目录

实验七：I2C 串行接口总线通讯实验

一.实验目的

1. 了解 IIC 总线接口的基本原理，掌握通过 IIC 总线访问器件的方法；
2. 掌握初始化 UDA1380 的流程，为下面的实时时钟实验、看门狗实验和 E2PROM 实验打下理论基础；

二.实验原理和说明

1. 引入 IIC 总线接口的原因

在消费电子、通信和工控的一些系统设计上都有很多的相似部分，其中包括：

(1) 都有一些智能控制，如使用微处理器进行控制。

(2) 都有一些通用的电路，如 LCD 驱动、I/O 端口、RAM、EEPROM 和一些数据转换电路。

(3) 有面向应用的电路，如数字解调和数字信号处理等。

为了充分利用以上设计的相似之处，使硬件工作更有效，电路更简单，引入了 IIC 总线。IIC 总线是一个简单的双向总线，通过此总线对 IC 进行控制非常方便。

2. IIC 总线的概念

IIC 总线支持任何 IC 生产过程。两线－－串行数据（SDA）和串行时钟（SCL）线在连接到总线的器件间传递信息。每个器件都有一个唯一的地址识别（无论是微控制器、LCD 驱动器、存储器或键盘接口），而且都可以作为一个发送器或接收器。显然，LCD 驱动器只能是一个接收器，而存储器则既可以接收又可以发送数据。除了发送器和接收器外，器件在执行数据传输时也可以被看做是主机或从机。主机是初始化总线、数据传输并产生允许传输的时钟信号器件。此时，任何被寻址的器件都被认为是从机。

3. IIC 总线的术语

在了解 IIC 总线的同时，必须对 IIC 总线的相关术语有所了解，这样才能更清楚地掌握 IIC 总线。表 1 给出了 IIC 总线的相关术语。

表 1 IIC 总线的相关术语

术语	描述
发送器	发送数据到总线的器件
接受器	从总线接收数据的器件
主机	初始化发送、产生时钟信号和终止发送的器件
从机	被主机寻址的器件
多主机	同时有多于一个主机尝试控制总线，但不破坏报文
竞争	在存在多主机时，这些主机必须竞争确保只有一个获得总线

4. IIC 总线的特点

(1) IIC 总线仅有两条总线信号线：串行数据信号线（SDA）和串行时钟信号线（SCL）。

(2) 它是一个真正的多主机总线，如果两个或者更多的主机同时初始化数据传输，可以通过冲突检测和仲裁防止数据被破坏。

(3) 每一个连在此总线的设备都是可编址的。采用 IIC 总线连接的设备处于主从模式，主方即可接

受数据，也可发送数据。

(4) IIC 总线是一个串行的 8 位双向数据传送总线。在标准模式下，数据传输速率为 100Kb/s；在快模式下，数据传输速率为 400Kb/s；在高速传输模式下，数据传输速率为 3.4Mb/s。

5. IIC 数据传输规范

在位传输时，有两个重要的传输位：START（开始位）和 STOP（结束位）。START 位处在当 SDA 信号线上的状态由高到低转换且 SCL 信号线为高时；STOP 位处在当 SDA 信号线上的状态由低到高转换且 SCL 信号线为高时。在位传输时，开始与结束的位置如图 1 所示。

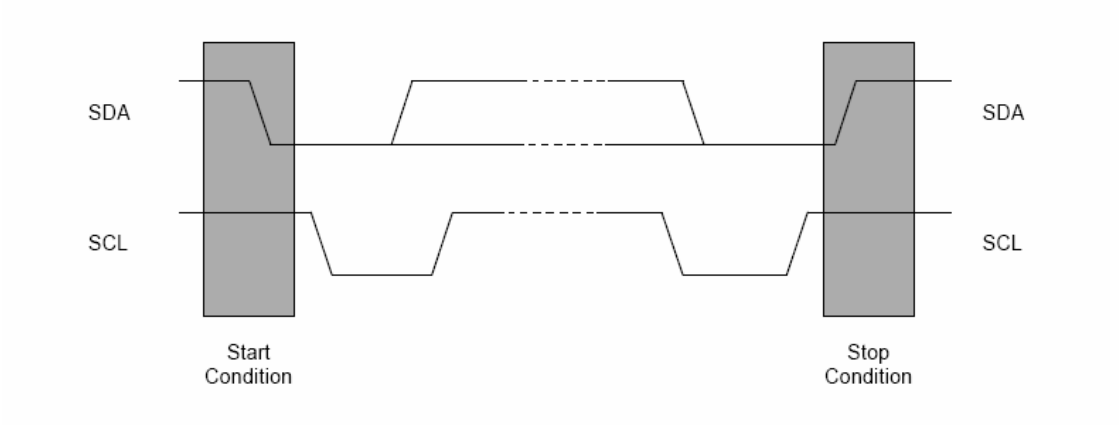


图 1 时序图中开始位置与停止位置图

在字节传输时，传送到 SDA 线上的每一个字节必须为 8 位：每次传送的字节数不限；每一个字节后面必须跟一个相应位。数据在传输时，首先传输最有意义位（Most Significant Bit，MSB）。如果在传输的过程中，从设备不能一次接受完一个字节，此时它就使时钟置为低电平，迫使主设备等待；当从设备能接收下一个数据字节后，将释放 SCL 线，继续后面的数据传输。如图 2 所示为数据传输时序图。

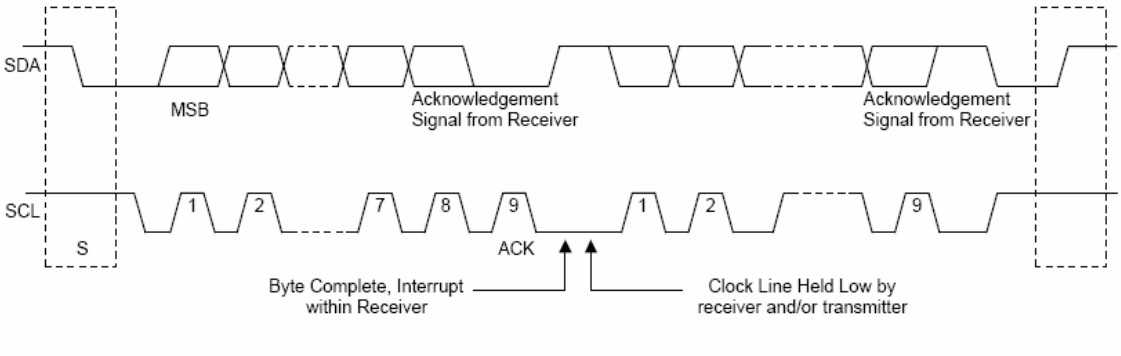


图 2 IIC 总线中数据传输时序图

6. S3C2410 IIC 总线寄存器

S3C2410 微处理器支持多主机 IIC 总线，并有 4 个寄存器来控制该总线的功能，这四个寄存器分别是多主机 IIC 总线控制寄存器、多主机 IIC 总线状态寄存器、多主机 IIC 总线数据寄存器、多主机 IIC 总线地址寄存器。下面对这四个寄存器分别予以介绍：

Register	Address	R/W	Description	Reset Value
IICCON	0x54000000	R/W	IIC-Bus control register	0x0X

IICCON	Bit	Description	Initial State
Acknowledge generation (note 1)	[7]	IIC-bus acknowledge enable bit. 0 = Disable, 1 = Enable In Tx mode, the IICSDA is free in the ack time. In Rx mode, the IICSDA is L in the ack time.	0
Tx clock source selection	[6]	Source clock of IIC-bus transmit clock prescaler selection bit. 0 = IICCLK = f_{PCLK} /16 1 = IICCLK = f_{PCLK} /512	0
Tx/Rx Interrupt (note 5)	[5]	IIC-Bus Tx/Rx interrupt enable/disable bit. 0 = Disable, 1 = Enable	0
Interrupt pending flag (note 2), (note 3)	[4]	IIC-bus Tx/Rx interrupt pending flag. This bit cannot be written to 1. When this bit is read as 1, the IICSCL is tied to L and the IIC is stopped. To resume the operation, clear this bit as 0. 0 = 1) No interrupt pending (when read). 2) Clear pending condition & Resume the operation (when write). 1 = 1) Interrupt is pending (when read) 2) N/A (when write)	0
Transmit clock value (note 4)	[3:0]	IIC-Bus transmit clock prescaler. IIC-Bus transmit clock frequency is determined by this 4-bit prescaler value, according to the following formula: Tx clock = IICCLK/(IICCON[3:0]+1).	Undefined

表 2 多主机 IIC 总线控制寄存器

这个寄存器特别重要，尤其是第 4 位。当一个字节的发送或者接收操作完成时，发生中断，第四位被置为 1，下面在程序中将具体予以分析。

Register	Address	R/W	Description	Reset Value
IICADD	0x54000008	R/W	IIC-Bus address register	0xXX

IICADD	Bit	Description	Initial State
Slave address	[7:0]	7-bit slave address, latched from the IIC-bus. When serial output enable = 0 in the IICSTAT, IICADD is write-enabled. The IICADD value can be read any time, regardless of the current serial output enable bit (IICSTAT) setting. Slave address = [7:1] Not mapped = [0]	XXXXXXXX

表 3 多主机 IIC 总线地址寄存器

Register	Address	R/W	Description	Reset Value
IICSTAT	0x54000004	R/W	IIC-Bus control/status register	0x0

IICSTAT	Bit	Description	Initial State
Mode selection	[7:6]	IIC-bus master/slave Tx/Rx mode select bits. 00: Slave receive mode 01: Slave transmit mode 10: Master receive mode 11: Master transmit mode	00
Busy signal status / START STOP condition	[5]	IIC-Bus busy signal status bit. 0 = read) Not busy (when read) write) STOP signal generation 1 = read) Busy (when read) write) START signal generation. The data in IICDS will be transferred automatically just after the start signal.	0
Serial output	[4]	IIC-bus data output enable/disable bit. 0 = Disable Rx/Tx, 1 = Enable Rx/Tx	0
Arbitration status flag	[3]	IIC-bus arbitration procedure status flag bit. 0 = Bus arbitration successful 1 = Bus arbitration failed during serial I/O	0
Address-as-slave status flag	[2]	IIC-bus address-as-slave status flag bit. 0 = Cleared when START/STOP condition was detected 1 = Received slave address matches the address value in the IICADD	0
Address zero status flag	[1]	IIC-bus address zero status flag bit. 0 = Cleared when START/STOP condition was detected. 1 = Received slave address is 00000000b.	0
Last-received bit status flag	[0]	IIC-bus last-received bit status flag bit. 0 = Last-received bit is 0 (ACK was received). 1 = Last-received bit is 1 (ACK was not received).	0

表 4 多主机 IIC 总线状态寄存器

Register	Address	R/W	Description	Reset Value
IICDS	0x5400000C	R/W	IIC-Bus transmit/receive data shift register	0xXX

IICDS	Bit	Description	Initial State
Data shift	[7:0]	8-bit data shift register for IIC-bus Tx/Rx operation. When serial output enable = 1 in the IICSTAT, IICDS is write-enabled. The IICDS value can be read any time, regardless of the current serial output enable bit (IICSTAT) setting.	XXXXXXXX

表 5 多主机 IIC 总线数据寄存器

7. S3C2410 IIC 对应的 I/O 端口介绍

Register	Address	R/W	Description	Reset Value
GPECON	0x56000040	R/W	Configure the pins of port E	0x0
GPEDAT	0x56000044	R/W	The data register for port E	Undefined
GPEUP	0x56000048	R/W	pull-up disable register for port E	0x0
Reserved	0x5600004C	–	Reserved	Undefined

表 6 I/0 寄存器介绍

GPECON	Bit	Description
GPE15	[31:30]	00 = Input 01 = Output (open drain output) 10 = IICSDA 11 = Reserved
GPE14	[29:28]	00 = Input 01 = Output (open drain output) 10 = IICSCL 11 = Reserved

表 7 端口 E 控制寄存器

GPEUP	Bit	Description
GPE[15:0]	[15:0]	0: The pull-up function attached to to the corresponding port pin is enabled. 1: The pull-up function is disabled.

表 8 端口 E 端口寄存器

8. S3C2410 IIC 总线上所接的 IC 设备

(1) UDA1380，原理图见图 3：

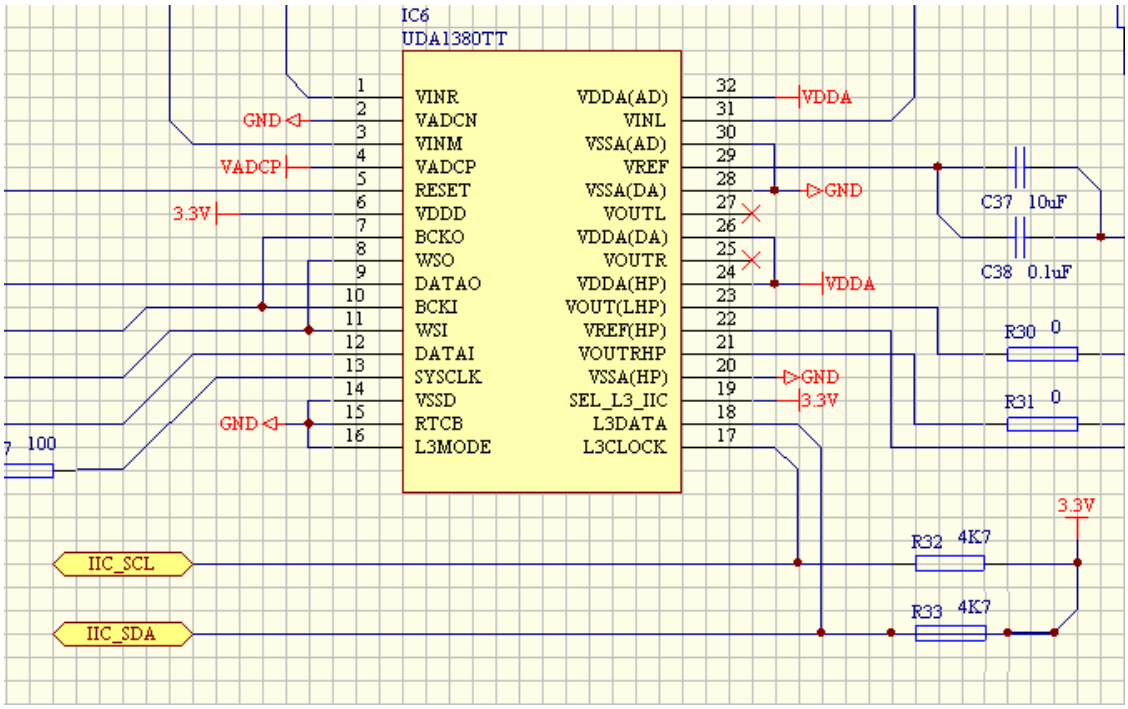


图 3 UDA1380 原理图

从图中可以看出 UDA1380 的 17 和 18 引脚，分别和 CPU 的 IIC_SCL 和 IIC_SDA 引脚相连。这里需要特别注意的是 UDA1380 的两个引脚都要上拉电阻，并且加上 3.3 伏的电压。

(2) X1227 芯片（包括实时时钟、看门狗定时器和 E2PROM），这个将在下面的实验中予以介绍。

9. UDA1380 读写模式及流程图介绍

(1) 写模式介绍

开始	设备地址	R/W		寄存器地址		MS data byte		LS data byte		停止
S	0011000	0	A	ADDR	A	MS1	A	LS1	A	P

表 9 IIC 模式下主机向 UDA1380 寄存器写

(2) 读模式介绍

开始	设备地址	R/W		寄存器地址			设备地址	R/W		MS data byte		LS data byte	
S	0011000	0	A	ADDR	A	Sr	0011000	1	A	MS1	A	LS1	NA

表 10 IIC 模式下主机向 UDA1380 寄存器写

读懂这两个模式对于编程非常的重要，从这两个表可以看出，读操作要比写操作复杂一些。读写流程图如下：

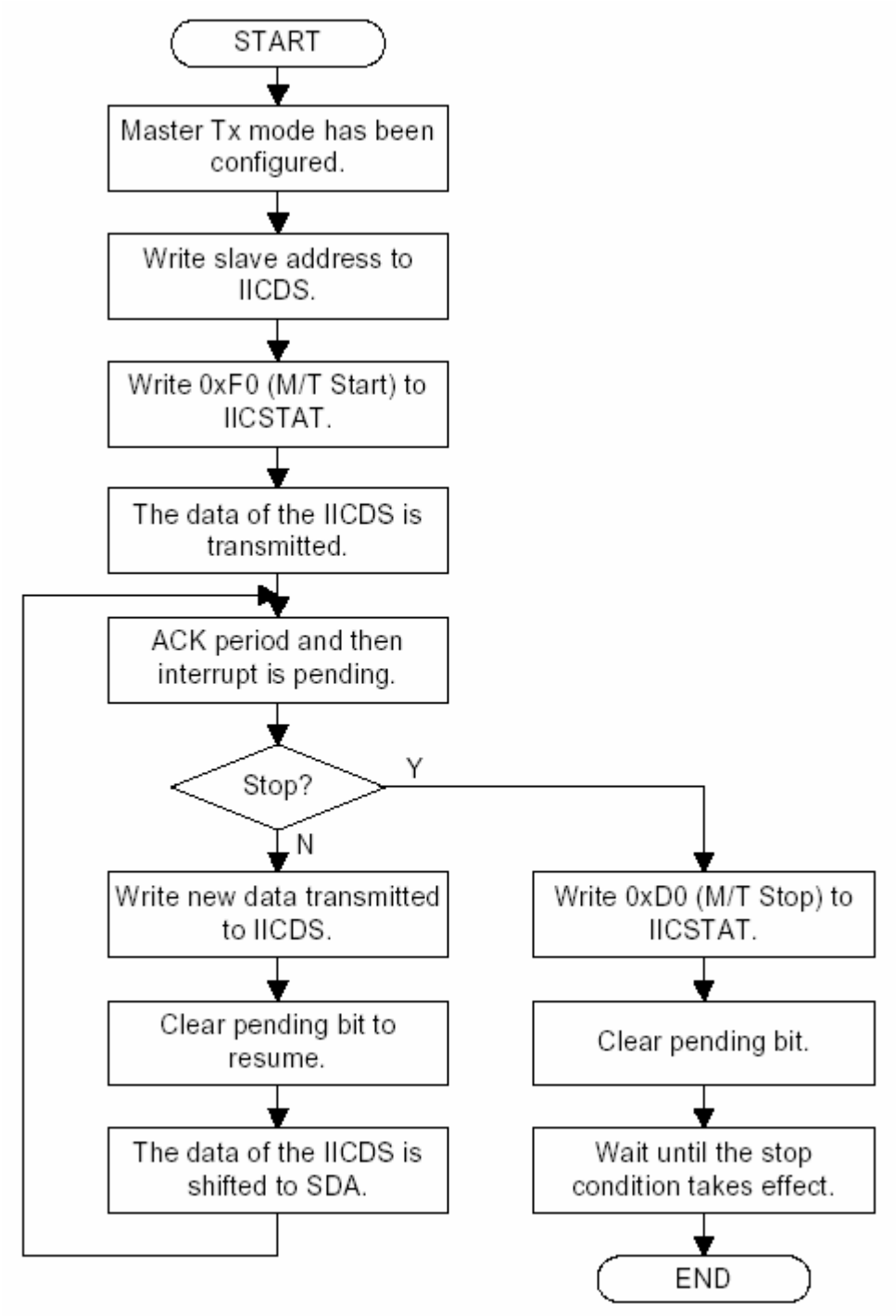


图 4 主发送模式流程图

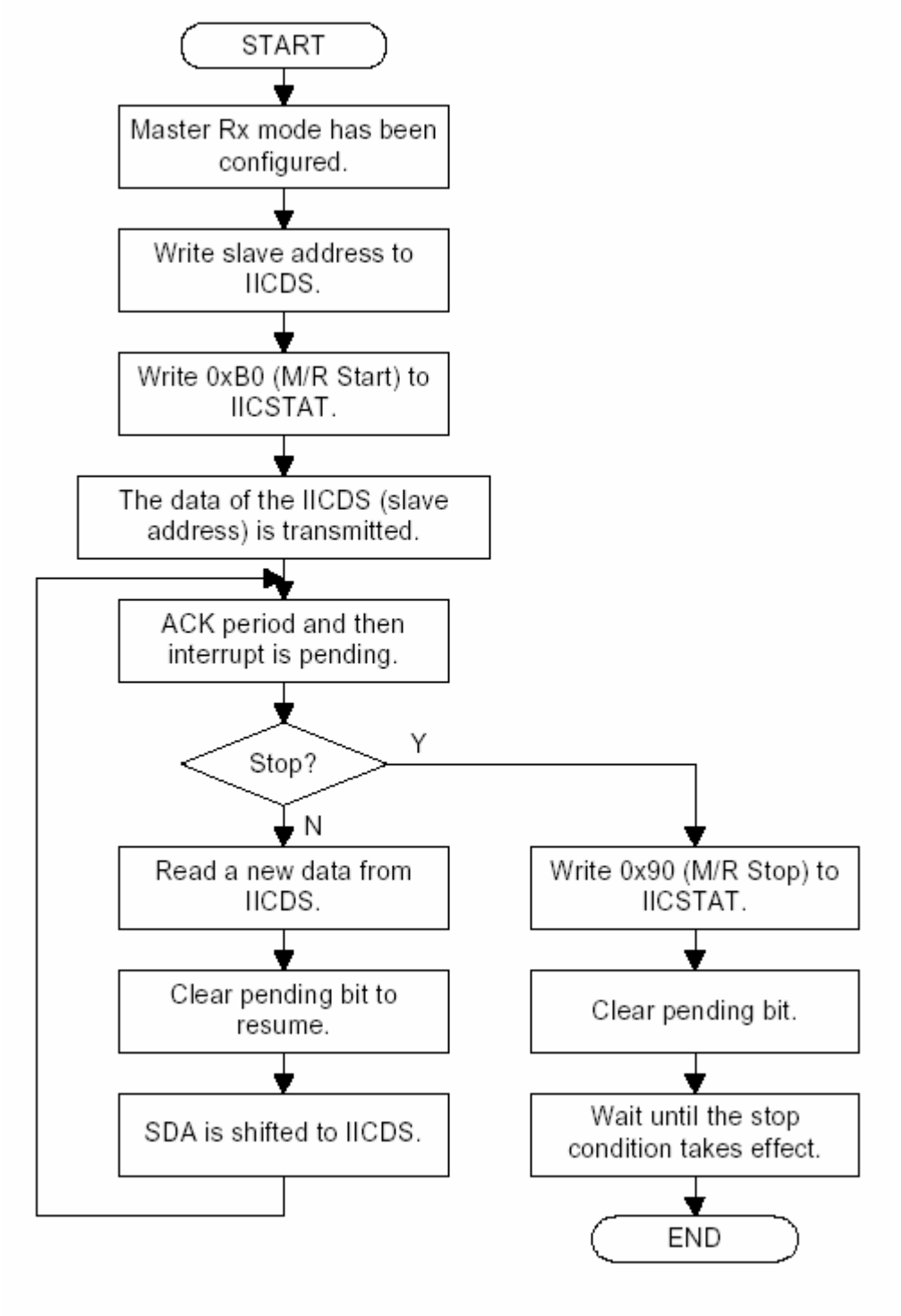


图 5 主接收模式流程图

10. 程序解读（实验一）

在 2410iic.c 文件中，主要有下面几个函数：

```
void   _Wr2410Iic(U32 slvAddr,U32 addr,U32 wdata)；
void   _Rd2410Iic(U32 slvAddr,U32 addr,U32 *rdata)；
void   Run_IicPoll(void)；
void   IicPoll(void)；
int    address_map(void)；
```

int Test_Iic(void)；

下面分别对后五个函数予以介绍，读者在此基础上编写_Wr2410Iic 函数。

(1) address_map 函数

该函数调用__ioremap 函数，把物理地址映射到内核空间的虚拟地址，以方便在内核中方便使用寄存器。

```
int   address_map(void)
{
r_GPECON  =__ioremap(0x56000040,4,0)；
r_GPEUP    =__ioremap(0x56000048,4,0)；
r_IICCON    =__ioremap(0x54000000,4,0)；
r_IICSTAT   =__ioremap(0x54000004,4,0)；
r_IICADD    =__ioremap(0x54000008,4,0)；
r_IICDS     =__ioremap(0x5400000c,4,0)；
      return 0；
}
```

(2) Run_IicPoll 函数

该函数的作用就是判断一个字节是否发送或者接收完毕，因为在表 2 已经说明了，只要传输完毕，就会产生中断，使 IIC 控制寄存器的 bit4 置 1。该函数判断该位，如果为 1，说明传输完毕，可以进行下面的字节传输，如果是 0，则继续等待，直到它变为 1。如果变为 1，就调用 IicPoll 函数。

```
void Run_IicPoll(void)
{
     if(rIICCON & 0x10)
        IicPoll()；
}
```

(3) IicPoll 函数

该函数的主体是一个 switch 语句，用来判断现在进行的是什么操作，并进行相应的读写操作。这个函数主要反映了 I2C 的时序，所以要认真理解。

```
case POLLACK：
      _iicStatus = iicSt；
      break；
case RDDATA：
      if((_iicDataCount--)==0)
      {
        _iicData[_iicPt++] = rIICDS；
        rIICSTAT = 0x90；            //停止读操作，见图 5 右面第一行
         rIICCON = 0xaf；
           Delay(1)；                //等待停止操作生效
         //Too long time...
        break；
       }
        _iicData[_iicPt++] = rIICDS；

       if((_iicDataCount)==0)              //准备读最后一个数据
         rIICCON = 0x2f；              //继续 IIC 操作不伴有 ACK
```

```
        else
          rIICCON = 0xaf;                //继续 IIC 操作伴有 ACK
      break;
    case WRDATA:
        if((_iicDataCount--)==0)
        {
            rIICSTAT = 0xd0;                    //stop MasTx condition
            rIICCON = 0xaf;                     //resumes IIC operation.
            Delay(1);                   //wait until stop condtion is in effect.
            //The pending bit will not be set after issuing stop condition.
            break;
         }
        rIICDS = _iicData[_iicPt++];
         for(i=0; i<10; i++);           //for setup time until rising edge of IICSCL
         rIICCON = 0xaf;                            //resumes IIC operation.
         break;

    case SETRDADDR:
        if((_iicDataCount--)==0)
        {
            break;    //IIC operation is stopped because of IICCON[4]
        }
        rIICDS = _iicData[_iicPt++];
        for(i=0; i<10; i++);   //for setup time until rising edge of IICSCL
        rIICCON = 0xaf;                 //resumes IIC operation.
        break;
    default:
        break;
```

(4) _Rd2410Iic 函数

该函数的编写要参考表 10 和图 5。具体代码如下：

```
  _iicMode = SETRDADDR;      //进行写操作
  _iicPt = 0;
  _iicData[0] = (U8)addr;
  _iicDataCount = 1;

  rIICDS = slvAddr&0xfe;      //设备地址，参见写操作图 4 的第三步
  rIICSTAT = 0xf0;              //参见写操作图 4 的第四步
  //发送设备地址

  while(_iicDataCount!=－1)
      Run_IicPoll();              //发送要读取寄存器地址
  _iicMode = RDDATA;            //进行读操作
  _iicPt = 0;
  _iicDataCount = 2;
```

```
    rIICDS     = slvAddr|0x01;     //设备地址，参见读操作图 5 的第三步
    rIICSTAT = 0xb0;          //主设备读，开始，参见读操作图 5 的第四步
    rIICCON    = 0xaf;          //继续 IIC 操作
    while(_iicDataCount!=－1)
        Run_IicPoll();

    *rdata = _iicData[1]*256+_iicData[2];   //所读到的数
```

(5) Test_Iic 函数

```
    unsigned int i，j，save_E，save_PE；
    static U32 data；
    static U32 rdata；
    static U32 wdata；

    address_map()；

    save_E     =    rGPECON；
    save_PE    =    rGPEUP；

    rGPEUP     |= 0xc000；                    //Pull-up disable
    rGPECON    |= 0xa00000；                 //GPE15：IICSDA , GPE14：IICSCL

    //Enable ACK, Prescaler IICCLK=PCLK/16, Enable interrupt, Transmit clock value Tx
      clock=IICCLK/16
    rIICCON    =    (1<<7) | (0<<6) | (1<<5) | (0xf)；

    rIICSTAT = 0x10；                         //IIC bus data output enable(Rx/Tx)

    printk("--------->1\n")；
    wdata    =    0x8aa8；
    i    =    0x11；
    _Wr2410Iic(0x30，(U8)i，wdata)；          //向寄存器写入
    printk("The writing data is    %x    \n"，wdata)；

    printk("--------->2\n")；
    _Rd2410Iic(0x31，(U8)I，&(rdata))；       //读出刚才写入的值
    printk("The reading data is    %x \n"，rdata)；

    rGPEUP    =    save_PE；
    rGPECON    =    save_E；
    return 0；
```

在 2410iic_test.c 文件中，只有两个驱动最基本的函数，如下：

```
int   init_module(void)
{
    int hehe；
    hehe   =   Test_lic()；
    printk("The test……insmod …….is   OK!\n")；
    return 0；
}

void cleanup_module(void)
{
   printk("The test..... rmmod...……is OK!\n")；
}
```

11. 程序解读（实验二）

和实验一的 2410iic.c 文件相比，该实验中的 2410iic.c 文件少了 Test_lic 函数，同时多了 uda1380_init 函数和一个初始化好的 command 数组。

(1) command 数组如下：

```
unsigned   char   command[]   =   {
      0x00, 0x0F, 0x06,   //Sysclock
      0x01, 0x00, 0x00,
      0x02, 0xaf, 0xff,     //line in
      0x03, 0x00, 0x00,
      0x04, 0x02, 0x02,     //default
      0x10, 0x00, 0x00,
      0x11, 0x00, 0x00,
      0x12, 0x00, 0x00,
      0x13, 0x00, 0x00,
      0x14, 0x00, 0x00,
      0x20, 0x00, 0x00,
      0x21, 0x00, 0x00,
      0x22, 0x00, 0x0c,     //mic
      0x23, 0x00, 0x00
};
```

为了方便理解，我们三个一行，每一行的第一个是 UDA1380 寄存器的地址，后两个是向该寄存器写入的值。这里把值设定好了，因为本实验的目的只是要学习如何使用 I2C 控制 IC 器件。

(2) uda1380_init 函数如下：

```
int   ic；
unsigned   int   msb，lsb，data，reg；
address_map()；
rGPEUP   |= 0xc000；                          //Pull-up disable
rGPECON |= 0xa00000；             //GPE15：IICSDA ，GPE14：IICSCL

rIICCON   = (1<<7) | (0<<6) | (1<<5) | (0xf)；
```

```
    rIICSTAT = 0x10；                    //IIC bus data output enable(Rx/Tx)

    for (ic = 0；  ic < 42；  ic += 3) {
  //这里需要添加几行，调用_Wr2410Iic 函数，初始化 UDA1380 的寄存器
    }
     return 0；
```

在 2410iic_init1380 文件里，有如下两个函数：

```
int init_module(void)
{
      int hehe；
      hehe   =   uda1380_init()；
      printk("Init uda1380 is OK!\n")；
      return 0；
}

void cleanup_module(void)
{
      printk("rmmod uda1380 is OK!\n")；
}
```

补充一下，上面两个实验的 Makefile 已经写好了，这两个实验均是采用模块的方式加载到系统中去；加载的名字可以在 Makefile 任意设定。

三.实验内容和步骤

1. 参考读函数，并根据表 9 和图 4 编写_Wr2410Iic()函数，并测试读写函数是否正确。

实验步骤：

(1) 把编写的_Wr2410Iic()函数加入到 2410IIC.c 文件里。

(2) 测试函数 Test_Iic 的功能是先通过 IIC 总线往 UDA1380 某寄存器写数据，然后把它读出来，看是否一致，以检测函数编写是否正确，IIC 工作是否正常，读懂它。

(3) 根据实验四介绍的驱动程序框架，读懂相应的驱动测试文件，2410IIC_Test.c。

(4) 读懂 Makefile，进行编译，先 make clean，然后 make。

(5) 把宿主机和开发板用串口线和对接线相连，在宿主机上输入命令 minicom。

(6) ctrl－c 进入命令行，lsmod 看现在已经加载的驱动。

(7) insmod 2410IIC_TEST.o，并观察相应的输出。

(8) lsmod 看看该驱动程序是否已经加载。

(9) rmmod 2410IIC_TEST 卸载相应的驱动程序，并观察相应的输出。

加载时正确的输出如下：

```
---------->1
The writing data is 8aa8
---------->2
The reading data is 8aa8
The test……insmod …….is    OK!
```

卸载时正确的输入是

```
The test.....rmmod...……is    OK!
```

2. 通过 IIC 总线对 UDA1380 进行初始化，编写 uda1380_init 函数

实验步骤：

(1) 读 command 数组，该数组中的数三个一组；每组的第一个是 UDA1380 的寄存器的地址，后两个是往该寄存器里写入的值。

(2) 运用上面编写好的_Wr24C080 函数已经设置好的 command 数组，对 UDA1380 的各个寄存器进行初始化，并把该函数加入到 2410IIC.c 文件里。

(3) 执行命令 make clean，然后 make。

(4) 把宿主机和开发板用串口线和对接线相连，在宿主机上输入命令 minicom。

(5) insmod 2410IIC_INIT1380.o，并观察相应的输出信息。

(6) lsmod 看看该驱动程序是否已经加载。

(7) 如果驱动加载成功，用音频对接线连接宿主机的音频输出端口和板子上的 Line 口，把耳机接在板子上的 Phone 口。

(8) 在宿主机上面的相应目录下可以看到 song.wav，输入命令 play song.wav，带上耳机可以听到歌声。

(9) 如果对声音大小等不满意，可以在 command 数组的相应位进行修改，这里就不做介绍。

四.思考题

1. UDA1380 的寄存器地址均是一个字节，而寄存器的大小均是 16 个 bit；但在下面的实时时钟实验中，所用的芯片 X1227 的寄存器地址是两个字节，而寄存器的大小均是 8 个 bit，如何把上面的读写函数改成适用于 X1227 芯片的读写函数？

实验八：实时时钟实验

一.实验目的

1. 了解 X1227 芯片的功能
2. 掌握通过 IIC 总线实现对 X1227 的控制，进一步加深对 IIC 总线接口的认识
3. 了解实时时钟的基本原理，掌握 X1227 芯片中实时时钟的应用

二.实验原理和说明

1. X1227 芯片介绍

X1227 是美国 XICOR 公司推出的实时时钟（RTC）芯片。与其它 RTC 芯片相比，X1227 除有基本的时钟和报警功能外，还有 4K 位 E2PROM 存储器和复位输出、电压监控、看门狗定时、频率输出等功能。

X1227 可以准确地用秒、分、时、日、星期、月、年来显示时间和日期，具有世纪字节，解决了两千年问题，自动实现闰年调整；有 2 路报警，可设置为按秒、分、时、日、月和星期任意组合的定时报警；内部的 4K 位 E2PROM 存储器，可用于存储用户的设置参数或其它数据，其内容在电源失效时不会丢失；采用 IIC 总线与单片机接口，一次可传送多个字节的数据，数据传送的速率为 400kHz；片内的看门狗定时器可在 0.25s、0.75s、1.75s 和关闭之间调整；还提供一个备用电源输入引脚（Vback），接一电池作为备用电源，可在主电源（Vcc）失效时保证芯片正常工作和时钟的连续运行。X1227 因其计时准确、体积小、功能强，且与单片机接口方便、性价比高，得到了广泛的应用。芯片结构图如下图 1 所示。

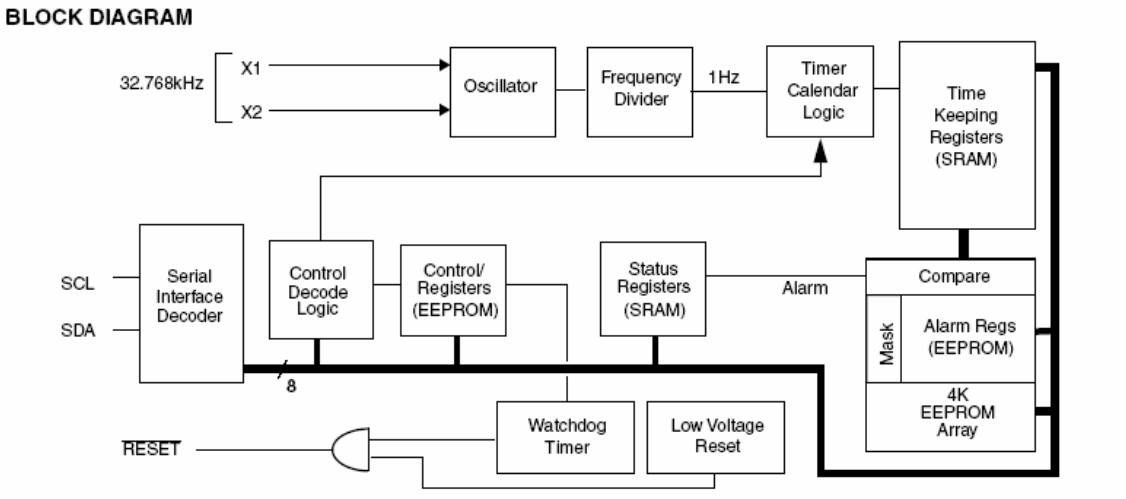


图 1 X1227 芯片结构图

1.1 X1227 芯片的特点

(1) 可选的看门狗定时器(0.25s，0.75s，1.75s，关闭)
(2) 上电复位(250ms)
(3) 低电压复位
(4) 与 IIC 兼容的的两线接口，传输速率可达 400kHZ
(5) 512 x 8 位EEPROM

(6) 具有内部切换电路的备用电源输入
(7) 片内振荡器补偿
(8) 实时时钟解决了千年问题

1.2 X1227 芯片引脚定义

X1227 有 SOIC 和 TSSOP 两种封装，如图 2 所示。

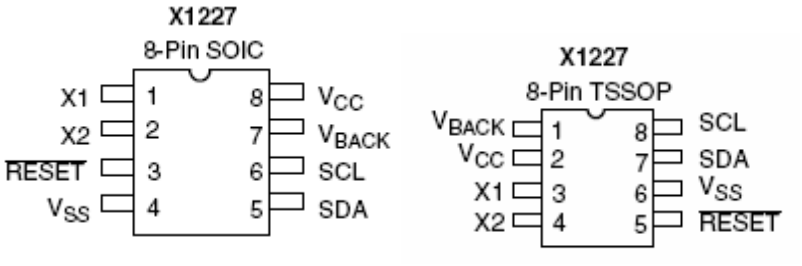


图 2 引脚结构图

引脚定义如下：

串行时钟（SCL）

----SCL 输入端被用来对所有输入和输出器件的数据进行计时。该输入端的缓冲器总是激活的（无门栅）。IIC 总线的时钟输入端。

串行数据端（SDA）

----SDA 是一个双向引脚,用于向器件输入或输出数据.它是一个漏极开路输出,可以与其它漏极开路或集电极开路。输入缓冲器总是激活的（无门栅）。IIC总线的数据输入/输出端。

漏极开路输出需要使用上拉电阻。输出电路使用一个斜率控制的下拉来控制输出信号的下降时间。电路被设计成400kHz速率的2线接口。

VBACK

----这个输入端为器件提供一个备用电源电压。当VCC电源失效时，VBACK为器件提供电源。

RESET输出

----这是复位信号输出端。这个信号告诉主处理器，看门狗超时时间已到或是电压跌落到固定的VTRIP门限以下，是一个漏极开路低有效输出端。若该输出端未用，则用5KΩ电阻上拉或接到VSS。

X1、X2

----X1和X2脚分别为用作片内振荡器的反相放大器的输入和输出端，需要一个32.768kHz石英晶体。建议使用Citizen CFS206-32.768KDZF 型的晶体晶体。晶体的作用是为时钟/振荡器提供时间基准。内部补偿电路也包括在组成一个完整的振荡器电路之内。

在我们的 RUIJIARM9-EDU 板子上的连接原理图见图 3 所示：

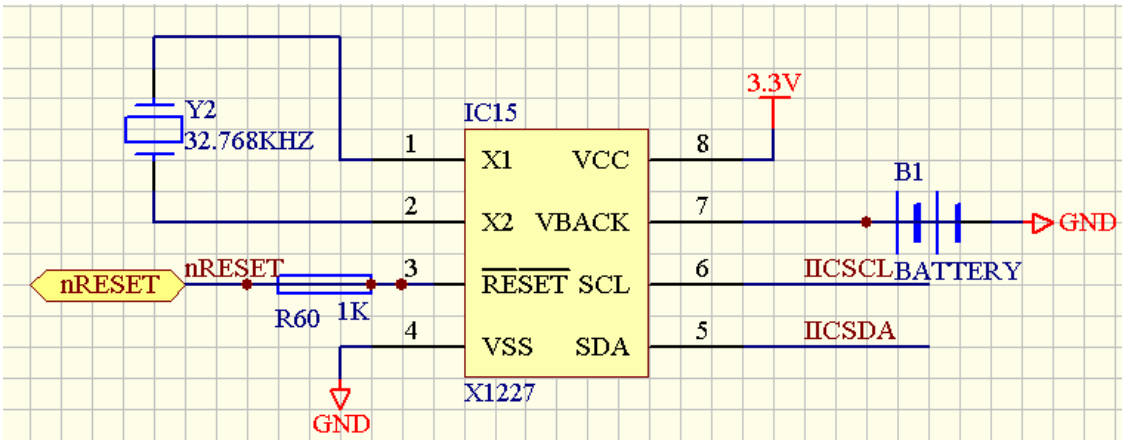


图 3 原理图

2. X1227 工作原理

2.1 IIC 总线的通信协议

X1227 通过串行的 IIC 总线实现与系统的通信，SDA 用来接受和发送数据，SCL 接收产生的同步脉冲，当 SCL 为低时 SDA 上的数据变化，SCL 为高时表明 SDA 上的数据可以接收。

任何命令的执行必须有一个启始位，通信的启始位条件是：当 SCL 为高时，SDA 线上有一个下降沿。任何命令的结束传输必须有一个停止位，通信的停止位条件是：当 SCL 为高时，SDA 线上有一个上升沿。详细介绍可以参见 IIC 总线接口实验。

系统的 IIC 总线上可同时接多个设备，因此每个 IIC 设备都有自己的地址，X1227 也不例外，X1227 有两个从设备地址，1010 为访问 4KB EEPROM；1101 为访问 CCR(时钟/控制寄存器)，用于控制 RTC 和 WatchDog。

完成一次字节访问的操作，一共由四个字节组成，一个从地址字节，一个要访问的地址字（16 位），一个 8 位的操作数，从地址字节最后一位决定了此次操作是读还是写，读是 1，写是 0。

2.2 电源控制操作

电源控制电路接受VCC和VBACK输入,当VCC< VBACK-0.2V时,电源控制电路将电源切换至VBACK。当VCC超过VBACK时它将切换回VCC，见图4所示。

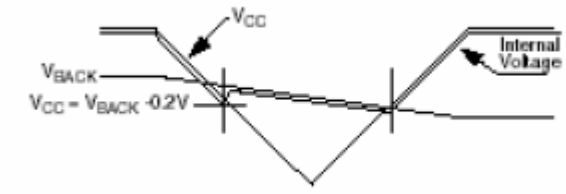


图4 电源控制过程

2.3 X1227 的主要功能

该芯片的主要功能有三个：实时时钟（RTC）,看门狗定时器（WatchDog Timer）和 E2PROM。

实时时钟：通过 X1227 的 IIC 接口可将时间写入 X1227 的实时时钟寄存器中，注意在 CCR 中存放的是 BCD 码，如在 CCR 的 HR(小时)寄存器写入 23 点时，必须写为 1010 0011，读 RTC 的时钟也得到 BCD 码，使用前必须转换。

看门狗定时器：通过写 CCR 中的 BL(块保护)寄存器中的 WD1、WD0 两位实现对 WatchDog 的

设置，控制喂狗的最短时间，注意上电缺省为 00，在初期的调试阶段，必须先用程序将其改为 11，使 WatchDog 无效，系统才能在不喂狗时正常工作。使 WatchDog 定时器清零采用的喂狗指令是通过 IIC 接口送一个启始位给 X1227，当 X1227 进入 Stand By 模式必须再送一个停止位，两个信号的时间间隔要小于 WatchDog 定时器设置的时间，才可将 WatchDog 定时器清零。

EEPROM ：X1227 还为系统提供了 4K EEPROM 可用于存放自检程序，或系统关键的数据，当系统掉电时关键数据不会丢失。

2.4 X1227 的寄存器分布

Addr.	Type	Reg Name	Bit								Range	Default
			7	6	5	4	3	2	1	0(optional)		
003F	Status	SR	BAT	AL1	AL0	0	0	RWEL	WEL	RTCF		01h
0037	RTC (SRAM)	Y2K	0	0	Y2K21	Y2K20	Y2K13	0	0	Y2K10	19/20	20h
0036		DW	0	0	0	0	0	DY2	DY1	DY0	0-6	00h
0035		YR	Y23	Y22	Y21	Y20	Y13	Y12	Y11	Y10	0-99	00h
0034		MO	0	0	0	G20	G13	G12	G11	G10	1-12	00h
0033		DT	0	0	D21	D20	D13	D12	D11	D10	1-31	00h
0032		HR	MIL	0	H21	H20	H13	H12	H11	H10	0-23	00h
0031		MN	0	M22	M21	M20	M13	M12	M11	M10	0-59	00h
0030		SC	0	S22	S21	S20	S13	S12	S11	S10	0-59	00h
0013	Control (EEPROM)	DTR	0	0	0	0	0	DTR2	DTR1	DTR0		00h
0012		ATR	0	0	ATR5	ATR4	ATR3	ATR2	ATR1	ATR0		00h
0011		INT	Unused									
0010		BL	BP2	BP1	BP0	WD1	WD0	0	0	0		00h
000F	Alarm1 (EEPROM)	Y2K1	0	0	A1Y2K21	A1Y2K20	A1Y2K13	0	0	A1Y2K10	19/20	20h
000E		DWA1	EDW1	0	0	0	0	DY2	DY1	DY0	0-6	00h
000D		YRA1	Unused - Default = RTC Year value (No EEPROM) - Future expansion									
000C		MOA1	EMO1	0	0	A1G20	A1G13	A1G12	A1G11	A1G10	1-12	00h
000B		DTA1	EDT1	0	A1D21	A1D20	A1D13	A1D12	A1D11	A1D10	1-31	00h
000A		HRA1	EHR1	0	A1H21	A1H20	A1H13	A1H12	A1H11	A1H10	0-23	00h
0009		MNA1	EMN1	A1M22	A1M21	A1M20	A1M13	A1M12	A1M11	A1M10	0-59	00h
0008		SCA1	ESC1	A1S22	A1S21	A1S20	A1S13	A1S12	A1S11	A1S10	0-59	00h
0007	Alarm0 (EEPROM)	Y2K0	0	0	A0Y2K21	A0Y2K20	A0Y2K13	0	0	A0Y2K10	19/20	20h
0006		DWA0	EDW0	0	0	0	0	DY2	DY1	DY0	0-6	00h
0005		YRA0	Unused - Default = RTC Year value (No EEPROM) - Future expansion									
0004		MOA0	EMO0	0	0	A0G20	A0G13	A0G12	A0G11	A0G10	1-12	00h
0003		DTA0	EDT0	0	A0D21	A0D20	A0D13	A0D12	A0D11	A0D10	1-31	00h
0002		HRA0	EHR0	0	A0H21	A0H20	A0H13	A0H12	A0H11	A0H10	0-23	00h
0001		MNA0	EMN0	A0M22	A0M21	A0M20	A0M13	A0M12	A0M11	A0M10	0-59	00h
0000		SCA0	ESC0	A0S22	A0S21	A0S20	A0S13	A0S12	A0S11	A0S10	0-59	00h

表 1 X1227 时钟/控制寄存器

时钟/控制寄存器位于由EEPROM阵列分离出来的一个区域，它只能跟随在从字节“1101111x”之后才能访问，并且只能在地址[0000h:003Fh]读出或写入。

X1227 对时钟的访问和设置都是通过时钟/控制寄存器 CCR 来实现的。相应的寄存器分成 5 段：

(1) 报警寄存器 0，非易失性 E2PROM 存储器；

(2) 报警寄存器 1，非易失性 E2PROM 存储器；

(3) 控制寄存器（Control）为 4 字节，地址 0010H～0013H，非易失性 E2PROM 存储器；

(4) 实时时钟（RTC）为 8 字节，地址 0030H～0037H，易失性 RAM 存储器；

(5) 状态寄存器（Status）为 1 字节，地址为 003FH，易失性 RAM 存储器。

从第1至第3段是非易失性而第4和第5段是易失性的。每个寄存器通过缓冲器进行读和写。非易失性部分（或RTC的计数器部分）只有在RWEL被置位并且当一次有效的写操作和停止位之后才能更

新。每次只能对CCR的一个段进行连续读或页面写操作。对另一段访问需要一次新的操作。连续的读或写一旦到达段的末端，将返回到段的开始再继续。读或页面写可以在CCR的任何地址开始。

第5段是一个易失性寄存器。不需要先设置RWEL位再写状态寄存器。第5段只支持单字节的读或写。连续读或写该段将终止操作。

任何时候通过完成一次在CCR中任何地址的随机读可以读出CCR的状态。这将返回那个寄存器地址的内容。其它的寄存器可通过完成一次连续读来读出。读指令将所有的时钟寄存器的内容锁存到一个缓冲器。因此时钟的更新不改变被读出的时间。对CCR的连续读不会导致从存储器阵列中输出数据。在读结束时，主机提供一个停止条件，以结束操作并释放总线。在读CCR之后，地址保留在先前的地址+1，因而用户可以执行CCR的当前地址读并继续读下一个寄存器。下面将对我们用到的寄存器进行详细的介绍。

2.5 实时时钟寄存器

2.5.1 2000年(Y2K)

X1227有一个世纪字节，当年字节从99变为00时,它从19变为20。Y2K字节只有19或20两个值。

2.5.2 星期寄存器（DW）

该寄存器提供星期中的日期状态。它用三位DY2至DY0来表示星期中的7天。计数器不断地作0-1-2-3-4-5-6-0-1-2-...的循环。数字值分配到星期中的某日是任意的，可以由系统软件设计者决定。时钟缺省值规定0=星期天。

2.5.3 时钟/日历寄存器（YR、MO、DT、HR、MN、SC）

这些寄存器采用BCD码表示时间。其中SC（秒）、MIN（分）的范围为0至59 ，HR（时）在带有AM或PM显示（H21位）时为1至12，或者为0至23（当MIL=1），DT（日期）为1至31，MO（月）为1至12，YR（年）为0至99。

2.5.4 24小时时间

如果HR寄存器中的MIL位为“1”， 则RTC使用24小时格式。如果MIL位为“0”， 则RTC使用12小时格式，这时位H21用作AM/PM指示，当H21为“1” 显示PM。时钟缺省为H21=0的标准时间。

2.5.5 闰年

闰年加一个2月29日，它是由年份数能被4除尽决定的。年份数能被100除尽不是闰年，除非它也能被400除尽。这表明2000年是闰年而2100年不是，X1227不校正2100年为闰年。

2.6 状态寄存器

状态寄存器位于RTC区，地址003Fh。这是一个易失性寄存器，它用来控制WEL和RWEL的写使能锁存和读两个电源状态和两个报警位。这个寄存器独立于存储阵列和时钟/控制寄存器。

Addr	7	6	5	4	3	2	1	0
003Fh	BAT	AL1	AL0	0	0	RWEL	WEL	RTCF
Default	0	0	0	0	0	0	0	1

表2 状态寄存器

BAT：电池供电－－－易失性

这位置“1”表示器件由电池VBACK供电而不是由VCC 。这是由硬件置位/复位的只读位。

AL1, AL0：报警位－－－易失性

这两位显示报警1或报警2是否与实时时钟匹配。如果有某一个匹配了，则相应位被置“1”。在状态寄存器读操作中最后一个数据位的下降沿复位该标志。注意：当SR读时只有被置位的AL位会被复位。在一次SR读操作期间发生的一次报警置位一个报警位，并将保持置位到读操作完成之后。

RWEL：寄存器写使能锁存－－－易失性

该位是一易失性锁存位，上电时为“低”（禁止）状态。在向任何时钟/控制寄存器写入之前RWEL位必须先置“1”。写RWEL位并不引起一次非易失性写周期，因此在停止（stop）条件之后器件可以立刻准备下一次操作。向CCR写入需要RWEL 和WEL这两位以一定的步骤都被置位。

WEL：写使能锁存－－－易失性

WEL位控制在一次写操作时对CCR和寄存器阵列的访问。该位是易失性锁存位，上电时处于“低”（禁止）状态。当WEL位为“低”时，对CCR或任何阵列地址写入都将被忽略（在数据字节之后将不发生应答）。通过对状态寄存器的WEL位写“1”对其它位写“0”来使WEL位置1。一旦被置1，WEL保持置1，直到被复位为0（通过向状态寄存器的WEL位写“0”， 其它位也写入“0”）或者直到器件再次上电，写WEL位并不引起一次非易失性写周期，因此在停止（stop）条件之后器件可以立刻准备下一次操作。

RTCF：实时时钟失效位－－－易失性

在全部电源失效后该位被置“1”。这是一个由硬件置位的只读位，在器件失去全部电源后再上电时该位被置位。该位的置位与先加VCC还是VBACK 无关。失去其中一个或另一个电源并不导致RTCF位的置位。向RTC的第一次有效写（只要写一个字节即可）即能将RTCF位复位为“0”。

不用的位

位3和位4没有使用，但在这些位必须置“0”。当向SR读出数据字节时，将包含这二位的“0”。

2.7 控制寄存器

块保护位－－－BP2、BP1、BP0（非易失性）

块保护位BP2、BP1、BP0决定了阵列中的哪些块是写保护的。对存储器中一个被保护的块写入被忽略。块保护位可以对阵列的8个段中的某个段提供写保护，如表3说明。

BP2	BP1	BP0	Protected Addresses X1227	Array Lock
0	0	0	None	None
0	0	1	180_h - $1FF_h$	Upper 1/4
0	1	0	100_h - $1FF_h$	Upper 1/2
0	1	1	000_h - $1FF_h$	Full Array
1	0	0	000_h - $03F_h$	First Page
1	0	1	000_h - $07F_h$	First 2 pgs
1	1	0	000_h - $0FF_h$	First 4 pgs
1	1	1	000_h - $1FF_h$	First 8 pgs

表3 块保护位

看门狗定时器控制位

WD1、WD0这二位控制看门狗定时器的超时时间，各可选项见表4。

WD1	WD0	Watchdog Time Out Period
0	0	1.75 seconds
0	1	750 milliseconds
1	0	250 milliseconds
1	1	Disabled

表4 看门狗定时器控制位

3 对X1227芯片的操作

3.1 向时钟/控制寄存器写

要改变时钟/控制寄存器的任何非易失性位，需要以下步骤：

－－写02h至状态寄存器将“写使能锁存”（WEL）位置1。这是一次易失性操作，所以在写后没有延迟。（操作由“开始”（start）引导，由“停止”（stop） 结束。）

－－写06h至状态寄存器，将“寄存器写使能锁存”（RWEL）和（WEL）这两位都置1。这也是易失性操作。在数据字节中的0是必须的。（操作由“开始”（start） 引导，由“停止”（stop）结束。）

－－向时钟/控制寄存器写1至8个字节，写入所需的时钟、报警或控制数据。这个序列由“开始”位引导，需要一个从地址字节“11011110”和CCR中的一个地址，并由“停止”位结束。向CCR写会改变EEPROM的值，这会启动一次非易失性的写周期且要用10ms来完成。对未定义的区域写没有影响。完成一次非易失性写周期将复位RWEL位，所以对CCR内容作另一次改变时，必须重复以上初始化序列。如果由于任何原因而上述序列没有完成（例如，发送一个不正确的位数或发送“开始”而未发送“停止”等），则RWEL位未被复位，器件保持在激活方式。

－－写全“0”至状态寄存器，将WEL和RWEL位都复位。

－－在前面的任意操作间发生一次读操作，将不影响寄存器的写操作。

－－RWEL和WEL位可以通过写“0”至状态寄存器来复位。

3.2 器件寻址

主机在发出开始条件后必须跟着输出一个从地址字节。从地址字节的前4位规定了访问EEPROM阵列还是访问CCR。“1010”表示访问EEPROM 阵列，而“1101”表示访问CCR。从地址字节的位3至位1规定了器件的选择位，它们是“111”。从地址字节的最后一位定义了要完成的操作。当R/*W*位为1时，选择了读操作；为0 时选择写操作，见图5。

从SDA总线输入了整个的从地址字节之后，X1227将器件辨识符和器件选择位与“1010111”或“1101111”比较。如果比较正确，器件输出一个应答至SDA线。

在从地址字节之后跟随一个两字节的字地址。这个字地址可以由主机提供也可以从内部计数器获得。在上电时内部地址计数器被设置为地址0h， 因此EEPROM 阵列的当前地址读从地址0开始。当需要时，如同随机读那样，主机必须如下图5所示提供两字节的字地址。

在随机读操作时，“伪”写操作中的从地址字节，必须与“读操作”时的从地址字节一致。即如果是从阵列中随机读，则在两种情况下的从地址字节必须为1010111x。同样的如果是对时钟/控制寄存器（CCR）的随机读，则两种从地址字节必须1101111x。

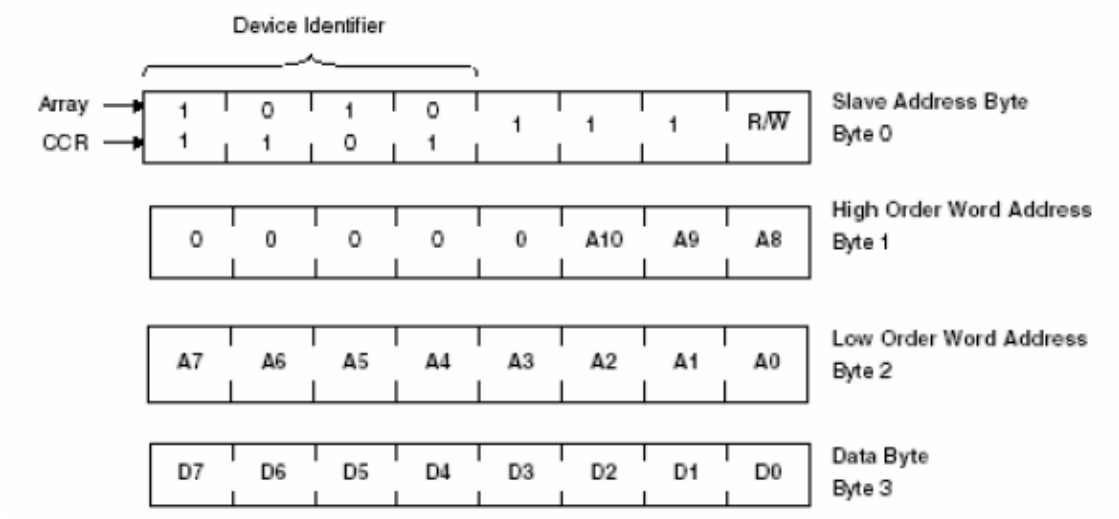


图5 从地址、字地址和数据字节（64字节页面）

3.3 X1227 写操作

写操作分为字节写和页面写两种方式，下面分别予以介绍：

3.3.1 字节写

要进行一次写操作，器件需要从地址字节和字地址字节。这使得主机可以访问存储器阵列或CCR中的任何一个字。（注意在写CCR之前，主机必须以两个前导的操作先向状态寄存器写02h，然后写06h以使能写操作。见“向时钟/控制寄存器写”一节中的说明。）收到每个地址字节后，X1227响应一个应答。在两个地址字节都收到之后，X1227 等待8位数据。在收到8位数据之后，X1227再响应一个应答。

然后主机通过产生一个停止条件来终止传送。X1227接着开始一次将数据写入非易失性存储器的内部写周期。在内部写周期时，器件的输入是禁止的，所以器件将不响应从主机来的任何请求。SDA输出端这进处于高阻状态，见图6。

向存储器的保护块写入是无效的，但仍会收到一个应答。在写命令结束时，X1227将不启动一次内部写周期，并继续应答命令。

字节模式如图 6 所示：

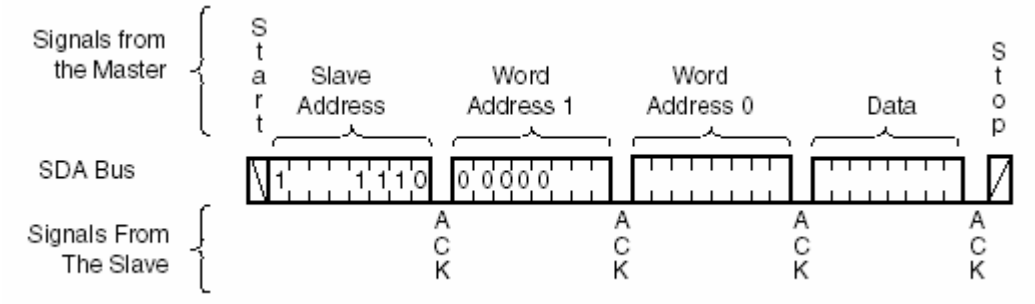


图 6 字节写时序

3.3.2 页面写

X1227 具有页面写操作功能。它可以以字节写操作同样的方式启动，但是在第一个数据字节传送之后不结束写周期，主机可以发送多于63 个字节至存储器阵列和多于7 个字节至时钟/控制寄存器（注意：在向CCR写入之前，主机必须以两个前导的操作写02h，然后写06h至状态寄存器以便使能写操作。见“向时钟/状态寄存器写”一节中的说明。）

在收到每个字节后，X1227响应一个应答，而内部将地址加1。当计数器达到该页的末尾时，它自动地“返回”到该页的首地址。这意味着主机可从某一页的任何位置开始向存储器阵列写64个字节或向CCR写8个字节。如果主机从存储器某页的地址40开始写入30个字节，则前23个字节写在地址40至63，而后的7个字节写在0至6。然后，地址计数器将指向该写入页的地址7。如果主机要写多于1 页的最大字节数，则前面已写入的数据将被新写入的数据覆盖，一次覆盖一个字节。

主机通过发出停止条件来终止数据字节的传送，这引起X1227启动非易失性写周期。同字节写操作一样，所有的输入都被禁止直到内部写周期完成，关于地址、应答和数据传送序列请参见图7、8。

