零死角玩转STM32

与野火同行 乐意惬无边

1 原创教程，完全开源。

2 通俗易懂，详尽解读。

3 由浅入深，结合实操。

4 配套板子，全面玩转。

5 强强联合，不断更新。

野火团队 Wild Fire Team

0、 友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，与野火同行，乐意惬无边。

另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

4、液晶触摸画板

4.1 实验简介

本实验向大家介绍如何使用 STM32 的 FSMC 接口驱动 LCD 屏，及使用触摸屏控制器检测触点坐标。

4.2 LCD 控制器简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，因而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

因为 STM32 内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路(液晶屏和触摸屏的驱动电路是独立的)，STM32 芯片通过驱动芯片来控制液晶屏和触摸屏。以野火 3.2 寸液晶屏(240*320)为例，它使用 *ILI9341* 芯片控制液晶屏，通过 *TSC2046* 芯片控制触摸屏。

4.2.1 ILI9341 控制器结构

液晶屏的控制芯片内部结构非常复杂，见**错误！未找到引用源。**。最主要的是位于中间 GRAM(Graphics RAM)，可以理解为*显存*。GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。

框图的左上角为 ILI9341 的主要控制信号线和配置引脚，根据其不同状态设置可以使芯片工作在不同的模式，如每个像素点的位数是 6、16 还是 18 位；使用 SPI 接口还是 8080 接口与 MCU 进行通讯；使用 8080 接口的哪种模式。MUC 通过 SPI 或 8080 接口与 ILI9341 进行通讯，从而访问它的控制寄存器(CR)、地址计数器(AC)、及 GRAM。

在 GRAM 的左侧还有一个 LED 控制器(LED Controller)。LCD 为非发光性的显示装置，它需要借助背光源才能达到显示功能，LED 控制器就是用来控制液晶屏中的 LED 背光源。

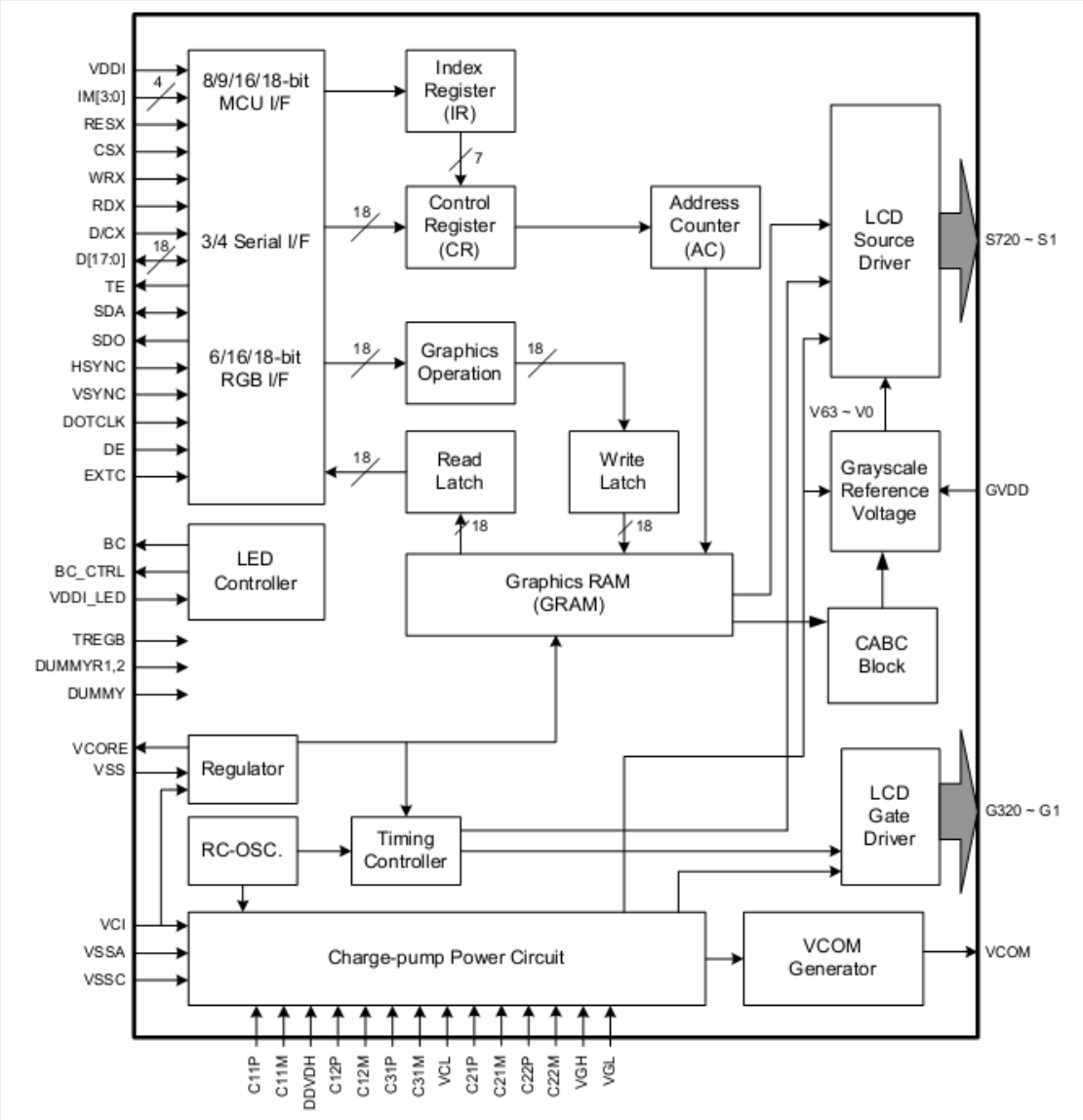


图 0-1 ILI9341 控制器内部框图

4.2.2 像素点的数据格式

图像数据的像素点由红(R)、绿(G)、蓝(B)三原色组成，三原色根据其深浅程度被分为 0~255 个级别，它们按不同比例的混合可以得出各种色彩。如 R：255，G255，B255 混合后为白色。根据描述像素点数据的长度，主要分为 8、16、24 及 32 位。如以 8 位来描述的像素点可表示 2^8=256 色，16 位描述的为

2^{16}=65536 色，称为真彩色，也称为 64K 色。实际上受人眼对颜色的识别能力的限制， 16 位色与 12 位色已经难以分辨了。

ILI9341 最高能够控制 18 位的 LCD，但为了数据传输简便，我们采用它的 16 位控制模式，以 16 位描述的像素点。按照标准格式，16 位的像素点的三原色描述的位数为 R：G：B =5：6：5，描述绿色的位数较多是因为人眼对绿色更为敏感。16 位的像素点格式见图 0-2。

16-bit data bus interface (D[17:13] & D[11:1] is used) , DPI[2:0] = 101, and RIM=0

	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D
16bpp Frame Memory Write	R[4]	R[3]	R[2]	R[1]	R[0]		G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	B[4]	B[3]	B[2]	B[1]	B

图 0-2 16 位像素点格式

图中的是默认 18 条数据线时，像素点三原色的分配状况，D1~D5 为蓝色，D6~D11 为绿色，D13~D17 为红色。这样分配有 D0 和 D12 位是无效的。若使用 16 根数据线传送像素点的数据，则 D0~D4 为蓝色，D5~D10 为绿色，D11~D15 为红色，使得刚好使用完整的 16 位。

RGB 比例为 5：6：5 是一个十分通用的颜色标准，在 GRAM 相应的地址中填入该颜色的编码，即可控制 LCD 输出该颜色的像素点。如黑色的编码为 0x0000，白色的编码为 0xffff，红色为 0xf800。

4.2.3 ILI9341 的通讯时序

目前，大多数的液晶控制器都使用 8080 或 6800 接口与 MCU 进行通讯，它们的时序十分相似，野火以 ILI9341 使用的 8080 通讯时序进行分析，实际上 ILI9341 也可以使用 SPI 接口来控制。

ILI9341 的 8080 接口有 5 条基本的控制信号线：

1. 用于片选的 CSX 信号线；
2. 用于写使能的 WRX 信号线；
3. 用于读使能的 RDX 信号线；
4. 用于区分数据和命令的 D/CX 信号线；
5. 用于复位的 RESX 信号线。

其中带 X 的表示低电平有效。除了控制信号，还有数据信号线，它的数目不定，可根据 ILI9341 框图中的 IM[3:0]来设定，这部分一般由制作液晶屏的厂家完成。为便于传输像素点数据，野火使用的液晶屏设定为 16 条数据线 D[15:0]。使用 8080 接口的写命令时序图见**错误！未找到引用源。**。

