
网名“鱼树”的学员聂龙浩,

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细,供大家参考。

也许有错漏,请自行分辨。

目录

单片机驱动 DM9000	3
简述	3
一、电路连接	3
二、编写驱动程序	4
1、读、写寄存器	4
2、初始化 DM9000 网卡芯片。	6
3、发送、接收数据包	7
DM9000 寄存器	11
DM9000 寄存器功能详细介绍	11
以下为 DM9000 的寄存器功能详解:	11
下面列出三个常用的 PHY 寄存器。	18
访问 PHY 寄存器的方法是:	19
DM9000C 确定相异性	20
差异性 和 寄存器设置:	20
一, 分析厂家提供的 DM9000C 网卡驱动程序:	20
1, 入口函数:	20
2, dmfe_probe1 函数: 设置 硬件 。	21
3, int dmfe_open(struct net_device *dev)	21
二, 从厂家提供的驱动程序上修改移植:	22
移植: 找出差异性来修改。	22
修改源代码:	22
移植修改:	24
4, 在完成基地址, 中断号的设置后, 直接在厂家驱动的基础上编译驱动成功。	28
设置位宽 DW4 bit[17:16]:	31
DM9000C 设置时序	32
DM9000 时序:	32
1. ST4[19]: SRAM 是否使用 UB/LB(upper byte/lower byte)。	32
2. WS4[18]: WAIT 状态。	32
3. DW4[16:17]: 数据总线宽度。	32

设置 BANK4 寄存器	32
看“时间参数”：	33
读时序：	34
写时序：	34
下面是要设置的位域 S3C2440A nGCS Timing Diagram	35
DM9000:读时序	35
DM9000 写信号 时序：	36
DM9000 的操作：从源代码上分析：	37
总结：	45
DM9000C 驱动侧式及内存控制器	47
测试 DM9000C 驱动程序：	47
1. 把 dm9dev9000c.c 放到内核的 drivers/net 目录下	47
2. 修改 drivers/net/Makefile.....	47
3. make uImage	47
4. 使用 NFS 启动	47
设置位宽和 BANKCON4。	47
当“Tacc[10:8]”：	47
内存控制器：	48
实例：	48
“内存控制器”要发起 2 次传输：	49
例 2：	50
DM9000 驱动在 MINI2440 上的移植	51
首先看一下 DM9000 的引脚和 MINI2440 的引脚连接	51
DM9000 的写时序	52
DM9000 的读时序如下	55
关于 BANCON4 的：看数据手册	60
我们找出 s3c2440 和 dm9000 的时序段对应关系：	61

单片机驱动 DM9000

简述

单片机驱动 DM9000 和其它网卡芯片不同，DM9000 系列网卡芯片在嵌入式开发板上很常见，尤其是有关 ARM-Linux 的开发板上的网络连接部分几乎都是采用该芯片完成的。当然，其它网卡芯片，如 RTL8019 的应用也很常见，在很多开发板上得到应用然而 RTL8019 的介绍在网上可以找到非常详细的介绍，尤其是用单片机对其做底层驱动的介绍非常丰富。下面的网站就介绍了用 AVR 驱动 RTL8019 网卡芯片的非常详细的过程，有兴趣的朋友可以参考一下。

<http://members.home.nl/bzijlstra/software/examples/RTL8019as.htm>

AVR 驱动 RTL8019 网卡芯片的详细介绍。

言归正传。在网上也能找到许多关于 DM9000 网卡芯片的介绍，然而这些介绍大多是关于 Linux 或 WinCE 下的驱动程序或移植，很少有介绍单片机驱动 DM9000 的例子。因此我在这里把我调试 DM9000E 的过程详细说明一下，仅供参考。

本文主要介绍单片机驱动 DM9000E 网卡芯片的详细过程。

从网卡电路的连接，到网卡初始化相关程序调试，再到 ARP 协议的实现，一步一步详细介绍调试过程。如果有时间也会把 UDP 和 TCP 通讯实验过程写出来。当然，会用单片机编写 DM9000 的驱动，再想编写 ARM 下的 Linux 的驱动就容易的多了。在调试之前，应该先参考两份技术文档，可以从下面网站中下载。

DM9000E.pdf (芯片数据资料) 和 DM9000 Application Notes Ver 1_22
061104.pdf (应用手册)

<http://www.davicom.com.tw>

或者

DM9000 Datasheet VF03:

<http://www.davicom.com.tw/userfile/24> ... M9000-DS-F03-
041906_1.pdf

DM9000A Datasheet:

[http://www.davicom.com.tw/userfile/24247/DM9000A-DS-F01-
101906.pdf](http://www.davicom.com.tw/userfile/24247/DM9000A-DS-F01-101906.pdf)

DM9000 Application Notes V1.22

<http://www.davicom.com.tw/big5/download> ...
tes_Ver_1_22%20061104.pdf

一、电路连接

DM9000E 网卡芯片支持 8 位、16 位、32 位模式的处理器，通过芯片引脚 EEDO (65 脚) 和 WAKEUP (79 脚) 的复位值设置支持的处理器类型，如 16 位处理器只需将这两个引脚接低电平即可，其中 WAKEUP 内部有 60K 下拉电阻，因此可悬空该引脚，或作为网卡芯片唤醒输出用。其它型号请参考相应的数据手册。

图 1 DM9000 引脚

如 图所示，对处理器驱动网卡芯片来说，我们比较关心的有以下几个引脚：IOR、IOW、AEN、CMD（SA2）、INT、RST，以及数据引脚 SD0- SD15-SD31 和地址引脚 SA4-SA9。其中，地址引脚配合 AEN 引脚来选通该网卡芯片，对于大多数的应用来说没有意义，因为在我们的应用中一般只 用一个网卡芯片，而这些地址引脚主要用于在多网卡芯片环境下选择其中之一。DM9000 工作的默认基地址为 0x300，这里我们按照默认地址选择，将 SA9、SA8 接高电平，SA7-DA4 接低电平。多网卡环境可以根据 TXD0-TXD3 配置 SA4-SA7 来选择不同的网卡，这里不做介绍，有兴趣的朋友请参考应用手册和数据手册。数据引脚 SD0-SD31 则根据前面所讲的配置处理器模式与处理器的数据总线进行选择连接即可，没用到的引脚悬空。那么，除了地址、数据引脚外，剩下的与处理器有关引脚对我们来说及其重要了，而与处理器无关的引脚，只需按照应用手册连接即可。

IOR 和 IOW 是 DM9000 的读写选择引脚，低电平有效，即低电平时进行读（IOR）写（IOW）操作；AEN 是芯片选通引脚，低电平有效，该引脚为低时才能进行读写操作；CMD 的命令/数据切换引脚，低电平时读写命令操作，高电平时读写数据操作。

图 2 读时序

图 3 写时序

这 些引脚接口和其它单片机外围器件的引脚接口基本相同，其使用也一样。对于有总线接口的单片机来说，如 51 系列，ARM 等直接连接即可。对于没有总线接口的 来说，如 AVR mega32 等可以直接用 I/O 引脚模拟总线时序进行连接。连接时要参考读写时序，如上图所示。具体连接电路，有时间我再画出来，暂时先略了。

二、编写驱动程序

在 这，我使用 C 语言编写驱动程序，这需要非常注意一点，即处理器所用的 C 编译器使用“大端格式”还是“小端格式”，这可以在相应处理器的 C 编译器说明上找 到。一般比较常见的是小端格式。而对于 8 位处理器来说，在编写驱动程序时，可以不考虑，但是在编写网络协议的时候，一定好考虑，因为网络协议的格式是大端 格式，而大部分编译器或者我们习惯的是小端格式，这一点需要注意。

在 DM9000 中，只有两个可以直接被处理器访问的寄存器，这里命名为 CMD 端口和 DATA 端口。

事实上，DM9000 中有许多控制和状态寄存器（这些寄存器在上一篇文章中有详细的使用说明），但它们都不能直接被处理器访问，访问这些控制、状态寄存器的方法是：

- （1）、将寄存器的地址写到 CMD 端口；
- （2）、从 DATA 端口读写寄存器中的数据；

1、读、写寄存器

其实，INDEX 端口和 DATA 端口的就是由芯片上的 CMD 引脚来区分的。低电平为 INDEX 端口，高电平为 DATA 端口。所以，要想实现读写寄存器，就必须先控制好 CMD 引脚。若使用总线接口连接 DM9000 的话，假设总线连接后芯片的基地址为 0x800300（24 根地址总线），只需如下方法：

```
#define DM_ADD (*[1]
#define DM_CMD (*[2]
//向DM9000寄存器写数据
```

```

void dm9000_reg_write(unsigned char reg, unsigned char data)
{
    udelay(20); //之前定义的微妙级延时函数，这里延时20us
    DM_ADD = reg; //将寄存器地址写到INDEX端口
    udelay(20);
    DM_CMD = data; //将数据写到DATA端口，即写进寄存器
}

//从DM9000寄存器读数据
unsigned int dm9000_reg_read(unsigned char reg)
{
    udelay(20);
    DM_ADD = reg;
    udelay(20);
    return DM_CMD; //将数据从寄存器中读出
}

```

值得注意的是前面的两个宏定义 `DM_ADD` 和 `DM_CMD`，定义的内容表示指向无符号整形变量的指针，在这里 `0x800300` 是 DM9000 命令端口的地址，对它的赋值操作就相当于把数据写到该地址中，即把数据写到 DM9000 的命令端口中。读的道理也一样。这是一种很常见的宏定义，一般在处理器中定义通用寄存器也是这样定义的。

若没有总线接口的话，可以使用 IO 口模拟总线时序的方法实现寄存器的读写。这里只说明实现步骤。首先将处理器的 I/O 端口与 DM9000 的 IOR 等引脚直接相连（电平匹配的情况下），又假设已经有宏定义“IOR”I/O 端口控制 DM9000 的 IOR 引脚，其它端口控制 DM9000 引脚的命名相同，“PIO1”（根据处理器情况，可以是 8 位、16 位或 32 位的 I/O 端口组成）控制数据端口。这样宏命名更直观些。写寄存器的函数如下：

```

void dm9000_reg_write(unsigned char reg, unsigned char data)
{
    PIO1 = reg;
    AEN = 0;
    CMD = 0;
    IOR = 1;
    IOW = 0;
    udelay(1);
    AEN = 1;
    IOW = 1;
    udelay(20);
    PIO1 = data;
    AEN = 0;
    CMD = 0;
    IOR = 1;
    IOW = 0;
    udelay(1);
    AEN = 1;
    IOW = 1;
}

```

读寄存器的写法类似，这里就略一下了。

这一过程看上去有些复杂，呵呵，其实执行起来也蛮有效率的，执行时间差不多。这种模拟总线时序的方式实际并不复杂，只是把总线方式下自动执行的过程手动的执行了一遍而已。

在 DM9000 中，还有一些 PHY 寄存器，也称之为介质无关接口 MII (Media Independent Interface) 寄存器。

对这些寄存器的操作会影响网卡芯片的初始化和网络连接，这里不对其进行操作，所以对这些寄存器的访问方法这里也略了（在上篇文章中有介绍）。操作不当反而使网卡不能连接到网络。

至此，我们已经写好了两个最基本的函数：dm9000_reg_write() 和 dm9000_reg_read()，以及前面的宏定义 DM_ADD 和 DM_CMD。下面将一直用到。

2、初始化 DM9000 网卡芯片。

初始化 DM9000 网卡芯片的过程，实质上就是填写、设置 DM9000 的控制寄存器的过程，这里以程序为例进行说明。其中寄存器的名称宏定义在 DM9000.H 中已定义好。

注：一下函数中 unsigned char 为一个字节 unsigned int 为两个字节

```
//DM9000初始化
void DM9000_init(void)
{
    unsigned int i;
    IOODIR |= 1 << 8;
    IO1CLR |= 1 << 8;
    udelay(500000);
    IO2SET |= 1 << 8;
    udelay(500000);
    IO1CLR |= 1 << 8;
    udelay(500000);
    /*以上部分是利用一个IO口控制DM9000的RST引脚，使其复位。这一步可以省略，可以用下面的软件复位代替*/

    dm9000_reg_write(GPCR, 0x01); //设置 GPCR(1EH) bit[0]=1, 使DM9000的GPIO3为输出。
    dm9000_reg_write(GPR, 0x00); //GPR bit[0]=0 使DM9000的GPIO3输出为低以激活内部PHY。
    udelay(5000); //延时2ms以上等待PHY上电。
    dm9000_reg_write(NCR, 0x03); //软件复位
    udelay(30); //延时20us以上等待软件复位完成
    dm9000_reg_write(NCR, 0x00); //复位完成，设置正常工作模式。
    dm9000_reg_write(NCR, 0x03); //第二次软件复位，为了确保软件复位完全成功。此步骤是必要的。
    udelay(30);
    dm9000_reg_write(NCR, 0x00);
    /*以上完成了DM9000的复位操作*/

    dm9000_reg_write(NSR, 0x2c); //清除各种状态标志位
    dm9000_reg_write(ISR, 0x3f); //清除所有中断标志位
    /*以上清除标志位*/

    dm9000_reg_write(RCR, 0x39); //接收控制
    dm9000_reg_write(TCR, 0x00); //发送控制
    dm9000_reg_write(BPTR, 0x3f);
```

```

    dm9000_reg_write(FCTR, 0x3a);
    dm9000_reg_write(RTFCR, 0xff);
    dm9000_reg_write(SMCR, 0x00);
/*以上是功能控制，具体功能参考上一篇文章中的说明，或参考数据手册的介绍*/
    for(i=0; i<6; i++)
        dm9000_reg_write(PAR + i, mac_addr[i]); //mac_addr[]自己定义一下吧，6个字节的MAC地址
/*以上存储MAC地址（网卡物理地址）到芯片中去，这里没有用EEPROM，所以需要自己写进去*/
/*关于MAC地址的说明，要参考网络相关书籍或资料*/
    dm9000_reg_write(NSR, 0x2c);
    dm9000_reg_write(ISR, 0x3f);
/*为了保险，上面有清除了一次标志位*/
    dm9000_reg_write(IMR, 0x81);
/*中断使能（或者说中断屏蔽），即开启我们想要的中断，关闭不想要的，这里只开启的一个接收中断*/
/*以上所有寄存器的具体含义参考上一篇文章，或参考数据手册*/
}

```

这样就对 DM9000 初始化完成了，怎么样，挺简单的吧。

3、发送、接收数据包

同样，以程序为例，通过注释说明。

```

//发送数据包
//参数：datas为要发送的数据缓冲区（以字节为单位），length为要发送的数据长度（两个字节）。
void sendpacket(unsigned char *datas, unsigned int length)
{
    unsigned int len, i;
    dm9000_reg_write(IMR, 0x80); //先禁止网卡中断，防止在发送数据时被中断干扰
    len = length;
    dm9000_reg_write(TXPLH, (len>>8) & 0x0ff);
    dm9000_reg_write(TXPLL, len & 0x0ff);
/*这两句是将要发送数据的长度告诉DM9000的寄存器*/
    DM_ADD = MWCMD; //这里的写法是针对有总线接口的处理器，没有总线接口的处理器要注意加上时序。
    for(i=0; i<len; i+=2) //16 bit mode
    {
        udelay(20);
        DM_CMD = datas[i] | (datas[i+1]<<8);
    }
/*上面是将要发送的数据写到DM9000的内部SRAM中的写FIFO中，注意没有总线接口的处理器要加上适当的时序*/
/*只需要向这个寄存器中写数据即可，MWCMD是DM9000内部SRAM的DMA指针，根据处理器模式，写后自动增加*/
    dm9000_reg_write(TCR, 0x01); //发送数据到以太网上
    while((dm9000_reg_read(NSR) & 0x0c) == 0); //等待数据发送完成
    udelay(20);
    dm9000_reg_write(NSR, 0x2c); //清除状态寄存器，由于发送数据没有设置中断，因此不必处理中断标志位
}

```

```
dm9000_reg_write(IMR, 0x81); //DM9000网卡的接收中断使能
}
```

以上是发送数据包，过程很简单。而接收数据包确需要些说明了。DM9000 从网络中接到一个数据包后，会在数据包前面加上 4 个字节，分别为“01H”、“status”（同 RSR 寄存器的值）、“LENL”（数据包长度低 8 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数据包，若为“00H”则表示没有数据包，若为其它值则表示网卡没有正确初始化，需要从新初始化。

如果接收到的数据包长度小于 60 字节，则 DM9000 会自动为不足的字节补上 0，使其达到 60 字节。同时，在接收到的数据包后 DM9000 还会自动添加 4 个 CRC 校验字节。可以不予处理。于是，接收到的数据包的最小长度也会是 64 字节。当然，可以根据 TCP/IP 协议从首部字节中出有效字节数，这部分在后面 讲解。下面为接收数据包的函数。

```
//接收数据包
//参数: datas为接收到是数据存储位置（以字节为单位）
//返回值: 接收成功返回数据包类型，不成功返回0
unsigned int receivepacket(unsigned char *datas)
{
    unsigned int i, tem;
    unsigned int status, len;
    unsigned char ready;
    ready = 0; //希望读取到“01H”
    status = 0; //数据包状态
    len = 0; //数据包长度
    /*以上为有效数据包前的4个状态字节*/
    if(dm9000_reg_read(ISR) & 0x01)
    {
        dm9000_reg_write(ISR, 0x01);
    }
    /*清除接收中断标志位*/
    /*****
    *****/
    /*这个地方遇到了问题，下面的黑色字体语句应该替换成红色字体，也就是说 MRCMDX
    寄存器如果第一次读不到数据，还要读一次才能确定完全没有数据。
    在做 PING 实验时证明：每个数据包都是通过第二次的读取 MRCMDX 寄存器操作而获知为
    有效数据包的，对初始化的寄存器做了多次修改依然是此结果，但是用如下方法来实现，
    绝不会漏掉数据包。*/
```

```
ready = dm9000_reg_read(MRCMDX); // 第一次读取，一般读取到的是 00H
if((ready & 0x0ff) != 0x01)
{
    ready = dm9000_reg_read(MRCMDX); // 第二次读取，总能获取到数据
    if((ready & 0x01) != 0x01)
    {
        if((ready & 0x01) != 0x00) //若第二次读取到的不是 01H 或 00H，则表示没有初始化
成功
```



```

        {
            dm9000_reg_write(IMR, 0x80); // 屏幕网卡中断
            DM9000_init(); // 重新初始化
            dm9000_reg_write(IMR, 0x81); // 打开网卡中断
        }

        return 0;
    }
}

/* ready = dm9000_reg_read(MRCMDX); // read a byte without pointer increment
   if(!(ready & 0x01))
   {
       return 0;
   }*/

/*****
/*以上表示若接收到的第一个字节不是“01H”，则表示没有数据包，返回0*/
   status = dm9000_reg_read(MRCMD);
   udelay(20);
   len = DM_CMD;
   if(!(status & 0xbf00) && (len < 1522))
   {
       for(i=0; i<len; i+=2) // 16 bit mode
       {
           udelay(20);
           tem = DM_CMD;
           datas[i] = tem & 0x0ff;
           datas[i + 1] = (tem >> 8) & 0x0ff;
       }
   }
   else
   {
       return 0;
   }
/*以上接收数据包，注意的地方与发送数据包的地方相同*/
   if(len > 1000) return 0;
   if( (HON( ETHBUF->type ) != ETHTYPE_ARP) &&
       (HON( ETHBUF->type ) != ETHTYPE_IP) )
   {
       return 0;
   }
   packet_len = len;
/*以上对接收到的数据包作一些必要的限制，去除大数据包，去除非ARP或IP的数据包*/
   return HON( ETHBUF->type ); // 返回数据包的类型，这里只选择是ARP或IP两种类型
}

