Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version: 2.0.40 2.2.26 2.4.37 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 ***4.2***

Linux/include/net/ip.h

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              Definitions for the IP module.
 *
 * Version:     @(#)ip.h        1.0.2   05/07/93
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 *
 * Changes:
 *              Mike McLagan    :       Routing by source
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 */
#ifndef _IP_H
#define _IP_H

#include <linux/types.h>
#include <linux/ip.h>
#include <linux/in.h>
#include <linux/skbuff.h>

#include <net/inet_sock.h>
#include <net/route.h>
#include <net/snmp.h>
#include <net/flow.h>
#include <net/flow_dissector.h>

struct sock;

struct inet_skb_parm {
        struct ip_options       opt;            /* Compiled IP options          */
        unsigned char           flags;

#define IPSKB_FORWARDED         BIT(0)
#define IPSKB_XFRM_TUNNEL_SIZE  BIT(1)
#define IPSKB_XFRM_TRANSFORMED  BIT(2)
#define IPSKB_FRAG_COMPLETE     BIT(3)
#define IPSKB_REROUTED          BIT(4)
#define IPSKB_DOREDIRECT        BIT(5)
#define IPSKB_FRAG_PMTU         BIT(6)

        u16                     frag_max_size;
};

static inline unsigned int ip_hdrlen(const struct sk_buff *skb)
{
        return ip_hdr(skb)->ihl * 4;
}

struct ipcm_cookie {
        __be32                  addr;
        int                     oif;
        struct ip_options_rcu   *opt;
        __u8                    tx_flags;
        __u8                    ttl;
        __s16                   tos;
        char                    priority;
};

#define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
#define PKTINFO_SKB_CB(skb) ((struct in_pktinfo *)((skb)->cb))

struct ip_ra_chain {
        struct ip_ra_chain __rcu *next;
        struct sock             *sk;
        union {
                void                    (*destructor)(struct sock *);
                struct sock             *saved_sk;
        };
        struct rcu_head         rcu;
};

extern struct ip_ra_chain __rcu *ip_ra_chain;

/* IP flags. */
#define IP_CE           0x8000          /* Flag: "Congestion"           */
#define IP_DF           0x4000          /* Flag: "Don't Fragment"       */
#define IP_MF           0x2000          /* Flag: "More Fragments"       */
#define IP_OFFSET       0x1FFF          /* "Fragment Offset" part       */

#define IP_FRAG_TIME    (30 * HZ)               /* fragment Lifetime    */

struct msghdr;
struct net_device;
struct packet_type;
struct rtable;
struct sockaddr;

int igmp_mc_init(void);

/*
 *      Functions provided by ip.c
 */

int ip_build_and_send_pkt(struct sk_buff *skb, struct sock *sk,
                          __be32 saddr, __be32 daddr,
                          struct ip_options_rcu *opt);
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt,
           struct net_device *orig_dev);
int ip_local_deliver(struct sk_buff *skb);
int ip_mr_input(struct sk_buff *skb);
int ip_output(struct sock *sk, struct sk_buff *skb);
int ip_mc_output(struct sock *sk, struct sk_buff *skb);
int ip_do_fragment(struct sock *sk, struct sk_buff *skb,
                   int (*output)(struct sock *, struct sk_buff *));
void ip_send_check(struct iphdr *ip);
int __ip_local_out(struct sk_buff *skb);
int ip_local_out_sk(struct sock *sk, struct sk_buff *skb);
static inline int ip_local_out(struct sk_buff *skb)
{
        return ip_local_out_sk(skb->sk, skb);
}

int ip_queue_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl);
void ip_init(void);
int ip_append_data(struct sock *sk, struct flowi4 *fl4,
                   int getfrag(void *from, char *to, int offset, int len,
                               int odd, struct sk_buff *skb),
                   void *from, int len, int protolen,
                   struct ipcm_cookie *ipc,
                   struct rtable **rt,
                   unsigned int flags);
```