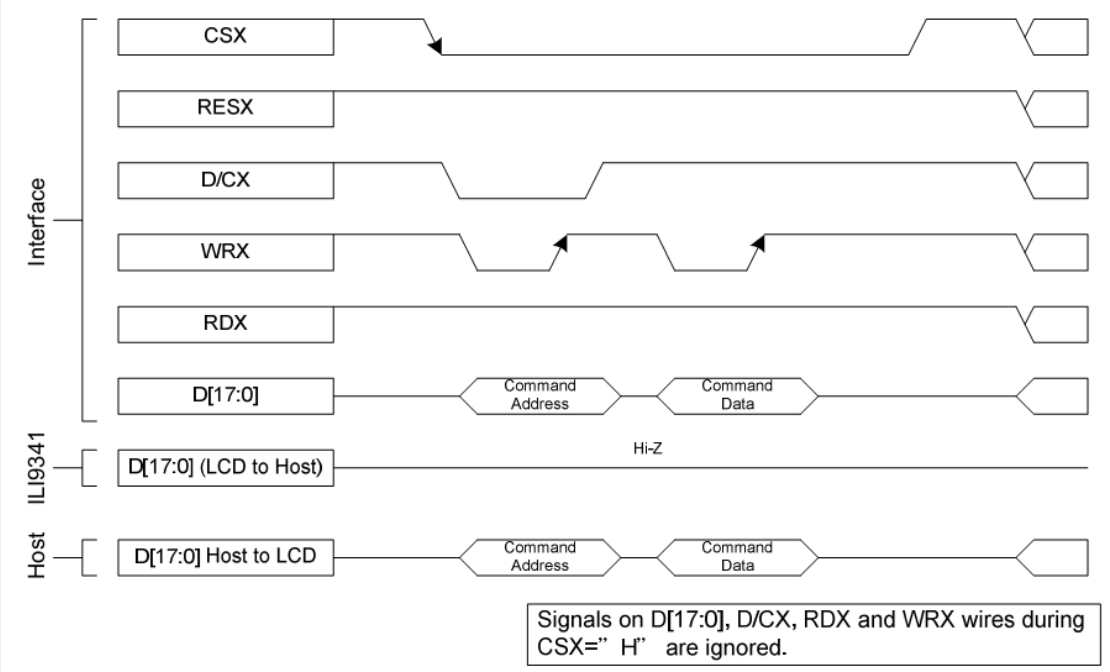


图 0-3 使用 18 条数据线的 8080 接口写命令时序

由图可知，写命令时序由 CSX 信号线拉低开始，*D/CX 信号线也置低电平*表示写入的是*命令地址*(可理解为命令编码，如软件复位命令：0x01)，以 WRX 信号线为低，RDX 信号为高表示数据传输方向为写入，同时，在数据线[17:0]输出命令地址，在第地二个传输阶段传送的为命令的参数，所以 *D/CX 要置高电平*，表示写入的是*命令数据*。

当我们需要向 GRAM 写入数据的时候，把 CSX 信号线拉低后，*把 D/CX 信号线置为高电平*，这时由 D[17:0]传输的数据则会被 ILI9341 保存至它的 GRAM 中。

4.3 用 STM32 驱动 LCD

ILI9341 的 8080 通讯接口时序可以由 STM32 使用普通 I/O 接口进行模拟，但这样效率较低，它提供了一种特别的控制方法——使用 FSMC 接口。

FSMC 简介

FSMC(flexible static memory controller)，译为静态存储控制器。可用于 STM32 芯片控制 NOR FLASH、PSRAM、和 NAND FLASH 存储芯片。其结构见图 0-4。

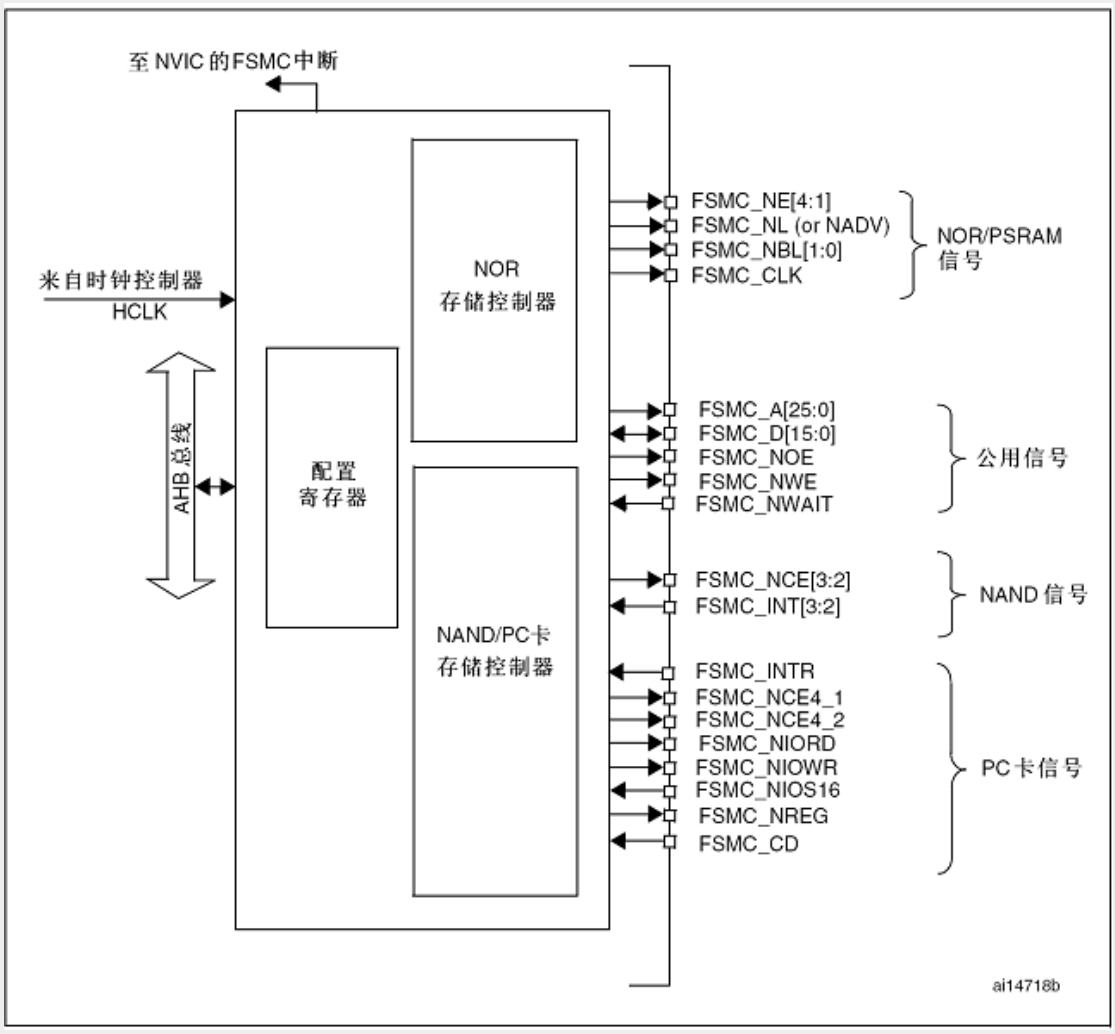


图 0-4 FSMC 结构图

我们是使用 FSMC 的 NOR\PSRAM 模式控制 LCD，所以我们重点分析框图中 NOR FLASH 控制信号线部分。控制 NOR FLASH 主要使用到如下信号线：

FSMC信号名称	信号方向	功能
CLK	输出	时钟(同步突发模式使用)
A[25:0]	输出	地址总线
D[15:0]	输入/输出	双向数据总线
NE[x]	输出	片选，x = 1...4
NOE	输出	输出使能
NWE	输出	写使能
NWAIT	输入	NOR闪存要求FSMC等待的信号

图 0-5 FSMC 控制 NOR FLASH 的信号线

根据 STM32 对寻址空间的地址映射，见前面的**错误！未找到引用源。**，*地址 0x6000 0000 ~0x9FFF FFFF 是映射到外部存储器的*，而其中的 0x6000 0000 ~0x6FFF FFFF 则是分配给 NOR FLASH、PSRAM 这类可直接寻址的器件。当 FSMC 外设被配置为正常工作，并且外部接了 NOR FLASH，这时若向 0x60000000 地址写入数据 0xffff，FSMC 会自动在各信号线上产生相应的电平信号，写入数据。该过程的时序图见图 0-6。

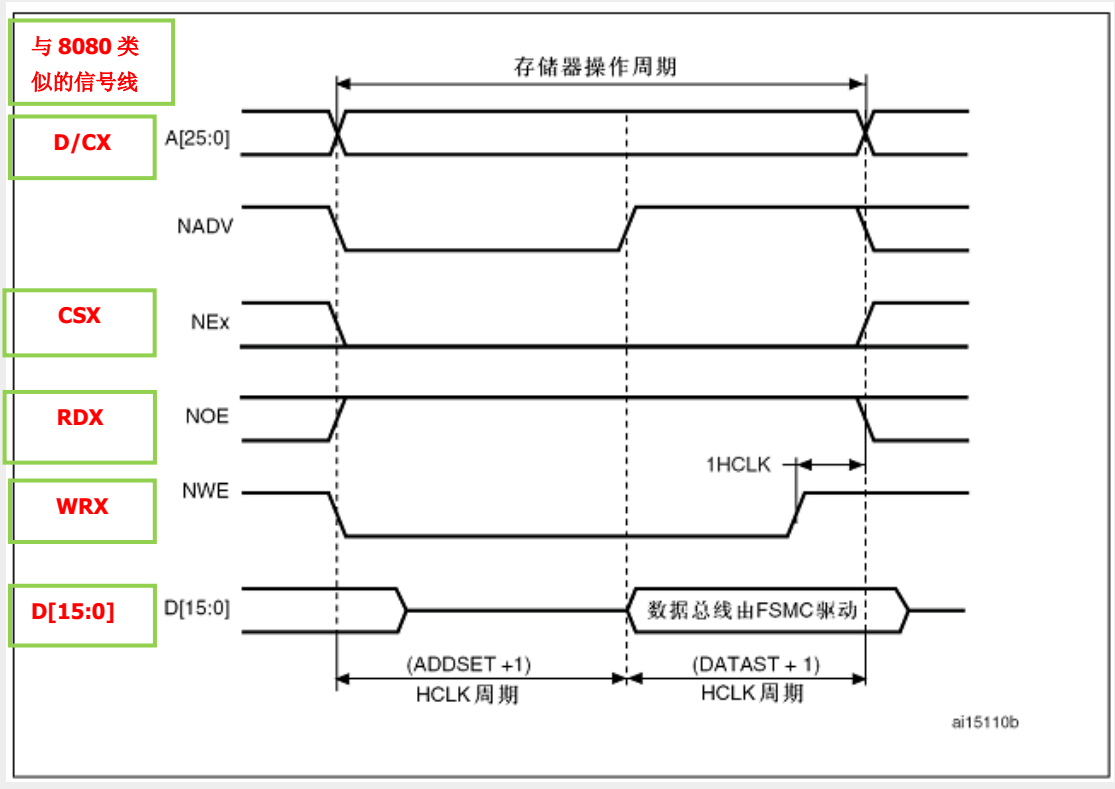


图 0-6 FSMC 写 NOR 时序图

它会控制*片选信号 NE[X]*选择相应的某块 NOR 芯片，然后使用*地址线 A[25:0]*输出 0x60000000，在 *NEW 写使能信号线*上发出写使能信号，而要写

