Inet Help Sheet For Minix

Please refer to http://www.nyx.net/~ctwong/minix/ first, which provides systematic explain on main functions in inet and some big pictures for it. Here I try to explain some undocumented questions for minix inet, which will be updated with time:

How inet bootup?

The entry of minix inet bootup is located in /usr/src/inet/inet.c, which is composed of two main parts:
1. nw_init()
   this is the networking initiation, including the following functions:
   a. read_conf(); before inet bootup, system need to know how many network interfaces have been equipped on, and what port number is assigned to them, all of which should been configured by user in /etc/inet.conf. The format is as following:
      eth0 LANCE 0 {default;};
      eth1 DP8390 1;
      ...
      eth0 means the first NIC, LANCE is the driver name, 0 is the port number, {default;} is to set eth0 as the default NIC. Only one NIC can be set to be default in system.
      eth1 means the second NIC, using DP8390 as its driver, occupying port 1;

      Based on configuration in /etc/inet.conf, system initialize the global variables eth_conf[], ip_conf[] and eth_conf_nr;

   b. sr_init(); It is used to initialize sr_fd_table[]. There are several important structures in inet, sr_fd_table[] is one of them, which plays the gateway to map system call from application level to kernel level. When using open("/dev/eth0") in application level, actually it calls eth_open() in kernel; while open("/dev/ip0"), it calls ip_open() in ip.c. Inet uses sr_fd_table to finish the mapping process. Let’s take open("/dev/eth0") as the example:
      Each device file in /dev has a major number and minor number, which can be checked by ls –l. (The detail explain for major and minor number can be found in http://www.nyx.net/~ctwong/minix/) In the following initiations, eth_init(), ip_init(), tcp_init(), udp_init(), all of them will call
      sr_add_minor(minor, port, openf, closef, readf, writef, ioctlf, cancelf)
      to register their system call function pointer to sr_fd_table[] as this way:
      sr_add_minor(eth_minor, i, eth_open, eth_close, eth_read, eth_write, eth_ioctl, eth_cancel);
      sr_add_minor(ip_minor, i, ip_open, ip_close, ip_read, ip_write, ip_ioctl, ip_cancel);
      sr_add_minor(tcp_minor, i, tcp_open, tcp_close, tcp_read, tcp_write, tcp_ioctl, tcp_cancel);
      sr_add_minor(udp_minor, i, udp_open, udp_close, udp_read, udp_write, udp_ioctl, udp_cancel);
      when open("/dev/eth0") in application level, inet will get /dev/eth0 minor number, eth_minor, as the index of sr_fd_table, call
      (*sr_fd_table[eth_minor]. srf_open)(…)
      which is actually point to eth_open(…)
c. eth_init(); The following graph describes the process of Ethernet initialization:

1. Eth_init()

2. Initialize
   - Eth_fd_table
   - Eth_port_table

3. osdep_eth_init()

4. Send(message)

5. lance_task()

6. do_init(&m)

7. mess_reply()

8. receive(message)

9. sr_add_minor()

10. setup_read
    - (eth_port)

11. send(message)

12. do_vread()

13. mess_reply()

14. receive(message)

15. eth_arrive()

---

(1) begin eth_init() in /usr/src/inet/generic/eth.c;
(2) initialize eth_fd_table[] and eth_port_table[]. Generally speaking, eth_fd_table[] contains the information of upper layer information, such as pointer to callback functions, or data buffer transferred between different layers, plays the gateway between ether and ip, arp, rarp; eth_port_table[] contains the Ethernet information, such as MAC address, buffer pointer transferred between ether and driver, port number etc. each item in eth_port_table corresponds to a NIC.
(3) Call osdep_eth_init() in /usr/src/inet/mnx_eth.c;
(4) Construct an init message, send to driver.
(5) Here we use amd lance as the NIC driver, so call lance_task() in /usr/src/kernel/lance.c. lance_task sits in an endless loop to get messages from eth.c, arp.c or hard interrupt, then process the message according to message type and forward message to the destination designated in original message.
(6) Lance_task() get a init message, then call do_init(&m), to initialize NIC and construct reply message containing MAC address of NIC.
(7) Send message back by mess_reply().
(8) Jump back to /usr/src/inet/mnx_eth.c, using receive(&m) to get message sent from driver, and set Ethernet address in eth_port_table[].
(9) Call sr_add_minor() in /usr/src/inet/sr.c to register Ethernet system call function pointer in sr_fd_table[].
(10) To get data from driver, ethernet will call setup_read(eth_port) in mnx_eth.c. Eth_port decides which NIC will be used to get data.
(11) Construct dev_read message, and send to driver lance_task() in /usr/src/kernel/lance.c.
(12) According to message type, dev_read, call do_vread() to check whether there are data in read buffer queue, if yes, transfer buffer data to upper layer.
(13) Send message back to original caller.
(14) Back to mnx_eth.c, call receive(&m) to get message sent from driver.
(15) Call eth_arrive() to indicate there are new packets arrive. Check ether header to decide where packets will be delivered to.
(16) In eth_init(), no packet gets from driver, so return to inet.c to continue next initialization.

