NAME

packet − packet interface on device level

SYNOPSIS

#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */

packet_socket = socket(AF_PACKET, int *socket_type***, int** *protocol***);**

DESCRIPTION

Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.

The *socket_type* is either **SOCK_RAW** for raw packets including the link-level header or **SOCK_DGRAM** for cooked packets with the link-level header removed. The link-level header information is available in a common format in a *sockaddr_ll* structure. *protocol* is the IEEE 802.3 protocol number in network byte order. See the<*linux/if_ether*.*h*> include file for a list of allowed protocols. When protocol is set to **htons(ETH_P_ALL)**, then all protocols are received. All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols implemented in the kernel. If *protocol* is set to zero, no packets are received. **bind**(2) can optionally be called with a nonzero *sll_protocol* to start receiving packets for the protocols specified.

In order to create a packet socket, a process must have the **CAP_NET_RAW** capability in the user namespace that governs its network namespace.

SOCK_RAW packets are passed to and from the device driver without any changes in the packet data. When receiving a packet, the address is still parsed and passed in a standard *sockaddr_ll* address structure. When transmitting a packet, the user-supplied buffer should contain the physical-layer header. That packet is then queued unmodified to the network driver of the interface defined by the destination address. Some device drivers always add other headers. **SOCK_RA W** is similar to but not compatible with the obsolete **AF_INET/SOCK_PACKET** of Linux 2.0.

SOCK_DGRAM operates on a slightly higher level. The physical header is removed before the packet is passed to the user. Packets sent through a **SOCK_DGRAM** packet socket get a suitable physical-layer header based on the information in the *sockaddr_ll* destination address before they are queued.

By default, all packets of the specified protocol type are passed to a packet socket. To get packets only from a specific interface use **bind**(2) specifying an address in a *struct sockaddr_ll* to bind the packet socket to an interface. Fields used for binding are*sll_family* (should be **AF_P ACKET**), *sll_protocol*, and *sll_ifindex*.

The **connect**(2) operation is not supported on packet sockets.

When the **MSG_TRUNC** flag is passed to **recvmsg**(2), **recv**(2), or **recvfrom**(2), the real length of the packet on the wire is always returned, even when it is longer than the buffer.

Address types

The *sockaddr_ll* structure is a device-independent physical-layer address.

```
struct sockaddr_ll {
    unsigned short sll_family;   /* Always AF_PACKET */
    unsigned short sll_protocol; /* Physical-layer protocol */
    int            sll_ifindex;  /* Interface number */
    unsigned short sll_hatype;   /* ARP hardware type */
    unsigned char  sll_pkttype;  /* Packet type */
    unsigned char  sll_halen;    /* Length of address */
    unsigned char  sll_addr[8];  /* Physical-layer address */
};
```

The fields of this structure are as follows:

sll_protocol
is the standard ethernet protocol type in network byte order as defined in the<*linux/if_ether.h*> include file. It defaults to the socket's protocol.

sll_ifindex
is the interface index of the interface (see **netdevice**(7)); 0 matches any interface (only permitted for binding). *sll_hatype* is an ARP type as defined in the <*linux/if_arp.h*> include file.

sll_pkttype
contains the packet type. Valid types are **PACKET_HOST** for a packet addressed to the local host, **PACKET_BROADCAST** for a physical-layer broadcast packet, **PACKET_MULTICAST** for a packet sent to a physical-layer multicast address, **PACKET_OTHERHOST** for a packet to some other host that has been caught by a device driver in promiscuous mode, and **PACKET_OUTGOING** for a packet originating from the local host that is looped back to a packet socket. These types make sense only for receiving.

sll_addr
sll_halen
contain the physical-layer (e.g., IEEE 802.3) address and its length. The exact interpretation depends on the device.

When you send packets, it is enough to specify *sll_family*, *sll_addr*, *sll_halen*, *sll_ifindex*, and *sll_protocol*. The other fields should be 0. *sll_hatype* and *sll_pkttype* are set on received packets for your information.

Socket options
Packet socket options are configured by calling **setsockopt**(2) with level **SOL_PACKET**.