```
int ip_generic_getfrag(void *from, char *to, int offset, int len, int odd,
                       struct sk_buff *skb);
ssize_t ip_append_page(struct sock *sk, struct flowi4 *fl4, struct page *page,
                       int offset, size_t size, int flags);
struct sk_buff *__ip_make_skb(struct sock *sk, struct flowi4 *fl4,
                              struct sk_buff_head *queue,
                              struct inet_cork *cork);
int ip_send_skb(struct net *net, struct sk_buff *skb);
int ip_push_pending_frames(struct sock *sk, struct flowi4 *fl4);
void ip_flush_pending_frames(struct sock *sk);
struct sk_buff *ip_make_skb(struct sock *sk, struct flowi4 *fl4,
                            int getfrag(void *from, char *to, int offset,
                                        int len, int odd, struct sk_buff *skb),
                            void *from, int length, int transhdrlen,
                            struct ipcm_cookie *ipc, struct rtable **rtp,
                            unsigned int flags);

static inline struct sk_buff *ip_finish_skb(struct sock *sk, struct flowi4 *fl4)
{
        return __ip_make_skb(sk, fl4, &sk->sk_write_queue, &inet_sk(sk)->cork.base);
}

static inline __u8 get_rttos(struct ipcm_cookie* ipc, struct inet_sock *inet)
{
        return (ipc->tos != -1) ? RT_TOS(ipc->tos) : RT_TOS(inet->tos);
}

static inline __u8 get_rtconn_flags(struct ipcm_cookie* ipc, struct sock* sk)
{
        return (ipc->tos != -1) ? RT_CONN_FLAGS_TOS(sk, ipc->tos) : RT_CONN_FLAGS(sk);
}

/* datagram.c */
int __ip4_datagram_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len);
int ip4_datagram_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len);

void ip4_datagram_release_cb(struct sock *sk);

struct ip_reply_arg {
        struct kvec iov[1];
        int         flags;
        __wsum      csum;
        int         csumoffset; /* u16 offset of csum in iov[0].iov_base */
                                /* -1 if not needed */
        int         bound_dev_if;
        u8          tos;
};

#define IP_REPLY_ARG_NOSRCCHECK 1

static inline __u8 ip_reply_arg_flowi_flags(const struct ip_reply_arg *arg)
{
        return (arg->flags & IP_REPLY_ARG_NOSRCCHECK) ? FLOWI_FLAG_ANYSRC : 0;
}

void ip_send_unicast_reply(struct sock *sk, struct sk_buff *skb,
                           const struct ip_options *sopt,
                           __be32 daddr, __be32 saddr,
                           const struct ip_reply_arg *arg,
                           unsigned int len);

#define IP_INC_STATS(net, field)        SNMP_INC_STATS64((net)->mib.ip_statistics, field)
#define IP_INC_STATS_BH(net, field)     SNMP_INC_STATS64_BH((net)->mib.ip_statistics, field)
#define IP_ADD_STATS(net, field, val)   SNMP_ADD_STATS64((net)->mib.ip_statistics, field, val)
#define IP_ADD_STATS_BH(net, field, val) SNMP_ADD_STATS64_BH((net)->mib.ip_statistics, field, val)
#define IP_UPD_PO_STATS(net, field, val) SNMP_UPD_PO_STATS64((net)->mib.ip_statistics, field, val)
#define IP_UPD_PO_STATS_BH(net, field, val) SNMP_UPD_PO_STATS64_BH((net)->mib.ip_statistics, field, val)
#define NET_INC_STATS(net, field)       SNMP_INC_STATS((net)->mib.net_statistics, field)
#define NET_INC_STATS_BH(net, field)    SNMP_INC_STATS_BH((net)->mib.net_statistics, field)
#define NET_INC_STATS_USER(net, field)  SNMP_INC_STATS_USER((net)->mib.net_statistics, field)
#define NET_ADD_STATS(net, field, adnd) SNMP_ADD_STATS((net)->mib.net_statistics, field, adnd)
#define NET_ADD_STATS_BH(net, field, adnd) SNMP_ADD_STATS_BH((net)->mib.net_statistics, field, adnd)
#define NET_ADD_STATS_USER(net, field, adnd) SNMP_ADD_STATS_USER((net)->mib.net_statistics, field, adnd)

unsigned long snmp_fold_field(void __percpu *mib, int offt);
```

```
#if BITS_PER_LONG==32
u64 snmp_fold_field64(void __percpu *mib, int offt, size_t sync_off);
#else
static inline u64 snmp_fold_field64(void __percpu *mib, int offt, size_t syncp_off)
{
        return snmp_fold_field(mib, offt);
}
#endif

void inet_get_local_port_range(struct net *net, int *low, int *high);

#ifdef CONFIG_SYSCTL
static inline int inet_is_local_reserved_port(struct net *net, int port)
{
        if (!net->ipv4.sysctl_local_reserved_ports)
                return 0;
        return test_bit(port, net->ipv4.sysctl_local_reserved_ports);
}

static inline bool sysctl_dev_name_is_allowed(const char *name)
{
        return strcmp(name, "default") != 0  && strcmp(name, "all") != 0;
}

#else
static inline int inet_is_local_reserved_port(struct net *net, int port)
{
        return 0;
}
#endif