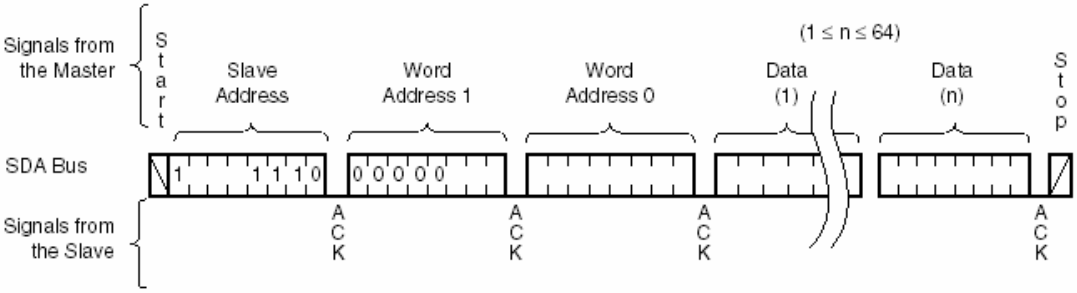


图 7　页面写时序

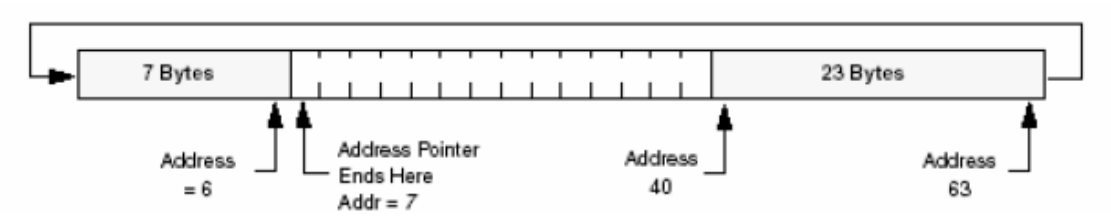


图 8　从地址40开始对一个64字节的页面写30个字节

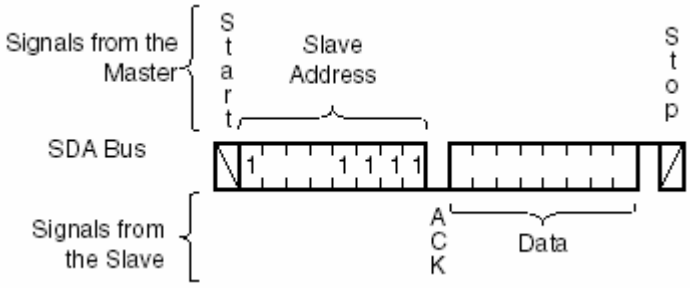
3.4 X1227 读操作

X1227 有三种基本的读操作：当前地址读(Current Address Read)、随机读(Random Read)和序列读(Sequential Read)。

3.4.1 当前地址读

X1227 内部有一个地址计数器，指向上次所读的最后一个数据的地址的下一位。因此，如果上次所读的最后一个数据的地址是 n，下次读操作将从地址 n＋1 开始读数据。当系统重启以后，这个 16bit 的地址位将被初始化为 0。这样，当系统重启后，当前地址读操作可以读取 memory 里的全部数据。当收到从设备地址和一个 R/W bit:1 以后，X1227 响应一个 ACK 信号，然后发送 8 个 bit 的数据。主设备通过发出一个 stop 停止信号来中止读操作。见图 9 所示。

图 9 当前地址读时序图



3.4.2 随机读

随机读操作允许主机访问X1227中的任何地址。在发出将R/*W* 位置1的从地址字节之前，主机必

须首先完成一次“伪”写操作。

主机发出开始条件和从地址字节，收到一个应答，然后发出字地址字节。在收到每个字地址字节的应答后，主机立即发出另一个开始条件和将R/*W*位置1的从地址字节。之后器件发出一个应答接着就是8位数据。主机要终止读操作时就不响应一个应答而发一个停止条件。关于地址应答和数据传送序列请参见图10。

有一种类似的操作称为“设置当前地址”，器件设置地址后在图10中第二个开始条件处主机再发一个停止条件。停止条件后X1227即进入等待方式，并且直到在检测到一个开始条件前对总线上所有的活动都忽略。这个操作在地址计数器中加载了新的地址。下一次当前地址读操作将从新的加载地址读出。这种读操作对主机知道它所需读出的下一个地址但数据尚未准备好的情况是有用的

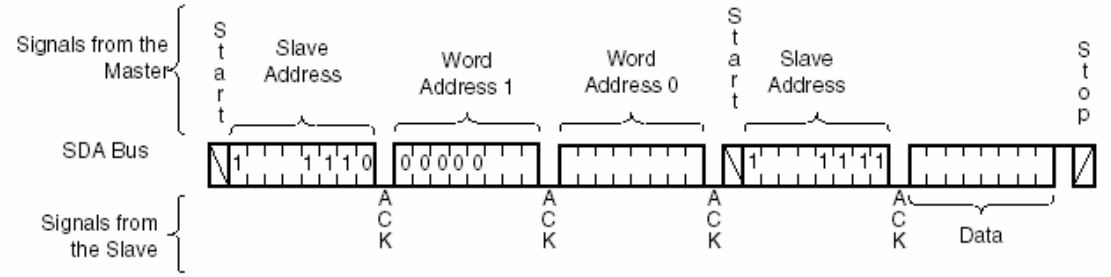


图 10 Random Read 模式

3.4.3 顺序读

从上面两种模式可以看出，无论是哪种模式，每次都只是读取其中的一个 byte 的数据。Sequential Read 则可以同时读取多个 byte 的数据。该模式可以是仿照 Current Address Read 模式，也可以仿照 Random Read 模式。和上面两种模式的最大区别就是该模式在收到一个 byte 的数据后，主设备响应 ACK 信号，告知 X1227 它需要额外的数据，这样从设备不停的向主设备发送数据。最后主设备通过发送一个 STOP 信号中止该次读操作。

4. X1227 中的实时时钟的应用

X1227 实时时钟（RTC）外部仅使用 1 个 32.768kHz 晶体来保持年、月、星期、日、时、分和秒的精确的内部表示。启动读命令并指定对应于 RTC 寄存器的地址可以读 RTC，也可以通过写 RTC 寄存器来设置时间和日期。模拟微调寄存器 ATR（低 6 位）用来调整 X1 和 X2 引脚间的片内负载电容，为 5～39.5pF，这将使晶体选择有较大的余地。数字微调寄存器 DTR（低 3 位）用来调整 RTC 的误差，达到长时间的高精度。

状态寄存器 SR 中复位读标志位 RWEL 为寄存器写使能锁存，位 WEL 为写使能锁存，上电时均为“0”，禁止状态。注意：要对 CCR 或存储器进行任何非易失性写操作，须首先写“02H”至 SR，将 WEL 位置“1”，其次写“06H”至 SR，将 RWEL 和 WEL 都置“1”，然后才能写实际数据到 CCR 或存储器。

实时时钟用一个外部的精震来保持一个精确的时间表示。当 X1227 的电源供应 Vcc 和 Vback 都不能正常供电的时候，该时钟将不会增加，直到向该时钟寄存器写入一个 byte。

4.1 写实时时钟

通过向 RTC 寄存器里写数据来设置相应的时间和日期。为了避免一个不完全的写操作改变了当前的时间，在向 RTC 寄存器写入数据之前，当前的时间被存放在一个缓存区里，而时钟继续运行，新的数据则代替缓存区里的数据。当有效写操作结束时，RTC 更新当前的时间。

根据上面的写操做介绍，写操作分为字节写和页面写两种方式。下面将采用字节写时序进行操作。

4.2 读实时时钟

因为时钟连续的运行，而一个读操作要花费一定的时间，这样在读操作的过程中，时钟是可能改变的。所以在这个设备中，读命令先将当前时间锁存起来，以避免在读过程中时间的值发生改变。这里采用随机读时序进行操作。

4.3 X1227 中 RTC 在 RUIJIARM9-EDU 开发板上的功能

RUIJIARM9-EDU 开发板上跑 ARMLinux，当系统每次断电后重新启动，必须通过读取 X1227 的时间来更新系统时间，所以我们的实验内容就是把系统时间和 RTC 挂接起来。

5. 程序解读（实验一）

在 2410Iic.c 的文件里中，address_map 函数、Run_IicPoll 函数、IicPoll 函数都和 I2C 实验中的完全一样，这是因为无论是 UDA1380 还是 X1227，I2C 的时序都是完全一样的。重要的区别是在读写函数的设计上。

下面我们看一下_Rd2410Iic 函数：

```
void _Rd2410Iic(U32 slvAddr，U32 addr，U8 *rdata)
{
    //X1227 random read mode
    _iicMode   =   ETRDADDR；
    _iicPt       ＝ 0；
    _iicData[0]   =   addr/256；
    _iicData[1]   =   addr；            //这个是最大的区别
    _iicDataCount   =   2；
    rIICDS    =   slvAddr&0xfe；
    rIICSTAT = 0xf0；

    while(_iicDataCount!=-1)
        Run_IicPoll()；

    _iicMode   =   RDDATA；
    _iicPt   =   1；
    _iicDataCount   =   1；

    rIICDS    =   slvAddr|0x01；
    rIICSTAT   =   0xb0；                    //Master Rx,Start
    rIICCON   = 0xaf；                       //Resumes IIC operation.
    while(_iicDataCount!=-1)
        Run_IicPoll()；

    *rdata = _iicData[2]；              //这个是最大的区别
}
```

下面是 Test_Iic 函数：

```
int Test_Iic(void)
{
```

```
    unsigned int save_E，save_PE；
    static U8 rdata；
    static U8 wdata；
    static U32 I；
    address_map()；

    save_E    = rGPECON；
    save_PE   = rGPEUP；

    rGPEUP   |= 0xc000；                          //Pull-up disable
    rGPECON |= 0xa00000；          //GPE15：IICSDA , GPE14：IICSCL

    rIICCON   = (1<<7) | (0<<6) | (1<<5) | (0xf)；
     rIICSTAT = 0x10；                  //IIC bus data output enable(Rx/Tx)

   printk("--------->1\n")；
   wdata   =   0x21；  //设置为 21
   i   =   0x0030；     //设置秒寄存器
  _Wr2410Iic(0xde，0x003f，0x02)；  //两个必须的步骤之一
  _Wr2410Iic(0xde，0x003f，0x06)；  //两个必须的步骤之二
  _Wr2410Iic(0xde，i，wdata)；        //写寄存器
  _Wr2410Iic(0xde，0x003f，0x0)；
  printk("The writing data is：%x \n"，wdata)；

 printk("--------->2\n")；
 _Rd2410Iic(0xdf，i，&(rdata))；
 printk("The reading data is：%x \n"，rdata)；

 rGPECON   =   save_E；
 rGPEUP   =   save_PE；
 return 0；
}
```

在 RTC_Driver.c 文件里，主要是下面几个函数：
读函数如下：（读函数中，每次读取全部 6 个寄存器）

```
static ssize_t
rtc_rd(struct file *filep，char * buffer，size_t length，loff_t * off)
{
    int   ret   =   0，i；
    U8   rdata；

    dbuf   =   kmalloc(8*sizeof(unsigned char) ，GFP_KERNEL)；

    Init_Iic()；
```

```
    /*Second*/
    _Rd2410lic(0xde，0x30，&(rdata))；
    dbuf[0]  =  rdata；

    /*Minute*/
    _Rd2410lic(0xde，0x31，&(rdata))；
    dbuf[1]  =  rdata；

    /*Hour*/
    _Rd2410lic(0xde，0x32，&(rdata))；
    dbuf[2]  =  rdata；

    /*Date*/
    _Rd2410lic(0xde，0x33，&(rdata))；
    dbuf[3]  =  rdata；

    /*Month*/
    _Rd2410lic(0xde，0x34，&(rdata))；
    dbuf[4]  =  rdata；

    /*Year*/
    _Rd2410lic(0xde，0x35，&(rdata))；
    dbuf[5]  =  rdata；
    copy_to_user(buffer，dbuf，6)；
    kfree(dbuf)；
    return ret；
}
```

写函数如下：（写函数中，可以每次只设置 RTC 中的一个寄存器，也可以同时设置 8 个寄存器）

```
static ssize_t
rtc_wr(struct file *filep,char * buffer,size_t length,loff_t * off)
{
U8 data；
int ic ，ret = 0；
unsigned int reg， mreg，lreg；
dbuf = kmalloc(length*sizeof(char)，GFP_KERNEL)；

copy_from_user(dbuf,buffer,length)；
Init_lic()；

_Wr2410lic(0xde,0x003f,0x02)；
    _Wr2410lic(0xde,0x003f,0x06)；
```

```
for(ic = 0；ic < length；  ic+=3)
{
      mreg = dbuf[ic]；
      lreg = dbuf[ic+1]；
      reg = mreg<<8|lreg；
      data = dbuf[ic+2]；
          _Wr2410Iic(0xde,reg,data)；
}
kfree(dbuf)；
return ret；
}
```

INIT 函数如下：

```
static int __init
RTC1227_init (void)
{
X1227_devfs_dir = devfs_mk_dir(NULL, "X1227", NULL)；
if(!X1227_devfs_dir)
      return –EBUSY；

devfs_register(X1227_devfs_dir,DEVICE_NAME,DEVFS_FL_AUTO_DEVNUM,0,0,S_IFCHR|
S_IRUGO|S_IWUGO,&Fops,NULL)；

printk ("X1227 RTC driver installed OK\n")；
printk ("Test is starting!\n")；
Test_Iic()；     //调用测试函数
return 0；
}
```

EXIT 函数如下：

```
void __exit
RTC1227_exit(void)
{
devfs_unregister(X1227_devfs_dir)；
printk("X1227 RTC driver uninstalled OK\n")；
}
module_init(RTC1227_init)；
module_exit(RTC1227_exit)；
```

6. 程序解读（实验二）

应用程序 RTC.c 文件中

```
int   fd；
int   i = 0；
unsigned char CurTime [6]；          //Current Time
```

```
unsigned char SetTime[] =          //Set Time
{
    0x00，0x30，0x08，   //second
    0x00，0x31，0x08，   //minute
    0x00，0x32，0x08，   //hour
    0x00，0x33，0x07，   //date
    0x00，0x34，0x08，   //month
   0x00，0x35，0x04，   //year
};
//请加入写函数
sleep(10)；
//请加入读函数，并把时间打印出来
```

7. 程序解读（实验三）

把实时时钟和操作系统挂接起来。

在 busybox 的 date.c 文件中：

与实时时钟有关的代码我们都标上了 RJcn add by it 字样。

在该文件头部加入如下代码：

```
int fd；
unsigned char SetTime[] =
        {
                0x00,0x30,0x00,   //second
                0x00,0x31,0x00,   //minute
                0x00,0x32,0x00,   //hour
                0x00,0x33,0x00,   //date
                0x00,0x34,0x00,   //month
                0x00,0x35,0x00,   //year
        };
```

在我们的 //get time 下面加入

```
fd  =  open("/dev/X1227/rtc"，O_RDWR)；
read(fd，CurTime，6)；
CurTime[2]  &=  0x7f；
//这一步的目的是因为时间的第七位是 1 的时候表示 24 小时
tm_time.tm_sec  =  CurTime[0]/16*10  +  CurTime[0]%16；
tm_time.tm_min  =  CurTime[1]/16*10  +  CurTime[1]%16；
tm_time.tm_hour  =  CurTime[2]/16*10  +  CurTime[2]%16；
tm_time.tm_mday  =  CurTime[3]/16*10  +  CurTime[3]%16；
tm_time.tm_mon  =  CurTime[4]/16*10  +  CurTime[4]%16；
tm_time.tm_year  =  CurTime[5]/16*10  +  CurTime[5]%16；
```

这里特别要注意的是 X1227 的时间寄存器均是用 BCD 表示的，而 tm_time 结构中的时间均是二进制表示的，所以要进行转换。

三.实验内容和步骤

1. 根据上面的 X1227 实时时钟的读写操作的介绍，并参考上面给出的读函数，写出相应的 X1227 的 IIC 写函数，并在实时时钟的驱动程序中进行相应的读写测试。（写函数采用字节写方式；读函数采用随机读方式）

实验步骤：

(1) 参考读函数，编写相应的写函数_Wr2410Iic(U32 slvAddr,U32 addr,U8 wdata)

(2) 在 2410Iic.c 里面，Test_Iic 函数的功能使用来测试读写函数是否正确，先设置秒为 21，然后把它读出来，正常情况下，读出的也是 21。

(3) 在 RTC_Driver.c 文件里，在 RTC1227_init 函数中调用 Test_Iic 函数，可以参见上面的程序代码。

(4) 在宿主机上 make。

(5) mount –o nolock 192.168.2.181：/ /mnt （该 ip 应该是宿主机的 IP）

(6) 在开发板子上 insmod RTC.o，看相应的输出结果，正常输出结果应该是：

```
---------->1
The writing data is 21
---------->2
The reading data is 21
```

2．为了测试上面的驱动程序的正确性，我们编写应用程序，该程序的作用是：用户可以对 X1227 设置新的时间；也可以读取当前的时间。

实验步骤：

(1) 按照上面的程序解读，填写读写函数；

(2) SetTime 数组的时间也可以自己设置；

(3) 因为在设置和读取函数中间加入了 sleep（10），所以我们读到的时间，应该比我们设置的时间多 10 秒钟；

3．嵌入式操作系统初始化的时候从 RTC 读取时间来初始化系统时间

我们的思路是这样的，每次操作系统重新启动以后，我们手动的加载 RTC 的驱动程序（当然，也可以把该驱动加入到内核里）；然后通过 busybox 里的 date 命令，调用 RTC 驱动的读写函数，当用户输入 date 命令读取时间时，调用驱动的读函数，读取 X1227 的时间；当用户输入 date 命令设置时间时，调用驱动的写函数，设置 X1227 的时间，同时设置系统的时间。

实验步骤：

(1) 打开/busybox-1.00-pre10/coreutils/date.c 文件

(2) 读我们加入的代码（都有标记），在 //set time 下加入设置时间的代码，可以仿照上面的 get time 的代码

(3) 输入 make 命令，得到可执行文件 busybox

(4) 把它放到 ramdisk 的 application 中

(5) 重新烧写 ramdisk

(6) 输入 date –help，可以看到 date 的命令格式，[MMDDhhmmYY]

(7) 设置时间，如 date 080715202004

(8) 读取时间，date

(9) 在 RTC 上装有备用电池的情况下，断电，重新启动后，再次读取当前时间，看看是否是刚才设置的时间，当然秒和分是会改变的。

四.思考题

1．文中介绍了读写的多种模式，如果用其它的读写模式，IIC 读写函数将要做哪些改动?

2．这里每次读取 RTC 的全部 8 个寄存器，如果用户每次只想读取其中的某一个或某几个寄存器时，驱动程序应该如何编写？

3. 该实验中，为了让学生更好的理解 X1227 寄存器的使用，我们在应用程序中涉及到了 X1227 的寄存器的地址，正常情况下，硬件的寄存器的地址都应该放在驱动程序里面；应用程序通常只是设置时间，如果这样，怎么才能实现？

实验九：看门狗实验

一.实验目的

1. 了解看门狗定时器的基本原理；
2. 掌握 X1227 芯片中看门狗定时器的使用方法；
3. 了解 X1227 芯片中看门狗定时器的应用；

二.实验原理和说明

1. 看门狗定时器的基本原理

在嵌入式系统中，有很多系统处在恶劣的环境（如电压供应变换、静电释放等）中，这很容易造成系统处理器执行错误的程序，使系统进入死循环。在这种环境下，看门狗定时器是一个有用的外围设备，用于捕获和复位已经失去了控制的处理器。

看门狗定时器是一个简单的计数器，经过一个指定的间隔时间来复位微处理器。在一个较好的系统中，软件将定时检测或重新置位看门狗定时器。当系统启动后，看门狗定时器每经过一个预定的的时间间隔来计数。当软件与设备正常工作时，软件通过定时来重置看门狗定时器。当软件和设备无效工作时，看门狗定时器将得不到置位，这样它将不断计数，直到溢出；一旦溢出，它将产生一个复位信号并重新复位系统。

2. X 1227 看门狗定时器的特点

X1227 看门狗定时器可通过向 BL 寄存器中 WD1、WD0 这两位的“写入”，设置为 3 种不同的时间隔或不工作，“00”为 1.75s，“01”为 750ms，“10”为 250ms，“11”为不工作。看门狗启动时，必须在规定间隔内对它进行刷新，方法是在 SCL 线为高时，SDA 线产生下降沿。如果看门狗在规定间隔内没有被刷新，则 RESET 脚变为有效。注意：如果使用开始条件来刷新看门狗定时器，必须跟着一个结束条件以复位 X1227。

可以用 X1227 的 WatchDog 对系统监控，当系统死机或程序异常时，不能及时喂狗，X1227 的 RESET/n 引脚将输出一个低电平信号，使系统可靠复位。关于 X1227 芯片的介绍请参见实时时钟实验的实验原理和说明部分。

特别注意的是上电缺省为 00，在初期的调试阶段，必须先用程序将其改为 11，使 WatchDog 无效，系统才能在不喂狗时正常工作。

3．X1227 看门狗定时器操作

X1227中的看门狗定时器是可以选择的，通过设置WD1和WD0两位，看门狗定时器可以设置3个不同的时间。当看门狗定时器被置为不工作时，看门狗电路被配置为低功耗工作。

当SCL电平高，同时SDA下降沿的时候，看门狗定时器重新启动，这也称为开始条件。重新启动信号重启看门狗定时器计数器，重新将计数的时间设置为最大。如果在看门狗定时器定时超时前检测到一次启动失败，则*RESET* 脚变为有效。正当在复位超时时间内发生重新启动信号，则重新启动将不起作用。当使用一个START信号来更新看门狗定时器时，必须跟着一个STOP位以复位器件回到等待方式。

为了更好的了解其原理，下面具体看一下看门狗时序图，如图 1 所示。

这里设定的定时器时间是 Twdo，当 Trsp<Twdo 的时候，看门狗定时器收到 RESTART 信号，重新开始计数。当 Trsp>Twdo 时，看门狗定时器的 RESET 引脚被激活；在 RESET 被激活的这段时间，

看门狗将将不响应任何的输入信号。

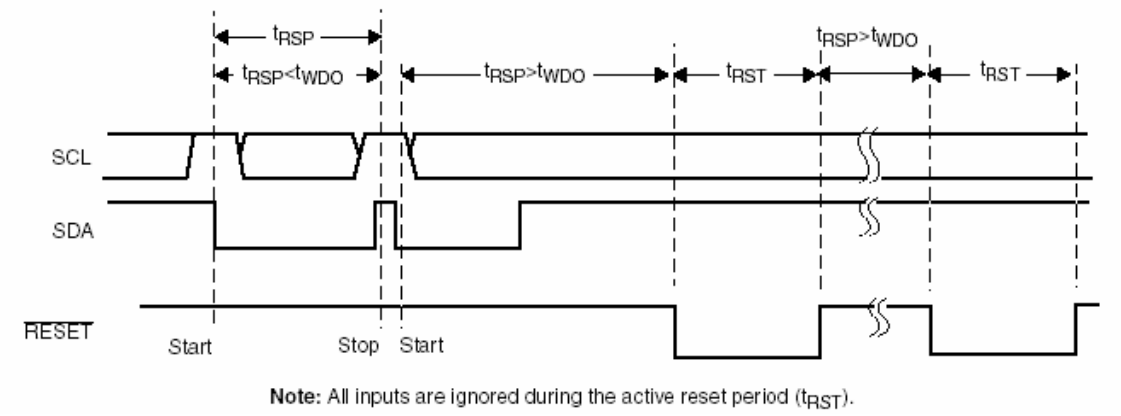


图 1 看门狗时序图

4. CPU 中自带的看门狗的应用

在以前的 2410 板子上，我们在 shell 下面，运行 reboot，系统将重新启动，这是 CPU 自带的看门狗起的作用，下面我们对这个进行分析，首先我们介绍一下相应的寄存器。

Register	Address	R/W	Description	Reset Value
WTCON	0x53000000	R/W	Watchdog timer control register	0x8021

WTCON	Bit	Description	Initial State
Prescaler Value	[15:8]	Prescaler value. The valid range is from 0 to (2^8-1).	0x80
Reserved	[7:6]	Reserved. These two bits must be 00 in normal operation.	00
Watchdog Timer	[5]	Enable or disable bit of Watchdog timer. 0 = Disable 1 = Enable	1
Clock Select	[4:3]	Determine the clock division factor. 00: 16 01: 32 10: 64 11: 128	00
Interrupt Generation	[2]	Enable or disable bit of the interrupt. 0 = Disable 1 = Enable	0
Reserved	[1]	Reserved. This bit must be 0 in normal operation.	0
Reset Enable/Disable	[0]	Enable or disable bit of Watchdog timer output for reset signal. 1: Assert reset signal of the S3C2410X at watchdog time-out 0: Disable the reset function of the watchdog timer.	1

图 2 看门狗控制定时器

Register	Address	R/W	Description	Reset Value
WTDAT	0x53000004	R/W	Watchdog timer data register	0x8000

WTDAT	Bit	Description	Initial State
Count Reload Value	[15:0]	Watchdog timer count value for reload.	0x8000

图 3 看门狗数据寄存器

Register	Address	R/W	Description	Reset Value
WTCNT	0x53000008	R/W	Watchdog timer count register	0x8000

WTCNT	Bit	Description	Initial State
Count Value	[15:0]	The current count value of the watchdog timer	0x8000

图 4 看门狗计数寄存器

第一步：当系统上电后，第一件事就是调用 ppcboot，ppcboot 中相应的代码如下：

```
//下面才是真正的上电复位的启动代码：
/*
* the actual reset code
*/
reset:

/* turn off the watchdog */
#if defined(CONFIG_S3C2400)
#define pWTCON        0x15300000
/* Interupt-Controller base addresses */
#define INTMSK        0x14400008
/* clock divisor register */
#define CLKDIVN       0x14800014
#elif defined(CONFIG_S3C2410)
#define pWTCON        0x53000000       //Watchdog timer control register
/* Interupt-Controller base addresses */
#define INTMSK        0x4A000008
#define INTSUBMSK   0x4A00001C
/* clock divisor register */
#define CLKDIVN       0x4C000014
#define MPLLCON       0x4C000004
#define CLK_CTL_BASE                0x4C000000
#endif

ldr     r0, =pWTCON
mov    r1, #0x0              //Disable the reset function of the watchdog timer
str    r1, [r0]
```

可以看出，在系统以上电做的第一件事情就是使看门狗不起作用。

第二步：当我们运行 reboot，实际是调用了 application/busybox/init/init.c 里的 init_reboot 函数，该函数如下：

```
if((pid = fork()) = =0){
      message(CONSOLE | LOG,   “reboot(magic=%08x\n)…”, magic);
      reboot(magic);
    _exit(0);
```

}
可以看出，该函数又调用了 reboot 函数。

第三步：该 reboot 会调用/kernel/kernel/sys.c 里的 sys_reboot 函数，magic 传递的参数是 01234567 和 LINUX_REBOOT_CMD_RESTART 相同，见下面代码：

```
switch(cmd){
case   LINUX_REBOOT_CMD_RESTART：
       ……..
        pirntk(KERN_EMERG   “Restarting system.\n”)；
        machine_restart(NULL)；
        _break；
```

可以看出，现在接着调用 machine_restart 函数。

第四步：下面我们看一下 machine_restart 函数，该函数定义在 kernel/arch/arm/kernel/process.c 文件里，该函数又调用了 arch_reset(reboot_mode)函数；

第五步：arch_reset 函数是定义在 kernel/include/asm-arm/arch-s3c2410/system.h 文件里，该函数如下：

```
if(mode = = ’s’){
   cpu_reset(0)；
}else{
     WTCNT = 0x100；
     WTDAT = 0x100；
     WTCON = 0x8021；        //使看门狗使能
}
```

现在看门狗已经基本实现了重新启动功能，但是，它存在一个严重的 bug，就是如果对 jffs2 进行了写操作后，reboot 命令将不起作用；所以我们决定采用 X1227 芯片的看门狗，这个我们将在下面进行介绍。

5. X1227 中 WatchDog 在 RUIJIARM9-EDU 开发板上的功能

ARM9-EDU 开发板上跑 ARMLinux，通过调用应用程序 reboot，使 WatchDog 重新启动操作系统。

6．程序解读（实验一）

我们在系统刚启动的时候，在 ppcboot 里使看门狗不使能，具体代码在 ppcboot-2.0.0/common/main.c 文件里，下面我们进行分析。

我们在该文件里加入了如下函数：

```
int Disable_WatchDog(void)；
void _Wr2410Iic(U32 slvAddr,U32 addr,U8 wdata)；
void _Rd2410Iic(U32 slvAddr,U32 addr,U8 *rdata)；
void Init_Iic(void)；
void Run_IicPoll(void)；
void IicPoll(void)；
```

这些函数和实时时钟实验里的代码基本相同，只有在对寄存器进行读写操作时，是直接对寄存器进行赋值，例如(*(volatile unsigned long *)0x54000004) = 0xf0，就是赋值 IIC 状态寄存器位 0xf0。其余完全相同，这里就不再介绍。下面我们重点介绍一下 Disable_WatchDog 函数，它的作用就是使看门狗不使能。

```
static U8 rdata；
static U8 wdata；
static U32 i；
  init_lic()；  //初始化 IIC

wdata = 0x18；
i = 0x0010;；
_Wr2410lic(0xde,0x003f,0x02)；
_Wr2410lic(0xde,0x003f,0x06)；
_Wr2410lic(0xde,i,wdata)；          //写控制寄存器使看门狗不使能
_Wr2410lic(0xde,0x003f,0x0)；
printf("Write %02x to Watchdog ",wdata)；

_Rd2410lic(0xdf,i,&(rdata))；
printf("and it is %02x now\n",rdata)；
```

我们把该函数加在系统启动的最一开始，具体位置见相应的代码。

7．程序解读（实验二）

我们这里分析一下驱动程序，我们将只介绍几个主要的函数，其余的函数和实时时钟相同，这里就不在介绍。

```
printk("--------->1\n")；
wdata = 0x00；
i = 0x0010；
_Wr2410lic(0xde,0x003f,0x02)；
_Wr2410lic(0xde,0x003f,0x06)；
_Wr2410lic(0xde,i,wdata)；  //使看门狗使能，但只喂一次狗
_Wr2410lic(0xde,0x003f,0x0)；
  printk("The writing data is:%x\n",wdata)；
  printk("--------->2\n")；
  _Rd2410lic(0xdf,i,&(rdata))；
  printk("The reading data is:%x \n",rdata)；
```

三.实验内容和步骤

1．在 ppcboot 里使看门狗不使能，这里要求学生理解我们的思路和如何编程。

实验步骤：

(1) 修改 main.c 文件，然后编译；

(2) 烧写 ppcboot，tftp 30008000 ppcboot.bin；fl 1000000 30008000 20000；

(3) 烧写完毕后，重新启动，观察相应的输出，下面是相应的代码：

PPCBoot 2.0.0 (Aug 11 2004 - 10:07:32)
PPCBoot code: 33F00000 -> 33F157A4 BSS: -> 33F189E0
DRAM Configuration:
Bank #0: 30000000 64 MB
Flash: 16 MB
Write 18 to Watchdog and it is 18 now //这句话说明我们的程序起作用了
start linux now(y/n):## Booting image at 30008000 ...
copy kernel done
copy ramdisk done

2. 结合IIC的工作原理,编写看门狗定时器的最简单的驱动程序，即只有module_init和module_exit函数，在 module_init 函数里，调用 iic 里的相应的函数，这里可以选择调用两个函数，一个是Reboot 函数，一个是 Enable_WatchDog 函数，两个函数的功能如下：
(1)Enable_WatchDog 函数里，定时喂狗，观察板子的工作情况，应该正常工作；
(2) Reboot 函数里，只喂一次狗，观察板子的工作情况，板子应该重新启动；
请自己编写 Enable_WatchDog 函数。

四.思考题

1. 为什么要在 ppcboot 里面使看门狗不使能，把这段代码放在内核里可以吗？
2. 列举出看门狗的一些优点。
3. 在 ppcboot 使看门狗不使能，为什么可以直接对寄存器进行操作，而不需要地址映射？

实验十：E2PROM 实验

一．实验目的

1．了解 X1227 芯片中 E2PROM 的作用；
2．掌握对该芯片中的 E2PROM 的读写操作；

二．实验原理和说明

1． X1227 芯片中 E2PROM 的读写操作

1.1 E2PROM 写操作

显然，对 E2PROM 的操作，无论是读还是写，每次只传输一个字节是不切实际的。所以，这里的读操作将采用页面写方式，见下图 1。

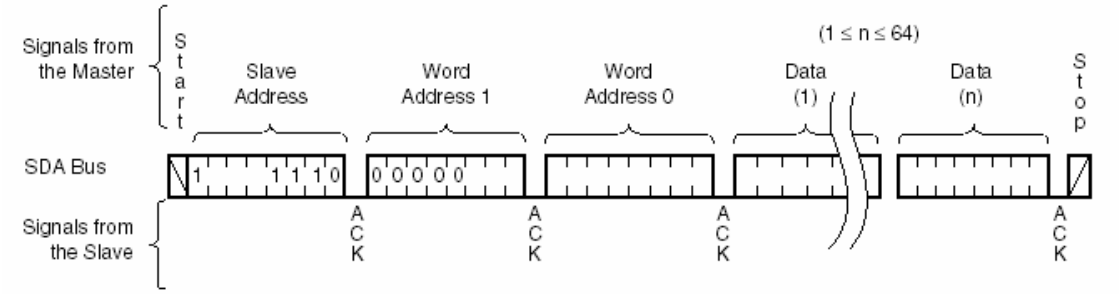


图 1 页面写时序

从图 1 可以看到，X1227 每接收到一个字节，发出一个 ACK。当计数器到达了该页的末尾，它会循环跳到该页的首地址。这意味着，该方式可以每次向 E2PROM 写 64 个字节。例如，如果开始的时候向这个页面的第 40 个地址写入 30 个字节，前 23 个字节将被写到地址 40 到 63 上；后 7 个字节被写到地址 0 到 6 上。地址计数器将指到地址的第 7 位，见图 2 所示。如果向该页写入的字节数大于一个页面说能存放的最大字节数时，过去的数据会被新的数据所覆盖。

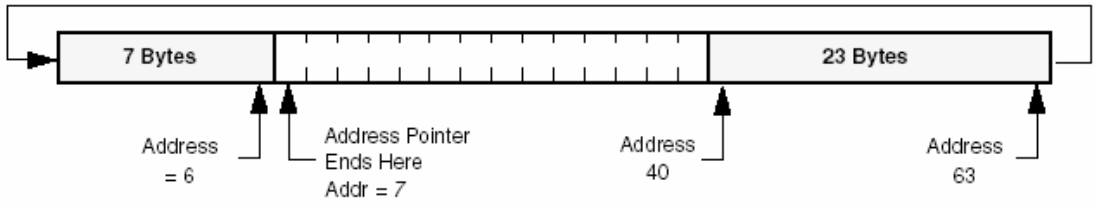


图 2 向一个 64 byte 的页面的第 40 个地址写入 30 个字节

1.2 E2PROM 读操作

这里采用随机读模式和连续读模式结合的方式读取 E2PROM。

随机读模式－－－随机读操作允许主机访问X1227中的任何地址。在发出将R/*W* 位置1的从地址字节之前，主机必须首先完成一次“伪”写操作。主机发出开始条件和从地址字节，收到一个应答，然后发出字地址字节。在收到每个字地址字节的应答后，主机立即发出另一个开始条件和将R/*W*位置1的从地址字节。之后器件发出一个应答接着就是8位数据。主机要终止读操作时就不响应一个应答而发一个停止条件。关于地址应答和数据传送序列请参见图3所示。

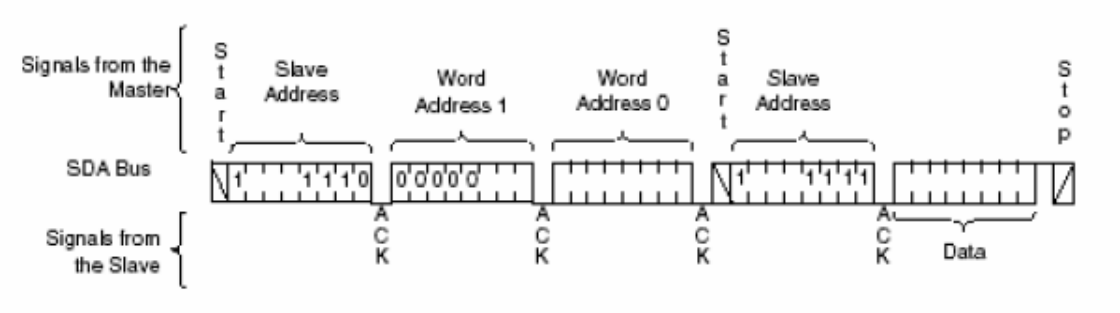


图 3 随机读时序图

连续读模式－－－连续读可以由当前地址读或是随机地址读任一种方法来启动。第一个数据字节可以用其它方式发送；但是现在主机响应一次应答，以表示它需要随后的数据。器件对每一个接收到的应答继续输出数据。主机若要终止读操作，则不响应应答而发出一个停止条件。

数据输出是顺序的，从地址n输出的数据后面跟着从地址n+1输出数据。用于读操作的地址计数器逐次加1，通过所有的页和栏地址，可以在一次操作中将全部存储器的内容连续读出。在地址空间的终端，计数器“返回”到地址空间的开始端，X1227继续在收到每个应答时输出数据。

1.3 特别注意事项

下面，我们先看一下状态寄存器的 WEL 位的介绍。

WEL：写使能锁存－－－易失性

WEL位控制在一次写操作时对CCR和寄存器阵列的访问。该位是易失性锁存位，上电时处于“低”（禁止）状态。当WEL位为“低”时，对CCR或任何阵列地址写入都将被忽略（在数据字节之后将不发生应答）。通过对状态寄存器的WEL位写“1”对其它位写“0”来使WEL位置1。一旦被置1，WEL保持置1，直到被复位为0（通过向状态寄存器的WEL位写“0”， 其它位也写入“0”）或者直到器件再次上电，写WEL位并不引起一次非易失性写周期，因此在停止（stop）条件之后器件可以立刻准备下一次操作。

可以看出，和向 CCR 寄存器写类似，我们在对 E2PROM 进行写操作前，需要使 WEL 位置 1；写完毕后，再把它复位。

2. X1227 中 E2PROM 在 RUIJIARM9-EDU 开发板上的作用

下面首先介绍一下嵌入式网络设备中的 MAC 及 IP 地址的设置特点。

(1) 在嵌入式设备中往往没有硬盘，它的操作系统和应用软件通常是打包放在 Flash 等存储设备中。系统启动时，把 Flash 中的代码释放到内存中，再在内存中运行。比如嵌入式操作系统 linux-2.4.*-rmk9，在用于 S3C2410 这样的带以太网接口的嵌入式设备时，把内核和应用程序代码压成一个映像文件包，在包中有网络部分 MAC 及 IP 地址。但这些 MAC 及 IP 地址的值是在编译映像文件时设定的，而且在编译后的映像文件中的值是不能直观地看到的，它是压缩了的二进制数据，不方便在映像文件中直接更改 MAC 及 IP 地址的值。

(2) 对于使用同一映像文件的嵌入式网络设备，如果不做进一步的处理，其 MAC 及 IP 地址是相同的。这显然不能满足应用，因为不同的设备应该有不同的 MAC 及 IP 地址。而编译生成映像文件往往要用十几甚至几十分钟。对于生产厂家，不可能为每台设备编译一个特定的映像文件。

针对以上问题，我们在 S3C2410 上运行 linux 时，使用了一些特殊的方法来解决它。嵌入式网络设备系统的 MAC 及 IP 地址设置的基本思想是：把 MAC 及 IP 地址存放在 X1227 芯片的 E2PROM 的未用扇区（一般在高扇区），嵌入式操作系统启动后，自动运行一个程序去读取 MAC 及 IP 地址并设置它。

3. 程序解读（实验一）

该实验的 2410iic.c 文件和上面的实时时钟有一定的区别，区别主要在如下几个函数：

_Wr2410Iic(U32 slvAddr，U32 addr，U8 wdata)

_Wr2410Iic_E2PROM(U32 slvAddr，U32 addr，U8 *buffer，U32 length)

_Rd2410Iic_E2PROM(U32 slvAddr，U32 addr，U8 *buffer，U32 length)

Test_Iic（void）

_Wr2410Iic(U32 slvAddr，U32 addr，U8 wdata)函数和上面一样，就是设置状态寄存器，每次只写入一个字节；而_Wr2410Iic_E2PROM 函数和_Rd2410Iic_E2PROM 函数是可以每次向 E2PROM 里面写入或者读出 8 个字节的数据。

下面我们重点看一下_Rd2410Iic_E2PROM 函数：

```
int   i = 0；
_iicMode   =   SETRDADDR；  //写入要读的地址
_iicPt   =   0；
_iicData[0]    =   addr/256；
_iicData[1]    =   addr；
_iicDataCount = 2；

rIICDS    =   slvAddr&0xfe；
rIICSTAT   =   0xf0；                          //Master Tx，Start

while(_iicDataCount!=-1)
    Run_IicPoll()；

_iicMode   =   RDDATA；
_iicPt      =   1；
_iicDataCount   =   length；

rIICDS    =   slvAddr|0x01；
rIICSTAT   =   0xb0；                           //Master Rx，Start
rIICCON   =   0xaf；                           //Resumes IIC operation
while(_iicDataCount!=-1)
    Run_IicPoll()；
for(i = 0；  i<length；i++)
{
     buffer[i] = _iicData[i+2]；       //读入 E2PROM 里面的数据
}
```

Test_Iic 函数如下：

```
unsigned int save_E，save_PE；
unsigned char buffer[3]，read[3]；
unsigned int len，j；
static U32 i；
address_map()；
```

```
save_E    =   rGPECON；
save_PE   =   rGPEUP；

rGPEUP    |= 0xc000；                    //Pull-up disable
rGPECON |= 0xa00000；                  //GPE15：IICSDA ，GPE14：IICSCL

 //Enable ACK, Prescaler IICCLK=PCLK/16, Enable interrupt, Transmit clock value Tx clock=IICCLK/16
rIICCON    = (1<<7) | (0<<6) | (1<<5) | (0xf)；

rIICSTAT = 0x10；                      //IIC bus data output enable(Rx/Tx)

 _Wr2410Iic(0xde,0x3f,0x02)；      //这一步非常的重要

printk("--------->1\n");
buffer[0] = 0x11；
buffer[1] = 0x22；
buffer[2] = 0x33；
len = 3；
i   =   0x0；

//向 E2PROM 里写入数据
_Wr2410Iic_E2PROM(0xae，i，buffer，len)；
for(j = 0；  j<len；j++)
{
        printk("The writing data is：x\n"，buffer[j])；
}

printk("--------->2\n")；

//读 E2PROM 里写入数据
_Rd2410Iic_E2PROM(0xaf，i，read，len)；
for(j = 0；  j<len；j++)
{
        printk("The reading data is：x \n"，ead[j])；
}

rGPECON = save_E；
rGPEUP = save_PE；
```

4. 程序解读（实验二）

编写应用程序，在 E2PROM.c 文件里写入读写函数。

```
#include <stdio.h>
#include <fcntl.h>
```

```
main()
{
    int fd，i；
    unsigned char send[3]， receiver[3]；
    //在这里加入代码，打开设备
    if(fd < 0)
    {
      printf("Error\n")；
      return 0；
    }
    //要写入的数
    send[0] = 0xcc；
    send[1] = 0xbb；
    send[2] = 0xaa；
    //在这里加入代码，首先向 E2PROM 里面写入，然后读出来
    for(i = 0；i<3 ； i++)    //打印上面读出的数
    {
        printf("receive the num[%d] is %x   \n"，I，receiver[i])；
    }
    close(fd)；    //关闭设备
    return 0；
}
```

三.实验内容和步骤

1. 根据上面的 X1227 中 E2PROM 的读写操作的介绍，编写适合于 X1227 E2PROM 的 IIC 读写函数，并在 E2PROM 的驱动程序中进行相应的读写测试。（写操作采用 Page Write 方式；读操作采用 Sequential Read 模式和 Random Read 模式结合的方式；为了方便，做了一个假设，就是用户每次读、写操作都是从 E2PROM 的初始地址，即 0 地址开始操作）

实验步骤：

(1) 参考读函数，编写相应的写函数
(2) 读懂相应的 Test_Iic 函数
(3) make，得到相应的 E2PROM.o
(4) 在板子上 insmod 2PROM.o，看相应的输出代码，正确的输出是

```
---------->1
The writing data is 11
The writing data is 22
The writing data is 33
---------->2
The reading data is 11
The reading data is 22
The reading data is 33
```

2．编写应用程序，该应用程序的作用是：用户可以直接对 E2PROM 进行读写操作。通过该应用程序，可以检测 E2PROM 驱动程序编写的是否正确。相应的代码见程序解读（实验二）。

四.思考题

1. 在上面的实验 1 中，假设了用户每次读写操作都是从 E2PROM 的初始地址即 0 地址开始操作；然而在实际情况中，我们经常会遇到在 E2PROM 的某一个指定位置进行读写操作，请进一步完善该驱动程序的这项功能，并在应用程序中进行测试。

2. 为什么在进行 E2PROM 写操作前，必须进行_Wr2410Iic(0xde,0x3f,0x02)如下的操作？

实验十一：A/D 接口实验

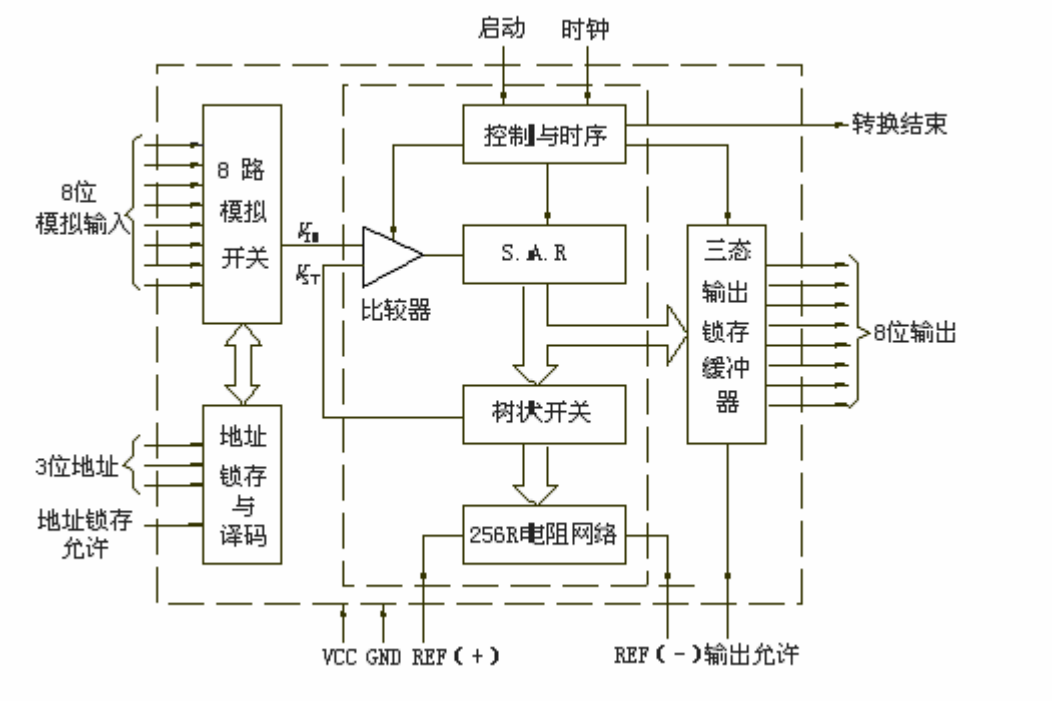
一．实验目的

通过本实验，使学生掌握 Linux 下 A/D 驱动程序的设计，了解在 S3C2410 平台上通过 CPLD 控制 A/D 的工作方式和原理。

二．实验原理和说明

1．A/D 转换芯片 ADC0809 及其接口

ADC0809 是采用 CMOS 工艺制成的 8 位 8 通道逐次渐近型 A/D 转换器。其原理框图如图所示。



图一

1.1 主要性能

8 位逐次逼近型 A/D 转换器，所有引脚的逻辑电平与 TTL 兼容；

带有锁存功能的 8 路模拟量转换开关，可对 8 路 0～5V 模拟量进行分时转换；

输出具有三态锁存/缓冲功能；

分辨率：8 位，转换时间：100us；

不可调误差：±1LSB，功耗：15mW；

工作电压：+5V，参考电压标准值+5V；

片内无时钟，一般需外加 640KHz 以下且不低于 100KHz 的时钟信号。

1.2 ADC0809 的内部结构与引脚功能

(1) 内部结构

有模拟多路转换开关和 A/D 转换两大部分。

模拟多路转换开关由 8 路模拟开关和 3 位地址锁存与译码器组成，地址锁存允许信号 ALE 将三位地址信号 ADDC、ADDB 和 ADDA 进行锁存，然后由译码电路选通其中一路摸信号加到 A/D 转换

部分进行转换。A/D 转换部分包括比较器、逐次逼近寄存器 SAR、256R 电阻网络、树状电子开关、控制与时序电路等，另外具有三态输出锁存缓冲器，其输出数据线可直接连 CPU 的数据线。

(2) 引脚功能

D7～D0：8 位数据输出线；

IN7～IN0：8 路模拟信号输入；

ADDC、ADDB、ADDA：8 路模拟信号输入通道的地址选择线；

ALE：地址锁存允许，其正跳变锁存地址选择线状态，经译码选通对应的模拟输入信号；

START：启动信号，上升沿使片内所有寄存器清零，下降沿启动 A/D 转换；

EOC：转换开始后，此引脚变为低电平，转换一结束，此引脚变为高电平；

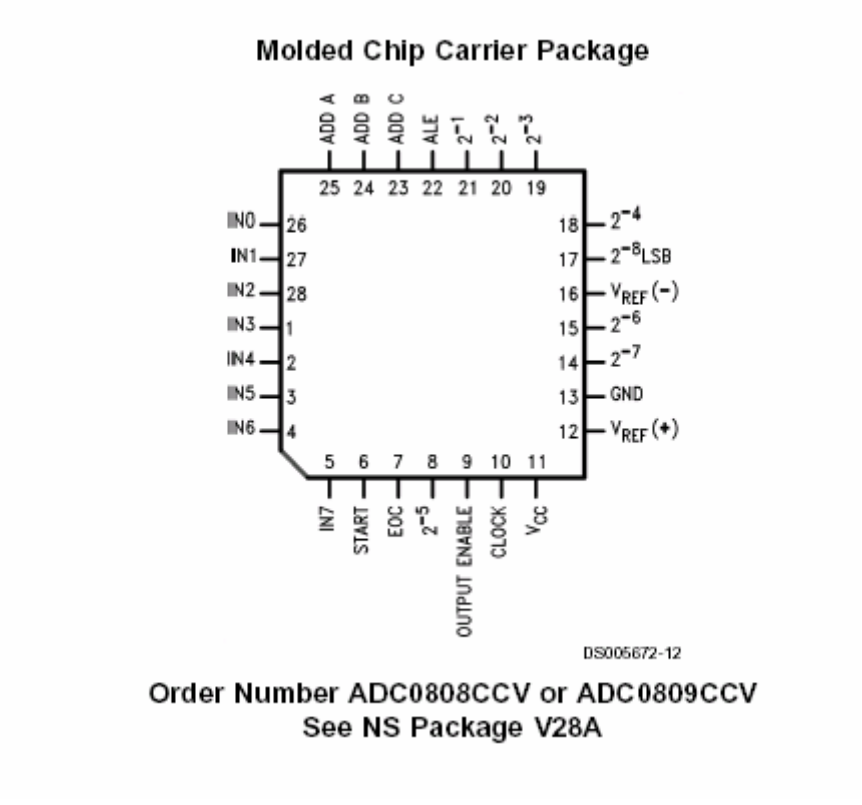
OE：输出允许，此引脚为高电平有效，当有效时，芯片内部三态数据输出锁存缓冲器被打开，转换结果送到 D7～D0；

CLOCK：时钟，最高可达 1280KHz，由外部提供；

REF（+）、REF（-）：参考电压正极、负极，通常 REF（+）接 Vcc，REF（-）接 GND；

Vcc：电源，+5V，GND：地线。

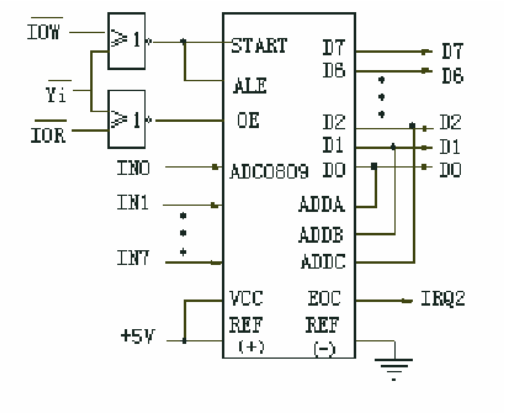
模拟输入与数字量输出的关系为 N＝（VIN-VREF（-））×256/（VREF（+）-VREF（-）），当 VREF（+）＝+5V，VREF（-）＝0V，若输入模拟电压为 2.5V，则转换后的数字量 N＝128，即 10000000B。



图二

1.3 ADC0809 的多路转换

参见图三。

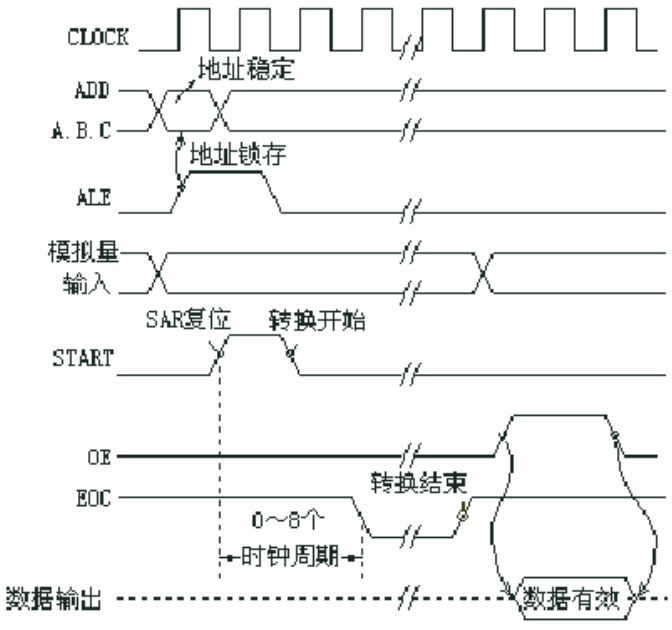


图三

例：当 ADDC、ADDB、ADDA 三个管脚接成“100”状态，ALE 有效时，ADC0809 将 IN4 管脚上的模拟输入信号进行转换。若三位地址输入信号接 CPU 的数据线 D2～D0，其状态由 CPU 提供，则可分时对 8 路不同的测量或控制电路进行 A/D 转换。

1.4 转换时序

参见图四。

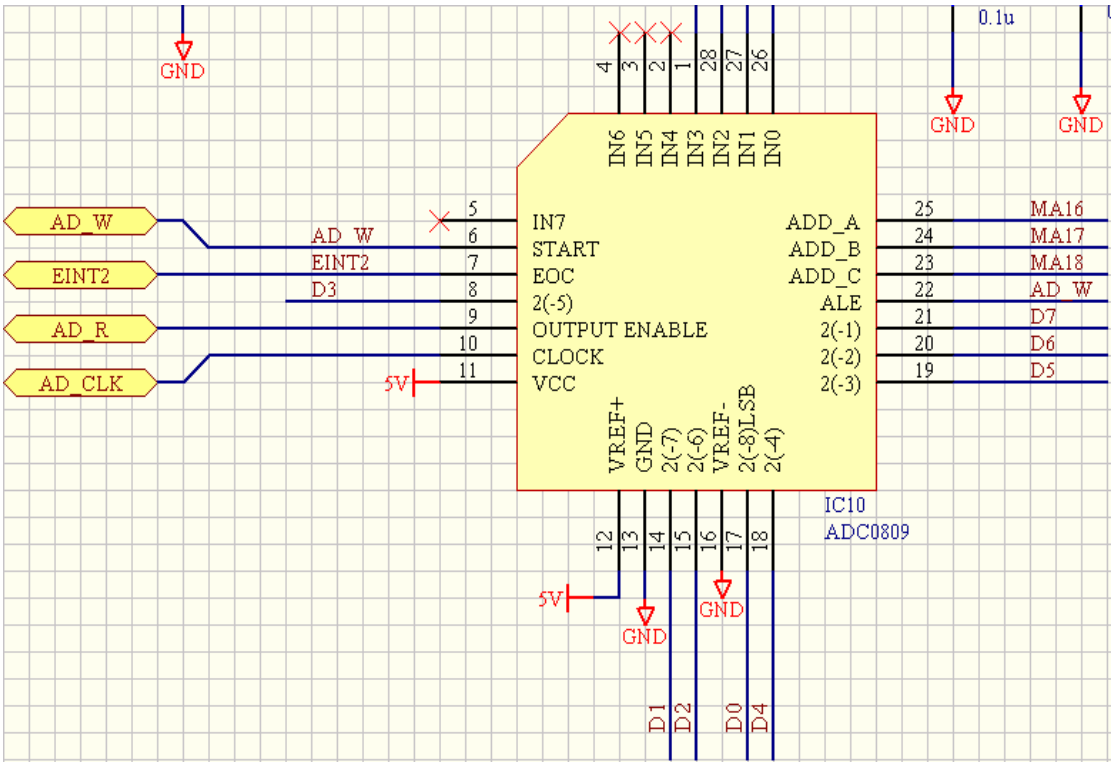


图四

2．ADC0809 在 RUIJIARM9-EDU 开发板上的连接

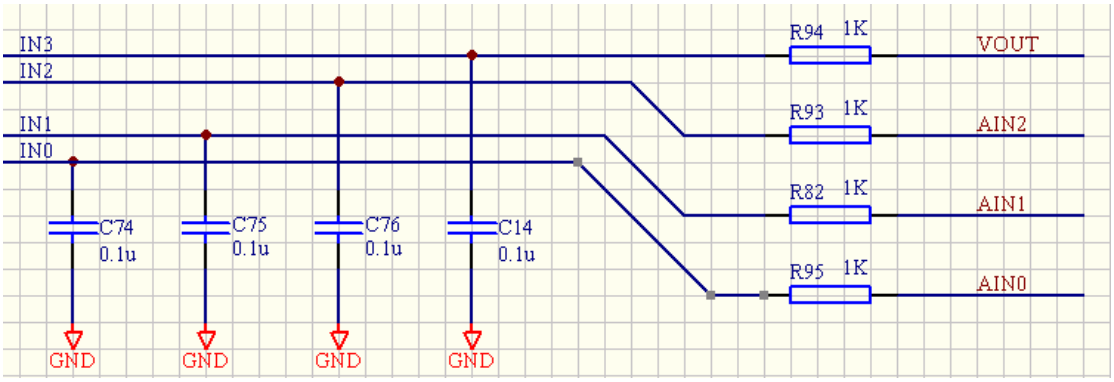
请参考 RUIJIARM9-EDU 原理图 AD，DA 转换部分。这里做一些简单的介绍：

DDA,ADDB,ADDC 分别连接到 S3C2410 的地址线 MA16,MA17,MA18 上，由于 ADC0809 可以有 8 路输入，我们可以通过 MA16,MA17,MA18 控制接受 8 路输入中某一路输入信号。



图五

(1) IN0－IN7 七路输入信号中用到了 IN0,IN1,IN2,IN3 四路信号，连接到接线电极。



图六

(2) START,OE，ALE，CLK 直接连接到 CPLD 的 IO 引脚上，由 CPLD 控制。EOC 连接到 2410 的 EINT2 引脚，做为中断信号。

(3) D0-D7 直接连接到 S3C2410 的数据线 D0-D7 上。

三．实验内容和步骤

1．CPLD 控制逻辑：

ADC0809 的 START 信号，OE（OUTPUT ENABLE）信号，CLOCK 时钟，ALE 信号都是连接在 CPLD 上，我们首先介绍一下 CPLD 的控制逻辑：

(1) 当 CPLD 判断到 NGCS2 选中，并且地址线 MA5－MA0 为 010000b 时，拉高 START 和 ALE 信号。

(2) 当 CPLD 判断到 NGCS2 选中，并且地址线 MA5－MA0 为 100000b 时，拉高 OE 信号。

2．A/D 驱动程序：

(1）下面是 A/D 驱动程序的初始化模块（驱动加载时运行）：

```
int __init adc0809_init(void)
{
   static int result;
   unsigned long gpfup;
   printk("**********************adc0809_init**************\n");

   ADC_GPACON = ioremap(0x56000000,4);
   ADC_GPADATA = ioremap(0x56000004,4);

   ADC_0 = ioremap(0x10000010,4);
   ADC_1 = ioremap(0x10010010,4);
   ADC_2 = ioremap(0x10020010,4);
   ADC_3 = ioremap(0x10030010,4);
   ADC_4 = ioremap(0x10040010,4);
   ADC_5 = ioremap(0x10050010,4);
   ADC_6 = ioremap(0x10060010,4);
   ADC_7 = ioremap(0x10070010,4);

   ADC_DATA = ioremap(0x10000020,4);

   set_external_irq(IRQ_EINT2, EXT_RISING_EDGE, GPIO_PULLUP_DIS);

   gpfup = ioremap(0x56000058,4);
   (*(volatile unsigned long *)gpfup) = 0;

   printk("adc0809_irq : EXTERNAL_IRQ = %d\n",IRQ_EINT2);

   disable_irq(IRQ_EINT2);
   enable_irq(IRQ_EINT2);
   result=request_irq(IRQ_EINT2,&adc0809_interrupt,SA_INTERRUPT,"adc0809",NULL);
   if (result)
   {
        printk("Can't get assigned irq %d,result=%d\n",IRQ_EINT2,result);
     }

   devfs_adc =
           devfs_register(NULL,"adc0809",DEVFS_FL_DEFAULT,
                       ADC0809_MAJOR, 0,
                            S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP,
                            &adc0809_fops,NULL);
```

```
    return 0;
}
```

在 adc0809_init 函数中我们主要做了如下几个工作：

(a) 映射 GCS2 地址，和 8 路 AD 输入地址；
(b) 设置 2 号外部中断；
(c) 请求中断；
(d) 申请注册设备。

(2）驱动程序实现了 file_operations 的回调函数如下：

```
static struct file_operations adc0809_fops = {
        ioctl:          adc0809_ioctl,
        open:           adc0809_open,
        read:           adc0809_read,
        write:          adc0809_write,
        release:        adc0809_release,
};
```

open 函数中选中片选 NGCS2，初始化 adc_read_addr；
ioctl 函数中根据参数确定将读取的引脚；
read 函数把中断处理函数中读到的数据发到用户程序；

(3）下面是中断处理函数：

```
void adc0809_interrupt(int irq,void *d,struct pt_regs *regs)
{
  int i;
  printk("**********************adc0809_interrupt ********************\n");
  SRCPND &= (~0x00000004);      //bit2
  INTPND = INTPND;

  printk("interrupt process!\n");
    adc_read_addr = ioremap(0x10000020,4);
    data = (*(volatile unsigned long *) adc_read_addr);
}
```

进入中断处理函数后，首先清除中断（参考 CPU 手册），紧接着把需要的数据读到一个全局变量 data 中。

3．编写 A/D 驱动测试程序（应用层）：

应用程序比较简单，有以下几个步骤：

(1) 打开设备（open）。
(2) 在应用程序中获得用户需要测量的引脚。
(3) 通过 ioctl 选择待测量的引脚（由用户输入），并且触发中断。
(4) 通过 read 调用，读取特定引脚的电平值。