```

注意：上面的函数用到了一些宏定义，已经在头文件中定义过，这里说明一下：其中 `uint16` 定义为两个字节的变量，根据 C 编译器进行定义。

```
unsigned char Buffer[1000]; //定义了一个1000字节的接收发送缓冲区
uint16 packet_len; //接收、发送数据包的长度，以字节为单位。
struct eth_hdr //以太网头部结构，为了以后使用方便
{
    unsigned char d_mac[6]; //目的地址
    unsigned char s_mac[6]; //源地址
    uint16 type; //协议类型
};
struct arp_hdr //以太网头部+ARP首部结构
{
    struct eth_hdr ethhdr; //以太网首部
    uint16 hwtype; //硬件类型(1表示传输的是以太网MAC地址)
    uint16 protocol; //协议类型(0x0800表示传输的是IP地址)
    unsigned char hwlen; //硬件地址长度(6)
    unsigned char protolen; //协议地址长度(4)
    uint16 opcode; //操作(1表示ARP请求, 2表示ARP应答)
    unsigned char smac[6]; //发送端MAC地址
    unsigned char sipaddr[4]; //发送端IP地址
    unsigned char dmac[6]; //目的端MAC地址
    unsigned char dipaddr[4]; //目的端IP地址
};
struct ip_hdr //以太网头部+IP首部结构
{
    struct eth_hdr ethhdr; //以太网首部
    unsigned char vhl, //4位版本号4位首部长度(0x45)
        tos; //服务类型(0)
    uint16 len, //整个IP数据报总字节长度
        ipid, //IP标识
        ipoffset; //3位标识13位偏移
    unsigned char ttl, //生存时间(32或64)
        proto; //协议(1表示ICMP, 2表示IGMP, 6表示TCP, 17表示UDP)
    uint16 ipchksum; //首部校验和
    unsigned char srcipaddr[4], //源IP
        destipaddr[4]; //目的IP
};
```

以上定义的三种首部结构，是根据 TCP/IP 协议的相关规范定义的，后面会对 ARP 协议进行详细讲解。

附注

```
(volatile unsigned int *) 0x8000300
(volatile unsigned int *) 0x8000304
```

DM9000 寄存器

DM9000 寄存器功能详细介绍

DM9000 (A) 是一个全集成、功能强大、性价比高的快速以太网 MAC 控制器，它带有一个通用处理器接口、EEPROM 接口、10/100 PHY 和 16KB 的 SRAM (13KB 作为接收 FIFO, 3KB 作为发送 FIFO)。它采用单电源供电，可兼容 3.3V、5V 的 IO 接口电平。

DM9000 (A) 同样支持 MII (Media Independent Interface 介质无关) 接口，连接到 HPNA (Home Phone-line Networking Alliance 家用电话网络联盟) 设备上或其它支持 MII 的设备。

DM9000 (A) 包含一系列可被访问的控制状态寄存器，这些寄存器是字节对齐的，他们在硬件或软件复位时被设置成初始值。

以下为 DM9000 的寄存器功能详解：

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。

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, 写 1 将清零该位(官方说明书上是说读或写都可清零, 但是经过实验, 只要写 1 才能清零!)。数据包指针 2 传输完成。
- 2: TX1END: TX (发送) 数据包 1 完成标志, 写 1 将清零该位(官方说明书上是说读或写都可清零, 但是经过实验, 只要写 1 才能清零!)。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。

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。多是由于工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_DIS1: 禁止为数据包指针 1 添加 PAD。
- 1: CRC_DIS1: 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。

TSR_I (03H): 数据包指针 1 的发送状态寄存器 1 (TX Status Register I)

- 7: TJT0: Jabber 传输超时。该位置位表示由于多于 2048 字节数据被传输而导致数据帧被截掉。
- 6: LC: 载波信号丢失。该位置位表示在帧传输时发生红载波信号丢失。在内部回环模式下该位无效。
- 5: NC: 无载波信号。该位置位表示在帧传输时无载波信号。在内部回环模式下该位无效。
- 4: LC: 冲突延迟。该位置位表示在 64 字节的冲突窗口后又发生冲突。
- 3: COL: 数据包冲突。该位置位表示传输过程中发生冲突。
- 2: EC: 额外冲突。该位置位表示由于发生了第 16 次冲突 (即额外冲突) 后, 传送被终止。
- 1-0: 保留。

TSR_II (04H): 数据包指针 2 的发送状态寄存器 2 (TX Status Register II)

同 TSR_I

略。

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。

RSR (06H): 接收状态寄存器 (RX Status Register)

- 7: RF: 不完整数据帧。该位置位表示接收到小于 64 字节的帧。
- 6: MF: 多点传送帧。该位置位表示接收到帧包含多点传送地址。
- 5: LCS: 冲突延迟。该位置位表示在帧接收过程中发生冲突延迟。
- 4: RWT0: 接收看门狗定时溢出。该位置位表示接收到大于 2048 字节数据帧。

- 3: PLE: 物理层错误。该位置位表示在帧接收过程中发生物理层错误。
- 2: AE: 对齐错误 (Alignment)。该位置位表示接收到的帧结尾处不是字节对齐, 即不是以字节为边界对齐。
- 1: CE: CRC 校验错误。该位置位表示接收到的帧 CRC 校验错误。
- 0: FOE: 接收 FIFO 缓存溢出。该位置位表示在帧接收时发生 FIFO 溢出。

ROCR (07H): 接收溢出计数寄存器 (Receive Overflow Counter Register)

- 7: RXFU: 接收溢出计数器溢出。该位置位表示 ROC (接收溢出计数器) 发生溢出。
- 6-0: ROC: 接收溢出计数器。该计数器为静态计数器, 指示 FIFO 溢出后, 当前接收溢出包的个数。

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 250us, 1001 为 300us, 1010 为 350us, 1011 为 400us, 1100 为 450us, 1101 为 500us, 1110 为 550us, 1111 为 600us。

FCTR (09H): 溢出控制门限寄存器 (Flow Control Threshold Register)

- 7-4: HWOT: 接收 FIFO 缓存溢出门限最高值。当接收 SRAM 空闲空间小于该门限值, 则发送一个暂停时间 (pause_time) 为 FFFFH 的暂停包。若该值为 0, 则无接收空闲空间。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: LWOT: 接收 FIFO 缓存溢出门限最低值。当接收 SRAM 空闲空间大于该门限值, 则发送一个暂停时间 (pause_time) 为 0000H 的暂停包。当溢出门限最高值的暂停包发送之后, 溢出门限最低值的暂停包才有效。默认值为 8K 字节。不要超过 SRAM 大小。

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。

EPCCR/PHY_CR (0BH): EEPROM 和 PHY 控制寄存器 (EEPROM & PHY Control Register)

- 7-6: 保留。
- 5: REEP: 重新加载 EEPROM。驱动程序需要在该操作完成后清零该位。
- 4: WEP: EEPROM 写使能。
- 3: EPOS: EEPROM 或 PHY 操作选择位。0 选择 EEPROM, 1 选择 PHY。

- 2: ERPRR: EEPROM 读, 或 PHY 寄存器读命令。驱动程序需要在该操作完成后清零该位。
- 1: ERPRW: EEPROM 写, 或 PHY 寄存器写命令。驱动程序需要在该操作完成后清零该位。
- 0: ERRE: EEPROM 或 PHY 的访问状态。1 表示 EEPROM 或 PHY 正在被访问。

EPAR/PHY_AR (0CH) : EEPROM 或 PHY 地址寄存器 (EEPROM & PHY Address Register)

- 7-6: PHY_ADR: PHY 地址的低两位 (bit1, bit0), 而 PHY 地址的 bit[4:2] 强制为 000。如果要选择内部 PHY, 那么此 2 位强制为 01, 实际应用中要强制为 01。
- 5-0: EROA: EEPROM 字地址或 PHY 寄存器地址。

EPDRL/PHY_DRL (0DH) : EEPROM 或 PHY 数据寄存器低半字节 (EEPROM & PHY Low Byte Data Register)

7-0: EE_PHY_L

EPDRL/PHY_DRH (0EH) : EEPROM 或 PHY 数据寄存器高半字节 (EEPROM & PHY High Byte Data Register)

7-0: EE_PHY_H

WUCR (0FH) : 唤醒控制寄存器 (Wake Up Control Register)

- 7-6: 保留。
- 5: LINKEN: 1 使能“连接状态改变”唤醒事件。该位不受软件复位影响。
- 4: SAMPLEEN: 1 使能“Sample 帧”唤醒事件。该位不受软件复位影响。
- 3: MAGICEN: 1 使能“Magic Packet”唤醒事件。该位不受软件复位影响。
- 2: LINKST: 1 表示发生了连接改变事件和连接状态改变事件。该位不受软件复位影响。
- 1: SAMPLEST: 1 表示接收到“Sample 帧”和发生了“Sample 帧”事件。该位不受软件复位影响。
- 0: MAGICST: 1 表示接收到“Magic Packet”和发生了“Magic Packet”事件。该位不受软件复位影响。

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。

MAR (16H -- 1DH) : 多点发送地址寄存器 (Multicast Address Register)

7-0: MAB0 -- MAB7: 多点发送地址字节 0 -- 字节 7 (16H -- 1DH)。

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。GPIO0 默认为输出做 POWER_DOWN 功能。其它默认为输入。因此默认值为 0001。

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

7-4: 保留。

3-1: GEPI03-1: 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: GEPI00: 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。

命令

TRPAL (22H): 发送 SRAM 读指针地址低半字节 (TX SRAM Read Pointer Address Low Byte)

7-0: TRPAL

TRPAH (23H): 发送 SRAM 读指针地址高半字节 (TX SRAM Read Pointer Address High Byte)

7-0: TRPAH

RWPAL (24H): 接收 SRAM 指针地址低半字节 (RX SRAM Write Pointer Address Low Byte)

7-0: RWPAL

RWPAH (25H): 接收 SRAM 指针地址高半字节 (RX SRAM Write Pointer Address High Byte)

7-0: RWPAH

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。

CHIPR (2CH): 芯片修订版本 (CHIP Revision)

7-0: PIDH: 只读, 默认 00H。

TCR2 (2DH): 传输控制寄存器 2 (TX Control Register 2)

7: LED: LED 模式。1 设置 LED 引脚为模式 1, 0 设置 LED 引脚为模式 0 或根据 EEPROM 的设置。

6: RLCP: 1 重新发送有冲突延迟的数据包。

5: DTU: 1 禁止重新发送“underruned”数据包。

4: ONEPM: 单包模式。1 发送完成前发送一个数据包的命令能被执行, 0 发送完成前发送两个以上数据包的命令能被执行。

3-0: IFGS: 帧间间隔设置。0XXX 为 96bit, 1000 为 64bit, 1001 为 72bit, 1010 为 80bit, 1011 为 88bit, 1100 为 96bit, 1101 为 104bit, 1110 为 112bit, 1111 为 120bit。

OCR (2EH): 操作测试控制寄存器 (Operation Control Register)

7-6: SCC: 设置内部系统时钟。00 为 50MHz, 01 为 20MHz, 10 为 100MHz, 11 保留。

5: 保留。

4: SOE: 内部 SRAM 输出使能始终开启。

3: SCS: 内部 SRAM 片选始终开启。

2-0: PHYOP: 为测试用内部 PHY 操作模式。

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。

ETXCSR (30H) : 传输前 (Early) 控制、状态寄存器 (Early Transmit Control/Status Register)

7: ETE: 传输前使能。

6: ETS2: 传输前状态 2。

5: ETS1: 传输前状态 1。

4-2: 保留。

1-0: ETT: 传输前门限。当写到发送 FIFO 缓存里的数据字节数达到该门限, 则开始传输。
00 为 12.5%, 01 为 25%, 10 为 50%, 11 为 75%。

TCSCR (31H) : 传输校验和控制寄存器 (Transmit Check Sum Control Register)

7-3: 保留。

2: UDPCSE: UDP 校验和产生使能。

1: TCPCSE: TCP 校验和产生使能。

0: IPCSE: IP 校验和产生使能。

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: 接收校验和校验使能。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。

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。

MRCMDX1 (F1H) : 存储器读地址不变的读数据命令 (Memory Data Read Command With Address Increment Register)

同上。

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)。

MRRL (F4H): 存储器读地址寄存器低半字节 (Memory Data Read_ address Register Low Byte)

7-0: MDRAL

MRRH (F5H): 存储器读地址寄存器高半字节 Memory Data Read_ address Register High Byte

7-0: MDRAH: 若 IMR 的 bit7=1, 则该寄存器设置为 0CH。

MWCMDX (F6H): 存储器读地址不变的读数据命令 (Memory Data Write Command Without Address Increment Register)

7-0: MWCMDX: 写数据到发送 SRAM 中, 之后指向内部 SRAM 的写地址指针不变。

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)。

MWRL (FAH): 存储器写地址寄存器低半字节 (Memory Data Write_ address Register Low Byte)

7-0: MDRAL

MWRH (FBH): 存储器写地址寄存器高半字节 (Memory Data Write _ address Register High Byte)

7-0:MDRAH

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

ISR (FEH): 中断状态寄存器 (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, 11 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

IMR (FFH): 中断屏蔽寄存器 (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。

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。

BMCR (00H) : 基本模式控制寄存器 (Basic Mode Control Register)

- 15: reset: 1PHY 软件复位，0 正常操作。复位操作使 PHY 寄存器的值为默认值。复位操作完成后，该位自动清零。
- 14: loopback: 1Loop-back 使能，0 正常操作。
- 13: speed selection: 1 为 100Mbps，0 为 10Mbps。连接速度即可以根据该位选择，也可以根据第 12 位，即自动协商选择。当自动协商使能时，即第 12 位为 1，该位将会返回自动协商后的速度值。
- 12: auto-negotiation enable: 1 自动协商使能。使得第 13 位和第 8 位的值反应自动协商后的状态。
- 11: power down: POWER_DOWN 模式。1 为 POWER_DOWN，0 为正常操作。在 POWER_DOWN 状态下，PHY 应当响应操作处理。在转变到 POWER_DOWN 状态或已经运行在 POWER_DOWN 状态下时，PHY 不会在 MII 上产生虚假信号。
- 10: isolate: 1 除了一些操作外，PHY 将从 MII 中隔离，0 为正常操作。当该位置位，PHY 不会响应 TXD[3:0]，TX_EN 和 TX_ER 输入，并且在 TX_CLK，RX_CLK，RX_DV，RX_ER，RXD[3:0]，COL 和 CRS 输出上为高阻态。当 PHY 被隔离，则它将响应操作处理。
- 9: restart auto-aegotiation: 1 重新初始化自动协商协议，0 为正常操作。当第 12 位禁止该功能，则该位无效。初始化后该位自动清零。
- 8: duplex mode: 1 为全双工操作，0 为正常操作。当第 12 位被禁止（置 0）时该位被置位，若第 12 位被置位，则该位反应自动协商后的状态。
- 7: collision test: 1 为冲突测试使能，0 为正常操作。若该位置位，声明 TX_EN 将引起 COL 信号被声明。
- 6-0: 保留。

ANAR (04H) : 自动协商广告寄存器 (Auto-negotiation Advertisement Register)

- 15: NP: 0 表示无有效的下一页，1 表示下一页有效。PHY 没有下一页，所以该位始终为 0。
- 14: ACK: 1 表示连接对象数据接收认证，0 表示无认证。PHY 的自动协商状态机会自动控制该位。
- 13: RF: 1 表示本地设备处于错误状态，0 为无错误检验。

12-11: 保留。
10: FCS: 1 表示处理器支持溢出控制能力, 0 表示不支持。
9: T4: 1 表示本地设备支持 100BASE-T4, 0 表示不支持。PHY 不支持 100BASE-T4, 所以该位永远是 0。
8: TX_FDX: 1 为本地设备支持 100BASE-TX 全双工模式, 0 为不支持。
7: TX_HDX: 1 为本地设备支持 100BASE-TX, 0 为不支持。
6: 10_FDX: 1 为本地设备支持 100BASE-T 全双工模式, 0 为不支持。
5: 10_HDX: 1 为本地设备支持 100BASE-T, 0 为不支持。
4-0: selecter: 协议选择位, 00001 为默认值, 表示设备支持 IEEE802.3CSMA/CD, 不用修改。

DSCR (16H) : DAVIDCOM 详细配置寄存器 (DAVIDCOM Specified Configuration Register)

15: BP_4B5B: 1 为绕过 4B5B 编码和 5B4B 解码功能, 0 为正常 4B5B 和 5B4B 功能。
14: BP_SCR: 1 为绕过扰频和解扰功能, 0 为正常操作。
13: BP_ALIGN: 1 为绕过接收时的解扰、符号队列、解码功能和发送时的符号编码、扰频功能, 0 正常操作。
12: BP_ADPOK: 1 为强制信号探测功能使能, 0 为正常操作。该位仅为调试使用
11: 保留。
10: TX: 1 表示 100BASE-TX 操作, 0 保留。
9-8: 保留。
7: F_LINK_100: 0 为正常 100Mbps, 1 为强制 100Mbps 良好连接状态。
6-5: 保留, 强制为 0。
4: RPDCTR-EN: 1 为使能自动简化 POWER_DOWN, 0 为禁止。
3: SMRST: 1 为重新初始化 PHY 的状态机, 初始化后该位自动清零。
2: MFPSC: 1 表示 MII 帧引导抑制开启, 0 表示关闭。
1: SLEEP: 睡眠模式。该位置位将导致 PHY 进入睡眠模式, 通过将该位清零唤醒睡眠模式, 其中配置将还原为睡眠模式之前的状态, 但状态机将重新初始化。
0: RLOUT: 该位置位将使接收到的数据放入发送通道中。

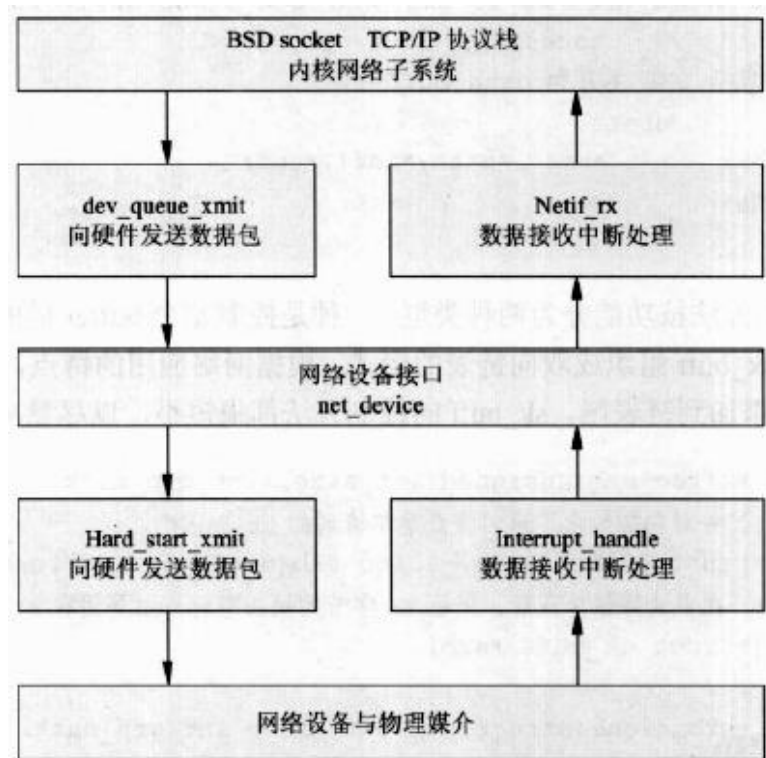
访问 PHY 寄存器的方法是:

- (1) 寄存器地址写到 EPAR/PHY_AR (0CH) 寄存器中, 注意将寄存器地址的第 6 位置 1 (地址与 0x40 或运算即可), 以表明写的是 PHY 地址, 而不是 EEPROM 地址。
- (2) 将数据高字节写到 PHY_DRH (0EH) 寄存器中。
- (3) 将数据低字节写到 PHY_DRL (0DH) 寄存器中。
- (4) 发送 PHY 命令 (0x0a) 到 EPCR/PHY_CR (0BH) 寄存器中。
- (5) 延时 5us, 发送命令 0x08 到 EPCR/PHY_CR (0BH) 寄存器中, 清除 PHY 写操作。

以上为 DM9000 (A) 常用寄存器功能的详细介绍, 通过对这些寄存器的操作访问, 我们便可以实现对 DM9000 的初始化、数据发送、接收等相关操作。而要实现 ARP、IP、TCP 等功能, 则需要对相关协议的理解, 由编写相关协议或移植协议栈来实现。

DM9000C 确定相异性

差异性 和 寄存器设置：



DM9000C 并不能使用内核提供的驱动，所以要从厂家提供的 DM9000C 的驱动上移植：

一，分析厂家提供的 DM9000C 网卡驱动程序：

```
#ifdef MODULE
... ..
int __init init_module(void)
void __exit cleanup_module(void)
... ..
#endif
```

若定义了宏“MODULE”才会使用到初始化“init_module”函数。

1，入口函数：

```
int __init init_module(void)
-->dmfe_dev = dmfe_probe(); probe函数
-->dev= alloc_etherdev(sizeof(struct board_info));分配一个net_device结构。
-->dmfe_probel(dev); 设置此结构
-->register_netdev(dev); 注册结构
```

2, dmfe_probel 函数：设置 硬件 。

```
int __init dmfe_probel(struct net_device *dev)
--> /* Search All DM9000 serial NIC */ 这个注释下是硬件相关的操作
    outb(DM9KS_VID_L, iobase);      iobase要自己设置
    id_val = inb(iobase + 4);        inb是读，这里是读这个网卡芯片
    outb(DM9KS_VID_H, iobase);
    id_val |= inb(iobase + 4) << 8;
    outb(DM9KS_PID_L, iobase);
    id_val |= inb(iobase + 4) << 16;
    outb(DM9KS_PID_H, iobase);
    id_val |= inb(iobase + 4) << 24;
```

网卡接口与内存接口一样，所以只需要去读某个地址即可。地址就表示网卡的某些寄存器。对里面寄存器的操作只有厂家才知道。所以网卡驱动程序厂家一般都会提供一个 demo。各个开发板的“iobase”基地址肯定不一样。中断引脚可能也不一样。

```
--> dev->hard_start_xmit      = &dmfe_start_xmit; 硬件启动传输函数。
--> dev->open                  = &dmfe_open; 使用网卡时会调用open函数
```

3, int dmfe_open(struct net_device *dev)

```
--> request_irq(dev->irq, &dmfe_interrupt, 0, dev->name, dev) 在open函数中注册中断。不同的开发板中断dev->irq也会不一样，这要自己设置。
```

下面是自己修改过的代码：

```
if (request_irq(dev->irq, &dmfe_interrupt, IRQF_TRIGGER_RISING, dev->name, dev))
```

对于 2440 来说可以配置此中断是“上升沿”触发还是“下降沿”触发。

在入口函数“int __init dm9000c_init(void)”中有修改过的：

```
volatile unsigned long *bwscon; // 0x48000000
volatile unsigned long *bankcon4; // 0x48000014
unsigned long val;

iobase = (int)ioremap(0x20000000, 1024); /* thisway.diy@163.com */
irq     = IRQ_EINT7;                    /* thisway.diy@163.com */
```

- 1, 具体开发板的基地址：下面显示物理基地址是“0x20000000”，接着 ioremap。
- 2, 也设置了中断号，此开发板的中断号是“IRQ_EINT7”

以上便为简单的厂家 DM9000C 驱动程序的分析。

二，从厂家提供的驱动程序上修改移植：

移植：找出差异性来修改。

DM9000 是一个内存接口的芯片，两个同样接有 DM9000 的开发板，最小差异：基地址、位宽和中断引脚。

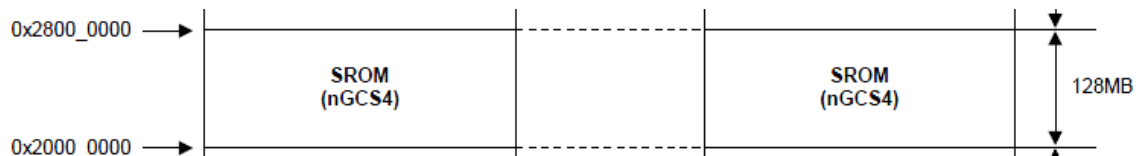
1，用什么地址访问此网卡芯片：

看片选引脚--nGCS4。

地址线和数据线上挂有网卡芯片、内存和 NORFLASH。如何访问一个网卡设备而不被其他诸如“内存”或“NORFLASH”影响，则是将每个设备选中再访问。

让“nGCS4”片选信号为低电平。

看 2440 芯片手册“MEMORY CONTROLLER”一章：



只要 CPU 发出的物理地址是“0x2000_0000 - 0x2800_0000”之间时，引脚“nGCS4”就会变成低电平。

(1). 接上板子上的外设，怎么确定它的地址？

1. CPU 发出一个地址 A
2. 存储控制器根据地址 A 的范围，决定让 nGCS0~nGCS7 中的哪个引脚输出低电平，假设是 nGCS3
3. 接在 nGCS3 的芯片就被选中
4. 访问这个被选中的芯片的哪个地址呢？由地址线 A26~A0 决定。
A26~A0 有 27 条地址线，寻址芯片是 128M

修改源代码：

```
iobase = (int)ioremap(0x20000000, 1024);
```

Ioremap 0x20000000，是因为基地址就是 0x20000000。

让片选引脚是低电平，CPU 发出来的地址必须是位于“0x2000_0000 - 0x2800_0000”之间。所以上面就把基地址设置为“0x2000_0000”。

2，入口函数 与 出口函数：

原来的入口与出口函数在一个“#ifden”下面：

```

#ifdef MODULE

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Davicom DM9000/DM9010 ISA/uP Fast Ethernet Driver");
#ifdef LINUX_VERSION_CODE < KERNEL_VERSION(2,5,0)
MODULE_PARM(mode, "i");
MODULE_PARM(irq, "i");
MODULE_PARM(iobase, "i");
#else
module_param(mode, int, 0);
module_param(irq, int, 0);
module_param(iobase, int, 0);
#endif
MODULE_PARM_DESC(mode, "Media Speed, 0:10MHD, 1:10MFD, 4:100MHD, 5:100MFD");
MODULE_PARM_DESC(irq, "EtherLink IRQ number");
MODULE_PARM_DESC(iobase, "EtherLink I/O base address");

```

```

int __init init_module(void)
{
    switch(mode) {
        case DM9KS_10MHD:
        case DM9KS_100MHD:
        case DM9KS_10MFD:
        case DM9KS_100MFD:
            media_mode = mode;
            break;
        default:
            media_mode = DM9KS_AUTO;
    }
    dmfe_dev = dmfe_probe();
    if(IS_ERR(dmfe_dev))
        return PTR_ERR(dmfe_dev);
    return 0;
}

```




```

void __exit cleanup_module(void)
{
    struct net_device *dev = dmfe_dev;
    DMFE_DEBUG(0, "clean_module()", 0);

    unregister_netdev(dmfe_dev);
    release_region(dev->base_addr, 2);
    #if LINUX_VERSION_CODE < KERNEL_VERSION(2,5,0)
        kfree(dev);
    #else
        free_netdev(dev);
    #endif

    DMFE_DEBUG(0, "clean_module() exit", 0);
}
#endif

```

移植修改：

去掉 #ifden。。。#endif；将 init_module() 改为 “dm9000c_init()” 入口函数。

将出口函数 “cleanup_module()” 改成 “dm9000c_exit()”，再加上修饰函数。如下：

```

//#ifdef MODULE
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Davicom DM9000/DM9010 ISA/uP Fast Ethernet Driver");
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,5,0)
MODULE_PARM(mode, "i");
MODULE_PARM(irq, "i");
MODULE_PARM(iobase, "i");
#else
module_param(mode, int, 0);
module_param(irq, int, 0);
module_param(iobase, int, 0);
#endif
MODULE_PARM_DESC(mode, "Media Speed, 0:10MHD, 1:10MFD, 4:100MHD, 5:100MFD");
MODULE_PARM_DESC(irq, "EtherLink IRQ number");
MODULE_PARM_DESC(iobase, "EtherLink I/O base address");

/* Description:
   when user used insmod to add module, system invoked init_module()
   to initilize and register.
*/
int __init dm9000c_init(void)
{
    switch(mode) {
    case DM9KS_10MHD:
    case DM9KS_100MHD:

```



```

case DM9KS_10MFD:
case DM9KS_100MFD:
media_mode = mode;
break;
default:
media_mode = DM9KS_AUTO;
}

dmfe_dev = dmfe_probe();
if(IS_ERR(dmfe_dev))
return PTR_ERR(dmfe_dev);
return 0;
}

/* Description:
   when user used rmmod to delete module, system invoked clean_module()
   to un-register DEVICE.
*/
void __exit dm9000c_exit(void)
{
struct net_device *dev = dmfe_dev;
DMFE_DEBUG(0, "clean_module()", 0);

unregister_netdev(dmfe_dev);
release_region(dev->base_addr, 2);
#if LINUX_VERSION_CODE < KERNEL_VERSION(2, 5, 0)
kfree(dev);
#else
free_netdev(dev);
#endif

DMFE_DEBUG(0, "clean_module() exit", 0);
}

module_init(dm9000c_init);
module_exit(dm9000c_exit);
//#endif

```

3, 查看入口函数中的“probe”函数:

找到设置结构体的函数。

```

int __init dm9000c_init(void)
-->dmfe_dev = dmfe_probe();
   -->struct net_device *dev = alloc_etherdev(sizeof(struct board_info)); 分配结构体
   -->dmfe_probel(dev); 设置
   -->register_netdev(dev); 注册

int __init dmfe_probel(struct net_device *dev)

```

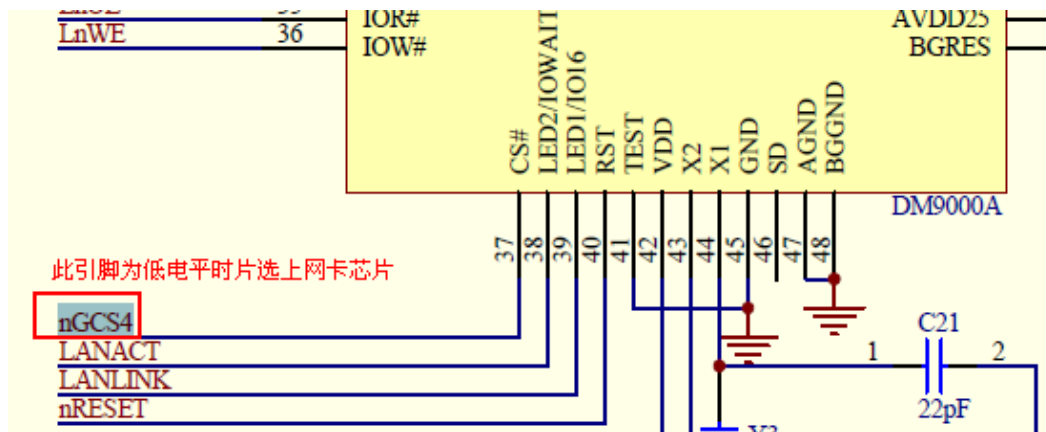
-->outb(DM9KS_VID_L, iobase); 往iobase地址处写入一个值“DM9KS_VID_L”。DM9000C的索引寄存器(cmd引脚为0)

```
id_val = inb(iobase + 4); 从iobase+4处读出某个值。读DM9000C的数据寄存器(cmd引脚为1)
outb(DM9KS_VID_H, iobase);
id_val |= inb(iobase + 4) << 8;
outb(DM9KS_PID_L, iobase);
id_val |= inb(iobase + 4) << 16;
outb(DM9KS_PID_H, iobase);
id_val |= inb(iobase + 4) << 24;
```

上面要设置“iobase”，这样就能outb(),和inb()能访问。

①，差异性：基地址。要设置 iobase：

当只要 CPU 发出的物理地址是“0x2000_0000 - 0x2800_0000”之间时，引脚“(nGCS4)”就会变成低电平。所以基地址根据这个物理地址“0x2000_0000 - 0x2800_0000”设置而来。



从上面的硬件设置中，发现有一个iobase，厂家驱动设置如下：

```
#define DM9KS_MIN_IO (MST_ETH_PHYS + 0x300)

static int iobase = DM9KS_MIN_IO;
```

最后要看宏“MST_ETH_PHYS”。

下面修改“基地址”：在“入口函数”中设置。

```
iobase = (int)ioremap(0x20000000, 1024);
```

大小的设置有些随意性，就是看用到了多少地址。这里用的不是很多，就直接把大小设置成了1M。因为iobase定义的是int型，ioremap()后要强制转换下，不然编译时会有一个警告：

```
dm9dev9000c.c:1641: warning: assignment makes integer from pointer without a cast
```

有ioremap(), 则在出口函数中要: iounmap((void *)iobase);

写成“iounmap(iobase)”时，也会有警告：需要类型转换

```
/work/drivers_and_test/17th_dm9000c/dm9dev9000c.c: In function 'dm9000c_exit':
/work/drivers_and_test/17th_dm9000c/dm9dev9000c.c:1675: warning: passing arg 1 of 'iounmap' makes pointer from integer without a cast
```

②，关于芯片版本：

```
if((db->chip_revision!=0x1A) || ((chip_info&(1<<5))!=0) || ((chip_info&(1<<2))!=1)) return -ENODEV;
```

若芯片版本不是“0x1A”等“chip_info”条件不满足时就返回一个错误。实际中的开发板网卡不适合这个条件，但注释掉也能使用此驱动，所以这里直接注释去。

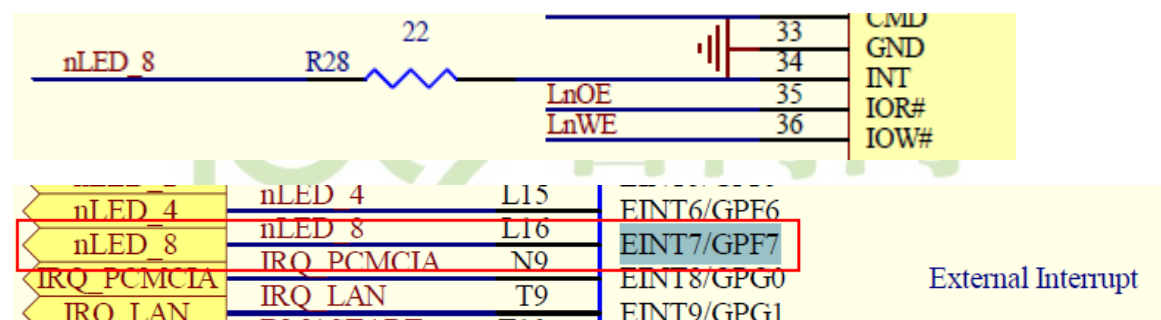
```
//if((db->chip_revision!=0x1A) || ((chip_info&(1<<5))!=0) ||  
((chip_info&(1<<2))!=1)) return -ENODEV;
```

在实际的移植中，因为此句的条件判断而返回时，在不同的地址加打印语句来判断的出错的地方而发现此句要修改。

③，中断号：

网卡芯片接收到数据后，会产生一个中断（网卡芯片里面也会有内存，因为它接收到数据后是先放在自己的内存里，然后再产生一个中断）。在“中断服务程序”里再将此缓存的数据从网卡芯片里拿出来构造成一个“skb_buff”，然后才提交上去（网络7层模型的其他层）。发送时也是从“skb_buff”里拿出数据交给网卡芯片。网卡芯片硬件就会再把数据从skb_buff中取出发送出去，发送完成后就会产生一个中断。

看原理图：DM9000C 与 2440 的连接引脚“INT”

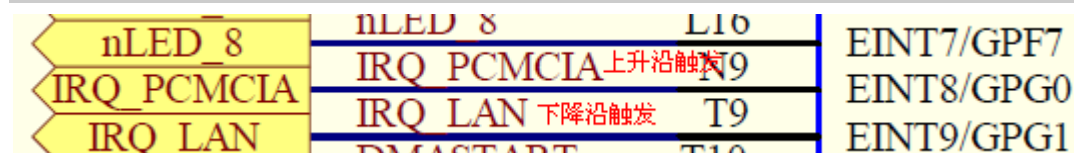


原理图上，DM9000C 的“INT”引脚最终接到了 2440 的外部中断 7 上“EINT7/GPF7”。
查找到中断号后，在入口函数中指定此中断号。

```
irq = IRQ_EINT7;
```

④，dev->open = &dmfe_open；里面有使用中断。

```
-->if (request_irq(dev->irq,&dmfe_interrupt, IRQF_TRIGGER_RISING,dev->name,dev)) return -  
EAGAIN;
```



对于 2440 来说，中断可以设置成“上升沿触发”或“下降沿触发”。可以参考以前“按键中断”驱动中的代码：

```
static struct file_operations sencod_drv_fops = {
    .owner    = THIS_MODULE, /* 这是一个宏，推向
    .open     = sixth_drv_open,
    .read     = sixth_drv_read,
    .release  = sixth_drv_close,
    .poll     = sixth_drv_poll,
    .fasync   = sixth_drv_fasync,
};
```

```
/* 配置GPF0, 2为输入引脚 */
/* 配置GPG3, 11为输入引脚 */
request_irq(IRQ_EINT0, buttons_irq, IRQT_BOTHEDGE, "S2", &pins_desc[0]);
request_irq(IRQ_EINT2, buttons_irq, IRQT_BOTHEDGE, "S3", &pins_desc[1]);
request_irq(IRQ_EINT11, buttons_irq, IRQT_BOTHEDGE, "S4", &pins_desc[2]);
request_irq(IRQ_EINT19, buttons_irq, IRQT_BOTHEDGE, "S5", &pins_desc[3]);
```

```
#define IRQT_RISING ( __IRQT_RISEEDGE)
#define IRQT_FALLING ( __IRQT_FALEEDGE)
#define IRQT_BOTHEDGE ( __IRQT_RISEEDGE | __IRQT_FALEEDGE)
```