d.  arp_init();
   (1) do nothing here, the real initialization of arp is in ip_init();

e.  ip_init();
   (1) begin ip_init() in /usr/src/inet/generic/ip.c;
   (2) initialize ip part of ip_port_table[];
   (3) initialize Ethernet part of ip_port_table[]
   (4) call eth_open() in eth.c to 1) get a slot index of eth_fd_table[] 2) register get_eth_data(), put_eth_data(), ip_eth_arrived() in eth_fd_table[];
   (5) in icmp_init(), initialize the icmp_port_table[];
   (6) since icmp is based on ip, icmp must register itself in ip_fd_table by ip_open(), including two call back functions: icmp_getdata(), icmp_putdata();
   (7) try to read icmp packets
   (9) calling ip_read() first
   (10) packet2user() is used to put data to upper layer
   (11) using bf_delhead() to delete ip header, so get icmp packets
   (12) initialize routing table. There are two routing tables in minix, oroute_table is used for find a entry to deliver outgoing packets; iroute_table is used for the incoming packets to find a entry to forward packets;
   (13) Call sr_add_minor() in /usr/src/inet/sr.c to register ip system call function pointers in sr_fd_table[];
   (14) Call (*ip_port->ip_dev_main)(ip_port), actually call ipeth_main() in ip_eth.c. Here is the interface between ip and Ethernet.
   (15) arp_set_cb() do the real job of arp initialization. Initialize arp_port_table[] first.
   (16) In arp_main(), finish two things: 1) call eth_open() to finish registration 2) get ethernet address, saving in arp_port_table[].
   (17) Since arp is based on Ethernet layer, it must register itself in eth_fd_table[], including two call function pointers, arp_getdata, arp_putdata;
   (18) try to read packets from Ethernet layer
   (19) calling eth_read()
f. tcp_init();
   (1) begin tcp_init() in /usr/src/inet/generic/tcp.c;
   (2) Initialize the tcp_fd_table, tcp_port_table;
   (3) Register tcp sys call into sr_fd_table;
   (4) tcp_main performs initialization routines depending on the state the tcp is in as determined
       by tcp_state.
   (5) Get index to ip channel array.
       tcp_get_data returns data specific to the tcp port table, used to be called in ip layer, to
       transfer data from tcp to ip;
       tcp_put_data put data (ip errors, ip ioctl calls or ip packets) from ip to tcp according to
       tcp_port->tp_state;
       tcp_put_pkt(fd, data, datalen) to do the real job to transfer ip data to tcp layer.
   (6) In tcp_main(), initialize the tcp connection table and call read_ip_packets(tcp_port) to get ip
       packets voluntarily;
   (7) Ip_read() will send packets saved in ip_fd->if_rdbuf_head back to call layer, here is tcp
       layer by calling packet2user();
   (8) Packet2user() actually send packets back to upper layer by calling the function pointer
       ip_fd->if_put_pkt, while it exactly points to tcp_put_pkt(fd, data, datalen).
2. Then inet will sit in a endless loop to get message and process it, the basic structure can be described as follows in the following c-like program:

```c
#include <minix/type.h>
#define TRUE 1

int main()
{
    init();
    while (TRUE)
```
{  
    message m;
    receive(&m);
    processmessage(&m);
  
}
How to send and receive packets from application layer to Ethernet layer (How ping.c works in minix)?

