网络对于嵌入式系统来说必不可少。可是s3c2440没有集成以太网接口，所以要想使s3c2440具备以太网的功能，就必须扩展网卡接口。在这里，我们外接DM9000，使其可以与以太网相连接。

DM9000可以直接与ISA总线相连，也可以与大多数CPU相连。在这里，我们当然是要让DM9000与s3c2440相连接了。DM9000对外来说只有两个端口——地址口和数据口，地址口用于输入内部寄存器的地址，而数据口则完成对某一寄存器的读写。DM9000的CMD引脚用来区分这两个端口，当CMD引脚为0时，DM9000的数据线上传输的是寄存器地址，当CMD引脚为1时，传输的是读写数据。我们把DM9000的A8和A9接为高电平，把A4~A7接为低电平，并且把DM9000的AEN接到s3c2440的nGCS4引脚上，则DM9000的端口基址为0x20000300，如果再把DM9000的CMD引脚接到s3c2440的ADDR2引脚上，则我们就可以定义DM9000的这两个端口地址，它们分别为：

#define DM\_ADDR\_PORT (\*((volatile unsigned short \*) 0x20000300)) //地址口

#define DM\_DATA\_PORT (\*((volatile unsigned short \*) 0x20000304)) //数据口

如果要写入DM9000中的某个寄存器，则先把该寄存器的地址赋予DM\_ADDR\_PORT，然后再把要写入的数据赋予DM\_DATA\_PORT即可。读取DM9000中的某个寄存器也类似。下面的函数的作用分别是DM9000的读、写寄存器操作：

//写DM9000寄存器

void \_\_inline dm\_reg\_write(unsigned char reg, unsigned char data)

{

DM\_ADDR\_PORT = reg; //将寄存器地址写到地址端口

DM\_DATA\_PORT = data; //将数据写到数据端口

}

//读DM9000寄存器

unsigned char \_\_inline dm\_reg\_read(unsigned char reg)

{

DM\_ADDR\_PORT = reg;

return DM\_DATA\_PORT; //将数据从数据端口读出

}

完成了对DM9000寄存器的读写函数的编写，下面我们就可以初始化DM9000，它的过程就是适当配置DM9000寄存器的过程。DM9000的内部寄存器在这里就不做介绍，而且DM9000的应用数据手册也有如何初始化DM9000的步骤，我们这里只给出具体的程序：

void dm\_init(void)

{

dm\_reg\_write(DM9000\_NCR,1); //软件复位DM9000

delay(30); //延时至少20μs

dm\_reg\_write(DM9000\_NCR,0); //清除复位位

dm\_reg\_write(DM9000\_NCR,1); //为了确保复位正确，再次复位

delay(30);

dm\_reg\_write(DM9000\_NCR,0);

dm\_reg\_write(DM9000\_GPCR,1); //设置GPIO0为输出

dm\_reg\_write(DM9000\_GPR,0); //激活内部PHY

dm\_reg\_write(DM9000\_NSR,0x2c); //清TX状态

dm\_reg\_write(DM9000\_ISR,0xf); //清中断状态

dm\_reg\_write(DM9000\_RCR,0x39); //设置RX控制

dm\_reg\_write(DM9000\_TCR,0); //设置TX控制

dm\_reg\_write(DM9000\_BPTR,0x3f);

dm\_reg\_write(DM9000\_FCTR,0x3a);

dm\_reg\_write(DM9000\_FCR,0xff);

dm\_reg\_write(DM9000\_SMCR,0x00);

dm\_reg\_write(DM9000\_PAR1,0x00); //设置MAC地址：00-01-02-03-04-05

dm\_reg\_write(DM9000\_PAR2,0x01);

dm\_reg\_write(DM9000\_PAR3,0x02);

dm\_reg\_write(DM9000\_PAR4,0x03);

dm\_reg\_write(DM9000\_PAR5,0x04);

dm\_reg\_write(DM9000\_PAR6,0x05);

