NAME

ifnet, **ifaddr**, **ifqueue**, **if_data** - kernel interfaces for manipulating network interfaces

SYNOPSIS

```
#include <sys/param.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <net/if.h>
#include <net/if_var.h>
#include <net/if_types.h>
```

Interface Manipulation Functions

*struct ifnet **
if_alloc(*u_char type*);

*struct ifnet **
if_alloc_dev(*u_char type*, *device_t dev*);

*struct ifnet **
if_alloc_domain(*u_char type*, *int numa_domain*);

void
if_attach(*struct ifnet *ifp*);

void
if_detach(*struct ifnet *ifp*);

void
if_free(*struct ifnet *ifp*);

void
if_free_type(*struct ifnet *ifp*, *u_char type*);

void
if_down(*struct ifnet *ifp*);

int
ifioctl(*struct socket *so*, *u_long cmd*, *caddr_t data*, *struct thread *td*);

int

ifpromisc(*struct ifnet *ifp, int pswitch*);

int
if_allmulti(*struct ifnet *ifp, int amswitch*);

*struct ifnet **
ifunit(*const char *name*);

*struct ifnet **
ifunit_ref(*const char *name*);

void
if_up(*struct ifnet *ifp*);

Interface Address Functions

*struct ifaddr **
ifa_ifwithaddr(*struct sockaddr *addr*);

*struct ifaddr **
ifa_ifwithdstaddr(*struct sockaddr *addr, int fib*);

*struct ifaddr **
ifa_ifwithnet(*struct sockaddr *addr, int ignore_ptp, int fib*);

*struct ifaddr **
ifaof_ifpforaddr(*struct sockaddr *addr, struct ifnet *ifp*);

void
ifa_ref(*struct ifaddr *ifa*);

void
ifa_free(*struct ifaddr *ifa*);

Interface Multicast Address Functions

int
if_addmulti(*struct ifnet *ifp, struct sockaddr *sa, struct ifmultiaddr **ifmap*);

int
if_delmulti(*struct ifnet *ifp, struct sockaddr *sa*);

*struct ifmultiaddr **
if_findmulti(*struct ifnet *ifp, struct sockaddr *sa*);

Output queue macros
IF_DEQUEUE(*struct ifqueue *ifq, struct mbuf *m*);

struct ifnet Member Functions
void
(*if_input)(*struct ifnet *ifp, struct mbuf *m*);

int
(*if_output)(*struct ifnet *ifp, struct mbuf *m, const struct sockaddr *dst, struct route *ro*);

void
(*if_start)(*struct ifnet *ifp*);

int
(*if_transmit)(*struct ifnet *ifp, struct mbuf *m*);

void
(*if_qflush)(*struct ifnet *ifp*);

int
(*if_ioctl)(*struct ifnet *ifp, u_long cmd, caddr_t data*);

void
(*if_init)(*void *if_softc*);

int
(*if_resolvemulti)(*struct ifnet *ifp, struct sockaddr **retsa, struct sockaddr *addr*);

struct ifaddr member function
void
(*ifa_rtrequest)(*int cmd, struct rtentry *rt, struct rt_addrinfo *info*);

Global Variables
extern struct ifnethead ifnet;
extern int if_index;
extern int ifqmaxlen;

DATA STRUCTURES

The kernel mechanisms for handling network interfaces reside primarily in the *ifnet*, *if_data*, *ifaddr*, and *ifmultiaddr* structures in <*net/if.h*> and <*net/if_var.h*> and the functions named above and defined in */sys/net/if.c*. Those interfaces which are intended to be used by user programs are defined in <*net/if.h*>; these include the interface flags, the *if_data* structure, and the structures defining the appearance of interface-related messages on the route(4) routing socket and in sysctl(3). The header file <*net/if_var.h*> defines the kernel-internal interfaces, including the *ifnet*, *ifaddr*, and *ifmultiaddr* structures and the functions which manipulate them. (A few user programs will need <*net/if_var.h*> because it is the prerequisite of some other header file like <*netinet/if_ether.h*>. Most references to those two files in particular can be replaced by <*net/ethernet.h*>.)

The system keeps a linked list of interfaces using the TAILQ macros defined in queue(3); this list is headed by a *struct ifnethead* called *ifnet*. The elements of this list are of type *struct ifnet*, and most kernel routines which manipulate interface as such accept or return pointers to these structures. Each interface structure contains an *if_data* structure used for statistics and information. Each interface also has a TAILQ of interface addresses, described by *ifaddr* structures. An AF_LINK address (see link_addr(3)) describing the link layer implemented by the interface (if any) is accessed by the *if_addr* structure. (Some trivial interfaces do not provide any link layer addresses; this structure, while still present, serves only to identify the interface name and index.)