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping.c 61</td>
<td>hostent=gethostbyname(argv[1]);</td>
<td>Get host information, saving in hostent</td>
</tr>
<tr>
<td>Ping.c 73</td>
<td>dst_addr=<em>ipaddr_t</em>&gt;(hostent-&gt;h_addr);</td>
<td>Get host ip address</td>
</tr>
<tr>
<td>Ping.c 87</td>
<td>fd=open(&quot;/dev/ip&quot;, O_RDWR);</td>
<td>Try to open ip device</td>
</tr>
<tr>
<td></td>
<td>Trap into kernel</td>
<td></td>
</tr>
<tr>
<td>Src 214</td>
<td>PRIVATE int sr_open(message m)</td>
<td>Transfer message from application layer to kernel and execute corresponding system call</td>
</tr>
<tr>
<td>Src 243</td>
<td>fd= (*sr_fd-&gt;sr_open)(sr_fd-&gt;sr_port, i, sr_get_userdata, sr_put_userdata, 0);</td>
<td>Call function pointer to execute the real system call</td>
</tr>
<tr>
<td>Ip.c 191</td>
<td>ip_open (port, srfd, get_userdata, put_userdata, put_pkt)</td>
<td>Ip_open() is the real system call for open(&quot;/dev/ip&quot;) in application layer</td>
</tr>
<tr>
<td>Ip.c 231</td>
<td>Return i;</td>
<td>i is the slot index in ip_fd_table[]</td>
</tr>
<tr>
<td>Go back ping.c 91</td>
<td>[popt.nwio_flags= NWIO_COPY</td>
<td>NWIO_PROTOSPEC restricts communication to one IP protocol, specified in nwio_proto. NWIO_PROTOANY allows any protocol to be sent or received.</td>
</tr>
<tr>
<td>Ping.c 94</td>
<td>result=ioctl (fd, NWIOSPOPT, &amp;ipopt);</td>
<td>Set ip operation mode</td>
</tr>
<tr>
<td></td>
<td>Trap into kernel</td>
<td></td>
</tr>
<tr>
<td>Src 270</td>
<td>PRIVATE int sr_rwio(message m)</td>
<td></td>
</tr>
<tr>
<td>Src 346</td>
<td>r=(*sr_fd-&gt;sr_ioctl)(sr_fd-&gt;sr_fd, request);</td>
<td>Do the real system call ioctl()</td>
</tr>
<tr>
<td>Ip_ioctl.c 26</td>
<td>PUBLIC int ip_ioctl (fd, req)</td>
<td>Do the real system call ioctl()</td>
</tr>
<tr>
<td>Ip_ioctl.c 51</td>
<td>case NWIOSPOPT:</td>
<td>Get the old operation flags</td>
</tr>
<tr>
<td>Ip_ioctl.c 52</td>
<td>data=(*ip_fd-&gt;ip_get_userdata)(ip_fd-&gt;ip_srfd, 0, sizeof(nwio_ipopt_t), TRUE);</td>
<td>Actually call this function to get flag</td>
</tr>
<tr>
<td>Src 487</td>
<td>acc_t *sr_get_userdata (fd, offset, count, for_ioctl)</td>
<td>Setting default operation parameters then return</td>
</tr>
<tr>
<td>Ip_ioctl.c 55-156</td>
<td></td>
<td>Setting default operation parameters then return</td>
</tr>
<tr>
<td>Ping.c 98</td>
<td>result=ioctl (fd, NWIOGIPOPT, &amp;ipopt);</td>
<td>Get ip operation flag, saving in ipopt</td>
</tr>
<tr>
<td></td>
<td>Trap into kernel</td>
<td></td>
</tr>
<tr>
<td>Src 270</td>
<td>PRIVATE int sr_rwio(message m)</td>
<td></td>
</tr>
<tr>
<td>Src 346</td>
<td>r=(*sr_fd-&gt;sr_ioctl)(sr_fd-&gt;sr_fd, request);</td>
<td>Do the real system call ioctl()</td>
</tr>
<tr>
<td>Ip_ioctl.c 26</td>
<td>PUBLIC int ip_ioctl (fd, req)</td>
<td>Do the real system call ioctl()</td>
</tr>
<tr>
<td>Ip_ioctl.c 51</td>
<td>case NWIOGIPOPT:</td>
<td>Get the old operation flags</td>
</tr>
<tr>
<td>Ip_ioctl.c 165</td>
<td>result= (*ip_fd-&gt;ip_put_userdata)(ip_fd-&gt;ip_srfd, 0, data, TRUE);</td>
<td>Put ip operation setting back to &amp;ipopt, deferred in ping.c 98</td>
</tr>
<tr>
<td>Ping.c 104-112</td>
<td>result=write(fd, buffer, length);</td>
