

ARP Receive

ARP packet structures

ARP packets consist of the protocol independent header shown in blue followed by a protocol dependent pair of hardware and protocol (IP) addresses.

```
128 struct arphdr
129 {
130     unsigned short ar_hrd; /* format of hardware address */
131     unsigned short ar_pro; /* format of protocol address */
132     unsigned char  ar_hln; /* length of hardware address */
133     unsigned char  ar_pln; /* length of protocol address */
134     unsigned short ar_op;  /* ARP opcode (command) */
135
136 #if 0
137 /*
138 * Ethernet looks like this :
139 */
140     unsigned char ar_sha[ETH_ALEN]; /* send hardware      */
141     unsigned char ar_sip[4];        /* sender IP address */
142     unsigned char ar_tha[ETH_ALEN]; /* target hardware    */
143     unsigned char ar_tip[4];        /* target IP address  */
144 #endif
145
146 };
```

```
09:05:57.478352 arp who-has jmw9 tell jmw7
0x0000      0001 0800 0604 0001 00b0 d0e9 0f5b c0a8      .....[...
0x0010      0221 0000 0000 0000 c0a8 022c                .....

09:05:57.478376 arp reply jmw9 is-at 0:9:6b:e3:7e:a2
0x0000      0001 0800 0604 0002 0009 6be3 7ea2 c0a8      .....k.~...
0x0010      022c 00b0 d0e9 0f5b c0a8 0221                .,.....[...!
```

The *arp_rcv()* function

The *arp_rcv()* function defined in *net/ipv4/arp.c* is the arp packet handler invoked by *net_rx_action()* when an ARP packet is received. In the following, the pointer, *arp*, references the ARP header structure and *arp_ptr* refers to the data consisting of the two MAC and two IP addresses.

```
576 /*
577 *      Receive an arp request by the device layer.
578 */
579
580 int arp_rcv(struct sk_buff *skb, struct net_device *dev,
              struct packet_type *pt)
581 {
582     struct arphdr *arp = skb->nh.arph;
583     unsigned char *arp_ptr= (unsigned char *) (arp+1);
584     struct rtable *rt;
585     unsigned char *sha, *tha;
586     u32 sip, tip;
587     u16 dev_type = dev->type;
588     int addr_type;
589     struct in_device *in_dev = in_dev_get(dev);
590     struct neighbour *n;
```

Validating the ARP packet

The following validity checks are performed on received packets.

- An *in_device* structure must be associated with the device on which the packet was received.
- The hardware header length specified in the arp header must match that of the device
- The protocol address length must be 4, the length of an IP address.
- The device must support ARP.
- The *pkt_type* must not indicate this packet is a loopback or destined for another host.

```
599     if (in_dev == NULL ||
600         arp->ar_hln != dev->addr_len    ||
601         dev->flags & IFF_NOARP ||
602         skb->pkt_type == PACKET_OTHERHOST ||
603         skb->pkt_type == PACKET_LOOPBACK ||
604         arp->ar_pln != 4)
605         goto out;
```

If the *sk_buff* is shared, then it is cloned by the *skb_share_check()* function.

```
607     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
608         goto out_of_mem;
```

If the *sk_buff* is non-linear, it is linearized by the *skb_linearize()* function and *arp* and *arp_ptr* are reset to refer to the new data.

```
610     if (skb_is_nonlinear(skb)) {
611         if (skb_linearize(skb, GFP_ATOMIC) != 0)
612             goto freeskb;
613         arp = skb->nh.arph;
614         arp_ptr = (unsigned char *) (arp+1);
615     }
```