dm\_reg\_write(DM9000\_NSR,0x2c); //再次清TX状态

dm\_reg\_write(DM9000\_ISR,0xf); //再次清中断状态

dm\_reg\_write(DM9000\_IMR,0x81); //打开接受数据中断

}

DM9000内部有0x3FF大小的SRAM用于接受和发送数据缓存。在发送或接收数据包之前，数据是暂存在这个SRAM中的。当需要连续发送或接收数据时，我们需要分别把DM9000寄存器MWCMD或MRCMD赋予数据端口，这样就指定了SRAM中的某个地址，并且在传输完一个数据后，指针会指向SRAM中的下一个地址，从而完成了连续访问数据的目的。但当我们在发送或接受一个数据后，指向SRAM的数据指针不需要变化时，则要把MWCMDX或MRCMDX赋予数据端口。下面的程序为DM9000发送数据的函数，它的两个输入参数分别为要发送数据数组首地址和数据数组长度。在这里我们已经知道数据的宽为16位，它是由DM9000的硬件引脚设置实现的。

void dm\_tran\_packet(unsigned char \*datas, int length)

{

int i;

dm\_reg\_write(DM9000\_IMR, 0x80); //在发送数据过程中禁止网卡中断

dm\_reg\_write(DM9000\_TXPLH, (length>>8) & 0x0ff); //设置发送数据长度

dm\_reg\_write(DM9000\_TXPLL, length & 0x0ff);

DM\_ADDR\_PORT = DM9000\_MWCMD; //发送数据缓存赋予数据端口

//发送数据

for(i=0;i<length;i+=2)

{

delay(50);

DM\_DATA\_PORT = datas[i]|(datas[i+1]<<8); //8位数据转换为16位数据输出

}

dm\_reg\_write(DM9000\_TCR, 0x01); //把数据发送到以太网上

while((dm\_reg\_read(DM9000\_NSR) & 0x0c) == 0)

; //等待数据发送完成

delay(50);

dm\_reg\_write(DM9000\_NSR, 0x2c); //清除TX状态

dm\_reg\_write(DM9000\_IMR, 0x81); //打开DM9000接收数据中断

}

发送数据比较简单，接收数据就略显复杂，因为它是有一定格式要求的。在接收到的一包数据中的首字节如果为0x01，则表示这是一个可以接收的数据包；如果为0x0，则表示没有可接收的数据包。因此在读取其他字节时，一定要先判断首字节是否为0x01。数据包的第二个字节为数据包的一些信息，它的高字节的格式与DM9000的寄存器RSR完全一致。第三个和第四个字节为数据包的长度。后面的数据就是真正要接收的数据了。下面就是DM9000接收数据的程序，其中输入参数为存放输入数据数组的首地址，输出参数为接收数据的长度。

int dm\_rec\_packet(unsigned char \*datas)

{

unsigned char int\_status;

unsigned char rx\_ready;

unsigned short rx\_status;

unsigned short rx\_length;

unsigned short temp;

int i;

int\_status = dm\_reg\_read(DM9000\_ISR); //读取ISR

if(int\_status & 0x1) //判断是否有数据要接受

{

rx\_ready = dm\_reg\_read(DM9000\_MRCMDX); //先读取一个无效的数据

rx\_ready = (unsigned char)DM\_DATA\_PORT; //真正读取到的数据包首字节

if(rx\_ready == 1) //判读首字节是否为1或0

{

DM\_ADDR\_PORT = DM9000\_MRCMD; //连续读取数据包内容

rx\_status = DM\_DATA\_PORT; //状态字节

rx\_length = DM\_DATA\_PORT; //数据长度

if(!(rx\_status & 0xbf00) && (rx\_length < 10000)) //判读数据是否符合要求

{

for(i=0; i<rx\_length; i+=2) //16位数据转换为8位数据存储

{

delay(50);

temp = DM\_DATA\_PORT;

datas[i] = temp & 0x0ff;

datas[i + 1] = (temp >> 8) & 0x0ff;