<td>Construct icmp and ip header</td>
</tr>
<tr>
<td>Ping.c 113</td>
<td>result=write(fd, buffer, length);</td>
<td>Send packet (buffer) out</td>
</tr>
<tr>
<td>Src 270</td>
<td>int sr_rwio(m)</td>
<td></td>
</tr>
<tr>
<td>Src 330</td>
<td>(*sr_fd-&gt;sr_write)(sr_fd-&gt;sr_fd, m-&gt;mq_msg.COUNT);</td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td><code>int ip_write(fd, count)</code></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td><code>ip_send(fd, pack, count);</code></td>
<td></td>
</tr>
<tr>
<td>129-136</td>
<td>Set ip header parameters</td>
<td></td>
</tr>
<tr>
<td>261</td>
<td>Check whether destination address is in the same sub network of its own ip address, if so, go to following steps; if not, call oroute_frag(ip_port - ip_port_table, dstaddr, ttl, &amp;nexthop); in line 271, to check whether there is a route in routing table. Here I using same network address as the example.</td>
<td></td>
</tr>
<tr>
<td>265</td>
<td>Calculate the broadcast address in the sub net</td>
<td></td>
</tr>
<tr>
<td>266</td>
<td>Send packet to Ethernet layer</td>
<td></td>
</tr>
<tr>
<td>279</td>
<td>This is the real function to be called, the registration is made in lp_eth.c 73, ip_port-&gt;ip_dev_send= ipeth_send; in inet initialization</td>
<td></td>
</tr>
<tr>
<td>297-306</td>
<td>Add ether header to data, now the packet becomes a link list as: ether header-&gt;ip header-&gt;icmp packet</td>
<td></td>
</tr>
<tr>
<td>309</td>
<td>To decide broadcast the packets or check arp cache to find the destination MAC</td>
<td></td>
</tr>
<tr>
<td>326</td>
<td>To check whether could find a entry in arp cache</td>
<td></td>
</tr>
<tr>
<td>676</td>
<td>Check arp cache</td>
<td></td>
</tr>
<tr>
<td>693</td>
<td>If find a entry, return to ip_eth.c, else allocate a cache entry by calling setup_write() in 735</td>
<td></td>
</tr>
<tr>
<td>699</td>
<td>If we have no write in progress, we can try to send the ethernet packet using eth_send. If the IP packet is larger than mss, unqueue the packet and let ipeth_restart_send deal with it.</td>
<td></td>
</tr>
<tr>
<td>364</td>
<td>Since packet is less than message size, call eth_send() to send packet out</td>
<td></td>
</tr>
<tr>
<td>395</td>
<td>Write packet to specific port</td>
<td></td>
</tr>
<tr>
<td>455</td>
<td>Translate packet address to ivector address</td>
<td></td>
</tr>
<tr>
<td>114-133</td>
<td>Construct message which contains the packet buffer address</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>Send message to driver task with the type of DL_WRITE</td>
<td></td>
</tr>
<tr>
<td>272</td>
<td>NIC driver</td>
<td></td>
</tr>
<tr>
<td>308</td>
<td>Call do_vwrite according to message type</td>
<td></td>
</tr>
<tr>
<td>1198</td>
<td>Copy write_iovec to the slot on DMA address</td>
<td></td>
</tr>
</tbody>
</table>
How an Udp program works?
To simplify the problem, I create two simple udp programs, talker and listener. listener is to listen on an udp port and display incoming messages sent by talker.c. Comments in code will help you to understand how to develop a simple udp program. Here I only trace an udp packet to show how it is sent out and received by peer.

