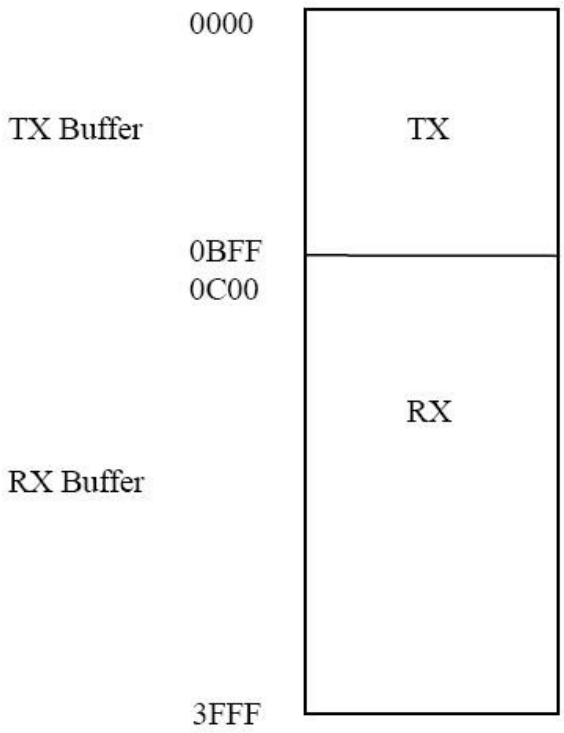
努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处
http://blog.csdn.net/woshixingaaa/archive/2011/06/18/6552911.aspx
这里主要讨论中断处理和数据的发送和接收，这个也是网卡驱动最重要的部分了。
中断处理函数：
发生中断的情况有 3 中:1)接收到数据 2)发送完数据 3)链路状态改变

```
1.  static irqreturn_t dm9000_interrupt(int irq, void *dev_id)
2.  {
3.      struct net_device *dev = dev_id;
4.      board_info_t *db = netdev_priv(dev);
5.      int int_status;
6.      unsigned long flags;
7.      u8 reg_save;
8.
9.      dm9000_dbg(db, 3, "entering %s/n", __func__);
10.
11.     /* A real interrupt coming */
12.
13.     /*保存中断到 flags 中*/
14.     spin_lock_irqsave(&db->lock, flags);
15.
16.     /*保存寄存器地址*/
17.     reg_save = readb(db->io_addr);
18.
19.     /*禁止 DM9000 的所有中断*/
20.     iow(db, DM9000_IMR, IMR_PAR);
21.
22.     /*获得 DM9000 的中断状态*/
23.     int_status = ior(db, DM9000_ISR);   /* Got ISR */
24.     iow(db, DM9000_ISR, int_status);    /* Clear ISR status */
25.
26.     if (netif_msg_intr(db))
27.         dev_dbg(db->dev, "interrupt status %02x/n", int_status);
28.
29.     /*检查 Interrupt Status Register 的第 0 位,看有没有接收数据*/
30.     if (int_status & ISR_PRS)
31.         dm9000_rx(dev);
32.
33.     /*检查 Interrupt Status Register 的第 1 位,看有没有发送完数据*/
34.     if (int_status & ISR_PTS)
```

```
35.     dm9000_tx_done(dev, db);
36.
37.   if (db->type != TYPE_DM9000E) {
38.     /*检查 Interrupt Status Register 的第 5 位,看链路状态有没有变化*/
39.     if (int_status & ISR_LNKCHNG) {
40.         /* fire a link-change request */
41.         schedule_delayed_work(&db->phy_poll, 1);
42.     }
43.   }
44.
45.   /*重新使能相应中断*/
46.   iow(db, DM9000_IMR, db->imr_all);
47.
48.   /*还原原先的 io 地址*/
49.   writeb(reg_save, db->io_addr);
50.   /*还原中断状态*/
51.   spin_unlock_irqrestore(&db->lock, flags);
52.
53.   return IRQ_HANDLED;
54.}
```

下面说一下 DM9000A 中的存储部分，DM9000A 内部有一个 4K Dword SRAM，其中 3KB 是作为发送，16KB 作为接收，如下图所示。其中 0x0000~0x0BFF 是传说中的 TX buffer(TX buffer 中只能存放两个包)，0x0C00~0x3FFF 是 RX buffer。因此在写内存操作时，当 IMR 的第 7 位被设置，如果到达了地址的结尾比如到了 3KB，则回卷到 0。相似的方式，在读操作中，当 IMR 的第 7 位被设置如果到达了地址的结尾比如 16K，则回卷到 0x0C00。