The [type of device](#) on which the arp packet arrived and the [arp hardware address type](#) should be the same and must be either ARPHRD_ETHER or ARPHRD_IEEE802. Similarly, the [protocol type](#) field of the arp packet should be ETH_P_IP.

```
617     switch (dev_type) {
618     default:
619         if (arp->ar_pro != __constant_htons(ETH_P_IP))
620             goto out;
621         if (htons(dev_type) != arp->ar_hrd)
622             goto out;
623         break;

624 #ifdef CONFIG_NET_ETHERNET
625     case ARPHRD_ETHER:
626         /*
627         ETHERNET devices will accept ARP hardware types of
628         either (Ethernet) or 6 (IEEE 802.2).
629         */
630         if (arp->ar_hrd != __constant_htons(ARPHRD_ETHER) &&
631             arp->ar_hrd !=
632                 __constant_htons(ARPHRD_IEEE802))
633             goto out;
634         if (arp->ar_pro != __constant_htons(ETH_P_IP))
635             goto out;
636         break;
637 #endif
```

A number of similar protocol dependent blocks (Token Ring, etc...) follow here.

```
695     :  
        }
```

Validation continues with the message type.

```
696     /* Understand only these message types */  
697  
698     if (arp->ar_op != __constant_htons(ARPOP_REPLY) &&  
699         arp->ar_op != __constant_htons(ARPOP_REQUEST))  
700         goto out;
```

At this point the packet header is thought to be valid, and data fields in the ARP packet are retrieved to local variables. *sha* is *sender_hardware_address* and *tip* is *target_ip* address, etc. Note that *sha* and *tha* are pointers, but *sip* and *tip* are values.

```
702 /*  
703  *      Extract fields  
704  */  
  
705     sha=arp_ptr;  
706     arp_ptr += dev->addr_len;  
707     memcpy(&sip, arp_ptr, 4);  
708     arp_ptr += 4;  
709     tha=arp_ptr;  
710     arp_ptr += dev->addr_len;  
711     memcpy(&tip, arp_ptr, 4);
```

Bad requests for loopback and multicast addresses are dropped.

```
712 /*
    Check for bad requests for 127.x.x.x and requests for
    multicast addresses. If this is one such, delete it.
715 */
716 if (LOOPBACK(tip) || MULTICAST(tip))
717     goto out;
718
```

If the packet arrived on a device of type ARPHRD_DLCI (**frame relay DLCI ??**), then the source hardware address of the packet is reset to broadcast address of the device ??

```
719 /*
    Special case: We must set Frame Relay source Q.922
    address
721 */
722 if (dev_type == ARPHRD_DLCI)
723     sha = dev->broadcast;
```

Processing of validated packets

Actual processing of the packet begins here.

- For ARP responses it is necessary to update an existing *neighbour* structure.
- For ARP requests it is necessary to create or update a *neighbour* structure and send the reply.

Duplicate address detection

If the source IP address is NULL, and the packet is an ARP request, and the target ip address is of type RTN_LOCAL indicating that it is owned by this machine, then this is an IPv4 duplicate address detection packet. An ARP reply is sent immediately and no updating of the *neighbour* structures occurs.

```
742     /* Special case: IPv4 duplicate address detection
743     packet (RFC2131) */
743     if (sip == 0) {
744         if (arp->ar_op ==
745             __constant_htons(ARPOP_REQUEST) &&
746             inet_addr_type(tip) == RTN_LOCAL)
747             arp_send(ARPOP_REPLY, ETH_P_ARP, tip, dev,
748                     tip, sha, dev->dev_addr, dev->dev_addr);
748     }
748     goto out;
```

Processing of ARP request packets

If the incoming ARP packet is an ARP request then, `ip_route_input()` is invoked. The objective of this is to determine if this host owns `tip`. The `ip_route_input()` function returns NULL if the packet is routeable and this host owns `tip` if the route type is RTN_LOCAL.

```
750     if (arp->ar_op == __constant_htons(ARPOP_REQUEST)
751         && ip_route_input(skb, tip, sip, 0, dev) == 0) {
753         rt = (struct rtable*)skb->dst;
754         addr_type = rt->rt_type;
```