Launch listener as: ./use/messager/listener

Launch listener as: ./use/messager/listener

| Listener.c 59 | if((udp_fd = open(udp_devic.e, O_RDWR)) <0) | Open a udp device, which points to /dev/udp
| Udp.c 232 | udp_open (port, sched, get_userdata, put_userdata, put_pkt) | To get a slot in udp_fd_table[], register get_userdata() and put_userdata in it, which are the main function to transfer data between udp and ip layer.
| Listener.c 77 | s = ioctl(udp_fd, NWIODUPOPT, &udpopt); | Call ioctl to set udp operation flags
| Udp.c 448-450 | result = udp_setopt(udp_fd); | The definition of udp_setopt(udp_fd) in Udp.c
| Udp.c 473-671 | The definition of udp_setopt(udp_fd) | Get udp operation flag, after verification, assign new operation flag to udp_fd->uf_userdata. In udp 486 (*udp_fd->uf_get_userdata)

| Lance.c 1226 | out_word(ioaddr+LANCE_ADDR, 0x0000); out_word(ioaddr+LANCE_DATA, 0x0048); | Send packets out

| Inet.c 140 | eth_rec(&mq->mq_mess); | Get message from NIC driver
| Max_eth.c 263 | read_int(loc_port, m->DL_COUNT); | Get icmp reply
| Max_eth.c 397 | eth_arrive(eth_port, cut_pack, count); | Report packet arrived
| Eth.c 781 | packet2user(eth_fd, pack, exp_time); | Send packets to ip layer
| Eth.c 861 | result = (*eth_fd->ef_put_userdata)(eth_fd->ef_srfd, size_tid, pack, FALSE); | Put user data to ip layer by calling function pointer
| Ip_eth.c 221 | put_eth_data(port, offset, data, for_ioctl); | The real function which is called
| Ip_eth.c 261 | ip_eth_arrived(port, data, bf_bufsize(data)); | Tell system ip packet has arrived, delete ethernet header
| Ip_eth.c 699 | ip_arrived(ip_port, pack); | Verify packets format
| Ip_read.c 599 | ip_port_arrive(ip_port, pack, ip_hdr); | Find the destination is the same as host address, reassemble packets
| Ip_read.c 414 | result = (*ip_fd->if_put_userdata)(ip_fd->if_srfd, size_tid, pack, FALSE); | Call function pointer to send packets to upper layer
| Ip_read.c 502 | packet2user(ip_fd, pack, exp_time); | Put packets to upper layer
| Icmp.c 222 | icmp_puddata(port, offset, data, for_ioctl); | The real function to be called
| Icmp.c 252 | proc_data(icmp_port, data); | To analyze the packet, get icmp data to see what type is it
| Icmp.c 406 | case ICMP_TYPE_ECHO_REPL: | Find it is a echo reply packet, do nothing, free the data
| Ping.c 130 | result = read(fd, buffer, sizeof(buffer)); | The write is over, then check whether could get reply from ethernet
| Inet.c 130 | fr_rec(mq); | Get request from filesystem, then map read() system call to ip_read()
| Ip_read.c 32 | ip_read(fd, count) | Get icmp echo reply packet from ip_fd->if_rdbuf_head, means reply ok, return;
| Ping.c 144 | printf("%s is alive\n", argv[1]); | Ping is over
`s = read(udp_fd, buf, sizeof(buf));`  
Try to get data in udp_fd->uf_rdbuf_head, if no udp packets received, udp_fd->uf_rdbuf_head is null and return NW_SUSPEND,

`sr_reply(m, result, FALSE);`  
Send replay to file system that read system call is suspending, then read() in listener.c is blocked here, waiting for incoming udp packets

Launch talker as: talker 192.168.163.122 “this is a test message”. 192.168.163.122 is the ip address of listener machine

```
if((udp_fd = open(udp_device, O_RDWR)) < 0)
Open a udp device /dev/udp to get a slot in udp_fd_table[], the same as listener
```

```
s = ioctl(udp_fd, NWIOSUDPOPT, &udpopt);
Set udp operation flags, the same as listener
```

```
s = write(udp_fd, buf, numbytes);
Write buf data to udp_fd, send data to remote listener defined in udpopt
```

```
udp_write(fd, count)
Verify and set udp_fd->uf_flags, call restart_write_fd(udp_fd) to do the real write
```

```
1. line 1201 (*udp_fd->uf_get_userdata)() get data from user space to buffer, actually call sr.c line 543 cp_u2b((*head_ptr)->mq_mess.PROC_NR, src, &acc, count)
2. add udp and ip header before data, as last wrap data as a link list: ip header ->udp header ->data
3. line 1324 ip_write() to write packet to ip layer
```

```
ip_write(fd, count)
1. line 40 (*ip_fd->if_get_userdata()) get data from udp layer, actually call udp_get_data() in udp.c 266. The function return in line 341 at return bf_cut (udp_port->up_wr_pack, offset, count);
2. line 44 call r= ip_send(fd, pack, count) to do real sending job
```

```
ip_send(fd, pack, count)
1. to verify and initialize ip header
2. to route packets by checking whether the destination ip address and local ip are in same network
3. if in same network, call r= (*ip_port->ip_dev_send) in line 266, which actually points to ip_eth.c 279 ipeth_send(ip_port, dest, pack, broadcast)
4. if not in same network, call oroute_frag () to find route in out routing table, then send packets by ipeth_send()
```

```
ip_eth.c 279-403
```

```
1. add ether header before ip packet;
2. get a arp entry to tell packet the destination MAC
```

```
ip_eth.c 405-496
```

```
1. to big packet, split packet to small ones and add ether header to each of them;
2. call eth_send() or eth_write() to send packets out
```

```
eth_send(fd, data, data_len)
1. to initialize the ether header
```

```
Udp.c 762-793
```

```
udp_read(fd, count)
```

```
Sr.c 177-180
```

```
sr_reply(m, result, FALSE);
```

```
Listener.c 87
```

```
s = read(udp_fd, buf, sizeof(buf));
```

```
Udp.c 1134-1335
```

```
restart_write_fd(udp_fd)
```

```
```
Ip_write.c 26-52
```

