dm9000 分析

----------参考:linux/driver/net/davicom/dm9000.c
arch/arm/mach-s5pv210.c

因 fs4412 在内核中没有现成的框架代码，故暂采用 smdk210 设备代码为其匹配的设备代码。经查看代码得知，该网卡驱动采用的是 platform 架构，故先查看其 probe 函数；

【备注：紫底黑字表示其他文件或者函数的具体实现中的代码，墨绿色底黑字表示该段语句不执行】

1.probe 分析

该部分程序主要有：
（1）申请 strutc net_device ,struct board_info
（2）从 platform_device 中获取 DM9000 的 data，address，irq 并建立映射代表其在内核中使用；
（3）根据 board_info 中的字节长度设定其相应字节长度的读写控制函数；
（4）对网卡进行模式及复位控制；
（5）读取并验证网卡的 vendor 及 device 的 id 号；
（6）读取网卡的 chip version，并根据不容的 version id 进行赋值；
（7）初始化填充 net_device 结构体；
（8）注册网卡结构体；

```
/*Search DM9000 board, allocate space and register it*/
static int dm9000_probe(struct platform_device *pdev)
{
    //定义并获取设备的 dm9000_plat_data
    arch/arm/mach-s5pv210.c 中与之对应的代码
    static struct dm9000_plat_data smdkv210_dm9000_platdata = {
        .flags          = DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM,
        .dev_addr       = { 0x00, 0x09, 0xc0, 0xff, 0xec, 0x48 },
    };

    struct dm9000_plat_data *pdata = dev_get_platdata(&pdev->dev);
    struct board_info *db;    /* Point a board information structure */
    struct net_device *ndev;//定义网络设备结构体
    const unsigned char *mac_src;
```

```
int ret = 0;
int iosize;
int i;
u32 id_val;

//如果在平台设备中没有获取到 struct dm9000_plat_data，则取设备树中尝试获取
if (!pdata) {
	pdata = dm9000_parse_dt(&pdev->dev);
	if (IS_ERR(pdata))
		return PTR_ERR(pdata);
}

/* Init network device *///申请 struct net_device+struct board_info
ndev = alloc_etherdev(sizeof(struct board_info));
if (!ndev)
	return -ENOMEM;
//将 ndev 与 platform_device 中的 dev 联系起来
SET_NETDEV_DEV(ndev, &pdev->dev);

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

//填充 struct board_info 信息
/* setup board info structure */
db = netdev_priv(ndev);

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

spin_lock_init(&db->lock);
mutex_init(&db->addr_lock);

INIT_DELAYED_WORK(&db->phy_poll, dm9000_poll_work);
//获取设备中的地址及中断资源
-----------------------------------------------------------------
arch/arm/mach-s5pv210.c 中与之对应的代码
static struct resource smdkv210_dm9000_resources[] = {
	[0] = DEFINE_RES_MEM(S5PV210_PA_SROM_BANK5, 1),
	[1] = DEFINE_RES_MEM(S5PV210_PA_SROM_BANK5 + 2, 1),
	[2] = DEFINE_RES_NAMED(IRQ_EINT(9), 1, NULL, IORESOURCE_IRQ \
				| IORESOURCE_IRQ_HIGHLEVEL),
};
-----------------------------------------------------------------
db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
db->irq_res    = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
//判断资源是否获取到，没有获取到则进入 out 标签对应的处理
```

```
if (db->addr_res == NULL || db->data_res == NULL ||
        db->irq_res == NULL) {
        dev_err(db->dev, "insufficient resources\n");
        ret = -ENOENT;
        goto out;
}
```

//获取设备资源中的第一个中断，如果没有则返回-ENXIO,由于此处的中断只有第 0 个
//中断，无第一个，所以返回-ENXIO;故 if (db->irq_wake >= 0) 该段函数无需执行

```
db->irq_wake = platform_get_irq(pdev, 1);
if (db->irq_wake >= 0) {
        dev_dbg(db->dev, "wakeup irq %d\n", db->irq_wake);

        ret = request_irq(db->irq_wake, dm9000_wol_interrupt,
                          IRQF_SHARED, dev_name(db->dev), ndev);
        if (ret) {
                dev_err(db->dev, "cannot get wakeup irq (%d)\n", ret);
        } else {

                /* test to see if irq is really wakeup capable */
                ret = irq_set_irq_wake(db->irq_wake, 1);
                if (ret) {
                        dev_err(db->dev, "irq %d cannot set wakeup (%d)\n",
                                db->irq_wake, ret);
                        ret = 0;
                } else {
                        irq_set_irq_wake(db->irq_wake, 0);
                        db->wake_supported = 1;
                }
        }
    }
```

//将获取的地址及数据资源进行字节长度测量、内存申请、进行 IO 内存映射

```
iosize = resource_size(db->addr_res);
db->addr_req = request_mem_region(db->addr_res->start, iosize,
                                  pdev->name);

if (db->addr_req == NULL) {
        dev_err(db->dev, "cannot claim address reg area\n");
        ret = -EIO;
        goto out;
}
db->io_addr = ioremap(db->addr_res->start, iosize);

if (db->io_addr == NULL) {
```

```
        dev_err(db->dev, "failed to ioremap address reg\n");
        ret = -EINVAL;
        goto out;
}

iosize = resource_size(db->data_res);
db->data_req = request_mem_region(db->data_res->start, iosize,
                                  pdev->name);

if (db->data_req == NULL) {
        dev_err(db->dev, "cannot claim data reg area\n");
        ret = -EIO;
        goto out;
}

db->io_data = ioremap(db->data_res->start, iosize);

if (db->io_data == NULL) {
        dev_err(db->dev, "failed to ioremap data reg\n");
        ret = -EINVAL;
        goto out;
}

/* fill in parameters for net-dev structure *
//将 platform_device 中的地址及中断号分别赋值给网络设备中的地址及中断号成员
ndev->base_addr = (unsigned long)db->io_addr;
ndev->irq= db->irq_res->start;

/* ensure at least we have a default set of IO routines */
//根据地址的字节长度设置 struct board_info 中数据处理的不同回调函数,一般纵使该
//函数进行了相应的设置，但是会是会由之后的 if (pdata != NULL) {}中的处理进行覆盖
dm9000_set_io(db, iosize);

/* check to see if anything is being over-ridden */
//根据地址的字节长度设置 struct board_info 中数据处理的不同回调函数
if (pdata != NULL) {
        /* check to see if the driver wants to over-ride the
          * default IO width */

        if (pdata->flags & DM9000_PLATF_8BITONLY)
                dm9000_set_io(db, 1);

        if (pdata->flags & DM9000_PLATF_16BITONLY)
                dm9000_set_io(db, 2);
```

```
        if (pdata->flags & DM9000_PLATF_32BITONLY)
            dm9000_set_io(db, 4);

        /* check to see if there are any IO routine
         * over-rides */

        if (pdata->inblk != NULL)
            db->inblk = pdata->inblk;

        if (pdata->outblk != NULL)
            db->outblk = pdata->outblk;

        if (pdata->dumpblk != NULL)
            db->dumpblk = pdata->dumpblk;

        db->flags = pdata->flags;
    }

#ifdef CONFIG_DM9000_FORCE_SIMPLE_PHY_POLL
    db->flags |= DM9000_PLATF_SIMPLE_PHY;
#endif

    /* Fixing bug on dm9000_probe, takeover dm9000_reset(db),
     * Need 'NCR_MAC_LBK' bit to indeed stable our DM9000 fifo
     * while probe stage.
     */
    //硬件操作：对相应的寄存器进行赋值
    iow(db, DM9000_NCR, NCR_MAC_LBK | NCR_RST);
```

