NAME

netlink − communication between kernel and user space (AF_NETLINK)

SYNOPSIS

#include <asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

netlink_socket = socket(AF_NETLINK, *socket_type***,** *netlink_family***);**

DESCRIPTION

Netlink is used to transfer information between the kernel and user-space processes. It consists of a standard sockets-based interface for user space processes and an internal kernel API for kernel modules. The internal kernel interface is not documented in this manual page. There is also an obsolete netlink interface via netlink character devices; this interface is not documented here and is provided only for backward compatibility.

Netlink is a datagram-oriented service. Both **SOCK_RAW** and **SOCK_DGRAM** are valid values for *socket_type*. However, the netlink protocol does not distinguish between datagram and raw sockets.

netlink_family selects the kernel module or netlink group to communicate with. The currently assigned netlink families are:

NETLINK_ROUTE
: Receives routing and link updates and may be used to modify the routing tables (both IPv4 and IPv6), IP addresses, link parameters, neighbor setups, queueing disciplines, traffic classes, and packet classifiers (see **rtnetlink**(7)).

NETLINK_W1 (Linux 2.6.13 to Linux 2.16.17)
: Messages from 1-wire subsystem.

NETLINK_USERSOCK
: Reserved for user-mode socket protocols.

NETLINK_FIREWALL (up to and including Linux 3.4)
: Transport IPv4 packets from netfilter to user space. Used by *ip_queue* kernel module. After a long period of being declared obsolete (in favor of the more advanced *nfnetlink_queue* feature), **NETLINK_FIREWALL** was removed in Linux 3.5.

NETLINK_SOCK_DIAG (since Linux 3.3)
: Query information about sockets of various protocol families from the kernel (see **sock_diag**(7)).

NETLINK_INET_DIAG (since Linux 2.6.14)
: An obsolete synonym for **NETLINK_SOCK_DIAG**.

NETLINK_NFLOG (up to and including Linux 3.16)
: Netfilter/iptables ULOG.

NETLINK_XFRM
: IPsec.

NETLINK_SELINUX (since Linux 2.6.4)
: SELinux event notifications.

NETLINK_ISCSI (since Linux 2.6.15)
: Open-iSCSI.

NETLINK_AUDIT (since Linux 2.6.6)
: Auditing.

NETLINK_FIB_LOOKUP (since Linux 2.6.13)
: Access to FIB lookup from user space.

NETLINK_CONNECTOR (since Linux 2.6.14)
Kernel connector. See*Documentation/driver−api/connector .rst* (or */Documentation/connector/connector.** in Linux 5.2 and earlier) in the Linux kernel source tree for further information.

NETLINK_NETFILTER (since Linux 2.6.14)
Netfilter subsystem.

NETLINK_SCSITRANSPORT (since Linux 2.6.19)
SCSI Transports.

NETLINK_RDMA (since Linux 3.0)
Infiniband RDMA.

NETLINK_IP6_FW (up to and including Linux 3.4)
Transport IPv6 packets from netfilter to user space. Used by *ip6_queue* kernel module.

NETLINK_DNRTMSG
DECnet routing messages.

NETLINK_KOBJECT_UEVENT (since Linux 2.6.10)
Kernel messages to user space.

NETLINK_GENERIC (since Linux 2.6.15)
Generic netlink family for simplified netlink usage.

NETLINK_CRYPTO (since Linux 3.2)
Netlink interface to request information about ciphers registered with the kernel crypto API as well as allow configuration of the kernel crypto API.

Netlink messages consist of a byte stream with one or multiple *nlmsghdr* headers and associated payload. The byte stream should be accessed only with the standard **NLMSG_*** macros. See **netlink**(3) for further information.

In multipart messages (multiple *nlmsghdr* headers with associated payload in one byte stream) the first and all following headers have the **NLM_F_MULTI** flag set, except for the last header which has the type **NLMSG_DONE**.

After each *nlmsghdr* the payload follows.

```
struct nlmsghdr {
    __u32 nlmsg_len;    /* Length of message including header */
    __u16 nlmsg_type;   /* Type of message content */
    __u16 nlmsg_flags;  /* Additional flags */
    __u32 nlmsg_seq;    /* Sequence number */
    __u32 nlmsg_pid;    /* Sender port ID */
};
```

nlmsg_type can be one of the standard message types: **NLMSG_NOOP** message is to be ignored, **NLMSG_ERROR** message signals an error and the payload contains an *nlmsgerr* structure, **NLMSG_DONE** message terminates a multipart message. Error messages get the original request appended, unless the user requests to cap the error message, and get extra error data if requested.

```
struct nlmsgerr {
    int error;        /* Negative errno or 0 for acknowledgements */
    struct nlmsghdr msg;  /* Message header that caused the error */
    /*
     * followed by the message contents
     * unless NETLINK_CAP_ACK was set
     * or the ACK indicates success (error == 0).
     * For example Generic Netlink message with attributes.
     * message length is aligned with NLMSG_ALIGN()
     */
    /*
```

```
          * followed by TLVs defined in enum nlmsgerr_attrs
          * if NETLINK_EXT_ACK was set
          */
     };
```

A netlink family usually specifies more message types, see the appropriate manual pages for that, for example, **rtnetlink**(7) for **NETLINK_ROUTE**.

