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:

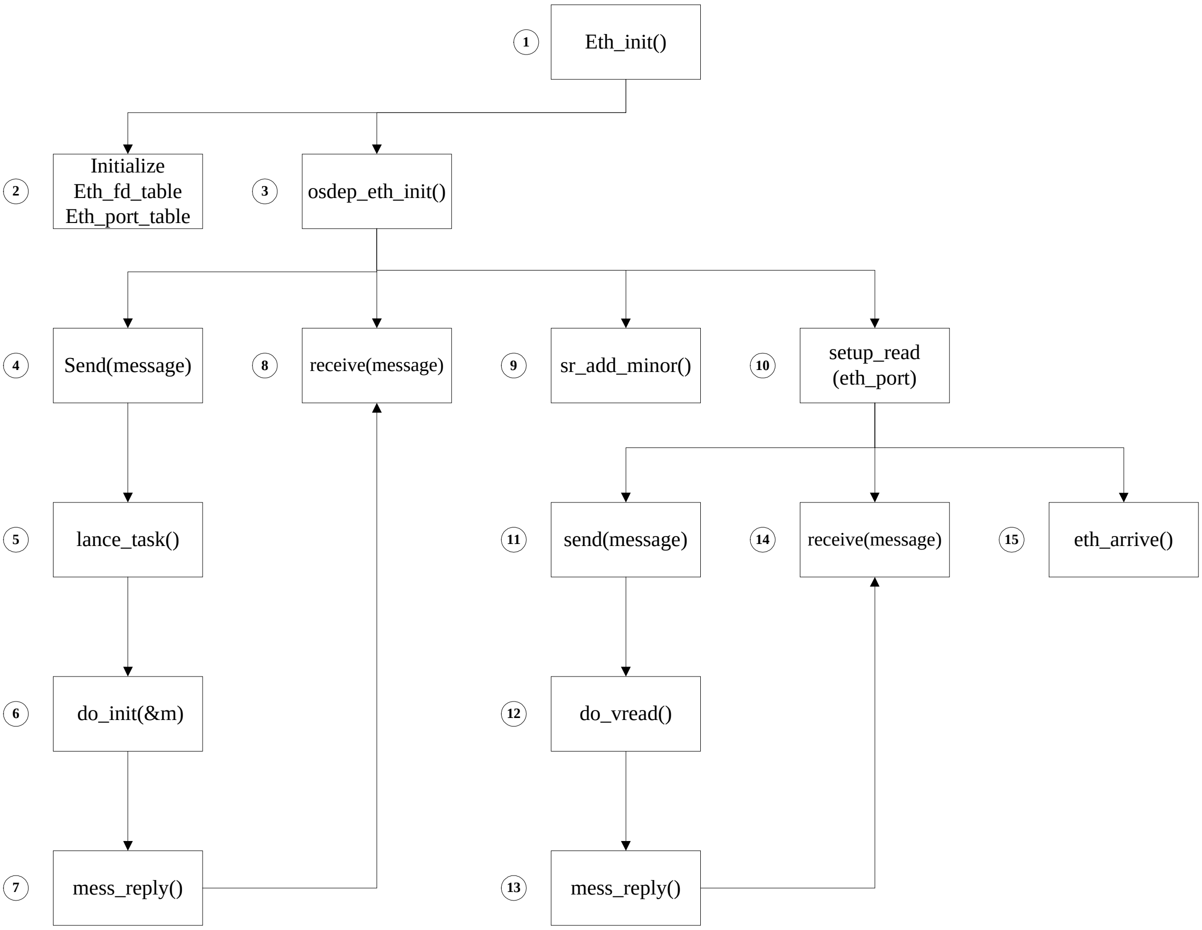
```
sr_add_minor(eth_minor,i, eth_open, eth_close, eth_read, eth_write, eth_ioctl, eth_cancel);
sr_add_minor(ip_minor, i, ip_open, ip_close, ip_read,  ip_write, ip_ioctl, ip_cancel);
sr_add_minor(tcp_minor,i, tcp_open, tcp_close, tcp_read,tcp_write, tcp_ioctl, tcp_cancel);
sr_add_minor(udp_minor, i, 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 describe the process of Ethernet initialization:



(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 a NIC.

(3) Call osdep_eth_init() in /usr/src/inet/mnx_eth.c;

(4) Construct a 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 a 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) Accoding 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) s

(7) 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();

(8) 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.

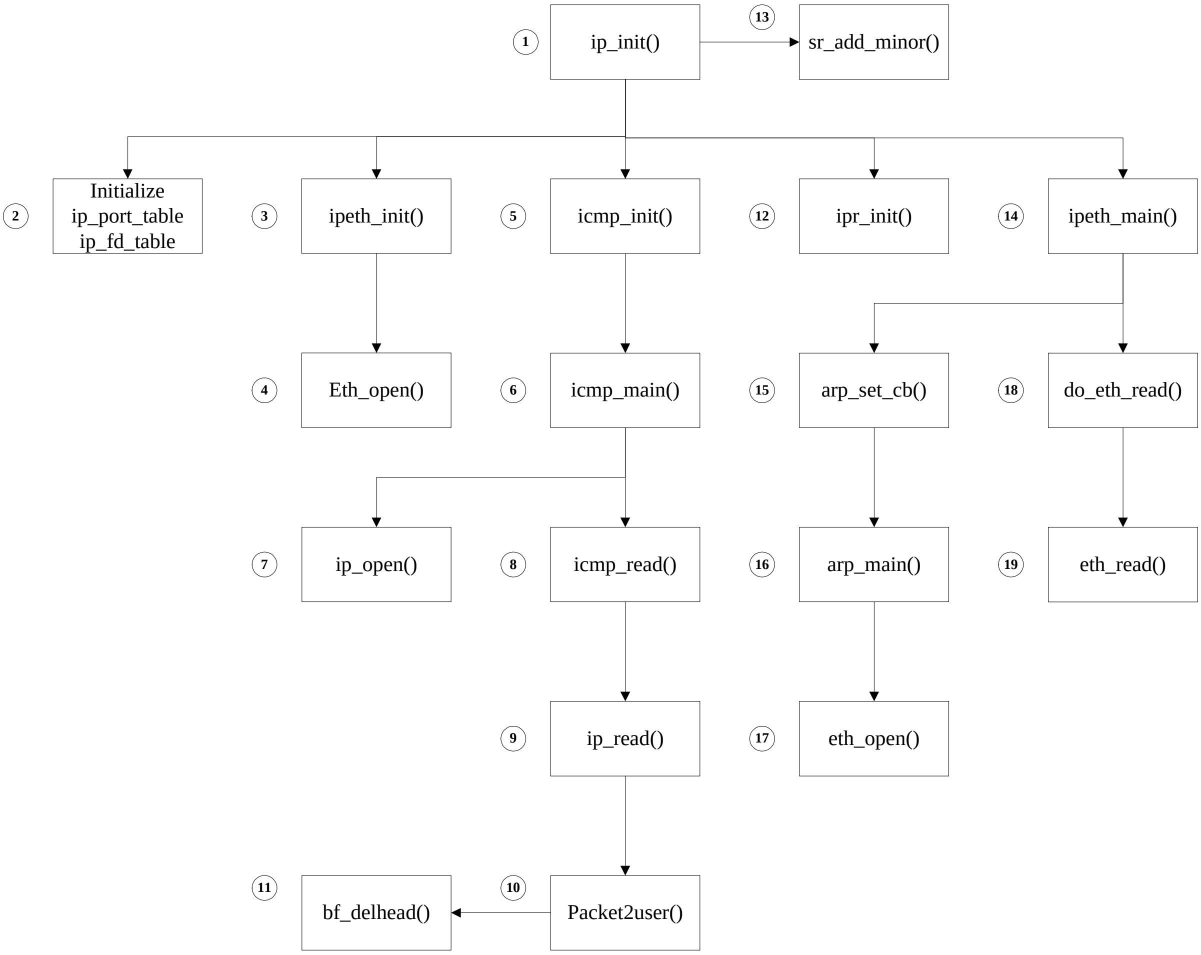
(15) arp_set_cb() do the real job of arp initialization. Initialize arp_port_table[] first.