入的数据信号 0xffff 则从*数据线 D[15:0]*输出，然后数据就被保存到 NOR FLASH 中了。

用 FSMC 模拟 8080 时序

在图 0-6 的时序图中 NADV 信号是在地址、信号线复用时作为锁存信号的，在此，我们略它。然后读者会发现，这个 FSMC 写 NOR 时序是跟 8080 接口的时序(见图 0-3)是十分相似的，对它们的信号线对比如下：

8080 信号线	功能	FSMC-NOR 信号线	功能
CSX	片选信号	NEx	片选
WRX	写使能	NWR	写使能
RDX	读使能	NOE	读使能
D[15:0]	数据信号	D[15:0]	数据信号
D\CX	数据/命令选择	A[25:0]	地址信号

前四种信号线都是完全一样的，仅在 8080 的数据\命令选择线与 FSMC 的地址信号线有区别。为了模拟出 8080 时序，我们把 FSMC 的 *A0 地址线*(也可以使用其它地址线)连接 8080 的 *D|CX*，即 A0 为*高电平*时，数据线 D[15:0]的信号会被理解 ILI9341 为*数值*，若 A0 为*低电平*时，传输的信号则会被理解为*命令*。

也就是说，当向地址为 0x6xxx xxx1、0x6xxx xxx3、0x6xxx xxx5…这些*奇数地址*写入数据时，地址线 A0(D/CX)会为*高电平*，这个数据被理解为*数值*；若向 0x6xxx xxx0 、0x6xxx xxx2、0x6xxx xxx4…这些*偶数地址*写入数据时，地址线 A0(D/CX)会为*低电平*，这个数据会被理解为*命令*。

有了这个基础，只要我们在代码中利用指针变量，向不同的地址单元写入数据，就能够由 FSMC 模拟出的 8080 接口向 ILI9341 写入控制命令或 GRAM 的数据了。

4.3.1 触摸屏感应原理

触摸屏常与液晶屏配套使用，组合成为一个可交互的输入输出系统。除了熟悉的电阻、电容屏外，触摸屏的种类还有超声波屏、红外屏。由于电阻屏的控制系统简单、成本低，且能适应各种上恶劣环境，被广泛采用。

电阻触摸屏的基本原理为分压，它由一层或两层阻性材料组成，在检测坐标时，在阻性材料的一端接参考电压 Vref，另一端接地，形成一个沿坐标方向的*均匀电场*。当触摸屏受到挤压时，阻性材料与下层电极接触，阻性材料被分为两部分，因而在*触摸点的电压，反映了触摸点与阻性材料的 Vref 端的距离，而且为线性关系*，而该触点的电压可由 ADC 测得。更改电场方向，以同样的方法，可测得另一方向的坐标。

4.3.2 TSC2046 触摸屏控制器

TSC2046 是专用在四线电阻屏的触摸屏控制器，MCU 可通过 SPI 接口向它写入控制字，由它测得 X、Y 方向的触点电压返回给 MCU。见图 0-7。

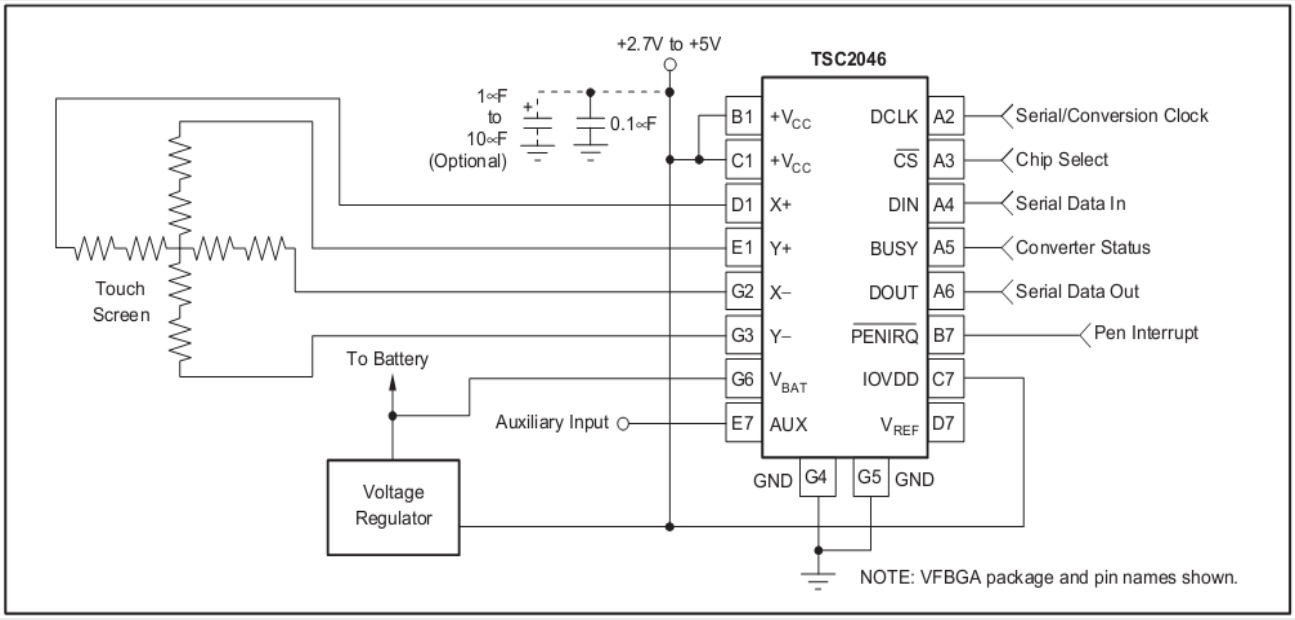


图 0-7 TSC2046 与电阻屏的连接图

图中，电阻屏两层阻性材料的两端分别接入到 TSC2046 的 X+、X-和 Y+、Y-。当要测量 X 坐标时，MCU 通过 SPI 接口写命令到 TSC2046，使它通过内部的*模拟开关*使 X+、X-接通电源，于是在电阻屏的 X 方向上产生一个匀强电场；把 Y+、Y-连接到 TSC2046 的 ADC。当电阻屏被触摸时，上、下两层的

阻性材料接触，在 *PENIRQ* 引脚产生一个 *中断信号*，通知 MCU。该触点的电压由 Y+或 Y-(此时的 Y+Y-电阻很小，可忽略)引入到 ADC 进行测量，MCU 读取该电压，进行软件转换，就可以测得触点 X 方向的坐标。同理可以测得 Y 方向的坐标。

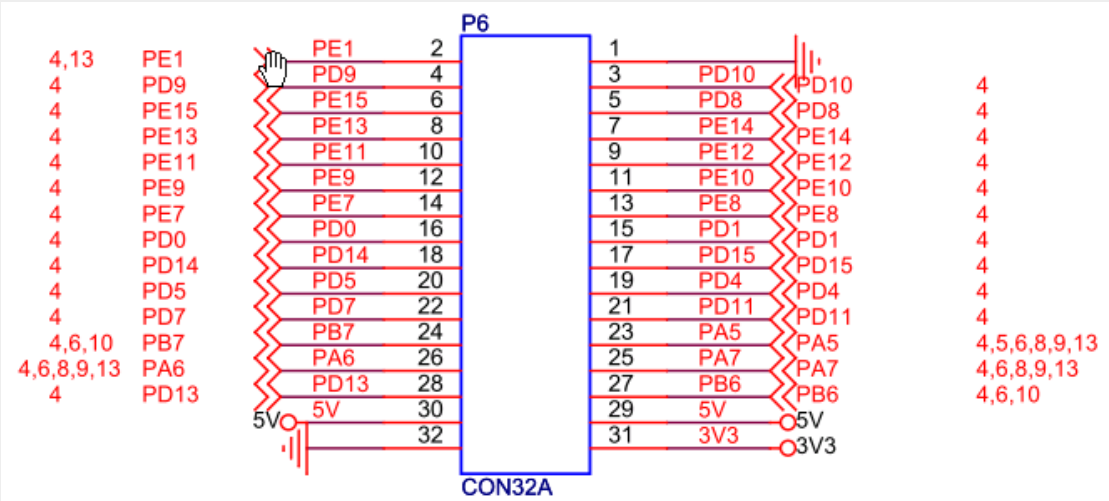
4.4 实验讲解

4.4.1 实验描述及工程文件清单

实验描述	野火 STM32 开发板驱动配套的 3.2 寸液晶、触摸屏，使用 FSMC 接口控制该屏幕自带的液晶控制器 ILI9341，使用 SPI 接口与触摸屏控制器 TSC2046 通讯。驱动成功后可在屏幕上使用基本的触摸绘图功能。
硬件连接	*TFT 数据线* PD14-FSMC-D0 ----LCD-DB0 PD15-FSMC-D1 ----LCD-DB1 PD0-FSMC-D2 ----LCD-DB2 PD1-FSMC-D3 ----LCD-DB3 PE7-FSMC-D4 ----LCD-DB4 PE8-FSMC-D5 ----LCD-DB5 PE9-FSMC-D6 ----LCD-DB6 PE10-FSMC-D7 ----LCD-DB7 PE11-FSMC-D8 ----LCD-DB8 PE12-FSMC-D9 ----LCD-DB9 PE13-FSMC-D10 ----LCD-DB10 PE14-FSMC-D11 ----LCD-DB11 PE15-FSMC-D12 ----LCD-DB12 PD8-FSMC-D13 ----LCD-DB13 PD9-FSMC-D14 ----LCD-DB14 PD10-FSMC-D15 ----LCD-DB15