那么传说中的发送函数又是哪个呢，在 probe 函数里进行了初始化函数指针操作。

```
ndev->open       = &dm9000_open;
ndev->hard_start_xmit    = &dm9000_start_xmit;
ndev->tx_timeout        = &dm9000_timeout;
ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
ndev->stop       = &dm9000_stop;
ndev->set_multicast_list = &dm9000_hash_table;
ndev->ethtool_ops     = &dm9000_ethtool_ops;
ndev->do_ioctl       = &dm9000_ioctl;
```

可以看到当上层调用 hard_start_xmit 时，在我们的驱动程序中实际调用的是 dm9000_start_xmit，下面来分析一 dm9000_start_xmit 的源码。

发送函数：

```
/*
 *  Hardware start transmission.
 *  Send a packet to media from the upper layer.
 */
static int
dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    unsigned long flags;
    /*获得 ndev 的私有数据，也就是芯片相关的信息*/
    board_info_t *db = netdev_priv(dev);

    dm9000_dbg(db, 3, "%s:/n", __func__);
    /只能存放两个，如果已经有两个就退出/
```

```
14.   if (db->tx_pkt_cnt > 1)
15.     return 1;
16.
17.   spin_lock_irqsave(&db->lock, flags);
18.   /*
19.    *MWCMD 是 Memory data write command with address increment Register(F8H)
20.    *写数据到 TX SRAM
21.    *写这个命令后，根据 IO 操作模式(8-bit or 16-bit)来增加写指针 1 或 2
22.    */
23.   writeb(DM9000_MWCMD, db->io_addr);
24.   /*把数据从 sk_buff 中拷贝到 TX SRAM 中*/
25.   (db->outblk)(db->io_data, skb->data, skb->len);
26.   /*为了统计发送了多少个字节，这个在使用 ifconfig 中显示出的那个发送了多少个字节就是这里计算的*/
27.   dev->stats.tx_bytes += skb->len;
28.   /*待发送的计数加一*/
29.   db->tx_pkt_cnt++;
30.   /*如果只有一个要发送就立即发送，如果这是第二个就应该进行排队了*/
31.   if (db->tx_pkt_cnt == 1) {
32.     /*把数据的长度填到 TXPLL(发送包长度低字节)和 TXPLH(发送包长度高字节)中*/
33.     iow(db, DM9000_TXPLL, skb->len);
34.     iow(db, DM9000_TXPLH, skb->len >> 8);
35.     /*置发送控制寄存器(TX Control Register)的发送请求位 TXREQ(Auto clears after sending completel
    y)，这样就可以发送出去了*/
36.     iow(db, DM9000_TCR, TCR_TXREQ); /* Cleared after TX complete */
37.     /*
38.      *记下此时的时间，这里起一个时间戳的作用，之后的超时会用到。如果当前的系统时间超过设备的
    trans_start 时间
39.      *至少一个超时周期，网络层将最终调用驱动程序的 tx_timeout。那个这个"一个超时周期"又是什么呢？
    这个是我们在
40.        *probe 函数中设置的,ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
41.      */
42.     dev->trans_start = jiffies;  /* save the time stamp */
43.   } else {
44.     /*如果是第二个包则把 skb->len 复制给队列的 queue_pkt_len，然后告诉上层停止发送队列，因为发送队
    列已经满了*/
45.     db->queue_pkt_len = skb->len;
46.     netif_stop_queue(dev);
47.   }
48.   spin_unlock_irqrestore(&db->lock, flags);
49.   /*每个数据包写入网卡 SRAM 后都要释放 skb*/
50.   dev_kfree_skb(skb);
51.
52.   return 0;
53. }
```

下面来看一下刚才提到的那个超时函数，发送超时函数：

```
1. /* Our watchdog timed out. Called by the networking layer */
2. static void dm9000_timeout(struct net_device *dev)
3. {
4.     board_info_t *db = netdev_priv(dev);
5.     u8 reg_save;
6.     unsigned long flags;
7.
8.     /* Save previous register address */
9.     reg_save = readb(db->io_addr);
10.    spin_lock_irqsave(&db->lock, flags);
11.
12.    netif_stop_queue(dev);
13.    dm9000_reset(db);
14.    dm9000_init_dm9000(dev);
15.    /* We can accept TX packets again */
16.    dev->trans_start = jiffies;
17.    netif_wake_queue(dev);
18.
19.    /* Restore previous register address */
20.    writeb(reg_save, db->io_addr);
21.    spin_unlock_irqrestore(&db->lock, flags);
22.}
```

