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/2011/06/18/6552910.aspx

网络设备的初始化：

通过模块的加载函数看出 DM9000A 的驱动是以平台驱动的形式注册进内核的，下边是模块的加载函数：

```
1. static int __init
2. dm9000_init(void)
3. {
4.     printk(KERN_INFO "%s Ethernet Driver, V%s/n", CARDNAME, DRV_VERSION);
5.
6.     return platform_driver_register(&dm9000_driver);
7. }
```

下边是平台驱动结构体：

```
1. static struct platform_driver dm9000_driver = {
2.     .driver = {
3.         .name    = "dm9000",
4.         .owner   = THIS_MODULE,
5.     },
6.     .probe   = dm9000_probe,
7.     .remove  = __devexit_p(dm9000_drv_re     ,
8.     .suspend = dm9000_drv_suspend,
9.     .resume  = dm9000_drv_resume,
10.};
```

下面来分析 probe 函数，用来执行分配的内核函数是 alloc_netdev，函数原型是：

```
1. struct net_device *alloc_netdev(int sizeof_pr     const char *name, void (*setup)(struct net_devi
   ce*));
```

这里的 sizeof_priv 是驱动程序私有数据区的大小；这个区成员和 net_device 结构一同分配给网络设备。实际上，他们都处于一大块内存中，但是驱动程序不需要知道这些。name 是接口的名字，其在用户空间可见；这个名字可以使用类似 printf 中%d 的格式，内核将用下一个可用的接口号替代%d，最后，setup 是一个初始化函数，用来设置 net_device 结构剩余的部分。网络子系统对 alloc_netdev，为不同种类的接口封装了许多函数。最常用的是 alloc_etherdev，它定义在 linux/etherdevice.h 中：

```
1. struct net_device *alloc_etherdev(int sizeof_priv);
```

该函数使用 eth%d 的形式指定分配给网络设备的名字。它提供了自己的初始化函数(ether_setup)，用正确的值为以太网设备设置 net_device 中的许多成员。那么在 DM9000A 中这个私有数据成员是什么呢，看下边的结构：

```
1. /* Structure/enum declaration ------------------------------ */
2. typedef struct board_info {
3. 
4.   void __iomem    *io_addr;   /* Register I/O base address */
5.   void __iomem    *io_data;   /* Data I/O address */
6.   u16      irq;       /* IRQ */
7. 
8.   u16     tx_pkt_cnt;
9.   u16     queue_pkt_len;
10.  u16     queue_start_addr;
11.  u16     dbug_cnt;
12.  u8      io_mode;        /* 0:word, 2:byte */
13.  u8      phy_addr;
14.  u8      imr_all;
15. 
16.  unsigned int    flags;
17.  unsigned int    in_suspend :1;
18.  int     debug_level;
19. 
20.  enum dm9000_type type;
21. 
22.  void (*inblk)(void __iomem *port, void *data, int length);
23.  void (*outblk)(void __iomem *port, void *data, int length);
24.  void (*dumpblk)(void __iomem *port, int length);
25. 
26.  struct device   *dev;        /* parent device */
27. 
28.  struct resource *addr_res;   /* resources found */
29.  struct resource *data_res;
30.  struct resource *addr_req;   /* resources requested */
31.  struct resource *data_req;
32.  struct resource *irq_res;
33. 
34.  struct mutex     addr_lock; /* phy and eeprom access lock */
35. 
36.  struct delayed_work phy_poll;
37.  struct net_device  *ndev;
38. 
```

```
    spinlock_t  lock;

    struct mii_if_info mii;
    u32     msg_enable;
} board_info_t;
```