IRQT_RISING 上升沿触发。这个宏是“#define IRQ_TYPE_EDGE_RISING 0x00000001 /* Edge rising type */”

```
int request_irq(unsigned int irq, irq_handler_t handler,
unsigned long irqflags, const char *devname, void *dev_id)
```

参 3 “irqflags” 就是设置“上升沿”或是“下降沿”。参考按键驱动的“上升沿”，在网卡驱动里我们也设置成“上升沿”触发。

```
#define IRQF_TRIGGER_RISING 0x00000001
#define IRQF_TRIGGER_FALLING 0x00000002
```

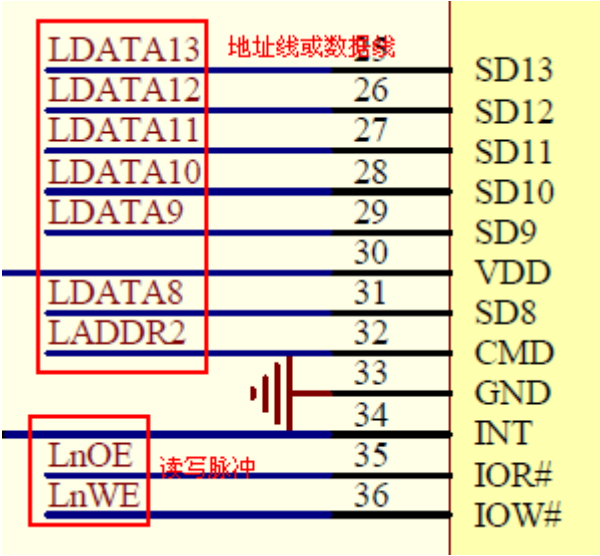
以上设置了差异性的部分：iobase 基地址，中断号。

4，在完成基地址，中断号的设置后，直接在厂家驱动的基础上编译驱动成功。

```
book@book-desktop:/work/drivers_and_test/17th_dm9000c$ make
make -C /work/system/linux-2.6.22.6 M='pwd' modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
CC [M] /work/drivers_and_test/17th_dm9000c/dm9dev9000c.o
Building modules. stage 2.
MODPOST 1 modules
LD [M] /work/drivers_and_test/17th_dm9000c/dm9dev9000c.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/17th_dm9000c$
```

编译通过。但这个驱动程序还是有一个缺点，就是各信号间时序的设置。但这里没有设置前，也能使用，是因为 UBOOT 中设置好了。要是想写一个不依赖于 UBOOT 的网卡驱动程序，就得自己设置。

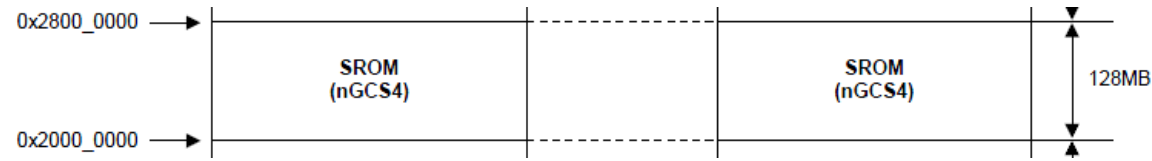
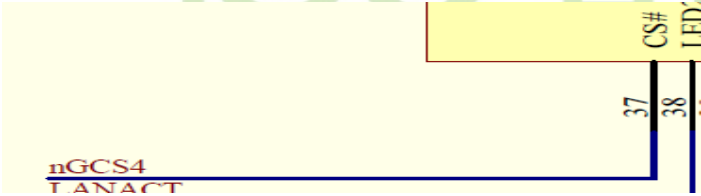
设置各信号间的时间：时序。



网卡是内存一样的接口，地址信号，读写脉冲，这些信号的读写时序会有要求。
CPU 发指令给内存控制器，再由“内存控制器”负责发读写、地址等信号给网卡芯片，这些信号间的时间要设置。所以还要设置一项“MEMORY CONTROLLER”。

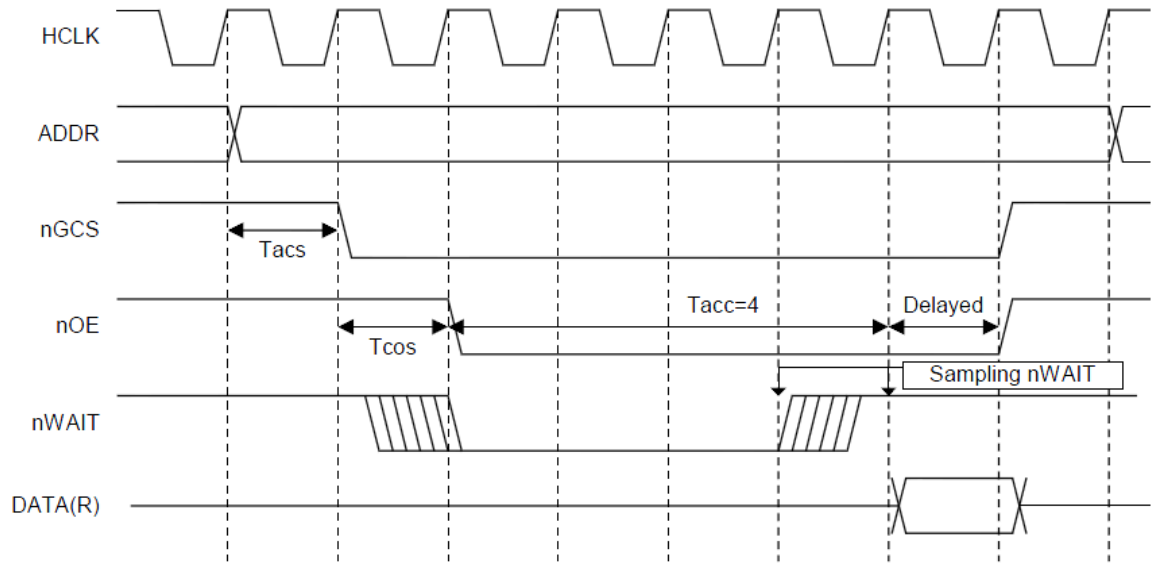
设置 S3C2440 的 memory controller:

①，此开发板的网卡是片选 4:



②，nWAIT PIN OPERATION:

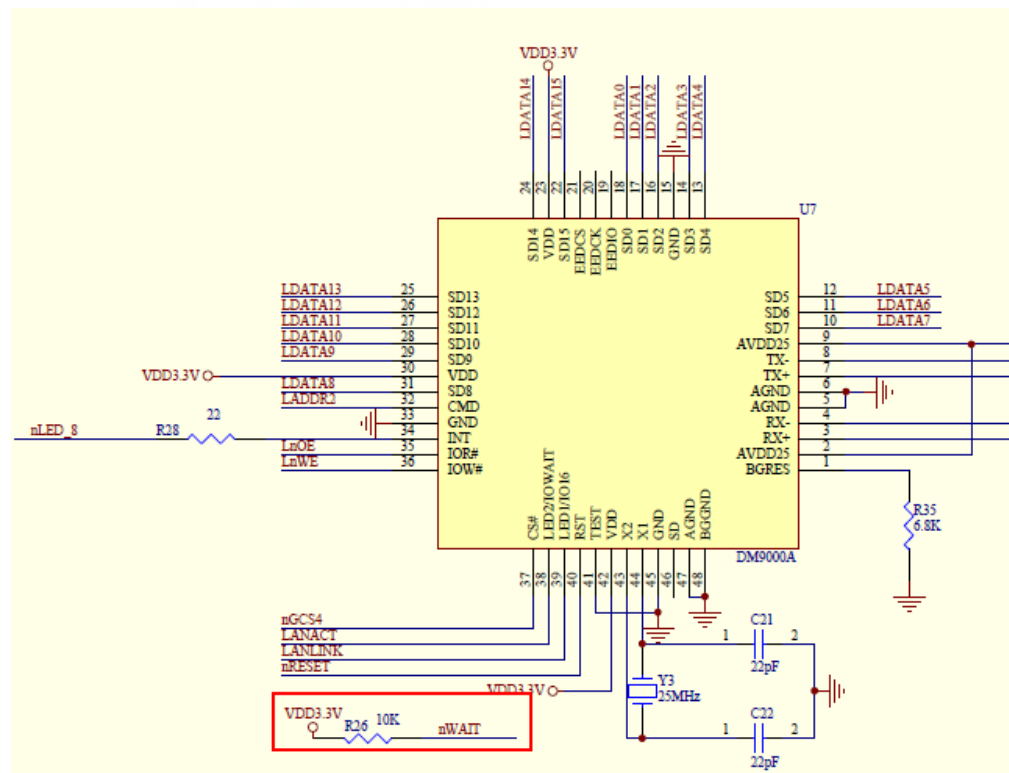
等待信号的时序。但最终 DM9000 的原理图上没有此信号，就不用设置。



S3C2440A External nWAIT Timing Diagram (Tacc=4)

等待信号：

当 CPU 访问网卡芯片时，若此时 DM9000 还没准备就绪，这时网卡就可以给 CPU 发出一个等待信号（nWAIT）。从原理图上看 DM9000 没有“nWAIT”等待信号。



③，BUS WIDTH & WAIT CONTROL REGISTER (BWSCON)：

总线位宽 和 等待寄存器。BWSCON 寄存器主要用来设置外接存储器的总线宽度和等待状态。

a. STx：启动/禁止 SDRAM 的数据掩码引脚，对于 SDRAM，此位为 0；对于 SRAM，此位为 1。

- b. WSx: 是否使用存储器的 WAIT 信号, 通常设为 0
- c. DWx: 使用两位来设置存储器的位宽: 00-8 位, 01-16 位, 10-32 位, 11-保留。
- d. 比较特殊的是 BANK0 对应的 4 位, 它们由硬件跳线决定, 只读。

Register	Address	R/W	Description	Reset Value
BWSCON	0x48000000	R/W	Bus width & wait status control register	0x000000

因为使用的是片选 4, 所以设置的位域如下:

ST4	[19]	Determines SRAM for using UB/LB for bank 4. 0 = Not using UB/LB (The pins are dedicated nWBE[3:0]) 1 = Using UB/LB (The pins are dedicated nBE[3:0])	0
WS4	[18]	Determines WAIT status for bank 4. 0 = WAIT disable 1 = WAIT enable	0
DW4	[17:16]	Determine data bus width for bank 4. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	0

DW4 的描述为 BANK4 的带宽, DM9000 接了 16 条地址线, 那么带宽就是 16, 这里选 01
WS4 的描述为是否为 BANK4 使用等待状态, DM9000 没有接 WAIT 引脚, 所以可以不管这个字段
ST4 的描述为是否为 BANK4 使用 UB/LB(写高/低字节使能), DM9000 没有接 nWBE[3:0] 这 4 个引脚, 所以也不管这个字段。

设置位宽 DW4 bit[17:16]:

看原理图上 DM9000 使用了多少条数据线。看上面的 DM9000 原理图, 数据线是 “LDATA0 ~ 15” 一共 16 条

即 16 位。

设置等待 WS4 bit[18]: 不等待。

设置功能引脚 ST4 bit[19]: 这个不很了解, 就直接按默认写 “0”。

在入口函数中设置位宽:

1, 定义寄存器:

```
volatile unsigned long *bwscon;           // 0x48000000 此寄存器设置位宽
```

2, ioremap().

```
bwscon = ioremap(0x48000000, 4);
```

3, 设置位宽为 16 bit

```
/* DW4[17:16]: 01-16bit 需要将DW4位域指定为16bit.
 * WS4[18] : 0-WAIT 即不使用等待信号
 * ST4[19] : 0 这个引脚没用到。
 */
val = *bwscon;           //先读出这个寄存器的值。
val &= ~(0xf<<16);       //再清零. 从bit16 - bit19
val |= (1<<16);          //再将bit16设置为1. 即1左右16位。
*bwscon = val;
```


DM9000C 设置时序

DM9000 时序：

位宽&等待状态寄存器 BWSCON (BUS WIDTH&WAIT STATUS CONTROL REGISTER)，每四位控制一个 memory bank，根据 S3C2440 手册，涉及 dm9000 的 4 个引脚是 16~19。默认值是 0x000000。

1. ST4[19]：SRAM 是否使用 UB/LB(upper byte/lower byte)。

0：不使用 UB/LB；1：使用 UB/LB。

2. WS4[18]：WAIT 状态。

0：禁止 WAIT；1：使能 WAIT。

3. DW4[16:17]：数据总线宽度。

00：8 位；01：16 位；10：32 位；11：保留。

接着，BANK4 控制寄存器 BANKCON4 (BANK CONTROL REGISTER)，用于控制 BANK4 外接设备的访问时序。默认值是 0x0700。

设置 BANK4 寄存器

1. Tacs[14:13]：地址建立时间。

00：0 时钟周期；01：1 时钟周期；10：2 时钟周期；11：4 时钟周期。

2. Tcos[12:11]：

片选建立时间。00：0 时钟周期；01：1 时钟周期；10：2 时钟周期；11：4 时钟周期。

3. Tacc[10:8]：地址周期。

000：1 时钟周期；001：2 时钟周期；... 111：14 时钟周期。

4. Tcoh[7:6]：片选保持时间。

00：0 时钟周期；01：1 时钟周期；10：2 时钟周期；11：4 时钟周期。

5. Tcah[5:4]: 地址保持时间。
00: 0 时钟周期; 01: 1 时钟周期; 10: 2 时钟周期; 11: 4 时钟周期。
6. Tcap[3:2]: 页模式存取周期。
00: 2 时钟周期; 01: 3 时钟周期; 10: 4 时钟周期; 11: 6 时钟周期。
7. PMC[1:0]: 页模式配置。
00: 1 data; 01: 4 data; 10: 8 data; 11: 16 data。

最后，在 arch/arm/mach-s3c2410/include/mach/regs-mem.h 中，

```
#define S3C2410_BWSCON_DW4_16    (1<<16)
#define S3C2410_BWSCON_WS4      (1<<18)
#define S3C2410_BWSCON_ST4      (1<<19)
```

理解了寄存器 BWSCON 和 BANKCON4 的含义之后，上述代码的作用就很明显了。

```
*((volatile unsigned int *)S3C2410_BWSCON) =
    (oldval_bwscon & ~(3<<16)) | S3C2410_BWSCON_DW4_16 | S3C2410_BWSCON_WS4 |
    S3C2410_BWSCON_ST4;// 数据总线宽度16bits，使用UB/LB，使能WAIT
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x1f7c; //修改了访问dm9000的时序
```

看“时间参数”：

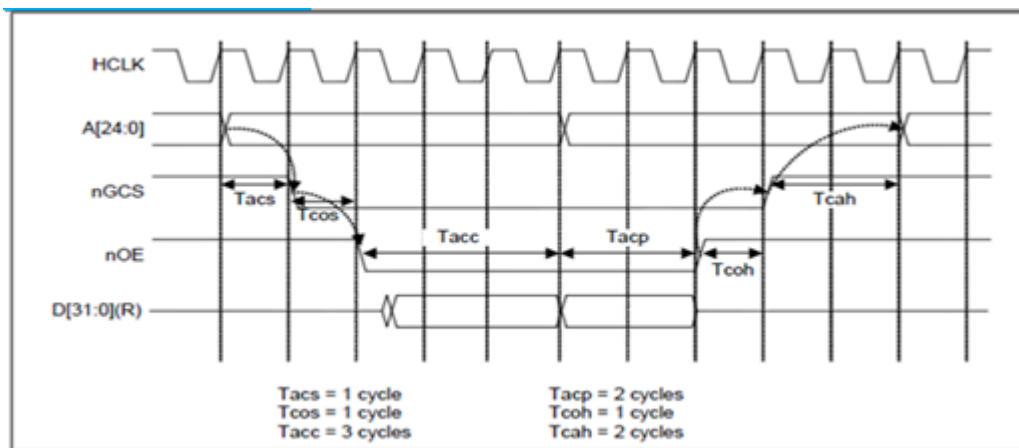
BANKCON4	0x48000014	R/W	Bank 4 control register	0x0700
----------	------------	-----	-------------------------	--------

这些值如何设置，只能看 DM9000 的芯片手册。若不想看，则可以使用上面 bit[0-14] 中每个位域的最大值。如 PMC bit[1:0]就使用“11=16data”这个值。用起来之后，你想让 DM9000 的访问速度更快时，就得去看 DM9000 的芯片手册来修改上面的值。

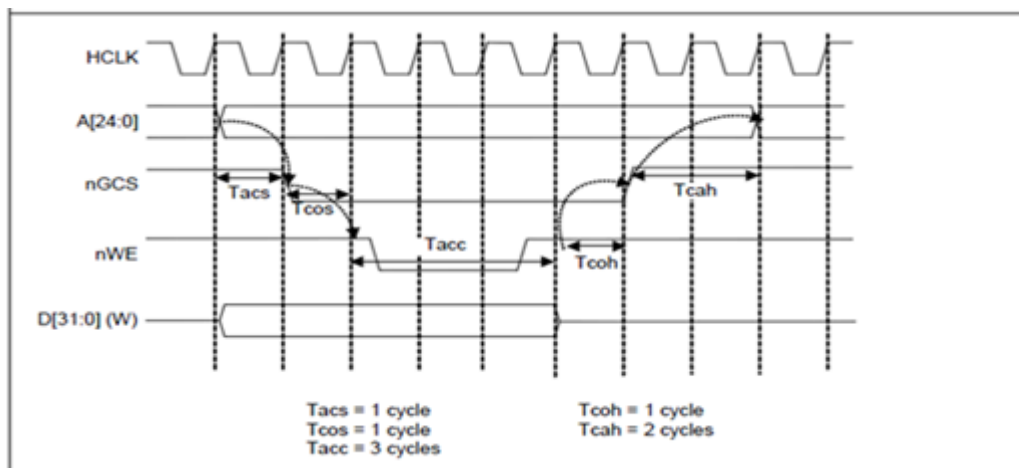
设置这个时间值的过程，先找到寄存器的位域定义，再找到 2440 上这些时间的时序图，按照时序图知道它们的关系。然后再找到 DM9000 的芯片手册看芯片上的规定读写时序，两两时序对比计算，得出这些寄存器位域值的设置结果。

2440, PROGRAMMABLE ACCESS CYCLE

读时序:



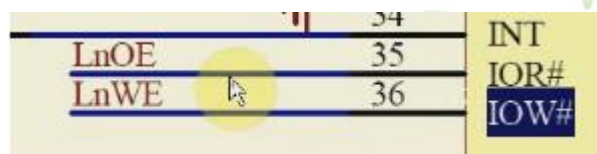
写时序:



下面是要设置的位域 S3C2440A nGCS Timing Diagram

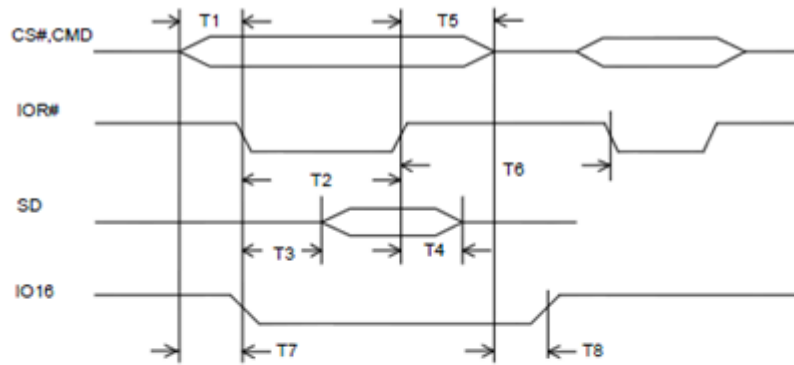
BANKCONn	Bit	Description	Initial State
Tacs	[14:13]	Address set-up time before nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tcos	[12:11]	Chip selection set-up time before nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacc	[10:8]	Access cycle 000 = 1 clock 001 = 2 clocks 010 = 3 clocks 011 = 4 clocks 100 = 6 clocks 101 = 8 clocks 110 = 10 clocks 111 = 14 clocks Note: When nWAIT signal is used, Tacc ≥ 4 clocks.	111
Tcoh	[7:6]	Chip selection hold time after nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	000
Tcah	[5:4]	Address hold time after nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacp	[3:2]	Page mode access cycle @ Page mode 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = 6 clocks	00
PMC	[1:0]	Page mode configuration 00 = normal (1 data) 01 = 4 data 10 = 8 data 11 = 16 data	00

DM9000: 读时序



原理图上，读写引脚连接的 DM9000 上的 IOR# 和 IOW#

10.3.4 Processor I/O Read Timing

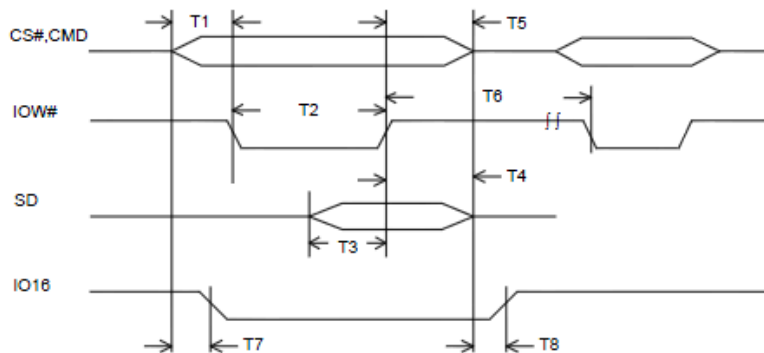


Symbol	Parameter	Min.	Typ.	Max.	Unit
T1	CS#,CMD valid to IOR# valid	0			ns
T2	IOR# width	10			ns
T3	System Data(SD) Delay time			19	ns *2
T4	IOR# invalid to System Data(SD) invalid			6	ns
T5	IOR# invalid to CS#,CMD invalid	0			ns
T6	IOR# invalid to next IOR#/IOW# valid When read DM9000C register	2			clk*
T2+T6	IOR# valid to next IOR#/IOW# valid When read DM9000C memory with F0h register	4			clk*
T2+T6	IOR# valid to next IOR#/IOW# valid When read DM9000C memory with F2h register	1			clk*
T7	CS#,CMD valid to IO16 valid			5	ns
T8	CS#,CMD invalid to IO16 invalid			5	ns



DM9000 写信号 时序:

10.3.5 Processor I/O Write Timing



Symbol	Parameter	Min.	Typ.	Max.	Unit
T1	CS#,CMD valid to IOW# valid	0			ns
T2	IOW# Width	10			ns
T3	System Data(SD) Setup Time	10			ns
T4	System Data(SD) Hold Time	3			ns
T5	IOW# Invalid to CS#,CMD Invalid	0			ns
T6	IOW# Invalid to next IOW#/IOR# valid When write DM9000C INDEX port	1			clk*
T6	IOW# Invalid to next IOW#/IOR# valid When write DM9000C DATA port	2			clk*
T2+T6	IOW# valid to next IOW#/IOR# valid When write DM9000C memory	1			clk*
T7	CS#,CMD Valid to IO16 valid			3	ns
T8	CS#,CMD Invalid to IO16 Invalid			3	ns

Tacs: acs 是地址设置时间。

从时序图上看意思是说，地址信号必须稳定之后再过多少时间，才会发出这个“片选nGCS”信号。

Tacs	[14:13]	Address set-up time before nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
-------------	----------------	--	-----------

LADDR2: 原理图上只用了一个引脚 LADDR2 表示，是命令或是其他。原画较上除此外都是数据引脚。

CMD 引脚用于设置 COMMAND 模式, CMD 为高时, 选择数据端口。CMD 为低时, 选地址端口。数据端口和地址端口的地址码由下式决定:

DM9000 地址端口=高位片选地址+300H+0

DM9000 数据端口=高位片选地址+300H+4H

其中, 高位片选地址由 S3C2410 的 NGCS3 提供, 即为: 0X10000000H。

驱动编写特点:

采用一个 CMD 信号来控制是对 DM9000 读还是写, 架构非常简单, 容易理解. 有点类似 CS8900A 的 PacketPage 结构。

DM9000 的操作: 从源代码上分析:

```
dmfe_dev = dmfe_probe();
-->err = dmfe_probe1(dev);
    -->outb(DM9KS_VID_L, iobase); /* DM9000C的索引寄存器(cmd引脚为0) */
    id_val = inb(iobase + 4); /* 读DM9000C的数据寄存器(cmd引脚为1) */
    outb(DM9KS_VID_H, iobase);
    id_val |= inb(iobase + 4) << 8;
    outb(DM9KS_PID_L, iobase);
    id_val |= inb(iobase + 4) << 16;
    outb(DM9KS_PID_H, iobase);
    id_val |= inb(iobase + 4) << 24;
```

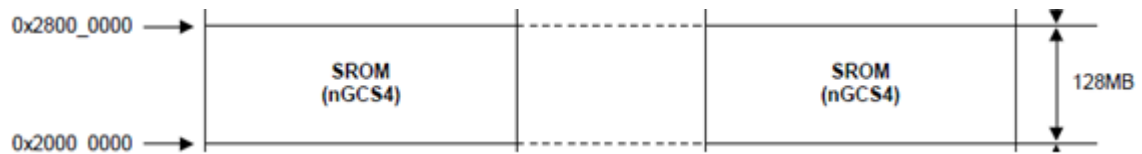
分析:

DM9000C 读数据: 先要把地址值写给某个寄存器, 再去读所谓的“数据寄存器”。

想读它的数据, 就先把地址写到“索引寄存器”里面, 再去读所谓的“数据寄存器”。

①, outb(DM9KS_VID_L, iobase);

先把一个值“DM9KS_VID_L (寄存器地址)”写到“iobase”基地址里面去。这个基地址就是:



想读它的数据，就先把地址写到“索引寄存器”里面，再去读所谓的“数据寄存器”。
那么：

因为，接上板子上的外设，怎么确定它的地址？

1. CPU 发出一个地址 A
2. 存储控制器根据地址 A 的范围，决定让 nGCS0~nGCS7 中的哪个引脚输出低电平，假设是 nGCS3
3. 接在 nGCS3 的芯片就被选中
4. 访问这个被选中的芯片的哪个地址呢？由地址线 A26~A0 决定。

A26~A0 有 27 条地址线，寻址芯片是 128M

就是说 CPU 对基地址“0x2000_0000 - 0x2800_0000”之间时发的都是低电平，因为从原理图上看 DM9000 接的是片选“nGCS4”，nGCS4 被选中时，这个区间的地址都是低电平“0”，则这个“LADDR2 (CMD 引脚)”就表示为“0”。

DM9000 的访问比较有意思，它只有 2 个地址。

一个地址称为命令端口，另一个地址称为数据端口

+4，就是让 addr2 为 1

其实不一定要这两个地址

只要一个 addr2 为 0，另一个 addr2 为 1 就够了

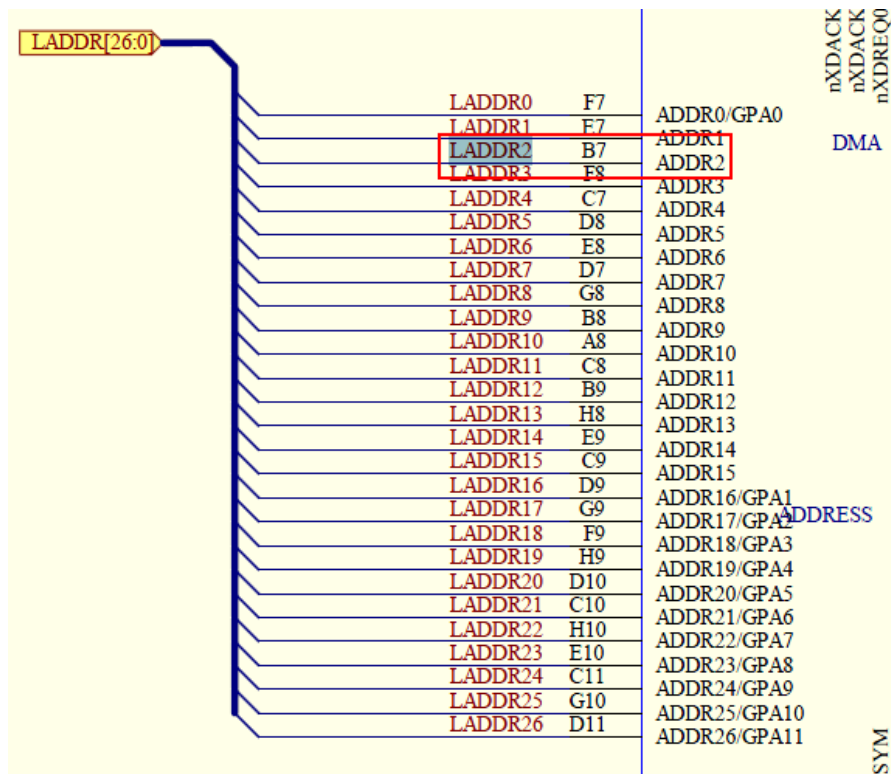
如果用 addr3 就是+8

假设想写 DM90000 里地址为 X 的寄存器，把数据 Y 给它，怎么办？

1. 把 X 作为数据写入 命令端口
2. 把 Y 作为数据写入 数据端口

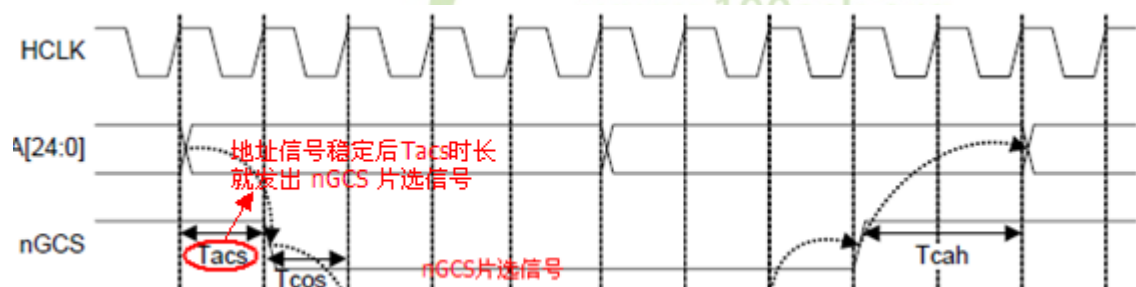
②, id_val = inb(iobase + 4);

这里是指从基地址读一个值，是读地址值加 4。4 就是 bit0~bit3.

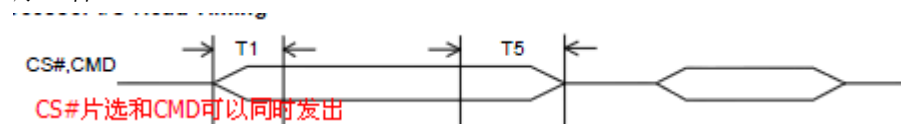


4 就是指“LADDR2(CMD)”引脚，读这个“iobase+4”就是这里“LADDR2”引脚等于“1”高电平。

2440 上的地址信号与片选信号时序时长间隔“Tacs”：

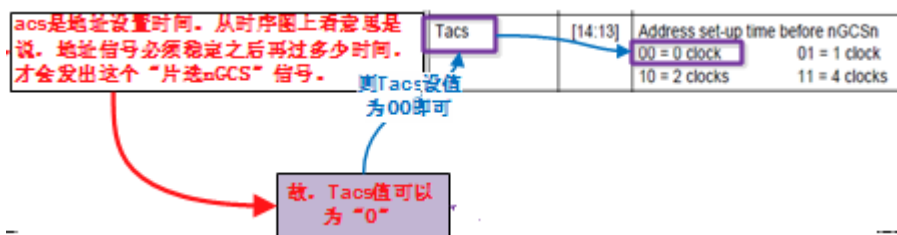


DM9000 芯片片选“CS#”和“CMD/地址”信号时序是显示为可以同时发出：（读和写时序一样）



没有要求说是先发出 CMD 信号后，再发出“片选”。所以 CMD 信号和片选信号可以同时发出，则 Tacs 这个值可以等于 0。

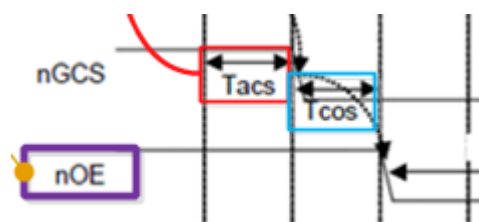
则 2440 “Tacs” 位域取值可以为“00”。



2, Tcos : “nOE” 是读信号，

这里的意思是说，片选信号 nGCS 发出多久之后，再发出“读”信号。

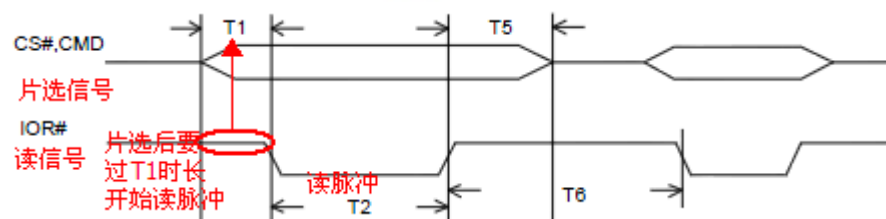
Tcos	[12:11]	Chip selection set-up time before nOE	00
		00 = 0 clock	01 = 1 clock
		10 = 2 clocks	11 = 4 clocks



上面就是 nGCS 信号低电平 Tcos 时间后，才发出“nOE”读信号低脉冲。原理图的读写引脚如下：

LnOE	35	INT IOR# IOW#
LnWE	36	

读“LnOE”对应的 DM9000 芯片引脚是“IOR#”，那么这时就去 DM9000 芯片手册上搜索此引脚的时序相关内容为：



从 T1 的最小取值为“0ns”可知：

Symbol	Parameter	Min	Typ.	Max.	Unit
T1	CS#,CMD valid to IOR# valid	0			ns
T1	CS#,CMD valid to IOW# valid	0			ns

CS#片选信号和 IOR#可以同时发出，则：Tcos 位域可以取值为 00。

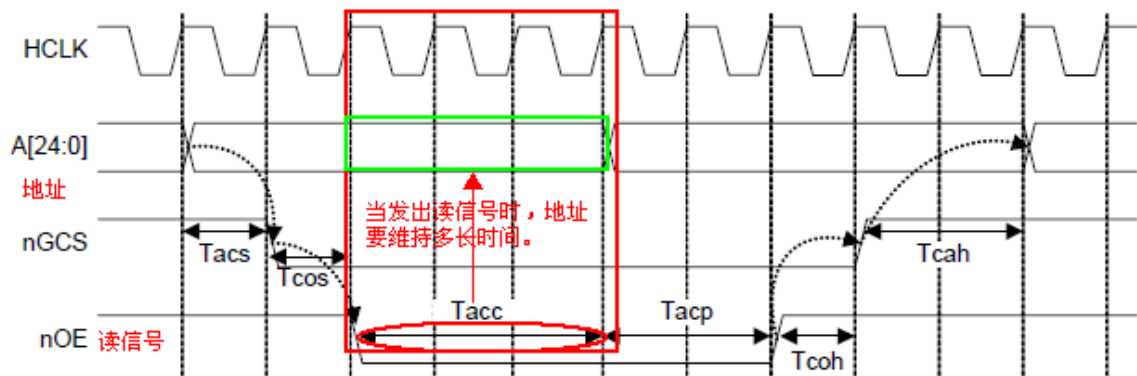
结果是：发出片选信号后，过 0ns 就可以发出读信号。即可同时发出。	Tcos	[12:11]	Chip selection set-up time before nOE	00
			00 = 0 clock	01 = 1 clock
			10 = 2 clocks	11 = 4 clocks

3, Tacc : 访问周期(地址周期)。

当发出读信号后，地址要维持多长时间。(读写信号的脉冲长度，)

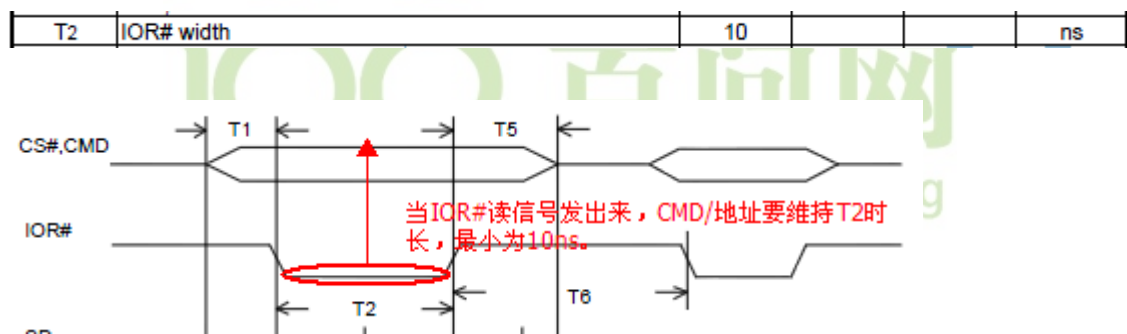
Tacc	[10:8]	Access cycle 000 = 1 clock 010 = 3 clocks 100 = 6 clocks 110 = 10 clocks Note: When nWAIT signal is used, Tacc ≥ 4 clocks.	001 = 2 clocks 011 = 4 clocks 101 = 8 clocks 111 = 14 clocks	111
------	--------	---	---	-----

2440 上时序:

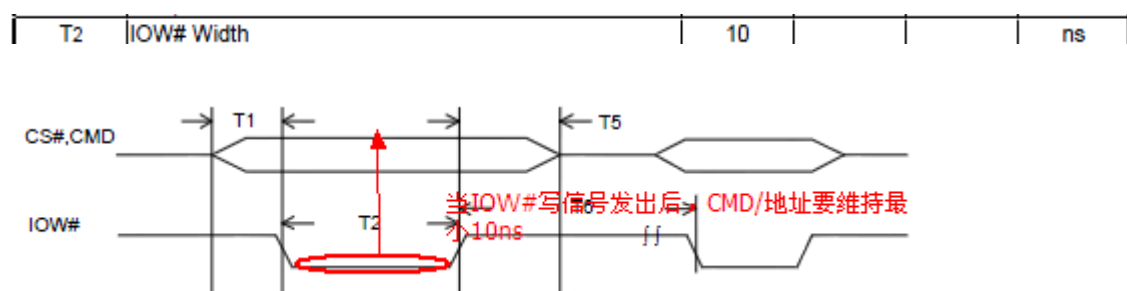


DM9000 上时序:

读:

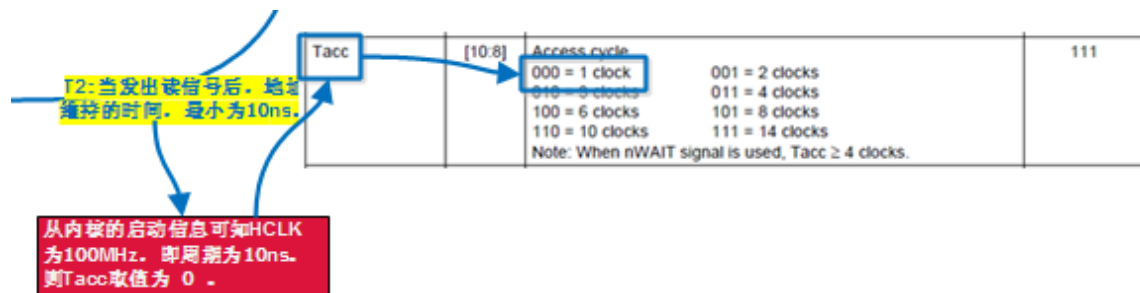


写:



则是读写脉冲要大于等于 10ns.

