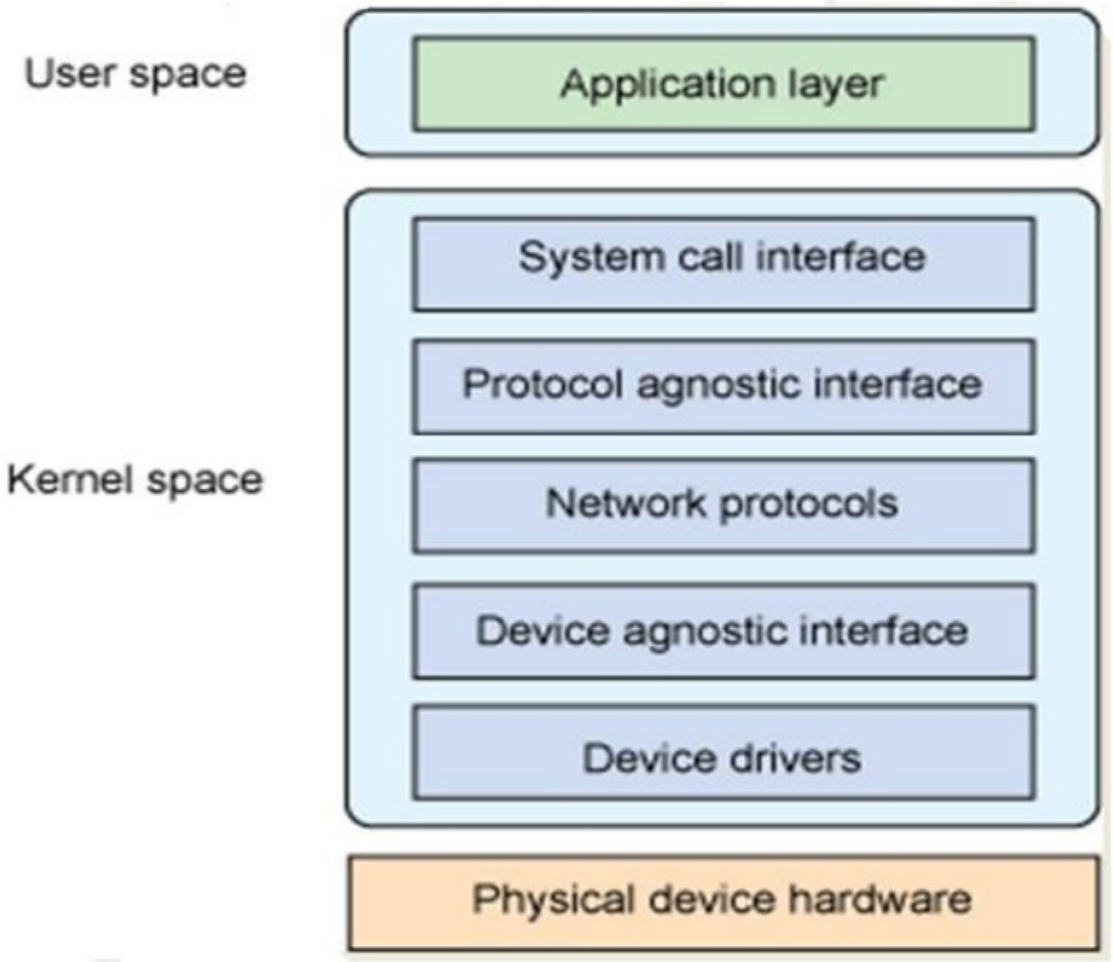
专题15-网卡驱动程序设计

一、网卡驱动架构分析

1.1、Linux网络子系统



系统调用接口层
为应用程序提供访问网络子系统的统一方法。
协议无关层
提供通用的方法来使用传输层协议。
协议栈的实现
实现具体的网络协议
设备无关层
协议与设备驱动之前通信的通用接口
设备驱动程序