这个 struct board_info 就是那个私有数据，用来保存芯片相关的一些私有信息。下面是 probe 函数的实现：

```
/*
 * Search DM9000 board, allocate space and register it
 */
static int __devinit
dm9000_probe(struct platform_device *pdev)
{
    /*获得平台数据，这个应该在 platform_device 那边指定了*/
    struct dm9000_plat_data *pdata = pdev->dev.platform_data;
    struct board_info *db;  /* Point a board information structure */
    struct net_device *ndev;
    const unsigned char *mac_src;
    int ret = 0;
    int iosize;
    int i;
    u32 id_val;

    /*分配以太网的网络设备*/
    ndev = alloc_etherdev(sizeof(struct board_info));
    if (!ndev) {
        dev_err(&pdev->dev, "could not allocate device./n");
        return -ENOMEM;
    }
    /*#define SET_NETDEV_DEV(net, pdev) ((net)->dev.parent = (pdev))*/
    SET_NETDEV_DEV(ndev, &pdev->dev);

    dev_dbg(&pdev->dev, "dm9000_probe()/n");

    /*设置 struct board_info 为 ndev 的私有数据*/
    db = netdev_priv(ndev);
    memset(db, 0, sizeof(*db));

    db->dev = &pdev->dev;
    db->ndev = ndev;

    spin_lock_init(&db->lock);
    mutex_init(&db->addr_lock);
    /*提交一个任务给一个工作队列，你需要填充一个 work_struct 结构 db->phy_poll*/
```

```
38.   INIT_DELAYED_WORK(&db->phy_poll, dm9000_poll_work);
39.   /*获取 IO 内存和中断资源*/
40.   db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
41.   db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
42.   db->irq_res  = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
43.
44.   if (db->addr_res == NULL || db->data_res == NULL ||
45.       db->irq_res == NULL) {
46.       dev_err(db->dev, "insufficient resources/n");
47.       ret = -ENOENT;
48.       goto out;
49.   }
50.   /*映射到内核，并获得 IO 内存的虚拟地址，ioremap 完成页表的建立，不同于 vmalloc，但是，它实际上不
      分配内存*/
51.   iosize = res_size(db->addr_res);
52.   db->addr_req = request_mem_region(db->addr_res->start, iosize,
53.                   pdev->name);
54.
55.   if (db->addr_req == NULL) {
56.       dev_err(db->dev, "cannot claim address reg area/n");
57.       ret = -EIO;
58.       goto out;
59.   }
60.
61.   db->io_addr = ioremap(db->addr_res->start, iosize);
62.
63.   if (db->io_addr == NULL) {
64.       dev_err(db->dev, "failed to ioremap address reg/n");
65.       ret = -EINVAL;
66.       goto out;
67.   }
68.
69.   iosize = res_size(db->data_res);
70.   db->data_req = request_mem_region(db->data_res->start, iosize,
71.                   pdev->name);
72.
73.   if (db->data_req == NULL) {
74.       dev_err(db->dev, "cannot claim data reg area/n");
75.       ret = -EIO;
76.       goto out;
77.   }
78.
79.   db->io_data = ioremap(db->data_res->start, iosize);
80.
81.   if (db->io_data == NULL) {
```

```
82.     dev_err(db->dev, "failed to ioremap data reg/n");
83.     ret = -EINVAL;
84.     goto out;
85.   }
86.
87.   /*获得网络设备的基地址*/
88.   ndev->base_addr = (unsigned long)db->io_addr;
89.   /*获得网络设备的中断号*/
90.   ndev->irq = db->irq_res->start;
91.
92.   /*设置默认的 IO 函数*/
93.   dm9000_set_io(db, iosize);
94.
95.   /*如果平台数据不为空*/
96.   if (pdata != NULL) {
97.     /* check to see if the driver wants to over-ride the
98.      * default IO width */
99.
100.      if (pdata->flags & DM9000_PLATF_8BITONLY)
101.        dm9000_set_io(db, 1);
102.
103.      if (pdata->flags & DM9000_PLATF_16BITONLY)
104.        dm9000_set_io(db, 2);
105.
106.      if (pdata->flags & DM9000_PLATF_32BITONLY)
107.        dm9000_set_io(db, 4);
108.
109.      /* check to see if there are any IO routine
110.       * over-rides */
111.
112.      if (pdata->inblk != NULL)
113.        db->inblk = pdata->inblk;
114.
115.      if (pdata->outblk != NULL)
116.        db->outblk = pdata->outblk;
117.
118.      if (pdata->dumpblk != NULL)
119.        db->dumpblk = pdata->dumpblk;
120.
121.      db->flags = pdata->flags;
122.    }
123.
124. #ifdef CONFIG_DM9000_FORCE_SIMPLE_PHY_POLL
125.    db->flags |= DM9000_PLATF_SIMPLE_PHY;
126. #endif
```

```
127.  /*dm9000 复位*/
128.  dm9000_reset(db);