需要注意的是，从驱动中获得的只是电压的相对值，需要根据参考电压进行转换。

四．思考题

1．试着结合 A/D 实验，测量步进电机控制电路输出电压。
2．请思考中断处理程序中为什么要清除中断，不这样做会怎么样？
3．有兴趣的同学可以用轮询方式实现本驱动。

实验十二：D/A 接口实验

一． 实验目的

通过本实验，使学生掌握 Linux 下 D/A 驱动程序的设计，了解 D/A 的工作方式和原理,了解 S3C2410 平台如何通过 CPLD 控制 D/A 工作。

二． 实验原理和说明

1．D/A 转换器芯片

D/A 转换器：数/模转换器，它把数字量转换成电模拟量。即把二进制数字量转换为与其数值成正比的电模拟量。 D/A 转换器的性能指标如下：

(1) 分辨率：是指 D/A 能转换的二进制位数，位数越多，分辨率越高；例：转换 8 位，若电压满量程为 5V，则能分辨的最小电压为：5V/256≈20mV；转换 10 位，若电压满量程为 5V，则能分辨的最小电压为：5V/1024≈5Mv。

(2) 转换时间：指数字量输入到转换输出稳定为止所需的时间；

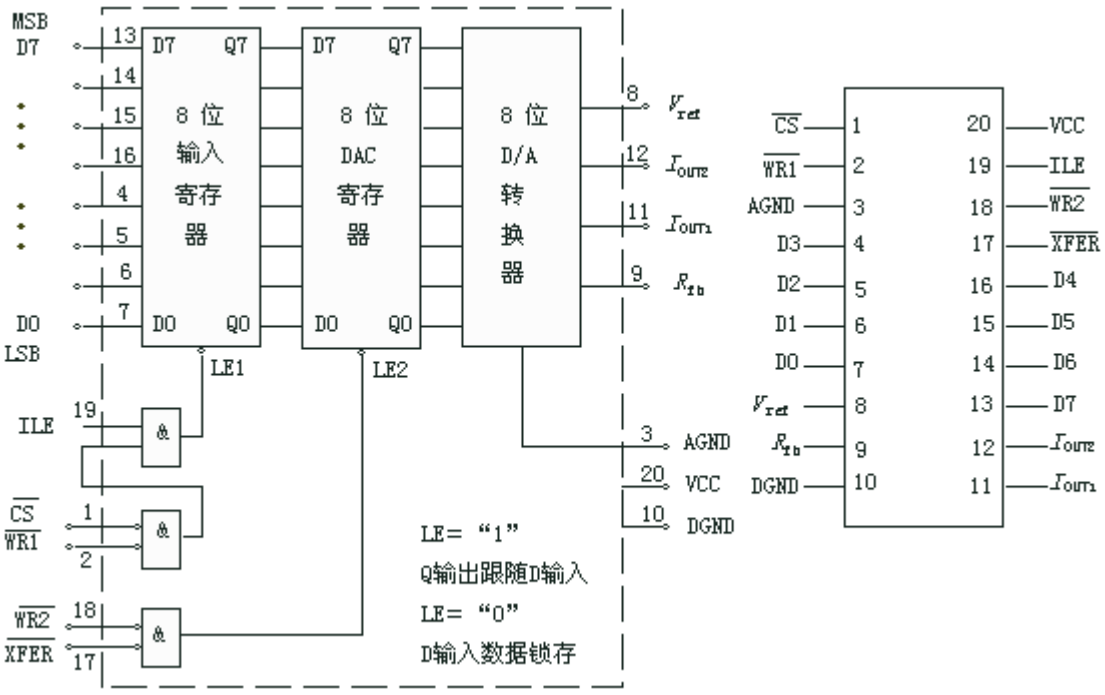
(3) 精度：指 D/A 实际输出与理论值之间的误差，一般采用数字量的最低有效位作为衡量单位；例：±1/2LSB，若是 8 位转换，则精度是±（1/2）×（1/256）满度=±1/512 满度。

(4) 线性度：当数字量变化时，D/A 输出的电模拟量按比例关系变化的程度。模拟量输出偏离理想输出的最大值称为线性误差。

2．DAC0832 及其接口

本试验使用 DAC0832 来实现 D/A 转换，其内部框图如图所示，由 8 位输入寄存器，8 位 DAC 寄存器，8 位 D/A 转换器及逻辑控制单元等功能电路构成。

DAC0832 芯片采用 CMOS 工艺，电流输出型 D/A，8 位，转换时间约 1us。



（a） 结构框图　　（b）引脚排列

图一

2.1 主要性能

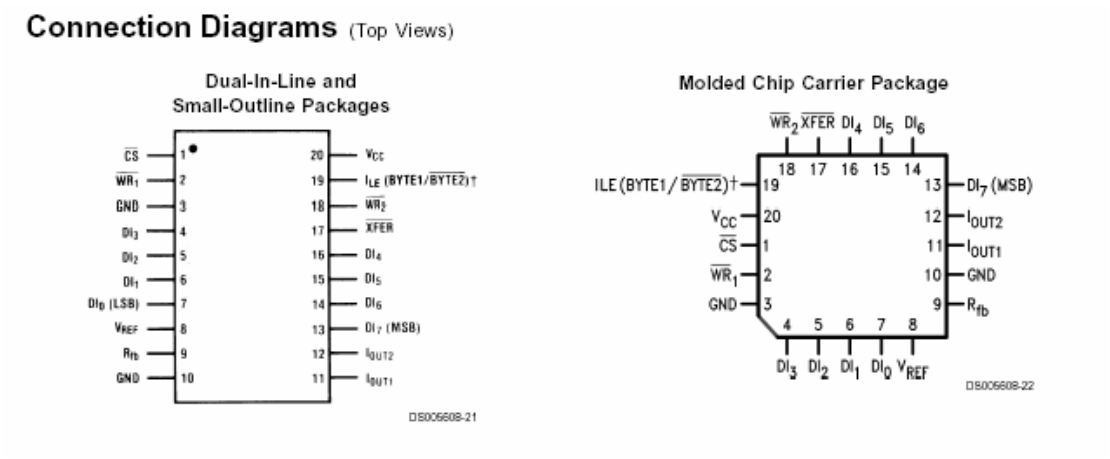
(1) 输入的数字量为 8 位；
(2) 采用 CMOS 工艺，所有引脚的逻辑电平与 TTL 兼容；
(3) 数字了输入可以采用双缓冲，单缓冲或直通方式；
(4) 转换时间：1us；
(5) 精度：±1LSB；
(6) 分辨率：8 位；
(7) 单一电源，5V～15V，功耗 20mW；
(8) 参考电压：+10V～-10V。

2.2 内部结构及引脚功能

(1) 内部结构

DAC0832 有两种封装形式，如下图，在我们的系统里使用的第二种封装。

具体的内部结构叙述如下：



图二

寄存器部分：

(a) 8 位输入寄存器：可作为输入数据第一级缓冲；

(b) 8 位 DAC 寄存器：可作为输入数据第二级缓冲；

(c) 8 位 D/A 转换器：将 DAC 寄存器中的数据转换成具有一定比例的直流电流。

逻辑控制部分：

DAC0832 芯片内部有两个数据缓冲器，分别由两组控制信号控制，

(a) 当 ILE=1∩ $\overline{CS}$=0∩ $\overline{WR1}$=0 时，D7～D0 上的数据锁存到输入寄存器中。

(b) 当 $\overline{XFER}$=0∩ $\overline{WR2}$=0 时，输入寄存器中的数据被锁存到 DAC 寄存器中。

(2) 引脚功能

D7～D0：8 位数据量输入；

ILE：数据输入锁存允许，高电平有效；

$\overline{CS}$：片选；

$\overline{WR1}$：输入寄存器写信号，当 ILE、 $\overline{CS}$、 $\overline{WR1}$同时有效，数据装入输入寄存器，实现输入数据的第一级缓冲；

$\overline{XFER}$：数据传送控制信号，控制从输入寄存器到 DAC 寄存器的内部数据传送；

$\overline{WR2}$：DAC 寄存器写信号，当 $\overline{XFER}$和 $\overline{WR2}$均有效时，将输入寄存器中的数据装入 DAC 寄存器并开始 D/A 转换，实现输入数据的第二级缓冲；

VREF：参考电压源，电压范围为-10V～+10V。

Rf0：内部反馈电阻接线端；

IOUT1：DAC 电流输出 1。其值随输入数字量线性变化；

IOUT2：DAC 电流输出 2。

当 DAC 寄存器内容全为 1 时，IOUT1 最大，IOUT2=0；

当 DAC 寄存器内容全为 0 时，IOUT1=0，IOUT2=最大；

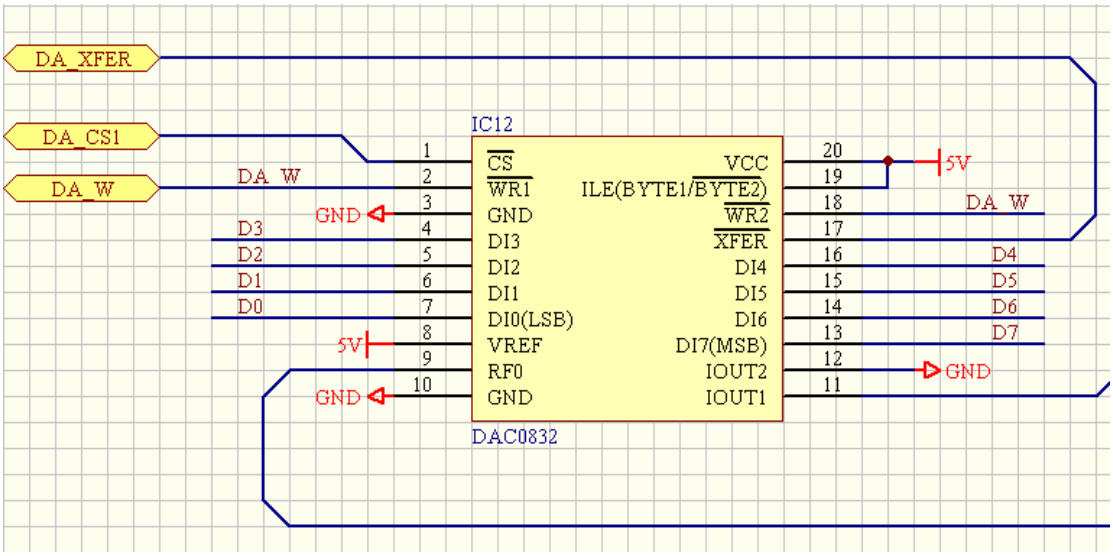
当 DAC 寄存器内容为 N 时，IOUT1=VREF×N/（256×Rf0），IOUT2= VREF/Rf0- IOUT1，

无论 N 值多大：IOUT1+IOUT2= VREF/Rf0（1-28）=常数≌VREF/Rf0。

Vcc：工作电源，其值范围为+5V～15V，典型值为+15V；
AGND：模拟信号地线；
DGND：数字信号地线。

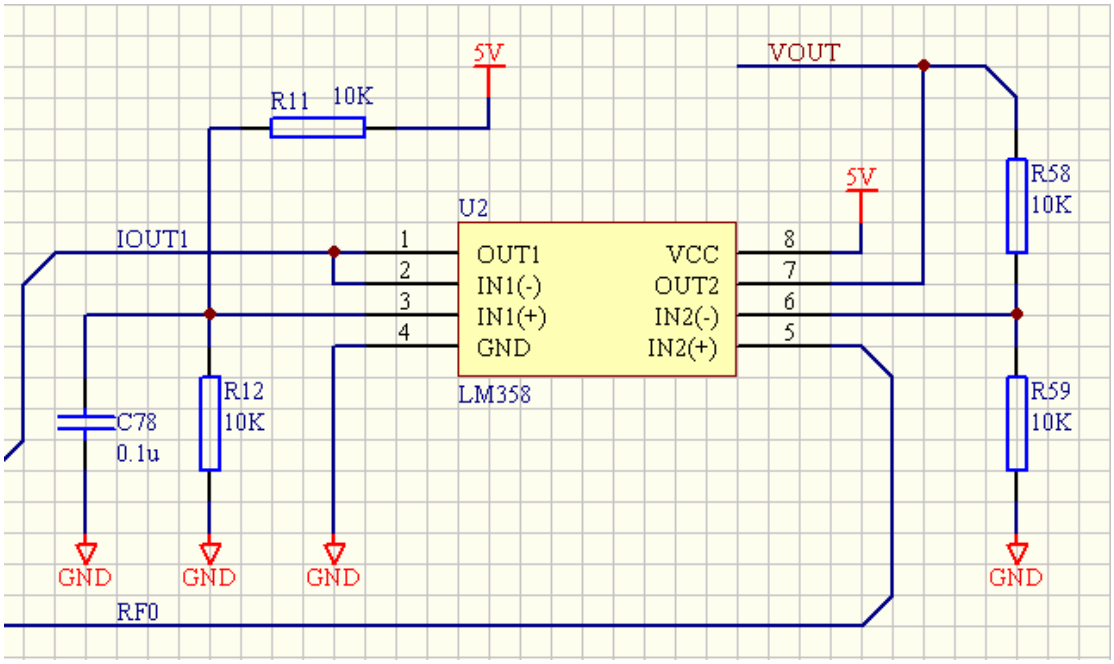
(3) 工作方式
DAC0832 有双缓冲、单缓冲和直通三种方式。
(a) 双缓冲工作方式：进行两级缓冲；
(b) 单缓冲工作方式：只进行一级缓冲；
(c) 直通工作方式：不进行缓冲。适用于比较简单的场合。

(4) 接口电路
DAC0832 为电流输出型 D/A 转换芯片，使用时，Rf0、IOUT1、IOUT2 3 个引脚外接运算放大器或其他转换电路，以便将转换后的电流变换成电压输出。若外接一个运算放大器为单极性输出，如图三 VOUT 输出。



图三

DAC0832 输出 的是电流，要转换成电压，还必须外接运算放大器。如下图所示。



图四

图中，VOUT1＝-IOUT1×Rf0＝-VREF×N/(256×Rf0)×Rf0 ＝-N/256×VREF,

DAC0832 对执行时序也有一定要求:第一，$\overline{WR}$选通脉冲应有一定宽度，通常要求≥500ns，当取VCC＝+15V 典型值时，$\overline{WR}$宽度只要≥100ns 就可以了。此时器件处于最佳工作状态。第二，数据输入保持时间应不小于 90ns。在满足这两个条件下，转换电流建立时间为 1.0μs。当 VCC 偏移典型值时，也要注意满足转换时序要求，否则将不能保证转换数据正确。

在图三中，若 $\overline{XFER}$不接地址译码器，改为接 $\overline{WR1}$、 $\overline{WR2}$或直接接地，则 DAC0832 为单缓冲方式。

三． 实验内容和步骤

1．CPLD 控制逻辑：

由于本系统上 DAC0832 芯片控制信号（$\overline{CS}$、$\overline{WR1}$、$\overline{WR2}$、$\overline{XFER}$）是连接在 CPLD 上的，所以驱动程序必须和 CPLD 逻辑程序相匹配，我们首先介绍一下 CPLD 程序的控制逻辑：

(1) CPLD 判断到 NGCS2 选中，并且地址线 MA5－MA0 为 001000b 时，拉低 $\overline{WR}$信号，D7-D0 的数据将锁存在 8 位输入寄存器中。

(2) 当 CPLD 判断到 NGCS2 选中，并且地址线 MA5－MA0 为 110000b 时，拉低$\overline{XFER}$信号，开始 D/A 转换。

2．编写 D/A 驱动程序：

驱动程序的工作就是要和 CPLD 逻辑程序相配合，下面我们介绍一下：

(1) 下面是 A/D 驱动程序的初始化模块（驱动加载时运行）：

```
int __init dac0832_init(void)
{
   printk("*****************dac0832_init**************\n");
```

```
    DAC_GPACON = ioremap(0x56000000,4);
    DAC_GPADATA = ioremap(0x56000004,4);

    DAC_DATA = ioremap(0x10000008,4);
    DAC_START = ioremap(0x10000030,4);

    devfs_dac =
        devfs_register(NULL,"dac0832",DEVFS_FL_DEFAULT,
                DAC0832_MAJOR, 0,
                S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP,
                &dac0832_fops,NULL);
    return 0;
}
```

在 dac0809_init 函数中我们主要做了如下几个工作：

(a) 映射 GCS2 地址，CPLD 逻辑控制第一步和第二步的入口地址；

(b) 申请注册设备；

(2) 驱动程序实现了 file_operations 的回调函数如下：

```
static struct file_operations dac0832_fops = {
        open:           dac0832_open,
        write:          dac0832_write,
        release:        dac0832_release,
};
```

(3) dac0832_open 代码：

```
int dac0832_open(struct inode *inode, struct file *filp)
{
    dac0832_sle |= 0x2000;
    dac0832_sle_data &= (~0x2000);
    printk("open ok\n");
    return 0;
}
```

dac0832_open 唯一做的工作就是把 NGCS2 的片选选中。

(4) dac0832_write 代码：

```
ssize_t dac0832_write(struct file *filp, const char *buf,
                      size_t size, loff_t *offp)
{
    char key;
    if (get_user(key, buf))
      return -EFAULT;

    (*(volatile unsigned char *) DAC_DATA) = key;
```

```
    udelay(100000);
    (*(volatile unsigned char *) DAC_START) = key;

    return 1;
}
```

dac0832_write 首先通过 get_user 函数得到用户传送过来的数据，接着(*(volatile unsigned char *) DAC_DATA) = key;语句写地址 M5-M0 位 001000b，也就是让 CPLD 拉低 $\overline{WR}$信号。最后通过语句(*(volatile unsigned char *) DAC_START) = key;写地址 M5-M0 为 110000b，让拉低 $\overline{XFER}$信号，开始 D/A 转换。

3. 写 D/A 驱动测试程序（应用层）。

测试程序很简单，只需要打开设备，调用驱动程序中 write 系统调用就可以了，在这里不列出代码，大家可以参考实验十二的 A/D 驱动测试程序。

四．思考题

1. 实验十二 A/D 驱动程序采用的是中断方式，本试验采用的是轮询方式，请思考驱动的中断方式和轮询方式的区别，以及在什么情况下使用中断，什么情况下使用轮询。
2. 试验采用的是单缓冲方式，有兴趣的同学可以考虑如何实现直通方式和双缓冲方式。

实验十三：CPLD 控制步进电机实验

一．实验目的

通过本实验，使学生掌握 Linux 下通过 CPLD 控制设备驱动程序的设计，了解步进电机的工作方式和原理,了解 S3C2410 平台如何通过 CPLD 控制步进电机工作。

二．实验原理和说明

1．步进电机简单工作原理

步进电机是将电脉冲信号转变为角位移或线位移的开环控制元件。在非超载的情况下，电机的转速、停止的位置只取决于脉冲信号的频率和脉冲数，而不受负载变化的影响，即给电机加一个脉冲信号，电机则转过一个步距角。这一线性关系的存在，加上步进电机只有周期性的误差而无累积误差等特点。使得在速度、位置等控制领域用步进电机来控制变的非常的简单。

虽然步进电机已被广泛地应用，但步进电机并不能象普通的直流电机，交流电机在常规下使用。它必须由双环形脉冲信号、功率驱动电路等组成控制系统方可使用。因此用好步进电机却非易事，它涉及到机械、电机、电子及计算机等许多专业知识。使用嵌入式系统能够很好的实现对步进电机的控制。

常用的步进电机分为反应式步进电机和感应子式步进电机，本实验中采用的是感应子式步进电机。

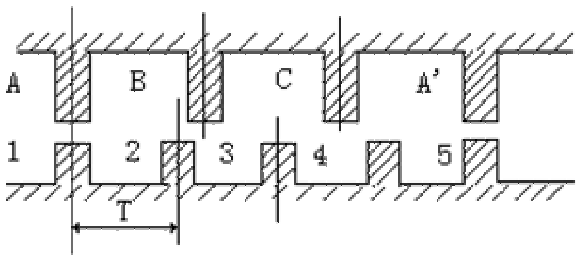
1.1 反应式步进电机原理

由于反应式步进电机工作原理比较简单。下面先叙述三相反应式步进电机原理。

(1) 结构：

电机转子均匀分布着很多小齿，定子齿有三个励磁绕阻，其几何轴线依次分别与转子齿轴线错开。

0、1/3 τ、2/3 τ,（相邻两轴子齿轴线间的距离为齿距以τ表示），即 A 与齿 1 相对齐，B 与齿 2 向右错开 1/3 τ，C 与齿 3 向右错开 2/3 τ，A'与齿 5 相对齐，（A'就是 A，齿 5 就是齿 1）下面是定转子的展开图：



图一

(2) 旋转：

如 A 相通电，B，C 相不通电时，由于磁场作用，齿 1 与 A 对齐，（转子不受任何力以下均同）。

如 B 相通电，A，C 相不通电时，齿 2 应与 B 对齐，此时转子向右移过 1/3 τ，此时齿 3 与 C 偏移为 1/3 τ，齿 4 与 A 偏移（τ-1/3 τ）=2/3 τ。

如 C 相通电，A，B 相不通电，齿 3 应与 C 对齐，此时转子又向右移过 1/3 τ，此时齿 4 与 A 偏移为 1/3 τ对齐。

如 A 相通电，B，C 相不通电，齿 4 与 A 对齐，转子又向右移过 1/3 て

这样经过 A、B、C、A 分别通电状态，齿 4（即齿 1 前一齿）移到 A 相，电机转子向右转过一个齿距，如果不断地按 A，B，C，A……通电，电机就每步（每脉冲）1/3 て,向右旋转。如按 A，C，B，A……通电，电机就反转。

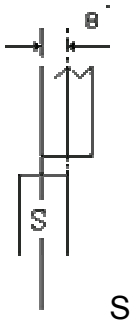
由此可见：电机的位置和速度由导电次数（脉冲数）和频率成一一对应关系。而方向由导电顺序决定。

不过，出于对力矩、平稳、噪音及减少角度等方面考虑。往往采用 A-AB-B-BC－C-C-A 这种导电状态，这样将原来每步 1/3 て改变为 1/6 て。甚至于通过二相电流不同的组合，使其 1/3 て变为 1/12 て，1/24 て，这就是电机细分驱动的基本理论依据。

不难推出：电机定子上有 m 相励磁绕阻，其轴线分别与转子齿轴线偏移 1/m,2/m……(m-1)/m,1。并且导电按一定的相序电机就能正反转被控制——这是步进电机旋转的物理条件。只要符合这一条件我们理论上可以制造任何相的步进电机，出于成本等多方面考虑，市场上一般以二、三、四、五相为多。

(3) 力矩：

电机一旦通电，在定转子间将产生磁场（磁通量Φ）当转子与定子错开一定角度产生力 F 与（dΦ/dθ）成正比



图二

其磁通量Φ=Br*S

Br 为磁密，S 为导磁面积

F 与 L*D*Br 成正比

L 为铁芯有效长度，D 为转子直径

Br=N・I/R

N・I 为励磁绕阻安匝数（电流乘匝数）R 为磁阻。

力矩=力*半径

力矩与电机有效体积*安匝数*磁密 成正比（只考虑线性状态）

因此，电机有效体积越大，励磁安匝数越大，定转子间气隙越小，电机力矩越大，反之亦然。

1.2 感应子式步进电机

(1) 特点：

感应子式步进电机与传统的反应式步进电机相比，结构上转子加有永磁体，以提供软磁材料的工作点，而定子激磁只需提供变化的磁场而不必提供磁材料工作点的耗能，因此该电机效率高，电流小，发热低。因永磁体的存在，该电机具有较强的反电势，其自身阻尼作用比较好，使其在运转过程中比较平稳、噪音低、低频振动小。

感应子式步进电机某种程度上可以看作是低速同步电机。一个四相电机可以作四相运行，也可以

作二相运行。（必须采用双极电压驱动），而反应式电机则不能如此。例如：四相，八相运行（A-AB-B-BC-C-CD-D-DA-A）完全可以采用二相八拍运行方式.不难发现其条件为 C= $\overline{A}$,D= $\overline{B}$ 。

一个二相电机的内部绕组与四相电机完全一致，小功率电机一般直接接为二相，而功率大一点的电机，为了方便使用，灵活改变电机的动态特点，往往将其外部接线为八根引线（四相），这样使用时，既可以作四相电机使用，可以作二相电机绕组串联或并联使用。

(2) 分类

感应子式步进电机以相数可分为：二相电机、三相电机、四相电机、五相电机等。以机座号（电机外径）可分为：42BYG(BYG 为感应子式步进电机代号)、57BYG、86BYG、110BYG、（国际标准），而像 70BYG、90BYG、130BYG 等均为国内标准。

(3) 步进电机的静态指标术语

相数：产生不同对极 N、S 磁场的激磁线圈对数。常用 m 表示。

拍数：完成一个磁场周期性变化所需脉冲数或导电状态用 n 表示，或指电机转过一个齿距角所需脉冲数，以四相电机为例，有四相四拍运行方式即 AB-BC-CD-DA-AB，四相八拍运行方式即 A-AB-B-BC-C-CD-D-DA-A.

步距角：对应一个脉冲信号，电机转子转过的角位移用 θ 表示。θ =360 度（转子齿数 J*运行拍数），以常规二、四相，转子齿为 50 齿电机为例。四拍运行时步距角为 θ =360 度/（50*4）=1.8 度（俗称整步），八拍运行时步距角为 θ =360 度/（50*8）=0.9 度（俗称半步）。

定位转矩：电机在不通电状态下，电机转子自身的锁定力矩（由磁场齿形的谐波以及机械误差造成的）

静转矩：电机在额定静态电作用下，电机不作旋转运动时，电机转轴的锁定力矩。此力矩是衡量电机体积（几何尺寸）的标准，与驱动电压及驱动电源等无关。

虽然静转矩与电磁激磁安匝数成正比，与定齿转子间的气隙有关，但过份采用减小气隙，增加激磁安匝来提高静力矩是不可取的，这样会造成电机的发热及机械噪音。

(3) 步进电机动态指标及术语：

(a) 步距角精度：

步进电机每转过一个步距角的实际值与理论值的误差。用百分比表示：误差/步距角*100%。不同运行拍数其值不同，四拍运行时应在 5%之内，八拍运行时应在 15%以内。

(b) 失步：

电机运转时运转的步数，不等于理论上的步数。称之为失步。

(c) 失调角：

转子齿轴线偏移定子齿轴线的角度，电机运转必存在失调角，由失调角产生的误差，采用细分驱动是不能解决的。

(d) 最大空载起动频率：

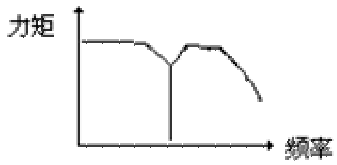
电机在某种驱动形式、电压及额定电流下，在不加负载的情况下，能够直接起动的最大频率。

(e) 最大空载的运行频率：

电机在某种驱动形式，电压及额定电流下，电机不带负载的最高转速频率。

(f) 运行矩频特性：

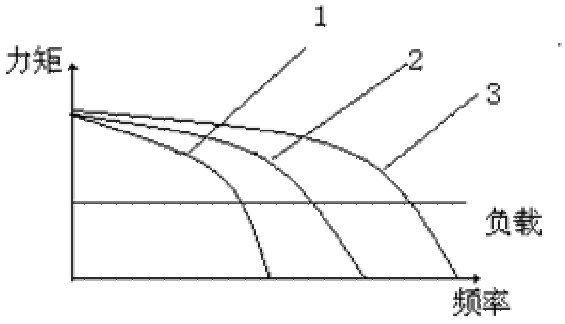
电机在某种测试条件下测得运行中输出力矩与频率关系的曲线称为运行矩频特性，这是电机诸多动态曲线中最重要的，也是电机选择的根本依据。如下图所示：



图三

其它特性还有惯频特性、起动频率特性等。

电机一旦选定，电机的静力矩确定，而动态力矩却不然，电机的动态力矩取决于电机运行时的平均电流（而非静态电流），平均电流越大，电机输出力矩越大，即电机的频率特性越硬。如下图所示：



图四

其中，曲线 3 电流最大、或电压最高;曲线 1 电流最小、或电压最低，曲线与负载的交流为负载的最大速度点。

要使平均电流大，尽可能提高驱动电压，使采用小电感大电流的电机。

(g) 电机的共振点：

步进电机均有固定的共振区域，二、四相感应子式步进电机的共振区一般在 180-250pps 之间（步距角 1.8 度）或在 400pps 左右（步距角为 0.9 度），电机驱动电压越高，电机电流越大，负载越轻，电机体积越小，则共振区向上偏移，反之亦然，为使电机输出电矩大，不失步和整个系统的噪音降低，一般工作点均应偏移共振区较多。

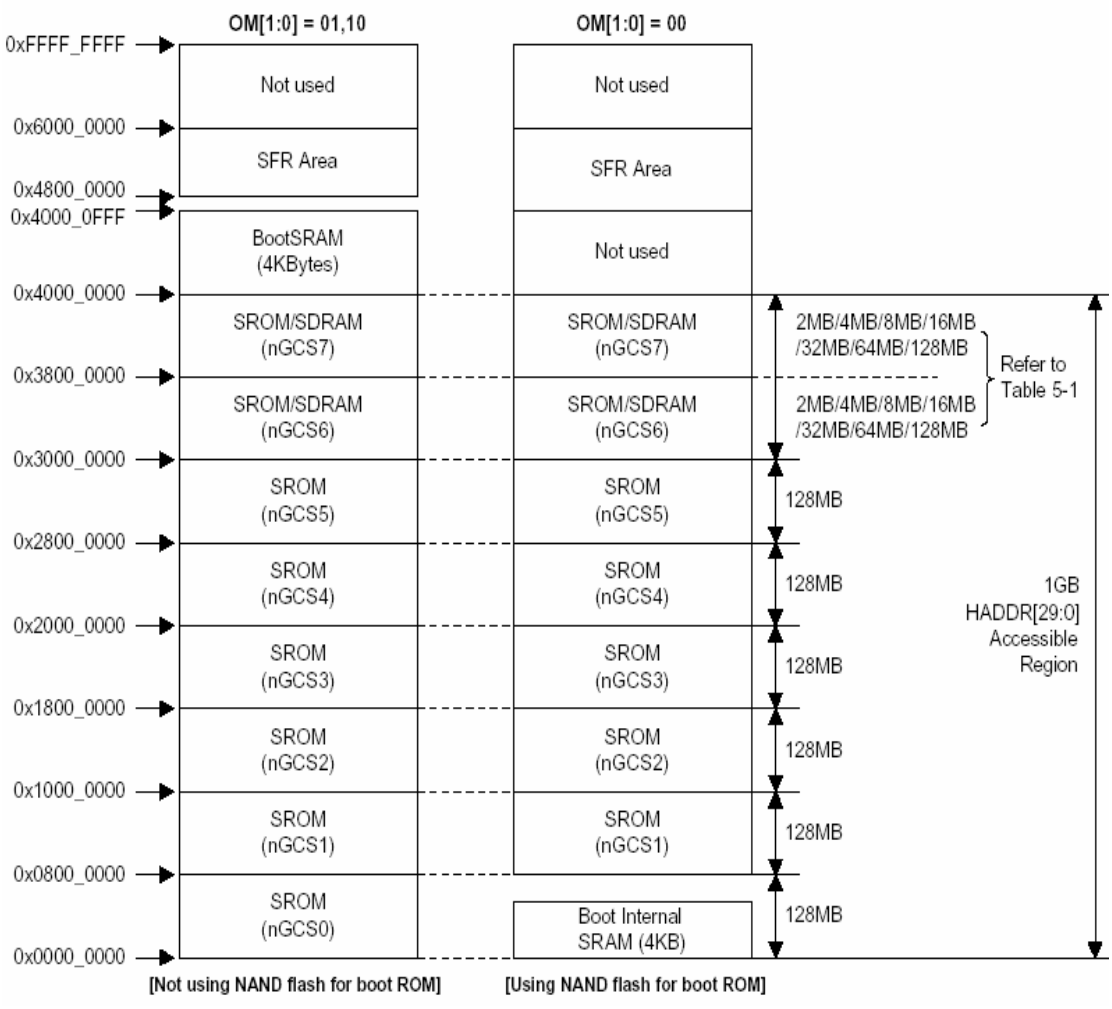
(h) 电机正反转控制：

当电机绕组通电时序为 AB-BC-CD-DA 或($A\bar{B}$ - $\bar{A}\bar{B}$ - $\bar{A}B$ - AB)时为正转，通电时序为 DA-CA-BC-AB 或(AB - $\bar{A}B$ - $\bar{A}\bar{B}$ - $A\bar{B}$)时为反转。

2．SAMSUNG 的 S3C2410 处理器相关寄存简介

与本实验相关的寄存器有nGCS2，GPA。

(1) nGCS2：片选。



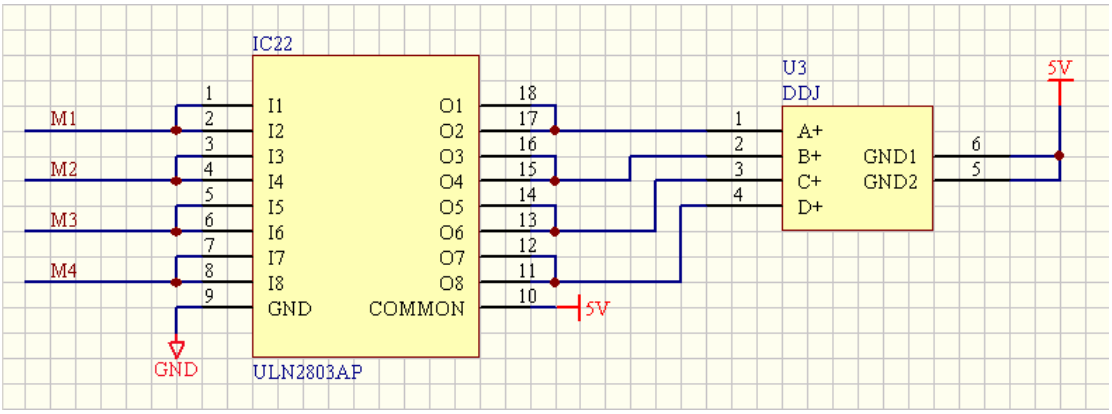
(2) GPA端口控制器

Register	Address	R/W	Description	Reset Value
GPACON	0x56000000	R/W	Configure the pins of port A	0x7FFFFF
GPADAT	0x56000004	R/W	The data register for port A	Undefined
Reserved	0x56000008	–	Reserved	Undefined
Reserved	0x5600000C	–	Reserved	Undefined

GPACON	Bit	Description
GPA22	[22]	0 = Output 1 = nFCE
GPA21	[21]	0 = Output 1 = nRSTOUT (nRSTOUT = nRESET & nWDTRST & SW_RESET(MISCCR[16]))
GPA20	[20]	0 = Output 1 = nFRE
GPA19	[19]	0 = Output 1 = nFWE
GPA18	[18]	0 = Output 1 = ALE
GPA17	[17]	0 = Output 1 = CLE
GPA16	[16]	0 = Output 1 = nGCS5
GPA15	[15]	0 = Output 1 = nGCS4
GPA14	[14]	0 = Output 1 = nGCS3
GPA13	[13]	0 = Output 1 = nGCS2
GPA12	[12]	0 = Output 1 = nGCS1
GPA11	[11]	0 = Output 1 = ADDR26
GPA10	[10]	0 = Output 1 = ADDR25
GPA9	[9]	0 = Output 1 = ADDR24
GPA8	[8]	0 = Output 1 = ADDR23
GPA7	[7]	0 = Output 1 = ADDR22
GPA6	[6]	0 = Output 1 = ADDR21
GPA5	[5]	0 = Output 1 = ADDR20
GPA4	[4]	0 = Output 1 = ADDR19
GPA3	[3]	0 = Output 1 = ADDR18
GPA2	[2]	0 = Output 1 = ADDR17
GPA1	[1]	0 = Output 1 = ADDR16
GPA0	[0]	0 = Output 1 = ADDR0

GPADAT	Bit	Description
GPA[22:0]	[22:0]	When the port is configured as output port, the pin state is the same as the that of the corresponding bit. When the port is configured as functional pin, undefined value will be read.

3. 步进电机连接



上图是步进电机的连接图，ULN2803AP 芯片的作用是放大输入电流，M1,M2,M3,M4 的输入电流大约是 10mA 左右，从 O1,O2,O3,O4,O5,O6,O7,O8 出来的电流放大到 100mA 左右。M1,M2,M3,M4 直接连接在 CPLD 的 IO 引脚上。我们使用的是四项电机，在图中标识位 DDJ。

三．试验内容和步骤

1． CPLD 控制逻辑：

在我们的嵌入式平台上是通过 CPLD 来控制步进电机的，所以步进电机的驱动程序需要和 CPLD 的接口程序相配合。

ULN2803AP 的 M1,M2,M3,M4 是直接连接在 CPLD 上的，由 CPLD 控制，控制逻辑如下：

CPLD 判断到 NGCS2 选中，并且地址线 MA5－MA0 为 000110b 时，读取数据线的低四位 D3-D0，并作为 M4-M1 的输出。

2． 步进电机驱动程序：

驱动程序相对来说比较简单，主要工作是和 CPLD 逻辑相配合，通过向 D3-D0 写不同的数据来控制步进电机工作，简单的代码介绍如下：

(1) 下面是 A/D 驱动程序的初始化模块（驱动加载时运行）：

```
int __init electromotor_init(void)
{
printk("********************electromotor_init**************\n");
ELECTROMOTOR_GPACON = ioremap(0x56000000,4);
ELECTROMOTOR_GPADATA = ioremap(0x56000004,4);

ELECTROMOTOR = ioremap(0x10000003,4);
//printk("electromotor_init: electromotor_addr_1: %x\n",ELECTROMOTOR_1);

devfs_electromotor =
   devfs_register(NULL, "electromotor", DEVFS_FL_DEFAULT,
            ELECTROMOTOR_MAJOR, 0,
            S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP,
            &electromotor_fops, NULL);
return 0;
}
```

在 electromotor_init 函数中我们主要做了如下几个工作：

(a) 映射 GCS2 地址和 CPLD 逻辑控制的入口地址；

(b) 申请注册设备；

(2) 驱动程序实现了 file_operations 的回调函数如下：

```
static struct file_operations electromotor_fops = {
open:           electromotor_open,
write:                  electromotor_write,
release:  electromotor_release,
};
```

(3) electromotor_open 代码：

```
int electromotor_open(struct inode *inode, struct file *filp)
{
electromotor_sle |= 0x2000;
```

```
electromotor_sle_data &= (~0x2000);
printk("open ok\n");
return 0;
}
```

electromotor_open 唯一做的工作就是把 NGCS2 的片选选中。

(4) electromotor_write 代码：

```
ssize_t electromotor_write(struct file *fp, char * buf, size_t size)
{
    char key;
    if (get_user(key, buf))
      return -EFAULT;
    (*(volatile unsigned char *) ELECTROMOTOR) = key;
    return 1;
}
```

electromotor_write 首先通过 get_user 函数得到用户传送过来的数据，接着(*(volatile unsigned char *) ELECTROMOTOR) = key;语句写地址 M5-M0 位 000110b，也就是让 CPLD 进入控制逻辑。这里要注意的是从应用程序中传来的 Key 值的低四位是传送给 M4-M1。

2. D/A 步进电机控制程序（应用层）。

下面是步进电机驱动测试程序的代码：

```
int main()
{
    int electromotor_fd,count;
    char ret;
    electromotor_fd = open("/dev/electromotor",O_RDWR);
    if (electromotor_fd <= 0){
        printf("open electromotor device error\n");
        return 0;
    }
    while(1)
    {
        ret = 0x7;
        count = write(electromotor_fd,&ret,1);
        usleep(10000);
        ret = 0x3;
        count = write(electromotor_fd,&ret,1);
        usleep(10000);
        ret = 0xb;
        count = write(electromotor_fd,&ret,1);
        usleep(10000);
        ret = 0x9;
        count = write(electromotor_fd,&ret,1);
        usleep(10000);
```

```
            ret = 0xd;
            count = write(electromotor_fd,&ret,1);
            usleep(10000);
            ret = 0xc;
            count = write(electromotor_fd,&ret,1);
            usleep(10000);
            ret = 0xe;
            count = write(electromotor_fd,&ret,1);
            usleep(10000);
        }
        close(electromotor_fd);
        return 0;
    }
```

代码程序本身比较简单，我们需要注意的是传送给 write 系统调用的 ret 值的低四位，大家可以结合步进电机的原理，分析传送这些值的道理。

四．思考题

1．修改驱动程序，使步进电机以不同速度转动（通过添加 ioctl 接口，可以在应用程序中控制转速）。

实验十四：小键盘+LED 驱动实验

一．实验目的

学习小键盘驱动原理和 LED 显示原理，掌握轮询方式获取键值的原理，理解驱动程序采用轮询方式和中断方式的区别。学习将最近的按键值移位显示在 LED 上的方法。

二．实验原理和说明

1．键盘有关概念

1.1 键盘扫描码

当键盘上的按键被按下或松开的时候，键盘将发送扫描码给计算机。扫描码告诉系统什么键被按下或松开。例如按键'A'。按键'A'的扫描码为 1CH。当按下键'A'时，键盘将发送 1CH 到串行线上。如果一直按住键'A'不放，键盘将以一定的重复率发送 1CH。这个过程将保持下去，直到另外的键被按下或者键'A'被释放。

而且，当按键被释放的时候，键盘也将发送其它代码给计算机。还是以键'A'为例。当键'A'被释放时，键盘将发送 F0H 给计算机，报告有键被释放，随后再发送 1CH，说明被释放的键是'A'，即，键'A'释放时键盘发送给计算机的扫描码是 F0H 1CH。

1.2 系统扫描码

在我们通常所使用的 pc 机上，系统（BIOS，linux 内核）处理的并不是键盘扫描码，而是系统扫描码。而键盘扫描码到系统扫描码的转换通常都是由硬件（如 PC 键盘接口模块 8042）完成。系统扫描码与键盘扫描码除了每个键的代码不同外，最大的差别是把键盘放下时产生的两个键盘扫描码（共两字节）转换为 1 个字节的系统扫描码表示。如‘A’被键释放时，发送键盘扫描码为：F0H，1CH，相应的系统扫描码为 9E。

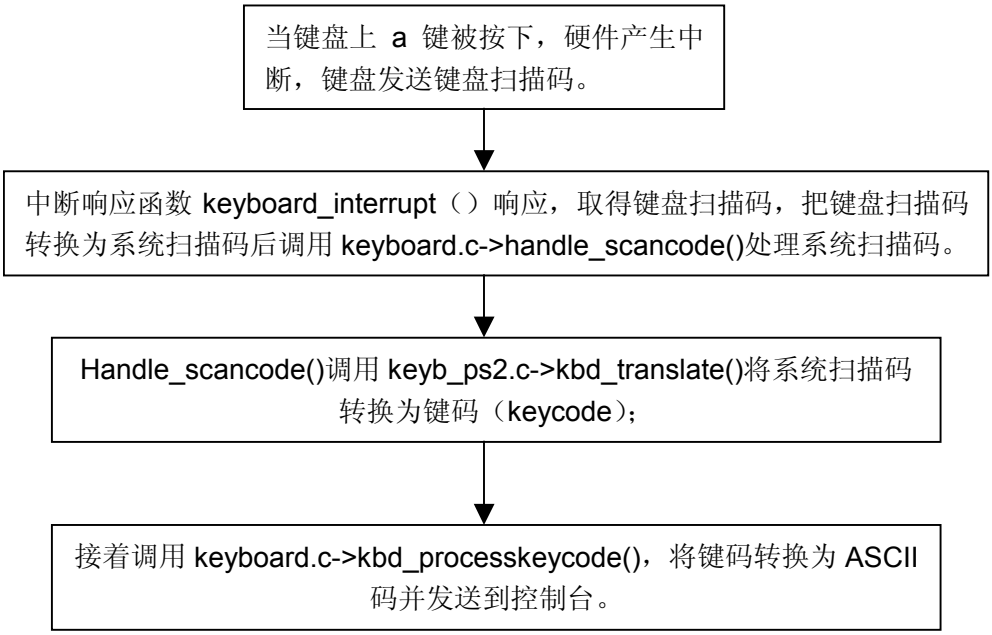
1.3 键码

内核中为了方便键盘按键的处理，把系统扫描码对应成键码（keycode）。键码在 1－128 之间（一般的键盘为 104 键），这样键码就可用 7 位来表示键盘上某个键（内核中用键码字节的高位表示此键是按下还是松开）。

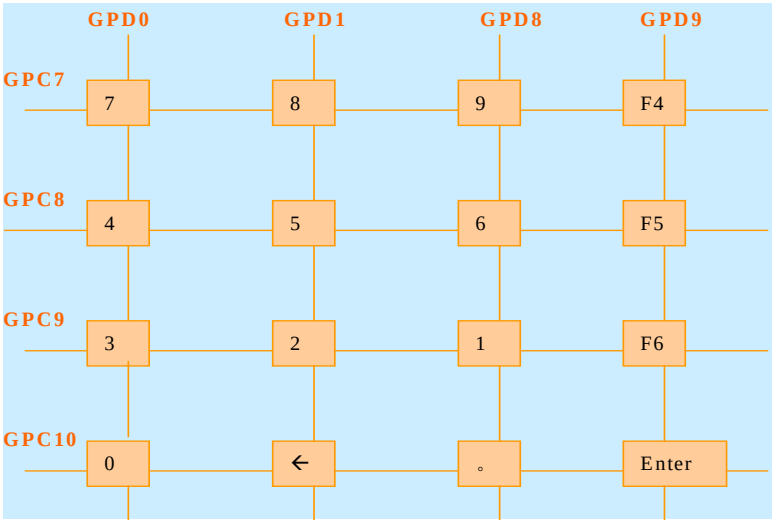
硬件上已经实现了串转并的功能，发送过来的是一个完整的键盘扫描码，软件上只要处理这些扫描码即可。

2．键盘按键值的获取

PC 机的 PS2 键盘按键太多，若是采用轮询方式将浪费 CPU 时间。因此采用的是中断方式。通用的键值获取流程如下：



而实验平台的小键盘是 4*4 的，比大键盘小得多，为简单起见，驱动采用轮询方式。小键盘是直接通过 GPIO 和 CPU 相连，4 行分别与 GPC7~GPC10 相连，4 列分别与 GPD 0，GPD1，GPD8，GPD9 相连，共占用 8 路 GPIO。连线如下：



当按下小键盘上的某个键时，和该键所在行列相连的两路 GPIO 就会导通，其电平就会相同。因此驱动中只需要轮询各路 GPIO 就可以知道该键的扫描码了。比如，先将所有行设为低电平，所有列设为高电平，再依次查询每列，如果哪列变为了低电平，就可以确定该列被按下了；再将所有列设为低电平，所有行设为高电平，依次查询每行，就可以确定哪行被按下了。由得到的行值和列值就可以确定是哪个键被按下了，因此得到按键的键值。

驱动 keybd.c 中部分关键代码如下：

```
//定义键值映射数组。其中 0 表示没有任何键被按下。a~f 表示功能键。
//数组中多余的 0 用于方便下界检查 。
unsigned char    keybd_arr[5][5] = {{0,0,0,0,0},{0,'7','8','9','a'},{0,'4','5','6','b'},
```

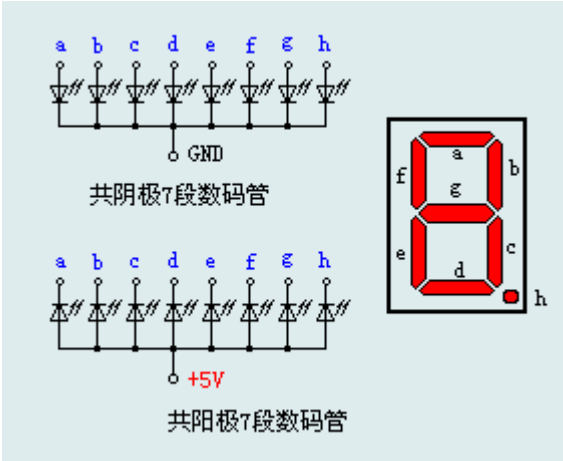
```
{0,'1','2','3','c'},{0,'0','e','.','f'}};

ssize_t keybd_read(struct file *fp, char * buf, size_t size)
{
    int row ,line;
     row = 0;                                  //初始化行下标
     line = 0;                                 //初始化列下标

      key_rGPDCON &= 0xfff5fff5;          //列作输出
      key_rGPDCON |= 0x00050005;
      key_rGPDDAT &= 0xfcfc;              //列设为低电平。
       key_rGPCCON &= 0xffc03fff;         //行做输入，依次查询行是否被按下
      udelay(10000);                           //延时，等待
      if ((key_rGPCDAT & 0x0080) == 0)        row = 1;      //查询第一行
      else if ((key_rGPCDAT & 0x0100) == 0)   row = 2;     //查询第二行                else if
      ((key_rGPCDAT & 0x0200) == 0)   row = 3;     //查询第三行
      else if ((key_rGPCDAT & 0x0400) == 0)   row = 4;     //查询第四行

      key_rGPCCON &= 0xffd57fff;           //行作输出
      key_rGPCCON |= 0x00154000;
      key_rGPCDAT &= 0xf87f;               //行设为低电平。
       key_rGPDCON &= 0xfff0fff0;          //列做输入，依次查询列是否被按下
       udelay(10000);                       //延时，等待
      if ((key_rGPDDAT & 0x0001) == 0)         line = 1;   //查询第一列
      else if ((key_rGPDDAT & 0x0002) == 0)   line = 2;   //查询第二列
      else if ((key_rGPDDAT & 0x0100) == 0)   line = 3;   //查询第三列
      else if ((key_rGPDDAT & 0x0200) == 0)   line = 4;   //查询第四列

       key = keybd_arr[row][line];             //通过映射数组，获得键值
       ……
}
```

需要提醒的是，代码中的两处 udelay(10000)。这是必须的。如果去掉，驱动将无法正常工作。而且时间太小了(比如 5000)也不行。因为行列线需要来回切换成输入或输出模式，硬件有延迟。

3．LED 发光原理

LED 的基本结构是一块电致发光的半导体材料，置于一个有引线的架子上，然后四周用环氧树脂密封，起到保护内部芯线的作用，所以 LED 的抗震性能好。发光二极管的核心部分是由 p 型半导体和 n 型半导体组成的晶片，在 p 型半导体和 n 型半导体之间有一个过渡层，称为 p-n 结。在某些半导体材料的 PN 结中，注入的少数载流子与多数载流子复合时会把多余的能量以光的形式释放出来，从而把电能直接转换为光能。PN 结加反向电压，少数载流子难以注入，故不发光。这种利用注入式电致发光原理制作的二极管叫发光二极管，通称 LED。 当它处于正向工作状态时（即两端加上正向电压），电流从 LED 阳极流向阴极时，半导体晶体就发出从紫外到红外不同颜色的光线，光的强弱与电流有关。

7 段 LED 数码管，则在一定形状的绝缘材料上，利用单只 LED 组合排列成“8”字型的数码管，分别引出它们的电极，点亮相应的点划来显示出 0-9 的数字。



LED 数码管根据 LED 的接法不同分为共阴和共阳两类，了解 LED 的这些特性，对编程是很重要的，因为不同类型的数码管，除了它们的硬件电路有差异外，编程方法也是不同的。左图是共阴和共阳极数码管的内部电路，它们的发光原理是一样的，只是它们的电源极性不同而已。

将多只 LED 的阴极连在一起即为共阴式，而将多只 LED 的阳极连在一起即为共阳式。以共阴式为例，如把阴极接地，在相应段的阳极接上电源，该段即会发光。当然，LED 的电流通常较小，一般均需在回路中接上限流电阻。假如我们将"b"和"c"段接上正电源，其它端接地或悬空，那么"b"和"c"段发光，此时，数码管显示将显示数字“1”。而将"a"、"b"、"d"、"e"和"g"段都接上正电源，其它引脚悬空，此时数码管将显示“2”。其它字符的显示原理类同，读者自行分析即可。

4．LED 驱动原理

在我们的嵌入式平台上是通过 CPLD 逻辑来控制 LED 的，所以 LED 的驱动程序需要和 CPLD 的接口程序相配合。6 个 7 段的 LED1～LED6 是直接连接在 CPLD 上的，由 CPLD 控制，控制逻辑如下：

CPLD 判断当 NGCS2(GPA13)选中，并且地址线低三位 MA2－MA0 分别为 001b，010b，011b，100b，101b，110b 时，读取数据线的低四位 D3-D0，分别作为 LED1～LED6 的输出。

下面给出输入二进制数据和 LED 显示的数字的对应关系。

LED 显示的数字←→	e c h b a d f g ----相应 LED 段
0	1 1 0 1 1 1 1 0
1	0 1 0 1 0 0 0 0
2	1 0 0 1 1 1 0 1
3	0 1 0 1 1 1 0 1
4	0 1 0 1 0 0 1 1
5	0 1 0 0 1 1 1 1
6	1 1 0 0 1 1 1 1
7	0 1 0 1 1 0 0 0
8	1 1 0 1 1 1 1 1
9	0 1 0 1 1 1 1 1
.	0 0 1 0 0 0 0 0
a	1 1 0 1 1 0 1 1

b	1 1 0 0 0 1 1 1
c	1 0 0 0 1 1 1 0
d	1 1 0 1 0 1 0 1
e	1 0 0 1 1 1 1 1
f	1 0 0 0 1 0 1 1

驱动程序相对来说比较简单，主要工作是和 CPLD 逻辑相配合，通过向数据总线低四位 D3-D0 写不同的数据来控制 LED 的输出。下面我们分析一下驱动中的关键函数和代码。

(1) 打开LED函数：使能逻辑

```
int led_open(struct inode *inode, struct file *filp)
{
    led_sle |= 0x2000;
    led_sle_data &= (~0x2000);        //将相应位设低
      ……
}
```

(2) 向LED写入数据函数

```
ssize_t led_write(struct file *fp, char * buf,size_t size)
{
    char key;
    if (get_user(key, buf))            //从用户缓冲中获取数据
        return -EFAULT;
    (*(volatile unsigned char *) led_write_addr) = key;   //写入数据
    ……
}
```

(3) LED的文件控制函数：判断点亮哪个LED灯

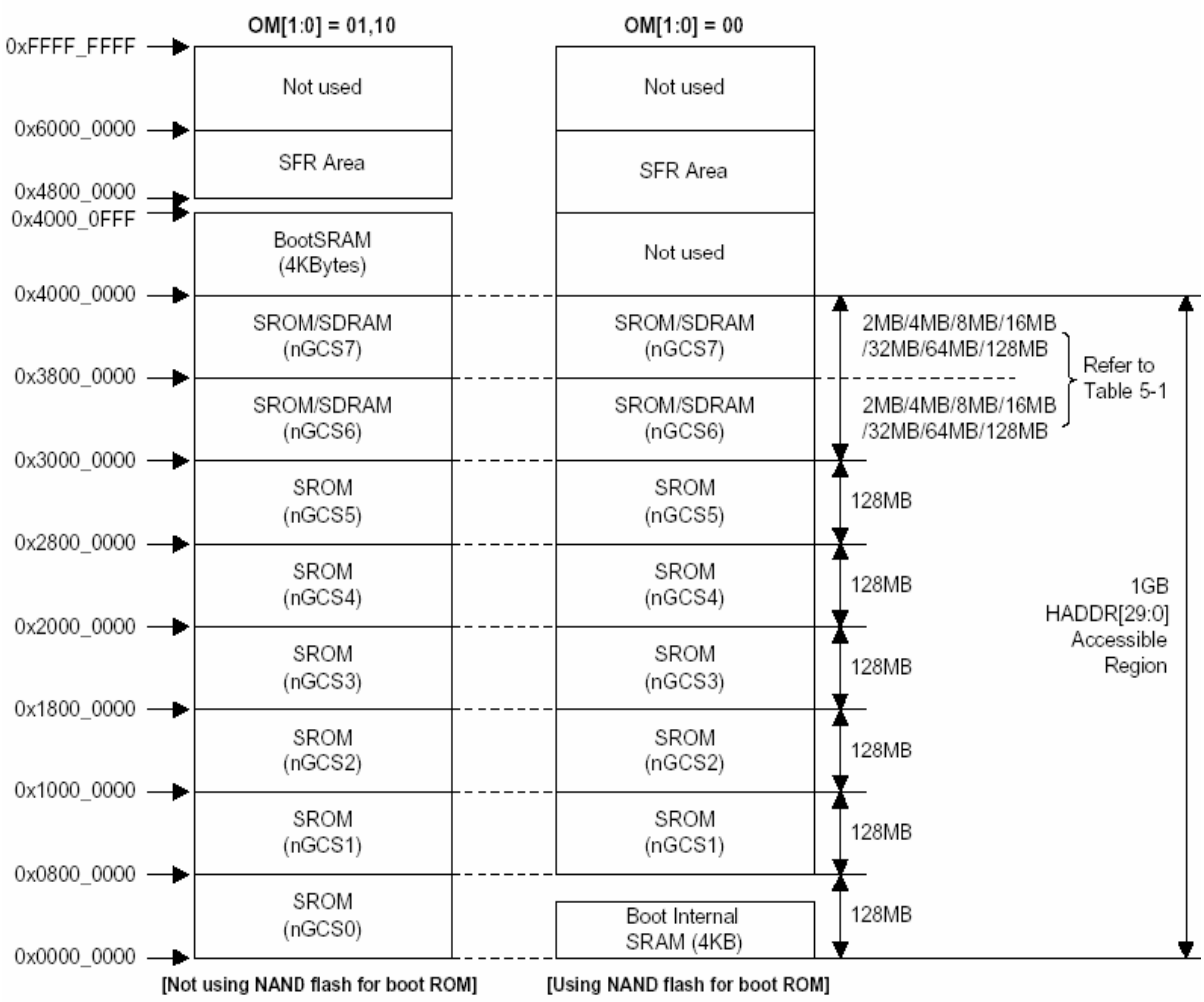
```
int led_ioctl(struct inode *inode, struct file *flip, unsigned int command,
          unsigned long arg)
{
    int err = 0;
    switch (command) {
        case IOCTRL_LED_1:
                        led_write_addr = LED_1;     //点亮第1个LED灯
                        break;
        case IOCTRL_LED_2:
                        led_write_addr = LED_2;     //点亮第2个LED灯
                        break;
        case IOCTRL_LED_3:
                        led_write_addr = LED_3;     //点亮第3个LED灯
                        break;
        case IOCTRL_LED_4:
                        led_write_addr = LED_4;     //点亮第4个LED灯
                        break;
        case IOCTRL_LED_5:
                        led_write_addr = LED_5;     //点亮第5个LED灯
```

```
                                    break;
            case IOCTRL_LED_6:
                                    led_write_addr = LED_6;     //点亮第6个LED灯
                                    break;
            default:
                                    err = -EINVAL;
        }
       ……
    }
```

5.S3C2410相关寄存器说明

与本实验相关的寄存器有nGCS2，GPA，GPC，GPD。

(1) nGCS2：片选。



(2) PGA端口控制器

GPA[13]复用为 nGCS2。

Register	Address	R/W	Description	Reset Value
GPACON	0x56000000	R/W	Configure the pins of port A	0x7FFFFF
GPADAT	0x56000004	R/W	The data register for port A	Undefined
Reserved	0x56000008	–	Reserved	Undefined
Reserved	0x5600000C	–	Reserved	Undefined

GPACON	Bit	Description
GPA22	[22]	0 = Output 1 = nFCE
GPA21	[21]	0 = Output 1 = nRSTOUT (nRSTOUT = nRESET & nWDTRST & SW_RESET(MISCCR[16]))
GPA20	[20]	0 = Output 1 = nFRE
GPA19	[19]	0 = Output 1 = nFWE
GPA18	[18]	0 = Output 1 = ALE
GPA17	[17]	0 = Output 1 = CLE
GPA16	[16]	0 = Output 1 = nGCS5
GPA15	[15]	0 = Output 1 = nGCS4
GPA14	[14]	0 = Output 1 = nGCS3
GPA13	[13]	0 = Output 1 = nGCS2
GPA12	[12]	0 = Output 1 = nGCS1
GPA11	[11]	0 = Output 1 = ADDR26
GPA10	[10]	0 = Output 1 = ADDR25
GPA9	[9]	0 = Output 1 = ADDR24
GPA8	[8]	0 = Output 1 = ADDR23
GPA7	[7]	0 = Output 1 = ADDR22
GPA6	[6]	0 = Output 1 = ADDR21
GPA5	[5]	0 = Output 1 = ADDR20
GPA4	[4]	0 = Output 1 = ADDR19
GPA3	[3]	0 = Output 1 = ADDR18
GPA2	[2]	0 = Output 1 = ADDR17
GPA1	[1]	0 = Output 1 = ADDR16
GPA0	[0]	0 = Output 1 = ADDR0

(4) PGC端口寄存器：

GPC[7,8,9,10]为小键盘行控制线。

Register	Address	R/W	Description	Reset Value
GPCCON	0x56000020	R/W	Configure the pins of port C	0x0
GPCDAT	0x56000024	R/W	The data register for port C	Undefined
GPCUP	0x56000028	R/W	Pull-up disable register for port C	0x0
Reserved	0x5600002C	–	Reserved	Undefined

GPCCON	Bit	Description
GPC15	[31:30]	00 = Input 01 = Output 10 = VD[7] 11 = Reserved
GPC14	[29:28]	00 = Input 01 = Output 10 = VD[6] 11 = Reserved
GPC13	[27:26]	00 = Input 01 = Output 10 = VD[5] 11 = Reserved
GPC12	[25:24]	00 = Input 01 = Output 10 = VD[4] 11 = Reserved
GPC11	[23:22]	00 = Input 01 = Output 10 = VD[3] 11 = Reserved
GPC10	[21:20]	00 = Input 01 = Output 10 = VD[2] 11 = Reserved
GPC9	[19:18]	00 = Input 01 = Output 10 = VD[1] 11 = Reserved
GPC8	[17:16]	00 = Input 01 = Output 10 = VD[0] 11 = Reserved
GPC7	[15:14]	00 = Input 01 = Output 10 = LCDVF2 11 = Reserved
GPC6	[13:12]	00 = Input 01 = Output 10 = LCDVF1 11 = Reserved
GPC5	[11:10]	00 = Input 01 = Output 10 = LCDVF0 11 = Reserved
GPC4	[9:8]	00 = Input 01 = Output 10 = VM 11 = Reserved
GPC3	[7:6]	00 = Input 01 = Output 10 = VFRAME 11 = Reserved
GPC2	[5:4]	00 = Input 01 = Output 10 = VLINE 11 = Reserved
GPC1	[3:2]	00 = Input 01 = Output 10 = VCLK 11 = Reserved
GPC0	[1:0]	00 = Input 01 = Output 10 = LEND 11 = Reserved

(5) GPD端口寄存器

GPD[0,1,8,9]为小键盘列控制线。

Register	Address	R/W	Description	Reset Value
GPDCON	0x56000030	R/W	Configure the pins of port D	0x0
GPDDAT	0x56000034	R/W	The data register for port D	Undefined
GPDUP	0x56000038	R/W	Pull-up disable register for port D	0xF000
Reserved	0x5600003C	–	Reserved	Undefined