PACKET_ADD_MEMBERSHIP
PACKET_DROP_MEMBERSHIP
Packet sockets can be used to configure physical-layer multicasting and promiscuous mode. **PACKET_ADD_MEMBERSHIP** adds a binding and **PACKET_DROP_MEMBERSHIP** drops it. They both expect a *packet_mreq* structure as argument:

```
struct packet_mreq {
    int            mr_ifindex;    /* interface index */
    unsigned short mr_type;       /* action */
    unsigned short mr_alen;       /* address length */
    unsigned char  mr_address[8]; /* physical-layer address */
};
```

mr_ifindex contains the interface index for the interface whose status should be changed. The *mr_type* field specifies which action to perform. **PACKET_MR_PROMISC** enables receiving all packets on a shared medium (often known as "promiscuous mode"), **PACKET_MR_MULTICAST** binds the socket to the physical-layer multicast group specified in *mr_address* and *mr_alen*, and **PACKET_MR_ALLMULTI** sets the socket up to receive all multicast packets arriving at the interface.

In addition, the traditional ioctls **SIOCSIFFLAGS**, **SIOCADDMULTI**, **SIOCDELMULTI** can be used for the same purpose.

PACKET_AUXDATA (since Linux 2.6.21)
If this binary option is enabled, the packet socket passes a metadata structure along with each packet in the **recvmsg**(2) control field. The structure can be read with **cmsg**(3). It is defined as

```
struct tpacket_auxdata {
    __u32 tp_status;
    __u32 tp_len;      /* packet length */
    __u32 tp_snaplen;  /* captured length */
    __u16 tp_mac;
    __u16 tp_net;
```

```
        __u16 tp_vlan_tci;
        __u16 tp_vlan_tpid; /* Since Linux 3.14; earlier, these
                               were unused padding bytes */
    };
```

PACKET_FANOUT (since Linux 3.1)
To scale processing across threads, packet sockets can form a fanout group. In this mode, each matching packet is enqueued onto only one socket in the group. A socket joins a fanout group by calling **setsockopt**(2) with level **SOL_PACKET** and option **PACKET_FANOUT**. Each network namespace can have up to 65536 independent groups. A socket selects a group by encoding the ID in the first 16 bits of the integer option value. The first packet socket to join a group implicitly creates it. To successfully join an existing group, subsequent packet sockets must have the same protocol, device settings, fanout mode, and flags (see below). Packet sockets can leave a fanout group only by closing the socket. The group is deleted when the last socket is closed.

Fanout supports multiple algorithms to spread traffic between sockets, as follows:

- The default mode, **PACKET_FANOUT_HASH**, sends packets from the same flow to the same socket to maintain per-flow ordering. For each packet, it chooses a socket by taking the packet flow hash modulo the number of sockets in the group, where a flow hash is a hash over network-layer address and optional transport-layer port fields.
- The load-balance mode **PACKET_FANOUT_LB** implements a round-robin algorithm.
- **PACKET_FANOUT_CPU** selects the socket based on the CPU that the packet arrived on.
- **PACKET_FANOUT_ROLLOVER** processes all data on a single socket, moving to the next when one becomes backlogged.
- **PACKET_FANOUT_RND** selects the socket using a pseudo-random number generator.
- **PACKET_FANOUT_QM** (available since Linux 3.14) selects the socket using the recorded queue_mapping of the received skb.

Fanout modes can take additional options. IP fragmentation causes packets from the same flow to have different flow hashes. The flag **PACKET_FANOUT_FLAG_DEFRAG**, if set, causes packets to be defragmented before fanout is applied, to preserve order even in this case. Fanout mode and options are communicated in the second 16 bits of the integer option value. The flag **PACKET_FANOUT_FLAG_ROLLOVER** enables the roll over mechanism as a backup strategy: if the original fanout algorithm selects a backlogged socket, the packet rolls over to the next available one.

PACKET_LOSS (with **PACKET_TX_RING**)
When a malformed packet is encountered on a transmit ring, the default is to reset its *tp_status* to **TP_STATUS_WRONG_FORMAT** and abort the transmission immediately. The malformed packet blocks itself and subsequently enqueued packets from being sent. The format error must be fixed, the associated *tp_status* reset to **TP_STATUS_SEND_REQUEST**, and the transmission process restarted via **send**(2). However, if **PACKET_LOSS** is set, any malformed packet will be skipped, its *tp_status* reset to **TP_STATUS_AVAILABLE**, and the transmission process continued.

