第十八章 TFTLCD 显示实验

上一章我们介绍了 OLED 模块及其显示，但是该模块只能显示单色/双色，不能显示彩色，而且尺寸也较小。本章我们将介绍 ALIENTEK 2.8 寸 TFT LCD 模块，该模块采用 TFTLCD 面板，可以显示 16 位色的真彩图片。在本章中，我们将使用探索者 STM32F4 开发板上的 LCD 接口，来点亮 TFTLCD，并实现 ASCII 字符和彩色的显示等功能，并在串口打印 LCD 控制器 ID，同时在 LCD 上面显示。本章分为如下几个部分：

18.1 TFTLCD & FSMC 简介

18.2 硬件设计

18.3 软件设计

18.4 下载验证

18.1 TFTLCD&FSMC 简介

本章我们将通过 STM32F4 的 FSMC 接口来控制 TFTLCD 的显示，所以本节分为两个部分，分别介绍 TFTLCD 和 FSMC。

18.1.1 TFTLCD 简介

TFT-LCD 即薄膜晶体管液晶显示器。其英文全称为：Thin Film Transistor-Liquid Crystal Display。TFT-LCD 与无源 TN-LCD、STN-LCD 的简单矩阵不同，它在液晶显示屏的每一个象素上都设置有一个薄膜晶体管（TFT），可有效地克服非选通时的串扰，使显示液晶屏的静态特性与扫描线数无关，因此大大提高了图像质量。TFT-LCD 也被叫做真彩液晶显示器。

上一章介绍了 OLED 模块，本章，我们给大家介绍 ALIENTEK TFTLCD 模块，该模块有如下特点：

1，2.4’/2.8’/3.5’/4.3’/7’ 5 种大小的屏幕可选。

2，320×240 的分辨率（3.5’分辨率为:320*480，4.3’和 7’分辨率为：800*480）。

3，16 位真彩显示。

4，自带触摸屏，可以用来作为控制输入。

本章，我们以 2.8 寸（其他 3.5 寸/4.3 寸等 LCD 方法类似，请参考 2.8 的即可）的 ALIENTEK TFTLCD 模块为例介绍，该模块支持 65K 色显示，显示分辨率为 320×240，接口为 16 位的 80 并口，自带触摸屏。

该模块的外观图如图 18.1.1.1 所示：

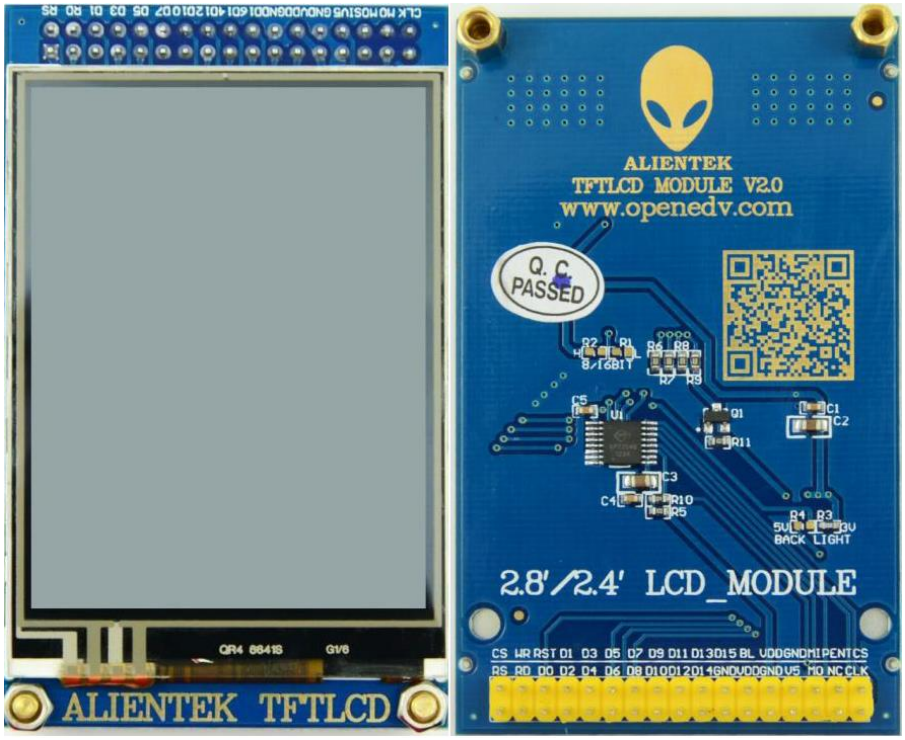

图 18.1.1.1 ALIENTEK 2.8 寸 TFTLCD 外观图

模块原理图如图 18.1.1.2 所示：

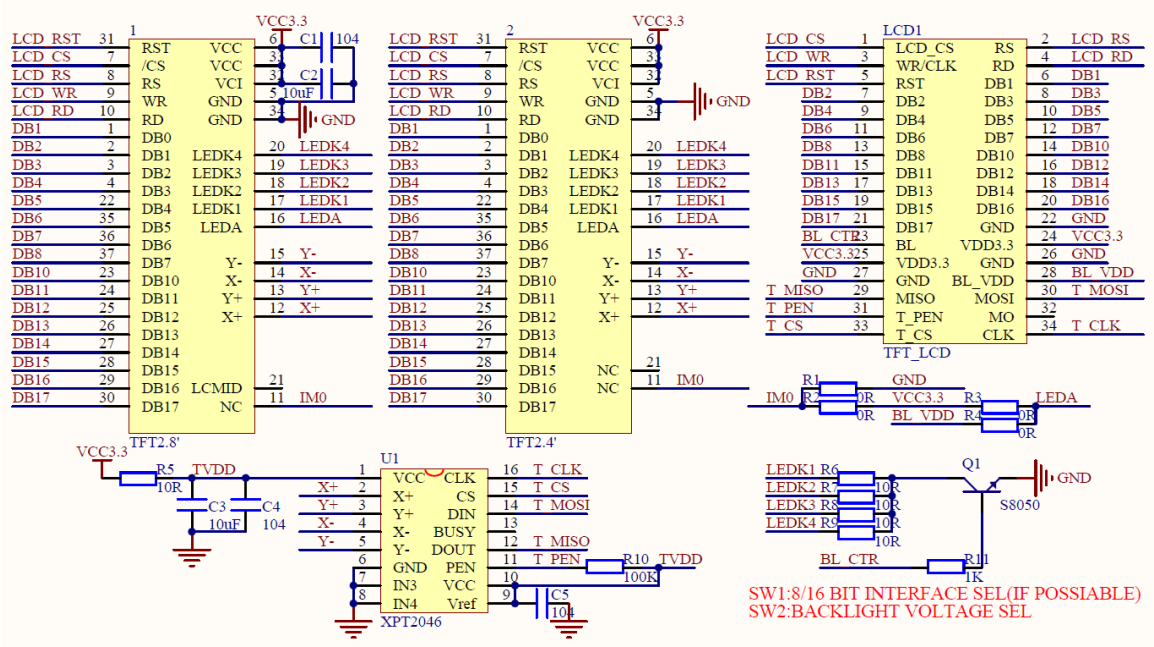

图 18.1.1.2　ALIENTEK 2.8 寸 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，接口定义如图 18.1.1.3 所示：

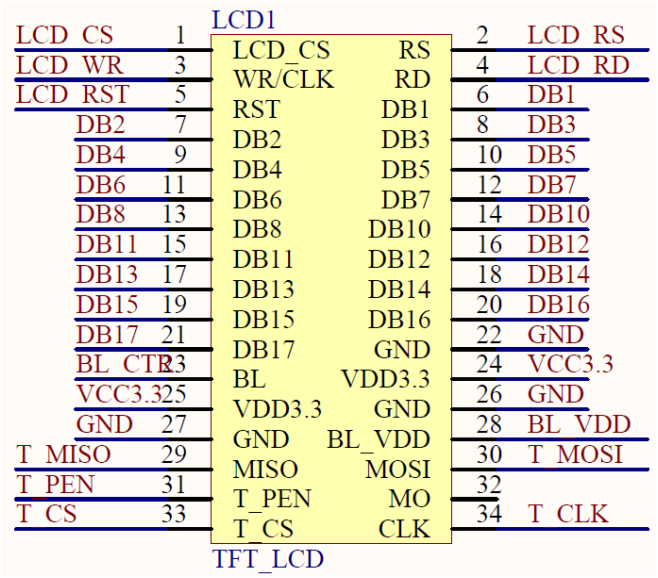

图 18.1.1.3　ALIENTEK 2.8 寸 TFTLCD 模块接口图

从图 18.1.1.3 可以看出，ALIENTEK TFTLCD 模块采用 16 位的并方式与外部连接，之所以不采用 8 位的方式，是因为彩屏的数据量比较大，尤其在显示图片的时候，如果用 8 位数据线，就会比 16 位方式慢一倍以上，我们当然希望速度越快越好，所以我们选择 16 位的接口。图 18.1.1.3 还列出了触摸屏芯片的接口，关于触摸屏本章我们不多介绍，后面的章节会有详细的介绍。该模块的 80 并口有如下一些信号线：

CS：TFTLCD 片选信号。

WR：向 TFTLCD 写入数据。

RD：从 TFTLCD 读取数据。

D[15：0]：16 位双向数据线。

RST：硬复位 TFTLCD。

RS：命令/数据标志（0，读写命令；1，读写数据）。

80 并口在上一节我们已经有详细的介绍了，这里我们就不再介绍，需要说明的是，TFTLCD 模块的 RST 信号线是直接接到 STM32F4 的复位脚上，并不由软件控制，这样可以省下来一个 IO 口。另外我们还需要一个背光控制线来控制 TFTLCD 的背光。所以，我们总共需要的 IO 口数目为 21 个。这里还需要注意，我们标注的 DB1~DB8，DB10~DB17，是相对于 LCD 控制 IC 标注的，实际上大家可以把他们就等同于 D0~D15，这样理解起来就比较简单一点。

ALIENTEK 提供 2.8/3.5/4.3/7 寸等不同尺寸的 TFTLCD 模块，其驱动芯片有很多种类型，比如有：ILI9341/ILI9325/RM68042/RM68021/ILI9320/ILI9328/LGDP4531/LGDP4535/SPFD5408/SSD1289/1505/B505/C505/NT35310/NT35510/SSD1963 等(具体的型号，大家可以通过下载本章实验代码，通过串口或者 LCD 显示查看)，这里我们仅以 ILI9341 控制器为例进行介绍，其他的控制基本都类似，我们就不详细阐述了。

ILI9341 液晶控制器自带显存，其显存总大小为 172800（240*320*18/8），即 18 位模式（26 万色）下的显存量。在 16 位模式下，ILI9341 采用 RGB565 格式存储颜色数据，此时 ILI9341 的 18 位数据线与 MCU 的 16 位数据线以及 LCD GRAM 的对应关系如图 18.1.1.4 所示：

9341总线	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MCU数据（16位）	D15	D14	D13	D12	D11	NC	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	NC
LCD GRAM（16位）	R[4]	R[3]	R[2]	R[1]	R[0]	NC	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	B[4]	B[3]	B[2]	B[1]	B[0]	NC

图 18.1.1.4　16 位数据与显存对应关系图

从图中可以看出，ILI9341 在 16 位模式下面，数据线有用的是：D17~D13 和 D11~D1，D0 和 D12 没有用到，实际上在我们 LCD 模块里面，ILI9341 的 D0 和 D12 压根就没有引出来，这样，ILI9341 的 D17~D13 和 D11~D1 对应 MCU 的 D15~D0。

这样 MCU 的 16 位数据，最低 5 位代表蓝色，中间 6 位为绿色，最高 5 位为红色。数值越大，表示该颜色越深。另外，特别注意 ILI9341 所有的指令都是 8 位的（高 8 位无效），且参数除了读写 GRAM 的时候是 16 位，其他操作参数，都是 8 位的，这个和 ILI9320 等驱动器不一样，必须加以注意。

接下来，我们介绍一下 ILI9341 的几个重要命令，因为 ILI9341 的命令很多，我们这里就不全部介绍了，有兴趣的大家可以找到 ILI9341 的 datasheet 看看。里面对这些命令有详细的介绍。我们将介绍：0XD3，0X36，0X2A，0X2B，0X2C，0X2E 等 6 条指令。

首先来看指令：0XD3，这个是读 ID4 指令，用于读取 LCD 控制器的 ID，该指令如表 18.1.1.1 所示：

顺序	控制			各位描述								HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	1	1	0	1	0	0	1	1	D3H
参数 1	1	↑	1	XX	X	X	X	X	X	X	X	X	X
参数 2	1	↑	1	XX	0	0	0	0	0	0	0	0	00H
参数 3	1	↑	1	XX	1	0	0	1	0	0	1	1	93H
参数 4	1	↑	1	XX	0	1	0	0	0	0	0	1	41H

表 18.1.1.1 0XD3 指令描述

从上表可以看出，0XD3 指令后面跟了 4 个参数，最后 2 个参数，读出来是 0X93 和 0X41，刚好是我们控制器 ILI9341 的数字部分，从而，通过该指令，即可判别所用的 LCD 驱动器是什么型号，这样，我们的代码，就可以根据控制器的型号去执行对应驱动 IC 的初始化代码，从而兼容不同驱动 IC 的屏，使得一个代码支持多款 LCD。

接下来看指令：0X36，这是存储访问控制指令，可以控制 ILI9341 存储器的读写方向，简单的说，就是在连续写 GRAM 的时候，可以控制 GRAM 指针的增长方向，从而控制显示方式（读 GRAM 也是一样）。该指令如表 18.1.1.2 所示：

顺序	控制			各位描述								HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	1	0	1	1	0	36H
参数	1	1	↑	XX	MY	MX	MV	ML	BGR	MH	0	0	0

表 18.1.1.2 0X36 指令描述

从上表可以看出，0X36 指令后面，紧跟一个参数，这里我们主要关注：MY、MX、MV 这三个位，通过这三个位的设置，我们可以控制整个 ILI9341 的全部扫描方向，如表 18.1.1.3 所示：

控制位			效果
MY	MX	MV	LCD 扫描方向（GRAM 自增方式）
0	0	0	从左到右, 从上到下
1	0	0	从左到右, 从下到上

0	1	0	从右到左,从上到下
1	1	0	从右到左,从下到上
0	0	1	从上到下,从左到右
0	1	1	从上到下,从右到左
1	0	1	从下到上,从左到右
1	1	1	从下到上,从右到左

表 18.1.1.3 MY、MX、MV 设置与 LCD 扫描方向关系表

这样，我们在利用 ILI9341 显示内容的时候，就有很大灵活性了，比如显示 BMP 图片，BMP 解码数据，就是从图片的左下角开始，慢慢显示到右上角，如果设置 LCD 扫描方向为从左到右，从下到上，那么我们只需要设置一次坐标，然后就不停的往 LCD 填充颜色数据即可，这样可以大大提高显示速度。