GPDCON	Bit	Description
GPD15	[31:30]	00 = Input 01 = Output 10 = VD23 11 = nSS0
GPD14	[29:28]	00 = Input 01 = Output 10 = VD22 11 = nSS1
GPD13	[27:26]	00 = Input 01 = Output 10 = VD21 11 = Reserved
GPD12	[25:24]	00 = Input 01 = Output 10 = VD20 11 = Reserved
GPD11	[23:22]	00 = Input 01 = Output 10 = VD19 11 = Reserved
GPD10	[21:20]	00 = Input 01 = Output 10 = VD18 11 = Reserved
GPD9	[19:18]	00 = Input 01 = Output 10 = VD17 11 = Reserved
GPD8	[17:16]	00 = Input 01 = Output 10 = VD16 11 = Reserved
GPD7	[15:14]	00 = Input 01 = Output 10 = VD15 11 = Reserved
GPD6	[13:12]	00 = Input 01 = Output 10 = VD14 11 = Reserved
GPD5	[11:10]	00 = Input 01 = Output 10 = VD13 11 = Reserved
GPD4	[9:8]	00 = Input 01 = Output 10 = VD12 11 = Reserved
GPD3	[7:6]	00 = Input 01 = Output 10 = VD11 11 = Reserved
GPD2	[5:4]	00 = Input 01 = Output 10 = VD10 11 = Reserved
GPD1	[3:2]	00 = Input 01 = Output 10 = VD9 11 = Reserved
GPD0	[1:0]	00 = Input 01 = Output 10 = VD8 11 = Reserved

三．实验内容和步骤

通过 6 个共阴极的 7 段码 LED，将 4*4 小键盘的最近 6 次的按键值显示在 6 个 LED 灯上。

1. 编写键盘驱动函数 keybd.c，实现函数 keybd_read，keybd_open，keybd_release 和 keybd_init。其中代码主要是 keybd_read，keybd_init 函数。注意 keybd_read 函数需要延时。
2. 编写 LED 驱动函数 led.c。实现函数 led_read，led_open，led_ioctl，led_write，led_release 和 led_init。其中代码主要是 led_write，led_ioctl，led_init 函数。
3. 在主程序 key.c 中编写代码，获取按键值。并根据获得的按键值，点亮相应的 LED 灯。6 个 LED 将显示最近的 6 次按键值。

程序框架如下：

```
int main()
{
    char ret[7]; // ret[0]为本次读入的按键值。ret[1..7]中保存最近 6 次的按键值，
                 //ret [i]为向 LED[i]写入的数据。数组初始化为 0xdf(对应数值 8)，
                 //即使得 6 个 LED 的所有段都点亮，显示数值’8’。
```

```
    打开键盘；
    打开 LED；
      初始化 ret 数组；
      向 LED[i]分别写入 ret[i]；              //点亮 6 个 LED，等待键盘输入。

      While   (1)                             //查询方式
       {
            读取键盘；
            if   (有键按下)
            {
                 ret 数组逻辑右移；
                 向 LED[i]分别写入 ret[i]；     //LED 显示最新的 6 次按键值
             }
            usleep(100000);                      //释放一下 CPU
      }

      关闭 LED；
      关闭键盘；

      return 0;
}
```

注意，while 循环中的 usleep 是必须的。因为采用的是轮询方式，如果不加该语句，CPU 将不停的查询，严重浪费资源。参数也需要适当。如果时间太小，那么按下一次键盘可能会有几次响应；如果太大，那么键盘响应将变得很慢，按键时间要求很长。

四．思考题

1．轮询方式和中断方式有什么区别？

2．在小键盘驱动代码中的 udelay 和主函数中的 usleep 各有什么作用？去掉可以吗？为什么？

实验十五：SPI 实验＋CAN 串行接口总线通讯实验

一．实验目的

1．了解 SPI 总线的原理，掌握通过 SPI 总线访问器件的方法；
2．掌握在 RUIJIARM9-EDU 上的 CAN 总线通讯原理；
3．学习编程实现 MCP2510 的 CAN 总线通讯；

二．实验原理和说明

1. CAN 总线概述

CAN 全称为 Controller Area Network，即控制器局域网，是国际上应用最广泛的现场总线之一。最初，CAN 总线被设计为汽车环境中的微控制器通讯，在车载的各电子控制装置 ECU 之间交换信息，形成汽车电子控制网络。比如，发动机管理系统、变速箱控制器、仪表装置和电子主干系统中均嵌入 CAN 控制装置。

一个由 CAN 总线构成的单一网络中，理论上可以挂接无数个节点。但是，实际应用中节点数目受网络硬件的电气特性所限制。

CAN 可提供高达 1Mbit/s 的数据传输速率，这使实时控制变的非常容易。另外，硬件的错误检测特性也增强了 CAN 的抗电磁干扰能力。

2. CAN 总线技术的应用优点

(1) CAN 网络上的任何一节点均可作为主结点主动地与其他节点交换数据，大大提高了系统的性能。
(2) CAN 网络节点的信息帧可分出优先级，且单帧字节长度短，有很好的实时性。
(3) CAN 的物理层及数据链路层采用独特的设计技术，使其在抗干扰、错误检测能力等方面的性能超过其它总线。
(4) CAN 的通信速率相当高。当网络线的长度不超过 40 米时，其通信速率可达 1Mbit/s。
(5) CAN 总线每帧数据都包含有 CRC 校验及其他校验措施，数据出错率低。
(6) CAN 总线节点在严重错误的情况下，可自动切断与总线的通信联系，以使总线上的其它操作不受影响。

3．典型系统实现方法图

图 3.1 所示的是典型系统实现方法图。在本实验设计中，CAN 控制器用的是 MCP2510 芯片，CAN 收发器用的是 PCA82C251 芯片，节点控制器是 S3C2410 微处理器，如图 3.2 所示。MCP2510 芯片具有 SPI 接口，所以先把 SPI 的相关知识介绍一下。

4. S3C2410 的 SPI 介绍

SPI（Serial Peripheral Interface）系统是一个同步串行外围接口，允许 MCU 与各种外围设备以串行方式进行通信。

S3C2410 微处理器包括两路 SPI，每一路分别有两个 8 位转移寄存器，用来发送和接收数据。在 SPI 进行传输时，数据同时发送和接收。如果只想发送数据，接收到的数据将是无效的；如果只想接收数据，应当发送全是 1 的数据。

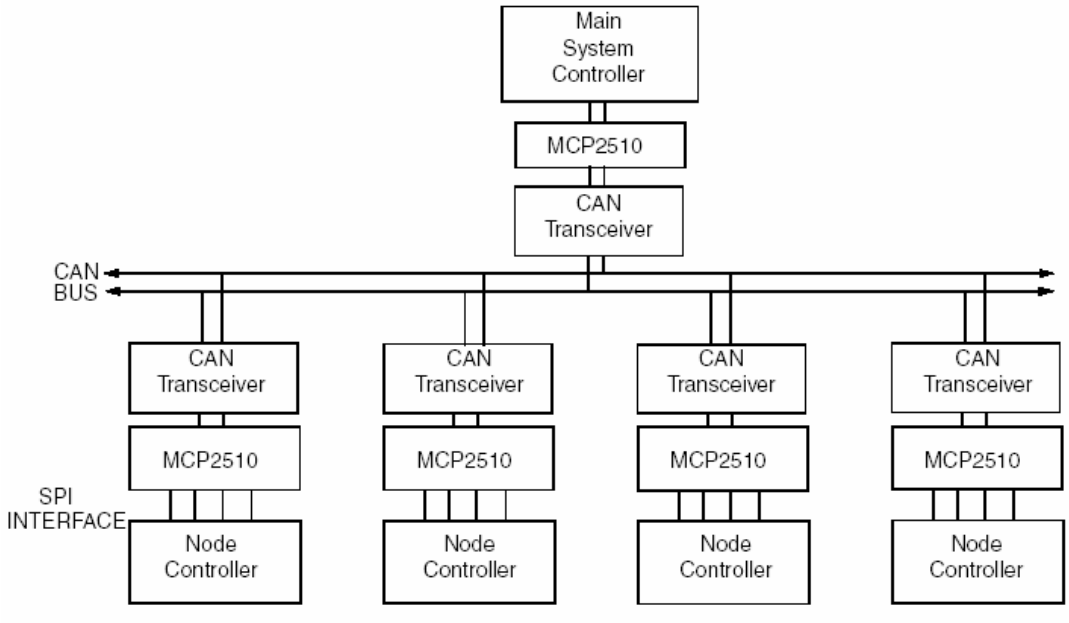


图 3.1 典型系统实现方法图

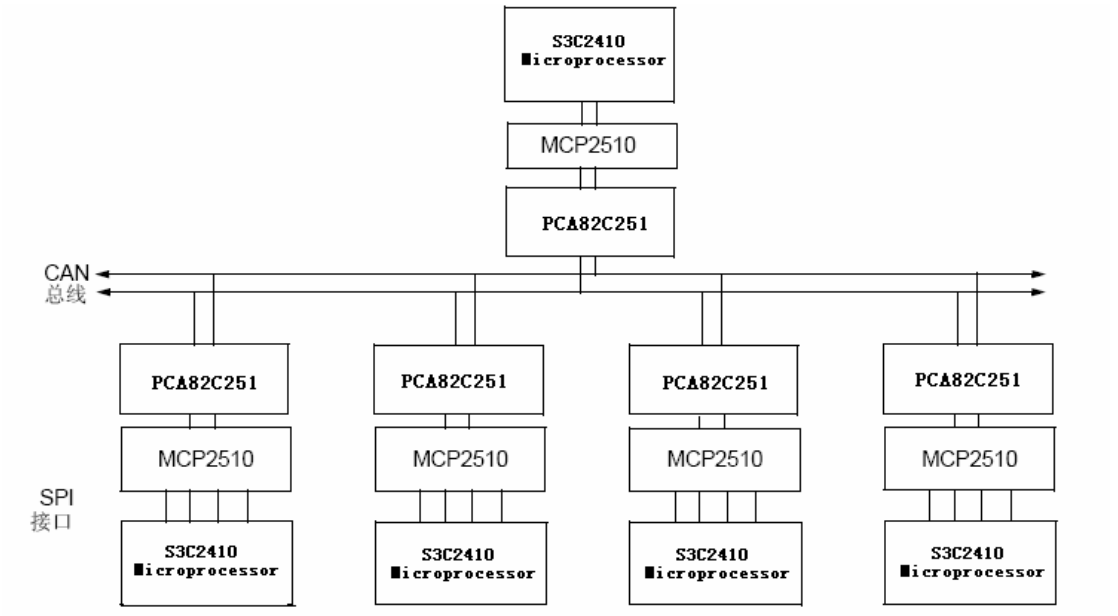


图 3.2 本实验系统实现方法图

4.1 SPI 引脚介绍

SPI 系统使用四个 I/O 引脚，它们是主机输入/从机输出数据线 MISO；主机输出/从机输入数据线 MOSI；串行时钟 SCK 和低有效的选择线 SS。

(1) 串行数据线(MISO、MOSI)

MISO 和 MOSI 用于串行接收和发送数据，先为 MSB(高位)，后为 LSB(低位)。在 SPI 设置为主机方式时，MISO 是主机数据输入线，MOSI 是主机数据输出线。在 SPI 设置为从机方式时，MISO 变成从机数据输出线，而 MOSI 成为从机数据输入线。

(2) 从机选择线(SS)

该控制线用来控制从机的打开或者关闭。

在数据传输过程中，一个 SPI 系统充当主机来控制数据流，其它的系统充当从机。一个主机可以同时转移数据到多个从机，然而，在某一时刻，只能有一个从机向主机发送数据。

在主模式下，外设选择有两种不同的方法：

固定外设选择：SPI 只和固定的一个外设进行数据交换

可变外设选择：数据能和一个以上外设的进行交换

4.2 SPI 传输模式

S3C2410 支持 4 种不同的数据传输模式，分别如下图所示：

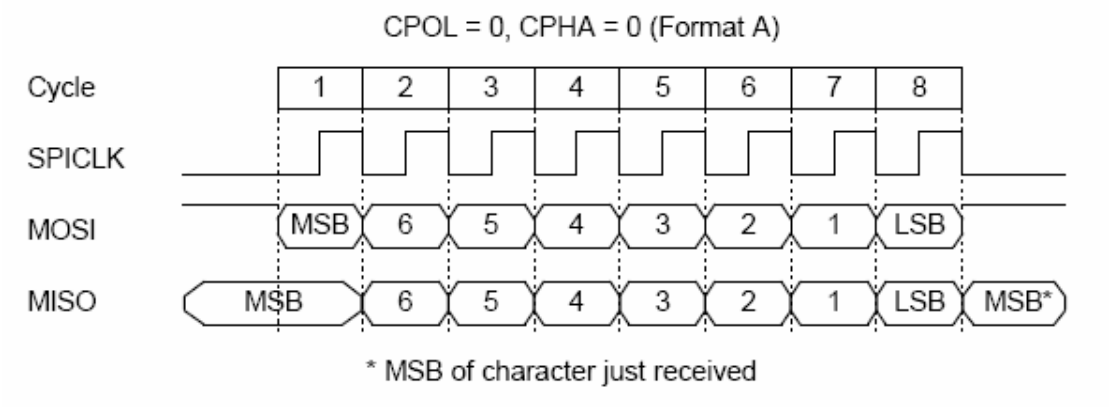


图 4.1 0-0 模式

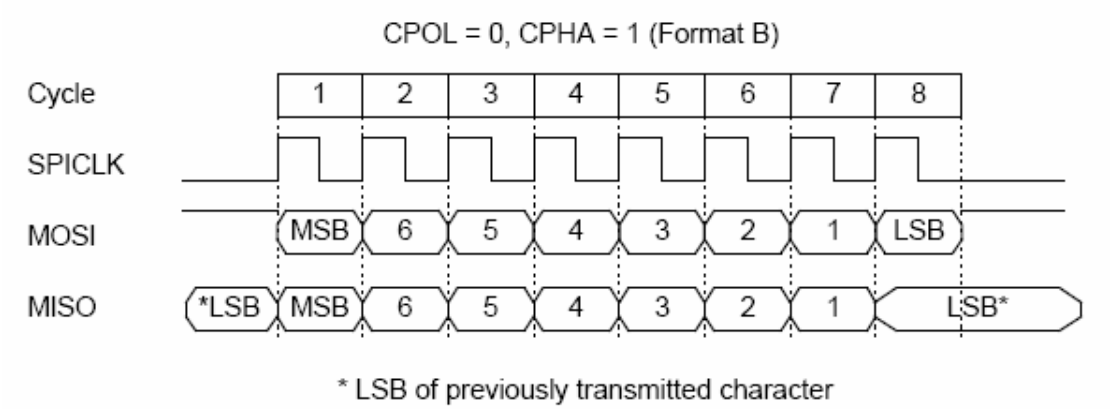


图 4.2 0-1 模式

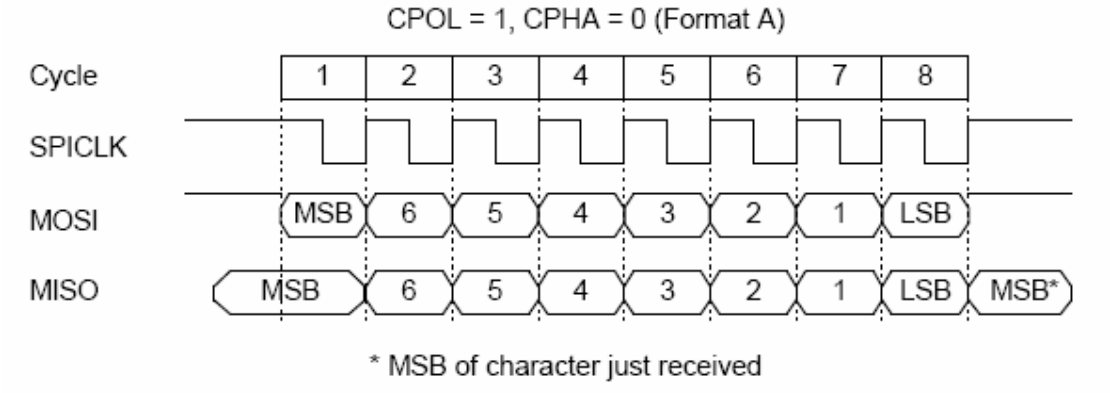


图 4.3 1-0 模式

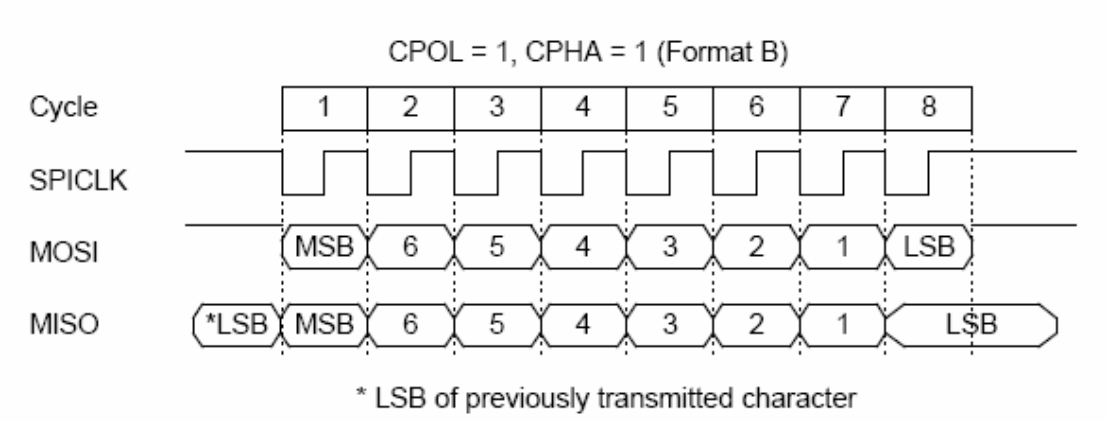


图 4.4 1-1 模式

MCP2510 芯片支持 0，0 和 1，1SPI 模式。这是我们在编程中要特别注意的。

4.3 SPI 寄存器介绍

由于我们准备采用 SPI channel 0，我们将只介绍相应的寄存器。

寄存器	地址	R/W	描述
SPCON0	0x59000000	R/W	SPI 通道 0 控制寄存器
SPSTA0	0x59000004	R	SPI 通道 0 状态寄存器
SPPIN0	0x59000008	R/W	SPI 通道 0 引脚控制寄存器
SPPRE0	0x5900000C	R/W	SPI 通道 0 波特率寄存器
SPTDAT0	0x59000010	R/W	SPI 通道 0 发送数据寄存器
SPRDAT0	0x59000014	R	SPI 通道 0 接收数据寄存器

表 4.1 SPI 寄存器

Register	Address	R/W	Description	Reset Value
SPCON0	0x59000000	R/W	SPI channel 0 control register	0x00
SPCON1	0x59000020	R/W	SPI channel 1 control register	0x00

SPCONn	Bit	Description	Initial State
SPI Mode Select (SMOD)	[6:5]	Determine how and by what SPTDAT is read/written. 00 = polling mode, 01 = interrupt mode 10 = DMA mode, 11 = reserved	00
SCK Enable (ENSCK)	[4]	Determine whether you want SCK enable or not (for only master). 0 = disable, 1 = enable	0
Master/Slave Select (MSTR)	[3]	Determine the desired mode (master or slave). 0 = slave, 1 = master **NOTE:** In slave mode, there should be set up time for master to initiate Tx/Rx.	0
Clock Polarity Select (CPOL)	[2]	Determine an active high or active low clock. 0 = active high, 1 = active low	0
Clock Phase Select (CPHA)	[1]	Select one of two fundamentally different transfer formats. 0 = format A, 1 = format B	0
Tx Auto Garbage Data mode enable (TAGD)	[0]	Decide whether the receiving data only needs or not. 0 = normal mode, 1 = Tx auto garbage data mode **NOTE:** In normal mode, if you only want to receive data, you should transmit dummy 0xFF data.	0

表 4.2 SPI 控制寄存器

Register	Address	R/W	Description	Reset Value
SPSTA0	0x59000004	R	SPI channel 0 status register	0x01
SPSTA1	0x59000024	R	SPI channel 1 status register	0x01

SPSTAn	Bit	Description	Initial State
Reserved	[7:3]		
Data Collision Error Flag (DCOL)	[2]	This flag is set if the SPTDATn is written or the SPRDATn is read while a transfer is in progress and cleared by reading the SPSTAn. 0 = not detect, 1 = collision error detect	0
Multi Master Error Flag (MULF)	[1]	This flag is set if the nSS signal goes to active low while the SPI is configured as a master, and SPPINn's ENMUL bit is multi master errors detect mode. MULF is cleared by reading SPSTAn. 0 = not detect, 1 = multi master error detect	0
Transfer Ready Flag (REDY)	[0]	This bit indicates that SPTDATn or SPRDATn is ready to transmit or receive. This flag is automatically cleared by writing data to SPTDATn. 0 = not ready, 1 = data Tx/Rx ready	1

表 4.3 SPI 状态寄存器

Register	Address	R/W	Description	Reset Value
SPPIN0	0x59000008	R/W	SPI channel 0 pin control register	0x02
SPPIN1	0x59000028	R/W	SPI channel 1 pin control register	0x02

SPPINn	Bit	Description	Initial State
Reserved	[7:3]		
Multi Master error detect Enable (ENMUL)	[2]	The /SS pin is used as an input to detect multi master error when the SPI system is a master. 0 = disable (general purpose) 1 = multi master error detect enable	0
Reserved	[1]	This bit should be '1'.	1
Master Out Keep (KEEP)	[0]	Determine MOSI drive or release when 1byte transmit is completed (only master). 0 = release, 1 = drive the previous level	0

表 4.4 引脚控制寄存器

Register	Address	R/W	Description	Reset Value
SPPRE0	0x5900000C	R/W	SPI cannel 0 baud rate prescaler register	0x00
SPPRE1	0x5900002C	R/W	SPI cannel 1 baud rate prescaler register	0x00

SPPREn	Bit	Description	Initial State
Prescaler Value	[7:0]	Determine SPI clock rate as above equation. Baud rate = PCLK / 2 / (Prescaler value + 1)	0x00

表 4.5 波特率寄存器

Register	Address	R/W	Description	Reset Value
SPTDAT0	0x59000010	R/W	SPI channel 0 Tx data register	0x00
SPTDAT1	0x59000030	R/W	SPI channel 1 Tx data register	0x00

SPTDATn	Bit	Description	Initial State
Tx Data Register	[7:0]	This field contains the data to be transmitted over the SPI channel.	0x00

表 4.6 发送数据寄存器

Register	Address	R/W	Description	Reset Value
SPRDAT0	0x59000014	R	SPI channel 0 Rx data register	0x00
SPRDAT1	0x59000034	R	SPI channel 1 Rx data register	0x00

SPRDATn	Bit	Description	Initial State
Rx Data Register	[7:0]	This field contains the data to be received over the SPI channel.	0x00

表 4.7 接收数据寄存器

4.4 SPI 程序流程

(1) 初始化 SPPRE0 寄存器，对波特率进行设置；
(2) 设置 SPCON0 寄存器，设置相应的 SPI 模式，这里采用 polling 模式，数据传输模式是 0，0 SPI 模式；
(3) 向 SPTDAT0 寄存器写 0xFF 十次，为了初始化 MCP2510；
(4) 设置 GPIO 引脚，用来充当片选，变低以激活 MCP2510；
(5) 发送数据：检查发送状态位 REDY 是否位 1，是就向 SPTDAT0 寄存器写数据；
(6) 接收数据：
SPCON0 寄存器的 TAGD 不使能（即普通模式），向 SPTDAT0 寄存器写入 FF，在确认 REDY 后，从读缓存读数据；TAGD 使能，确认 REDY 后，从读缓存读数据，然后自动的开始发送数据；
(7) 设置 GPIO 引脚，信号变高，片选不使能；

5．PCA82C251 芯片介绍

PCA82C251是CAN 协议控制器和物理总线之间的接口。它主要在卡车和公共汽车中速度达1Mbaud的应用中使用。这个器件向总线提供了差动的发送能力，向CAN 控制器提供了差动的接收能力。

5.1 PCA82C251 芯片的特性

(1) 热保护
(2) 在24V系统中防止电池对地的短路
(3) 待机模式电流低
(4) 不上电的节点不会影响总线线路
(5) 至少可以连接110个节点
(6) 高速（可达1Mbaud）
(7) 对电磁干扰有高的抗干扰性

5.2 PCA82C251 芯片管脚

助记符	管脚	描述	实际接法
TXD	1	发送数据输入	连 MCP2510 的 TXCAN 引脚
GND	2	地	接地
Vcc	3	电源电压	接 5V 电压
RXD	4	接收数据输出	连 MCP2510 的 RXCAN 引脚
Vref	5	参考电压输出	悬空
CANL	6	输入/输出低电平 CAN 电压	引出
CANH	7	输入/输出高电平 CAN 电压	引出
Rs	8	斜率电阻输入	接地

表 5.1 PCA82C251 管脚图

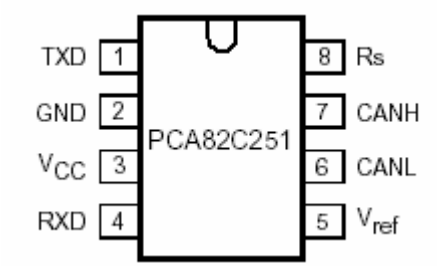


图 5.1 管脚配置图

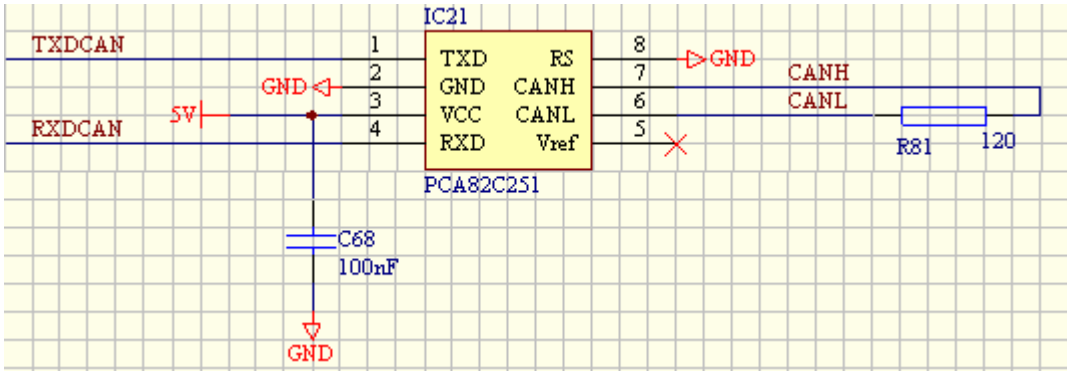


图 5.2 PCA82C251 原理图

5.3 PCA82C251 功能描述

PCA82C251 是CAN 协议控制器和物理总线之间的接口。它主要在卡车和公共汽车中速度达1Mbaud的应用中使用。这个器件向总线提供了差动的发送能力，向CAN 控制器提供了差动的接收能力。它完全符合“ISO 11898-24 V”标准。

一个限流电路可防止发送器的输出级对电池电压的正端和负端短路。虽然在出现这种故障条件时功耗将增加，但这种特性可以防止破坏发送器的输出级。

在节点温度超过大约160度时，两个发送器输出端的极限电流将减少。由于发送器是功耗的主要部分，因此芯片温度会迅速降低。IC 的所有其他部分将继续工作。当总线短路时尤其需要这个热保护电路。

CANH、 CANL两条线也能防止受到在汽车环境下可能发生的电气瞬变现象的影响。

管脚8 Rs可以选择三种不同的工作模式：高速模式、斜率控制模式和待机模式。

在高速工作模式下，发送器输出级晶体管将以尽可能快的速度开、闭。在这种模式下，不采取任何措施限制上升斜率和下降斜率。建议使用屏蔽电缆以避免射频干扰RFI问题。把管脚8接地可选择高速模式。

斜率控制模式允许使用非屏蔽双绞线或平行线作为总线。为降低射频干扰RFI，应限制上升斜率和下降斜率。上升斜率和下降斜率可通过由管脚8接至地的连接电阻进行控制。斜率正比于管脚8的电流输出。

如果向管脚8加高电平，则电路进入低电流待机模式。在这种模式下，发送器被关闭，而接收器转至低电流。若在总线上检测到显性位（差动总线电压>0.9V），RXD将变为低电平。微控制器应通过将收发器切换至正常工作状态（通过管脚8），对此信号作出响应。由于在待机方式下接收器是慢速的，因此，当位速率很高时，第一个报文将丢失。

由上面的原理图可以看到，我们采用的是高速工作模式。

6. MCP2510 芯片介绍

该部分节选自 MCP2510 芯片手册，主要介绍了一些重要的知识。如果读者感兴趣，可以参考该手册。

6.1 器件功能介绍

MCP2510是一款独立CAN控制器，是为简化连接CAN总线的应用而开发的。图6.1简要显示了MCP2510的结构框图。该器件主要由三个部分组成：

(1) CAN 协议引擎
(2) 用来为器件及其运行进行配置的控制逻辑和SRAM寄存器
(3) SPI协议模块

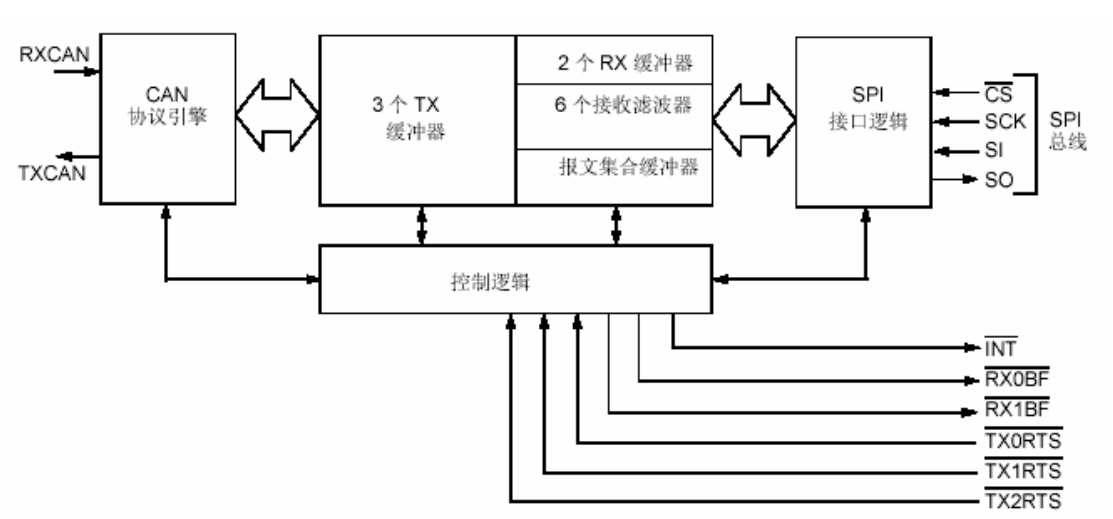


图6.1 结构框图

CAN 协议引擎的功能是处理所有总线上的报文发送和接收。报文发送时，首先将报文装载到正确的报文缓冲器和控制寄存器中。利用控制寄存器位，通过SPI 接口或使用发送使能引脚均可启动发送操作。通过读取相应的寄存器可以检查通信状态和错误。任何在CAN总线上侦测到的报文都会进行错误检测，然后与用户定义的滤波器进行匹配，以确定是否将其转移到两个接收缓冲之一中。

MCU 通过SPI 接口与器件进行通信。通过使用标准SPI读写命令对寄存器进行读写操作。

所提供的中断引脚提高了系统的灵活性。器件上有一个多用途中断引脚，以及各接收缓冲器专用的中断引脚，以及各接收缓冲器专用的中断引脚，可用于指示有效报文是否被接收和载入各接收缓冲

器。是否使用专用中断引脚由用户决定，若不使用，也可用通用中断引脚和状态寄存器（通过SPI 接口访问）确定有效报文是否已被接收。

器件还有三个引脚，用来将装载在三个发送缓冲器之一中的报文立即发送出去。是否使用这些引脚由用户决定，若不使用，也可通过SPI 接口访问控制寄存器的方式来启动报文发送。

6.2 芯片管脚介绍

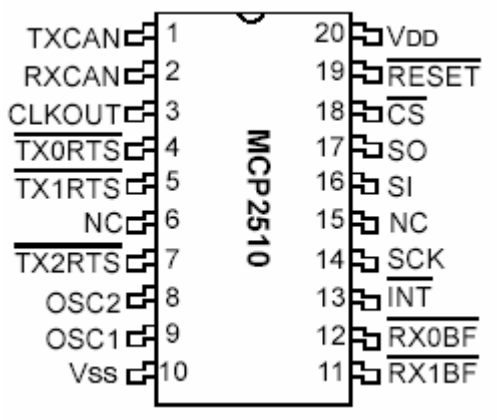


图 6.2 管脚配置图

在我们的实验中，并不需要用到所有的引脚，下面的表格将只对我们用到的引脚进行介绍。如果读者想了解其它的引脚信息，请参考该芯片手册。

名称	管脚	描述	实际接法
TXCAN	1	连接到 CAN 总线的发送输出引脚	接 PCA82C251 的 TXD 引脚
RXCAN	2	连接到 CAN 总线的接收输入引脚	接 PCA82C251 的 RXD 引脚
OCS2	8	振荡器输出	接 16MHZ 的晶振
OCS1	9	振荡器输入	接 16MHZ 的晶振
Vss	10	逻辑和 I/O 引脚的参考地端	接地
INT	13	中断输出引脚	接 GPG13 引脚
SCK	14	SPI 接口时钟输入引脚	接 CPU 的 SPCK 引脚
SI	16	SPI 接口数据输入引脚	接 CPU 的 MOSI 引脚
SO	17	SPI 接口输入输出引脚	接 CPU 的 MISO 脚
CS	18	SPI 接口片选输入引脚	接 CPU 的 GPIO 引脚
RESET	19	低电平有效器件复位输入引脚	接 CPU 的 TIOB5 引脚
VDD	20	逻辑和 I/O 引脚的正电源	接 5V 电压

表 6.1 管脚介绍

MCP2510 芯片原理图如下图 6.3 所示。

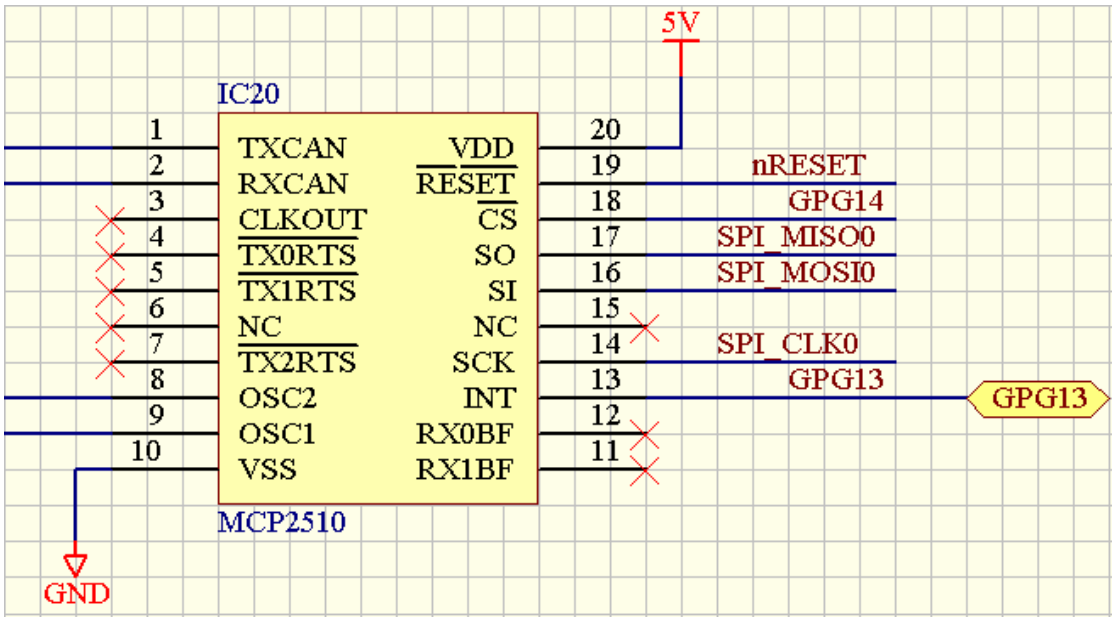


图6.3 MCP2510芯片原理图

7. CAN 报文帧

MCP2510 支持CAN 2.0B技术规范中所定义的标准数据帧、扩展数据帧以及远程帧（标准和扩展）。这里将只介绍标准数据帧和扩展数据帧。

(1) 标准数据帧

CAN 标准数据帧如图7.1所示。与其它所有帧相同，帧以起始帧(SOF) 位开始。SOF 为显性状态，允许所有节点进行硬同步。在SOF之后是仲裁字段，由12 个位组成，分别为11个识别位和一个远程发送请求 (RTR) 位。RTR 位用于

区分报文是数据帧 (RTR位为显性) 还是远程帧(RTR位为隐性状态)。在仲裁字段之后是控制字段，由6 个位组成。控制字段的第一位为识别扩展（IDE） 位，该位为显性状态时，说明这是标准帧。识别扩展位的下一位为零保留位(RB0)，这一保留位将由CAN 协议定义为显性位。控制字段的其余4位为数据长度码（DLC），说明了报文中包含的数据字节数。控制字段之后为数据字段，包含正在发送的数据字节。数据字段长度由上述数据长度码DLC定义（0-8字节）。数据字段后为循环冗余校验字段(CRC)，用来检测报文传输错误。CRC字段包含一个15位的CRC序列，之后是隐性CRC定界位。最后一个字段是确认字段，由两个位组成。在确认间隙(ACK slot) 位执行期间，发送节点发出一个接收位。任何收到无错误帧的节点会发回一个显性位(无论该节点是否配置为接收该报文与否)，确认帧收到无误。确认字段以隐性确认定界符结束，该字符可能不允许被改写为显性位。

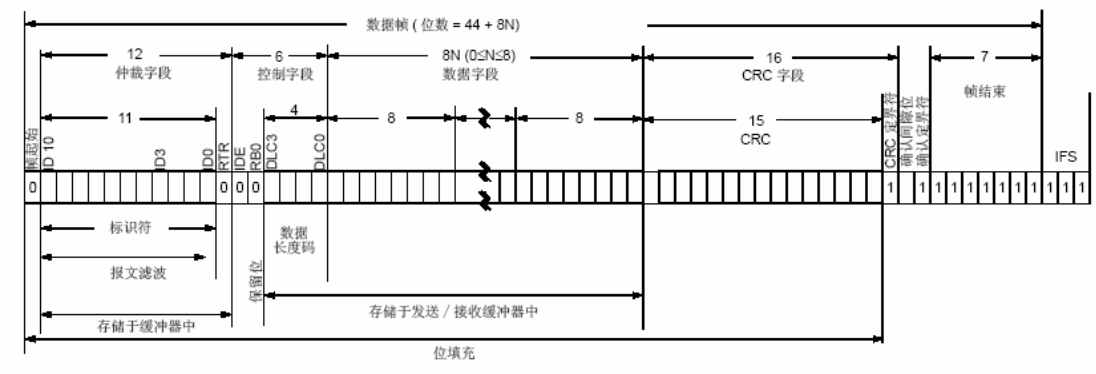


图7.1 标准数据帧

(2) 扩展数据帧

在扩展数据帧中，紧随SOF位的是32位仲裁字段，如图7.2所示。仲裁字段的前11位为29位标识符的最有效位(基本ID)。紧随这11位的是替换远程请求(SRR)位，定义为隐性状态。SRR位之后是IDE位，该位隐性时表示这是扩展的CAN帧。应注意的是，如果在扩展帧标识符的前11位发送完后，总线仲裁无果，而此时仲裁中的节点之一发出标准数据帧(11位标识符)，那么，由于节点发出了显性IDE位而使标准CAN帧赢得总线仲裁。另外，扩展CAN帧的SRR位应为隐性，以允许正在发送标准CAN远程帧的节点发出显性RTR位。SRR位和IDE位之后是标识符的其余18位（扩展ID）以及一个远程发送请求位。为使标准帧和扩展帧都能在共享网络上发送，应将29位的扩展报文标识符拆分成最高11位和最低18位两部分。拆分后可确保IDE位在标准数据帧和扩展帧中的位置保持不变。仲裁字段之后是6位控制字段。控制字段前两位为保留位，必须定义为显性位。其余4位为数据长度码(DLC),说明报文中包含的数据字节数。扩展数据帧的其它部分（数据字段，CRC字段，确认字段，帧结尾和间断）与标准数据帧的结构相同。

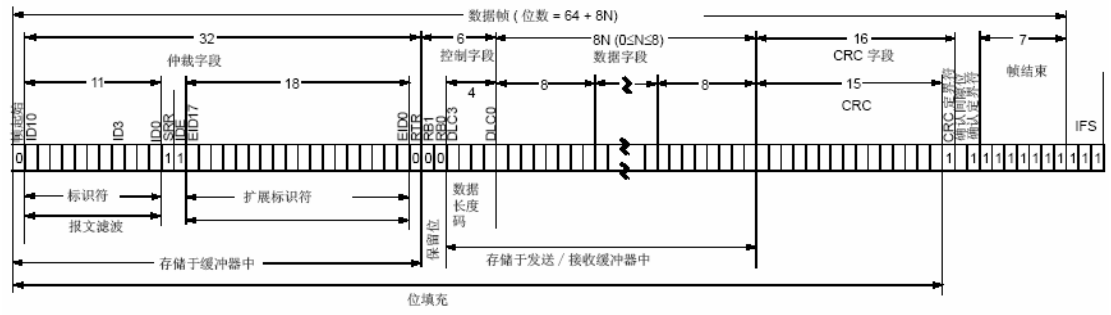


图7.2 扩展数据帧

8. 报文发送

8.1 发送缓冲器

MCP2510采用三个发送缓冲器。每个发送缓冲器占据14字节的SRAM，并映射到存储器中。其中第一字节TXBNCTRL是与报文缓冲器相关的控制寄器。该寄存器中的信息决定了报文在何种条件下被发送，并在报文发送时指示其状态。用5个字节用来装载标准和扩展标识符以及其它报文仲裁信息。最后8个字节用来装载等待发送的报文的八个可能的数据字节。

对于可访问报文缓冲器的单片机来说，必须清除TXBNCTRL寄存器TXREQ 位，表明发送缓冲器无等待发送的报文。至少须将TXBNSIDH、TXBNSIDL 和 TXBNDLC寄存器装载数据。如果报文包含数据字节，还需对TXBNDm寄存器进行装载。若报文采用扩展标识符，应对TXBNEIDm寄存器进行装载，并将 TXBNSIDL寄存器的EXIDE位置位。在报文发送之前，单片机应CANINTE寄存器TXINE位进行初始化，以便在报文发送时使能或禁止中断的产生。MCU还应对TXBNCTRL寄存器的TXP优先级控制位进行初始化。

8.2 发送优先级

发送优先级是指MCP2510内部等待发送报文之间的优先级。它与CAN协议中固有的报文仲裁的优先级无关。在发送起始帧SOF之前，器件将所有等待发送报文的发送缓冲器的优先级进行比较。具有较高优先级的发送缓冲器将首先发送。例如，如果发送缓冲器0的优先级设定比发送缓冲器1高，缓冲器0将首先发送。如果两个缓冲器的优先级相同，编号较高的发送缓冲器将优先发送。例如，如果发送缓冲器1与发送缓冲器0的优先级设定相同，缓冲器1将优先发送。发送优先级的设定共有4个等级。如果某个发送缓冲器的TXBNCTRL寄存器的TXP<1:0> 设定为11，该发送缓冲器具有最高的发送优先级。如果TXBNCTRL寄存器的TXP<1:0> 设定为00，该发送缓冲的发送优先级最低。

8.3 发送启动

通过设定控制寄存器中TXBNCTRL寄存器的TXREQ发送控制位可以启动相应发送缓冲器的报文发送。通过SPI接口写寄存器或向某一发送缓冲器TXNRTS 引脚输入低电平可以进行设定。如果选择SPI接口方式进行位设定以启动报文发送，可以同时设定TXREQ位和TXP优先级控制位。当TXBNCTRL寄存器TXREQ 置位后，TXBNCTRL寄存器ABTF，TXBNCTRL寄存器的MLOA位和TXERR位都将被清除。

将TXBNCTRL寄存器的TXREQ位置位并不能启动报文发送，仅将发送缓冲器标记为准备发送。当器件检测到总线空闲时，才会启动报文发送。优先级最高的报文将首先发送。

报文发送成功后，TXBNCTRL寄存器的TXREQ位将被清除，CANINTF寄存器的TXNIF位将被置位，置位后将产生中断。如果报文发送失败，TXBNCTRL寄存器的TXREQ将保持置位，表明该报文仍在等待发送。此时以下条件标志之一将被置位。如果报文发送已开始但发生错误，TXBNCTRL寄存器的TXERR和 CANINTF寄存器的MERRF位将被置位，此时在CANINTE寄存器的MERRE位置位后，器件将会在INT引脚产生中断。若发送报文总线仲裁失败，TXBNCTRL寄存器的MLOA位将被置位。发送流程图如下：

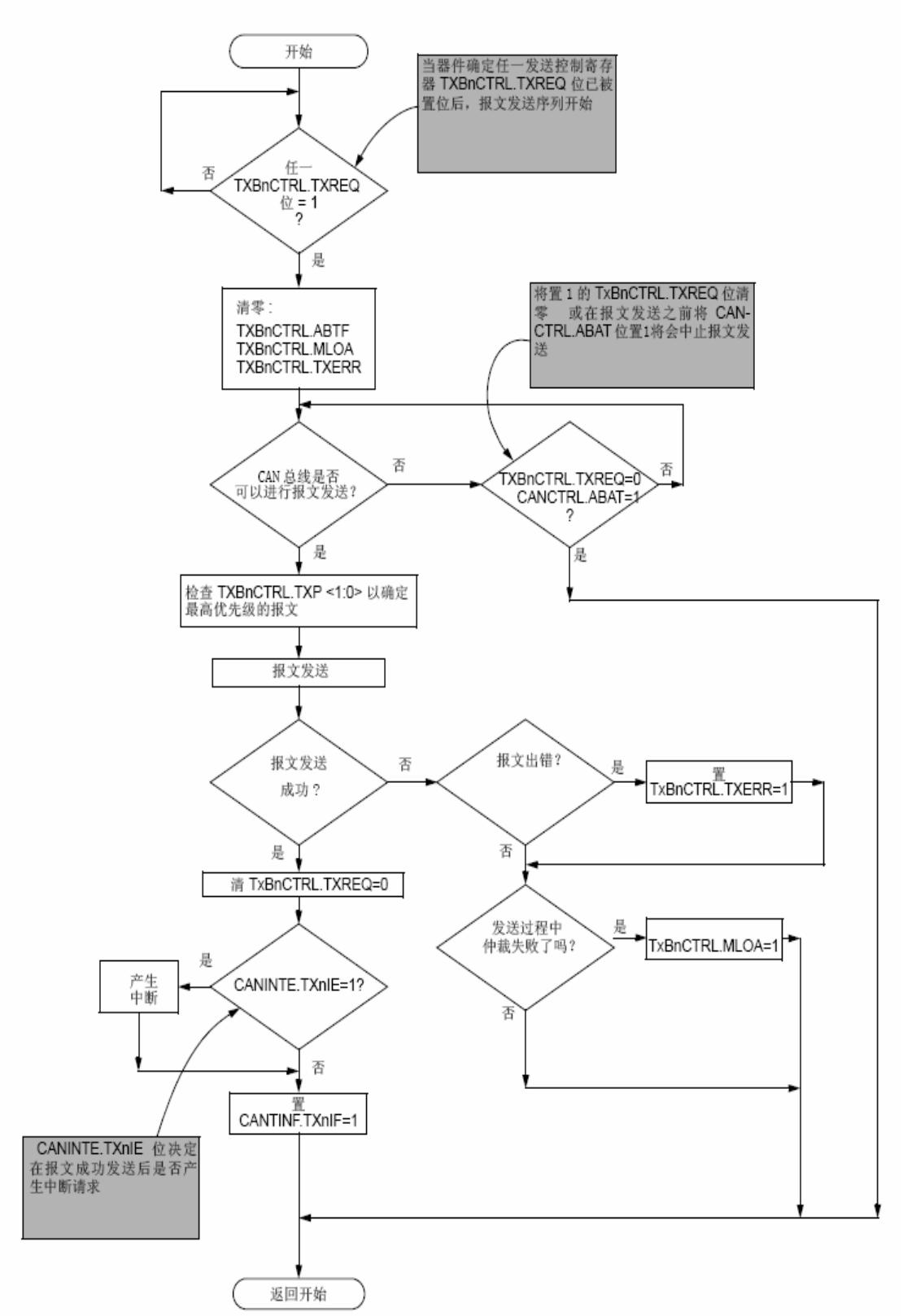


图 8.1 报文发送流程图

8.4 寄存器介绍

U-0	R-0	R-0	R-0	R/W-0	U-0	R/W-0	R/W-0
—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0
bit 7							bit 0

bit 7 **未用：**读作 '0'

bit 6 **ABTF**: 报文发送中止标志
1 = 报文中止
0 = 报文发送成功

bit 5 **MLOA**: 报文仲裁失败
1 = 报文发送中仲裁失败
0 = 报文发送中仲裁未失败

bit 4 **TXERR**: 检测到的发送错误
1 = 报文发送中发生了总线错误
0 = 报文发送中未发生总线错误

bit 3 **TXREQ**: 报文发送请求
1 = 缓冲器等待报文发送
(单片机将此位置位以请求报文发送 - 报文发送后该位自动清零)
0 = 缓冲器无报文发送
(单片机将此位清零以请求中止报文发送)

bit 2 **未使用：读作 '0'**

bit 1-0 **TXP<1:0>**: 发送缓冲器优先级
11 = 最高的报文发送优先级
10 = 中偏高的报文发送优先级
11 = 中偏低的报文发送优先级
00 = 最低的报文发送优先级

图8.2 TXBNCTRL 发送缓冲器N 控制寄存器

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID<10:3>**: 标准标识符 <10:3>

其中：			
R = 可读位	W = 可写位	U = 未用，读作 '0'	
-n = 上电复位值	'1' = 置位值	'0' = 清零值	x = 未知值

图8.3 TXBNSIDH 发送缓冲器N 标准标识符高位

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID<2:0>**: 标准标识符位数 <2:0>

bit 4 **未用**：读作 '0'

bit 3 **EXIDE**: 扩展标识符使能
1 = 报文将发送扩展标识符
0 = 报文将发送标准标识符

bit 2 **未用**：读作 '0'

bit 1-0 **EID<17:16>**: 扩展标识符位数 s <17:16>

图8.4 TXBNSIDL 发送缓冲器N 标准标识符低位

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

bit 7-0 **EID<15:8>**: 扩展标识符位数 <15:8>

图8.5 TXBNEID8 发送缓冲器N 扩展标识符高位

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

bit 7-0 **EID<7:0>**: 扩展标识符位数 <7:0>

图8.6 TXBNEID0 发送缓冲器N 扩展标识符低位

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	RTR	—	—	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

bit 7 **未用**：读作 '0'

bit 6 **RTR**: 远程发送请求位
1 = 发送的报文为远程帧
0 = 发送的报文为数据帧

bit 5-4 **未用**：读作 '0'

bit 3-0 **DLC<3:0>**: 数据长度码
设定发送的数据长度 (0 到 8 字节)

Note: 可以将 DLC 设定为大于 8 的值，但只发送 8 个字节

图8.7 TXBNDLC 发送缓冲器N 数据长度码

TXBNDM - 发送缓冲器 N 数据段字节 m
(地址 : 36h-3Dh, 46h-4Dh, 56h-5Dh)

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
TXBNDm 7	TXBNDm 6	TXBNDm 5	TXBNDm 4	TXBNDm 3	TXBNDm 2	TXBNDm 1	TXBNDm 0
bit 7							bit 0

TXBNDM7:TXBNDM0: 发送缓冲器 n 数据字段字节 m

图 8.8 发送缓冲器寄存器

9. 报文接收

9.1 报文接收缓冲器

MCP2510具有两个全文接收缓冲器。每个接收缓冲器配备有多个验收滤波器。除上述专用接收缓冲器外，MCP2510还具有单独的报文集成缓冲器(MAB)，可作为第三个接收缓冲器，如图9.1所示。

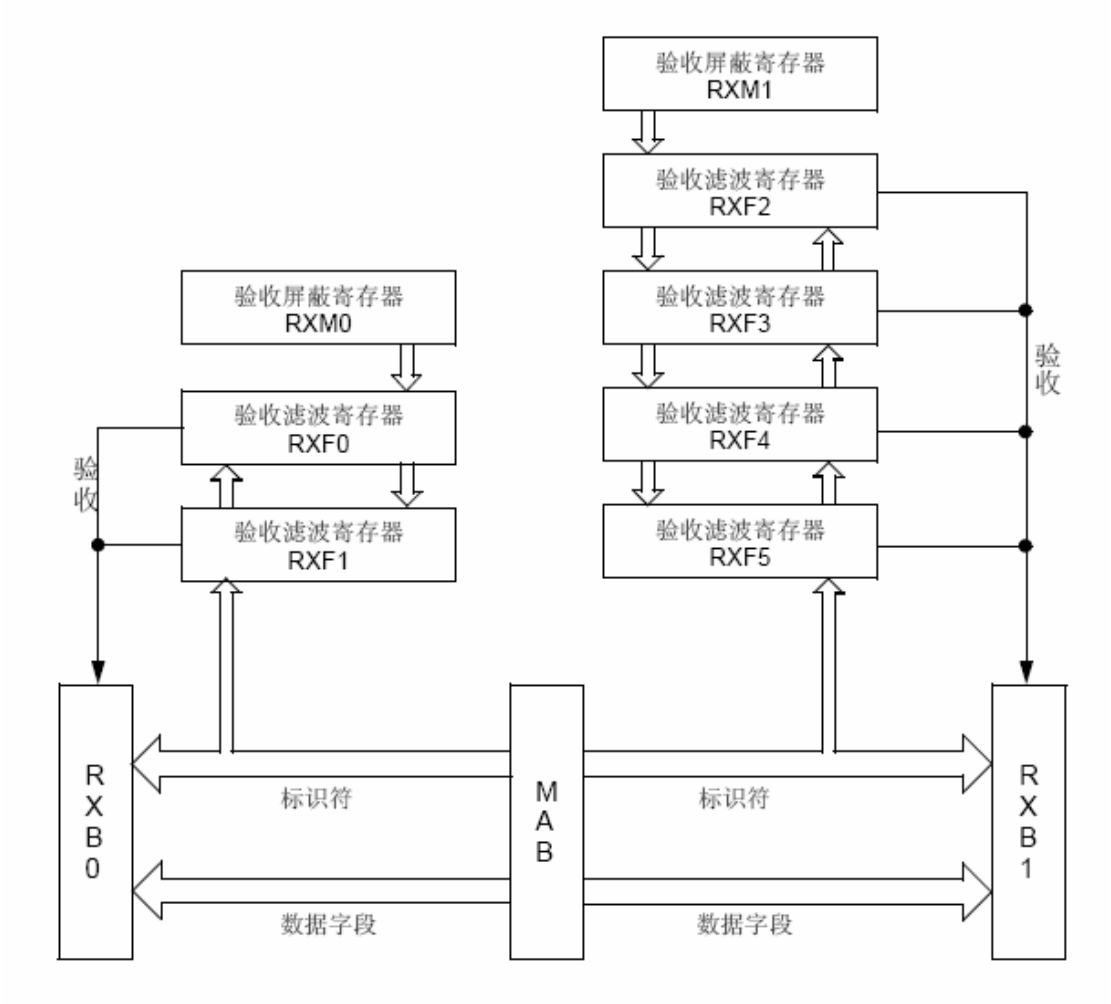


图 9.1 接收缓冲器原理图

9.2 接收缓冲器

在三个接收缓冲器中，MAB总能够接收来自总线的下一条报文。其余两个接收缓冲器RXB0和RXB1则从协议引擎接收完整的报文。当其中一个缓冲器处于接收等待或保存着上一条接收到的报文时，MCU可对另一缓冲器进行访问。

MAB对接收到的报文进行组合，并将满足验收滤波器条件的报文传送到至RXBN缓冲器。

当报文传送至某一接收缓冲器，与该接收缓冲器对应的CANINTF寄存器的RXNIF位将置1。一旦缓冲器中的报文处理完毕，MCU就必须将该位清除以接收下一条报文。 该控制位提供的锁定功能确保在MCU尚未处理完上一条报文前， MCP2510不会将新的报文载入接收缓冲器。如果CANINTE寄存器的RXNIE位被置1，器件会在INT引脚产生一个中断，显示接收到有效报文。

9.3 接收优先级

RXB0是具有较高优先级的寄存器，并配置有2个报文验收滤波寄存器。RXB1优先级较低，配置有4个验收滤波器寄存器。RXB0的验收滤波寄存器数量较少，因此RXB0接受匹配条件更为严格，表明RXB0具有较高的优先级。此外通过配置RXB0CTRL寄存器，还可以实现以下功能：如果RXB0中装有上一条有效报文，而另一条报文也正被接收，该项设置将避免发生溢出错误。无论新的报文是否符合RXB1验收条件，都将被滚存至RXB1。每个接收缓冲器还分别配置有一个可编程验收滤波屏蔽寄存器(见第6.5.4节)。

当报文被接收时，RXBNCTRL 寄存器的<3:0>位状态将显示使能该接收操作的验收滤波器的编号，以及接收到的报文是否为远程传输请求。通过RXBNCTRL寄存器的RXM位可以设定特殊接收工作模式。该位通常设置为00，以接收所有被验收滤波器器认可的报文。在这种情况下，标准或扩展帧报文的接收与否取决于验收滤波寄存器中RFXNSIDL寄存器的EXIDE控制位的状态。如果RXBNCTRL寄存器的RXM设定值为01或10，接收缓冲器将分别只接收标准帧或扩展帧。如果验收滤波寄存器RFXNSIDL中的EXIDE位的设置不对应于RXBNCTRL寄存器的RXM工作模式，验收滤波器将不起作用。上述两种由RXBNCTRL寄存器的RXM控制位决定的接收模式可以应用在总线上只有标准帧或扩展帧的系统中。如果RXBNCTRL寄存器的RXM位设置为11，无论验收滤波器的设置值是什么，缓冲器都将接收所有报文。如果报文在帧结束前出错，在MAB中组合的出错前的那部分报文将被移入缓冲器。该工作模式可在CAN系统调试时使用，一般不在实际系统环境中使用。

9.4 报文验收滤波器及屏蔽寄存器

验收滤波器及屏蔽寄存器用来确定报文集成缓冲器中的报文是否应被载入接收缓冲器。一旦MAB接收到有效报文，报文中的标识符字段将与过滤寄存器中的值进行比较。如果两者匹配，该报文将被载入相应的接收缓冲器。滤波屏蔽寄存器用来确定滤波器对标识符中的哪些位进行校验。表9.1所示的真值表显示了标识符中每一位是如何与验收屏蔽器和滤波器进行比较，以确定该报文是否应被载入接收缓冲器。屏蔽寄存器主要确定对标识符中的哪一位进行滤波。如果某屏蔽位设置为零，对应的标识符位将被自动接收而不被滤波。

屏蔽位 n	过滤位 n	报文标识符位n001	接收或拒绝位 n
0	X	X	接受
1	0	0	接受
1	0	1	拒绝
1	1	0	拒绝
1	1	1	接受

表 9.1 滤波/屏蔽寄存器真值表（X 可为任意值）

RXB0接收缓冲器配备有验收滤波寄存器RXF0和RXF1，以及过滤屏蔽寄存器RXM0。RXB1配备有验收滤波寄存器RXF2、RXF3、RXF4和RXF5以及滤波屏蔽寄存器RXM1。当新报文符合验收滤波条件并被载入接收缓冲器时，使能报文接收的滤波器编号将被装载到RXBNCTRL寄存器FILHIT位中。对于RXB1，RXB1CTRL寄存器包含FILHIT<2:0>位。

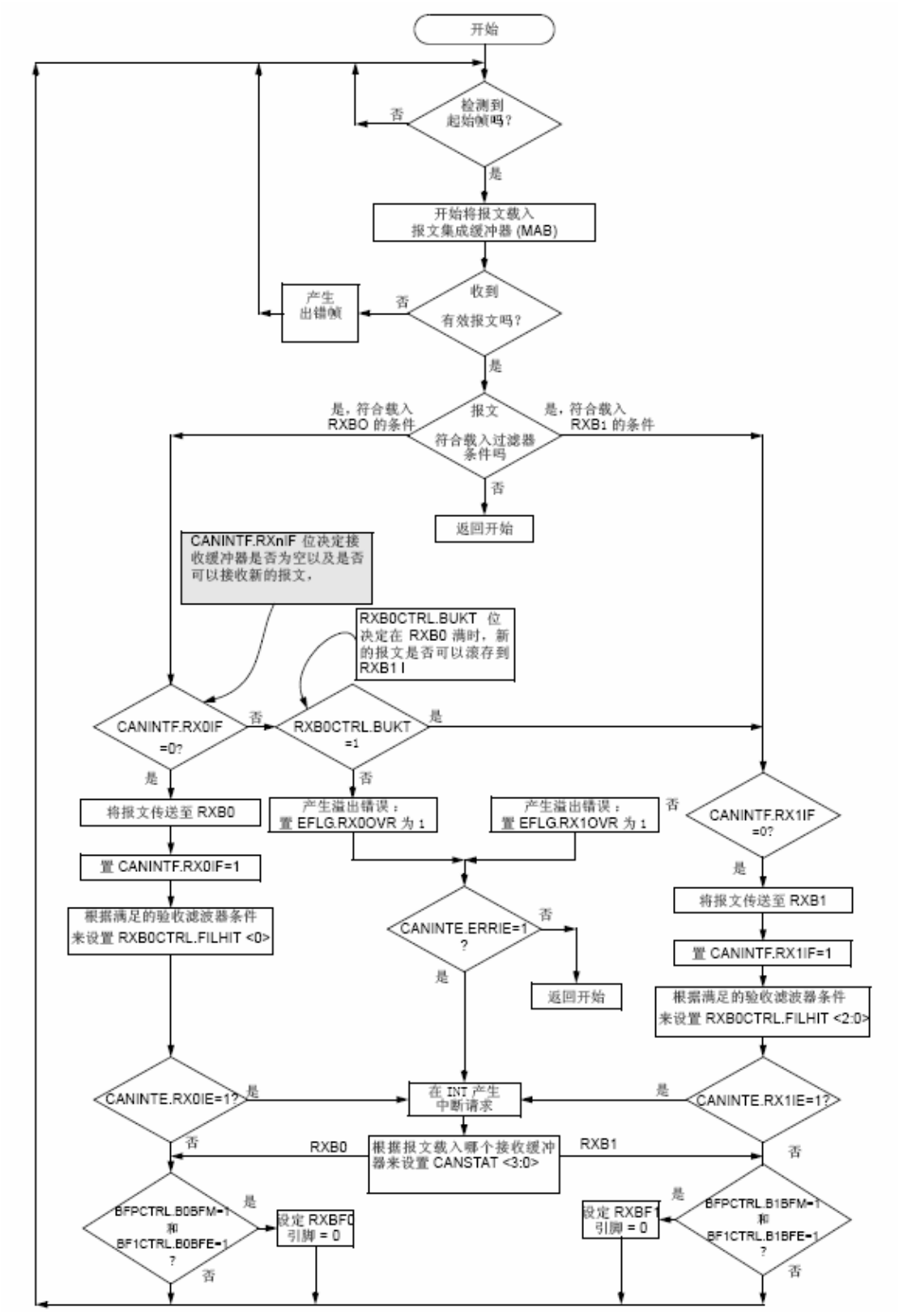


图9.2 报文接收流程图

如果接收报文符合一个以上滤波寄存器的接受条件，FILHIT位中的二进制代码将反映其中编号最小的寄存器。例如，如果滤波器RXF2和RXF4同时与接收报文匹配，FILHIT中将装载RXF2编码值。这实际上为编号较小的验收滤波寄存器赋予较高的优先级。接收报文将按照编号上升的原则依次与滤波寄存器进行匹配比较。只有MCP2510处于配置模式时，才能对屏蔽和滤波寄存器中的内容进行修

