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/sock.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 AF_INET socket handler.
 *
 * Version:     @(#)sock.h      1.0.4   05/13/93
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Corey Minyard <wf-rch!minyard@relay.EU.net>
 *              Florian La Roche <flla@stud.uni-sb.de>
 *
 * Fixes:
 *              Alan Cox        :       Volatiles in skbuff pointers. See
 *                                      skbuff comments. May be overdone,
 *                                      better to prove they can be removed
 *                                      than the reverse.
 *              Alan Cox        :       Added a zapped field for tcp to note
 *                                      a socket is reset and must stay shut up
 *              Alan Cox        :       New fields for options
 *      Pauline Middelink       :       identd support
 *              Alan Cox        :       Eliminate low level recv/recvfrom
 *              David S. Miller :       New socket lookup architecture.
 *              Steve Whitehouse:       Default routines for sock_ops
 *              Arnaldo C. Melo :       removed net_pinfo, tp_pinfo and made
 *                                      protinfo be just a void pointer, as the
 *                                      protocol specific parts were moved to
 *                                      respective headers and ipv4/v6, etc now
 *                                      use private slabcaches for its socks
 *              Pedro Hortas    :       New flags field for socket options
 *
 *
 *              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 _SOCK_H
#define _SOCK_H

#include <linux/hardirq.h>
#include <linux/kernel.h>
#include <linux/list.h>
#include <linux/list_nulls.h>
#include <linux/timer.h>
```

```
#include <linux/cache.h>
#include <linux/bitops.h>
#include <linux/lockdep.h>
#include <linux/netdevice.h>
#include <linux/skbuff.h>       /* struct sk_buff */
#include <linux/mm.h>
#include <linux/security.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/page_counter.h>
#include <linux/memcontrol.h>
#include <linux/static_key.h>
#include <linux/sched.h>

#include <linux/filter.h>
#include <linux/rculist_nulls.h>
#include <linux/poll.h>

#include <linux/atomic.h>
#include <net/dst.h>
#include <net/checksum.h>
#include <net/tcp_states.h>
#include <linux/net_tstamp.h>

struct cgroup;
struct cgroup_subsys;
#ifdef CONFIG_NET
int mem_cgroup_sockets_init(struct mem_cgroup *memcg, struct cgroup_subsys *ss);
void mem_cgroup_sockets_destroy(struct mem_cgroup *memcg);
#else
static inline
int mem_cgroup_sockets_init(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
{
        return 0;
}
static inline
void mem_cgroup_sockets_destroy(struct mem_cgroup *memcg)
{
}
#endif
/*
 * This structure really needs to be cleaned up.
 * Most of it is for TCP, and not used by any of
 * the other protocols.
 */

/* Define this to get the SOCK_DBG debugging facility. */
#define SOCK_DEBUGGING
#ifdef SOCK_DEBUGGING
#define SOCK_DEBUG(sk, msg...) do { if ((sk) && sock_flag((sk), SOCK_DBG)) \
                                        printk(KERN_DEBUG msg); } while (0)
#else
/* Validate arguments and do nothing */
static inline __printf(2, 3)
void SOCK_DEBUG(const struct sock *sk, const char *msg, ...)
{
}
#endif

/* This is the per-socket lock.  The spinlock provides a synchronization
 * between user contexts and software interrupt processing, whereas the
 * mini-semaphore synchronizes multiple users amongst themselves.
 */
typedef struct {
        spinlock_t              slock;
        int                     owned;
        wait_queue_head_t       wq;
        /*
```

```
         * We express the mutex-alike socket_lock semantics
         * to the lock validator by explicitly managing
         * the slock as a lock variant (in addition to
         * the slock itself):
         */
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map dep_map;
#endif
} socket_lock_t;

struct sock;
struct proto;
struct net;

typedef __u32 __bitwise __portpair;
typedef __u64 __bitwise __addrpair;

/**
 *      struct sock_common - minimal network layer representation of sockets
 *      @skc_daddr: Foreign IPv4 addr
 *      @skc_rcv_saddr: Bound local IPv4 addr
 *      @skc_hash: hash value used with various protocol lookup tables
 *      @skc_u16hashes: two u16 hash values used by UDP lookup tables
 *      @skc_dport: placeholder for inet_dport/tw_dport
 *      @skc_num: placeholder for inet_num/tw_num
 *      @skc_family: network address family
 *      @skc_state: Connection state
 *      @skc_reuse: %SO_REUSEADDR setting
 *      @skc_reuseport: %SO_REUSEPORT setting
 *      @skc_bound_dev_if: bound device index if != 0
 *      @skc_bind_node: bind hash linkage for various protocol lookup tables
 *      @skc_portaddr_node: second hash linkage for UDP/UDP-Lite protocol
 *      @skc_prot: protocol handlers inside a network family
 *      @skc_net: reference to the network namespace of this socket
 *      @skc_node: main hash linkage for various protocol lookup tables
 *      @skc_nulls_node: main hash linkage for TCP/UDP/UDP-Lite protocol
 *      @skc_tx_queue_mapping: tx queue number for this connection
 *      @skc_refcnt: reference count
 *
 *      This is the minimal network layer representation of sockets, the header
 *      for struct sock and struct inet_timewait_sock.
 */
struct sock_common {
        /* skc_daddr and skc_rcv_saddr must be grouped on a 8 bytes aligned
         * address on 64bit arches : cf INET_MATCH()
         */
        union {
                __addrpair      skc_addrpair;
                struct {
                        __be32  skc_daddr;
                        __be32  skc_rcv_saddr;
                };
        };
        union  {
                unsigned int    skc_hash;
                __u16           skc_u16hashes[2];
        };
        /* skc_dport && skc_num must be grouped as well */
        union {
                __portpair      skc_portpair;
                struct {
                        __be16  skc_dport;
                        __u16   skc_num;
                };
        };

        unsigned short          skc_family;
        volatile unsigned char  skc_state;
        unsigned char           skc_reuse:4;
        unsigned char           skc_reuseport:1;
        unsigned char           skc_ipv6only:1;
        unsigned char           skc_net_refcnt:1;
        int                     skc_bound_dev_if;
        union {
                struct hlist_node       skc_bind_node;
                struct hlist_nulls_node skc_portaddr_node;
        };
        struct proto            *skc_prot;
        possible_net_t          skc_net;

#if IS_ENABLED(CONFIG_IPV6)
        struct in6_addr         skc_v6_daddr;
        struct in6_addr         skc_v6_rcv_saddr;
#endif

        atomic64_t              skc_cookie;

        /*
         * fields between dontcopy_begin/dontcopy_end
         * are not copied in sock_copy()
         */
        /* private: */
        int                     skc_dontcopy_begin[0];
        /* public: */
        union {
                struct hlist_node       skc_node;
                struct hlist_nulls_node skc_nulls_node;
        };
        int                     skc_tx_queue_mapping;
        atomic_t                skc_refcnt;
        /* private: */
        int                     skc_dontcopy_end[0];
        /* public: */
};

struct cg_proto;
/**
 *      struct sock - network layer representation of sockets
 *      @__sk_common: shared layout with inet_timewait_sock
 *      @sk_shutdown: mask of %SEND_SHUTDOWN and/or %RCV_SHUTDOWN
 *      @sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
 *      @sk_lock:       synchronizer
 *      @sk_rcvbuf: size of receive buffer in bytes
 *      @sk_wq: sock wait queue and async head
 *      @sk_rx_dst: receive input route used by early demux
 *      @sk_dst_cache: destination cache
 *      @sk_dst_lock: destination cache lock
 *      @sk_policy: flow policy
 *      @sk_receive_queue: incoming packets
 *      @sk_wmem_alloc: transmit queue bytes committed
 *      @sk_write_queue: Packet sending queue
 *      @sk_omem_alloc: "o" is "option" or "other"
 *      @sk_wmem_queued: persistent queue size
 *      @sk_forward_alloc: space allocated forward
 *      @sk_napi_id: id of the last napi context to receive data for sk
 *      @sk_ll_usec: usecs to busypoll when there is no data
 *      @sk_allocation: allocation mode
 *      @sk_pacing_rate: Pacing rate (if supported by transport/packet scheduler)
 *      @sk_max_pacing_rate: Maximum pacing rate (%SO_MAX_PACING_RATE)
 *      @sk_sndbuf: size of send buffer in bytes
 *      @sk_flags: %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPALIVE,
 *                 %SO_OOBINLINE settings, %SO_TIMESTAMPING settings
 *      @sk_no_check_tx: %SO_NO_CHECK setting, set checksum in TX packets
 *      @sk_no_check_rx: allow zero checksum in RX packets
 *      @sk_route_caps: route capabilities (e.g. %NETIF_F_TSO)
 *      @sk_route_nocaps: forbidden route capabilities (e.g NETIF_F_GSO_MASK)
 *      @sk_gso_type: GSO type (e.g. %SKB_GSO_TCPV4)
 *      @sk_gso_max_size: Maximum GSO segment size to build
 *      @sk_gso_max_segs: Maximum number of GSO segments
 *      @sk_lingertime: %SO_LINGER l_linger setting
 *      @sk_backlog: always used with the per-socket spinlock held
 *      @sk_callback_lock: used with the callbacks in the end of this struct
 *      @sk_error_queue: rarely used
 *      @sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
 *                        IPV6_ADDRFORM for instance)
 *      @sk_err: last error
 *      @sk_err_soft: errors that don't cause failure but are the cause of a
 *                    persistent failure not just 'timed out'
 *      @sk_drops: raw/udp drops counter
 *      @sk_ack_backlog: current listen backlog
 *      @sk_max_ack_backlog: listen backlog set in listen()
 *      @sk_priority: %SO_PRIORITY setting
 *      @sk_cgrp_prioidx: socket group's priority map index
 *      @sk_type: socket type (%SOCK_STREAM, etc)
 *      @sk_protocol: which protocol this socket belongs in this network family
 *      @sk_peer_pid: &struct pid for this socket's peer
 *      @sk_peer_cred: %SO_PEERCRED setting
 *      @sk_rcvlowat: %SO_RCVLOWAT setting
 *      @sk_rcvtimeo: %SO_RCVTIMEO setting
 *      @sk_sndtimeo: %SO_SNDTIMEO setting
 *      @sk_rxhash: flow hash received from netif layer
 *      @sk_incoming_cpu: record cpu processing incoming packets
 *      @sk_txhash: computed flow hash for use on transmit
 *      @sk_filter: socket filtering instructions
 *      @sk_timer: sock cleanup timer
 *      @sk_stamp: time stamp of last packet received
 *      @sk_tsflags: SO_TIMESTAMPING socket options
 *      @sk_tskey: counter to disambiguate concurrent tstamp requests
 *      @sk_socket: Identd and reporting IO signals
 *      @sk_user_data: RPC layer private data
 *      @sk_frag: cached page frag
 *      @sk_peek_off: current peek_offset value
 *      @sk_send_head: front of stuff to transmit
 *      @sk_security: used by security modules
 *      @sk_mark: generic packet mark
 *      @sk_classid: this socket's cgroup classid
 *      @sk_cgrp: this socket's cgroup-specific proto data
 *      @sk_write_pending: a write to stream socket waits to start
 *      @sk_state_change: callback to indicate change in the state of the sock
 *      @sk_data_ready: callback to indicate there is data to be processed
 *      @sk_write_space: callback to indicate there is bf sending space available
 *      @sk_error_report: callback to indicate errors (e.g. %MSG_ERRQUEUE)
 *      @sk_backlog_rcv: callback to process the backlog
 *      @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
 */
struct sock {
        /*
         * Now struct inet_timewait_sock also uses sock_common, so please just
         * don't add nothing before this first member (__sk_common) --acme
         */
        struct sock_common      __sk_common;
#define sk_node                 __sk_common.skc_node
#define sk_nulls_node           __sk_common.skc_nulls_node
#define sk_refcnt               __sk_common.skc_refcnt
#define sk_tx_queue_mapping     __sk_common.skc_tx_queue_mapping