接下来看指令：0X2A，这是列地址设置指令，在从左到右，从上到下的扫描方式（默认）下面，该指令用于设置横坐标（x 坐标），该指令如表 18.1.1.4 所示：

顺序	控制			各位描述								HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	0	1	0	1	0	2AH
参数 1	1	1	↑	XX	SC15	SC14	SC13	SC12	SC11	SC10	SC9	SC8	SC
参数 2	1	1	↑	XX	SC7	SC6	SC5	SC4	SC3	SC2	SC1	SC0	
参数 3	1	1	↑	XX	EC15	EC14	EC13	EC12	EC11	EC10	EC9	EC8	EC
参数 4	1	1	↑	XX	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0	

表 18.1.1.4 0X2A 指令描述

在默认扫描方式时，该指令用于设置 x 坐标，该指令带有 4 个参数，实际上是 2 个坐标值：SC 和 EC，即列地址的起始值和结束值，SC 必须小于等于 EC，且 0≤SC/EC≤239。一般在设置 x 坐标的时候，我们只需要带 2 个参数即可，也就是设置 SC 即可，因为如果 EC 没有变化，我们只需要设置一次即可（在初始化 ILI9341 的时候设置），从而提高速度。

与 0X2A 指令类似，指令：0X2B，是页地址设置指令，在从左到右，从上到下的扫描方式（默认）下面，该指令用于设置纵坐标（y 坐标）。该指令如表 18.1.1.5 所示：

顺序	控制			各位描述								HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	0	1	0	1	0	2BH
参数 1	1	1	↑	XX	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SP
参数 2	1	1	↑	XX	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	
参数 3	1	1	↑	XX	EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	EP
参数 4	1	1	↑	XX	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0	

表 18.1.1.5 0X2B 指令描述

在默认扫描方式时，该指令用于设置 y 坐标，该指令带有 4 个参数，实际上是 2 个坐标值：SP 和 EP，即页地址的起始值和结束值，SP 必须小于等于 EP，且 0≤SP/EP≤319。一般在设置 y 坐标的时候，我们只需要带 2 个参数即可，也就是设置 SP 即可，因为如果 EP 没有变化，我们只需要设置一次即可（在初始化 ILI9341 的时候设置），从而提高速度。

接下来看指令：0X2C，该指令是写 GRAM 指令，在发送该指令之后，我们便可以往 LCD 的 GRAM 里面写入颜色数据了，该指令支持连续写，指令描述如表 18.1.1.6 所示：

顺序	控制			各位描述								HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	

指令	0	1	↑	XX	0	0	1	0	1	1	0	0	2CH
参数 1	1	1	↑	D1[15：0]									XX
……	1	1	↑	D2[15：0]									XX
参数 n	1	1	↑	Dn[15：0]									XX

表 18.1.1.6 0X2C 指令描述

从上表可知，在收到指令 0X2C 之后，数据有效位宽变为 16 位，我们可以连续写入 LCD GRAM 值，而 GRAM 的地址将根据 MY/MX/MV 设置的扫描方向进行自增。例如：假设设置的是从左到右，从上到下的扫描方式，那么设置好起始坐标（通过 SC，SP 设置）后，每写入一个颜色值，GRAM 地址将会自动自增 1（SC++），如果碰到 EC，则回到 SC，同时 SP++，一直到坐标：EC，EP 结束，其间无需再次设置的坐标，从而大大提高写入速度。

最后，来看看指令：0X2E，该指令是读 GRAM 指令，用于读取 ILI9341 的显存（GRAM），该指令在 ILI9341 的数据手册上面的描述是有误的，真实的输出情况如表 18.1.1.7 所示：

顺序	控制			各位描述											HEX
	RS	RD	WR	D15~D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX				0	0	1	0	1	1	1	0	2EH
参数 1	1	↑	1	XX												dummy
参数 2	1	↑	1	R1[4:0]	XX			G1[5:0]						XX		R1G1
参数 3	1	↑	1	B1[4:0]	XX			R2[4:0]						XX		B1R2
参数 4	1	↑	1	G2[5:0]			XX	B2[4:0]						XX		G2B2
参数 5	1	↑	1	R3[4:0]	XX			G3[5:0]						XX		R3G3
参数 N	1	↑	1	按以上规律输出												

表 18.1.1.7 0X2E 指令描述

该指令用于读取 GRAM，如表 18.1.1.7 所示，ILI9341 在收到该指令后，第一次输出的是 dummy 数据，也就是无效的数据，第二次开始，读取到的才是有效的 GRAM 数据（从坐标：SC，SP 开始），输出规律为：每个颜色分量占 8 个位，一次输出 2 个颜色分量。比如：第一次输出是 R1G1，随后的规律为：B1R2→G2B2→R3G3→B3R4→G4B4→R5G5... 以此类推。如果我们只需要读取一个点的颜色值，那么只需要接收到参数 3 即可，如果要连续读取（利用 GRAM 地址自增，方法同上），那么就按照上述规律去接收颜色数据。

以上，就是操作 ILI9341 常用的几个指令，通过这几个指令，我们便可以很好的控制 ILI9341 显示我们所要显示的内容了。

一般 TFTLCD 模块的使用流程如图 18.1.1.5：

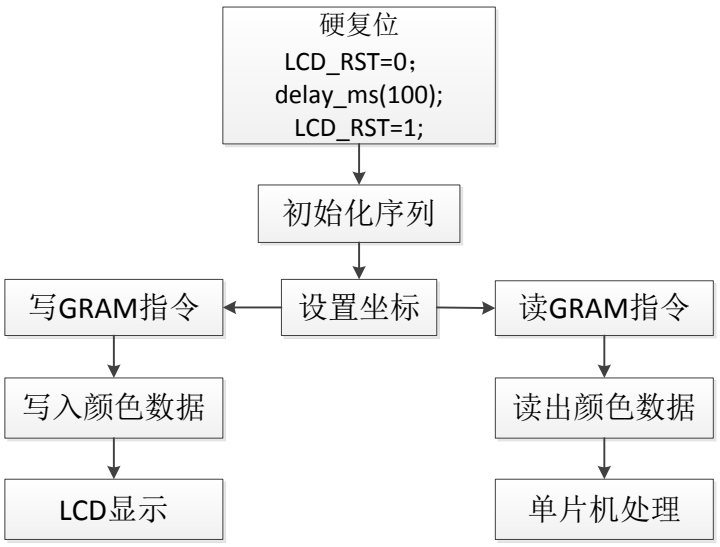

图 18.1.1.5 TFTLCD 使用流程

任何 LCD，使用流程都可以简单的用以上流程图表示。其中硬复位和初始化序列，只需要执行一次即可。而画点流程就是：设置坐标→写 GRAM 指令→写入颜色数据，然后在 LCD 上面，我们就可以看到对应的点显示我们写入的颜色了。读点流程为：设置坐标→读 GRAM 指令→读取颜色数据，这样就可以获取到对应点的颜色数据了。

以上只是最简单的操作，也是最常用的操作，有了这些操作，一般就可以正常使用 TFTLCD 了。接下来我们将该模块用来来显示字符和数字，通过以上介绍，我们可以得出 TFTLCD 显示需要的相关设置步骤如下：

1）设置 STM32F4 与 TFTLCD 模块相连接的 IO。

这一步，先将我们与 TFTLCD 模块相连的 IO 口进行初始化，以便驱动 LCD。这里我们用到的是 FSMC，FSMC 将在 18.1.2 节向大家详细介绍。

2）初始化 TFTLCD 模块。

即图 18.1.1.5 的初始化序列，这里我们没有硬复位 LCD，因为探索者 STM32F4 开发板的 LCD 接口，将 TFTLCD 的 RST 同 STM32F4 的 RESET 连接在一起了，只要按下开发板的 RESET 键，就会对 LCD 进行硬复位。初始化序列，就是向 LCD 控制器写入一系列的设置值（比如伽马校准），这些初始化序列一般 LCD 供应商会提供给客户，我们直接使用这些序列即可，不需要深入研究。在初始化之后，LCD 才可以正常使用。

3）通过函数将字符和数字显示到 TFTLCD 模块上。

这一步则通过图 18.1.1.5 左侧的流程，即：设置坐标→写 GRAM 指令→写 GRAM 来实现，但是这个步骤，只是一个点的处理，我们要显示字符/数字，就必须要多次使用这个步骤，从而达到显示字符/数字的目的，所以需要设计一个函数来实现数字/字符的显示，之后调用该函数，就可以实现数字/字符的显示了。

18.1.2 FSMC 简介

STM32F407 或 STM32F417 系列芯片都带有 FSMC 接口，ALIENTEK 探索者 STM32F4 开发板的主芯片为 STM32F407ZGT6，是带有 FSMC 接口的。

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡连接，STM32F4 的 FSMC 接口支持包括 SRAM、NAND FLASH、NOR FLASH 和 PSRAM 等存储器。FSMC 的框图如图 18.1.2.1 所示：

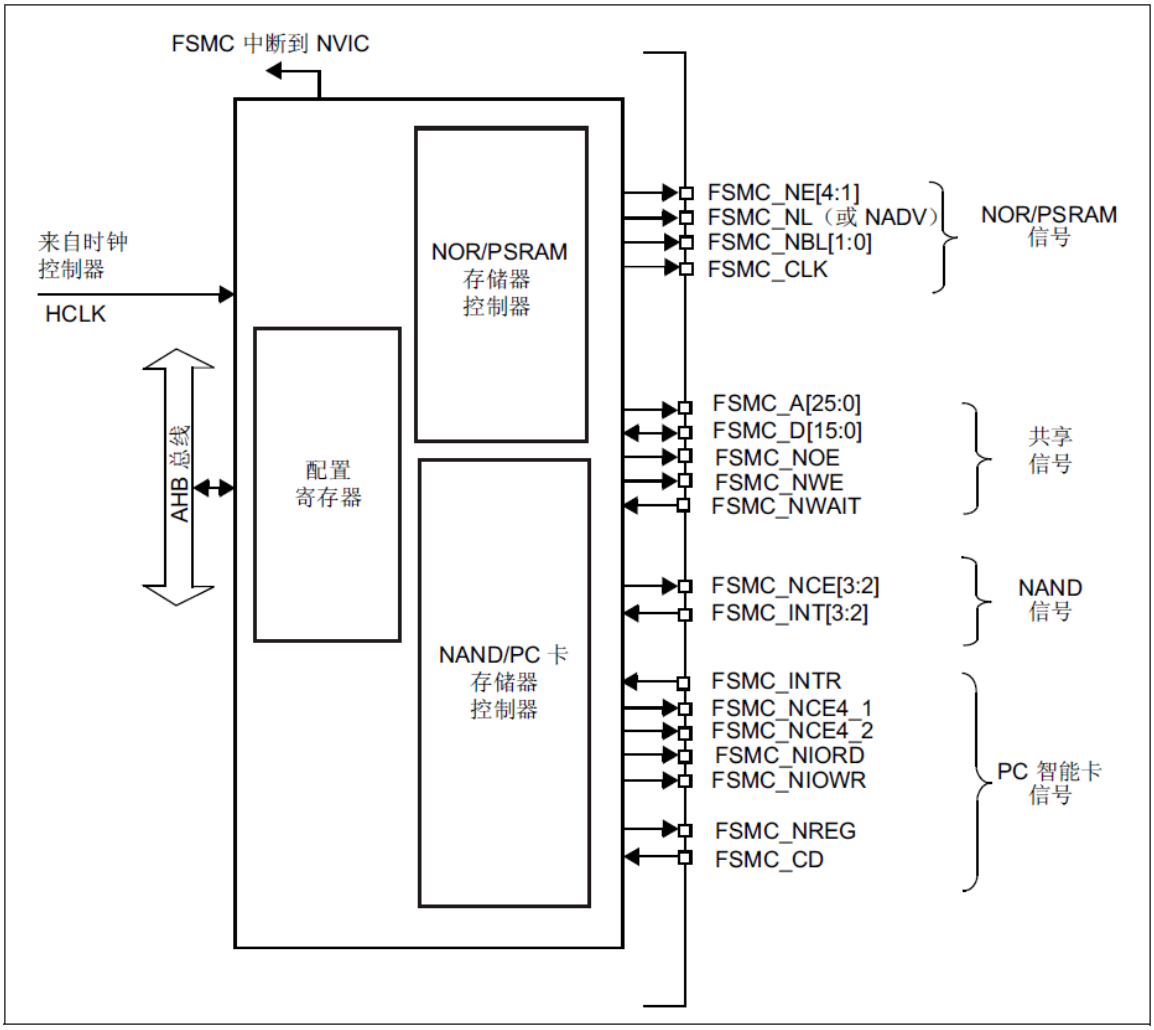

图 18.1.2.1 FSMC 框图

从上图我们可以看出，STM32F4 的 FSMC 将外部设备分为 2 类：NOR/PSRAM 设备、NAND/PC 卡设备。他们共用地址数据总线等信号，他们具有不同的 CS 以区分不同的设备，比如本章我们用到的 TFTLCD 就是用的 FSMC_NE4 做片选，其实就是将 TFTLCD 当成 SRAM 来控制。

这里我们介绍下为什么可以把 TFTLCD 当成 SRAM 设备用：首先我们了解下外部 SRAM 的连接，外部 SRAM 的控制一般有：地址线（如 A0~A18）、数据线（如 D0~D15）、写信号（WE）、读信号（OE）、片选信号（CS），如果 SRAM 支持字节控制，那么还有 UB/LB 信号。而 TFTLCD 的信号我们在 18.1.1 节有介绍，包括：RS、D0~D15、WR、RD、CS、RST 和 BL 等，其中真正在操作 LCD 的时候需要用到的就只有：RS、D0~D15、WR、RD 和 CS。其操作时序和 SRAM 的控制完全类似，唯一不同就是 TFTLCD 有 RS 信号，但是没有地址信号。

TFTLCD 通过 RS 信号来决定传送的数据是数据还是命令，本质上可以理解为一个地址信号，比如我们把 RS 接在 A0 上面，那么当 FSMC 控制器写地址 0 的时候，会使得 A0 变为 0，对 TFTLCD 来说，就是写命令。而 FSMC 写地址 1 的时候，A0 将会变为 1，对 TFTLCD 来说，就是写数据了。这样，就把数据和命令区分开了，他们其实就是对应 SRAM 操作的两个连续地址。当然 RS 也可以接在其他地址线上，探索者 STM32F4 开发板是把 RS 连接在 A6 上面的。

STM32F4 的 FSMC 支持 8/16/32 位数据宽度，我们这里用到的 LCD 是 16 位宽度的，

所以在设置的时候，选择 16 位宽就 OK 了。我们再来看看 FSMC 的外部设备地址映像，STM32F4 的 FSMC 将外部存储器划分为固定大小为 256M 字节的四个存储块，如图 18.1.2.2 所示：