	TFT 控制信号线 PD4-FSMC-NOE ----LCD-RD PD5-FSMC-NEW ----LCD-WR PD7-FSMC-NE1 ----LCD-CS PD11-FSMC-A16 ----LCD-DC PE1-FSMC-NBL1 ----LCD-RESET PD13-FSMC-A18 ----LCD-BLACK-LIGHT *触摸屏 TSC2046 控制线* PA5-SPI1-SCK ----TSC2046-SPI -SCK PA7-SPI1-MOSI ----TSC2046-SPI - MOSI PA6-SPI1-MISO ----TSC2046-SPI – MISO PB7-I2C1-SDA ----TSC2046-SPI-CS PB6-I2C1-SCL ----TSC2046- INT_IRQ
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c *FWlib/misc.c* *FWlib/stm32f10x_spi.c* *FWlib/stm32f10x_rcc.c* *FWlib/stm32f10x_exti.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_fsmc.c*
用户编写的文件	USER/main.c USER/stm32f10x_it.c *USER/lcd.c* *USER/SysTick.c* *USER/lcd_botton.c* *USER/Touch.c*

野火 STM32 开发板 3.2 寸 LCD 硬件连接图(兼容 V1 版本的 2.4 寸屏)



4.4.2 配置工程环境

本 LCD 触摸屏画板实验中我们用到了 *GPIO、RCC、SPI、EXTI、FSMC* 外设，所以我们先要把以下库文件添加到工程：*stm32f10x_gpio.c* 、*stm32f10x_rcc.c*、*stm32f10x_spi.c*、*stm32f10x_exti.c*、*stm32f10x_fsmc.c*。由于在 TSC2046 的触摸检测中使用了中断，所以还要把 *misc.c* 文件添加进工程。

本工程使用了旧的用户文件 *SysTick.c*，用作定时，把它添加到新工程之中，并新建 *lcd_botton.c*、*lcd.c*、*Touch.c* 及相应的头文件。其中 *lcd_botton.c* 文件定义了最底层的 LCD 控制函数，LCD 上层的函数如画点、显示字符等位于 *lcd.c* 文件中。

最后在 *stm32f10x_conf.h* 中把使用到的 ST 库的头文件注释去掉。

```
1. /**
2.   ******************************************************
3.   * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.   * @author  MCD Application Team
5.   * @version V3.5.0
6.   * @date    08-April-2011
7.   * @brief   Library configuration file.
8.   ******************************************************/
9.
10.#include "stm32f10x_exti.h"
11.#include "stm32f10x_fsmc.h"
12.#include "stm32f10x_gpio.h"
13.#include "stm32f10x_rcc.h"
14.#include "stm32f10x_spi.h"
```

```
15. #include "misc.h"
```

4.4.3 main 文件

从本工程的 main 文件分析代码的执行流程：

```
1.  /*
2.   * 函数名：main
3.   * 描述  ：主函数
4.   * 输入  ：无
5.   * 输出  ：无
6.   */
7.  int main(void)
8.  {
9.      SysTick_Init();                         /*systick 初始化*/
10.     LCD_Init();                             /*LCD 初始化*/
11.     Touch_init();                           /*触摸初始化*/
12.
13.     while(Touchl_Calibrate() !=0);          /*等待触摸屏校准完毕*/
14.     Init_Palette();                         /*画板初始化*/
15.
16.     while (1)
17.     {
18.         if(touch_flag == 1)             /*如果触笔按下了*/
19.         {
20.                         /*获取点的坐标*/
21.             if(Get_touch_point(&display, Read_2046_2(), &touch_para )
    !=DISABLE)
22.             {
23.                         /*画点*/
24.                 Palette_draw_point(display.x,display.y);
                                        /*画点*/
25.             }
26.         }
27.     }
28. }
```

其执行流程如下：

1. 调用 *SysTick_Init()*、*LCD_Init()*、*Touch_init()*初始化了 STM32 的 Systick、FSMC、SPI 外设，并用 FSMC 和 SPI 接口初始化了 ILI9341 和 TSC2046 控制器。
2. 调用 *Touch1_Calibrate()*函数进行触摸屏校准，使得触摸屏与液晶屏的坐标匹配。
3. 调用 *Init_Palette()*函数初始化触摸画板的应用程序，使得在 LCD 上显示画板界面，并能够正常响应触摸屏的信号。
4. 第 16~27 行的 *while* 循环，通过不断检测触笔按下标志 *touch_flag*，判断触摸屏是否被触笔按下。触摸屏控制器 *TSC2046* 检测到触笔信号

时，由它的 *PENIRQ* 引脚触发 STM32 的中断，在中断服务函数中对 *touch_flag* 标志置 1。

5. 第 21 行，若检测到触笔按下，调用 *Get_touch_point()*函数读取 *TSC2046* 的寄存器，获得与触点的 X、Y 坐标相关的电压信号，转化成 LCD 的 X、Y 坐标。
6. 获取了触点坐标后，使 LCD 液晶屏在该坐标点显示为对应的颜色。
7. 循环触点捕捉、画点过程，就实现了触摸画板的功能。

4.4.5 初始化 FSMC 模式

4.4.5.1 初始化液晶屏流程

在 main 函数中调用的 *LCD_Init()*函数，它对液晶控制器 ILI9341 用到的 GPIO、FSMC 接口进行了初始化，并且向该控制器写入了命令参数，配置好了 LCD 液晶屏的基本功能。其函数定义位于 lcd_botton.c 文件，如下：

```
1.  /*****************************************
2.   * 函数名：LCD_Init
3.   * 描述   ：LCD 控制 I/O 初始化
4.   *          LCD FSMC 初始化
5.   *          LCD 控制器 HX8347 初始化
6.   * 输入   ：无
7.   * 输出   ：无
8.   * 举例   ：无
9.   * 注意   ：无
10. *****************************************/
11. void LCD_Init(void)
12. {
13.     unsigned long i;
14.
15.     LCD_GPIO_Config();       //初始化使用到的 GPIO
16.     LCD_FSMC_Config();       //初始化 FSMC 模式
17.     LCD_Rst();               //复位 LCD 液晶屏
18.     Lcd_init_conf();         //写入命令参数，对液晶屏进行基本的初始化配置
19.     Lcd_data_start();        //发送写 GRAM 命令
20.     for(i=0; i<(320*240); i++)
21.     {
22.         LCD_WR_Data(GBLUE); //发送颜色数据，初始化屏幕为 GBLUE 颜色
23.
24.     }
25. }
```

*LCD_Init()*函数执行后，最直观的结果是使 LCD 整个屏幕显示编码为 0X07FF 的 GBLUE 颜色。

函数中调用的 *LCD_GPIO_Config()*主要工作是把液晶屏(不包括触摸屏)中使用到的 GPIO 引脚和使能外设时钟，除了*背光、复位*用的 *PD13* 和 *PD1* 设置为*通用推挽输出*外，其它的与 FSMC 接口相关的*地址信号、数据信号、控制信号*的端口均设置为*复用推挽输出*。

4.4.5.2 初始化 FSMC 模式

接下来 *LCD_Init()*函数调用 *LCD_FSMC_Config()*设置 FSMC 的模式，我们的目的是使用它的 NOR FLASH 模式模拟出 8080 接口，在 LCD 接口中我们使用的是 FSMC 地址线 *A16* 作为 8080 的 *D/CX* 命令选择信号的。

*LCD_FSMC_Config()*具体代码如下：

```
/*******************************************
 * 函数名：LCD_FSMC_Config
 * 描述   ：LCD  FSMC 模式配置
 * 输入   ：无
 * 输出   ：无
 * 举例   ：无
 * 注意   ：无
*******************************************/
void LCD_FSMC_Config(void)
{
    FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
    FSMC_NORSRAMTimingInitTypeDef  p;


    p.FSMC_AddressSetupTime = 0x02;  //地址建立时间
    p.FSMC_AddressHoldTime = 0x00;   //地址保持时间
    p.FSMC_DataSetupTime = 0x05;          //数据建立时间
    p.FSMC_BusTurnAroundDuration = 0x00;   //总线恢复时间
    p.FSMC_CLKDivision = 0x00;            //时钟分频
    p.FSMC_DataLatency = 0x00;            //数据保持时间
    p.FSMC_AccessMode = FSMC_AccessMode_B;    //在地址数\据线不复用的情况下，ABCD 模式的区别不大
                                              //本成员配置只有使用扩展模式才有效


    FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1;
   //NOR  FLASH 的 BANK1
    FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;       //数据线与地址线不复用
    FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_NOR;
                //存储器类型 NOR  FLASH
    FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;         //数据宽度为 16 位
    FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;     //使用异步写模式，禁止突发模式
    FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;   //本成员的配置只在突发模式下有效，等待信号极性为低
```

```
31.     FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
                  //禁止非对齐突发模式
32.     FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalA
    ctive_BeforeWaitState;    //本成员配置仅在突发模式下有效。NWAIT 信号在什么时期
    产生
33.     FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disabl
    e;                //本成员的配置只在突发模式下有效，禁用 NWAIT 信号
34.     FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disabl
    e;                //禁止突发写操作
35.     FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperatio
    n_Enable;          //写使能
36.     FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Di
    sable;            //禁止扩展模式，扩展模式可以使用独立的读、写模式
37.     FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p;
                   //配置读写时序
38.     FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p;
                   //配置写时序
39.
40.
41.     FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
42.
43.     /* 使能 FSMC Bank1_SRAM Bank */
44.     FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);
45. }
```

本函数主要使用了两种类型的结构体对 FSMC 进行配置，第一种为 *FSMC_NORSRAMInitTypeDef* 类型的结构体主要用于 NOR FLASH 的模式配置，包括存储器类型、数据宽度等。另一种的类型为 *FSMC_NORSRAMTimingInitTypeDef*，这是用于配置 FSMC 的 NOR FLASH 模式下读写时序中的地址建立时间、地址保持时间等，代码中用它定义了结构体 *p*，这个第二种类型的结构体在前一种结构体中被指针调用。

