What is a Network Device in Linux?

Network device is the representation of NIC. It resides in L2 (Link layer) in the Linux network stack. Exact implementation of Network Device can be found in include/linux/netdevice.h. This file has the definition of the net_device structure. Net_device structure holds information such as name, mac address, MTU, a list of multicast addresses, pointers to IPv4 (in_device struct) and IPv6 (inet6_dev struct) interfaces which holds IPv4 and IPv6 specific data associated to that network device respectively. There are many more fields in net_device struct not mentioned here.

What is IP Interface in Linux?

Ns-3's IP Interface class is equivalent to in_device structure found in include/linux/inetdevice.h. This structure seems to store IP addresses, IP multicast addresses, ARP params?, pointer to net_device struct and a whole lot of other data.

The file also seems to hold function prototypes that are used to register and unregister to inet_addr notification chain.

```
int register_inetaddr_notifier(struct notifier_block *nb);
int unregister_inetaddr_notifier(struct notifier_block *nb);
int register_inetaddr_validator_notifier(struct notifier_block *nb);
int unregister_inetaddr_validator_notifier(struct notifier_block *nb);
```

Relationship between net_device and in_device

net_device struct has a pointer to in_device struct at line 1971:

```
struct in_device __rcu    *ip_ptr;
```

Similarly, in_device has a pointer to net_device struct at line 26:

```
struct net_device    *dev;
```

Quoting from [2], The in_device structure stores all of the IPv4-related configuration for a network device, such as changes made by a user with the ifconfig or ip command. This structure is linked to the net_device structure via net_device->ip_ptr and can be retrieved with in_dev_get and _ _in_dev_get .

The in_device structure is allocated and linked to the device with inetdev_init , which is called when the first IPv4 address is configured on the device.

Furthermore, the in_device struct stores IP address configured for the device in in_ifaddr struct present in in_device at line 27:

```
struct in_ifaddr    __rcu *ifa_list;   /* IP ifaddr chain */
```

Therefore here is my conclusion: net_device is the representation of a NIC in Linux and in_device stores all IP related information of a particular net_device and IP addresses are stored in in_ifaddr.

What is network notification chain??

Network devices state can change dynamically; from time to time, the user/administrator can register/unregister network devices, change their MAC address, change their MTU, etc. The network stack and other subsystems and modules should be able to be notified about these events and handle them properly. The network notifications chains provide a mechanism for handling such events[1]. Every subsystem and every module can register itself to notification chains. Thereafter those subscribed subsystems will be notified about events. **The generation of notification events is done by calling the notifier_call_chain() method.**

Inetaddr_chain Notification Chain

The IP subsystem uses the inetaddr_chain notification chain to notify other kernel subsystems about changes to the IPv4 configuration of the local devices. Changes to IPv6 configuration is taken care of by inet6addr_chain. A kernel subsystem can register and unregister itself with inetaddr_chain by means of the register_inetaddr_notifier and unregister_inetaddr_notifier functions[2]. Here are two examples of users for this notification chain:

- Routing
- Netfilter masquerading

The two NETDEV_DOWN and NETDEV_UP events, respectively, are notified when an IP address is removed and when it is added to a local device. Such notifications are generated by the inet_del_ifa and inet_insert_ifa routines.

Instances of inetaddr_chain notification generation

Inetaddr_chain notifications are generated in the following functions:

In net/ipv4/devinet.c,

-
  ```
  static int __inet_insert_ifa(struct in_ifaddr *ifa, struct nlmsghdr *nlh,
                               u32 portid, struct netlink_ext_ack *extack)
  ```

-
  ```
  static void __inet_del_ifa(struct in_device *in_dev,
                             struct in_ifaddr __rcu **ifap,
                             int destroy, struct nlmsghdr *nlh, u32 portid)
  ```

__inet_insert_ifa ( ) function adds ifaddr struct to the list of ifaddr structs in the corresponding net_device and then calls for notification using

blocking_notifier_call_chain(&inetaddr_chain, NETDEV_UP, ifa)

method. The function takes in 3 arguments which specifies the notification chain, the state of netdevice, and the newly added IP address struct.

