Operating Systems

Network Stack

Network Communication

- Key functionality in an OS
  - Drove the development of BSD and Xinu
  - Everything interesting communicates in some way
- The Internet Protocol suite enables the internetworking of different network types and technologies
  - Even though Ethernet is now ubiquitous, the addressing mechanism does not allow location-based routing
- Layers of network functionality and protocols give rise to the network stack

Internet Protocols

- IP – Internet Protocol, Layer 3
  - Operates on packets, or datagrams, generally embedded in one or more Data Link (Ethernet) frames
- UDP – User Datagram protocol
  - Minimal Transport (Layer 4) protocol which includes port numbers to demultiplex inside a host
- ARP – Address Resolution Protocol
  - Binds global Layer 3 addresses to local Layer 2 addresses

Internet Protocols

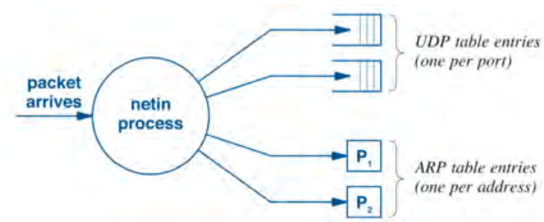
- DHCP – Dynamic Host Configuration Protocol
  - Allows a server to provide an IP address, default gateway, nameserver, etc.
- ICMP – Internet Control Message Protocol
  - Error and informational messages
  - Useful for debugging
  - ping uses Echo Request and Echo Response ICMP messages

Xinu Network Processes

- Single network input process called *netin*
- IP output process called *ipout*
- The software uses the timed message receipt call recvtime
- When a process sends a network message, the netin process sends an internal message (with send) when the response is received
- If the timer expires first, recvtime returns the message TIMEOUT

Xinu Network Processes

- Coordination between the netin process and other waiting processes occurs via UDP-specific queues or ARP table entries



Kernel Threads

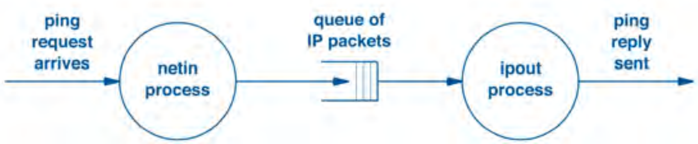
- The Xinu network stack is realized in terms of processes, which we have considered to be application programs
- Unix/Linux does essentially the same thing with threads running in kernel space – kernel threads
  - Show up in ps output surrounded by [brackets]
- Background system-level tasks like protocol processing, buffer management are implemented this way
  - A pure microkernel approach would put them in userspace, interacting with messages

Kernel Threads

- One of the Linux kernel threads is called ksoftirqd
- Soft IRQs are raised by interrupt handlers to request finishing bottom half processing tasks that can be deferred
  - The goal is to minimize processing time while interrupts are deferred
- Another is events/0, which provides a work queue for other kernel tasks
- Either could be used to restructure netin and ipout in Xinu

Xinu Network Processes

- netin is the only process that reads network packets – it cannot block waiting for a packet
- Incoming IP packets may require the generation of a response (e.g. ping)
- Transmission of outgoing IP packets may require an ARP exchange
- The processes are decoupled and utilize a queue



arp.h

```c
/* Items related to ARP - definition of cache and the packet format */

#define ARP_HALEN 6   /* Size of Ethernet MAC address */
#define ARP_PALEN 4   /* Size of IP address     */

#define ARP_HTYPE 1   /* Ethernet hardware type */
#define ARP_PTYPE 0x0800    /* IP protocol type   */

#define ARP_OP_REQ  1   /* Request op code     */
#define ARP_OP_RPLY 2   /* Reply op code     */

#define ARP_SIZ    16   /* Number of entries in a cache */

#define ARP_RETRY 3   /* Num. retries for ARP request */

#define ARP_TIMEOUT 300   /* Retry timer in milliseconds   */

/* State of an ARP cache entry */

#define AR_FREE    0   /* Slot is unused    */
#define AR_PENDING  1   /* Resolution in progress */
#define AR_RESOLVED 2   /* Entry is valid    */
```

arp.h

```c
#pragma pack(2)
struct arppacket {      /* ARP packet for IP & Ethernet */
  byte  arp_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC addr */
  byte  arp_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
  uint16  arp_ethtype;    /* Ethernet type field     */
  uint16  arp_htype;     /* ARP hardware type     */
  uint16  arp_ptype;     /* ARP protocol type     */
  byte  arp_hlen;    /* ARP hardware address length  */
  byte  arp_plen;    /* ARP protocol address length  */
  uint16  arp_op;      /* ARP operation     */
  byte  arp_sndha[ARP_HALEN]; /* ARP sender's Ethernet addr   */
  uint32  arp_sndpa;     /* ARP sender's IP address  */
  byte  arp_tarha[ARP_HALEN]; /* ARP target's Ethernet addr */
  uint32  arp_tarpa;     /* ARP target's IP address  */
};
#pragma pack()

struct arpentry {      /* Entry in the ARP cache */
  int32 arstate;    /* State of the entry    */
  uint32  arpaddr;     /* IP address of the entry   */
  pid32 arpid;      /* Waiting process or -1  */
  byte  arhaddr[ARP_HALEN]; /* Ethernet address of the entry*/
};
```

arp.c

```c
struct  arpentry  arpcache[ARP_SIZ];  /* ARP cache      */

/*------------------------------------------------------------------------
 * arp_init  -  Initialize ARP cache for an Ethernet interface
 *------------------------------------------------------------------------
 */
void  arp_init(void)
{
  int32 i;      /* ARP cache index    */

  for (i=1; i<ARP_SIZ; i++) { /* Initialize cache to empty  */
    arpcache[i].arstate = AR_FREE;
  }
}
```

arp.c

```
/*------------------------------------------------------------------------
 * arp_resolve  -  Use ARP to resolve an IP address to an Ethernet address
 *------------------------------------------------------------------------
 */
status  arp_resolve (
   uint32 nxthop,     /* Next-hop address to resolve  */
   byte mac[ETH_ADDR_LEN] /* Array into which Ethernet  */
  )       /*   address should be placed */
{
  intmask mask;      /* Saved interrupt mask   */
  struct  arppacket apkt;   /* Local packet buffer    */
  int32 i;       /* Index into arpcache     */
  int32 slot;     /* ARP table slot to use  */
  struct  arpentry  *arptr; /* Ptr to ARP cache entry */
  int32 msg;      /* Message returned by recvtime */

  /* Use MAC broadcast address for IP limited broadcast */

  if (nxthop == IP_BCAST) {
    memcpy(mac, NetData.ethbcast, ETH_ADDR_LEN);
    return OK;
  }
```