#define sk_dontcopy_begin       __sk_common.skc_dontcopy_begin
#define sk_dontcopy_end         __sk_common.skc_dontcopy_end
#define sk_hash                 __sk_common.skc_hash
#define sk_portpair             __sk_common.skc_portpair
#define sk_num                  __sk_common.skc_num
#define sk_dport                __sk_common.skc_dport
#define sk_addrpair             __sk_common.skc_addrpair
#define sk_daddr                __sk_common.skc_daddr
```

```
#define sk_rcv_saddr            __sk_common.skc_rcv_saddr
#define sk_family               __sk_common.skc_family
#define sk_state                __sk_common.skc_state
#define sk_reuse                __sk_common.skc_reuse
#define sk_reuseport            __sk_common.skc_reuseport
#define sk_ipv6only             __sk_common.skc_ipv6only
#define sk_net_refcnt           __sk_common.skc_net_refcnt
#define sk_bound_dev_if         __sk_common.skc_bound_dev_if
#define sk_bind_node            __sk_common.skc_bind_node
#define sk_prot                 __sk_common.skc_prot
#define sk_net                  __sk_common.skc_net
#define sk_v6_daddr             __sk_common.skc_v6_daddr
#define sk_v6_rcv_saddr __sk_common.skc_v6_rcv_saddr
#define sk_cookie               __sk_common.skc_cookie

        socket_lock_t           sk_lock;
        struct sk_buff_head     sk_receive_queue;
        /*
         * The backlog queue is special, it is always used with
         * the per-socket spinlock held and requires low latency
         * access. Therefore we special case it's implementation.
         * Note : rmem_alloc is in this structure to fill a hole
         * on 64bit arches, not because its logically part of
         * backlog.
         */
        struct {
                atomic_t        rmem_alloc;
                int             len;
                struct sk_buff  *head;
                struct sk_buff  *tail;
        } sk_backlog;
#define sk_rmem_alloc sk_backlog.rmem_alloc
        int                     sk_forward_alloc;
#ifdef CONFIG_RPS
        __u32                   sk_rxhash;
#endif
        u16                     sk_incoming_cpu;
        /* 16bit hole
         * Warned : sk_incoming_cpu can be set from softirq,
         * Do not use this hole without fully understanding possible issues.
         */

        __u32                   sk_txhash;
#ifdef CONFIG_NET_RX_BUSY_POLL
        unsigned int            sk_napi_id;
        unsigned int            sk_ll_usec;
#endif
        atomic_t                sk_drops;
        int                     sk_rcvbuf;

        struct sk_filter __rcu  *sk_filter;
        struct socket_wq __rcu  *sk_wq;

#ifdef CONFIG_XFRM
        struct xfrm_policy      *sk_policy[2];
#endif
        unsigned long           sk_flags;
        struct dst_entry        *sk_rx_dst;
        struct dst_entry __rcu  *sk_dst_cache;
        spinlock_t              sk_dst_lock;
        atomic_t                sk_wmem_alloc;
        atomic_t                sk_omem_alloc;
        int                     sk_sndbuf;
        struct sk_buff_head     sk_write_queue;
        kmemcheck_bitfield_begin(flags);
        unsigned int            sk_shutdown  : 2,
                                sk_no_check_tx : 1,
                                sk_no_check_rx : 1,
                                sk_userlocks : 4,
                                sk_protocol  : 8,
                                sk_type      : 16;
        kmemcheck_bitfield_end(flags);
        int                     sk_wmem_queued;
        gfp_t                   sk_allocation;
        u32                     sk_pacing_rate; /* bytes per second */
        u32                     sk_max_pacing_rate;
        netdev_features_t       sk_route_caps;
        netdev_features_t       sk_route_nocaps;
        int                     sk_gso_type;
        unsigned int            sk_gso_max_size;
        u16                     sk_gso_max_segs;
        int                     sk_rcvlowat;
        unsigned long           sk_lingertime;
        struct sk_buff_head     sk_error_queue;
        struct proto            *sk_prot_creator;
        rwlock_t                sk_callback_lock;
        int                     sk_err,
                                sk_err_soft;
        u32                     sk_ack_backlog;
        u32                     sk_max_ack_backlog;
        __u32                   sk_priority;
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
        __u32                   sk_cgrp_prioidx;
#endif
        struct pid              *sk_peer_pid;
        const struct cred       *sk_peer_cred;
        long                    sk_rcvtimeo;
        long                    sk_sndtimeo;
        struct timer_list       sk_timer;
        ktime_t                 sk_stamp;
        u16                     sk_tsflags;
        u32                     sk_tskey;
        struct socket           *sk_socket;
        void                    *sk_user_data;
        struct page_frag        sk_frag;
        struct sk_buff          *sk_send_head;
        __s32                   sk_peek_off;
        int                     sk_write_pending;
#ifdef CONFIG_SECURITY
        void                    *sk_security;
#endif
        __u32                   sk_mark;
        u32                     sk_classid;
        struct cg_proto         *sk_cgrp;
        void                    (*sk_state_change)(struct sock *sk);
        void                    (*sk_data_ready)(struct sock *sk);
        void                    (*sk_write_space)(struct sock *sk);
        void                    (*sk_error_report)(struct sock *sk);
        int                     (*sk_backlog_rcv)(struct sock *sk,
                                                  struct sk_buff *skb);
        void                    (*sk_destruct)(struct sock *sk);
};

#define __sk_user_data(sk) ((*((void __rcu **)&(sk)->sk_user_data)))

#define rcu_dereference_sk_user_data(sk)        rcu_dereference(__sk_user_data((sk)))
#define rcu_assign_sk_user_data(sk, ptr)        rcu_assign_pointer(__sk_user_data((sk)), ptr)

/*
 * SK_CAN_REUSE and SK_NO_REUSE on a socket mean that the socket is OK
 * or not whether his port will be reused by someone else. SK_FORCE_REUSE
 * on a socket means that the socket will reuse everybody else's port
 * without looking at the other's sk_reuse value.
 */

#define SK_NO_REUSE     0
```

```
#define SK_CAN_REUSE    1
#define SK_FORCE_REUSE  2

static inline int sk_peek_offset(struct sock *sk, int flags)
{
        if ((flags & MSG_PEEK) && (sk->sk_peek_off >= 0))
                return sk->sk_peek_off;
        else
                return 0;
}

static inline void sk_peek_offset_bwd(struct sock *sk, int val)
{
        if (sk->sk_peek_off >= 0) {
                if (sk->sk_peek_off >= val)
                        sk->sk_peek_off -= val;
                else
                        sk->sk_peek_off = 0;
        }
}

static inline void sk_peek_offset_fwd(struct sock *sk, int val)
{
        if (sk->sk_peek_off >= 0)
                sk->sk_peek_off += val;
}