1.2、重要数据结构

1.2.1、网卡描述结构

在Linux内核中，每个网卡都由一个net_device结构来 描述，其中的一些重要成员有：

```
char name[IFNAMSIZ]
```

设备名,如：eth%d

```
unsigned long base_addr
```

I/O 基地址

```
const struct net_device_ops *netdev_ops;
```

1.2.2、网卡操作集合

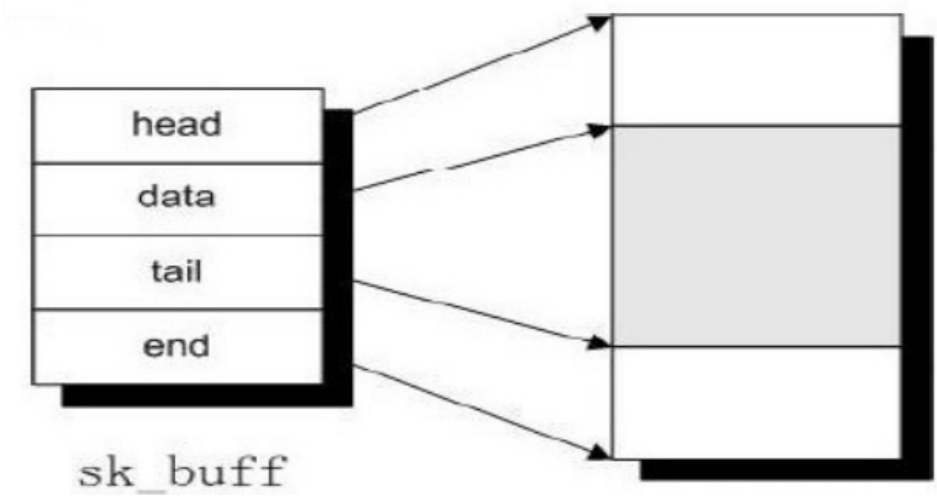
类似于字符设备驱动中的file_operations结构， net_device_ops结构记录了网卡所支持的操作。

```
static const struct net_device_ops dm9000_netdev_ops =
{
    .ndo_open = dm9000_open,
   .ndo_stop = dm9000_stop,
```

```
    .ndo_start_xmit = dm9000_start_xmit,
    .ndo_do_ioctl = dm9000_ioctl,
    .ndo_validate_addr = eth_validate_addr,
    .ndo_set_mac_address = eth_mac_addr,
};
```

