

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.

```
210 struct net_device *init_etherdev(struct net_device *dev,
211                                     int sizeof_priv)
212 {
213     return init_netdev(dev, sizeof_priv, "eth%d",
214                         ether_setup);
215 }
216
217 static struct net_device *init_netdev(struct net_device *dev,
218                                       int sizeof_priv, char *mask,
219                                       void (*setup)(struct net_device *))
220 {
221     int new_device = 0;
222
223     /*
224      *Allocate a device if one is not provided.
225     */
226
227     if (dev == NULL) {
228         dev=init_alloc_dev(sizeof_priv);
229         if(dev==NULL)
230             return NULL;
231         new_device = 1;
232     }
233
234 }
```

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*.

```
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 }
```

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         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 }
```

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.

```
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,
110                                         GFP_KERNEL);
111     if (dev == NULL)
112     {
113         printk(KERN_ERR "alloc_dev: Unable to allocate
114               device memory.\n");
115         return NULL;
116     }
117 }
```

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

```
116     memset(dev, 0, alloc_size);
117
118     if (sizeof_priv)
119         dev->priv = (void *) (((long)(dev + 1) + 31) & ~31);
120
121     return dev;
122 }
123 }
```

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.

```

560 int dev_alloc_name(struct net_device *dev, const char *name)
561 {
562     int i;
563     char buf[32];
564     char *p;
565
566 /*
567  * Verify the string as this thing may have come from
568  * the user. There must be either one "%d" and no other "%
569  * characters, or no "%" characters at all.
570 */
571     p = strchr(name, '%');
572     if (p && (p[1] != 'd' || strchr(p+2, '%')))
573         return -EINVAL;
574

```

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.

```

575 /*
576  * If you need over 100 please also fix the algorithm...
577 */
578     for (i = 0; i < 100; i++) {
579         snprintf(buf, sizeof(buf), name, i);
580         if (__dev_get_by_name(buf) == NULL) {
581             strcpy(dev->name, buf);
582             return i;
583         }
584     }
585     return -ENFILE; /* Over 100 of the things... bail out! */
586 }
587
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 (strcmp(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

```
405 void ether_setup(struct net_device *dev)
406 {
407 /* Fill in the fields of the device structure with ethernet-
408 generic values.
409 This should be in a common file instead of per-driver. */
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
```

Indicate that the interface supports both hardware broadcast and multicast.

```
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.

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

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.

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

It would be nice to understand diverters.

```
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 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.

```

2511     dev->iflink = -1;
2512
2513 /* Init, if this function is available */
2514 if (dev->init && dev->init(dev) != 0) {
2515 #ifdef CONFIG_NET_DIVER
2516     free_device_link(dev);
2517 #endif
2518     return -EINVAL;
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->index = dev_new_index();
2522     if (dev->iflink == -1)
2523         dev->iflink = dev->index;
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_DIVER
2529         free_device_link(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 if (dev->rebuild_header == NULL)
2539     dev->rebuild_header = default_rebuild_header;
2541

```

```

2542 /*
2543  *Default initial state at registry is that the
2544  *device is present.
2545 */
2546     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);
2556

```

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
2560     net_run_sbini_hotplug(dev, "register");
2561
2562     return 0;
2563 }

```

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().

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

```
14 struct notifier_block
15 {
16     int (*notifier_call)(struct notifier_block *self,
17                          unsigned long, void *);
18     struct notifier_block *next;
19     int priority;
20 };

21
22 static struct notifier_block *netdev_chain=NULL;
23
```

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.

```
850 int register_netdevice_notifier(struct notifier_block *nb)
851 {
852     return notifier_chain_register(&netdev_chain, nb);
853 }

181
182 static struct notifier_block *netdev_chain=NULL;
183
```

Adding the notifier_block to the chain.

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

```
63
64 int notifier_chain_register(struct notifier_block **list,
65     struct notifier_block *n)
66 {
67     write_lock(&notifier_lock);
68     while(*list)
69     {
70         if(n->priority > (*list)->priority)
71             break;
72         list= &((*list)->next);
73     }
74     n->next = *list;
75     *list=n;
76     write_unlock(&notifier_lock);
77 }
```

Here are the notifiers associated with net_device events.