__inet_del_ifa ( ) function deletes the ifaddr struct given in the argument. This function generates the notification in three separate places. I think if there is no IP promotion, then primary and secondary IP addresses are deleted and a notification is generated like:

blocking_notifier_call_chain(&inetaddr_chain, NETDEV_DOWN, ifa1)

Whereas if promotion is enabled, then secondary address becomes primary and a notification is generated like

blocking_notifier_call_chain(&inetaddr_chain, NETDEV_UP, promote);

A grep search through the source code reveals that inetaddr_chain notifications are generated only in net/ipv4/devinet.c, i.e in the aforementioned functions only.

What one can deduct from these function calls is that, function calls notify the state of net_device. When an IP address is inserted or when a secondary IP address is promoted to primary, the state passed on to other sub-systems is NETDEV_UP and when the last IP address is deleted or promotion is disabled, NETDEV_DOWN state is passed.

Instances of inet6addr_chain notification generation

Inet6addr_chain is used to notify changes done to IPv6 configuration of a device. Similar to inetaddr_chain, these notifications are generated in ipv6_add_addr() function used to add IPv6 addresses and ipv6_del_addr() function used to delete IPv6 addresses. Additionally, notification is also generated in addrconf_ifdown().

Sub-systems subscribed to inet6addr_chain

A grep search `grep -Rwn "register_inet6addr_notifier"` showed no results that seemed relevant to the project. One entry that I may have overlooked is in net/mac80211/main.c.

Sub-systems subscribed to inetaddr_chain

A grep search for “register_inetaddr_notifier” which is the function that must be used to register to inetaddr_chain reveals 20 instances of function calls.

```
grep -iRwn "register_inetaddr_notifier"
```

The search is done on the root directory of Linux source code recursively. Following is my understanding of where these calls come from and the functions that are triggered by the notification. This will provide an understanding of what happens when IP Interface state changes.

- SCTP Protocol
- FIB Frontend
- Netfilter masquerading
- ATM
- Filesystem (2 matches)
- IPVLAN Driver
- Infiniband drivers (3 matches)
- Broadcom wireless driver
- Ethernet Drivers: Via, Mellanox, Qlogic (4 matches)
- S390 Driver
- Usermode linux kernel driver (2 matches)
- Mac80211

Out of the above entries, the only one useful for our purposes is FIB Frontend. The problem with entries concerning Ethernet drivers is that only the aforementioned drivers use them. Drivers like Intel or Realtek do not use the notification. So that implies there are drivers that work without using the notification.

Usage of inetaddr_chain in FIB

During the initialization of FIB, we register ourselves with the notifier facility in the net_device. We do this to put ourselves on the chain of functions called when network interface devices change status. **Now we can be sure we are informed when we have to add or delete routes from the FIB as devices go up or down.**

In net/ipv4/fib_frontend.c,

```
void __init ip_fib_init(void)
{
      ....code...

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

      ...code....
}
```

The above functions register for notifications from ineraddr_chain as well as netdevice_chain. Upon getting inetaddr_chain notification, the function

static int fib_inetaddr_event(struct notifier_block *this, unsigned long event, void *ptr)

Is executed. If the received event is NETDEV_UP, A new IP address has been configured on a local device. The handler must add the necessary routes to the local_table routing table. The routine responsible for this is fib_add_ifaddr. After this, the routing cache is flushed.

If the event is NETDEV_DOWN, An IP address has been removed from a local device. The handler must remove these routes that were added by the previous NETDEV_UP event. The routine responsible for this is fib_del_ifaddr. Moreover, if it is determined that the last IP address is removed from the Interface, the IP protocol is disabled on the device. If routes are deleted, the function also flushes the routing table immediately. It also flushes the routing cache immediately and asks ARP to clear from its cache all the entries that refer to the device where the IP protocol is being shut down.

netdev_chain Notification Chain

The netdev_chain is used to notify subsystems about changes in the status of a Network Device. To receive notifications from netdev_chain, a subsystem must register to the chain using the following function:

```
int register_netdevice_notifier(struct notifier_block *nb);
int unregister_netdevice_notifier(struct notifier_block *nb);
```

The unregister_netdevice_notifier is used to unregister from the notification chain.