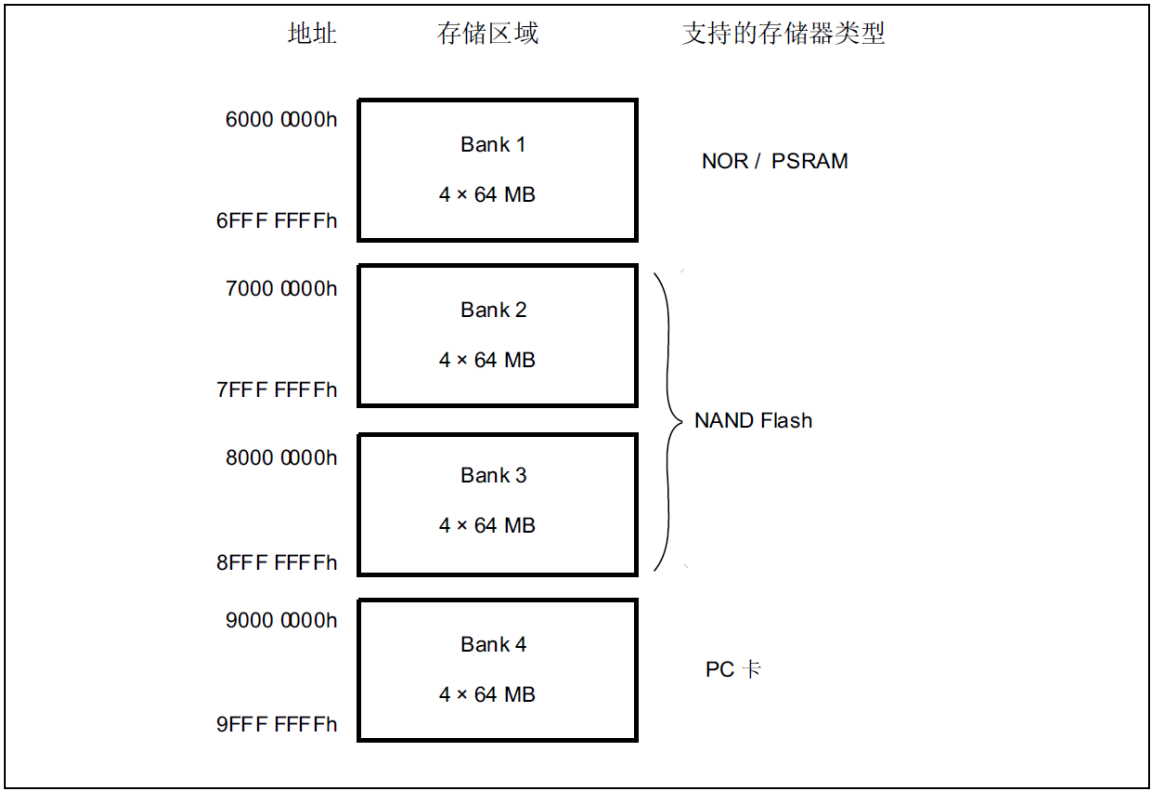

图 18.1.2.2 FSMC 存储块地址映像

从上图可以看出，FSMC 总共管理 1GB 空间，拥有 4 个存储块（Bank），本章，我们用到的是块 1，所以在本章我们仅讨论块 1 的相关配置，其他块的配置，请参考《STM32F4xx 中文参考手册》第 32 章（1191 页）的相关介绍。

STM32F4 的 FSMC 存储块 1（Bank1）被分为 4 个区，每个区管理 64M 字节空间，每个区都有独立的寄存器对所连接的存储器进行配置。Bank1 的 256M 字节空间由 28 根地址线（HADDR[27:0]）寻址。

这里 HADDR 是内部 AHB 地址总线，其中 HADDR[25:0]来自外部存储器地址 FSMC_A[25:0]，而 HADDR[26:27]对 4 个区进行寻址。如表 18.1.2.1 所示：

Bank1 所选区	片选信号	地址范围	HADDR	
			[27:26]	[25:0]
第 1 区	FSMC_NE1	0X6000,0000~63FF,FFFF	00	FSMC_A[25:0]
第 2 区	FSMC_NE2	0X6400,0000~67FF,FFFF	01	
第 3 区	FSMC_NE3	0X6800,0000~6BFF,FFFF	10	
第 4 区	FSMC_NE4	0X6C00,0000~6FFF,FFFF	11	

表 18.1.2.1 Bank1 存储区选择表

表 18.1.2.1 中，我们要特别注意 HADDR[25:0]的对应关系：

当 Bank1 接的是 16 位宽度存储器的时候：HADDR[25:1]→ FSMC_A[24:0]。

当 Bank1 接的是 8 位宽度存储器的时候：HADDR[25:0]→ FSMC_A[25:0]。

不论外部接 8 位/16 位宽设备，FSMC_A[0]永远接在外部设备地址 A[0]。这里，TFTLCD 使用的是 16 位数据宽度，所以 HADDR[0]并没有用到，只有 HADDR[25:1]是有效的，对应关系变为：HADDR[25:1]→ FSMC_A[24:0]，相当于右移了一位，这里请大家特别留意。另

外，HADDR[27:26]的设置，是不需要我们干预的，比如：当你选择使用 Bank1 的第三个区，即使用 FSMC_NE3 来连接外部设备的时候，即对应了 HADDR[27:26]=10，我们要做的就是配置对应第 3 区的寄存器组，来适应外部设备即可。STM32F4 的 FSMC 各 Bank 配置寄存器如表 18.1.2.2 所示：

内部控制器	存储块	管理的地址范围	支持的设备类型	配置寄存器
NOR FLASH 控制器	Bank1	0X6000,0000~ 0X6FFF,FFFF	SRAM/ROM NOR FLASH PSRAM	FSMC_BCR1/2/3/4 FSMC_BTR1/2/2/3 FSMC_BWTR1/2/3/4
NAND FLASH /PC CARD 控制器	Bank2	0X7000,0000~ 0X7FFF,FFFF	NAND FLASH	FSMC_PCR2/3/4 FSMC_SR2/3/4 FSMC_PMEM2/3/4 FSMC_PATT2/3/4 FSMC_PIO4 FSMC_ECCR2/3
	Bank3	0X8000,0000~ 0X8FFF,FFFF		
	Bank4	0X9000,0000~ 0X9FFF,FFFF	PC Card	

表 18.1.2.2 FSMC 各 Bank 配置寄存器表

对于 NOR FLASH 控制器，主要是通过 FSMC_BCRx、FSMC_BTRx 和 FSMC_BWTRx 寄存器设置（其中 x=1~4，对应 4 个区）。通过这 3 个寄存器，可以设置 FSMC 访问外部存储器的时序参数，拓宽了可选用的外部存储器的速度范围。FSMC 的 NOR FLASH 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要的设置的时间参数有 2 个：

1，HCLK 与 FSMC_CLK 的分频系数(CLKDIV)，可以为 2～16 分频；

2，同步突发访问中获得第 1 个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。FSMC 综合了 SRAM / ROM、PSRAM 和 NOR Flash 产品的信号特点，定义了 4 种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数，如表 18.1.2.3 所列：

时序模型		简单描述	时间参数
异步	Mode1	SRAM/CRAM 时序	DATAST、ADDSET
	ModeA	SRAM/CRAM OE 选通型时序	DATAST、ADDSET
	Mode2/B	NOR FLASH 时序	DATAST、ADDSET
	ModeC	NOR FLASH OE 选通型时序	DATAST、ADDSET
	ModeD	延长地址保持时间的异步时序	DATAST、ADDSET、ADDHLK
同步突发		根据同步时钟 FSMC_CK 读取多个顺序单元的数据	CLKDIV、DATLAT

表 18.1.2.3 NOR FLASH 控制器支持的时序模型

在实际扩展时，根据选用存储器的特征确定时序模型，从而确定各时间参数与存储器读 / 写周期参数指标之间的计算关系；利用该计算关系和存储芯片数据手册中给定的参数指标，可计算出 FSMC 所需要的各时间参数，从而对时间参数寄存器进行合理的配置。

本章，我们使用异步模式 A（ModeA）方式来控制 TFTLCD，模式 A 的读操作时序如图 18.1.2.3 所示：

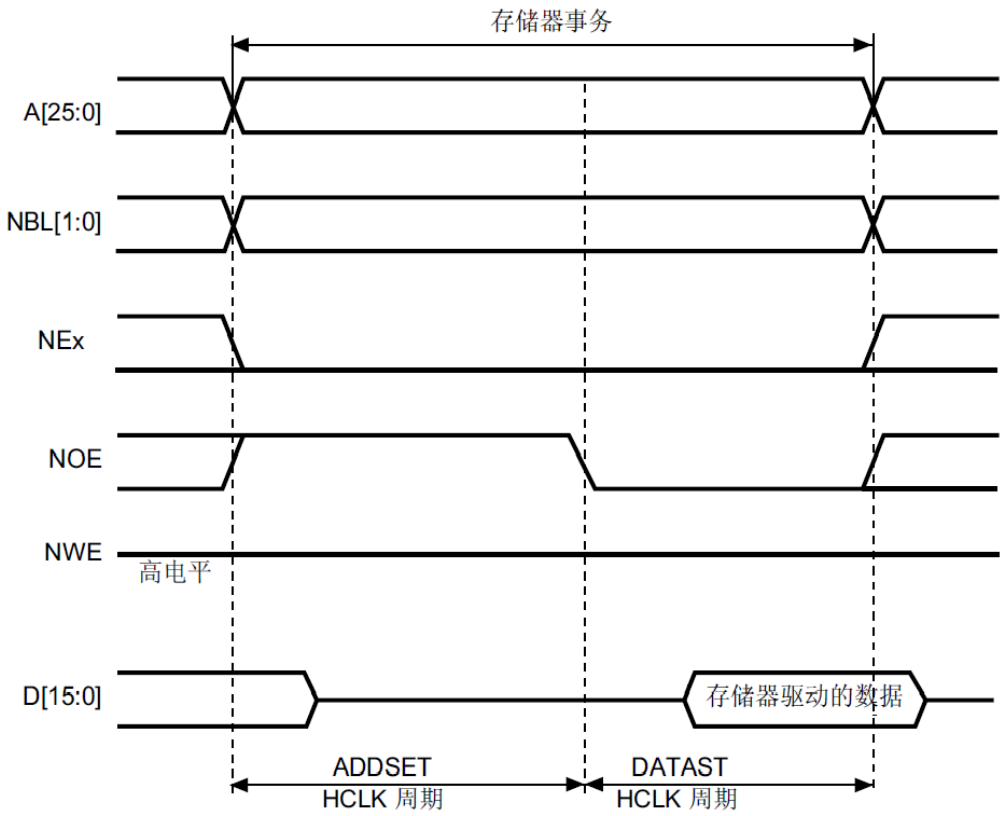

图 18.1.2.3 模式 A 读操作时序图

模式 A 支持独立的读写时序控制，这个对我们驱动 TFTLCD 来说非常有用，因为 TFTLCD 在读的时候，一般比较慢，而在写的时候可以比较快，如果读写用一样的时序，那么只能以读的时序为基准，从而导致写的速度变慢，或者在读数据的时候，重新配置 FSMC 的延时，在读操作完成的时候，再配置回写的时序，这样虽然也不会降低写的速度，但是频繁配置，比较麻烦。而如果有独立的读写时序控制，那么我们只要初始化的时候配置好，之后就不用再配置，既可以满足速度要求，又不需要频繁改配置。

模式 A 的写操作时序如图 18.1.2.4 所示：

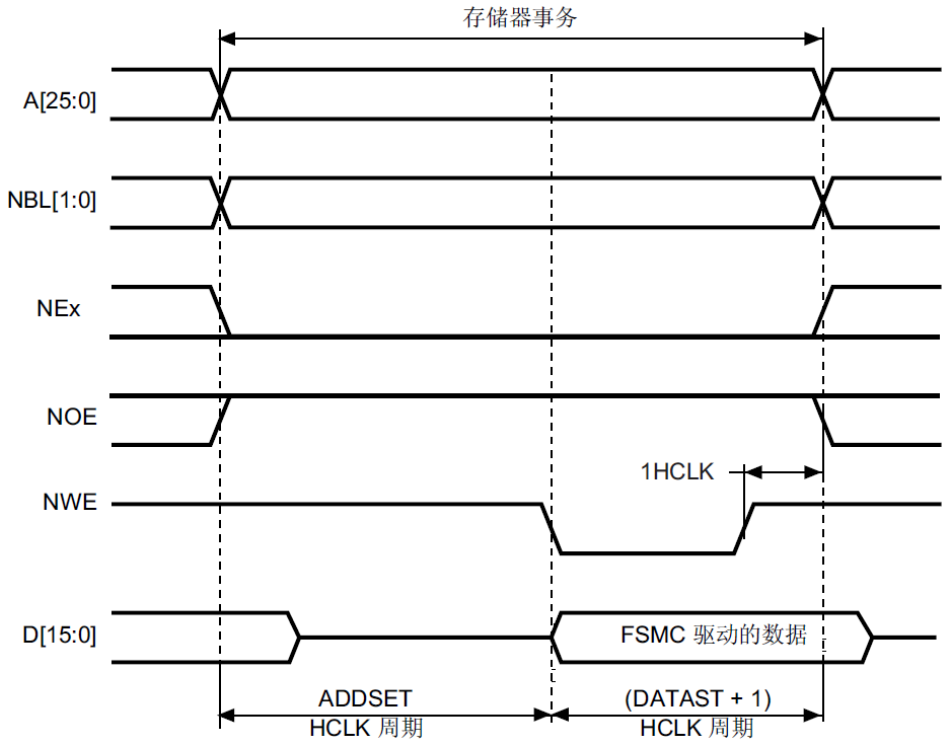

图 18.1.2.4 模式 A 写操作时序

图 18.1.2.3 和图 18.1.2.4 中的 ADDSET 与 DATAST，是通过不同的寄存器设置的，接下来我们讲解一下 Bank1 的几个控制寄存器

首先，我们介绍 SRAM/NOR 闪存片选控制寄存器：FSMC_BCRx（x=1~4），该寄存器各位描述如图 18.1.2.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												CBURSTRW	Reserved			ASCYCWAIT	EXTMOD	WAITEN	WREN	WAITCFG	WRAPMOD	WAITPOL	BURSTEN	Reserved	FACCEN	MWID		MTYP		MUXEN	MBKEN
												rw				rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

图 18.1.2.5 FSMC_BCRx 寄存器各位描述

该寄存器我们在本章用到的设置有：EXTMOD、WREN、MWID、MTYP 和 MBKEN 这几个设置，我们将逐个介绍。

EXTMOD：扩展模式使能位，也就是是否允许读写不同的时序，很明显，我们本章需要读写不同的时序，故该位需要设置为 1。

WREN：写使能位。我们需要向 TFTLCD 写数据，故该位必须设置为 1。

MWID[1:0]：存储器数据总线宽度。00，表示 8 位数据模式；01 表示 16 位数据模式；10 和 11 保留。我们的 TFTLCD 是 16 位数据线，所以设置 WMID[1:0]=01。

MTYP[1:0]：存储器类型。00 表示 SRAM、ROM；01 表示 PSRAM；10 表示 NOR FLASH;11 保留。前面提到，我们把 TFTLCD 当成 SRAM 用，所以需要设置 MTYP[1:0]=00。

MBKEN：存储块使能位。这个容易理解，我们需要用到该存储块控制 TFTLCD，当然要使能这个存储块了。

接下来，我们看看 SRAM/NOR 闪存片选时序寄存器：FSMC_BTRx（x=1~4），该寄存器各位描述如图 18.1.2.6 所示：

31 30	29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
Reserved	ACCMOD	DATLAT	CLKDIV	BUSTURN	DATAST	ADDHLD	ADDSET
	rw rw	rw rw rw rw	rw rw rw rw	rw rw rw rw	rw rw rw rw rw rw rw rw	rw rw rw rw	rw rw rw rw