}

}

}

else if(rx\_ready !=0) //停止设备

{

//dm\_reg\_write(DM9000\_IMR,0x80); //停止中断

//dm\_reg\_write(DM9000\_ISR,0x0F); //清中断状态

//dm\_reg\_write(DM9000\_RCR,0x0); //停止接收

//还需要复位系统，这里暂时没有处理

}

}

dm\_reg\_write(DM9000\_ISR, 0x1); //清中断

return rx\_length;

}

关于DM9000的设置我们就介绍到这里，下面就是s3c2440的设置。在这里，网卡发送数据利用的是查询方式，接收数据利用的是中断方式，因此我们把DM9000的INT引脚连接到了s3c2440的EINT7上。另外我们还是用UART0接口来控制和显示网卡数据。这两个接口的初始化为：

//uart0 port

rGPHCON = 0x00faaa;

rGPHUP = 0x7ff;

rULCON0 = 0x3;

rUCON0 = 0x5;

rUFCON0 = 0;

rUMCON0 = 0;

rUBRDIV0 = 26;

rSRCPND = (0x1<<27)|(0x1<<28);

rSUBSRCPND = 0x1;

rINTPND = (0x1<<27)|(0x1<<28);

rINTSUBMSK = ~(0x1);

rINTMSK = ~((0x1<<27)|(0x1<<28));

pISR\_UART0 = (U32)uartISR;

//EINT7

rGPFCON = 2<<14;

rEXTINT0 = (rEXTINT0 & (~(0x07<<28))) | (0x01<<28);

rEINTMASK &= ~(1<<7);

rSRCPND = rSRCPND | (0x1<<4);

rINTPND = rINTPND | (0x1<<4);

rINTMSK &= ~(1<<4);

pISR\_EINT4\_7 = (U32)DM9000ISR;

下面就利用DM9000来进行简单的网卡传输数据的测验。由于以太网传输数据都是基于某种协议的，因此要传输数据，必须遵循一定的协议格式。这里我们实现较为简单的ARP协议。用于以太网的ARP请求/应答分组格式为：14个字节的以太网首部＋28个字节ARP请求/应答。以太网首部的格式为：6个字节的以太网目标地址＋6个字节以太网源地址＋2个字节帧类型，对于ARP来说，帧类型为0x0806。ARP请求/应答的格式为：2个字节的硬件类型＋2个字节的协议类型＋1个字节的硬件地址长度＋1个字节的协议地址长度＋2个字节的操作码＋6个字节的发送端以太网地址＋4个字节的发送端IP地址＋6个字节的目标以太网地址＋4个字节的目标IP地址。硬件类型为1表示的是以太网，协议类型为0x0800表示的是IP地址，硬件地址长度和协议地址长度分别为6和4，它们都是以字节为单位的，操作码为1表示的是ARP请求，为2表示的是ARP应答。

在下面的测试程序中，我们用交叉网线把开发板与PC机（操作系统为Windows XP，网卡的IP地址为192.168.1.120）相连接，我们通过UART发出一个命令，让开发板发出一个ARP请求数据包，然后接收来自PC机的应答，并把该应答信息通过UART显示出来。其中UART的中断复位程序为：

void \_\_irq uartISR(void)

{

char ch;

rSUBSRCPND |= 0x1;

rSRCPND |= 0x1<<28;

rINTPND |= 0x1<<28;

ch=rURXH0;

if(ch == 0x33)

comm=3; //表示发送一个ARP数据请求包

rUTXH0=ch;

}

另外我们还要事先定义一个遵循ARP协议格式的数组：

unsigned char arpsendbuf[42]={

0xff,0xff,0xff,0xff,0xff,0xff, //以太网目标地址，全1表示为广播地址

0x00,0x01,0x02,0x03,0x04,0x05, //以太网源地址

0x08,0x06, //帧类型：ARP帧

0x00,0x01, //硬件类型：以太网

0x08,0x00, //协议类型：IP协议

0x06, //硬件地址长度：6字节

0x04, //协议地址长度：4字节

0x00,0x01, //操作码：ARP请求

0x00,0x01,0x02,0x03,0x04,0x05, //发送端以太网硬件地址

192, 168, 1, 50, //发送端IP协议地址

0x00,0x00,0x00,0x00,0x00,0x00, //接收端以太网硬件地址

192, 168, 1, 120 //接收端IP协议地址

};