```
41 /* netdevice notifier chain */
42 #define NETDEV_UP          0x0001
43 /* For now you can't veto a device up/down */
44 #define NETDEV_DOWN         0x0002
45 #define NETDEV_REBOOT        0x0003
46 /* Tell a protocol stack a network interface
47 detected a hardware crash and restarted
48 - we can use this e.g. to kick tcp sessions
49 once done */
50 #define NETDEV_CHANGE        0x0004
51 /* Notify devstate change */
52 #define NETDEV_REGISTER       0x0005
53 #define NETDEV_UNREGISTER      0x0006
54 #define NETDEV_CHANGEMTU      0x0007
55 #define NETDEV_CHANGEADDR      0x0008
56 #define NETDEV_GOING_DOWN      0x0009
57 #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.

```
2557 /* Notify protocols, that a new device appeared. */
2558     notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
```

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

```
487 static int rtnealink_event(struct notifier_block *this,
488                             unsigned long event, void *ptr)
489 {
490     struct net_device *dev = ptr;
491     switch (event) {
492     case NETDEV_UNREGISTER:
493         rtmsg_info(RTM_DELLINK, dev, ~0U);
494         break;
495     case NETDEV_REGISTER:
496         rtmsg_info(RTM_NEWWLINK, dev, ~0U);
497         break;
498     case NETDEV_UP:
499     case NETDEV_DOWN:
500         rtmsg_info(RTM_NEWWLINK, dev,
501                    IFF_UP|IFF_RUNNING);
502         break;
503     case NETDEV_CHANGE:
504     case NETDEV_GOING_DOWN:
505         rtmsg_info(RTM_NEWWLINK, dev, 0);
506         break;
507     }
508     return NOTIFY_DONE;
509 }
```

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	register_netdevice_notifier
net/netsyms.c, line 465	dst_dev_event()
net/appletalk/aarp.c, line 859	rtnetlink_dev_notifier()
net/appletalk/ddp.c, line 1974	ip_netdev_notifier()
net/ax25/af_ax25.c, line 1851	ip_mr_notifier()
net/core/dev.c, line 850	fib_netdev_notifier()
net/core/dst.c, line 214	fib_rules_notifier()
net/core/rtnetlink.c, line 526	ipq_dev_notifier()
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/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()
494 case NETDEV_REGISTER: 495 rtmsg_ifinfo(RTM_NEWTLINK, dev, ~OU); 496 break;	
net/ipv4/devinet.c, line 1140	inetdev_event()
802 case NETDEV_REGISTER: 803 prntk(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.	

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`.

```
487 static int rtnetlink_event(struct notifier_block *this,
488                             unsigned long event, void *ptr)
489 {
490     struct net_device *dev = ptr;
491     switch (event) {
492     case NETDEV_UNREGISTER:
493         rtmsg_info(RTM_DELLINK, dev, ~0U);
494         break;
495     case NETDEV_REGISTER:
496         rtmsg_info(RTM_NEWWLINK, dev, ~0U);
497         break;
498     case NETDEV_UP:
499     case NETDEV_DOWN:
500         rtmsg_info(RTM_NEWWLINK, dev,
501                    IFF_UP|IFF_RUNNING);
502         break;
503     case NETDEV_CHANGE:
504     case NETDEV_GOING_DOWN:
505         rtmsg_info(RTM_NEWWLINK, dev, 0);
506         break;
507     default:
508     }
509     return NOTIFY_DONE;
510 }
```

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_NEWINLINK`. The change parameter is `0xffffffff`.

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

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.

```
260     NETLINK_CB(skb).dst_groups = RTMGRP_LINK;
261     netlink_broadcast(rtnl, skb, 0, RTMGRP_LINK, GFP_KERNEL);
262 }
```

Netlink message headers

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

```
26 struct nlmsghdr
27 {
28     __u32 nlmsg_len;          /* Len of msg incl uding hdr */
29     __u16 nlmsg_type;        /* Message content */
30     __u16 nlmsg_flags;       /* Additioinal 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.