arp.c

```
/* Use MAC broadcast address for IP network broadcast */

if (nxthop == NetData.ipbcast) {
  memcpy(mac, NetData.ethbcast, ETH_ADDR_LEN);
  return OK;
}

/* Ensure only one process uses ARP at a time */

mask = disable();

/* See if next hop address is already present in ARP cache */

for (i=0; i<ARP_SIZ; i++) {
  arptr = &arpcache[i];
  if (arptr->arstate == AR_FREE) {
    continue;
  }
  if (arptr->arpaddr == nxthop) { /* Address is in cache */
    break;
  }
}
```

arp.c

```
if (i < ARP_SIZ) {  /* Entry was found */

  /* If entry is resolved - handle and return */

  if (arptr->arstate == AR_RESOLVED) {
    memcpy(mac, arptr->arhaddr, ARP_HALEN);
    restore(mask);
    return OK;
  }

  /* Entry is already pending -  return error because */
  /*  only one process can be waiting at a time */

  if (arptr->arstate == AR_PENDING) {
    restore(mask);
    return SYSERR;
  }
}

/* IP address not in cache -  allocate a new cache entry and  */
/*  send an ARP request to obtain the answer    */

slot = arp_alloc();
if (slot == SYSERR) {
  restore(mask);
  return SYSERR;
}
```

arp.c

```
arptr = &arpcache[slot];
arptr->arstate = AR_PENDING;
arptr->arpaddr = nxthop;
arptr->arpid = currpid;

/* Hand-craft an ARP Request packet */

memcpy(apkt.arp_ethdst, NetData.ethbcast, ETH_ADDR_LEN);
memcpy(apkt.arp_ethsrc, NetData.ethucast, ETH_ADDR_LEN);
apkt.arp_ethtype = ETH_ARP;   /* Packet type is ARP   */
apkt.arp_htype = ARP_HTYPE;   /* Hardware type is Ethernet  */
apkt.arp_ptype = ARP_PTYPE;   /* Protocol type is IP  */
apkt.arp_hlen = 0xff & ARP_HALEN; /* Ethernet MAC size in bytes */
apkt.arp_plen = 0xff & ARP_PALEN; /* IP address size in bytes */
apkt.arp_op = 0xffff & ARP_OP_REQ;/* ARP type is Request  */
memcpy(apkt.arp_sndha, NetData.ethucast, ARP_HALEN);
apkt.arp_sndpa = NetData.ipucast; /* IP address of interface  */
memset(apkt.arp_tarha, '\0', ARP_HALEN); /* Target HA is unknown*/
apkt.arp_tarpa = nxthop;    /* Target protocol address  */
```

arp.c

```c
/* Convert ARP packet from host to net byte order */

arp_hton(&apkt);

/* Convert Ethernet header from host to net byte order */

eth_hton((struct netpacket *)&apkt);

/* Send the packet ARP_RETRY times and await response */

msg = recvclr();
for (i=0; i<ARP_RETRY; i++) {
  write(ETHER0, (char *)&apkt, sizeof(struct arppacket));
  msg = recvtime(ARP_TIMEOUT);
  if (msg == TIMEOUT) {
    continue;
  } else if (msg == SYSERR) {
    restore(mask);
    return SYSERR;
  } else {  /* entry is resolved */
    break;
  }
}
```

arp.c

```c
  /* If no response, return TIMEOUT */

  if (msg == TIMEOUT) {
    arptr->arstate = AR_FREE;   /* Invalidate cache entry */
    restore(mask);
    return TIMEOUT;
  }

  /* Return hardware address */

  memcpy(mac, arptr->arhaddr, ARP_HALEN);
  restore(mask);
  return OK;
}
```

arp.c

```c
/*------------------------------------------------------------------------
 * arp_in  -  Handle an incoming ARP packet
 *------------------------------------------------------------------------
 */
void arp_in (
    struct arppacket *pktptr  /* Ptr to incoming packet */
  )
{
  intmask mask;     /* Saved interrupt mask   */
  struct  arppacket apkt;  /* Local packet buffer    */
  int32 slot;     /* Slot in cache    */
  struct  arpentry  *arptr; /* Ptr to ARP cache entry */
  bool8 found;      /* Is the sender's address in */
          /*   the cache?     */

  /* Convert packet from network order to host order */

  arp_ntoh(pktptr);

  /* Verify ARP is for IPv4 and Ethernet */

  if ( (pktptr->arp_htype != ARP_HTYPE) ||
       (pktptr->arp_ptype != ARP_PTYPE) ) {
    freebuf((char *)pktptr);
    return;
  }
```

arp.c

```c
  /* Ensure only one process uses ARP at a time */

  mask = disable();

  /* Search cache for sender's IP address */

  found = FALSE;

  for (slot=0; slot < ARP_SIZ; slot++) {
    arptr = &arpcache[slot];

    /* Skip table entries that are unused */

    if (arptr->arstate == AR_FREE) {
      continue;
    }

    /* If sender's address matches, we've found it */

    if (arptr->arpaddr == pktptr->arp_sndpa) {
      found = TRUE;
      break;
    }
  }
```

arp.c

```
if (found) {

  /* Update sender's hardware address */

  memcpy(arptr->arhaddr, pktptr->arp_sndha, ARP_HALEN);

  /* If a process was waiting, inform the process */

  if (arptr->arstate == AR_PENDING) {
    /* Mark resolved and notify waiting process */
    arptr->arstate = AR_RESOLVED;
    send(arptr->arpid, OK);
  }
}

/* For an ARP reply, processing is complete */

if (pktptr->arp_op == ARP_OP_RPLY) {
  freebuf((char *)pktptr);
  restore(mask);
  return;
}
```

arp.c

```
/* The following is for an ARP request packet: if the local */
/*  machine is not the target or the local IP address is not  */
/*  yet known, ignore the request (i.e., processing is complete)*/

if ((!NetData.ipvalid) ||
    (pktptr->arp_tarpa != NetData.ipucast)) {
  freebuf((char *)pktptr);
  restore(mask);
  return;
}

/* Request has been sent to the local machine's address.  So, */
/*   add sender's info to cache, if not already present   */

if (!found) {
  slot = arp_alloc();
  if (slot == SYSERR) { /* Cache is full */
    kprintf("ARP cache overflow on interface\n");
    freebuf((char *)pktptr);
    restore(mask);
    return;
  }
  arptr = &arpcache[slot];
  arptr->arpaddr = pktptr->arp_sndpa;
  memcpy(arptr->arhaddr, pktptr->arp_sndha, ARP_HALEN);
  arptr->arstate = AR_RESOLVED;
}
```

