

RTL8139网卡驱动程序分析

独孤求真

www.osplay.org

Email: Addylee2004@163.com

OSP1ay 原创文章 转载请注明出处。

© 2007-7-23

根据sinister的建议，在接收部分加入了对NAPI和非NAPI方式的分析。在此对sinister大虾表示感谢。

Contents

1 预备知识	3
2 驱动的初始化	4
3 中断处理	18
4 软中断请求	20
4.1 NAPI方式	20
4.2 非NAPI方式	24
5 网卡接收操作	26
6 网卡发送操作	31

Abstract

对多数驱动程序开发的学习者来说，总是感觉很难入门，不能从整体上把握驱动程序是如何驱动硬件设备工作的。本文以Linux内核中8139网卡驱动为例，对驱动程序的工作过程进行详细的分析，为初学者拨开迷雾，走出雾里看花的迷茫。本文虽然以Linux驱动为例，但是技术总是相通的，为了给Windows驱动初学者同样的启发，我有意的借用了许多Windows驱动中的名词，同时顺便略述了Windows驱动中的一些容易让初学者感到迷惑的概念。

本人水平有限，纰漏之处在所难免，希望读者海涵，并不吝赐教。

1 预备知识

Rt18139是一个PCI网卡，老式的设备地址是固定的，对设备的扩充通常通过跳线等方式来更改地址以避免地址冲突，例如通过跳线来设置IED主盘，从盘。PCI总线设备可以通过软件编程灵活的设置各个设备的地址。在操作系统启动的时候，系统根据PCI总线协议规范对主板上的PCI进行扫描，同时为发现的设备配置相关资源，包括中断请求号，地址空间等。每一个PCI设备上有一个配置空间，配置空间中包含了设备的基本信息，例如设备类别ID，厂商ID，设备板载存储空间等信息，操作系统在扫描所有的PCI设备后，可以根据这些信息统一分配地址资源以避免地址冲突。系统通过在PCI设备中的基地址寄存器中写入一个分配到的基地址，之后CPU在指令执行的时候给出一个地址，这个地址首先送到Host-PCI桥，就是我们通常所说的北桥，北桥判断出这个地址是落在内存或是PCI等设备的地址空间上¹，如果落在PCI空间地址中，则北桥通过PCI总线仲裁申请，把地址送到PCI总线上，总线上的每一个PCI设备会根据自己的存储空间大小以及基地址寄存器中的值来比较，如果被寻址的地址落在自己的地址空间范围内，则该设备会作为PCI从设备响应完成数据传输。这就是说，系统上的设备都有自己的地址空间，以总线带宽为32位的系统为例，可以容纳的地址空间为4G，CPU在这4G的地址空间一部分划分到内存空间，还有以部分很可能划分给PCI等外部设备，这通常取决于硬件系统设计者。PCI设备灵活的配置方式也不可避免的带来了复杂性。PCI协议对设备的枚举，检测，配置的过程是复杂的，通常操作系统提供了PCI总线协议驱动程序，并在启动的时候完成了这一复杂的过程，这样大大减小了PCI设备驱动程序开发者的工作量。这就好像平时大家做网络程序开发的时候都没必要自己实现TCP/IP协议一样。用Windows的术语来说，对PCI设备的枚举由总线驱动程序完成，而具体的对PCI设备的控制是功能驱动程序的工作。本文要描述的是Rt18139网卡功能驱动程序。对PCI协议有一定了解是必要的。关于PCI总线驱动程序的知识可以

¹新的集成内存控制器的CPU自己能判断对内存的寻址。

阅读《Linux内核情景分析》。

2 驱动的初始化

驱动程序的入口函数`rtl8139_init_module`调用了`pci_register_driver` (`&rtl8139_pci_driver`): 其中参数`rtl8139_pci_driver`结构如下:

```
1 // 该结构相当于 Windows 中的功能驱动程序对象
2 static struct pci_driver rtl8139_pci_driver = {
3     .name          = DRV_NAME,
4     .id_table      = rtl8139_pci_tbl,
5     .probe         = rtl8139_init_one,
6     .remove        = __devexit_p(rtl8139_remove_one),
7 #ifdef CONFIG_PM
8     .suspend       = rtl8139_suspend,
9     .resume        = rtl8139_resume,
10 #endif /* CONFIG_PM */
11 };
12
13 // 最重要的参数。 rtl8139_pci_tbl
14 static struct pci_device_id rtl8139_pci_tbl[] = {
15     {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0,
16      RTL8139 },
17     {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0,
18      RTL8139 },
19     {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0,
20      RTL8139 },
21     .....
22 };
```

rtl8139_pci_tbl中是由VerdonID, DeviceID等组成, 表示该设备驱动程序可以控制的设备, 每一个PCI设备在配置空间中固化了自己的基本信息, 前面说过内核启动的时候PCI总线驱动程序会扫描PCI总线上的设备, 并且把这些信息收集起来, 并且每一个设备的信息由一个专门的结构保存起来, 保存在一个pci_dev结构中。pci_register_driver会根据rtl8139_pci_tbl中的信息和内核中扫描到的对比, 如果有匹配的话, 就把功能驱动程序和目标设备对应起来了。在Windows系统中维护设备信息的那个结构被称为物理设备对象, 该对象由总线驱动程序创建管理。之后pci_register_driver进行必要的初始化后, 调用参数指定的rtl8139_init_one。

```
1 static int __devinit rtl8139_init_one (struct pci_dev
    *pdev, const struct pci_device_id *ent)
2 {
3     struct net_device *dev = NULL;
4     struct rtl8139_private *tp;
5     int i, addr_len, option;
6     void __iomem *ioaddr;
7     static int board_idx = -1;
8     u8 pci_rev;
9
10    .....
11    i = rtl8139_init_board (pdev, &dev);
12    .....
13 }
```