Processing of ARP requests for this host

If the route cache entry is of type RTN_LOCAL (i.e. the packet is for local delivery), then the packet is an ARP request for this host. The `neigh_event_ns()` function updates the ARP cache by [creating a neighbour structure](#) if necessary and [caching the hardware address](#) of the neighbour that initiated the ARP request.

```
756         if (addr_type == RTN_LOCAL) {
757             n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
```

`neigh_event_ns` returns the updated neighbour structure on success and returns NULL on error.

```
758             if (n) {
759                 int dont_send = 0;
```

If the ARP filter option is enabled for IPV4 or in the `in_device` structure and the `arp_filter()` function validates the request. It return true when the ARP request is invalid.

```
760                 if (IN_DEV_ARPFILTER(in_dev))
761                     dont_send |=
                        arp_filter(sip, tip, dev);
```

If `dont_send` remains false, an ARP reply is sent to the requesting neighbour. Parameters to ARP send are the *target ip*, *source ip*, *target hardware address*, *source hardware address*.

```
762                 if (!dont_send)
763                     arp_send(ARPOP_REPLY,
                        ETH_P_ARP, sip, dev, tip, sha,
                        dev->dev_addr, sha);
765                 neigh_release(n);
766             }
```

This concludes the processing of [ARP requests](#) destined for this host. A jump is taken to the exit point, `out`.

```
767             goto out;
```

ARP requests for other hosts

If route type of the ARP request packet was not local and if forwarding is enabled on the input device, then a check is made to see if this request requires a proxy arp reply.

```
768             } else if (IN_DEV_FORWARD(in_dev)) {
```

If this is an ARP request for one of the neighbours for which we are acting as a proxy then one of the following conditions should hold true:

- RTCF_DNAT (destination NAT) flag is set in the route cache entry indicating that the packet destination address must be translated. For an ARP request, this indicates that intended destination is a neighbour "behind" this host.
- The address type of the next route is RTN_UNICAST *and* the device for the next hop is different from the device the packet arrived on and either proxy ARP is supported by the device or a proxy neighbour structure of the target host behind this host is already present in the cache.

If one of the above conditions is true, the hardware address of the ARP request source host is added to the neighbour cache by the *neigh_event_ns* routine.

```
769             if ((rt->rt_flags&RTCF_DNAT) ||
770                 addr_type == RTN_UNICAST  &&
771                 rt->u.dst.dev != dev &&
772                 (IN_DEV_PROXY_ARP(in_dev) ||
773                 pneigh_lookup(&arp_tbl, &tip,
774                               dev, 0))) {
775                 n = neigh_event_ns(&arp_tbl,
776                                   sha, &sip, dev);
```

The *neigh_lookup()* function called by *neigh_event_ns()* above increments the reference count of the structure. Here, *neigh_release* is called to decrement the reference count after it has been used and updated above.

```
773                 if (n)
774                     neigh_release(n);
```

Proxy ARP replies

The *proxy_delay* parameter in the *arp_tbl* set to $(8 * \text{HZ}) / 10$, so by default all proxy ARP replies are delayed. The ARP reply is sent without delay only if any one of the following conditions is true. Otherwise it is queued by the *pneigh_enqueue* routine.

- If the *stamp.tv_sec* field in the *sk_buff* has been reset to zero by *pneigh_enqueue* i.e. this *sk_buff* has been queued for a specific period of time. (Note: All incoming packets were time-stamped (i.e. *do_gettimeofday(&skb->stamp)*) back in *netif_rx* routine.) **While it is true that *pneigh_enqueue()* does zero the time stamp and put the packet on the proxy_queue, it remains unclear how control could reach line 776 after proceeding down that path.**
- If the *sk_buff* packet type is *PACKET_HOST* or if the *proxy_delay* field in the *arp_parms* structure of *in_device* equals zero.

```
776         if (skb->stamp.tv_sec == 0
            || skb->pkt_type==PACKET_HOST
            || in_dev->arp_parms->
               proxy_delay == 0) {
            arp_send(ARPOP_REPLY,
                    ETH_P_ARP, sip, dev, tip,
                    sha, dev->dev_addr, sha);
780     } else {
            pneigh_enqueue(&arp_tbl,
            in_dev->arp_parms, skb);
782     in_dev_put(in_dev);
783     return 0;
784     }
785     goto out;
786     }
787     }
788 }
```

