any specific post related on how to access nested attribute such as IFLA_LINKINFO.

★ Loading...

Reply

Leave a Reply

Enter your comment here...

This site uses Akismet to reduce spam. Learn how your comment data is processed.