sudo ./run

Two interfaces created: os0, os1

Two networks created: (add to /etc/networks)
  peanut where os0 will connect – 192.168.0.0
  grape where os1 will connect – 192.168.1.0

Two IP addresses in peanut: (add to /etc/hosts)
  butter-near - 192.168.0.1
  butter-far   - 192.168.0.2

Two IP addresses in grape:  (add to /etc/hosts)
  jelly-near  - 192.168.1.1
  jelly-far   - 192.168.1.2
route:

Kernel IP routing table

<table>
<thead>
<tr>
<th>Destination</th>
<th>Gateway</th>
<th>Genmask</th>
<th>Flags</th>
<th>Metric</th>
<th>Ref</th>
<th>Use</th>
<th>Iface</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>10.52.240.1</td>
<td>0.0.0.0</td>
<td>UG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth1</td>
</tr>
<tr>
<td>10.52.240.0</td>
<td>*</td>
<td>255.255.240.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth1</td>
</tr>
<tr>
<td>peanut</td>
<td>*</td>
<td>255.255.255.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>os0</td>
</tr>
<tr>
<td>grape</td>
<td>*</td>
<td>255.255.255.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>os1</td>
</tr>
</tbody>
</table>

ssh butter-near:

Last login: Tue Dec 4 08:12:06 2012 from jelly-near

You have no mail.

08:35:31 up 2:54, 8 users, load average: 0.39, 0.37, 0.30

[franco@franco ~]$}

ping jelly-near:

PING jelly-near (192.168.1.2) 56(84) bytes of data.
64 bytes from jelly-near (192.168.1.2): icmp_req=1 ttl=64 time=0.046 ms
64 bytes from jelly-near (192.168.1.2): icmp_req=2 ttl=64 time=0.051 ms

sudo tcpdump -i os0:

tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on os0, link-type EN10MB (Ethernet), capture size 65535 bytes
08:40:08.847331 IP butter-far > butter-near: ICMP echo request, id 10511, seq 1, length 64
08:40:08.847365 IP butter-near > butter-far: ICMP echo reply, id 10511, seq 1, length 64
Device Registration

No major/minor numbers with network drivers
Instead, driver adds a data structure for each detected interface into a global list of network devices
Interfaces are described in:

```
struct net_device (in linux/netdevice.h)
```

Keep track of os0 and os1 with this:

```
struct net_device *os0, *os1;
```

which are allocated with this:

```
os0=alloc_etherdev(struct sizeof(struct os_priv));
```

where

```
struct os_priv {
    struct net_device_stats stats;
    struct os_packet *pkt;
    struct sk_buff *skb; /* Socket buffer */
    struct net_device *dev;
};
```
Device Initialization

Initialization must be complete before calling `register_netdev`

Initialization is in `init_module`:

```c
os0->dev_addr[i]  set MAC address octets
os0->broadcast[i] set broadcast octets
os0->netdv_ops = &os0_device_ops;
os0->header_ops = &os0_header_ops;
mem_cpy(os0->name, “os0\0”, 4) copies device name
os0->flags |= IFF_NOARP; disable ARP
priv = netdev_priv(os0); access private area
priv->dev = os0;
priv->pkt = kmalloc(sizeof(struct os_pkt)..);
priv->pkt->dev = os0;
register_netdev(os0); register device
```
Device Operations

**open:**
- Opens the interface when `ifconfig` activates it
- Registers system resource it needs (I/O ports, IRQ, DMA, etc.)
- Turns on the hardware
- Performs any other setup the device requires

**stop:**
- Stops the interface when it is brought down
- Should reverse operations performed in `open`

**hard_start_xmit:**
- Transmission of packet contained in socket buffer (sk_buff)

**hard_header:**
- Builds the hardware header from src and dest hrdw addresses previously retrieved
- Organizes info passed to it into device-specific hrdw header
Hardware Header