改。报文接收流程图见图9.2。

9.5 寄存器介绍

U-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	BUKT	BUKT1	FILHIT0
bit 7							bit 0

bit 7 未用：读作 '0'

bit 6-5 **RXM<1:0>**：接收缓冲器工作模式
11 =关闭屏蔽 / 滤波功能：接收所有报文
10 =只接收符合滤波器条件的带扩展标识符的有效报文
01 =只接收符合滤波器条件的带标准标识符的有效报文
00 =接收符合滤波器条件的所有带扩展标识符或标准标识符的有效报文

bit 4 未用：读作 '0'

bit 3 **RXRTR**: 是否接收到远程传送请求
1 = 接收到远程传送请求
0 = 没有接收到远程传送请求

bit 2 **BUKT**：滚存使能
1 = 如果 RXB0 满， RXB0 接收到的报文将被滚存至 RXB1
0 = 滚存禁止

bit 1 **BUKT1**：只读位， BUKT 位备份 （只在 MCP2510 器件内部使用).

bit 0 **FILHIT<0>**：滤波器指示 - 指明使能报文接收的验收滤波寄存器编号
1 = 验收滤波寄存器 1 (RXF1)
0 = 验收滤波寄存器 0 (RXF0)

图9.3 RXB0CTRL 接收缓冲器0 控制寄存器

U-0	R/W-0	R/W-0	U-0	R-0	R-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0
bit 7							bit 0

bit 7 未用：读作 '0'

bit 6-5 **RXM<1:0>**：接收缓冲器工作模式
11 =关闭屏蔽 / 滤波功能：接收任何报文
10 =只接收符合滤波器条件的带扩展标识符的有效报文
01 =只接收符合滤波器条件的带标准标识符的有效报文
00 =接收符合滤波器条件的所有带扩展标识符或标准标识符的有效报文

bit 4 未用：读作 '0'

bit 3 **RXRTR**：是否收到远程传递请求
1 = 接收到远程传递请求
0 = 未收到远程传递请求

bit 2-0 **FILHIT<2:0>**：滤波器指示 - 显示使能报文接收的过滤寄存器编号
101 = 验收滤波寄存器 5 (RXF5)
100 = 验收滤波寄存器 4 (RXF4)
011 = 验收滤波寄存器 3 (RXF3)
010 = 验收滤波寄存器 2 (RXF2)
001 = 验收滤波寄存器 1 (RXF1) (只有当 RXB0CTRL 中的 BUKT 位置 1 时)
000 = 验收滤波寄存器 0 (RXF0) (只有当 RXB0CTRL 中的 BUKT 位置 1 时)

图9.4 RXB1CTRL 接收缓冲器1 控制寄存器

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID<10:3>**: 标准标识符位 <10:3>
这些数据位装载接收报文标准标识符中最高 8 位。

图9.5 RXBNSIDH 接收缓冲器N 标准标识符高位

R-x	R-x	R-x	R-x	R-x	U-0	R-x	R-x
SID2	SID1	SID0	SRR	IDE	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID<2:0>**: 标准标识符位 <2:0>
这些数据位装载接收报文中标准标识符的最低 3 位。

bit 4 **SRR**: 远程发送请求位 （只有当 IDE 位 = '0' 时有效）
1 = 收到远程发送请求
0 = 收到标准数据帧

bit 3 **IDE**: 扩展标识符标志
该位表明收到报文是标准帧还是扩展帧
1 = 收到的报文为扩展帧
0 = 收到的报文为标准帧

bit 2 **未用**：读作 '0'

bit 1-0 **EID<17:16>**: 扩展标识符位 <17:16>
这些位装载接收报文中扩展标识符中的最高 2 位。

图9.6 RXBNSIDL 接收缓冲器N 标准标识符低位

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

bit 7-0 **EID<15:8>**: 扩展标识符位 <15:8>
这些数据位装载接收报文扩展标识符的第 8 到 15 位。

图9.7 RXBNEID8 接收缓冲器N 扩展标识符中间位

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

bit 7-0 **EID<7:0>**: 扩展标识符位 <7:0>
这些数据位装载接收报文扩展标识符中最低的 8 位。

图9.8 RXBNEID0 - 接收缓冲器N 扩展标识符低位

U-0	R-x	R-x	R-x	R-x	R-x	R-x	R-x
—	RTR	RB1	RB0	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

bit 7 未用：读作 '0'

bit 6 **RTR**：扩展帧远程发送请求位 (只有当 RXBnSIDL.IDE = 1 时有效)
1 = 收到扩展远程 （发送请求）帧
0 = 收到扩展数据帧

bit 5 **RB1:** 保留位 1

bit 4 **RB0:** 保留位 0

bit 3-0 **DLC<3:0>**：数据长度代码
表明接收到的数据字节个数

图9.9 RXBNDLC 接收缓冲器N 数据长度码

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
RBNDm7	RBNDm6	RBNDm5	RBNDm4	RBNDm3	RBNDm2	RBNDm1	RBNDm0
bit 7							bit 0

bit 7-0 **RBNDm7:RBNDm0**: 接收缓冲器 N 数据场字节 m
这 8 个字节包含接收报文中的数据信息

图9.10 RXBNDM 接收缓冲器N 数据字段字节M

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID<10:3>**: 标准标识符滤波控制位 <10:3>
这些数据位装载了用来对接收报文中标准标识码位 <10:3> 进行滤波判断的滤波寄存器位

图9.12 RXFNSIDH 验收滤波寄存器N 标准标识符的高位

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID<2:0>**：标准标识符滤波控制位 <2:0>
这些数据位装载用来对接收报文中的标准标识码位 <2:0> 进行滤波判断的滤波寄存器位

bit 4 未用：读作 '0'

bit 3 **EXIDE**: 扩展标识符使能
1 = 报文滤波仅应用于扩展帧
0 = 报文滤波仅应用于标准帧

bit 2 未用：读作 '0

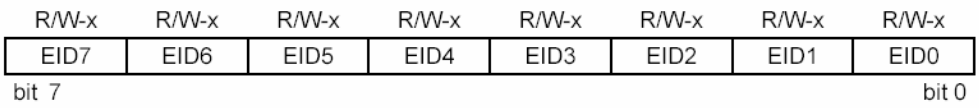
bit 1-0 **EID<17:16>**：扩展标识符滤波控制位 <17:16>
这些数据位装载用来对接收报文中的扩展标识码位 <17:16> 进行滤波判断的滤波寄存器位

图9.13 RXFNSIDL 验收滤波寄存器**N** 标准标识符的低位

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

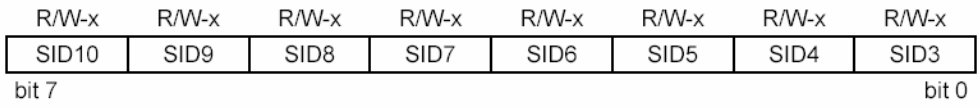
bit 7-0 **EID<15:8>**：扩展标识符过滤控制位 <15:8>
这些数据位装载 用来对接收报文中扩展标识码 位 <15:8> 进行过滤判断的过滤寄存器位

图9.14 RXFNEID8 验收滤波器N 扩展标识符的高位



bit 7-0 **EID<7:0>**：扩展标识符过滤控制位 <7:0>
这些数据位装载用来对接收报文中扩展标识码位 <7:0> 进行滤波判断的滤波寄存器位

图9.15 RXFNEID0 验收滤波寄存器N 扩展标识符的低位

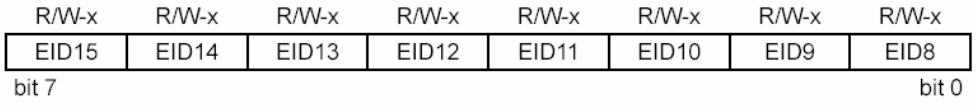


bit 7-0 **SID<10:3>**：标准标识符屏蔽控制位 <10:3>
这些数据位装载用来对接收报文中标准标识码位 <10:3> 进行屏蔽控制的屏蔽寄存器位

图9.16 RXMNSIDH 验收滤波屏蔽寄存器N 标准标识符的高位

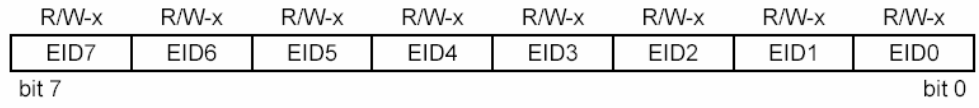
bit 7-5 **SID<2:0>**：标准标识符屏蔽控制位 <2:0>
这些数据位装载用来对接收报文中标准标识码位 <2:0> 进行屏蔽控制的屏蔽寄存器位。
bit 4-2 **未用**：读作 '0'
bit 1-0 **EID<17:16>**: 扩展标识符屏蔽控制位 <17:16>
这些数据位装载用来对接收报文中扩展标识码位 <17:16> 进行屏蔽控制的屏蔽寄存器位

图9.17 RXMNSIDL 验收滤波屏蔽寄存器N 标准标识符低位



bit 7-0 **EID<15:8>**：扩展标识码屏蔽控制位 <15:8>
这些数据位装载用来对接收报文中扩展标识码位 <15:8> 进行屏蔽控制的屏蔽寄存器位

图9.18 RXMNEID8 验收滤波屏蔽寄存器N 扩展标识码的高位



bit 7-0 **EID<7:0>**:：扩展标识符屏蔽控制位 <7:0>
这些数据位装载用来对接收报文中扩展标识符位 <7:0> 进行屏蔽控制的屏蔽寄存器位

图9.19 RXMNEID0 接受屏蔽寄存器N 扩展标识符的低位字节

10. 位定时

10.1 位定时配置寄存器

CAN总线接口的位定时由配置寄存器（CNF1，CNF2，CNF3）控制。只有当MCP2510处于配置模式时，才能对这些寄存器进行修改。

下面对这三个寄存器简单的进行介绍：

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
bit 7							bit 0

bit 7-6 **SJW<1:0>**：同步跳转宽度

11 = 长度 = 4xTQ
10 = 长度 = 3xTQ
01 = 长度 = 2xTQ
00 = 长度 = 1xTQ

bit 5-0 **BRP<5:0>**：波特率预分频器

TQ = 2 x (BRP + 1) / FOSC

图10.1 CNF1 配置寄存器1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0
bit 7							bit 0

bit 7 **BTLMODE**: 相位段 2 位时间长度

1 = 相位段 2 位时间长度由 CNF3 中的 PHSEG22:PHSEG20 确定
0 = 相位段 2 位时间长度取相位缓冲段 1 和 IPT (2TQ) 之间的较大值

bit 6 **SAM**: 采样点配置

1 = 在采样点对总线进行三次采样
0 = 在采样点对总线进行一次采样

bit 5-3 **PHSEG1<2:0>**：相位段 1 位时间长度

(PHSEG1 + 1) x TQ

bit 2-0 **PRSEG<2:0>**：传播段长度

(PRSEG + 1) x TQ

图10.2 CNF2 配置寄存器2

U-0	R/W-0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0
—	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20
bit 7							bit 0

bit 7 未用：读作 '0'

bit 6 **WAKFIL**：唤醒输入引脚滤波使能位

1 = 唤醒输入引脚滤波使能
0 = 唤醒输入引脚滤波禁止

bit 5-3 未用：读作 '0'

bit 2-0 **PHSEG2<2:0>**：相位段 2 长度

(PHSEG2 + 1) x TQ

注： 相位段 2 的最小有效设定为 2TQ

图 10.3 CNF3 配置寄存器 3

11. 错误检测

略。

12. 中断

该器件具有8个中断源。CANINTE 寄存器中包含了使能各个中断源的中断使能控制位。CANINTF寄存器中包含了各个中断源的中断标志位。当有中断请求发生，INT引脚将置为低电平，并维持低电平状态直至MCU清除中断标志。中断标志只有在引起相应中断请求条件消失后，才能被清除。建议在对CANINTF 寄存器中的中断标志位进行复位操作时，采用位修改命令而不要使用普通的

写操作。这是为了避免在写命令执行中无意间修改了标志位，从而导致中断请求信号的丢失。应注意，CANINTF中的中断标志位为可读写位，因此在相关CANINTE中断使能位置1的前提下，对上述任何一位进行置位均可使MCU产生中断请求。

12.1 发送中断

在发送中断使能（CANINTE.TXNIE = 1）的条件下，如果相关发送缓冲器空并处于新报文装载就绪状态时，器件会在INT引脚产生中断请求信号。CANINTF.TXNIF发送中断标志位将被置位以显示中断源。MCU通过将TXNIF位置0来清除中断。

12.2 接收中断

在接收中断使能（CANINTE.RXNIE = 1）的条件下，如果报文成功接收并被载入相关接收缓冲器时，器件会在INT引脚产生中断请求信号。在接收到EOF 字段后，该中断立即被激活。CANINTF.RXNIF接收中断标志位将被置位以显示中断源。MCU通过将RXNIF位复位来清除中断。

12.3 报文错误中断

如果报文发送和接收过程中出现错误，报文出错标志（CANINTF.MERRF）将被置1，此时若相应的CANINTE.MERRE中断使能位也被置1，器件将在INT引脚产生中断请求信号。该中断功能在与只听模式联用时被用来加快波特率的确定。

12.4 中断确认

中断与CANINTF寄存器中的一个或多个状态标志直接相关。只要其中一个标志位置位，所有中断就将保持等待发送状态。一旦器件设置了中断标志，在中断条件消除之前MCU将不能将其复位。

12.5 错误中断

在错误中断使能时（CANINTE.ERRIE = 1），如果发生溢出或发送/ 接收节点的状态发生改变，器件将在INT引脚产生中断请求。错误标志寄存器（EFLG）将会显示以下错误中断状况之一。

(1) 接收缓冲器溢出
(2) 接收节点警告
(3) 发送节点警告
(4) 接收节点错误消极模式
(5) 发送节点错误消极模式
(6) 总线关闭模式

12.6 中断寄存器

寄存器 7-1: **CANINTE - I 中断使能寄存器 (地址 : 2Bh)**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE
bit 7							bit 0

bit 7 **MERRE**：报文错误中断使能
1 = 报文接收或发送错误中断
0 = 中断禁止

bit 6 **WAKIE**：唤醒中断使能
1 = CAN 总线活动中断
0 = 中断禁止

bit 5 **ERRIE**：错误中断使能 （ EFLG 寄存器中包含了多种引起错误中断的状态标志位)
1 = EFLG 错误状态变化中断
0 = 中断禁止

bit 4 **TX2IE**：发送缓冲器 2 空中断使能
1 = TXB2 为空时中断
0 = 中断禁止

bit 3 **TX1IE**：发送缓冲器 1 空中断使能
1 = TXB1 为空时中断
0 = 中断禁止

bit 2 **TX0IE**：发送缓冲器 0 空中断使能
1 = TXB0 为空时中断
0 = 中断禁止

bit 1 **RX1IE**：接收缓冲器 1 满中断使能
1 = RXB1 装入报文时中断
0 = 中断禁止

bit 0 **RX0IE**：接收缓冲器 0 满中断使能
1 = RXB0 装入报文时中断
0 = 中断禁止

图12.1 中断使能寄存器

寄存器 7-2: **CANINTF - 中断标志寄存器 (地址 : 2Ch)**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF
bit 7							bit 0

bit 7 **MERRF**：报文出错中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 6 **WAKIF**：唤醒中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 5 **ERRIF**：出错中断标志 （ EFLG 寄存器中包含了多种引起出错中断的状态标志位
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 4 **TX2IF**：发送缓冲器 2 空中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 3 **TX1IF**：发送缓冲器 1 空中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 2 **TX0IF**：发送缓冲器 0 空中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 1 **RX1IF**：接收缓冲器 1 满中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

bit 0 **RX0IF**：接收缓冲器 0 满中断标志
1 = 有等待处理的中断 （此位应由 MCU 清除以使中断复位）
0 = 无等待处理的中断

图12.2 中断标志寄存器

13. 时钟振荡器

MCP2510设计使用晶体振荡器或陶瓷振荡器作为时钟振荡器，它们应连接在OSC1和OSC2引脚上。MCP2510的振荡器设计要求选用并联切割晶体振荡器。若使用串联晶振，其产生的时钟频率可能超出厂商规定值。图13.1显示了一个典型时钟振荡器电路。

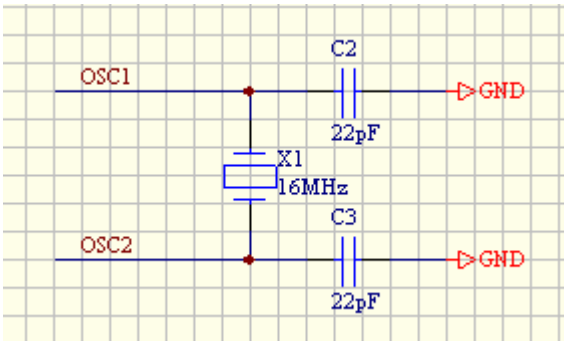


图 13.1 振荡电路图

14. 工作模式

MCP2510 具有 5 种工作模式，分别为：

(1) 配置模式
(2) 正常模式
(3) 睡眠模式

(4) 监听模式
(5) 环回模式

通过设定CANCTRL寄存器的REQOP位，可选择工作模式。改变工作模式时，新的工作模式需等到所有报文传输完毕之后才能生效。因此在运行另一种模式之前，用户在进行下一步操作时应先确认器件是否已进入该工作模式。通过读取CANSTAT寄存器的OPMODE位可以查验当前工作模式。

(1) 配置模式

正常运行之前，必须对MCP2510 进行初始化。只有在配置模式下，才能对器件进行初始化。在初始上电或复位时，器件自动进入配置模式。将CANTRL寄存器的REQOP 设置为‘100’ 也可以使器件进入配置模式。当进入配置模式时，所有错误计数器将被清零。只有在配置模式下，才能对下列的寄存器进行修改。

(a) CNF1, CNF2, CNF3
(b) TXRTSCTRL
(c) 接收滤波寄存器
(d) 接收屏蔽寄存器

只有当CANSTAT寄存器OPMODE读数为‘100’ 时，才能进行初始化操作，并对配置寄存器，接收过滤寄存器以及接收屏蔽寄存器进行写操作。配置完成之后，可以通过设定CANCTRL寄存器的REQOP位以使器件进入正常工作模式（或其他工作模式）。

(2) 正常模式

该模式为MCP2510标准工作模式。该模式下，器件主动监视总线上的所有报文，并产生确认位和错误帧等。只有在正常工作模式下，MCP2510才能在CAN 总线上进行报文的传输。

(3) 睡眠模式（略）
(4) 监听模式（略）
(5) 回环模式

该模式可使器件内部发送缓冲器和接收缓冲器之间进行报文自发自收，而无须通过CAN 总线。该模式可用于系统研发和测试。回环模式下应答位ACK无效，器件接收自己发送的报文就如同接收来自其它节点的报文。回环模式是一个静音模式，即器件不能通过总线发送包括错误标志或确认信号在内的任何报文。在该模式下，器件的TXCAN引脚处于隐性状态。可通过设定滤波器和屏蔽器以接收特定的报文。也可以将滤波屏蔽位全部置零来接收所有的报文。通过设定CANCTRL寄存器中的模式请求位可激活回环模式。

R/W-1	R/W-1	R/W-1	R/W-0	U-0	R/W-1	R/W-1	R/W-1
REQOP2	REQOP1	REQOP0	ABAT	—	CLKEN	CLKPRE1	CLKPRE0
bit 7							bit 0

bit 7-5 **REQOP<2:0>:** 工作模式设定
000 = 设定为正常工作模式
001 = 设定为休眠模式
010 = 设定为回环模式
011 = 设定为监听模式
100 = 设定为配置模式
REQOP 不应设定为其它任何值，任何其它设定值皆为无效。
注： 上电复位时，REQOP = b'111'

bit 4 **ABAT:** 报文发送中止设定
1 = 请求中止所有等待发送的发送缓冲器
0 = 终止报文发送中止请求

bit 3 **未用：**读作 '0'

bit 2 **CLKEN:** CLKOUT 引脚使能设定
1 = CLKOUT 引脚使能
0 = CLKOUT 引脚禁止（引脚处于高阻状态）

bit 1-0 **CLKPRE <1:0>:** CLKOUT 引脚预分频器设定
00 = FCLKOUT = 系统时钟频率 /1
01 = FCLKOUT = 系统时钟频率 /2
10 = FCLKOUT = 系统时钟频率 /4
11 = FCLKOUT = 系统时钟频率 /8

图14.1 CANCTRL CAN 控制寄存器

R-1	R-0	R-0	U-0	R-0	R-0	R-0	U-0
OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—
bit 7							bit 0

bit 7-5 **OPMOD<2:0>：**工作模式
000 = 器件处于正常工作模式
001 = 器件处于休眠模式
010 = 器件处于回环模式
011 = 器件处于监听模式
100 = 器件处于配置模式

bit 4 **未用：**读作 '0'

bit 3-1 **ICOD<2:0>：**中断标志码
000 = 无中断
001 = 出错中断
010 = 唤醒中断
011 = TXB0 中断
100 = TXB1 中断
101 = TXB2 中断
110 = RXB0 中断
111 = RXB1 中断

bit 0 **未用：**读作 '0'

图14.2 CANSTAT CAN 状态寄存器

15. 寄存器映射表

低地址位	高地址位							
	x000 xxxx	x001 xxxx	x010 xxxx	x0011 xxxx	x100 xxxx	x101 xxxx	x110 xxxx	x111 xxxx
0000	RXF0SIDH	RXF3SIDH	RXM0SIDH	TXB0CTRL	TXB1CTRL	TXB2CTRL	RXB0CTRL	RXB1CTRL
0001	RXF0SIDL	RXF3SIDL	RXM0SIDL	TXB0SIDH	TXB1SIDH	TXB2SIDH	RXB0SIDH	RXB1SIDH
0010	RXF0EID8	RXF3EID8	RXM0EID8	TXB0SIDL	TXB1SIDL	TXB2SIDL	RXB0SIDL	RXB1SIDL
0011	RXF0EID0	RXF3EID0	RXM0EID0	TXB0EID8	TXB1EID8	TXB2EID8	RXB0EID8	RXB1EID8
0100	RXF1SIDH	RXF4SIDH	RXM1SIDH	TXB0EID0	TXB1EID0	TXB2EID0	RXB0EID0	RXB1EID0
0101	RXF1SIDL	RXF4SIDL	RXM1SIDL	TXB0DLC	TXB1DLC	TXB2DLC	RXB0DLC	RXB1DLC
0110	RXF1EID8	RXF4EID8	RXM1EID8	TXB0D0	TXB1D0	TXB2D0	RXB0D0	RXB1D0
0111	RXF1EID0	RXF4EID0	RXM1EID0	TXB0D1	TXB1D1	TXB2D1	RXB0D1	RXB1D1
1000	RXF2SIDH	RXF5SIDH	CNF3	TXB0D2	TXB1D2	TXB2D2	RXB0D2	RXB1D2
1001	RXF2SIDL	RXF5SIDL	CNF2	TXB0D3	TXB1D3	TXB2D3	RXB0D3	RXB1D3
1010	RXF2EID8	RXF5EID8	CNF1	TXB0D4	TXB1D4	TXB2D4	RXB0D4	RXB1D4
1011	RXF2EID0	RXF5EID0	CANINTE	TXB0D5	TXB1D5	TXB2D5	RXB0D5	RXB1D5
1100	BFPCTRL	TEC	CANINTF	TXB0D6	TXB1D6	TXB2D6	RXB0D6	RXB1D6
1101	TXRTSCTRL	REC	EFLG	TXB0D7	TXB1D7	TXB2D7	RXB0D7	RXB1D7
1110	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT
1111	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL

图 15.1　寄存器映射表

从该表可以看出，第一列、第二列主要是验收滤波寄存器；第三列前 8 个寄存器是验收滤波屏蔽寄存器；第四、五、六列分别是发送缓冲区 0、1、2；第七、八列分别是接收缓冲区 0、1。该表可以很清晰的看出 MCP2510 的全部寄存器的分布情况。

16. MCP2510 的 SPI 接口

MCP2510设计可与许多微控制器的串行外设接口（SPI）直接相连，支持0,0 和1,1 运行模式。外部数据和命令通过SI引脚传送到器件中，而数据在SCK时钟信号的上升沿传送进去。MCP2510在SCK下降沿通过SO引脚发送。表列出了所有操作的指令字节。

指令名称	指令格式	功能介绍
复位	1100 0000	将内部寄存器复位为缺省状态，并将器件设定为配置模式
读	0000 0011	从以指定地址起始的寄存器读取数据
写	0000 0010	向以指定地址起始的寄存器写入数据
RTS （发送请求）	1000 0nnn	设置 TXBnCTRL.TXREQ 位以启动一个或多个发送缓冲器的报文发送 1000 0nnn TXB2 请求发送　TXB0 请求发送 TXB1 请求发送
状态读	1010 0000	读取 MCP2510 的状态（包括发送接收中断标志位和各请求发送位）
位修改	0000 0101	对指定寄存器进行位修改

表16.1　SPI指令集

(1) 读指令

在读操作开始时，CS引脚将被置为低电平。随后读指令和8位地址码（A7至A0）将被依次送入MCP2510。在接收到读指令和地址码之后，MCP2510指定地址寄存器中的数据将被移出通过SO引脚进行发送。每一数据字节移出后，器件内部的地址指针将自动加1以指向下一地址。因此可以对下一个连续地址寄存器进行读操作。通过该方法可以顺序读取任意个连续地址寄存器中的数据。通过拉高CS引脚电平可以结束读操作，见图16.1。

(2) 字节写指令

置CS引脚为低电平启动写操作。启动写指令后，地址码以及至少一个字节的数据被依次发送到MCP2510。只要CS保持低电平，就可以对连续地址寄存器进行顺序写操作。在SCK引线上的上升沿，数据字节将从D0位开始依次被写入。如果CS引脚在字节的8位数据尚未发送完之前跳变到高电平，该字节的写操作将被中止，而之前发送的字节已经写入。有关详细的字节写操作时序请参见图16.2。

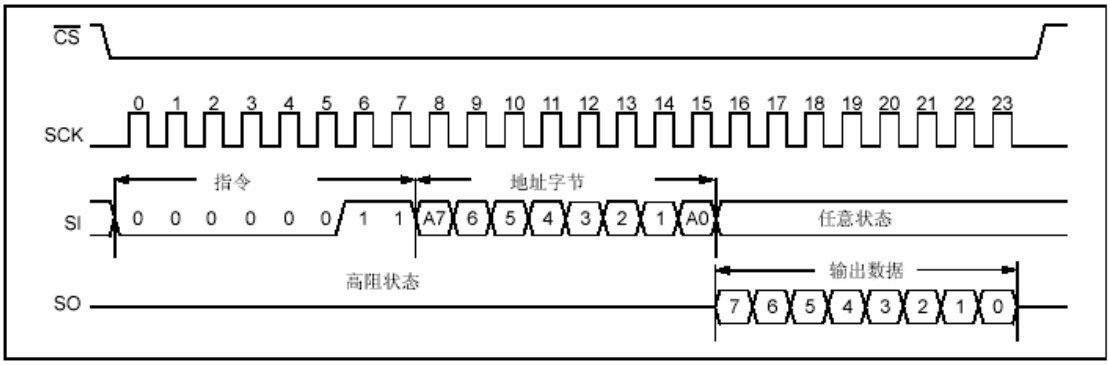


图16.1 读指令

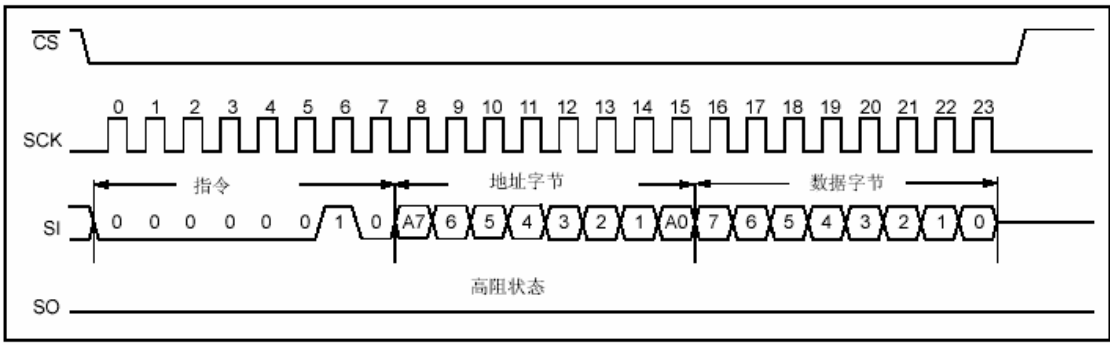


图16.2 字节写指令

(3) 请求发送指令

使用RTS命令可以启动一个或多个发送缓冲器的报文发送。置某器件的CS为低电平选择该器件，之后RTS命令字节会被发送给MCP2510。如图15.3 所示，该命令的最后3位显示了被使能发送的缓冲器编号。执行该命令后相应缓冲器的TxBnCTRL寄存器的TXREQ位被置1。用一条RTS命令即可对这三位中的一位或全部三位进行设定。如果发送的RTS命令中nnn = 000，该命令将被视为无效。

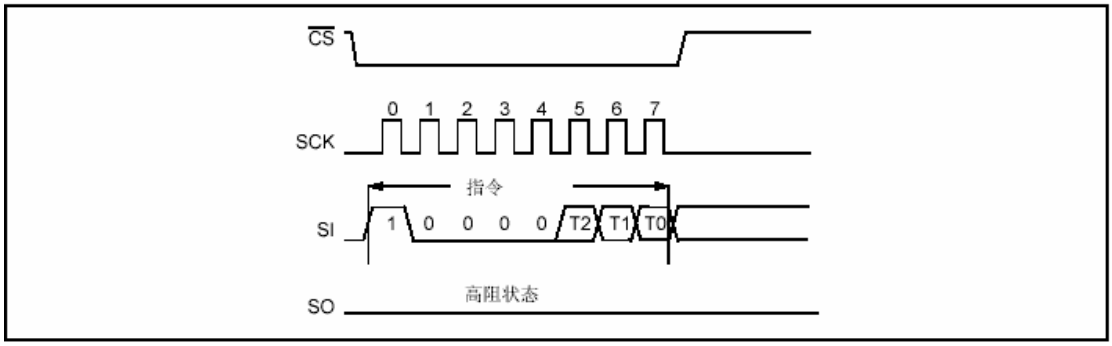


图16.3 请求发送指令

(4) 状态读指令

状态读指令允许单条指令对一些常用的报文接收和发送状态位进行访问。

置某器件的CS为低电平选择该器件，如图15.4所示，状态读命令字节将被发送给MCP2510。命令字节发出后， MCP2510将发回一个包含状态信息的8位数据。

在发送完最初8位数据之后，如果还有时钟信号发出，只要CS引脚保持低电平并且时钟信号是通过SCK引脚提供的，MCP2510将继续发送状态位。该命令中发回

的每个位的状态也可通过带相应寄存器地址的标准读命令读取。

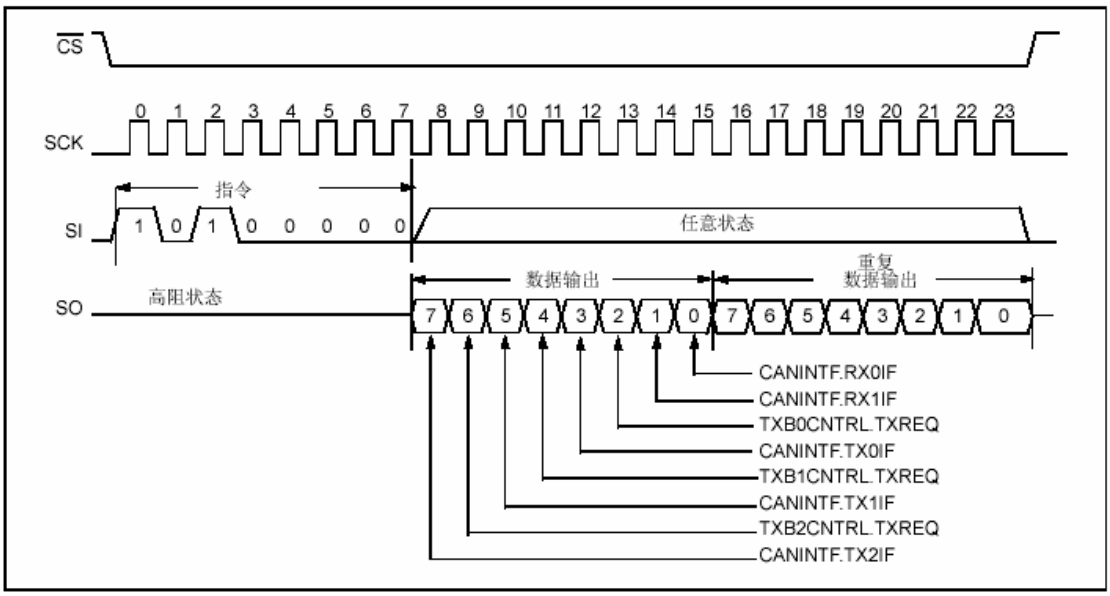


图16.4 状态读指令

(5) 位修改指令

位修改命令提供了一种对特定控制和状态寄存器中单独的位进行设定和清除的方法。该命令并非对所有寄存器都有效。允许进行位修改的寄存器（寄置某器件的CS为低电平选择该器件，之后位修改命令字节会被发送给MCP2510。命令字节发送后，寄存器地址，屏蔽字节以及数据字节被依次发出。屏蔽字节决定寄存器中的哪一位将被修改。屏蔽字节中的1表示允许对寄存器相应的位进行修改，0则禁止修改。数据字节确定寄存器位修改后的最终结果。如果屏蔽字节相应位设置为1，数据字节中的1表示将对寄存器对应位置1，而0则将对该位清零。

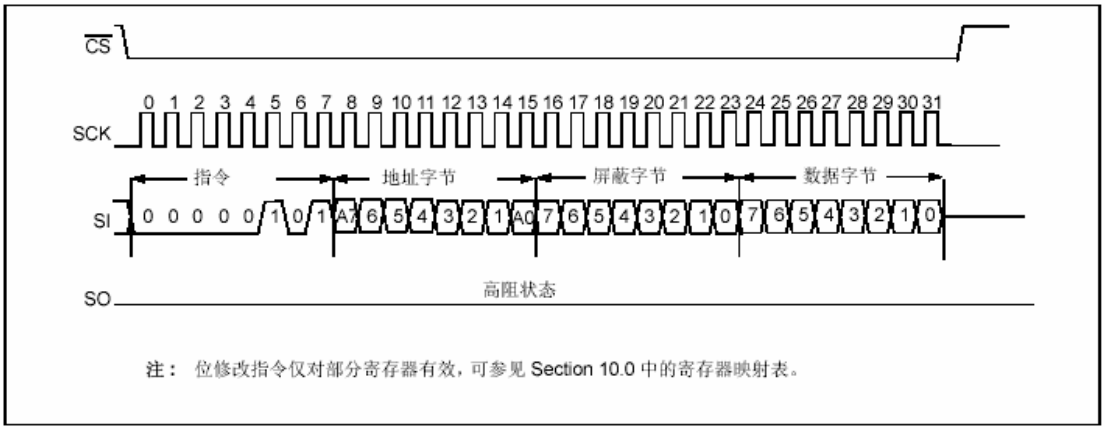


图16.5 位修改指令\

(6) 复位指令

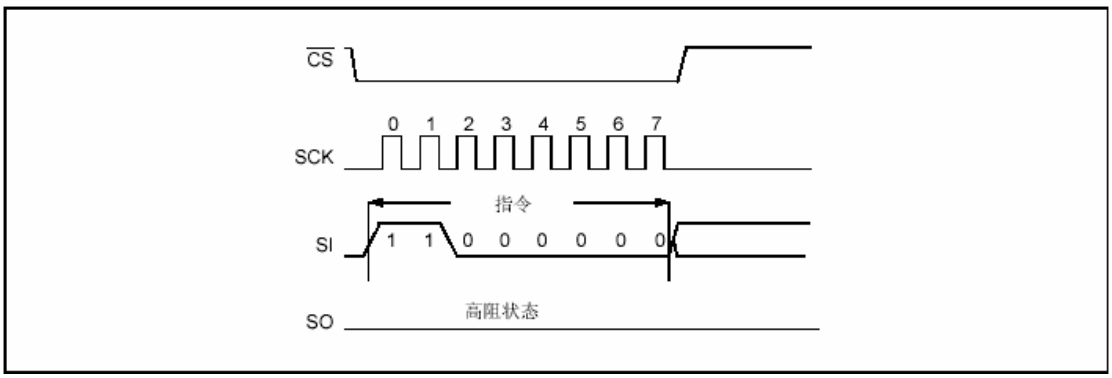


图 16.6 复位指令

17. 一些注意事项

(1) MCP2510 只支持 SPI 0,0 和 1,1 模式；
(2) 片选接在 GPG14 处，中断接在 GPG13 引脚，对应的是中断 21；
(3) 我们在下面的实验里，发送用发送寄存器 0，接收用接收寄存器 1；
(4) 下面我们将详细介绍三个实验，前一个实验是后一个实验的基础，也是我们进行调试 CAN 的一个最基本的过程。第一个实验是关于 SPI 驱动的，我们在这个实验中通过 SPI 驱动对 MCP2510 进行读写操作，以检测 SPI 驱动是否编写正确；第二个实验是用 MCP2510 的回环模式对编写的程序进行检测，以检测发送、接收程序是否正确；第三个实验是两个板子之间进行通讯。
(5) 这三个实验的文件有 mcp2510.c 和 mcp2510.h，在该头文件里，定义了所有 MCP2510 的寄存器的地址，下面我们将只介绍 mcp2510.c 文件。

18. 程序解读（实验一）

我们介绍该实验的主要函数：

(1) address_map 函数，把寄存器的物理地址映射到内核空间地址；
(2) spi_poll_done 函数，判断 SPI 传输是否结束；
(3) spi_tx_data 函数，进行 SPI 数据传输；
(4) Write_2510 函数，通过 SPI 对 CAN 的一个寄存器进行写操作；
(5) Read_2510 函数，通过 SPI 对 CAN 的一个寄存器进行读操作；
(6) Reset_2510 函数，通过 SPI 对 CAN 进行复位操作；
(7) Modify_2510 函数，通过 SPI 对 CAN 的一个寄存器进行位修改操作；
(8) Init_MCP2510 函数，驱动初始化程序；
(9) Exit_MCP2510 函数，驱动结束程序；
(10) Init_SPI 函数，初始化 SPI；
(11) Test 函数，实验一要实现的程序；

下面我们分别对上面的函数进行介绍：

(1) address_map 函数：

```
// SPI 寄存器
r_SPCON0 = ioremap(0x59000000，4)；
r_SPSTA0 = ioremap(0x59000004，4)；
r_SPPIN0 = ioremap(0x59000008，4)；
r_SPPRE0 =  ioremap(0x5900000C，4)；
r_SPTDAT0 = ioremap(0x59000010，4)；
```

```
r_SPRDAT0 = ioremap(0x59000014，4)；

//I/O 寄存器充当 SPI 引脚
r_GPECON = ioremap(0x56000040，4)；
r_GPEUP  = ioremap(0x56000048，4)；

//片选寄存器
r_GPGCON = ioremap(0x56000060，4)；
r_GPGUP = ioremap(0x56000068，4)；
r_GPGDAT = ioremap(0x56000064，4)；
```

(2) spi_poll_done 函数：

```
int   nCount = 0；
while(!(rSPSTA0 & 0x01) )   //判断 SPI 发送寄存器是否准备好
{
        nCount++；
        if(nCount>=5000)
         {
                printk("SPI state poll failed\n")；
                break；
        }
}//endwhile
```

(3) spi_tx_data 函数：

```
spi_poll_done()；
rSPTDAT0 = data；     //  通过 SPI，发送数据
spi_poll_done()；
```

(4) Read_2510 函数：

```
unsigned char buffer；
rGPGDAT &=(~0x4000)；          //选中该芯片
udelay(100000)；                //进行延迟，以确保确实选中，这是必须的

spi_tx_data(CMD_READ)；        //参见 16 节图 1
spi_tx_data(R_ADD)；               //所读寄存器的地址
spi_tx_data(0xff)；                //发送 0xff 读该寄存器数据
buffer = rSPRDAT0；

rGPGDAT |=0x4000；              //不选中该芯片
return buffer；
```

(5) Init_SPI 函数：

```
int   i；
rSPPRE0 = 0xff；    //设置 SPI 波特率
```

```
    rSPCON0 = 0x18;    //设置为查询方式
    for(i = 0 ;  i < 10 ;  i++)
    {
     rSPTDAT0 = 0xff;
     }//初始化设备

     rGPECON |=    0x0a800000;       //设置 SPI 的三个引脚
     rGPECON &= (~0x05400000);
    rGPEUP |= 0x3800;

    rGPGCON |=    0x10000000;//设置 GPG14 引脚为输出
    rGPGCON &= (~0x20000000);
    rGPGUP &= (~0x4000);
    rGPGDAT |=0x4000;            //不选中该芯片
```

(6) Init_MCP2510 函数：

```
//设置中断
//Set Interrupt
set_external_irq(IRQ_EINT21, EXT_FALLING_EDGE, GPIO_PULLUP_DIS);

gpfup = ioremap(0x56000068,4);
(*(volatile unsigned long *)gpfup) = 0;

disable_irq(IRQ_EINT21);
enable_irq(IRQ_EINT21);

result = request_irq(IRQ_EINT21,&can_interrupt,SA_INTERRUPT,"can",NULL);
if (result)
{
    printk("Can't get assigned irq %d,result=%d\n",IRQ_EINT21,result);
    return result;
}
//注册设备
mcp2510_devfs_dir = devfs_mk_dir(NULL,"CAN",NULL);
devfs_register(mcp2510_devfs_dir,DEVICE_NAME,DEVFS_FL_AUTO_DEVNUM,0,0,S_IFCHR|S
_IRUGO|S_IWUGO,&mcp2510_fops,NULL);

printk ("MCP2510 driver installed OK\n");

printk("---------->1\n");
printk("Initialize SPI\n");
address_map();  //寄存器地址映射
Init_SPI();        //初始化 SPI
```

```
spi_tx_data(0xff)；//初始化该芯片
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
spi_tx_data(0xff)；
#ifdef DEBUG_SPI1     //因为这是实验一，所以写 DEBUG_SPI1，不进行编译
printk("-------------->2\n")；
printk("Initialize MCP2510\n")；
      Init_MCP2510()；
#endif

#ifdef DEBUG_SPI      //在前面定义了 DEBUG_SPI，所以下面的代码有效
printk("-------------->2\n")；
printk("Test Funciton 1\n")；
      Test()；
#endif
```

(7) mcp2510_exit 函数：

```
//Disable Interrupt
disable_irq(IRQ_EINT21)；
free_irq(IRQ_EINT21, NULL)；
devfs_unregister(mcp2510_devfs_dir)；
printk ("MCP2510 driver uninstalled OK\n")；
```

(8) Test 函数：

```
unsigned char buffer[5]；

//Reset MCP2510
Reset_2510()；

//设置系统为配置模式
Modify_2510(CANCTRL, 0xe0, 0x80)；
Modify_2510(CANCTRL, 0x07, 0x00)；

//设置 CNF 寄存器
Write_2510(CNF1,0x07)；
Write_2510(CNF2,0x10)；
```

```
Write_2510(CNF3,0x02);

Write_2510(RXB0CTRL,0x60);

//读寄存器
buffer[0] = Read_2510(CANSTAT);
buffer[1] = Read_2510(CNF1);
buffer[2] = Read_2510(CNF2);
buffer[3] = Read_2510(CNF3);
buffer[4] = Read_2510(RXB0CTRL);

printk("The CANSTAT register is    %x\n",buffer[0]);
printk("The CNF1 register is    %x\n",buffer[1]);
printk("The CNF2 register is    %x\n",buffer[2]);
printk("The CNF3 register is    %x\n",buffer[3]);
printk("The RXB0CTRL register is    %x\n",buffer[4]);
```

19. 程序解读（实验二）

上面已经介绍的函数我们这里将省略，这里只介绍该实验中用到的并且和上面的实验不同的函数。

(1) Test_can_bus 函数，测试在回环模式下的收发；
(2) send_test_frame 函数，发送样本数据，调用 can_data_send 函数；
(3) can_data_send 函数，发送数据到总线；
(4) can_data_receive 函数，接收总线上的数据；
(5) can_interrupt 函数，中断处理函数；
(6) Init_MCP2510 函数，初始化 MCP2510；

下面对这些函数分别进行说明：

(1) Test_can_bus 函数：

```
unsigned char buffer;
int i;
//设置进入回环模式
Modify_2510(CANCTRL,0xe0,0x40);

//Read the Mode
buffer = Read_2510(CANSTAT);
printk("The CANSTAT register is %x\n",buffer);

//发送样本数据
printk("Transmit the sample data\n");
send_test_frame();

//等待进入中断，读取接收到的数据
while(1)
{
```

```
if(flag == 0xff)    //判断是否接收到数据
{
        for(i = 0；i < 8；i++)
            printk("Rxdata[%d] is %c\n",i,RXdata[i])；
        break；
}
}//endwhile
```

(2) send_test_frame 函数：

```
TXdata[0] =  0xa0；  //这个是发送数据的标志符
TXdata[1] = 0x00；
TXdata[2] =   0x08；  //发送数据的长度

TXdata[3] =  'A' ；
TXdata[4] =  'B' ；
TXdata[5] =  'C' ；
TXdata[6] =  'D' ；
TXdata[7] =  'E' ；
TXdata[8] =  'F' ；
TXdata[9] =  'G' ；
TXdata[10]=  'H' ；

can_data_send()；  //发送数据到总线
```

(3) can_data_send 函数：

```
unsigned char length，i；
length = TXdata[2]；
Write_2510(TXB0SIDH,TXdata[0])；
Write_2510(TXB0SIDL,TXdata[1])；
Write_2510(TXB0DLC, TXdata[2])；

rGPGDAT &=(~0x4000)；  //选中该芯片
udelay(100000)；

spi_tx_data(CMD_WRITE)；
spi_tx_data(TXB0D0)；
for(i = 0；  i<length；  i++)
{
    spi_tx_data(TXdata[i+3])；      //发送数据到 SPI 寄存器
}

rGPGDAT |=0x4000；              //Unselect the chip

SEND_TXB0()；                   //是 CAN 发送寄存器的数据发送到总线
```

```
printk("Transmit is over!\n");
```

(4) can_interrupt 函数：

```
unsigned char buffer[2];

buffer[0] = Read_2510(CANSTAT);

/*clear interrupt register for EINT21*/
SRCPND &= (~0x00000020);
INTPND = INTPND;
EINTPEND &= (~0x00200000);

#ifdef DEBUG_SPI
    printk("Entered an interrupt! \n");
    printk("CANSTAT register is %x\n",buffer[0]);
   buffer[1] = Read_2510(RXB1CTRL);
   //观察该接收操作的验收滤波器的编号
    printk("CANSTAT register is %x\n",buffer[1]);
#endif

if((buffer&0x0e) == 0x0e)
{
    printk("Recevie the sample data!\n");
    can_data_receive();
flag = 0xff;    //表示收到数据
}
```

(5) Init_MCP2510 函数：

```
unsigned char buffer;
//Reset MCP2510
Reset_2510();

//设置为配置模式
Modify_2510(CANCTRL, 0xe0, 0x80);
Modify_2510(CANCTRL, 0x07, 0x00);

//等待进入配置模式
while(1)
{
buffer = Read_2510(CANSTAT);
printk("CANSTAT register is %x\n,buffer&0xe0);
if((buffer&0xe0) == 0x80)
{
```

```
        printk("In the Configure Mode\n");
        break;
}
}

//Set physical layer configuration
Write_2510(CNF1,0x07);
Write_2510(CNF2,0x90);
Write_2510(CNF3,0x02);
/Set Interput , receive register 1 empty interrupt enable
Write_2510(CANINTE, 0x02);

//Configure Receive buffer 0 Mask and Filters
//Receive buffer 0 will not be used
Write_2510(RXM0SIDH,0xff);
Write_2510(RXM0SIDL,0xff);
Write_2510(RXF0SIDH,0xff);
Write_2510(RXF0SIDL,0xff);
Write_2510(RXF1SIDH,0xff);
Write_2510(RXF1SIDL,0xff);

//Configure Receive buffer 1 Mask and Filters
//这里是该实验要改的地方
#if 0
Write_2510(RXM1SIDH,0xff);       //RXB1 mathces all filters for Standad messages
Write_2510(RXM1SIDL,0xe0);
#endif

Write_2510(RXM1SIDH,0x00);   //接收全部数据
Write_2510(RXM1SIDL,0x00);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//------------------Receives 1 message-----------------------
Write_2510(RXF2SIDH,0xa0);
//Initialize Filter 2 (will receive only b'1010 0000 000' message)
Write_2510(RXF2SIDL,0x00);
//Make sure EXIDE bit (bit 3) is set correctly in filter also

//------------------Receives 2 message-----------------------
Write_2510(RXF3SIDH,0xa1);
Write_2510(RXF3SIDL,0x00);

//------------------Receives 3 message-----------------------
Write_2510(RXF4SIDH,0xa2);
```

```
Write_2510(RXF4SIDL,0x00);

//------------------Receives 4 message-----------------------
Write_2510(RXF5SIDH,0xa3);
Write_2510(RXF5SIDL,0x00);

//Set RXB1CTRL Register，只接收标准报文
Write_2510(RXB1CTRL, 0x20);

//Set to Normal Mode
Modify_2510(CANCTRL, 0xe0, 0x00);

//Wait into Normal Mode
while(1)
{
    buffer = Read_2510(CANSTAT);
    printk("CANSTAT register is %x\n",buffer&0xe0);
    if((buffer&0xe0) == 0x0)
    {
            printk("In the Normal Mode\n");
            break;
    }
}

printk("Init MCP2510 is over!\n");
flag == 0x88;
```

20. 程序解读（实验三）

驱动程序：

A 板的 Init_MCP2510 函数：

```
Write_2510(RXM1SIDH,0xff);      //RXB1 mathces all filters for Standad messages
Write_2510(RXM1SIDL,0xe0);

//------------------Receives 1 message-----------------------
Write_2510(RXF2SIDH,0xa0);
//Initialize Filter 2 (will receive only b'1010 0000 000' message)
Write_2510(RXF2SIDL,0x00);
//Make sure EXIDE bit (bit 3) is set correctly in filter also

//------------------Receives 2 message-----------------------
Write_2510(RXF3SIDH,0xa1);
Write_2510(RXF3SIDL,0x00);

//------------------Receives 3 message-----------------------
```

```
Write_2510(RXF4SIDH,0xa2);
Write_2510(RXF4SIDL,0x00);

//-----------------Receives 4 message----------------------
Write_2510(RXF5SIDH,0xa3);
Write_2510(RXF5SIDL,0x00);
```

A 板的 mcp2510_wr 函数：

```
dbuf = kmalloc(count*sizeof(unsigned char) , GFP_KERNEL);

copy_from_user(dbuf,buf,count);

TXdata[0] = 0xb0;
TXdata[1] = 0x00;
TXdata[2] = count;

for(i = 0;  i < count; i++)
{
    TXdata[i+3] = dbuf[i];
}
#ifdef DEBUG_SPI
for(i = 0; i < 11; i++)
{
        printk("Txdata[%d] is %x\n",i,TXdata[i]);
}
#endif
//Transmit data to Bus
can_data_send();
kfree(dbuf);
```

下面我们来看一下 send 应用程序：

```
int i = 0;
unsigned char Send[8] =
    {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
int fd;
fd = open("/dev/CAN/MCP2510",O_RDWR);
printf("Send data to Can bus\n");
write(fd,Send,8);
for(i = 0 ; i < 8 ; i++)
    printf("Send num[%d] is %x\n",i,Send[i]);
close(fd);
```

下面我们来看一下 receive 应用程序：

```
int i = 0;
unsigned char Receive[8];
```

```
int fd；
fd = open("/dev/CAN/MCP2510",O_RDWR)；
printf("Receive data to Can bus\n")；
read(fd,Receive,8)；
for(i = 0； i < 8； i++)
    printf("Receive num[%d] is %x\n",i,Receive[i])；
close(fd)；
return；
```

三．实验内容和步骤

学习 CAN 总线通讯原理，了解 CAN 总线的结构，阅读本实验原理和说明（或者阅读 CAN 控制器 MCP2510 的芯片文档），掌握 MCP2510 的相关寄存器的功能和使用方法。

1. SPI 驱动程序的编写，并通过读写相应的 MCP2510 的寄存器，检验该驱动编写是否正确。

实验步骤：

(1) 参照 Read_2510 函数和第 16 节的 MCP2510 的 SPI 接口的介绍，编写 Write_2510、Modify_2510、Reset_2510 函数；特别注意的是在 Reset_2510 函数中，发送 RESET 命令后，拉高片选，然后要等待 128 个晶振时间，才能进行读写操作；

(2) 参考前面相关各节的介绍，读懂实验解读一的代码分析；

(3) 编译并加载该驱动程序，并看相应的输出结果，正确的输出结果如下：

```
MCP2510 driver installed OK
------------------------------>1；
    Initialize SPI.
    ------------------------------>2；
  Test Funciton 1
The CANSTAT register is  80
The CNF1 register is  7
The CNF2 register is  10
The CNF3 register is  2
The RXB0CTRL register is  60
```

2. 编写 CAN 的收发程序，并设置 CAN 为回环模式对编写的程序进行检测，以检测发送、接收程序是否正确，CAN 是否工作正常。

实验步骤：

(1) 编写 can_data_receive 函数；

(2) 读懂上面的程序，了解 CAN 的工作机制：发送数据后，如果发送的标志符和验收滤波器相同，就进入中断，在中断里判断是否是接收缓冲器有数据，如果是，调用 can_data_receive 函数，接收数据；

(3) 在 Init_MCP2510 里，如果设置 RXM1SIDH 为 00，则无论 TXdata[0]是什么都可以进入中断，接收数据；如果 RXM1SIDH 为 FF，则 TXdata[0]必须和滤波寄存器 2 到 5 里的一个相同才可以接收；分别把 Txdata[0]设置为 0xa0，0xa1，0xa4，进行相应的实验，并把 RXB1CTRL 打印出来，看实验结果有什么区别。

3. 编程实现 RUIJIARM2410 之间的 CAN 总线通讯：两个 RUIJIARM2410 通过 CAN 总线相连接。分别在两个板子上加载驱动，然后板子 A 上运行 send 程序，监视另一个板子 B 的串行口，如果板子 B 接收到的数据，将显示进入中断，这时，在板子 B 上运行 receive 程序，读接收到的数据。注意事项就是我们在 A 板和 B 板加载的驱动略有不同，两个不同点分别在 Init_MCP2510 函数和驱动的写函数里：Init_MCP2510 函数设置只能接收特定的报文，在驱动里写函数里设定发送到的标志符；可以假设 A 板只能接收 a0，a1，a2，a3 开头的数据，但发送的标志符为 b0；B 板只能接收 b0，b1，b2，b3 开头的数据，但发送的标志符为 a0。

实验步骤：

(1) 编写驱动程序的读写函数；
(2) 读懂应用程序 send 和 receive；
(3) 分别在两个板子上加载驱动，看相应的输出是否正确；
(4) 在一个板子上运行 send 应用程序，发送数据到总线；
(5) 观察到另一个板子的反映，如果显示进入中断，则自动打印出收到的数据来，当然也可以运行 receive 应用程序来读取，观察相应的结果；

四．思考题

1. 在本实验中，SPI 驱动用的是 polling 模式，如果用 DMA 模式，驱动程序将怎么写？
2. 实现多个板子上的 CAN 总线连接，并进行相应的收发实验。

实验十六：以太网驱动实验

一．实验目的

通过本实验，使学生了解 MAC 地址的作用，了解 MAC 地址的保存方式，了解网卡芯片 DM9000 的功能，了解嵌入式系统中以太网的驱动原理。

二．实验原理

1．IP 地址和 MAC 地址：

1.1 IP 地址

对于 IP 地址，相信大家都很熟悉，即指使用 TCP/IP 协议指定给主机的 32 位地址。IP 地址由用点分隔开的 4 个 8 八位组构成，如 192.168.0.1 就是一个 IP 地址，这种写法叫点分十进制格式。IP 地址由网络地址和主机地址两部分组成，分配给这两部分的位数随地址类（A 类、B 类、C 类等）的不同而不同。网络地址用于路由选择，而主机地址用于在网络或子网内部寻找一个单独的主机。一个 IP 地址使得将来自源地址的数据通过路由而传送到目的地址变为可能。

1.2 MAC 地址

对于 MAC 地址，由于我们不直接和它接触，所以大家不一定很熟悉。在 OSI（Open System Interconnection，开放系统互连）7 层网络协议（物理层，数据链路层，网络层，传输层，会话层，表示层，应用层）参考模型中，第二层为数据链路层（Data Link）。它包含两个子层，上一层是逻辑链路控制（LLC：Logical Link Control），下一层即是我们前面所提到的 MAC（Media Access Control）层，即介质访问控制层。所谓介质（Media），是指传输信号所通过的多种物理环境。常用网络介质包括电缆（如：双绞线，同轴电缆，光纤），还有微波、激光、红外线等，有时也称介质为物理介质。MAC 地址也叫物理地址、硬件地址或链路地址，由网络设备制造商生产时写在硬件内部。这个地址与网络无关，也即无论将带有这个地址的硬件（如网卡、集线器、路由器等）接入到网络的何处，它都有相同的 MAC 地址，MAC 地址一般不可改变，不能由用户自己设定。

1.3 MAC 地址的长度、表示方法、分配方法及其唯一性

MAC 地址的长度为 48 位（6 个字节），通常表示为 12 个 16 进制数，每 2 个 16 进制数之间用冒号隔开，如：08:00:20:0A:8C:6D 就是一个 MAC 地址，其中前 6 位 16 进制数 08:00:20 代表网络硬件制造商的编号，它由 IEEE（Istitute of Electrical and Electronics Engineers，电气与电子工程师协会）分配，而后 3 位 16 进制数 0A:8C:6D 代表该制造商所制造的某个网络产品（如网卡）的系列号。每个网络制造商必须确保它所制造的每个以太网设备都具有相同的前三字节以及不同的后三个字节。这样就可保证世界上每个以太网设备都具有唯一的 MAC 地址。

1.4 IP 地址与 MAC 地址在互连网中的作用

既然每个以太网设备在出厂时都有一个唯一的 MAC 地址了，那为什么还需要为每台主机再分配一个 IP 地址呢？或者说为什么每台主机都分配唯一的 IP 地址了，为什么还要在网络设备（如网卡，集线器，路由器等）生产时内嵌一个唯一的 MAC 地址呢？主要原因有以下几点：

(1) IP 地址的分配是根据网络的拓朴结构，而不是根据谁制造了网络设置。若将高效的路由选择方案建立在设备制造商的基础上而不是网络所处的拓朴位置基础上，这种方案是不可行的。

(2) 当存在一个附加层的地址寻址时，设备更易于移动和维修。例如，如果一个以太网卡坏了，可以被更换，而无须取得一个新的 IP 地址。如果一个 IP 主机从一个网络移到另一个网络，可以给它一个新的 IP 地址，而无须换一个新的网卡。

(3) 无论是局域网，还是广域网中的计算机之间的通信，最终都表现为将数据包从某种形式的链路上的初始节点出发，从一个节点传递到另一个节点，最终传送到目的节点。数据包在这些节点之间的移动都是由 ARP（Address Resolution Protocol：地址解析协议）负责将 IP 地址映射到 MAC 地址上来完成的。

1.5 嵌入式网络设备中 MAC 及 IP 地址的特点

在嵌入式系统中，操作系统和所有的应用软件都被固化到 Flash 等存储设备中。在嵌入式系统中很少使用外存。嵌入式系统的启动往往也是“自动”的，即从上电到处于工作状态，不用人的介入。这是嵌入式设备应用的要求和特点。嵌入式网络设备的启动，很自然会遇到 MAC 及 IP 地址的设置问题。

嵌入式网络设备中的 MAC 及 IP 地址的设置有它的特点：

(1) 关心和接触嵌入式网络设备 MAC 地址的人比关心和接触通用计算机 MAC 地址的人多得多。因为设计、研究和生产嵌入式网络设备的厂家比网卡的厂家多得多。

(2) 在嵌入式设备中往往没有硬盘，它的操作系统和应用软件通常是打包放在 Flash 等存储设备中。系统启动时，把 Flash 中的代码释放到内存中，再在内存中运行。比如嵌入式操作系统 linux-2.4.*-rmk9，在用于 S3C2410 这样的带以太网接口的嵌入式设备时，把内核和应用程序代码压成一个映像文件包，在包中有网络部分 MAC 及 IP 地址。但这些 MAC 及 IP 地址的值是在编译映像文件时设定的，而且在编译后的映像文件中的值是不能直观地看到的，而且在编译后的映像文件中的值是不能直观地看到的，它是压缩了的二进制数据，不方便地映像文件中直接更改 MAC 及 IP 地址的值。

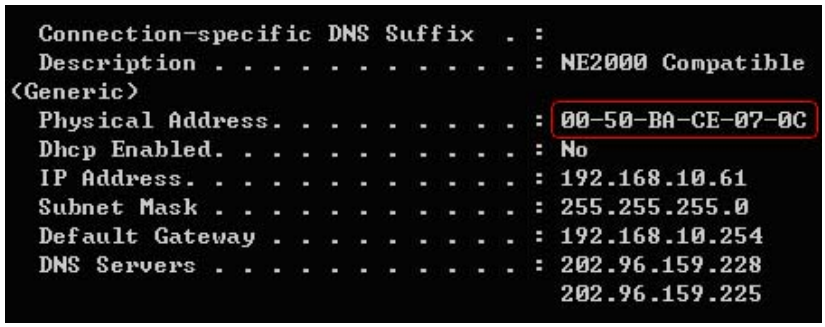
(3) 对于使用同一映像文件的嵌入式网络设备，如果不做进一步的处理，其 MAC 及 IP 地址是相同的。这显然不能满足应用，因为不同的设备应该有不同的 MAC 及 IP 地址。而编译生成映像文件往往要用十几甚至几十分钟。对于生产厂家，不可能为每台设备编译一个特定的映像文件。