```
419 struct iinfomsg
420 {
421     unsigned char iifi_familly;
422     unsigned char iifi_pad;
423     unsigned short iifi_type;           /* ARPHRD_* */
424     int iifi_index;                   /* Link index */
425     unsigned iifi_flags;             /* IFF_* flags */
426     unsigned iifi_change;            /* IFF_* change mask */
427 };
```



```
152 static int rtnetlink_ifinfo(struct sk_buff *skb,
153                                struct net_device *dev,
154                                int type, u32 pid, u32 seq, u32 change)
155 {
156     struct iinfomsg *r;
157     struct nlmsghdr *nlh;
158     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.

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

Fill in the interface information header.

```
161     r = NLMSG_DATA(nlh);
162     r->ifi_family = AF_UNSPEC;
163     r->ifi_type = dev->type;
164     r->ifi_index = dev->interface_index;
165     r->ifi_flags = dev->iflags;
166     r->ifi_change = change;
167
168     if (!netif_running(dev) || !netif_carrier_ok(dev))
169         r->iflags |= IFF_RUNNING;
170     else
171         r->iflags |= 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, si_sizeof(mtu), &mtu);
184     }
185     if (dev->interface != dev->link)
186         RTA_PUT(skb, IFLA_LINK, si_sizeof(int), &dev->link);
187     if (dev->qdisc_steepling)
188         RTA_PUT(skb, IFLA_QDISC,
189                 strlen(dev->qdisc_steepling->ops->id) + 1,
190                 dev->qdisc_steepling->ops->id);
191     if (dev->master)
192         RTA_PUT(skb, IFLA_MASTER, si_sizeof(int),
193                 &dev->master->interface);
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, si_sizeof(*stats), stats);
199     }
200     nlh->nmsg_len = skb->tai - b;
201     return skb->len;
202 }
```

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

```
564 #define RTA_PUT(skb, attrtype, attrlen, data) \
565 ({ if (skb_tail_room(skb) < (int)RTA_SPACE(attrlen)) goto \
566     rtattr_failure; \
567     __rta_fill(skb, attrtype, attrlen, data); })
```

It relies upon a collection of related macros...

```
64
65 #define RTA_ALIGNTO    4
66 #define RTA_ALIGN(len)  (((len)+RTA_ALIGNTO-1) &
67 ~RTA_ALIGNTO)
68 #define RTA_OK(rta, len) ((len) > 0 && (rta)->rta_len >=
69 sizeof(struct rtattr) && \
70 (rta)->rta_len <= (len))
71 #define RTA_NEXT(rta, attrlen) ((attrlen) -=
72 RTA_ALIGN((rta)->rta_len), \
73 (struct \
74 rtattr*)((char*)(rta) + RTA_ALIGN((rta)->rta_len)))
75 #define RTA_LENGTH(len)  (RTA_ALIGN(sizeof(struct rtattr)) +
76 (len))
77 #define RTA_SPACE(len)   RTA_ALIGN(RTA_LENGTH(len))
78 #define RTA_DATA(rta)   ((void*)((char*)(rta) + \
79 RTA_LENGTH(0)))
80 #define RTA_PAYLOAD(rta) ((int)((rta)->rta_len) - \
81 RTA_LENGTH(0))
```

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

```
107 void __rta_fill(struct sk_buff *skb, int attrtype, int
108 attrlen, const void *data)
109 {
110     struct rtattr *rta;
111     int size = RTA_LENGTH(attrlen);
112 }
```