rtl8139_init_one的参数就是PCI总线驱动程序在设备枚举过程中创建的。首先调用rtl8139_init_board。

```
1 static int __devinit rtl8139_init_board (struct
    pci_dev *pdev, struct net_device **dev_out)
```

```

2 {
3     void __iomem *ioaddr;
4     struct net_device *dev;
5     struct rtl8139_private *tp;
6     u8 tmp8;
7     int rc, disable_dev_on_err = 0;
8     unsigned int i;
9     unsigned long pio_start, pio_end, pio_flags, pio_len
        ;
10    unsigned long mmio_start, mmio_end, mmio_flags,
        mmio_len;
11    u32 version;
12
13    assert (pdev != NULL);
14
15    *dev_out = NULL;
16
17    /* dev and priv zeroed in alloc_etherdev */
18    /* 每一个网络设备驱动程序为了设备管理方便，统一接口等目的，
        需要创建自己的一个设备对象来维护设备信息，在 Windows 系
        统中，该对象称为功能设备对象。 */
19    /* 每一个设备对象是标准的结构，但是不同的驱动程序可能都要维
        护不同的私有信息，所以在分配 net_dev 结构的同时可以多分
        配出 rtl8139_private 结构来。比如应用程序可能会经常查
        询网卡地址，虽然驱动程序可以通过访问网卡上的存储空间来获
        取网卡地址，但是驱动程序可不希望每次都通过慢速的 IO 访问
        来获取这些信息，通常驱动程序会为这些信息维护内存中的数据
        结构中，这些信息都可以放在 tp 中。 */
20    dev = alloc_etherdev (sizeof (*tp));
21    if (dev == NULL) {
22        dev_err(&pdev->dev, "Unable to alloc new net
            device\n");

```

```

23     return -ENOMEM;
24 }
25 SET_MODULE_OWNER(dev);
26 SET_NETDEV_DEV(dev, &pdev->dev);
27
28 tp = netdev_priv(dev);
29 tp->pci_dev = pdev;
30
31 /* enable device (incl. PCI PM wakeup and hotplug
    setup) */
32 /* 启用设备的 memory/Io 译码, 如果设备处于休眠状态, 则唤醒
    设备。在启用设备之前, 虽然设备的基址寄存器中设置了值, 但
    是当总线有主设备对该地址进行寻址的时候, 设备是不会响应
    的。具体启用操作由总线驱动程序封装, 这里通过调用 pci 总线
    驱动程序提供的函数来完成该任务, 在 windows 中某些写操作
    由功能驱动向下层的总线驱动发送 IRP 完成同样的任务。 */
33 rc = pci_enable_device (pdev);
34 if (rc)
35     goto err_out;
36
37 /* 大多数设备上有自己的板载存储空间, 其中包括 memory 空间
    和 IO 空间, 在 X86 这样的 IO 与 Memory 分立编址的系统
    上, 他们的区别就在于独立的 IO 访问指令, 独立的地址译码,
    在多数统一编址的系统上, memeor 空间和 IO 空间没有本质区
    别。CPU 给出的指令中的地址落在哪里, 设备在电路级别会访问
    到对应地址的板载存储空间。PCI 总线驱动程序已经为设备分配
    地址空间, 并配置了基址寄存器, 通常 BIOS 已经为这些设备配
    置了互不冲突的地址, 系统的 PCI 总线驱动程序可以直接扫
    描 PCI 设备, 并从基址寄存器中读出各个设备的基地址, 也可以
    推倒从新统一分配。这里功能驱动程序需要知道自己如何访问到
    目标设备, 所以从总线物理设备对象中读取基址信息。 */
38 pio_start = pci_resource_start (pdev, 0);
39 pio_end = pci_resource_end (pdev, 0);
40 pio_flags = pci_resource_flags (pdev, 0);
41 pio_len = pci_resource_len (pdev, 0);

```

```

42
43 mmio_start = pci_resource_start (pdev, 1);
44 mmio_end = pci_resource_end (pdev, 1);
45 mmio_flags = pci_resource_flags (pdev, 1);
46 mmio_len = pci_resource_len (pdev, 1);
47
48 /*
49  set this immediately, we need to know before
50  we talk to the chip directly
51  */
52 DPRINTF("PIO region size == 0x%02X\n", pio_len);
53 DPRINTF("MMIO region size == 0x%02lX\n", mmio_len);
54
55 /* PCI 设备板载存储空间可以通过 IO 或者 Memory 方式来访
    问, PCI 设备上分别有 IO 和 Memory 基地址寄存器,
    在 X86 上的有独立的 IO 指令, 对于某些统一编址的体系结构
    则只有通过 Memory 方式来访问, CPU 执行 IO/Memory 指
    令时, 其地址送到总线上, PCI 设备会根据 IO/Memory 基址
    寄存器判断出目标设备是不是自己。无论是 IO 方式还
    是 Memory 方式, 访问到通常是板载存储空间上的同一个地方。
    由被寻址的从设备内部处理。例如: 假设 8139 网
    卡 IO / Memory 基址寄存器的值分别是 X / Y, 则使
    用 IO 指令 ( IN / OUT ) 访问 X, 以及使用 Memory 指令
    ( MOV ) 访问 Y, 结果是一样的。这样保证了设备在统一编址
    和独立编址的体系结构下的兼容性, 为什么通过 MOV 访问的地址
    没有访问到内存上去呢? 通常 CPU 给出一条访存指令, 地址被
    发到北桥, 北桥会根据地址空间的划分情况判别出该地址是落在
    内存空间上还是其它总线上的设备空间上。如果是内存的话, 则
    向内存控制器发起访问操作, 如果是 PCI 空间的话, 则同样的
    向 PCI 总线总裁提出申请。 */
56
57 #ifdef USE_IO_OPS
58  /* make sure PCI base addr 0 is PIO */
59  if (!(pio_flags & IORESOURCE_IO)) {
60      dev_err(&pdev->dev, "region #0 not a PIO resource,
          aborting\n");