/*
 * Hashed lists helper routines
 */
static inline struct sock *sk_entry(const struct hlist_node *node)
{
        return hlist_entry(node, struct sock, sk_node);
}

static inline struct sock *__sk_head(const struct hlist_head *head)
{
        return hlist_entry(head->first, struct sock, sk_node);
}

static inline struct sock *sk_head(const struct hlist_head *head)
{
        return hlist_empty(head) ? NULL : __sk_head(head);
}

static inline struct sock *__sk_nulls_head(const struct hlist_nulls_head *head)
{
        return hlist_nulls_entry(head->first, struct sock, sk_nulls_node);
}

static inline struct sock *sk_nulls_head(const struct hlist_nulls_head *head)
{
        return hlist_nulls_empty(head) ? NULL : __sk_nulls_head(head);
}

static inline struct sock *sk_next(const struct sock *sk)
{
        return sk->sk_node.next ?
                hlist_entry(sk->sk_node.next, struct sock, sk_node) : NULL;
}

static inline struct sock *sk_nulls_next(const struct sock *sk)
{
        return (!is_a_nulls(sk->sk_nulls_node.next)) ?
                hlist_nulls_entry(sk->sk_nulls_node.next,
                                  struct sock, sk_nulls_node) :
                NULL;
}
```

```

static inline bool sk_unhashed(const struct sock *sk)
{
        return hlist_unhashed(&sk->sk_node);
}

static inline bool sk_hashed(const struct sock *sk)
{
        return !sk_unhashed(sk);
}

static inline void sk_node_init(struct hlist_node *node)
{
        node->pprev = NULL;
}

static inline void sk_nulls_node_init(struct hlist_nulls_node *node)
{
        node->pprev = NULL;
}

static inline void __sk_del_node(struct sock *sk)
{
        __hlist_del(&sk->sk_node);
}

/* NB: equivalent to hlist_del_init_rcu */
static inline bool __sk_del_node_init(struct sock *sk)
{
        if (sk_hashed(sk)) {
                __sk_del_node(sk);
                sk_node_init(&sk->sk_node);
                return true;
        }
        return false;
}

/* Grab socket reference count. This operation is valid only
   when sk is ALREADY grabbed f.e. it is found in hash table
   or a list and the lookup is made under lock preventing hash table
   modifications.
 */

static inline void sock_hold(struct sock *sk)
{
        atomic_inc(&sk->sk_refcnt);
}

/* Ungrab socket in the context, which assumes that socket refcnt
   cannot hit zero, f.e. it is true in context of any socketcall.
 */
static inline void __sock_put(struct sock *sk)
{
        atomic_dec(&sk->sk_refcnt);
}

static inline bool sk_del_node_init(struct sock *sk)
{
        bool rc = __sk_del_node_init(sk);

        if (rc) {
                /* paranoid for a while -acme */
                WARN_ON(atomic_read(&sk->sk_refcnt) == 1);
                __sock_put(sk);
        }
        return rc;
}
#define sk_del_node_init_rcu(sk)        sk_del_node_init(sk)
```

```

static inline bool __sk_nulls_del_node_init_rcu(struct sock *sk)
{
        if (sk_hashed(sk)) {
                hlist_nulls_del_init_rcu(&sk->sk_nulls_node);
                return true;
        }
        return false;
}

static inline bool sk_nulls_del_node_init_rcu(struct sock *sk)
{
        bool rc = __sk_nulls_del_node_init_rcu(sk);

        if (rc) {
                /* paranoid for a while -acme */
                WARN_ON(atomic_read(&sk->sk_refcnt) == 1);
                __sock_put(sk);
        }
        return rc;
}

static inline void __sk_add_node(struct sock *sk, struct hlist_head *list)
{
        hlist_add_head(&sk->sk_node, list);
}

static inline void sk_add_node(struct sock *sk, struct hlist_head *list)
{
        sock_hold(sk);
        __sk_add_node(sk, list);
}

static inline void sk_add_node_rcu(struct sock *sk, struct hlist_head *list)
{
        sock_hold(sk);
        hlist_add_head_rcu(&sk->sk_node, list);
}

static inline void __sk_nulls_add_node_rcu(struct sock *sk, struct hlist_nulls_head *list)
{
        hlist_nulls_add_head_rcu(&sk->sk_nulls_node, list);
}

static inline void sk_nulls_add_node_rcu(struct sock *sk, struct hlist_nulls_head *list)
{
        sock_hold(sk);
        __sk_nulls_add_node_rcu(sk, list);
}

static inline void __sk_del_bind_node(struct sock *sk)
{
        __hlist_del(&sk->sk_bind_node);
}

static inline void sk_add_bind_node(struct sock *sk,
                                        struct hlist_head *list)
{
        hlist_add_head(&sk->sk_bind_node, list);
}

#define sk_for_each(__sk, list) \
        hlist_for_each_entry(__sk, list, sk_node)
#define sk_for_each_rcu(__sk, list) \
        hlist_for_each_entry_rcu(__sk, list, sk_node)
#define sk_nulls_for_each(__sk, node, list) \
        hlist_nulls_for_each_entry(__sk, node, list, sk_nulls_node)
#define sk_nulls_for_each_rcu(__sk, node, list) \
        hlist_nulls_for_each_entry_rcu(__sk, node, list, sk_nulls_node)
#define sk_for_each_from(__sk) \
        hlist_for_each_entry_from(__sk, sk_node)
#define sk_nulls_for_each_from(__sk, node) \
        if (__sk && ({ node = &(__sk)->sk_nulls_node; 1; })) \
                hlist_nulls_for_each_entry_from(__sk, node, sk_nulls_node)
#define sk_for_each_safe(__sk, tmp, list) \
        hlist_for_each_entry_safe(__sk, tmp, list, sk_node)
#define sk_for_each_bound(__sk, list) \
        hlist_for_each_entry(__sk, list, sk_bind_node)

/**
 * sk_nulls_for_each_entry_offset - iterate over a list at a given struct offset
 * @tpos:       the type * to use as a loop cursor.
 * @pos:        the &struct hlist_node to use as a loop cursor.
 * @head:       the head for your list.
 * @offset:     offset of hlist_node within the struct.
 *
 */
#define sk_nulls_for_each_entry_offset(tpos, pos, head, offset)                 \
        for (pos = (head)->first;                                               \
             (!is_a_nulls(pos)) &&                                              \
                ({ tpos = (typeof(*tpos) *)((void *)pos - offset); 1;});        \
             pos = pos->next)

static inline struct user_namespace *sk_user_ns(struct sock *sk)
{
        /* Careful only use this in a context where these parameters
         * can not change and must all be valid, such as recvmsg from
         * userspace.
         */
        return sk->sk_socket->file->f_cred->user_ns;
}

/* Sock flags */
enum sock_flags {
        SOCK_DEAD,
        SOCK_DONE,
        SOCK_URGINLINE,
        SOCK_KEEPOPEN,
        SOCK_LINGER,
        SOCK_DESTROY,
        SOCK_BROADCAST,
        SOCK_TIMESTAMP,
        SOCK_ZAPPED,
        SOCK_USE_WRITE_QUEUE, /* whether to call sk->sk_write_space in sock_wfree */
        SOCK_DBG, /* %SO_DEBUG setting */
        SOCK_RCVTSTAMP, /* %SO_TIMESTAMP setting */
        SOCK_RCVTSTAMPNS, /* %SO_TIMESTAMPNS setting */
        SOCK_LOCALROUTE, /* route locally only, %SO_DONTROUTE setting */
        SOCK_QUEUE_SHRUNK, /* write queue has been shrunk recently */
        SOCK_MEMALLOC, /* VM depends on this socket for swapping */
        SOCK_TIMESTAMPING_RX_SOFTWARE,  /* %SOF_TIMESTAMPING_RX_SOFTWARE */
        SOCK_FASYNC, /* fasync() active */
        SOCK_RXQ_OVFL,
        SOCK_ZEROCOPY, /* buffers from userspace */
        SOCK_WIFI_STATUS, /* push wifi status to userspace */
        SOCK_NOFCS, /* Tell NIC not to do the Ethernet FCS.
                     * Will use last 4 bytes of packet sent from
                     * user-space instead.
                     */
        SOCK_FILTER_LOCKED, /* Filter cannot be changed anymore */
        SOCK_SELECT_ERR_QUEUE, /* Wake select on error queue */
};

static inline void sock_copy_flags(struct sock *nsk, struct sock *osk)
{
        nsk->sk_flags = osk->sk_flags;
}

static inline void sock_set_flag(struct sock *sk, enum sock_flags flag)
{
        __set_bit(flag, &sk->sk_flags);
}

static inline void sock_reset_flag(struct sock *sk, enum sock_flags flag)
{
        __clear_bit(flag, &sk->sk_flags);
}

static inline bool sock_flag(const struct sock *sk, enum sock_flags flag)
{
        return test_bit(flag, &sk->sk_flags);
}

#ifdef CONFIG_NET
extern struct static_key memalloc_socks;
static inline int sk_memalloc_socks(void)
{
        return static_key_false(&memalloc_socks);
}
#else

static inline int sk_memalloc_socks(void)
{
        return 0;
}

#endif

static inline gfp_t sk_gfp_atomic(struct sock *sk, gfp_t gfp_mask)
{
        return GFP_ATOMIC | (sk->sk_allocation & __GFP_MEMALLOC);
}

static inline void sk_acceptq_removed(struct sock *sk)
{
        sk->sk_ack_backlog--;
}

static inline void sk_acceptq_added(struct sock *sk)
{
        sk->sk_ack_backlog++;
}

static inline bool sk_acceptq_is_full(const struct sock *sk)
{
        return sk->sk_ack_backlog > sk->sk_max_ack_backlog;
}

/*
 * Compute minimal free write space needed to queue new packets.
 */
static inline int sk_stream_min_wspace(const struct sock *sk)
{
        return sk->sk_wmem_queued >> 1;
}

static inline int sk_stream_wspace(const struct sock *sk)
{
        return sk->sk_sndbuf - sk->sk_wmem_queued;
}

void sk_stream_write_space(struct sock *sk);