arp.c

```
/* Hand-craft an ARP reply packet and send back to requester  */

memcpy(apkt.arp_ethdst, pktptr->arp_sndha, ARP_HALEN);
memcpy(apkt.arp_ethsrc, NetData.ethucast, ARP_HALEN);
apkt.arp_ethtype= ETH_ARP;    /* Frame carries ARP  */
apkt.arp_htype  = ARP_HTYPE;    /* Hardware is Ethernet */
apkt.arp_ptype  = ARP_PTYPE;    /* Protocol is IP */
apkt.arp_hlen = ARP_HALEN;    /* Ethernet address size*/
apkt.arp_plen = ARP_PALEN;    /* IP address size  */
apkt.arp_op = ARP_OP_RPLY;    /* Type is Reply  */

/* Insert local Ethernet and IP address in sender fields  */

memcpy(apkt.arp_sndha, NetData.ethucast, ARP_HALEN);
apkt.arp_sndpa = NetData.ipucast;

/* Copy target Ethernet and IP addresses from request packet */

memcpy(apkt.arp_tarha, pktptr->arp_sndha, ARP_HALEN);
apkt.arp_tarpa = pktptr->arp_sndpa;
```

arp.c

```
  /* Convert ARP packet from host to network byte order */

  arp_hton(&apkt);

  /* Convert the Ethernet header to network byte order */

  eth_hton((struct netpacket *)&apkt);

  /* Send the reply */

  write(ETHER0, (char *)&apkt, sizeof(struct arppacket));
  freebuf((char *)pktptr);
  restore(mask);
  return;
}
```

arp.c

```
/*------------------------------------------------------------------------
 * arp_alloc  -  Find a free slot or kick out an entry to create one
 *------------------------------------------------------------------------
 */
int32 arp_alloc ()
{
  int32 slot;      /* Slot in ARP cache    */

  /* Search for a free slot */

  for (slot=0; slot < ARP_SIZ; slot++) {
    if (arpcache[slot].arstate == AR_FREE) {
      memset((char *)&arpcache[slot],
          NULLCH, sizeof(struct arpentry));
      return slot;
    }
  }
```

arp.c

```
  /* Search for a resolved entry */

  for (slot=0; slot < ARP_SIZ; slot++) {
    if (arpcache[slot].arstate == AR_RESOLVED) {
      memset((char *)&arpcache[slot],
          NULLCH, sizeof(struct arpentry));
      return slot;
    }
  }

  /* At this point, all slots are pending (should not happen) */

  kprintf("ARP cache size exceeded\n");

  return SYSERR;
}
```

arp.c

```
/*------------------------------------------------------------------------
 * arp_ntoh  -  Convert ARP packet fields from net to host byte order
 *------------------------------------------------------------------------
 */
void  arp_ntoh(
  struct arppacket *pktptr
  )
{
  pktptr->arp_htype = ntohs(pktptr->arp_htype);
  pktptr->arp_ptype = ntohs(pktptr->arp_ptype);
  pktptr->arp_op    = ntohs(pktptr->arp_op);
  pktptr->arp_sndpa = ntohl(pktptr->arp_sndpa);
  pktptr->arp_tarpa = ntohl(pktptr->arp_tarpa);
}

/*------------------------------------------------------------------------
 * arp_hton  -  Convert ARP packet fields from net to host byte order
 *------------------------------------------------------------------------
 */
void  arp_hton(
  struct arppacket *pktptr
  )
{
  pktptr->arp_htype = htons(pktptr->arp_htype);
  pktptr->arp_ptype = htons(pktptr->arp_ptype);
  pktptr->arp_op    = htons(pktptr->arp_op);
  pktptr->arp_sndpa = htonl(pktptr->arp_sndpa);
  pktptr->arp_tarpa = htonl(pktptr->arp_tarpa);
}
```

net.h

```
#define NETSTK      8192   /* Stack size for network setup */
#define NETPRIO     500     /* Network startup priority    */
#define NETBOOTFILE 128    /* Size of the netboot filename */

/* Constants used in the networking code */

#define ETH_ARP     0x0806    /* Ethernet type for ARP  */
#define ETH_IP      0x0800    /* Ethernet type for IP   */
#define ETH_IPv6    0x86DD    /* Ethernet type for IPv6 */

/* Format of an Ethernet packet carrying IPv4 and UDP */

#pragma pack(2)
struct  netpacket {
  byte  net_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC address  */
  byte  net_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
  uint16  net_ethtype;    /* Ethernet type field     */
  byte  net_ipvh;     /* IP version and hdr length  */
  byte  net_iptos;     /* IP type of service    */
  uint16  net_iplen;     /* IP total packet length */
  uint16  net_ipid;     /* IP datagram ID    */
  uint16  net_ipfrag;     /* IP flags & fragment offset */
  byte  net_ipttl;     /* IP time-to-live     */
```

net.h

```
  byte   net_ipproto;   /* IP protocol (actually type)  */
  uint16  net_ipcksum;    /* IP checksum      */
  uint32  net_ipsrc;    /* IP source address    */
  uint32  net_ipdst;    /* IP destination address */
  union {
   struct {
    uint16  net_udpsport; /* UDP source protocol port */
    uint16  net_udpdport; /* UDP destination protocol port*/
    uint16  net_udplen; /* UDP total length   */
    uint16  net_udpcksum; /* UDP checksum      */
    byte    net_udpdata[1500-28];/* UDP payload (1500-above)*/
   };
   struct {
    byte    net_ictype; /* ICMP message type    */
    byte    net_iccode; /* ICMP code field (0 for ping) */
    uint16  net_icccksum;  /* ICMP message checksum  */
    uint16  net_icident;  /* ICMP identifier    */
    uint16  net_icseq;  /* ICMP sequence number   */
    byte    net_icdata[1500-28];/* ICMP payload (1500-above)*/
   };
  };
};
```

net.h

```
#define PACKLEN sizeof(struct netpacket)

extern  bpid32  netbufpool;   /* ID of net packet buffer pool */

struct  network {
  uint32  ipucast;
  uint32  ipbcast;
  uint32  ipmask;
  uint32  ipprefix;
  uint32  iprouter;
  uint32  bootserver;
  bool8 ipvalid;
  byte  ethucast[ETH_ADDR_LEN];
  byte  ethbcast[ETH_ADDR_LEN];
  char  bootfile[NETBOOTFILE];
};

extern  struct  network NetData;  /* Local Network Interface  */
```

net.c

```
#include <xinu.h>
#include <stdio.h>

struct  network NetData;
bpid32  netbufpool;

/*------------------------------------------------------------------------
 * net_init  -  Initialize network data structures and processes
 *------------------------------------------------------------------------
 */

void  net_init (void)
{
  int32 nbufs;      /* Total no of buffers    */

  /* Initialize the network data structure */

  memset((char *)&NetData, NULLCH, sizeof(struct network));

  /* Obtain the Ethernet MAC address */

  control(ETHER0, ETH_CTRL_GET_MAC, (int32)NetData.ethucast, 0);

  memset((char *)NetData.ethbcast, 0xFF, ETH_ADDR_LEN);
```

net.c

```
  /* Create the network buffer pool */

  nbufs = UDP_SLOTS * UDP_QSIZ + ICMP_SLOTS * ICMP_QSIZ + 1;

  netbufpool = mkbufpool(PACKLEN, nbufs);

  /* Initialize the ARP cache */

  arp_init();

  /* Initialize UDP */

  udp_init();

  /* Initialize ICMP */

  icmp_init();

  /* Initialize the IP output queue */

  ipoqueue.iqhead = 0;
  ipoqueue.iqtail = 0;
  ipoqueue.iqsem = semcreate(0);
```