这个函数首先停止了发送队列，然后复位 dm9000,初始化 dm9000,重新设置了时间戳，然后唤醒发送队列，通知网络子系统可再次传输数据包。

发送完成后的中断处理函数：

```
1. /*
2.  * DM9000 interrupt handler
3.  * receive the packet to upper layer, free the transmitted packet
4.  */
5.
6. static void dm9000_tx_done(struct net_device *dev, board_info_t *db)
7. {
8.     /*从网络状态寄存器(Network Status Register)中获得发送状态*/
9.     int tx_status = ior(db, DM9000_NSR);
10.    /*如果发送状态为 NSR_TX2END(第二个包发送完毕)或 NSR_TX1END(第一个包发送完毕)*/
11.    if (tx_status & (NSR_TX2END | NSR_TX1END)) {
12.        /*如果一个数据包发送完，待发送数据包个数减 1*/
13.        db->tx_pkt_cnt--;
14.        /*如果一个数据包发送完，已发送数据包个数加 1*/
15.        dev->stats.tx_packets++;
16.
17.        if (netif_msg_tx_done(db))
18.            dev_dbg(db->dev, "tx done, NSR %02x/n", tx_status);
19.
20.        /*如果还有数据包要发送*/
```

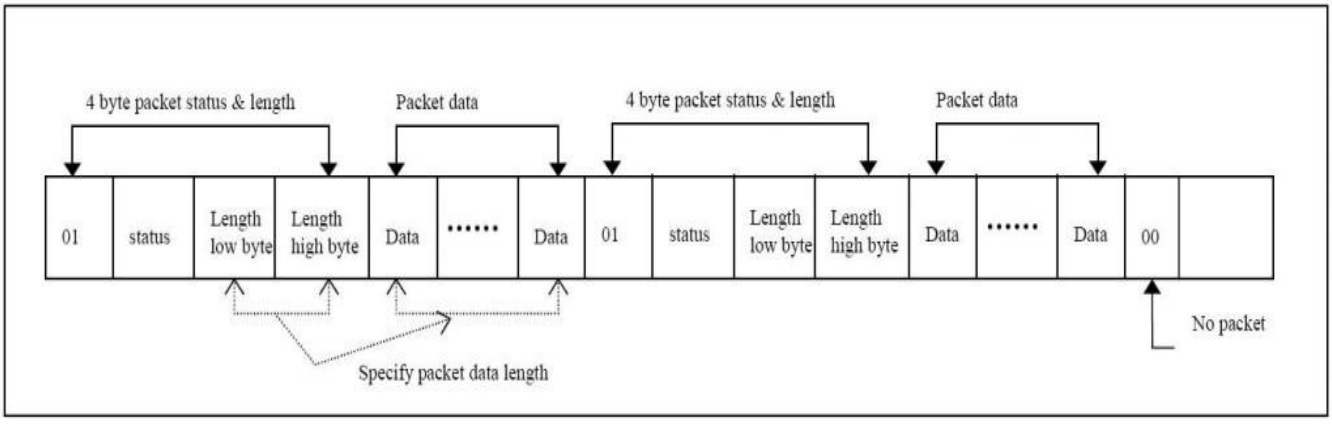
```
    if (db->tx_pkt_cnt > 0) {
        /*将发送的长度放到 TXPLL，TXPLH 寄存器中*/
        iow(db, DM9000_TXPLL, db->queue_pkt_len);
        iow(db, DM9000_TXPLH, db->queue_pkt_len >> 8);
        /*置发送请求位*/
        iow(db, DM9000_TCR, TCR_TXREQ);
        /*保存时间戳*/
        dev->trans_start = jiffies;
    }
    /*通知内核将待发送数据包放入发送队列*/
    netif_wake_queue(dev);
  }
}
```

接收函数：

每接受到一个包，DM9000 都会在包的前面加上 4 个字节，"01H"，status 与 RSR(RX Status Register)的内容相同，"LENL"(数据包长度低位位)，"LENH"(数据包长度高 8 位)。所以首先要读取这 4 个字节来确定数据包的状态，第一个字节"01H"表示接下来的是有效的数据包，"00H"表示没有数据包，若为其他值则表示网卡没有正确初始化，需要从新初始化。这 4 个字节封装在一起:

```
struct dm9000_rxhdr {
  u8  RxPktReady;
  u8  RxStatus;
  __le16  RxLen;
} __attribute__((__packed__));
```