PACKET_RESERVE (with **PACKET_RX_RING**)
By default, a packet receive ring writes packets immediately following the metadata structure and alignment padding. This integer option reserves additional headroom.

PACKET_RX_RING
Create a memory-mapped ring buffer for asynchronous packet reception. The packet socket reserves a contiguous region of application address space, lays it out into an array of packet slots and copies packets (up to *tp_snaplen*) into subsequent slots. Each packet is preceded by a metadata structure similar to *tpacket_auxdata*. The protocol fields encode the offset to the data from the start of the metadata header. *tp_net* stores the offset to the network layer. If the packet socket is of type **SOCK_DGRAM**, then *tp_mac* is the same. If it is of type **SOCK_RAW**, then that field

stores the offset to the link-layer frame. Packet socket and application communicate the head and tail of the ring through the *tp_status* field. The packet socket owns all slots with *tp_status* equal to **TP_STATUS_KERNEL**. After filling a slot, it changes the status of the slot to transfer ownership to the application. During normal operation, the new *tp_status* value has at least the **TP_STATUS_USER** bit set to signal that a received packet has been stored. When the application has finished processing a packet, it transfers ownership of the slot back to the socket by setting *tp_status* equal to **TP_STATUS_KERNEL**.

Packet sockets implement multiple variants of the packet ring. The implementation details are described in *Documentation/networking/packet_mmap.rst* in the Linux kernel source tree.

PACKET_STATISTICS
Retrieve packet socket statistics in the form of a structure

```
struct tpacket_stats {
    unsigned int tp_packets;  /* Total packet count */
    unsigned int tp_drops;    /* Dropped packet count */
};
```

Receiving statistics resets the internal counters. The statistics structure differs when using a ring of variant **TPACKET_V3**.

PACKET_TIMESTAMP (with **PACKET_RX_RING**; since Linux 2.6.36)
The packet receive ring always stores a timestamp in the metadata header. By default, this is a software generated timestamp generated when the packet is copied into the ring. This integer option selects the type of timestamp. Besides the default, it support the two hardware formats described in *Documentation/networking/timestamping.rst* in the Linux kernel source tree.

PACKET_TX_RING (since Linux 2.6.31)
Create a memory-mapped ring buffer for packet transmission. This option is similar to **PACKET_RX_RING** and takes the same arguments. The application writes packets into slots with *tp_status* equal to **TP_STATUS_AVAILABLE** and schedules them for transmission by changing *tp_status* to **TP_STATUS_SEND_REQUEST**. When packets are ready to be transmitted, the application calls **send**(2) or a variant thereof. The *buf* and *len* fields of this call are ignored. If an address is passed using**sendto**(2) or **sendmsg**(2), then that o verrides the socket default. On successful transmission, the socket resets *tp_status* to **TP_STATUS_AVAILABLE**. It immediately aborts the transmission on error unless **PACKET_LOSS** is set.

PACKET_VERSION (with **PACKET_RX_RING**; since Linux 2.6.27)
By default, **PACKET_RX_RING** creates a packet receive ring of variant **TPACKET_V1**. To create another variant, configure the desired variant by setting this integer option before creating the ring.

PACKET_QDISC_BYPASS (since Linux 3.14)
By default, packets sent through packet sockets pass through the kernel's qdisc (traffic control) layer, which is fine for the vast majority of use cases. For traffic generator appliances using packet sockets that intend to brute-force flood the network—for example, to test devices under load in a similar fashion to pktgen—this layer can be bypassed by setting this integer option to 1. A side effect is that packet buffering in the qdisc layer is avoided, which will lead to increased drops when network device transmit queues are busy; therefore, use at your own risk.

Ioctls

SIOCGSTAMP can be used to receive the timestamp of the last received packet. Argument is a *struct timeval* variable.

In addition, all standard ioctls defined in **netdevice**(7) and **socket**(7) are valid on packet sockets.

Error handling

Packet sockets do no error handling other than errors occurred while passing the packet to the device driver. They don't have the concept of a pending error.

