第020课 SPI裸板

目录

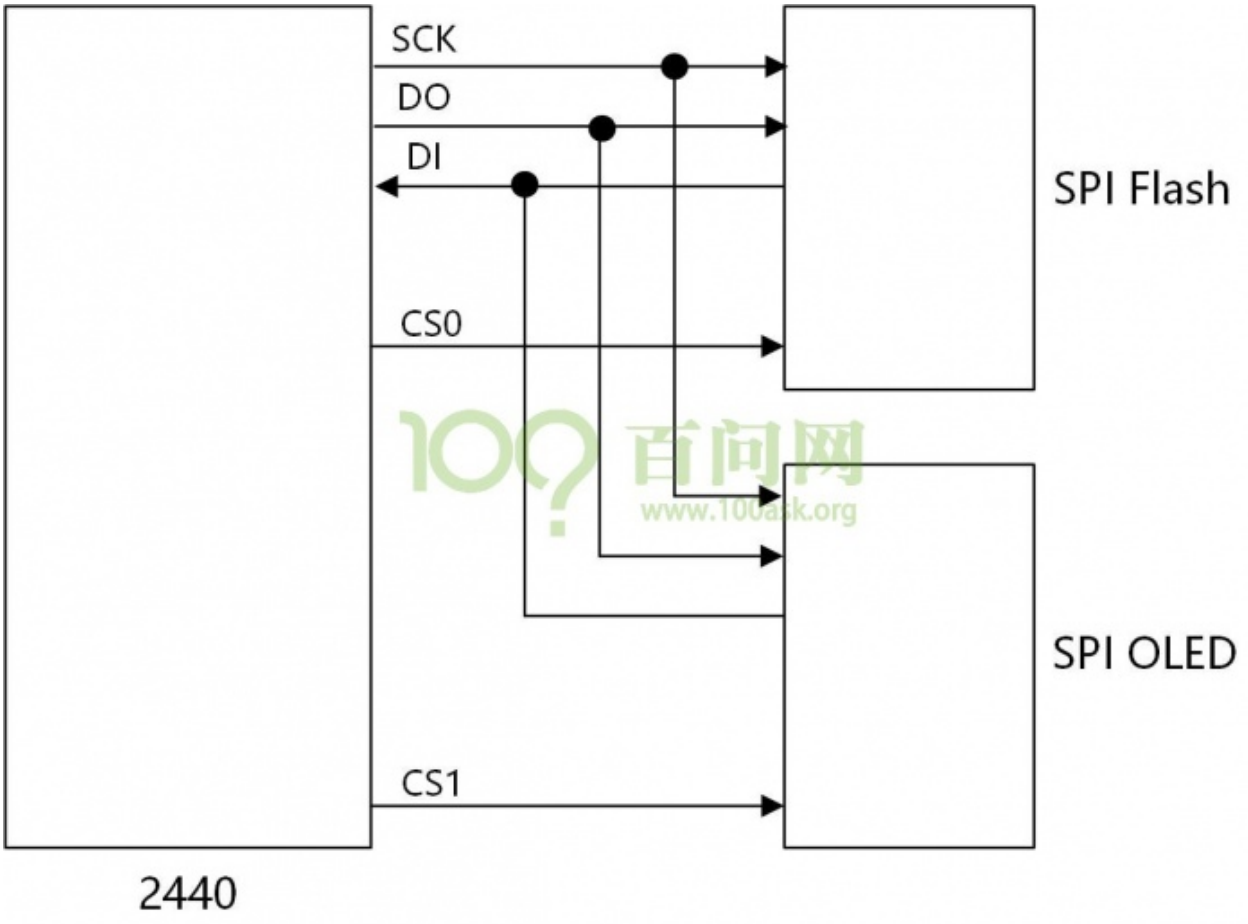
第001节_SPI协议介绍

市面上的开发板很少接有SPI设备，但是SPI协议在工作中经常用到。我们开发了SPI模块，上面有SPI Flash和SPI OLED。OLED就是一块显示器。

我们裸板程序会涉及两部分：

1. 用GPIO模拟SPI
2. 用S3C2440的SPI控制器

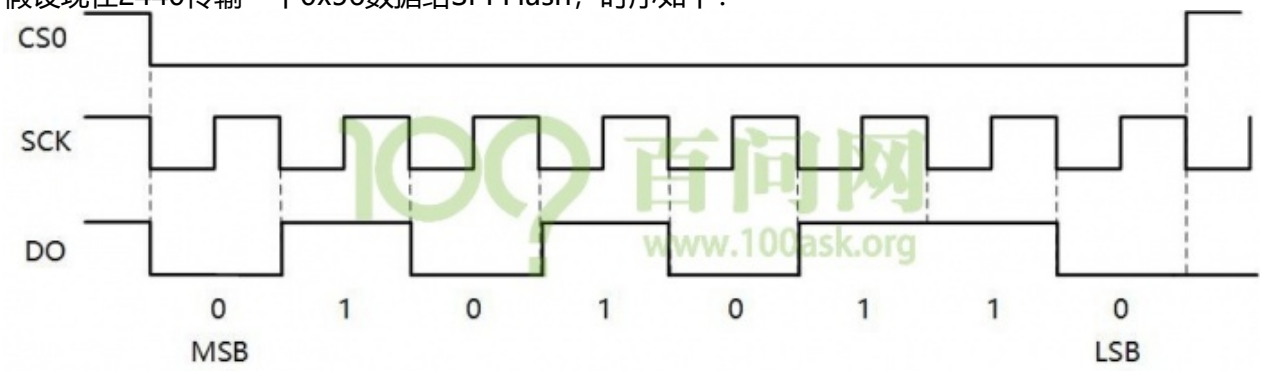
我们先介绍下SPI协议，硬件框架如下：



```
SCK：提供时钟
DO:作为数据输出
DI:作为数据输入
CS0/CS1:作为片选
```

同一时刻只能有一个SPI设备处于工作状态。

假设现在2440传输一个0x56数据给SPI Flash，时序如下：



首先CS0先拉

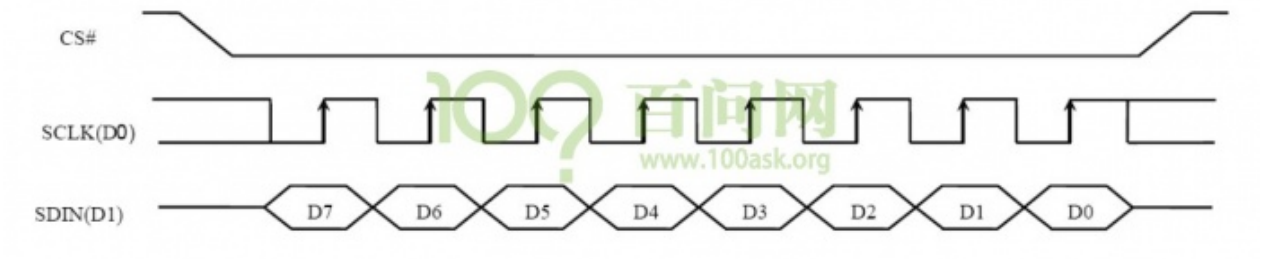
低选中SPI Flash，0x56的二进制就是0b0101 0110，因此在每个SCK时钟周期，DO输出对应的电平。 SPI Flash会在每个时钟周期的上升沿读取D0上的电平。

在SPI协议中，有两个值来确定SPI的模式。 CPOL:表示SPICLK的初始电平，0为电平，1为高电平 CPHA:表示相位，即第一个还是第二个时钟沿采样数据，0为第一个时钟沿，1为第二个时钟沿

CPOL	CPHA	模式	含义
0	0	0	初始电平为低电平，在第一个时钟沿采样数据
0	1	1	初始电平为低电平，在第二个时钟沿采样数据
1	0	2	初始电平为高电平，在第一个时钟沿采样数据
1	1	3	初始电平为高电平，在第二个时钟沿采样数据