清晰一些如下图：



接收函数如下：

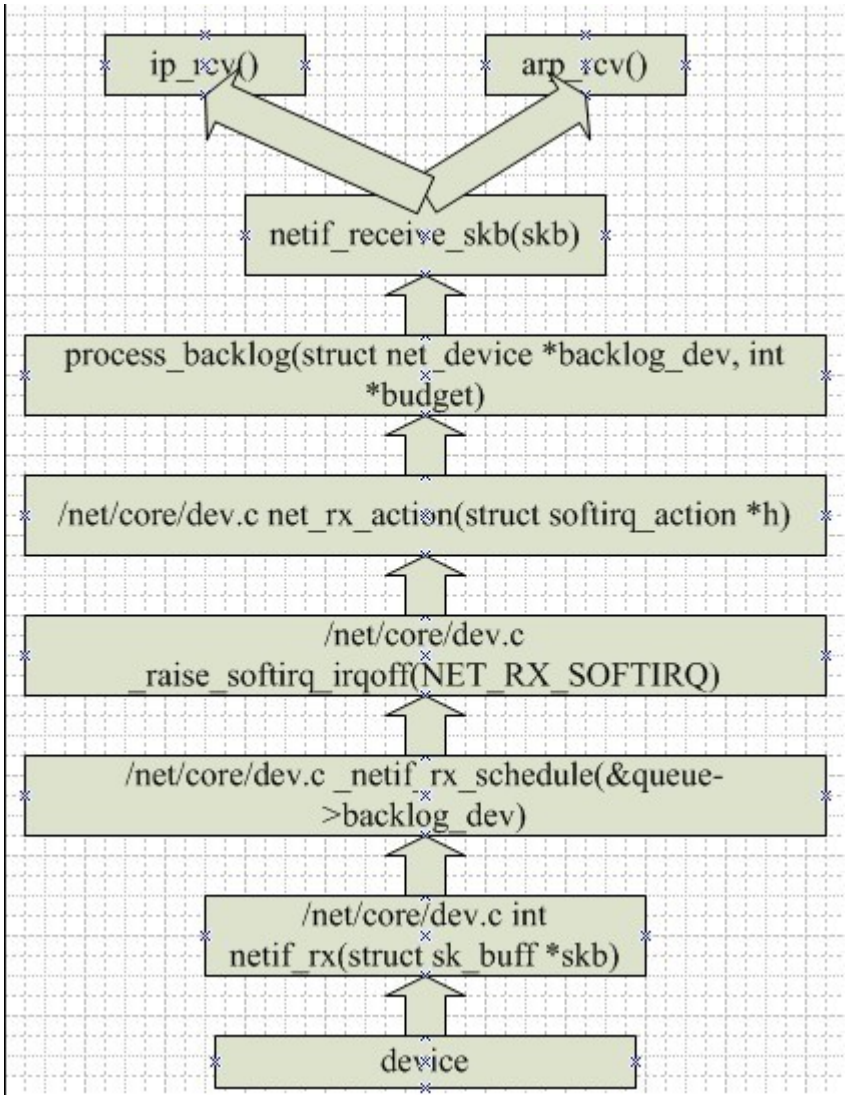
```
/*
 * Received a packet and pass to upper layer
 */
```

```
4. static void
5. dm9000_rx(struct net_device *dev)
6. {
7.     board_info_t *db = netdev_priv(dev);
8.     struct dm9000_rxhdr rxhdr;
9.     struct sk_buff *skb;
10.    u8 rxbyte, *rdptr;
11.    bool GoodPacket;
12.    int RxLen;
13.
14.    /* Check packet ready or not */
15.    do {
16.        /*MRCMDX 为内存数据预取读命令，并且没有地址增加，读数据的第一个字节，直到读到 0x01(数据有效)
    为止。*/
17.        ior(db, DM9000_MRCMDX); /* Dummy read */
18.        /*获得数据*/
19.        rxbyte = readb(db->io_data);
20.        /*因为只能为 0x00 或 0x01,所以如果大于 0x01，则返回*/
21.        if (rxbyte > DM9000_PKT_RDY) {
22.            dev_warn(db->dev, "status check fail: %d/n", rxbyte);
23.            /*停止设备*/
24.            iow(db, DM9000_RCR, 0x00);
25.            /*停止中断请求*/
26.            iow(db, DM9000_ISR, IMR_PAR);
27.            return;
28.        }
29.        /*如果为 0x00，则返回*/
30.        if (rxbyte != DM9000_PKT_RDY)
31.            return;
32.
33.        /*如果有有效数据包，设置标志标量*/
34.        GoodPacket = true;
35.        /*MRCMD 是地址增加的内存数据读命令*/
36.        writeb(DM9000_MRCMD, db->io_addr);
37.        /*读取 RX SRAM 中的数据放入 struct dm9000_rxhdr 中*/
38.        (db->inblk)(db->io_data, &rxhdr, sizeof(rxhdr));
39.        /*将一个无符号的 26 位小头数值转换成 CPU 使用的值*/
40.        RxLen = le16_to_cpu(rxhdr.RxLen);
41.
42.        if (netif_msg_rx_status(db))
43.            dev_dbg(db->dev, "RX: status %02x, length %04x/n",
44.                rxhdr.RxStatus, RxLen);
45.
46.        /*一个数据包的长度应大于 64 字节*/
47.        if (RxLen < 0x40) {
```

```
48.         GoodPacket = false;
49.         if (netif_msg_rx_err(db))
50.             dev_dbg(db->dev, "RX: Bad Packet (runt)/n");
51.     }
52.     /*一个数据包的长度应小于 1536 字节*/
53.     if (RxLen > DM9000_PKT_MAX) {
54.         dev_dbg(db->dev, "RST: RX Len:%x/n", RxLen);
55.     }
56.