针对以上问题，我们在嵌入式系统上运行 linux 时，使用了一些特殊的方法来解决它。嵌入式网络设备系统的 MAC 及 IP 地址设置的基本思想是：把 MAC 及 IP 地址存放在 Flash 的未用扇区（一般在高扇区），嵌入式操作系统启动后，自动运行一个程序去读取 MAC 及 IP 地址并设置它。在嵌入式系统 S3C2410 中，我们将 MAC 地址保存在 EEPROM 中的固定位置处，当以太网驱动加载时，从 EEPROM 中获取 MAC 地址并赋给以太网，而通过 ifconfig 命令在脚本中设置 IP 地址。

1.6 本机 MAC 地址的获取

在 Windows 98/Me 中，依次单击“开始”→“运行” →输入“winipcfg”→回车。即可看到 MAC 地址。

在 Windows 2000/XP 中，依次单击“开始”→“运行”→输入“CMD”→回车→输入“ipconfig /all”→回车。即可看到 MAC 地址。



在 linux 系统中，运行 ifconfig eth0，即可看到 MAC 地址。

eth0 Link encap:Ethernet **HWaddr 00:E0:4C:81:87:D6**
inet addr:192.168.2.37 Bcast:192.168.2.255 Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:29769 errors:0 dropped:0 overruns:0 frame:0
TX packets:9871 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:100
RX bytes:7858345 (7.4 Mb) TX bytes:1764628 (1.6 Mb)
Interrupt:10 Base address:0xa000

下面我们来通过一个例子看看 IP 地址和 MAC 地址是怎样结合来传送数据包的：

假设网络上要将一个数据包（名为 PAC）由北京的一台主机（名称为 A，IP 地址为 IP_A，MAC 地址为 MAC_A）发送到华盛顿的一台主机（名称为 B，IP 地址为 IP_B，MAC 地址为 MAC_B）。这两台主机之间不可能是直接连接起来的，因而数据包在传递时必然要经过许多中间节点（如路由器，服务器等等），我们假定在传输过程中要经过 C1、C2、C3（其 MAC 地址分别为 M1，M2，M3）三个节点。A 在将 PAC 发出之前，先发送一个 ARP 请求，找到其要到达 IP_B 所必须经历的第一个中间节点 C1 的 MAC 地址 M1，然后在其数据包中封装（Encapsulation）这些地址：IP_A、IP_B，MAC_A 和 M1。当 PAC 传到 C1 后，再由 ARP 根据其目的 IP 地址 IP_B，找到其要经历的第二个中间节点 C2 的 MAC 地址 M2，然后再将带有 M2 的数据包传送到 C2。如此类推，直到最后找到带有 IP 地址为 IP_B 的 B 主机的地址 MAC_B，最终传送给主机 B。在传输过程中，IP_A、IP_B 和 MAC_A 不变，而中间节点的 MAC 地址通过 ARP 在不断改变（M1，

M2，M3），直至目的地址 MAC_B。

综合上面所述，我们可以归纳出 IP 地址和 MAC 地址相同点是它们都唯一，不同的特点主要有：

(1) 网络上的某一设备，如一台计算机或一台路由器，其 IP 地址可变（但必须唯一），而 MAC 地址不可变。我们可以根据需要给一台主机指定任意的 IP 地址，如我们可以给局域网上的某台计算机分配 IP 地址为 192.168.0.112 ，也可以将它改成 192.168.0.200。而任一网络设备（如网卡，路由器）一旦生产出来以后，其 MAC 地址永远唯一且不能由用户改变。

(2) 长度不同。IP 地址为 32 位，MAC 地址为 48 位。

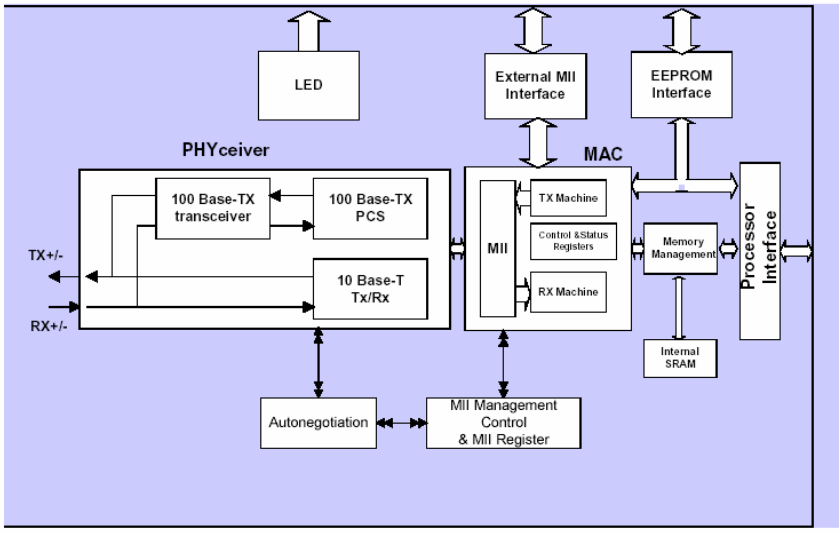
(3) 分配依据不同。IP 地址的分配是基于网络拓朴，MAC 地址的分配是基于制造商。

(4) 寻址协议层不同。IP 地址应用于 OSI 第三层，即网络层，而 MAC 地址应用在 OSI 第二层，即数据链路层。数据链路层协议可以使数据从一个节点传递到相同链路的另一个节点上(通过 MAC 地址)，而网络层协议使数据可以从一个网络传递到另一个网络上（ARP 根据目的 IP 地址，找到中间节点的 MAC 地址，通过中间节点传送，从而最终到达目的网络)。

2．DM9000：

2.1 概述

DM9000 是完全综合的、成本较低的单一快速以太网控制器芯片，具有通用的处理器接口，10/100M 自适应，以及 4K 双字节静态存取存储器。它被设计为低功耗、高处理性能，支持 3.3V 到 5V 的容差。DM9000 提供一个 MII 接口来连接 HPNA 设备或者其他支持 MII 接口的收发器，并支持 8 位、16 位、32 位的接口来适应不同的处理器对内部存储器的访问，它完全支持 IEEE 802.3u 规格，还支持 IEEE 802.3x 全双工流控制。DM9000 的设计非常简单，所以可以容易的完成不同系统的软件驱动开发。



DM9000 block diagram

2.2 特点

(1) 支持 ISA/uP 接口：位/字节/双字对内部存储器数据操作的 I/O 控制；

(2) 10/100M 的收发器；

(3) 支持 MII 和相反的 MII 接口；

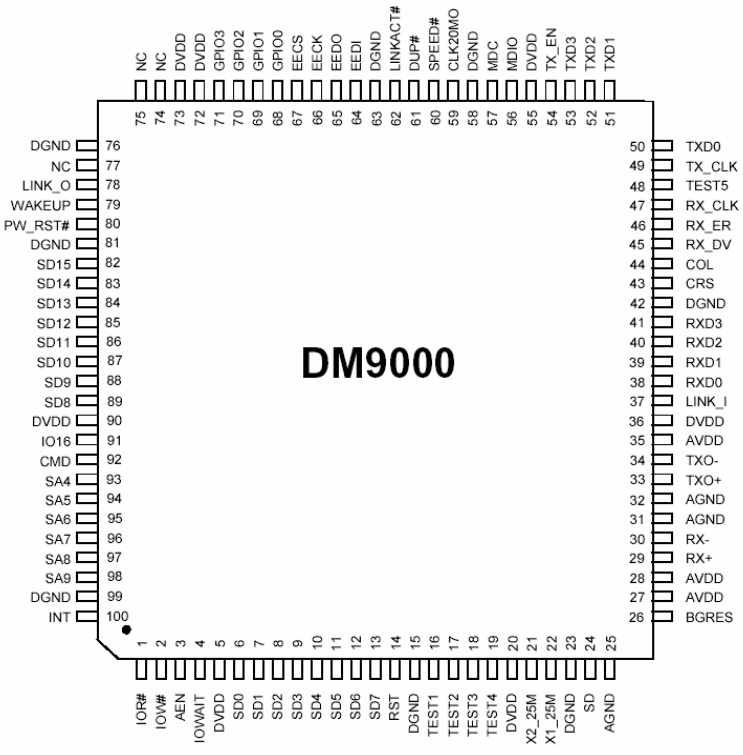
(4) 支持半双工流控制下的反向电压模式；

(5) 支持全双工模式下的 IEEE 802.3x 流控制；

(6) 支持连接状况变更、远程唤醒；

(7) 完整的 4K 双字节静态存取存储器；
(8) 支持从 EEPROM 中自动获取厂家及产品 ID；
(9) 支持 4 路 GPIO 引脚；
(10)可选的 EEPROM 配置；
(11)超低功耗模式；
(12)3.3V 到 5V 的容差；
(13)100 引脚的 LQFP。

2.3 引脚配置



Pin configurations（with MII interface）

2.4 DM9000 内部寄存器

寄存器	描述	偏移地址	初始状态
NCR	Network Control Register	00H	00H
NSR	Network Status Register	01H	00H
TCR	TX Control Register	02H	00H
TSR I	TX Status Register I	03H	00H
TSR II	TX Status Register II	04H	00H
RCR	RX Control Register	05H	00H
RSR	RX Status Register	06H	00H
ROCR	Receive Overflow Counter Register	07H	00H
BPTR	Back Pressure Threshold Register	08H	37H
寄存器	描述	偏移地址	初始状态
FCTR	Flow Control Threshold Register	09H	38H
FCR	RX Flow Control Register	0AH	00H
EPCR	EEPROM & PHY Control Register	0BH	00H
EPAR	EEPROM & PHY Address Register	0CH	40H
EPDRL	EEPROM & PHY Low Byte Data	0DH	Unknown

	Register		
EPDRH	EEPROM & PHY High Byte Data Register	0EH	Unknown
WCR	Wake Up Control Register	0FH	00H
PAR	Physical Address Register	10H-15H	Determined by EEPROM
MAR	Multicast Address Register	16H-1DH	Unknown
GPCR	General Purpose Control Register	1EH	01H
GPR	General Purpose Register	1FH	Unknown
TRPAL	TX SRAM read pointer address low byte	22H	00H
TRPAH	TX SRAM read pointer address high byte	23H	00H
RWPAL	RX SRAM write pointer address low byte	24H	04H
REPAH	RX SRAM write pointer address high byte	25H	0CH
VID	Vendor ID	28H-29H	0A46H
PID	Product ID	2AH-2BH	9000H
CHIPP	CHIP revision	2CH	00H
SMCR	Special Mode Control Register	2FH	00H
MRCMDX	Memory data read command without address increment Register	F0H	Unknown
MRCMD	Memory data read command with address increment Register	F2H	Unknown
MRRL	Memory data read_address Register low byte	F4H	00H
MRRH	Memory data read_address Register high byte	F5H	00H
MWCMDX	Memory data write command without address increment Register	F6H	Unknown
MWCMD	Memory data write command with address increment Register	F8H	Unknown
MWRL	Memory data write_address Register low byte	FAH	00H
MWRH	Memory data write_address Register high byte	FBH	00H
TXPLL	TX packet length low byte register	FCH	Unknown
TXPLH	TX packet length high byte register	FDH	Unknown
ISR	Interrupt Status Register	FEH	00H
IMR	Interrupt Mask Register	FFH	00H

Vendor Control and Status Register set

DM9000 的内部寄存器的访问不是直接读写的，而是通过地址 PORT 和数据 PORT（即 DM9000 留给外部的两个访问口，而其他的端口都不能读写）来访问的，其实现代码在：/RUIJIARM9-EDU/kernel/driver/net/dm9000.c 中：

```
/*因为 S3C2410 保存数据使用的是 little endian，与网络字节序相同，所以不需要替换数据格式
*/
#define   DATA(a)       (a)
/*数据以 word 或 byte 方式读出*/
inline void outw(unsigned short value,unsigned long addr)
{
```

```
*(volatile unsigned short *)addr = DATA(value);
}
inline void outb(unsigned char value,unsigned long addr)
{
*(volatile unsigned short *)addr = (value & 0x00FF);
}
/*数据以 byte 或 word 方式写入*/
inline unsigned char inb(unsigned long addr)
{
unsigned short val;
val = *(volatile unsigned short *)addr;
return (unsigned char)(val & 0x00FF);
}
inline unsigned short inw(unsigned long addr)
{
unsigned short val;
val = *(volatile unsigned short *)addr;
return (DATA(val));
}
```

3．以太网的驱动原理

3.1 网卡驱动程序

驱动程序提供了面向操作系统核心的接口和面向物理层的接口。驱动程序的操作系统接口是一些用于发现网卡、检测网卡参数以及发送接收数据的例程。当驱动程序开始运作时，操作系统首先调用检测例程以发现系统中安装的网卡。如果该网卡支持即插即用，那么检测例程应该可以自动发现网卡的各种参数；否则你就要在驱动程序运作前，设置好网卡的参数供驱动程序使用。当核心要发送数据时，它调用驱动程序的发送例程。发送例程将数据写入正确的空间，然后激活物理发送过程。驱动程序面向物理层的接口是中断处理例程。当网卡接收到数据、发送过程结束，或者发现错误时，网卡产生一个中断，然后核心调用该中断的处理例程。中断处理例程判断中断发生的原因，并进行响应的处理。比如当网卡接收到数据而发生中断时，中断处理例程调用接收例程进行接收。

3.2 驱动程序工作参数

驱动程序的工作参数因网卡性质的不同而不同，大致包括 I/O 端口号、中断号、DMA 通道、共享存储区等。输入输出端口号又被称为输入输出基地址，当网卡工作于端口输入输出模式时被使用。端口输入输出模式需要 CPU 的全程干预，但所需硬件及存储空间要求较低。CPU 通过端口号指定的空间与网卡交换数据。中断号是网卡的中断序号，只要不与其它设备冲突即可。当网卡使用 DMA 方式时，它要使用 DMA 通道批量传输数据而不需要 CPU 的干预。对于一块具体的网卡，如果网卡支持完全自动检测，那么一个参数也不用指定，驱动程序的检测例程会自动设定所需参数。一般情况，你需要人工设定这些参数的一部分。如果你的网卡使用端口输入输出模式，你要设定端口号和中断号。如果你的网卡使用 DMA 模式，你要设定 DMA 通道和中断号。如果你的网卡使用共享存储区的模式，那你就得设定共享存储区的地址范围。

3.3 驱动程序的使用方式

有了网卡的驱动程序后，你可以选择是把驱动程序加入到 Linux 核心之中还是把驱动程序加工成

独立模块。Linux 系统一个引人入胜的长处就是可以定制系统的核心。把需要频繁调用的功能加入系统核心，可以大大提高系统的效率。在这种情况下系统启动时，系统核心自动加载网卡的驱动程序。驱动程序的参数可以通过 LILO 命令参数加以指定。系统启动后驱动程序永久驻留核心，不能用常规的方法将其卸载。至于定制的系统核心，是通过重新编译得到的；如何编译核心将在后文叙及。如果把驱动程序编译成可装载模块，就可以用系统提供的命令在系统启动后随时加载。随时加载的好处是减少内存开销，易于管理，但同时也牺牲了一点网络传输的效率。驱动程序的参数是在命令行中直接输入或通过配置文件指定。

3.4 嵌入式系统 S3C2410 中的以太网驱动程序解析

嵌入式系统 S3C2410 中，以太网的驱动是通过/RUIJIARM9-EDU/kernel/driver/net/dm9000x.c 来实现的。

```
int init_module(void)
{
DMFE_DBUG(0, "init_module() ", debug);
if (debug) dmfe_debug = debug;                    /在此设置 debug 标志
switch(mode) {
        case DM9000_10MHD:
        case DM9000_100MHD:
        case DM9000_10MFD:
        case DM9000_100MFD:
        case DM9000_1M_HPNA:
            dia_mode = mode;
            break;
        default:
            dia_mode = DM9000_AUTO;    //判断 DM9000 的模式
}
     loor = (nfloor > 15) ? 0:nfloor;
     return dmfe_probe(0);
    //search board and register,跳到函数 dmfe_probe()
}

int __init dmfe_probe(struct DEVICE *dev)
{
unsigned char buff[7];
int j;
static char sEthernet[5];
struct board_info *db;        /* Point a board information structure */
u32 id_val;
u32 iobase;
u16 i, dm9000_count = 0;
u8   irqline;

BWSCON     |= 0xc0;
GPFCON     |= 0x2;        //EINT0 from PORT F control register
```

```
EXTINT0      |= 0x4；          //EINT0 Rising edge triggered
INTMSK       &= 0xfff7；  //EINT0 Enable     //在此将中断使能

net_init_iic()；                                    //初始化 IIC
net_iic_read(0xaf,0x20,buff,12)；                       //调用函数 net_iic_read()从 EEPROM
                                                    //中地址 0x20 处读取 MAC 地址

if(strcmp(dev->name,sEthernet)!=0)
     strcpy(sEthernet,dev->name);
else return;                                        //当有多个网卡时，通过它循环检测
if(strcmp(dev->name,"eth0")==0)                     //检查网卡设备是否为 eth0
{
     iobase = ioremap(DM9000_MIN_IO_ETH1,0x400)；
     dev->irq = 0；                                 //EINT0, 设置设备中断号为：0
         for (i=0；  i<6；  i++)
     {
          j = (2 * i) + 1;
          dev->dev_addr[i] = buff[j]；              //将从 EEPROM 中获取得 MAC 地
                                                        //址赋给网卡设备
               printk("%02X:",dev->dev_addr[i])；
     }
}else
if(strcmp(dev->name, "eth1")==0)                        //如果有其它的网卡设备
{
     ……
}
     else                                           //所有的网卡设备都被检测
     return 0；

/* Disable all interrupt */
/* Search All DM9000 NIC */
outb(DM9000_VID_L, iobase)；
id_val = inb(iobase + 4)；
outb(DM9000_VID_H, iobase)；
id_val |= inb(iobase + 4) << 8；
outb(DM9000_PID_L, iobase)；
id_val |= inb(iobase + 4) << 16；
outb(DM9000_PID_H, iobase)；
id_val |= inb(iobase + 4) << 24；
                                                    // 读 取 网 卡 芯 片 中 的 固 定 地 址
                                                    //0x28,0x29,0x2A,0x2B 中的值，并赋给//id_val

if (id_val == DM9000_ID) {
                                                //DM9000 芯片的 0x28,0x29,0x2A,0x2B
```

```
                                        //中的值固定为 0x90000A46，如果 id_val
                                        //的值也为 0x90000A46，则可确定使用
                                        //的网卡芯片是 DM9000
        dm9000_count++;
        /* Init network device */
        dev = init_etherdev(dev, 0);          //调用函数 init_etherdev()，此函数在
                                              //net_init.c 中

        /* Allocated board information structure */
        rqline = 3;
        db=(void*)(kmalloc(sizeof(*db), GFP_KERNEL|GFP_DMA));
        emset(db, 0, sizeof(*db));
        dev->priv = db;    /* link device and board info */
        db->next_dev = dmfe_root_dev;
        dmfe_root_dev = dev;
        db->ioaddr = iobase;
        db->io_data = iobase + 4;

        /* driver system function */
        dev->base_addr = iobase;
        dev->open = &dmfe_open;
        dev->hard_start_xmit = &dmfe_start_xmit;
        dev->stop = &dmfe_stop;
        dev->get_stats = &dmfe_get_stats;
        dev->set_multicast_list = &dm9000_hash_table;
        dev->do_ioctl = &dmfe_do_ioctl;
                                        //当调用设备打开、读写、关闭等操作时，
                                        //调用相应的函数，如：设备打开，调用
                                        //函数 dmfe_open（）
    }

    return dm9000_count ? 0:-ENODEV;
    }
```

三．实验内容和步骤

1．获取宿主机与开发板的 MAC 地址：

(1) 根据宿主机的系统环境，获取宿主机的 MAC 地址和 IP 地址并记录；

(2) 在开发板运行 ifconfig，获取开发板的 MAC 地址和 IP 地址并记录。

2．用对接线将开发板与宿主机相连，改写开发板的 MAC 地址，观察结果：

(1) 在开发板上运行程序 setmac ff ff ff ff ff ff，向 EEPROM 中地址 0x20 处写入实新的 MAC 地址 FF:FF:FF:FF:FF:FF（无效地址）；

(2) 重新启动开发板，运行 ifconfig，查看开发板的 IP 地址和 MAC 地址，ping、mount 宿主机，观

察结果，并记录此时宿主机和开发板的网络状态；

(3) 向 EEPROM 中地址 0x20 处写入宿主机的 MAC 地址；

(4) 重新启动开发板，运行 ifconfig，查看开发板的 IP 地址和 MAC 地址，ping、mount 宿主机，观察结果，并记录此时宿主机和开发板的网络状态；

(5) 向 EEPROM 中地址 0x20 处写入原开发板的 MAC 地址；

(6) 重新启动开发板，运行 ifconfig，查看开发板的 IP 地址和 MAC 地址，ping、mount 宿主机，观察结果，并记录此时宿主机和开发板的网络状态。

四．思考题

1. 结合实验中的结果，分析 MAC 地址和 IP 地址对网络状态的影响。
2. 如果网络内有两台机器具有相同的 MAC 地址，那同时发送信息给这两台机器(比如同时 ping 这两台机器)，会发生什么情况？为什么？

实验十七： LCD 实验

一．实验目的

理解 lcd 驱动的移植原理，理解 CPU 手册中 LCD 相关寄存器的配置。能够使用基于 Frame Buffer 的应用程序的实现。对各种 LCD 屏能够正确移植。理解 Minigui 的工作原理，实现对 minigui 的支持。

二．实验原理和说明

1. LCD 原理综述

液晶显示器 LCD（Liquid Crystal Display），特别是点阵式液晶，已经成为现代仪器仪表用户界面的主要发展方向，它不仅省电，而且能够显示大量的信息，如各种文字、曲线等等。它比传统的数码管显示器有了质的提高，但是点阵式液晶的驱动电路相对复杂一些，价格页比较高。因此，降低成本，减少系统的复杂成都，对点阵式液晶的应用具有重要的意义。

液晶是一种介于液体和固体之间的热力学的中间稳定物质形态。其特点是在一定的温度范围内既有液体的流动性和连续性，又有晶体的各向异性，其分子呈长棒形，长宽之比较大，分子不能弯曲，是一个刚性体，中心一般有一个桥链，分子两头有极性。

LCD 器件的结构如图 1 所示。由于液晶的四壁效应，在定向膜的作用下，液晶分子在正、背玻璃电极上呈水平排列，但排列方向为正交，而玻璃间的分子呈连续扭转过度，这样的构造能使液晶对光产生旋光作用，使光偏转方向旋转 90 度。

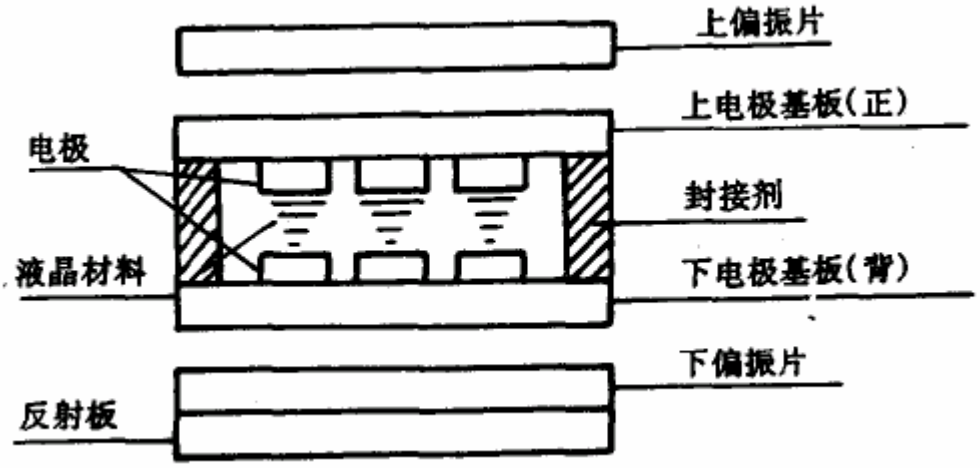


图 1 液晶显示器基本构造

图 2 显示了液晶显示器的工作原理。当外部光线通过上偏振片后形成偏振光，偏振方向成垂直排列，当此偏振光通过液晶材料以后，被旋转 90 度，变成方向成水平方向，此方向和下偏振片的偏振方向一致，因此此光能完全穿过下偏振片而达到反射极，经反射后沿远路返回，从而呈现出透明状态。当液晶盒的上、下电极加上一定的电压后，电极部分的液晶分子转成垂直排列，从而失去旋光性。因此，从上偏振片入射的偏振光不被旋转，当此偏振光到达下偏振片时，因为其偏振方向与下偏振片的方向垂直，因而被下偏振片吸收，无法到达反射板形成反射，所以呈现出黑色。根据需要，将电极做成各种文字、数字或者点阵，就可以获得所需的各种显示。

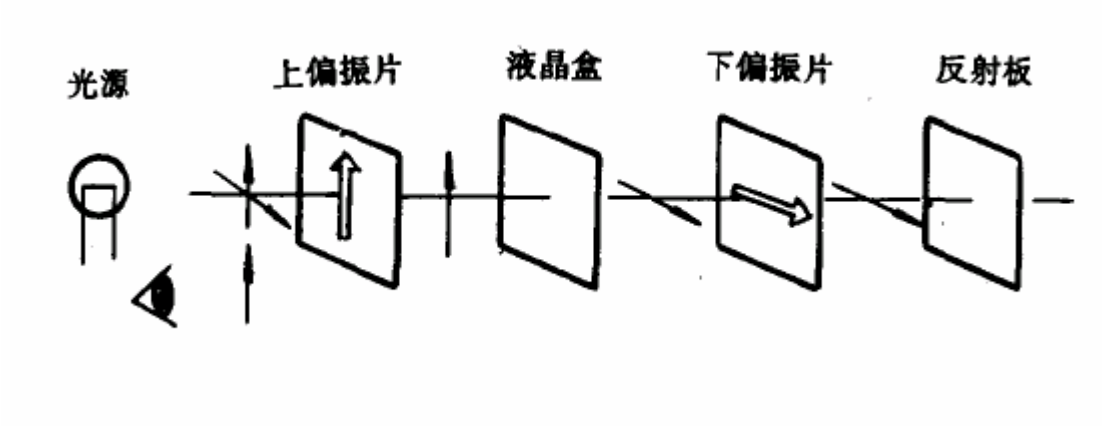


图 2

液晶显示器的工作原理

2. LCD 的驱动方式

液晶显示器的驱动方式由电极引线的选择方向确定，因此，在选择号液晶显示器之后，用户无法改变驱动方式。

液晶显示器的驱动方式一般由静态驱动和动态驱动两种。由于电流电压驱动 LCD 会使液晶体产生电解和电极老化，从而大大降低 LCD 的使用寿命，所以现在的驱动方式多属于交流电压驱动。

2.1 静态驱动方式

静态驱动回路以及波形图如图 3 所示。其中 LCD 表示某个液晶显示字段，当此字段上两个电极的电压相同时，两电极之间的电位差为零，该字段不显示，当此字段上两个电极的电压相位相反时，两电极之间的电位差不为零，为二倍幅值的方波电压，该字段呈现出黑色显示。

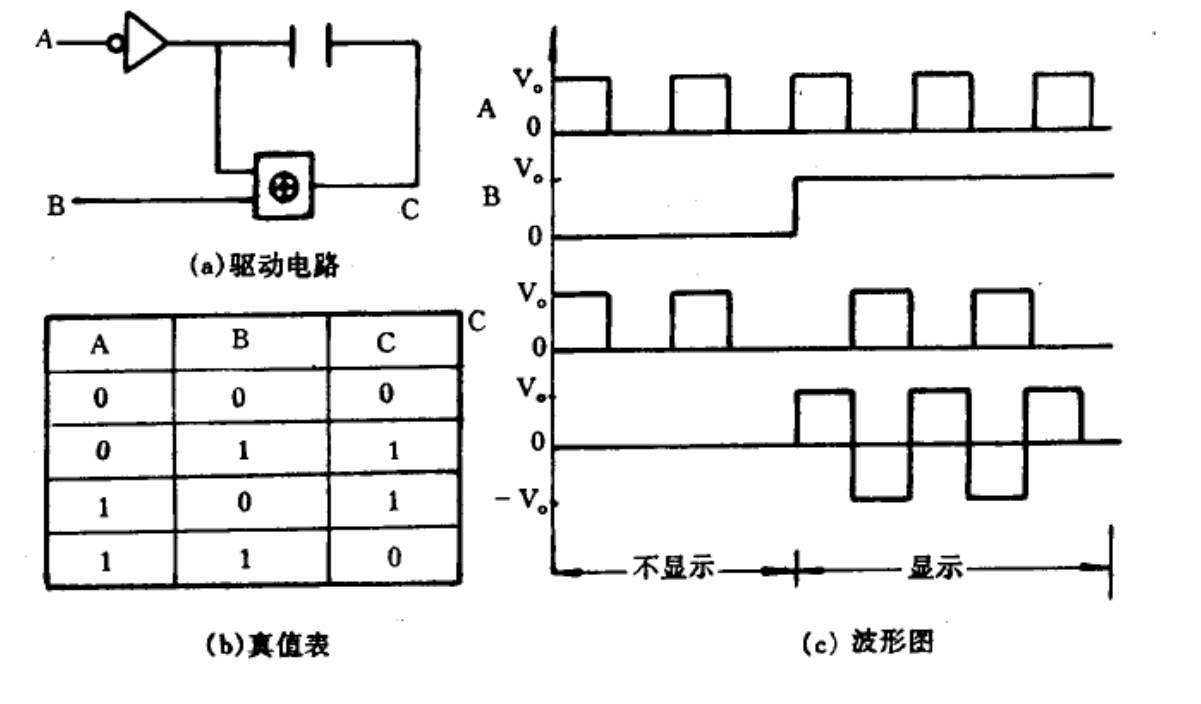


图 3 静态驱动回路及波形

液晶显示的驱动鱼 LED 的驱动有很大的不同，对于 LED 而言，当在 LED 两端加上恒定的导通兆截止电压便可控制其亮或者暗。而 LCD，由于其两极不能加恒定的电流电压，因而给驱动带来复杂性。一般应在 LCD 的公共极（一般为背极）加上恒定的交变方波信号，通过控制前极的

电压变化而在 LCD 两极间产生所需的零电压或二倍幅值的交变电压，以达到 LCD 亮、灭的控制。目前已经有很多 LCD 驱动集成芯片，这些芯片已经将多个 LCD 驱动电路集成到一起，使用起来根 LED 驱动芯片一样方便，而且形式非常相似。

图 4 是七段液晶显示器的电极配置和静态驱动电路。七段共用一个背极 BP（Black Place），前极 a、b、c、d、e、f、g 互相独立，每段各加上一个异或门进行驱动，显示字符同 LED。

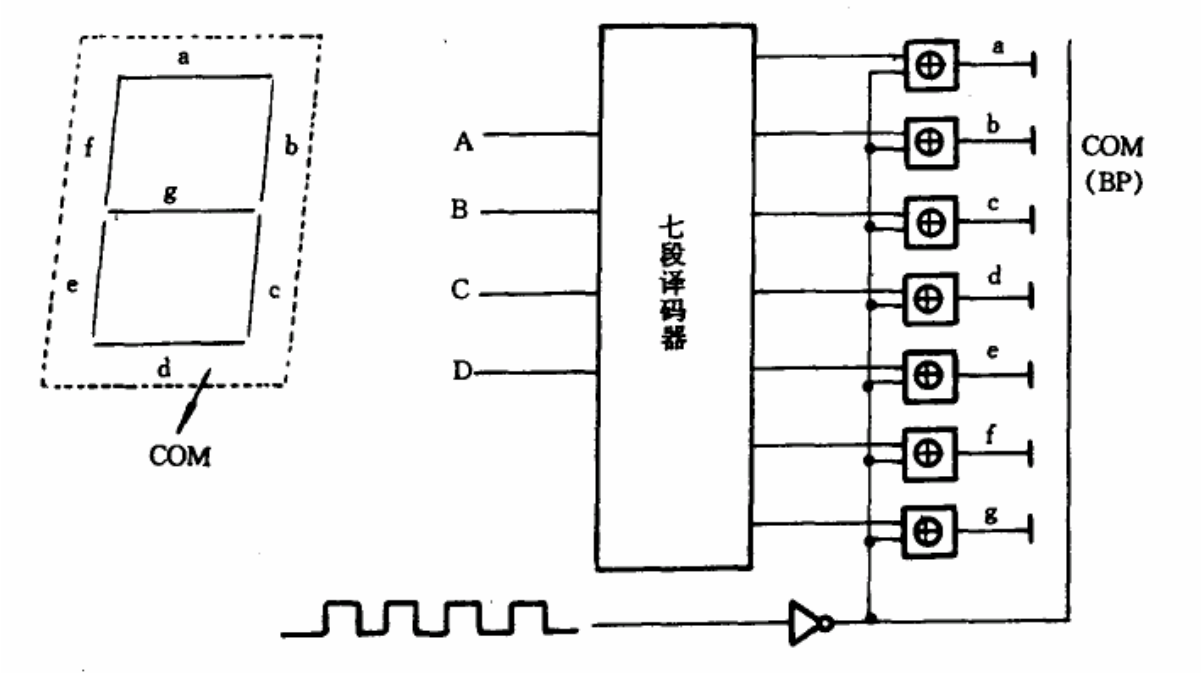


图 4 七段 LCD 显示电路

2.2 动态驱动方式

动态驱动实质上是矩阵扫描驱动，可用于多位的 8 段数码显示和点阵显示。点阵显示是把液晶置于处置的条状电极之间，异条状电极交点的组合来显示，可组成图形和各种字符显示。这种显示形状如果只在显示点有关的行列上加电压，则非显示点也会因为有线电压，产生所谓的“交叉效应”，是对比度下降。一般在非选中点页加上低于阀值的电压，以清除交叉效应的影响。常用的方法有偏压法和双频法。此外，对矩阵各点的驱动要采用分时的方法，其背电极 BP 为行线，分时对各行加上阀值电压。因此，在行线扫描周期内（称为帧周期）阀值电压的占空比为 1/行数。

时分割驱动方式通常采用电压平均化法，常用的动态驱动有 2 分时、3 分时和 4 分时等等动态驱动，或称 1/2、1/3、1/4 占空系数等等。

下面以点阵式 LCD 的驱动来介绍时分割驱动方法。

在点阵式 LCD 中，使用了行驱动和列驱动，使所选通点上的选通电压大于开启电压，但由于多点共用一个电极，在选通点外的非选通点上也加有电压，从而使清晰度下降，这就是交叉效应现象。如果在非选通点上十佳只有选通电压的 1/2，使非选通的电压值低于显示的截止电压，这样将减少交叉效应现象，这就是 1/2 偏压法，波形见图 5 所示。实际应用中常用 1/3、1/4、1/7 等偏压法，使选通电压与非选通电压之间的差距加大，以提高显示的清晰度。

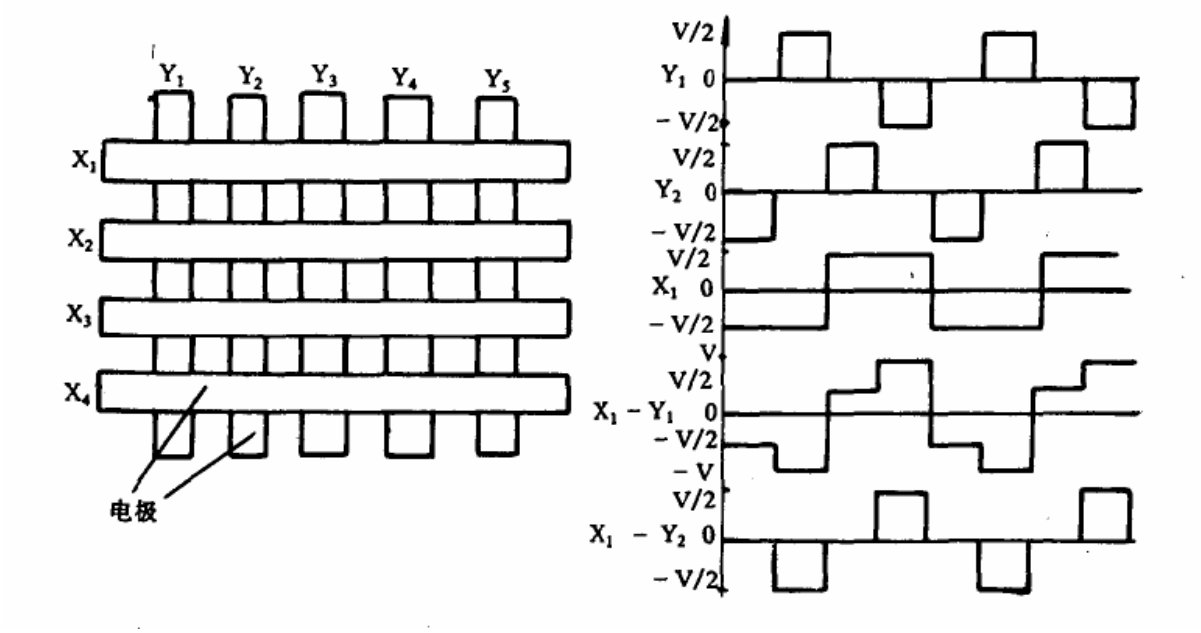


图 5 点阵 LCD 以及驱动波形

点阵式 LCD 的控制一般采用行扫描方式，原理参见图 6 所示。各行所施加的电压脉冲占空比为 1/行数，占空比越小，清晰度就越差，甚至还会产生闪烁现象。图 6 显示仅为一个字符，当一行有多个字符时，先把列数据以串行码方式输出给列驱动器，然后产生行扫描，以实现显示状态。

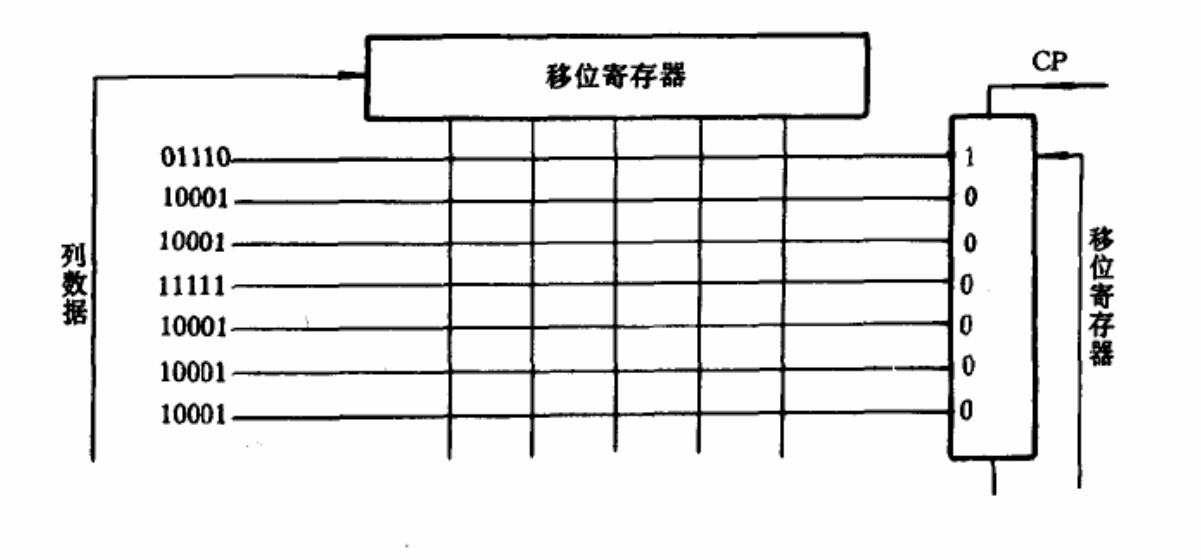


图 6 行扫描驱动原理

2.3 LCD 显示控制和驱动接口电路

随着液晶显示技术的迅速发展，各家场商竞相推出各种专用的控制和驱动大规模集成电路 LSI，使得液晶显示的控制和驱动极为方便，而且可以由 CPU 直接控制，以满足用户液晶显示的多种要求。

目前，这类 LSI 可大致分为如下五类：

(1) 自带 4 位 CPU 的段式液晶显示驱动 LSI；
(2) 控制驱动点阵位图式液晶显示 LSI；
(3) 控制驱动点阵字符式液晶显示 LSI；
(4) 控制驱动点阵图形式液晶显示 LSI；
(5) 视频 LCD 接口控制和驱动 LSI。

下面要介绍 TOSHIBA 的 T6963C 点阵图形式液晶显示 LSI。这类 LSI 的特点是：自带字符 ROM，可以产生标准的 128 个 ASCII 字符供用户调用。还用外接扩展 RAM 存储若干屏的显示数据，即几幅图形的显示数据。还可以在图形模式下显示汉字，具有丰富的显示控制指令，常用于驱动点阵图形式液晶显示模块。

T6963C 控制与驱动点阵图形式液晶显示 LSI，通过对其管脚的不同预置可支持多种 LCD 的显示格式，还可以进行文本显示和文本、图形混合显示等等。T6963C 的原理框图和应用举例如图 7 和图 8 所示。

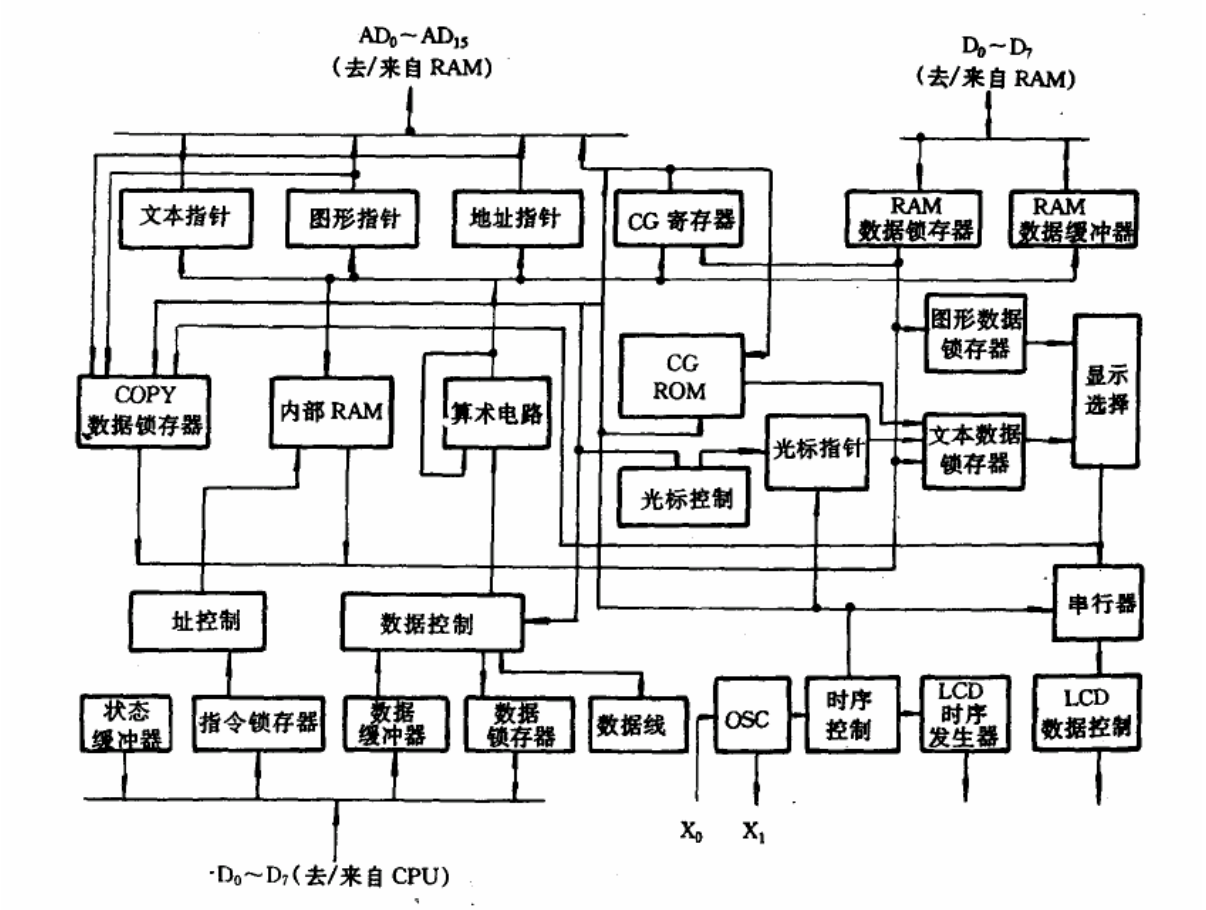


图 7：T6963C 方框图

T6963C 具有如下特性：

(1) 显示最多可达 80 字符×8 行，自行可选 5×8、6×8、7×8、8×8 四种。
(2) 占空比选择：1/16～1/128。
(3) 字符 ROM 有 128 字符，显示 RAM 可扩展到 64KB。
(4) 文本显示，图形显示，文本/图形的与、或、异或显示。
(5) 控制指令 10 条，有指针指令、控制字设置、模式设置、读写控制等。
(6) 显示驱动能力单屏 640×128，双屏 640×256。

与 LCD 显示相对的是 CRT 显示，多采用 VGA 显示系统。

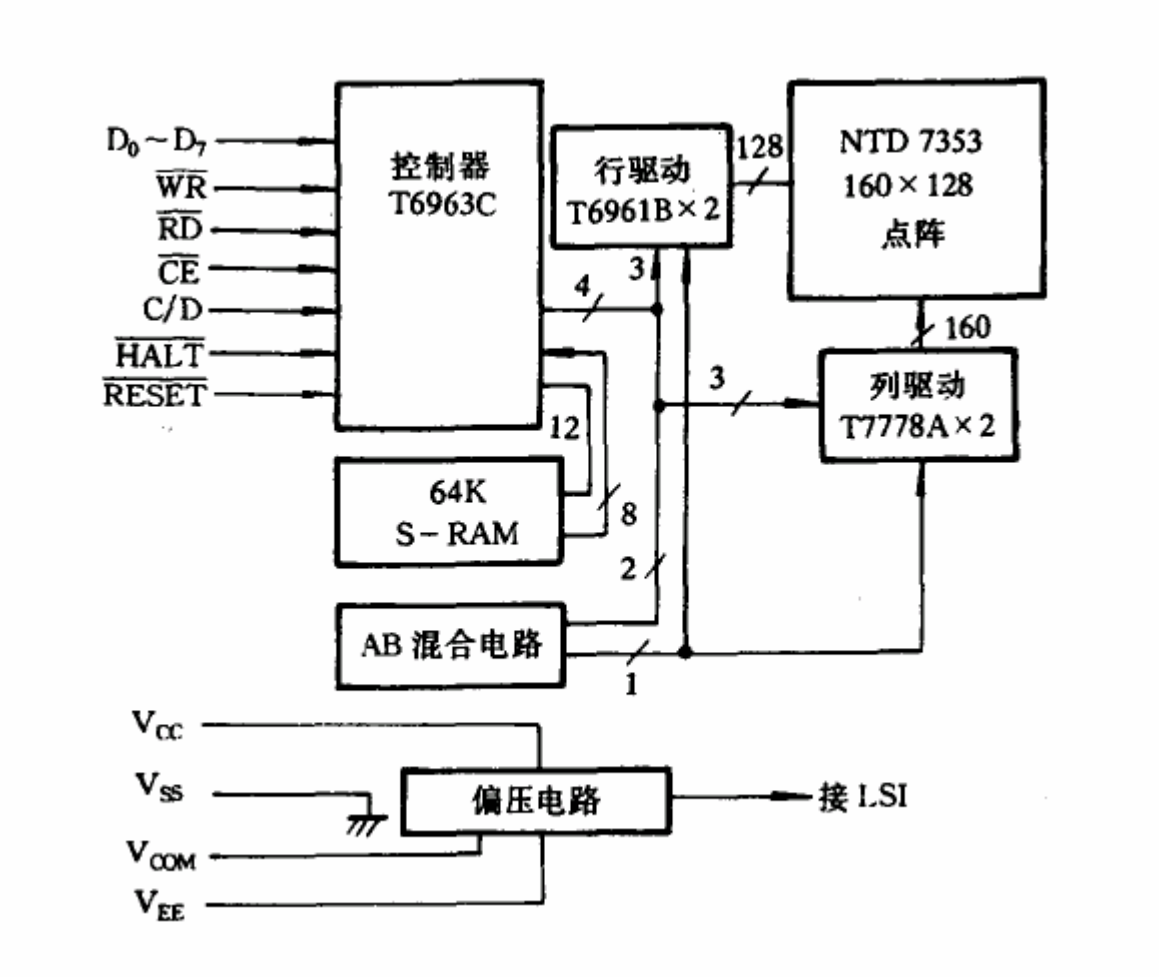


图 8 DM5001N 的驱动和控制原理

LCD 显示器通常与 Frame Buffer 设备结合使用，系统定时将 Frame Buffer 中的内容写道 LCD 显示器中，在驱动的编写过程中，需要根据实现 LCD 的大小来调整 CPU 寄存器中相关寄存器中保存 LCD 大小的参数，即 x，y 坐标的最大值，同时根据 CPU 的时钟频率来修改 LCD 的刷新频率，保证 lcd 的刷新频率在 50～60hz 即可。

用户程序进行 lcd 写操作的时候，通常在应用程序初始化的时候映射 Frame Buffer 到内存中的一个地址空间，这部分地址空间中的内容就对应着 LCD 显示器中的显示内容，lcd 中每一点对应的 bit 宽度不同，将导致 lcd 显示器的内容在内存空间的分布宽度。内存空间的内存分布一一对应于 LCD 上的坐标点。我们通过改变内存空间的值例如从 0～0xffff 来使屏幕上的点由白到黑。

根据上面阐述的激励可以将各点连接起来，实现在屏幕上画线，甚至写汉字，画图等等一些功能。一些图形操作平台如 minigui 也是根据该原理实现的。

3．LCD 的选型心德

目前市场上主要有 Samsung 和 Sharp 这 2 个品牌的 LCD。其中 3.5"寸的 LCD 基本都是 240×320 的 QVGA，视角为 6 点钟方向。

Samsung 能提供的产品为 LTS350Q1_PD1（CCFL 背光）和 LTS350Q1_PE1（LED 背光）。LTS350Q1_PD1 是 Samsung 的早期产品，虽然可以直接由 S3C2410 直接控制驱动，但是 LTS350Q1_PD1 已经停产，半年前在市场上就已不见踪影。

LTS350Q1_PE1 是 Samsung 后来推出的产品，它内部采用 6 只 LED 背光，但是它必须通过 1 颗叫 LCC3600 的 IC 才能被 S3C2410 驱动，我习惯称它为"LCD 伴侣 IC"。但是采用 LTS350Q1_PE1

会遇到以下一些问题：

(1) LCC3600 这颗 IC 本身并不是 Samsung 公司自己的产品，而是当年有几个从 Samsung 出走的工程师自己开办公司研发的。由于 LCC3600 是专门与 LTS350Q1_PE1 这类 LCD 配套的，因此自然不得不利用 Samsung 的销售渠道，但是这家公司与 Samsung 的关系并不是很融洽，这使得 LCC3600 的供货始终不是很稳定。

(2) Samsung LCD 的产能虽然很大，但是由于 Samsung 本身的 LCD 用量很大（例如 LCD 显示器和彩屏手机），再扣除为其它公司做 OEM，或者是其它大公司的海量采购（例如 HP 之类的），这就使得 Samsung 的 LCD 产能还是不够，LTS350Q1_PE1 在一般市场上的供应量比较紧缺，供货也不是很好，除非你的用量很大，可以得到 Samsung 足够的重视那又另当别论。

(3) Sharp 的 3.5" LCD 的型号是 LQ035Q7DB02，采用 LED 背光；

(4) 另外还有 1 款 3.9"的型号是 LQ039Q2DS54/LQ039Q2DS53，采用 CCFL 背光。它们均为 6 点钟视角，系 Sharp 的 HR-TFT 系列。与 LTS350Q1_PE1 一样，它们也不能直接与 S3C2410 接口，也必须通过额外的 1 颗 IC 进行时序转换，从而能被 S3C2410 驱动。从性能上来比较 Samsung 和 Sharp 并没有太大的差异，Sharp 作为 LCD 的老牌厂家，与 Samsung 相比应该只会有过之而无不及。并且 Sharp LCD 的供货渠道非常多。甚至在有些对成本非常敏感的应用场合选用 LQ039Q2DS54 这样的"新屏"就可以降低 20+USD，几乎可以买到 1 颗 S3C2410＋64MB SDRAM 了。

4． 不同 LCD 驱动程序的移植

LCD 驱动编写的主要工作就是正确设置对应预新的 LCD 屏的 CPU 寄存器的设置，和 LCD 对应的 CPU 寄存器主要如下：

LCD Control 1 Register

Register	Address	R/W	Description	Reset Value
LCDCON1	0X4D000000	R/W	LCD control 1 register	0x00000000

LCD Control 2 Register

Register	Address	R/W	Description	Reset Value
LCDCON2	0X4D000004	R/W	LCD control 2 register	0x00000000

LCD Control 3 Register

Register	Address	R/W	Description	Reset Value
LCDCON3	0X4D000008	R/W	LCD control 3 register	0x00000000

LCD Control 4 Register

Register	Address	R/W	Description	Reset Value
LCDCON4	0X4D00000C	R/W	LCD control 4 register	0x00000000

LCD Control 5 Register

Register	Address	R/W	Description	Reset Value
LCDCON5	0X4D000010	R/W	LCD control 5 register	0x00000000

FRAME BUFFER START ADDRESS 1 REGISTER

Register	Address	R/W	Description	Reset Value
LCDSADDR1	0X4D000014	R/W	**STN/TFT**: Frame buffer start address 1 register	0x00000000

FRAME Buffer Start Address 2 Register

Register	Address	R/W	Description	Reset Value
LCDSADDR2	0X4D000018	R/W	**STN/TFT**: Frame buffer start address 2 register	0x00000000

FRAME Buffer Start Address 3 Register

Register	Address	R/W	Description	Reset Value
LCDSADDR3	0X4D00001C	R/W	**STN/TFT**: Virtual screen address set	0x00000000

RED Lookup Table Register

Register	Address	R/W	Description	Reset Value
REDLUT	0X4D000020	R/W	**STN**: Red lookup table register	0x00000000

GREEN Lookup Table Register

Register	Address	R/W	Description	Reset Value
GREENLUT	0X4D000024	R/W	**STN**: Green lookup table register	0x00000000

BLUE Lookup Table Register

Register	Address	R/W	Description	Reset Value
BLUELUT	0X4D000028	R/W	**STN**: Blue lookup table register	0x0000

Dithering Mode Register

Register	Address	R/W	Description	Reset Value
DITHMODE	0X4D00004C	R/W	**STN**: Dithering mode register. This register reset value is 0x00000 But, user can change this value to 0x12210. (Refer to a sample program source for the latest value of this register.)	0x00000

Temp Palette Register

Register	Address	R/W	Description	Reset Value
TPAL	0X4D000050	R/W	**TFT**: Temporary palette register. This register value will be video data at next frame.	0x00000000

LCD Interrupt Pending Register

Register	Address	R/W	Description	Reset Value
LCDINTPND	0X4D000054	R/W	Indicate the LCD interrupt pending register	0x0

LCD Source Pending Register

Register	Address	R/W	Description	Reset Value
LCDSRCPND	0X4D000058	R/W	Indicate the LCD interrupt source pending register	0x0

LCD Interrupt Mask Register

Register	Address	R/W	Description	Reset Value
LCDINTMSK	0X4D00005C	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced.	0x3

LPC3600 Control Register

Register	Address	R/W	Description	Reset Value
LPCSEL	0X4D000060	R/W	This register controls the LPC3600 modes.	0x4

关于各个寄存器的详细设置清参照2410的用户手册P372～P413部分。

如果系统使用的 LCD 发生了变化，则对应的驱动程序就需要更改，现在我们针对各种系统的 LCD 的移植分别加以阐述。

LCD 的移植由小屏换大屏需要更改的地方主要有：点频、帧频、场频、屏的长宽以及多少位的颜色、屏的点极性等等。

以 MXL 为例，屏的大小长宽信息可由 LCD 的 SIZE 寄存器直接设置，屏的多少位色，显示极性等信息可由 PCR 寄存器设置，这些可直接参见 CPU 的手册，通常没有什么难度。

屏的大小设置：参见 mxl 的代码，在 dbmx1fb.h 中，修改宏定义

```
#define LCD_MAXX        640    //240
#define LCD_MAXY        480    //320
```

这将使寄存器 DBMX1_LCD_XYMAX 的设置发生变化，DBMX1_LCD_XYMAX 寄存器就是用来设置 LCD 屏的大小的。

参见代码 dbmx1fb.c 文件：

```
val =0;
val = current_par.xres/16;
val = val<<20;
val += current_par.yres;
DPRINTK(KERN_ERR"par.x=%x, y=%x\n",
                current_par.xres,
                current_par.yres);
WRITEREG(DBMX1_LCD_XYMAX, val);
```

调试 LCD 最关键的一步在于频率的调整，这部分需要软件和硬件人员配合调整。

以 MXL 为例，240×320 的小屏的点频、帧频和场频分别为 5.3Mhz,15Khz,60hz。现在需要更改为 640×480 的大屏，通常的屏都需要保持固定的场频，就是 50～60hz 左右，这样我们新的大屏也就需要 50～60hz 的场频（场频就是指屏幕上每一点的刷新频率），对应的帧频应该为 30Khz（原来的 2 倍左右），对应的点频应该为原来的 4~5 倍。

屏幕的总大小变为原来的 4 倍，所以屏幕的点频应该变为原来的 4 倍左右，增加一定的富裕度，通常可选择原来的 5 倍左右。

屏的点频（pixel clock）是由用户设定的，而对应的帧频和场频是由屏幕的长宽等信息和屏幕刷新完一点的等待周期等等信息对应后自动给出的。

小屏的点频为 5.3Mhz，根据 PCR 寄存器的设置，为 LCDC_CLOCK 时钟 3 分频以后的结果，则 LCDC_CLOCK 时钟的频率应该为 16Mhz 左右，即使以该时钟频率不再分频而直接用于 640×480 大屏的时钟，也不足以提供大屏 25Mhz 的时钟频率的需求，所以重新调整 LCDC_CLOCK 时钟的频率，使其至少为 25Mhz，以满足新的大屏的时钟需求。

Mxl 的 LCDC_CLOCK 即 perclock2 时钟是由系统时钟分频所得，对系统时钟对 perclock2 的分

频设置由 PCDR 寄存器进行设置，系统默认的是 5 分频，可见系统时钟应该在 80Mhz 左右，可以采用 3 分频可得 25M 左右的系统时钟。

上述为大致理论计算的结果，实际使用过程中，因为系统时钟的分频不可能做到连续变化，只能选取间断的值，所以在实际应用中往往要反复实验各个时钟的选取，以选择一个最理想的时钟。

其中修改系统 PCDR 寄存器的代码如下所示：

```
 val=tmp=0;
val= READREG(0X0021B020);
tmp=0x00000040;
val&=0xffffff0f;
val+=tmp;
WRITEREG(0X0021B020, val);
```

其中 val 首先读 PCDR 寄存器的值，然后修改 4～7 位，为 perclock2 的分频，上述代码是将其改为 4 分频。即设置 perclock2 的时钟位 80Mhz/5=20Mhz。修改 PCDR 寄存器的值应该在系统启动后尽量早的地方执行，我们将其放置于 init/main.c 文件中。

设置 lcd 的时钟分频：

```
WRITEREG(DBMX1_LCD_PANELCFG, 0xF8C88B80);
```

其中 PCR 寄存器的最后六位为 LCD 的时钟分频，设为 0 表示使用 perclock2 时钟，如果设置为 1～63，表示对应的 2～64 倍的分频。

上述设置表示 lcd 的时钟不再使用分频，如果要改为 3 分频的话，只要设置为：

```
WRITEREG(DBMX1_LCD_PANELCFG, 0xF8C88B82);
```

即可。

经过反复测试，发现 PCDR 寄存器对 perclock2 采用 5 分频而 lcd 的时钟不再分频的效果最好。

2410 240×320 的小屏 更换 640×480 的大屏的更改代码如下：

240×320 小屏：

```
#ifdef CONFIG_S3C2410_SMDK
static struct s3c2410fb_mach_info xxx_stn_info __initdata = {
        pixclock:       174757,         bpp:            16,
#ifdef CONFIG_FB_S3C2410_EMUL
        xres:           96,
#else
        xres:           240,
#endif
        yres:           320,

        hsync_len    :  5,     vsync_len     :  1,
        left_margin :   7,     upper_margin :   1,
        right_margin:   3,     lower_margin :   3,

        sync:           0,              cmap_static:    1,
        reg : {
            lcdcon1: LCD1_BPP_16T | LCD1_PNR_TFT | LCD1_CLKVAL(7) ,
            lcdcon2 : LCD2_VBPD(4) | LCD2_VFPD(1) | LCD2_VSPW(1),
            lcdcon3 : LCD3_HBPD(6) | LCD3_HFPD(30),
```

```
            lcdcon4 : LCD4_HSPW(3) | LCD4_MVAL(13),
            lcdcon5 : LCD5_FRM565 | LCD5_HWSWP | LCD5_PWREN,
        },
};
```

640×480 的代码如下：

```
#ifdef CONFIG_S3C2410_SMDK
static struct s3c2410fb_mach_info xxx_stn_info __initdata = {
        pixclock:       174757,         bpp:            16,
#ifdef CONFIG_FB_S3C2410_EMUL
        xres:           96,
#else
        xres:           640,
#endif
        yres:           480,

        hsync_len   :  5,    vsync_len    :  1,
        left_margin :  7,    upper_margin :  1,
        right_margin:  3,    lower_margin :  3,

        sync:           0,              cmap_static:    1,
        reg : {
          lcdcon1 : LCD1_BPP_16T | LCD1_PNR_TFT | LCD1_CLKVAL(1) ,
          lcdcon2 : LCD2_VBPD(60) | LCD2_VFPD(60) | LCD2_VSPW(20),
          lcdcon3 : LCD3_HBPD(48) | LCD3_HFPD(6),
          lcdcon4 : LCD4_HSPW(96) | LCD4_MVAL(13),
          lcdcon5 : LCD5_FRM565 | LCD5_INVVLINE | LCD5_INVVFRAME |
                LCD5_HWSWP | LCD5_PWREN,
        },
};
#endif
```

更改的部分主要为：屏的大小尺寸由 xres，yres。

LCD1_BPP_16T：还是 16 位彩色

LCD1_PNR_TFT：采用 TFT 模式，能采用这种，当然用这种。

CLKVAL 原来的值为 7，则原来的时钟频率为 60/((7+1)*2)，则现在屏的大小变为原来的 4 倍，则时钟频率也应该为原来的 4 倍，设现在的 clkval 为 x，则 60/((x+1)*2)=4*60((7+1)*2)，即 x＝1。故设 LCD1_CLKVAL(1)。

```
        lcdcon2 : LCD2_VBPD(60) | LCD2_VFPD(60) | LCD2_VSPW(20),
        lcdcon3 : LCD3_HBPD(48) | LCD3_HFPD(6),
        lcdcon4 : LCD4_HSPW(96) | LCD4_MVAL(13),
```

LCD2_VBPD 表示垂直同步周期之后，在一帧开始的时候的非活动周期

LCD2_VFPD,表示垂直同步周期之前，在一帧结束的时候的非活动周期

LCD2_VSPW 垂直同步脉冲宽度，通过检测非活动周期的宽度来确定垂直脉冲的高电平的宽度。

LCD3_HBPD，LCD3_HFPD，LCD4_HSPW 为水平同步的对应值。

这六个值都可以查找 LCD 的手册，均可以查到对应该 LCD 的各个值的最小值。更改上述值使之大于对应的最小值即可。

LCD4_MVAL 是 STN 模式下使用的参数，不需要进行调整。

LCD5_INVVLINE 用来设置 vline/hsync 脉冲的极性，可根据具体的硬件参数进行调整。

LCD5_INVVFRAME用来设置VFRAME/VSYNC脉冲的极性，可根据具体的硬件参数进行调整。

更改屏的尺寸时主要更改以上参数就可以了。

如下所示采用单色屏的驱动：

```
#ifdef CONFIG_S3C2410_SMDK
static struct s3c2410fb_mach_info xxx_stn_info __initdata = {
        pixclock:       174757,         bpp:            1,
        xres:           160,
        yres:           160,
    sync:           0,              cmap_static:    1,
        reg : {
          lcdcon1 : LCD1_ENVID |LCD1_BPP_1S | LCD1_MMODE| LCD1_PNR_4S|
                    LCD1_CLKVAL(12) ,
           lcdcon2 : LCD2_VBPD(0) | LCD2_VFPD(0) | LCD2_VSPW(0),
           lcdcon3 : LCD3_WDLY_16,
           lcdcon4 : LCD4_WLH(0) | LCD4_MVAL(40),
           lcdcon5 : LCD5_BSWP | LCD5_PWREN,
        },
};
#endif
```

单色屏 bpp 改为 1

大小设置为 160×160

LCD1_ENVID：输出使能

LCD1_BPP_1S：单色屏，采用 STN 模式

LCD1_MMODE采用Mmode模式，由下设LCD4_MVAL(40)决定Vm rate，VM Rate = VLINE Rate / (2 * MVAL)，Vm信号用来驱动LCD改变行，列电压的极性。

LCD1_CLKVAL(12)表示lcd的时钟为60/((12+1)*2)Mhz

LCD2_VBPD(0) | LCD2_VFPD(0) | LCD2_VSPW(0)因为采用STN模式，这些TFT模式使用的寄存器不需要设置。

LCD3_WDLY_16：该寄存器用来设置VLINE和VCLK之间的延时，单位为HCLK。可选的值为16，32，64，128。

LCD4_WLH(0)：用来设置VLINE脉冲高电平的宽度，单位为HCLK，可选值为0，1，2，3对应16，32，48，64HCLK。

LCD5_BSWP：Byte swap control bit.