我们常用的是模式0和模式3，因为它们都是在上升沿采样数据，不用去在乎时钟的初始电平是什么，只要在上升沿采集数据就行。

极性选什么？格式选什么？通常去参考外接的模块的芯片手册。比如对于OLED，查看它的芯片手册时序部分：



SCLK的初始电平我们并不需要关心，只要保证在上升沿采样数据就行。

第002节_使用GPIO实现SPI协议操作OLED

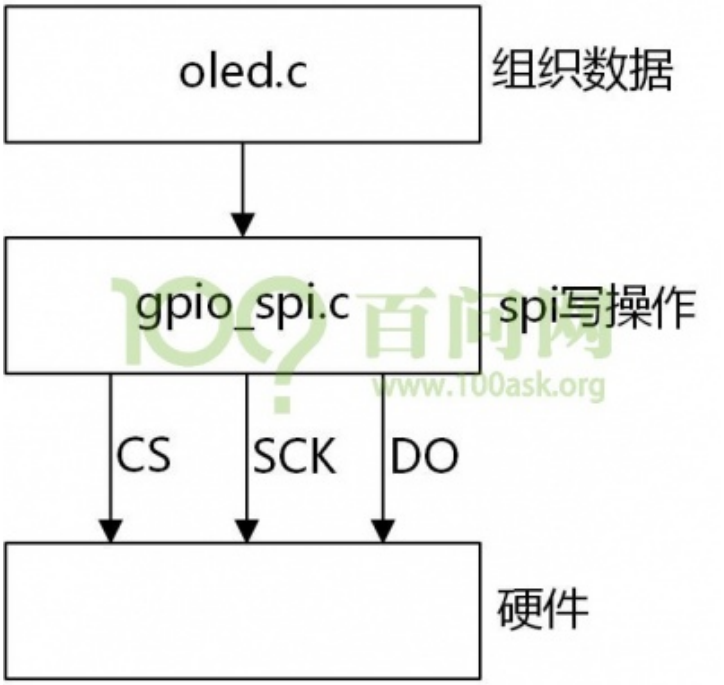
现在开始写代码，使用GPIO实现SPI协议操作。

我们现在想要操作OLED，通过三条线(SCK、DO、CS)与OLED相连，这里没有DI是因为2440只会向OLED传数据而不用接收数据。

我们要用GPIO来实现SOC向OLED写数据，这一层用gpio_spi.c来实现，负责发送数据。

对于OLED，有专门的指令和数据格式，要传输的数据内容，在oled.c这一层来实现，负责组织数据。

因此，我们需要实现以上两个文件。

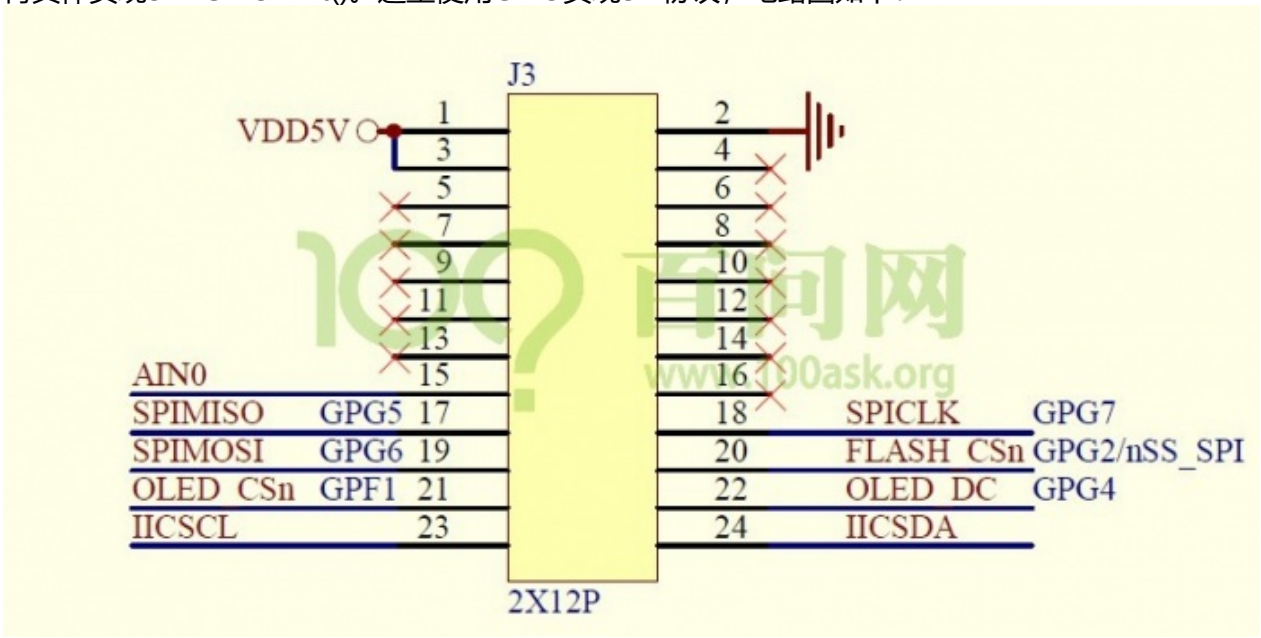


需要实现的函数：先SPI初始化SPIInt()，再初始化OLEDOLEDInit()，最后再显示OLEDPrint()。

新建一个gpio_spi.c文件，实现SPI初始化SPIInt()

```
void SPIInit(void)
{
    /* 初始化引脚 */
    SPI_GPIO_Init();
}
```

再具体实现SPI GPIO Init()。这里使用GPIO实现SPI协议，电路图如下：




```
GPF1作为OLED片选引脚，设置为输出；
GPG2作为FLASH片选引脚，设置为输出；
GPG4作为OLED的数据(Data)/命令(Command)选择引脚，设置为输出；
GPG5作为SPI的MISO，设置为输入；
GPG6作为SPI的MOSI，设置为输出；
GPG7作为SPI的时钟CLK，设置为输出；
```

```
/* 用GPIO模拟SPI */
static void SPI_GPIO_Init(void)
{
    /* GPF1 OLED_CSn output */
    GPFCON &= ~(3<<(1*2));
    GPFCON |= (1<<(1*2));
    GPFDAT |= (1<<1);

    /* GPG2 FLASH_CSn output
    * GPG4 OLED_DC   output
    * GPG5 SPIMISO   input
    * GPG6 SPIMOSI   output
    * GPG7 SPICLK    output
    */
    GPGCON &= ~((3<<(2*2)) | (3<<(4*2)) | (3<<(5*2)) | (3<<(6*2)) | (3<<(7*2)));
    GPGCON |= ((1<<(2*2)) | (1<<(4*2)) | (1<<(6*2)) | (1<<(7*2)));
    GPGDAT |= (1<<2);
}
```

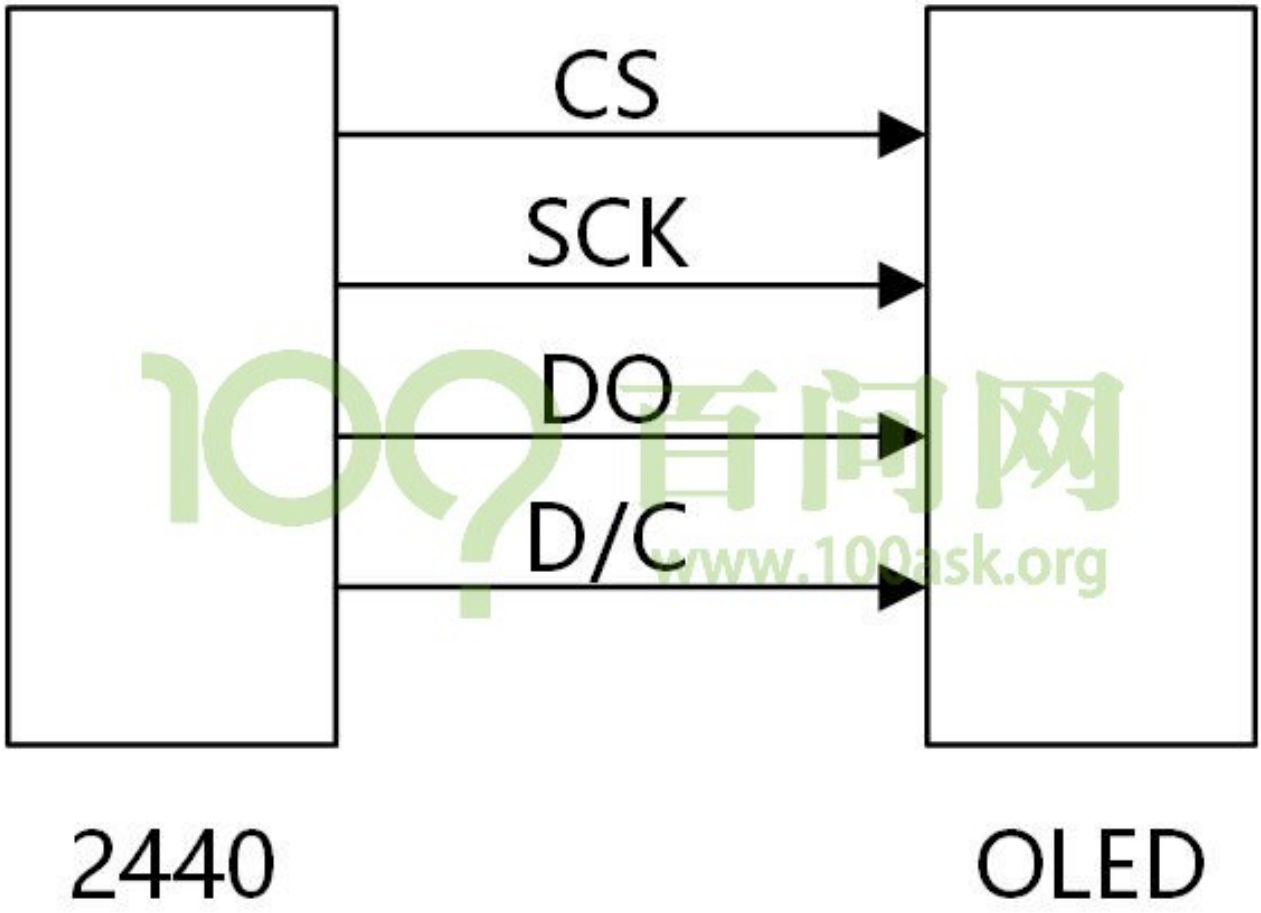
再新建一个oled.c文件，以实现初始化OLEDOLEDInit()

```
void OLEDInit(void)
{
/* 向OLED发命令以初始化 */
}
```

查阅OLED数据手册SPEC UG-2864TMBEG01.pdf可以得知其初始化流程和参考的初始化代码：

```
void OLEDInit(void)
{
    /* 向OLED发命令以初始化 */
    OLEDWriteCmd(0xAE); /*display off*/
    OLEDWriteCmd(0x00); /*set lower column address*/
    OLEDWriteCmd(0x10); /*set higher column address*/
    OLEDWriteCmd(0x40); /*set display start line*/
    OLEDWriteCmd(0xB0); /*set page address*/
    OLEDWriteCmd(0x81); /*contract control*/
    OLEDWriteCmd(0x66); /*128*/
    OLEDWriteCmd(0xA1); /*set segment remap*/
    OLEDWriteCmd(0xA6); /*normal / reverse*/
    OLEDWriteCmd(0xA8); /*multiplex ratio*/
    OLEDWriteCmd(0x3F); /*duty = 1/64*/
    OLEDWriteCmd(0xC8); /*Com scan direction*/
    OLEDWriteCmd(0xD3); /*set display offset*/
    OLEDWriteCmd(0x00);
    OLEDWriteCmd(0xD5); /*set osc division*/
    OLEDWriteCmd(0x80);
    OLEDWriteCmd(0xD9); /*set pre-charge period*/
    OLEDWriteCmd(0x1f);
    OLEDWriteCmd(0xDA); /*set COM pins*/
    OLEDWriteCmd(0x12);
    OLEDWriteCmd(0xdb); /*set vcomh*/
    OLEDWriteCmd(0x30);
    OLEDWriteCmd(0x8d); /*set charge pump enable*/
    OLEDWriteCmd(0x14);
}
```

因此我们还要先实现OLEDWriteCmd()函数，对于OLED，除了SPI的片选、时钟、数据引脚，还有一个数据/命令切换引脚。



这里的D/C即数据(Data)/命令(Command)选择引脚，它为高电平时，OLED即认为收到的是数据；它为低电平时，OLED即认为收到的是命令。

对于OLED，命令由开启/关闭显示、背光亮度等，具体有什么命令，可以查阅OLED的主控芯片手册SSD1306-Revision 1.1 (Charge Pump).pdf，在9 COMMAND TABLE 有相关命令的介绍。

因此，在编写OLEDWriteCmd()时，需要先设置为命令模式：

```
static void OLEDWriteCmd(unsigned char cmd)
{
    OLED_Set_DC(0); /* command */
    OLED_Set_CS(0); /* select OLED */

    SPISendByte(cmd);

    OLED_Set_CS(1); /* de-select OLED */
    OLED_Set_DC(1); /*  */
}
```

即：先设置为命令模式，再片选OLED，再传输命令，再恢复成原来的模式和取消片选。

片选函数和模式切换函数都比较简单，设置为对应的高低电平即可：

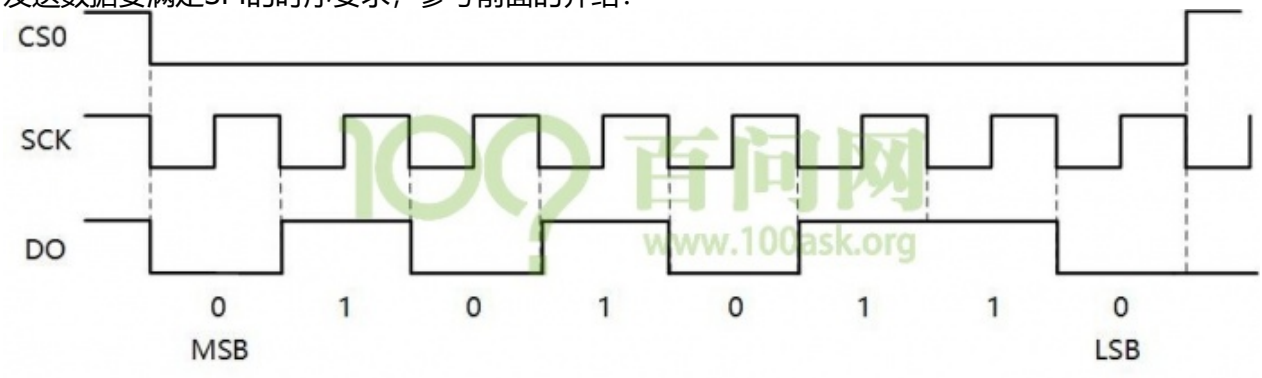
```
static void OLED_Set_DC(char val)
{
    if (val)
        GPGDAT |= (1<<4);
    else
        GPGDAT &= ~(1<<4);
}

static void OLED_Set_CS(char val)
{
    if (val)
```

```
        GPFDAT |= (1<<1);
    else
        GPFDAT &= ~(1<<1);
}
```