Finally, those interfaces supporting reception of multicast datagrams have a TAILQ of multicast group memberships, described by *ifmultiaddr* structures. These memberships are reference-counted.

Interfaces are also associated with an output queue, defined as a *struct ifqueue*; this structure is used to hold packets while the interface is in the process of sending another.

The ifnet Structure

The fields of *struct ifnet* are as follows:

if_softc (*void* *) A pointer to the driver's private state block. (Initialized by driver.)

if_l2com (*void* *) A pointer to the common data for the interface's layer 2 protocol. (Initialized by **if_alloc**().)

if_vnet (*struct vnet* *) A pointer to the virtual network stack instance. (Initialized by **if_attach**().)

if_home_vnet (*struct vnet* *) A pointer to the parent virtual network stack, where this *struct ifnet* originates from. (Initialized by **if_attach**().)

if_link (**TAILQ_ENTRY**(*ifnet*)) queue(3) macro glue.

if_xname (*char* *) The name of the interface, (e.g., "fxp0" or "lo0"). (Initialized by driver (usually via **if_initname**()).)

if_dname (*const char* *) The name of the driver. (Initialized by driver (usually via **if_initname**()).)

if_dunit (*int*) A unique number assigned to each interface managed by a particular driver. Drivers may choose to set this to IF_DUNIT_NONE if a unit number is not associated with the device. (Initialized by driver (usually via **if_initname**()).)

if_refcount (*u_int*) The reference count. (Initialized by **if_alloc**().)

if_addrhead (*struct ifaddrhead*) The head of the queue(3) TAILQ containing the list of addresses assigned to this interface.

if_pcount (*int*) A count of promiscuous listeners on this interface, used to reference-count the IFF_PROMISC flag.

if_carp (*struct carp_if* *) A pointer to the CARP interface structure, carp(4). (Initialized by the driver-specific **if_ioctl**() routine.)

if_bpf (*struct bpf_if* *) Opaque per-interface data for the packet filter, bpf(4). (Initialized by **bpf_attach**().)

if_index (*u_short*) A unique number assigned to each interface in sequence as it is attached. This number can be used in a *struct sockaddr_dl* to refer to a particular interface by index (see link_addr(3)). (Initialized by **if_alloc**().)

if_vlantrunk (*struct ifvlantrunk* *) A pointer to 802.1Q trunk structure, vlan(4). (Initialized by the driver-specific **if_ioctl**() routine.)

if_flags (*int*) Flags describing operational parameters of this interface (see below). (Manipulated by generic code.)

if_drv_flags (*int*) Flags describing operational status of this interface (see below). (Manipulated by driver.)

if_capabilities (*int*) Flags describing the capabilities the interface supports (see below).

if_capenable (*int*) Flags describing the enabled capabilities of the interface (see below).

if_linkmib (*void* *) A pointer to an interface-specific MIB structure exported by ifmib(4). (Initialized by driver.)

if_linkmiblen (*size_t*) The size of said structure. (Initialized by driver.)

if_data (*struct if_data*) More statistics and information; see *The if_data structure*, below. (Initialized by driver, manipulated by both driver and generic code.)

if_multiaddrs (*struct ifmultihead*) The head of the queue(3) TAILQ containing the list of multicast addresses assigned to this interface.

if_amcount (*int*) A number of multicast requests on this interface, used to reference-count the IFF_ALLMULTI flag.

if_addr (*struct ifaddr* *) A pointer to the link-level interface address. (Initialized by **if_alloc**().)

if_snd (*struct ifaltq*) The output queue. (Manipulated by driver.)

if_broadcastaddr
(*const u_int8_t* *) A link-level broadcast bytestring for protocols with variable address length.

if_bridge (*void* *) A pointer to the bridge interface structure, if_bridge(4). (Initialized by the driver-specific **if_ioctl**() routine.)

if_label (*struct label* *) A pointer to the MAC Framework label structure, mac(4). (Initialized by **if_alloc**().)

if_afdata (*void* *) An address family dependent data region.

if_afdata_initialized
(*int*) Used to track the current state of address family initialization.

if_afdata_lock
(*struct rwlock*) An rwlock(9) lock used to protect *if_afdata* internals.

if_linktask (*struct task*) A taskqueue(9) task scheduled for link state change events of the

interface.