LCD5_PWREN：Half-Word swap control bit.

这两位用来控制字节交换和半字交换，主要用来大小头的问题，如果输出到屏上的汉字左右互换了，或者输出到屏上的图花屏了，可以更改这个选项。

在上述寄存器的设置中：
lcdcon1
lcdcon2
lcdcon3
lcdcon4
lcdcon5
分别表示 LCD 的 5 个控制寄存器，如果你要查看各个寄存器的详细说明，可以查看 2410 的 CPU 手册的 P397~P402.

如果您想进行其他的设置，请相信参照 CPU 手册的上述部分。

5．MiniGUI 免费版本的移植过程。

北京飞漫软件有限责任公司开发出适用于嵌入式 linux 的 MiniGUI，用户可以从 http://www.minigui.com/download/cindex.shtml 可以下载到 minigui 的源代码，当前的版本是 libminigui-1.3.3.tar.gz、minigui-res-1.3.3.tar.gz、mde-1.3.0.tar.gz、mg-samples-1.3.0.tar.gz。你也可以下载相应的手册。

在 pc 机的根目录下建立 minigui-free 目录，将下载的源代码包 copy 到该目录，解压缩，生成 libminigui-1.3.3、minigui-res-1.3.3、mde-1.3.0、mg-samples-1.3.1。我在该目录中建立 nfsroot 目录，用以存放生成的文件。

在 libminigui-1.3.3 目录中配置 lib

./configure --host=arm-unknown-linux --enable-jpgsupport=no --enable-pngsupport=no --enable-gifsupport=no --disable-lite --prefix=/minigui-free/nfsroot --enable-smdk2410ial=yes

修改configure，在文件开头处增加

########RJTECH##

CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc

CPP=/opt/host/armv4l/bin/armv4l-unknown-linux-cpp

LD=/opt/host/armv4l/bin/armv4l-unknown-linux-ld

AR=/opt/host/armv4l/bin/armv4l-unknown-linux-ar

RANLIB=/opt/host/armv4l/bin/armv4l-unknown-linux-ranlib

STRIP=/opt/host/armv4l/bin/armv4l-unknown-linux-strip

################RJTECH##

指明编译器，所以必须保证在你的pc机的根目录的/opt目录中存在上述编译器。

对lib中源文件的修改仅限于2410.c，它存在于src/ial目录中。集体细节参见其中含有RJtech标记的部分，只是告诉minigui确定lcd的设备名和如何从内核中读取触摸屏的坐标值。

然后，make，make install。

生成的文件在/minigui-free/nfsroot目录中，包括lib等目录

到/minigui-free/nfsroot目录中，删除minigui目录，因为该目录中的内容在res中都有

删除*.a 和 *.la文件，因为我们采用的动态连接，删除所有的静态链接库

执行/opt/host/armv4l/bin/armv4l-unknown-linux-strip *之后，库文件的大小应该在1M以内。

在mde-1.3.0目录中配置mde

./configure --build=i686-pc-linux-gnu --host=arm-unknown-linux --prefix=/minigui-free/nfsroot/ LDFLAGS=-L/minigui-free/nfsroot/lib CPPFLAGS=-I/minigui-free/nfsroot/include

CFLAGS=-I/minigui-free/nfsroot/include
修改configure，在文件开头处增加
########RJTECH##
CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc
CPP=/opt/host/armv4l/bin/armv4l-unknown-linux-cpp
LD=/opt/host/armv4l/bin/armv4l-unknown-linux-ld
AR=/opt/host/armv4l/bin/armv4l-unknown-linux-ar
RANLIB=/opt/host/armv4l/bin/armv4l-unknown-linux-ranlib
STRIP=/opt/host/armv4l/bin/armv4l-unknown-linux-strip
###############RJTECH##

make即可
我们采用的非lite模式，所以不需要执行mginit，所以这一步其实可以不必执行，后来也没有用到。

在minigui-res-1.3.3目录中配置res
修改config.linux文件，指明TOPDIR=/minigui-free/nfsroot。
Make install即可。

在mg-samples-1.3.1目录中配置mg－sample。
./configure --build=i686-pc-linux-gnu --host=arm-unknown-linux --prefix=/minigui-free/nfsroot/ LDFLAGS=-L/minigui-free/nfsroot/lib CPPFLAGS=-I/minigui-free/nfsroot/include CFLAGS=-I/minigui-free/nfsroot/include
修改configure，在文件开头处增加
########RJTECH##
CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc
CPP=/opt/host/armv4l/bin/armv4l-unknown-linux-cpp
LD=/opt/host/armv4l/bin/armv4l-unknown-linux-ld
AR=/opt/host/armv4l/bin/armv4l-unknown-linux-ar
RANLIB=/opt/host/armv4l/bin/armv4l-unknown-linux-ranlib
STRIP=/opt/host/armv4l/bin/armv4l-unknown-linux-strip
###############RJTECH##
make即可
生成的可执行文件在src目录中。

制作ramdisk，我制作了一个5M的ramdisk。
将/minigui-free/nfsroot/lib中所有的库文件copy到ramdisk的/lib中，将/minigui-free/nfsroot/usr/local/lib/minigui目录copy到ramdisk的/lib目录中。同时在ramdisk中建立/usr/local目录，在该目录中建立连接ln –s /lib lib.
Copy /minigui-free/MiniGUI.cfg文件到ramdiak中的/mnt/etc目录中。这个文件你们也可以在发给你们的早期的ramdisk中找到，在/mnt/etc目录，是minigui的配置文件。
Copy /minigui-free/mg-samples-1.3.1/src/treeview文件到ramdisk中的/bin目录，这个文件是minigui的演示程序。
将该ramdisk制作好以后，烧到板子上就可以了。
直接执行treeview，就可以看到minigui的例子。

6．MiniGUI源代码分析

MiniGUI的代码的核心部分在lib中，对lib的配置我们使用的是：

./configure --host=arm-unknown-linux --enable-jpgsupport=no --enable-pngsupport=no --enable-gifsupport=no --disable-lite --prefix=/minigui-free/nfsroot --enable-smdk2410ial=yes

--enable-jpgsupport=no：对jpg文件不支持是因为以前版本的lib中没有jpg的库，如果现在的用户手中取得的版本中已经存在jpg的库的话，是可以启动对jpg文件的支持的。

--disable-lite： 表示使用非lite模式，这是minigui默认的模式。这种模式使用pthread库，每个程序都启用一个新的线程来执行；相对的一种模式是lite模式，这种模式在每个程序启动之前都需要执行mg-init,在mg-init中完成所有的初始化操作，然后才可以执行用户的程序。

--prefix=/minigui-free/nfsroot：指明编译好的需要安装的文件放到系统的什么地方，在这里我们放在/minigui-free/nfsroot目录中。

--enable-smdk2410ial=yes：表明我们是使用smdk2410ial功能，这意味这对LCD的相关操作将对应src/ial/2410.c文件。我们对触摸屏的坐标读取就是对这个文件进行修改获得的。

该文件内容如下：

```
typedef struct {
    unsigned short pressure;
    unsigned short x;
    unsigned short y;
    unsigned short pad;
} TS_EVENT;
```

这个结构用来保存从触摸屏得到的坐标信息，x，y的大小。 Pressure用来表示用户当前是否有触摸笔在触摸屏上按下，如果按下，在该值大于0，否则等于0。当该值等于0的时候，x，y值没有意义，通常，x和y都等于0。

```
static unsigned char state [NR_KEYS];
static int ts = -1;
static int mousex = 0;
static int mousey = 0;
```

mousex和mousey用来保存当前从触摸屏取得的坐标x和y的值。

```
static TS_EVENT ts_event;

#undef _DEBUG

/************************  Low Level Input Operations **********************/
/*
 * Mouse operations -- Event
 */
static int mouse_update(void)
{
    return 1;
}

static void mouse_getxy(int *x, int* y)
{
```

```
#ifdef _DEBUG
    printf ("mousex = %d, mousey = %d\n", mousex, mousey);
#endif

    if (mousex < 0) mousex = 0;
    if (mousey < 0) mousey = 0;
    if (mousex > 239) mousex = 239;
    if (mousey > 319) mousey = 319;
```

这里是对mousex和mousey进行校验，这个是对240×320的小屏进行的校验，如果使用640×480的大屏，需要将上述坐标的限制改变239变为479，319变为630。以240×320的屏幕为例，屏幕x取值的变化范围为0~239，随意对该值做了上述的限制性设定。

```
    *x = mousex;
    *y = mousey;
}
static int mouse_getbutton(void)
{
    return ts_event.pressure;
}

#ifdef _LITE_VERSION
static int wait_event (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
                struct timeval *timeout)
#else
static int wait_event (int which, fd_set *in, fd_set *out, fd_set *except,
                struct timeval *timeout)
#endif
{
    fd_set rfds;
    int     retvalue = 0;
    int     e;

        // RJtech RJTECH modified it
    if (!in) {
        in = &rfds;
        FD_ZERO (in);
    }
```

第一次启动的时候，要初始化in变量。

```
    if ((which & IAL_MOUSEEVENT) && ts >= 0) {
        FD_SET (ts, in);
#ifdef _LITE_VERSION
        if (ts > maxfd) maxfd = ts;
#endif
    }
```

这里设置系统开始检测ts的信号，ts表示触摸屏的设备号。

```
    #ifdef _LITE_VERSION
        e = select (maxfd + 1, in, out, except, timeout) ;
    #else
        e = select (FD_SETSIZE, in, out, except, timeout) ;
    #endif

      if (e > 0)//这里e>0表示已经得到了触摸屏的信号。即触摸笔为按下状态
              {
             if (ts >= 0 && FD_ISSET (ts, in))
              {
                  FD_CLR (ts, in);
                   ts_event.x=0;
                   ts_event.y=0;
    初始化坐标值，用于当系统出错，或者没有读到坐标时的返回值。
                   read (ts, &ts_event, sizeof (TS_EVENT));
    读取坐标值。
                   if (ts_event.pressure > 0) {
                        mousex = ts_event.x;
                        mousey = ts_event.y;
                   }

    #ifdef _DEBUG
                   if (ts_event.pressure > 0) {
                        printf ("mouse down: ts_event.x = %d, ts_event.y = %d\n", ts_event.x,
ts_event.y);
                   }
    #endif

                   ts_event.pressure = ( ts_event.pressure > 0 ? 4:0);
                   retvalue |= IAL_MOUSEEVENT;
              设定是否按下的状态，读到坐标则返回正确的状态。
              }

        }
        else if (e < 0) {
             return -1;
        如果e<0,表示当前没有触摸屏鼠标事件发生
        }
    如果e＝0，则返回默认的retvalue，其中retvalue＝0。
        return retvalue;
    }

    BOOL Init2410Input (INPUT* input, const char* mdev, const char* mtype)
    {
```

```
        //rjtech RJTECH modified it
    ts = open ("/dev/touchscreen/0raw", O_RDONLY);
```

在初始化的时候，打开ts设备，即触摸屏设备。不同的操作平台这里应该不同，向不同的操作平台移植的时候清注意修改这里。

```
    if (ts < 0) {
        fprintf (stderr, "2410: Can not open touch screen!\n");
        return FALSE;
    }

    input->update_mouse = mouse_update;
    input->get_mouse_xy = mouse_getxy;
    input->set_mouse_xy = NULL;
    input->get_mouse_button = mouse_getbutton;
    input->set_mouse_range = NULL;

    input->wait_event = wait_event;
    mousex = 0;
    mousey = 0;
    ts_event.x = ts_event.y = ts_event.pressure = 0;
```

这里设定各种默认值，和各类函数的出世化。

```
    return TRUE;
}

void Term2410Input(void)
{
    if (ts >= 0)
        close(ts);
}
```

该文件结束。

在移植minigui的时候需要注意的领一个文件是./src/newgal/fbcon/fbvideo.c
在将minigui移植到mxl上的时候，该文件移植出错出错部分如下：

```
static int FB_EnterGraphicsMode (_THIS)
{
#ifdef _HAVE_TEXT_MODE
#if defined(_LITE_VERSION) && !defined(_STAND_ALONE)
    if (mgIsServer) {
#endif
        char* tty_dev;
        if (geteuid() == 0)
            tty_dev = "/dev/tty0";
        else      /* not a super user, so try to open the control terminal */
            tty_dev = "/dev/tty";
```

在这里设置系统的终端，在2410上采用的是以前的中断模式，即在/dev目录下存在tty0和tty

两个设备，但是在mxl上采用了新的devfs文件系统以后，就只有tty这一个设备了，这部分移植的时候，系统的tty设备不支持现面的ioctl，所以会出错，导致初始化失败。

```
        /* open tty, enter graphics mode */
        ttyfd = open (tty_dev, O_RDWR);
        if (ttyfd < 0) {
            fprintf (stderr,"GAL ENGINE: Can't open %s: %m\n", tty_dev);
            goto fail;
        }
if (ioctl (ttyfd, KDSETMODE, KD_GRAPHICS) == -1) {
            fprintf (stderr,"GAL ENGINE: Error when setting the terminal to graphics mode:
  %m\n");
            fprintf (stderr,"GAL ENGINE: Maybe is not a console.\n");
            goto fail;
```

在mxl的这部分就会ioctl失败，从而导致该函数执行错误，minigui初始化失败而使应用程序直接退出。我们在移植的时候注释掉了这部分代码，程序就可以顺利运行了。

```
            }
    #if defined(_LITE_VERSION) && !defined(_STAND_ALONE)
        }
        else {
            ttyfd = 0;
        }
    #endif

        return ttyfd;

    fail:
        FB_LeaveGraphicsMode (this);
        return -1;

    #else
        return 0;
    #endif
    }
```

三．实验内容和步骤

1． LCD 实验：更改 240×320－－》640×480LCD 屏的驱动

(1) 在板子的 ppcboot 模式下，使用 tftp，fl 等命令烧写 240×320LCD 的内核和 ramdisk。

```
tftp 30008000 zImage
fl 1040000 30008000 e0000
tftp 30800000 ramdisk.image.gz
fl 1140000 30800000 220000
```

(2) 插上 240×320 的 LCD，观察屏幕的正常显示。

修改驱动 driver/video/s3c2410fb.c 文件

xres:640
yres:480
reg 改为：

```
reg : {
    lcdcon1 : LCD1_BPP_16T | LCD1_PNR_TFT | LCD1_CLKVAL(1) ,
    lcdcon2 : LCD2_VBPD(60) | LCD2_VFPD(60) | LCD2_VSPW(20),
    lcdcon3 : LCD3_HBPD(48) | LCD3_HFPD(6),
    lcdcon4 : LCD4_HSPW(96) | LCD4_MVAL(13),
    lcdcon5 : LCD5_FRM565 | LCD5_INVVLINE | LCD5_INVVFRAME |
        LCD5_HWSWP | LCD5_PWREN,
     },
```

(3) 在 kernel 目录下 make zImage，程序会自动将 zImage copy 到/tftpboot 目录
(4) 在板子的 ppcboot 方式下执行：

```
tftp 30008000 zImage
fl 1040000 30008000 e0000
tftp 30800000 ramdisk.image.gz   //烧写新的用于大屏的 ramdisk
fl 1140000 30800000 220000
```

(5) 断电，去掉 240×320 的小 LCD 屏，换上 640×480 的大 LCD 屏。
(6) 重新启动板子，观察 640×480 的大屏的正确显示结果，显示正确表示大屏的驱动移植成功。

2. MiniGUI 应用程序的编写
(1) 在 mg-sample-1.3.1 目录下执行 make，可执行程序将自动生成在 src 目录下。
(2) 在板子的操作系统上 mount pc 机的这个目录

```
mount –o nolock 192.168.2.**:/
    /RJtek2410-1/application/minigui/mg-sample-1.3.1/src   /mnt
```

(3) 在板子上执行/mnt 目录下的各个可执行程序，观察 minigui 在 LCD 上的显示效果。
(4) 修改 mg-sample-1.3.1/src 下的源程序，观察 LCD 上的显示变化。

四．思考题

1. 在屏幕上写汉字显示和 CPU 大小头之间的关系，分析如果汉字左右写反了可能存在的原因。
2. 不同 BPP LCD 屏的坐标定位方法有何不同。

实验十八　图形界面设计实验

一. 实验目的

通过本实验，使学生掌握 Linux 下图形界面的设计，了解图形界面的创建、显示过程。

二. 实验原理和说明

S3C2410 内置 LCD 控制器，它支持通用 LCD 液晶显示屏，开发板支持彩色显示的点阵大小为：320×240。RJtek2410-1 开发板提供的液晶屏上附有一层触摸屏膜，它们与板子通过一个 20 线的排线插座相连接。见图 18_1。

锐极 RJtek2410-1 开发平台提供的 GUI API 仿照 WIN32 API 的接口，使客户能够以最短的时间熟悉并使用它们，实现从 WINDOWS 平台到 LINUX 平台开发者的角色转变。

在/ RJtek2410-1/application/microwin 目录下，有很多基于 micro windows 的 DEMO 样例程序可供参考和演示，可供选择。

1. S3C2410 的 LCD 驱动配置。

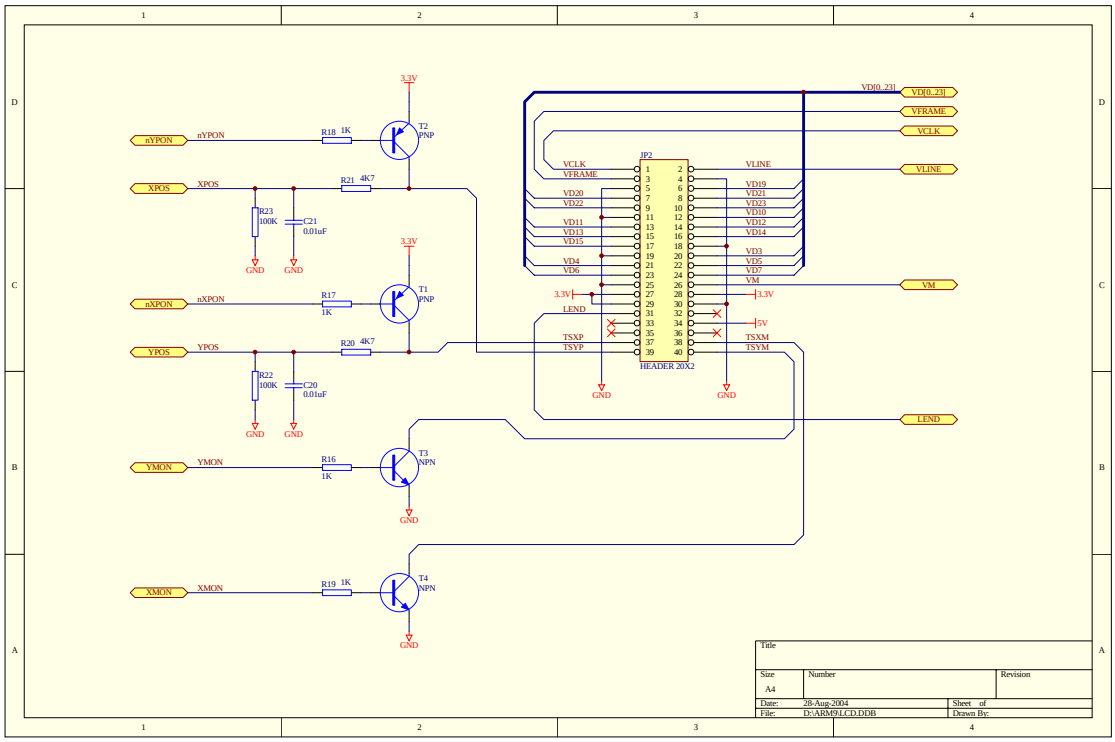


图 18_1　RJARM2410-R3 开发板 LCD 和触摸屏引脚图

S3C2410MCU内置了可编程LCD控制模块。

它支持STN LCD和TFT LCD两个系列的显示屏，包括单色16级灰度的单色和64K种颜色的彩色。屏幕可以是160*160、240*320、640*480、2048*1024等多种尺寸。支持一个象素可以用 1、2、4、8、16、24或32等bit来描述，通过用户对LCD控制器的编程来选择其中的一种摸式，以便和LCD屏匹配。RJARM2410-R3开发扳采用的是TFT LCD屏 16 bit 无调色板彩色工作摸式，图18_2是32位数据字在屏幕上象素的映射关系。RGB彩色描述采用的是5：6：5的比例位模式。

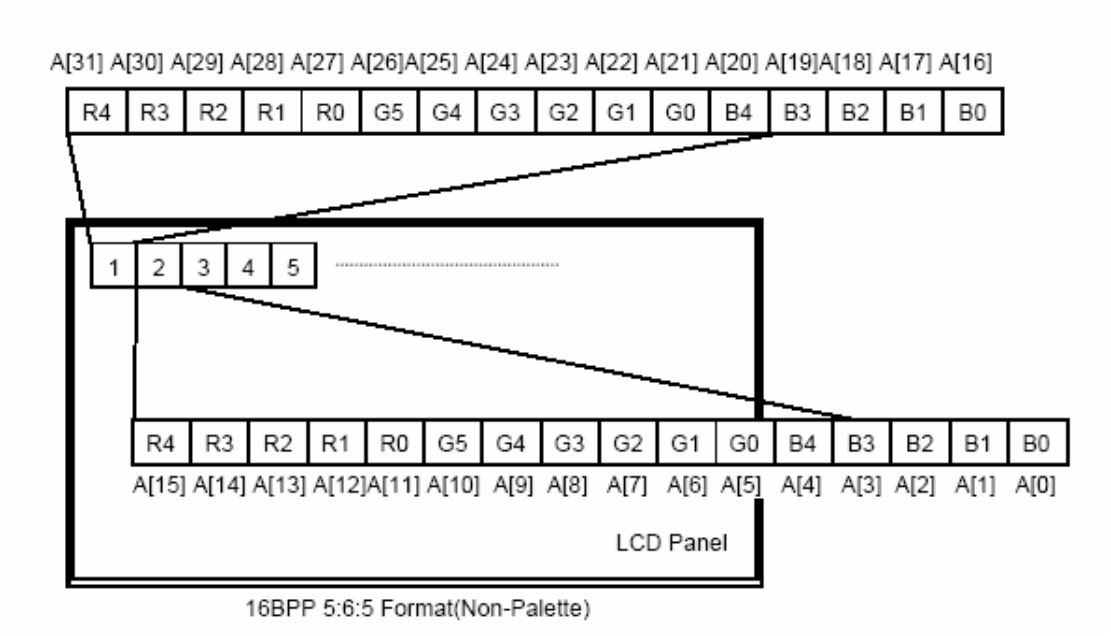


图18_2 32位数据字在屏幕上象素的映射关系

图18_3 是16位彩色调色板工作模式中调色板在内存地址空间的映射关系，它可以有256种颜色。这里要注意的是RJARM2410-R3开发板上LCD的引脚图中，LCD数据总线VD[23：19]表述Red色，VD[15：10]，VD[15：10]表述Green色，VD[7：3]表述Blue色。VRAME引脚是桢同步信号，VLINE引脚是行同步信号，VCLK引脚是象素同步信号，VM引脚是数据使能同步信号，LEND引脚是行结束信号。

Table 15-4. 5:6:5 Format

INDEX\Bit Pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Address
00H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	1)0X4D000400
01H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000404
.....																	.....
FFH	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D0007FC
Number of VD	23	22	21	20	19	15	14	13	12	11	10	7	6	5	4	3	

图18_3 16位调色板工作模式中调色板在内存地址空间的映射关系

S3C2410 可以在内存中开辟一个LCD 显示缓冲区，可以不断从VD[23，0]送出象素信号，和显示同步信号，大大简化了LCD 液晶屏的外围电路。

2. BMP图形文件格式

嵌入式LCD显示的原理，是对frame buffer即帧缓冲的内容进行改写，把frame buffer的内容写入显示缓冲区中，进行LCD显示。为了了解LCD显示文字和BMP图象，下面说明BMP文件的基本结构。

BMP(Bitmap-File)图形文件是Windows采用的图形文件格式，在Windows环境下运行的所图象处理软件都支持BMP图象文件格式。Windows系统内部各图像绘制操作都是以BMP为基础的。Windows 3.0以前的BMP图文件格式与显示设备有关，因此把这种BMP图象文件格式称为设备相关位图DDB(device-dependent bitmap)文件格式。Windows 3.0以后的BMP图象文件与显示设备无关，因此把这种BMP图象文件格式称为设备无关位图DIB(**d**evice-**i**ndependent **b**itmap)格式（注：Windows 3.0以后，在系统中仍然存在DDB位图，象BitBlt()这种函数就是基于DDB位图的，只不过如果你想将图

像以BMP格式保存到磁盘文件中时，微软极力推荐你以DIB格式保存），目的是为了让Windows能够在任何类型的显示设备上显示所存储的图象。BMP位图文件默认的文件扩展名是BMP或者bmp（有时它也会以.DIB或.RLE作扩展名）。

位图文件可看成由4个部分组成：位图文件头(bitmap-file header)、位图信息头(bitmap-information header)、彩色表(color table)和定义位图的字节阵列，它具有如下所示的形式：

位图文件的组成	结构名称	符号
位图文件头(bitmap-file header)	BITMAPFILEHEADER	bmfh
位图信息头(bitmap-information header)	BITMAPINFOHEADER	bmih
彩色表(color table)	RGBQUAD	aColors[]
图象数据阵列字节	BYTE	aBitmapBits[]

位图文件结构可综合在表 01 中。

表 01 位图文件结构内容摘要

	偏移量	域的名称	大小	内容
图象文件头	0000h	文件标识	2 bytes	两字节的内容用来识别位图的类型： ‘BM’ ： Windows 3.1x, 95, NT, … ‘BA’ ：OS/2 Bitmap Array ‘CI’ ：OS/2 Color Icon ‘CP’ ：OS/2 Color Pointer ‘IC’ ： OS/2 Icon ‘PT’ ：OS/2 Pointer 注：因为 OS/2 系统并没有被普及开，所以在编程时，你只需判断第一个标识“BM”就行。
	0002h	File Size	1 dword	用字节表示的整个文件的大小
	0006h	Reserved	1 dword	保留，必须设置为 0
	000Ah	Bitmap Data Offset	1 dword	从文件开始到位图数据开始之间的数据(bitmap data)之间的偏移量
	000Eh	Bitmap Header Size	1 dword	位图信息头(Bitmap Info Header)的长度，用来描述位图的颜色、压缩方法等。下面的长度表示： 28h - Windows 3.1x, 95, NT, … 0Ch - OS/2 1.x F0h - OS/2 2.x 注：在 Windows95、98、2000 等操作系统中，位图信息头的长度并不一定是 28h，因为微软已经制定出了新的 BMP 文件格式，其中的信息头结构变化比较大，长度加长。

				所以最好不要直接使用常数 28h，而是应该从具体的文件中读取这个值。这样才能确保程序的兼容性。
	0012h	Width	1 dword	位图的宽度，以象素为单位
	0016h	Height	1 dword	位图的高度，以象素为单位
	001Ah	Planes	1 word	位图的位面数（注：该值将总是 1）
图象信息头	001Ch	Bits Per Pixel	1 word	每个象素的位数 1 - 单色位图（实际上可有两种颜色，缺省情况下是黑色和白色。你可以自己定义这两种颜色） 4 - 16 色位图 8 - 256 色位图 16 - 16bit 真彩色位图 24 - 24bit 真彩色位图 32 - 32bit 增强型真彩色位图
	001Eh	Compression	1 dword	压缩说明： 0 - 不压缩 (使用 BI_RGB 表示) 1 - RLE 8-使用 8 位 RLE 压缩方式(用 BI_RLE8 表示) 2 - RLE 4-使用 4 位 RLE 压缩方式(用 BI_RLE4 表示) 3 - Bitfields- 位域存放方式(用 BI_BITFIELDS 表示)

(1) 位图文件头

位图文件头包含有关于文件类型、文件大小、存放位置等信息，在 Windows 3.0 以上版本的位图文件中用 BITMAPFILEHEADER 结构来定义：

```
typedef struct tagBITMAPFILEHEADER { /* bmfh */
    UINT bfType;
    DWORD bfSize;
    UINT bfReserved1;
    UINT bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER;
```

其中：

bfType	说明文件的类型.（该值必需是 0x4D42，也是字符'BM'。我们不需要判断 OS/2 的位图标识，这么做现在来看似乎已经没有什么意义了，而且如果要支持 OS/2 的位图，程序将变得很繁琐。所以，在此只建议你检察'BM'标识）
bfSize	说明文件的大小，用字节为单位
bfReserved1	保留，必须设置为 0

bfReserved2	保留，必须设置为 0
bfOffBits	说明从文件头开始到实际的图象数据之间的字节的偏移量。这个参数是非常有用的，因为位图信息头和调色板的长度会根据不同情况而变化，所以你可以用这个偏移值迅速的从文件中读取到位数据。

(2) 位图信息头

位图信息用 BITMAPINFO 结构来定义，它由位图信息头(bitmap-information header)和彩色表(color table)组成，前者用 BITMAPINFOHEADER 结构定义，后者用 RGBQUAD 结构定义。BITMAPINFO 结构具有如下形式：

```
typedef struct tagBITMAPINFO { /* bmi */
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1];
} BITMAPINFO;
```

其中：

bmiHeader	说明 BITMAPINFOHEADER 结构，其中包含了有关位图的尺寸及位格式等信息
bmiColors	说明彩色表 RGBQUAD 结构的阵列，其中包含索引图像的真实 RGB 值。

BITMAPINFOHEADER 结构包含有位图文件的大小、压缩类型和颜色格式，其结构定义为：

```
typedef struct tagBITMAPINFOHEADER { /* bmih */
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

其中：

biSize	说明 BITMAPINFOHEADER 结构所需要的字数。注：这个值并不一定是 BITMAPINFOHEADER 结构的尺寸，它也可能是 sizeof(BITMAPV4HEADER)的值，或是 sizeof(BITMAPV5HEADER)的值。这要根据该位图文件的格式版本来决定，不过，就现在的情况来看，绝大多数的 BMP 图像都是 BITMAPINFOHEADER 结构的（可能是后两者太新的缘故吧:-）。
biWidth	说明图象的宽度，以象素为单位

biHeight	说明图象的高度，以象素为单位。注：这个值除了用于描述图像的高度之外，它还有另一个用处，就是指明该图像是倒向的位图，还是正向的位图。如果该值是一个正数，说明图像是倒向的，如果该值是一个负数，则说明图像是正向的。大多数的 BMP 文件都是倒向的位图，也就是时，高度值是一个正数。（注：当高度值是一个负数时（正向图像），图像将不能被压缩（也就是说 biCompression 成员将不能是 BI_RLE8 或 BI_RLE4）。
biPlanes	为目标设备说明位面数，其值将总是被设为 1
biBitCount	说明比特数/象素，其值为 1、4、8、16、24、或 32
biCompression	说明图象数据压缩的类型。其值可以是下述值之一： BI_RGB：没有压缩； BI_RLE8：每个象素 8 比特的 RLE 压缩编码，压缩格式由 2 字节组成(重复象素计数和颜色索引)； BI_RLE4：每个象素 4 比特的 RLE 压缩编码，压缩格式由 2 字节组成 BI_BITFIELDS：每个象素的比特由指定的掩码决定。
biSizeImage	说明图象的大小，以字节为单位。当用 BI_RGB 格式时，可设置为 0
BiXPelsPerMeter	说明水平分辨率，用象素/米表示
biYPelsPerMeter	说明垂直分辨率，用象素/米表示
biClrUsed	说明位图实际使用的彩色表中的颜色索引数（设为 0 的话，则说明使用所有调色板项）
biClrImportant	说明对图象显示有重要影响的颜色索引的数目，如果是 0，表示都重要。

现就 BITMAPINFOHEADER 结构作如下说明：

(a) 彩色表的定位

应用程序可使用存储在 biSize 成员中的信息来查找在 BITMAPINFO 结构中的彩色表，如下所示：

pColor = ((LPSTR) pBitmapInfo + (WORD)(pBitmapInfo->bmiHeader.biSize))

(b) biBitCount

biBitCount=1 表示位图最多有两种颜色，缺省情况下是黑色和白色，你也可以自己定义这两种颜色。图像信息头装调色板中将有两个调色板项，称为索引 0 和索引 1。图象数据阵列中的每一位表示一个象素。如果一个位是 0，显示时就使用索引 0 的 RGB 值，如果位是 1，则使用索引 1 的 RGB 值。

biBitCount=4 表示位图最多有 16 种颜色。每个象素用 4 位表示，并用这 4 位作为彩色表的表项来查找该象素的颜色。例如，如果位图中的第一个字节为 0x1F，它表示有两个象素，第一象素的颜色就在彩色表的第 2 表项中查找，而第二个象素的颜色就在彩色表的第 16 表项中查找。此时，调色板中缺省情况下会有 16 个 RGB 项。对应于索引 0 到索引 15。

biBitCount=8 表示位图最多有 256 种颜色。每个象素用 8 位表示，并用这 8 位作为彩色表的表项来查找该象素的颜色。例如，如果位图中的第一个字节为 0x1F，这个象素的颜色就在彩色表的第 32 表项中查找。此时，缺省情况下，调色板中会有 256 个 RGB 项，对应于索引 0 到索引 255。

biBitCount=16 表示位图最多有 2^{16} 种颜色。每个色素用 16 位（2 个字节）表示。这种格式叫作高彩色，或叫增强型 16 位色，或 64K 色。它的情况比较复杂，当 biCompression 成员的值是 BI_RGB 时，它没有调色板。16 位中，最低的 5 位表示蓝色分量，中间的 5 位表示绿色分量，高的 5 位表示红色分量，一共占用了 15 位，最高的一位保留，设为 0。这种格式也被称作 555 16 位位图。如果 biCompression 成员的值是 BI_BITFIELDS，那么情况就复杂了，首先是原来调色板的位置被三个 DWORD 变量占据，称为红、绿、蓝掩码。分别用于描述红、绿、蓝分量在 16 位中所占的位置。在 Windows 95（或 98）中，系统可接受两种格式的位域：555 和 565，在 555 格式下，

红、绿、蓝的掩码分别是：0x7C00、0x03E0、0x001F，而在 565 格式下，它们则分别为：0xF800、0x07E0、0x001F。你在读取一个像素之后，可以分别用掩码“与”上像素值，从而提取出想要的颜色分量（当然还要再经过适当的左右移操作）。在 NT 系统中，则没有格式限制，只不过要求掩码之间不能有重叠。（注：这种格式的图像使用起来是比较麻烦的，不过因为它的显示效果接近于真彩，而图像数据又比真彩图像小的多，所以，它更多的被用于游戏软件）。

biBitCount=24 表示位图最多有 2^{24} 种颜色。这种位图没有调色板（bmiColors 成员尺寸为 0），在位数组中，每 3 个字节代表一个象素，分别对应于颜色 R、G、B。

biBitCount=32 表示位图最多有 2^{32} 种颜色。这种位图的结构与 16 位位图结构非常类似，当 biCompression 成员的值是 BI_RGB 时，它也没有调色板，32 位中有 24 位用于存放 RGB 值，顺序是：最高位—保留，红 8 位、绿 8 位、蓝 8 位。这种格式也被成为 888 32 位图。如果 biCompression 成员的值是 BI_BITFIELDS 时，原来调色板的位置将被三个 DWORD 变量占据，成为红、绿、蓝掩码，分别用于描述红、绿、蓝分量在 32 位中所占的位置。在 Windows 95(or 98)中，系统只接受 888 格式，也就是说三个掩码的值将只能是：0xFF0000、0xFF00、0xFF。而在 NT 系统中，你只要注意使掩码之间不产生重叠就行。（注：这种图像格式比较规整，因为它是 DWORD 对齐的，所以在内存中进行图像处理时可进行汇编级的代码优化（简单））。

(c) ClrUsed

BITMAPINFOHEADER 结构中的成员 ClrUsed 指定实际使用的颜色数目。如果 ClrUsed 设置成 0，位图使用的颜色数目就等于 biBitCount 成员中的数目。请注意，如果 ClrUsed 的值不是可用颜色的最大值或不是 0，则在编程时应该注意调色板尺寸的计算，比如在 4 位位图中，调色板的缺省尺寸应该是 16*sizeof(RGBQUAD)，但是，如果 ClrUsed 的值不是 16 或者不是 0，那么调色板的尺寸就应该是 ClrUsed*sizeof(RGBQUAD)。

了解了 BMP 在 WINDOW 下的格式之后，用 VC++(以二进制)打开查看里面的数据，里面包含图象文件头和图象信息头(包含 BMP 文件的宽度，高度及用 biBitCount 控制的文件的具体格式如黑白，16 位，24 彩图)以及实际数据等等。

设计一结构体存放 BMP 文件的信息头，结构体内容与图象文件头和图象信息头相对应如下：

```
typedef struct{
    short bfType; //__attribute__((packed));
    long   bfSize; //__attribute__((packed));
    short bfReserved1;
    short bfReserved2;
    long bfOffBits;
    long biSize;
    long biWidth;
    long biHeight;
    short biPlanes;
    short biBitCount;
    long biCompression;
    long biSizeImage;
    long biXpp;
    long biYpp;
    long biClrUsed;
    long biClrImportant;
}BMPHEAD;
```

该结构体中的总字节数为 54 个字节。由于 long 表示四个字节，则对应搜索以 4 为倍数的地址申请。对实际数据操作时 bfType 没用到，为了能使图象文件头和图象信息头里的关键参数如宽度等能与结构里的一一对应，采用如下语句：

fseek(fp,2L,0); (作用去掉 bfType2 个字节,从 bfSize 开始读入)

fread(&bmp_inf.bfSize,1,52,fp);

把 fp（对应的 BMP）的信息(从 bfSize 开始)放入结构体 bmp_inf 内，接着就可以读取该 BMP 的文件信息，如文件长度 biWidth（以象素为单位），宽度 biHeight（以象素为单位），biBitCount(控制图象的格式)

3. 编写程序步骤如下：

(1) 判断 biBitCount 是否等于 1，16 或 24

注意事项：由于 BMP 文件在每行都以 4 字节的倍数进行结尾，也就是说当宽度（以象素为单位）/8（对 biBitCount=1）转换为字节后的字节数要是 4 的倍数，若不是则补 0，补满为止。全部的 BMP 文件，包括 16 与 24 位都是以 4 字节的倍数结尾的。

(a) 若为 1，表示该 BMP 为黑白图片

若不满 4 的倍数的，控制补 0 的语句如下：aver_size=((bmp_inf.biWidth+15)/16)*2; (由于 1 位表示一个象素，所以申请缓冲区(以字节为单位)，除 8)。

由于图像信息头带有调色板，8 个字节，所以 BMP 图象的实际数据为从 BMP 开始地址的第 63 字节，编写语句如下：

fseek(fp,62L,0);

(b) 若为 16,表示该 BMP 为 16 位的彩图

若不满 4 的倍数的，控制补 0 的语句如下：

aver_size=(bmp_inf.biWidth*2+3)/4*4;(2 个字节表示一个象素)

该 BMP 文件不带调色板，实际数据的定位如下：

fseek(fp,54L,0);

(c) 若为 24,表示该 BMP 为 24 位的彩图

若不满 4 的倍数的，控制补 0 的语句如下：

aver_size=(bmp_inf.biWidth*3+3)/4*4;(3 个字节表示一个象素)

24 位 BMP 文件信息头中同样不带调色板，则实际数据定位如下：

fseek(fp,54L,0);

(2) 找到结构体中关于文件的宽度和高度，申请缓冲区大小。

(a) biBitCount=1，图象数据阵列中的每一位表示一个象素。

语句如下：

size=aver_size*bmp_inf.biHeight; (aver_size 参照如上)

buf=(char*)malloc(size);

(b) biBitCount=16

语句如下：

size=bmp_inf.biWidth*bmp_inf.biHeight*2;（由于是 16 位，则字节数*2）

buf=(char*)malloc(size);

(c) biBitCount=24

语句如下：

size=bmp_inf.biHeight*((bmp_inf.biWidth*3+3)/4*4); （由于是 24 位，则字节数*3）

buf=(char*)malloc(size);

```
buf1=(char*)malloc(bmp_inf.biWidth*bmp_inf.biHeight*2);
```

由于只支持 16 位显示所以要把 24 位转换成 16 位，转换语句如下：

```
for(k=i*((bmp_inf.biWidth*3+3)/4)*4,m=i*bmp_inf.biWidth*2,j=0;j<bmp_inf.biWidth;j++)
  { red=((float)(buf[k+j*3+2]))/255*31+0.5;
    green=((float)(buf[k+j*3+1]))/255*63+0.5;
    blue=((float)(buf[k+j*3]))/255*31+0.5;
   *(unsigned short*)(buf1+m+j*2)=(red<<11)|(green<<5)|(blue);
   }
```

转换原则如下：按各分量的大小所占的比例进行，即 24 位中 R(红色分量)实际大小所占总分量(8 位)的比例，与 16 位中 R(红色分量)实际大小所占总分量(5 位)的比例等价，其他分量也一样，G(绿色分量)8 位转换成 6 位，B(蓝色分量)8 位转换成 5 位。

(3) 对 BMP 存放的缓冲区里的内容进行调整，由于 BMP 实际数据存放的格式是 Height 是上下反向的，即第一行显示的内容存放在最后一行

转换成实际显示的内容的语句如下：

```
for(i=0;i<(bmp_inf.biHeight>>1);i++)
     for(k=i*aver_size,j=0;j<aver_size;j++)
         {
           Tmp=buf[k+j];
           buf[k+j]=buf[size-k-aver_size+j];
           buf[size-k-aver_size+j]=Tmp;
}
```

对 biBitCount=1，16 或 24 位的都根据这个原则进行编写。

(4) 对调整过后的缓冲区内的内容进行显示调用的函数如下：

(1) biBitCount=1

因为图象数据阵列中的每一位表示一个象素，则调用的子函数如下：

```
void my_draw_bmp(short x,short y,unsigned short width,unsigned short height,char *pixel)
{short i,j;
 long aver_size=((width+31)/32)*4;
 char masktab[8]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
 for(j=0;(y+j)<ScreenHeight&&(y+j)>=0&&j<height;j++)
  for(i=0;(x+i)<ScreenWidth&&(x+i)>=0&&i<width;i++)
  if(*(pixel+j*aver_size+i/8)        &        masktab[i%8])//judge        pixel        "0"or"1";
{*(screen_ptr+(y+j)*ScreenWidth*2+2*(i+x))=0x00;
*(screen_ptr+(y+j)*ScreenWidth*2+2*(i+x)+1)=0x00;
                    }
 else{
*(screen_ptr+(y+j)*ScreenWidth*2+2*(i+x))=0xff;
        *(screen_ptr+(y+j)*ScreenWidth*2+2*(i+x)+1)=0xff;
}
}
```

(2) biBitCount=16

因为图象数据阵列中的每两个字节表示一个象素，则调用的子函数如下：

```
void color_565_draw_bmp(short x,short y,unsigned short width,unsigned short height,char
*buf)
{    long i,j,offset;
     unsigned short tmp,red,green,blue;
     for(j=y;j<ScreenHeight&&(j-y)<height;j++)
          for(i=x;i<ScreenWidth&&(i-x)<width;i++)
              { offset=(j-y)*width*2+(i-x)*2;
                tmp=*(unsigned short*)(buf+offset);
                *(unsigned short*)(screen_ptr+j*ScreenWidth*2+2*i)=tmp;
               }
}
```

(3) biBitCount=24

因为通过转换后 24 位已经转换成 16 位，存放在缓冲区内，所以跟 16 位的显示相差不大，即把 16 位的信息直接对应地放入进行象素控制的 16 位中，调用的子函数与 16 位彩图显示一样，调用 color_565_draw_bmp(short x,short y,unsigned short width,unsigned short height,char *buf) 函数。

4. 图形界面（GUI）接口函数 API

在 graphic.h 和 graphic.c 中，存在以下函数：

short initgraph(void)

初始化显示环境,返回结果表示初始化是否成功；

void closegraph(void)

关闭显示环境；

void clearscreen(void)

清屏。

void setpixel(short x,short y,short color)

在（x,y）坐标处画点；

short getpixel(short x,short y)

返回（x,y）点的颜色信息；

void setmode(CopyMode mode)

设置显示模式。

参数mode 可为下值：

MODE_SRC 从源到目的COPY；

MODE_NOT_SRC 目的为源的反；

MODE_SRC_OR_DST 目的为与源相或后的结果；

MODE_SRC_AND_DST 目的为与源相与后的结果；

MODE_SRC_XOR_DST 目的为与源相异或后的结果；

MODE_SRC_OR_NOT_DST 目的为与源的反相或后的结果；

MODE_SRC_AND_NOT_DST 目的为目的的反与源相与后的结果；

MODE_SRC_XOR_NOT_DST 目的为目的的反与源相异或后的结果；
MODE_NOT_SRC_OR_DST 目的为与源的反相或后的结果；
MODE_NOT_SRC_AND_DST 目的为与源相与后的结果；
MODE_NOT_SRC_XOR_DST 目的为与源的反相异或后的结果；
MODE_NOT_SRC_OR_NOT_DST 目的为目的的反与源的反相或后的结果；
MODE_NOT_SRC_AND_NOT_DST 目的为目的的反与源的反相与后的结果；
MODE_NOT_SRC_XOR_NOT_DST 目的为目的的反与源的反相异或后的结果。

CopyMode getmode(void);
获取当前显示模式。

void setcolor(short color)
设置显示前景色。

UINT getcolor(void)
获取当前显示前景色。

void setfillpattern(PatternIndex index)
设置填充模式。

PatternIndex getfillpattern(void)
获取当前填充模式。

void bar(short x1,short y1,short x2,short y2)
以实填充模式在（x1,y1,x2,y2）绘制矩形。

void ellipse(short x1,short y1,short x2,short y2)
在矩形（x1,y1,x2,y2）中绘制椭圆。

void line(short x1, short y1, short x2,short y2)
从点（x1,y1）到点（x2,y2）画一条直线。
void lineto(short x, short y)
从当前所在点到点（x,y）之间画一条直线。

void moveto(short x,short y) 设置当前所在点为点（x,y）。

void rectangle(short x1,short y1,short x2,short y2)
在（x1,y1,x2,y2）中绘制矩形。

void textout(short x,short y,unsigned char *s)
在（x,y）坐标处输出字符串 s。

void bitblt(short src_x, short src_y, short w, short h, short dest_x, short dest_y,unsigned char *src, short src_units_per_line, unsigned char *dest, short dest_units_per_line)

位块传送，源地址(src_x,src_y),宽度w,高度h，目标地址(dest_x,dest_y)，源块的数据指针src，src_units_per_line 指明了数据源中每行的宽度，dest指明了目的数据地址，dest_units_per_line 指明了目的地址方的每行的宽度。

void ShowBMP(char *filename,short x,short y)
在x，y 处显示位图。

void draw_bmp(short sx, short sy, short rwidth, short height, char* pixel)
位图显示函数在指定的sx,sy 处显示每行逻辑宽度为rwidth 字节的位图，pixel 中指定了位图的图象信息，通常的使用可以参考ShowBMP 中的使用。

void V_scroll_screen(short height)
竖直方向滚动屏幕，以height 为单位。height>0，屏幕上滚；height<0，屏幕下滚。
void H_scroll_screen(short height)
水平方向滚动屏幕，以height 为单位。height>0，屏幕向左滚动；height<0，屏幕向右滚动。

5. GUI 应用程序

(1) gui应用程序在宿主机 /Rjtek2410-1/application/gui 目录下，可以参考相应的源代码，修改或编写自己的嵌入式应用程序。

图形界面的创建可以直接由photoshop、firework、GIMP之类创建，也可以由抓屏粘贴获得，但必须转化格式到液晶屏支持的类型（我们使用的是16位的RGB位图，注意：如果是索引BMP，需要将图像模式转化到RGB）。

(2) 然后可以调用ShowBMP直接显示图形图像，也可以将图形图像文件先读入缓冲区，再调用ShowBuf函数显示图形图像，这样做的好处是显示速度比使用ShowBMP快很多，因为已经读入到缓冲区了。下面给出这几个函数的使用样例，仅供参考。

(a) ShowBMP（）方式

```
/* gui.c: Graphics demos
 *
 *    Programmed By Chen: Yang (support@ruijitek.com)
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
#include <stdio.h>      /* Standard input/output definitions */
#include <string.h>    /* String function definitions */
#include <unistd.h>    /* UNIX standard function definitions */
#include <fcntl.h>     /* File control definitions */
#include <errno.h>      /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
#include<sys/types.h>
```

```
#include<sys/stat.h>
#include <stdlib.h>

#include "gui.h"
void delay()
{int i,j;

 for(i=0;i<1000;i++)
   for(j=0;j<20000;j++) {}
 }
main(int argc,char*argv[])
{
    short i,j,w,h;
    struct stat st;
    int color0,filelength;
 PatternIndex p1=BlackPattern;
    char buf[20]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
    char buf1[700000];
    short   *o;int *p;long *q;char *c;
    FILE *fp;
   initgraph();
   if((fp=fopen("test3.bmp","r"))==NULL)
            {printf("Can't find the file");
              return 1;
            }
   stat("test3.bmp",&st);
   filelength=st.st_size;
   clearscreen();
   fillrect(10,10,100,100);
textout(0,130,"中国计量学院嵌入式系统实验室\n 热烈欢迎各位同学、各位朋友！\n",0xf800,0x0000);
   delay();
   delay();
   textout(0,30,"中国计量学院\n在美丽的杭州热烈欢迎你们！ \n",0xf900,0x0000);
 delay();
 delay();
   clearscreen();
   fread(buf1,1,filelength,fp);
   ShowBuf(buf1,0,0);
ShowBMP(" test3.bmp",0,0);
delay();

clearscreen();
ShowBMP("test4.bmp",0,0);
delay();
```

```
delay();
clearscreen();
ShowBMP("12.bmp",0,0);
}
```

该程序用到二个主要函数，textout()和ShowBMP()，它们的函数在graphic.c 中定义。

(b) ShowBuf方式

以下程序可以改用ShowBMP函数实现，该函数使用很方便，但显示时没有ShowBuf流畅！因为读取比较大的位图需要一段时间，改用ShowBMP时会造成一段时间的响应停滞（严重的时候可能会导致几秒钟的黑屏）。

```
/* gui.c: Graphics demos
 *
 *    Programmed By Chen: Yang (support@ruijitek.com)
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
#include <stdio.h>      /* Standard input/output definitions */
#include <string.h>    /* String function definitions */
#include <unistd.h>    /* UNIX standard function definitions */
#include <fcntl.h>     /* File control definitions */
#include <errno.h>      /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
#include<sys/types.h>
#include<sys/stat.h>
#include <stdlib.h>
#include "gui.h"
void delay()          /*延时函数*/
{
      int i,j;

      for(i=0;i<1000;i++)
         for(j=0;j<7000;j++) {}
}
main(int argc,char*argv[])
{
        short i,j,w,h;
        struct stat st;
        int color0,filelength;
        PatternIndex p1=BlackPattern;
```

```
 /*定义缓冲区，并初始化*/
 char buf[20]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
 char buf1[700000];
 short   *o,*c;
 int *p;
 long *q;

 FILE *fp;

/*初始化液晶屏*/
initgraph();
clearscreen();

/*画出方框*/
fillrect(10,10,100,100);

/*在指定的坐标出显出彩色文本*/
textout(0,130,"中国计量学院\n\nA Student",0xf800,0x0000);

/*以只读方式打开文件*/
if((fp=fopen("bqb4.bmp","r"))==NULL)
        {
          /*如文件缺失则报错，返回*/
          printf("Can't find the file");
          return 1;
       }

/*取得文件的状态参数，存放于结构体内*/
stat("bqb4.bmp",&st);

/*从结构体中取出文件长度*/
filelength=st.st_size;
delay();

/*文本输出*/
textout(0,30,"这只是一个小实验\n",0xf900,0x0000);

/*将文件内容读入缓冲区，刚才取得的文件长度应用于此*/
fread(buf1,1,filelength,fp);

/*关闭文件，释放资源*/
fclose(fp);
```

```
    /*从缓冲区进行液晶屏输出*/
    ShowBuf(buf1,0,0);

    /*打开文件*/
    fp=fopen("bqb6.bmp","r");

    /*取得文件参数*/
    stat("bqb6.bmp",&st);

    /*取得文件长度*/
    filelength=st.st_size;

    /*读入缓冲区*/
    fread(buf1,1,filelength,fp);
    fclose(fp);

    /*显示缓冲区*/
    ShowBuf(buf1,0,0);