图 18.1.2.6 FSMC_BTRx 寄存器各位描述

这个寄存器包含了每个存储器块的控制信息，可以用于 SRAM、ROM 和 NOR 闪存存储器。如果 FSMC_BCRx 寄存器中设置了 EXTMOD 位，则有两个时序寄存器分别对应读(本寄存器)和写操作(FSMC_BWTRx 寄存器)。因为我们要求读写分开时序控制，所以 EXTMOD 是使能了的，也就是本寄存器是读操作时序寄存器，控制读操作的相关时序。本章我们要用到的设置有：ACCMOD、DATAST 和 ADDSET 这三个设置。

ACCMOD[1:0]：访问模式。00 表示访问模式 A；01 表示访问模式 B；10 表示访问模式 C；11 表示访问模式 D，本章我们用到模式 A，故设置为 00。

DATAST[7:0]：数据保持时间。0 为保留设置，其他设置则代表保持时间为: DATAST 个 HCLK 时钟周期，最大为 255 个 HCLK 周期。对 ILI9341 来说，其实就是 RD 低电平持续时间，一般为 355ns。而一个 HCLK 时钟周期为 6ns 左右（1/168Mhz），为了兼容其他屏，我们这里设置 DATAST 为 60，也就是 60 个 HCLK 周期，时间大约是 360ns。

ADDSET[3:0]：地址建立时间。其建立时间为：ADDSET 个 HCLK 周期，最大为 15 个 HCLK 周期。对 ILI9341 来说，这里相当于 RD 高电平持续时间，为 90ns，我们设置 ADDSET 为 15，即 15*6=90ns。

最后，我们再来看看 SRAM/NOR 闪写时序寄存器：FSMC_BWTRx（x=1~4），该寄存器各位描述如图 18.1.2.7 所示：

31 30	29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
Res.	ACCMOD	DATLAT	CLKDIV	BUSTURN	DATAST	ADDHLD	ADDSET
	rw rw	rw rw rw rw	rw rw rw rw	rw rw rw rw	rw rw rw rw rw rw rw rw	rw rw rw rw	rw rw rw rw

图 18.1.2.7 FSMC_BWTRx 寄存器各位描述

该寄存器在本章用作写操作时序控制寄存器，需要用到的设置同样是：ACCMOD、DATAST 和 ADDSET 这三个设置。这三个设置的方法同 FSMC_BTRx 一模一样，只是这里对应的是写操作的时序，ACCMOD 设置同 FSMC_BTRx 一模一样，同样是选择模式 A，另外 DATAST 和 ADDSET 则对应低电平和高电平持续时间，对 ILI9341 来说，这两个时间只需要 15ns 就够了，比读操作快得多。所以我们这里设置 DATAST 为 2，即 3 个 HCLK 周期，时间约为 18ns。然后 ADDSET 设置为 3，即 3 个 HCLK 周期，时间为 18ns。

至此，我们对 STM32F4 的 FSMC 介绍就差不多了，通过以上两个小节的了解，我们可以开始写 LCD 的驱动代码了。不过，这里还要给大家做下科普，在 MDK 的寄存器定义里面，并没有定义 FSMC_BCRx、FSMC_BTRx、FSMC_BWTRx 等这个单独的寄存器，而是将他们进行了一些组合。

FSMC_BCRx 和 FSMC_BTRx，组合成 BTCR[8]寄存器组，他们的对应关系如下：

BTCR[0]对应 FSMC_BCR1，BTCR[1]对应 FSMC_BTR1

BTCR[2]对应 FSMC_BCR2，BTCR[3]对应 FSMC_BTR2

BTCR[4]对应 FSMC_BCR3，BTCR[5]对应 FSMC_BTR3

BTCR[6]对应 FSMC_BCR4，BTCR[7]对应 FSMC_BTR4

FSMC_BWTRx 则组合成 BWTR[7]，他们的对应关系如下：

BWTR[0]对应 FSMC_BWTR1，BWTR[2]对应 FSMC_BWTR2，

BWTR[4]对应 FSMC_BWTR3，BWTR[6]对应 FSMC_BWTR4，

BWTR[1]、BWTR[3]和 BWTR[5]保留，没有用到。

通过上面的讲解，通过对 FSMC 相关的寄存器的描述，大家对 FSMC 的原理有了一个初步的认识，如果还不熟悉的朋友，请一定要搜索网络资料理解 FSMC 的原理。只有理解了原理，使用库函数才可以得心应手。那么在库函数中是怎么实现 FSMC 的配置的呢？FSMC_BCRx，FSMC_BTRx 寄存器在库函数是通过什么函数来配置的呢？下面我们来讲解一下 FSMC 相关的库函数：

1) FSMC 初始化函数

根据前面的讲解，初始化 FSMC 主要是初始化三个寄存器 FSMC_BCRx，FSMC_BTRx，FSMC_BWTRx，那么在固件库中是怎么初始化这三个参数的呢？

固件库提供了 3 个 FSMC 初始化函数分别为

```
FSMC_NORSRAMInit()；
FSMC_NANDInit()；
FSMC_PCCARDInit()；
```

这三个函数分别用来初始化 4 种类型存储器。这里根据名字就很好判断对应关系。用来初始化 NOR 和 SRAM 使用同一个函数 FSMC_NORSRAMInit()。所以我们之后使用的 FSMC 初始化函数为 FSMC_NORSRAMInit()。下面我们看看函数定义：

```
void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct)；
```

这个函数只有一个入口参数，也就是 FSMC_NORSRAMInitTypeDef 类型指针变量，这个结构体的成员变量非常多，因为 FSMC 相关的配置项非常多。

```
typedef struct
{
  uint32_t FSMC_Bank;
  uint32_t FSMC_DataAddressMux;
  uint32_t FSMC_MemoryType;
  uint32_t FSMC_MemoryDataWidth;
  uint32_t FSMC_BurstAccessMode;
  uint32_t FSMC_AsynchronousWait;
  uint32_t FSMC_WaitSignalPolarity;
  uint32_t FSMC_WrapMode;
  uint32_t FSMC_WaitSignalActive;
  uint32_t FSMC_WriteOperation;
  uint32_t FSMC_WaitSignal;
  uint32_t FSMC_ExtendedMode;
  uint32_t FSMC_WriteBurst;
  FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;
  FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;
}FSMC_NORSRAMInitTypeDef;
```

从这个结构体我们可以看出，前面有 13 个基本类型（unit32_t）的成员变量，这 13 个参数是用来配置片选控制寄存器 FSMC_BCRx。最后面还有两个 SMC_NORSRAMTimingInitTypeDef 指针类型的成员变量。前面我们讲到，FSMC 有读时序和写时序之分，所以这里就是用来设置读时序和写时序的参数了， 也就是说，这两个参数是用来配置寄存器 FSMC_BTRx 和 FSMC_BWTRx，后面我们会讲解到。下面我们主要来看看模式 A 下的相关配置参数：

参数 FSMC_Bank 用来设置使用到的存储块标号和区号，前面讲过，我们是使用的存储块 1

区号 4，所以选择值为 FSMC_Bank1_NORSRAM4。
参数 FSMC_MemoryType 用来设置存储器类型，我们这里是 SRAM，所以选择值为 FSMC_MemoryType_SRAM。
参数 FSMC_MemoryDataWidth 用来设置数据宽度，可选 8 位还是 16 位，这里我们是 16 位数据宽度，所以选择值为 FSMC_MemoryDataWidth_16b。
参数 FSMC_WriteOperation 用来设置写使能，毫无疑问，我们前面讲解过我们要向 TFT 写数据，所以要写使能，这里我们选择 FSMC_WriteOperation_Enable。
参数 FSMC_ExtendedMode 是设置扩展模式使能位，也就是是否允许读写不同的时序，这里我们采取的读写不同时序，所以设置值为 FSMC_ExtendedMode_Enable。
上面的这些参数是与模式 A 相关的，下面我们也来稍微了解一下其他几个参数的意义吧：
参数 FSMC_DataAddressMux 用来设置地址/数据复用使能，若设置为使能，那么地址的低 16 位和数据将共用数据总线，仅对 NOR 和 PSRAM 有效，所以我们设置为默认值不复用，值
FSMC_DataAddressMux_Disable。
参数 FSMC_BurstAccessMode， FSMC_AsynchronousWait， FSMC_WaitSignalPolarity，FSMC_WaitSignalActive， FSMC_WrapMode， FSMC_WaitSignal FSMC_WriteBurst 和 FSMC_WaitSignal 这些参数在成组模式同步模式才需要设置，大家可以参考中文参考手册了解相关参数的意思。
接下来我们看看设置读写时序参数的两个变量 FSMC_ReadWriteTimingStruct 和 FSMC_WriteTimingStruct，他们都是 FSMC_NORSRAMTimingInitTypeDef 结构体指针类型，这两个参数在初始化的时候分别用来初始化片选控制寄存器 FSMC_BTRx 和写操作时序控制寄存器 FSMC_BWTRx。下面我们看看 FSMC_NORSRAMTimingInitTypeDef 类型的定义：

```
typedef struct
{
  uint32_t FSMC_AddressSetupTime;
  uint32_t FSMC_AddressHoldTime;
  uint32_t FSMC_DataSetupTime;
  uint32_t FSMC_BusTurnAroundDuration;
  uint32_t FSMC_CLKDivision;
  uint32_t FSMC_DataLatency;
  uint32_t FSMC_AccessMode;
}FSMC_NORSRAMTimingInitTypeDef;
```

这个结构体有 7 个参数用来设置 FSMC 读写时序。其实这些参数的意思我们前面在讲解 FSMC 的时序的时候有提到，主要是设计地址建立保持时间，数据建立时间等等配置，对于我们的实验中，读写时序不一样，读写速度要求不一样，所以对于参数 FSMC_DataSetupTime 设置了不同的值，大家可以对照理解一下。记住，这些参数的意义在前面讲解 FSMC_BTRx 和 FSMC_BWTRx 寄存器的时候都有提到，大家可以翻过去看看。

2) FSMC 使能函数

FSMC 对不同的存储器类型同样提供了不同的使能函数：

```
void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);
void FSMC_NANDCmd(uint32_t FSMC_Bank, FunctionalState NewState);
void FSMC_PCCARDCmd(FunctionalState NewState);
```

这个就比较好理解，我们这里不讲解，我们是 SRAM,所以使用的是第一个函数。

18.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） TFTLCD 模块

TFTLCD 模块的电路见图 18.1.1.2，这里我们介绍 TFTLCD 模块与 ALIETEK 探索者 STM32F4 开发板的连接，探索者 STM32F4 开发板底板的 LCD 接口和 ALIENTEK TFTLCD 模块直接可以对插，连接关系如图 18.2.1 所示：

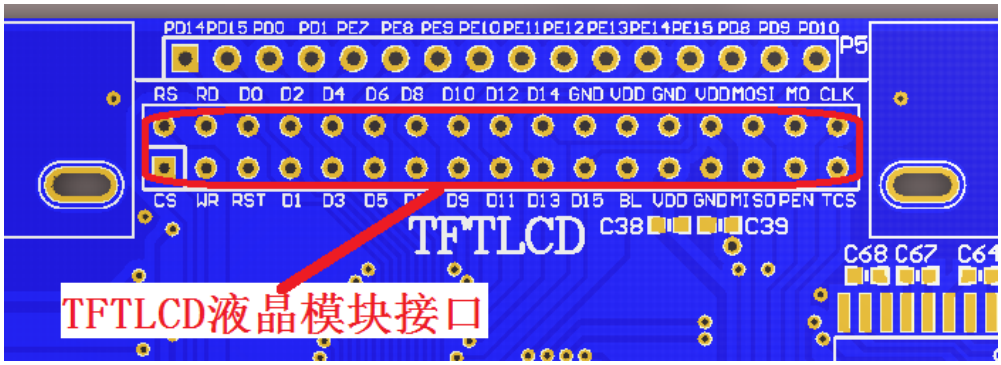

图 18.2.1 TFTLCD 与开发板连接示意图

图 18.2.1 中圈出来的部分就是连接 TFTLCD 模块的接口，液晶模块直接插上去即可。

在硬件上，TFTLCD 模块与探索者 STM32F4 开发板的 IO 口对应关系如下：

LCD_BL(背光控制)对应 PB0;

LCD_CS 对应 PG12 即 FSMC_NE4;

LCD _RS 对应 PF12 即 FSMC_A6;

LCD _WR 对应 PD5 即 FSMC_NWE;

LCD _RD 对应 PD4 即 FSMC_NOE;

LCD _D[15:0]则直接连接在 FSMC_D15~FSMC_D0;

这些线的连接，探索者 STM32F4 开发板的内部已经连接好了，我们只需要将 TFTLCD 模块插上去就好了。实物连接如图 18.2.2 所示：

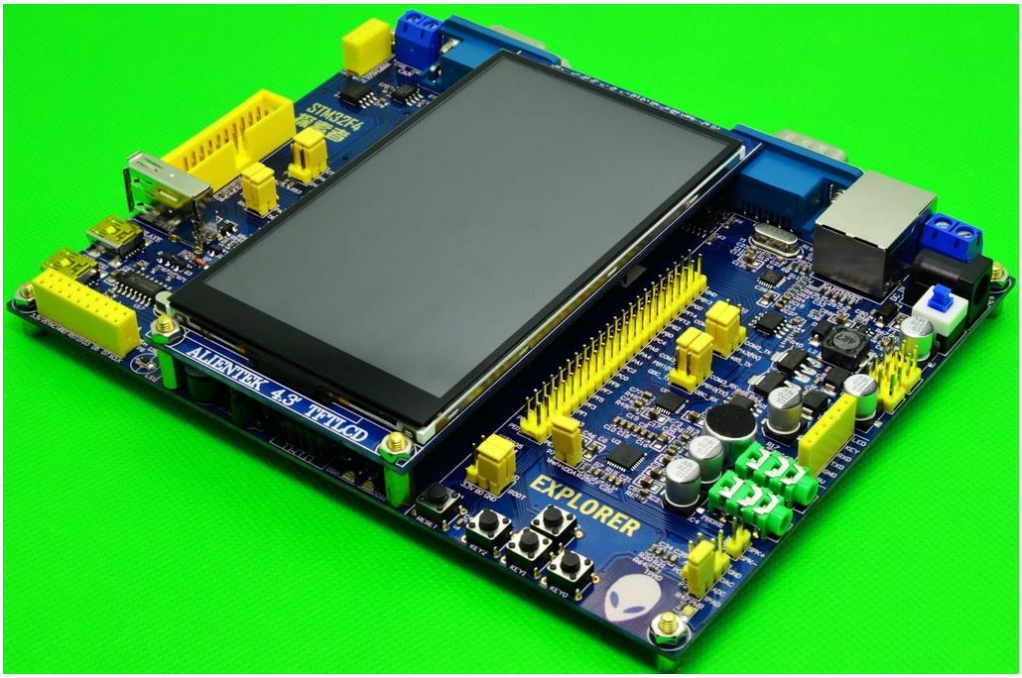

图 18.2.2 TFTLCD 与开发板连接实物图

18.3 软件设计

打开我们光盘的 TFT LCD 显示实验工程可以看到我们添加了两个文件 lcd.c 和头文件 lcd.h。同时，FSMC 相关的库函数分布在 stm32f4xx_fsmc.c 文件和头文件 stm32f4xx_fsmc.h 中。所以我们在工程中要引入 stm32f4xx_fsmc.c 源文件。