if_addr_lock (*struct rwlock*) An rwlock(9) lock used to protect interface-related address lists.

if_clones (**LIST_ENTRY**(*ifnet*)) queue(3) macro glue for the list of clonable network interfaces.

if_groups (**TAILQ_HEAD**(, *ifg_list*)) The head of the queue(3) TAILQ containing the list of groups per interface.

if_pf_kif (*void* *) A pointer to the structure used for interface abstraction by pf(4).

if_lagg (*void* *) A pointer to the lagg(4) interface structure.

if_alloctype (*u_char*) The type of the interface as it was at the time of its allocation. It is used to cache the type passed to **if_alloc**(), but unlike *if_type*, it would not be changed by drivers.

if_numa_domain
(*uint8_t*) The NUMA domain of the hardware device associated with the interface. This is filled in with a wildcard value unless the kernel is NUMA aware, the system is a NUMA system, and the ifnet is allocated using **if_alloc_dev**() or **if_alloc_domain**().

References to *ifnet* structures are gained by calling the **if_ref**() function and released by calling the **if_rele**() function. They are used to allow kernel code walking global interface lists to release the *ifnet* lock yet keep the *ifnet* structure stable.

There are in addition a number of function pointers which the driver must initialize to complete its interface with the generic interface layer:

if_input()
Pass a packet to an appropriate upper layer as determined from the link-layer header of the packet. This routine is to be called from an interrupt handler or used to emulate reception of a packet on this interface. A single function implementing **if_input**() can be shared among multiple drivers utilizing the same link-layer framing, e.g., Ethernet.

if_output()
Output a packet on interface *ifp*, or queue it on the output queue if the interface is already active.

if_transmit()
Transmit a packet on an interface or queue it if the interface is in use. This function will return ENOBUFS if the devices software and hardware queues are both full. This function must be installed after **if_attach**() to override the default implementation. This function is exposed in order to allow drivers to manage their own queues and to reduce the latency caused by a frequently gratuitous enqueue / dequeue pair to ifq. The suggested internal software queuing mechanism is buf_ring.

if_qflush()
Free mbufs in internally managed queues when the interface is marked down. This function must be installed after **if_attach**() to override the default implementation. This function is exposed in order to allow drivers to manage their own queues and to reduce the latency caused by a frequently gratuitous enqueue / dequeue pair to ifq. The suggested internal software queuing mechanism is buf_ring.

if_start()
Start queued output on an interface. This function is exposed in order to provide for some interface classes to share a **if_output**() among all drivers. **if_start**() may only be called when the IFF_DRV_OACTIVE flag is not set. (Thus, IFF_DRV_OACTIVE does not literally mean that output is active, but rather that the device's internal output queue is full.) Please note that this function will soon be deprecated.

if_ioctl()
Process interface-related ioctl(2) requests (defined in <*sys/sockio.h*>). Preliminary processing is done by the generic routine **ifioctl**() to check for appropriate privileges, locate the interface being manipulated, and perform certain generic operations like twiddling flags and flushing queues. See the description of **ifioctl**() below for more information.

if_init()
Initialize and bring up the hardware, e.g., reset the chip and enable the receiver unit. Should mark the interface running, but not active (IFF_DRV_RUNNING, ~IIF_DRV_OACTIVE).

if_resolvemulti()
Check the requested multicast group membership, *addr*, for validity, and if necessary compute a link-layer group which corresponds to that address which is returned in **retsa*. Returns zero on success, or an error code on failure.

Interface Flags

Interface flags are used for a number of different purposes. Some flags simply indicate information about the type of interface and its capabilities; others are dynamically manipulated to reflect the current

state of the interface. Flags of the former kind are marked <S> in this table; the latter are marked <D>. Flags which begin with "IFF_DRV_" are stored in *if_drv_flags*; all other flags are stored in *if_flags*.

The macro IFF_CANTCHANGE defines the bits which cannot be set by a user program using the SIOCSIFFLAGS command to ioctl(2); these are indicated by an asterisk ('*') in the following listing.