(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 system 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;

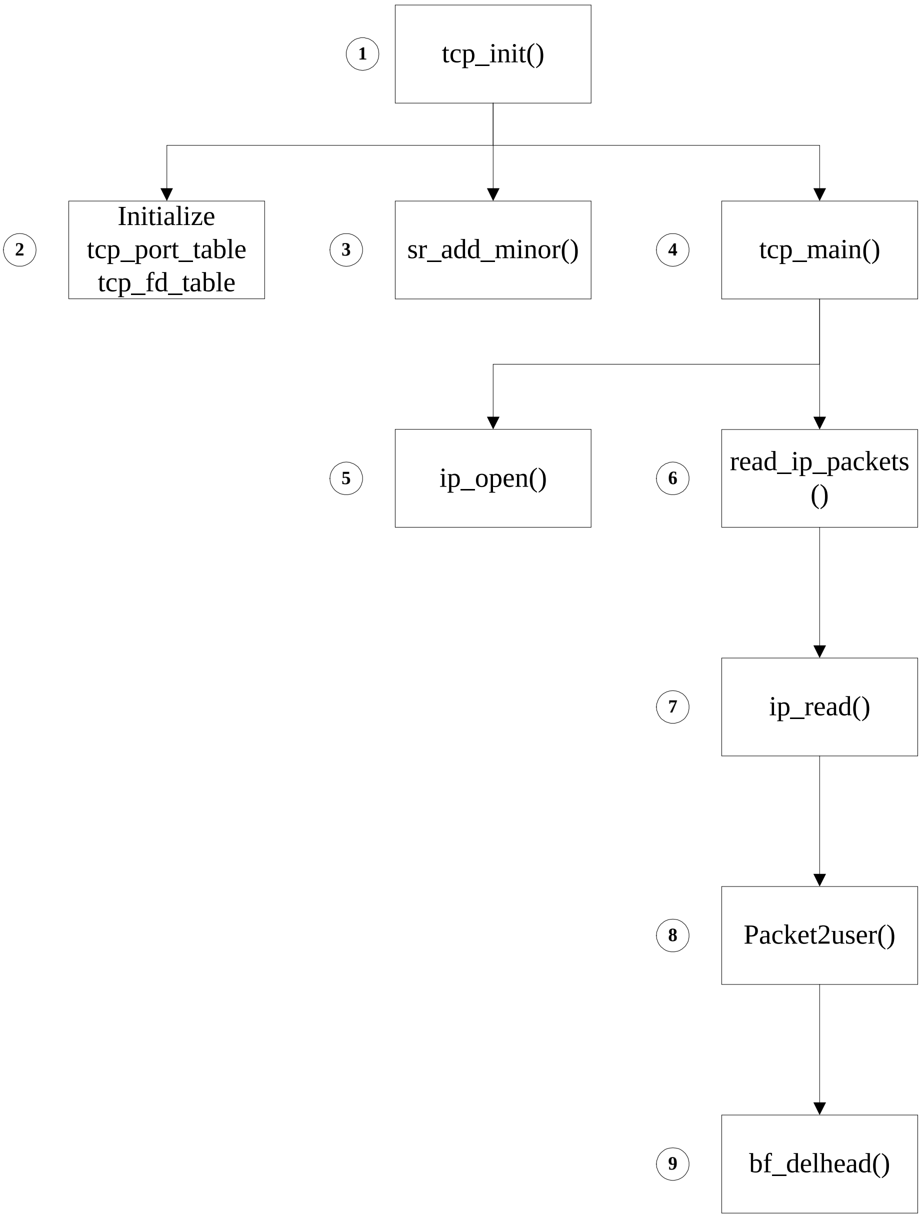
tcp_put_data put data (ip errors, ip ioctl calls or ip packets) from ip to tcp according to tcp_port->tp_state;

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).



g. udp_init();
almost the same as tcp_init(), please refer to http://www.nyx.net/~ctwong/minix/Minix_udp.htm.

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:

```
#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)?

Ping.c 61	hostent= gethostbyname(argv[1]);	Get host information, saving in hostent
Ping.c 73	dst_addr= *(ipaddr_t *)(hostent->h_addr);	Get host ip address
Ping .c 87	fd= open ("/dev/ip", O_RDWR);	Try to open ip device
Trap into kernel		
Sr.c 214	PRIVATE int sr_open(message m)	Transfer message from application layer to kernel and execute corresponding system call
Sr.c 243	fd= (*sr_fd->srf_open)(sr_fd->srf_port, i, sr_get_userdata, sr_put_userdata, 0);	Call function pointer to execute the real system call
Ip.c 191	ip_open (port, srfd, get_userdata, put_userdata, put_pkt)	Ip_open() is the real system call for open(“dev/ip”) in application layer
Ip.c 231	Return i	i is the slot index in ip_fd_table[]
Go back ping.c 91	ipopt.nwio_flags= NWIO_COPY | NWIO_PROTOSPEC;	NWIO_PROTOSPEC restricts communication to one IP protocol, specified in nwio_proto. NWIO_PROTOANY allows any protocol to be sent or received.
Ping.c 94	result= ioctl (fd, NWIOSIPOPT, &ipopt);	Set ip operation mode
Trap into kernel		
Sr.c 270	PRIVATE int sr_rwio(message m)	
Sr.c 346	r= (*sr_fd->srf_ioctl)(sr_fd->srf_fd, request);	
Ip_ioctl.c 26	PUBLIC int ip_ioctl (fd, req)	Do the real system call ioctl()
Ip_ioctl.c 51	case NWIOSIPOPT:	
Ip_ioctl.c 52	data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, 0, sizeof(nwio_ipopt_t), TRUE);	Get the old operation flags
Sr.c 487	acc_t *sr_get_userdata (fd, offset, count, for_ioctl)	Actually call this function to get flag
Ip_ioctl.c 55-156		Setting default operation parameters then return
Ping.c 98	result= ioctl (fd, NWIOGIPOPT, &ipopt);	Get ip operation flag, saving in ipopt
Trap into kernel		
Sr.c 270	PRIVATE int sr_rwio(message m)	
Sr.c 346	r= (*sr_fd->srf_ioctl)(sr_fd->srf_fd, request);	
Ip_ioctl.c 26	PUBLIC int ip_ioctl (fd, req)	Do the real system call ioctl()
Ip_ioctl.c 51	case NWIOGIPOPT:	
Ip_ioctl.c 165	result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0, data, TRUE);	Put ip operation setting back to &ipopt, deferred in ping.c 98
Ping.c 104-112		Construct icmp and ip header
Ping.c 113	result= write(fd, buffer, length);	Send packet (buffer) out
Sr.c 270	int sr_rwio(m)	
Sr.c 330	(*sr_fd->srf_write)(sr_fd->srf_fd, m->mq_mess.COUNT);	

