Ethernet Device Initialization

The *init_etherdev()* function is called by most Ethernet drivers at initialization time to initialize and possibly allocate a *net_device* structure. It is a convienience front end that forwards the call to the more generic *init_netdev()* providing the naming string *eth%d* that is eventually used in the construction of the interface name eth*n.* This module resides in drivers/net/net_init.c.
If the *dev* parameter is NULL, then the *net_device* structure is allocated and initialized. Otherwise it is only initialized.

```
struct net_device *init_etherdev(struct net_device *dev,
                                      int sizeof_priv)
{
    return init_netdev(dev, sizeof_priv, "eth%d",
                       ether_setup);
}

static struct net_device *init_netdev(struct net_device *dev,
    int sizeof_priv, char *mask,
    void (*setup)(struct net_device *))
{
    int new_device = 0;

/*
 *Allocate a device if one is not provided.
 */

    if (dev == NULL) {
         dev=init_alloc_dev(sizeof_priv);
         if(dev==NULL)
              return NULL;
         new_device = 1;
    }

```

On return to *init_netdev()* an attempt is made to try to allocate a name. For ethernet devices, *mask* points to the string *eth%d.*

```
/*
 *Allocate a name
 */

    if (dev->name[0] == '\0' || dev->name[0] == ' ') {
         strcpy(dev->name, mask);
         if (dev_alloc_name(dev, mask)<0) {
              if (new_device)
                   kfree(dev);
              return NULL;
         }
    }

```

On return to init_netdev() , the call at line 158 is used to copy any boot time options. This is a vestige of the bad old days of ISA NICs in which it might be necessary to supply I/O addresses and irqs manually.

```
158     netdev_boot_setup_check(dev);
159
```

The setup() function referenced here is an input parameter to this routine. For ethernet devices the call is actually to ether_setup().

```
160 /*
161  *Configure via the caller provided setup function then
162  *register if needed.
163  */
164
165     setup(dev);
166
```

Finally, if this was not a previously allocated device, init_netdev() attempts to register the device with the protocols.

```
167     if (new_device) {
168          int err;
169
170          rtnl_lock();
171          err = register_netdevice(dev);
172          rtnl_unlock();
173
174          if (err < 0) {
175               kfree(dev);
176               dev = NULL;
177          }
178     }
179     return dev;
180 }
181
```

Allocation of the ***net_device*** structure

The init_alloc_dev() function allocates a struct net_device plus whatever memory is requested for the device driver private area. The private area is a 32 byte aligned structure that follows the struct net_device and is accessed via the dev->priv pointer.

```
static struct net_device *init_alloc_dev(int sizeof_priv)
{
    struct net_device *dev;
    int alloc_size;

/* ensure 32-byte alignment of the private area */
    alloc_size = sizeof (*dev) + sizeof_priv + 31;

    dev = (struct net_device *) kmalloc (alloc_size,
               GFP_KERNEL);
    if (dev == NULL)
    {
         printk(KERN_ERR "alloc_dev: Unable to allocate
                   device memory.\n");
         return NULL;
    }

```

Initialize the net_device structure and link in the private area if it was provided.

```
    memset(dev, 0, alloc_size);

    if (sizeof_priv)
         dev->priv = (void *) (((long)(dev + 1) + 31) & ~31);

    return dev;
}

```

The dev_alloc_name() function attempts to allocate an available device name in the space name0 through name99 where the name of an Ethernet device is eth. It uses a serial search through the namespace which terminates when an unallocated name is encountered.

```
int dev_alloc_name(struct net_device *dev, const char *name)
{
	int i;
	char buf[32];
	char *p;

/*
 * Verify the string as this thing may have come from
 * the user.  There must be either one "%d" and no other "%"
 * characters, or no "%" characters at all.
 */
	p = strchr(name, '%');
	if (p && (p[1] != 'd' || strchr(p+2, '%')))
		return -EINVAL;