//数据手册信息

NCR(00H)：网络控制寄存器(Network Control Register)

7：EXT_PHY：1 选择外部 PHY，0 选择内部 PHY，不受软件复位影响。

6：WAKEEN：事件唤醒使能，1 使能，0 禁止并清除事件唤醒状态，不受软件复位影响。

5：保留。

4：FCOL：1 强制冲突模式，用于用户测试。

3：FDX：全双工模式。内部 PHY 模式下只读，外部 PHY 下可读写。

2-1：LBK：回环模式(Loopback)00 通常，01MAC 内部回环，10 内部 PHY 100M 模式数字回环，11 保留。

0：RST：1 软件复位，10us 后自动清零。

```
    /* try multiple times, DM9000 sometimes gets the read wrong */
    //验证 DM9000ID
    for (i = 0; i < 8; i++) {
```

```
        id_val    = ior(db, DM9000_VIDL);
        id_val |= (u32)ior(db, DM9000_VIDH) << 8;
        id_val |= (u32)ior(db, DM9000_PIDL) << 16;
        id_val |= (u32)ior(db, DM9000_PIDH) << 24;
```

//数据手册信息

VID(28H -- 29H)：生产厂家序列号(Vendor ID)

7-0：VIDL：低半字节(28H)，只读，默认 46H。

7-0：VIDH：高半字节(29H)，只读，默认 0AH。

PID(2AH --2BH)：产品序列号(Product ID)

7-0：PIDL：低半字节(2AH)，只读，默认 00H。

7-0：PIDH：高半字节(2BH)，只读，默认 90H。

```
        if (id_val == DM9000_ID)
            break;
        dev_err(db->dev, "read wrong id 0x%08x\n", id_val);
    }
    //如果读取的 ID 不等于 DM9000 的 id 则退出并返回错误
    if (id_val != DM9000_ID) {
        dev_err(db->dev, "wrong id: 0x%08x\n", id_val);
        ret = -ENODEV;
        goto out;
    }

    /* Identify what type of DM9000 we are working on */
    //读取芯片版本号
    id_val = ior(db, DM9000_CHIPR);
```

//数据手册

CHIPR(2CH)：芯片修订版本(CHIP Revision)

7-0：PIDH：只读，默认 00H。

```
    dev_dbg(db->dev, "dm9000 revision 0x%02x\n", id_val);
    //根据不同的芯片版本号进行不同的 struct board_info 中 type 成员的赋值
    switch (id_val) {
    case CHIPR_DM9000A:
        db->type = TYPE_DM9000A;
        break;
    case CHIPR_DM9000B:
        db->type = TYPE_DM9000B;
        break;
    default:
        dev_dbg(db->dev, "ID %02x => defaulting to DM9000E\n", id_val);
        db->type = TYPE_DM9000E;
```

```
}

/* dm9000a/b are capable of hardware checksum offload */
//根据不同的设备类型可以开启不同的用户可更改特性
if (db->type == TYPE_DM9000A || db->type == TYPE_DM9000B) {
	ndev->hw_features = NETIF_F_RXCSUM | NETIF_F_IP_CSUM;
	ndev->features |= ndev->hw_features;
}

/* from this point we assume that we have found a DM9000 */

/* driver system function */
//填充 struct net_device 的成员
ether_setup(ndev);
//填充该网络设备的操作方法集合
ndev->netdev_ops  = &dm9000_netdev_ops;
ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
//支持上层 ethtool 工具【设置得到的网卡信息】
ndev->ethtool_ops = &dm9000_ethtool_ops;
//填充 struct board_info
db->msg_enable       = NETIF_MSG_LINK;
db->mii.phy_id_mask    = 0x1f;
db->mii.reg_num_mask = 0x1f;
db->mii.force_media    = 0;
db->mii.full_duplex    = 0;
db->mii.dev            = ndev;
db->mii.mdio_read      = dm9000_phy_read;
db->mii.mdio_write     = dm9000_phy_write;

mac_src = "eeprom";

/* try reading the node address from the attached EEPROM */
//读取 MAC 地址的多种方式
//（1）读取 eeprom 中存储的 MAC
for (i = 0; i < 6; i += 2)
	dm9000_read_eeprom(db, i / 2, ndev->dev_addr+i);
//(2)读取 struct board_info 存储的 MAC
if (!is_valid_ether_addr(ndev->dev_addr) && pdata != NULL) {
	mac_src = "platform data";
	memcpy(ndev->dev_addr, pdata->dev_addr, ETH_ALEN);
}
//（3）读取寄存器中的 MAC 地址----->采用该方式
if (!is_valid_ether_addr(ndev->dev_addr)) {
	/* try reading from mac */
```

```
            mac_src = "chip";
            for (i = 0; i < 6; i++)
                ndev->dev_addr[i] = ior(db, i+DM9000_PAR);
        }
```

//数据手册

PAR(10H -- 15H)：物理地址(MAC)寄存器(Physical Address Register)