```
```
Ip_write.c 54-302
```

```
ip_eth.c 395-457
```

```
```
90x728

```
```
90x712

```
```
87x52

```
```
329x759

```
```
329x744

```
```
329x712

```
```
90x712

```
```
90x701

```
```
90x585

```
```
90x569

```
```
90x553

```
```
90x537

```
```
90x505

```
```
90x490

```
```
90x474

```
```
90x473

```
```
90x458

```
```
90x452

```
```
90x380

```
```
90x379

```
```
90x357

```
```
90x333

```
```
90x317

```
```
90x285

```
```
90x285

```
```
90x263

```
```
90x254

```
```
90x239

```
```
90x223

```
```
90x207

```
```
90x180

```
```
90x160

```
```
90x145

```
```
90x128

```
```
90x105

```
```
90x81

```
```
1. call eth_write_port(eth_port, eth_pack) for further forwarding

**Mnx_eth.c 98-232**

Construct message and send it to driver, driver will forward packets to destination. As to detail information about driver, please refer to "how ping.c works"

Listener will get the packet sent out if the port number and destination ip are match, following is how listener process message when it get message

**Inet.c 140**

Get message from driver, which tell system some packs have arrived

**Mnx_eth.c 234-264**

1. search which Ethernet port should be delivered;
2. according to m->DL_STAT to decide read input or write input. Here call read_int(loc_port, m->DL_COUNT) at 263

**Mnx_eth.c 385-402**

1. check whether there is packet which read before, if so send to up layer;
2. call setup_read() to get data from driver

**Mnx_eth.c 403-506**

1. construct message and data space, sending to driver then get data back from it;
2. call eth_arrive() sending packets to upper layer

**Eth.c 699-816**

1. check ether header to see whether it's the packet sent to itself;
2. call packet2user() to send packets to upper layer

**Eth.c 820-870**

1. to verify the validation of ether packets
2. call (*eth_fd->ef_put_userdata)() to put data to ip layer, actually call put_eth_data (port, offset, data, for_ioctl) in ip_eth.c

**Ip_eth.c 221-270**

Call ip_eth_arrived() to tell ip layer new packets arrived

**Ip_eth.c 683-700**

1. delete ether header;
2. call ip_arrived(ip_port, pack) to send packet

**Ip_read.c 540-725**

1. check ip packets size and fragmentation
2. if it's a local forwarding, call ip_port_arrive ();
3. if destination address is not the same as local address, check in routing table to see whether could find a entry to forward packet again. This option is used when make a minix as router;

**Ip_read.c 429-538**

1. check fragmentation and reassemble the packets;
2. do verification;
3. call packet2user(first_fd, pack, exp_time) to send packets to upper layer.

**Ip_read.c 351-428**

1. if *ip_fd->if_flags is not set to IFF_READ_IP, just copy data to ip_fd->if_rbuff_head, waiting for a ip read request;
2. if set, call (*ip_fd->if_put_userdata()) in line 414 to send packet to upper layer

**Udp.c 354-423**

Since udp_port->up_flags has been set UPF_READ_IP, go to line 412 call udp_ip_arrived().
How a tcp program works?