net.c

```
  if((int32)ipoqueue.iqsem == SYSERR) {
    panic("Cannot create ip output queue semaphore");
    return;
  }

  /* Create the IP output process */

  resume(create(ipout, NETSTK, NETPRIO, "ipout", 0, NULL));

  /* Create a network input process */

  resume(create(netin, NETSTK, NETPRIO, "netin", 0, NULL));
}
```

net.c

```
/*------------------------------------------------------------------------
 * netin  -  Repeatedly read and process the next incoming packet
 *------------------------------------------------------------------------
 */

process netin ()
{
  struct  netpacket *pkt;   /* Ptr to current packet  */
  int32 retval;      /* Return value from read */

  /* Do forever: read a packet from the network and process */

  while(1) {

    /* Allocate a buffer */

    pkt = (struct netpacket *)getbuf(netbufpool);

    /* Obtain next packet that arrives */

    retval = read(ETHER0, (char *)pkt, PACKLEN);
    if(retval == SYSERR) {
      panic("Cannot read from Ethernet\n");
    }
```

net.c

```
    /* Convert Ethernet Type to host order */

    eth_ntoh(pkt);

    /* Demultiplex on Ethernet type */

    switch (pkt->net_ethtype) {

        case ETH_ARP:      /* Handle ARP */
      arp_in((struct arppacket *)pkt);
      continue;

        case ETH_IP:       /* Handle IP  */
      ip_in(pkt);
      continue;

        case ETH_IPv6:       /* Handle IPv6  */
      freebuf((char *)pkt);
      continue;

        default:  /* Ignore all other incoming packets  */
      freebuf((char *)pkt);
      continue;
    }
  }
}
```

net.c

```
/*------------------------------------------------------------------------
 * eth_hton  -  Convert Ethernet type field to network byte order
 *------------------------------------------------------------------------
 */
void  eth_hton(
    struct netpacket *pktptr
  )
{
  pktptr->net_ethtype = htons(pktptr->net_ethtype);
}


/*------------------------------------------------------------------------
 * eth_ntoh  -  Convert Ethernet type field to host byte order
 *------------------------------------------------------------------------
 */
void  eth_ntoh(
    struct netpacket *pktptr
  )
{
  pktptr->net_ethtype = ntohs(pktptr->net_ethtype);
}
```

ip.h

```c
/* ip.h  -  Constants related to Internet Protocol version 4 (IPv4) */

#define IP_BCAST   0xffffffff  /* IP local broadcast address */
#define IP_THIS    0xffffffff  /* "this host" src IP address */
#define IP_ALLZEROS 0x00000000  /* The all-zeros IP address     */

#define IP_ICMP   1   /* ICMP protocol type for IP  */
#define IP_UDP    17    /* UDP protocol type for IP   */

#define IP_ASIZE  4   /* Bytes in an IP address */
#define IP_HDR_LEN  20    /* Bytes in an IP header  */
#define IP_VH    0x45    /* IP version and hdr length  */

#define IP_OQSIZ  8   /* Size of IP output queue  */

/* Queue of outgoing IP packets waiting for ipout process */

struct  iqentry {
  int32 iqhead;      /* Index of next packet to send */
  int32 iqtail;      /* Index of next free slot  */
  sid32 iqsem;       /* Semaphore that counts pkts */
  struct  netpacket *iqbuf[IP_OQSIZ];/* Circular packet queue */
};

extern  struct  iqentry ipoqueue; /* Network output queue   */
```

ip.c

```c
#include <xinu.h>

struct  iqentry ipoqueue;   /* Queue of outgoing packets  */

/*------------------------------------------------------------------------
 * ip_in  -  Handle an IP packet that has arrived over a network
 *------------------------------------------------------------------------
 */

void  ip_in(
    struct netpacket *pktptr  /* Pointer to the packet  */
  )
{
  int32 icmplen;    /* Length of ICMP message */

  /* Verify checksum */

  if (ipcksum(pktptr) != 0) {
    kprintf("IP header checksum failed\n\r");
    freebuf((char *)pktptr);
    return;
  }

  /* Convert IP header fields to host order */

  ip_ntoh(pktptr);
```

ip.c

```c
  /* Verify encapsulated prototcol checksums and then convert */
  /*  the encapsulated headers to host byte order    */

  switch (pktptr->net_ipproto) {

      case IP_UDP:
        /* Skipping UDP checksum for now */
        udp_ntoh(pktptr);
        break;

      case IP_ICMP:
        icmplen = pktptr->net_iplen - IP_HDR_LEN;
        if (icmp_cksum((char *)&pktptr->net_ictype,icmplen) != 0){
          freebuf((char *)pktptr);
          return;
        }
        icmp_ntoh(pktptr);
        break;

      default:
        break;
  }
```

ip.c

```c
  /* Deliver 255.255.255.255 to local stack */

  if (pktptr->net_ipdst == IP_BCAST) {
    ip_local(pktptr);
    return;
  }

  /* If we do not yet have a valid address, accept UDP packets  */
  /*  (to get DHCP replies) and drop others      */

  if (!NetData.ipvalid) {
    if (pktptr->net_ipproto == IP_UDP) {
      ip_local(pktptr);
      return;
    } else {
      freebuf((char *)pktptr);
      return;
    }
  }
```

ip.c

```
/* If packet is destined for us, accept it; otherwise, drop it  */

if ( (pktptr->net_ipdst == NetData.ipucast) ||
     (pktptr->net_ipdst == NetData.ipbcast) ||
     (pktptr->net_ipdst == IP_BCAST)  ) {
  ip_local(pktptr);
  return;
} else {

  /* Drop the packet */
  freebuf((char *)pktptr);
  return;
}
}
```

ip.c

```
/*------------------------------------------------------------------------
 * ip_send  -  Send an outgoing IP datagram from the local stack
 *------------------------------------------------------------------------
 */

status  ip_send(
    struct netpacket *pktptr  /* Pointer to the packet  */
  )
{
  intmask mask;     /* Saved interrupt mask   */
  uint32  dest;     /* Destination of the datagram  */
  int32 retval;     /* Return value from functions  */
  uint32  nxthop;     /* Next-hop address   */

  mask = disable();

  /* Pick up the IP destination address from the packet */

  dest = pktptr->net_ipdst;
```

ip.c

```
/* Loop back to local stack if destination 127.0.0.0/8 */

if ((dest&0xff000000) == 0x7f000000) {
  ip_local(pktptr);
  restore(mask);
  return OK;
}

/* Loop back if the destination matches our IP unicast address  */

if (dest == NetData.ipucast) {
  ip_local(pktptr);
  restore(mask);
  return OK;
}

/* Broadcast if destination is 255.255.255.255 */

if ( (dest == IP_BCAST) ||
     (dest == NetData.ipbcast) ) {
  memcpy(pktptr->net_ethdst, NetData.ethbcast,
          ETH_ADDR_LEN);
  retval = ip_out(pktptr);
  restore(mask);
  return retval;
}
```