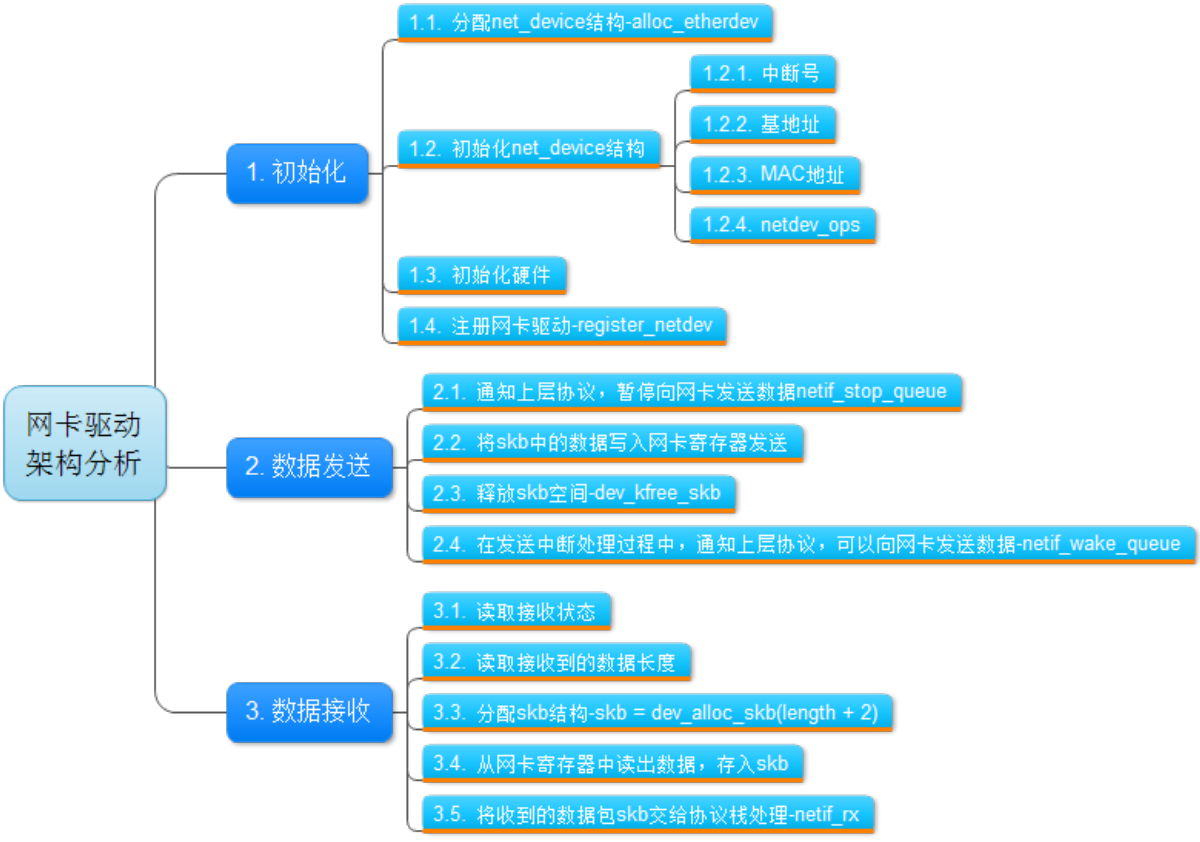
1.2.3、网络数据包

Linux内核中的每个网络数据包都由一个套接字 缓冲区结构 struct sk_buff 描述，即一个 sk_buff结构就是一个网络包,指向sk_buff的指针通常被称做skb。