I create two simple tcp programs, client and server, which use emulation socket lib developed by Claudio Tantignone. Code in socket.c is clear and simple, which is very useful to learn what’s the main purpose of these system calls in minix, such as ioctl(), read()… Especially for those students who are very familiar with socket, to learn it is really easy. Since most of the procedures under ip layer are almost same to tcp and udp, I only explain those working on ip layer.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>udp.c 855-1112</td>
<td>udp_ip_arrived(port, pack, pack_size)</td>
</tr>
<tr>
<td></td>
<td>delete ip head;</td>
</tr>
<tr>
<td></td>
<td>verify udp header;</td>
</tr>
<tr>
<td></td>
<td>add udp io header, which contains some information about ip layer, such as</td>
</tr>
<tr>
<td></td>
<td>source ip, port and des ip, port;</td>
</tr>
<tr>
<td></td>
<td>call udp_rd_enqueue() to copy udp packets to udp_fd-&gt;uf_rdbuf_head</td>
</tr>
<tr>
<td></td>
<td>since listener is still suspending on read() system call, the udp_fd-&gt;uf</td>
</tr>
<tr>
<td></td>
<td>flags is still UFF_READ_IP, then call udp_packet2user(user_fd) at line 1083;</td>
</tr>
<tr>
<td>udp.c 795-853</td>
<td>udp_packet2user(udp_fd)</td>
</tr>
<tr>
<td></td>
<td>Call (*udp_fd-&gt;uf_put_userdata()) at line 848 to put data to user layer,</td>
</tr>
<tr>
<td></td>
<td>actually call sr_put_userdata() at line 548 in sr.c.</td>
</tr>
<tr>
<td>Sr.c 548-606</td>
<td>sr_put_userdata(fd, offset, data, for_ioctl)</td>
</tr>
<tr>
<td></td>
<td>Return cp_b2u(data, (*head_ptr)-&gt;mq_mess.PROC_NR, dst) to file system, so</td>
</tr>
<tr>
<td></td>
<td>listener get data from read system call.</td>
</tr>
<tr>
<td>Listener.c 103-104</td>
<td>udp_io_hdr =((udp_io_hdr_t *)buf);</td>
</tr>
<tr>
<td></td>
<td>s = s - sizeof(udp_io_hdr_t);</td>
</tr>
<tr>
<td></td>
<td>Remove udp_io_hdr to get the real data</td>
</tr>
<tr>
<td>Listener.c 112</td>
<td>printf(&quot;Listener: from %s, %u\n&quot;, inet_ntoa(udp_io_hdr-&gt;uih_src_addr),</td>
</tr>
<tr>
<td></td>
<td>ntohs(udp_io_hdr-&gt;uih_src_port));</td>
</tr>
<tr>
<td></td>
<td>Show src address and port in udp_io_hdr.</td>
</tr>
<tr>
<td></td>
<td>All set.</td>
</tr>
</tbody>
</table>

Launch server as: /usr/messager/server

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server.c 46</td>
<td>if ((sockfd = socket(AF_INET, 0, 0)) == -1)</td>
</tr>
<tr>
<td></td>
<td>Create a socket file descriptor, actually call mnx_socket(int proto) in</td>
</tr>
<tr>
<td></td>
<td>socket.c</td>
</tr>
<tr>
<td>Socket.c 48</td>
<td>if ((fd = open(device, O_RDWR)) &lt; 0)</td>
</tr>
<tr>
<td></td>
<td>Return a tcp device file descriptor</td>
</tr>
<tr>
<td>Server.c 56</td>
<td>bind(sockfd, (struct sockaddr *)&amp;my_addr, sizeof(struct sockaddr))</td>
</tr>
<tr>
<td></td>
<td>Bind sockfd to server’s address, my_addr</td>
</tr>
<tr>
<td>Socket.c 246-292</td>
<td>int mnx_bind(int fd, struct sockaddr *addr)</td>
</tr>
<tr>
<td></td>
<td>1. construct a tcpconf and initialize it;</td>
</tr>
<tr>
<td></td>
<td>2. call ioctl(fd, NWIOD, TCPCONF, &amp;tcpconf) to set operation flag, which</td>
</tr>
<tr>
<td></td>
<td>actually call tcp_ioctl(fd, req), line 651-660;</td>
</tr>
<tr>
<td>Tcp.c 751-938</td>
<td>tcp_setconf(tcp_fd)</td>
</tr>
<tr>
<td></td>
<td>Do verification of tcpconf, and saving the configuration in tcp_fd-&gt;tf_tcpconf;</td>
</tr>
<tr>
<td>Server.c 61</td>
<td>if (listen(sockfd, 0) == -1)</td>
</tr>
<tr>
<td></td>
<td>Listening on a port number configured</td>
</tr>
<tr>
<td>Socket.c 214</td>
<td>mnx_listen(int fd)</td>
</tr>
<tr>
<td></td>
<td>Add a new flag in by tcpopt.nwto_flags</td>
</tr>
<tr>
<td></td>
<td>option delays a failure response on a connect to the same port as the</td>
</tr>
<tr>
<td></td>
<td>current open connection. Without this option a connect would fail if a</td>
</tr>
<tr>
<td></td>
<td>server is not yet listening. With this option a connect will linger on</td>
</tr>
<tr>
<td></td>
<td>until the server starts listening.</td>
</tr>
<tr>
<td>Server.c 70</td>
<td>new_fd = accept(sockfd, (struct sockaddr)</td>
</tr>
<tr>
<td></td>
<td>Actually call mnx_accept() in socket.c</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>mnx_accept()</code></td>
<td>Accept a connection on a new socket channel</td>
</tr>
<tr>
<td><code>ioctl(fd, NWIOGTCPCONF, &amp;tcpconf);</code></td>
<td>Get operation flags of present socket</td>
</tr>
<tr>
<td><code>ioctl(chan, NWIOGTCPCONF, &amp;tcpconf);</code></td>
<td>Assign operation flags to new socket channel</td>
</tr>
<tr>
<td><code>ioctl(chan, NWIOTCPLISTEN, &amp;tcplistenopt);</code></td>
<td>Make new socket listen</td>
</tr>
<tr>
<td><code>ioctl(chan, NWIOGTCPCONF, &amp;tcpconf2);</code></td>
<td>Get connection information</td>
</tr>
<tr>
<td><code>write(new_fd, buf, strlen(buf);</code></td>
<td>Write packets to remote peer</td>
</tr>
<tr>
<td><code>tcp_write(fd, count)</code></td>
<td>Send packet out by a tcp port</td>
</tr>
<tr>
<td><code>tcp_conn_write(tcp_conn, enq)</code></td>
<td>Call <code>ip_write(fd, count)</code> to send packets to ip layer</td>
</tr>
</tbody>
</table>