下面先来分析 *FSMC_NORSRAMInitTypeDef* 的结构体成员：

1. *FSMC_Bank*

用于*选择外接存储器的区域*(或地址)，见图 0-8 FSMC 存储块，STM32 的存储器映射中，把 0x6000 0000~0x9fff ffff 的地址都映射到被 FSMC 控制的外存储器中，其中属于 NOR FLASH 的为 0x6000 0000~0x6fff ffff。*而属于 NOR FLASH 的这部分地址空间又被分为 4 份，每份大小为 64MB，编号为 BANK1 ~BANK4*。分 BANK 是由 FSMC 寻址范围决定的，该接口的地址线最多为 26 条，即最大寻址空间为 2^{26} =64MB。为了扩展寻址空间，可把地址与数据线与多片 NOR FLASH 并联，由不同的*片选信号 NE[3:0]*区分不同的 BANK。

在本实验中，我们使用的是 FSMC 的信号线 *NE1* 作为控制 8080 的 CSX 片选信号，所以我们把本成配置为 *FSMC_Bank1_NORSRAM1* (NE1 片选 BANK1)。

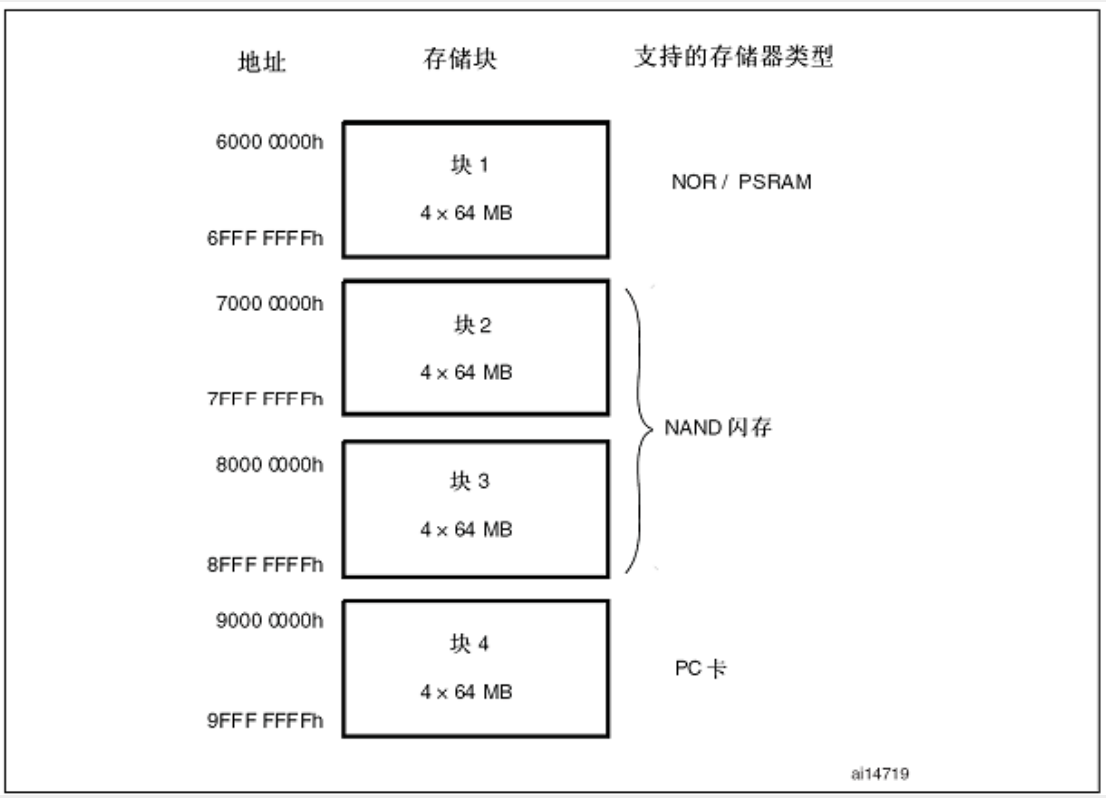


图 0-8 FSMC 存储块

2. *FSMC_DataAddressMux*

本成员用于配置 FSMC 的*数据线与地址线是否复用*。FSMC 支持数据与地址线复用或非复用两种模式。在*非复用模式*下 16 位数据线及 26 位地址线分开始用；*复用模式*则低 16 位数据/地址线复用。在复用模式下，推荐使用地址锁存器以区分数据与地址。当 *NADV 信号线*为低时，复用信号线 ADx(x=0...15)上出现地址信号 Ax，当 NADV 变高时，ADx 上出现数据信号 Dx。

本实验中用 FSMC 模拟 8080 接口，*地址线 A16* 提供 8080 的 *D/CX 信号*，实际上就只使用了这一条地址线，I\O 资源并不紧张，所以把本成员配置为 *FSMC_DataAddressMux_Disable* (非复用模式)。

3. *FSMC_MemoryType*

本成员用于配置 FSMC 外接的*存储器的类型*，可被配置为 *NOR FLASH 模式、PSARM 模式及 SRAM 模式*。

在本实验的应用中，由于 *NOR FLASH 模式的时序与 8080 更接近*，所以本结构体被配置为 *FSMC_MemoryType_NOR*(NOR FLASH 模式)。

4. *FSMC_MemoryDataWidth*

本成员用于设置 FSMC 接口的*数据宽度*，可被设置为 8Bit 或 16bit。对于 16 位宽度的外部存储器。在 STM32 地址映射到 FSMC 接口的结构中，HADDR 信号线是需要转换到外部存储器的内部 AHB 地址线，是*字节地址*。

若存储器的数据线宽为 8Bit，FSMC 的 *26* 条地址信号线 FSMC_A[25:0]直接可以引入到与 AHB 相连的 HADDR[25:0]，*26* 条*字节地址*信号线最大寻址空间为 64MB。见图 0-9。

数据宽度(1)	连到存储器的地址线	最大访问存储器空间(位)
8位	HADDR[25:0]与FSMC_A[25:0]对应相连	64M字节 x 8 = 512 M位
16位	HADDR[25:1]与FSMC_A[24:0]对应相连，HADDR[0]未接	64M字节/2 x 16 = 512 M位

图 0-9 外部存储器地址

若存储器的数据线宽 16Bit，则存储器的地址信号线是*半字地址(16Bit)*。为了使 HADDR 的*字节地址*信号线与存储器匹配，FSMC 的 *25* 条地址信号线 FSMC_A[24:0]与 HADDR[25:1]相连，由于变成了*半字地址(16Bit)*，仅需要 *25* 条*半字字地址*信号线就达到最大寻址空间 64MB。正因地址线的不对称相连，*16bit 数据线宽下，实际的访问地址为右移一位之后的地址*。

本实验中 8080 接口采用 16bit 模式，所以我们把本成员配置为 *FSMC_MemoryDataWidth_16b*，由于地址线不对称相连，这会影响到我们用*地址信号线 A16* 控制的 8080 接口的 *D/CX 信号*。

5. *FSMC_BurstAccessMode*

本成员用于配置*访问模式*。FSMC 对存储器的访问分为异步模式和突发模式(同步模式)。在*异步模式*下，每次传送数据都需要产生一个确定的地址，而*突发模式*可以在开始始提供一个地址之后，把数据成组地连续写入。

本实验中使用 FSMC 模拟 8080 端口，更适合使用异步模式，因而向本成员赋值为 *FSMC_WriteBurst_Disable*。

6. 突发模式参数配置

代码中的 30~34 行，都是关于使用突发模式时的一些参数配置，这些成员为： *FSMC_WaitSignalPolarity(配置等待信号极性)*、*FSMC_WrapMode(配置是否使用非对齐方式)*、*FSMC_WaitSignalActive(配置等待信号什么时期产生)*、*FSMC_WaitSignal（配置是否使用等待信号）*、*FSMC_WriteBurst(配置是否允许突发写操作)*，这些成员均需要在突发模式开启后配置才有效。

这些成员在开启突发模式时才有效，本实验使用的是异步模式，所以这些成员的参数没有意义。

7. *FSMC_WriteOperation*

本成员用于配置*写操作使能*，如果禁止了写操作，FSMC 不会产生写时序，但仍可从存储器中读出数据。

本实验需要写时序，所以向本成员赋值为 *FSMC_WriteOperation_Enable*(写使能)

8. *FSMC_ExtendedMode*

本成员用于配置是否使用*扩展模式*，在扩展模式下，读时序和写时序可以使用独立时序模式。如读时序使用模式 A，写时序使用模式 B，这些 A、B、C、D 模式实际上差别不大，主要是在使用*数据/地址线*复用的情况下，NADV 信号产生的时序不一样，具体的时序图可查阅《STM32 参考手册》。

本实验中数据/地址线不复用，所以读写时序中不同的 NADV 信号并没影响，禁止使用扩展模式 *SMC_ExtendedMode_Disable*。

9. *FSMC_ReadWriteTimingStruct 及 FSMC_WriteTimingStruct*

这两个参数分别用来设置 FSMC 的*读时序及写时序的时间参数*。若使用了扩展模式，则前者配置的是读时序，后者为写时序；若禁止了扩展模式，则读写时序都使用 *FSMC_ReadWriteTimingStruct* 结构体中的参数。