Standard flag bits in *nlmsg_flags*

NLM_F_REQUEST	Must be set on all request messages.
NLM_F_MULTI	The message is part of a multipart message terminated by **NLMSG_DONE**.
NLM_F_ACK	Request for an acknowledgement on success.
NLM_F_ECHO	Echo this request.

Additional flag bits for GET requests

NLM_F_ROOT	Return the complete table instead of a single entry.
NLM_F_MATCH	Return all entries matching criteria passed in message content. Not implemented yet.
NLM_F_ATOMIC	Return an atomic snapshot of the table.
NLM_F_DUMP	Convenience macro; equivalent to (NLM_F_ROOT|NLM_F_MATCH).

Note that **NLM_F_ATOMIC** requires the **CAP_NET_ADMIN** capability or an effective UID of 0.

Additional flag bits for NEW requests

NLM_F_REPLACE	Replace existing matching object.
NLM_F_EXCL	Don't replace if the object already exists.
NLM_F_CREATE	Create object if it doesn't already exist.
NLM_F_APPEND	Add to the end of the object list.

nlmsg_seq and *nlmsg_pid* are used to track messages. *nlmsg_pid* shows the origin of the message. Note that there isn't a 1:1 relationship between *nlmsg_pid* and the PID of the process if the message originated from a netlink socket. See the**ADDRESS FORMA TS** section for further information.

Both *nlmsg_seq* and *nlmsg_pid* are opaque to netlink core.

Netlink is not a reliable protocol. It tries its best to deliver a message to its destination(s), but may drop messages when an out-of-memory condition or other error occurs. For reliable transfer the sender can request an acknowledgement from the receiver by setting the **NLM_F_ACK** flag. An acknowledgement is an **NLMSG_ERROR** packet with the error field set to 0. The application must generate acknowledgements for received messages itself. The kernel tries to send an **NLMSG_ERROR** message for every failed packet. A user process should follow this convention too.

However, reliable transmissions from kernel to user are impossible in any case. The kernel can't send a netlink message if the socket buffer is full: the message will be dropped and the kernel and the user-space process will no longer have the same view of kernel state. It is up to the application to detect when this happens (via the **ENOBUFS** error returned by **recvmsg**(2)) and resynchronize.

Address formats

The *sockaddr_nl* structure describes a netlink client in user space or in the kernel. A*soc kaddr_nl* can be either unicast (only sent to one peer) or sent to netlink multicast groups (*nl_groups* not equal 0).

```
struct sockaddr_nl {
    sa_family_t     nl_family;  /* AF_NETLINK */
    unsigned short  nl_pad;     /* Zero */
    pid_t           nl_pid;     /* Port ID */
    __u32           nl_groups;  /* Multicast groups mask */
};
```

nl_pid is the unicast address of netlink socket. It's always 0 if the destination is in the kernel. For a user-space process, *nl_pid* is usually the PID of the process owning the destination socket. However, *nl_pid* identifies a netlink socket, not a process. If a process owns several netlink sockets, then *nl_pid* can be equal to the process ID only for at most one socket. There are two ways to assign *nl_pid* to a netlink socket. If the application sets*nl_pid* before calling **bind**(2), then it is up to the application to mak e sure that *nl_pid* is unique. If the application sets it to 0, the kernel takes care of assigning it. The kernel assigns the process ID to the first netlink socket the process opens and assigns a unique *nl_pid* to every netlink socket that the process subsequently creates.

nl_groups is a bit mask with every bit representing a netlink group number. Each netlink family has a set of 32 multicast groups. When **bind**(2) is called on the socket, the *nl_groups* field in the *sockaddr_nl* should be set to a bit mask of the groups which it wishes to listen to. The default value for this field is zero which means that no multicasts will be received. A socket may multicast messages to any of the multicast groups by setting *nl_groups* to a bit mask of the groups it wishes to send to when it calls **sendmsg**(2) or does a **connect**(2). Only processes with an effective UID of 0 or the **CAP_NET_ADMIN** capability may send or listen to a netlink multicast group. Since Linux 2.6.13, messages can't be broadcast to multiple groups. Any replies to a message received for a multicast group should be sent back to the sending PID and the multicast group. Some Linux kernel subsystems may additionally allow other users to send and/or receive messages. As at Linux 3.0, the **NETLINK_KOBJECT_UEVENT**, **NETLINK_GENERIC**, **NETLINK_ROUTE**, and **NETLINK_SELINUX** groups allow other users to receive messages. No groups allow other users to send messages.