1.3、网卡驱动架构分析

cs890x.c



二、回环网卡驱动设计

loopback.c

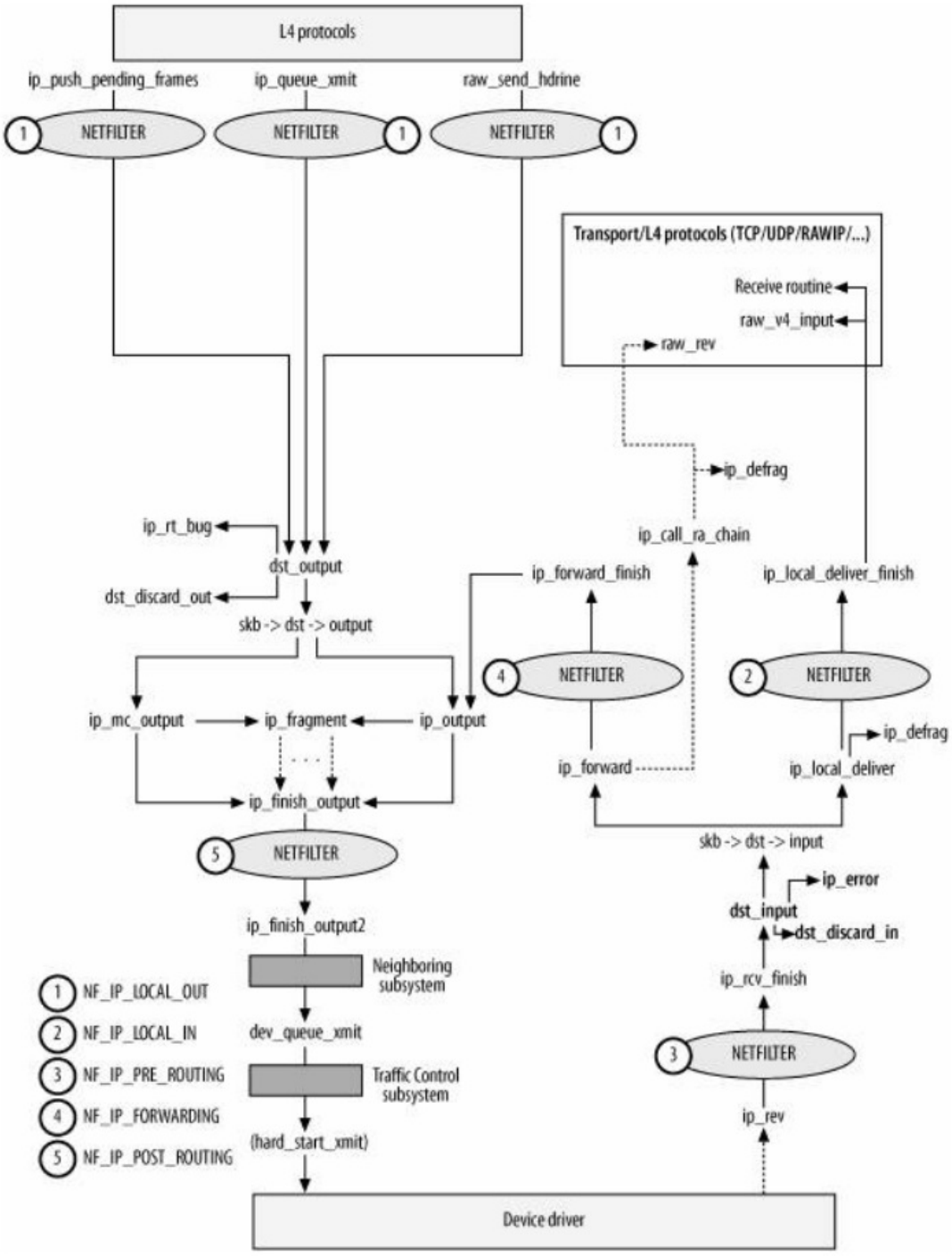
```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
```

```
#include <linux/skbuff.h>
#include <linux/if_ether.h> /* For the statistics structure. */


unsigned long bytes = 0;
unsigned long packets = 0;


static int loopback_xmit(struct sk_buff *skb, struct net_device *dev)
{

 skb->protocol = eth_type_trans(skb,dev);


 bytes += skb->len;
 packets++;

 netif_rx(skb);

 return 0;
}

static struct net_device_stats *loopback_get_stats(struct net_device *dev)
{
 struct net_device_stats *stats = &dev->stats;

 stats->rx_packets = packets;
 stats->tx_packets = packets;
 stats->rx_bytes = bytes;
 stats->tx_bytes = bytes;
 return stats;
}

static const struct net_device_ops loopback_ops = {
 .ndo_start_xmit= loopback_xmit,
 .ndo_get_stats = loopback_get_stats,
};

/*
 * The loopback device is special. There is only one instance
 * per network namespace.
 */
static void loopback_setup(struct net_device *dev)
{
 dev->mtu = (16 * 1024) + 20 + 20 + 12;
 dev->flags = IFF_LOOPBACK;
 dev->header_ops = &eth_header_ops;
 dev->netdev_ops = &loopback_ops;

}

/* Setup and register the loopback device. */
static __net_init int loopback_net_init(struct net *net)
{
 struct net_device *dev;
 int err;
 err = -ENOMEM;
 dev = alloc_netdev(0, "lo", loopback_setup);
 if (!dev)
  goto out;

 err = register_netdev(dev);
 if (err)
  goto out_free_netdev;

 net->loopback_dev = dev;
 return 0;
```

```
out_free_netdev:
 free_netdev(dev);
out:
 if (net == &init_net)
  panic("loopback: Failed to register netdevice: %d\n", err);
 return err;
}

static __net_exit void loopback_net_exit(struct net *net)
{
 struct net_device *dev = net->loopback_dev;

 unregister_netdev(dev);
}

/* Registered in net/core/dev.c */
struct pernet_operations __net_initdata loopback_net_ops = {
      .init = loopback_net_init,
      .exit = loopback_net_exit,
};
```