7-0：PAD0 -- PAD5：物理地址字节 0 -- 字节 5(10H -- 15H)。用来保存 6 个字节的 MAC 地址。

```
        if (!is_valid_ether_addr(ndev->dev_addr)) {
            dev_warn(db->dev, "%s: Invalid ethernet MAC address. Please "
                  "set using ifconfig\n", ndev->name);
            //生成一个只读的虚拟 MAC 地址供 net_device 填充其成员
            eth_hw_addr_random(ndev);
            mac_src = "random";
        }

        //将 ndev 填充至 struct platform_device 中的 struct device 中的 struct private_data 中的
        //driver_data
        platform_set_drvdata(pdev, ndev);
        //注册网络设备
        ret = register_netdev(ndev);

        if (ret == 0)
            printk(KERN_INFO "%s: dm9000%c at %p,%p IRQ %d MAC: %pM (%s)\n",
                   ndev->name, dm9000_type_to_char(db->type),
                   db->io_addr, db->io_data, ndev->irq,
                   ndev->dev_addr, mac_src);
        return 0;

out:
        dev_err(db->dev, "not found (%d).\n", ret);

        dm9000_release_board(pdev, db);
        free_netdev(ndev);

        return ret;
}
```

2.struct net_device_ops 中的 open 分析

在 struct net_device_ops 中的 open 与 close 回调函数都在何时进行调用呢？当该网卡设

备在开启或者关闭的时候进行分别调用，即当 ifconfig eth0 up/down 时分别调用 open,close。

```
static int dm9000_open(struct net_device *dev)
{
    //获取 board_info
    board_info_t *db = netdev_priv(dev);
    //获取中断标志
    unsigned long irqflags = db->irq_res->flags & IRQF_TRIGGER_MASK;

    if (netif_msg_ifup(db))
        dev_dbg(db->dev, "enabling %s\n", dev->name);

    /* If there is no IRQ type specified, default to something that
     * may work, and tell the user that this is a problem */

    if (irqflags == IRQF_TRIGGER_NONE)
        dev_warn(db->dev, "WARNING: no IRQ resource flags set.\n");

    irqflags |= IRQF_SHARED;

    /* GPIO0 on pre-activate PHY, Reg 1F is not set by reset */
    iow(db, DM9000_GPR, 0);     /* REG_1F bit0 activate phyxcer */
```

//数据手册

GPR(1FH)：GPIO 寄存器(General Purpose Register)

7-4：保留。

3-1：GEPIO3-1：GPIO 为输出时，相关位控制对应 GPIO 端口状态，GPIO 为输入时，相关位反映对应 GPIO 端口状态。(类似于单片机对 IO 端口的控制)。

0：GEPIO0：功能同上。该位默认为输出 1 到 POWER_DEWN 内部 PHY。若希望启用 PHY，则驱动程序需要通过写“0”将 PWER_DOWN 信号清零。该位默认值可通过 EEPROM 编程得到。参考 EEPROM 相关描述。

```
    mdelay(1); /* delay needs by DM9000B */

    /* Initialize DM9000 board */
    //通过对网卡中寄存器的配置对网卡实现复位重启
    dm9000_reset(db);
```

```
static void dm9000_reset(board_info_t *db)
{
    dev_dbg(db->dev, "resetting device\n");

    /* Reset DM9000, see DM9000 Application Notes V1.22 Jun 11, 2004 page 29
     * The essential point is that we have to do a double reset, and the
     * instruction is to set LBK into MAC internal loopback mode.
     */
    iow(db, DM9000_NCR, 0x03);
```

//数据手册

NCR(00H)：网络控制寄存器(Network Control Register)

7：EXT_PHY：1 选择外部 PHY，0 选择内部 PHY，不受软件复位影响。

6：WAKEEN：事件唤醒使能，1 使能，0 禁止并清除事件唤醒状态，不受软件复位影响。

5：保留。

4：FCOL：1 强制冲突模式，用于用户测试。

3：FDX：全双工模式。内部 PHY 模式下只读，外部 PHY 下可读写。

2-1：LBK：回环模式(Loopback)00 通常，01MAC 内部回环，10 内部 PHY 100M 模式数字回环，11 保留。

0：RST：1 软件复位，10us 后自动清零。

```
    udelay(100); /* Application note says at least 20 us */
    if (ior(db, DM9000_NCR) & 1)
        dev_err(db->dev, "dm9000 did not respond to first reset\n");

    iow(db, DM9000_NCR, 0);
    iow(db, DM9000_NCR, 0x03);
    udelay(100);
    if (ior(db, DM9000_NCR) & 1)
        dev_err(db->dev, "dm9000 did not respond to second reset\n");
}
```

//网卡的初始化设置，主要是控制各个寄存器的操作

dm9000_init_dm9000(dev);

```
static void    dm9000_init_dm9000(struct net_device *dev)
{
    board_info_t *db = netdev_priv(dev);//获取  struct board_info  结构体指针
    unsigned int imr;
    unsigned int ncr;

    dm9000_dbg(db, 1, "entering %s\n", __func__);

    /* I/O mode */
    db->io_mode = ior(db, DM9000_ISR) >> 6;    /* ISR bit7:6 keeps I/O mode */
```

ISR(FEH)：终端状态寄存器(Interrupt Status Register)

7-6：IOMODE：处理器模式。00 为 16 位模式，01 为 32 位模式，10 为 8 位模式，00 保留。

5：LNKCHG：连接状态改变。

4：UDRUN：传输“Underrun”

3：ROOS：接收溢出计数器溢出。

2：ROS：接收溢出。

1：PTS：数据包传输。

0：PRS：数据包接收。

ISR 寄存器各状态写 1 清除

```
/* Checksum mode */
if (dev->hw_features & NETIF_F_RXCSUM)
        iow(db, DM9000_RCSR,
                (dev->features & NETIF_F_RXCSUM) ? RCSR_CSUM : 0);
```

RCSCSR(32H)：接收校验和控制状态寄存器(Receive Check Sum Control Status Register)