Ip_write.c 26	int ip_write (fd, count)	Make the real write() system call
Ip_write.c 44	ip_send(fd, pack, count);	
Ip_write.c 129-136		Set ip header parameters
Ip_write.c 261	if (((dstaddr ^ ip_port->ip_ipaddr) & ip_port->ip_subnetmask) == 0)	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, &nexthop); in line 271, to check whether there is a route in routing table. Here I using same network address as the example.
Ip_write.c 265	broadcast= (dstaddr == (ip_port->ip_ipaddr | ~ip_port->ip_subnetmask));	Calculate the broadcast address in the sub net
Ip_write.c 266	r= (*ip_port->ip_dev_send)(ip_port, dstaddr, data,broadcast);	send packet to Ethernet layer
Ip_eth.c 279	int ipeth_send(ip_port, dest, pack, broadcast)	This is the real function to be called, the registration is made in Ip_eth.c 73, ip_port->ip_dev_send= ipeth_send; in inet initialization
Ip_eth.c 297-306		Add ether header to data, now the packet becomes a link list as: ether header->ip header->icmp packet
Ip_eth.c 309	if (broadcast) eth_hdr->eh_dst= broadcast_ethaddr;	To decide broadcast the packets or check arp cache to find the destination MAC
Ip_eth.c 326	r= arp_ip_eth(ip_port->ip_dl.dl_eth.de_port, dest, &eth_hdr->eh_dst);	To check whether could find a entry in arp cache
Arp.c 676	arp_ip_eth (eth_port, ipaddr, ethaddr)	
Arp.c 693	ce= find_cache_ent (arp_port, ipaddr);	Check arp cache
Arp.c 699		If find a entry, return to ip_eth.c, else allocate a cache entry by calling setup_write() in 735
Ip_eth.c 356		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.
Ip_eth.c 364	r= eth_send(ip_port->ip_dl.dl_eth.de_fd, eth_pack, pack_size);	Since packet is less than message size, call eth_send() to send packet out
Eth.c 395	int eth_send(fd, data, data_len)	
Eth.c 455	eth_write_port(eth_port, eth_pack);	Write packet to specific port
Mnx_eth.c 98	void eth_write_port(eth_port, pack)	
Mnx_eth.c 114-133		Translate packet address to ivector address
Mnx_eth.c 137-154		Construct message which contains the packet buffer address
Mnx_eth.c 158	r= send (eth_port->etp_osdep.etp_task, &mess1);	Send message to driver task with the type of DL_WRITE
Lance.c 272	void lance_task()	NIC driver
Lance.c 308	case DL_WRITE: do_vwrite(&m, FALSE, FALSE); break;	Call do_vwrite according to message type
Lance.c 1198	ec_user2nic(ec, &ec->write_iovec, 0, (int)(lp->tbuf[tx_slot_nr]), ec->write_s);	copy write_iovec to the slot on DMA address

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
Mnx_eth.c 263	read_int(loc_port, m->DL_COUNT);	Get icmp reply
Mnx_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_t)0, 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_t)0, 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_putdata(port, offset, data, for_ioctl)	The real function to be called
Icmp.c 252	process_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	sr_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

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: /usr/messager/listener		
Listener.c 59	if((udp_fd = open(udp_device, O_RDWR)) < 0)	Open a udp device, which points to “/dev/udp”
Udp.c 232	udp_open (port, srfd, 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, NWIOSUDPOPT, &udpopt);	Call ioctl to set udp operation flags
Udp.c 448-450	result= udp_setopt(udp_fd);	
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_udpopt. In udp 486 (*udp_fd->uf_get_userdata)

		actually point to sr.c 487 sr_get_userdata (fd, offset, count, for_ioctl)
Listener.c 87	s = read(udp_fd, buf, sizeof(buf));	
Udp.c 762-793	udp_read (fd, count)	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.c 177-180	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 listner machine		
Talker.c 72	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
Talker.c 90	s = ioctl(udp_fd, NWIOSUDPOPT, &udpopt);	Set udp operation flags, the same as listener
Talker.c 101	s = write(udp_fd, buf, numbytes);	Write buf data to udp_fd, send data to remote listener defined in udpopt
Udp.c 1138-1172	udp_write(fd, count)	Verify and set udp_fd->uf_flags, call restart_write_fd(udp_fd) to do the real write
Udp.c 1174-1335	restart_write_fd(udp_fd)	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 hearder ->udp header ->data 3. line 1324 ip_write() to write packet to ip layer
Ip_write.c 26-52	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_write.c 54-302	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	ipeth_send(ip_port, dest, pack, broadcast)	1. add ether header before ip packet; 2. get a arp entry to tell packet the destination MAC
Ip_eth.c 405-496	ipeth_restart_send(ip_port)	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.c 395-457	eth_send(fd, data, data_len)	1. to initialize the ether header