```

```

61     rc = -ENODEV;
62     goto err_out;
63 }
64 /* check for weird/broken PCI region reporting */
65 if (pio_len < RTL_MIN_IO_SIZE) {
66     dev_err(&pdev->dev, "Invalid PCI I/O region size(s
        ), aborting\n");
67     rc = -ENODEV;
68     goto err_out;
69 }
70 #else
71 /* make sure PCI base addr 1 is MMIO */
72 if (!(mmio_flags & IORESOURCE_MEM)) {
73     dev_err(&pdev->dev, "region #1 not an MMIO
        resource, aborting\n");
74     rc = -ENODEV;
75     goto err_out;
76 }
77 if (mmio_len < RTL_MIN_IO_SIZE) {
78     dev_err(&pdev->dev, "Invalid PCI mem region size(s
        ), aborting\n");
79     rc = -ENODEV;
80     goto err_out;
81 }
82 #endif
83
84 /* PCI 设备的地址是由软件配置的，那么必然需要一种机制来防止
    地址冲突，系统维护
    了 io resource 和 memory resource ，分别登记了已经分
    配出去的地址空间，当为新设备分配地址的时候，必须根据登记

```

的信息找到空闲的地址空间，这里就是在系统中登记，表示改设备要使用这个地址空间了，具体的地址在总线设备对象 pdev 结构中指定。注意把地址写入到设备的基址寄存器早就已经由 bios 或者内核完成了。这里只是进行登记。 */

```

85 rc = pci_request_regions (pdev, DRV_NAME);
86 if (rc)
87     goto err_out;
88 disable_dev_on_err = 1;
89
90 /* enable PCI bus-mastering */
91 pci_set_master (pdev);
92
93 #ifdef USE_IO_OPS
94 ioaddr = ioport_map(pio_start, pio_len);
95 if (!ioaddr) {
96     dev_err(&pdev->dev, "cannot map PIO, aborting\n");
97     rc = -EIO;
98     goto err_out;
99 }
100 dev->base_addr = pio_start;
101 tp->mmio_addr = ioaddr;
102 tp->regs_len = pio_len;
103 #else
104 /* ioremap MMIO region */
105 /* 映射设备地址，这里需要搞清楚 3 种地址：
106 1. 虚拟地址：经过页表页目录映射，访问的时候通过 CPU 的 MMU 转换得到物理地址。
107 2. 物理地址：在实模式下使用的或保护模式下经过 CPU 的 MMU 转换后的地址。
108 3. 总线地址：经过各种总线桥接器转换后在出现在总线的地址线上的地址。 */
109

```

```

110  /* 通常程序指令中访问的是虚拟地址， CPU 在指令执行的时候通
      过 MMU 把虚拟地址转换成物理地址，这个地址被送到北桥，北桥
      根据自己地址空间判断出这个地址是在内存上还是在外部总线上的
      设备上，如果在外部设备上，北桥可能要根据总线地址空间的
      分配情况将这个物理地址变换成总线地址，再发送到总线上面去。
      总线上的 Host - PCI 以及 PCI - PCI 都可以做这样的变换，大多
      数情况下总线地址和物理地址是一样的，这取决于系统设计。需
      要注意的是写入 PCI 设备基址寄存器中的值必须匹配到出现在
      总线上的地址，从而响应寻址操作。而 CPU 则要记住物理地址，
      并根据物理地址映射虚拟地址。ioaddr 是映射后的虚拟地址，以
      后程序中将通过 ioaddr 访问设备。 */
111
112  ioaddr = pci_iomap(pdev, 1, 0);
113  if (ioaddr == NULL) {
114      dev_err(&pdev->dev, "cannot remap MMIO, aborting\n
          ");
115      rc = -EIO;
116      goto err_out;
117  }
118  dev->base_addr = (long) ioaddr;
119  tp->mmio_addr = ioaddr;
120  tp->regs_len = mmio_len;
121 #endif /* USE_IO_OPS */
122
123  .....
124
125  // reset 就是向命令寄存器写个 reset 命令。
126  rt18139_chip_reset (ioaddr);
127
128  *dev_out = dev;
129  return 0;
130
131 err_out:

```

```
132  __rtl8139_cleanup_dev (dev);
133  if (disable_dev_on_err)
134      pci_disable_device (pdev);
135  return rc;
136 }
```