在配置这两个参数时，使用的是类型 *FSMC_NORSRAMTimingInitTypeDef* 时序初始化结构体，对这种类型结构体的成员进行赋值。它的成员分别有：*FSMC_AddressSetupTime(地址建立时间)*、*FSMC_AddressHoldTime(地址保持时间)*、*FSMC_DataSetupTime(数据建立时间)*、*FSMC_DataLatency(数据保持时间)* 、*FSMC_BusTurnAroundDuration(总线恢复时间)*、*FSMC_CLKDivision(时钟分频)*、*FSMC_AccessMode(访问模式)*。对以上各个时间成员赋的数值 X 表示 X 个时钟周期，它的时钟是由 HCLK 经过成员时钟分频得来的，该分频值在成员 *FSMC_CLKDivision(时钟分频)*中设置。其中 *FSMC_AccessMode(访问模式)*成员的设置只在开启了扩展模式才有效，而且开启了扩展模式后，读时序和写时序的设置可以是独立的。

本实验中的时序设置是根据 ILI9341 的 datasheet 设置的，调试的时候可以先把这些值设置得大一些，然后慢慢靠近 datasheet 要求的最小值，这样会取得比较好的效果。时序的参数设置对 LCD 的显示效果有一定的影响。

配置完初始化结构体后，要调用库函数 *FSMC_NORSRAMInit()*把这些配置参数写到控制寄存器，还要调用 *FSMC_NORSRAMCmd()*使能 *BANK1*。如果是使用 FSMC 配置其它存储器如 NAND FLASH，要使用其它的库函数及初始化结构体。

4.4.6 FSMC 模拟 8080 读写参数、命令

回到 *LCD_Init()*函数的执行流程，初始化完成 FSMC 接口后，就可以使用它控制 ILI9341 了。在 *LCD_Init()*中调用了 *Lcd_init_conf()*函数向 ILI9341 写入了一系列的控制参数：

```
1.  /*********************************
2.   * 函数名：Lcd_init_conf
3.   * 描述   ：ILI9341 LCD 寄存器初始配置
4.   * 输入   ：无
5.   * 输出   ：无
6.   * 举例   ：无
7.   * 注意   ：无
8.  *********************************/
9.  void Lcd_init_conf(void)
10. {
11.     DEBUG_DELAY();
12.     LCD_ILI9341_CMD(0xCF);
13.     LCD_ILI9341_Parameter(0x00);
```

```
    LCD_ILI9341_Parameter(0x81);
    LCD_ILI9341_Parameter(0x30);

    DEBUG_DELAY();
    LCD_ILI9341_CMD(0xED);
    LCD_ILI9341_Parameter(0x64);
    LCD_ILI9341_Parameter(0x03);
    LCD_ILI9341_Parameter(0x12);
    LCD_ILI9341_Parameter(0x81);

    DEBUG_DELAY();
    LCD_ILI9341_CMD(0xE8);
    LCD_ILI9341_Parameter(0x85);
    LCD_ILI9341_Parameter(0x10);
    LCD_ILI9341_Parameter(0x78);
  // ·····此处省略几十行·····
```

本函数十分长，由于篇幅问题，以上只是该函数其中的一部分，省略部分的代码也是这样的模板，只是写入的命令和参数不一样而已，这些命令和参数设置了像素点颜色格式、屏幕扫描方式、横屏\竖屏等初始化配置，这些命令的意义从 ILI9341 的 datasheet 命令列表中可以查到。该函数通过调用 *LCD_ILI9341_CMD()*写入*命令*，用 *LCD_ILI9341_Parameter()*写入*参数*。它们实质是两个宏：

```
/*****************************************************/
#define LCD_ILI9341_CMD(index)          LCD_WR_REG(index)
#define LCD_ILI9341_Parameter(val)      LCD_WR_Data(val)

/****为了移植方便，上面的宏只是封装，以下才是最底层的宏**********/
/* 选择 BANK1-BORSRAM1 连接 TFT，地址范围为 0X60000000~0X63FFFFFF
 * FSMC_A16 接 LCD 的 DC(寄存器/数据选择)脚
 * 16 bit => FSMC[24:0]对应 HADDR[25:1]
 * 寄存器基地址 = 0X60000000
 * RAM 基地址 = 0X60020000 = 0X60000000+2^16*2 = 0X60000000 + 0X20000 = 0X60020000
 * 当选择不同的地址线时，地址要重新计算。
 */
#define Bank1_LCD_D     ((u32)0x60020000)       //Disp Data ADDR
#define Bank1_LCD_C     ((u32)0x60000000)       //Disp Reg ADDR

/*选定 LCD 指定寄存器（命令编码）*/
#define LCD_WR_REG(index)     ((*(__IO u16 *) (Bank1_LCD_C)) = ((u16)index))

/*往 LCD 写入数据*/
#define LCD_WR_Data(val)        ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))
```

这部分是 FSMC 模拟 8080 接口的精髓。

读写参数、命令

先来看第 21 行的宏，*LCD_WR_Data(val)*，这是一个带参宏，用于向 LCD 控制器写入参数，参数为 val。它的宏展开为：

```
1.  ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))
```

宏展开中的*(Bank1_LCD_D)*是一个在第 14 行定义的宏，它的值为 0x6002 0000，实质是一个地址，这个地址的计算在后面介绍。

*(__IO u16 *) (Bank1_LCD_D)*表示把*(Bank1_LCD_D)*强制转换成一个 16 位的地址。*((*(__IO u16 *) (Bank1_LCD_D))*表示再对这个地址作”*”指针运算，取该指针对象的内容，并把它的内容赋值为 *= ((u16)(val)))*。所以整个宏的操作就是：把参数 val 写入到地址为 0x6002 0000 的地址空间。

由于这个地址被 STM32 映射到外存储器，所以会由 FSMC 外设以访问 NOR FLASH 的形式、时序，在地址线上发出 0x6002 0000 地址信号，在数据线上发出 val 数据信号，写入参数到外存储器中。而 FSMC 接口又被我们模拟成了 8080 接口，最终 val 被 8080 接口理解为参数，传输到 ILI9341 控制器中。

计算地址

见图 0-10。计算地址前，再明确一下在本实验中，使用的是 *FSMC_NE1* 作为 *8080_CS 片选信号*，以 *FSMC_A16* 作为 *8080_D/CX 数据/命令信号*(图中为 RS，意义相同)。

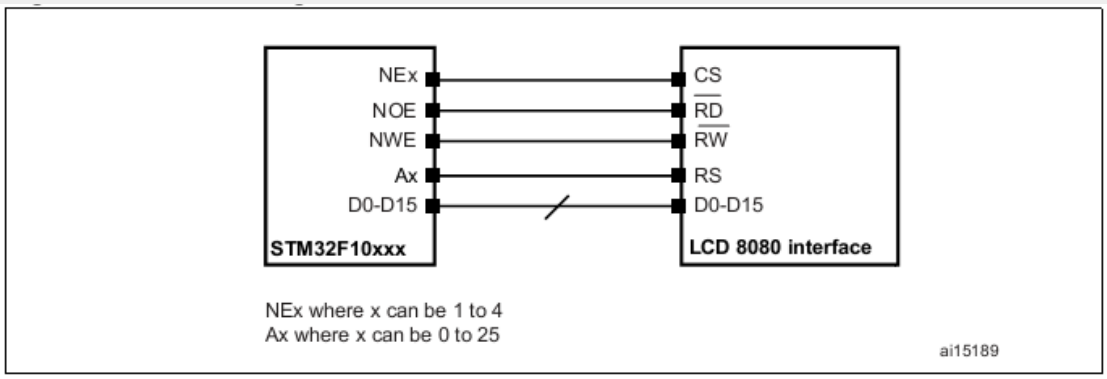


图 0-10 FSMC 与 8080 端口连接简图

按这种连接时，FSMC_NE1 为*低电平*、FSMC_A16 为*高电平*，表示通过 D[15:0]发送\接收的数据被 8080 接口解释为*参数(数值)*，当我们访问 *0x6002 0000* 这个地址的时候，正好符合这个条件。该地址的计算过程如下：

由于选择的是使用 FSMC_NE1 片选信号线，片选的为 BANK1，所以基地址为 0x6000 0000。要把地址线 FSMC_A16 置为*高电平*，可以采用下列算式：

```
1.  0x6000 0000 |= 1<<16;  //结果 = 0x6001 0000
```

但是，这样计算出来的地址只是数据线为 8Bit 模式下的字节地址。由于我们采用的是 16Bit 数据线，FSMC[24:0]与 HADDR[25:1]对齐，HADDR 地址要左移一位才是 FSMC 的访问地址。因此为了把 FSMC 中的 *FSMC_A16 (D/CX 线)*置 1，实际上要对应到 HADDR 地址(AHB 地址)的 *HADDR_A17*，即正确的计算地址公式应为：

```
1.  0x6000 0000 |= 1<< (16+1) ;  //此为正确结果 = 0x6002 0000
```

对于 16 位数据线模式，能使 FSMC_NEX 为低电平，FSMC_A16 为高电平的地址，并不只有 *0x6002 0000* 一个，只要是属于 *0X60000000~0X63FFFFFF* 范围内(BANK1 地址范围)，HADDR_A17 位为高电平的地址均可，这是因为我们只采用了 FSMC_A16 用于 8080 的 D/CX 信号，所以地址线的电平状态并无影响。如 0x6002 0001 地址，在本实验中是与 0x6002 0000 等价的。

若修改这个地址，使 FSMC_NEX 为低电平，FSMC_A16 为*低电平*，即 D/CX 被置 0，*8080 会把由数据线传输的信号理解为命令*。以同样的方式计算，符合这样要求的其中一个地址为 *0x6000 0000*，向这个地址空间赋值，这个值最终会被 8080 接口解释为*命令*。宏 *LCD_WR_REG(index)* 就是这样实现的。

给整个屏幕上色

再次回到 *LCD_Init()*函数，它调用完 *Lcd_init_conf()*初始化了液晶屏后，向使用 *Lcd_data_start()*函数发送了一个命令——写 GRAM 内容，即后面发送的数据都被解析为显示到屏幕像素点的数据。代码中使用 for 循环把语句