/* OOB backlog add */
```

```
static inline void __sk_add_backlog(struct sock *sk, struct sk_buff *skb)
{
        /* dont let skb dst not refcounted, we are going to leave rcu lock */
        skb_dst_force(skb);

        if (!sk->sk_backlog.tail)
                sk->sk_backlog.head = skb;
        else
                sk->sk_backlog.tail->next = skb;

        sk->sk_backlog.tail = skb;
        skb->next = NULL;
}

/*
 * Take into account size of receive queue and backlog queue
 * Do not take into account this skb truesize,
 * to allow even a single big packet to come.
 */
static inline bool sk_rcvqueues_full(const struct sock *sk, unsigned int limit)
{
        unsigned int qsize = sk->sk_backlog.len + atomic_read(&sk->sk_rmem_alloc);

        return qsize > limit;
}

/* The per-socket spinlock must be held here. */
static inline __must_check int sk_add_backlog(struct sock *sk, struct sk_buff *skb,
                                              unsigned int limit)
{
        if (sk_rcvqueues_full(sk, limit))
                return -ENOBUFS;

        __sk_add_backlog(sk, skb);
        sk->sk_backlog.len += skb->truesize;
        return 0;
}

int __sk_backlog_rcv(struct sock *sk, struct sk_buff *skb);

static inline int sk_backlog_rcv(struct sock *sk, struct sk_buff *skb)
{
        if (sk_memalloc_socks() && skb_pfmemalloc(skb))
                return __sk_backlog_rcv(sk, skb);

        return sk->sk_backlog_rcv(sk, skb);
}

static inline void sk_incoming_cpu_update(struct sock *sk)
{
        sk->sk_incoming_cpu = raw_smp_processor_id();
}

static inline void sock_rps_record_flow_hash(__u32 hash)
{
#ifdef CONFIG_RPS
        struct rps_sock_flow_table *sock_flow_table;

        rcu_read_lock();
        sock_flow_table = rcu_dereference(rps_sock_flow_table);
        rps_record_sock_flow(sock_flow_table, hash);
        rcu_read_unlock();
#endif
}

static inline void sock_rps_record_flow(const struct sock *sk)
{
#ifdef CONFIG_RPS
```

```
        sock_rps_record_flow_hash(sk->sk_rxhash);
#endif
}

static inline void sock_rps_save_rxhash(struct sock *sk,
                                        const struct sk_buff *skb)
{
#ifdef CONFIG_RPS
        if (unlikely(sk->sk_rxhash != skb->hash))
                sk->sk_rxhash = skb->hash;
#endif
}

static inline void sock_rps_reset_rxhash(struct sock *sk)
{
#ifdef CONFIG_RPS
        sk->sk_rxhash = 0;
#endif
}

#define sk_wait_event(__sk, __timeo, __condition)                       \
        ({      int __rc;                                               \
                release_sock(__sk);                                     \
                __rc = __condition;                                     \
                if (!__rc) {                                            \
                        *(__timeo) = schedule_timeout(*(__timeo));      \
                }                                                       \
                sched_annotate_sleep();                                         \
                lock_sock(__sk);                                        \
                __rc = __condition;                                     \
                __rc;                                                   \
        })

int sk_stream_wait_connect(struct sock *sk, long *timeo_p);
int sk_stream_wait_memory(struct sock *sk, long *timeo_p);
void sk_stream_wait_close(struct sock *sk, long timeo_p);
int sk_stream_error(struct sock *sk, int flags, int err);
void sk_stream_kill_queues(struct sock *sk);
void sk_set_memalloc(struct sock *sk);
void sk_clear_memalloc(struct sock *sk);

int sk_wait_data(struct sock *sk, long *timeo, const struct sk_buff *skb);

struct request_sock_ops;
struct timewait_sock_ops;
struct inet_hashinfo;
struct raw_hashinfo;
struct module;

/*
 * caches using SLAB_DESTROY_BY_RCU should let .next pointer from nulls nodes
 * un-modified. Special care is taken when initializing object to zero.
 */
static inline void sk_prot_clear_nulls(struct sock *sk, int size)
{
        if (offsetof(struct sock, sk_node.next) != 0)
                memset(sk, 0, offsetof(struct sock, sk_node.next));
        memset(&sk->sk_node.pprev, 0,
               size - offsetof(struct sock, sk_node.pprev));
}

/* Networking protocol blocks we attach to sockets.
 * socket layer -> transport layer interface
 */
struct proto {
        void                    (*close)(struct sock *sk,
                                        long timeout);
        int                     (*connect)(struct sock *sk,
                                        struct sockaddr *uaddr,
                                        int addr_len);
        int                     (*disconnect)(struct sock *sk, int flags);

        struct sock *           (*accept)(struct sock *sk, int flags, int *err);

        int                     (*ioctl)(struct sock *sk, int cmd,
                                         unsigned long arg);
        int                     (*init)(struct sock *sk);
        void                    (*destroy)(struct sock *sk);
        void                    (*shutdown)(struct sock *sk, int how);
        int                     (*setsockopt)(struct sock *sk, int level,
                                        int optname, char __user *optval,
                                        unsigned int optlen);
        int                     (*getsockopt)(struct sock *sk, int level,
                                        int optname, char __user *optval,
                                        int __user *option);
#ifdef CONFIG_COMPAT
        int                     (*compat_setsockopt)(struct sock *sk,
                                        int level,
                                        int optname, char __user *optval,
                                        unsigned int optlen);
        int                     (*compat_getsockopt)(struct sock *sk,
                                        int level,
                                        int optname, char __user *optval,
                                        int __user *option);
        int                     (*compat_ioctl)(struct sock *sk,
                                        unsigned int cmd, unsigned long arg);
#endif
        int                     (*sendmsg)(struct sock *sk, struct msghdr *msg,
                                           size_t len);
        int                     (*recvmsg)(struct sock *sk, struct msghdr *msg,
                                           size_t len, int noblock, int flags,
                                           int *addr_len);
        int                     (*sendpage)(struct sock *sk, struct page *page,
                                        int offset, size_t size, int flags);
        int                     (*bind)(struct sock *sk,
                                        struct sockaddr *uaddr, int addr_len);

        int                     (*backlog_rcv) (struct sock *sk,
                                                struct sk_buff *skb);

        void            (*release_cb)(struct sock *sk);

        /* Keeping track of sk's, looking them up, and port selection methods. */
        void                    (*hash)(struct sock *sk);
        void                    (*unhash)(struct sock *sk);
        void                    (*rehash)(struct sock *sk);
        int                     (*get_port)(struct sock *sk, unsigned short snum);
        void                    (*clear_sk)(struct sock *sk, int size);

        /* Keeping track of sockets in use */
#ifdef CONFIG_PROC_FS
        unsigned int            inuse_idx;
#endif

        bool                    (*stream_memory_free)(const struct sock *sk);
        /* Memory pressure */
        void                    (*enter_memory_pressure)(struct sock *sk);
        atomic_long_t           *memory_allocated;      /* Current allocated memory. */
        struct percpu_counter   *sockets_allocated;     /* Current number of sockets. */
        /*
         * Pressure flag: try to collapse.
         * Technical note: it is used by multiple contexts non atomically.
         * All the __sk_mem_schedule() is of this nature: accounting
         * is strict, actions are advisory and have some latency.
         */
        int                     *memory_pressure;
         long                    *sysctl_mem;
         int                     *sysctl_wmem;
         int                     *sysctl_rmem;
         int                     max_header;
         bool                    no_autobind;

         struct kmem_cache       *slab;
         unsigned int            obj_size;
         int                     slab_flags;

         struct percpu_counter   *orphan_count;

         struct request_sock_ops *rsk_prot;
         struct timewait_sock_ops *twsk_prot;

         union {
                 struct inet_hashinfo    *hashinfo;
                 struct udp_table        *udp_table;
                 struct raw_hashinfo     *raw_hash;
         } h;

         struct module           *owner;

         char                    name[32];

         struct list_head        node;
 #ifdef SOCK_REFCNT_DEBUG
         atomic_t                socks;
 #endif
 #ifdef CONFIG_MEMCG_KMEM
         /*
          * cgroup specific init/deinit functions. Called once for all
          * protocols that implement it, from cgroups populate function.
          * This function has to setup any files the protocol want to
          * appear in the kmem cgroup filesystem.
          */
         int                     (*init_cgroup)(struct mem_cgroup *memcg,
                                                struct cgroup_subsys *ss);
         void                    (*destroy_cgroup)(struct mem_cgroup *memcg);
         struct cg_proto         *(*proto_cgroup)(struct mem_cgroup *memcg);
 #endif
 };

 /*
  * Bits in struct cg_proto.flags
  */
 enum cg_proto_flags {
         /* Currently active and new sockets should be assigned to cgroups */
         MEMCG_SOCK_ACTIVE,
         /* It was ever activated; we must disarm static keys on destruction */
         MEMCG_SOCK_ACTIVATED,
 };

 struct cg_proto {
         struct page_counter     memory_allocated;       /* Current allocated memory. */
         struct percpu_counter   sockets_allocated;      /* Current number of sockets. */
         int                     memory_pressure;
         long                    sysctl_mem[3];
         unsigned long           flags;
         /*
          * memcg field is used to find which memcg we belong directly
          * Each memcg struct can hold more than one cg_proto, so container_of
          * won't really cut.
          *
          * The elegant solution would be having an inverse function to
          * proto_cgroup in struct proto, but that means polluting the structure
          * for everybody, instead of just for memcg users.
          */
         struct mem_cgroup       *memcg;
 };

 int proto_register(struct proto *prot, int alloc_slab);
 void proto_unregister(struct proto *prot);

 static inline bool memcg_proto_active(struct cg_proto *cg_proto)
 {
         return test_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
 }