回到rtl8139_init_one:

```
1 static int __devinit rtl8139_init_one (struct pci_dev
    *pdev, const struct pci_device_id *ent)
2 {
3     .....
4
5     i = rtl8139_init_board (pdev, &dev);
6
7     .....
8
9     tp = netdev_priv(dev);
10    ioaddr = tp->mmio_addr;
11
12    // 从 ioaddr 中读出 Mac 地址
13    addr_len = read_eeprom (ioaddr, 0, 8) == 0x8129 ? 8
        : 6;
14    for (i = 0; i < 3; i++)
15        ((u16 *) (dev->dev_addr))[i] = le16_to_cpu (
            read_eeprom (ioaddr, i + 7, addr_len));
16
17    memcpy(dev->perm_addr, dev->dev_addr, dev->addr_len)
        ;
18
```

```
19  /* The Rt18139-specific entries in the device
    structure. */
20  /* 设置功能驱动程序对象的函数指针集 */
21  dev->open = rt18139_open;
22  dev->hard_start_xmit = rt18139_start_xmit;
23  dev->poll = rt18139_poll;
24  dev->weight = 64;
25  dev->stop = rt18139_close;
26  dev->get_stats = rt18139_get_stats;
27  dev->set_multicast_list = rt18139_set_rx_mode;
28  dev->do_ioctl = netdev_ioctl;
29  dev->ethtool_ops = &rt18139_ethtool_ops;
30  dev->tx_timeout = rt18139_tx_timeout;
31  dev->watchdog_timeo = TX_TIMEOUT;
32  #ifdef CONFIG_NET_POLL_CONTROLLER
33  dev->poll_controller = rt18139_poll_controller;
34  #endif
35
36  .....
37
38  /* Put the chip into low-power mode. */
39  /* 'R' would leave the clock running. */
40  if (rtl_chip_info[tp->chipset].flags & HasH1tC1k)
41      RTL_W8 (H1tC1k, 'H');
42
43  return 0;
44
45 err_out:
46  __rt18139_cleanup_dev (dev);
```

```

47  pci_disable_device (pdev);
48  return i;
49  }

```

8139网卡的有一个接收缓冲寄存器，用于存放接收缓存的首地址，网卡一边把网线上的发出的数据放到内部FIFO，一边从FIFO中把数据通过DMA传送到由接收寄存器指定的内存地址中，接收到的数据依次排放，当长度超过默认的缓冲区长度时，会回过头来放到开始的地方，所以接收缓冲区被称为环形缓冲区。发送方面：8139有四个发送地址寄存器，CPU将要发送的数据在内存中的地址写入这四个寄存器中的任何一个，网卡就会通过DMA操作把数据发送出去。当发送或者接收完成后，网卡会发出中断，中断处理程序通过读取网卡的中断状态寄存器来识别出是发送完成发出的中断，接收到数据包的中断，还是错误中断。

当运行`ifconfig ethx up`的时候，`rt18139_open`得到调用。该函数的任务就是分配，初始化接收，发送缓冲区，分配中断号等。

```

1  static int rt18139_open (struct net_device *dev)
2  {
3      struct rt18139_private *tp = netdev_priv(dev);
4      int retval;
5      void __iomem *ioaddr = tp->mmio_addr;
6
7      // 为网卡申请中断，当中断到来时 rt18139_interrupt 会被调用。
8      retval = request_irq (dev->irq, rt18139_interrupt,
9                          IRQF_SHARED, dev->name, dev);
9      if (retval)
10         return retval;
11     // 分配接收，发送缓冲区，DMA 没有 CPU 的 MMU 单元，因此只能使用物理地址上连续的内存空间。

```

```

12  tp->tx_bufs = pci_alloc_consistent(tp->pci_dev ,
    TX_BUF_TOT_LEN , &tp->tx_bufs_dma);
13  tp->rx_ring = pci_alloc_consistent(tp->pci_dev ,
    RX_BUF_TOT_LEN , &tp->rx_ring_dma);
14  if (tp->tx_bufs == NULL || tp->rx_ring == NULL)
15  {
16      free_irq(dev->irq , dev);
17
18      if (tp->tx_bufs)
19          pci_free_consistent(tp->pci_dev , TX_BUF_TOT_LEN ,
    tp->tx_bufs , tp->tx_bufs_dma);
20      if (tp->rx_ring)
21          pci_free_consistent(tp->pci_dev , RX_BUF_TOT_LEN ,
    tp->rx_ring , tp->rx_ring_dma);
22      return -ENOMEM;
23  }
24
25  tp->mii.full_duplex = tp->mii.force_media;
26  tp->tx_flag = (TX_FIFO_THRESH << 11) & 0x003f0000;
27
28  /* 初始化接送发送缓冲区，由于有四个发送地址寄存器，因此把发
    送缓冲区分成 4 组，以后发送请求到来的时候，将待发送内容拷
    贝到第一组，再将第一组的地址写入寄存器，之后依次轮流使用
    第二，三，四，一组。... */
29  rt18139_init_ring (dev);
30  /* 对网卡硬件进行相关的初始化。 */
31  rt18139_hw_start (dev);
32
33  return 0;
34  }