在 lcd.c 里面代码比较多，我们这里就不贴出来了，只针对几个重要的函数进行讲解。完整版的代码见光盘→4，程序源码→标准例程-库函数版本→实验 13 TFTLCD 显示实验的 lcd.c 文件。

本实验，我们用到 FSMC 驱动 LCD，通过前面的介绍，我们知道 TFTLCD 的 RS 接在 FSMC 的 A6 上面，CS 接在 FSMC_NE4 上，并且是 16 位数据总线。即我们使用的是 FSMC 存储器 1 的第 4 区，我们定义如下 LCD 操作结构体（在 lcd.h 里面定义）：

```
//LCD 操作结构体
typedef struct
{
    vu16 LCD_REG;
    vu16 LCD_RAM;
} LCD_TypeDef;
//使用 NOR/SRAM 的 Bank1.sector4,地址位 HADDR[27,26]=11 A6 作为数据命令区分线
//注意 16 位数据总线时，STM32 内部地址会右移一位对齐!
#define LCD_BASE        ((u32)(0x6C000000 | 0x0000007E))
#define LCD             ((LCD_TypeDef *) LCD_BASE)
```

其中 LCD_BASE，必须根据我们外部电路的连接来确定，我们使用 Bank1.sector4 就是从地址 0X6C000000 开始，而 0X0000007E，则是 A6 的偏移量，这里很多朋友不理解这个偏移量的概念，简单说明下：以 A6 为例，7E 转换成二进制就是：1111110，而 16 位数据时，地址右移一位对齐，那么实际对应到地址引脚的时候，就是：A6:A0=0111111，此时 A6 是 0，但是如果 16 位地址再加 1（注意：对应到 8 位地址是加 2，即 7E+0X02），那么：A6:A0=1000000，此时 A6 就是 1 了，即实现了对 RS 的 0 和 1 的控制。

我们将这个地址强制转换为 LCD_TypeDef 结构体地址，那么可以得到 LCD->LCD_REG 的地址就是 0X6C00,007E，对应 A6 的状态为 0(即 RS=0)，而 LCD-> LCD_RAM 的地址就是 0X6C00,0080（结构体地址自增），对应 A6 的状态为 1（即 RS=1）。

所以，有了这个定义，当我们要往 LCD 写命令/数据的时候，可以这样写：

```
LCD->LCD_REG=CMD;  //写命令
LCD->LCD_RAM=DATA; //写数据
```

而读的时候反过来操作就可以了，如下所示：

```
CMD= LCD->LCD_REG;//读 LCD 寄存器
 DATA = LCD->LCD_RAM;//读 LCD 数据
```

这其中，CS、WR、RD 和 IO 口方向都是由 FSMC 控制，不需要我们手动设置了。接下来，我们先介绍一下 lcd.h 里面的另一个重要结构体:

```
//LCD 重要参数集
typedef struct
{
    u16 width;          //LCD  宽度
    u16 height;         //LCD  高度
```

```
    u16 id;                 //LCD ID
    u8  dir;                //横屏还是竖屏控制：0，竖屏；1，横屏。
    u16 wramcmd;            //开始写 gram 指令
    u16 setxcmd;            //设置 x 坐标指令
    u16 setycmd;            //设置 y 坐标指令
}_lcd_dev;
//LCD 参数
extern _lcd_dev lcddev; //管理 LCD 重要参数
```

该结构体用于保存一些 LCD 重要参数信息，比如 LCD 的长宽、LCD ID（驱动 IC 型号）、LCD 横竖屏状态等，这个结构体虽然占用了十几个字节的内存，但是却可以让我们的驱动函数支持不同尺寸的 LCD，同时可以实现 LCD 横竖屏切换等重要功能，所以还是利大于弊的。有了以上了解，下面我们开始介绍 lcd.c 里面的一些重要函数。

先看 7 个简单，但是很重要的函数：

```
//写寄存器函数
//regval:寄存器值
void LCD_WR_REG(vu16 regval)
{   regval=regval;      //使用-O2 优化的时候,必须插入的延时
    LCD->LCD_REG=regval;//写入要写的寄存器序号
}
//写 LCD 数据
//data:要写入的值
void LCD_WR_DATA(vu16 data)
{   data=data;          //使用-O2 优化的时候,必须插入的延时
    LCD->LCD_RAM=data;
}
//读 LCD 数据
//返回值:读到的值
u16 LCD_RD_DATA(void)
{   vu16 ram;           //防止被优化
    ram=LCD->LCD_RAM;
    return ram;
}
//写寄存器
//LCD_Reg:寄存器地址
//LCD_RegValue:要写入的数据
void LCD_WriteReg(vu16 LCD_Reg, vu16 LCD_RegValue)
{   LCD->LCD_REG = LCD_Reg;         //写入要写的寄存器序号
    LCD->LCD_RAM = LCD_RegValue;  //写入数据
}
//读寄存器
//LCD_Reg:寄存器地址
//返回值:读到的数据
u16 LCD_ReadReg(vu16 LCD_Reg)
{   LCD_WR_REG(LCD_Reg);        //写入要读的寄存器序号
```

```
    delay_us(5);
    return LCD_RD_DATA();          //返回读到的值
}
//开始写 GRAM
void LCD_WriteRAM_Prepare(void)
{   LCD->LCD_REG=lcddev.wramcmd;
}
//LCD 写 GRAM
//RGB_Code:颜色值
void LCD_WriteRAM(u16 RGB_Code)
{   LCD->LCD_RAM = RGB_Code;//写十六位 GRAM
}
```

因为 FSMC 自动控制了 WR/RD/CS 等这些信号，所以这 7 个函数实现起来都非常简单，我们就不多说，注意，上面有几个函数，我们添加了一些对 MDK －O2 优化的支持，去掉的话，在-O2 优化的时候会出问题。这些函数实现功能见函数前面的备注，通过这几个简单函数的组合，我们就可以对 LCD 进行各种操作了。

第七个要介绍的函数是坐标设置函数，该函数代码如下：

```
//设置光标位置
//Xpos:横坐标
//Ypos:纵坐标
void LCD_SetCursor(u16 Xpos, u16 Ypos)
{
    if(lcddev.id==0X9341||lcddev.id==0X5310)
    {
        LCD_WR_REG(lcddev.setxcmd);
        LCD_WR_DATA(Xpos>>8);
        LCD_WR_DATA(Xpos&0XFF);
        LCD_WR_REG(lcddev.setycmd);
        LCD_WR_DATA(Ypos>>8);
        LCD_WR_DATA(Ypos&0XFF);
    }else if(lcddev.id==0X6804)
    {
        if(lcddev.dir==1)Xpos=lcddev.width-1-Xpos;//横屏时处理
        LCD_WR_REG(lcddev.setxcmd);
        LCD_WR_DATA(Xpos>>8);
        LCD_WR_DATA(Xpos&0XFF);
        LCD_WR_REG(lcddev.setycmd);
        LCD_WR_DATA(Ypos>>8);
        LCD_WR_DATA(Ypos&0XFF);
    }else if(lcddev.id==0X5510)
    {
        LCD_WR_REG(lcddev.setxcmd);
        LCD_WR_DATA(Xpos>>8);
        LCD_WR_REG(lcddev.setxcmd+1);
```

```
            LCD_WR_DATA(Xpos&0XFF);
            LCD_WR_REG(lcddev.setycmd);
            LCD_WR_DATA(Ypos>>8);
            LCD_WR_REG(lcddev.setycmd+1);
            LCD_WR_DATA(Ypos&0XFF);
        }else
        {
            if(lcddev.dir==1)Xpos=lcddev.width-1-Xpos;//横屏其实就是调转 x,y 坐标
            LCD_WriteReg(lcddev.setxcmd, Xpos);
            LCD_WriteReg(lcddev.setycmd, Ypos);
        }
    }
```

该函数实现将 LCD 的当前操作点设置到指定坐标(x,y)。因为 9341/5310/6804/5510 等的设置同其他屏有些不太一样，所以进行了区别对待。

接下来我们介绍第八个函数：画点函数。该函数实现代码如下：

```
    //画点
    //x,y:坐标
    //POINT_COLOR:此点的颜色
    void LCD_DrawPoint(u16 x,u16 y)
    {
        LCD_SetCursor(x,y);          //设置光标位置
        LCD_WriteRAM_Prepare(); //开始写入 GRAM
        LCD->LCD_RAM=POINT_COLOR;
    }
```

该函数实现比较简单，就是先设置坐标，然后往坐标写颜色。其中 POINT_COLOR 是我们定义的一个全局变量，用于存放画笔颜色，顺带介绍一下另外一个全局变量：BACK_COLOR，该变量代表 LCD 的背景色。LCD_DrawPoint 函数虽然简单，但是至关重要，其他几乎所有上层函数，都是通过调用这个函数实现的。

有了画点，当然还需要有读点的函数，第九个介绍的函数就是读点函数，用于读取 LCD 的 GRAM，这里说明一下，为什么 OLED 模块没做读 GRAM 的函数，而这里做了。因为 OLED 模块是单色的，所需要全部 GRAM 也就 1K 个字节，而 TFTLCD 模块为彩色的，点数也比 OLED 模块多很多，以 16 位色计算，一款 320×240 的液晶，需要 320×240×2 个字节来存储颜色值，也就是也需要 150K 字节，这对任何一款单片机来说，都不是一个小数目了。而且我们在图形叠加的时候，可以先读回原来的值，然后写入新的值，在完成叠加后，我们又恢复原来的值。这样在做一些简单菜单的时候，是很有用的。这里我们读取 TFTLCD 模块数据的函数为 LCD_ReadPoint，该函数直接返回读到的 GRAM 值。该函数使用之前要先设置读取的 GRAM 地址，通过 LCD_SetCursor 函数来实现。LCD_ReadPoint 的代码如下：

```
    //读取个某点的颜色值
    //x,y:坐标
    //返回值:此点的颜色
    u16 LCD_ReadPoint(u16 x,u16 y)
    {
        vu16 r=0,g=0,b=0;
        if(x>=lcddev.width||y>=lcddev.height)return 0;          //超过了范围,直接返回
```

```
        LCD_SetCursor(x,y);
        if(lcddev.id==0X9341||lcddev.id==0X6804||lcddev.id==0X5310)LCD_WR_REG(0X2E
);
        //9341/6804/3510  发送读 GRAM 指令
        else if(lcddev.id==0X5510)LCD_WR_REG(0X2E00);//5510  发送读 GRAM 指令
        else LCD_WR_REG(R34);                                   //其他 IC 发送读 GRAM 指令
        if(lcddev.id==0X9320)opt_delay(2);                      //FOR 9320,延时 2us
        LCD_RD_DATA();                                          //dummy Read
        opt_delay(2);
        r=LCD_RD_DATA();                                        //实际坐标颜色
        if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510)

        {     //9341/NT35310/NT35510 要分 2 次读出

              opt_delay(2);
              b=LCD_RD_DATA();
              g=r&0XFF;//9341/5310/5510 等,第一次读取的是 RG 的值,R 在前,G 在后,各占 8
位
              g<<=8;
        }
        if(lcddev.id==0X9325||lcddev.id==0X4535||lcddev.id==0X4531||lcddev.id==0XB505||
           lcddev.id==0XC505)return r;    //这几种 IC 直接返回颜色值
        else                  if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510)return
(((r>>11)<<11)
                    |((g>>10)<<5)|(b>>11));    //ILI9341/NT35310/NT35510 需要公式转换一下
        else return LCD_BGR2RGB(r);    //其他 IC
    }
```

在 LCD_ReadPoint 函数中，因为我们的代码不止支持一种 LCD 驱动器，所以，我们根据不同的 LCD 驱动器（(lcddev.id）型号，执行不同的操作，以实现对各个驱动器兼容，提高函数的通用性。

第十个要介绍的是字符显示函数 LCD_ShowChar，该函数同前面 OLED 模块的字符显示函数差不多，但是这里的字符显示函数多了 1 个功能，就是可以以叠加方式显示，或者以非叠加方式显示。叠加方式显示多用于在显示的图片上再显示字符。非叠加方式一般用于普通的显示。该函数实现代码如下：

```
    //在指定位置显示一个字符
    //x,y:起始坐标
    //num:要显示的字符:" "--->"~"
    //size:字体大小  12/16/24
    //mode:叠加方式(1)还是非叠加方式(0)
    void LCD_ShowChar(u16 x,u16 y,u8 num,u8 size,u8 mode)
    {
        u8 temp,t1,t; u16 y0=y;
        u8  csize=(size/8+((size%8)?1:0))*(size/2);//得到字体一个字符对应点阵集所占的字
节数
```

```
        //设置窗口
        num=num-' ';//得到偏移后的值
        for(t=0;t<csize;t++)
        {
            if(size==12)temp=asc2_1206[num][t];         //调用 1206 字体
            else if(size==16)temp=asc2_1608[num][t];  //调用 1608 字体
            else if(size==24)temp=asc2_2412[num][t];  //调用 2412 字体
            else return;                                          //没有的字库
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)LCD_Fast_DrawPoint(x,y,POINT_COLOR);
                else if(mode==0)LCD_Fast_DrawPoint(x,y,BACK_COLOR);
                temp<<=1;
                y++;
                if(y>=lcddev.height)return;          //超区域了
                if((y-y0)==size)
                {
                    y=y0; x++;
                    if(x>=lcddev.width)return;    //超区域了
                    break;
                }
            }
        }
    }
```

在 LCD_ShowChar 函数里面，我们采用快速画点函数 LCD_Fast_DrawPoint 来画点显示字符，该函数同 LCD_DrawPoint 一样，只是带了颜色参数，且减少了函数调用的时间，详见本例程源码。该代码中我们用到了三个字符集点阵数据数组 asc2_2412、asc2_1206 和 asc2_1608，这几个字符集的点阵数据的提取方式，同十七章介绍的提取方法是一模一样的。详细请参考第十七章。