 #ifdef SOCK_REFCNT_DEBUG
 static inline void sk_refcnt_debug_inc(struct sock *sk)
 {
         atomic_inc(&sk->sk_prot->socks);
 }

 static inline void sk_refcnt_debug_dec(struct sock *sk)
 {
         atomic_dec(&sk->sk_prot->socks);
         printk(KERN_DEBUG "%s socket %p released, %d are still alive\n",
                sk->sk_prot->name, sk, atomic_read(&sk->sk_prot->socks));
 }

 static inline void sk_refcnt_debug_release(const struct sock *sk)
 {
         if (atomic_read(&sk->sk_refcnt) != 1)
                 printk(KERN_DEBUG "Destruction of the %s socket %p delayed, refcnt=%d\n",
                        sk->sk_prot->name, sk, atomic_read(&sk->sk_refcnt));
 }
 #else /* SOCK_REFCNT_DEBUG */
 #define sk_refcnt_debug_inc(sk) do { } while (0)
 #define sk_refcnt_debug_dec(sk) do { } while (0)
 #define sk_refcnt_debug_release(sk) do { } while (0)
 #endif /* SOCK_REFCNT_DEBUG */

 #if defined(CONFIG_MEMCG_KMEM) && defined(CONFIG_NET)
 extern struct static_key memcg_socket_limit_enabled;
 static inline struct cg_proto *parent_cg_proto(struct proto *proto,
                                                struct cg_proto *cg_proto)
 {
         return proto->proto_cgroup(parent_mem_cgroup(cg_proto->memcg));
 }
 #define mem_cgroup_sockets_enabled static_key_false(&memcg_socket_limit_enabled)
 #else
 #define mem_cgroup_sockets_enabled 0
 static inline struct cg_proto *parent_cg_proto(struct proto *proto,
                                                struct cg_proto *cg_proto)
 {
         return NULL;
 }
 #endif

 static inline bool sk_stream_memory_free(const struct sock *sk)
 {
         if (sk->sk_wmem_queued >= sk->sk_sndbuf)
                 return false;

         return sk->sk_prot->stream_memory_free ?
                 sk->sk_prot->stream_memory_free(sk) : true;
 }

 static inline bool sk_stream_is_writeable(const struct sock *sk)
 {
         return sk_stream_wspace(sk) >= sk_stream_min_wspace(sk) &&
                sk_stream_memory_free(sk);
 }

```

```

static inline bool sk_has_memory_pressure(const struct sock *sk)
{
        return sk->sk_prot->memory_pressure != NULL;
}

static inline bool sk_under_memory_pressure(const struct sock *sk)
{
        if (!sk->sk_prot->memory_pressure)
                return false;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
                return !!sk->sk_cgrp->memory_pressure;

        return !!*sk->sk_prot->memory_pressure;
}

static inline void sk_leave_memory_pressure(struct sock *sk)
{
        int *memory_pressure = sk->sk_prot->memory_pressure;

        if (!memory_pressure)
                return;

        if (*memory_pressure)
                *memory_pressure = 0;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
                struct cg_proto *cg_proto = sk->sk_cgrp;
                struct proto *prot = sk->sk_prot;

                for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
                        cg_proto->memory_pressure = 0;
        }

}

static inline void sk_enter_memory_pressure(struct sock *sk)
{
        if (!sk->sk_prot->enter_memory_pressure)
                return;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
                struct cg_proto *cg_proto = sk->sk_cgrp;
                struct proto *prot = sk->sk_prot;

                for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
                        cg_proto->memory_pressure = 1;
        }

        sk->sk_prot->enter_memory_pressure(sk);
}

static inline long sk_prot_mem_limits(const struct sock *sk, int index)
{
        long *prot = sk->sk_prot->sysctl_mem;
        if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
                prot = sk->sk_cgrp->sysctl_mem;
        return prot[index];
}

static inline void memcg_memory_allocated_add(struct cg_proto *prot,
                                              unsigned long amt,
                                              int *parent_status)
{
        page_counter_charge(&prot->memory_allocated, amt);

        if (page_counter_read(&prot->memory_allocated) >
            prot->memory_allocated.limit)
                *parent_status = OVER_LIMIT;
}

static inline void memcg_memory_allocated_sub(struct cg_proto *prot,
                                              unsigned long amt)
{
        page_counter_uncharge(&prot->memory_allocated, amt);
}

static inline long
sk_memory_allocated(const struct sock *sk)
{
        struct proto *prot = sk->sk_prot;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
                return page_counter_read(&sk->sk_cgrp->memory_allocated);

        return atomic_long_read(prot->memory_allocated);
}

static inline long
sk_memory_allocated_add(struct sock *sk, int amt, int *parent_status)
{
        struct proto *prot = sk->sk_prot;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
                memcg_memory_allocated_add(sk->sk_cgrp, amt, parent_status);
                /* update the root cgroup regardless */
                atomic_long_add_return(amt, prot->memory_allocated);
                return page_counter_read(&sk->sk_cgrp->memory_allocated);
        }

        return atomic_long_add_return(amt, prot->memory_allocated);
}

static inline void
sk_memory_allocated_sub(struct sock *sk, int amt)
{
        struct proto *prot = sk->sk_prot;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
                memcg_memory_allocated_sub(sk->sk_cgrp, amt);

        atomic_long_sub(amt, prot->memory_allocated);
}

static inline void sk_sockets_allocated_dec(struct sock *sk)
{
        struct proto *prot = sk->sk_prot;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
                struct cg_proto *cg_proto = sk->sk_cgrp;

                for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
                        percpu_counter_dec(&cg_proto->sockets_allocated);
        }

        percpu_counter_dec(prot->sockets_allocated);
}

static inline void sk_sockets_allocated_inc(struct sock *sk)
{
        struct proto *prot = sk->sk_prot;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
                struct cg_proto *cg_proto = sk->sk_cgrp;

                for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
                        percpu_counter_inc(&cg_proto->sockets_allocated);
        }

        percpu_counter_inc(prot->sockets_allocated);
}

static inline int
sk_sockets_allocated_read_positive(struct sock *sk)
{
        struct proto *prot = sk->sk_prot;

        if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
                return percpu_counter_read_positive(&sk->sk_cgrp->sockets_allocated);

        return percpu_counter_read_positive(prot->sockets_allocated);
}

static inline int
proto_sockets_allocated_sum_positive(struct proto *prot)
{
        return percpu_counter_sum_positive(prot->sockets_allocated);
}

static inline long
proto_memory_allocated(struct proto *prot)
{
        return atomic_long_read(prot->memory_allocated);
}

static inline bool
proto_memory_pressure(struct proto *prot)
{
        if (!prot->memory_pressure)
                return false;
        return !!*prot->memory_pressure;
}


#ifdef CONFIG_PROC_FS
/* Called with local bh disabled */
void sock_prot_inuse_add(struct net *net, struct proto *prot, int inc);
int sock_prot_inuse_get(struct net *net, struct proto *proto);
#else
static inline void sock_prot_inuse_add(struct net *net, struct proto *prot,
                int inc)
{
}
#endif


/* With per-bucket locks this operation is not-atomic, so that
 * this version is not worse.
 */
static inline void __sk_prot_rehash(struct sock *sk)
{
        sk->sk_prot->unhash(sk);
        sk->sk_prot->hash(sk);
}

void sk_prot_clear_portaddr_nulls(struct sock *sk, int size);

/* About 10 seconds */
#define SOCK_DESTROY_TIME (10*HZ)

/* Sockets 0-1023 can't be bound to unless you are superuser */
#define PROT_SOCK       1024

```

```
#define SHUTDOWN_MASK   3
#define RCV_SHUTDOWN    1
#define SEND_SHUTDOWN   2

#define SOCK_SNDBUF_LOCK        1
#define SOCK_RCVBUF_LOCK        2
#define SOCK_BINDADDR_LOCK      4
#define SOCK_BINDPORT_LOCK      8

struct socket_alloc {
        struct socket socket;
        struct inode vfs_inode;
};

static inline struct socket *SOCKET_I(struct inode *inode)
{
        return &container_of(inode, struct socket_alloc, vfs_inode)->socket;
}

static inline struct inode *SOCK_INODE(struct socket *socket)
{
        return &container_of(socket, struct socket_alloc, socket)->vfs_inode;
}

/*
 * Functions for memory accounting
 */
int __sk_mem_schedule(struct sock *sk, int size, int kind);
void __sk_mem_reclaim(struct sock *sk, int amount);