129.  /*读取 Vendor ID Register,Product ID Register 中的值，与 0x90000A46 比较，如果相等，则说明是
    DM9000*/
130.  /* try multiple times, DM9000 sometimes gets the read wrong */
131.  for (i = 0; i < 8; i++) {
132.      id_val  = ior(db, DM9000_VIDL);
133.      id_val |= (u32)ior(db, DM9000_VIDH) << 8;
134.      id_val |= (u32)ior(db, DM9000_PIDL) << 16;
135.      id_val |= (u32)ior(db, DM9000_PIDH) << 24;
136.
137.      if (id_val == DM9000_ID)
138.          break;
139.      dev_err(db->dev, "read wrong id 0x%08x/n", id_val);
140.  }
141.
142.  if (id_val != DM9000_ID) {
143.      dev_err(db->dev, "wrong id: 0x%08x/n", id_val);
144.      ret = -ENODEV;
145.      goto out;
146.  }
147.
148.  /* Identify what type of DM9000 we are working on */
149.  /*读取 Chip Revision Register 中的值*/
150.  id_val = ior(db, DM9000_CHIPR);
151.  dev_dbg(db->dev, "dm9000 revision 0x%02x/n", id_val);
152.
153.  switch (id_val) {
154.  case CHIPR_DM9000A:
155.      db->type = TYPE_DM9000A;
156.      break;
157.  case CHIPR_DM9000B:
158.      db->type = TYPE_DM9000B;
159.      break;
160.  default:
161.      dev_dbg(db->dev, "ID %02x => defaulting to DM9000E/n", id_val);
162.      db->type = TYPE_DM9000E;
163.  }
164.
165.  /* from this point we assume that we have found a DM9000 */
166.
167.  /* driver system function */
168.  /*设置部分 net_device 字段*/
169.  ether_setup(ndev);
170.
```

```
171.    ndev->open       = &dm9000_open;
172.    ndev->hard_start_xmit    = &dm9000_start_xmit;
173.    ndev->tx_timeout         = &dm9000_timeout;
174.    ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
175.    ndev->stop       = &dm9000_stop;
176.    ndev->set_multicast_list = &dm9000_hash_table;
177.    /*对 ethtool 支持的相关声明可在<linux/ethtool.h>中找到。它的核心是一个 ethtool_ops 类型的结构，
    里边包含一个全部的 24 个不同的方法来支持 ethtool*/
178.    ndev->ethtool_ops     = &dm9000_ethtool_ops;
179.    ndev->do_ioctl       = &dm9000_ioctl;
180.
181.#ifdef CONFIG_NET_POLL_CONTROLLER
182.    ndev->poll_controller     = &dm9000_poll_controller;
183.#endif
184.
185.    db->msg_enable       = NETIF_MSG_LINK;
186.    db->mii.phy_id_mask  = 0x1f;
187.    db->mii.reg_num_mask = 0x1f;
188.    db->mii.force_media  = 0;
189.    db->mii.full_duplex  = 0;
190.    db->mii.dev       = ndev;
191.    db->mii.mdio_read    = dm9000_phy_read;
192.    db->mii.mdio_write   = dm9000_phy_write;
193.    /*MAC 地址的源是 eeprom*/
194.    mac_src = "eeprom";
195.
196.    /* try reading the node address from the attached EEPROM */
197.    for (i = 0; i < 6; i += 2)
198.        dm9000_read_eeprom(db, i / 2, ndev->dev_addr+i);
199.    /*如果从 eeprom 中读取的地址无效，并且私有数据不为空，从 platform_device 的私有数据中获取
    dev_addr*/
200.    if (!is_valid_ether_addr(ndev->dev_addr) && pdata != NULL) {
201.        mac_src = "platform data";
202.        memcpy(ndev->dev_addr, pdata->dev_addr, 6);
203.    }
204.    /*如果地址依然无效，从 PAR:物理地址(MAC)寄存器(Physical Address Register)中读取*/
205.    if (!is_valid_ether_addr(ndev->dev_addr)) {
206.        /* try reading from mac */
207.
208.        mac_src = "chip";
209.        for (i = 0; i < 6; i++)
210.            ndev->dev_addr[i] = ior(db, i+DM9000_PAR);
211.    }
212.    /*查看以太网网卡设备地址是否有效*/
213.    if (!is_valid_ether_addr(ndev->dev_addr))
```

```
214.        dev_warn(db->dev, "%s: Invalid ethernet MAC address. Please "
215.            "set using ifconfig/n", ndev->name);
216.    /*将 ndev 保存到 pdev->dev->driver_data 中*/
217.    platform_set_drvdata(pdev, ndev);
218.    /*一切都初始化好后，注册网络设备*/
219.    ret = register_netdev(ndev);
220.
221.    if (ret == 0)
222.        printk(KERN_INFO "%s: dm9000%c at %p,%p IRQ %d MAC: %pM (%s)/n",
223.            ndev->name, dm9000_type_to_char(db->type),
224.            db->io_addr, db->io_data, ndev->irq,
225.            ndev->dev_addr, mac_src);
226.    return 0;
227.
228.out:
229.    dev_err(db->dev, "not found (%d)./n", ret);
230.
231.    dm9000_release_board(pdev, db);
232.    free_netdev(ndev);
233.
234.    return ret;
235.}
```