		2. call eth_write_port(eth_port, eth_pack) for further fawarding
Mnx_eth.c 98-232	eth_write_port(eth_port, pack)	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	eth_rec(&mq->mq_mess);	get message from driver, which tell system some packs have arrived
Mnx_eth.c 234-264	eth_rec(m)	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	read_int(eth_port, count)	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	setup_read(eth_port)	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	eth_arrive (eth_port, pack, pack_size)	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	packet2user (eth_fd, pack, exp_time)	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	put_eth_data (port, offset, data, for_ioctl)	Call ip_eth_arrived() to tell ip layer new packets arrived
Ip_eth.c 683-700	ip_eth_arrived(port, pack, pack_size)	1. delete ether header; 2. call ip_arrived(ip_port, pack) to send packet
Ip_read.c 540-725	ip_arrived(ip_port, pack)	1. check ip packets size and fragmention 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	ip_port_arrive (ip_port, pack, ip_hdr)	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	packet2user (ip_fd, pack, exp_time)	1. if ip_fd->if_flags is not set to IFF_READ_IP, just copy data to ip_fd->if_rdbuf_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	udp_put_data (fd, offset, data, for_ioctl)	Since udp_port->up_flags has been set UPF_READ_IP, go to line 412 call udp_ip_arrived().

Udp.c 855-1112	udp_ip_arrived(port, pack, pack_size)	1. delete ip heard; 2. verify udp header; 3. add udp io header, which contains some information about ip layer, such as source ip, port and des ip, port; 4. call udp_rd_enqueue() to copy udp packets to udp_fd->uf_rdbuf_head 5. since listener is still suspending on read() system call, the udp_fd->uf_flags is still UFF_READ_IP, then call udp_packet2user(share_fd) at line 1083;
Udp.c 795-853	udp_packet2user (udp_fd)	Call (*udp_fd->uf_put_userdata)() at line 848 to put data to user layer, actually call sr_put_userdata () at line 548 in sr.c.
Sr.c 548-606	sr_put_userdata (fd, offset, data, for_ioctl)	Return cp_b2u (data, (*head_ptr)->mq_mess.PROC_NR, dst) to file system, so listener get data from read system call.
Listener.c 103-104	udp_io_hdr = (udp_io_hdr_t *)buf; s = s - sizeof(udp_io_hdr_t);	Remove udp_io_hdr to get the real data
Listener.c 112	printf("listener: from %s, %u \n", inet_ntoa(udp_io_hdr->uih_src_addr), ntohs(udp_io_hdr->uih_src_port));	Show src address and port in udp_io_hdr. All set.

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 pcocedures under ip layer are almost same to tcp and udp, I only explain those working on ip layer.

Launch server as: /usr/messager/server		
Server.c 46	if ((sockfd = socket(AF_INET, 0, 0)) == -1)	Create a socket file descriptor, actually call mnx_socket(int proto) in socket.c
Socket.c 48	if ((fd = open(device, O_RDWR)) < 0)	Return a tcp device file descriptor
Server.c 56	bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)	Bind sockfd to server's address, my_addr
Socket.c 246-292	int mnx_bind(int fd, struct sockaddr *addr)	1. construct a tcpconf and initialize it; 2. call ioctl(fd, NWIOSTCPCONF, &tcpconf) to set operation flag, which actually call tcp_ioctl (fd, req), line 651-660;
Tcp.c 751-938	tcp_setconf(tcp_fd)	Do verification of tcpconf, and saving the configuration in tcp_fd->tf_tcpconf.
Server.c 61	if (listen(sockfd, 0) == -1)	Listening on a port number configured
Socket.c 214	mnx_listen(int fd)	Add a new flag in by tcpopt.nwto_flags |= NWTO_DEL_RST. The **NWTO_DEL_RST** option delays a failure response on a connect to the same port as the current open connection. Without this option a connect would fail if a server is not yet listening. With this option a connect will linger on until the server starts listening.
Server.c 70	new_fd = accept(sockfd, (struct sockaddr	Actually call mnx_accept() in socket.c