ETHERNET FRAME
- Destination ethernet address
- Source ethernet address
- Protocol
- Data
- Checksum

IP PACKET
- Length
- Protocol
- Checksum
- Source IP address
- Destination IP address
- Data

TCP PACKET
- Source TCP address
- Destination TCP address
- SEQ
- ACK
- Data
Device Operations

`rebuild_header:`
Rebuilds hardware header after ARP resolution completes but before a packet is transmitted

`get_stats:`
Exec when `ifconfig` or `netstat -i` is run

`set_mac_address:`
Can be implemented if interface supports it

`do_ioctl:`
Performs interface-specific ioctl commands
The corresponding field in `struct net_device` can be left as `NULL` if the interface doesn’t need any interface-specific commands.
Device Fields

void *priv:
  Private data (see os_priv)

unsigned long last_rx:
  Time since last receive

int watchdog_timeo:
  minimum time elapsed before deciding on a timeout

int xmit_lock_owner:
  Used to avoid multiple simultaneous calls to the driver’s hard_start_xmit function
  Is the number of the CPU that has obtained xmit_lock
Open Device

os_open:
Assign the hardware address of the board:
Use "\1\2\3\4\5\6" >> 01:02:03:04:05:06
the first byte of multicast addr is odd so this is even.

Start the interface's transmit queue allowing it to accept packets for transmission once it is ready to start sending data.
Close Device

**os_release:**
Just stop the transmit queue
Packet Transmission

int os_tx:
Call hard_start_xmit to put data in an outgoing queue

Each packet is in a socket buffer structure (struct sk_buff) and is complete including transmission headers.

Actual transmission is hardware dependent.
Code for this is in snull_hw_tx.
packet is manipulated and finally enqueued with
os_nqueue_buf(dest, tx_buffer);
which links the packet (tx_buffer) into the queue.

Scatter/Gather I/O:
Packet data & headers may be copied from user space.
This may mean a lot of data copying.
May avoid this with scatter/gather I/O – send the pieces as one chunk.
Packet Reception

**int os_rx:**
Called on interrupt from os_interrupt

Allocate an `sk_buff`, fill with packet, write metadata, send to receive level with `netif_rx`

packet data is copied with

```
memcpy(skb_put(skb, pkt->len), pkt->data, pkt->len);
```

The `dev`, `protocol` fields must be filled so network level can make sense of the packet

**Scatter/Gather I/O:**
Packet data & headers may be copied from user space
This may mean a lot of data copying
May avoid this with scatter/gather I/O – send the pieces as one chunk
Interrupt Handler

```c
int os_regular_interrupt:
    Retrieve priv from netdev_priv(dev)
    Get statusword from priv
    If statusword is OK and interrupt is from receiver
    get packet from priv->rx_queue
    Dequeue packet with priv->rx_queue = pkt->next
    Send packet to network level with netif_rx(dev, pkt)
    level can make sense of the packet
    
Otherwise if statusword OK and interrupt is from Xmit
    free the sk_buff (transmission is over)
```
Socket Buffer

Fields

struct net_device:
    The device sending or receiving data

union {} h, nh, mac:
    Pointers to levels of headers contained within the packet
    h hosts pointers to transport layers
    nh includes network layer headers
    mac collects pointers to link-layer headers
    src and dest addresses are in skb->h.th

unsigned char *head, *tail, *end, *data:
    Pointers to space in the packets

unsigned char ip_summed:
    Checksum policy
Socket Buffer

Functions

`struct sk_buff *alloc_skb(len, priority):`
Allocate the buffer

`void dev_kfree_skb(skb):`
Free a buffer

`unsigned char *skb_put(skb, len):`
Update the tail and len fields of the sk_buff structure
Drivers use the return value to copy data with memcpy()

`unsigned char *skb_push:`
Decrement skb->data and increment skb->len
Used to add a hardware header before transmitting a packet

`unsigned char *skb_pull:`
Removes data from the head of the packet