```

`rtl8139_hw_start`主要是对网卡芯片进行初始化，主要就是根据网卡的硬件手册，Programming guide向一些寄存器写入某些值。接下来我们将看到大量类似的操作。

```

1 static void rtl8139_hw_start (struct net_device *dev)
2 {
3     struct rtl8139_private *tp = netdev_priv(dev);
4     /* ioaddr 就是访问设备上的存储空间的基址。 */
5     void __iomem *ioaddr = tp->mmio_addr;
6
7     .....
8     /* 向网卡命令寄存器写入一个 RESET 命令，这样网卡的各个寄存
          器恢复到默认状态。 */
9     rtl8139_chip_reset (ioaddr);
10
11     /* unlock Config[01234] and BMCR register writes */
12     RTL_W8_F (Cfg9346, Cfg9346_Unlock);
13     /* MAC0 被定义成 0，在 8139 网卡 PCI 空间基址偏移为的个
          字节是用于存放网卡 MAC 地址的，现在把之前从 EEPROM 中
          读出来的 MAC 地址写入这个地址，将来网卡在收包的时候，就会
          根据这个寄存器中的值来确定自己的 MAC 地址。现在大家该明白
          为什么我们平时能改 MAC 地址了吧。 */
14     /* Restore our idea of the MAC address. */
15     RTL_W32_F (MAC0 + 0, cpu_to_le32 (*(u32 *) (dev->
          dev_addr + 0)));
16     RTL_W32_F (MAC0 + 4, cpu_to_le32 (*(u32 *) (dev->
          dev_addr + 4)));
17
18     /* Must enable Tx/Rx before setting transfer
          thresholds! */
19     /* 向命令写入命令，允许发送和接送。 */
20     RTL_W8 (ChipCmd, CmdRxEnb | CmdTxEnb);

```

```

21
22  /* 向接收配置寄存器写入配置，以后该网卡只接收广播帧和目
      的 MAC 地址是自己的帧。设为混杂模式的时候，则是向这个寄存
      器写入。 AcceptAllPhys */
23  tp->rx_config = rtl8139_rx_config | AcceptBroadcast
      | AcceptMyPhys;
24  RTL_W32 (RxConfig, tp->rx_config);
25  RTL_W32 (TxConfig, rtl8139_tx_config);
26
27  .....
28  /* 向中断屏蔽寄存器写入中断允许位。默认允许接送，发送，错误
      等等中断。如果屏蔽了接送中断，那么当网卡接收到帧的时候就
      不会发出中断了，NAPI 就是通过屏蔽这里的接收中断，而通
      过轮询接收状态寄存器来查看是不是有帧收到了来减少中断次
      数，提高效率的。由于网卡接收小包的速度快，如果按常规处理
      流程，中断 -> CPU 保存一堆寄存器然后中断处
      理 -> CPU 恢复一堆寄存器 -> 调度决策-> ..... 在小包
      高速发送的环境下，尤其是在测试的时候，很可能在刚一恢复线
      程上下文，中断又来了。又得重新保存上下文进入中断处理，关
      闭中断进行轮询就是要节省这几个NAPI CPU 时钟周期。对大包
      来说，一个包收的要慢一点，很可能在执行 poll 轮询的时候，
      第一次检测网卡状态寄存器发现有包了，CPU Copy 出来处理，
      之后再检测那个寄存器的时候，下一个大包的接收还没完成，于
      是开了中断，结束一次，然后恢线程上下文，然后过了很小的一
      段时间，中断又来了，所以大包省不了几个中断切换的时钟周
      期，效果不明显。poll*/
29  RTL_W16 (IntrMask, rtl8139_intr_mask);

```

以上都是向某个地址写入一些值，基地址是PCI总线驱动程序配置的，某个偏移位置的地址代表什么意思，该写入什么值是功能设备的芯片逻辑规定的。以接送配置寄存器为例，RxConfig被定义为0x44，则该寄存器地址偏移是0x44,AcceptAllPhys被定义为0x01，AcceptMyPhys被定义为0x02，就是说该寄存器的最低位为1时，网卡会进入混杂模式接收所有的帧，第一位为0,第二位为1时，只接收目的MAC地址为自己的帧。彻底的搞清楚每一个寄存器的每个BIT代表什么实在是没有必要，不同的网卡芯片都是不一样的。所有这里我将略去细节的分析，力求从主线上把握就可以了，如果确实需要，可以查阅相关芯片的Datasheet和

Programming guide。

3 中断处理

当网卡收到数据，发送数据完成，或收发出错都可能发出中断，在中断处理中根据网卡中断状态寄存器的值来判断是什么情况的中断，然后调用相应的处理函数。

```

1 static irqreturn_t rtl8139_interrupt (int irq, void *
    dev_instance)
2 {
3     struct net_device *dev = (struct net_device *)
        dev_instance;
4     struct rtl8139_private *tp = netdev_priv(dev);
5     void __iomem *ioaddr = tp->mmio_addr;
6     u16 status, ackstat;
7     int link_changed = 0; /* avoid bogus "uninit"
        warning */
8     int handled = 0;
9
10    spin_lock (&tp->lock);
11    /* 读取中断状态寄存器的值。 */
12    status = RTL_R16 (IntrStatus);
13
14    .....
15
16    /* Receive packets are processed by poll routine. If
        not running start it now. */
17    /* 如果状态寄存器的接收位置 1，则进入接收处理函数。根
        据 NAPI 机制。这里先向中断屏蔽寄存器中写
        入 rtl8139_norx_intr_mask，关闭接收中
    
```