7：UDPS：UDP 校验和状态。1 表示 UDP 数据包校验失败。

6：TCPS：TCP 校验和状态。1 表示 TCP 数据包校验失败。

5：IPS：IP 校验和状态。1 表示 IP 数据包校验失败。

4：UDPP：1 表示 UDP 数据包。

3：TCPP：1 表示 TCP 数据包。

2：IPP：1 表示 IP 数据包。

1：RCSEN：接收检验和检验使能。1 使能校验和校验，将校验和状态位(bit7-2)存储到数据包的各自的报文头的第一个字节。

0：DCSE：丢弃校验和错误的数据包。1 使能丢弃校验和错误的数据包，若 IP/TCP/UDP 的校验和域错误，则丢弃该数据包。

```
iow(db, DM9000_GPCR, GPCR_GEP_CNTL);  /* Let GPIO0 output */
```

GPCR(1FH)：GPIO 控制寄存器(General Purpose Control Register)

7-4：保留。

3-0：GEP_CNTL：GPIO 控制。定义 GPIO 的输入输出方向。1 为输出，0 为输入。GPIO 0 默认为输出做 POWER_DOWN 功能。其它默认为输入。因此默认值为 0001。

```
iow(db, DM9000_GPR, 0);
```

GPR(1FH)：GPIO 寄存器(General Purpose Register)

7-4：保留。

3-1：GEPIO3-1：GPIO 为输出时，相关位控制对应 GPIO 端口状态，GPIO 为输入时，相关位反映对应 GPIO 端口状态。(类似于单片机对 IO 端口的控制)。

0：GEPIO0：功能同上。该位默认为输出 1 到 POWER_DEWN 内部 PHY。若希望启用 PHY，则驱动程序需要通过写“0”将 PWER_DOWN 信号清零。该位默认值可通过 EEPROM 编程得到。参考 EEPROM 相关描述。

```
    /* If we are dealing with DM9000B, some extra steps are required: a
     * manual phy reset, and setting init params.
     */
    if (db->type == TYPE_DM9000B) {
            dm9000_phy_write(dev, 0, MII_BMCR, BMCR_RESET);
            dm9000_phy_write(dev, 0, MII_DM_DSPCR, DSPCR_INIT_PARAM);
    }

    ncr = (db->flags & DM9000_PLATF_EXT_PHY) ? NCR_EXT_PHY : 0;

    /* if wol is needed, then always set NCR_WAKEEN otherwise we end
     * up dumping the wake events if we disable this. There is already
     * a wake-mask in DM9000_WCR */
    if (db->wake_supported)
            ncr |= NCR_WAKEEN;
```

NCR(00H)：网络控制寄存器(Network Control Register)

7：EXT_PHY：1 选择外部 PHY，0 选择内部 PHY，不受软件复位影响。

6：WAKEEN：事件唤醒使能，1 使能，0 禁止并清除事件唤醒状态，不受软件复位影响。

5：保留。

4：FCOL：1 强制冲突模式，用于用户测试。

3：FDX：全双工模式。内部 PHY 模式下只读，外部 PHY 下可读写。

2-1：LBK：回环模式(Loopback)00 通常，01MAC 内部回环，10 内部 PHY 100M 模式数字回环，11 保留。

0：RST：1 软件复位，10us 后自动清零。

```
    iow(db, DM9000_NCR, ncr);

    /* Program operating register */
    iow(db, DM9000_TCR, 0);                 /* TX Polling clear */
```

TCR(02H)：发送控制寄存器(TX Control Register)

7：保留。

6：TJDIS：Jabber 传输使能。1 使能 Jabber 传输定时器(2048 字节)，0 禁止。

注释：Jabber 是一个有 CRC 错误的长帧(大于 1518byte 而小于 6000byte)或是数据包重组错误。原因：它可能导致网络丢包。多是由于作站有硬件或软件错误。

5：EXCECM：额外冲突模式控制。0 当额外的冲突计数多于 15 则终止本次数据包，1 始终尝试发发送本次数据包。

4：PAD_DIS2：禁止为数据包指针 2 添加 PAD。

3：CRC_DIS2：禁止为数据包指针 2 添加 CRC 校验。

2：PAD_DIS2：禁止为数据包指针 1 添加 PAD。

1：CRC_DIS2：禁止为数据包指针 1 添加 CRC 校验。

0：TXREQ：TX(发送)请求。发送完成后自动清零该位。

```
iow(db, DM9000_BPTR, 0x3f);      /* Less 3Kb, 200us */
```

BPTR(08H)：背压门限寄存器(Back Pressure Threshold Register)

7-4：BPHW：背压门限最高值。当接收 SRAM 空闲空间低于该门限值，则 MAC 将产生一个拥挤状态。1=1K 字节。默认值为 3H，即 3K 字节空闲空间。不要超过 SRAM 大小。

3-0：JPT：拥挤状态时间。默认为 200us。0000 为 5us，0001 为 10us，0010 为 15us，0011 为 25us，0100 为 50us，0101 为 100us，0110 为 150us，0111 为 200us，1000 为 250 us，1001 为 300us，1010 为 350us，1011 为 400us，1100 为 450us，1101 为 500us， 111 0 为 550us，1111 为 600us。

```
iow(db, DM9000_FCR, 0xff);  /* Flow Control */
```

RTFCR(0AH)：接收/发送溢出控制寄存器(RX/TX Flow Control Register)

7：TXP0：1 发送暂停包。发送完成后自动清零，并设置 TX 暂停包时间为 0000H。

6：TXPF：1 发送暂停包。发送完成后自动清零，并设置 TX 暂停包时间为 FFFFH。

5：TXPEN：强制发送暂停包使能。按溢出门限最高值使能发送暂停包。

4：BKPA：背压模式。该模式仅在半双工模式下有效。当接收 SRAM 超过 BPHW 并且接收新数据包时，产生一个拥挤状态。

3：BKPM：背压模式。该模式仅在半双工模式下有效。当接收 SRAM 超过 BPHW 并数据包 DA 匹配时，产生一个拥挤状态。

2：RXPS：接收暂停包状态。只读清零允许。

1：RXPCS：接收暂停包当前状态。

0：FLCE：溢出控制使能。1 设置使能溢出控制模式。

```
iow(db, DM9000_SMCR, 0);                /* Special Mode */
```

SMCR(2FH)：特殊模式控制寄存器(Special Mode Control Register)

7：SM_EN：特殊模式使能。

6-3：保留。

2：FLC：强制冲突延迟。

1：FB1：强制最长“Back-off”时间。

0：FB0：强制最短“Back-off”时间。

```
/* clear TX status */
iow(db, DM9000_NSR, NSR_WAKEST | NSR_TX2END | NSR_TX1END);
```