ip.c

```
/* If destination is on the local network, next hop is the  */
/*  destination; otherwise, next hop is default router  */


if ( (dest & NetData.ipmask) == NetData.ipprefix) {

  /* Next hop is the destination itself */
  nxthop = dest;

} else {

  /* Next hop is default router on the network */
  nxthop = NetData.iprouter;

}

if (nxthop == 0) {  /* Dest. invalid or no default route  */
  freebuf((char *)pktptr);
  return SYSERR;
}
```

ip.c

```
  /* Resolve the next-hop address to get a MAC address */

  retval = arp_resolve(nxthop, pktptr->net_ethdst);
  if (retval != OK) {
    freebuf((char *)pktptr);
    return SYSERR;
  }

  /* Send the packet */

  retval = ip_out(pktptr);
  restore(mask);
  return retval;
}
```

ip.c

```
/*------------------------------------------------------------------------
 * ip_local  -  Deliver an IP datagram to the local stack
 *------------------------------------------------------------------------
 */
void  ip_local(
    struct netpacket *pktptr  /* Pointer to the packet  */
  )
{
  /* Use datagram contents to determine how to process */

  switch (pktptr->net_ipproto) {

      case IP_UDP:
    udp_in(pktptr);
    return;

      case IP_ICMP:
    icmp_in(pktptr);
    return;

      default:
    freebuf((char *)pktptr);
    return;
  }
}
```

ip.c

```
/*------------------------------------------------------------------------
 *  ip_out  -  Transmit an outgoing IP datagram
 *------------------------------------------------------------------------
 */
status  ip_out(
    struct netpacket *pktptr  /* Pointer to the packet  */
  )
{
  uint16  cksum;      /* Checksum in host byte order  */
  int32 len;      /* Length of ICMP message */
  int32 pktlen;     /* Length of entire packet  */
  int32 retval;     /* Value returned by write  */

  /* Compute total packet length */

  pktlen = pktptr->net_iplen + ETH_HDR_LEN;

  /* Convert encapsulated protocol to network byte order */

  switch (pktptr->net_ipproto) {

      case IP_UDP:

      pktptr->net_udpcksum = 0;
      udp_hton(pktptr);

      /* ...skipping UDP checksum computation */

      break;
```

ip.c

```
      case IP_ICMP:
      icmp_hton(pktptr);

      /* Compute ICMP checksum */

      pktptr->net_iccksum = 0;
      len = pktptr->net_iplen-IP_HDR_LEN;
      cksum = icmp_cksum((char *)&pktptr->net_ictype,
            len);
      pktptr->net_iccksum = 0xffff & htons(cksum);
      break;

      default:
      break;
  }

  /* Convert IP fields to network byte order */

  ip_hton(pktptr);
```

ip.c

```
/* Compute IP header checksum */

pktptr->net_ipcksum = 0;
cksum = ipcksum(pktptr);
pktptr->net_ipcksum = 0xffff & htons(cksum);

/* Convert Ethernet fields to network byte order */

eth_hton(pktptr);

/* Send packet over the Ethernet */

retval = write(ETHER0, (char*)pktptr, pktlen);
freebuf((char *)pktptr);

if (retval == SYSERR) {
  return SYSERR;
} else {
  return OK;
}
}
```

ip.c

```
/*------------------------------------------------------------------------
 * ipcksum  -  Compute the IP header checksum for a datagram
 *------------------------------------------------------------------------
 */

uint16  ipcksum(
  struct  netpacket *pkt   /* Pointer to the packet  */
 )
{
 uint16  *hptr;      /* Ptr to 16-bit header values  */
 int32 i;      /* Counts 16-bit values in hdr  */
 uint16  word;      /* One 16-bit word    */
 uint32  cksum;      /* Computed value of checksum */

 hptr= (uint16 *) &pkt->net_ipvh;

 /* Sum 16-bit words in the packet */

 cksum = 0;
 for (i=0; i<10; i++) {
   word = *hptr++;
   cksum += (uint32) htons(word);
 }
```

ip.c

```
/* Add in carry, and take the ones-complement */

cksum += (cksum >> 16);
cksum = 0xffff & ~cksum;

/* Use all-1s for zero */

if (cksum == 0xffff) {
  cksum = 0;
}
return (uint16) (0xffff & cksum);
}
```

ip.c

```
/*------------------------------------------------------------------------
 * ip_ntoh  -  Convert IP header fields to host byte order
 *------------------------------------------------------------------------
 */
void  ip_ntoh(
    struct netpacket *pktptr
  )
{
  pktptr->net_iplen = ntohs(pktptr->net_iplen);
  pktptr->net_ipid = ntohs(pktptr->net_ipid);
  pktptr->net_ipfrag = ntohs(pktptr->net_ipfrag);
  pktptr->net_ipsrc = ntohl(pktptr->net_ipsrc);
  pktptr->net_ipdst = ntohl(pktptr->net_ipdst);
}

/*------------------------------------------------------------------------
 * ip_hton  -  Convert IP header fields to network byte order
 *------------------------------------------------------------------------
 */
void  ip_hton(
    struct netpacket *pktptr

  )
{
  pktptr->net_iplen = htons(pktptr->net_iplen);
  pktptr->net_ipid = htons(pktptr->net_ipid);
  pktptr->net_ipfrag = htons(pktptr->net_ipfrag);
  pktptr->net_ipsrc = htonl(pktptr->net_ipsrc);
  pktptr->net_ipdst = htonl(pktptr->net_ipdst);
}
```

ip.c

```
/*------------------------------------------------------------------------
 *  ipout  -  Process that transmits IP packets from the IP output queue
 *------------------------------------------------------------------------
 */

process ipout(void)
{
  struct  netpacket *pktptr;  /* Pointer to next the packet */
  struct  iqentry   *ipqptr;  /* Pointer to IP output queue */
  uint32  destip;      /* Destination IP address */
  uint32  nxthop;      /* Next hop IP address    */
  int32 retval;      /* Value returned by functions  */

  ipqptr = &ipoqueue;

  while(1) {

    /* Obtain next packet from the IP output queue */

    wait(ipqptr->iqsem);
    pktptr = ipqptr->iqbuf[ipqptr->iqhead++];
    if (ipqptr->iqhead >= IP_OQSIZ) {
      ipqptr->iqhead= 0;
    }
```

ip.c

```
    /* Fill in the MAC source address */

    memcpy(pktptr->net_ethsrc, NetData.ethucast, ETH_ADDR_LEN);

    /* Extract destination address from packet */

    destip = pktptr->net_ipdst;


    /* Sanity check: packets sent to ioout should *not* */
    /*  contain a broadcast address.     */

    if ((destip == IP_BCAST)||(destip == NetData.ipbcast)) {
      kprintf("ipout: encountered a broadcast\n");
      freebuf((char *)pktptr);
      continue;
    }

    /* Check whether destination is the local computer */

    if (destip == NetData.ipucast) {
      ip_local(pktptr);
      continue;
    }
```