What are the possible notifications?

```
/* netdevice notifier chain. Please remember to update netdev_cmd_to_name()
 * and the rtnetlink notification exclusion list in rtnetlink_event() when
 * adding new types.
 */
enum netdev_cmd {
    NETDEV_UP    = 1,    /* For now you can't veto a device up/down */
    NETDEV_DOWN,
    NETDEV_REBOOT,       /* Tell a protocol stack a network interface
                        detected a hardware crash and restarted
                        - we can use this eg to kick tcp sessions
                        once done */
    NETDEV_CHANGE,       /* Notify device state change */
    NETDEV_REGISTER,
    NETDEV_UNREGISTER,
    NETDEV_CHANGEMTU,    /* notify after mtu change happened */
    NETDEV_CHANGEADDR,    /* notify after the address change */
    NETDEV_PRE_CHANGEADDR,    /* notify before the address change */
    NETDEV_GOING_DOWN,
    NETDEV_CHANGENAME,
    NETDEV_FEAT_CHANGE,
    NETDEV_BONDING_FAILOVER,
    NETDEV_PRE_UP,
    NETDEV_PRE_TYPE_CHANGE,
    NETDEV_POST_TYPE_CHANGE,
    NETDEV_POST_INIT,
    NETDEV_RELEASE,
    NETDEV_NOTIFY_PEERS,
    NETDEV_JOIN,
    NETDEV_CHANGEUPPER,
    NETDEV_RESEND_IGMP,
    NETDEV_PRECHANGEMTU,    /* notify before mtu change happened */
    NETDEV_CHANGEINFODATA,
    NETDEV_BONDING_INFO,
    NETDEV_PRECHANGEUPPER,
    NETDEV_CHANGELOWERSTATE,
    NETDEV_UDP_TUNNEL_PUSH_INFO,
    NETDEV_UDP_TUNNEL_DROP_INFO,
    NETDEV_CHANGE_TX_QUEUE_LEN,
    NETDEV_CVLAN_FILTER_PUSH_INFO,
    NETDEV_CVLAN_FILTER_DROP_INFO,
    NETDEV_SVLAN_FILTER_PUSH_INFO,
    NETDEV_SVLAN_FILTER_DROP_INFO,
};
```

Instances of netdev_chain notification generation

Subsystems subscribed to netdev_chain notifications

A grep search for a function call to register for netdev_chain revealed 130 entries.

```
grep -Rwn "register_netdevice_notifier"
```

Obviously irrelevant ones were removed and what I think is within scope of the project is explored. Apart from that, there may be some entries which I may have overlooked. Those which I suspect are given below:

- net/ieee802154/core.c
- net/ieee802154/6lowpan/core.c
- net/bluetooth/6lowpan.c
- net/bridge/br_netfilter_hooks.c
- net/bridge/br.c
- net/packet/af_packet.c
- net/mac802154/iface.c
- net/6lowpan/core.c
- net/ipv4/ipmr.c
- net/ipv4/igmp.c
- net/ipv4/netfilter/ipt_CLUSTERIP.c
- net/netfilter/
- net/wireless/core.c
- net/ipv6/mcast.c
- net/ipv6/ip6mr.c
- drivers/net/ethernet/

Above entries may or maynot have some importance; It is not known right now and I won't be looking into it.

Routing sub-system

The routing subsystem registers three different handlers with the netdev_chain notification chain to handle changes in the status of a device:

- fib_rules_event() in net/core/fibrules.c updates the policy database, when policy routing is in effect.
- fib_netdev_event() in net/ipv4/fib_frontend.c updates the routing tables.
- inetdev_event() in net/ipv4/devinet.c updates the device's IP configuration.

Policy routing is not available in ns-3 hence fib_rules_event() is not relevant.

fib_netdev_event()

When a network device associated with one or more routes is disabled, a NETDEV_DOWN notification is sent. The FIB callback for handling this event is the fib_netdev_event() method.
Following events are handled by this function:

NETDEV_UNREGISTER: When a device is unregistered, all the routes that use this device are removed from the routing tables (cache included). Multipath routes are also removed if at least one of the next hops uses this device[2].

NETDEV_DOWN:fib_disable_ip() method is called and it performs the following steps[1]:

- First, the fib_sync_down_dev() method is called (net/ipv4/fib_semantics.c). In the fib_sync_down_dev() method, the RTNH_F_DEAD flag of the nexthop flags (nh_flags) is set and the FIB info flags (fib_flags) is set.
- The routes are flushed by the fib_flush() method.
- The rt_cache_flush() method and the arp_ifdown() method are invoked. The arp_ifdown() method deletes all ARP entries associated with the device and rt_flush_cache() flushes routing cache.

