NAME

packet, PF_PACKET − packet interface on device level.

SYNOPSIS

```
#include <sys/socket.h>
#include <features.h>     /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>          /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>        /* The L2 protocols */
#endif
```

packet_socket = socket(PF_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**. *protocol* is the IEEE 802.3 protocol number in network 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.

Only processes with effective uid 0 or the **CAP_NET_RAW** capability may open packet sockets.

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_RAW** is similar to but not compatible with the obsolete **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 only get packets from a specific interface use **bind**(2) specifying an address in a **struct sockaddr_ll** to bind the packet socket to an interface. Only the **sll_protocol** and the **sll_ifindex** address fields are used for purposes of binding.

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

When the **MSG_TRUNC** flag is passed to **recvmsg**(2), **recv**(2), **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 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;   /* Header type */
    unsigned char   sll_pkttype;  /* Packet type */
    unsigned char   sll_halen;    /* Length of address */
    unsigned char   sll_addr[8];  /* Physical layer address */
};
```

sll_protocol is the standard ethernet protocol type in network 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 legal for binding). **sll_hatype** is a 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 originated from the local host that is looped back to a packet socket. These types make only sense for receiving. **sll_addr** and **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**. The other fields should be 0. **sll_hatype** and **sll_pkttype** are set on received packets for your information. For bind only **sll_protocol** and **sll_ifindex** are used.

SOCKET OPTIONS

Packet sockets can be used to configure physical layer multicasting and promiscuous mode. It works by calling **setsockopt**(2) on a packet socket for SOL_PACKET and one of the options **PACKET_ADD_MEMBERSHIP** to add a binding or **PACKET_DROP_MEMBERSHIP** to drop 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** parameter 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.

IOCTLS

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

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.

COMPATIBILITY

In Linux 2.0, the only way to get a packet socket was by calling **socket(PF_INET, SOCK_PACKET,** *protocol***)**. This is still supported but strongly deprecated. 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, e.g. eth0.

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

NOTES

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

ERRORS

ENETDOWN
Interface is not up.

ENOTCONN
No interface address passed.

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

EMSGSIZE
Packet is bigger than interface MTU.

ENOBUFS
Not enough memory to allocate the packet.

EFAULT
User passed invalid memory address.

EINVAL
Invalid argument.

ENXIO
Interface address contained illegal interface index.

EPERM
User has insufficient privileges to carry out this operation.

EADDRNOTAVAIL
Unknown multicast group address passed.

ENOENT
No packet received.

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

VERSIONS

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

BUGS

glibc 2.1 does not have a define for **SOL_PACKET.** The suggested workaround is to use

```
#ifndef SOL_PACKET
#define SOL_PACKET 263
#endif
```

This is fixed in later glibc versions and also does not occur on libc5 systems.

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

Socket filters are not documented.

The *MSG_TRUNC* recvmsg 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.

CREDITS

This man page was written by Andi Kleen with help from Matthew Wilcox. PF_PACKET in Linux 2.2 was implemented by Alexey Kuznetsov, based on code by Alan Cox and others.

SEE ALSO

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

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.