IFF_UP	<D> The interface has been configured up by the user-level code.
IFF_BROADCAST	<S*> The interface supports broadcast.
IFF_DEBUG	<D> Used to enable/disable driver debugging code.
IFF_LOOPBACK	<S> The interface is a loopback device.
IFF_POINTOPOINT	<S*> The interface is point-to-point; "broadcast" address is actually the address of the other end.
IFF_DRV_RUNNING	<D*> The interface has been configured and dynamic resources were successfully allocated. Probably only useful internal to the interface.
IFF_NOARP	<D> Disable network address resolution on this interface.
IFF_PROMISC	<D*> This interface is in promiscuous mode.
IFF_PPROMISC	<D> This interface is in the permanently promiscuous mode (implies IFF_PROMISC).
IFF_ALLMULTI	<D*> This interface is in all-multicasts mode (used by multicast routers).
IFF_DRV_OACTIVE	<D*> The interface's hardware output queue (if any) is full; output packets are to be queued.
IFF_SIMPLEX	<S*> The interface cannot hear its own transmissions.
IFF_LINK0	
IFF_LINK1	
IFF_LINK2	<D> Control flags for the link layer. (Currently abused to select among multiple physical layers on some devices.)
IFF_MULTICAST	<S*> This interface supports multicast.
IFF_CANTCONFIG	<S*> The interface is not configurable in a meaningful way. Primarily useful for IFT_USB interfaces registered at the interface list.
IFF_MONITOR	<D> This interface blocks transmission of packets and discards incoming packets after BPF processing. Used to monitor network traffic but not interact with the network in question.
IFF_STATICARP	<D> Used to enable/disable ARP requests on this interface.
IFF_DYING	<D*> Set when the *ifnet* structure of this interface is being released and still has *if_refcount* references.
IFF_RENAMING	<D> Set when this interface is being renamed.

Interface Capabilities Flags

Interface capabilities are specialized features an interface may or may not support. These capabilities are very hardware-specific and allow, when enabled, to offload specific network processing to the interface or to offer a particular feature for use by other kernel parts.

It should be stressed that a capability can be completely uncontrolled (i.e., stay always enabled with no way to disable it) or allow limited control over itself (e.g., depend on another capability's state.) Such peculiarities are determined solely by the hardware and driver of a particular interface. Only the driver possesses the knowledge on whether and how the interface capabilities can be controlled. Consequently, capabilities flags in *if_capenable* should never be modified directly by kernel code other than the interface driver. The command SIOCSIFCAP to **ifioctl**() is the dedicated means to attempt altering *if_capenable* on an interface. Userland code shall use ioctl(2).

The following capabilities are currently supported by the system:

IFCAP_RXCSUM	This interface can do checksum validation on receiving data. Some interfaces do not have sufficient buffer storage to store frames above a certain MTU-size completely. The driver for the interface might disable hardware checksum validation if the MTU is set above the hardcoded limit.
IFCAP_TXCSUM	This interface can do checksum calculation on transmitting data.
IFCAP_HWCSUM	A shorthand for (IFCAP_RXCSUM | IFCAP_TXCSUM).
IFCAP_NETCONS	This interface can be a network console.
IFCAP_VLAN_MTU	The vlan(4) driver can operate over this interface in software tagging mode without having to decrease MTU on vlan(4) interfaces below 1500 bytes. This implies the ability of this interface to cope with frames somewhat longer than permitted by the Ethernet specification.
IFCAP_VLAN_HWTAGGING	This interface can do VLAN tagging on output and demultiplex frames by their VLAN tag on input.
IFCAP_JUMBO_MTU	This Ethernet interface can transmit and receive frames up to 9000 bytes long.
IFCAP_POLLING	This interface supports polling(4). See below for details.

IFCAP_VLAN_HWCSUM	This interface can do checksum calculation on both transmitting and receiving data on vlan(4) interfaces (implies IFCAP_HWCSUM).
IFCAP_TSO4	This Ethernet interface supports TCP4 Segmentation offloading.
IFCAP_TSO6	This Ethernet interface supports TCP6 Segmentation offloading.
IFCAP_TSO	A shorthand for (IFCAP_TSO4 | IFCAP_TSO6).
IFCAP_TOE4	This Ethernet interface supports TCP4 Offload Engine.
IFCAP_TOE6	This Ethernet interface supports TCP6 Offload Engine.
IFCAP_TOE	A shorthand for (IFCAP_TOE4 | IFCAP_TOE6).
IFCAP_WOL_UCAST	This Ethernet interface supports waking up on any Unicast packet.
IFCAP_WOL_MCAST	This Ethernet interface supports waking up on any Multicast packet.
IFCAP_WOL_MAGIC	This Ethernet interface supports waking up on any Magic packet such as those sent by wake(8).
IFCAP_WOL	A shorthand for (IFCAP_WOL_UCAST | IFCAP_WOL_MCAST | IFCAP_WOL_MAGIC).
IFCAP_VLAN_HWFILTER	This interface supports frame filtering in hardware on vlan(4) interfaces.
IFCAP_VLAN_HWTSO	This interface supports TCP Segmentation offloading on vlan(4) interfaces (implies IFCAP_TSO).
IFCAP_LINKSTATE	This Ethernet interface supports dynamic link state changes.
IFCAP_NETMAP	This Ethernet interface supports netmap(4).