HCLK 是 100MHz. 即一个周期是 10ns. 则这个“Tacc”可以取为 0。Tacc=0 就表示一个周期。



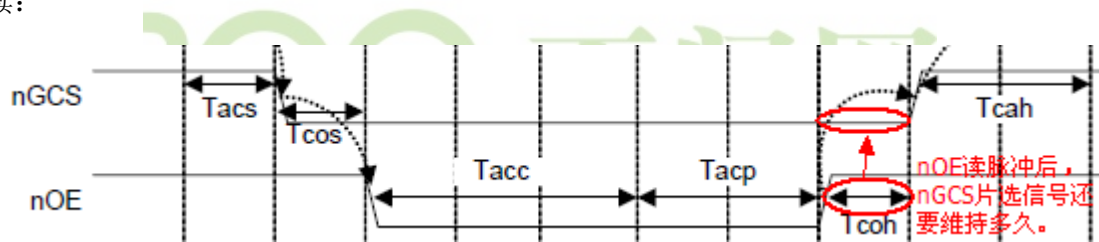
若觉得这个 Tacc=0 的取值太临界, 那么就取它为“1”, 因为 T2 的最小值为 10ns, 当 T2=10ns 时即 1 个周期, 则当读信号发出后, 地址要维持的时长就是 1 个周期, 1 个周期时长时, Tacc=0. 要让读信号发出后, 地址信号维持 2 个周期, 则 Tacc=001 = 2clocks. 2 个周期的维持时间, 读信号肯定足够。

4, Tcoh: 是读信号之后, 片选信号还要维持多长时间。

Tcoh	[7:6]	Chip selection hold time after nOE	000
		00 = 0 clock	
		01 = 1 clock	
		10 = 2 clocks	
		11 = 4 clocks	

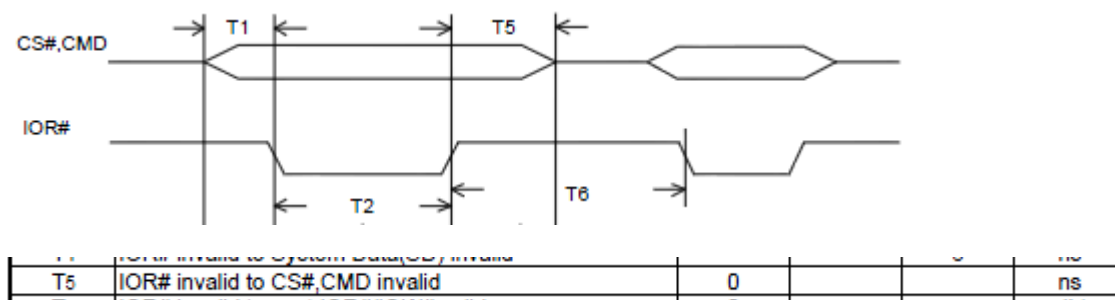
2440 时序:

读:

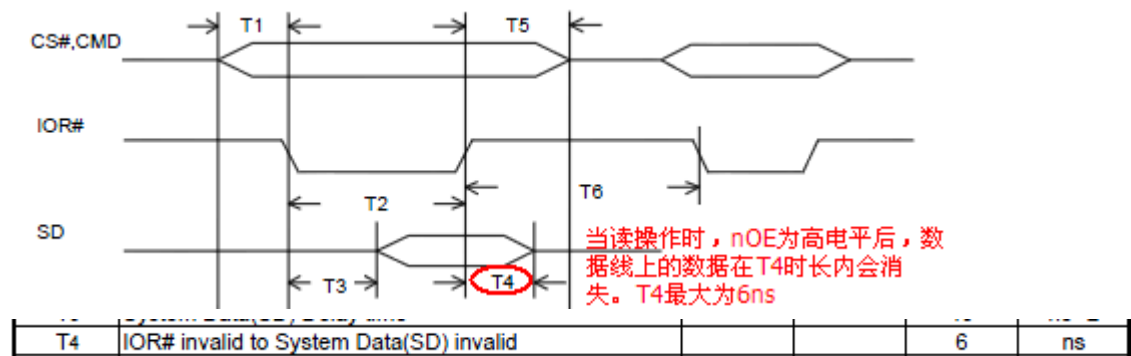


DM9000 时序:

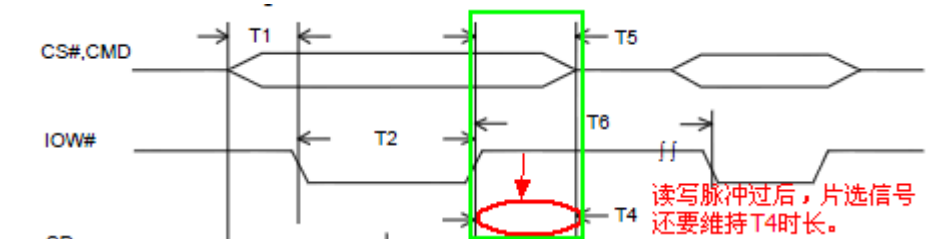
读:



DM9000C 进行读操作时, nOE 变为高电平之后, 数据线上的数据在 6ns 之内会消失:



DM9000C 进行写操作时，nWE 变为高电平之后，数据线上的数据还要维持最少 3ns：



从 DM9000 上时序知道，T4 最小为 3ns，则 Tcoh 要大于等于 3，HCLK 为 100MHz（1 周期为 10ns），则 Tcoh 取值：

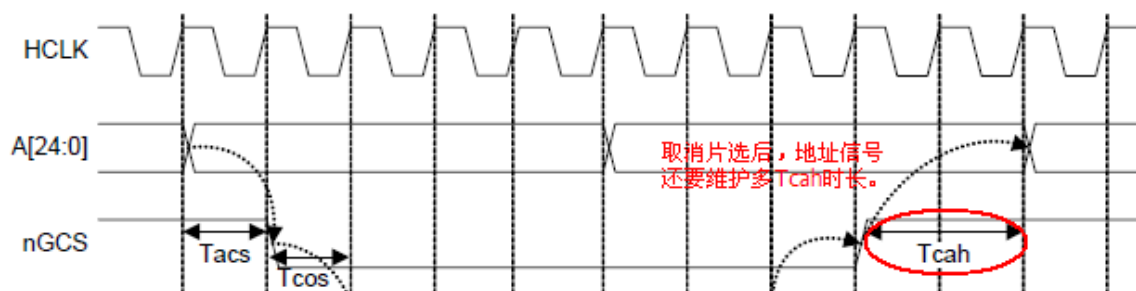
Tcoh	[7:6]	Chip selection hold time after nOE	000
		00 = 0 clock	
		01 = 1 clock	
		10 = 2 clocks	
		11 = 4 clocks	

不能为 00，00 则为 0 周期（小于了 3ns），则这里选择为“01”。

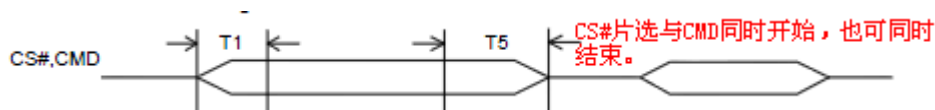
5, Tcah: 在取消片选后，地址信号要维持的时间。

DM9000C 的片选信号和 CMD 信号可以同时出现，同时消失，所以设为 0

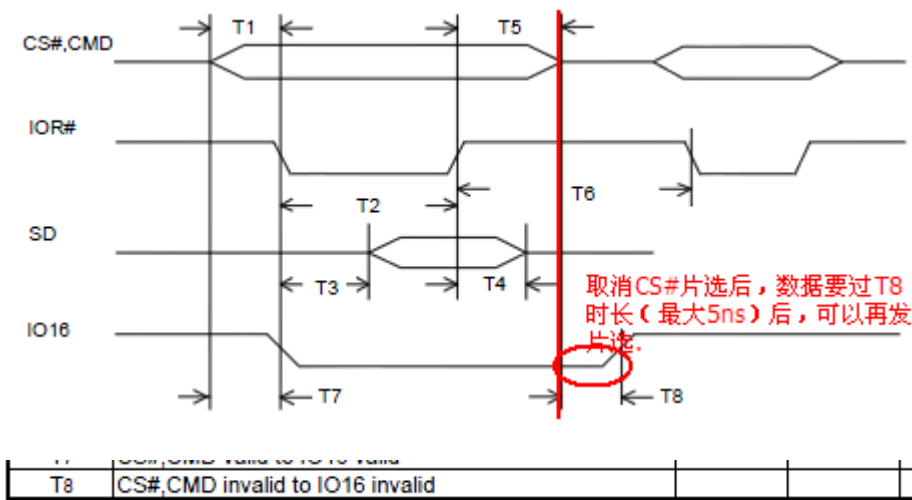
2440: 读写



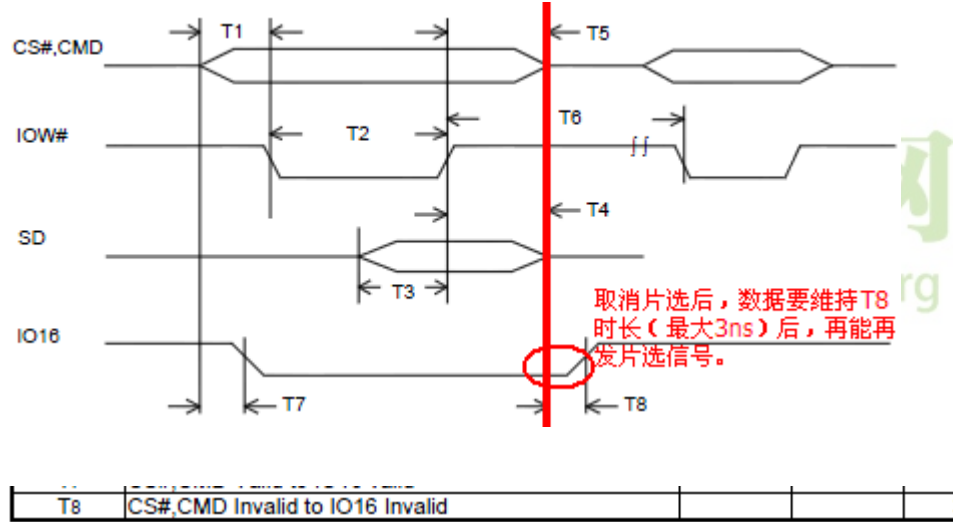
DM9000:



读：



写：



取消片选后，数据要维持时间 T8，最大要过 3ns，在这个 3ns 时间内，再发出片选信号是无效的。所以保持 3ns 后再发出片选信号。

因为是说在一个片选信号后，只要 T8 时长内不再发出另一个片选信号即可，故这个 "Tach" 只要大于等于 T8 时长即可。T8 要么为最大为 5ns，要么最大为 3ns，则由于一周期为 10ns，故 Tach = 01（1 周期）。

6, Tacp: 页模式。此位域不用配置。

Tacp	[3:2]	Page mode access cycle @ Page mode 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = 6 clocks	00
------	-------	--	----

7, PMC: 页模式配置。

PMC	[1:0]	Page mode configuration 00 = normal (1 data) 01 = 4 data 10 = 8 data 11 = 16 data	00
-----	-------	---	----

我们是正常模式，一次读一个数据。

总结：

其实 2440, DM9000 芯片的时序上都有分读、写两种时序，里面有点小差别。下面总结的是写时序时的取值：

BANKCONn	Bit	HCLK为100MHz, 即1周期为10ns Description	Initial State
Tacs	[14:13]	Address set-up time before nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tcos	[12:11]	Chip selection set-up time before nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacc	[10:8]	Access cycle 000 = 1 clock 001 = 2 clocks 010 = 3 clocks 011 = 4 clocks 100 = 6 clocks 101 = 8 clocks 110 = 10 clocks 111 = 14 clocks Note: When nWAIT signal is used, Tacc ≥ 4 clocks.	111
Tcoh	[7:6]	Chip selection hold time after nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	000
Tcah	[5:4]	Address hold time after nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacp	[3:2]	Page mode access cycle @ Page mode 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = 6 clocks	00
PMC	[1:0]	Page mode configuration 00 = normal (1 data) 01 = 4 data 10 = 8 data 11 = 16 data	00

设置：

```
/*
* Tacs[14:13]: 发出片选信号之前, 多长时间内要先发出地址信号
*              DM9000C的片选信号和CMD信号可以同时发出,
*              所以它设为0
* Tcos[12:11]: 发出片选信号之后, 多长时间才能发出读信号nOE
*              DM9000C的T1>=0ns,
*              所以它设为0
* Tacc[10:8] : 读写信号的脉冲长度,
*              DM9000C的T2>=10ns,
*              所以它设为1, 表示2个hclk周期, hclk=100MHz, 就是20ns
* Tcoh[7:6]  : 当读信号nOE变为高电平后, 片选信号还要维持多长时间
*              DM9000C进行写操作时, nWE变为高电平之后, 数据线上的数据还要维持最少3ns
*              DM9000C进行读操作时, nOE变为高电平之后, 数据线上的数据在6ns之内会消失
*              我们取一个宽松值: 让片选信号在nOE放为高电平后, 再维持10ns,
*              所以设为01
* Tcah[5:4]  : 当片选信号变为高电平后, 地址信号还要维持多长时间
*              DM9000C的片选信号和CMD信号可以同时出现, 同时消失
*              所以设为0
* PMC[1:0]   : 00-正常模式
*
*/
*bankcon4 = (1<<8) | (1<<6); /* 对于DM9000C可以设Tacc为1, 对于DM9000E, Tacc要设大一点, 比如最大值7 */
**bankcon4 = (7<<8) | (1<<6); /* MINI2440使用DM9000E, Tacc要设大一点 */
```



DM9000C 驱动侧式及内存控制器

测试 DM9000C 驱动程序：

1. 把 dm9dev9000c.c 放到内核的 drivers/net 目录下
2. 修改 drivers/net/Makefile

把

```
obj-$(CONFIG_DM9000) += dm9000.o
```

改为

```
obj-$(CONFIG_DM9000) += dm9dev9000c.o
```

3. make uImage

使用新内核启动

4. 使用 NFS 启动

或

```
ifconfig eth0 192.168.1.17
```

```
ping 192.168.1.1
```

之前没有设置 memory controller 时网卡可以使用，是因为之前在 UBOOT 中设置好了。这个 DM9000 能使用依赖于 UBOOT 的设置。要是想让网卡不依赖于 UBOOT 时，就需要自己来设置位宽和 BANKCON4。

设置位宽和 BANKCON4。

当“Tacc[10:8]”：

设置为 0 时。即“000=1clock”一个时钟。

Tacc[10:8] : 读写信号的脉冲长度，DM9000C的T2>=10ns，所以它设为0，表示一个hclk周期，hclk=100MHz，就是10ns

Tacc	[10:8]	Access cycle	
		000 = 1 clock	001 = 2 clocks
		010 = 3 clocks	011 = 4 clocks
		100 = 6 clocks	101 = 8 clocks
		110 = 10 clocks	111 = 14 clocks
Note: When nWAIT signal is used, Tacc ≥ 4 clocks.			

写的太临界，如按上面的取值“Tacc = 000”时，网卡驱动崩溃了。所以将此值放大。取默认值时，为 111 即 14 个时钟，这又太慢。

```
22.6$ cp arch/arm/boot/uImage
22.6$ vi drivers/net/Makefile
```

内存控制器：

CPU 执行的指令要么是：

“ldr r0,[某个地址 A]”这是从这个地址里读取某个数据存到 r0 里去。ldr 是装载 4 个字节。

A 地址的 1 字节数据。

A+1 地址的 1 字节数据。

A+2 地址的 1 字节数据。

A+3 地址的 1 字节数据。

即，ldr 是得到 4 个地址上的数据共4字节。

“ldb r0,[某个地址A]”，ldb是装载1个字节。对CPU来说一个地址对应的数据是 1 字节。

CPU执行这两种指令时，就要得到4字节或1字节的数据。

实例：

```
Mov R1,#0
```

```
Ldr R0,[R1]
```

从R1把的地方即0地址读到4字节。

相当于：

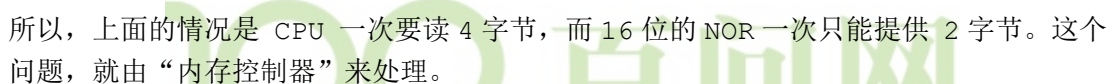
从0地址取1字节

从1地址取1字节

从2地址取1字节

从3地址取1字节

假设 NOR 是 16 位（有 16 条数据线），CPU 发出“ldr r0,[0]”，则 0 是对应 NORFLASH。这里是想得到 4 个字节的数据。NORFLASH 是以 16 位来访问的，意思是说访问 NOR 时，一次只能返回一个 16 位的数据。（16 位的 NORFLASH 硬件上只能一次提供 2 字节数据）。



www.100ask.org

- 所以要设置“内存控制器”，当 CPU 想得到 4 字节数据时，内存控制器要发出 2 次操作。这就要设置“位宽-WSx”。

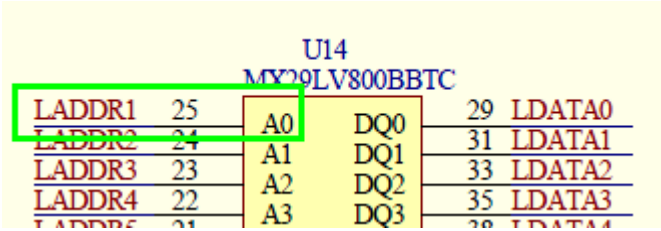
Register	Address	R/W	Description	Reset Value
BWSCON	0x48000000	R/W	Bus width & wait status control register	0x000000
DW0	[2:1]		Indicate data bus width for bank 0 (read only). 01 = 16-bit, 10 = 32-bit The states are selected by OM[1:0] pins	—

ST4	[19]	Determines SRAM for using UB/LB for bank 4. 0 = Not using UB/LB (The pins are dedicated nWBE[3:0]) 1 = Using UB/LB (The pins are dedicated nBE[3:0])	0
WS4	[18]	Determines WAIT status for bank 4. 0 = WAIT disable 1 = WAIT enable	0
DW4	[17:16]	Determine data bus width for bank 4. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	0