三、网络子系统深度剖析

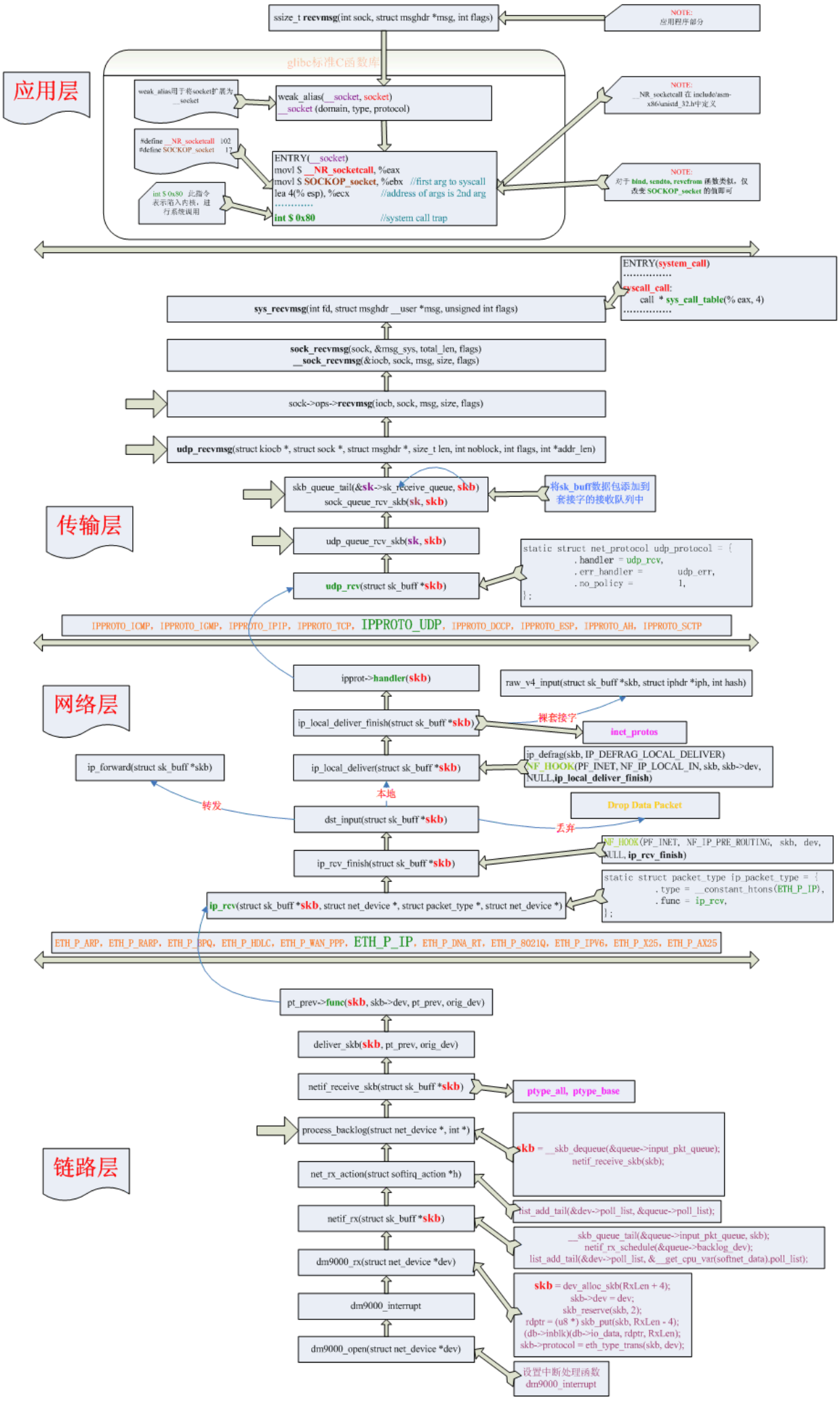
3.1、网络包发送模型



3.1.1、选择路由

3.1.2、邻居子系统

3.2、网络包接收模型


ssize_t recvmsg(int sock, struct msghdr *msg, int flags)
NOTE:
应用程序部分
glibc标准C函数库
应用层
weak_alias用于将socket扩展为__socket
weak_alias(__socket, socket)
__socket (domain, type, protocol)
#define __NR_socketcall 102
#define SOCKOP_socket 17
int $ 0x80 此指令表示陷入内核，进行系统调用
ENTRY(__socket)
movl $ __NR_socketcall, %eax
movl $ SOCKOP_socket, %ebx //first arg to syscall
lea 4(% esp), %ecx //address of args is 2nd arg
.....
int $ 0x80 //system call trap
NOTE:
__NR_socketcall 在 include\asm-x86\unistd_32.h中定义
NOTE:
对于 bind, sendto, revcfrom 函数类似，仅改变 SOCKOP_socket 的值即可
ENTRY(system_call)
.....
syscall_call:
call * sys_call_table(% eax, 4)
.....
sys_recvmsg(int fd, struct msghdr __user *msg, unsigned int flags)
sock_recvmsg(sock, &msg_sys, total_len, flags)
__sock_recvmsg(&iocb, sock, msg, size, flags)
sock->ops->recvmsg(iocb, sock, msg, size, flags)
udp_recvmsg(struct kiocb *, struct sock *, struct msghdr *, size_t len, int noblock, int flags, int *addr_len)
skb_queue_tail(&sk->sk_receive_queue, skb)
sock_queue_rcv_skb(sk, skb)
将sk_buff数据包添加到套接字的接收队列中
传输层
udp_queue_rcv_skb(sk, skb)
udp_rcv(struct sk_buff *skb)
static struct net_protocol udp_protocol = {
.handler = udp_rcv,
.err_handler = udp_err,
.no_policy = 1,
};
IPPROTO_ICMP, IPPROTO_IGMP, IPPROTO_IPIP, IPPROTO_TCP, IPPROTO_UDP, IPPROTO_DCCP, IPPROTO_ESP, IPPROTO_AH, IPPROTO_SCTP
ipprot->handler(skb)
raw_v4_input(struct sk_buff *skb, struct iphdr *iph, int hash)
网络层
ip_local_deliver_finish(struct sk_buff *skb)
裸套接字
inet_protos