*LCD_WR_Data(GBLUE)*执行了 320*240 次，即把所有像素点都显示为 GBLUE 颜色。

4.4.6.1 液晶屏画点函数

初始化了液晶屏后，就可以控制液晶上每个像素点的颜色了。如果能够实现一个*画点函数*，在指定的(x,y)坐标像素点上显示指定的颜色，那么就能够实现一切液晶屏最复杂的显示功能，如在液晶屏指定位置显示形状、文字、图像，都可以通过调用画点函数或以类似的方式控制液晶的像素点。本实验中的画点函数 *LCD_Point()*在 *lcd.c* 文件中定义如下：

```
/*****************************************************
 * 函数名：LCD_Point
 * 描述   ：在指定坐标处显示一个点
 * 输入   ：-x 横向显示位置 0~319
 *          -y 纵向显示位置 0~239
 * 输出   ：无
 * 举例   ：    LCD_Point(100,200);
 *            LCD_Point(10,200);
 *            LCD_Point(300,220);
 * 注意   ：    (0,0)位置为液晶屏左上角 已测试
*****************************************************/
void LCD_Point(u16 x,u16 y)
{
    LCD_open_windows(x,y,1,1);
    LCD_WR_Data(POINT_COLOR);
}
```

本函数首先调用了一个液晶显示窗口函数 *LCD_open_windows()*用于开辟液晶屏上的显示区域，后面写入的颜色数据将被显示到该区域中。示窗函数按*(x,y,1,1)*的参数调用时，该函数开辟了一个坐标为(x,y)的像素点。接着调用 *LCD_WR_Data()*向 ILI9341 写入颜色数据，像素的坐标即为示窗函数开辟的显示坐标。于是 *LCD_Point()*函数就实现了控制特定坐标的像素点。

接下来分析 *LCD_open_windows()*函数是如何开辟显示区域的：

```
/*********************************
 * 函数名：LCD_open_windows
 * 描述   ：开窗(以 x,y 为坐标起点，长为 len,高为 wid)
 * 输入   ：-x     窗户起点
 *          -y       窗户起点
 *          -len   窗户长
 *          -wid 窗户宽
 * 输出   ：无
 * 举例   ：无
```

```
 * 注意  : 无
**********************************/
void LCD_open_windows(u16 x,u16 y,u16 len,u16 wid)
{

    if(display_direction == 0)       /*如果是横屏选项*/
    {

        LCD_ILI9341_CMD(0X2A);
        LCD_ILI9341_Parameter(x>>8);     //start 起始位置的高 8 位
        LCD_ILI9341_Parameter(x-((x>>8)<<8));  //起始位置的低 8 位
        LCD_ILI9341_Parameter((x+len-1)>>8);     //end 结束位置的高 8 位
        LCD_ILI9341_Parameter((x+len-1)-(((x+len-1)>>8)<<8));  //结束位置的低 8 位

        LCD_ILI9341_CMD(0X2B);
        LCD_ILI9341_Parameter(y>>8);    //start
        LCD_ILI9341_Parameter(y-((y>>8)<<8));
        LCD_ILI9341_Parameter((y+wid-1)>>8);    //end
        LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-1)>>8)<<8));

    }
    else
    {
        LCD_ILI9341_CMD(0X2B);
        LCD_ILI9341_Parameter(x>>8);
        LCD_ILI9341_Parameter(x-((x>>8)<<8));
        LCD_ILI9341_Parameter((x+len-1)>>8);
        LCD_ILI9341_Parameter((x+len-1)-(((x+len-1)>>8)<<8));

        LCD_ILI9341_CMD(0X2A);
        LCD_ILI9341_Parameter(y>>8);
        LCD_ILI9341_Parameter(y-((y>>8)<<8));
        LCD_ILI9341_Parameter((y+wid-1)>>8);
        LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-1)>>8)<<8));

    }

      LCD_ILI9341_CMD(0x2c);
}
```

本函数主要使用了三个 ILI9341 的控制命令：

1. *0x2A 列地址控制命令*

   用于设置显示区域的*列像素区域*。它有四个参数，分别为 SC 的高 8 位、SC 的低 8 位及 EC 的高 8 位、EC 的低 8 位。SC 和 EC 的意义见图 0-11。*SC*表示要控制的显示区域的*列起始坐标*，*EC*表示显示区域的*列结束坐标*。

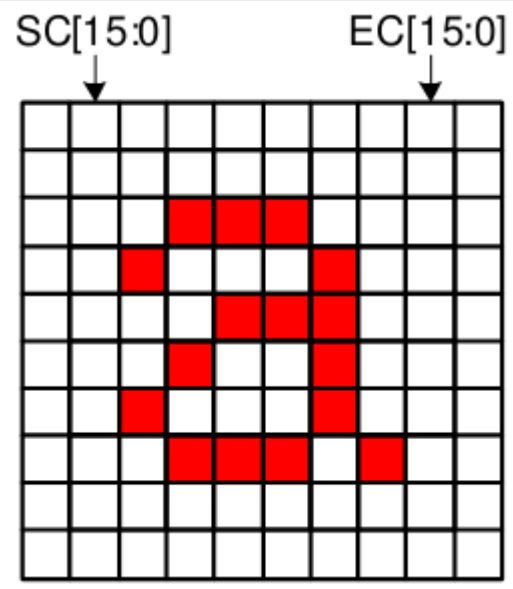
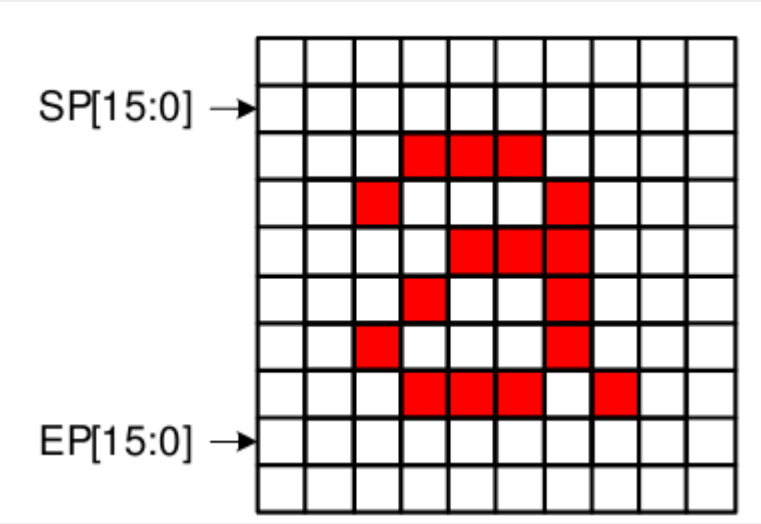


图 0-11 列控制命令 SC 和 EC

2. *0x2B 页地址控制命令*

用于设置显示区域的 *页(行)像素区域*。与上一命令类似，也有四个参数，分别为 SP 的高 8 位、SP 的低 8 位及 EP 的高 8 位、EP 的低 8 位。SP 和 EP 的意义见**错误！未找到引用源。**。*SP*表示要控制的显示区域的*行起始坐标*，*EP*表示显示区域的*行结束坐标*。

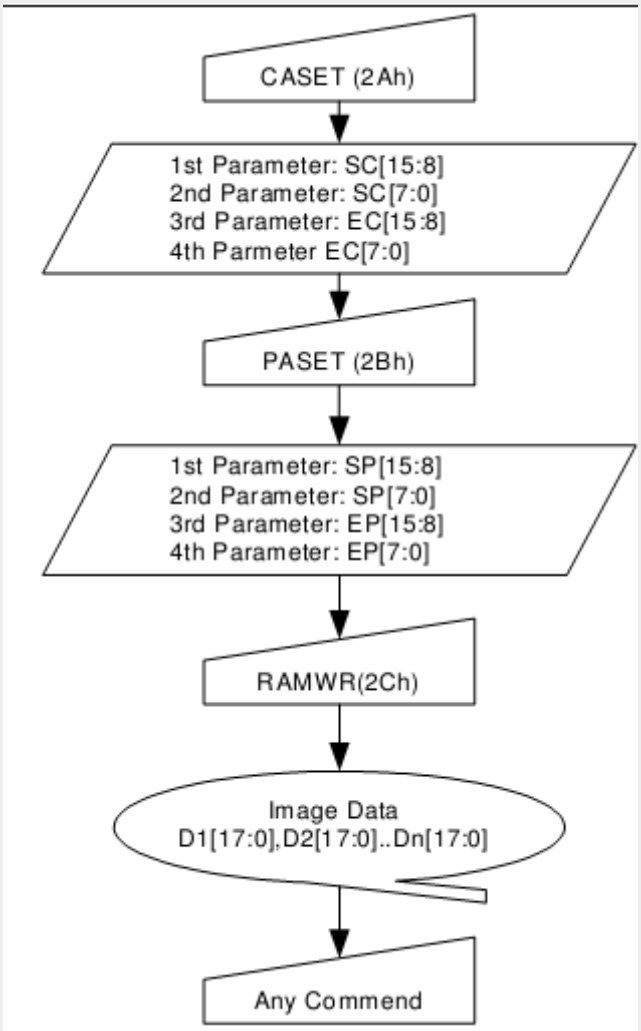


3. *0x2C 写 RAM 命令*

本命令用于表示*开始写入像素显示数据*，紧跟着本命令后面的即为写入到 GRAM 的 RGB 5：6：5 的颜色数据。在初始化液晶屏函数中也调用到本命令。使用本命令，后面的颜色数据按照一个接着一个预设的扫描方式(扫描方式决定了是横屏和竖屏)写入到由 0x2A 和 0x2B 设置的显

示区域中。横屏扫描方式为从左到右，扫描完一行(页)像素点再扫描下一行(页)。竖屏扫描为从上到下，扫描完一列再扫描下一列。

了解这三个命令后，就知道示窗函数的执行流程了。



调用本示窗函数后，就可以开辟一个起始坐标为(x,y)，长为 len 宽为 wid 的矩形显示窗口了。特别地，当 len=wid=1 时， 开辟的为一个坐标为(x,y)的像素点。