	*)&their_addr, &sin_size)	
Socket.c 136-209	mnx_accept(int fd, struct sockaddr *addr)	1. create a new socket chan, mnx_socket(IPPROTO_TCP) 2. get operation flag of present socket, ioctl(fd, NWIOGTCPCONF, &tcpconf); 3. assign operation flags to new socket chan, ioctl(chan, NWIOSTCPCONF, &tcpconf); 4. make new socket listen, ioctl(chan, NWIOTCPLISTEN, &tcplistenopt); 5. get connection information, ioctl(chan, NWIOGTCPCONF, &tcpconf2).
Server.c 81	write(new_fd, buf, strlen(buf)	Write packets to remote peer
Tcp.c 1508-1559	tcp_write(fd, count)	1. do flag verification 2. call tcp_fd_write(tcp_conn) to get data from user layer and copy them to tcp_conn->tc_send_data; 3. call tcp_conn_write (tcp_conn, enq) to send data out by established connection.
Tcp_send.c 25-63	tcp_conn_write (tcp_conn, enq)	Send packet out by a tcp port
Tcp_send.c 77-146	tcp_port_write(tcp_port)	Call ip_write (fd, count) to send packets to ip layer
Following refer to udp about how ip packets are sent out and how they get to the destination.		

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

add_route/del_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.c 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:

add_route.c 54-55	if (strcmp(prog_name, "add_route") == 0) action= ADD;	According to program's name, decide it should add route in routing table

add_route.c 74-153	while ((c= getopt(argc, argv, "iovDg:d:m:n:I:?")) != -1)……………
	gateway_str= g_arg;
	destination_str= d_arg;
		Analyze the parameters set in command line. Since we set –d 192.168.2.3, string "192.168.2.3" is assigned to destination_str, string "192.168.163.2" is assigned to gateway_str.

	metric_str= m_arg;
	netmask_str= n_arg;
	ip_device= I_arg;	

add_route.c 155	if (!name_to_ip(gateway_str, &gateway))	Convert gateway_str to a valid ip address, saved in gateway
add_route.c 166-207	if (destination_str) ……..	Analyze destination address, convert host name to be a valid ip address and calculate default netmask
add_route.c 209-236	if (netmask_str)....	If other parameters are not set, assign default value to them
add_route.c 238	ip_fd= open(ip_device, O_RDWR);	Open IP device, get a file descriptor of it
add_route.c 259-266	route.nwr_ent_no= 0; …..	Construct route, put set value and default value in it
add_route.c 269	r= ioctl(ip_fd, itab ? NWIOSIPIROUTE : NWIOSIPOROUTE, &route);	Call ioctl system call, putting data saved in route into kernel
Ip_ioctl.c 26		Begin ioctl system call
ip_ioctl.c 333	case NWIOSIPOROUTE:	From the flag set in ioctl, jump here to do functionality of setting ip out routing table
Ip_ioctl.c 344	result= ipr_add_oroute()	Add an route entry into out routing table
ipr.c 209	PUBLIC int ipr_add_oroute()	Ipr_add_oroute() definition
ipr.c 209-402		Add a new entry into routing array
The del_route.c is the same file as add_route.c, which only has a different name with the different parameters in configuration.		
The pr_routes.c does the similar things, get a ip device file descriptor and call system call ioctl.		

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.