```

Here a series of names (e.g., eth0, eth1, eth2, ...) are generated. The name parameter that is passed to the snprintf() function would be eth%d in this case, and the value of i the index. The __dev_get_by_name() function returns NULL if the name is not already in use.

```
/*
 * If you need over 100 please also fix the algorithm...
 */
	for (i = 0; i < 100; i++) {
		snprintf(buf,sizeof(buf),name,i);
		if (__dev_get_by_name(buf) == NULL) {
			strcpy(dev->name, buf);
			return i;
		}
	}
	return -ENFILE;/* Over 100 of the things .. bail out! */
}

struct net_device *__dev_get_by_name(const char *name)
{
	struct net_device *dev;

	for (dev = dev_base; dev != NULL; dev = dev->next) {
		if (strncmp(dev->name, name, IFNAMSIZ) == 0)
			return dev;
	}
	return NULL;
}
```

The ether_setup() function just fills in spots in the dev structure that are common to all ethernet drivers. The labels of the form eth_ are references to functions the live in net/ethernet/eth.c

```
void ether_setup(struct net_device *dev)
{
/* Fill in the fields of the device structure with ethernet-
   generic values.
   This should be in a common file instead of per-driver.  */

    dev->change_mtu = eth_change_mtu;
    dev->hard_header = eth_header;
    dev->rebuild_header = eth_rebuild_header;
    dev->set_mac_address = eth_mac_addr;
    dev->hard_header_cache = eth_header_cache;
    dev->header_cache_update = eth_header_cache_update;
    dev->hard_header_parse = eth_header_parse;

    dev->type= ARPHRD_ETHER;
    dev->hard_header_len = ETH_HLEN;
    dev->mtu= 1500; /* eth_mtu */
    dev->addr_len= ETH_ALEN;
    dev->tx_queue_len= 100;/* Ethernet wants good queues */

    memset(dev->broadcast,0xFF, ETH_ALEN);

```

Indicate that the interface supports both hardware broadcast and multicast.

```
/* New-style flags. */
    dev->flags= IFF_BROADCAST|IFF_MULTICAST;
}
```

Registering a net_device

We shall see that the call to register_netdevice() will trigger indirects calls to a relatively large collection of change–of–state handlers associated with various protocol routines. The recipients of these indirect calls will in turn use the netlinks interface to send routing updates to the FIB manager.

```
int register_netdevice(struct net_device *dev)
{
    struct net_device *d, **dp;
    #ifdef CONFIG_NET_DIVERT
    int ret;
    #endif

    spin_lock_init(&dev->queue_lock);
    spin_lock_init(&dev->xmit_lock);
    dev->xmit_lock_owner = -1;
#ifdef CONFIG_NET_FASTROUTE
    dev->fastpath_lock = RW_LOCK_UNLOCKED;
#endif

```

The value of dev_boot_phase is statically initialized to 1 at compile time. It is reset to 0 during the call to net_dev_init() ensuring that the code is executed exactly once at boot time when the first network driver initializes.

```
    if (dev_boot_phase)
         net_dev_init();

```

It would be nice to understand diverters.

```
#ifdef CONFIG_NET_DIVERT
    ret = alloc_divert_blk(dev);
    if (ret)
         return ret;
#endif /* CONFIG_NET_DIVERT */

```

The iflink element is an alternative identifying index that can be set by the device driver. It is initialized to −1 before calling the drivers initialization routine, dev−>init(). If control reaches this point via any path (including the init_etherdev() path) which includes the creation of the net_device structure, dev−>init() will necessarily be NULL. Device drivers which allocate the net_device structure and later register can specific a callback that will be activated here.

```
    dev->iflink = -1;

/* Init, if this function is available */
    if (dev->init && dev->init(dev) != 0) {
#ifdef CONFIG_NET_DIVERT
         free_divert_blk(dev);
#endif
         return -EIO;

```

Each interface is given a unique identifier number. This number is also inherited by dev−>iflink if the device driver didn't provide the info in its init routine.

```
    dev->ifindex = dev_new_index();
    if (dev->iflink == -1)
         dev->iflink = dev->ifindex;

/* Check for existence, and append to tail of chain */
    for (dp=&dev_base; (d=*dp) != NULL; dp=&d->next) {
         if (d == dev || strcmp(d->name, dev->name) == 0) {
#ifdef CONFIG_NET_DIVERT
              free_divert_blk(dev);
#endif
              return -EEXIST;
         }
    }
/*
 *nil rebuild_header routine,
 *that should be never called and used as just bug trap.
 */

    if (dev->rebuild_header == NULL)
         dev->rebuild_header = default_rebuild_header;