/* From inetpeer.c */
extern int inet_peer_threshold;
extern int inet_peer_minttl;
extern int inet_peer_maxttl;

/* From ip_input.c */
extern int sysctl_ip_early_demux;

/* From ip_output.c */
extern int sysctl_ip_dynaddr;

void ipfrag_init(void);

void ip_static_sysctl_init(void);

#define IP4_REPLY_MARK(net, mark) \
        ((net)->ipv4.sysctl_fwmark_reflect ? (mark) : 0)

static inline bool ip_is_fragment(const struct iphdr *iph)
{
        return (iph->frag_off & htons(IP_MF | IP_OFFSET)) != 0;
}

#ifdef CONFIG_INET
#include <net/dst.h>

/* The function in 2.2 was invalid, producing wrong result for
 * check=0xFEFF. It was noticed by Arthur Skawina _year_ ago. --ANK(000625) */
static inline
int ip_decrease_ttl(struct iphdr *iph)
{
        u32 check = (__force u32)iph->check;
        check += (__force u32)htons(0x0100);
        iph->check = (__force __sum16)(check + (check>=0xFFFF));
        return --iph->ttl;
}

static inline
int ip_dont_fragment(struct sock *sk, struct dst_entry *dst)
{
        return  inet_sk(sk)->pmtudisc == IP_PMTUDISC_DO ||
                (inet_sk(sk)->pmtudisc == IP_PMTUDISC_WANT &&
                 !(dst_metric_locked(dst, RTAX_MTU)));
}
```

```c

static inline bool ip_sk_accept_pmtu(const struct sock *sk)
{
        return inet_sk(sk)->pmtudisc != IP_PMTUDISC_INTERFACE &&
               inet_sk(sk)->pmtudisc != IP_PMTUDISC_OMIT;
}

static inline bool ip_sk_use_pmtu(const struct sock *sk)
{
        return inet_sk(sk)->pmtudisc < IP_PMTUDISC_PROBE;
}

static inline bool ip_sk_ignore_df(const struct sock *sk)
{
        return inet_sk(sk)->pmtudisc < IP_PMTUDISC_DO ||
               inet_sk(sk)->pmtudisc == IP_PMTUDISC_OMIT;
}

static inline unsigned int ip_dst_mtu_maybe_forward(const struct dst_entry *dst,
                                                    bool forwarding)
{
        struct net *net = dev_net(dst->dev);

        if (net->ipv4.sysctl_ip_fwd_use_pmtu ||
            dst_metric_locked(dst, RTAX_MTU) ||
            !forwarding)
                return dst_mtu(dst);

        return min(dst->dev->mtu, IP_MAX_MTU);
}

static inline unsigned int ip_skb_dst_mtu(const struct sk_buff *skb)
{
        if (!skb->sk || ip_sk_use_pmtu(skb->sk)) {
                bool forwarding = IPCB(skb)->flags & IPSKB_FORWARDED;
                return ip_dst_mtu_maybe_forward(skb_dst(skb), forwarding);
        } else {
                return min(skb_dst(skb)->dev->mtu, IP_MAX_MTU);
        }
}

u32 ip_idents_reserve(u32 hash, int segs);
void __ip_select_ident(struct net *net, struct iphdr *iph, int segs);

static inline void ip_select_ident_segs(struct net *net, struct sk_buff *skb,
                                        struct sock *sk, int segs)
{
        struct iphdr *iph = ip_hdr(skb);

        if ((iph->frag_off & htons(IP_DF)) && !skb->ignore_df) {
                /* This is only to work around buggy Windows95/2000
                 * VJ compression implementations.  If the ID field
                 * does not change, they drop every other packet in
                 * a TCP stream using header compression.
                 */
                if (sk && inet_sk(sk)->inet_daddr) {
                        iph->id = htons(inet_sk(sk)->inet_id);
                        inet_sk(sk)->inet_id += segs;
                } else {
                        iph->id = 0;
                }
        } else {
                __ip_select_ident(net, iph, segs);
        }
}

static inline void ip_select_ident(struct net *net, struct sk_buff *skb,
                                   struct sock *sk)
{
        ip_select_ident_segs(net, skb, sk, 1);
}

static inline __wsum inet_compute_pseudo(struct sk_buff *skb, int proto)
{
        return csum_tcpudp_nofold(ip_hdr(skb)->saddr, ip_hdr(skb)->daddr,
                                  skb->len, proto, 0);
}