ip.c

```
    /* Check whether destination is on the local net */

    if ( (destip & NetData.ipmask) == NetData.ipprefix) {

      /* Next hop is the destination itself */

      nxthop = destip;
    } else {

      /* Next hop is default router on the network */

      nxthop = NetData.iprouter;
    }

    if (nxthop == 0) {  /* Dest. invalid or no default route*/
      freebuf((char *)pktptr);
      continue;
    }
```

ip.c

```
    /* Use ARP to resolve next-hop address */

    retval = arp_resolve(nxthop, pktptr->net_ethdst);
    if (retval != OK) {
      freebuf((char *)pktptr);
      continue;
    }

    /* Use ipout to Convert byte order and send */

    ip_out(pktptr);
  }
}
```

ip.c

```
/*------------------------------------------------------------------------
 *  ip_enqueue  -  Deposit an outgoing IP datagram on the IP output queue
 *------------------------------------------------------------------------
 */
status  ip_enqueue(
    struct netpacket *pktptr  /* Pointer to the packet  */
  )
{
  intmask mask;      /* Saved interrupt mask   */
  struct  iqentry *iptr;     /* Ptr. to network output queue */

  /* Ensure only one process accesses output queue at a time */

  mask = disable();

  /* Enqueue packet on network output queue */

  iptr = &ipoqueue;
```

ip.c

```
  if (semcount(iptr->iqsem) >= IP_OQSIZ) {
    kprintf("ipout: output queue overflow\n");
    freebuf((char *)pktptr);
    restore(mask);
    return SYSERR;
  }
  iptr->iqbuf[iptr->iqtail++] = pktptr;
  if (iptr->iqtail >= IP_OQSIZ) {
    iptr->iqtail = 0;
  }
  signal(iptr->iqsem);
  restore(mask);
  return OK;
}
```

udp.h

```
/* udp.h - Declarations pertaining to User Datagram Protocol (UDP) */

#define UDP_SLOTS 6     /* Number of open UDP endpoints */
#define UDP_QSIZ  8   /* Packets enqueued per endpoint*/

#define UDP_DHCP_CPORT  68    /* Port number for DHCP client  */
#define UDP_DHCP_SPORT  67    /* Port number for DHCP server  */

/* Constants for the state of an entry */

#define UDP_FREE  0     /* Entry is unused     */
#define UDP_USED  1   /* Entry is being used    */
#define UDP_RECV  2   /* Entry has a process waiting  */

#define UDP_ANYIF -2    /* Register an endpoint for any */
          /*   interface on the machine */

#define UDP_HDR_LEN 8   /* Bytes in a UDP header  */
```

udp.h

```
struct  udpentry {      /* Entry in the UDP endpoint tbl*/
  int32 udstate;     /* State of entry: free/used  */
  uint32  udremip;     /* Remote IP address (zero  */
          /*  means "don't care")  */
  uint16  udremport;    /* Remote protocol port number  */
  uint16  udlocport;    /* Local protocol port number */
  int32 udhead;     /* Index of next packet to read */
  int32 udtail;     /* Index of next slot to insert */
  int32 udcount;    /* Count of packets enqueued  */
  pid32 udpid;      /* ID of waiting process  */
  struct  netpacket *udqueue[UDP_QSIZ];/* Circular packet queue */
};

extern  struct  udpentry udptab[];
```

udp.c

```
/* udp.c - udp_init, udp_in, udp_register, udp_send, udp_sendto,  */
/*         udp_recv, udp_recvaddr, udp_release, udp_ntoh, udp_hton */

#include <xinu.h>

struct  udpentry udptab[UDP_SLOTS]; /* Table of UDP endpoints */

/*------------------------------------------------------------------------
 * udp_init  -  Initialize all entries in the UDP endpoint table
 *------------------------------------------------------------------------
 */
void  udp_init(void)
{

  int32 i;       /* Index into the UDP table */

  for(i=0; i<UDP_SLOTS; i++) {
    udptab[i].udstate = UDP_FREE;
  }

  return;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_in  -  Handle an incoming UDP packet
 *------------------------------------------------------------------------
 */
void  udp_in(
    struct netpacket *pktptr  /* Pointer to the packet  */
  )
{
  intmask mask;     /* Saved interrupt mask   */
  int32 i;       /* Index into udptab    */
  struct  udpentry *udptr;  /* Pointer to a udptab entry  */


  /* Ensure only one process can access the UDP table at a time */

  mask = disable();

  for (i=0; i<UDP_SLOTS; i++) {
      udptr = &udptab[i];
      if (udptr->udstate == UDP_FREE) {
      continue;
      }
```

udp.c

```
    if ((pktptr->net_udpdport == udptr->udlocport)  &&
                  ((udptr->udremport == 0) ||
                    (pktptr->net_udpsport == udptr->udremport)) &&
               (  ((udptr->udremip==0)     ||
                    (pktptr->net_ipsrc == udptr->udremip)))     ) {

  /* Entry matches incoming packet */

  if (udptr->udcount < UDP_QSIZ) {
    udptr->udcount++;
    udptr->udqueue[udptr->udtail++] = pktptr;
    if (udptr->udtail >= UDP_QSIZ) {
      udptr->udtail = 0;
    }
    if (udptr->udstate == UDP_RECV) {
      udptr->udstate = UDP_USED;
      send (udptr->udpid, OK);
    }
    restore(mask);
    return;
  }
 }
}
```

udp.c

```
  /* No match - simply discard packet */

  freebuf((char *) pktptr);
  restore(mask);
  return;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_register  -  Register a remote IP, remote port & local port to
 *           receive incoming UDP messages from the specified
 *           remote site sent to the specified local port
 *------------------------------------------------------------------------
 */
uid32 udp_register (
   uint32 remip,       /* Remote IP address or zero  */
   uint16 remport,     /* Remote UDP protocol port */
   uint16 locport      /* Local UDP protocol port  */
  )
{
  intmask mask;      /* Saved interrupt mask   */
  int32 slot;      /* Index into udptab    */
  struct  udpentry *udptr;  /* Pointer to udptab entry  */

  /* Ensure only one process can access the UDP table at a time */

  mask = disable();
```

udp.c

```
  /* See if request already registered */

  for (slot=0; slot<UDP_SLOTS; slot++) {
    udptr = &udptab[slot];
    if (udptr->udstate == UDP_FREE) {
      continue;
    }

    /* Look at this entry in table */

    if ( (remport == udptr->udremport) &&
         (locport == udptr->udlocport) &&
         (remip   == udptr->udremip   ) ) {

      /* Request is already in the table */

      restore(mask);
      return SYSERR;
    }
  }
```