```

```
/*
 *Default initial state at registry is that the
 *device is present.
 */

    set_bit(__LINK_STATE_PRESENT, &dev->state);

    dev->next = NULL;
    dev_init_scheduler(dev);
    write_lock_bh(&dev_base_lock);
    *dp = dev;
    dev_hold(dev);
    dev->deadbeaf = 0;
    write_unlock_bh(&dev_base_lock);

```

The call to notifier_call_chain() results in a call to the callback function associated with every notifier block in the netdev_chain passing them the event code NETDEV_REGISTER and a pointer to the struct netdevice. As will be shown there may be twenty or more such functions, but in this case they really don't do very much.

```
/* Notify protocols, that a new device appeared. */
    notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);

    net_run_sbin_hotplug(dev, "register");

    return 0;
}
```

Device scheduler initialization

The scheduler is initially configure to support no queuing at all. This appears to get rectified during the call to dev_activate().

```
void dev_init_scheduler(struct net_device *dev)
{
	write_lock(&qdisc_tree_lock);
	spin_lock_bh(&dev->queue_lock);
	dev->qdisc = &noop_qdisc;
	spin_unlock_bh(&dev->queue_lock);
	dev->qdisc_sleeping = &noop_qdisc;
	dev->qdisc_list = NULL;
	write_unlock(&qdisc_tree_lock);

	dev_watchdog_init(dev);
}

```

Notifier Chains

The notifier chain facility is a general mechanism provided by the kernel. It is designed to provide a way for kernel elements to express interest in being informed about the occurence of general asynchronous events. The basic building block of the mechanism is the struct notifier_block which is defined in include/linux/notifier.h. The block contains a pointer to the function to be called when the event occurs. The parameters passed to the function include:

- a pointer to the notifier block itself,
- an event code such as NETDEV_REGISTER or NETDEV_UNREGISTER,
- and a pointer to an unspecified private data type which in the case of the network chain points to the associated struct netdevice.

```
struct notifier_block
{
    int (*notifier_call)(struct notifier_block *self,
              unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

```
static struct notifier_block *netdev_chain=NULL;
```

The kernel function notifier_chain_register() assembles related notifier blocks into notifier chains. Modules within the networking subsystem use the register_netdevice_notifier() function defined in net/core/dev.c to add their own notifier blocks to the netdev_chain which is statically initialized as NULL in dev.c.

```
int register_netdevice_notifier(struct notifier_block *nb)
{
    return notifier_chain_register(&netdev_chain, nb);
}
```


```
static struct notifier_block *netdev_chain=NULL;
```

Adding the notifier_block to the chain.

The kernel routine notifier_chain_register() links the notifier block into the specified chain in priority order.

```

int notifier_chain_register(struct notifier_block **list,
            struct notifier_block *n)
{
    write_lock(&notifier_lock);
    while(*list)
    {
        if(n->priority > (*list)->priority)
            break;
        list= &((*list)->next);
    }
    n->next = *list;
    *list=n;
    write_unlock(&notifier_lock);
    return 0;
}
```

Here are the notifiers associated with net_device events.

```
/* netdevice notifier chain */
#define NETDEV_UP                  0x0001
    /* For now you can't veto a device up/down */
#define NETDEV_DOWN                0x0002
#define NETDEV_REBOOT              0x0003
    /* Tell a protocol stack a network interface
    detected a hardware crash and restarted
   - we can use this eg to kick tcp sessions
                                      once done */
#define NETDEV_CHANGE              0x0004
    /* Notify devstate change */
#define NETDEV_REGISTER            0x0005
#define NETDEV_UNREGISTER          0x0006
#define NETDEV_CHANGEMTU           0x0007
#define NETDEV_CHANGEADDR          0x0008
#define NETDEV_GOING_DOWN          0x0009
#define NETDEV_CHANGENAME          0x000A

```

Invoking notifier_call_chain()

When a function such as netdev_init() makes the call to notifier_call_chain(), it results it a callback being made for every notifier block that is in the chain. These notifier callback functions typically contain a switch() block which they used to select and process only those event types in which they are interested.

```
/* Notify protocols, that a new device appeared. */
    notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
```

This structure is illustrated below in the rtnetlink_event() callback.

```
static int rtnetlink_event(struct notifier_block *this,
                           unsigned long event, void *ptr)
{
    struct net_device *dev = ptr;
    switch (event) {
    case NETDEV_UNREGISTER:
         rtmsg_ifinfo(RTM_DELLINK, dev, ~0U);
         break;
    case NETDEV_REGISTER:
         rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);
         break;
    case NETDEV_UP:
    case NETDEV_DOWN:
         rtmsg_ifinfo(RTM_NEWLINK, dev,
                      IFF_UP|IFF_RUNNING);
         break;
    case NETDEV_CHANGE:
    case NETDEV_GOING_DOWN:
         break;
    default:
         rtmsg_ifinfo(RTM_NEWLINK, dev, 0);
         break;
    }
    return NOTIFY_DONE;
}
```

The entire collection of callers of register_netdevice_notifier() is quite large. Each of the modules shown below has a callback function in the netdev chain. However, only the notifiers shown in red have any impact on IP_V4.

Referenced (in 35 files total) in:

include/linux/netdevice.h, line 454	
net/netsyms.c, line 465	
net/appletalk/aarp.c, line 859	
net/appletalk/ddp.c, line 1974	
net/ax25/af_ax25.c, line 1851	
net/core/dev.c, line 850	register_netdevice_notifier
net/core/dst.c, line 214	dst_dev_event()
net/core/rtnetlink.c, line 526	rtnetlink_dev_notifier()
net/ipv4/devinet.c, line 1140	ip_netdev_notifier()
net/ipv4/ipmr.c, line 1756	ip_mr_notifiier()
net/ipv4/fib_frontend.c, line 652	fib_netdev_notifier()
net/ipv4/fib_rules.c, line 466	fib_rules_notifier()
net/ipv4/netfilter/ip_queue.c, line 647	ipq_dev_notifier()
net/ipv4/netfilter/ipfwadm_core.c, line 1385	
net/ipv4/netfilter/ipt_MASQUERADE.c, line 190	

- net/ipx/af_ipx.c, line 2562
- net/netrom/af_netrom.c, line 1311
- net/decnet/af_decnet.c, line 2260
- net/decnet/dn_rules.c, line 363
- net/ipv6/ipv6_sockglue.c, line 563
- net/ipv6/netfilter/ip6_queue.c, line 703
- net/bridge/br.c, line 51
- net/econet/af_econet.c, line 1125
- net/x25/af_x25.c, line 1324
- net/rose/af_rose.c, line 1463
- net/wanrouter/af_wanpipe.c, line 2762
- net/packet/af_packet.c, line 1896
- net/irda/af_irda.c, line 2590
- net/atm/clip.c:
  - line 739
  - line 740
- net/atm/mpc.c, line 768
- net/8021q/vlan.c, line 99
- drivers/net/wan/lapbether.c, line 478
- drivers/net/hamradio/bpqether.c, line 614
- drivers/net/pppoe.c, line 1065
- drivers/net/bonding.c, line 2010

Actions associated with NETDEV_REGISTER

net/core/dst.c, line 214 dst_dev_event()

No action is taken on REGISTER. On UNREGISTER/DOWN dst->output is set to BLACKHOLE.

net/core/rtnetlink.c, line 526 rtnetlink_event()

```
     case NETDEV_REGISTER:
          rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);
          break;
```

net/ipv4/devinet.c, line 1140 inetdev_event()

```
     case NETDEV_REGISTER:
          printk(KERN_DEBUG "inetdev_event: bug\n");
          dev->ip_ptr = NULL;
          break;
```

net/ipv4/ipmr.c, line 1756 ipmr_device_event()

Multicast routing support via mrouted.

net/ipv4/fib_frontend.c, line 652 fib_netdev_event()

No action.

net/ipv4/fib_rules.c, line 466 fib_rules_event()

```
else if (event == NETDEV_REGISTER)
     fib_rules_attach(dev);
```

Recall that fib_rules aren't in play unless IP_MULTIPLE_TABLES is configured.

net/ipv4/netfilter/ip_queue.c, line 647 ipq_rcv_dev_event()

No action is taken on REGISTER. The packet queue is dumped on DOWN.

The functions rtnetlink_event() and inetdev_event() (among many others) are indirectly called. This whole gruesome collection is shown on the previous pages. Here we focus upon rtnetlink_event() when called with the parameter NETDEV_REGISTER.

```
static int rtnetlink_event(struct notifier_block *this,
                  unsigned long event, void *ptr)
{
    struct net_device *dev = ptr;
    switch (event) {
    case NETDEV_UNREGISTER:
         rtmsg_ifinfo(RTM_DELLINK, dev, ~0U);
         break;
    case NETDEV_REGISTER:
         rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);
         break;
    case NETDEV_UP:
    case NETDEV_DOWN:
         rtmsg_ifinfo(RTM_NEWLINK, dev,
                   IFF_UP|IFF_RUNNING);
         break;
    case NETDEV_CHANGE:
    case NETDEV_GOING_DOWN:
         break;
    default:
         rtmsg_ifinfo(RTM_NEWLINK, dev, 0);
         break;
    }
    return NOTIFY_DONE;
}
```

Construction of netlink messages

The rtmsg_ifinfo() function is called in response to register, unregister, interface up, and interface down events. For register and interface up the message type is RTM_NEWLINK. The change parameter is 0xffffffff.

```
void rtmsg_ifinfo(int type, struct net_device *dev,
                                 unsigned change)
{
        struct sk_buff *skb;
        int size = NLMSG_GOODSIZE;

        skb = alloc_skb(size, GFP_KERNEL);
        if (!skb)
             return;

        if (rtnetlink_fill_ifinfo(skb, dev,
                              type, 0, 0, change) < 0) {
             kfree_skb(skb);
             return;
        }
```

The updating of the control buffer appears to be establishing the target recipients of this message. The call to netlink_broadcast() actually effects the delivery.

```
        NETLINK_CB(skb).dst_groups = RTMGRP_LINK;
        netlink_broadcast(rtnl, skb, 0, RTMGRP_LINK, GFP_KERNEL);
}
```

Netlink message headers

Each of thes messages begins with a header of the following layout. In the present context the type is always RTM_NEWLINK.

```
struct nlmsghdr
{
        __u32  nlmsg_len;       /* Len of msg including hdr */
        __u16  nlmsg_type;      /* Message content */
        __u16  nlmsg_flags;     /* Additional flags */
        __u32  nlmsg_seq;       /* Sequence number */
        __u32  nlmsg_pid;       /* Sending process PID */
};
```

For messages of the interface information class, a fixed structure follows the netlink header.

```
struct ifinfomsg
{
        unsigned char   ifi_family;
        unsigned char   __ifi_pad;
        unsigned short  ifi_type;       /* ARPHRD_* */
        int             ifi_index;      /* Link index   */
        unsigned        ifi_flags;      /* IFF_* flags  */
        unsigned        ifi_change;     /* IFF_* change mask */
};
```

```
static int rtnetlink_fill_ifinfo(struct sk_buff *skb,
                struct net_device *dev,
                int type, u32 pid, u32 seq, u32 change)
{
        struct ifinfomsg *r;
        struct nlmsghdr  *nlh;
        unsigned char *b = skb->tail;

```

The NLMSG_PUT macro builds the header. In this context pid which plays an important role in routing of these messages is 0.

```
        nlh = NLMSG_PUT(skb, pid, seq, type, sizeof(*r));
        if (pid) nlh->nlmsg_flags |= NLM_F_MULTI;
```

Fill in the interface information header.

```
    r = NLMSG_DATA(nlh);
    r->ifi_family = AF_UNSPEC;
    r->ifi_type = dev->type;
    r->ifi_index = dev->ifindex;
    r->ifi_flags = dev->flags;
    r->ifi_change = change;

    if (!netif_running(dev) || !netif_carrier_ok(dev))
         r->ifi_flags &= ~IFF_RUNNING;
    else
         r->ifi_flags |= IFF_RUNNING;


    RTA_PUT(skb, IFLA_IFNAME, strlen(dev->name)+1,
              dev->name);
    if (dev->addr_len) {
         RTA_PUT(skb, IFLA_ADDRESS, dev->addr_len,
              dev->dev_addr);
         RTA_PUT(skb, IFLA_BROADCAST, dev->addr_len,
                   dev->broadcast);
    }
    if (1) {
         unsigned mtu = dev->mtu;
         RTA_PUT(skb, IFLA_MTU, sizeof(mtu), &mtu);
    }
    if (dev->ifindex != dev->iflink)
         RTA_PUT(skb, IFLA_LINK, sizeof(int), &dev->iflink);
    if (dev->qdisc_sleeping)
         RTA_PUT(skb, IFLA_QDISC,
              strlen(dev->qdisc_sleeping->ops->id) + 1,
                        dev->qdisc_sleeping->ops->id);
    if (dev->master)
         RTA_PUT(skb, IFLA_MASTER, sizeof(int),
                             &dev->master->ifindex);
    if (dev->get_stats) {
         struct net_device_stats *stats =
                             dev->get_stats(dev);
         if (stats)
         RTA_PUT(skb, IFLA_STATS, sizeof(*stats), stats);
    }
    nlh->nlmsg_len = skb->tail - b;
    return skb->len;

    nlmsg_failure:
    rtattr_failure:
    skb_trim(skb, b - skb->data);
    return -1;
}
```

RTA_PUT is a macro used to add information elements to the message.

```
#define RTA_PUT(skb, attrtype, attrlen, data) \
({ if (skb_tailroom(skb) < (int)RTA_SPACE(attrlen)) goto
        rtattr_failure; \
       __rta_fill(skb, attrtype, attrlen, data); })
```

It relies upon a collection of related macros...

```
#define RTA_ALIGNTO     4
#define RTA_ALIGN(len) ( ((len)+RTA_ALIGNTO-1) &
        ~(RTA_ALIGNTO-1) )
#define RTA_OK(rta,len) ((len) > 0 && (rta)->rta_len >=
        sizeof(struct rtattr) && \
                             (rta)->rta_len <= (len))
#define RTA_NEXT(rta,attrlen)   ((attrlen) -=
        RTA_ALIGN((rta)->rta_len), \
                                     (struct
        rtattr*)(((char*)(rta)) + RTA_ALIGN((rta)->rta_len)))
#define RTA_LENGTH(len) (RTA_ALIGN(sizeof(struct rtattr)) +
        (len))
#define RTA_SPACE(len)  RTA_ALIGN(RTA_LENGTH(len))
#define RTA_DATA(rta)   ((void*)(((char*)(rta)) +
        RTA_LENGTH(0)))
#define RTA_PAYLOAD(rta) ((int)((rta)->rta_len) -
        RTA_LENGTH(0))
```

The RTA_PUT macro invokes the __rta_fill function to add the data to the packet.

```
void __rta_fill(struct sk_buff *skb, int attrtype, int
                  attrlen, const void *data)
{
        struct rtattr *rta;
        int size = RTA_LENGTH(attrlen);

```

Here is where the actual (T, L, V) data is stored in the message.

```
        rta = (struct rtattr*)skb_put(skb, RTA_ALIGN(size));
        rta->rta_type = attrtype;
        rta->rta_len = size;
        memcpy(RTA_DATA(rta), data, attrlen);
}
```

```
void netlink_broadcast(struct sock *ssk, struct sk_buff
    *skb, u32 pid, u32 group, int allocation)
{
    struct sock *sk;
    struct sk_buff *skb2 = NULL;
    int protocol = ssk->protocol;
    int failure = 0;

/* While we sleep in clone, do not allow to change socket
    list */

    netlink_lock_table();


    for (sk = nl_table[protocol]; sk; sk = sk->next) {
         if (ssk == sk)
              continue;

         if (sk->protinfo.af_netlink->pid == pid ||
              !(sk->protinfo.af_netlink->groups&group))
              continue;

         if (failure) {
              netlink_overrun(sk);
              continue;
         }

         sock_hold(sk);
         if (skb2 == NULL) {
              if (atomic_read(&skb->users) != 1) {
                   skb2 = skb_clone(skb, allocation);
              } else {
                   skb2 = skb;
                   atomic_inc(&skb->users);
              }
         }
         if (skb2 == NULL) {
              netlink_overrun(sk)
/* Clone failed. Notify ALL listeners. */
              failure = 1;
         } else if (netlink_broadcast_deliver(sk, skb2)) {
              netlink_overrun(sk);
         } else
              skb2 = NULL;
         sock_put(sk);
    }

    netlink_unlock_table();

    if (skb2)
    kfree_skb(skb2);
    kfree_skb(skb);
}
```

```
static __inline__ int netlink_broadcast_deliver(struct sock
         *sk, struct sk_buff
{
```

This is not normally configured in kernels I build. Here is the description from make xconfig:

> This option will be removed soon. Any programs that want to use character special nodes like /dev/tap0 or /dev/route (all with major number 36) need this option, and need to be rewritten soon to use the real netlink socket.

```
#ifdef NL_EMULATE_DEV
    if (sk->protinfo.af_netlink->handler) {
         skb_orphan(skb);
         sk->protinfo.af_netlink->handler(sk->protocol, skb);
         return 0;
    } else
#endif
    if (atomic_read(&sk->rmem_alloc) <= sk->rcvbuf &&
         !test_bit(0, &sk->protinfo.af_netlink->state)) {
         skb_orphan(skb);
         skb_set_owner_r(skb, sk);
         skb_queue_tail(&sk->receive_queue, skb);
         sk->data_ready(sk, skb->len);
         return 0;
    }
    return -1;
}
```

This code in net/ipv4/fib_frontend.c contains the glue that eventually causes fib_magic to be called. The resulting call causes the local and main tables to be updated.

```
struct notifier_block fib_inetaddr_notifier = {
        notifier_call:  fib_inetaddr_event,
};

struct notifier_block fib_netdev_notifier = {
        notifier_call:  fib_netdev_event,
};

void __init ip_fib_init(void)
{
#ifdef CONFIG_PROC_FS
        proc_net_create("route",0,fib_get_procinfo);
#endif          /* CONFIG_PROC_FS */

#ifndef CONFIG_IP_MULTIPLE_TABLES
        local_table = fib_hash_init(RT_TABLE_LOCAL);
        main_table = fib_hash_init(RT_TABLE_MAIN);
#else
        fib_rules_init();
#endif

        register_netdevice_notifier(&fib_netdev_notifier);
        register_inetaddr_notifier(&fib_inetaddr_notifier);
}
```

For the purposes of FIB (routing table) management , two important notifiers reside in net/ipv4/fib_frontend.c. These notifiers were registered during IP initialization at boot time.

```
struct notifier_block fib_inetaddr_notifier = {
    notifier_call:fib_inetaddr_event,
};

struct notifier_block fib_netdev_notifier = {
    notifier_call:fib_netdev_event,
};

void __init ip_fib_init(void)
{
#ifdef CONFIG_PROC_FS
    proc_net_create("route",0,fib_get_procinfo);
#endif/* CONFIG_PROC_FS */

#ifndef CONFIG_IP_MULTIPLE_TABLES
    local_table = fib_hash_init(RT_TABLE_LOCAL);
    main_table = fib_hash_init(RT_TABLE_MAIN);
#else
    fib_rules_init();
#endif

    register_netdevice_notifier(&fib_netdev_notifier);
    register_inetaddr_notifier(&fib_inetaddr_notifier);
}

```

```
struct notifier_block
{
    int (*notifier_call)(struct notifier_block *self, unsigned
          long, void *);
    struct notifier_block *next;
    int priority;
};
```

```
int register_netdevice_notifier(struct notifier_block *nb)
{
    return notifier_chain_register(&netdev_chain, nb);
}
```

```
static int fib_inetaddr_event(struct notifier_block *this,
         unsigned long event, v
{
    struct in_ifaddr *ifa = (struct in_ifaddr*)ptr;

    switch (event) {
    case NETDEV_UP:
         fib_add_ifaddr(ifa);
         rt_cache_flush(-1);
         break;

    case NETDEV_DOWN:
         fib_del_ifaddr(ifa);
         if (ifa->ifa_dev && ifa->ifa_dev->ifa_list == NULL){
         /* Last address was deleted from this interface.
            Disable IP.
         */
              fib_disable_ip(ifa->ifa_dev->dev, 1);
         } else {
              rt_cache_flush(-1);
         }
         break;
    }
    return NOTIFY_DONE;
}

static void fib_add_ifaddr(struct in_ifaddr *ifa)
{
    struct in_device *in_dev = ifa->ifa_dev;
    struct net_device *dev = in_dev->dev;
    struct in_ifaddr *prim = ifa;
    u32 mask = ifa->ifa_mask;
    u32 addr = ifa->ifa_local;
    u32 prefix = ifa->ifa_address&mask;

    if (ifa->ifa_flags&IFA_F_SECONDARY) {
         prim = inet_ifa_byprefix(in_dev, prefix, mask);
         if (prim == NULL) {
              printk(KERN_DEBUG "fib_add_ifaddr: bug: prim ==
                                    NULL\n");
              return;
         }
    }

    fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);

    if (!(dev->flags&IFF_UP))
         return;

```

```
/* Add broadcast address, if it is explicitly assigned. */
    if (ifa->ifa_broadcast && ifa->ifa_broadcast !=
              0xFFFFFFFF)
         fib_magic(RTM_NEWROUTE, RTN_BROADCAST,
              ifa->ifa_broadcast, 32, prim);

    if (!ZERONET(prefix) &&
              !(ifa->ifa_flags&IFA_F_SECONDARY) &&
              (prefix != addr || ifa->ifa_prefixlen < 32)) {
         fib_magic(RTM_NEWROUTE, dev->flags&IFF_LOOPBACK ?
                   RTN_LOCAL : RTN_UNICAST, prefix,
                   ifa->ifa_prefixlen, prim);

/* Add network specific broadcasts, when it takes a sense */
         if (ifa->ifa_prefixlen < 31) {
              fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix,
                        32, prim);
              fib_magic(RTM_NEWROUTE, RTN_BROADCAST,
                        prefix|~mask, 32, prim);
         }
    }
}
```

```
static void fib_magic(int cmd, int type, u32 dst, int
                             dst_len, struct in_ifaddr *ifa)
{
    struct fib_table * tb;
    struct {
         struct nlmsghdrnlh;
         struct rtmsgrtm;
    } req;
    struct kern_rta rta;

    memset(&req.rtm, 0, sizeof(req.rtm));
    memset(&rta, 0, sizeof(rta));

    if (type == RTN_UNICAST)
         tb = fib_new_table(RT_TABLE_MAIN);
    else
         tb = fib_new_table(RT_TABLE_LOCAL);

    if (tb == NULL)
         return;

    req.nlh.nlmsg_len = sizeof(req);
    req.nlh.nlmsg_type = cmd;
    req.nlh.nlmsg_flags =
         NLM_F_REQUEST|NLM_F_CREATE|NLM_F_APPEND;
    req.nlh.nlmsg_pid = 0;
    req.nlh.nlmsg_seq = 0;

    req.rtm.rtm_dst_len = dst_len;
    req.rtm.rtm_table = tb->tb_id;
    req.rtm.rtm_protocol = RTPROT_KERNEL;
    req.rtm.rtm_scope = (type != RTN_LOCAL ? RT_SCOPE_LINK :
         RT_SCOPE_HOST);
    req.rtm.rtm_type = type;

    rta.rta_dst = &dst;
    rta.rta_prefsrc = &ifa->ifa_local;
    rta.rta_oif = &ifa->ifa_dev->dev->ifindex;

    if (cmd == RTM_NEWROUTE)
         tb->tb_insert(tb, &req.rtm, &rta, &req.nlh, NULL);
    else
         tb->tb_delete(tb, &req.rtm, &rta, &req.nlh, NULL);
}
```