ip_forward(struct sk_buff *skb)
ip_local_deliver(struct sk_buff *skb)
ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER)
NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,ip_local_deliver_finish)
转发
本地
Drop Data Packet
dst_input(struct sk_buff *skb)
丢弃
ip_rcv_finish(struct sk_buff *skb)
NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish)
ip_rcv(struct sk_buff *skb, struct net_device *, struct packet_type *, struct net_device *)
static struct packet_type ip_packet_type = {
.type = __constant_htons(ETH_P_IP),
.func = ip_rcv,
};
ETH_P_ARP, ETH_P_RARP, ETH_P_SPQ, ETH_P_HDLC, ETH_P_WAN_PPP, ETH_P_IP, ETH_P_DNA_RT, ETH_P_8021Q, ETH_P_IPV6, ETH_P_X25, ETH_P_AX25
pt_prev->func(skb, skb->dev, pt_prev, orig_dev)
deliver_skb(skb, pt_prev, orig_dev)
netif_receive_skb(struct sk_buff *skb)
ptype_all, ptype_base
process_backlog(struct net_device *, int *)
skb = __skb_dequeue(&queue->input_pkt_queue);
netif_receive_skb(skb);
链路层
net_rx_action(struct softirq_action *h)
list_add_tail(&dev->poll_list, &queue->poll_list);
netif_rx(struct sk_buff *skb)
__skb_queue_tail(&queue->input_pkt_queue, skb);
netif_rx_schedule(&queue->backlog_dev);
list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);
dm9000_rx(struct net_device *dev)
dm9000_interrupt
skb = dev_alloc_skb(RxLen + 4);
skb->dev = dev;
skb_reserve(skb, 2);
rdptr = (u8 *) skb_put(skb, RxLen - 4);
(db->inblk)(db->io_data, rdptr, RxLen);
skb->protocol = eth_type_trans(skb, dev);
dm9000_open(struct net_device *dev)
设置中断处理函数 dm9000_interrupt

四、DM9000网卡驱动深度分析

4.1、发送

4.2、接收

五、DM9000网卡驱动实现

Linux驱动由Linux驱动模型+裸机驱动构成。