断，`netif_rx_schedule_prep` 检查网卡是不是处于 up 状态，然后在 `dev->state` 上设置 `__LINK_STATE_RX_SCHEDULE` 标记，然后通过把接收的 `__netif_rx_schedule_poll` 函数加入软中断队列。将来软中断调度的时候，会调用 `rt18139_poll`，进行轮询。轮询完成的时候，会清除 `dev->state` 上的 `__LINK_STATE_RX_SCHEDULE` 标记。这主要是避免软中断队列中出现多余的 `poll` 请求。我们都知道中断的优先级比较高，如果直接在这里用 `__netif_rx_schedule` 把 `poll` 请求加入软中断队列中，那么很可能在软中断还没被调度的时候，又来了一次接收中断，于是又有一个 `poll` 请求被加入队列中。等软中断被调度的时候，很可能在第一次 `poll` 的时候就处理完成了所有的接收，而后来的那些中断所收到的数据也被第一个处理了。`poll*/`

```

18  if (status & RxAckBits){
19      if (netif_rx_schedule_prep(dev)) {
20          RTL_W16_F (IntrMask, rt18139_norx_intr_mask);
21          __netif_rx_schedule (dev);
22      }
23  }
24
25  /* Check uncommon events with one test. */
26  /* 如果状态寄存器的相关错误位置 1，则进入错误处理函数。 */
27  if (unlikely(status & (PCIErr | PCSTimeout |
28      RxUnderrun | RxErr)))
29      rt18139_weird_interrupt (dev, tp, ioaddr, status,
30          link_changed);
31
32  /* 如果状态寄存器的发送位置 1，则进入发送中断的处理函
33      数。 */
34  if (status & (TxOK | TxErr)) {
35      rt18139_tx_interrupt (dev, tp, ioaddr);
36      if (status & TxErr)
37          RTL_W16 (IntrStatus, TxErr);
38  }

```

```

36
37 out:
38     spin_unlock (&tp->lock);
39
40     DPRINTK ("%s: exiting interrupt, intr_status=%#4.4x
41             .\n", dev->name, RTL_R16 (IntrStatus));
42     return IRQ_RETVAL(handled);

```

这里有必要说明几种关中断的方式以免读者混淆。

- CPU指令执行要经过取指令，指令译码，执行，中断检查等过程，通过CPU标志寄存器的IF位，可以让CPU在中断检查的时候忽略可屏蔽中断信号。
- 8259A 等中断控制器上面可以设置中断屏蔽位，当外设发出中断请求时，8259A先检测该中断是否被屏蔽，如果没被屏蔽，8259A才会让CPU的intr管脚有效，从而向CPU通知中断。
- 每一个可中断的外设有一个中断屏蔽寄存器，来指明哪种情况下需要向8259A中断控制器发出中断信号。

显然，这里的NAPI关闭接收中断是最后一种情况。在它关闭中断进行轮询处理过程中，随时都可能被系统上的其它设备中断。

4 软中断请求

4.1 NAPI方式

`__netif_rx_schedule(dev)`是把`poll`函数加入软中断调度队列。 ;

```

1 void __netif_rx_schedule(struct net_device *dev)
2 {
3     unsigned long flags;

```

```
4
5  local_irq_save(flags);
6  dev_hold(dev);
7  /* 每一个 CPU 有一个软中断调度队列，这里 poll_list 只是一个双向链表结构，没别的意思，当软中断调度的时候，它会循环处理队列中的调度请求，然后利用 list_move_tail，直到队列为空。 */
8  list_add_tail(&dev->poll_list, &__get_cpu_var(
        softnet_data).poll_list);
9  if (dev->quota < 0)
10     dev->quota += dev->weight;
11  else
12     dev->quota = dev->weight;
13  /* 系统启动的时候，通过 open_softirq(NET_RX_SOFTIRQ,
        net_rx_action, NULL); 把 NET_RX_SOFTIRQ 的处理函数
        设置为 net_rx_action，触发软中断
        后，net_rx_action 在适当的时候会被调用。
14  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
15  local_irq_restore(flags);
16 }
```

再看看net_rx_action。

```
1 static void net_rx_action(struct softirq_action *h)
2 {
3     struct softnet_data *queue = &__get_cpu_var(
        softnet_data);
4     unsigned long start_time = jiffies;
5     int budget = netdev_budget;
6     void *have;
7
8     local_irq_disable();
```

```
9
10 /* 循环处理软中断队列。 */
11 while (!list_empty(&queue->poll_list)) {
12     struct net_device *dev;
13
14     /* 一次软中断轮询时间不能过长。 */
15     if (budget <= 0 || jiffies - start_time > 1)
16         goto softnet_break;
17
18     local_irq_enable();
19
20     dev = list_entry(queue->poll_list.next, struct
21                     net_device, poll_list);
22     have = netpoll_poll_lock(dev);
23
24     /* 调用 poll 函数进入轮询处理。这里
25        是 rt18139_poll 。 */
26     if (dev->quota <= 0 || dev->poll(dev, &budget)) {
27         netpoll_poll_unlock(have);
28         local_irq_disable();
29         list_move_tail(&dev->poll_list, &queue->
30                       poll_list);
31         if (dev->quota < 0)
32             dev->quota += dev->weight;
33         else
34             dev->quota = dev->weight;
35     } else {
36         netpoll_poll_unlock(have);
37         dev_put(dev);
38     }
39 }
```

```
35     local_irq_disable();
36     }
37 }
38 out:
39 #ifdef CONFIG_NET_DMA
40     /*
41      * There may not be any more sk_buffs coming right
42      * now, so push
43      * any pending DMA copies to hardware
44      */
45     if (net_dma_client) {
46         struct dma_chan *chan;
47         rcu_read_lock();
48         list_for_each_entry_rcu(chan, &net_dma_client
49             ->channels, client_node)
50             dma_async_memcpy_issue_pending(chan);
51         rcu_read_unlock();
52     }
53 #endif
54     local_irq_enable();
55     return;
56
57 softnet_break:
58     __get_cpu_var(netdev_rx_stat).time_squeeze++;
59     __raise_softirq_irqoff(NET_RX_SOFTIRQ);
60     goto out;
61 }
```