还剩下SPISendByte()函数，它属于SPI协议，放在gpio_spi.c里面：

```
void SPISendByte(unsigned char val)
{
    int i;
    for (i = 0; i < 8; i++)
    {
        SPI_Set_CLK(0);
        SPI_Set_DO(val & 0x80);
        SPI_Set_CLK(1);
        val <<= 1;
    }

}
```

发送数据要满足SPI的时序要求，参考前面的介绍：



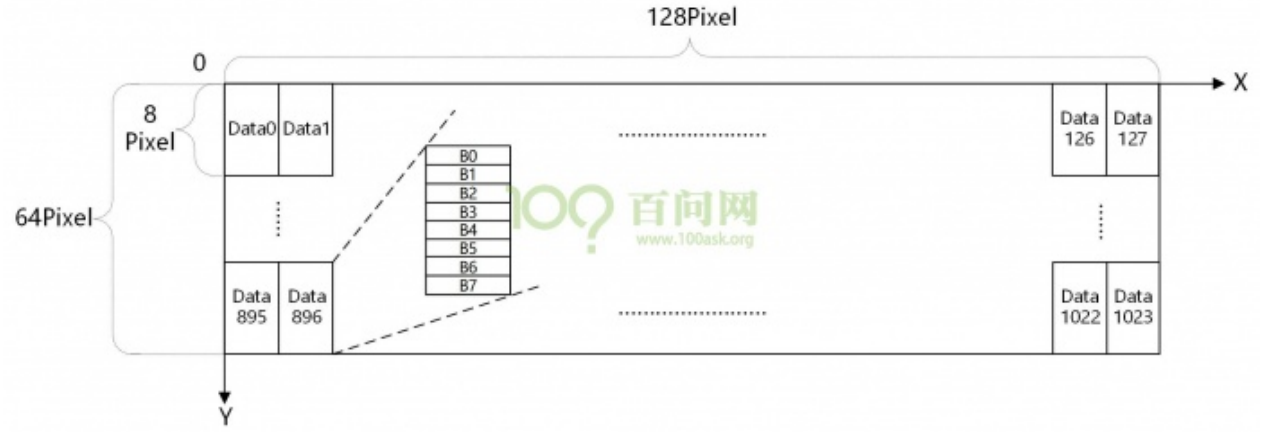
先设置CLK为低，然后数据引脚输出数据的最高位，然后CLK为高，在CLK这个上升沿中,OLED就读取了一位数据。接着左移一位，将原来的第7位移动到了第8位，重复8次，传输完成。

再完成SPI_Set_CLK()和SPI_Set_DO()：

```
static void SPI_Set_CLK(char val)
{
    if (val)
        GPGDAT |= (1<<7);
    else
        GPGDAT &= ~(1<<7);
}

static void SPI_Set_DO(char val)
{
    if (val)
        GPGDAT |= (1<<6);
    else
        GPGDAT &= ~(1<<6);
}
```

至此，SPI初始化和OLED初始化就基本完成了，接下来就是OLED显示部分。

先了解一下OLED显示的原理：



OLED长有128个像素，宽有64个像素，每个像素用一位来表示，为1则亮，为0则灭。

每一个字节数据Datax控制每列8个像素，在显存里面存放Data数据。

之后所需的操作就是把数据写到显存里面去，如何写到显存可以拆分成两个问题：

①怎么发地址

②怎么发数据

OLED主控的手册里介绍了三种地址模式，我们常用的是页地址模式(Page addressing mode (A[1:0]=10xb))，它把显存的64行分为8页，每页对应8行；选中某页后，再选择某列，然后就可以往里面写数据了，每写一个数据，地址就会加1，一直写到最右端的位置，他会自动跳到最左端。

通过命令来实现发送页地址和列地址，其中列地址分为两次发送，先发送低字节，再发送高字节。

假设每个字符数据大小为8x16，假如第一个字符位置为(page,col)，相邻的右边就是(page,col+8)，写满一行跳至下一行的坐标就是(page+2,col)。

```
/* page: 0-7
 * col : 0-127
 * 字符: 8x16象素
 */
void OLEDPrint(int page, int col, char *str)
{
    int i = 0;
    while (str[i])
    {
        OLEDPutChar(page, col, str[i]);
        col += 8;
        if (col > 127)
        {
            col = 0;
            page += 2;
        }
        i++;
    }
}
```

只要字符数组str[i]有数据，就调用OLEDPutChar(page, col, str[i])在指定位置显示第一个字符，然后位置向右移动一个字符的大小，如果遇到行尾，再进行换行，就这样依次显示完所有字符。

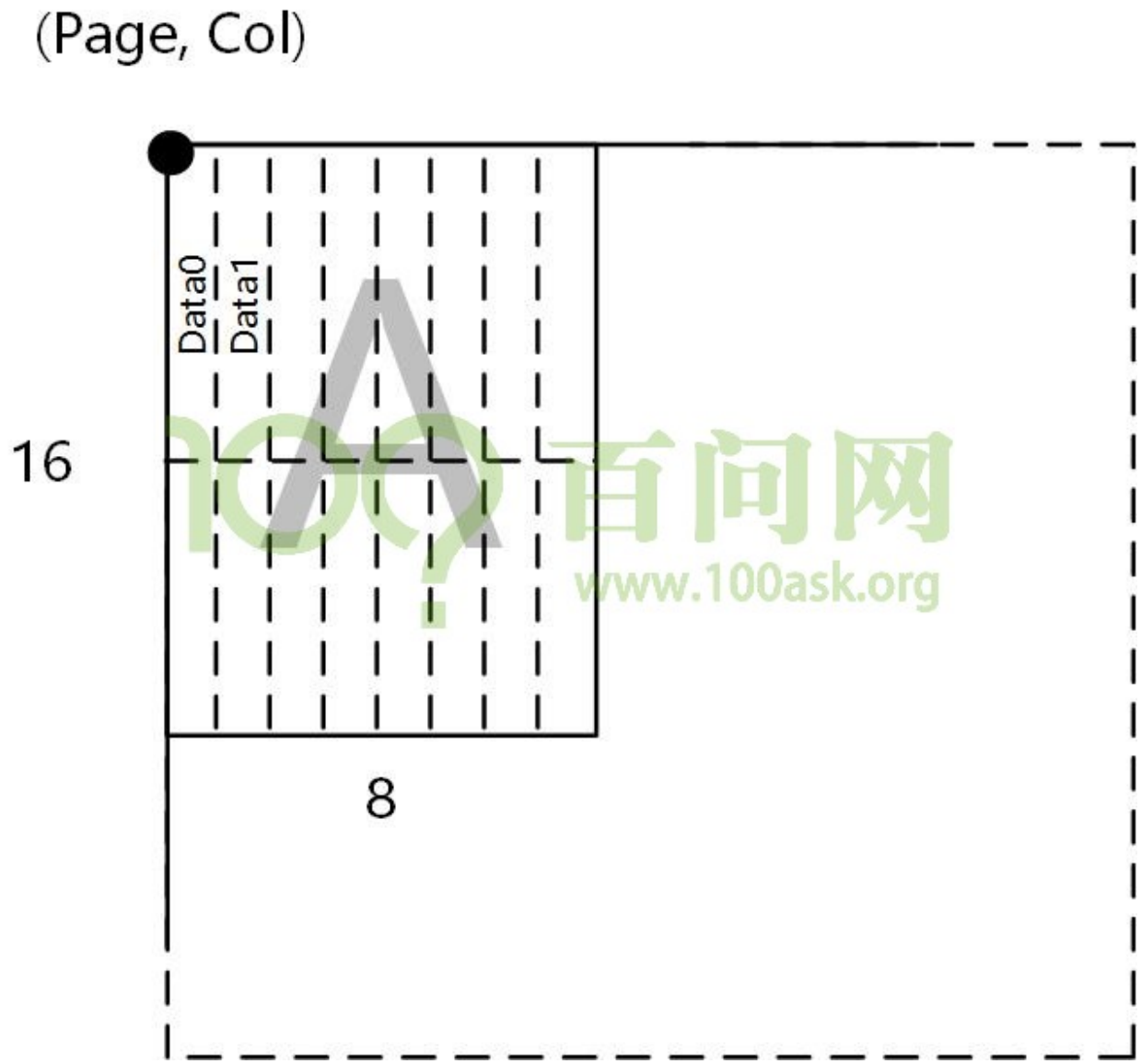
现在开始实现最重要的OLEDPutChar()函数。把一个字符在OLED上显示出来需要以下几个步骤：

a. 得到字模

b. 发给OLED

字模我们可以从网上搜索相关资料获取到，将字模的数组oled_asc2_8x16[95][16]放在oledfont.c里面，字符从空格开始，因此每次减去一个空格才是我们想要的字符。

如图所示一个字符，先以(page, col)为起点，显示8位数据，再换行，以(page+1, col)为起点显示8位数据。




```
/* page: 0-7
 * col : 0-127
 * 字符: 8x16象素
 */
void OLEDPutChar(int page, int col, char c)
{
    int i = 0;
    /* 得到字模 */
    const unsigned char *dots = oled_asc2_8x16[c - ' '];

    /* 发给OLED */
    OLEDSetPos(page, col);
    /* 发出8字节数据 */
    for (i = 0; i < 8; i++)
        OLEDWriteDat(dots[i]);

    OLEDSetPos(page+1, col);
    /* 发出8字节数据 */
    for (i = 0; i < 8; i++)
        OLEDWriteDat(dots[i+8]);
}
```

显示一个字符，就先获取字模数据，接着发出8字节数据，再换行发出8字节数。

再来实现OLED设置坐标位置函数，先设置page：

3. Addressing Setting Command Table											
D/C#	Hex	D7	D6	D5	D4	D3	D2	D1	D0	Command	Description
0	B0~B7	1	0	1	1	0	X_2	X_1	X_0	Set Page Start Address for Page Addressing Mode	Set GDDRAM Page Start Address (PAGE0~PAGE7) for Page Addressing Mode using X[2:0]. **Note** (1) This command is only for page addressing mode

D0~D2表示page数据，D3-D7是固定的值，因此每次写的命令内容为0xB0+page;