下边看看挂起和唤醒函数：

挂起函数完成了设置挂起标志，并没有真正把设备移除而只是设置了移除标志，复位 PHY，停止 PHY,禁止所有中断，禁止接受引脚。

```
1.  static int
2.  dm9000_drv_suspend(struct platform_device *dev, pm_message_t state)
3.  {
4.      struct net_device *ndev = platform_get_drvdata(dev);
5.      board_info_t *db;
6.
7.      if (ndev) {
8.          db = netdev_priv(ndev);
9.          db->in_suspend = 1;
10.
11.         if (netif_running(ndev)) {
12.             netif_device_detach(ndev);
13.             dm9000_shutdown(ndev);
14.         }
15.     }
16.     return 0;
17. }
```

唤醒函数完成了复位 dm9000,初始化 dm9000,标记设备为 attached，清除挂起标志。

```
1. static int
2. dm9000_drv_resume(struct platform_device *dev)
3. {
4.     struct net_device *ndev = platform_get_drvdata(dev);
5.     board_info_t *db = netdev_priv(ndev);
6.
7.     if (ndev) {
8.
9.         if (netif_running(ndev)) {
10.            dm9000_reset(db);
11.            dm9000_init_dm9000(ndev);
12.            netif_device_attach(ndev);
13.        }
14.        db->in_suspend = 0;
15.    }
16.    return 0;
17.}
```

网络设备的打开与释放：

首先来看这个 open 函数：

```
1. static int
2. dm9000_open(struct net_device *dev)
3. {
4.     board_info_t *db = netdev_priv(dev);
5.     unsigned long irqflags = db->irq_res->flags & IRQF_TRIGGER_MASK;
6.
7.     if (netif_msg_ifup(db))
8.         dev_dbg(db->dev, "enabling %s/n", dev->name);
9.
10.    /* If there is no IRQ type specified, default to something that
11.     * may work, and tell the user that this is a problem */
12.
13.    if (irqflags == IRQF_TRIGGER_NONE)
14.        dev_warn(db->dev, "WARNING: no IRQ resource flags set./n");
15.
16.    irqflags |= IRQF_SHARED;
17.    /*注册中断处理函数*/
18.    if (request_irq(dev->irq, &dm9000_interrupt, irqflags, dev->name, dev))
19.        return -EAGAIN;
20.
21.    /* Initialize DM9000 board */
```

```
    /*复位 DM9000*/
    dm9000_reset(db);
    /*初始化 DM9000 的寄存器*/
    dm9000_init_dm9000(dev);

    /* Init driver variable */
    db->dbug_cnt = 0;
    /*检查链路载波状况*/
    mii_check_media(&db->mii, netif_msg_link(db), 1);
    /*启动发送队列*/
    netif_start_queue(dev);
    /*之前在 probe 函数中调用 INIT_DELAYED_WORK 初始化了工作队列，并关联了一个操作函数
dm9000_poll_work()，此时运行 dm9000_schedule_poll 来调用这个函数*/
    dm9000_schedule_poll(db);

    return 0;
}
```

然后是 stop 函数：

```
static int
dm9000_stop(struct net_device *ndev)
{
    board_info_t *db = netdev_priv(ndev);

    if (netif_msg_ifdown(db))
        dev_dbg(db->dev, "shutting down %s/n", ndev->name);
    /*杀死延时工作队列 phy_poll*/
    cancel_delayed_work_sync(&db->phy_poll);
    /*停止发送队列*/
    netif_stop_queue(ndev);
    /*通知内核链路失去连接*/
    netif_carrier_off(ndev);
    /* free interrupt */
    free_irq(ndev->irq, ndev);
    /*关闭 DM9000*/
    dm9000_shutdown(ndev);
    return 0;
}
```

复位 PHY，停止 PHY，禁止所有中断，禁止接收引脚。

```
static void
dm9000_shutdown(struct net_device *dev)
{
```

```
4.    board_info_t *db = netdev_priv(dev);
5.
6.    /* RESET device */
7.    dm9000_phy_write(dev, 0, MII_BMCR, BMCR_RESET); /* PHY RESET */
8.    iow(db, DM9000_GPR, 0x01);  /* Power-Down PHY */
9.    iow(db, DM9000_IMR, IMR_PAR);   /* Disable all interrupt */
10.   iow(db, DM9000_RCR, 0x00);  /* Disable RX */
11.}
```