最后，我们再介绍一下 TFTLCD 模块的初始化函数 LCD_Init，该函数先初始化 STM32 与 TFTLCD 连接的 IO 口，并配置 FSMC 控制器，然后读取 LCD 控制器的型号，根据控制 IC 的型号执行不同的初始化代码，其简化代码如下：

```
    void LCD_Init(void)
    {
      vu32 i=0;
      GPIO_InitTypeDef    GPIO_InitStructure;
      FSMC_NORSRAMInitTypeDef    FSMC_NORSRAMInitStructure;
      FSMC_NORSRAMTimingInitTypeDef    readWriteTiming;
      FSMC_NORSRAMTimingInitTypeDef    writeTiming;

      //① GPIO,FSMC 时钟使能
      RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOD
          |RCC_AHB1Periph_GPIOE|RCC_AHB1Periph_GPIOF|RCC_AHB1Periph_GPIO
G,
```

```
                                                    ENABLE);//使能 PD,PE,PF,PG 时钟
RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC,ENABLE);//使能 FSMC 时钟

//② GPIO 初始化设置
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;//PB15 推挽输出,控制背光
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化 //PB15 推挽输出,控制背光

GPIO_InitStructure.GPIO_Pin = (3<<0)|(3<<4)|(7<<8)|(3<<14);
                                                    //PD0,1,4,5,8,9,10,14,15 AF OUT
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOD, &GPIO_InitStructure);//初始化

GPIO_InitStructure.GPIO_Pin = (0X1FF<<7);//PE7~15,AF OUT
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOE, &GPIO_InitStructure);//初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;//PF12,FSMC_A6
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;//PF12,FSMC_A6
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOG, &GPIO_InitStructure);//初始化

//③ 引脚复用映射设置
GPIO_PinAFConfig(GPIOD,GPIO_PinSource0,GPIO_AF_FSMC);//PD0,AF12
GPIO_PinAFConfig(GPIOD,GPIO_PinSource1,GPIO_AF_FSMC);//PD1,AF12
GPIO_PinAFConfig(GPIOD,GPIO_PinSource4,GPIO_AF_FSMC);
```

```
	GPIO_PinAFConfig(GPIOD,GPIO_PinSource5,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOD,GPIO_PinSource8,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOD,GPIO_PinSource9,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOD,GPIO_PinSource10,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOD,GPIO_PinSource14,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOD,GPIO_PinSource15,GPIO_AF_FSMC);//PD15,AF12

	GPIO_PinAFConfig(GPIOE,GPIO_PinSource7,GPIO_AF_FSMC);//PE7,AF12
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource8,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource9,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource10,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource11,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource12,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource13,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource14,GPIO_AF_FSMC);
	GPIO_PinAFConfig(GPIOE,GPIO_PinSource15,GPIO_AF_FSMC);//PE15,AF12

	GPIO_PinAFConfig(GPIOF,GPIO_PinSource12,GPIO_AF_FSMC);//PF12,AF12
	GPIO_PinAFConfig(GPIOG,GPIO_PinSource12,GPIO_AF_FSMC);

	//④FSMC 初始化
	readWriteTiming.FSMC_AddressSetupTime = 0XF; //地址建立时间为 16 个 HCLK
	readWriteTiming.FSMC_AddressHoldTime = 0x00;  //地址保持时间模式 A 未用到
	readWriteTiming.FSMC_DataSetupTime = 24;//数据保存时间为 25 个 HCLK
	readWriteTiming.FSMC_BusTurnAroundDuration = 0x00;
	readWriteTiming.FSMC_CLKDivision = 0x00;
	readWriteTiming.FSMC_DataLatency = 0x00;
	readWriteTiming.FSMC_AccessMode = FSMC_AccessMode_A;    //模式 A

	writeTiming.FSMC_AddressSetupTime =8;      //地址建立时间（ADDSET）为 8 个
HCLK
	writeTiming.FSMC_AddressHoldTime = 0x00; //地址保持时间
	writeTiming.FSMC_DataSetupTime = 8;    //数据保存时间为 6ns*9 个 HCLK=54ns
	writeTiming.FSMC_BusTurnAroundDuration = 0x00;
	writeTiming.FSMC_CLKDivision = 0x00;
	writeTiming.FSMC_DataLatency = 0x00;
	writeTiming.FSMC_AccessMode = FSMC_AccessMode_A;    //模式 A

	FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM4;
                                        //这里我们使用 NE4 ，也就对应 BTCR[6],[7]。
	FSMC_NORSRAMInitStructure.FSMC_DataAddressMux
                                        =FSMC_DataAddressMux_Disable;  //  不复用数据地
址
	FSMC_NORSRAMInitStructure.FSMC_MemoryType =FSMC_MemoryType_SRAM;
```

```
                                                // FSMC_MemoryType_SRAM;
        FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth
                                        =  FSMC_MemoryDataWidth_16b;//存储器数据宽度为
16bit
        FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode
                                =FSMC_BurstAccessMode_Disable;//
FSMC_BurstAccessMode_Disable;
        FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity
                                                =FSMC_WaitSignalPolarity_Low;
        FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait
                                                =FSMC_AsynchronousWait_Disable;
        FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
        FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive
                                                  =FSMC_WaitSignalActive_BeforeWaitState;
        FSMC_NORSRAMInitStructure.FSMC_WriteOperation                                   =
FSMC_WriteOperation_Enable;
                                                        //存储器写使能
        FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
        FSMC_NORSRAMInitStructure.FSMC_ExtendedMode                                     =
FSMC_ExtendedMode_Enable;
                                                        // 读写使用不同的时序
        FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
        FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &readWriteTiming;
                                                        //读写时序
        FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &writeTiming;   //写时序

        FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);   //初始化 FSMC 配置

        //⑤使能 FSMC
        FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM4, ENABLE);   // 使能 BANK1

        delay_ms(50); // delay 50 ms
        lcddev.id = LCD_ReadReg(0x0000);
        //⑥不同的 LCD 驱动器不同的初始化设置
        if(lcddev.id<0XFF||lcddev.id==0XFFFF||lcddev.id==0X9300)
          //ID 不正确,新增 0X9300 判断，因为 9341 在未被复位的情况下会被读成 9300
          {
              //尝试 9341 ID 的读取
              LCD_WR_REG(0XD3);
              lcddev.id=LCD_RD_DATA();      //dummy read
              lcddev.id=LCD_RD_DATA();      //读到 0X00
              lcddev.id=LCD_RD_DATA();      //读取 93
              lcddev.id<<=8;
              lcddev.id|=LCD_RD_DATA();     //读取 41
```

```
        if(lcddev.id!=0X9341)                //非 9341,尝试是不是 6804
        {
            LCD_WR_REG(0XBF);
            lcddev.id=LCD_RD_DATA();//dummy read
            lcddev.id=LCD_RD_DATA();//读回 0X01
            lcddev.id=LCD_RD_DATA();//读回 0XD0
            lcddev.id=LCD_RD_DATA();//这里读回 0X68
            lcddev.id<<=8;
            lcddev.id|=LCD_RD_DATA();//这里读回 0X04
            if(lcddev.id!=0X6804)          //也不是 6804,尝试看看是不是 NT35310
            {
                LCD_WR_REG(0XD4);
                lcddev.id=LCD_RD_DATA();       //dummy read
                lcddev.id=LCD_RD_DATA();       //读回 0X01
                lcddev.id=LCD_RD_DATA();       //读回 0X53
                lcddev.id<<=8;
                lcddev.id|=LCD_RD_DATA();      //这里读回 0X10
                if(lcddev.id!=0X5310)   //也不是 NT35310,尝试看看是不是 NT35510
                {
                    LCD_WR_REG(0XDA00);
                    lcddev.id=LCD_RD_DATA();//读回 0X00
                    LCD_WR_REG(0XDB00);
                    lcddev.id=LCD_RD_DATA();//读回 0X80
                    lcddev.id<<=8;
                    LCD_WR_REG(0XDC00);
                    lcddev.id|=LCD_RD_DATA();//读回 0X00
                    if(lcddev.id==0x8000)lcddev.id=0x5510;
                    //NT35510 读回的 ID 是 8000H,为方便区分,我们强制设置为 5510
                }
            }
        }
    }
    if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510)
    {    //如果是这三个 IC,则设置 WR 时序为最快
        //重新配置写时序控制寄存器的时序
        FSMC_Bank1E->BWTR[6]&=~(0XF<<0); //地址建立时间(ADDSET)清零
        FSMC_Bank1E->BWTR[6]&=~(0XF<<8); //数据保存时间清零
        FSMC_Bank1E->BWTR[6]|=3<<0;          //地址建立时间为 3 个 HCLK =18ns
        FSMC_Bank1E->BWTR[6]|=2<<8;          // 数 据 保 存 时 间 为  6ns*3  个
HCLK=18ns
    }else if(lcddev.id==0X6804||lcddev.id==0XC505)//6804/C505 速度上不去,得降低
    {
        //重新配置写时序控制寄存器的时序
        FSMC_Bank1E->BWTR[6]&=~(0XF<<0); //地址建立时间(ADDSET)清零
```

```
        FSMC_Bank1E->BWTR[6]&=~(0XF<<8); //数据保存时间清零
        FSMC_Bank1E->BWTR[6]|=10<<0;     //地址建立时间为 10 个 HCLK =60ns
        FSMC_Bank1E->BWTR[6]|=12<<8;     // 数 据 保 存 时 间 为  6ns*13  个
HCLK=78ns
    }
    printf(" LCD ID:%x\r\n",lcddev.id); //打印 LCD ID
    if(lcddev.id==0X9341)            //9341 初始化
    {
        ……//9341 初始化代码
    }else if(lcddev.id==0xXXXX)      //其他 LCD 初始化代码
    {
        ……//其他 LCD 驱动 IC，初始化代码
    }
    LCD_Display_Dir(0);              //默认为竖屏显示
    LCD_LED=1;                       //点亮背光
    LCD_Clear(WHITE);
}
```

从初始化代码可以看出，LCD 初始化步骤为①~⑥在代码中标注：

① GPIO,FSMC 使能。

② GPIO 初始化：GPIO_Init()函数。

③ 设置引脚复用映射。

④ FSMC 初始化：FSMC_NORSRAMInit()函数。

⑤ FSMC 使能：FSMC_NORSRAMCmd()函数。

⑥ 不同的 LCD 驱动器的初始化代码。

该函数先对 FSMC 相关 IO 进行初始化，然后是 FSMC 的初始化，这个我们在前面都有介绍，最后根据读到的 LCD ID，对不同的驱动器执行不同的初始化代码，从上面的代码可以看出，这个初始化函数可以针对十多款不同的驱动 IC 执行初始化操作，这样大大提高了整个程序的通用性。大家在以后的学习中应该多使用这样的方式，以提高程序的通用性、兼容性。

特别注意：本函数使用了 printf 来打印 LCD ID，所以，如果你在主函数里面没有初始化串口，那么将导致程序死在 printf 里面!!如果不想用 printf，那么请注释掉它。

LCD 驱动相关的函数就给大家讲解到这里。接下来，我们看看主函数代码如下：

```
int main(void)
{
    u8 x=0;
    u8 lcd_id[12];                  //存放 LCD ID 字符串
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);  //初始化延时函数
    uart_init(115200);        //初始化串口波特率为 115200

    LED_Init();                         //初始化 LED
    LCD_Init();             //初始化 LCD FSMC 接口
    POINT_COLOR=RED;
    sprintf((char*)lcd_id,"LCD ID:%04X",lcddev.id);//将 LCD ID 打印到 lcd_id 数组。
```

```
        while(1)
        {
            switch(x)
            {
                case 0:LCD_Clear(WHITE);break;
                case 1:LCD_Clear(BLACK);break;
                case 2:LCD_Clear(BLUE);break;
                case 3:LCD_Clear(RED);break;
                case 4:LCD_Clear(MAGENTA);break;
                case 5:LCD_Clear(GREEN);break;
                case 6:LCD_Clear(CYAN);break;
                case 7:LCD_Clear(YELLOW);break;
                case 8:LCD_Clear(BRRED);break;
                case 9:LCD_Clear(GRAY);break;
                case 10:LCD_Clear(LGRAY);break;
                case 11:LCD_Clear(BROWN);break;
            }
            POINT_COLOR=RED;
            LCD_ShowString(30,40,210,24,24,"Explorer STM32F4");
            LCD_ShowString(30,70,200,16,16,"TFTLCD TEST");
            LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
            LCD_ShowString(30,110,200,16,16,lcd_id);        //显示 LCD ID

            LCD_ShowString(30,130,200,12,12,"2014/5/4");
            x++;
            if(x==12)x=0;
            LED0=!LED0;delay_ms(1000);
        }
    }
```

该部分代码将显示一些固定的字符，字体大小包括 24*12、16*8 和 12*6 等三种，同时显示 LCD 驱动 IC 的型号，然后不停的切换背景颜色，每 1s 切换一次。而 LED0 也会不停的闪烁，指示程序已经在运行了。其中我们用到一个 sprintf 的函数，该函数用法同 printf，只是 sprintf 把打印内容输出到指定的内存区间上，sprintf 的详细用法，请百度。

另外**特别注意**：uart_init 函数，不能去掉，因为在 LCD_Init 函数里面调用了 printf，所以一旦你去掉这个初始化，就会死机了!实际上，只要你的代码有用到 printf，就必须初始化串口，否则都会死机，即停在 usart.c 里面的 fputc 函数，出不来。

在编译通过之后，我们开始下载验证代码。

18.4 下载验证

将程序下载到探索者 STM32F4 开发板后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块的显示如图 18.4.1 所示：

图 18.4.1 TFTLCD 显示效果图

我们可以看到屏幕的背景是不停切换的，同时 DS0 不停的闪烁，证明我们的代码被正确的执行了，达到了我们预期的目的。

第三十三章 触摸屏实验

本章，我们将介绍如何使用 STM32F4 来驱动触摸屏，ALIENTEK 探索者 STM32F4 开发板本身并没有触摸屏控制器，但是它支持触摸屏，可以通过外接带触摸屏的 LCD 模块（比如 ALIENTEK TFTLCD 模块），来实现触摸屏控制。在本章中，我们将向大家介绍 STM32 控制 ALIENTKE TFTLCD 模块（包括电阻触摸与电容触摸），实现触摸屏驱动，最终实现一个手写板的功能。本章分为如下几个部分：

33.1 电阻与电容触摸屏简介

33.2 硬件设计

33.3 软件设计

33.4 下载验证

33.1 触摸屏简介

目前最常用的触摸屏有两种：电阻式触摸屏与电容式触摸屏。下面，我们来分别介绍。

33.1.1 电阻式触摸屏

在 Iphone 面世之前，几乎清一色的都是使用电阻式触摸屏，电阻式触摸屏利用压力感应进行触点检测控制，需要直接应力接触，通过检测电阻来定位触摸位置。

ALIENTEK 2.4/2.8/3.5 寸 TFTLCD 模块自带的触摸屏都属于电阻式触摸屏，下面简单介绍下电阻式触摸屏的原理。

电阻触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，它以一层玻璃或硬塑料平板作为基层，表面涂有一层透明氧化金属（透明的导电电阻）导电层，上面再盖有一层外表面硬化处理、光滑防擦的塑料层、它的内表面也涂有一层涂层、在他们之间有许多细小的（小于 1/1000 英寸）的透明隔离点把两层导电层隔开绝缘。当手指触摸屏幕时，两层导电层在触摸点位置就有了接触，电阻发生变化，在 X 和 Y 两个方向上产生信号，然后送触摸屏控制器。控制器侦测到这一接触并计算出（X，Y）的位置，再根据获得的位置模拟鼠标的方式运作。这就是电阻技术触摸屏的最基本的原理。

电阻触摸屏的优点：精度高、价格便宜、抗干扰能力强、稳定性好。

电阻触摸屏的缺点：容易被划伤、透光性不太好、不支持多点触摸。

从以上介绍可知，触摸屏都需要一个 AD 转换器， 一般来说是需要一个控制器的。ALIENTEK TFTLCD 模块选择的是四线电阻式触摸屏，这种触摸屏的控制芯片有很多，包括：ADS7843、ADS7846、TSC2046、XPT2046 和 AK4182 等。这几款芯片的驱动基本上是一样的，也就是你只要写出了 ADS7843 的驱动，这个驱动对其他几个芯片也是有效的。而且封装也有一样的，完全 PIN TO PIN 兼容。所以在替换起来，很方便。