The ability of advanced network interfaces to offload certain computational tasks from the host CPU to the board is limited mostly to TCP/IP. Therefore a separate field associated with an interface (see

ifnet.if_data.ifi_hwassist below) keeps a detailed description of its enabled capabilities specific to TCP/IP processing. The TCP/IP module consults the field to see which tasks can be done on an *outgoing* packet by the interface. The flags defined for that field are a superset of those for *mbuf.m_pkthdr.csum_flags*, namely:

CSUM_IP The interface will compute IP checksums.

CSUM_TCP The interface will compute TCP checksums.

CSUM_UDP The interface will compute UDP checksums.

An interface notifies the TCP/IP module about the tasks the former has performed on an *incoming* packet by setting the corresponding flags in the field *mbuf.m_pkthdr.csum_flags* of the *mbuf chain* containing the packet. See mbuf(9) for details.

The capability of a network interface to operate in polling(4) mode involves several flags in different global variables and per-interface fields. The capability flag IFCAP_POLLING set in interface's *if_capabilities* indicates support for polling(4) on the particular interface. If set in *if_capabilities*, the same flag can be marked or cleared in the interface's *if_capenable* within **ifioctl**(), thus initiating switch of the interface to polling(4) mode or interrupt mode, respectively. The actual mode change is managed by the driver-specific **if_ioctl**() routine. The polling(4) handler returns the number of packets processed.

The if_data Structure

The *if_data* structure contains statistics and identifying information used by management programs, and which is exported to user programs by way of the ifmib(4) branch of the sysctl(3) MIB. The following elements of the *if_data* structure are initialized by the interface and are not expected to change significantly over the course of normal operation:

ifi_type (*u_char*) The type of the interface, as defined in <*net/if_types.h*> and described below in the *Interface Types* section.

ifi_physical (*u_char*) Intended to represent a selection of physical layers on devices which support more than one; never implemented.

ifi_addrlen (*u_char*) Length of a link-layer address on this device, or zero if there are none. Used to initialized the address length field in *sockaddr_dl* structures referring to this interface.

ifi_hdrlen (*u_char*) Maximum length of any link-layer header which might be prepended by the driver to a packet before transmission. The generic code computes the

maximum over all interfaces and uses that value to influence the placement of data in *mbuf*s to attempt to ensure that there is always sufficient space to prepend a link-layer header without allocating an additional *mbuf*.

ifi_datalen (*u_char*) Length of the *if_data* structure. Allows some stabilization of the routing socket ABI in the face of increases in the length of *struct ifdata*.

ifi_mtu (*u_long*) The maximum transmission unit of the medium, exclusive of any link-layer overhead.

ifi_metric (*u_long*) A dimensionless metric interpreted by a user-mode routing process.

ifi_baudrate (*u_long*) The line rate of the interface, in bits per second.

ifi_hwassist (*u_long*) A detailed interpretation of the capabilities to offload computational tasks for *outgoing* packets. The interface driver must keep this field in accord with the current value of *if_capenable*.

ifi_epoch (*time_t*) The system uptime when interface was attached or the statistics below were reset. This is intended to be used to set the SNMP variable *ifCounterDiscontinuityTime*. It may also be used to determine if two successive queries for an interface of the same index have returned results for the same interface.

The structure additionally contains generic statistics applicable to a variety of different interface types (except as noted, all members are of type *u_long*):

ifi_link_state (*u_char*) The current link state of Ethernet interfaces. See the *Interface Link States* section for possible values.

ifi_ipackets Number of packets received.

ifi_ierrors Number of receive errors detected (e.g., FCS errors, DMA overruns, etc.). More detailed breakdowns can often be had by way of a link-specific MIB.

ifi_opackets Number of packets transmitted.

ifi_oerrors Number of output errors detected (e.g., late collisions, DMA overruns, etc.). More detailed breakdowns can often be had by way of a link-specific MIB.

ifi_collisions Total number of collisions detected on output for CSMA interfaces. (This member is sometimes [ab]used by other types of interfaces for other output error counts.)

ifi_ibytes Total traffic received, in bytes.

ifi_obytes Total traffic transmitted, in bytes.

ifi_imcasts Number of packets received which were sent by link-layer multicast.

ifi_omcasts Number of packets sent by link-layer multicast.

ifi_iqdrops Number of packets dropped on input. Rarely implemented.

ifi_oqdrops Number of packets dropped on output.

ifi_noproto Number of packets received for unknown network-layer protocol.

ifi_lastchange (*struct timeval*) The time of the last administrative change to the interface (as required for SNMP).