对于其他的 BANK，其他片选信号，需要设置位宽。

要设置位宽，就是因为 CPU 发出命令就是要读到 4 字节。至于如何将数据准备交过来，就只由内存控制器来确定。

这也说明了为什么地址信号错开一位的原因。



如上 NORFLASH，CPU 的发出的地址 1 “LADDR1” 接到了 NORFLASH 的 “A0” 地址。则：“内存控制器” 要发起 2 次传输：

- NOR:
- ①，第一次，发出 0 地址，得到 2 字节。 NOR 得到 0 地址，返回第 0 个 2 字节。
 - ②，第二次，发出 2 地址，得到 2 字节。 2 为 000010, 而 LADDR0 没有接，而是接的 LADDR1-A0，这时 NOR 看到的是地址 1 。故返回第一个 2 字节。
 - ③，将两次得到的字节数据返回给 CPU。

例 2：

```
Mov R1, #3
Ldb R0, [R1]
```

CPU想得到地址 3 的一个字节数据。

- “内存控制器” 要发起 2 次传输： NOR:
- ①，发地址 3 (000011)
 - NOR 得到地址 1 (000011, LADDR0 没接，则是从第 2 个 1 接的)，返回第 0 个 2 字节。
 - ②，内存控制器从这 2 字节里取出高字节那一位，给 CPU。
- NORFLASH 中位宽是由硬件来设置的 (0M[1:0]设置了)。而那些时间参数是使用了默认值 (像 DM9000 中设置时序中的时间一样)。

DM9000 驱动在 MINI2440 上的移植

DM9000 驱动在 MINI2440 上的移植学习笔记

想了解一下 DM9000 的移植修改原理,所以分析了一下时序图和引脚连接

首先看一下 DM9000 的引脚和 MINI2440 的引脚连接

DM9000 MINI2440 功能描述

SD0 DATA0 数据信号

| |

SD15 DATA15 数据信号

CMD ADDR2 识别为地址还是数据

INT EINT7 中断

IOR# nOE 读命令使能

IOW# nWE 写命令使能

AEN nGCS4 片选使能

可以看出连接了 16 条数据线,1 条地址线,而这唯一的一条地址线用于判断数据线传输的是地址还是数据,所以这 16 条数据线为数据和地址复用

而片选信号使用的 BANK4,则访问 0x2000 0000 - 0x27FF FFFF 这个范围的地址时会激活片选使能信号 nGCS4

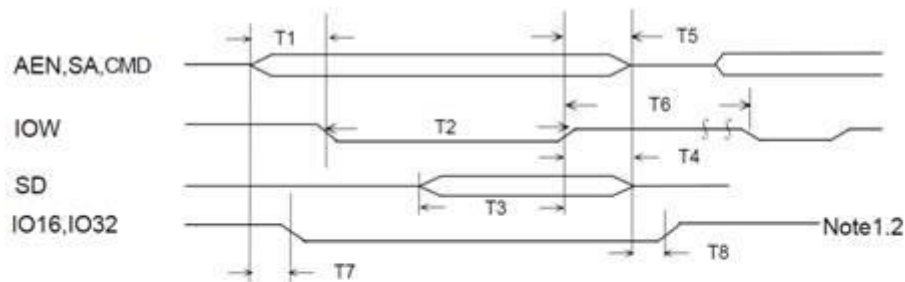
而在 MINI2440 提供的内核中,DM9000 的地址 IO 地址为 0x2000 0000,数据 IO 为 0x2000 0004

则向地址 IO 写数据的时候不会激活 ADDR2,所以向 DM9000 传送的数据为地址,而向数据 IO 写数据的时候会激活 ADDR2,所以向 DM9000 传送的数据为数据

现在看看 DM9000 和 S3C2440 的时序信号

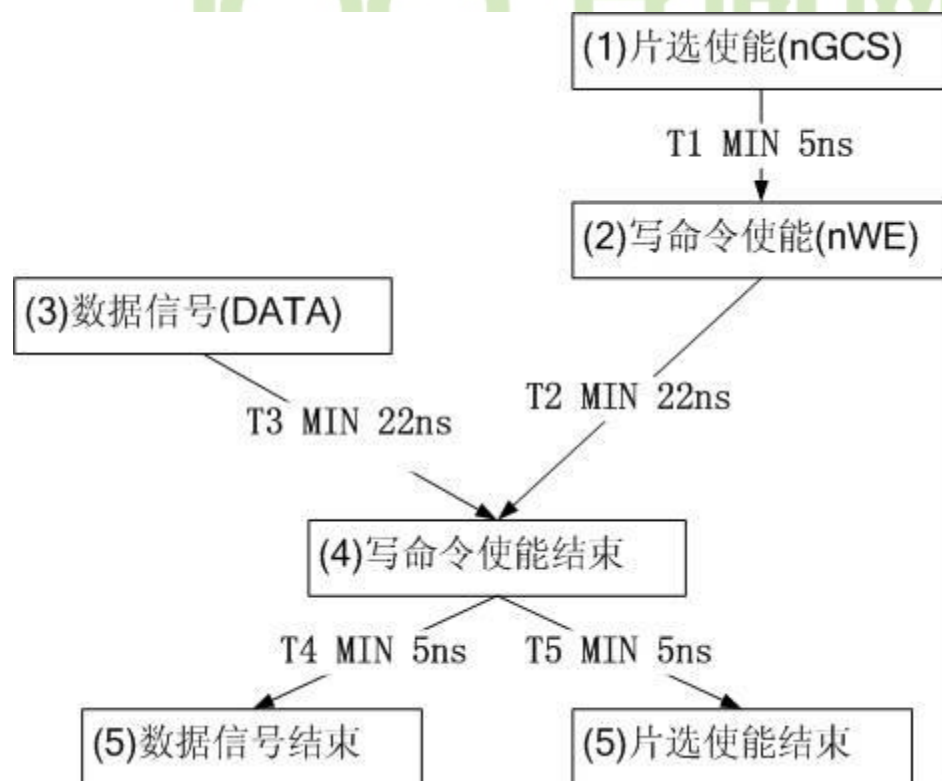
DM9000 的写时序

10.4.4 Processor Register Write Timing

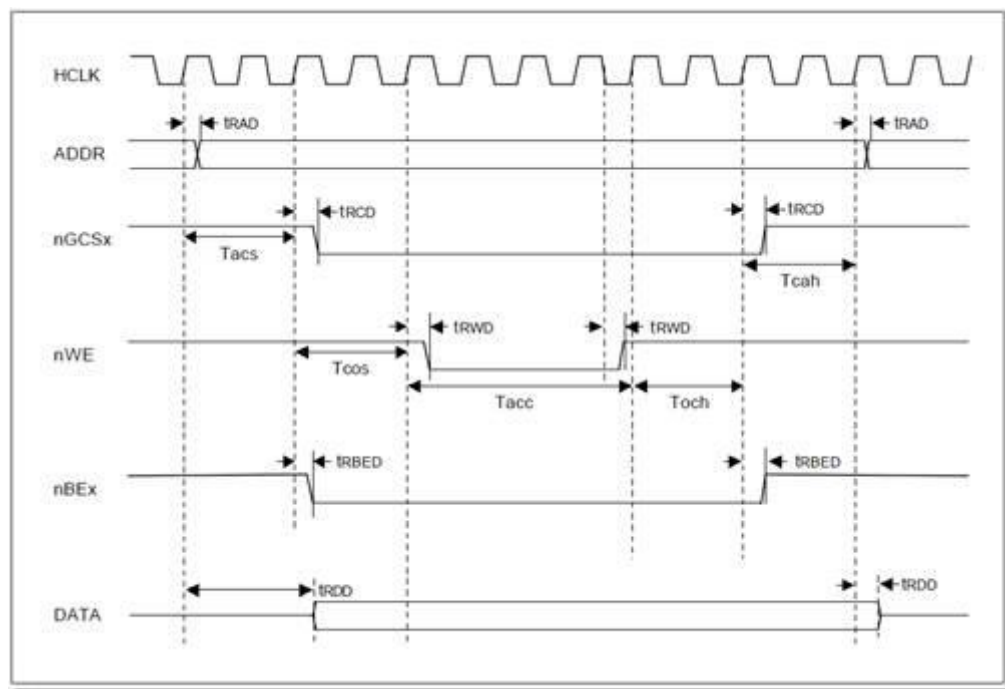


Symbol	Parameter	Min.	Typ.	Max.	Unit
T1	System Address Valid to IOW Valid	5			ns
T2	IOW Width	22			ns
T3	SD Setup Time	22			ns
T4	SD Hold Time	5			ns
T5	IOW Invalid to System Address Invalid	5			ns
T6	IOW Invalid to Next IOW valid access DM9000	84			ns
T7	System Address Valid to IO16, IO32 Valid			5	ns
T8	System Address Invalid to IO16, IO32 Invalid			5	ns

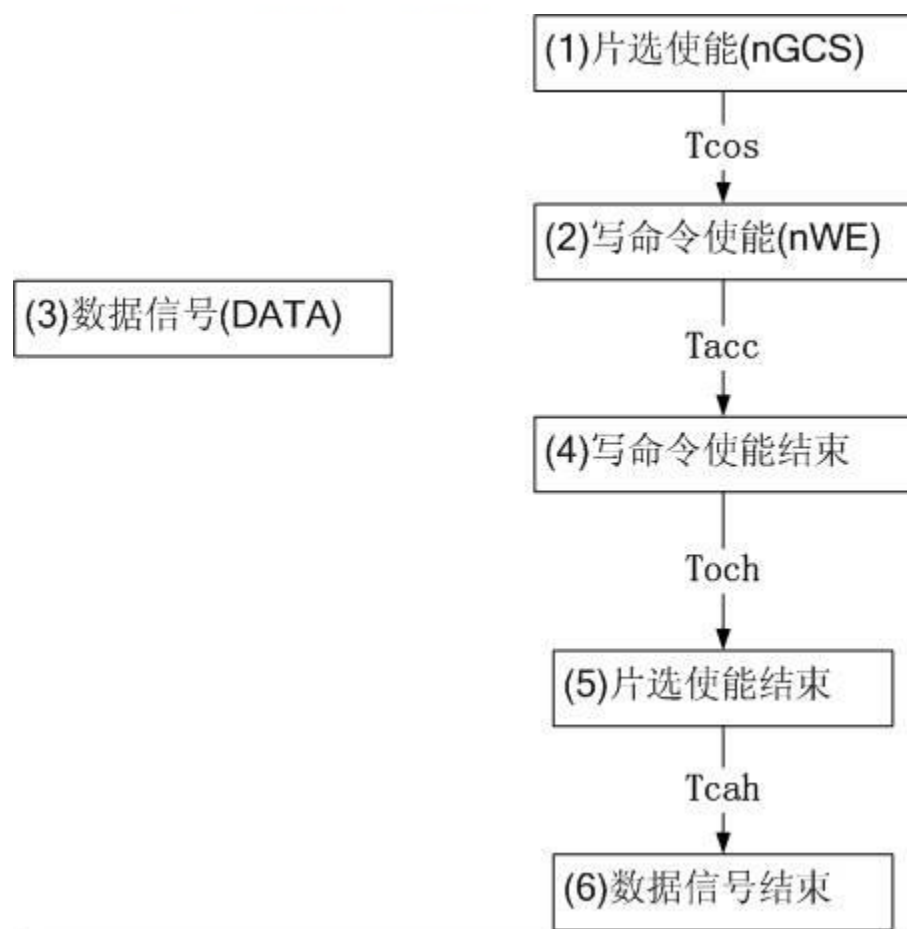
IO16, IO32 这两个引脚在 MINI2440 并没有连接, 所以不看这两个引脚的时序那么整理如下:



还有就是写命令使能结束后到下一个写命令使能需要最少 84ns 的间隔时间, 为 T6
 然后是 S3C2440 的写时序, 由于 DM9000 是连接在 BANK4 上的, 而 BANK 的写时序如下



由于 DM9000 在 MINI2440 上只需要片选使能, 写命令使能和数据信号, 所以我们不看 ADDR 和 nBE 信号, 那么整理如下



那么这些值为多少呢?~
来看看 BANKCON4

BANKCONn	Bit	Description	Initial State
Tacs	[14:13]	Address set-up time before nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tcos	[12:11]	Chip selection set-up time before nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacc	[10:8]	Access cycle 000 = 1 clock 001 = 2 clocks 010 = 3 clocks 011 = 4 clocks 100 = 6 clocks 101 = 8 clocks 110 = 10 clocks 111 = 14 clocks Note: When nWAIT signal is used, Tacc ≥ 4 clocks.	111
Tcoh	[7:6]	Chip selection hold time after nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	000
Tcah	[5:4]	Address hold time after nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacp	[3:2]	Page mode access cycle @ Page mode 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = 6 clocks	00
PMC	[1:0]	Page mode configuration 00 = normal (1 data) 01 = 4 data 10 = 8 data 11 = 16 data	00

这 里的值以时钟为周期,而 BANKCON 是接在 Memory Controller 上的(参考 S3C2440A 数据手册的表 1-4),而 Memory Controller 使用的是 Hclk 总线时钟信号(参考 S3C2440A 数据手册的图 7-1,感谢 kasim 大大指点),根据 S3C2440 手 册,Hclk 是由 Fclk 分频来的,具体的分频比每个板子的设置不一样,所以这里频率的设定要自己根据板子的设置来分析,假设主频为 400MHz,然后 Fclk,Hclk,Pclk 的分频比为 1:2:4,那么 Hclk 就是 200MHz,那么每个时钟周期就是 5ns

开始和 DM9000 的时序图进行对比,计算

Tcos对应T1, 那么最少应该为5ns, 也就是1个clock

Tacc对应T2, 那么最少应该为22ns, 那么我们这里最少也要选6个clock, 也就是30ns

Toch对应T5, 在这里无设置, 不过根据字面意思, 我认为Tcoh就是Toch, Toch最少应该为5ns, 也就是1个clock

Tcah对应T4, 由于之前已经有Toch了, 那么这里可以设置为0ns, 也就是0个clock

在S3C2440中, 一个写命令使能结束到下一个写命令使能开始的时间间隔为Toch + Tcah + Tacp + Tacs + Tcos

Tacs 是地址信号之后片选信号的起始间隔,我们这里先设为 0ns, 也就是 0 个 clock

Toch + Tcah + Tacp + Tacs + Tcos应该 > 84

5 + 0 + Tacp + 0 + 5 > 84

Tacp > 74

但是 Tacp 的最大值为 6 个 clock, 也就是 30ns, 还少了 44ns, 大概 9 个 clock

只要修改 Toch Tcah Tacs 和 Tcos 了, 虽然我们给的都是最小值, 但是为了信号稳定, 可以放宽其范围,

将Tcos和Toch设置为4个clock

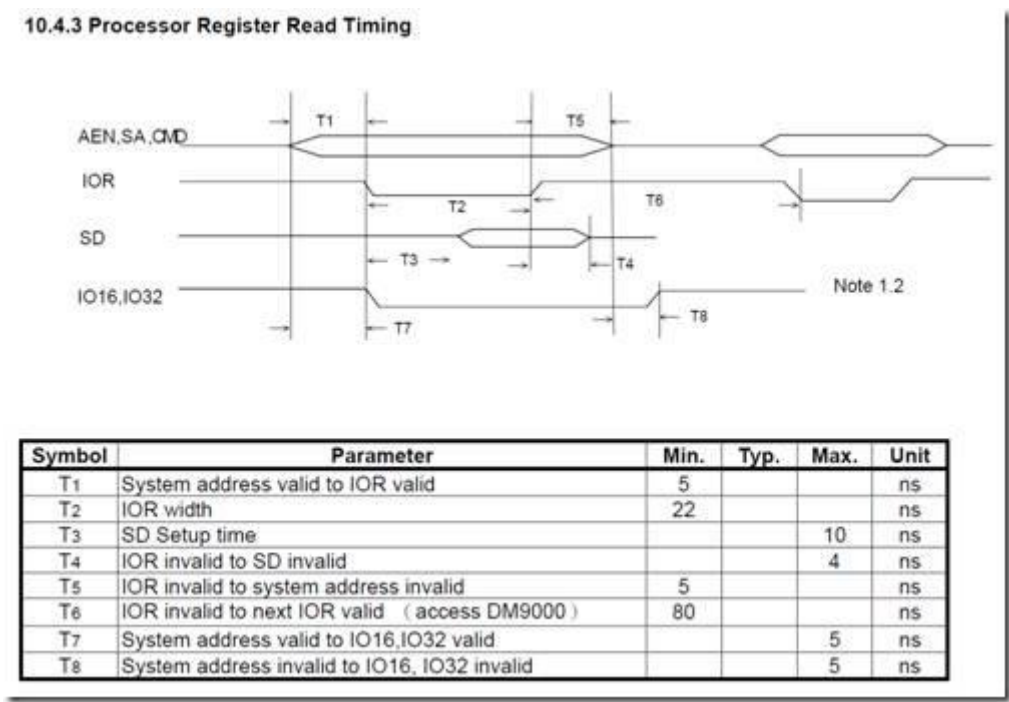
将Tacs和Tcah设置为2个Clock

这样总时间为 (4 + 2 + 6 + 2 + 4)*5 = 90ns

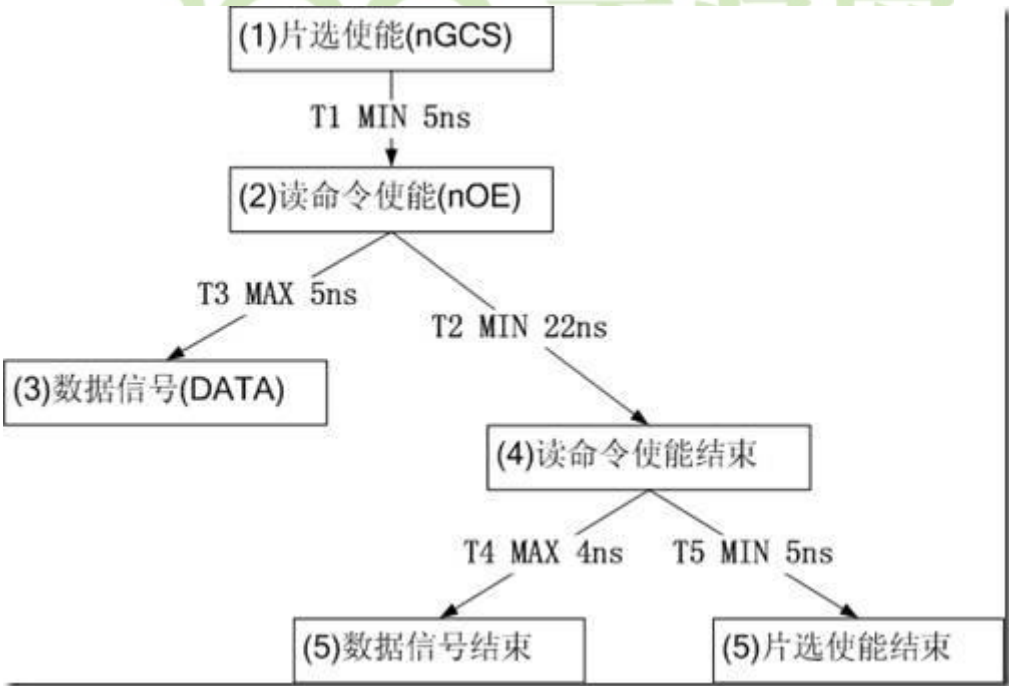
最后 DM9000 1 个周期只能处理 1 个数据, 所以 PMC 应该为 normal (1data)