再设置列：

3. Addressing Setting Command Table											
D/C#	Hex	D7	D6	D5	D4	D3	D2	D1	D0	Command	Description
0	00~0F	0	0	0	0	X_3	X_2	X_1	X_0	Set Lower Column Start Address for Page Addressing Mode	Set the lower nibble of the column start address register for Page Addressing Mode using X[3:0] as data bits. The initial display line register is reset to 0000b after RESET. **Note** (1) This command is only for page addressing mode
0	10~1F	0	0	0	1	X_3	X_2	X_1	X_0	Set Higher Column Start Address for Page Addressing Mode	Set the higher nibble of the column start address register for Page Addressing Mode using X[3:0] as data bits. The initial display line register is reset to 0000b after RESET. **Note** (1) This command is only for page addressing mode

分两次发送，显示发送低字节4位，再发送高字节四位；

```
static void OLEDSetPos(int page, int col)
{
    OLEDWriteCmd(0xB0 + page); /* page address */

    OLEDWriteCmd(col & 0xf);   /* Lower Column Start Address */
    OLEDWriteCmd(0x10 + (col >> 4));   /* Lower Higher Start Address */
}
```

前面提到了OLED主控有三种地址模式，我们常用的是页地址模式(Page addressing mode (A[1:0]=10xb))，虽然这是默认的摸索，但还是设置一下比较好：

0 0	20 A[1:0]	0 *	0 *	1 *	0 *	0 *	0 *	0 A_1	0 A_0	Set Memory Addressing Mode	A[1:0] = 00b, Horizontal Addressing Mode A[1:0] = 01b, Vertical Addressing Mode A[1:0] = 10b, Page Addressing Mode (RESET) A[1:0] = 11b, Invalid

即先发送0x20，再设置A[1:0]=10：

```
static void OLEDSetPageAddrMode(void)
{
    OLEDWriteCmd(0x20);
    OLEDWriteCmd(0x02);
}
```

在显示中，一般都需一个清屏函数来清空当前可能显示的数据。清屏函数比较简单，往所有位置里面写0即可：

```
static void OLEDClear(void)
{
    int page, i;
    for (page = 0; page < 8; page ++)
    {
        OLEDSetPos(page, 0);
        for (i = 0; i < 128; i++)
            OLEDWriteDat(0);
```

```
    }
}
```

再把地址模式OLEDSetPageAddrMode()和清屏函数OLEDClear()放在SPI_GPIO_Init()里，在Makefile加上gpio_spi.o和oled.o。

最后在主函数里加上初始化和显示函数：

```
SPIInit();
OLEDInit();
OLEDPrint(0,0,"www.100ask.net, 100ask.taobao.com");
```

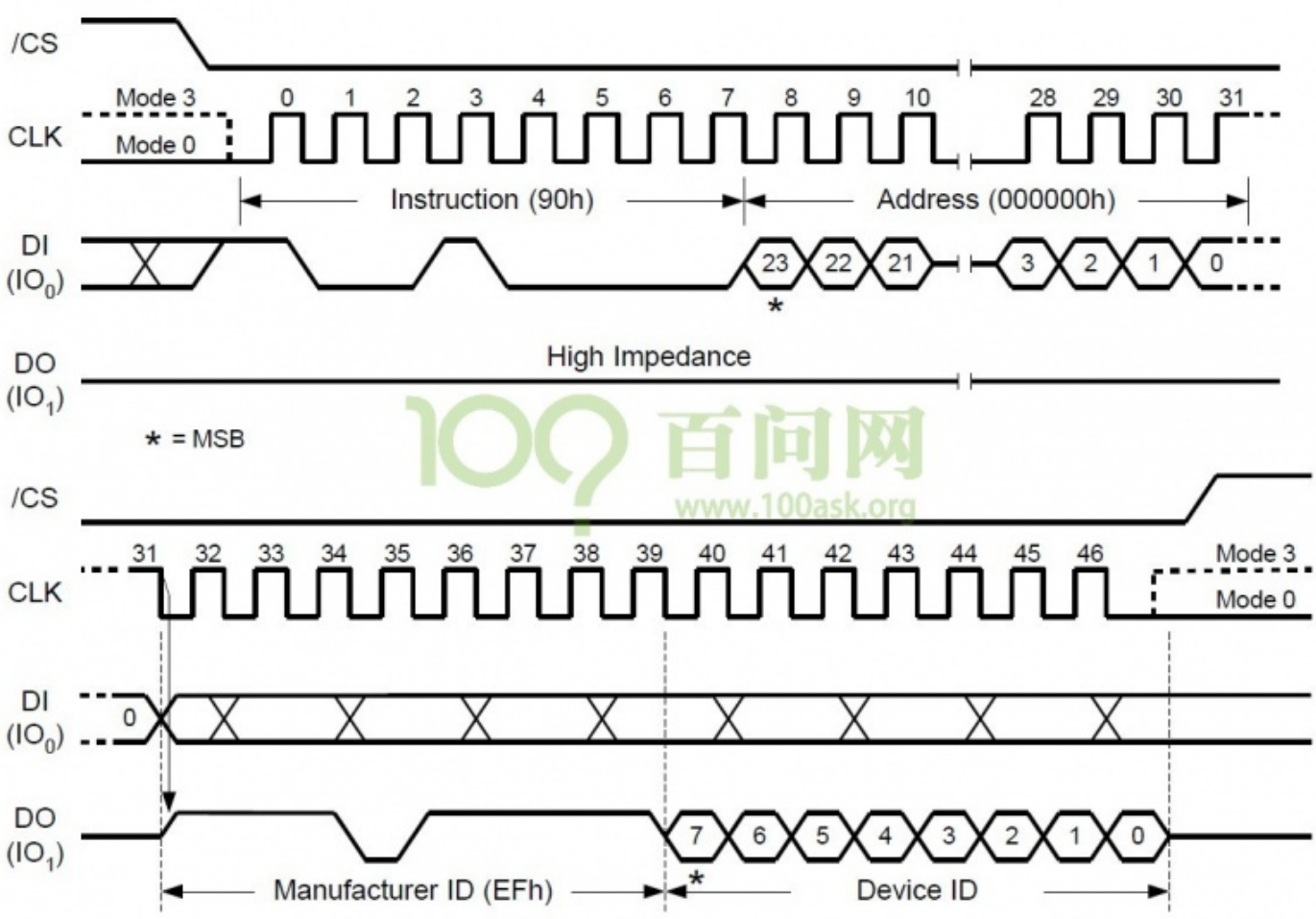
第003节_SPI_FLASH编程_读ID

这节讲解如何使用SPI操作Flash，我们在上节课的代码上进行修改，添加一个文件 spi_flash.c 和其头文件 spi_flash.h 。

我们先做一个最简单的spi操作，读取Flash的ID， SPIFlashID() 。

Flash的ID有厂家ID和设备ID，分别用pMID和pDID来保存。

根据Flash的芯片手册 W25Q16DV.pdf 可以知道需要先发出一个指令0x90，再发送24位的地址0，再读取数据前8位是设备ID，然后是8位设备ID。进行操作前必须要片选SPI Flash，片选完还是释放SPI Flash:




```
void SPIFlashReadID(int *pMID, int *pDID)
{
    SPIFlash_Set_CS(0); /* 选中SPI FLASH */

    SPISendByte(0x90);
```

```
    SPIFlashSendAddr(0);

    *pMID = SPIRecvByte();
    *pDID = SPIRecvByte();

    SPIFlash_Set_CS(1);
}
```

把其中的发送24地址封装成了一个函数 SPIFlashSendAddr()：

```
static void SPIFlashSendAddr(unsigned int addr)
{
    SPISendByte(addr >> 16);
    SPISendByte(addr >> 8);
    SPISendByte(addr & 0xff);
}
```

依次完成上面的子函数，先是SPI片选，上一节的原理图可以看到SPI Flash的片选是GPG2:

```
static void SPIFlash_Set_CS(char val)
{
    if (val)
        GPGDAT |= (1<<2);
    else
        GPGDAT &= ~(1<<2);
}
```

SPISendByte() 和前面OLED的是一样的，就不用写了，因此就只剩下 SPIRecvByte() ，放在 gpio_spi.c 里面实现：

```
unsigned char SPIRecvByte(void)
{
    int i;
    unsigned char val = 0;
    for (i = 0; i < 8; i++)
    {
        val <<= 1;
        SPI_Set_CLK(0);
        if (SPI_Get_DI())
            val |= 1;
        SPI_Set_CLK(1);
    }
    return val;
}
```

在每个时钟周期读取DI引脚上的值，对于SOC就是MISO引脚：

```
static char SPI_Get_DI(void)
{
    if (GPGDAT & (1<<5))
        return 1;
    else
        return 0;
}
```

至此，读取Flash的ID基本实现，最后在主函数里调用打印，分别在串口和OLED上显示：

```
    SPIFlashReadID(&mid, &pid);
    printf("SPI Flash : MID = 0x%02x, PID = 0x%02x\n\r", mid, pid);

    sprintf(str, "SPI : %02x, %02x", mid, pid);
    OLEDPrint(4,0,str);
```