Interface Types

The header file <*net/if_types.h*> defines symbolic constants for a number of different types of interfaces. The most common are:

IFT_OTHER	none of the following
IFT_ETHER	Ethernet
IFT_ISO88023	ISO 8802-3 CSMA/CD
IFT_ISO88024	ISO 8802-4 Token Bus
IFT_ISO88025	ISO 8802-5 Token Ring
IFT_ISO88026	ISO 8802-6 DQDB MAN
IFT_FDDI	FDDI
IFT_PPP	Internet Point-to-Point Protocol (ppp(8))
IFT_LOOP	The loopback (lo(4)) interface
IFT_SLIP	Serial Line IP
IFT_PARA	Parallel-port IP ("PLIP")
IFT_ATM	Asynchronous Transfer Mode
IFT_USB	USB Interface

Interface Link States

The following link states are currently defined:

LINK_STATE_UNKNOWN	The link is in an invalid or unknown state.
LINK_STATE_DOWN	The link is down.
LINK_STATE_UP	The link is up.

The ifaddr Structure

Every interface is associated with a list (or, rather, a TAILQ) of addresses, rooted at the interface structure's *if_addrhead* member. The first element in this list is always an AF_LINK address representing the interface itself; multi-access network drivers should complete this structure by filling in their link-layer addresses after calling **if_attach**(). Other members of the structure represent network-layer addresses which have been configured by means of the SIOCAIFADDR command to ioctl(2), called on a socket of the appropriate protocol family. The elements of this list consist of *ifaddr* structures. Most protocols will declare their own protocol-specific interface address structures, but all begin with a *struct ifaddr* which provides the most-commonly-needed functionality across all protocols. Interface addresses are reference-counted.

The members of *struct ifaddr* are as follows:

ifa_addr (*struct sockaddr* *) The local address of the interface.

ifa_dstaddr (*struct sockaddr* *) The remote address of point-to-point interfaces, and the broadcast address of broadcast interfaces. (*ifa_broadaddr* is a macro for *ifa_dstaddr*.)

ifa_netmask (*struct sockaddr* *) The network mask for multi-access interfaces, and the confusion generator for point-to-point interfaces.

ifa_ifp (*struct ifnet* *) A link back to the interface structure.

ifa_link (**TAILQ_ENTRY**(*ifaddr*)) queue(3) glue for list of addresses on each interface.

ifa_rtrequest See below.

ifa_flags (*u_short*) Some of the flags which would be used for a route representing this address in the route table.

ifa_refcnt (*short*) The reference count.

References to *ifaddr* structures are gained by calling the **ifa_ref**() function and released by calling the **ifa_free**() function.

ifa_rtrequest() is a pointer to a function which receives callouts from the routing code (**rtrequest**()) to perform link-layer-specific actions upon requests to add, or delete routes. The *cmd* argument indicates the request in question: RTM_ADD, or RTM_DELETE. The *rt* argument is the route in question; the *info* argument contains the specific destination being manipulated.

FUNCTIONS

The functions provided by the generic interface code can be divided into two groups: those which manipulate interfaces, and those which manipulate interface addresses. In addition to these functions, there may also be link-layer support routines which are used by a number of drivers implementing a specific link layer over different hardware; see the documentation for that link layer for more details.

The ifmultiaddr Structure

Every multicast-capable interface is associated with a list of multicast group memberships, which indicate at a low level which link-layer multicast addresses (if any) should be accepted, and at a high level, in which network-layer multicast groups a user process has expressed interest.

The elements of the structure are as follows:

ifma_link (**LIST_ENTRY**(*ifmultiaddr*)) queue(3) macro glue.

ifma_addr (*struct sockaddr* *) A pointer to the address which this record represents. The memberships for various address families are stored in arbitrary order.

ifma_lladdr (*struct sockaddr* *) A pointer to the link-layer multicast address, if any, to which the network-layer multicast address in *ifma_addr* is mapped, else a null pointer. If this element is non-nil, this membership also holds an invisible reference to another membership for that link-layer address.

ifma_refcount (*u_int*) A reference count of requests for this particular membership.