NSR (01H)：网络状态寄存器(Network Status Register)

7：SPEED：媒介速度，在内部 PHY 模式下，0 为 100Mbps，1 为 10Mbps。当 LINKST=0 时，此位不用。

6：LINKST：连接状态，在内部 PHY 模式下，0 为连接失败，1 为已连接。

5：WAKEST：唤醒事件状态。读取或写 1 将清零该位。不受软件复位影响。

4：保留。

3：TX2END：TX(发送)数据包 2 完成标志，读取或写 1 将清零该位。数据包指针 2 传输完成。

2：TX2END：TX(发送)数据包 1 完成标志，读取或写 1 将清零该位。数据包指针 1 传输完成。

1：RXOV：RX(接收)FIFO(先进先出缓存)溢出标志。

0：保留。

```
iow(db, DM9000_ISR, ISR_CLR_STATUS); /* Clear interrupt status */
```

ISR(FEH)：终端状态寄存器(Interrupt Status Register)

7-6：IOMODE：处理器模式。00 为 16 位模式，01 为 32 位模式，10 为 8 位模式，00 保留。

5：LNKCHG：连接状态改变。

4：UDRUN：传输“Underrun”

3：ROOS：接收溢出计数器溢出。

2：ROS：接收溢出。

1：PTS：数据包传输。

0：PRS：数据包接收。

ISR 寄存器各状态写 1 清除

```
/* Set address filter table */
dm9000_hash_table_unlocked(dev);

imr = IMR_PAR | IMR_PTM | IMR_PRM;
if (db->type != TYPE_DM9000E)
        imr |= IMR_LNKCHNG;

db->imr_all = imr;
```

```
    /* Enable TX/RX interrupt mask */
    iow(db, DM9000_IMR, imr);
```

IMR(FFH)：终端屏蔽寄存器(Interrupt Mask Register)

7：PAR：1 使能指针自动跳回。当 SRAM 的读、写指针超过 SRAM 的大小时，指针自动跳回起始位置。需要驱动程序设置该位，若设置则 REG_F5(MDRAH)将自动位 0CH。

6：保留。

5：LNKCHGI：1 使能连接状态改变中断。

4：UDRUNI：1 使能传输"Underrun"中断。

3：ROOI：1 使能接收溢出计数器溢出中断。

2：ROI：1 使能接收溢出中断。

1：PTI：1 使能数据包传输终端。

0：PRI：1 使能数据包接收中断。

注释：表示在 DM9000 初始化中要用到的寄存器。

访问以上寄存器的方法是通过总线驱动的方式，即通过对 IOR、IOW、AEN、CMD 以及 SD0--SD15 等相关引脚的操作来实现。其中 CMD 引脚为高 电平时为写寄存器地址，为低电平时为写数据到指定地址的寄存器中。详细过程请参考数据手册中"读写时序"部分。

在 DM9000(A)中，还有一些 PHY 寄存器，也称之为介质无关接口 MII 寄存器，需要我们去访问。这些寄存器是字对齐的，即 16 位宽。下面列出三个常用的 PHY 寄存器。

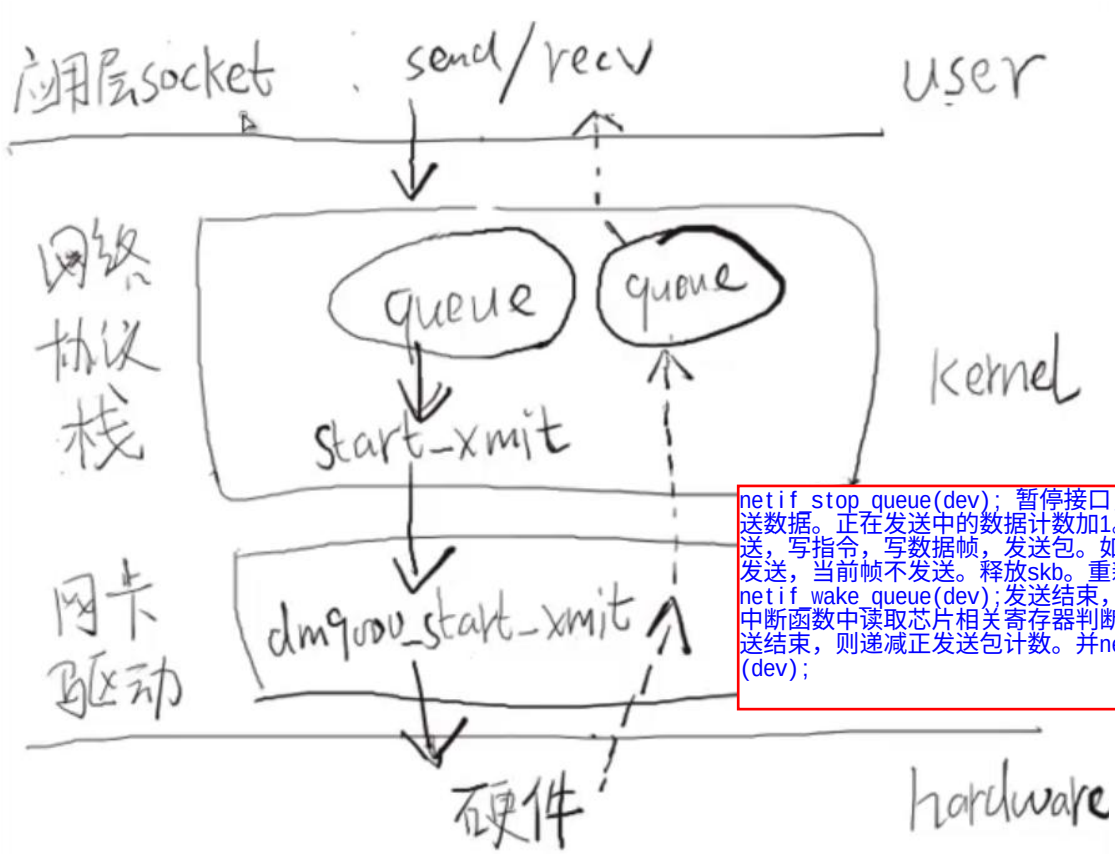
```
    /* Init Driver variable */
    db->tx_pkt_cnt = 0;
    db->queue_pkt_len = 0;
    dev->trans_start = jiffies;
}
```

```
    //申请中断
    if (request_irq(dev->irq, dm9000_interrupt, irqflags, dev->name, dev))
        return -EAGAIN;

    /* Init driver variable */
    db->dbug_cnt = 0;

    mii_check_media(&db->mii, netif_msg_link(db), 1);
    //允许发送
    netif_start_queue(dev);
    /*当应用层发送数据时，该数据被传送至网络协议栈，该协议栈维护一个数据的队列，然后该队列中的成员通过 start_xmit 将数据发送至网卡驱动最后发送至设备【ps:start_xmit 是由驱动实现，但是该函数被协议栈调用，将数据从协议栈的队列中发送至驱动。】*/
    dm9000_schedule_poll(db);
    return 0;
}
```