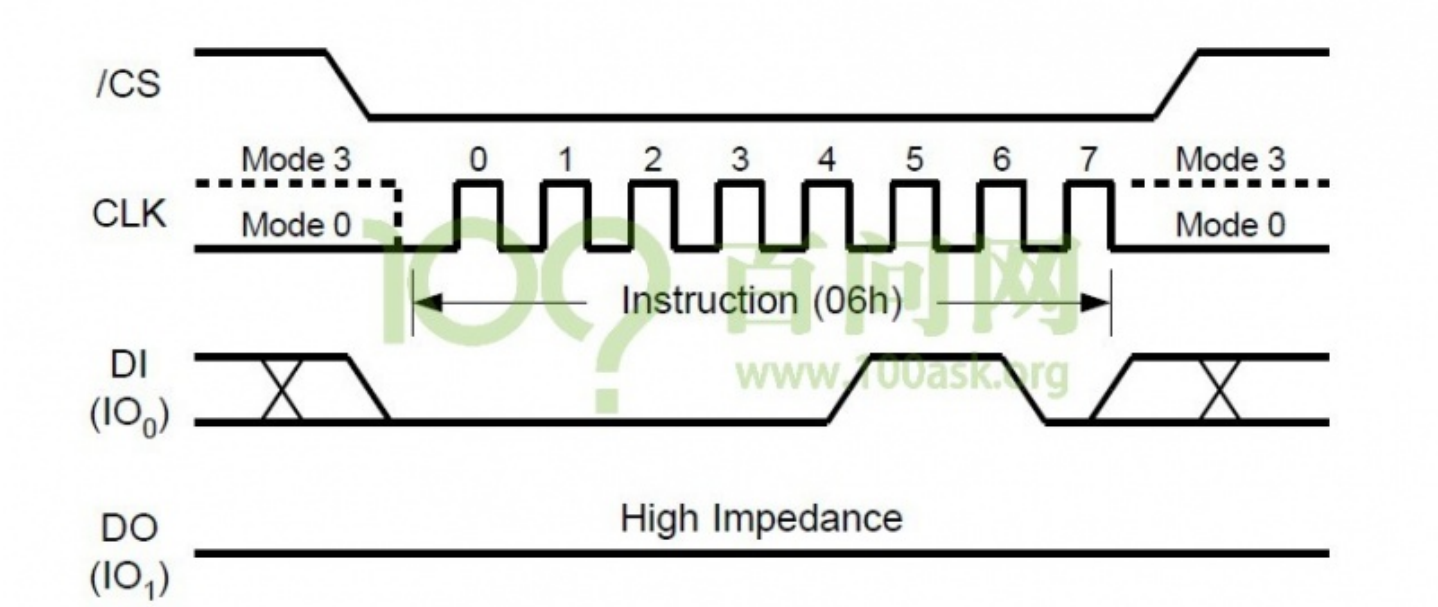
Makefile记得加上新生成的 spi_flash.o 。

第004节_SPI_FLASH编程_读写

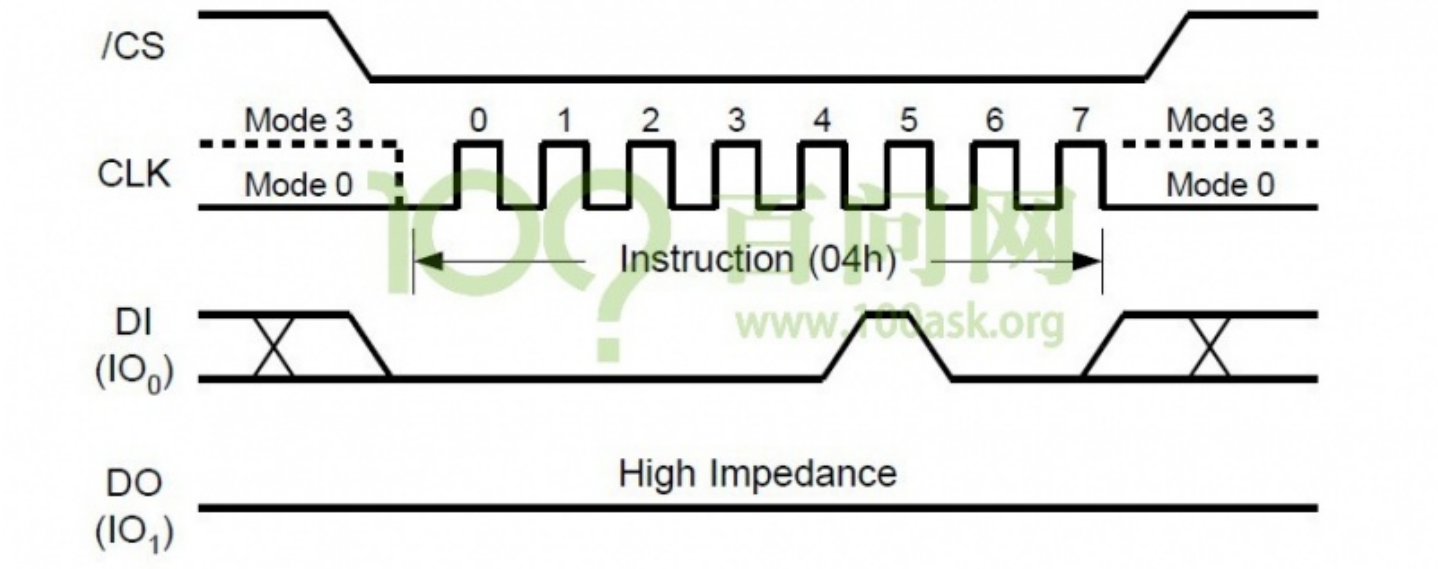
Flash作为一个存储芯片，最重要的就是存储和读取存储的数据，这节我们就实现Flash里数据的读写。 对于Flash，每次写操作需要的步骤如下:

1. 去保护（写使能、写状态寄存器）；
2. 擦除（写使能）
3. 编写入数据（写使能）

可以看出对于写操作，每次都要写使能，查阅芯片手册，可以看出写使能比较简单，只需要发送0x06命令即可：



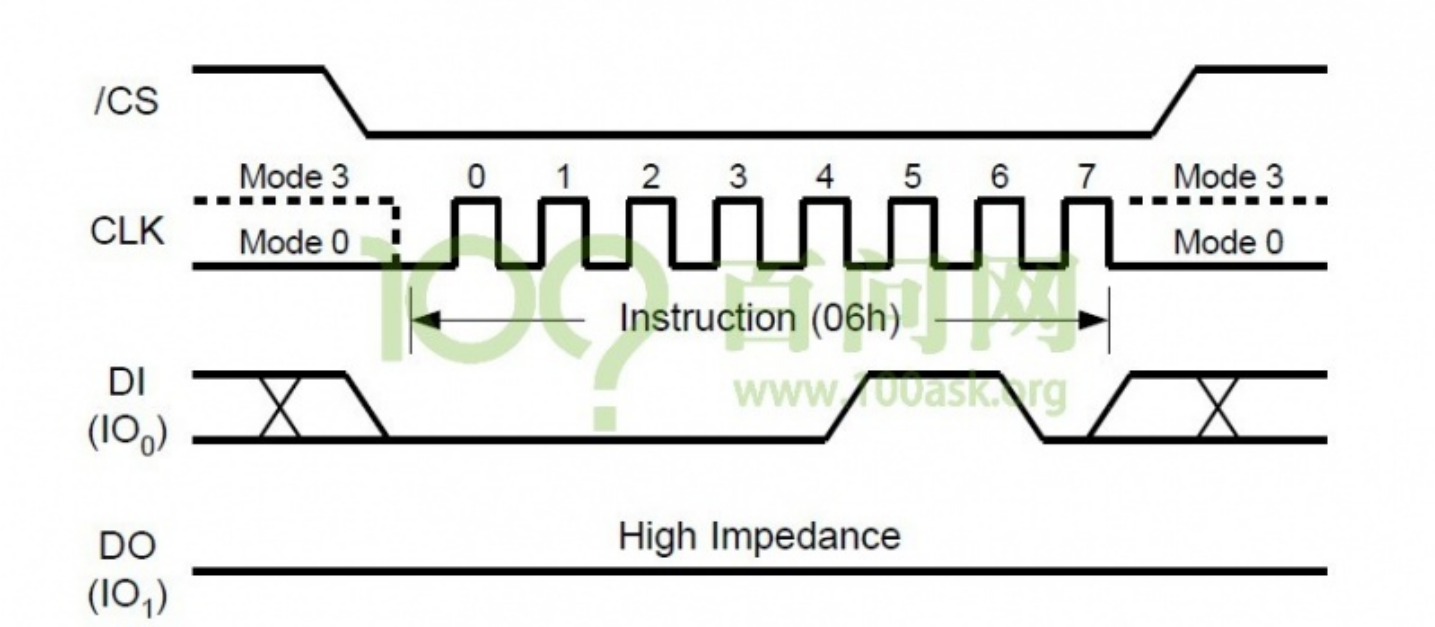
反之，写保护则是写入0x04:




```
static void SPIFlashWriteEnable(int enable)
{
    if (enable)
    {
        SPIFlash_Set_CS(0);
        SPISendByte(0x06);
        SPIFlash_Set_CS(1);
    }
    else
    {
        SPIFlash_Set_CS(0);
        SPISendByte(0x04);
```

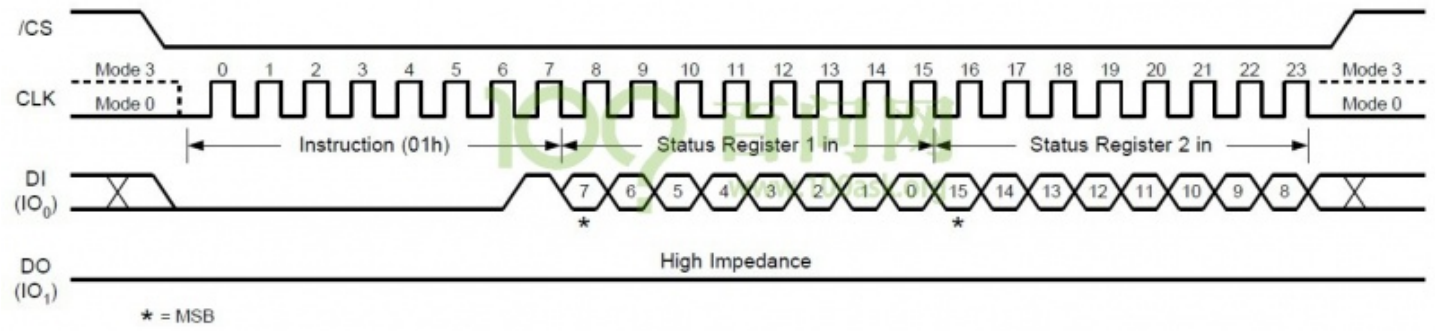
```
        SPIFlash_Set_CS(1);
    }
}
```

然后是读写状态寄存器，状态寄存器有两个，通过0x05读取状态寄存器1，通过0x35读取状态寄存器2：




```
static unsigned char SPIFlashReadStatusReg1(void)
{
    unsigned char val;
    SPIFlash_Set_CS(0);
    SPISendByte(0x05);
    val = SPIRecvByte();
    SPIFlash_Set_CS(1);
    return val;
}

static unsigned char SPIFlashReadStatusReg2(void)
{
    unsigned char val;
    SPIFlash_Set_CS(0);
    SPISendByte(0x35);
    val = SPIRecvByte();
    SPIFlash_Set_CS(1);
    return val;
}
```

写状态寄存器则是先发出0x01命令，再依次发送状态寄存器1、状态寄存器2：




```
static void SPIFlashWriteStatusReg(unsigned char reg1, unsigned char reg2)
{
    SPIFlashWriteEnable(1);

    SPIFlash_Set_CS(0);
    SPISendByte(0x01);
    SPISendByte(reg1);
```

```
    SPISendByte(reg2);
    SPIFlash_Set_CS(1);

    SPIFlashWaitWhenBusy();
}
```

写状态寄存器还需要去保护，默认的是发出`SPIFlashWriteEnable()`后，即可写状态寄存器，但为了确保万无一失，还是手动在将SRP1和SRP2设置为0，即将状态寄存器1的最高位清零和状态寄存器最低位清零：

SRP1	SRP0	/WP	Status Register	Description
0	0	X	Software Protection	/WP pin has no control. The Status register can be written to after a Write Enable instruction, WEL=1. [Factory Default]
0	1	0	Hardware Protected	When /WP pin is low the Status Register locked and can not be written to.
0	1	1	Hardware Unprotected	When /WP pin is high the Status register is unlocked and can be written to after a Write Enable instruction, WEL=1.
1	0	X	Power Supply Lock-Down	Status Register is protected and can not be written to again until the next power-down, power-up cycle.(1)
1	1	X	One Time Program(2)	Status Register is permanently protected and can not be written to.

Notes:
1. When SRP1, SRP0 = (1, 0), a power-down, power-up cycle will change SRP1, SRP0 to (0, 0) state.
2. This feature is available upon special order. Please contact Winbond for details.