4.2 非NAPI方式

在2.6的内核中，许多驱动默认就直接使用NAPI方式了，在没使用NAPI方式的情况下，网卡驱动在接收中断的时候，调用`netif_rx`而不是`netif_rx_action`。

```
1 int netif_rx(struct sk_buff *skb)
2 {
3     struct softnet_data *queue;
4     unsigned long flags;
5
6     /* if netpoll wants it, pretend we never saw it */
7     /* 注意这个 netpoll 和我们说的轮询那个 poll 无关，在内核调
      试过程中，用 printk 输出东西的时候，有时还来不急显示，
      就 panic 了，可以通过配置 netpoll 把 printk 的数据发
      送到另外一台机器，从而可以在 panic 看的时候看到正确的输
      出。 */
8     if (netpoll_rx(skb))
9         return NET_RX_DROP;
10
11     if (!skb->tstamp.off_sec)
12         net_timestamp(skb);
13
14     /*
15      * The code is rearranged so that the path is the
16      * most
17      * short when CPU is congested, but is still
18      * operating.
19      */
20     local_irq_save(flags);
21     queue = &__get_cpu_var(softnet_data);
```

```
21  __get_cpu_var(netdev_rx_stat).total++;
22  if (queue->input_pkt_queue.qlen <=
    netdev_max_backlog) {
23      if (queue->input_pkt_queue.qlen) {
24  enqueue:
25      dev_hold(skb->dev);
26      __skb_queue_tail(&queue->input_pkt_queue, skb);
27      local_irq_restore(flags);
28      return NET_RX_SUCCESS;
29  }
30      /* 注意这个 backlog_dev 是不是接收到数据包的那个 net_dev , 每一个 NET_RX_SOFTIRQ 接收软中断对象中有一个默认的 net_dev 结构 backlog_dev , 这个是系统在 net_dev_init 初始化网络系统的时候, 把 backlog_dev 的 poll 函数设置为 process_backlog , 这里和 NAPI 的区别就在于在加入队列的时候, 使用的是网卡的 NAPI net_dev , 而这里使用的是系统默认的 backlog_dev 。所以当 netif_rx_schedule 之后, 软中断调度的是 process_backlog 。*/
31      netif_rx_schedule(&queue->backlog_dev);
32      goto enqueue;
33  }
34
35      __get_cpu_var(netdev_rx_stat).dropped++;
36      local_irq_restore(flags);
37
38      kfree_skb(skb);
39      return NET_RX_DROP;
40  }
```

5 网卡接收操作

当RJ-45那个接口有数据从网线上“流入”的时候，网卡把它放到内部FIFO中，同时进行DMA传输到接收环形缓冲区。之后网卡发出中断。中断后进入接收处理函数。它先关闭接收中断后，将轮询函数挂入软中断调度队列。之后在软中断处理过程中将调用`rt18139_poll`。

```

1 static int rt18139_poll(struct net_device *dev, int *
    budget)
2 {
3     .....
4
5     spin_lock(&tp->rx_lock);
6     if (likely(RTL_R16(IntrStatus) & RxAckBits)) {
7         int work_done;
8         /* 在 rt18139_rx 中将接送到的数据拷贝出来并传递给上层协
            议驱动。*/
9         work_done = rt18139_rx(dev, tp, orig_budget);
10        if (likely(work_done > 0)) {
11            *budget -= work_done;
12            dev->quota -= work_done;
13            done = (work_done < orig_budget);
14        }
15    }
16
17    if (done) {
18        /* 轮询结束开启接收中断。 */
19        local_irq_save(flags);
20        RTL_W16_F(IntrMask, rt18139_intr_mask);
21        __netif_rx_complete(dev);
22        local_irq_restore(flags);

```

```

23 }
24 spin_unlock(&tp->rx_lock);
25 return !done;
26 }

```

rtl8139_rx把数据从网卡接收缓存中拷贝出来。数据在环形缓冲区的存放格式如下:

| 长度 | 状态位 | 内容 | 长度 | 状态位 | 内容 | ...

```

1 static int rtl8139_rx(struct net_device *dev, struct
    rtl8139_private *tp, int budget)
2 {
3     void __iomem *ioaddr = tp->mmio_addr;
4     int received = 0;
5     unsigned char *rx_ring = tp->rx_ring;
6     /* 网卡不断的把数据放进环形接收缓冲区, CPU 读出来的时候,
    读到哪里的顺序需要自己维护, tp->cur_rx 记录上次读到哪
    里, 这里将接着从上一次的地方拷贝。 */
7     unsigned int cur_rx = tp->cur_rx;
8     unsigned int rx_size = 0;
9
10    .....
11
12    /* 轮询寄存器, 当 ChipCmd RxBufEmpty 位没被网卡设置的时
    候, 则说明环形缓冲区中有接收到的数据等待处理。 */
13    while (netif_running(dev) && received < budget && (
        RTL_R8 (ChipCmd) & RxBufEmpty) == 0) {
14        u32 ring_offset = cur_rx % RX_BUF_LEN;
15        u32 rx_status;
16        unsigned int pkt_size;
17        struct sk_buff *skb;
18