3.网卡发送流程--->start_xmit



网卡发送流程：

应用层调用 send 发送数据---->网络协议栈--->将发送出来的数据存入 queue--->协议栈调用 start_xmit 发送队列中的 skb 数据--->发送至硬件的 tx_ram 中

```
static int dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    unsigned long flags;
    //获取 board_info_t
    board_info_t *db = netdev_priv(dev);

    dm9000_dbg(db, 3, "%s:\n", __func__);
    //因为 DM9000 上的 tx_ram 最大只允许 2 个发送的数据包，若已经有两个数据包在等
    //待，此时返回忙
    if (db->tx_pkt_cnt > 1)
        return NETDEV_TX_BUSY;

    spin_lock_irqsave(&db->lock, flags);

    /* Move data to DM9000 TX RAM */
    //内存地址写命令，可将数据写入 DM9000 的 tx_ram 中
    writeb(DM9000_MWCMD, db->io_addr);
```

//数据手册

MWCMD(F8H)：存储器读地址自动增加的读数据命令(Memory Data Write Command With Address Increment Register)

7-0：MWCMD：写数据到发送 SRAM 中，之后指向内部 SRAM 的读指针自动增加 1、2 或 4，根据处理器的操作模式而定(8 位、16 位或 32 位)。

//将 skb 数据拷贝至 tx_ram 中，并将其发送数据长度及包的数量进行相应的增加。

```
(db->outblk)(db->io_data, skb->data, skb->len);
dev->stats.tx_bytes += skb->len;

db->tx_pkt_cnt++;
/* TX control: First packet immediately send, second packet queue */
if (db->tx_pkt_cnt == 1) {
    dm9000_send_packet(dev, skb->ip_summed, skb->len);
```

```
static void dm9000_send_packet(struct net_device *dev,
                               int ip_summed,
                               u16 pkt_len)
{
    board_info_t *dm = to_dm9000_board(dev);

    /* The DM9000 is not smart enough to leave fragmented packets alone. */
    if (dm->ip_summed != ip_summed) {
        if (ip_summed == CHECKSUM_NONE)
            iow(dm, DM9000_TCCR, 0);
        else
            iow(dm, DM9000_TCCR, TCCR_IP | TCCR_UDP | TCCR_TCP);
        dm->ip_summed = ip_summed;
    }

    /* Set TX length to DM9000 */
//设置 DM9000 待发送的数据长度
    iow(dm, DM9000_TXPLL, pkt_len);
    iow(dm, DM9000_TXPLH, pkt_len >> 8);
```

TXPLL(FCH)：发送数据包长度寄存器低半字节(TX Packet Length Low Byte Register)

7-0：TXPLL

TXPLH(FDH)：发送数据包长度寄存器高半字节(TX Packet Length High Byte Register)

7-0：TXPLH

```
    /* Issue TX polling command */
//dm9000 开始数据发送
    iow(dm, DM9000_TCR, TCR_TXREQ);    /* Cleared after TX complete */
```

TCR(02H)：发送控制寄存器(TX Control Register)

7：保留。

6：TJDIS：Jabber 传输使能。1 使能 Jabber 传输定时器(2048 字节)，0 禁止。

注释：Jabber 是一个有 CRC 错误的长帧(大于 1518byte 而小于 6000byte)或是数据包重组错误。原因：它可能导致网络丢包。多是由于作站有硬件或软件错误。

5：EXCECM：额外冲突模式控制。0 当额外的冲突计数多于 15 则终止本次数据包，1 始终尝试发发送本次数据包。

4：PAD_DIS2：禁止为数据包指针 2 添加 PAD。

3：CRC_DIS2：禁止为数据包指针 2 添加 CRC 校验。

2：PAD_DIS2：禁止为数据包指针 1 添加 PAD。

1：CRC_DIS2：禁止为数据包指针 1 添加 CRC 校验。

0：TXREQ：TX(发送)请求。发送完成后自动清零该位。

}

```
//当 DM9000 发送完毕后会产生一个中断
static irqreturn_t dm9000_interrupt(int irq, void *dev_id)
{
	struct net_device *dev = dev_id;
	board_info_t *db = netdev_priv(dev);
	int int_status;
	unsigned long flags;
	u8 reg_save;

	dm9000_dbg(db, 3, "entering %s\n", __func__);

	/* A real interrupt coming */

	/* holders of db->lock must always block IRQs */
	spin_lock_irqsave(&db->lock, flags);

	/* Save previous register address */
	reg_save = readb(db->io_addr);

	/* Disable all interrupts */
//使能指针自动跳回
	iow(db, DM9000_IMR, IMR_PAR);
```

IMR(FFH)：终端屏蔽寄存器(Interrupt Mask Register)

7：PAR：1 使能指针自动跳回。当 SRAM 的读、写指针超过 SRAM 的大小时，指针自动跳回起始位置。需要驱动程序设置该位，若设置则 REG_F5(MDRAH)将自动位 0CH。

6：保留。

5：LNKCHGI：1 使能连接状态改变中断。

4：UDRUNI：1 使能传输"Underrun"中断。

3：ROOI：1 使能接收溢出计数器溢出中断。

2：ROI：1 使能接收溢出中断。

1：PTI：1 使能数据包传输终端。

0：PRI：1 使能数据包接收中断。

注释：表示在 DM9000 初始化中要用到的寄存器。

访问以上寄存器的方法是通过总线驱动的方式，即通过对 IOR、IOW、AEN、CMD 以及 SD0--SD15 等相关引脚的操作来实现。其中 CMD 引脚为高 电平时为写寄存器地址，为低电平时为写数据到指定地址的寄存器中。详细过程请参考数据手册中"读写时序"部分。