NETDEV_UP: When a device comes up, routing entries for all its IP addresses must be added to the local routing table ip_fib_local_table . This is accomplished by calling fib_add_ifaddr for each IP configured on the device[2].

NETDEV_CHANGE, NETDEV_CHANGEMTU: When a configuration change is applied to a device, the routing table cache is flushed. Among the most common notified changes are modifications of the MTU or the PROMISCUITY status.

Unregistering a device and shutting down a device can have different effects on the routing table. Some of the reasons a device can be unregistered include a user removing the driver from the kernel or unplugging a hotplug device such as a PCMCIA Ethernet card. Some of the reasons a device can be shut down include a user unplugging the cable or issuing an administrative command[2].
The route to the IP address is not removed when the device is shut down because its IP address belongs to the host, not to the interface. This address exists as long as its associated device exists i.e registered.

inetdev_event()
Here, the function inetdev_event handles changes to IP configuration as follows:

NETDEV_UNREGISTER: Disables the IP protocol on the device.
NETDEV_UP: Enables the multicast configuration (if present) with ip_mc_up . When the device going up is the loopback device, configure the 127.0.0.1/8 address on it. This notification is ignored if the device going up has a configured MTU smaller than the minimum value. It also sends gratuitous ARP if IN_DEV_ARP_NOTIFY(in_dev) macro returns success.

IN_DEV_ARP_NOTIFY (in_dev) returns the max value of proc/sys/net/ipv4/conf/<netDevice>/arp_notify and /proc/sys/net/ipv4/conf/all/arp_notify, where netDevice is the network device associated with the specified in_dev.

NETDEV_DOWN: Disables the multicast configuration (if present) with ip_mc_down .

NETDEV_CHANGEMTU: Checks whether the device's MTU has been set to a value smaller than the minimum necessary to run the IP protocol (68), and if so, disables the IP protocol on the device.
NETDEV_CHANGENAME: Updates the name of the directories /proc/sys/net/ipv4/conf/ devname and /proc/sys/net/ipv4/neigh/devname to reflect the new device name.

For both NETDEV_UNREGISTER and NETDEV_CHANGEMTU, the IP protocol is disabled with inetdev_destroy . That function removes all IP configurations from the device and clears the ARP cache accordingly with arp_ifdown.

ARP Module (net/ipv4/arp.c)

File Description: This module implements the Address Resolution Protocol ARP (RFC 826), which is used to convert IP addresses (or in the future maybe other high-level addresses) into a low-level hardware address (like an Ethernet address).

ARP module has a registered function arp_netdev_event() to handle notifications from netdevice. This function handles 2 cases:

- NETDEV_CHANGEADDR: It handles changes of MAC address events by calling the generic neigh_changeaddr() method and by calling the rt_cache_flush() method. rt_cache_flush flushes the IPv4 routing cache so that the IP layer is forced to start using the new L2 address. This function does not selectively delete the entries associated with the device that generated the notification, but simply removes everything in the cache.

- NETDEV_CHANGE: From kernel 3.11, you handle a NETDEV_CHANGE event when there was a change of the IFF_NOARP flag by calling the neigh_changeaddr() method. IFF_NOARP is set for devices which do not use the ARP protocol. neigh_changeaddr()is the function that neighboring protocols can invoke to update a protocol's cache when the L2 address of a local device has changed.

IPv6 Interface Configuration (net/ipv6/addrconf.c)

Here, NetDevice's IPv6 related configuration details are managed. NetDevice events are handled by addrconf_notify() function. It handles the following events:

NETDEV_REGISTER: An IPv6 device (interface) is configured for the device. Values such as MTU are set and device is added to mandatory multicast groups, etc. This is done by ipv6_add_dev() function. This function is called only when MTU is greater than or equal to minimum MTU value.
NETDEV_CHANGEMTU: If MTU is less than minimum set value, IPv6 is stopped on this device. Change in IP configuration is notified via inet6addr_chain notification. Otherwise MTU is set.
NETDEV_UP, NETDEV_CHANGE: If IPv6 is disabled on this device, nothing is done. Otherwise routes for each IP address in the device is added.

NETDEV_DOWN, NETDEV_UNREGISTER: All addresses are removed from this interface. Under some conditions, this change in IP configuration triggers a inet6addr_chain notification. Other events handled are: NETDEV_PRE_TYPE_CHANGE,NETDEV_POST_TYPE_CHANGE and NETDEV_CHANGEUPPER.