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

```
112     rta = (struct rtattr*)skb_put(skb, RTA_ALIGN(size));
113     rta->rta_type = attrtype;
114     rta->rta_len = size;
115     memcpy(RTA_DATA(rta), data, attrlen);
116 }
```

```

478 void netlink_broadcast(struct sock *ssk, struct sk_buff
479     *skb, u32 pid, u32 group, int allocation)
480 {
481     struct sock *sk;
482     struct sk_buff *skb2 = NULL;
483     int protocol = ssk->protocol;
484     int failure = 0;
485
486 /* While we sleep in clone, do not allow to change socket
487    list */
488     netlink_lock_table();
489
490     for (sk = nl_table[protocol]; sk; sk = sk->next) {
491         if (ssk == sk)
492             continue;
493
494         if (sk->protoinfo.af_netlink->pid == pid ||
495             !(sk->protoinfo.af_netlink->groups&group))
496             continue;
497
498         if (failure) {
499             netlink_overrun(sk);
500             continue;
501         }
502
503         sock_hold(sk);
504         if (skb2 == NULL) {
505             if (atomic_c_read(&skb->users) != 1) {
506                 skb2 = skb_clone(skb, allocation);
507             } else {
508                 skb2 = skb;
509                 atomic_inc(&skb->users);
510             }
511         }
512         if (skb2 == NULL) {
513             netlink_overrun(sk)
514         /* Clone failed. Notify ALL listeners. */
515             failure = 1;
516         } else if (netlink_broadcast_deliver(sk, skb2)) {
517             netlink_overrun(sk);
518         } else
519             skb2 = NULL;
520         sock_put(sk);
521     }
522
523     netlink_unlock_table();
524
525     if (skb2)
526         kfree_skb(skb2);
527         kfree_skb(skb);
528 }

```

```

458 static __inline__ int netlink_broadcast_deliver(struct sock
        *sk, struct sk_buff
459 {

```

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.

```

460 #ifdef NL_EMULATE_DEV
461     if (sk->protoinfo.af_netlink->handler) {
462         skb_orphan(skb);
463         sk->protoinfo.af_netlink->handler(sk->protocol, skb);
464         return 0;
465     } else
466 #endif
467     if (atomic_c_read(&sk->rmem_alloc) <= sk->rcvbuf &&
468         !test_bit(0, &sk->protoinfo.af_netlink->state)) {
469         skb_orphan(skb);
470         skb_set_owner_r(skb, sk);
471         skb_queue_tail(&sk->receive_queue, skb);
472         sk->data_ready(sk, skb->len);
473         return 0;
474     }
475     return -1;
476 }

```

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.

```
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_proci_nfo);  
643 #endif /* CONFIG_PROC_FS */  
644  
645 #ifndef CONFIG_IP_MULTI_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 /* CONFIG_IP_MULTI_TABLES */  
651  
652     register_netdevice_notifier(&fib_netdev_notifier);  
653     register_inetaddr_notifier(&fib_inetaddr_notifier);  
654 }
```