其中发送端硬件地址，即以太网源地址（00-01-02-03-04-05）是我们初始化DM9000时定义的。而发送端IP协议地址是我们任意定义的。

该测试程序的主程序为：

void Main(void)

{

…… ……

//一些必要的初始化

comm=0; //命令

flag=0; //发送ARP请求包标识

dm\_init(); //DM9000初始化

while(1)

{

if(comm.==3)

{

comm=0;

dm\_tran\_packet(arpsendbuf, 42 ); //发送ARP请求包

flag=1; //置标识

}

}

}

接收网络上的数据是通过外部中断方式的，在这个中断处理程序中，主要完成的是接收网卡数据，并把接收到的数据发送到UART，让其显示到PC机上。这里我们还需解决一个问题，那就是当我们发送一个ARP请求包的时候，XP系统并不会应答一个ARP数据包，而是应答一个IP协议数据包，当再多次发出ARP请求包后，才会得到ARP应答包。因此当s3c2440发送ARP请求包后，它首先要检查所接收到的数据包，如果不是ARP应答包，它就要再次发送ARP请求包，直到得到ARP应答包为止。因此中断处理程序为：

void \_\_irq DM9000ISR(void)

{

int i;

rSRCPND = rSRCPND | (0x1<<4);

rINTPND = rINTPND | (0x1<<4);

if(rEINTPEND&(1<<7))

{

rEINTPEND = rEINTPEND | (0x1<<7);

packet\_len = dm\_rec\_packet(buffer); //接收网卡数据

if((buffer[12]==0x08)&&(buffer[13]==0x06)) //是ARP协议

{

//通过UART显示出来

for(i=0;i<packet\_len;i++)

{

while(!(rUTRSTAT0 & 0x2)) ;

rUTXH0 = buffer[i];

}

flag=0; //清标志

}

else if(flag==1) //如果在发出ARP请求包后，接收到的数据不是ARP协议

{

comm=3; //继续发送ARP请求包

}

}

}

这样，整个网卡程序就编写完毕。为了使大家对程序的因果关系认识得更加清晰，我们再叙述一遍程序的流程：首先初始化UART0，使其用中断方式接收数据，查询方式发送数据；初始化EINT7，这是因为DM9000的数据中断引脚INT是连接到s3c2440的外部中断7引脚上的；然后初始化DM9000，主要是配置一些它的寄存器，并使其用中断方式接收网卡数据，查询方式发送数据，这与UART0相似，最后是死循环等待UART0接收中断服务程序中得到的发送ARP请求包命令。当得到发送ARP请求包命令后，调用DM9000发送数据命令，发送事先准备好的一组数据。在发送完ARP数据后，PC机会应答该请求，从而引发s3c2440外部中断7中断，在该中断服务程序中，主要是完成接收ARP应答包的任务，并把它通过UART0显示出来。

当程序被执行完，并在PC机上通过串口调试软件显示出了一个正确的ARP应答包后，我们还可以通过下列方法来进一步验证该程序的正确性：打开Windows XP系统只带的“命令提示符”小软件，在提示符下输入：arp –a，会出现我们所设置的开发板的MAC地址（00-01-02-03-04-05）和IP地址（192.168.1.50），则说明Windows XP系统已经把我们开发板上的网卡信息添加到了它的静态列表中。

我们对该系统进一步分析还会发现，当开发板上电并且DM9000初始化完成后，Windows XP系统会向该开发板发送一些目标地址为广播地址（FF-FF-FF-FF-FF-FF）的ARP数据包和IP数据包，只要我们正确读取它们，就可以在开发板上电后，自动知道与其相连的系统的MAC地址和IP地址了。另外，如果对这一部分感兴趣，还可以编写ICMP协议的数据包，这样就可以让PC机ping通我们的开发板了。