udp.c

```
  /* Find a free slot and allocate it */

  for (slot=0; slot<UDP_SLOTS; slot++) {
    udptr = &udptab[slot];
    if (udptr->udstate != UDP_FREE) {
      continue;
    }
    udptr->udlocport = locport;
    udptr->udremport = remport;
    udptr->udremip = remip;
    udptr->udcount = 0;
    udptr->udhead = udptr->udtail = 0;
    udptr->udpid = -1;
    udptr->udstate = UDP_USED;
    restore(mask);
    return slot;
  }

  restore(mask);
  return SYSERR;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_recv  -  Receive a UDP packet
 *------------------------------------------------------------------------
 */
int32 udp_recv (
   uid32  slot,      /* Slot in table to use   */
   char   *buff,      /* Buffer to hold UDP data  */
   int32  len,       /* Length of buffer    */
   uint32 timeout      /* Read timeout in msec    */
  )
{
  intmask mask;      /* Saved interrupt mask   */
  struct  udpentry *udptr;  /* Pointer to udptab entry  */
  umsg32  msg;       /* Message from recvtime()  */
  struct  netpacket *pkt;   /* Pointer to packet being read */
  int32 i;       /* Counts bytes copied      */
  int32 msglen;      /* Length of UDP data in packet */
  char  *udataptr;     /* Pointer to UDP data     */

  /* Ensure only one process can access the UDP table at a time */

  mask = disable();
```

udp.c

```
/* Verify that the slot is valid */

if ((slot < 0) || (slot >= UDP_SLOTS)) {
  restore(mask);
  return SYSERR;
}

/* Get pointer to table entry */

udptr = &udptab[slot];

/* Verify that the slot has been registered and is valid */

if (udptr->udstate != UDP_USED) {
  restore(mask);
  return SYSERR;
}
```

udp.c

```
/* Wait for a packet to arrive */

if (udptr->udcount == 0) {     /* No packet is waiting */
  udptr->udstate = UDP_RECV;
  udptr->udpid = currpid;
  msg = recvclr();
  msg = recvtime(timeout);  /* Wait for a packet  */
  udptr->udstate = UDP_USED;
  if (msg == TIMEOUT) {
    restore(mask);
    return TIMEOUT;
  } else if (msg != OK) {
    restore(mask);
    return SYSERR;
  }
}
```

udp.c

```
  /* Packet has arrived -- dequeue it */

  pkt = udptr->udqueue[udptr->udhead++];
  if (udptr->udhead >= UDP_QSIZ) {
    udptr->udhead = 0;
  }
  udptr->udcount--;

  /* Copy UDP data from packet into caller's buffer */

  msglen = pkt->net_udplen - UDP_HDR_LEN;
  udataptr = (char *)pkt->net_udpdata;
  if (len < msglen) {
    msglen = len;
  }
  for (i=0; i<msglen; i++) {
    *buff++ = *udataptr++;
  }
  freebuf((char *)pkt);
  restore(mask);
  return msglen;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_recvaddr  -  Receive a UDP packet and record the sender's address
 *------------------------------------------------------------------------
 */
int32 udp_recvaddr (
   uid32  slot,      /* Slot in table to use   */
   uint32 *remip,      /* Loc for remote IP address  */
   uint16 *remport,    /* Loc for remote protocol port */
   char   *buff,       /* Buffer to hold UDP data  */
   int32  len,      /* Length of buffer    */
   uint32 timeout      /* Read timeout in msec    */
  )
{
  intmask mask;      /* Saved interrupt mask    */
  struct  udpentry *udptr;  /* Pointer to udptab entry  */
  umsg32  msg;       /* Message from recvtime()  */
  struct  netpacket *pkt;   /* Pointer to packet being read */
  int32 msglen;      /* Length of UDP data in packet */
  int32 i;        /* Counts bytes copied     */
  char   *udataptr;    /* Pointer to UDP data     */

  /* Ensure only one process can access the UDP table at a time */

  mask = disable();
```

udp.c

```
/* Verify that the slot is valid */

if ((slot < 0) || (slot >= UDP_SLOTS)) {
  restore(mask);
  return SYSERR;
}

/* Get pointer to table entry */

udptr = &udptab[slot];

/* Verify that the slot has been registered and is valid */

if (udptr->udstate != UDP_USED) {
  restore(mask);
  return SYSERR;
}
```

udp.c

```
/* Wait for a packet to arrive */

if (udptr->udcount == 0) {   /* No packet is waiting */
  udptr->udstate = UDP_RECV;
  udptr->udpid = currpid;
  msg = recvclr();
  msg = recvtime(timeout);  /* Wait for a packet  */
  udptr->udstate = UDP_USED;
  if (msg == TIMEOUT) {
    restore(mask);
    return TIMEOUT;
  } else if (msg != OK) {
    restore(mask);
    return SYSERR;
  }
}

/* Packet has arrived -- dequeue it */

pkt = udptr->udqueue[udptr->udhead++];
if (udptr->udhead >= UDP_QSIZ) {
  udptr->udhead = 0;
}
```

udp.c

```
  /* Record sender's IP address and UDP port number */

  *remip = pkt->net_ipsrc;
  *remport = pkt->net_udpsport;

  udptr->udcount--;

  /* Copy UDP data from packet into caller's buffer */

  msglen = pkt->net_udplen - UDP_HDR_LEN;
  udataptr = (char *)pkt->net_udpdata;
  if (len < msglen) {
    msglen = len;
  }
  for (i=0; i<msglen; i++) {
    *buff++ = *udataptr++;
  }
  freebuf((char *)pkt);
  restore(mask);
  return msglen;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_send  -  Send a UDP packet using info in a UDP table entry
 *------------------------------------------------------------------------
 */
status udp_send (
   uid32  slot,     /* Table slot to use    */
   char   *buff,     /* Buffer of UDP data   */
   int32  len     /* Length of data in buffer */
  )
{
  intmask mask;     /* Saved interrupt mask    */
  struct  netpacket *pkt;   /* Pointer to packet buffer */
  int32 pktlen;     /* Total packet length     */
  static  uint16 ident = 1; /* Datagram IDENT field   */
  char  *udataptr;     /* Pointer to UDP data    */
  uint32  remip;      /* Remote IP address to use */
  uint16  remport;     /* Remote protocol port to use */
  uint16  locport;     /* Local protocol port to use */
  uint32  locip;      /* Local IP address taken from */
          /*  the interface      */
  struct  udpentry *udptr;  /* Pointer to table entry */

  /* Ensure only one process can access the UDP table at a time */

  mask = disable();
```

udp.c

```
/* Verify that the slot is valid */

if ( (slot < 0) || (slot >= UDP_SLOTS) ) {
  restore(mask);
  return SYSERR;
}

/* Get pointer to table entry */

udptr = &udptab[slot];

/* Verify that the slot has been registered and is valid */

if (udptr->udstate == UDP_FREE) {
  restore(mask);
  return SYSERR;
}

/* Verify that the slot has a specified remote address */

remip = udptr->udremip;
if (remip == 0) {
  restore(mask);
  return SYSERR;
}
```