    /*以下同理，可以改用ShowBMP函数，使用很方便，但显示时没有ShowBuf流畅！*/
    fp=fopen("bqb8.bmp","r");
    stat("bqb8.bmp",&st);
    fread(buf1,1,st.st_size,fp);
    fclose(fp);
    delay();
    ShowBuf(buf1,0,0);
    fp=fopen("bqb10.bmp","r");
    stat("bqb10.bmp",&st);
    fread(buf1,1,st.st_size,fp);
    fclose(fp);
    delay();
    ShowBuf(buf1,0,0);
    delay();
    fp=fopen("bqb12.bmp","r");
    stat("bqb12.bmp",&st);
    fread(buf1,1,st.st_size,fp);
    fclose(fp);
    delay();
    ShowBuf(buf1,0,0);
}
```

三. 实验内容和步骤

1．获得图形图像：

自己画，或者从网上下载，并转化成小于16位的RGB位图；

2. 选择合适的显示方式，主要从图形图像显示的连续性需求考虑；

3. 调试例程，结合自己的图形图像，完成界面设计。

四.思考题

1. 试着以图形显示方式，改写以前的文字界面程序。

实验十九：触摸屏实验

一． 实验目的

通过本实验，使学生掌握 Linux 下触摸屏程序的设计，了解触摸屏的工作方式和原理。可以参考相应的源代码，修改或编写自己的嵌入式应用程序。

二． 实验原理和说明

触摸屏可以被用来输入命令和文字，它是嵌入式设备最常用的输入接口之一。触摸屏和 LCD 并不是同一个物理设备。触摸屏附着在 LCD 上，是透明的一层薄膜，它负责将受压的位置转换成为模拟电信号，再经过 A/D 转换，成为数字量表示的 X、Y 坐标，送入 CPU 进行处理，为了可视化还可在 LCD 上显示出来。看起来，好像是直接在 LCD 上“触摸”一样。触摸屏的大小也并不与 LCD 大小严格相等。

触控屏的基本原理是，用手指或其他物体触摸安装在显示器上面的触控屏时，所触摸的坐标位置由触控屏控制器检测，并通过接口送到 CPU，从而确定输入的 X、Y 坐标信息。

触控屏系统一般包括触控屏控制器(卡)和触摸检测装置两个部分。其中，触控屏控制器(卡)的主要作用是从触摸点检测装置上接收触摸信息，并将它转换成触点坐标，再送给 CPU，它同时能接收 CPU 发来的命令并加以执行；触摸检测装置一般安装在显示器的前端，主要作用是检测用户的触摸位置，并传送给触控屏控制卡。触摸屏按其转换原理可分为五类：矢量压力传感式、电阻式、电容式、红外线式和表面声波式，其中电阻式触摸屏在嵌入式系统中用的较多。S3C2410 内置了触控屏控制器，它要求外接简单的接口电路与触控屏相连。

触摸屏可以看成是一个二维精密电阻网络，可以进一步等效成沿 x 方向的电阻 rx 和沿 y 方向的电阻 ry。rx 是指 xe-x0 这段电阻，ry 是指 ye-y0 这段电阻。如图 17-1 所示。当笔尖压在 K 点时，rx 方向的电阻可等效成两段：rxk-x0 和 rxe-xk。ry 方向上的电阻分成 ryk-y0 和 rye-yk。触摸屏控制接口有 6 根线引向 S3C2410 的引脚。当 nXPON、XMON 有效，T2、T4 导通，rx 电阻被加电，xk 点处的模拟电压经过 rye-yk 电阻、YPOS 引线进入 ADC 的通道 AIN[0]（在 RJARM2410 开发板的实际连线）。该通道以 10bit 的精度完成模数转化，该电压的数字量 Dxk 存入 S3C2410 内部 ADCDAT0 寄存器的 Xpdata 域。

同样，当 nYPON、YMON 有效，T1、T3 导通，K 点沿 y 方向的座标点 yk 处形成 ry 电阻的分割，该点的模拟电压经 rxe-xk 电阻、XPOS 引线进入 AIN[2]通道，模数转化后的数字电压 Dyk 信号存入 ADCDAT1 寄存器的 Ypdata 域。

从 x0 到 xk 之间的电阻值 rxk-x0 和从 yp 到 yk 之间的电阻值 ryk-yp 可以很容易的被求出：

rxk-x0=rxe-x0*Vxk-x0 /VE

ryk-y0=rye-yp*Vyk-y0 /VE

若 ADC 的参考电压 Vref=VE，转化精度为 10bit，则 xk 点和 yk 点的数字量 Dxk、Dyk 可近似用下面的表达式描述：

Dxk=1024*Vxk-x0 / VE

Dyk=1024*Vyk-yp / VE

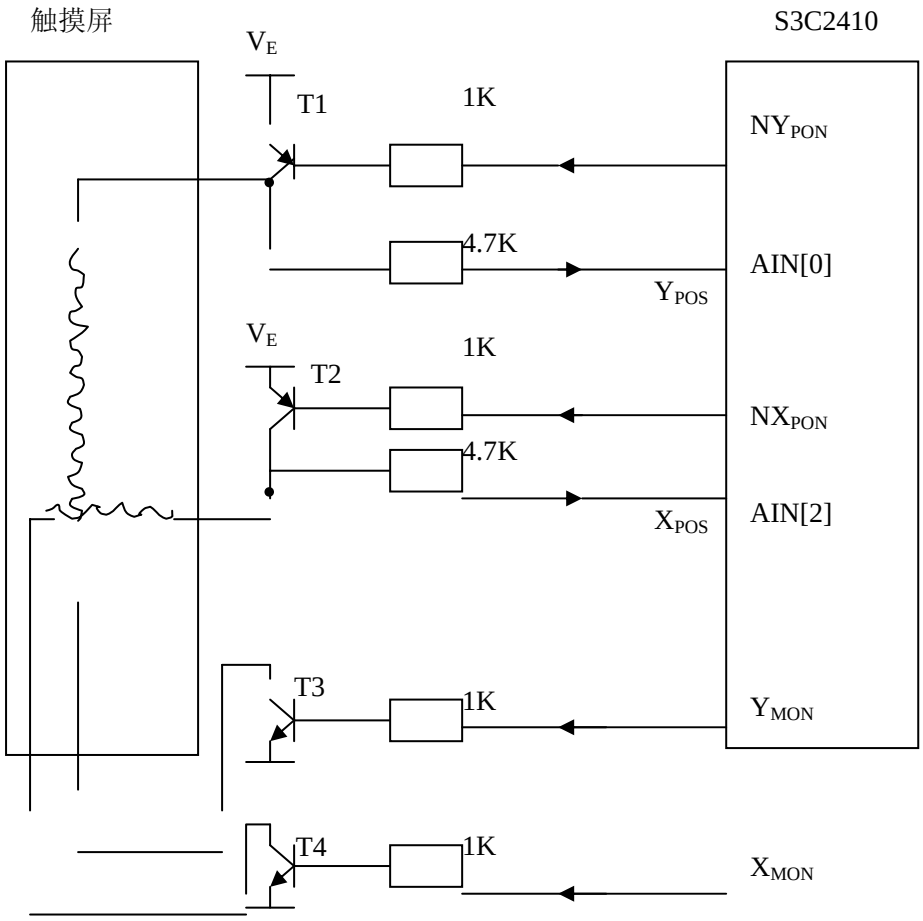


图 17-1

上述的处理简化了许多细节。如 T1、T2、T3、T4 控制用晶体管的导通电阻不可能等于 0。另外触摸屏的分度是按 10 比特 ADC 的转化值表达的，它和 LCD 显示屏的原点并不重合，如图 17-2 所示。这里以 240*320LCD 为例，X1 是 LCD 的起点；X2 是 LCD 的终点。xk 是触针按下的 K 点的 X 位置。要转换成 LCD 像素点阵的位置，LCD Kx 则用下述公式计算：

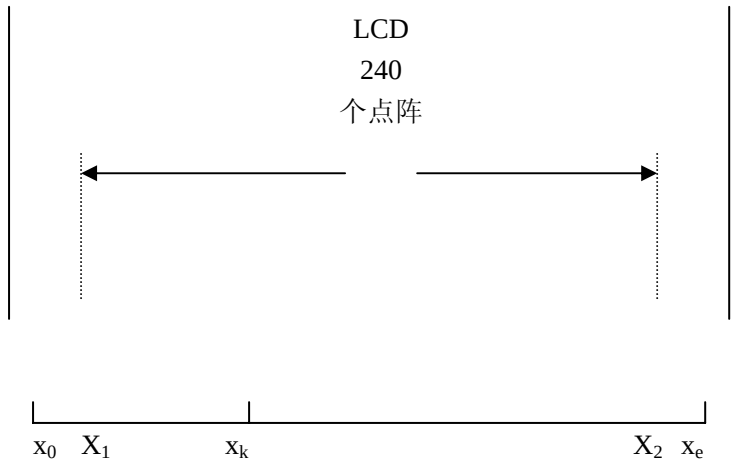


图 17-2

[x1，x2]在下面的例子中，有效取值区间是[90，1022]。[y1，y2] 在下面的例子中，有效取值区间是

[72，1022] (严格说，开机后要运行校准程序，应当在 LCD 上显示两个坐标校准点，通过触笔点触校准点，完成位置的校准)。

LCD Kx=240*(Dx_k-90)/(1022-90)

LCD Ky=320*(Dy_k-72)/(1022-72)

要想得到与 LCD 点阵相一致的触摸屏位置，必须按公式作转化，其中 LCD 原点和满量程的 X 位置的 AD 值（Dx1，Dx2）和 Y 位置的 AD 值（Dy1，Dy2）需要实测。这里从 AD 转换值到 LCD 像素值的映射计算需要用软件来完成。

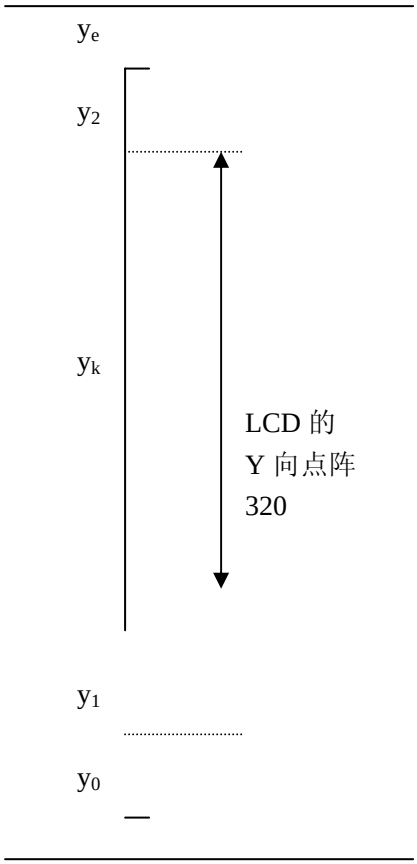


图 17-3

上述公式表述的是 LCD 座标原点在左下角的情况，如果 LCD 原点在左上角（如同一般的 CRT 显示屏），那么

LCD Ky=320-320*(Dy_k-72)/(1022-72)

S3C2410 在完成一次上述 X、Y 座标测试，要求控制信号的状态如表 17-1 所示

表 17-1

	nXpon	Xmon	nYpon	Ymon	Ypos	Xpos
测量 X 座标	√	√	×	×	√	×
测量 Y 座标	×	×	√	√	×	√

表 17-1 中 √ 表示有效操作，×表示无效操作

以输入大小字母为基本功能的应用程序为例，简要介绍一下编写触摸屏应用程序的基本步骤和模块：

(1) 首先要设计输入界面，如图 17-4、图 17-5、图 17-6 所示，它们是 BMP 格式的图形文件，通过调用 ShowBMP（s1，s2，s3）函数，被显示在 LCD 上：

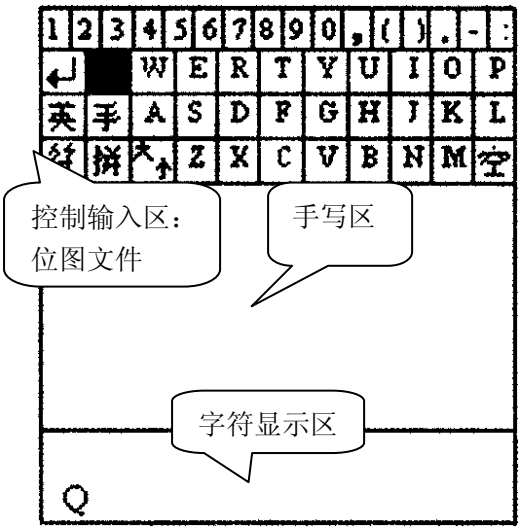


图 17-4 手写输入大写字母

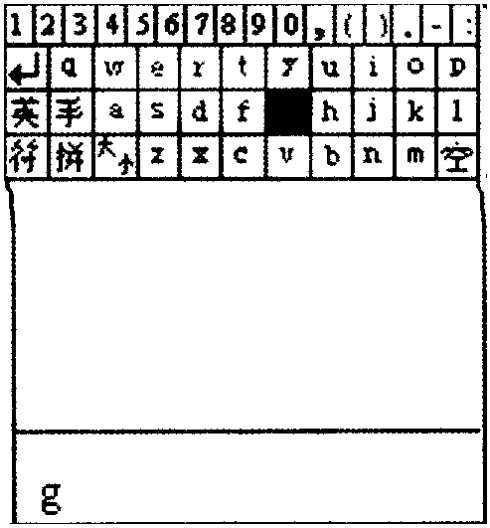


图 17-5 手写输入小写字母

(a) 图 17-4：触摸字符输入区，对应的字符区就会变暗，在 LCD 下部显示区中显示该字符。

(b) 图 17-5：先单击“大/小”软按键，软键盘输入字母就变成了小写，可以输入小写字母。

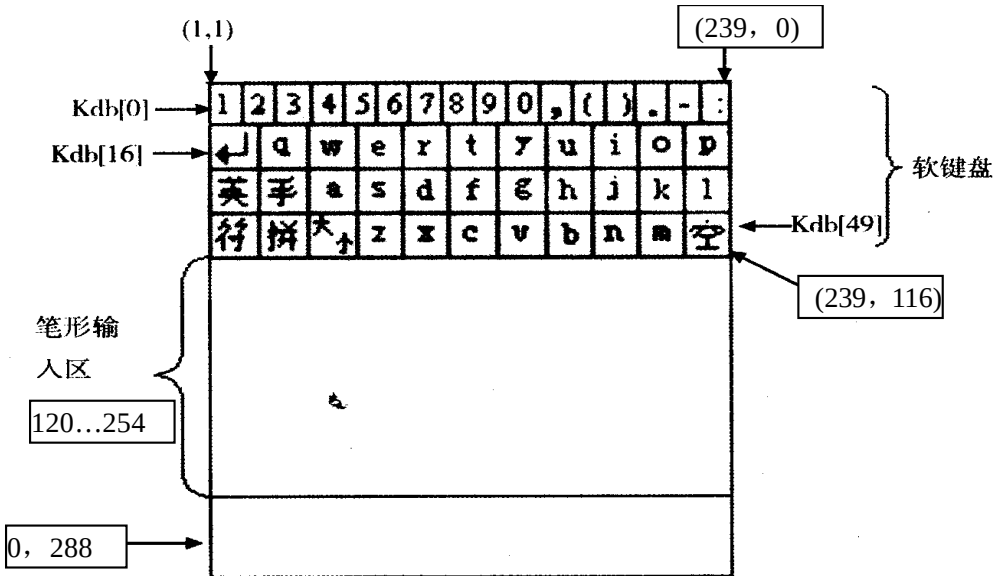


图 17-6 软键盘、笔形输入区和数据输出区在 LCD 上的分布

(2) 定义 kdb[]结构数组。该数组是同软键盘的分布密切相关的，数组元素与字符输入区的对应关系见上图 17-6。该数组的每一个元素都是一个数据结构，对应着软键盘上的一个按键。如图 17-6，kdb[0]对应着“1”这个按键。这个元素的信息如下：

{'1', 1+STARTX, 1+STARTY, 15+STARTX, 24+STARTY}

“1”表示在软键盘对应的字符是“1”，1+STARTX，1+STARTY，15+STARTX，24+STARTY 表示

其所处的小矩形的各个位置坐标。已经定义了 STARTX=STARTY=0，所以在软键盘上包围“1”的矩形坐标位置为(1, 1, 15, 24)，如图 17-6 所示。仔细看可以发现，不同行字符的大小是不同的，这就要求定义一个软键盘的结构体，在识别不同行的键时，如判别 z 键的 X、Y 坐标区间[kbd[z].startx, kbd[z].starty ; kbd[z].endy, kbd[z].endy]也要有区别。这里的 z 代表某一个键的索引值。

笔形输入区在如图 17-6 的 120<y<254 部分。余下的 260<y<320 区域为数据输出区。

(3) 初始化触摸屏 init_handpad()

void init_handpad()/*初始化触摸屏*/

(4) 获取落点坐标 get_handpad()

get_handpad(x，y)的功能是获取触摸在触摸面板上的点的位置。如果成功地获取了落点位置，则函数返回 1，否则返回 0。(x，y)为所检测到落点位置的 AD 转换值。在这里作参数的是指向 x 和 y 的指针。请注意关键字 static 的存在，即 buff 中的字符都是静态存储变量。在函数调用结束后其占用的存储单元并不释放，下一次调用时该变量的初始值就是上一次调用结束后的值。从后面可以发现，这是为了保证采样的有效性。如果两次采样一致，表明该采样数值有效，可以把确认标记发送到 LCD 上去，否则必须重新采样。因此，在程序中设置了一个标志 flag（相当于后述的 X、Y 坐标的 AD 采样成功的 cBuffer.pressure 标志）表示采样是否有效。

注意：字符数组 buff[4]是静态的。

handpad ：触摸屏输入与交互演示例程（含软键盘）。位于：../uClinux-dist/user 目录。

```
void main(void)
{
    unsigned short  i,x,y,index=0xffff;    /*index 对应了触摸板软面板上的字符表的各个字符在
    kdb[]结构中的目录项*/
    char buf[10],can[20];
    short Upper=1,count=0,start,end;       /*Upper 用来控制软面板的大/小写*/
    FILE *fp=fopen("gb.txt","rb");

    init_handpad();    /*初始化触摸板*/
    initgraph();       /*初始显示环境*/
    clearscreen();     /*清屏*/
    ShowBMP(KEYBOARD,0,0);          /*在(0,0)处显示大写软面板*/
    setmode(MODE_SRC_XOR_DST);    /*设置显示环境*/
    rectangle(0,120,239,254);              /*在(0,120,239,254) 中做矩阵*/
    if(!Initialize()) return;
    while(1){
        if(get_handpad(&x,&y))          /*获取落点*/
        {
            if(index!=0xffff)                /*如果落点被成功地检测到,且落在软面板键盘上*/
        {
            fillrect(kbd[index].startx,kbd[index].starty,
            kbd[index].endx+1,kbd[index].endy+1);
        }                              /*将对应落点处的小键盘按键填充为黑色*/

        if(y<116) /*y<116,在软键盘字符区*/
    {
```

```
if((y>kbd[0].starty)&&(y<kbd[0].endy))     {/*如果落点在第一行*/
   for(i=0;i<16;i++)
      I f((x>kbd[i].startx)\
          &&(x<kbd[i].endx))
           break;
           if(i==16) index=0xffff;
           else index=i; /*如果落点不在 kdb[16]处,就将 kdb[]数组的索引项赋给 index,否则将
             0xffff 赋给 index,跳到下面的笔形输入区代码段*/
      } else
if((y>kbd[16].starty)&&(y<kbd[16].endy)) {/*如果落点在第二行*/
       for(i=16;i<27;i++)
           if((x>kbd[i].startx)\
              &&(x<kbd[i].endx))
                break;
             if(i==27) index=0xffff;else index=i;/*如果落点不在 kdb[16]处,就给 kdb[]数组的索
               引项赋给 index,否则将 0xffff 赋给 index,跳到下面的笔形输入区代码段*/
       if(index==16) { buf[0]=0;count=0;textout(0,288,"\n");
        FindCandidate(buf,&start,&end);continue;}
} else
if ((y>kbd[27].starty)&&(y<kbd[27].endy)) {/*如果落点在第三行*/
  for(i=27;i<38;i++)
      if((x>kbd[i].startx)\
          &&(x<kbd[i].endx))
           break;
        if(i==38) index=0xffff;else index=i;/*如果落点不在 kdb[16]处,就把  kdb[]数组的索引
          项赋给 index,否则将 0xffff 赋给 index,跳到下面的笔形输入区代码段*/
}else
if((y>kbd[38].starty)&&(y<kbd[38].endy))   {/*如果落点在第四行*/
    for(i=38;i<49;i++)
       if((x>kbd[i].startx)\
          &&(x<kbd[i].endx))
           break;
    if(i==49) index=0xffff;
else
    if(i==40){/*kdb[40]处为大/小,然后进行大/小转换*/
       if(Upper)/*如果原来为大写状态,则转换为小写状态*/
           ShowBMP(L_KEYBOARD,0,0);
       else
           ShowBMP(KEYBOARD,0,0);
       Upper=!Upper;
       index=0xffff;
    }
  else
       index=i;   /*将目录项赋予 index,以输入 Z,X..M 等字符*/
```

```
    }    /*控制在(0,288)处显示字符*/
    if(index<49){
    fillrect(kbd[index].startx,kbd[index].starty,
              kbd[index].endx+1,kbd[index].endy+1);
        if(Upper)  buf[count]=toupper(kbd[index].ch);/*如果为大写状态,就转换为小写状态；如
          果为小写状态，转换为大写状态*/
     else
        buf[count]=kbd[index].ch;
    buf[++count]=0;
    textout(0,288,buf);/*在(0,288)处显示字符*/
    FindCandidate(buf,&start,&end);
    fseek(fp,start,SEEK_SET);
    if(end-start>18)
    {
        fread(can,18,1,fp);
        can[18]=0;
    }
    else
    {
        fread(can,end-start,1,fp);
        can[end-start]=0;
    }
    #ifdef DEBUGTRACE
        printf("count:%d buf:%s can: %s\n",count,buf,can);
    #endif

    textout(0,288,"              ");
    textout(0, 288,buf);
    textout(0,256,can);
    //printf("%d->%d\n",start,end);
    count=strlen(buf);
    }
}
else/*如果 index=0xffff,那么就进入笔形输入区*/
{
    static unsigned short oldx,oldy,err;/*err 为静态变量.通过判断 err 的数值来*决定是否连笔输
          入*/
if(y>120&&y<254)/*如果落点在笔形输入区*/
    {
       err=(oldx>x)? oldx-x:x-oldx;
       err+=(oldy>y)? oldy-y:y-oldy;

        if(err>0&&err<20){/*判断两个落点的间距大小,如果过大,就不连笔适淙?*直接位移到该
            点显示.*否则从起始点到落点间做直线*/
```

```
                lineto(oldx=x,oldy=y);
            }else{
                moveto(oldx=x,oldy=y);
            }

        }
    index=0xffff;
  }
  }//end while
}
```

以下给出的是另一个全屏触摸的应用例程参考实现：

(1) 初始化触摸屏

```
void init_handpad()
{
screen_tp_fd = open(device, O_RDONLY);
        if (screen_tp_fd == -1) {
                printf("Unable to open touch screen");
                exit(0);
        }
}
```

(2) 取得触摸的位置

```
int get_handpad(unsigned short *x,unsigned short *y)
{
int i=0,x_sum=0,y_sum=0;
int xa[3],ya[3];
for(i=0;i<3;)
{
    TS_RET_HANDPAD   cBuffer;
    read(screen_tp_fd,&cBuffer,sizeof(TS_RET_HANDPAD));
            //read 函数在 cBuffer 结构体内获得 X、Y 坐标的 AD 转换值。
    if(cBuffer.pressure){
            // cBuffer.pressure 相当于前述的 X、Y 坐标的 AD 采样成功的标志。
        if(cBuffer.y >= 1022 || cBuffer.x >= 1022
                ||cBuffer.y<=72 || cBuffer.x<=90){
        }else{
            xa[i] = cBuffer.x;
            ya[i] = cBuffer.y;
            i++;
        }
    }
}
```

```
*x = (get_average_num(xa[0],xa[1],xa[2])-167)*(220-20)/(924-167)+20;
*y = 300-(get_average_num(ya[0],ya[1],ya[2])-128)*(300-20)/(945-128);
    //对 X 和 Y 的 AD 转换结果求三次平均值，然后按比例换算成 LCD 的像素位置，该转换方
    式和上述公式和示意图基本符合，读者可自行推断。
return 1;
}
```

下面介绍一下触摸屏驱动的移植。

触摸屏的输出为两路模拟信号，采用外部的芯片将模拟信号转化为数字信号，通过串行总线输入到 cpu。驱动程序可获取相关坐标的 x，y 值。

驱动的移植主要包括触摸屏中断的设定，spi 片选的选取，以及根据具体的 cpu 来调整驱动的框架使之正确运行。

(1) Mxl 的触摸屏移植

Mxl 的触摸屏的硬件中断使用 GPIO 中断，从原来的 PD31 变成现在使用的 PB19，对应硬件初始化的操作需要更改中断的操作。在 digi_init()中添加：

```
//////////////////////////////////////////////////////////
    *(volatile U32*)PTB_GIUS |= 0x1<<16; //设置 PB16 为 GPIO 模式
    *(volatile U32*)PTB_DDIR &= ~(0x1<<16); //设置 PB16 为输入模式
    *(volatile U32*)PTB_PUEN |= 0x1<<16; //上拉
    *(volatile U32*)PTB_ICR2 |= 0x00000003; //中断触发模式，设为低电平触发，而事实上如论如何设置，都是低电平触发，此处比较奇怪。
    *(volatile U32*)PTB_IMR  &= ~(0x1<<16);   //启动时，使中断无效先
    *(volatile U32*)PTB_ISR  |= 0x1<<16;      //清除所有的中断
    ////////////////////////////////////////////////
```

来完成中断从 pd31 到 pb16 的配置。同时需要对应更改一些对中断进行配置的函数以及对应的宏。

```
static void touch_pan_set_inter(void)
{
        *(volatile U32*)PTB_GIUS |= 0x1<<16; //设置 PB16 为 GPIO 模式
        *(volatile U32*)PTB_IMR  &= ~(0x1<<16);  //启动时，使中断无效先
        *(volatile U32*)PTB_PUEN |= 0x1<<16; //上拉
        *(volatile U32*)PTB_DDIR &= ~(0x1<<6); //设置成输入模式
        *(volatile U32*)PTB_ICR2 |= 0x00000003; //中断触发模式，设为低电平触发
        temp = REG_PB_SSR;//当前的中断状态，第 16 位有效
}
static void touch_pan_enable_inter(void)
{
        *(volatile U32*)PTB_IMR |= 0x1<<16; //  中断使能
}
static void touch_pan_disable_inter(void)
{
        *(volatile U32*)PTB_IMR  &= ~(0x1<<16);     //屏蔽中断
}

static void touch_pan_clear_inter(void)
```

```
{
        *(volatile U32*)PTB_ISR   |= 0x1<<16;      //清中断，写 1 清中断
 }
static void touch_pan_set_pos_inter(void)
{
        *(volatile U32*)PTB_ICR2 &= 0xfffffffc; //设置中断模式，上升沿触发
 }
static void touch_pan_set_neg_inter(void)
{
        *(volatile U32*)PTB_ICR2 &= 0xfffffffd; //设置中断模式
        *(volatile U32*)PTB_ICR2 |= 0x00000001; //下降沿触发
}
```

(2) 驱动流程介绍

原来的驱动流程是在中断没来之前将中断设为下降沿有效（中断默认为高），当中断到来之后，将中断设为上升沿有效，这样保证在触摸屏被连续按下的时候，不会连续的触发中断，然后在中断处理程序中申请一个时钟 timer，这样连续检测从触摸屏取得的值，送到缓冲区中，直到触摸屏画笔抬起的时候，中断被触发，这时在中断处理程序中再将中断设为下降沿有效，保证下一次触摸屏被按下的时候可以触发中断。

但是我们的中断我发现无论如何设置中断，中断都是低电平有效。所以我修改的中断处理的流程，下面我主要介绍我编写的中断处理流程。

Digi_isr 为中断处理程序

```
digi_isr(int irq, void *dev_id, struct pt_regs *regs)
{
        if((REG_PB_ISR & (~TOUCH_INT_MASK))!=0)
        {
                touch_pan_clear_inter();//清中断
                touch_pan_disable_inter();//屏蔽中断
                tasklet_schedule(&digi_tasklet);//将 digi_tasklet 加入进度调度
        }
}
```

其中 digi_tasklet 已经将 digi_tasklet_init 初始化为其调度的任务。tasklet_init(&digi_tasklet, digi_tasklet_action,(unsigned long)0);

我们在中断到来的时候就立即屏蔽中断，使中断不再进入，并在中断处理函数中，设置一个 tmer，来不停的从触摸屏读取数据，这其中触摸屏应该不停的被按下。

参见 digi_tasklet_action（）的源代码：

```
        touch_pan_read_data(&newX, &newY);
        while((newX == TPNL_PEN_UPVALUE)&&(maxDelay <= 3))
                {
                        touch_pan_read_data(&newX, &newY);
                        maxDelay++;
                }
                if(newX == TPNL_PEN_UPVALUE)
                {
```

```
                touch_pan_enable_inter();
                return;
            }
            add_x_y(newX, newY, PENDOWN);
            wake_up_interruptible(&digi_wait);
            spin_lock_irq(&pen_lock);
            pen_timer.expires = jiffies + HZ/100;
            add_timer(&pen_timer);
            pen_timer_status = 1;
            spin_unlock_irq (&pen_lock);
```

该 timer 的处理函数就是 digi_sam_callback（）函数，具体代码参见下面两句，在启动函数中。

```
    init_timer(&pen_timer);
    pen_timer.function = digi_sam_callback;
```

这样 digi_sam_callback（）函数就会在 HZ/100 以后被调用。

```
static void digi_sam_callback(unsigned long data)
{
    u32 newX,newY;
    u32   maxDelay;
    maxDelay = 0;
    newX = TPNL_PEN_UPVALUE;
    if(pen_timer_status !=0)
        mod_timer(&pen_timer, jiffies + HZ/100);
    //在每一次该函数被调用的时候，设定下一次该函数被调用的时间
    if((REG_PB_SSR & 0x00010000) != 0)//这里表示触摸笔抬起来了
    {
            add_x_y(0, 0, PENUP);
            if (!NODATA())
            {
                wake_up_interruptible(&digi_wait);
                if(ts_fasync)
                kill_fasync(&ts_fasync, SIGIO, POLL_IN);
            }
            spin_lock_irq(&pen_lock);
            pen_timer_status = 0;
            del_timer(&pen_timer);
            spin_unlock_irq(&pen_lock);
              //因为触摸笔已经抬起来了，删除该计数器
              touch_pan_enable_inter();
            touch_pan_clear_inter();
              //使能中断，下一次笔被按下的时候，就响应中断

        return;
    }else//这里表示笔依然被按下着，马上要读取触摸屏坐标
    {
```

```
                //read data with MAX delay from ADC
                while((newX == TPNL_PEN_UPVALUE)&&(maxDelay <= 100))
                {
                        touch_pan_read_data(&newX, &newY);
                        maxDelay++;
                }
                if(newX == TPNL_PEN_UPVALUE)
                {
                        goto add_x_y_line;
                        return;
                }
                add_x_y(newX, newY, PENDOWN);
                wake_up_interruptible(&digi_wait);
                return;
        }
}
```

这样在每一次 timer 到来的时候，执行 digi_sam_callback（）函数，设置下一次 timer 的时候，并检测(REG_PB_SSR & 0x00010000)信号，如果不为 0 表示触摸笔抬起，否则为按下，按下的时候就读取触摸屏坐标，放到缓冲区当中，如果触摸笔抬起，就在缓冲区中放入抬起信号，同时删除 timer。该驱动框架的优点是几乎实时检测和绘制 LCD 屏，实时在内核和用户态切换。

(3) s3c2410 触摸屏的移植

2410 采用外部中断 1 作为触摸屏的中断，中断的初始化设置在 touch_pan_set_inter()函数中设定：

```
set_external_irq(IRQ_EINT1,EXT_FALLING_EDGE, GPIO_PULLUP_EN);
```

设定外部中断 1 为下降沿触发并且使能上拉的触摸屏中断。

然后设定中断处理程序：

```
tmp = request_irq(IRQ_EINT1,
                        (void * )digi_isr,
                        TPNL_INTR_MODE,
                        MODULE_NAME,
                        MODULE_NAME);
```

指定 digi_isr 为触摸屏的中断处理程序。

然后在 spi_init()函数中初始化 spi:

rSPPRE0 = 0x10;//2410SPI_BAUD; 设定 spi 的时钟，该时钟必须小于 25M，若触摸屏的处理速度过慢，可适当减小该值。

rSPCON0 = 0x18;//设定 spi 为 poll 模式，为减少中断 IO 的使用，我们采用 poll 模式，直接查询 spi 的寄存器来获取当前 spi 的工作状态。

```
for(i=0;i<10;i++)
        rSPTDAT0 = 0xff;
```

按照 cpu 手册的介绍，将 rSPTDAT0 寄存器写 10 次 0xff 来初始化与 spi 连接的外部设备，笔者怀疑应该是将 10 个字节发送到外设来初始化外设，而不是简单的读写寄存器就可以了，所以在驱动初始化的后面，我们又调用

```
spi_tx_data(0xd000);
spi_tx_data(0x0000);
```

```
        spi_tx_data(0x9000);
        spi_tx_data(0x0000);
        spi_tx_data(0xD000);
        spi_tx_data(0x0000);
        spi_tx_data(0x9000);
        spi_tx_data(0x0000);
        spi_tx_data(0x0000);
        spi_tx_data(0x0000);
        spi_tx_data(0x0000);
        spi_tx_data(0x0000);
        spi_tx_data(0x0000);
        spi_tx_data(0x0000);
```

来进一步初始化与 spi 连接的外设。

```
    rGPECON |=     0x0a800000;
```

rGPECON &= (~0x05400000);//将 GPE11、12、13 依次设置为 SPIMISO0、SPIMOSI0、SPICLOCK0。

```
    rGPEUP |= 0x3800;//使能上拉
```

这样就完成 spi 的初始化。Digi_init()结束。

当触摸笔在触摸屏按下的程序，内核相应中断，进入中断处理程序，按照上面的设定，进入 digi_isr()函数。

```
    static void digi_isr(int irq, void *dev_id, struct pt_regs *regs)
    {
            if((rGPFDAT & 0x00000002)==0)//查询当前中断是否为触摸屏中断，这种查询用于多个共享中断的情况。
            {
                    touch_pan_clear_inter();
                    touch_pan_disable_inter();
                    tasklet_schedule(&digi_tasklet);
                      //为了加快中断的相应速度，通常不在一个中断处理程序中停留过久的时间，所以这里将 digi_tasklet 压入任务调度之后就立即退出了，这里牺牲一个触摸屏的中断处理速度，而不是整个系统的相应速度变得过慢。
            }
            return;
    }
```

当 digi_tasklet 被调度的时候，就会调用 digi_tasklet_action()函数来提取触摸屏坐标，相应的处理与 mxl 基本相同。唯一的不同在于 mxl 的 cpu 在笔者调试的板子上，如果如何设置中断的触发方式，都是低电平有效，这意味着当触摸笔按下的时候，会不断进入中断；而 s3c2410 的中断被设置成下降沿有效的时候，是在触摸笔按下和抬起的时候都会响应中断。不同的 cpu 响应中断的方式可能不同，这样在移植驱动的时候需要特别注意。

为了增加驱动代码的通用性，笔者将触摸屏对 LCD 的坐标变化放到内核中来作，这样用户程序可以对触摸屏和 LCD 进行一体化的编程。

下面是读取触摸屏坐标的 read 函数。

```
    ssize_t digi_read(struct file * filp, char * buf, size_t count, loff_t * l)
    {
```

```
int nonBlocking = 1;//设置成非阻塞方式，因为我们的驱动支持用户层的 select 操作，如果没有数据的话，用户通过 select()立刻可以知道当前没有数据，就立即返回了。
int minor;
ts_event_t *ev;
int cnt=0;
minor = check_device( filp->f_dentry->d_inode );
if ( minor == - 1)
        return -ENODEV;
if( nonBlocking )
{
        if( !NODATA() )
        {
                cnt = get_block(&ev, count) ;//这里是获取触摸屏坐标的关键函数。//我们对屏幕的坐标变换也是在这里进行。
                if(cnt > count){
                        printk(KERN_ERR"Error read spi buffer\n");
                        return -EINVAL;
                }
                __copy_to_user(buf, (char*)ev, cnt );
                return cnt;
        }
        else
                return -EINVAL;
}
  ……
```

下面来查看 get_block（）函数。

这样，我们在 get_block()函数的结尾部分加入坐标转换命令。

```
static int get_block(ts_event_t * * block, int size)
{
        int cnt, rd;
        unsigned long flags;
        ts_event_t *p,*pTmp;
        if(NODATA())
                return 0;
        cnt = 0;
        save_flags(flags);
        cli();
        if(DSIZE()*sizeof(ts_event_t) >= size)
        {
                rd = size/sizeof(ts_event_t);
        }
        else
        {
                rd = DSIZE();
```

```
        }
        * block = p = get_data();
        cnt ++;
        while(p && (cnt < rd))
        {
                if(rptr == 0)
                        break;
                p = get_data();
                cnt ++;
        }
        restore_flags(flags);
        ////////////////////////////////add by lyk RJtech
        pTmp=*block; //*block 中保存的是当前的坐标的一组参数
        if(pTmp->pressure!=0){//如果鼠标信息为按下，则进行坐标转换
                pTmp->x=10+(pTmp->x-430)*(230-10)/(3558-430);
                pTmp->y=10+(pTmp->y-290)*(310-10)/(2679-400);
                  //这些坐标变换的信息需要对触摸屏反复取值作为平均值进行校正。
        }
        ////////////////////////////////////////////////////
        return (cnt)*sizeof(ts_event_t);
}
```

这样用户进程就可以获取触摸屏的坐标信息，从而来操作 LCD 等等。

三．实验内容和步骤

1．编写关键子函数，并调试通过；
2．使用轮询方式，调用第一步编写的子函数进行触摸位置判断，并应用于具体应用程序。
3．更换一个不同大小的触摸屏，校正触摸屏对 LCD 的左边变换，使触摸屏增长工作。

四．思考题

1．试着以触摸屏为输入途径，增强以前做过的实验，提高人机交互的友好度。

实验二十：以太网实验 SOCKET 通信

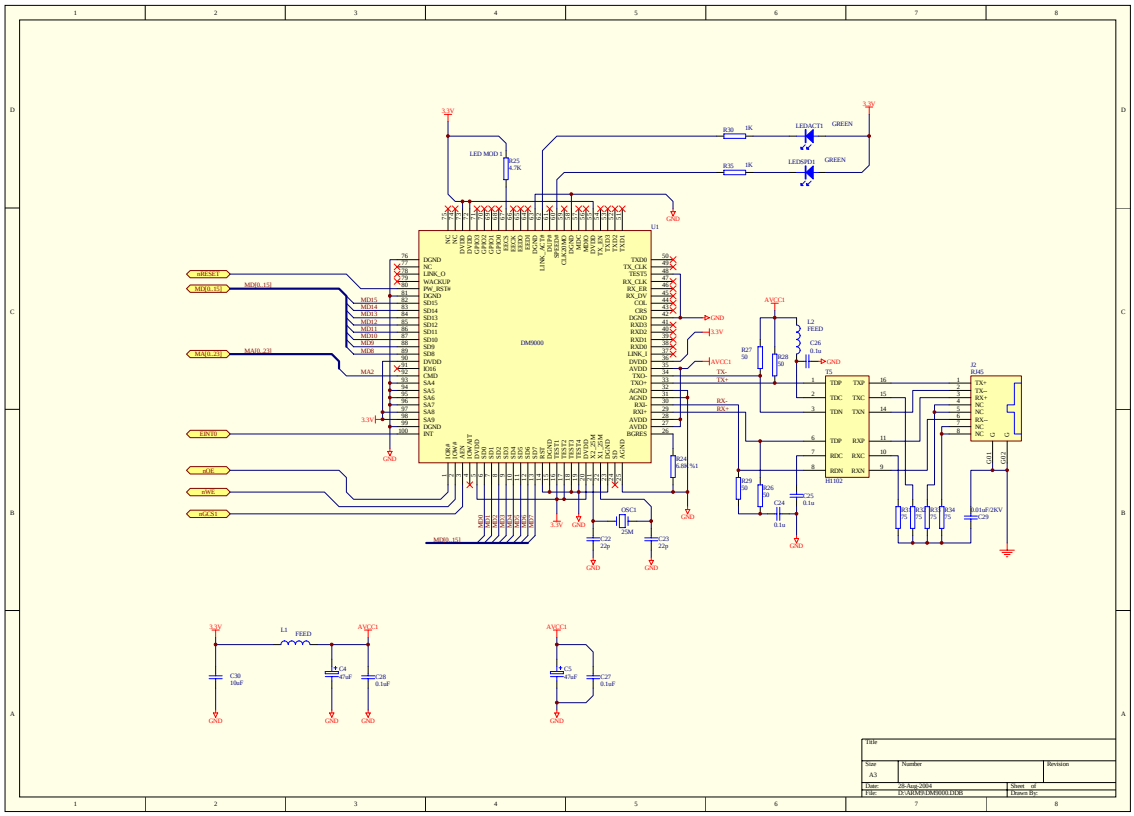
一. 实验目的

通过本实验，使学生掌握 Linux 下 SOCKET 通信程序的设计，了解 SOCKET 的工作方式和原理。

二. 实验原理和说明

1. Rjtek2410-1 开发板采用外扩以太网专用芯片的方法，建立网络端口的，如图 19_1 所示。图中，MD[15：0]是数据总线，nOE 是读有效，nWE 是写有效，nCS1 是 S3C2410 的地址片选信号，它映射的地址在 0x8000300,VID 在 0x090000a46。它的趋动模块在：

/ Rjtek2410-1/kernel/drivers/net/dm9000x.c



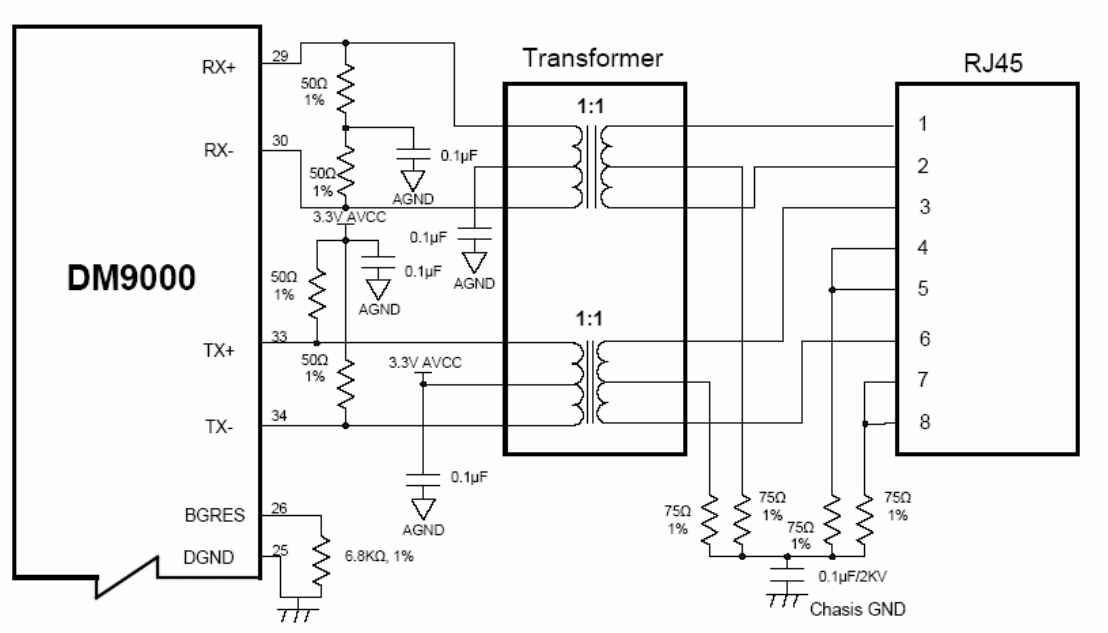


图 19_1 外扩以太网专用芯片电路原理和接线

2. 介绍如何在 Linux 环境下进行 Socket 编程和其常用函数的用法，客户/服务器模型的编程应注意的事项和常遇问题的解决方法。通过几个实例，最后学习网络通信功能是如何实现的。

2.1 TCP/IP 协议概述

2.1.1 TCP/IP 的参考模型

TCP/IP 参考模型是计算机网络的始祖 ARPANET 和其后继的因特网使用的参考模型。ARPANET 是由美国国防部 DoD(U.S.Department 0f Defense)赞助的研究网络，它通过租用的电话线连结了数百所大学和政府部门。当无线网络和卫星出现以后，现有的协议在和它们相连的时候出现了问题，所以需要一种新的参考体系结构。这个体系结构在它的两个主要协议出现以后，被称为 TCP/IP 参考模型。模型结构如图 19_2 所示。

图 19_2 TCP/IP 的参考模型

由于国防部担心一些珍贵的主机、路由器和互联网关可能会突然崩溃，所以网络必须实现的另一目标是网络不受子网硬件损失的影响，已经建立的会话不会被取消，而且整个体系结构必须相当灵活。

TCP/IP 参考模型共有四层：应用层、传输层、互联网层和主机至网络层。与 OSI 参考模型相比，TCP/IP 参考模型没有表示层和会话层。互联网层相当于 OSI 模型的网络层，主机至网络层相当于 OSI 模型中的物理层和数据链路层。

下面通过一个 www 浏览器来看看这四层的概况：

(1) 应用层：WWW、SMTP、DNS 和 FTP。

(2) 传输层：解释数据。
(3) 互联网层：定位 IP 地址与确定连接路径。
(4) 主机至网络层：与硬件驱动程序对话。

(1) 应用层

首先 WWW 浏览器必须调用 HTTP 协议，这个协议规定用什么样的命令来得到 WWW 文本，这种协议构成了 TCP/IP 的最高层——应用层。应用层包含所有的高层的协议。这些高层协议有：虚拟终端协议 TELNET、文件传输协议 FTP、电子邮件传输协议 SMTP、域名系统服务 DNS 和超文本传输协议 HTTP 等。

(a) 虚拟终端协议 TELNET：允许一台机器上的用户登录到远程机器上并进行工作。
(b) 文件传输协议 FTP：提供有效地将数据从一台机器上移动到另一台机器上的方法。
(c) 电子邮件协议 SMTP：最初仅是一种文件传输。但是后来为它提出了专门的协议。
(d) 域名系统服务 DNS：用于把主机名映射到网络地址。
(e) 超文本传输协议 HTTP：用于在万维网(WWW)上获取主页等。

(2) 传输层

为了使用 HTTP 协议，浏览器要把命令发送到服务器上去，并且从服务器得出回答。但必须记住网络上传输的总是些字节，哪些字节是命令，哪些是回送数据，又有哪些是用于表示“就绪”、“传输中”或者“停止”的验证码。这些解释工作需要一串复杂的协议进行控制，这构成了 TCP/IP 分层结构的第三层——传输层。它的功能是使源端和目标主机上的对等实体可以进行会话。在这一层定义了两个端到端的协议。一个是传输控制协议 TCP。它是一个面向连接的协议，允许从一台机器发出的字节流无差错地发往另一台机器。它将输入的字节流分成报文段并传给互联网层。TCP 还要处理流量控制，以避免快速发送方向低速接收方发送过多的报文而使接收方无法处理。另一个协议是用户数据报协议 UDP，它是一个不可靠的、无连结的协议，用于不需要 TCP 排序和流量控制而是自行完成这些功能的应用程序中。

(3) 互联网层

所有上述的需求导致了基于无连结互联网络层的分组交换网络。这一层被称作互联网层，它是整个体系结构的关键部分。它的功能是使主机把分组发往任何网络并使分组独立地传向目标(可能经由不同的网络)。这些分组到达的顺序和发送的顺序可能不同，因此如果需要按顺序发送和接收时，高层必须对分组进行排序。

互联网层定义了正式的分组格式和协议，即 IP 协议。互联网层的功能就是把 IP 分组发送到应该去的地方。分组路由和避免阻塞是这里主要的设计问题。TCP/IP 互联网层和 OSI 网络层在功能上非常相似。

2.1.2 IP 地址和子网

互联网用 IP 地址来标识主机地址，而数据传输则是通过把数据分成一列小“包”来完成。TCP/IP 世界里的每一台主机都有唯一的 IP 地址，这个 IP 地址是一个 32 位无符号整数，不过通常使用点分十进制表示。例如，00010000110000011111111000000001 是十进制的 281148929，但是实际应用中把它按照 8 位一段的方法分成四段，第一段是 00010000，也就是 16，第二段是 11000000，十进制是 192，同样，剩下来的两段是 254 和 1．因此这个地址是 16.192.254.1。这听起来很简单，但是在实现中有些复杂，主要问题是 TCP/IP 必须兼顾许多困难的问题，其中之一是网络联接。全世界有太多的网络主机，通过一个主机号很难知道它在哪里。最大的难题是，也许主机 13.2.0.5 在纽约，而主机 64.0.7.6 却在北京。如果搜索一台主机地址就要查阅全世界的主机列表，那么互联网将立即崩溃。

解决这一问题的第一个要点是网络分类。把世界上的 IP 地址分段，各段之间独立管理，通常每一段用同样的方式连接到一起，本段的机器之间可以直接互相访问，这样的一个段称为一个系列网络

地址。

为了管理方便，分段用一种特殊的方式。例如，对一些公司(如 IBM)，一个上万台机器的网络很正常，而小公司也许只有十几台机器，TCP/IP 实现中用 A、B、c 类地址来处理这个问题。

A 类地址用于超过 65534 个主机的网络，例如，一个前缀为 18 的网络使用 18.0.0.1～18.255.255.254 的网络地址，这些地址之间是直通的，主机之间可以彼此访问。为了说明这一点，TCP/IP 使用网络掩码和网络地址的概念。在上面的例子中，网络地址是 18.0.0.0，这表示网络的包含地址段是 18.0.0.1～18.255.255.254，即地址部分除去前缀后，余下的部分由全零变成全 1，不过全 1 的地址将保留为广播使用。与网络地址相应，对在此网络中的主机，TCP/IP 使用网络掩码。在上面的例子中，掩码是 255.0.0.0，其含义是：假设 18.0.0.3 想与 18.11.0.75 对话，那么，将这两个地址相互异或(XOR)，再与掩码取“与”(&)，结果是零，说明这两个地址可以直接对话。相反，如果要与 19.1.1.1 对话，那么运算之后不为零，说明必须使用间接方式才能到达。

可以用直接方式理解掩码，掩码中的“1”用来描述网络地址(前缀)，“0”用来描述子网的主机，如 255.0.0.0 意味着将主机地址的前 8 位解释为网络号，而后 24 位解释成网内的主机地址。也就是说，与某个 A 类主机地址前 8 位相同的主机地址被认为是在同一子网内。

A、B、C 类用下面的方式规定：

(1) A 类地址使用 255.0.0.0 的掩码。为了管理方便。规定 A 类地址总是使用头一位为 0 的地址值，这意味着 A 类地址是从 1.0.0.0 到 126.0.0.0。另外还有一个特殊的 A 类地址 127.0.0.0，它用来表示“本机”，即自身。

(2) B 类地址使用 255.255.0.0 的掩码。B 类网从 128.0.0.1 到 191.255.0.0。

(3) C 类地址的掩码是 255.255.255.0。网络地址从 192.0.0.0 到 233.255.255.0。首字节在 224 以上的地址用于实验和开发，部分用于组播，一般用的不多，可不必太关心。地址 255 为特殊的含义，它用于广播，即“向本网上所有人通话”。这个概念对任何人来说，都是很容易理解的。

使用广播有两种方法，并且几乎是等价的，即：

(a) 使用全 1 的地址：即使用 255.255.255.255。

(b) 使用本址广播．即用网络号加上全 1。

A 类网 180.0.0.0 可以使用 18.255.255.255 广播，B 类网 190.4.0.0 的广播地址则是 190.4.255.255。无论哪一种，对于广播地址的访问都将引发将数据发送给本网络上的所有机器。

计算网络掩码和标志子网地址是人们经常需要做的工作。第一种计算是根据某个确定的主机地址和它的网络掩码求出所有可以和它直接通信(在同一子网之内)的主机地址。例如，主机地址是 202.112.50.3，掩码是 255.255.255.0，与它可以直接通话的主机地址可以这样计算：掩码是 255.255.255.0，因此 32 位网络地址的前 24 位被解释为网络地址。凡是与 202.112.50.3 的前 24 位内容相同的主机地址就和它处在同一子网之内，后 8 位是子网里面的机器地址，它可以从全 0 变到全 I，所以所有和这个主机在同一子网之内的主机地址是 202.112.50.0～202.112.50.255。

网络地址/掩码一般用斜线分开，如网络地址是 202.199.248.0，掩码 255.255.255.0 在 UNIX 中一般写成 202.199.248.0/255.255.255.0。但也有另外一种写法，就是用掩码中 I 的个数来代替掩码的实际形式。例如，255.255.255.0 的前 24 位是 1，其他位是 0，因此可以写成 202.199.248.0/24。同样，下面的两栏地址形式是彼此等效的：

202.199.248.0/255.255.255.0	202.199.248.0/24
122.24.0.0/255.255.255.0	122.24.0.0/16
13.4.5.7/255.0.0.0	13.4.5.7/8

在前面一直设置掩码的 0/1 分界在字节边界处，但是这其实并不是必须的，完全可以有其他形式的掩码。例如，240 等于二进制的 11110000，因此一个 255.255.255.240 的掩码意味着前面 28 位是网络地址，而后面 4 位是子网内的 IP。同样，网络地址也可以不由 0 结尾。举个例子来说，202.112.50.16/255.255.255.240 是什么意思?202.112.50.16 是二进制的

11001010011100000011001000010000，套上一个有 28 位为 1 的网络掩码，意味着最后四位由全 0 变成全 1，就是最小是 11001010011100000011001000010000，最大是 11001010011100000011001000011111，这两个数是 202.112.50.16 和 202.112.50.31，所以这个网络地址代表 202.112.50.16 到 202.112.50.31 的所有主机。这个网络地址也可以写成 202.112.50.16/2。

2.1.3 Linux 中的 TCP/IP 网络层次结构

图 19-3 描述了 Linux 对 IP 协议族的实现机制，如同网络协议自身一样，Linux 也是通过视其为一组相连的软件层来实现的。其中 BSD 套接字由通用的套接字管理软件所支持，该软件是 INET 套接字层，用来管理基于 IP 的 TCP 与 UDP 的端到端互联。如前所述，TCP 是一个面向连接协议，而 UDP 则是一个非面向连接协议。当一个 UDP 报文发送出去后，Linux 并不知道也不去关心它是否成功地到达了目的主机。对于 TCP 传输，传输节点间先要建立连接．然后通过该连接传输已排好序的报文，以保证传输的正确性。IP 层中代码用以实现网际协议，这些代码将 IP 头增加到传输数据中，同时也把收到的 IP 报文正确地转送到 TCP 层或 UDP 层。在 IP 层之下，是支持所有 Linux 网络应用的网络设备层，例如点到点协议(PPP)和以太网层。网络设备并非总代表物理设备，其中有一些(例如回送设备)则是纯粹的软件设备。网络设备与标准的 Linux 设备不同，它们不是通过 mknod 命令创建的，必须是底层软件找到并进行了初始化之后，这些设备才被创建并可用。因此只有当启动了正确设置了以太网设备驱动程序的内核后，才会有/dev/eth0 文件。ARP 协议位于 IP 层和支持地址解析的协议层之间。

4．BSD 套接字接口

它是一个通用接口，它不仅支持不同的网络结构，同时也是一个内部进程间通信机制。一个套接字描述了一个连结的端点，因此两个互联的进程都要有一个描述它们之间连接的套接字。可以把套接字看作是一种特殊的管道，只是这种管道对于所包含的数据量没有限制。Linux 支持套接字地址族中的多个，如下所示：

UNIX　　UNIX 域套接字
INET　　使用 TCP/IP 的因特网地址族
IPX　　IPX
APPLETALK　　APPLETALK
X25　　：X25

Linux 支持多种套接字类型。套接字类型是指创建套接字的应用程序所希望的通信服务类型。同一协议族可能提供多种服务类型，比如 TCP/IP 协议族提供的虚电路与数据报就是两种不同的通信服务类型。Linux BSD 支持如下几种套接字类型：

Stream：即流式套接字，它提供可靠的面向连接传输的数据流，保证数据传输过程中无丢失、无损坏和无冗余。INET 地址族中的 TCP 协议支持该套接字。

(1) Datagram：即数据报套接字。它提供数据的双向传输，但不保证消息报文的准确到达，即使消息能够到达，也无法保证其顺序正确，并可能有冗余或损坏。INET 地址族中的 UDP 协议支持该套接字。

(2) Raw：它是低于传输层的低级协议或物理网络提供的套接字类型．比如通过分析为以太网设备所创建的 RAW 套接字，可看到裸 IP 数据流。

(3) Reliable Delivered Messages：它类似于数据报套接字．但能保证数据的正确到达。

(4) Sequenced Packets：类似于 Stream 套接字，但它的报文大小是可变的。

(5) Packet：这是 Linux 对标准 BSD 套接字类型的扩展，它允许应用程序在设备层直接访问报文数据。

2.2 Linux 环境下的 socket 编程

2.2.1 socket 定义

前面已经讲过一些关于套接字(socket)的基本概念。网络的 socket 数据传输是一种特殊的 i/o，socket 也是一种文件描述符。socket 也具有一个类似于打开文件的函数调用 socket()，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过该 socket 实现的。常用的 socket 类型有两种：流式 Socket,SOCK STREAM 和数据报式 socket，SOCK DGRAM。

流式 socket 是一种针对面向连接的 socket，对应于有连接的 TCP 服务应用。数据报式 socket 是一种无连接的 socket，对应于无连接的 UDP 服务应用。。

2.2.2 socket 编程相关数据类型定义

在计算机中，数据存储有两种字节优先顺序：高位字节优先和低位字节优先。Internet 上数据以高位字节优先顺序在网络上传输，所以对于在内部以低位字节优先方式存储数据的机器，在 Internet 上传输数据时就需要进行转换。

(1)struct soekaddr

要讨论的第一个结构类型是：struct sockaddr，该类型是用来保存 socket 信息的。

```
struct sockaddr{
  unsigned short sa family；/*地址族，AF XXX*/
  char sa_data[14]；/*14 字节的协议地址*/
  };
  struct sockaddr_in 定义为：
  struct sockaddr_in{
  short int sin_family；/*地址族*/
  unsigned shortint sin_port；/*端口号*/
  struct in_addr sin_addr；/*IP 地址*/
  unsigned char sin_zero[8]；/*填充 0 以保持与 struct sockaddr 同样大小*/
  };
```

这个结构使用更为方便。sin_zero(它用来将 sockaddr_in 结构填充到与 struct sockaddr 同样的长度)应该用 bzero()或 memset()函数将其置为零。指向 sockaddr_in 的指针和指向 sockaddr 的指针可以相互转换。这意味着如果一个函数所需参数类型是 sockaddr 时，就可以在函数调用时将一个指向 sockaddr_in 的指针转换为指向 sockaddr 的指针；或者相反。

(2) 几个字节顺序转换函数：

(a) htons()：htons 表示“Host to Network Short”，主机地址字节顺序转向网络字节顺序(对短型数据操作)。

(b) htonl()：htonl 表示“Host to Network Long”，主机地址字节顺序转向网络字节顺序(对长型数据操作)。

(c) ntohs()：ntohs 表示“Network to Host Short”，网络字节顺序转向主机地址顺序(对短型数据操作)。

(d) ntohl()：ntohl 表示“NetworktoHost Long”，网络字节顺序转向主机地址顺序(对长型数据操作)。

在这里，h 表示“host”；n 表示“network”；S 表示“short”，短整型数据；l 表示“long”，长整型数据。

(3) Bind()函数原型为：

int bind(int sockfd，struct sockaddr *my_addr，int addrlen)；

sockfd 是一个 socket 描述符，my_addr 是一个指向包含有本机 IP 地址及端口号等信息的 sockaddr 类型的指针，addrlen 常被设置为 sizeof(struct sockaddr)。

最后，对于 bind 函数要说明的一点是，可以用下面的赋值实现自动获得本机 IP 地址和随机获取

一个没有被占用的端口号。

my_addr．sin_port=0；/*系统随机选择一个未被使用的端 121 号*/

my_addr．sin_addr．s_addr=INADDR_ANY；/*内核将自动填入本机 IP 地址*/

通过将 my_addr．sin_port 置为 0，函数会自动选择一个未占用的端口使用。同样，通过将 my_addr．sin_addr．s_addr 置为 INADDR_ANY，系统会自动填入本机 IP 地址。

bind()函数在成功被调用时返回 O；遇到错误时返回“-l”并将 errno 置为相应的错误号。当调用函数时，一般不要将端口号置为小于 1024 的值，因为 l～1024 是保留端口号，可以使用大于 1024 的任何一个没有被占用的端口号。使用它，一旦通过 socket 调用返回一个 socket 描述符，应该将该 socket 与本机上的一个端口相关联，即“绑定”(往往在设计服务器端程序时，需要调用该函数。随后就可以在该端口监听服务请求；而客户端一般无须调用该函数)。

(4) connect()函数

用来与远端服务器建立一个 TCP 连接，其函数原型为：

int connect(int sockfd，struct sockaddr *serv_addr，int addrlen)；

sockfd 是目的服务器的 socket 描述符；serv_addr 是包含目的机 IP 地址和端口号的指针。遇到错误时返回-1，并且 errno 中包含相应的错误码。进行客户端程序设计无须调用 bind()，因为这种情况下只需知道目的机器的 IP 地址，而客户通过哪个端口与服务器建立连接并不需要关心，内核会自动选择一个未被占用的端口供客户端使用。

(5) listen()函数：用来监听是否有服务请求。

在服务器端程序中，当 socket 与某一端口捆绑以后，就需要监听该端口，以便对到达的服务清求加以处理。

int listen(int sockfd，int backlog)；

sockfd 是 socket 系统调用返回的 socket 描述符；backlog 指定在请求队列中允许的最大请求数，进入的连接请求将在队列中等待 accept()函数接受它们(关于 accept()函数，请见下文)。backlog 对队列中等待服务请求的数目进行了限制，大多数系统缺省值为 20。当 listen 遇到错误时返回-1，错误码 errno 被置为相应的值。

一般服务器端程序通常按下列顺序进行函数调用：

socket()->bind()->listen()；/*接下来就是 accept()函数*/

accept() 函数：连接端口的服务请求。

当某个客户端试图与服务器监听的端口连接时，该连接请求将排队等待,服务器调用 accept()函数接受它。程序调用 accept()函数为该请求建立一个连接，accept()函数就返回一个新的 socket 描述符，供这个新连接使用。而服务器可以继续在以前的那个 socket 上监听，同时可以在新的 socket 描述符上进行数据 send()(发送)和 recv()(接收)操作。

int accept(int sockfd，void *addr，int *addrlen)；

sockfd 是被监听的 socket 描述符。addr 通常是一个指向 sockaddr_in 变量的指针，该变量用来存放提出连接请求服务的主机信息(某台主机从某个端口发出该请求)。addrlen 一般是一个整型指针变量，它指向的整型数值为 sizeof(struct sockaddr in)。发生错误时返回一个-1，并且设置相应的 errno 值。

(6) send()和 recv() 函数：数据的发送和接收。

这两个函数使用面向连接的 socket 来进行数据传输。

send()函数原型为：

int send(int Sockfd，const void *msg，int len，int flags)；

sockfd 是用来传输数据的 socket 描述符，msg 是一个指向要发送数据的指针。len 是以字节为

单位的数据长度，flags 一般情况下置为 O(关于该参数的用法可参照 man 手册)。

send()函数返回实际上发送出的字节数，可能会少于希望发送的数据，所以需对 send()的返回值进行测量。当 send()返回值与 len 不匹配时，应该对这种情况进行处理。

recv()函数原型为：

int recv(int sockfd，void *buf，int len，unsigned int flags)；

sockfd 是接受数据的 socket 描述符；buf 是存放接收数据的缓冲区；len 是缓冲的长度。flags 也被置为 0。函数 recv()返回实际上接收的字节数，或当出现错误时，返回-1 并置相应的 errno 值。

(7) sendto()和 recvfrom() 函数：利用数据报方式进行数据传输。

在无连接的数据报 socket 方式下，由于本地 socket 并没有与远端机器建立连接，所以在发送数据时应指明目的地址。sendto()函数原型为：

int sendto(int sockfd，eonst void *msg，int len，unsigned int flags，

const struet sockaddr *to，int tolen)；

该函数比 send()函数多了两个参数，to 表示目的机的 IP 地址和端口号信息，而 tolen 常常被赋值为 sizeof(struct sockaddr)。sendto 函数也返回实际发送的数据字节长度或在
出现发送错误时返回-1。

recvfrom()函数原型为：

int reevfrom(int sockfd，void *buf，int len，unsigned int flags，struct sockaddr *from，int *fromlen)；

from 是一个 struct sockaddr 类型的指针变量，该变量保存源机的 IP 地址及端口号。fromlen 常置为 sizeof(struct sockaddr)。当 recvfrom()返同时，fromlen 包含实际存入 from 中的数据字节数。

recvfrom()函数返回接收到的字节数，当出现错误时则返回-1，并设置相应的 errno 值。当数据报 socket 调用 connect()函数时，可以利用 send()和 recv()进行数据传输，但该 socket 仍然是数据报 socket，并且利用传输层的 UDP 服务。在发送或接收数据报文时，内核会自动为之加上目的地址和源地址信息。

(8) close()和 shutdown() 函数：结束数据传输。

当所有的数据操作结束后，可以调用 close()函数来释放该 socket，从而停止在该 socket 上的任何数据操作，如：

close(sockfd)；

也可以凋用 shutdown()函数来关闭该 socket。该函数允许用户只停止在某个方向上的数据传输，而其他方向上的数据传输继续进行。可以关闭某 socket 的写操作而允许继续在该 socket 上接受数据，直至读入所有数据，如：

int shutdown(int sockfd，int how)；

sockfd 的含义是显而易见的，而参数 how 可以设为下列值：

O——不允许继续接收数据。

I——不允许继续发送数据。

2——不允许继续发送和接收数据，均为允许则调用。

close()和 shutdown 在操作成功时返回 0，在出现错误时返回-1(并置相应 errno)。

(9) DNS：域名服务相关函数：

由于 IP 地址难以记忆和读写，为了读写记忆方便，所以人们常用域名来表示主机．这就需要进行域名和 IP 地址的转换。函数 gethostbyname()就是完成这种转换的，函数原型为：

struct hostent *gethostbyname(const chat *name)；

函数返回一种名为 hosten 的结构类型，它的定义如下：

```
struct hostent{
char *h_name；       /*主机的官方域名*/
char **h_aliases；     /*一个以 N1JLL 结尾的主机别名数组*/
int h_addrtype；      /*返回的地址类型，在 Internet 环境下为 AF_INE*/
int h_length；      /*地址的字节长度*/
char **h_addr_list；/*一个以 0 结尾的数组，包含该主机的所有地址*/
};
#define h_addr h_addr_list[0] /*在 h_addr_list 中的第一个地址*/
```

当 gethostname()调用成功时，返回指向 struct hosten 的指针，当调用失败时返回-1。当调用 gethostbymame 时，不能使用 perror()函数来输出错误信息，而应使用 herror()函数来输出。

2.2.3 面向连接的客户/服务器程序

一个建立分布式应用时最常用的范例便是客户机/服务器模型。在这种方案中客户应用程序向服务器程序请求服务，这种方式隐含了在建立客户机/服务器间通信的非对称性。客户机/服务器模型工作时要求有一套为客户机和服务器所共识的惯例，以保证服务能够被提供(或被接收)。这一套惯例包含一套协议，它必须在通信的两端都被实现。在非对称协议中，一方不可改变的认为是主机，另一方则是从机。当服务被提供时必然存在“客户进程”和“服务进程”。一个服务器通常在一个众所周知的地址监听对服务的请求。也就是说，服务器一直处于休眠状态，直到一个客户对这个服务的地址提出连接请求。在这个时刻，服务程序被唤醒并且为客户提供服务，对客户的请求做出了适当的反应。

通信需要一方作为服务器端，另一方作为客户端。由于我们在 pc 机上应用交叉编译环境，已经具有服务器端（PC 机）和客户端（目标板）了。所以做实验时，硬件环境不需要另外配置。

服务器端需要编写一个监听的服务例程，提示如下

```
int sock_fd,new_fd[5], numbytes,t;
 char buf[Max],filename[10];                    /*sock_fd,new_fd 是套接字描述*/
 int nsize,nnsize,i,j,filelength;
 struct sockaddr_in my_addr;                   /*服务器的地址结构体*/
 struct sockaddr_in their_addr;                /*客户机的地址结构体*/
 int sin_size;
 int allsize=0;
 FILE *fp;
 struct stat st;
 char szsendbuf[128],head[8],buf1[10],buf2[10],buf3[2],length;
 int *phead=head+4;
if((sock_fd=socket(AF_INET,SOCK_STREAM,0))==-1)      /*建立流式套接字描述符，错误?*/
    {perror("socket");
     exit(1);
    }

/*服务器结构体的地址赋初值*/
 my_addr.sin_family=AF_INET;
 my_addr.sin_port=htons(MYPORT);                    /*服务器的端口号 2000*/
 my_addr.sin_addr.s_addr=INADDR_ANY;
 bzero(&(my_addr.sin_zero),8);                    /*填充 0，凑齐长度*/
```

```
/*绑定*/
 if(bind(sock_fd,(struct sockaddr*)&my_addr,sizeof(struct sockaddr))==-1)
     {perror("bindB");                                    /*绑定失败*/
      exit(1);
      }
 if(listen(sock_fd,BACKLOG)==-1)                         /*监听端口是否有请求*/
     {perror("listen");                                   /*监听失败*/
      exit(1);
     }
```

客户端接受例程

```
void recvfile(char *bmpbuf,char *filename)
     {char ip[]="192.168.2.100";
      trans(ip,bmpbuf,filename);
     }
```

当然，这里的 ip 可以设置成任何对应的 ip，只要能 ping 得通。

2.2.4 实例

三. 实验内容和步骤

1. 调试通过以上提示的例程：能够通过 socket 发送和接收简单的文本文件。
2. 学有余力的同学，可以尝试结合触摸屏技术编写简易考试系统，构想如下：主机随机抽出一些问题，通过 socket，目标板上接收并显示问题，并提供选择项，用户在目标板上通过触摸屏进行触摸选择，触摸有效位置后目标板将选择项发回主机，进行记录，最后根据答案统计成绩记录在数据库中，同时也将发回目标板。

四. 思考题

1. 试着结合触摸屏、图形界面显示，构成以触摸屏输入选项、SOCKET 传输文件、最后以图形界面显示出该文件内容。