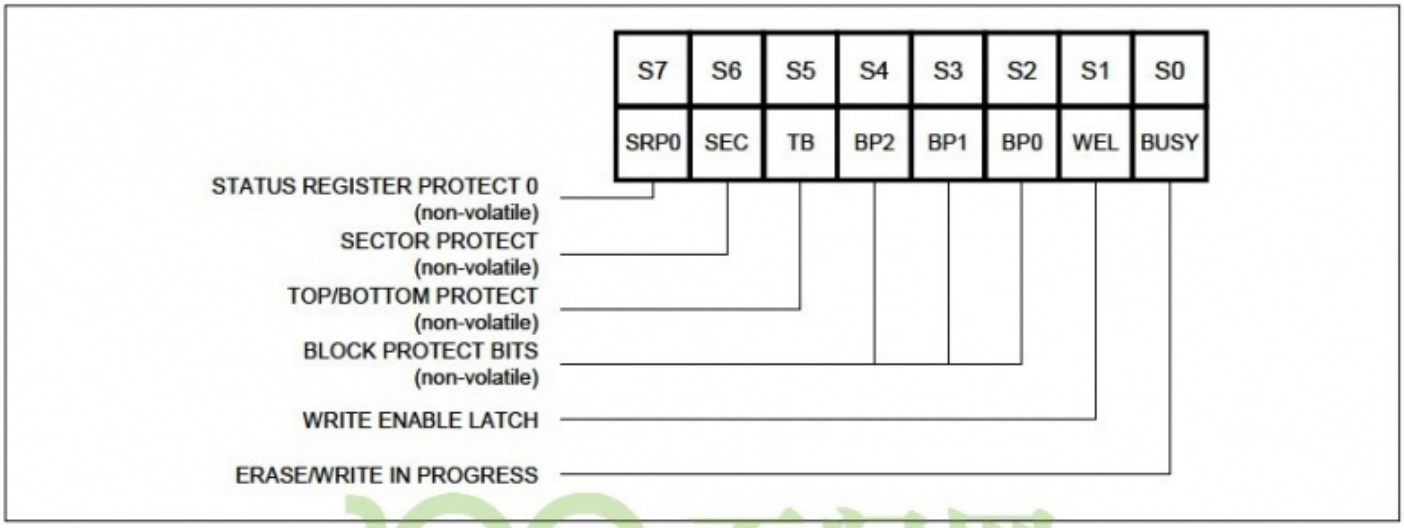


Figure 3a. Status Register-1

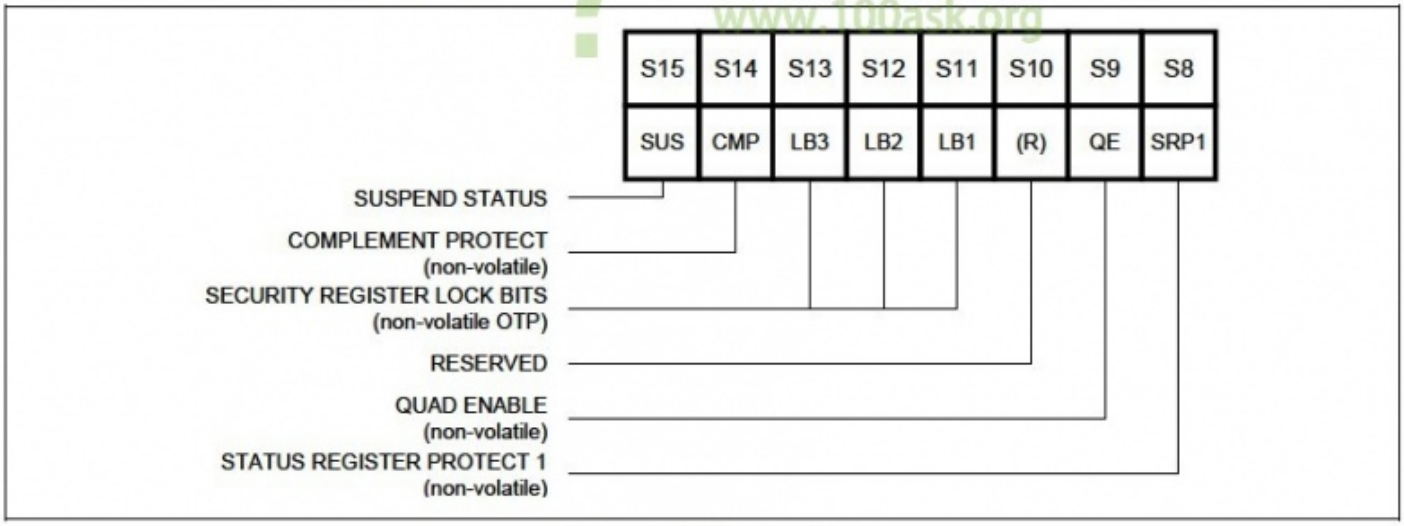


Figure 3b. Status Register-2

```
static void SPIFlashClearProtectForStatusReg(void)
{
    unsigned char reg1, reg2;

    reg1 = SPIFlashReadStatusReg1();
    reg2 = SPIFlashReadStatusReg2();

    reg1 &= ~(1<<7);
    reg2 &= ~(1<<0);

    SPIFlashWriteStatusReg(reg1, reg2);
}
```

Flash有两种保护机制，一个是保护状态寄存器，一种是保护存储数据，现在再来清除数据保护。 需要将CMP设置为0的同时，将BP0、BP1、BP2都设置为0：

7.1.11 Status Register Memory Protection (CMP = 0)

STATUS REGISTER(1)					W25Q16DV (16M-BIT) MEMORY PROTECTION(3)			
SEC	TB	BP2	BP1	BP0	PROTECTED BLOCK(S)	PROTECTED ADDRESSES	PROTECTED DENSITY	PROTECTED PORTION(2)
X	X	0	0	0	NONE	NONE	NONE	NONE