57.     /* rxhdr.RxStatus is identical to RSR register. */
58.     if (rxhdr.RxStatus & (RSR_FOE | RSR_CE | RSR_AE |
59.                 RSR_PLE | RSR_RWTO |
60.                 RSR_LCS | RSR_RF)) {
61.         GoodPacket = false;
62.         if (rxhdr.RxStatus & RSR_FOE) {
63.             if (netif_msg_rx_err(db))
64.                 dev_dbg(db->dev, "fifo error/n");
65.             dev->stats.rx_fifo_errors++;
66.         }
67.         if (rxhdr.RxStatus & RSR_CE) {
68.             if (netif_msg_rx_err(db))
69.                 dev_dbg(db->dev, "crc error/n");
70.             dev->stats.rx_crc_errors++;
71.         }
72.         if (rxhdr.RxStatus & RSR_RF) {
73.             if (netif_msg_rx_err(db))
74.                 dev_dbg(db->dev, "length error/n");
75.             dev->stats.rx_length_errors++;
76.         }
77.     }
78.
79.     /* Move data from DM9000 */
80.     if (GoodPacket
81.         /*分配一个套接字缓冲区*/
82.         && ((skb = dev_alloc_skb(RxLen + 4)) != NULL)) {
83.         skb_reserve(skb, 2);
84.         rdptr = (u8 *) skb_put(skb, RxLen - 4);
85.
86.         /**/
87.         (db->inblk)(db->io_data, rdptr, RxLen);
88.         dev->stats.rx_bytes += RxLen;
89.
90.         /*以太网支持代码导出了辅助函数 eth_type_trans，用来查找填入 protocol 中的正确值*/
91.         skb->protocol = eth_type_trans(skb, dev);
92.         /*将套接字缓冲区发向上层*/
```

```
93.             netif_rx(skb);
94.             /*增加发送包的引用计数*/
95.             dev->stats.rx_packets++;
96.
97.         } else {
98.             /*如果该数据包是坏的，则清理该数据包*/
99.             (db->dumpblk)(db->io_data, RxLen);
100.        }
101.    } while (rxbyte == DM9000_PKT_RDY);
102.}
```

如下图链路层包的传输过程：



在 netif_rx 中会调用 napi_schedule，然后该函数又会去调用 __napi_schedule()。在函数__napi_schedule()中会去调用设备的 poll 函数，它是设备自己注册的。在设备的 poll 函数中，会去调用 netif_receive_skb 函数，在该函数中有下面一条语句 pt_prev->func，此处的 func 为一个函数指针，在之前的注册中设置为 ip_rcv。因此，就完成了从链路层上传到网络层的这一个过程了。