#define SK_MEM_QUANTUM ((int)PAGE_SIZE)
#define SK_MEM_QUANTUM_SHIFT ilog2(SK_MEM_QUANTUM)
#define SK_MEM_SEND     0
#define SK_MEM_RECV     1

static inline int sk_mem_pages(int amt)
{
        return (amt + SK_MEM_QUANTUM - 1) >> SK_MEM_QUANTUM_SHIFT;
}

static inline bool sk_has_account(struct sock *sk)
{
        /* return true if protocol supports memory accounting */
        return !!sk->sk_prot->memory_allocated;
}

static inline bool sk_wmem_schedule(struct sock *sk, int size)
{
        if (!sk_has_account(sk))
                return true;
        return size <= sk->sk_forward_alloc ||
                __sk_mem_schedule(sk, size, SK_MEM_SEND);
}

static inline bool
sk_rmem_schedule(struct sock *sk, struct sk_buff *skb, int size)
{
        if (!sk_has_account(sk))
                return true;
        return size<= sk->sk_forward_alloc ||
                __sk_mem_schedule(sk, size, SK_MEM_RECV) ||
                skb_pfmemalloc(skb);
}

static inline void sk_mem_reclaim(struct sock *sk)
{
        if (!sk_has_account(sk))
                return;
        if (sk->sk_forward_alloc >= SK_MEM_QUANTUM)
                __sk_mem_reclaim(sk, sk->sk_forward_alloc);
}

static inline void sk_mem_reclaim_partial(struct sock *sk)
{
        if (!sk_has_account(sk))
                return;
        if (sk->sk_forward_alloc > SK_MEM_QUANTUM)
                __sk_mem_reclaim(sk, sk->sk_forward_alloc - 1);
}

static inline void sk_mem_charge(struct sock *sk, int size)
{
        if (!sk_has_account(sk))
                return;
        sk->sk_forward_alloc -= size;
}

static inline void sk_mem_uncharge(struct sock *sk, int size)
{
        if (!sk_has_account(sk))
                return;
        sk->sk_forward_alloc += size;
}

static inline void sk_wmem_free_skb(struct sock *sk, struct sk_buff *skb)
{
        sock_set_flag(sk, SOCK_QUEUE_SHRUNK);
        sk->sk_wmem_queued -= skb->truesize;
        sk_mem_uncharge(sk, skb->truesize);
        __kfree_skb(skb);
}

/* Used by processes to "lock" a socket state, so that
 * interrupts and bottom half handlers won't change it
 * from under us. It essentially blocks any incoming
 * packets, so that we won't get any new data or any
 * packets that change the state of the socket.
 *
 * While locked, BH processing will add new packets to
 * the backlog queue.  This queue is processed by the
 * owner of the socket lock right before it is released.
 *
 * Since ~2.3.5 it is also exclusive sleep lock serializing
 * accesses from user process context.
 */
#define sock_owned_by_user(sk)  ((sk)->sk_lock.owned)

static inline void sock_release_ownership(struct sock *sk)
{
        sk->sk_lock.owned = 0;
}

/*
 * Macro so as to not evaluate some arguments when
 * lockdep is not enabled.
 *
 * Mark both the sk_lock and the sk_lock.slock as a
 * per-address-family lock class.
 */
#define sock_lock_init_class_and_name(sk, sname, skey, name, key)        \
do {                                                                    \
        sk->sk_lock.owned = 0;                                          \
        init_waitqueue_head(&sk->sk_lock.wq);                           \
        spin_lock_init(&(sk)->sk_lock.slock);                           \
        debug_check_no_locks_freed((void *)&(sk)->sk_lock,              \
                        sizeof((sk)->sk_lock));                         \
        lockdep_set_class_and_name(&(sk)->sk_lock.slock,                \
                                (skey), (sname));                               \
        lockdep_init_map(&(sk)->sk_lock.dep_map, (name), (key), 0);     \
} while (0)

void lock_sock_nested(struct sock *sk, int subclass);

static inline void lock_sock(struct sock *sk)
{
        lock_sock_nested(sk, 0);
}

void release_sock(struct sock *sk);

/* BH context may only use the following locking interface. */
#define bh_lock_sock(__sk)      spin_lock(&((__sk)->sk_lock.slock))
#define bh_lock_sock_nested(__sk) \
                                spin_lock_nested(&((__sk)->sk_lock.slock), \
                                SINGLE_DEPTH_NESTING)
#define bh_unlock_sock(__sk)    spin_unlock(&((__sk)->sk_lock.slock))

bool lock_sock_fast(struct sock *sk);
/**
 * unlock_sock_fast - complement of lock_sock_fast
 * @sk: socket
 * @slow: slow mode
 *
 * fast unlock socket for user context.
 * If slow mode is on, we call regular release_sock()
 */
static inline void unlock_sock_fast(struct sock *sk, bool slow)
{
        if (slow)
                release_sock(sk);
        else
                spin_unlock_bh(&sk->sk_lock.slock);
}


struct sock *sk_alloc(struct net *net, int family, gfp_t priority,
                      struct proto *prot, int kern);
void sk_free(struct sock *sk);
void sk_destruct(struct sock *sk);
struct sock *sk_clone_lock(const struct sock *sk, const gfp_t priority);

struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force,
                             gfp_t priority);
void sock_wfree(struct sk_buff *skb);
void skb_orphan_partial(struct sk_buff *skb);
void sock_rfree(struct sk_buff *skb);
void sock_efree(struct sk_buff *skb);
#ifdef CONFIG_INET
void sock_edemux(struct sk_buff *skb);
#else
#define sock_edemux(skb) sock_efree(skb)
#endif

int sock_setsockopt(struct socket *sock, int level, int op,
                    char __user *optval, unsigned int optlen);

int sock_getsockopt(struct socket *sock, int level, int op,
                    char __user *optval, int __user *optlen);
struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size,
                                    int noblock, int *errcode);
struct sk_buff *sock_alloc_send_pskb(struct sock *sk, unsigned long header_len,
                                     unsigned long data_len, int noblock,
                                     int *errcode, int max_page_order);
void *sock_kmalloc(struct sock *sk, int size, gfp_t priority);
void sock_kfree_s(struct sock *sk, void *mem, int size);
void sock_kzfree_s(struct sock *sk, void *mem, int size);
void sk_send_sigurg(struct sock *sk);

/*
 * Functions to fill in entries in struct proto_ops when a protocol
 * does not implement a particular function.
 */
int sock_no_bind(struct socket *, struct sockaddr *, int);
int sock_no_connect(struct socket *, struct sockaddr *, int, int);
int sock_no_socketpair(struct socket *, struct socket *);
int sock_no_accept(struct socket *, struct socket *, int);
int sock_no_getname(struct socket *, struct sockaddr *, int *, int);
unsigned int sock_no_poll(struct file *, struct socket *,
                          struct poll_table_struct *);
int sock_no_ioctl(struct socket *, unsigned int, unsigned long);
int sock_no_listen(struct socket *, int);
int sock_no_shutdown(struct socket *, int);
int sock_no_getsockopt(struct socket *, int , int, char __user *, int __user *);
int sock_no_setsockopt(struct socket *, int, int, char __user *, unsigned int);
int sock_no_sendmsg(struct socket *, struct msghdr *, size_t);
int sock_no_recvmsg(struct socket *, struct msghdr *, size_t, int);
int sock_no_mmap(struct file *file, struct socket *sock,
                 struct vm_area_struct *vma);
ssize_t sock_no_sendpage(struct socket *sock, struct page *page, int offset,
                         size_t size, int flags);

/*
 * Functions to fill in entries in struct proto_ops when a protocol
 * uses the inet style.
 */
int sock_common_getsockopt(struct socket *sock, int level, int optname,
                                  char __user *optval, int __user *optlen);
int sock_common_recvmsg(struct socket *sock, struct msghdr *msg, size_t size,
                        int flags);
int sock_common_setsockopt(struct socket *sock, int level, int optname,
                                  char __user *optval, unsigned int optlen);
int compat_sock_common_getsockopt(struct socket *sock, int level,
                int optname, char __user *optval, int __user *optlen);
int compat_sock_common_setsockopt(struct socket *sock, int level,
                int optname, char __user *optval, unsigned int optlen);

void sk_common_release(struct sock *sk);

/*
 *      Default socket callbacks and setup code
 */

/* Initialise core socket variables */
void sock_init_data(struct socket *sock, struct sock *sk);

/*
 * Socket reference counting postulates.
 *
 * * Each user of socket SHOULD hold a reference count.
 * * Each access point to socket (an hash table bucket, reference from a list,
 *   running timer, skb in flight MUST hold a reference count.
 * * When reference count hits 0, it means it will never increase back.
 * * When reference count hits 0, it means that no references from
 *   outside exist to this socket and current process on current CPU
 *   is last user and may/should destroy this socket.
 * * sk_free is called from any context: process, BH, IRQ. When
 *   it is called, socket has no references from outside -> sk_free
 *   may release descendant resources allocated by the socket, but
 *   to the time when it is called, socket is NOT referenced by any
 *   hash tables, lists etc.
 * * Packets, delivered from outside (from network or from another process)
 *   and enqueued on receive/error queues SHOULD NOT grab reference count,
 *   when they sit in queue. Otherwise, packets will leak to hole, when
 *   socket is looked up by one cpu and unhasing is made by another CPU.
 *   It is true for udp/raw, netlink (leak to receive and error queues), tcp
 *   (leak to backlog). Packet socket does all the processing inside
 *   BR_NETPROTO_LOCK, so that it has not this race condition. UNIX sockets
 *   use separate SMP lock, so that they are prone too.
 */