```

```

19     rmb();
20
21     /* read size+status of next frame from DMA ring
        buffer */
22     rx_status = le32_to_cpu (*(u32 *) (rx_ring +
        ring_offset));
23     rx_size = rx_status >> 16;
24     pkt_size = rx_size - 4;
25
26     .....
27
28     /* Packet copy from FIFO still in progress.
        * Theoretically, this should never happen
        * since EarlyRx is disabled.
        */
32     /* 当 EarlyRX 允许的时候, 可能会发生这种情况, 一个完整的
        数据包的一部分已经通过 DMA 传送到了内存中, 而另外一部
        分还在网卡内部 FIFO 中, 网卡的 DMA 操作还在进行
        中。 */
33     if (unlikely(rx_size == 0xffff0)) {
34         if (!tp->fifo_copy_timeout)
35             tp->fifo_copy_timeout = jiffies + 2;
36         else if (time_after(jiffies, tp->
            fifo_copy_timeout)) {
37             DPRINTK ("%s: hung FIFO. Reset.", dev->name);
38             rx_size = 0;
39             goto no_early_rx;
40         }
41         if (netif_msg_intr(tp)) {

```

```

42         printk(KERN_DEBUG "%s: fifo copy in progress."
43                , dev->name);
44     }
45     tp->xstats.early_rx++;
46     break;
47 }
48 no_early_rx:
49     tp->fifo_copy_timeout = 0;
50
51     /* If Rx err or invalid rx_size/rx_status received
52      * (which happens if we get lost in the ring),
53      * Rx process gets reset, so we abort any further
54      * Rx processing.
55      */
56     if (unlikely((rx_size > (MAX_ETH_FRAME_SIZE+4)) ||
57                 (rx_size < 8) || (!(rx_status & RxStatusOK))))
58     {
59         rtl8139_rx_err (rx_status, dev, tp, iaddr);
60         received = -1;
61         goto out;
62     }
63
64     /* Malloc up new buffer, compatible with net-2e.
65      */
66     /* Omit the four octet CRC from the length. */
67
68     /* 把数据拷贝到 SKB 中来。 */
69     skb = dev_alloc_skb (pkt_size + 2);

```

```

67     if (likely(skb)) {
68         skb->dev = dev;
69         skb_reserve (skb, 2);
70 #if RX_BUF_IDX == 3
71         wrap_copy(skb, rx_ring, ring_offset+4, pkt_size)
72         ;
73 #else
74     eth_copy_and_sum (skb, &rx_ring[ring_offset +
75         4], pkt_size, 0);
76 #endif
77     skb_put (skb, pkt_size);
78
79     /* 判断包的协议。 */
80     skb->protocol = eth_type_trans (skb, dev);
81
82     dev->last_rx = jiffies;
83     /* 更新统计信息, 这些信息就是我们用 ifconfig 命令看到
84     的。 */
85     tp->stats.rx_bytes += pkt_size;
86     tp->stats.rx_packets++;
87
88     /* 调用系统函数通知上层协议驱动数据包的到来。 */
89     netif_receive_skb (skb);
90 } else {
91     if (net_ratelimit())
92         printk (KERN_WARNING "%s: Memory squeeze,
93         dropping packet.\n", dev->name);
94     tp->stats.rx_dropped++;
95 }

```

```

92     received++;
93
94     cur_rx = (cur_rx + rx_size + 4 + 3) & ~3;
95     RTL_W16 (RxBufPtr, (u16) (cur_rx - 16));
96
97     rtl8139_isr_ack(tp);
98 }
99
100 .....
101
102 out:
103     return received;
104 }

```

6 网卡发送操作

当上层协议驱动要发送数据的时候，最终会调用到`hard_start_xmit`指定的函数。发送过程很简单，只需要把待发数据拷贝到缓冲区里面，然后在把缓冲区的地址写如发送地址寄存器就可以了。

```

1 static int rtl8139_start_xmit (struct sk_buff *skb,
2     struct net_device *dev)
3 {
4     struct rtl8139_private *tp = netdev_priv(dev);
5     void __iomem *ioaddr = tp->mmio_addr;
6     unsigned int entry;
7     unsigned int len = skb->len;
8     unsigned long flags;

```

```

 9  /* Calculate the next Tx descriptor entry. */
10  /* 从四个发送寄存器中选取一个。 */
11  entry = tp->cur_tx % NUM_TX_DESC;
12
13  /* Note: the chip doesn't have auto-pad! */
14  if (likely(len < TX_BUF_SIZE)) {
15      if (len < ETH_ZLEN)
16          memset(tp->tx_buf[entry], 0, ETH_ZLEN);
17
18      /* 把待发送的数据拷贝到发送缓冲区中。 */
19      skb_copy_and_csum_dev(skb, tp->tx_buf[entry]);
20      dev_kfree_skb(skb);
21  } else {
22      dev_kfree_skb(skb);
23      tp->stats.tx_dropped++;
24      return 0;
25  }
26
27  spin_lock_irqsave(&tp->lock, flags);
28  /* 把发送缓冲区的地址写入发送缓冲区地址寄存器，之后网卡芯片
    就会把数据发送出去，发送结束的时候会发出中断请求。 */
29  RTL_W32_F (TxStatus0 + (entry * sizeof (u32)), tp->
    tx_flag | max(len, (unsigned int)ETH_ZLEN));
30
31  dev->trans_start = jiffies;
32  .....
33  return 0;
34  }

```