ALIENTEK TFTLCD 模块自带的触摸屏控制芯片为 XPT2046。XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置， 除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。XPT2046 采用微小的封装形式：TSSOP-16,QFN-16(0.75mm 厚度)和 VFBGA－48。工作温度范围为-40℃～+85℃。

该芯片完全是兼容 ADS7843 和 ADS7846 的，关于这个芯片的详细使用，可以参考这两个芯片的 datasheet。

电阻式触摸屏就介绍到这里。

33.1.2 电容式触摸屏

现在几乎所有智能手机，包括平板电脑都是采用电容屏作为触摸屏，电容屏是利用人体感应进行触点检测控制，不需要直接接触或只需要轻微接触，通过检测感应电流来定位触摸坐标。

ALIENTEK 4.3/7 寸 TFTLCD 模块自带的触摸屏采用的是电容式触摸屏，下面简单介绍下电容式触摸屏的原理。

电容式触摸屏主要分为两种：

1、 表面电容式电容触摸屏。

表面电容式触摸屏技术是利用 ITO(铟锡氧化物，是一种透明的导电材料)导电膜，通过电场感应方式感测屏幕表面的触摸行为进行。但是表面电容式触摸屏有一些局限性，它只能识别一个手指或者一次触摸。

2、 投射式电容触摸屏。

投射电容式触摸屏是传感器利用触摸屏电极发射出静电场线。一般用于投射电容传感技术的电容类型有两种：自我电容和交互电容。

自我电容又称绝对电容，是最广为采用的一种方法，自我电容通常是指扫描电极与地构成的电容。在玻璃表面有用 ITO 制成的横向与纵向的扫描电极，这些电极和地之间就构成一个电容的两极。当用手或触摸笔触摸的时候就会并联一个电容到电路中去，从而使在该条扫描线上的总体的电容量有所改变。在扫描的时候，控制 IC 依次扫描纵向和横向电极，并根据扫描前后的电容变化来确定触摸点坐标位置。笔记本电脑触摸输入板就是采用的这种方式，笔记本电脑的输入板采用 X*Y 的传感电极阵列形成一个传感格子，当手指靠近触摸输入板时，在手指和传感电极之间产生一个小量电荷。采用特定的运算法则处理来自行、列传感器的信号来确定手指的位置。

交互电容又叫做跨越电容，它是在玻璃表面的横向和纵向的 ITO 电极的交叉处形成电容。交互电容的扫描方式就是扫描每个交叉处的电容变化，来判定触摸点的位置。当触摸的时候就会影响到相邻电极的耦合，从而改变交叉处的电容量，交互电容的扫面方法可以侦测到每个交叉点的电容值和触摸后电容变化，因而它需要的扫描时间与自我电容的扫描方式相比要长一些，需要扫描检测 X*Y 根电极。目前智能手机/平板电脑等的触摸屏，都是采用交互电容技术。

ALIENTEK 所选择的电容触摸屏，也是采用的是投射式电容屏（交互电容类型），所以后面仅以投射式电容屏作为介绍。

透射式电容触摸屏采用纵横两列电极组成感应矩阵，来感应触摸。以两个交叉的电极矩阵，即： X 轴电极和 Y 轴电极，来检测每一格感应单元的电容变化，如图 33.1.2.1 所示：

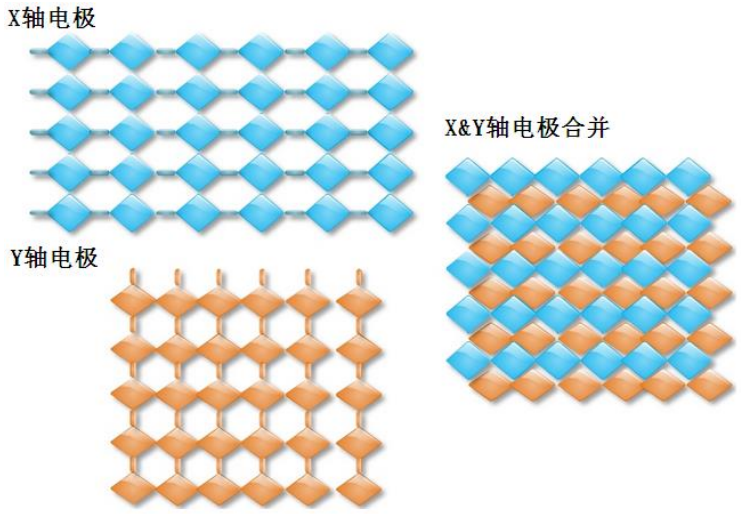

图 33.1.2.1 投射式电容屏电极矩阵示意图

示意图中的电极，实际是透明的，这里是为了方便大家理解。图中，X、Y 轴的透明电极电容屏的精度、分辨率与 X、Y 轴的通道数有关，通道数越多，精度越高。以上就是电容触摸屏的基本原理，接下来看看电容触摸屏的优缺点：

电容触摸屏的优点：手感好、无需校准、支持多点触摸、透光性好。

电容触摸屏的缺点：成本高、精度不高、抗干扰能力差。

这里特别提醒大家电容触摸屏对工作环境的要求是比较高的，在潮湿、多尘、高低温环境下面，都是不适合使用电容屏的。

电容触摸屏一般都需要一个驱动 IC 来检测电容触摸，且一般是通过 IIC 接口输出触摸数据的。ALIENTEK 7’ TFTLCD 模块的电容触摸屏，采用的是 15*10 的驱动结构（10 个感应通道，15 个驱动通道），采用的是 GT811/FT5206 做为驱动 IC。ALIENTEK 4.3’ TFTLCD 模块有两种成触摸屏：1，使用 OTT2001A 作为驱动 IC，采用 13*8 的驱动结构（8 个感应通道，13 个驱动通道）；2，使用 GT9147 作为驱动 IC，采用 17*10 的驱动结构（10 个感应通道，17 个驱动通道）。

这两个模块都只支持最多 5 点触摸，本例程支持 ALIENTEK 的 4.3 寸屏模块和新版的 7 寸屏模块（采用 SSD1963+FT5206 方案），电容触摸驱动 IC，这里只介绍 OTT2001A 和 GT9147，GT811/FT5206 的驱动方法同这两款 IC 是类似的，大家可以参考着学习即可。

OTT2001A 是台湾旭曜科技生产的一颗电容触摸屏驱动 IC，最多支持 208 个通道。支持 SPI/IIC 接口，在 ALIENTEK 4.3’ TFTLCD 电容触摸屏上，OTT2001A 只用了 104 个通道，采用 IIC 接口。IIC 接口模式下，该驱动 IC 与 STM32F4 的连接仅需要 4 根线：SDA、SCL、RST 和 INT，SDA 和 SCL 是 IIC 通信用的，RST 是复位脚（低电平有效），INT 是中断输出信号，关于 IIC 我们就不详细介绍了，请参考第二十九章。

OTT2001A 的器件地址为 0X59（不含最低位，换算成读写命令则是读：0XB3，写：0XB2），接下来，介绍一下 OTT2001A 的几个重要的寄存器。

1， 手势 ID 寄存器

手势 ID 寄存器（00H）用于告诉 MCU，哪些点有效，哪些点无效，从而读取对应的数据，该寄存器各位描述如表 33.1.2.1 所示：

手势 ID 寄存器（00H）				
位	BIT8	BIT6	BIT5	BIT4
说明	保留	保留	保留	0，(X1，Y1)无效 1，(X1，Y1)有效

位	BIT3	BIT2	BIT1	BIT0
说明	0,(X4,Y4)无效 1,(X4,Y4)有效	0,(X3,Y3)无效 1,(X3,Y3)有效	0,(X2,Y2)无效 1,(X2,Y2)有效	0,(X1,Y1)无效 1,(X1,Y1)有效

表 33.1.2.1 手势 ID 寄存器

OTT2001A 支持最多 5 点触摸，所以表中只有 5 个位用来表示对应点坐标是否有效，其余位为保留位（读为 0），通过读取该寄存器，我们可以知道哪些点有数据，哪些点无数据，如果读到的全是 0，则说明没有任何触摸。

2， 传感器控制寄存器（ODH）

传感器控制寄存器（ODH），该寄存器也是 8 位，仅最高位有效，其他位都是保留，当最高位为 1 的时候，打开传感器（开始检测），当最高位设置为 0 的时候，关闭传感器（停止检测）。

3， 坐标数据寄存器（共 20 个）

坐标数据寄存器总共有 20 个，每个坐标占用 4 个寄存器，坐标寄存器与坐标的对应关系如表 33.1.2.2 所示：

寄存器编号	01H	02H	03H	04H
坐标 1	X1[15:8]	X1[7:0]	Y1[15:8]	Y1[7:0]
寄存器编号	05H	06H	07H	08H
坐标 2	X2[15:8]	X2[7:0]	Y2[15:8]	Y2[7:0]
寄存器编号	10H	11H	12H	13H
坐标 3	X3[15:8]	X3[7:0]	Y3[15:8]	Y3[7:0]
寄存器编号	14H	15H	16H	17H
坐标 4	X4[15:8]	X4[7:0]	Y4[15:8]	Y4[7:0]
寄存器编号	18H	19H	1AH	1BH
坐标 5	X5[15:8]	X5[7:0]	Y5[15:8]	Y5[7:0]

表 33.1.2.2 坐标寄存器与坐标对应表

从表中可以看出，每个坐标的值，可以通过 4 个寄存器读出，比如读取坐标 1（X1，Y1），我们则可以读取 01H~04H，就可以知道当前坐标 1 的具体数值了，这里我们也可以只发送寄存器 01，然后连续读取 4 个字节，也可以正常读取坐标 1，寄存器地址会自动增加，从而提高读取速度。

OTT2001A 相关寄存器的介绍就介绍到这里，更详细的资料，请参考：OTT2001A IIC 协议指导.pdf 这个文档。OTT2001A 只需要经过简单的初始化就可以正常使用了，初始化流程：复位→延时 100ms→释放复位→设置传感器控制寄存器的最高位位 1，开启传感器检查。就可以正常使用了。

另外，OTT2001A 有两个地方需要特别注意一下：

1， OTT2001A 的寄存器是 8 位的，但是发送的时候要发送 16 位（高八位有效），才可以正常使用。

2， OTT2001A 的输出坐标，默认是以：X 坐标最大值是 2700，Y 坐标最大值是 1500 的分辨率输出的，也就是输出范围为：X：0~2700，Y：0~1500；MCU 在读取到坐标后，必须根据 LCD 分辨率做一个换算，才能得到真实的 LCD 坐标。

下面我们简单介绍下 GT9147，该芯片是深圳汇顶科技研发的一颗电容触摸屏驱动 IC，支持 100Hz 触点扫描频率，支持 5 点触摸，支持 18*10 个检测通道，适合小于 4.5 寸的电容触摸屏使用。

和 OTT2001A 一样，GT9147 与 MCU 连接也是通过 4 根线：SDA、SCL、RST 和 INT。不过，GT9147 的 IIC 地址，可以是 0X14 或者 0X5D，当复位结束后的 5ms 内，如果 INT

是高电平，则使用 0X14 作为地址，否则使用 0X5D 作为地址，具体的设置过程，请看：GT9147 数据手册.pdf 这个文档。本章我们使用 0X14 作为器件地址（不含最低位，换算成读写命令则是读：0X29，写：0X28），接下来，介绍一下 GT9147 的几个重要的寄存器。

1，控制命令寄存器（0X8040）

该寄存器可以写入不同值，实现不同的控制，我们一般使用 0 和 2 这两个值，写入 2，即可软复位 GT9147，在硬复位之后，一般要往该寄存器写 2，实行软复位。然后，写入 0，即可正常读取坐标数据（并且会结束软复位）。

2，配置寄存器组（0X8047~0X8100）

这里共 186 个寄存器，用于配置 GT9147 的各个参数，这些配置一般由厂家提供给我们（一个数组），所以我们只需要将厂家给我们的配置，写入到这些寄存器里面，即可完成 GT9147 的配置。由于 GT9147 可以保存配置信息（可写入内部 FLASH，从而不需要每次上电都更新配置），我们有几点注意的地方提醒大家：1，0X8047 寄存器用于指示配置文件版本号，程序写入的版本号，必须大于等于 GT9147 本地保存的版本号，才可以更新配置。2，0X80FF 寄存器用于存储校验和，使得 0X8047~0X80FF 之间所有数据之和为 0。3，0X8100 用于控制是否将配置保存在本地，写 0，则不保存配置，写 1 则保存配置。

3，产品 ID 寄存器（0X8140~0X8143）

这里总共由 4 个寄存器组成，用于保存产品 ID，对于 GT9147，这 4 个寄存器读出来就是：9，1，4，7 四个字符（ASCII 码格式）。因此，我们可以通过这 4 个寄存器的值，来判断驱动 IC 的型号，从而判断是 OTT2001A 还是 GT9147，以便执行不同的初始化。

4，状态寄存器（0X814E）

该寄存器各位描述如表 33.1.2.3 所示：

寄存器	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0X814E	buffer 状态	大点	接近有效	按键	有效触点个数			

表 33.1.2.3 状态寄存器各位描述

这里，我们仅关心最高位和最低 4 位，最高位用于表示 buffer 状态，如果有数据（坐标/按键），buffer 就会是 1，最低 4 位用于表示有效触点的个数，范围是：0~5，0，表示没有触摸，5 表示有 5 点触摸。这和前面 OTT2001A 的表示方法稍微有点区别，OTT2001A 是每个位表示一个触点，这里是有多少有效触点值就是多少。最后，该寄存器在每次读取后，如果 bit7 有效，则必须写 0，清除这个位，否则不会输出下一次数据!! 这个要特别注意!!!