/* Ungrab socket and destroy it, if it was the last reference. */
static inline void sock_put(struct sock *sk)
{
        if (atomic_dec_and_test(&sk->sk_refcnt))
                sk_free(sk);
}
/* Generic version of sock_put(), dealing with all sockets
 * (TCP_TIMEWAIT, TCP_NEW_SYN_RECV, ESTABLISHED...)
 */
void sock_gen_put(struct sock *sk);

int sk_receive_skb(struct sock *sk, struct sk_buff *skb, const int nested);

static inline void sk_tx_queue_set(struct sock *sk, int tx_queue)
{
        sk->sk_tx_queue_mapping = tx_queue;
}

static inline void sk_tx_queue_clear(struct sock *sk)
{
        sk->sk_tx_queue_mapping = -1;
}

static inline int sk_tx_queue_get(const struct sock *sk)
{
        return sk ? sk->sk_tx_queue_mapping : -1;
}

static inline void sk_set_socket(struct sock *sk, struct socket *sock)
{
        sk_tx_queue_clear(sk);
        sk->sk_socket = sock;
}

static inline wait_queue_head_t *sk_sleep(struct sock *sk)
{
        BUILD_BUG_ON(offsetof(struct socket_wq, wait) != 0);
        return &rcu_dereference_raw(sk->sk_wq)->wait;
}
/* Detach socket from process context.
 * Announce socket dead, detach it from wait queue and inode.
 * Note that parent inode held reference count on this struct sock,
 * we do not release it in this function, because protocol
 * probably wants some additional cleanups or even continuing
 * to work with this socket (TCP).
 */
static inline void sock_orphan(struct sock *sk)
{
        write_lock_bh(&sk->sk_callback_lock);
        sock_set_flag(sk, SOCK_DEAD);
        sk_set_socket(sk, NULL);
        sk->sk_wq  = NULL;
        write_unlock_bh(&sk->sk_callback_lock);
}

static inline void sock_graft(struct sock *sk, struct socket *parent)
{
        write_lock_bh(&sk->sk_callback_lock);
        sk->sk_wq = parent->wq;
        parent->sk = sk;
        sk_set_socket(sk, parent);
        security_sock_graft(sk, parent);
        write_unlock_bh(&sk->sk_callback_lock);
}

kuid_t sock_i_uid(struct sock *sk);
unsigned long sock_i_ino(struct sock *sk);

static inline struct dst_entry *
__sk_dst_get(struct sock *sk)
{
        return rcu_dereference_check(sk->sk_dst_cache, sock_owned_by_user(sk) ||
                                                       lockdep_is_held(&sk->sk_lock.slock));
}

static inline struct dst_entry *
sk_dst_get(struct sock *sk)
{
        struct dst_entry *dst;

        rcu_read_lock();
        dst = rcu_dereference(sk->sk_dst_cache);
        if (dst && !atomic_inc_not_zero(&dst->__refcnt))
                dst = NULL;
        rcu_read_unlock();
        return dst;
}

static inline void dst_negative_advice(struct sock *sk)
{
        struct dst_entry *ndst, *dst = __sk_dst_get(sk);

        if (dst && dst->ops->negative_advice) {
                ndst = dst->ops->negative_advice(dst);

                if (ndst != dst) {
                        rcu_assign_pointer(sk->sk_dst_cache, ndst);
                        sk_tx_queue_clear(sk);
                }
        }
}

static inline void
__sk_dst_set(struct sock *sk, struct dst_entry *dst)
{
        struct dst_entry *old_dst;

        sk_tx_queue_clear(sk);
        /*
         * This can be called while sk is owned by the caller only,
         * with no state that can be checked in a rcu_dereference_check() cond
         */
        old_dst = rcu_dereference_raw(sk->sk_dst_cache);
        rcu_assign_pointer(sk->sk_dst_cache, dst);
        dst_release(old_dst);
}

static inline void
sk_dst_set(struct sock *sk, struct dst_entry *dst)
{
        struct dst_entry *old_dst;

        sk_tx_queue_clear(sk);
        old_dst = xchg((__force struct dst_entry **)&sk->sk_dst_cache, dst);
        dst_release(old_dst);
}

static inline void
```

```
__sk_dst_reset(struct sock *sk)
{
        __sk_dst_set(sk, NULL);
}

static inline void
sk_dst_reset(struct sock *sk)
{
        sk_dst_set(sk, NULL);
}

struct dst_entry *__sk_dst_check(struct sock *sk, u32 cookie);

struct dst_entry *sk_dst_check(struct sock *sk, u32 cookie);

bool sk_mc_loop(struct sock *sk);

static inline bool sk_can_gso(const struct sock *sk)
{
        return net_gso_ok(sk->sk_route_caps, sk->sk_gso_type);
}

void sk_setup_caps(struct sock *sk, struct dst_entry *dst);

static inline void sk_nocaps_add(struct sock *sk, netdev_features_t flags)
{
        sk->sk_route_nocaps |= flags;
        sk->sk_route_caps &= ~flags;
}

static inline int skb_do_copy_data_nocache(struct sock *sk, struct sk_buff *skb,
                                           struct iov_iter *from, char *to,
                                           int copy, int offset)
{
        if (skb->ip_summed == CHECKSUM_NONE) {
                __wsum csum = 0;
                if (csum_and_copy_from_iter(to, copy, &csum, from) != copy)
                        return -EFAULT;
                skb->csum = csum_block_add(skb->csum, csum, offset);
        } else if (sk->sk_route_caps & NETIF_F_NOCACHE_COPY) {
                if (copy_from_iter_nocache(to, copy, from) != copy)
                        return -EFAULT;
        } else if (copy_from_iter(to, copy, from) != copy)
                return -EFAULT;

        return 0;
}

static inline int skb_add_data_nocache(struct sock *sk, struct sk_buff *skb,
                                       struct iov_iter *from, int copy)
{
        int err, offset = skb->len;

        err = skb_do_copy_data_nocache(sk, skb, from, skb_put(skb, copy),
                                       copy, offset);
        if (err)
                __skb_trim(skb, offset);

        return err;
}

static inline int skb_copy_to_page_nocache(struct sock *sk, struct iov_iter *from,
                                           struct sk_buff *skb,
                                           struct page *page,
                                           int off, int copy)
{
        int err;

        err = skb_do_copy_data_nocache(sk, skb, from, page_address(page) + off,
                                       copy, skb->len);
        if (err)
                return err;

        skb->len             += copy;
        skb->data_len        += copy;
        skb->truesize        += copy;
        sk->sk_wmem_queued   += copy;
        sk_mem_charge(sk, copy);
        return 0;
}

/**
 * sk_wmem_alloc_get - returns write allocations
 * @sk: socket
 *
 * Returns sk_wmem_alloc minus initial offset of one
 */
static inline int sk_wmem_alloc_get(const struct sock *sk)
{
        return atomic_read(&sk->sk_wmem_alloc) - 1;
}

/**
 * sk_rmem_alloc_get - returns read allocations
 * @sk: socket
 *
 * Returns sk_rmem_alloc
 */
static inline int sk_rmem_alloc_get(const struct sock *sk)
{
        return atomic_read(&sk->sk_rmem_alloc);
}

/**
 * sk_has_allocations - check if allocations are outstanding
 * @sk: socket
 *
 * Returns true if socket has write or read allocations
 */
static inline bool sk_has_allocations(const struct sock *sk)
{
        return sk_wmem_alloc_get(sk) || sk_rmem_alloc_get(sk);
}

/**
 * wq_has_sleeper - check if there are any waiting processes
 * @wq: struct socket_wq
 *
 * Returns true if socket_wq has waiting processes
 *
 * The purpose of the wq_has_sleeper and sock_poll_wait is to wrap the memory
 * barrier call. They were added due to the race found within the tcp code.
 *
 * Consider following tcp code paths:
 *
 * CPU1                  CPU2
 *
 * sys_select            receive packet
 *   ...                 ...
 *   __add_wait_queue    update tp->rcv_nxt
 *   ...                 ...
 *   tp->rcv_nxt check   sock_def_readable
 *   ...                 {
 *   schedule               rcu_read_lock();
 *                          wq = rcu_dereference(sk->sk_wq);
 *                          if (wq && waitqueue_active(&wq->wait))
 *                               wake_up_interruptible(&wq->wait)
 *                           ...
 *                        }
 *
 * The race for tcp fires when the __add_wait_queue changes done by CPU1 stay
 * in its cache, and so does the tp->rcv_nxt update on CPU2 side.  The CPU1
 * could then endup calling schedule and sleep forever if there are no more
 * data on the socket.
 *
 */
static inline bool wq_has_sleeper(struct socket_wq *wq)
{
        /* We need to be sure we are in sync with the
         * add_wait_queue modifications to the wait queue.
         *
         * This memory barrier is paired in the sock_poll_wait.
         */
        smp_mb();
        return wq && waitqueue_active(&wq->wait);
}

/**
 * sock_poll_wait - place memory barrier behind the poll_wait call.
 * @filp:           file
 * @wait_address:   socket wait queue
 * @p:              poll_table
 *
 * See the comments in the wq_has_sleeper function.
 */
static inline void sock_poll_wait(struct file *filp,
                wait_queue_head_t *wait_address, poll_table *p)
{
        if (!poll_does_not_wait(p) && wait_address) {
                poll_wait(filp, wait_address, p);
                /* We need to be sure we are in sync with the
                 * socket flags modification.
                 *
                 * This memory barrier is paired in the wq_has_sleeper.
                 */
                smp_mb();
        }
}

static inline void skb_set_hash_from_sk(struct sk_buff *skb, struct sock *sk)
{
        if (sk->sk_txhash) {
                skb->l4_hash = 1;
                skb->hash = sk->sk_txhash;
        }
}

/*
 *      Queue a received datagram if it will fit. Stream and sequenced
 *      protocols can't normally use this as they need to fit buffers in
 *      and play with them.
 *
 *      Inlined as it's very short and called for pretty much every
 *      packet ever received.
 */

static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
{
        skb_orphan(skb);
        skb->sk = sk;
        skb->destructor = sock_wfree;
        skb_set_hash_from_sk(skb, sk);
        /*
         * We used to take a refcount on sk, but following operation
```

```c
         * is enough to guarantee sk_free() wont free this sock until
         * all in-flight packets are completed
         */
        atomic_add(skb->truesize, &sk->sk_wmem_alloc);
}

static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
{
        skb_orphan(skb);
        skb->sk = sk;
        skb->destructor = sock_rfree;
        atomic_add(skb->truesize, &sk->sk_rmem_alloc);
        sk_mem_charge(sk, skb->truesize);
}

void sk_reset_timer(struct sock *sk, struct timer_list *timer,
                    unsigned long expires);

void sk_stop_timer(struct sock *sk, struct timer_list *timer);

int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb);

int sock_queue_err_skb(struct sock *sk, struct sk_buff *skb);
struct sk_buff *sock_dequeue_err_skb(struct sock *sk);