Handling ARP responses

Before an ARP request is sent, a *neighbour* structure must be created. Thus the *neigh_lookup()* function is called with the *creat* flag set to NULL indicating that a new *neighbour* should *not* be created if the lookup fails. If the lookup should fail, this packet is an unsolicited ARP response.

```
790     /* Update our ARP tables */
792     n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
```

Handling of unsolicited ARP responses

Unsolicited ARP responses are not accepted unless CONFIG_IP_ACCEPT_UNSOLICITED_ARP is defined. If the neighbour lookup failed, and if the packet is an ARP reply with the IP source address of type RTN_UNICAST, then this is an unsolicited ARP reply. In this case *__neigh_lookup* is invoked a second time but with the *creat* flag set to create a new neighbour structure.

```
794 #ifdef CONFIG_IP_ACCEPT_UNSOLICITED_ARP
795     /*
           Unsolicited ARP is not accepted by default.
           It is possible, that this option should be
           enabled for some devices (strip is candidate)
798     */
799     if (n == NULL &&
800         arp->ar_op == __constant_htons(ARPOP_REPLY) &&
801         inet_addr_type(sip) == RTN_UNICAST)
802         n = __neigh_lookup(&arp_tbl, &sip, dev, -1);
803 #endif
```

Handling of solicited (and acceptable unsolicited) ARP responses

If the neighbour lookup or creation is successful, the default new state of the neighbour structure in the cache is NUD_REACHABLE.

```
805     if (n) {
806         int state = NUD_REACHABLE;
807         int override = 0;
808
809     /*
      If several different ARP replies follows
      back-to-back, use the FIRST one. It is
      possible, if several proxy agents are
      active. Taking the first reply prevents
      arp trashing and chooses the fastest router.
813     */
```

If the last update time of the neighbour structure is greater than the *locktime* parameter (set to 1*Hz in *arp_tbl*) of the neighbour, the *override* flag is set to true. [The *override* flag permits a new hardware address to replace an existing one.](#)

```
814         if (jiffies - n->updated >= n->parms->locktime)
815             override = 1;
```

If the ARP packet is not an ARP reply (**how could control reach here in that case???**) or if the packet type is not `PACKET_HOST` (unicast packet destined to this host) then the default state of the neighbour is reset to `NUD_STALE` .

```
817         /* Broadcast replies and request packets
818         do not assert neighbour reachability.
819         */
820         if (arp->ar_op !=
            __constant_htons(ARPOP_REPLY) ||
821             skb->pkt_type != PACKET_HOST)
822             state = NUD_STALE;
```

The call to `neigh_update()` generally sets the state to `NUD_REACHABLE` if it's a direct ARP reply and to `NUD_STALE` if it is not. The call to `neigh_release()` decrements the reference count.

```
823         neigh_update(n, sha, state, override, 1);
824         neigh_release(n);
825     }
826
827 out:
828     if (in_dev)
829         in_dev_put(in_dev);
830 freeskb:
831     kfree_skb(skb);
832 out_of_mem:
833     return 0;
834 }
```