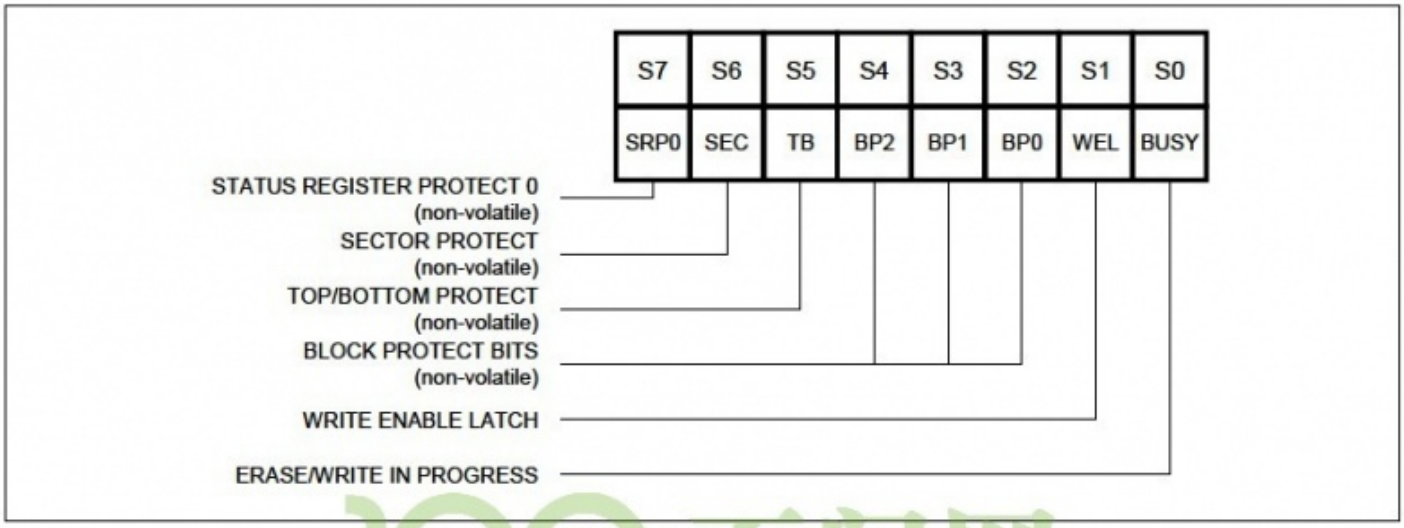


Figure 3a. Status Register-1

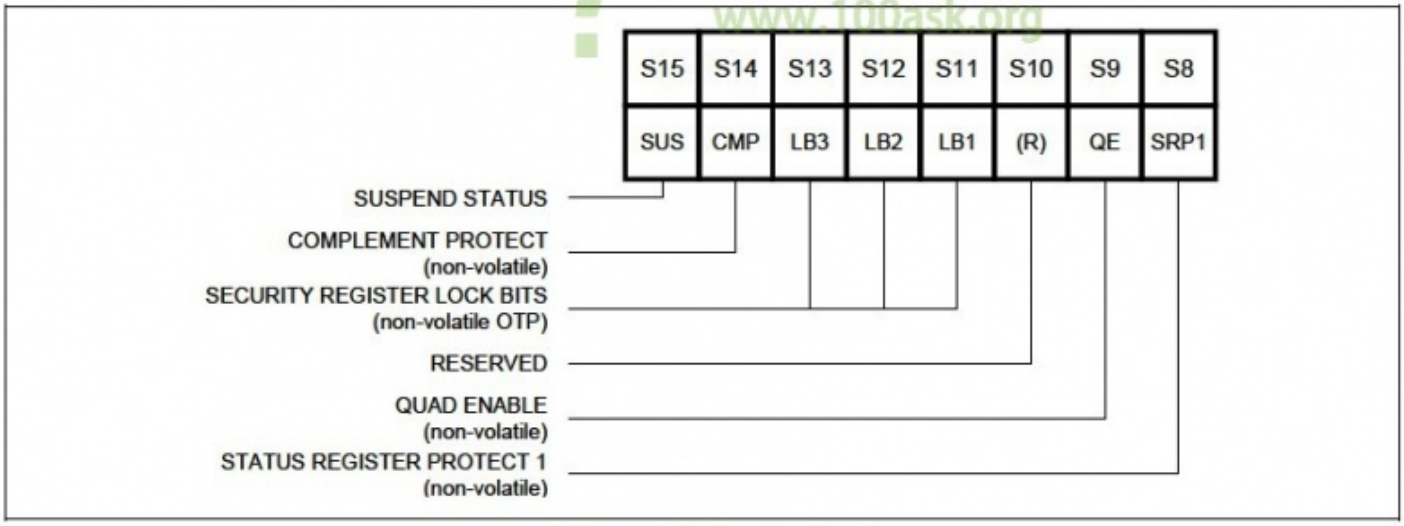


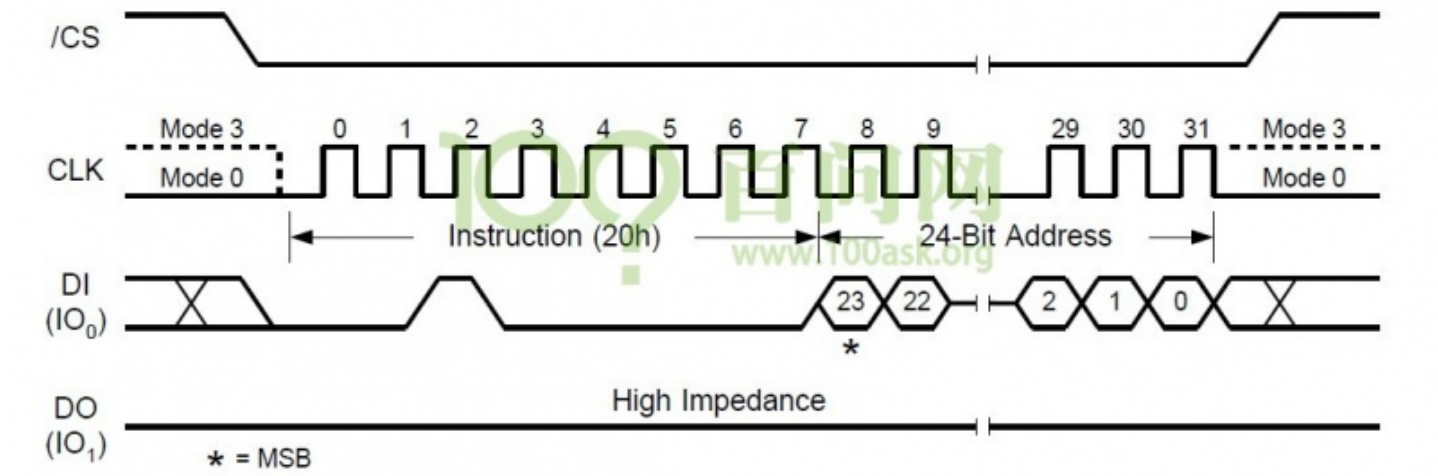
Figure 3b. Status Register-2

```
static void SPIFlashClearProtectForData(void)
{
    /* cmp=0,bp2,1,0=0b000 */
    unsigned char reg1, reg2;

    reg1 = SPIFlashReadStatusReg1();
    reg2 = SPIFlashReadStatusReg2();

    reg1 &= ~(7<<2);
    reg2 &= ~(1<<6);

    SPIFlashWriteStatusReg(reg1, reg2);
}
```

将两个清除写保护都放在一起，作为一个SPI Flash初始化函数：

```
void SPIFlashInit(void)
{
    SPIFlashClearProtectForStatusReg();
    SPIFlashClearProtectForData();
}
```

再来实现擦除，擦除命令需要先发一个0x20的命令，再发出24位的想擦除位置的地址：

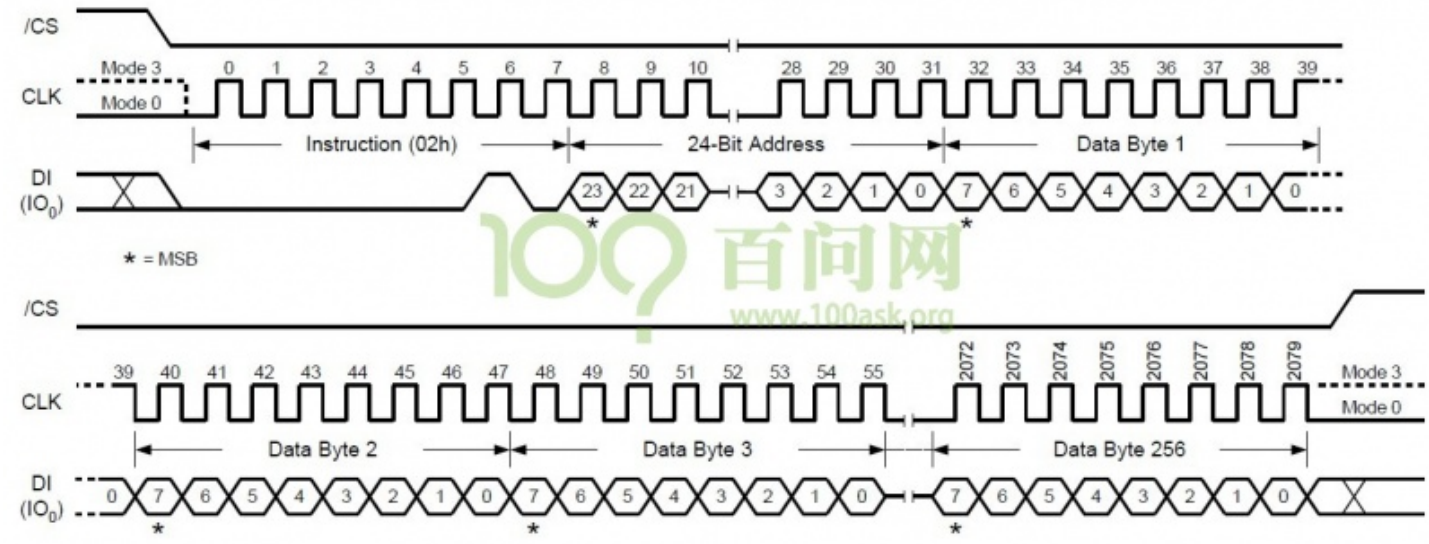



```
/* erase 4K */
void SPIFlashEraseSector(unsigned int addr)
{
    SPIFlashWriteEnable(1);

    SPIFlash_Set_CS(0);
    SPISendByte(0x20);
    SPIFlashSendAddr(addr);
    SPIFlash_Set_CS(1);

    SPIFlashWaitWhenBusy();
}
```

为了保证擦除成功，需要读取状态寄存器1的的第1位：

```
static void SPIFlashWaitWhenBusy(void)
{
    while (SPIFlashReadStatusReg1() & 1);
}
```

然后是烧写函数，先发命令0x02，再发出24位地址，最后再逐个发送数据：

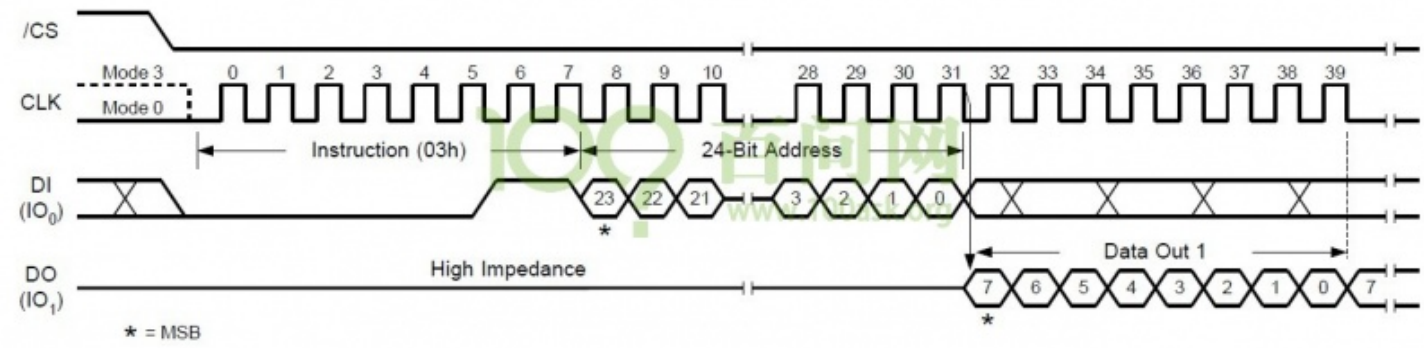



```
/* program */
void SPIFlashProgram(unsigned int addr, unsigned char *buf, int len)
{
    int i;

    SPIFlashWriteEnable(1);
```

```
    SPIFlash_Set_CS(0);
    SPISendByte(0x02);
    SPIFlashSendAddr(addr);

    for (i = 0; i < len; i++)
        SPISendByte(buf[i]);

    SPIFlash_Set_CS(1);

    SPIFlashWaitWhenBusy();

}
```

同前面的擦除操作一样，烧写操作也不是一定是实时的，需要读取状态标志位来判断是否完成。

读函数也是类似的操作，先发命令0x03，再发出24位地址，再逐个读取数据：




```
void SPIFlashRead(unsigned int addr, unsigned char *buf, int len)
{
    int i;

    SPIFlash_Set_CS(0);
    SPISendByte(0x03);
    SPIFlashSendAddr(addr);
    for (i = 0; i < len; i++)
        buf[i] = SPIRecvByte();

    SPIFlash_Set_CS(1);
}
```

至此，基本的Flash读写功能已经完成，在主函数调用擦除函数擦除4096这个扇区的数据，再往4096这个地方写入字符串，再从该地址读取出来，在串口和OLED打印出来：

```
SPIFlashEraseSector(4096);
SPIFlashProgram(4096, "100ask", 7);
SPIFlashRead(4096, str, 7);
printf("SPI Flash read from 4096:  %s\n\r", str);
OLEDPrint(4,0,str);
```

第005节_在OLED上显示ADC的值

这节我们在OLED显示ADC电压值，通过调节可调电阻，让ADC的值在屏幕上不断变化。

在JZ2440的主光盘的hardware里面有一个adc_ts触摸屏的程序，把里面的adc_ts.c和adc_ts.h提取出来放在本节视频待写的代码里面。

主函数调用的是Test_Adc.c进行测试adc，因此在里面加上打印和OLED显示函数。

```
/*
 * 测试ADC
 * 通过A/D转换，测量可变电阻器的电压值
 */
void Test_Adc(void)
{
```

```
    float vol0, vol1;
    int t0, t1;
    char buf[100];

    printf("Measuring the voltage of AIN0 and AIN1, press any key to exit\n\r");
    while (!awaitkey(0))    // 串口无输入，则不断测试
    {
        vol0 = ((float)ReadAdc(0)*3.3)/1024.0;  // 计算电压值
        vol1 = ((float)ReadAdc(1)*3.3)/1024.0;  // 计算电压值
        t0   = (vol0 - (int)vol0) * 1000;   // 计算小数部分，本代码中的printf无法打印浮点数
        t1   = (vol1 - (int)vol1) * 1000;   // 计算小数部分，本代码中的printf无法打印浮点数
        printf("AIN0 = %d.%-3dV    AIN1 = %d.%-3dV\r", (int)vol0, t0, (int)vol1, t1);

        sprintf(buf,"ADC: %d.%-3d, %d.%-3d", (int)vol0, t0, (int)vol1, t1);

        OLEDPrint(6, 0, buf);
    }
    printf("\n");
}
```

这里调用了一个awaitkey()函数，需要再复制adc_ts触摸屏的程序里serial.c的该函数到本工程里面。

```
/*
 * 接收字符，若有数据直接返回，否则等待规定的时间
 * 输入参数：
 *     timeout：等待的最大循环次数，0表示不等待
 * 返回值：
 *    0     ：无数据，超时退出
 *    其他值：串口接收到的数据
 */
unsigned char awaitkey(unsigned long timeout)
{
    while (!(UTRSTAT0 & RXDOREADY))
    {
        if (timeout > 0)
            timeout--;
        else
            return 0;   // 超时，返回0
    }

    return URXH0;       // 返回接收到的串口数据
}
```

修改Makefile，加入adc_ts.o，编译，报错，涉及除法操作，需要加入数学库：

```
LDFLASG  := -L $(shell dirname $(CC) $(CFLAGS) -print-libgcc-file-name) -lgcc
```

现在重新编译即可通过。

现在将IIC的的结果也在OLED上显示出来，在主函数添加如下代码：

```
    i2c_init();

    at24cxx_write(0, 0x55);
    data = at24cxx_read(0);

    OLEDClearPage(2);
    OLEDClearPage(3);

    if (data == 0x55)
        OLEDPrint(2,0,"I2C OK!");
    else
        OLEDPrint(2,0,"I2C Err!");
```