/* copy IPv4 saddr & daddr to flow_keys, possibly using 64bit load/store
 * Equivalent to :      flow->v4addrs.src = iph->saddr;
 *                      flow->v4addrs.dst = iph->daddr;
 */
static inline void iph_to_flow_copy_v4addrs(struct flow_keys *flow,
                                            const struct iphdr *iph)
{
        BUILD_BUG_ON(offsetof(typeof(flow->addrs), v4addrs.dst) !=
                     offsetof(typeof(flow->addrs), v4addrs.src) +
                              sizeof(flow->addrs.v4addrs.src));
        memcpy(&flow->addrs.v4addrs, &iph->saddr, sizeof(flow->addrs.v4addrs));
        flow->control.addr_type = FLOW_DISSECTOR_KEY_IPV4_ADDRS;
}

static inline void inet_set_txhash(struct sock *sk)
{
        struct inet_sock *inet = inet_sk(sk);
        struct flow_keys keys;

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

        keys.addrs.v4addrs.src = inet->inet_saddr;
        keys.addrs.v4addrs.dst = inet->inet_daddr;
        keys.control.addr_type = FLOW_DISSECTOR_KEY_IPV4_ADDRS;
        keys.ports.src = inet->inet_sport;
        keys.ports.dst = inet->inet_dport;

        sk->sk_txhash = flow_hash_from_keys(&keys);
}

static inline __wsum inet_gro_compute_pseudo(struct sk_buff *skb, int proto)
{
        const struct iphdr *iph = skb_gro_network_header(skb);

        return csum_tcpudp_nofold(iph->saddr, iph->daddr,
                                  skb_gro_len(skb), proto, 0);
}

/*
 *      Map a multicast IP onto multicast MAC for type ethernet.
 */

static inline void ip_eth_mc_map(__be32 naddr, char *buf)
{
        __u32 addr=ntohl(naddr);
        buf[0]=0x01;
        buf[1]=0x00;
        buf[2]=0x5e;
        buf[5]=addr&0xFF;
        addr>>=8;
        buf[4]=addr&0xFF;
        addr>>=8;
        buf[3]=addr&0x7F;
}

/*
 *      Map a multicast IP onto multicast MAC for type IP-over-InfiniBand.
 *      Leave P_Key as 0 to be filled in by driver.
 */

static inline void ip_ib_mc_map(__be32 naddr, const unsigned char *broadcast, char *buf)
{
        __u32 addr;
        unsigned char scope = broadcast[5] & 0xF;

        buf[0]  = 0;            /* Reserved */
        buf[1]  = 0xff;         /* Multicast QPN */
        buf[2]  = 0xff;
        buf[3]  = 0xff;
        addr    = ntohl(naddr);
        buf[4]  = 0xff;
        buf[5]  = 0x10 | scope; /* scope from broadcast address */
        buf[6]  = 0x40;         /* IPv4 signature */
        buf[7]  = 0x1b;
        buf[8]  = broadcast[8];         /* P_Key */
        buf[9]  = broadcast[9];
        buf[10] = 0;
        buf[11] = 0;
        buf[12] = 0;
        buf[13] = 0;
        buf[14] = 0;
        buf[15] = 0;
        buf[19] = addr & 0xff;
        addr  >>= 8;
        buf[18] = addr & 0xff;
        addr  >>= 8;
        buf[17] = addr & 0xff;
        addr  >>= 8;
        buf[16] = addr & 0x0f;
}

static inline void ip_ipgre_mc_map(__be32 naddr, const unsigned char *broadcast, char *buf)
{
        if ((broadcast[0] | broadcast[1] | broadcast[2] | broadcast[3]) != 0)
                memcpy(buf, broadcast, 4);
        else
                memcpy(buf, &naddr, sizeof(naddr));
}

#if IS_ENABLED(CONFIG_IPV6)
#include <linux/ipv6.h>
#endif

static __inline__ void inet_reset_saddr(struct sock *sk)
{
        inet_sk(sk)->inet_rcv_saddr = inet_sk(sk)->inet_saddr = 0;
#if IS_ENABLED(CONFIG_IPV6)
        if (sk->sk_family == PF_INET6) {
                struct ipv6_pinfo *np = inet6_sk(sk);

                memset(&np->saddr, 0, sizeof(np->saddr));
                memset(&sk->sk_v6_rcv_saddr, 0, sizeof(sk->sk_v6_rcv_saddr));
        }
#endif
}

#endif

bool ip_call_ra_chain(struct sk_buff *skb);