ARP filters

IN_DEV_ARPFILTER has been defined in *include/linux/inetdevice.h*. The *ipv4_devconf* structure holds various IPv4 configuration values. A static variable of this structure named *ipv4_devconf* is declared in *net/ipv4/devinet.c* and initialized with the default values. By default *arp_filter* is turned off.

```
#define IN_DEV_ARPFILTER(in_dev) (ipv4_devconf.arp_filter
                                || (in_dev)->cnf.arp_filter)

6 struct ipv4_devconf
7 {
8     int     accept_redirects;
9     int     send_redirects;
10    int     secure_redirects;
11    int     shared_media;
12    int     accept_source_route;
13    int     rp_filter;
14    int     proxy_arp;
15    int     bootp_relay;
16    int     log_martians;
17    int     forwarding;
18    int     mc_forwarding;
19    int     tag;
20    int     arp_filter;
21    void    *sysctl;
22 };

63 struct ipv4_devconf ipv4_devconf = { 1, 1, 1, 1, 0, };
```

Note: These values can be configured using the old `sysctl` command interface or the present `proc` file system interface. These configuration values are rooted in the `/proc/sys/net/ipv4/conf` directory.

The *arp_filter* function

The *arp_filter* function defined in *net/ipv4/arp.c* rejects the packet

- when a return route for the reply cannot be determined and
- when an output route is available but the output device is different from the device, the arp request arrived on.

In the case of a rejection, the neighbour structure is released and the packet is dropped in the *out* block of *arp_rcv()*.

```
348 static int arp_filter(__u32 sip, __u32 tip,
                        struct net_device *dev)
349 {
350     struct rtable *rt;
351     int flag = 0;
352     /*unsigned long now; */
353
354     if (ip_route_output(&rt, sip, tip, 0, 0) < 0)
355         return 1;
356     if (rt->u.dst.dev != dev) {
357         NET_INC_STATS_BH(ArpFilter);
358         flag = 1;
359     }
360     ip_rt_put(rt);
361     return flag;
362 }
```

Updating ARP Cache entries

The `neigh_event_ns()` function defined in `net/core/neighbour.c` is called when ARP *requests* are received. It attempts to locate a *neighbour* structure with key equal to the source address of the ARP request packet. If successful, `neigh_update()` updates the structure using link layer address in the ARP packet. Note that `neigh_event_ns` sets the *neighbour* state to `NUD_STALE`, as it is not called in response to a direct ARP reply from the neighbour.

```
883 struct neighbour * neigh_event_ns(struct neigh_table *tbl,
884                                 u8 *lladdr, void *saddr,
885                                 struct net_device *dev)
886 {
887     struct neighbour *neigh;
888
889     neigh = __neigh_lookup(tbl, saddr, dev,
890                          lladdr || !dev->addr_len);
891     if (neigh)
892         neigh_update(neigh, lladdr, NUD_STALE, 1, 1);
893     return neigh;
894 }
```

The *neigh_update* function

The *neigh_update()* function is defined in *net/core/neighbour.c*. The input parameters are described in the comment block below. The parameter *lladdr* refers to the Link Layer or MAC address.

```
766 /* Generic update routine.
    -- lladdr is new lladdr or NULL, if it is not supplied.
    -- new is new state.
    -- override==1 allows to override existing lladdr,
       if it is different.
    -- arp==0 means that the change is administrative (i.e
       not generated by the arp protocol.
    Caller MUST hold reference count on the entry.
773 */
774
775 int neigh_update(struct neighbour *neigh, const u8 *lladdr,
                  u8 new, int override, int arp)
776 {
777     u8 old;
778     int err;
779     int notify = 0;
780     struct net_device *dev = neigh->dev;
781
782     write_lock_bh(&neigh->lock);
783     old = neigh->nud_state;
784
785     err = -EPERM;
```

If the present neighbour state is either NUD_NOARP or NUD_PERMANENT, then it should not be changed regardless of what the caller might think!

```
786     if (arp && (old & (NUD_NOARP | NUD_PERMANENT)))
787         goto out;
```