**How add_route.c, pr_routes.c works in application level and kernel level?**

`add_route` is a command to add/del static route in routing table. When inet server bootup, during ip initialization in `/usr/src/inet/generic/ip.c` line 130, call `ipr_init()`, in which a routing table array is created and maintained during the lifetime of the inet server. `add_route` take advantage of IO control system call to add/del the entries in routing table array. Let’s see how it works step by step:

**a. Create routing table array**

From line 38 to 41 in `/usr/src/inet/generic/ipr.c`, create static out routing table. There are two routing table in minix system: out routing table is used for the output packets; in routing table is used for forwarding packets to other machines, which is only used when minix acts as a router or gateway.

```
PRIVATE oroute_t oroute_table[OROUTE_NR];
PRIVATE oroute_t *oroute_head;
PRIVATE int static_oroute_nr;
PRIVATE oroute_hash_t oroute_hash_table[OROUTE_HASH_NR][OROUTE_HASH_ASS_NR];
```

**b. If add a route in routing table, using command**

```
add_route –d 192.168.2.3 –g 192.168.163.2
```

Let’s see how the command is executed:

<table>
<thead>
<tr>
<th>Code Snippet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (strcmp(prog_name, &quot;add_route&quot;) == 0)</code></td>
<td>According to program’s name, decide it should add route in routing table</td>
</tr>
<tr>
<td><code>action=ADD;</code></td>
<td>Analyze the parameters set in command line. Since we set <code>-d 192.168.2.3</code>, string “192.168.2.3” is assigned to <code>destination_str</code>, string “192.168.163.2” is assigned to <code>gateway_str</code>.</td>
</tr>
</tbody>
</table>
Src\lib\ip\inet_ntoa.c
char * inet_ntoa(ipaddr_t in)
//Convert network-format internet address to base 256 d.d.d.d representation.

Src\lib\ip\inet_addr.c
ipaddr_t inet_addr(char *cp)
// Ascii internet address interpretation routine. The value returned is in network order.

Int inet_aton(char *cp, ipaddr_t *addr)
//Check whether "cp" is a valid ascii representation of an Internet address and convert to a binary address.
Returns 1 if the address is valid, 0 if not. This replaces inet_addr, the return value from which cannot distinguish between failure and a local broadcast address.

inet/inet_config.c
void read_conf(void)
This is the first step to start network. Checking /etc/inet.conf in system, which is usally configured as following format:
Eth0 LANCE 0 {default;};
Eth1 LANCE 1;
LANCE is the name of driver, which is defined in include\minix\com.h. 0/1 is the port number, that means one NIC can only occupy one port number. “default” can only be set on one NIC.