/*
 *      Recover an error report and clear atomically
 */

static inline int sock_error(struct sock *sk)
{
        int err;
        if (likely(!sk->sk_err))
                return 0;
        err = xchg(&sk->sk_err, 0);
        return -err;
}

static inline unsigned long sock_wspace(struct sock *sk)
{
        int amt = 0;

        if (!(sk->sk_shutdown & SEND_SHUTDOWN)) {
                amt = sk->sk_sndbuf - atomic_read(&sk->sk_wmem_alloc);
                if (amt < 0)
                        amt = 0;
        }
        return amt;
}

static inline void sk_wake_async(struct sock *sk, int how, int band)
{
        if (sock_flag(sk, SOCK_FASYNC))
                sock_wake_async(sk->sk_socket, how, band);
}

/* Since sk_{r,w}mem_alloc sums skb->truesize, even a small frame might
 * need sizeof(sk_buff) + MTU + padding, unless net driver perform copybreak.
 * Note: for send buffers, TCP works better if we can build two skbs at
 * minimum.
 */
#define TCP_SKB_MIN_TRUESIZE    (2048 + SKB_DATA_ALIGN(sizeof(struct sk_buff)))

#define SOCK_MIN_SNDBUF         (TCP_SKB_MIN_TRUESIZE * 2)
#define SOCK_MIN_RCVBUF          TCP_SKB_MIN_TRUESIZE

static inline void sk_stream_moderate_sndbuf(struct sock *sk)
{
        if (!(sk->sk_userlocks & SOCK_SNDBUF_LOCK)) {
                sk->sk_sndbuf = min(sk->sk_sndbuf, sk->sk_wmem_queued >> 1);
                sk->sk_sndbuf = max_t(u32, sk->sk_sndbuf, SOCK_MIN_SNDBUF);
        }
}

struct sk_buff *sk_stream_alloc_skb(struct sock *sk, int size, gfp_t gfp,
                                    bool force_schedule);

/**
 * sk_page_frag - return an appropriate page_frag
 * @sk: socket
 *
 * If socket allocation mode allows current thread to sleep, it means its
 * safe to use the per task page_frag instead of the per socket one.
 */
static inline struct page_frag *sk_page_frag(struct sock *sk)
{
        if (sk->sk_allocation & __GFP_WAIT)
                return &current->task_frag;

        return &sk->sk_frag;
}

bool sk_page_frag_refill(struct sock *sk, struct page_frag *pfrag);

/*
 *      Default write policy as shown to user space via poll/select/SIGIO
 */
static inline bool sock_writeable(const struct sock *sk)
{
        return atomic_read(&sk->sk_wmem_alloc) < (sk->sk_sndbuf >> 1);
}

static inline gfp_t gfp_any(void)
{
        return in_softirq() ? GFP_ATOMIC : GFP_KERNEL;
}

static inline long sock_rcvtimeo(const struct sock *sk, bool noblock)
{
        return noblock ? 0 : sk->sk_rcvtimeo;
}

static inline long sock_sndtimeo(const struct sock *sk, bool noblock)
{
        return noblock ? 0 : sk->sk_sndtimeo;
}

static inline int sock_rcvlowat(const struct sock *sk, int waitall, int len)
{
        return (waitall ? len : min_t(int, sk->sk_rcvlowat, len)) ? : 1;
}

/* Alas, with timeout socket operations are not restartable.
 * Compare this to poll().
 */
static inline int sock_intr_errno(long timeo)
{
        return timeo == MAX_SCHEDULE_TIMEOUT ? -ERESTARTSYS : -EINTR;
}

struct sock_skb_cb {
        u32 dropcount;
};

/* Store sock_skb_cb at the end of skb->cb[] so protocol families
 * using skb->cb[] would keep using it directly and utilize its
 * alignement guarantee.
 */
#define SOCK_SKB_CB_OFFSET ((FIELD_SIZEOF(struct sk_buff, cb) - \
                            sizeof(struct sock_skb_cb)))

#define SOCK_SKB_CB(__skb) ((struct sock_skb_cb *)((__skb)->cb + \
                            SOCK_SKB_CB_OFFSET))

#define sock_skb_cb_check_size(size) \
        BUILD_BUG_ON((size) > SOCK_SKB_CB_OFFSET)

static inline void
sock_skb_set_dropcount(const struct sock *sk, struct sk_buff *skb)
{
        SOCK_SKB_CB(skb)->dropcount = atomic_read(&sk->sk_drops);
}

void __sock_recv_timestamp(struct msghdr *msg, struct sock *sk,
                           struct sk_buff *skb);
void __sock_recv_wifi_status(struct msghdr *msg, struct sock *sk,
                             struct sk_buff *skb);

static inline void
sock_recv_timestamp(struct msghdr *msg, struct sock *sk, struct sk_buff *skb)
{
        ktime_t kt = skb->tstamp;
        struct skb_shared_hwtstamps *hwtstamps = skb_hwtstamps(skb);

        /*
         * generate control messages if
         * - receive time stamping in software requested
         * - software time stamp available and wanted
         * - hardware time stamps available and wanted
         */
        if (sock_flag(sk, SOCK_RCVTSTAMP) ||
            (sk->sk_tsflags & SOF_TIMESTAMPING_RX_SOFTWARE) ||
            (kt.tv64 && sk->sk_tsflags & SOF_TIMESTAMPING_SOFTWARE) ||
            (hwtstamps->hwtstamp.tv64 &&
             (sk->sk_tsflags & SOF_TIMESTAMPING_RAW_HARDWARE)))
                __sock_recv_timestamp(msg, sk, skb);
        else
                sk->sk_stamp = kt;

        if (sock_flag(sk, SOCK_WIFI_STATUS) && skb->wifi_acked_valid)
                __sock_recv_wifi_status(msg, sk, skb);
}

void __sock_recv_ts_and_drops(struct msghdr *msg, struct sock *sk,
                              struct sk_buff *skb);

static inline void sock_recv_ts_and_drops(struct msghdr *msg, struct sock *sk,
                                          struct sk_buff *skb)
{
#define FLAGS_TS_OR_DROPS ((1UL << SOCK_RXQ_OVFL)                       | \
                           (1UL << SOCK_RCVTSTAMP))
#define TSFLAGS_ANY       (SOF_TIMESTAMPING_SOFTWARE                    | \
                           SOF_TIMESTAMPING_RAW_HARDWARE)

        if (sk->sk_flags & FLAGS_TS_OR_DROPS || sk->sk_tsflags & TSFLAGS_ANY)
                __sock_recv_ts_and_drops(msg, sk, skb);
        else
                sk->sk_stamp = skb->tstamp;
}

void __sock_tx_timestamp(const struct sock *sk, __u8 *tx_flags);

/**
 * sock_tx_timestamp - checks whether the outgoing packet is to be time stamped
 * @sk:         socket sending this packet
 * @tx_flags:   completed with instructions for time stamping
 *
 * Note : callers should take care of initial *tx_flags value (usually 0)
 */
static inline void sock_tx_timestamp(const struct sock *sk, __u8 *tx_flags)
{
        if (unlikely(sk->sk_tsflags))
                __sock_tx_timestamp(sk, tx_flags);
        if (unlikely(sock_flag(sk, SOCK_WIFI_STATUS)))
                *tx_flags |= SKBTX_WIFI_STATUS;
}

/**
 * sk_eat_skb - Release a skb if it is no longer needed
 * @sk: socket to eat this skb from
 * @skb: socket buffer to eat
 *
 * This routine must be called with interrupts disabled or with the socket
 * locked so that the sk_buff queue operation is ok.
*/
static inline void sk_eat_skb(struct sock *sk, struct sk_buff *skb)
{
        __skb_unlink(skb, &sk->sk_receive_queue);
        __kfree_skb(skb);
}

static inline
struct net *sock_net(const struct sock *sk)
{
        return read_pnet(&sk->sk_net);
}

static inline
void sock_net_set(struct sock *sk, struct net *net)
{
        write_pnet(&sk->sk_net, net);
}

static inline struct sock *skb_steal_sock(struct sk_buff *skb)
{
        if (skb->sk) {
                struct sock *sk = skb->sk;

                skb->destructor = NULL;
                skb->sk = NULL;
                return sk;
        }
        return NULL;
}

/* This helper checks if a socket is a full socket,
 * ie _not_ a timewait or request socket.
 */
static inline bool sk_fullsock(const struct sock *sk)
{
        return (1 << sk->sk_state) & ~(TCPF_TIME_WAIT | TCPF_NEW_SYN_RECV);
}

void sock_enable_timestamp(struct sock *sk, int flag);
int sock_get_timestamp(struct sock *, struct timeval __user *);
int sock_get_timestampns(struct sock *, struct timespec __user *);
int sock_recv_errqueue(struct sock *sk, struct msghdr *msg, int len, int level,
                       int type);

bool sk_ns_capable(const struct sock *sk,
                   struct user_namespace *user_ns, int cap);
bool sk_capable(const struct sock *sk, int cap);
bool sk_net_capable(const struct sock *sk, int cap);

extern __u32 sysctl_wmem_max;
extern __u32 sysctl_rmem_max;

extern int sysctl_tstamp_allow_data;
extern int sysctl_optmem_max;

extern __u32 sysctl_wmem_default;
extern __u32 sysctl_rmem_default;

#endif  /* _SOCK_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