4.4.6.2 触摸屏校正

在 main 函数初始化完成 LCD 之后，调用了 *Touchl_Calibrate()*函数进行触摸屏校正。

```
1. /*******************************************************
2. * 函数名：Touchl_Calibrate
3. * 描述   ：触摸屏校正函数
```

```
* 输入  : 无
* 输出  : 0     --- 校正成功
                  1    --- 校正失败
* 举例  : 无
* 注意  : 无
*******************************************************/
int Touchl_Calibrate(void)
{
    uint8_t i;
    u16 test_x=0, test_y=0;
    u16 gap_x=0, gap_y=0;
    Coordinate * Ptr;
   // delay_init();
    Set_direction(0);    //设置为横屏
    for(i=0;i<4;i++)
    {
        LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);   //使整个屏幕显示背景颜色
        LCD_Str_6x12_O(10, 10,"Touch Calibrate", 0);   //显示提示信息
        LCD_Num_6x12_O(10,25, i+1,  0);                //显示触点次数

        delay_ms(500);
        DrawCross(DisplaySample[i].x,DisplaySample[i].y);  //显示校正用的"十"字
        do
        {
            Ptr=Read_2046();  //读取 TSC2046 数据到变量 ptr
        }
        while( Ptr == (void*)0 );  //当 ptr 为 0 时表示没有触点被按下
        ScreenSample[i].x= Ptr->x;     //把读取的原始数据存放到 ScreenSample 结构体
        ScreenSample[i].y= Ptr->y;

    }

        /* 用原始参数计算出 原始参数与坐标的转换系数。 */
    Cal_touch_para( &DisplaySample[0],&ScreenSample[0],&touch_para ) ;

        /*计算 X 值*/
    test_x = ( (touch_para.An * ScreenSample[3].x) +
               (touch_para.Bn * ScreenSample[3].y) +
               touch_para.Cn
               ) / touch_para.Divider ;

        /*计算 Y 值*/
    test_y = ( (touch_para.Dn * ScreenSample[3].x) +
               (touch_para.En * ScreenSample[3].y) +
               touch_para.Fn
               ) / touch_para.Divider ;

    gap_x = (test_x > DisplaySample[3].x)?(test_x - DisplaySample[3].x):(DisplaySample[3].x - test_x);
    gap_x = (test_y > DisplaySample[3].y)?(test_y - DisplaySample[3].y):(DisplaySample[3].y - test_y);


     LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);
    if((gap_x>11)||(gap_y>11))
    {

        LCD_Str_6x12_O(100, 100,"Calibrate fail", 0);
        LCD_Str_6x12_O(100, 120,"  try again   ", 0);
        delay_ms(2000);
        return 1;
```

```
63.     }
64.
65.
66.     aa1 = (touch_para.An*1.0)/touch_para.Divider;
67.     bb1 = (touch_para.Bn*1.0)/touch_para.Divider;
68.     cc1 = (touch_para.Cn*1.0)/touch_para.Divider;
69.
70.     aa2 = (touch_para.Dn*1.0)/touch_para.Divider;
71.     bb2 = (touch_para.En*1.0)/touch_para.Divider;
72.     cc2 = (touch_para.Fn*1.0)/touch_para.Divider;
73.
74.     LCD_Str_6x12_O(100, 100,"Calibrate Success", 0);
75.     delay_ms(1000);
76.
77.     return 0;
78. }
```

本函数的主要作用是在指定的几个*液晶屏坐标*(逻辑坐标)显示"十"字交叉点，由用户使用触笔点击触摸屏交叉点，读取由 TSC2046 测得的*触点电压*(物理坐标)。采集 4 个不同位置的触点电压(物理坐标)，然后根据触摸校准算法*把逻辑坐标与物理坐标转换公式的系数 A、B、C、D、E、F 计算出来*。

若使用此函数校准成功后，用户再点击触摸屏时，*可把测量出的触点电压(物理坐标)代入已知系数的转换公式，计算出对应的液晶屏坐标(逻辑坐标)*。

转换公式的系数为以上代码 66~72 行中的 *aa1、bb1、cc1、aa2、bb2、cc3* 这几个全局变量，如果把这几个数据保存在非易失性存储器（SD 卡、EEPROM 等）中，上电后向这几个变量赋值，就不需要每次上电都进行一次触屏校准了。

本函数中大部分都是关于触摸屏校准算法的数学运算，有兴趣的读者可查阅其它相关资料来理解。在代码中的 18~30 行，与触摸屏的触点电压获取有关，分析如下：

1. 第 20~25 行，调用 *LCD_Rectangle()、LCD_Str_6x12_O()、LCD_Num_6x12_O()、DrawCross()*由液晶屏显示背景、提示信息及校准用的“十”字。这些函数都与液晶的画点函数原理类似，关于字符显示的在下一个章节进行说明。
2. 第 28 行，调用 *Read_2046()*函数获取触点的电压，该函数通过向 TSC2046 控制器发送控制命令：若触笔点击触摸屏时采集触点的电压，采集 10 个电压取平均值，结果返回给变量 Ptr；若没有触点，则

Ptr 的值为 0，由 do-while 循环等待至采集到数据为止。Ptr 中保存的电压数据在后面被用于校准算法计算。

*Read_2046()*函数定义如下：

```
/*******************************************************
* 函数名：Read_2046
* 描述   ：得到滤波之后的 X Y
* 输入   ：无
* 输出   ：Coordinate 结构体地址
* 举例   ：无
* 注意   ：速度相对比较慢
*******************************************************/
Coordinate *Read_2046(void)
{
    static Coordinate  screen;
    int m0,m1,m2,TP_X[1],TP_Y[1],temp[3];
    uint8_t count=0;

        /* 坐标 X 和 Y 进行 9 次采样*/
    int buffer[2][9]={{0},{0}};
    do
    {
        Touch_GetAdXY(TP_X,TP_Y);
        buffer[0][count]=TP_X[0];
        buffer[1][count]=TP_Y[0];
        count++;

    }   /*用户点击触摸屏时即 TP_INT_IN 信号为低 并且 count<9*/
    while(!INT_IN_2046&& count<9);

//由于篇幅问题，此处省略很多行，省略部分主要为计算 10 个采样电压的平均值

                ...................................
```

在 *Read_2046()*函数中，调用了 *Touch_GetAdXY()*，它用于获取一次触点(x,y)电压。实际上，驱动 TSC2046 最底层的是命令 *WR_CMD(CHX)*和 *WR_CMD(CHY)*，发送了这两个命令后，TSC2046 开始采集相应的触点电压，通过 SPI 传送触点电压数据到 STM32。

命令语句中的 CHX 宏展开为 0xd0，CHY 为 0x90，它们是根据 TSC2046 的命令格式设定的。驱动 TSC2046 的命令控制字格式。

位 7 (MSB)	位 6	位 5	位 4	位 3	位 2	位 1	位 0 (LSB)
S	A2	A1	A0	MODE	SER/$\overline{\text{DFR}}$	PD1	PD0

其中 S 为数据传输起始标志位该位必为 1，A2~A0 进行通道选择 MOD 用于转换 精度选择，1 为 8 位精度，0 为 12 位精度。

A2	A1	A0	+REF	-REF	Y-	X+	Y+	Y-位置	X-位置	Z1-位置	Z2-位置	驱 动
0	0	1	Y+	Y-		+IN		M				Y+,Y-
0	1	1	Y+	X-		+IN				M		Y+,X-
1	0	0	Y+	Y+	+IN						M	Y+,X-
1	0	1	X+	X-			+IN		M			X+,X-

所谓通道选择即为检测哪一个通道的坐标。如 A2~A0 为 001 时，即命令控制字为 0x90，根据表格知，芯片会给触摸屏的 Y 阻性材料层的两端提供 Y+、Y-的电压，若有触笔点击，则 Y 触点电压可经过 X+利用 ADC 读取得。同理命令控制字为 0xd0 时，A2~A0 为 101，即给 X 阻性材料层提供电压，触点电压经过 Y+由 ADC 读取得。这就是 TSC2046 采集触点电压的原理。

4.4.6.3 检测触点、画点

回到 main 函数。触摸屏也校准好后，剩下的就是应用程序代码了，调用 *Init_Palette()*使液晶屏显示出画板的界面， (由于印刷原因，无法分辨具体颜色)

该画板的界面左侧为各种颜色方块，右侧为提供给用户进行绘画的空间。显示完该界面后，循环检测 touch_flag 标志，它在中 *stm32f10x_it.c* 文件中的中断服务函数被赋值。当触摸屏被按下时会进入该函数，对 touch_flag 赋值，如下：

```
1.  void EXTI9_5_IRQHandler(void)
2.  {
3.
4.
5.    if(EXTI_GetITStatus(EXTI_Line6) != RESET)
6.    {
```

```
7.       GPIO_ResetBits(GPIOB, GPIO_Pin_5);

8.
9.              touch_flag=1;
10.
11.     EXTI_ClearITPendingBit(EXTI_Line6);
12.   }
13. }
```

若 touch_flag 标志为 1，即触摸屏被按下，main 中调用函数 *Get_touch_point()*，该函数通过 *Read_2046_2()*获取触点电压，根据公式把电压转换为液晶坐标，并保存到 *display* 结构体中。得到液晶坐标后，main 调用 *Palette_draw_point()*对相应的坐标点进行处理，若触点位于画板界面的颜色方块中，则使画笔变为该颜色，其后在空白界面的触点将显示该颜色的笔迹。

4.5 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，接上液晶屏，将编译好的程序下载到开发板。运行后，可在 LCD 屏幕看到提示信息“Touch Calibrate” 和触摸屏校正用的“十”字。点击“十”字的中间（共四个），若校正成功后会出现画板界面，在画板界面的右侧可进行绘画，点击左侧的颜色块可选择笔迹的颜色。