Point-to-Point NetDevice in Linux and netdev_chain notifications

The following observations are made by examining drivers/net/ppp/ppp_generic.c file. The description of PPP Generic provided in this file is given below:

```
// SPDX-License-Identifier: GPL-2.0-or-later
/*
 * Generic PPP layer for Linux.
 *
 * Copyright 1999-2002 Paul Mackerras.
 *
 * The generic PPP layer handles the PPP network interfaces, the
 * /dev/ppp device, packet and VJ compression, and multilink.
 * It talks to PPP `channels' via the interface defined in
 * include/linux/ppp_channel.h.  Channels provide the basic means for
 * sending and receiving PPP frames on some kind of communications
 * channel.
 *
 * Part of the code in this driver was inspired by the old async-only
 * PPP driver, written by Michael Callahan and Al Longyear, and
 * subsequently hacked by Paul Mackerras.
 *
 * ==FILEVERSION 20041108==
 */
```

When the driver is loaded, the kernel calls an initialization function that was registered by the driver. This function in ppp_generic is

```
static int __init ppp_init(void)
```

This function registers callbacks that can handle situations such as:

- The device is added to a network namespace.
- Reading/ Writing from the device.
- Handling network devices.

Here I am concerned with the third case: Callbacks that handle network devices.

```
static struct rtnl_link_ops ppp_link_ops __read_mostly = {
    .kind          = "ppp",
    .maxtype    = IFLA_PPP_MAX,
    .policy        = ppp_nl_policy,
    .priv_size    = sizeof(struct ppp),
    .setup         = ppp_setup,
    .validate    = ppp_nl_validate,
    .newlink    = ppp_nl_newlink,
    .dellink    = ppp_nl_dellink,
    .get_size    = ppp_nl_get_size,
    .fill_info    = ppp_nl_fill_info,
    .get_link_net    = ppp_nl_get_link_net,
```

The above struct contains callbacks for handling network devices. The documentation provided for the struct in the code is given below:

```
/**
 *    @kind: Identifier
 *    @maxtype: Highest device specific netlink attribute number
 *    @policy: Netlink policy for device specific attribute validation
 *    @priv_size: sizeof net_device private space
 *    @validate: Optional validation function for netlink/changelink
parameters
 *    @setup: net_device setup function
 *    @newlink: Function for configuring and registering a new device
 *    @dellink: Function to remove a device
 *    @get_size: Function to calculate required room for dumping device
 *          specific netlink attributes
 *    @fill_info: Function to dump device specific netlink attributes
 *    @get_link_net: Function to get the i/o netns of the device
 */
```

`ppp_setup` fills up some details of Network Device in net_device struct such as device type as PPP, MTU, flags, etc. For example, flags are set as:

```
dev->flags = IFF_POINTOPOINT | IFF_NOARP | IFF_MULTICAST;
```

IFF_POINTTOPOINT: This flag signals that the interface is connected to a point-to-point link.
IFF_NOARP: This means that the interface can’t perform ARP.
IFF_MULTICAST: This flag is set by drivers to mark interfaces that are capable of multicast transmission.

`ppp_nl_newlink` is used to configure and register a new device. This fills in private data of net_device, initializes net_device using a given callback, fills in flags and sends a notification

```
call_netdevice_notifiers(NETDEV_POST_INIT, dev);
```

After that, it initializes tc, sets device address and sends another notification

```
call_netdevice_notifiers(NETDEV_REGISTER, dev);
```

This notifies protocols that a new device has appeared.

`ppp_nl_dellink` is used to remove a device. This function shuts down a device interface and removes it from the kernel tables. First, the function has to deregister the device from the kernel. Deregistering involves closing the device, Removing it from the notification chain, processing already received packets, shutdown queueing discipline, etc.

After closing the device, a notification is sent

```
call_netdevice_notifiers(NETDEV_DOWN, dev);
```

And after shutting down queuing discipline, a notification is sent

```
call_netdevice_notifiers(NETDEV_UNREGISTER, dev);
```

References

[1] Rami Rosen - Linux Kernel Networking: Implementation and Theory.

[2] Christian Benvenuti - Understanding Linux network internals.

[3] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman - Linux Device Drivers.

Note: This document refers to Linux kernel 5.7-rc6. The books [1],[2] and [3] are most probably referring to an older version.