655

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.

```

631 struct notifier_block_fib_i_netaddr_notifier = {
632     notifier_call : fib_i_netaddr_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_proci_nfo);
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_i_netaddr_notifier(&fib_i_netaddr_notifier);
654 }
655

14 struct notifier_block
15 {
16     int (*notifier_call)(struct notifier_block *self, unsigned
17                         long, void *);
17     struct notifier_block *next;
18     int priority;
19 };

866 int register_netdevice_notifier(struct notifier_block *nb)
867 {
868     return notifier_chain_register(&netdev_chain, nb);
869 }
```

```

575 static int fib_i netaddr_event(struct notifier_block *this,
576                                unsigned long event, v
577 {
578     struct in_ifaddr *ifa = (struct in_ifaddr*)ptr;
579     switch(event) {
580     case NETDEV_UP:
581         fib_add_ifaddr(ifa);
582         rt_cache_flush(-1);
583         break;
584     case NETDEV_DOWN:
585         fib_del_ifaddr(ifa);
586         if (ifa->iface && ifa->iface->iflist == NULL) {
587             /* Last address was deleted from this interface.
588              * Disable IP.
589             */
590             fib_desable_ip(ifa->iface->dev, 1);
591         } else {
592             rt_cache_flush(-1);
593         }
594         break;
595     }
596     return NOTIFY_DONE;
597 }
598
599 static void fib_add_ifaddr(struct in_ifaddr *ifa)
600 {
601     struct in_device *in_dev = ifa->iface;
602     struct net_device *dev = in_dev->dev;
603     struct in_ifaddr *prim = ifa;
604     u32 mask = ifa->if_mask;
605     u32 addr = ifa->if_local;
606     u32 prefix = ifa->if_address&mask;
607
608     if ((ifa->if_flags&IFA_F_SECONDARY) &lt;>
609         prim = inet_ifa_byprefix(in_dev, prefix, mask);
610         if (prim == NULL) {
611             printk(KERN_DEBUG "fib_add_ifaddr: bug: prim == "
612                               "NULL\n");
613             return;
614         }
615     }
616     fib_magic(RTM_NEWRROUTE, RTN_LOCAL, addr, 32, prim);
617
618     if (!(dev->flags&IFF_UP))
619         return;
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
```

```

481 /* Add broadcast address, if it is explicitly assigned. */
482     if (ifa->ifa_broadcast && ifa->ifa_broadcast != 0xFFFFFFFF)
483         fib_magic(RTM_NEWRROUTE, RTN_BROADCAST,
484                     ifa->ifa_broadcast, 32, prim);
485     if (!ZERONET(prefix) &&
486         !(ifa->ifa_flags&IFA_F_SECONDARY) &&
487         (prefix != addr || ifa->ifa_prefixlen < 32)) {
488         fib_magic(RTM_NEWRROUTE, dev->flags&IFF_LOOPBACK ?
489                     RTN_LOCAL : RTN_UNICAST, prefix,
490                     ifa->ifa_prefixlen, prim);
491     /* Add network specific broadcasts, when it takes a sense */
492     if (ifa->ifa_prefixlen < 31) {
493         fib_magic(RTM_NEWRROUTE, RTN_BROADCAST, prefix,
494                     32, prim);
495         fib_magic(RTM_NEWRROUTE, RTN_BROADCAST,
496                     prefix|~mask, 32, prim);
497     }
498 }

```

```

417 static void fib_magic(int cmd, int type, u32 dst, int
418 {           dst_len, struct in_ifaddr *ifa)
419   struct fib_table *tb;
420   struct {
421     struct nlmsghdrnlh;
422     struct rtmsgrtm;
423   } req;
424   struct kern_rta rta;
425
426   memset(&req.rtm, 0, sizeof(req.rtm));
427   memset(&rta, 0, sizeof(rta));
428
429   if (type == RTN_UNICAST)
430     tb = fib_new_table(RT_TABLE_MAIN);
431   else
432     tb = fib_new_table(RT_TABLE_LOCAL);
433
434   if (tb == NULL)
435     return;
436
437   req.nlh.nlmsg_len = sizeof(req);
438   req.nlh.nlmsg_type = cmd;
439   req.nlh.nlmsg_flags =
440     NLM_F_REQUEST | NLM_F_CREATE | NLM_F_APPEND;
441   req.nlh.nlmsg_pid = 0;
442   req.nlh.nlmsg_seq = 0;
443
444   req.rtm.rtm_dst_len = dst_len;
445   req.rtm.rtm_table = tb->tb_id;
446   req.rtm.rtm_protocol = RTPROT_KERNEL;
447   req.rtm.rtm_scope = (type != RTN_LOCAL ? RT_SCOPE_LINK :
448                         RT_SCOPE_HOST);
449   req.rtm.rtm_type = type;
450
451   rta.rta_dst = &dst;
452   rta.rta_PREFSRC = &ifa->ifalocal;
453   rta.rta_OIF = &ifa->ifadev->dev->ifindex;
454
455   if (cmd == RTM_NEWRROUTE)
456     tb->tb_insert(tb, &req.rtm, &rta, &req.nlh, NULL);
457   else
458     tb->tb_delete(tb, &req.rtm, &rta, &req.nlh, NULL);

```