在 DM9000(A)中，还有一些 PHY 寄存器，也称之为介质无关接口 MII 寄存器，需要我们去访问。这些寄存器是字对齐的，即 16 位宽。下面列出三个常用的 PHY 寄存器。

```
    /* Got DM9000 interrupt status */
//获取 DM9000 的中断状态，并在此写入以清除该中断标志
    int_status = ior(db, DM9000_ISR); /* Got ISR */
    iow(db, DM9000_ISR, int_status);  /* Clear ISR status */
```

ISR(FEH)：终端状态寄存器(Interrupt Status Register)

7-6：IOMODE：处理器模式。00 为 16 位模式，01 为 32 位模式，10 为 8 位模式，00 保留。

5：LNKCHG：连接状态改变。

4：UDRUN：传输"Underrun"

3：ROOS：接收溢出计数器溢出。

2：ROS：接收溢出。

1：PTS：数据包传输。

0：PRS：数据包接收。

ISR 寄存器各状态写 1 清除

```
    if (netif_msg_intr(db))
        dev_dbg(db->dev, "interrupt status %02x\n", int_status);

    /* Received the coming packet */
//根据不同的中断状态进行收数据及发数据的收尾工作
    if (int_status & ISR_PRS)
        dm9000_rx(dev);

    /* Trnasmit Interrupt check */
```

```
if (int_status & ISR_PTS)
        dm9000_tx_done(dev, db);
```

```
static void dm9000_tx_done(struct net_device *dev, board_info_t *db)
{
    int tx_status = ior(db, DM9000_NSR);   /* Got TX status */
//获取发送状态
```

NSR (01H)：网络状态寄存器(Network Status Register)

7：SPEED：媒介速度，在内部 PHY 模式下，0 为 100Mbps，1 为 10Mbps。当 LINKST=0 时，此位不用。

6：LINKST：连接状态，在内部 PHY 模式下，0 为连接失败，1 为已连接。

5：WAKEST：唤醒事件状态。读取或写 1 将清零该位。不受软件复位影响。

4：保留。

3：TX2END：TX(发送)数据包 2 完成标志，读取或写 1 将清零该位。数据包指针 2 传输完成。

2：TX2END：TX(发送)数据包 1 完成标志，读取或写 1 将清零该位。数据包指针 1 传输完成。

1：RXOV：RX(接收)FIFO(先进先出缓存)溢出标志。

0：保留。

```
    //如果发送完第一个或第二个包则待发送包的数量减 1，已发送的数
        //量加一
if (tx_status & (NSR_TX2END | NSR_TX1END)) {
        /* One packet sent complete */
        db->tx_pkt_cnt--;
        dev->stats.tx_packets++;

        if (netif_msg_tx_done(db))
            dev_dbg(db->dev, "tx done, NSR %02x\n", tx_status);

        /* Queue packet check & send */
        //如果仍有待发送的包，则继续发送
        if (db->tx_pkt_cnt > 0)
            dm9000_send_packet(dev, db->queue_ip_summed,
                    db->queue_pkt_len);
        //再次调用 start_xmit 进行数据发送
        netif_wake_queue(dev);
    }
```

```
        }
    if (db->type != TYPE_DM9000E) {
        if (int_status & ISR_LNKCHNG) {
            /* fire a link-change request */
            schedule_delayed_work(&db->phy_poll, 1);
        }
    }

    /* Re-enable interrupt mask */
    iow(db, DM9000_IMR, db->imr_all);

    /* Restore previous register address */
    writeb(reg_save, db->io_addr);

    spin_unlock_irqrestore(&db->lock, flags);

    return IRQ_HANDLED;
}
```

```
} else {
//当前一个包未发送完成，后一个包又来了，此时进入 else，调用 netif_stop_queue 通
//知协议栈不要再次调用 start_xmit 了
    /* Second packet */
    db->queue_pkt_len = skb->len;
    db->queue_ip_summed = skb->ip_summed;
    netif_stop_queue(dev);
}

spin_unlock_irqrestore(&db->lock, flags);

/* free this SKB */
dev_kfree_skb(skb);

return NETDEV_TX_OK;
}
```

网络数据包到达，DM9000自动接收并存放在DM内部RAM中，产生中断。在中断处理中识别中断原因并调用接收处理函数dm9000_rx (struct net_device *dev)。
dm9000_rx：
读取芯片相关寄存器确认DM9000正确的收到一帧数据。
申请skb_buffer，将数据从DM9000中拷贝到skb_buffer中。设置skb->dev=nev,skb->protocol=eth_type_trans(skb, dev)。

然后把skb_buffer交给上层协议：netif_rx(skb);

最后更新接口统计信息：db->stats.rx_packets++; 收到包总数+1。

整个DM9000驱动的移植和源码主要部分的简要分析至此结束。

4.DM9000 网卡接收

数据接收流程如下：

（1）读取一个字节需要用到 MRCMDX 寄存器，MRCMDX 寄存器是存储数据读命令寄存器

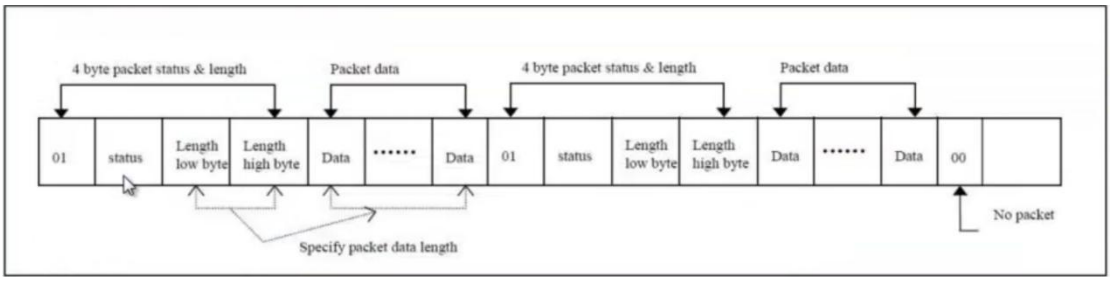
（2）读取头部并放入 struct dm9000_rxhdr 中，需要用到 MRCMD 寄存器（存储数据读取命