Interface Manipulation Functions

if_alloc()

Allocate and initialize *struct ifnet*. Initialization includes the allocation of an interface index and may include the allocation of a *type* specific structure in *if_l2com*.

if_alloc_dev()

Allocate and initialize *struct ifnet* as **if_alloc**() does, with the addition that the ifnet can be tagged with the appropriate NUMA domain derived from the *dev* argument passed by the caller.

if_alloc_domain()

Allocate and initialize *struct ifnet* as **if_alloc**() does, with the addition that the ifnet will be tagged with the NUMA domain via the *numa_domain* argument passed by the caller.

if_attach()
Link the specified interface *ifp* into the list of network interfaces. Also initialize the list of addresses on that interface, and create a link-layer *ifaddr* structure to be the first element in that list. (A pointer to this address structure is saved in the *ifnet* structure.) The *ifp* must have been allocated by **if_alloc**(), **if_alloc_dev**() or **if_alloc_domain**().

if_detach()
Shut down and unlink the specified *ifp* from the interface list.

if_free()
Free the given *ifp* back to the system. The interface must have been previously detached if it was ever attached.

if_free_type()
Identical to **if_free**() except that the given *type* is used to free *if_l2com* instead of the type in *if_type*. This is intended for use with drivers that change their interface type.

if_down()
Mark the interface *ifp* as down (i.e., IFF_UP is not set), flush its output queue, notify protocols of the transition, and generate a message from the route(4) routing socket.

if_up()
Mark the interface *ifp* as up, notify protocols of the transition, and generate a message from the route(4) routing socket.

ifpromisc()
Add or remove a promiscuous reference to *ifp*. If *pswitch* is true, add a reference; if it is false, remove a reference. On reference count transitions from zero to one and one to zero, set the IFF_PROMISC flag appropriately and call **if_ioctl**() to set up the interface in the desired mode.

if_allmulti()
As **ifpromisc**(), but for the all-multicasts (IFF_ALLMULTI) flag instead of the promiscuous flag.

ifunit()
Return an *ifnet* pointer for the interface named *name*.

ifunit_ref()

Return a reference-counted (via **ifa_ref**()) *ifnet* pointer for the interface named *name*. This is the preferred function over **ifunit**(). The caller is responsible for releasing the reference with **if_rele**() when it is finished with the ifnet.

ifioctl()
Process the ioctl request *cmd*, issued on socket *so* by thread *td*, with data parameter *data*. This is the main routine for handling all interface configuration requests from user mode. It is ordinarily only called from the socket-layer ioctl(2) handler, and only for commands with class 'i'. Any unrecognized commands will be passed down to socket *so*'s protocol for further interpretation. The following commands are handled by **ifioctl**():

SIOCGIFCONF	Get interface configuration. (No call-down to driver.)
SIOCSIFNAME	Set the interface name. RTM_IFANNOUNCE departure and arrival messages are sent so that routing code that relies on the interface name will update its interface list. Caller must have appropriate privilege. (No call-down to driver.)
SIOCGIFCAP SIOCGIFDATA SIOCGIFFIB SIOCGIFFLAGS SIOCGIFMETRIC SIOCGIFMTU SIOCGIFPHYS	Get interface capabilities, data, FIB, flags, metric, MTU, medium selection. (No call-down to driver.)
SIOCSIFCAP	Enable or disable interface capabilities. Caller must have appropriate privilege. Before a call to the driver-specific **if_ioctl**() routine, the requested mask for enabled capabilities is checked against the mask of capabilities supported by the interface, *if_capabilities*. Requesting to enable an unsupported capability is invalid. The rest is supposed to be done by the driver, which includes updating *if_capenable* and *if_data.ifi_hwassist* appropriately.
SIOCGIFCAPNV	NV(9) version of the SIOCGIFCAP ioctl. Caller must provide a pointer to *struct ifreq_cap_nv* as *data*, where the member buffer points to some buffer containing buf_length bytes. The serialized nvlist with description of the device capabilities is written to the buffer. If buffer is too short, the structure is updated with buffer

member set to NULL, length set to the minimal required length, and error EFBIG is returned.

Elements of the returned nvlist for simple capabilities are boolean, identified by names. Presence of the boolean element means that corresponding capability is supported by the interface. Element's value describes the current configured state: true means that the capability is enabled, and false that it is disabled.

Driver indicates support for both SIOCGIFCAPNV and SIOCSIFCAPNV requests by setting IFCAP_NV non-modifiable capability bit in if_capabilities.