/*
 *      Functions provided by ip_fragment.c
 */

enum ip_defrag_users {
        IP_DEFRAG_LOCAL_DELIVER,
        IP_DEFRAG_CALL_RA_CHAIN,
        IP_DEFRAG_CONNTRACK_IN,
        __IP_DEFRAG_CONNTRACK_IN_END    = IP_DEFRAG_CONNTRACK_IN + USHRT_MAX,
        IP_DEFRAG_CONNTRACK_OUT,
        __IP_DEFRAG_CONNTRACK_OUT_END   = IP_DEFRAG_CONNTRACK_OUT + USHRT_MAX,
        IP_DEFRAG_CONNTRACK_BRIDGE_IN,
        __IP_DEFRAG_CONNTRACK_BRIDGE_IN = IP_DEFRAG_CONNTRACK_BRIDGE_IN + USHRT_MAX,
        IP_DEFRAG_VS_IN,
        IP_DEFRAG_VS_OUT,
        IP_DEFRAG_VS_FWD,
        IP_DEFRAG_AF_PACKET,
        IP_DEFRAG_MACVLAN,
};

/* Return true if the value of 'user' is between 'lower_bond'
 * and 'upper_bond' inclusively.
 */
static inline bool ip_defrag_user_in_between(u32 user,
                                             enum ip_defrag_users lower_bond,
                                             enum ip_defrag_users upper_bond)
{
        return user >= lower_bond && user <= upper_bond;
}

int ip_defrag(struct sk_buff *skb, u32 user);
#ifdef CONFIG_INET
struct sk_buff *ip_check_defrag(struct sk_buff *skb, u32 user);
#else
static inline struct sk_buff *ip_check_defrag(struct sk_buff *skb, u32 user)
{
        return skb;
}
#endif
int ip_frag_mem(struct net *net);

/*
 *      Functions provided by ip_forward.c
 */

int ip_forward(struct sk_buff *skb);

/*
 *      Functions provided by ip_options.c
 */

void ip_options_build(struct sk_buff *skb, struct ip_options *opt,
                      __be32 daddr, struct rtable *rt, int is_frag);

int __ip_options_echo(struct ip_options *dopt, struct sk_buff *skb,
                      const struct ip_options *sopt);
static inline int ip_options_echo(struct ip_options *dopt, struct sk_buff *skb)
{
        return __ip_options_echo(dopt, skb, &IPCB(skb)->opt);
}

void ip_options_fragment(struct sk_buff *skb);
int ip_options_compile(struct net *net, struct ip_options *opt,
                       struct sk_buff *skb);
int ip_options_get(struct net *net, struct ip_options_rcu **optp,
                   unsigned char *data, int optlen);
int ip_options_get_from_user(struct net *net, struct ip_options_rcu **optp,
                             unsigned char __user *data, int optlen);
void ip_options_undo(struct ip_options *opt);
void ip_forward_options(struct sk_buff *skb);
int ip_options_rcv_srr(struct sk_buff *skb);

/*
 *      Functions provided by ip_sockglue.c
 */

void ipv4_pktinfo_prepare(const struct sock *sk, struct sk_buff *skb);
void ip_cmsg_recv_offset(struct msghdr *msg, struct sk_buff *skb, int offset);
int ip_cmsg_send(struct net *net, struct msghdr *msg,
                 struct ipcm_cookie *ipc, bool allow_ipv6);
int ip_setsockopt(struct sock *sk, int level, int optname, char __user *optval,
                  unsigned int optlen);
int ip_getsockopt(struct sock *sk, int level, int optname, char __user *optval,
                  int __user *optlen);
int compat_ip_setsockopt(struct sock *sk, int level, int optname,
                         char __user *optval, unsigned int optlen);
int compat_ip_getsockopt(struct sock *sk, int level, int optname,
                         char __user *optval, int __user *optlen);
int ip_ra_control(struct sock *sk, unsigned char on,
                  void (*destructor)(struct sock *));

int ip_recv_error(struct sock *sk, struct msghdr *msg, int len, int *addr_len);
void ip_icmp_error(struct sock *sk, struct sk_buff *skb, int err, __be16 port,
                   u32 info, u8 *payload);
void ip_local_error(struct sock *sk, int err, __be32 daddr, __be16 dport,
                    u32 info);

static inline void ip_cmsg_recv(struct msghdr *msg, struct sk_buff *skb)
{
        ip_cmsg_recv_offset(msg, skb, 0);
}

```

```
bool icmp_global_allow(void);
extern int sysctl_icmp_msgs_per_sec;
extern int sysctl_icmp_msgs_burst;

#ifdef CONFIG_PROC_FS
int ip_misc_proc_init(void);
#endif

#endif  /* _IP_H */

```

This page was automatically generated by LXR 0.3.1 (source). • Linux is a registered trademark of Linus Torvalds • Contact us

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog