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 convenience 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 `ethn`. 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.

```c
210 struct net_device *init_etherdev(struct net_device *dev,
          int sizeof_priv)
211 { return init_netdev(dev, sizeof_priv, "eth%d",
                    ether_setup);
213 }

129 static struct net_device *init_netdev(struct net_device *dev,
130     int sizeof_priv, char *mask,
131     void (*setup)(struct net_device *))
132 { int new_device = 0; /* flag that controls registration */
133  /*
134   *Allocate a device if one is not provided.
135   */
136  if (dev == NULL) {
137      dev=init_alloc_dev(sizeof_priv);
138      if(dev==NULL)
139          return NULL;
140      new_device = 1;
143  }
144  ```
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*.

```c
145 /*
146  *Allocate a name
147  */
148
149 if (dev->name[0] == '0' || dev->name[0] == ') { 
150     strcpy(dev->name, mask);
151     if (dev_alloc_name(dev, mask)<0) {
152         if (new_device)
153             kfree(dev);
154         return NULL;
155     }
156 }
157
158 }
```
On return to \texttt{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.

\begin{verbatim}
158   netdev_boot_setup_check(dev);
159

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

\begin{verbatim}
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, \texttt{init_netdev()} attempts to register the device with the protocols.

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

The struct net_device is the root of a collection of data structures used to manage an IP capable device.

```c
235 struct net_device
236 {
237 #238 /*
239 * This is the start of the "visible" part of this structure
240 * (i.e. as seen by users in the "Space.c" file). It is the
241 * name of the interface.
242 */
243     char  name[IFNAMSIZ];  /* e.g. eth0 */
244 */
245 /* I/O specific fields
246 * FIXME: Merge these and struct ifmap into one
247 */
248 */
249     unsigned long rmem_end;  /* shmem "recv" end */
250     unsigned long rmem_start; /* shmem "recv" start */
251     unsigned long mem_end;    /* shared mem end */
252     unsigned long mem_start;  /* shared mem start */
253     unsigned long base_addr; /* device I/O address */
254     unsigned int  irq;       /* device IRQ number */
255 */
256 /* Some hardware also needs these fields, but they are not
257 * part of the usual set specified in Space.c.
258 */
259 */
260     unsigned char if_port;  /* Selectable AUI, TP,..*/
261     unsigned char dma;      /* DMA channel */
262 */
263     unsigned long state;
264     struct net_device *next;
265 */
266 /* The device initialization function. Called only once. */
267     int      (*init)(struct net_device *dev);
268 */
269 /* ------- Fields preinitialized in Space.c finish here
270 * ------- */
271 *next_sched;
272 */ Interface index. Unique device identifier */
273     int  ifindex;
274     int  iflink;
275 ```
struct net_device_stats*
    (*get_stats)(struct net_device *dev);
struct iw_statistics*
    (*get_wireless_stats)(struct net_device *dev);

/* List of functions to handle Wireless Extensions
   (instead of ioctl).
   * See <net/iw_handler.h> for details. Jean II */
struct iw_handler_def * wireless_handlers;

/* These may be needed for future network-power-down code. */
unsigned long trans_start; /* Time of last Tx */
unsigned long last_rx; /* Time of last Rx */
unsigned short flags; /* interface flags (a la BSD) */
unsigned short qflags;
unsigned short priv_flags; /* Like 'flags' but
                               invisible to userspace. */
unsigned short unused_alignment_fixer;

unsigned mtu; /* interface MTU value */
unsigned short type; /* interface hardware type */
unsigned short hard_header_len; /* hardware hdr len */
void *priv; /* pointer to private data */

struct net_device *master; /* Pointer to master of a group,
                              * which this device is member of. */

/* Interface address info. */
unsigned char broadcast[MAX_ADDR_LEN]; /* hw bcast add */
unsigned char dev_addr[MAX_ADDR_LEN]; /* hw address */
unsigned char addr_len; /* hardware address length */
struct dev_mc_list *mc_list; /* Multicast mac addrs */
int mc_count; /* Number of installed mcasts */
int promiscuity;
int allmulti;

int watchdog_timeo;
struct timer_list watchdog_timer;
/* Protocol specific pointers */

void     *atalk_ptr;     /* AppleTalk link */
void     *ip_ptr;        /* IPv4 specific data */
void     *dn_ptr;        /* DECnet specific data */
void     *ip6_ptr;       /* IPv6 specific data */
void     *ec_ptr;        /* Econet specific data */

struct list_head poll_list; /* Link to poll list */
int       quota;
int       weight;

struct Qdisc  *qdisc;
struct Qdisc *qdisc_sleeping;
struct Qdisc  *qdisc_list;
struct Qdisc  *qdisc_ingress;
unsigned long tx_queue_len; /* Max frames per queue*/

/* hard_start_xmit synchronizer */

spinlock_t              xmit_lock;
/* cpu id of processor entered to hard_start_xmit or -1,
if nobody entered there. */
int  xmit_lock_owner;
/* device queue lock */
spinlock_t queue_lock;
/* Number of references to this device */
atomic_t refcnt;
/* Flag marking device unregistered, but held by user */
int    deadbeaf;

/* Net device features */

int features;
#define NETIF_F_SG        1 /* Scatter/gather IO. */
#define NETIF_F_IP_CSUM   2 /* Can checksum only TCP/UDP over IPv4. */
#define NETIF_F_NO_CSUM   4 /* Does not require checksum. */
#define NETIF_F_HW_CSUM   8 /* Can checksum all packets. */
#define NETIF_F_DYNALLOC  16 /* Self-destructable device. */
#define NETIF_F_HIGHDMA  32 /* Can DMA to high memory. */
#define NETIF_F_FRAGLIST  64 /* Scatter/gather IO. */
#define NETIF_F_HW_VLAN_TX 128 /* Transmit VLAN hw accel */
#define NETIF_F_HW_VLAN_RX 256 /* Rx VLAN hw acceleration */
#define NETIF_F_HW_VLAN_FILTER  512 /* Rx filtering on VLAN */
#define NETIF_F_VLAN_CHALLENGED 1024/* Device cannot handle VLAN packets */
/* Called after device is detached from network. */
void (*uninit)(struct net_device *dev);

/* Called after last user reference disappears. */
void (*destructor)(struct net_device *dev);

/* Pointers to interface service routines. */

int (*open)(struct net_device *dev);
int (*stop)(struct net_device *dev);
int (*hard_start_xmit)(struct sk_buff *skb,
struct net_device *dev);

#define HAVE_NETDEV_POLL
int (*poll)(struct net_device *dev, int *quota);
int (*hard_header)(struct sk_buff *skb,
struct net_device *dev,
unsigned short type,
void *daddr,
void *saddr,
unsigned len);
int (*rebuild_header)(struct sk_buff *skb);

#define HAVE_MULTICAST
void (*set_multicast_list)(struct net_device *dev);

#define HAVE_SET_MAC_ADDR
int (*set_mac_address)(struct net_device *dev,
void *addr);

#define HAVE_PRIVATE_IOCTL
int (*do_ioctl)(struct net_device *dev,
struct ifreq *ifr, int cmd);

#define HAVE_SET_CONFIG
int (*set_config)(struct net_device *dev,
struct ifmap *map);

#define HAVE_HEADER_CACHE
int (*hard_header_cache)(struct neighbour *neigh,
struct hh_cache *hh);
void (*header_cache_update)(struct hh_cache *hh,
struct net_device *dev,
unsigned char *haddr);

#define HAVE_CHANGE_MTU
int (*change_mtu)(struct net_device *dev, int new_mtu);
```c
#define HAVE_TX_TIMEOUT

void (*tx_timeout) (struct net_device *dev);

void (*vlan_rx_register)(struct net_device *dev,
                        struct vlan_group *grp);
void (*vlan_rx_add_vid)(struct net_device *dev,
                        unsigned short vid);
void (*vlan_rx_kill_vid)(struct net_device *dev,
                        unsigned short vid);

int (*hard_header_parse)(struct sk_buff *skb,
                         unsigned char *haddr);

int (*neigh_setup)(struct net_device *dev, struct
                   neigh_parms *);

int (*accept_fastpath)(struct net_device *, struct
                       dst_entry*);

/* open/release and usage marking */
struct module *owner;

/* bridge stuff */
struct net_bridge_port *br_port;

#ifdef CONFIG_NET_FASTROUTE
#define NETDEV_FASTROUTE_HMASK 0xF
/* Semi-private data. Keep it at the end of device struct. */
rwlock_t fastpath_lock;
struct dst_entry *fastpath[NETDEV_FASTROUTE_HMASK+1];
#endif

#ifdef CONFIG_NET_DIVERT
/* this will get initialized at each interface type init
   routine */
struct divert_blk *divert;
#endif
#endif /* CONFIG_NET_DIVERT */
};
```
**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.

```c
101 static struct net_device *init_alloc_dev(int sizeof_priv)
102 {
103     struct net_device *dev;
104     int alloc_size;
105
106     /* ensure 32-byte alignment of the private area */
107     alloc_size = sizeof (*dev) + sizeof_priv + 31;
108
109     dev = (struct net_device *) kmalloc (alloc_size, GFP_KERNEL);
110     if (dev == NULL)
111         { printk(KERN_ERR "alloc_dev: Unable to allocate device memory.\n"); return NULL; }
112
113     memset(dev, 0, alloc_size);
114
115     if (sizeof_priv)
116         { dev->priv = (void *) (((long)(dev + 1) + 31) & ~31); }
117     return dev;
118 }
```

Initialize the `net_device` structure and link in the private area if it was provided.
The \texttt{dev_alloc_name()} function

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

```c
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;

  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! */
}
```

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

411 struct net_device *__dev_get_by_name(const char *name) 
412 { 
413     struct net_device *dev;
414
415     for (dev = dev_base; dev != NULL; dev = dev->next) { 
416         if (strncmp(dev->name, name, IFNAMSIZ) == 0) 
417             return dev;
418     }
419     return NULL;
420 }
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`.

```c
405 void ether_setup(struct net_device *dev) 
406  { 
407      /* Fill in the fields of the device structure with ethernet-generic values. */
408      /* This should be in a common file instead of per-driver. */
409
410      dev->change_mtu = eth_change_mtu; 
411      dev->hard_header = eth_header; 
412      dev->rebuild_header = eth_rebuild_header; 
413      dev->set_mac_address = eth_mac_addr; 
414      dev->hard_header_cache = eth_header_cache; 
415      dev->header_cache_update = eth_header_cache_update; 
416      dev->hard_header_parse = eth_header_parse; 
417
418      dev->type= ARPHRD_ETHER; 
419      dev->hard_header_len = ETH_HLEN; 
420      dev->mtu= 1500; /* eth_mtu */
421      dev->addr_len= ETH_ALEN; 
422      dev->tx_queue_len= 100; /* Ethernet wants good queues */
423
424      memset(dev->broadcast,0xFF, ETH_ALEN); 
425
426   /* New-style flags. */
427      dev->flags= IFF_BROADCAST|IFF_MULTICAST; 
428  }
```

Indicate that the interface supports both hardware broadcast and multicast.

```c
426  /* New-style flags. */
427      dev->flags= IFF_BROADCAST|IFF_MULTICAST; 
428  }
```
Registering a `net_device`

We shall see that the call to `register_netdevice()` will trigger indirect 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.

```c
2488 int register_netdevice(struct net_device *dev)  
2489 {  
2490     struct net_device *d, **dp;  
2491     #ifdef CONFIG_NET_DIVERT  
2492         int ret;  
2493     #endif  
2494     spin_lock_init(&dev->queue_lock);  
2495     spin_lock_init(&dev->xmit_lock);  
2496     dev->xmit_lock_owner = -1;  
2497     #ifdef CONFIG_NET_FASTROUTE  
2498         dev->fastpath_lock = RW_LOCK_UNLOCKED;  
2499     #endif  
2500     if (dev_boot_phase)  
2501         net_dev_init();
```

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.

```c
2502     if (dev_boot_phase)  
2503         net_dev_init();
2504```

It would be nice to understand diverters.

```c
2505     #ifdef CONFIG_NET_DIVERT  
2506         ret = alloc_divert_blk(dev);  
2507     if (ret)  
2508         return ret;  
2509     #endif  /* CONFIG_NET_DIVERT */  
2510```
The iflink element is an alternative identifying index that can be set by the device driver. It is initialized to -1 before calling the driver's 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.

2511   dev->iflink = -1;
2512
2513 /* Init, if this function is available */
2514 if (dev->init && dev->init(dev) != 0) {
2515   #ifdef CONFIG_NET_DIVERT
2516     free_divert_blk(dev);
2517   #endif
2518   return -EIO;
2519 }
2520
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.

2521   dev->ifindex = dev_new_index();
2522   if (dev->iflink == -1)
2523     dev->iflink = dev->ifindex;
2524
2525 /* Check for existence, and append to tail of chain */
2526 for (dp=&dev_base; (d=*dp) != NULL; dp=&d->next) {
2527   if (d == dev || strcmp(d->name, dev->name) == 0) {
2528     #ifdef CONFIG_NET_DIVERT
2529       free_divert_blk(dev);
2530     #endif
2531     return -EEXIST;
2532   }
2533 }
2534 /*
2535  *nil rebuild_header routine,
2536  *that should be never called and used as just bug trap.
2537 */
2538
2539   if (dev->rebuild_header == NULL)
2540     dev->rebuild_header = default_rebuild_header;
2542 /*
2543 * Default initial state at registry is that the
2544 * device is present.
2545 */
2546
2547    set_bit(__LINK_STATE_PRESENT, &dev->state);
2548
2549    dev->next = NULL;
2550    dev_init_scheduler(dev);
2551    write_lock_bh(&dev_base_lock);
2552    *dp = dev;
2553    dev_hold(dev);
2554    dev->deadbeaf = 0;
2555    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.

2557 /* Notify protocols, that a new device appeared. */
2558    notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
2559    net_run_sbin_hotplug(dev, "register");
2560    return 0;
2561 }

The device index allocation algorithm

2432 int dev_new_index(void)
2433 {
2434     static int ifindex;
2435     for (;;) {
2436         if (++ifindex <= 0)
2437             ifindex=1;
2438         if (__dev_get_by_index(ifindex) == NULL)
2439             return ifindex;
2440     }
2441 }

15
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()`.

```c
486 void dev_init_scheduler(struct net_device *dev) {
487     write_lock(&qdisc_tree_lock);
488     spin_lock_bh(&dev->queue_lock);
489     dev->qdisc = &noop_qdisc;
490     spin_unlock_bh(&dev->queue_lock);
491     dev->qdisc_sleeping = &noop_qdisc;
492     dev->qdisc_list = NULL;
493     write_unlock(&qdisc_tree_lock);
494     dev_watchdog_init(dev);
495 }
496
497
498```
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 occurrence 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.

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

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.

```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.

```c
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.

```c
#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 in 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.

```c
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/netdymos.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
- net/core/dst.c, line 214
- net/core/rtnetlink.c, line 526
- net/ipv4/devinet.c, line 1140
- net/ipv4/ipmr.c, line 1756
- net/ipv4/fib_frontend.c, line 652
- net/ipv4/fib_rules.c, line 466
- net/ipv4/netfilter/ip_queue.c, line 647
- net/ipv4/netfilter/ipfwadm_core.c, line 1385
- net/ipv4/netfilter/ipt_MASQUERADE.c, line 190
- net/px/af_ipx.c, line 2562
- net/ntrom/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/rd/afr_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()

494    case NETDEV_REGISTER:
495        rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);
496    break;

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

802    case NETDEV_REGISTER:
803        printk(KERN_DEBUG "inetdev_event: bug\n");
804        dev->ip_ptr = NULL;
805        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()

388    else if (event == NETDEV_REGISTER)
389        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.
Construction of netlink messages

Recall that the \texttt{rtnetlink\_event()} function invoked the \texttt{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 \texttt{RTM\_NEWLINK}. The \texttt{change} parameter is \texttt{0xffffffff}.

\begin{verbatim}
 247 void rtmsg_ifinfo(int type, struct net_device *dev,
                unsigned change)
 248 {
 249     struct sk_buff *skb;
 250     int size = NLMSG_GOODSIZE;
 251
 252     skb = alloc_skb(size, GFP_KERNEL);
 253     if (!skb)
 254         return;
 255
 256     if (rtnetlink_fill_ifinfo(skb, dev,
                type, 0, 0, change) < 0) {
 257         kfree_skb(skb);
 258         return;
 259 }
\end{verbatim}

The updating of the control buffer appears to be establishing the target recipients of this message. The call to \texttt{netlink\_broadcast()} actually effects the delivery.

\begin{verbatim}
 260     NETLINK_CB(skb).dst_groups = RTMGRP_LINK;
 261     netlink_broadcast(rtnl, skb, 0, RTMGRP_LINK, GFP_KERNEL);
 262 }
\end{verbatim}
Netlink message headers

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