udp.c

```
locip = NetData.ipucast;
remport = udptr->udremport;
locport = udptr->udlocport;

/* Allocate a network buffer to hold the packet */

pkt = (struct netpacket *)getbuf(netbufpool);

if ((int32)pkt == SYSERR) {
  restore(mask);
  return SYSERR;
}

/* Compute packet length as UDP data size + fixed header size */

pktlen = ((char *)&pkt->net_udpdata - (char *)pkt) + len;
```

udp.c

```
/* Create a UDP packet in pkt */

memcpy((char *)pkt->net_ethsrc,NetData.ethucast,ETH_ADDR_LEN);
pkt->net_ethtype = 0x0800;  /* Type is IP      */
pkt->net_ipvh = 0x45;   /* IP version and hdr length  */
pkt->net_iptos = 0x00;    /* Type of service    */
pkt->net_iplen= pktlen - ETH_HDR_LEN;/* Total IP datagram length*/
pkt->net_ipid = ident++;  /* Datagram gets next IDENT */
pkt->net_ipfrag = 0x0000; /* IP flags & fragment offset */
pkt->net_ipttl = 0xff;    /* IP time-to-live     */
pkt->net_ipproto = IP_UDP;  /* Datagram carries UDP   */
pkt->net_ipcksum = 0x0000;  /* initial checksum    */
pkt->net_ipsrc = locip;   /* IP source address     */
pkt->net_ipdst = remip;   /* IP destination address */

pkt->net_udpsport = locport;  /* Local UDP protocol port  */
pkt->net_udpdport = remport;  /* Remote UDP protocol port */
pkt->net_udplen = (uint16)(UDP_HDR_LEN+len); /* UDP length  */
pkt->net_udpcksum = 0x0000; /* Ignore UDP checksum     */
udataptr = (char *) pkt->net_udpdata;
for (; len>0; len--) {
  *udataptr++ = *buff++;
}
```

udp.c

```
  /* Call ipsend to send the datagram */

  ip_send(pkt);
  restore(mask);
  return OK;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_sendto  -  Send a UDP packet to a specified destination
 *------------------------------------------------------------------------
 */
status  udp_sendto (
   uid32  slot,      /* UDP table slot to use  */
   uint32 remip,       /* Remote IP address to use */
   uint16 remport,     /* Remote protocol port to use  */
   char   *buff,       /* Buffer of UDP data   */
   int32  len      /* Length of data in buffer */
  )
{
  intmask mask;      /* Saved interrupt mask   */
  struct  netpacket *pkt;   /* Pointer to a packet buffer */
  int32 pktlen;      /* Total packet length     */
  static  uint16 ident = 1; /* Datagram IDENT field   */
  struct  udpentry *udptr;  /* Pointer to a UDP table entry */
  char  *udataptr;     /* Pointer to UDP data    */

  /* Ensure only one process can access the UDP table at a time */

  mask = disable();
```

udp.c

```
  /* Verify that the slot is valid */

  if ( (slot < 0) || (slot >= UDP_SLOTS) ) {
    restore(mask);
    return SYSERR;
  }

  /* Get pointer to table entry */

  udptr = &udptab[slot];

  /* Verify that the slot has been registered and is valid */

  if (udptr->udstate == UDP_FREE) {
    restore(mask);
    return SYSERR;
  }

  /* Allocate a network buffer to hold the packet */

  pkt = (struct netpacket *)getbuf(netbufpool);

  if ((int32)pkt == SYSERR) {
    restore(mask);
    return SYSERR;
  }
```

udp.c

```
  /* Compute packet length as UDP data size + fixed header size */

  pktlen = ((char *)&pkt->net_udpdata - (char *)pkt) + len;

  /* Create UDP packet in pkt */

  memcpy((char *)pkt->net_ethsrc,NetData.ethucast,ETH_ADDR_LEN);
        pkt->net_ethtype = 0x0800;  /* Type is IP */
  pkt->net_ipvh = 0x45;    /* IP version and hdr length  */
  pkt->net_iptos = 0x00;     /* Type of service    */
  pkt->net_iplen= pktlen - ETH_HDR_LEN;/* total IP datagram length*/
  pkt->net_ipid = ident++;  /* Datagram gets next IDENT */
  pkt->net_ipfrag = 0x0000; /* IP flags & fragment offset */
  pkt->net_ipttl = 0xff;     /* IP time-to-live    */
  pkt->net_ipproto = IP_UDP;  /* Datagram carries UDP    */
  pkt->net_ipcksum = 0x0000;  /* Initial checksum    */
```

udp.c

```
  pkt->net_ipsrc = NetData.ipucast;/* IP source address    */
  pkt->net_ipdst = remip;    /* IP destination address */
  pkt->net_udpsport = udptr->udlocport;/* local UDP protocol port */
  pkt->net_udpdport = remport;  /* Remote UDP protocol port */
  pkt->net_udplen = (uint16)(UDP_HDR_LEN+len); /* UDP length  */
  pkt->net_udpcksum = 0x0000; /* Ignore UDP checksum    */
  udataptr = (char *) pkt->net_udpdata;
  for (; len>0; len--) {
    *udataptr++ = *buff++;
  }

  /* Call ipsend to send the datagram */

  ip_send(pkt);
  restore(mask);
  return OK;
}
```

udp.c

```
/*------------------------------------------------------------------------
 * udp_release  -  Release a previously-registered UDP slot
 *------------------------------------------------------------------------
 */
status  udp_release (
   uid32  slot        /* Table slot to release  */
  )
{
  intmask mask;       /* Saved interrupt mask   */
  struct  udpentry *udptr;  /* Pointer to udptab entry  */
  struct  netpacket *pkt;   /* pointer to packet being read */

  /* Ensure only one process can access the UDP table at a time */

  mask = disable();

  /* Verify that the slot is valid */

  if ( (slot < 0) || (slot >= UDP_SLOTS) ) {
    restore(mask);
    return SYSERR;
  }

  /* Get pointer to table entry */

  udptr = &udptab[slot];
```

udp.c

```
  /* Verify that the slot has been registered and is valid */

  if (udptr->udstate == UDP_FREE) {
    restore(mask);
    return SYSERR;
  }

  /* Defer rescheduling to prevent freebuf from switching context */

  resched_cntl(DEFER_START);
  while (udptr->udcount > 0) {
    pkt = udptr->udqueue[udptr->udhead++];
    if (udptr->udhead >= UDP_QSIZ) {
      udptr->udhead = 0;
    }
    freebuf((char *)pkt);
    udptr->udcount--;
  }
  udptr->udstate = UDP_FREE;
  resched_cntl(DEFER_STOP);
  restore(mask);
  return OK;
}
```

Summary

- Simple protocol stack
- Operation similar to device drivers with components coupled by queues
- Message passing mechanism used like a soft interrupt (with timeout)