SIOCSIFCAPNV NV(9) version of the SIOCSIFCAP ioctl. Caller must provide the pointer to *struct ifreq_cap_nv* as *data*, where the member buffer points to serialized nvlist of length bytes. Each element of nvlist describes a requested update of one capability, identified by the element name. For simple capabilities, the element must be boolean. Its true value means that the caller asks to enable the capability, and false value to disable. Only capabilities listed in the nvlist are affected by the call.

SIOCSIFFIB Sets interface FIB. Caller must have appropriate privilege. FIB values start at 0 and values greater or equals than *net.fibs* are considered invalid.

SIOCSIFFLAGS Change interface flags. Caller must have appropriate privilege. If a change to the IFF_UP flag is requested, **if_up**() or **if_down**() is called as appropriate. Flags listed in IFF_CANTCHANGE are masked off, and the field *if_flags* in the interface structure is updated. Finally, the driver **if_ioctl**() routine is called to perform any setup requested.

SIOCSIFMETRIC
SIOCSIFPHYS Change interface metric or medium. Caller must have appropriate privilege.

SIOCSIFMTU Change interface MTU. Caller must have appropriate privilege. MTU values less than 72 or greater than 65535 are considered invalid. The driver **if_ioctl**() routine is called to implement the change; it is responsible for any additional sanity checking and for actually modifying the MTU in the interface structure.

SIOCADDMULTI
SIOCDELMULTI — Add or delete permanent multicast group memberships on the interface. Caller must have appropriate privilege. The **if_addmulti**() or **if_delmulti**() function is called to perform the operation; qq.v.

SIOCAIFADDR
SIOCDIFADDR — The socket's protocol control routine is called to implement the requested action.

Interface Address Functions

Several functions exist to look up an interface address structure given an address. **ifa_ifwithaddr**() returns an interface address with either a local address or a broadcast address precisely matching the parameter *addr*. **ifa_ifwithdstaddr**() returns an interface address for a point-to-point interface whose remote ("destination") address is *addr* and a fib is *fib*. If *fib* is RT_ALL_FIBS, then the first interface address matching *addr* will be returned.

ifa_ifwithnet() returns the most specific interface address which matches the specified address, *addr*, subject to its configured netmask, or a point-to-point interface address whose remote address is *addr* if one is found. If *ignore_ptp* is true, skip point-to-point interface addresses. The *fib* parameter is handled the same way as by **ifa_ifwithdstaddr**().

ifaof_ifpforaddr() returns the most specific address configured on interface *ifp* which matches address *addr*, subject to its configured netmask. If the interface is point-to-point, only an interface address whose remote address is precisely *addr* will be returned.

All of these functions return a null pointer if no such address can be found.

Interface Multicast Address Functions

The **if_addmulti**(), **if_delmulti**(), and **if_findmulti**() functions provide support for requesting and relinquishing multicast group memberships, and for querying an interface's membership list, respectively. The **if_addmulti**() function takes a pointer to an interface, *ifp*, and a generic address, *sa*. It also takes a pointer to a *struct ifmultiaddr* * which is filled in on successful return with the address of the group membership control block. The **if_addmulti**() function performs the following four-step process:

1. Call the interface's **if_resolvemulti**() entry point to determine the link-layer address, if any, corresponding to this membership request, and also to give the link layer an opportunity to veto this membership request should it so desire.

2. Check the interface's group membership list for a pre-existing membership for this group. If one is not found, allocate a new one; if one is, increment its reference count.

3. If the **if_resolvemulti**() routine returned a link-layer address corresponding to the group, repeat the previous step for that address as well.

4. If the interface's multicast address filter needs to be changed because a new membership was added, call the interface's **if_ioctl**() routine (with a *cmd* argument of SIOCADDMULTI) to request that it do so.

The **if_delmulti**() function, given an interface *ifp* and an address, *sa*, reverses this process. Both functions return zero on success, or a standard error number on failure.

The **if_findmulti**() function examines the membership list of interface *ifp* for an address matching *sa*, and returns a pointer to that *struct ifmultiaddr* if one is found, else it returns a null pointer.

SEE ALSO

ioctl(2), link_addr(3), queue(3), sysctl(3), bpf(4), ifmib(4), lo(4), netintro(4), polling(4), config(8), ppp(8), mbuf(9), rtentry(9)

Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated*, Vol. 2, Addison-Wesley, ISBN 0-201-63354-X.

AUTHORS

This manual page was written by Garrett A. Wollman.