```c
26 struct nlmsghdr
27 {
28     __u32 nlmsg_len;    /* Len of msg including hdr */
29     __u16 nlmsg_type;  /* Message content */
30     __u16 nlmsg_flags; /* Additional flags */
31     __u32 nlmsg_seq;   /* Sequence number */
32     __u32 nlmsg_pid;   /* Sending process PID */
33 };
```

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

```c
419 struct ifinfomsg
420 {
421     unsigned char ifi_family;
422     unsigned char __ifi_pad;
423     unsigned short ifi_type;  /* ARPHRD_* */
424     int ifi_index;  /* Link index */
425     unsigned ifi_flags; /* IFF_* flags */
426     unsigned ifi_change; /* IFF_* change mask */
427 };
```

```c
152 static int rtnetlink_fill_ifinfo(struct sk_buff *skb,
153                                       struct net_device *dev,
154                                       int type, u32 pid, u32 seq, u32 change)
155 {
156     struct ifinfomsg *r;
157     struct nlmsghdr *nlh;
158     unsigned char *b = skb->tail;
159     nlh = NLMSG_PUT(skb, pid, seq, type, sizeof(*r));
160     if (pid) nlh->nlmsg_flags |= NLM_F_MULTI;
```

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

```c
161 r = NLMSG_DATA(nlh);
162 r->ifi_family = AF_UNSPEC;
163 r->ifi_type = dev->type;
164 r->ifi_index = dev->ifindex;
165 r->ifi_flags = dev->flags;
166 r->ifi_change = change;
167
168 if (!netif_running(dev) || !netif_carrier_ok(dev))
169       r->ifi_flags &= ~IFF_RUNNING;
170 else
171       r->ifi_flags |= IFF_RUNNING;
172
173 RTA_PUT(skb, IFLA_IFNAME, strlen(dev->name)+1,
174          dev->name);
175 if (dev->addr_len) {
176     RTA_PUT(skb, IFLA_ADDRESS, dev->addr_len,
177              dev->dev_addr);
178     RTA_PUT(skb, IFLA_BROADCAST, dev->addr_len,
179              dev->broadcast);
180 }
181 if (1) {
182     unsigned mtu = dev->mtu;
183     RTA_PUT(skb, IFLA_MTU, sizeof(mtu), &mtu);
184 }
185 if (dev->ifindex != dev->iflink)
186     RTA_PUT(skb, IFLA_LINK, sizeof(int), &dev->iflink);
187 if (dev->qdisc_sleeping)
188     RTA_PUT(skb, IFLA_QDISC,
189              strlen(dev->qdisc_sleeping->ops->id) + 1,
190              dev->qdisc_sleeping->ops->id);
191 if (dev->master)
192     RTA_PUT(skb, IFLA_MASTER, sizeof(int),
193               &dev->master->ifindex);
194 if (dev->get_stats) {
195     struct net_device_stats *stats =
196         dev->get_stats(dev);
197     if (stats)
198         RTA_PUT(skb, IFLA_STATS, sizeof(*stats), stats);
199 }
200 nlh->nlmsg_len = skb->tail - b;
201 return skb->len;
202 }
```

24
$RTA\_PUT$ is a macro used to add information elements to the message.

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

It relies upon a collection of related macros...

```c
#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.

```c
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.

```c
  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);
            }
        } else if (netlink_broadcast_deliver(sk, skb2)) {
            netlink_overrun(sk);
        } else
            skb2 = NULL;
        sock_put(sk);
    }

    netlink_unlock_table();
    if (skb2)
        kfree_skb(skb2);
}
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.

```c
#define 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;
} else
    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.

```c
631 struct notifier_block fib_inetaddr_notifier = {
632     notifier_call: fib_inetaddr_event,
633 };  
634
635 struct notifier_block fib_netdev_notifier = {
636     notifier_call: fib_netdev_event,
637 };  
638
639 void __init ip_fib_init(void)  
640 {
641 #ifdef CONFIG_PROC_FS
642     proc_net_create("route",0,fib_get_procinfo);  
643 #endif /* CONFIG_PROC_FS */
644
645 #ifndef CONFIG_IP_MULTIPLE_TABLES
646     local_table = fib_hash_init(RT_TABLE_LOCAL);  
647     main_table = fib_hash_init(RT_TABLE_MAIN);  
648 #else
649     fib_rules_init();  
650 #endif
651
652     register_netdevice_notifier(&fib_netdev_notifier);  
653     register_inetaddr_notifier(&fib_inetaddr_notifier);  
654 }
655
```