先初始化iic，在0地址写入0x55，然后再读取出来，判断是否与写入的一样，一样则打印OK，否则打印Err。

为了防止OLED出现之前显示的数据残留，需要再写一个清除Page的函数：

```
void OLEDClearPage(int page)
{
    int i;
    OLEDSetPos(page, 0);
```

```
    for (i = 0; i < 128; i++)
        OLEDWriteDat(0);
}
```

第006节_使用SPI控制器

前面我们都是通过GPIO管脚来实现的SPI通信，这节我们使用2440里面的GPIO控制器来实现SPI通信。

前面使用GPIO发送数据时，是手工的控制时钟线、数据线，我们使用SPI控制器的话，只需要 把数据写入寄存器，它就可以帮我自动那些时钟线和数据线，我们继续在上一节的基础上修改，添加一个文件s3c2440_spi.c和s3c2440_spi.h，同时修改Makefile，替换gpio_spi.c为s3c2440_spi.o。

从初始化函数开始，需要管脚初始化和SPI控制器初始化：

```
void SPIInit(void)
{
    /* 初始化引脚 */
    SPI_GPIO_Init();

    SPIControllerInit();
}
```

管脚初始化即需要把SPI相关的CLK、MOSI、MISO配置为对应的功能引脚：

```
static void SPI_GPIO_Init(void)
{
    /* GPF1 OLED_CSn output */
    GPFCON &= ~(3<<(1*2));
    GPFCON |= (1<<(1*2));
    GPFDAT |= (1<<1);

    /* GPG2 FLASH_CSn output
    * GPG4 OLED_DC   output
    * GPG5 SPIMISO
    * GPG6 SPIMOSI
    * GPG7 SPICLK
    */
    GPGCON &= ~((3<<(2*2)) | (3<<(4*2)) | (3<<(5*2)) | (3<<(6*2)) | (3<<(7*2)));
    GPGCON |= ((1<<(2*2)) | (1<<(4*2)) | (3<<(5*2)) | (3<<(6*2)) | (3<<(7*2)));
    GPGDAT |= (1<<2);
}
```

然后是SPI控制器的初始化，控制器的初始化可以参考芯片手册介绍的编程步骤：

PROGRAMMING PROCEDURE

When a byte data is written into the SPTDATn register, SPI starts to transmit if ENSCK and MSTR of SPCONn register are set. You can use a typical programming procedure to operate an SPI card.

To program the SPI modules, follow these basic steps:

1. Set Baud Rate Prescaler Register (SPPREn).
2. Set SPCONn to configure properly the SPI module.
3. Write data 0xFF to SPTDATn 10 times in order to initialize MMC or SD card.
4. Set a GPIO pin, which acts as nSS, low to activate the MMC or SD card.
5. Tx data ¡ Check the status of Transfer Ready flag (REDY=1), and then write data to SPTDATn.
6. Rx data(1): SPCONn's TAGD bit disable = normal mode
7. ¡ write 0xFF to SPTDATn, then confirm REDY to set, and then read data from Read Buffer.
8. Rx data(2): SPCONn's TAGD bit enable = Tx Auto Garbage Data mode
9. ¡ confirm REDY to set, and then read data from Read Buffer (then automatically start to transfer).
10. Set a GPIO pin, which acts as nSS, high to deactivate the MMC or SD card.

首先是设置波特率，要根据外设所能接受的范围来设置，比如查阅OLED的芯片手册得知其时钟最小值为100ns，即最小为10MHz；Flash时钟支持最大104MHz，为了代码简单，就直接取10MHz，根据等式推出寄存器值：

```
Baud rate = PCLK / 2 / (Prescaler value + 1)
10 = 50 / 2 / (Prescaler value + 1)
Prescaler value = 1.5 = 2
```

实际的波特率为：50/2/3=8.3MHz

根据参考流程，接下来设置SPI控制寄存器：

PROGRAMMING PROCEDURE

When a byte data is written into the SPTDATn register, SPI starts to transmit if ENSCK and MSTR of SPCONn register are set. You can use a typical programming procedure to operate an SPI card.

To program the SPI modules, follow these basic steps:

1. Set Baud Rate Prescaler Register (SPPREn).
2. Set SPCONn to configure properly the SPI module.
3. Write data 0xFF to SPTDATn 10 times in order to initialize MMC or SD card.
4. Set a GPIO pin, which acts as nSS, low to activate the MMC or SD card.
5. Tx data ¡ Check the status of Transfer Ready flag (REDY=1), and then write data to SPTDATn.
6. Rx data(1): SPCONn's TAGD bit disable = normal mode
7. ¡ write 0xFF to SPTDATn, then confirm REDY to set, and then read data from Read Buffer.
8. Rx data(2): SPCONn's TAGD bit enable = Tx Auto Garbage Data mode
9. ¡ confirm REDY to set, and then read data from Read Buffer (then automatically start to transfer).
10. Set a GPIO pin, which acts as nSS, high to deactivate the MMC or SD card.

```
[6:5]设置为查询模式: 00 polling mode
[4]设置时钟使能: 1 = enable
[3]设置为主机模式: 1 = master
[2]设置无数据时时钟为低电平: 0
[1]设置工作模式为模式A: 0 = format A
[0]设置发送数据时无需读取数据: 0 = normal mode
```

```c
static void SPIControllerInit(void)
{
    /* OLED  : 100ns, 10MHz
    * FLASH : 104MHz
    * 取10MHz
    * 10 = 50 / 2 / (Prescaler value + 1)
    * Prescaler value = 1.5 = 2
    * Baud rate = 50/2/3=8.3MHz
    */
    SPPRE0 = 2;
    SPPRE1 = 2;

    /* [6:5] : 00, polling mode
    * [4]   : 1 = enable
    * [3]   : 1 = master
    * [2]   : 0
    * [1]   : 0 = format A
    * [0]   : 0 = normal mode
    */
    SPCON0 = (1<<4) | (1<<3);
    SPCON1 = (1<<4) | (1<<3);

}
```

发送数据时，先检查状态寄存器，判断发送/接收数据是否准备好了，准备好后就把数据放在寄存器SPTDAT1里，SPI控制器就自己控制时序把数据自动发送出去了。

```c
void SPISendByte(unsigned char val)
{
    while (!(SPSTA1 & 1));
    SPTDAT1 = val;
}
```

接收数据时，先写0xFF到寄存器SPTDAT1，再检查状态寄存器，判断发送/接收数据是否准备好了，准备好后就读取寄存器SPTDAT1，读取出来的就是接收到的数据。

```
unsigned char SPIRecvByte(void)
{
    SPTDAT1 = 0xff;
    while (!(SPSTA1 & 1));
    return SPRDAT1;
}
```

第007节_移植到MINI2440_TQ2440

前面在JZ2440上操作了SPI Flash和OLED，这节视频是件前面的代码移植到MINI2440和TQ2440上，如果你使用的是JZ2440，本节视频就不用看了。

MINI2440和TQ2440上的SPI管脚是完全一样的，因此只需移植一个，两者就通用了，先移植GPIO模式版本的，复制前面 04th_spi_i2c_adc_jz2440_ok_020_005 里的代码，复制后的新的命名为 06th_spi_i2c_adc_mini2440_tq2440_gpio_020_007 。

修改 gpio_spi.c ，里面的管脚几乎都变化了，因此需要改 SPI_GPIO_Init() ：

```
static void SPI_GPIO_Init(void)
{
    /* GPG1 OLED_CSn output
    * GPG10 FLASH_CSn output
    */
    GPGCON &= ~((3<<(1*2)) | (3<<(10*2)));
    GPGCON |= (1<<(1*2)) | (1<<(10*2));
    GPGDAT |= (1<<1) | (1<<10);

    /*
    * GPF3  OLED_DC   output
    * GPE11 SPIMISO   input
    * GPE12 SPIMOSI   output
    * GPE13 SPICLK    output
    */
    GPFCON &= ~(3<<(3*2));
    GPFCON |= (1<<(3*2));

    GPECON &= ~((3<<(11*2)) | (3<<(12*2)) | (3<<(13*2)));
    GPECON |= ((1<<(12*2)) | (1<<(13*2)));
}
```

CLK引脚也变了，修改如下：

```
static void SPI_Set_CLK(char val)
{
    if (val)
        GPEDAT |= (1<<13);
    else
        GPEDAT &= ~(1<<13);
}
```

SPI的MOSI和MISO也要变化如下：

```
static void SPI_Set_DO(char val)
{
    if (val)
        GPEDAT |= (1<<12);
    else
        GPEDAT &= ~(1<<12);
}

static char SPI_Get_DI(void)
{
```

```
    if (GPEDAT & (1<<11))
        return 1;
    else
        return 0;
}
```

对于SPI Flash需要修改其片选引脚，修改 spi_flash.c 里面的片选函数如下:

```
static void SPIFlash_Set_CS(char val)
{
    if (val)
        GPGDAT |= (1<<10);
    else
        GPGDAT &= ~(1<<10);
}
```

重新编译烧写，测试正常。 再移植SPI控制器版本的，复制前面 05th_spi_i2c_adc_jz2440_spi_controller_020_006 里的代码，复制后的新的命名为 07th_spi_i2c_adc_mini2440_tq2440_spi_controller_020_007 。 同样的首先修改GPIO初始化，修改为配套引脚:

```
static void SPI_GPIO_Init(void)
{
    /* GPG1 OLED_CSn output
    * GPG10 FLASH_CSn output
    */
    GPGCON &= ~((3<<(1*2)) | (3<<(10*2)));
    GPGCON |= (1<<(1*2)) | (1<<(10*2));
    GPGDAT |= (1<<1) | (1<<10);

    /*
    * GPF3  OLED_DC   output
    * GPE11 SPIMISO
    * GPE12 SPIMOSI
    * GPE13 SPICLK
    */
    GPFCON &= ~(3<<(3*2));
    GPFCON |= (1<<(3*2));

    GPECON &= ~((3<<(11*2)) | (3<<(12*2)) | (3<<(13*2)));
    GPECON |= ((2<<(11*2)) | (2<<(12*2)) | (2<<(13*2)));
}
```

SPI Flash使用的是SPI0，因此将 SPTDAT1 改为 *SPTDAT1* :

```
void SPISendByte(unsigned char val)
{
    while (!(SPSTA0 & 1));
    SPTDAT0 = val;
}

unsigned char SPIRecvByte(void)
{
    SPTDAT0 = 0xff;
    while (!(SPSTA0 & 1));
    return SPRDAT0;
}
```

修改SPI Flash的片选引脚:

```
static void SPIFlash_Set_CS(char val)
{
    if (val)
        GPGDAT |= (1<<10);
    else
        GPGDAT &= ~(1<<10);
}
```

最后是OLED的片选和数据/命令控制引脚：

```
static void OLED_Set_DC(char val)
{
    if (val)
        GPFDAT |= (1<<3);
    else
        GPFDAT &= ~(1<<3);
}

static void OLED_Set_CS(char val)
{
    if (val)
        GPGDAT |= (1<<1);
    else
        GPGDAT &= ~(1<<1);
}
```

重新编译、烧写，测试。

《《所有章节目录》》