Socket options

To set or get a netlink socket option, call **getsockopt**(2) to read or **setsockopt**(2) to write the option with the option level argument set to **SOL_NETLINK**. Unless otherwise noted,*optval* is a pointer to an *int*.

NETLINK_PKTINFO (since Linux 2.6.14)
: Enable **nl_pktinfo** control messages for received packets to get the extended destination group number.

NETLINK_ADD_MEMBERSHIP, **NETLINK_DROP_MEMBERSHIP** (since Linux 2.6.14)
: Join/leave a group specified by *optval*.

NETLINK_LIST_MEMBERSHIPS (since Linux 4.2)
: Retrieve all groups a socket is a member of. *optval* is a pointer to **__u32** and *optlen* is the size of the array. The array is filled with the full membership set of the socket, and the required array size is returned in *optlen*.

NETLINK_BROADCAST_ERROR (since Linux 2.6.30)
: When not set, **netlink_broadcast()** only reports **ESRCH** errors and silently ignore **ENOBUFS** errors.

NETLINK_NO_ENOBUFS (since Linux 2.6.30)
: This flag can be used by unicast and broadcast listeners to avoid receiving **ENOBUFS** errors.

NETLINK_LISTEN_ALL_NSID (since Linux 4.2)
: When set, this socket will receive netlink notifications from all network namespaces that have an *nsid* assigned into the network namespace where the socket has been opened. The *nsid* is sent to user space via an ancillary data.

NETLINK_CAP_ACK (since Linux 4.3)
: The kernel may fail to allocate the necessary room for the acknowledgement message back to user space. This option trims off the payload of the original netlink message. The netlink message header is still included, so the user can guess from the sequence number which message triggered the acknowledgement.

VERSIONS

The socket interface to netlink first appeared Linux 2.2.

Linux 2.0 supported a more primitive device-based netlink interface (which is still available as a compatibility option). This obsolete interface is not described here.

NOTES

It is often better to use netlink via *libnetlink* or *libnl* than via the low-level kernel interface.

BUGS

This manual page is not complete.

EXAMPLES

The following example creates a **NETLINK_ROUTE** netlink socket which will listen to the **RTMGRP_LINK** (network interface create/delete/up/down events) and **RTMGRP_IPV4_IFADDR** (IPv4 addresses add/delete events) multicast groups.

```
struct sockaddr_nl sa;

memset(&sa, 0, sizeof(sa));
sa.nl_family = AF_NETLINK;
sa.nl_groups = RTMGRP_LINK | RTMGRP_IPV4_IFADDR;

fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
bind(fd, (struct sockaddr *) &sa, sizeof(sa));
```

The next example demonstrates how to send a netlink message to the kernel (pid 0). Note that the application must take care of message sequence numbers in order to reliably track acknowledgements.

```
struct nlmsghdr *nh;    /* The nlmsghdr with payload to send */
struct sockaddr_nl sa;
struct iovec iov = { nh, nh−>nlmsg_len };
struct msghdr msg;

msg = { &sa, sizeof(sa), &iov, 1, NULL, 0, 0 };
memset(&sa, 0, sizeof(sa));
sa.nl_family = AF_NETLINK;
nh−>nlmsg_pid = 0;
nh−>nlmsg_seq = ++sequence_number;
/* Request an ack from kernel by setting NLM_F_ACK */
nh−>nlmsg_flags |= NLM_F_ACK;

sendmsg(fd, &msg, 0);
```

And the last example is about reading netlink message.

```
int len;
/* 8192 to avoid message truncation on platforms with
   page size > 4096 */
struct nlmsghdr buf[8192/sizeof(struct nlmsghdr)];
struct iovec iov = { buf, sizeof(buf) };
struct sockaddr_nl sa;
struct msghdr msg;
struct nlmsghdr *nh;

msg = { &sa, sizeof(sa), &iov, 1, NULL, 0, 0 };
len = recvmsg(fd, &msg, 0);

for (nh = (struct nlmsghdr *) buf; NLMSG_OK (nh, len);
     nh = NLMSG_NEXT (nh, len)) {
    /* The end of multipart message */
    if (nh−>nlmsg_type == NLMSG_DONE)
        return;

    if (nh−>nlmsg_type == NLMSG_ERROR)
        /* Do some error handling */
    ...
```

```
            /* Continue with parsing payload */
            ...
        }
```

SEE ALSO

cmsg(3), **netlink**(3), **capabilities**(7), **rtnetlink**(7), **sock_diag**(7)

information about libnetlink ⟨ftp://ftp.inr.ac.ru/ip−routing/iproute2*⟩

information about libnl ⟨http://www.infradead.org/~tgr/libnl/⟩

RFC 3549 "Linux Netlink as an IP Services Protocol"