ERRORS

EADDRNOTAVAIL
Unknown multicast group address passed.

EFAULT
User passed invalid memory address.

EINVAL
Invalid argument.

EMSGSIZE
Packet is bigger than interface MTU.

ENETDOWN
Interface is not up.

ENOBUFS
Not enough memory to allocate the packet.

ENODEV
Unknown device name or interface index specified in interface address.

ENOENT
No packet received.

ENOTCONN
No interface address passed.

ENXIO
Interface address contained an invalid interface index.

EPERM
User has insufficient privileges to carry out this operation.

In addition, other errors may be generated by the low-level driver.

VERSIONS

AF_PACKET is a new feature in Linux 2.2. Earlier Linux versions supported only **SOCK_PACKET**.

NOTES

For portable programs it is suggested to use **AF_PACKET** via **pcap**(3); although this covers only a subset of the **AF_PACKET** features.

The **SOCK_DGRAM** packet sockets make no attempt to create or parse the IEEE 802.2 LLC header for a IEEE 802.3 frame. When **ETH_P_802_3** is specified as protocol for sending the kernel creates the 802.3 frame and fills out the length field; the user has to supply the LLC header to get a fully conforming packet. Incoming 802.3 packets are not multiplexed on the DSAP/SSAP protocol fields; instead they are supplied to the user as protocol **ETH_P_802_2** with the LLC header prefixed. It is thus not possible to bind to **ETH_P_802_3**; bind to **ETH_P_802_2** instead and do the protocol multiplex yourself. The default for sending is the standard Ethernet DIX encapsulation with the protocol filled in.

Packet sockets are not subject to the input or output firewall chains.

Compatibility

In Linux 2.0, the only way to get a packet socket was with the call:

```
socket(AF_INET, SOCK_PACKET, protocol)
```

This is still supported, but deprecated and strongly discouraged. The main difference between the two methods is that **SOCK_PACKET** uses the old *struct sockaddr_pkt* to specify an interface, which doesn't provide physical-layer independence.

```
struct sockaddr_pkt {
    unsigned short spkt_family;
    unsigned char  spkt_device[14];
    unsigned short spkt_protocol;
```

```
};
```

spkt_family contains the device type, *spkt_protocol* is the IEEE 802.3 protocol type as defined in <*sys/if_ether.h*> and *spkt_device* is the device name as a null-terminated string, for example, eth0.

This structure is obsolete and should not be used in new code.

BUGS

LLC header handling

The IEEE 802.2/803.3 LLC handling could be considered as a bug.

MSG_TRUNC issues

The **MSG_TRUNC recvmsg**(2) extension is an ugly hack and should be replaced by a control message. There is currently no way to get the original destination address of packets via **SOCK_DGRAM**.

spkt_device device name truncation

The *spkt_device* field of *sockaddr_pkt* has a size of 14 bytes, which is less than the constant **IFNAMSIZ** defined in<*net/if .h*> which is 16 bytes and describes the system limit for a network interface name. This means the names of network devices longer than 14 bytes will be truncated to fit into *spkt_device*. All these lengths include the terminating null byte ('\0').

Issues from this with old code typically show up with very long interface names used by the **Predictable Network Interface Names** feature enabled by default in many modern Linux distributions.

The preferred solution is to rewrite code to avoid **SOCK_PACKET**. Possible user solutions are to disable **Predictable Network Interface Names** or to rename the interface to a name of at most 13 bytes, for example using the **ip**(8) tool.

Documentation issues

Socket filters are not documented.

SEE ALSO

socket(2), **pcap**(3), **capabilities**(7), **ip**(7), **raw**(7), **socket**(7), **ip**(8),

RFC 894 for the standard IP Ethernet encapsulation. RFC 1700 for the IEEE 802.3 IP encapsulation.

The <*linux/if_ether.h*> include file for physical-layer protocols.

The Linux kernel source tree. *Documentation/networking/filter.rst* describes how to apply Berkeley Packet Filters to packet sockets. *tools/testing/selftests/net/psock_tpacket.c* contains example source code for all available versions of **PACKET_RX_RING** and **PACKET_TX_RING**.