New state not VALID

If the *new* state is *not* in the NUD_VALID set $\{NUD_REACHABLE, NUD_PROBE, NUD_STALE, NUD_DELAY, NUD_PERMANENT, NUD_NOARP\}$, then any timer that was set up before is deleted. It appears that the only way this can occur is when *neigh_delete()* sets the new state to *NUD_FAILED*.

```
789     if (!(new & NUD_VALID)) {
790         neigh_del_timer(neigh);
```

New state not VALID and old state REACHABLE

If the *old* state was in the NUD_CONNECTED set $\{NUD_REACHABLE, NUD_PERMANENT, NUD_NOARP\}$ and the new state is not in the NUD_VALID set, *neigh_suspect()* is called to update the output function pointers so that the sending of an ARP request will be triggered. The *notify* field is used to communicate the new state to the user-space [ARP daemon](#).

```
791         if (old & NUD_CONNECTED)
792             neigh_suspect(neigh);
793         neigh->nud_state = new;
794         err = 0;
795         notify = old & NUD_VALID;
796         goto out;
797     }
```

New state is VALID

Devices not using link layer addresses

If the address length of the device is zero i.e. the device doesn't need an address, the *lladdr* field is set to point to the neighbour structure's hardware address field which presumably contains *nothing*.

```
799     /* Compare new lladdr with cached one */
800     if (dev->addr_len == 0) {
801     /* First case: device needs no address. */
802         lladdr = neigh->ha;
```

Device requires link layer address and one is specified

If the old state of the neighbour cache entry is valid then both the new hardware address and the cached hardware address are compared. If both are equal, then the *lladdr* pointer is reset to the existing the hardware address.. **what does this accomplish?** If they are not the same and the *override* flag is false, then processing is aborted.

```
803     } else if (lladdr) {
804     /* The second case: if something is already cached
805     and a new address is proposed:
806     - compare new & old
807     - if they are different, check override flag
808     */
809     if (old & NUD_VALID) {
810         if (memcmp(lladdr, neigh->ha,
811                 dev->addr_len) == 0)
812             lladdr = neigh->ha;
813         else if (!override)
814             goto out;
```

Device requires link layer address but it is not specified

```
815     } else {
816     /* No address is supplied; if we know
      something, use it, otherwise discard the request.
818     */
819         err = -EINVAL;
```

If no new hardware address is supplied and the old state is not VALID there is nothing more that can be done. However, if the state *is* valid the present link level address is used.

```
820         if (!(old & NUD_VALID))
821             goto out;
822         lladdr = neigh->ha;
823     }
```

Recovering the *old* state, part II.

The `neigh_sync()` function is called to determine the current state of the neighbor. This function effects the transitions between `NUD_REACHABLE` and `NUD_STALE` based upon whether or not the entry has last been *confirmed* within the `base_reachable_time` interval.

```
825     neigh_sync(neigh);
826     old = neigh->nud_state;
```

If the new state is one of the `NUD_CONNECTED` states (i.e. `NUD_REACHABLE` or `NUD_NOARP` or `NUD_PERMANENT`), then the confirmed time is updated. **This appears to be the only place `neigh->confirmed` gets updated.**

```
827     if (new & NUD_CONNECTED)
828         neigh->confirmed = jiffies;
829     neigh->updated = jiffies;
```

If the present state of the neighbour is in the `NUD_VALID` set { `NUD_REACHABLE`, `NUD_PROBE`, `NUD_STALE` } and there is no change in the neighbour hardware address then if either of the following are true, the state is not updated.