写时序分析完了, 现在来看看读时序

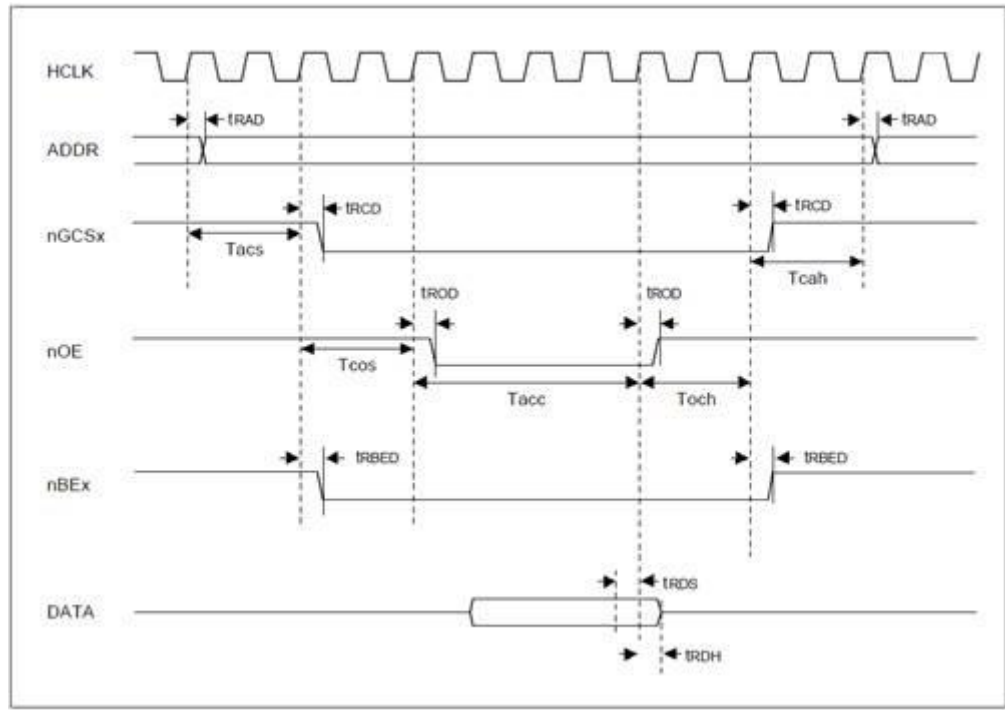
DM9000 的读时序如下



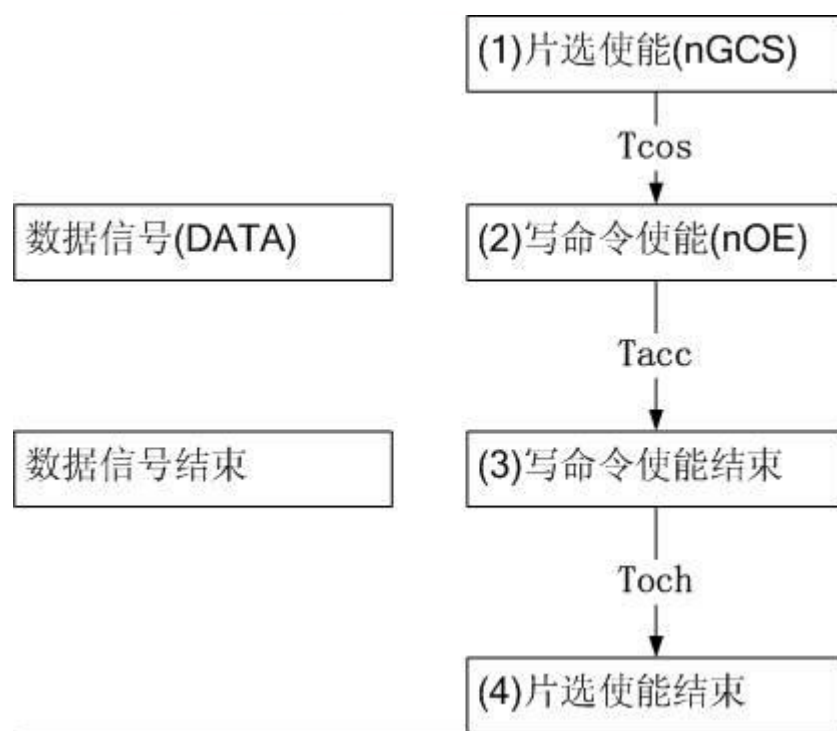
呢么整理如下:



读命令使能结束后到下一个读命令使能需要最少 80ns 的间隔时间, 为 T6
然后是 S3C2440 的读时序, 时序如下



整理如下：



T_{cos} 对应T1, 呢么最少应该为5ns, 也就是1个clock, 这里设置为和写操作一样的4个clock

T_{acc} 对应T2, 呢么最少应该为22ns, 这里设置为和写操作一样的6个clock

T_{och} 对应T5, 呢么最少应该为5ns, 也就是1个clock, 这里设置为和写操作一样的4个clock

其它时间间隔先设置和写操作一样

T_{cah} 为2个clock

T_{acp} 为6个clock

T_{acs} 为2个clock

PMC为normal(1data)

然后看看满足读命令使能结束后到下一个读命令使能的时间间隔 80ns 不

还是 $T_{och} + T_{cah} + T_{acp} + T_{acs} + T_{cos}$

$(4 + 1 + 6 + 1 + 4) * 5 = 15 * 5 = 90ns$, 能符合条件

那么 BANKCON4 的设置如下

$T_{acs} = 2 \text{ 个 clock} = 10$

$T_{cos} = 4 \text{ 个 clock} = 11$

$T_{acc} = 6 \text{ 个 clock} = 100$

$T_{coh} = 4 \text{ 个 clock} = 11$

$T_{cah} = 2 \text{ 个 clock} = 10$

$T_{acp} = 6 \text{ 个 clock} = 11$

PMC = normal(1data) = 00

也就是 0x5CEC

再来看 BWSCON, 这个寄存器负责配置 BANK 的带宽和等待状态

我们接的是 nGCS4, 那么主要就看 ST4, WS4 和 DW4 这几个字段

DW4 的描述为 BANK4 的带宽, DM9000 接了 16 条地址线, 那么带宽就是 16, 这里选 01

WS4 的描述为是否为 BANK4 使用等待状态, DM9000 没有接 WAIT 引脚, 所以可以不管这个字段

ST4 的描述为是否为 BANK4 使用 UB/LB (写高/低字节使能), DM9000 没有接

nWBE[3:0] 这 4 个引脚, 所以也不管这个字段

现在看看友善的 Linux 下 DM9000 驱动为适应 S3C2440 做了什么修改

```
#if defined(CONFIG_ARCH_S3C2410)
#include <mach/regs-mem.h>
#endif

#if defined(CONFIG_ARCH_S3C2410)
//取得带宽及等待状态控制寄存器的地址
unsigned int oldval_bwscon = *(volatile unsigned int *)S3C2410_BWSCON;
//取得4号BANK的控制寄存器的地址
unsigned int oldval_bankcon4 = *(volatile unsigned int *)S3C2410_BANKCON4;
#endif

#if defined(CONFIG_ARCH_S3C2410)
//先清除BWSCON上的DW4为0
//然后设置带宽为16位
//启用BANK4的WAIT状态
//启用BANK4的SRAM的写高低字节使能
*((volatile unsigned int *)S3C2410_BWSCON) =
(oldval_bwscon & ~(3<<16)) | S3C2410_BWSCON_DW4_16 | S3C2410_BWSCON_WS4 |
S3C2410_BWSCON_ST4;
//设置PMC - Page mode configuration - 1 data
// Tacc - Page mode access cycle @ Page mode - 6 clocks
// Tcah - Address hold time after nGCSn - 4 clocks
// Tcoh - Chip selection hold time after nOE - 1 clock
// Tacc - Access cycle - 14 clocks
// Tcos - Chip selection set-up time before nOE - 4 clocks
```

```
// Tacs - Address set-up time before nGCSn - 0 clock
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x1f7c;
#endif
#if defined(CONFIG_ARCH_S3C2410)
printf("Now use the default MAC address: 08:90:90:90:90:90\n");
mac_src = "friendly-arm";
ndev->dev_addr[0] = 0x08;
ndev->dev_addr[1] = 0x90;
ndev->dev_addr[2] = 0x90;
ndev->dev_addr[3] = 0x90;
ndev->dev_addr[4] = 0x90;
ndev->dev_addr[5] = 0x90;
#else
#if defined(CONFIG_ARCH_S3C2410)
*((volatile unsigned int *)S3C2410_BWCON = oldval_bwcon;
*((volatile unsigned int *)S3C2410_BANKCON4 = oldval_bankcon4;
#endif
#endif
```

主要就是执行 3 个功能

修改 BWCON 寄存器

修改 BANKCON4 寄存器

修改 MAC 信息

以前看别人移植 UBoot 给 MINI2440, Fclk, Hclk, Pclk 的分频比 1:4:8

那么 MINI2440 上的 Hclk 就是 100MHz, 也就是 1 个时钟 10ns, 刚好比上面分析的大 2 倍,

那么我们就可以将时钟数/2

```
Tacs = 1个clock = 01
Tcos = 2个clock = 10
Tacc = 3个clock = 010
Tcoh = 2个clock = 10
Tcah = 1个clock = 01
Tcap = 3个clock = 01
PMC = normal(ldata) = 00
```

也就是 0x3294

这里要注意的是使用 WAIT 信号的时候 Tacc 要大于等于 4 个 clock

所以将

```
((volatile unsigned int *)S3C2410_BWCON) =
    (oldval_bwcon & ~(3<<16)) | S3C2410_BWCON_DW4_16 | S3C2410_BWCON_WS4 |
S3C2410_BWCON_ST4;
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x1f7c;
```

改为

```
((volatile unsigned int *)S3C2410_BWCON) =
    (oldval_bwcon & ~(3<<16 | S3C2410_BWCON_WS4 | S3C2410_BWCON_ST4 )) |
S3C2410_BWCON_DW4_16;
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x3294;
```

大家喜欢的还可以把

```
#if defined(CONFIG_ARCH_S3C2410)
```

```
printk("Now use the default MAC address: 08:90:90:90:90:90\n");
```

改为

```
#if defined(CONFIG_ARCH_N02410)
```

```
printk("Now use the default MAC address: 08:90:90:90:90:90\n");
```

这样就会通过读取DM9000来得到MAC地址, 我经过试验, 得出的MAC地址为ff:ff:ff:ff:ff:ff

不知道会对TCP/IP协议栈有什么影响

这是使用

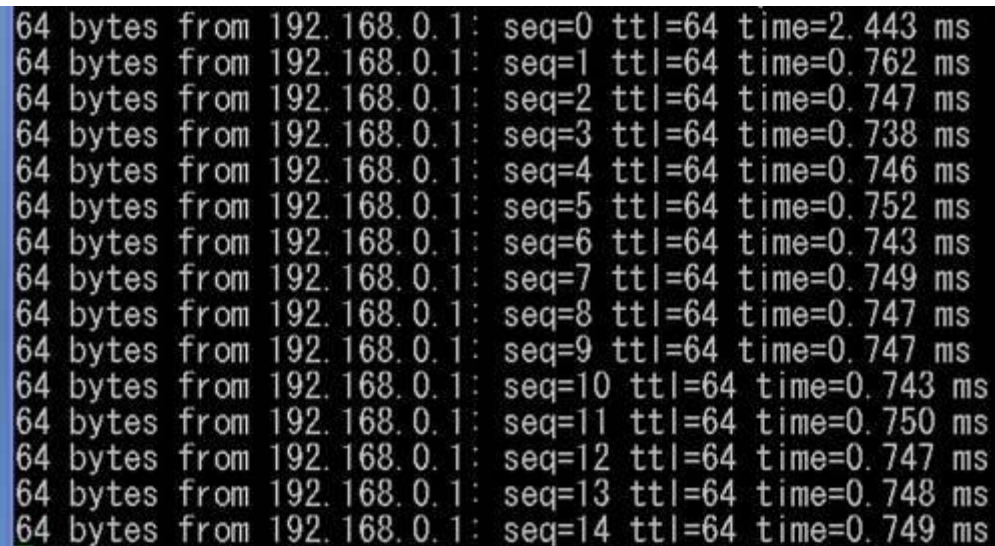
```
*((volatile unsigned int *)S3C2410_BWSCON) =
```

```
(oldval_bwscon & ~(3<<16)) | S3C2410_BWSCON_DW4_16 | S3C2410_BWSCON_WS4 |
```

```
S3C2410_BWSCON_ST4;
```

```
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x1f7c;
```

时候的延迟



```
64 bytes from 192.168.0.1: seq=0 ttl=64 time=2.443 ms
64 bytes from 192.168.0.1: seq=1 ttl=64 time=0.762 ms
64 bytes from 192.168.0.1: seq=2 ttl=64 time=0.747 ms
64 bytes from 192.168.0.1: seq=3 ttl=64 time=0.738 ms
64 bytes from 192.168.0.1: seq=4 ttl=64 time=0.746 ms
64 bytes from 192.168.0.1: seq=5 ttl=64 time=0.752 ms
64 bytes from 192.168.0.1: seq=6 ttl=64 time=0.743 ms
64 bytes from 192.168.0.1: seq=7 ttl=64 time=0.749 ms
64 bytes from 192.168.0.1: seq=8 ttl=64 time=0.747 ms
64 bytes from 192.168.0.1: seq=9 ttl=64 time=0.747 ms
64 bytes from 192.168.0.1: seq=10 ttl=64 time=0.743 ms
64 bytes from 192.168.0.1: seq=11 ttl=64 time=0.750 ms
64 bytes from 192.168.0.1: seq=12 ttl=64 time=0.747 ms
64 bytes from 192.168.0.1: seq=13 ttl=64 time=0.748 ms
64 bytes from 192.168.0.1: seq=14 ttl=64 time=0.749 ms
```

响应时间在 0.747 左右

这是使用

```
*((volatile unsigned int *)S3C2410_BWSCON) =
```

```
(oldval_bwscon & ~(3<<16 | S3C2410_BWSCON_WS4 | S3C2410_BWSCON_ST4 )) |
```

```
S3C2410_BWSCON_DW4_16;
```

```
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x3294;
```

时候的延迟

```
64 bytes from 192.168.0.1: seq=0 ttl=64 time=2.493 ms
64 bytes from 192.168.0.1: seq=1 ttl=64 time=0.746 ms
64 bytes from 192.168.0.1: seq=2 ttl=64 time=0.729 ms
64 bytes from 192.168.0.1: seq=3 ttl=64 time=0.725 ms
64 bytes from 192.168.0.1: seq=4 ttl=64 time=0.726 ms
64 bytes from 192.168.0.1: seq=5 ttl=64 time=0.728 ms
64 bytes from 192.168.0.1: seq=6 ttl=64 time=0.715 ms
64 bytes from 192.168.0.1: seq=7 ttl=64 time=0.727 ms
64 bytes from 192.168.0.1: seq=8 ttl=64 time=0.724 ms
64 bytes from 192.168.0.1: seq=9 ttl=64 time=0.731 ms
64 bytes from 192.168.0.1: seq=10 ttl=64 time=0.733 ms
64 bytes from 192.168.0.1: seq=11 ttl=64 time=0.727 ms
64 bytes from 192.168.0.1: seq=12 ttl=64 time=0.729 ms
64 bytes from 192.168.0.1: seq=13 ttl=64 time=0.725 ms
```

可见响应时间有所改善~ 不过我没有示波仪, 所以不知道这样的设置会不会对 DM9000 造成不良影响~

所以大家的 DM9000 挂掉的话不要来找我哈~ 哈哈 (逃~

关于 BANCON4 的：看数据手册

这个上面写的是啥啊，呵呵，我们操作一个时序器件的话时序是很重要的，而那写就是操作器件时序，我们设置好了的话，以后操作硬件 S3C2440 会自动给相应的器件时序，这样我们就设置一次就可以了，方便吧。那么这些时序是什么意思呢？那我们来看看。对于器件时序那我们就要分析器件的数据手册了，我们找到 dm9000 的数据手册，先看写时序：

T1: IOW有效之前地址信号有效的时间
T2: 写信号的有效期
T3: 数据在IOW信号消失之前数据的时间
T4: 数据的保持时间
T5: 地址在IOW信号消失之前的保持时间
T6: 下一个写信号与这个信号结束之间的时间间隔
T7: 地址信号有效IO16, IO32的有效时间
T8: 地址信号无效IO16, IO32有效的时间

各个时间段我们都看清楚了，现在我们看我们的 S3C2440 怎么给这个时间序了，对了，我们只要找到我们对应的连线就可以了，参考 S3C2440 原理图我们的 S3C2440 和 DM9000 的连线如下：

DM9000 简介层（为了好看加的） s3c2440

PW_RST	->nREST	->nREST
SD0-SD15	aLDATA0-LDATA15	->DATA0-DATA15
CMD	->LADDR2	->ADDR2
IRT	->I_LAN	->EINT7
IOR	->LnOE	->nOE
IOW	->LnWE	->nWE
IOWAIT	->Nwait	->nWAIT
AEN	-> nLAN_cs	->GPS4

所以要知道 IOW, IOR 等给出的时间序列，也就是看 nOE, new 等的序列就是了，我们继续看 S3C2440 找关于 BANK4 的时序序列有：

我们找出 s3c2440 和 dm9000 的时序段对应关系：

Dm9000	s3c2440
T1	Tcos
T2	Tacc
T3	Tacs+Tcos+Tacc
T4	Tacp+Toch+Tcah
T5	Tacp+Toch
T6	Tacp+Toch+Tcah+Tacs+Tcos

在 HCLK 为 100MHZ 的情况下（BANK4 使用 HCLK），一个时钟周期为 10ns, 根据 T1-T6 的最少时间（注：我们是可以把时间拉长，但是也要有个适度，因为时间的拉长应该会导致网卡的处理速度减慢，这个可以通过设置两种不同长短的时序，在看 ping 的延时，在同一网络）我们给 Tcos 等赋值为：

T1----Tcos	2个时钟20ns>5ns
------------	--------------

T2---Tacc	6个时钟 60ns>22ns	注意的是使用WAIT信号的时候Tacc要大于等于4个clock
T3---Tacs+Tcos+Tacc	Tacs 一个时钟:10+20+60=80ns>22ns	
T5--- Tacp+Toch	Toch一个时钟, Tacp六个时钟:10+60=70ns>5ns	
T4--- Tacp+Toch+Tcah	Tcah一个时钟:60+10+10=80ns>5ns	
T6--- Tacp+Toch+Tcah+Tacs+Tcos	60+10+10+10+20=110>84ns	

我们在对照: Tacs:01, Tcos:10, Tacc:100, Tcoh:01, Tcah:01, Tacp:11, PMC:00(1 个周期只能处理 1 个数据, 所以 PMC 应该为 normal (1data)), 所以对应的 BANCON4 为:0x345c