令，改指针自动增加）判断是否是正常封包

（3）读取包中的数据，也需要用到 MRCMD 寄存器

（4）分配一个 struct sk_buffer,并把包中的数据复制到 sk_buffer

（5）调用 netifx_rx(struct sk_buffer *skb);把 skb 提交给协议栈




```
    struct dm9000_rxhdr {
        u8    RxPktReady;
        u8    RxStatus;
        __le16    RxLen;
    } __packed;
static void       dm9000_rx(struct net_device *dev)
{
    board_info_t *db = netdev_priv(dev);
    struct dm9000_rxhdr rxhdr;
    struct sk_buff *skb;
    u8 rxbyte, *rdptr;
    bool GoodPacket;
    int RxLen;

    /* Check packet ready or not */
    do {
        ior(db, DM9000_MRCMDX);  /* Dummy read */
```

//数据手册

MRCMDX(F0H)：存储器地址不变的读数据命令(Memory Data Pre-Fetch Read Command Without Address Increment Register)

7-0：MRCMDX：从接收 SRAM 中读数据，读取之后，指向内部 SRAM 的读指针不变。

```
        /* Get most updated data */
        //读到 rx_ram 中
        rxbyte = readb(db->io_data);

        /* Status check: this byte must be 0 or 1 */
        if (rxbyte & DM9000_PKT_ERR) {
            dev_warn(db->dev, "status check fail: %d\n", rxbyte);
            iow(db, DM9000_RCR, 0x00);/* Stop Device */
```

//数据手册

RCR(05H)：接收控制寄存器(RX Control Register)

7：保留。

6：WTDIS：看门狗定时器禁止。1 禁止，0 使能。

5：DIS_LONG：丢弃长数据包。1 为丢弃数据包长度超过 1522 字节的数据包。

4：DIS_CRC：丢弃 CRC 校验错误的数据包。

3：ALL：忽略所有多点传送。

2：RUNT：忽略不完整的数据包。

1：PRMSC：混杂模式(Promiscuous Mode)

0：RXEN：接收使能。

```
iow(db, DM9000_ISR, IMR_PAR);   /* Stop INT request */
```

//数据手册

ISR(FEH)：终端状态寄存器(Interrupt Status Register)

7-6：IOMODE：处理器模式。00 为 16 位模式，01 为 32 位模式，10 为 8 位模式，00 保留。

5：LNKCHG：连接状态改变。

4：UDRUN：传输“Underrun”

3：ROOS：接收溢出计数器溢出。

2：ROS：接收溢出。

1：PTS：数据包传输。

0：PRS：数据包接收。

ISR 寄存器各状态写 1 清除

```
        return;
    }

    if (!(rxbyte & DM9000_PKT_RDY))
        return;

    /* A packet ready now    & Get status/length */
    GoodPacket = true;
    writeb(DM9000_MRCMD, db->io_addr);
```

//数据手册

MRCMD(F2H)：存储器读地址自动增加的读数据命令(Memory Data Read Command With Address Increment Register)

7-0：MRCMD：从接收 SRAM 中读数据，读取之后，指向内部 SRAM 的读指针自动增加 1、2 或 4，根据处理器的操作模式而定(8 位、16 位或 32 位)。

```
(db->inblk)(db->io_data, &rxhdr, sizeof(rxhdr));
//将 DM9000 的字节序转换为 cpu 字节序
RxLen = le16_to_cpu(rxhdr.RxLen);

if (netif_msg_rx_status(db))
    dev_dbg(db->dev, "RX: status %02x, length %04x\n",
        rxhdr.RxStatus, RxLen);
//数据长度小于 64 或者大于 1536 的包丢掉
/* Packet Status check */
if (RxLen < 0x40) {
    GoodPacket = false;
    if (netif_msg_rx_err(db))
        dev_dbg(db->dev, "RX: Bad Packet (runt)\n");
}

if (RxLen > DM9000_PKT_MAX) {
    dev_dbg(db->dev, "RST: RX Len:%x\n", RxLen);
}

//根据接受包的不同的错误状态标识进行不同的操作错误数量统计
/* rxhdr.RxStatus is identical to RSR register. */
if (rxhdr.RxStatus & (RSR_FOE | RSR_CE | RSR_AE |
                      RSR_PLE | RSR_RWTO |
                      RSR_LCS | RSR_RF)) {
    GoodPacket = false;
    if (rxhdr.RxStatus & RSR_FOE) {
        if (netif_msg_rx_err(db))
            dev_dbg(db->dev, "fifo error\n");
        dev->stats.rx_fifo_errors++;
    }
    if (rxhdr.RxStatus & RSR_CE) {
        if (netif_msg_rx_err(db))
            dev_dbg(db->dev, "crc error\n");
        dev->stats.rx_crc_errors++;
    }
    if (rxhdr.RxStatus & RSR_RF) {
        if (netif_msg_rx_err(db))
            dev_dbg(db->dev, "length error\n");
        dev->stats.rx_length_errors++;
```

```
                }
            }

            //从 DM9000 中拷贝数据
            /* Move data from DM9000 */
            if (GoodPacket &&
                ((skb = netdev_alloc_skb(dev, RxLen + 4)) != NULL)) {//设备申请 skb
                skb_reserve(skb, 2);//协议栈对其机制
                rdptr = (u8 *) skb_put(skb, RxLen - 4);//空出 data 域给待接收的数据

                /* Read received packet from RX SRAM */

                (db->inblk)(db->io_data, rdptr, RxLen);//将接收到的数据放入 skb
                dev->stats.rx_bytes += RxLen;

                /* Pass to upper layer */
                skb->protocol = eth_type_trans(skb, dev);//skb 的网络接口协议赋值
                if (dev->features & NETIF_F_RXCSUM) {//数据校验
                    if ((((rxbyte & 0x1c) << 3) & rxbyte) == 0)
                        skb->ip_summed = CHECKSUM_UNNECESSARY;
                    else
                        skb_checksum_none_assert(skb);
                }
                netif_rx(skb);//将 skb 提交给协议栈
                dev->stats.rx_packets++;

            } else {
                /* need to dump the packet's data */
                (db->dumpblk)(db->io_data, RxLen);
            }
        } while (rxbyte & DM9000_PKT_RDY);
}
```