- Both the proposed new state and the current state are equal
- The proposed new state is `NUD_STALE` and the current state is one of `NUD_CONNECTED` states. (Note: The value of *old* was set *after* the call to `neigh_sync()`. Thus this condition ensures that connected neighbour entries are not overridden when the input parameter is `NUD_STALE`).

```
831     /* If entry was valid and address is not changed,
832        do not change entry state, if new one is STALE.
833        */
834     err = 0;
835     if (old & NUD_VALID) {
836         if (lladdr == neigh->ha)
837             if (new == old || (new == NUD_STALE &&
838                 (old & NUD_CONNECTED)))
839                 goto out;
```

Updating the neighbour state

Any pending timer is deleted and the new state is assigned to the neighbour.

```
840     neigh_del_timer(neigh);
841     neigh->nud_state = new;
```

If there is a change in the hardware address of the neighbour, the new address is copied to the neighbour and all cached hardware headers of the neighbour are updated by the `neigh_update_hhs()` function. **This is the reason for the obscure reset of `lladdr` that was noted earlier.**

```
842     if (lladdr != neigh->ha) {
843         memcpy(&neigh->ha, lladdr, dev->addr_len);
844         neigh_update_hhs(neigh);
```

If the new state is not one of the `NUD_CONNECTED` states, then the `confirmed` ticks field is reset back by twice the `base_reachable_time`.

```
845         if (!(new&NUD_CONNECTED))
846             neigh->confirmed = jiffies -
                (neigh->parms->base_reachable_time<<1);
```

If user space ARP daemon is configured, the notify flag is set to enable the kernel to notify it later.

```
847 #ifdef CONFIG_ARPD
848     notify = 1;
849 #endif
850 }
```

If the `state` has not changed there is nothing more to be done. If the state is now in the `NUD_CONNECTED` set `neigh_connect()` is called to setup the fast transmit path. Otherwise the slow path is setup by `neigh_suspect()`.

```
851     if (new == old)
852         goto out;
853     if (new&NUD_CONNECTED)
854         neigh_connect(neigh);
855     else
856         neigh_suspect(neigh);
```

Draining the *arp_queue*

If the old state was not one of the NUD_VALID states and the new state *is* one of the NUD_VALID states, there may be *sk_buffs* awaiting output on the neighbour's *arp_queue* which can now be successfully transmitted. These *sk_buffs* are dequeued and queue for transmission.

```
857     if (!(old & NUD_VALID)) {
858         struct sk_buff *skb;

860         /* Again: avoid dead loop if something went wrong */
861
862         while (neigh->nud_state & NUD_VALID &&
                (skb= __skb_dequeue(&neigh->arp_queue))
                != NULL) {
864             struct neighbour *nl = neigh;
865             write_unlock_bh(&neigh->lock);
866             /* On shaper/eql
                skb->dst->neighbour != neigh :( */
867             if (skb->dst && skb->dst->neighbour)
                nl = skb->dst->neighbour;
```

The output function of the neighbour is called here to push the packet on to the device after setting up the hardware header. And the *arp_queue* is purged after dequeuing them above, in case anything went wrong.

```
869                 nl->output(skb);
870                 write_lock_bh(&neigh->lock);
871             }
872             skb_queue_purge(&neigh->arp_queue);
873     }

874 out:
875     write_unlock_bh(&neigh->lock);
876 #ifdef CONFIG_ARPD
877     if (notify && neigh->parms->app_probes)
878         neigh_app_notify(neigh);
879 #endif
880     return err;
881 }
```

The *neigh_sync()* function -

The *neigh_sync()* function updates the value of *nud_state* based upon the *confirmed* and *reachable* times.

```
527 static void neigh_sync(struct neighbour *n)
528 {
529     unsigned long now = jiffies;
530     u8 state = n->nud_state;
531
532     ASSERT_WL(n);
533     if (state & (NUD_NOARP|NUD_PERMANENT))
534         return;
535     if (state & NUD_REACHABLE) {
536         if (now-n->confirmed > n->parms->reachable_time){
537             n->nud_state = NUD_STALE;
538             neigh_suspect(n);
539         }
540     } else if (state & NUD_VALID) {
541         if (now-n->confirmed < n->parms->reachable_time) {
542             neigh_del_timer(n);
543             n->nud_state = NUD_REACHABLE;
544             neigh_connect(n);
545         }
546     }
547 }
```