5，坐标数据寄存器（共 30 个）

这里共分成 5 组（5 个点），每组 6 个寄存器存储数据，以触点 1 的坐标数据寄存器组为例，如表 33.1.2.4 所示：

寄存器	bit7~0	寄存器	bit7~0
0X8150	触点 1 x 坐标低 8 位	0X8151	触点 1 x 坐标低高位
0X8152	触点 1 y 坐标低 8 位	0X8153	触点 1 y 坐标低高位
0X8154	触点 1 触摸尺寸低 8 位	0X8155	触点 1 触摸尺寸高 8 位

表 33.1.2.4 触点 1 坐标寄存器组描述

我们一般只用到触点的 x，y 坐标，所以只需要读取 0X8150~0X8153 的数据，组合即可得到触点坐标。其他 4 组分别是：0X8158、0X8160、0X8168 和 0X8170 等开头的 16 个寄存器组成，分别针对触点 2~4 的坐标。同样 GT9147 也支持寄存器地址自增，我们只需要发送寄存器组的首地址，然后连续读取即可，GT9147 会自动地址自增，从而提高读取速度。

GT9147 相关寄存器的介绍就介绍到这里，更详细的资料，请参考：GT9147 编程指南.pdf

这个文档。

GT9147 只需要经过简单的初始化就可以正常使用了，初始化流程：硬复位→延时 10ms→结束硬复位→设置 IIC 地址→延时 100ms→软复位→更新配置（需要时）→结束软复位。此时 GT9147 即可正常使用了。

然后，我们不停的查询 0X814E 寄存器，判断是否有有效触点，如果有，则读取坐标数据寄存器，得到触点坐标，特别注意，如果 0X814E 读到的值最高位为 1，就必须对该位写 0，否则无法读到下一次坐标数据。

电容式触摸屏部分，就介绍到这里。

33.2 硬件设计

本章实验功能简介：开机的时候先初始化 LCD，读取 LCD ID，随后，根据 LCD ID 判断是电阻触摸屏还是电容触摸屏，如果是电阻触摸屏，则先读取 24C02 的数据判断触摸屏是否已经校准过，如果没有校准，则执行校准程序，校准过后再进入电阻触摸屏测试程序，如果已经校准了，就直接进入电阻触摸屏测试程序。

如果是 4.3 寸电容触摸屏，则先读取芯片 ID，判断是不是 GT9147，如果是则执行 GT9147 的初始化代码，如果不是，则执行 OTT2001A 的初始化代码；如果是 7 寸电容触摸屏（仅支持新款 7 寸屏，使用 SSD1963+FT5206 方案），则执行 FT5206 的初始化代码，在初始化电容触摸屏完成后，进入电容触摸屏测试程序（电容触摸屏无需校准！！）。

电阻触摸屏测试程序和电容触摸屏测试程序基本一样，只是电容触摸屏支持最多 5 点同时触摸，电阻触摸屏只支持一点触摸，其他一模一样。测试界面的右上角会有一个清空的操作区域（RST），点击这个地方就会将输入全部清除，恢复白板状态。使用电阻触摸屏的时候，可以通过按 KEY0 来实现强制触摸屏校准，只要按下 KEY0 就会进入强制校准程序。

所要用到的硬件资源如下：

1） 指示灯 DS0

2） KEY0 按键

3） TFTLCD 模块（带电阻/电容式触摸屏）

4） 24C02

所有这些资源与 STM32F4 的连接图，在前面都已经介绍了，这里我们只针对 TFTLCD 模块与 STM32F4 的连接端口再说明一下，TFTLCD 模块的触摸屏（电阻触摸屏）总共有 5 跟线与 STM32F4 连接，连接电路图如图 33.2.1 所示：

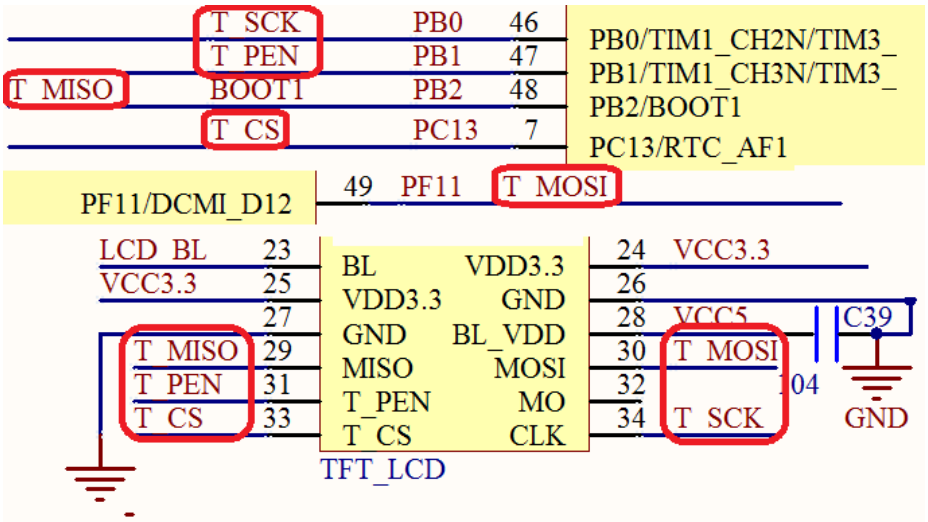

图 33.2.1 触摸屏与 STM32F4 的连接图

从图中可以看出，T_MOSI、T_MISO、T_SCK、T_CS 和 T_PEN 分别连接在 STM32F4 的：PF11、PB2、PB0、PC13 和 PB1 上。

如果是电容式触摸屏，我们的接口和电阻式触摸屏一样（上图右侧接口），只是没有用到五根线了，而是四根线，分别是：T_PEN(CT_INT)、T_CS(CT_RST)、T_CLK(CT_SCL)和 T_MOSI(CT_SDA)。其中：CT_INT、CT_RST、CT_SCL 和 CT_SDA 分别是 OTT2001A/GT9147/FT5206 的：中断输出信号、复位信号，IIC 的 SCL 和 SDA 信号。这里，我们用查询的方式读取 OTT2001A/GT9147/FT5206 的数据，对于 OTT2001A/FT5206 没有用到中断信号（CT_INT），所以同 STM32F4 的连接，只需要 3 根线即可，不过 GT9147 还需要用到 CT_INT 做 IIC 地址设定，所以需要 4 根线连接。

33.3 软件设计

打开本章实验工程目录可以看到，我们在 HARDWARE 文件夹下新建了一个 TOUCH 文件夹，然后新建了 touch.c、touch.h、ctiic.c、ctiic.h、ott2001a.c、ott2001a.h、gt9147.c、gt9147.h、ft5206.c 和 ft5206.h 等十个文件用来存放触摸屏相关的代码。同时引入这些源文件到工程 HARDWARE 分组之下，并将 TOUCH 文件夹加入头文件包含路径。其中，touch.c 和 touch.h 是电阻触摸屏部分的代码，顺带兼电容触摸屏的管理控制，其他则是电容触摸屏部分的代码。

打开 touch.c 文件，里面主要是与触摸屏相关的代码（主要是电阻触摸屏的代码），这里我们也不全部贴出来了，仅介绍几个重要的函数。

首先我们要介绍的是 TP_Read_XY2 这个函数，该函数专门用于从电阻式触摸屏控制 IC 读取坐标的值（0~4095），TP_Read_XY2 的代码如下：

```
//连续 2 次读取触摸屏 IC,且这两次的偏差不能超过
//ERR_RANGE,满足条件,则认为读数正确,否则读数错误.
//该函数能大大提高准确度
//x,y:读取到的坐标值
//返回值:0,失败;1,成功。
#define ERR_RANGE 50 //误差范围
u8 TP_Read_XY2(u16 *x,u16 *y)
{
    u16 x1,y1;
    u16 x2,y2;
    u8 flag;
    flag=TP_Read_XY(&x1,&y1);
    if(flag==0)return(0);
    flag=TP_Read_XY(&x2,&y2);
    if(flag==0)return(0);
    //前后两次采样在+-50 内
    if(((x2<=x1&&x1<x2+ERR_RANGE)||(x1<=x2&&x2<x1+ERR_RANGE))
    &&((y2<=y1&&y1<y2+ERR_RANGE)||(y1<=y2&&y2<y1+ERR_RANGE)))
    {
        *x=(x1+x2)/2;*y=(y1+y2)/2;
        return 1;
    }else return 0;
}
```

该函数采用了一个非常好的办法来读取屏幕坐标值，就是连续读两次，两次读取的值之

差不能超过一个特定的值（ERR_RANGE），通过这种方式，我们可以大大提高触摸屏的准确度。另外该函数调用的 TP_Read_XY 函数，用于单次读取坐标值。TP_Read_XY 也采用了一些软件滤波算法，具体见光盘的源码。接下来，我们介绍另外一个函数 TP_Adjust，该函数源码如下：

```
//触摸屏校准代码
//得到四个校准参数
void TP_Adjust(void)
{
    u16 pos_temp[4][2];//坐标缓存值
    u8  cnt=0; u32 tem1,tem2;
    u16 d1,d2; u16 outtime=0;
    double fac;
    POINT_COLOR=BLUE;
    BACK_COLOR =WHITE;
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=RED;//红色
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=BLACK;
    LCD_ShowString(40,40,160,100,16,(u8*)TP_REMIND_MSG_TBL);//显示提示信息
    TP_Drow_Touch_Point(20,20,RED);//画点 1
    tp_dev.sta=0;//消除触发信号
    tp_dev.xfac=0;//xfac 用来标记是否校准过,所以校准之前必须清掉!以免错误
    while(1)//如果连续 10 秒钟没有按下,则自动退出
    {
        tp_dev.scan(1);                                 //扫描物理坐标
        if((tp_dev.sta&0xc0)==TP_CATH_PRES)   //按键按下了一次(此时按键松开了.)
        {
            outtime=0;
            tp_dev.sta&=~(1<<6);//标记按键已经被处理过了.

            pos_temp[cnt][0]=tp_dev.x;
            pos_temp[cnt][1]=tp_dev.y;
            cnt++;
            switch(cnt)
            {
                case 1:
                    TP_Drow_Touch_Point(20,20,WHITE);                 //清除点 1
                    TP_Drow_Touch_Point(lcddev.width-20,20,RED);      //画点 2
                    break;
                case 2:
                    TP_Drow_Touch_Point(lcddev.width-20,20,WHITE); //清除点 2
                    TP_Drow_Touch_Point(20,lcddev.height-20,RED);     //画点 3
                    break;
                case 3:
```

```
                    TP_Drow_Touch_Point(20,lcddev.height-20,WHITE);//清除点 3
                    TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,RED);
                    //画点 4
                    break;
                case 4://全部四个点已经得到
                    //对边相等
                    tem1=abs(pos_temp[0][0]-pos_temp[1][0]);//x1-x2
                    tem2=abs(pos_temp[0][1]-pos_temp[1][1]);//y1-y2
                    tem1*=tem1;
                    tem2*=tem2;
                    d1=sqrt(tem1+tem2);//得到 1,2 的距离
                    tem1=abs(pos_temp[2][0]-pos_temp[3][0]);//x3-x4
                    tem2=abs(pos_temp[2][1]-pos_temp[3][1]);//y3-y4
                    tem1*=tem1;tem2*=tem2;
                    d2=sqrt(tem1+tem2);//得到 3,4 的距离
                    fac=(float)d1/d2;
                    if(fac<0.95||fac>1.05||d1==0||d2==0)//不合格
                    {
                        cnt=0;

TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,WHITE);
    //清除点 4
                        TP_Drow_Touch_Point(20,20,RED); //画点 1
                        TP_Adj_Info_Show(pos_temp[0][0],pos_temp[0][1],pos_temp
[1]
                        [0],pos_temp[1][1],pos_temp[2][0],pos_temp[2][1],pos_temp[3
]
                        [0],pos_temp[3][1],fac*100);//显示数据
                        continue;
                    }
                    tem1=abs(pos_temp[0][0]-pos_temp[2][0]);//x1-x3
                    tem2=abs(pos_temp[0][1]-pos_temp[2][1]);//y1-y3
                    tem1*=tem1;tem2*=tem2;
                    d1=sqrt(tem1+tem2);//得到 1,3 的距离
                    tem1=abs(pos_temp[1][0]-pos_temp[3][0]);//x2-x4
                    tem2=abs(pos_temp[1][1]-pos_temp[3][1]);//y2-y4
                    tem1*=tem1;tem2*=tem2;
                    d2=sqrt(tem1+tem2);//得到 2,4 的距离
                    fac=(float)d1/d2;
                    if(fac<0.95||fac>1.05)//不合格
                    {
                        cnt=0;
                        TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,
                        WHITE); //清除点 4
```

```
                                TP_Drow_Touch_Point(20,20,RED); //画点 1

TP_Adj_Info_Show(pos_temp[0][0],pos_temp[0][1],pos_temp[1]
                                [0],pos_temp[1][1],pos_temp[2][0],pos_temp[2][1],pos_temp[3
]
                                [0],pos_temp[3][1],fac*100);//显示数据
                                continue;
                            }//正确了
                            //对角线相等
                            tem1=abs(pos_temp[1][0]-pos_temp[2][0]);//x1-x3
                            tem2=abs(pos_temp[1][1]-pos_temp[2][1]);//y1-y3
                            tem1*=tem1;tem2*=tem2;
                            d1=sqrt(tem1+tem2);//得到 1,4 的距离
                            tem1=abs(pos_temp[0][0]-pos_temp[3][0]);//x2-x4
                            tem2=abs(pos_temp[0][1]-pos_temp[3][1]);//y2-y4
                            tem1*=tem1;tem2*=tem2;
                            d2=sqrt(tem1+tem2);//得到 2,3 的距离
                            fac=(float)d1/d2;
                            if(fac<0.95||fac>1.05)//不合格
                            {
                                cnt=0;
                                TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,
                                WHITE); //清除点 4
                                TP_Drow_Touch_Point(20,20,RED);//画点 1

    TP_Adj_Info_Show(pos_temp[0][0],pos_temp[0][1],pos_temp[1]
                                [0],pos_temp[1][1],pos_temp[2][0],pos_temp[2][1],pos_temp[3
]
                                [0],pos_temp[3][1],fac*100);//显示数据
                                continue;
                            }//正确了
                            //计算结果

tp_dev.xfac=(float)(lcddev.width-40)/(pos_temp[1][0]-pos_temp[0][0]);
                            //得到 xfac
                            tp_dev.xoff=(lcddev.width-tp_dev.xfac*(pos_temp[1][0]+pos_temp[
0]
                            [0]))/2;//得到 xoff

tp_dev.yfac=(float)(lcddev.height-40)/(pos_temp[2][1]-pos_temp[0][1]
                            );//得到 yfac

tp_dev.yoff=(lcddev.height-tp_dev.yfac*(pos_temp[2][1]+pos_temp[0]
                            [1]))/2;//得到 yoff
```

```
                        if(abs(tp_dev.xfac)>2||abs(tp_dev.yfac)>2)//触屏和预设的相反了.
                        {
                            cnt=0;
                            TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,WHI
TE
                            );//清除点 4
                            TP_Drow_Touch_Point(20,20,RED); //画点 1

    LCD_ShowString(40,26,lcddev.width,lcddev.height,16,"TP Need
                              readjust!");
                            tp_dev.touchtype=!tp_dev.touchtype;//修改触屏类型.
                            if(tp_dev.touchtype)//X,Y 方向与屏幕相反
                            {CMD_RDX=0X90; CMD_RDY=0XD0;}

                            else {CMD_RDX=0XD0;CMD_RDY=0X90;}
                            //X,Y 方向与屏幕相同
                            continue;
                        }
                        POINT_COLOR=BLUE;
                        LCD_Clear(WHITE);//清屏
                        LCD_ShowString(35,110,lcddev.width,lcddev.height,16,"Touch
Screen
                        Adjust OK!");//校正完成
                        delay_ms(1000);
                        TP_Save_Adjdata();
                        LCD_Clear(WHITE);//清屏
                        return;//校正完成
                    }
                }
                delay_ms(10); outtime++;
                if(outtime>1000) { TP_Get_Adjdata();break; }
            }
    }
```

TP_Adjust 是此部分最核心的代码，在这里，给大家介绍一下我们这里所使用的触摸屏校正原理：我们传统的鼠标是一种相对定位系统，只和前一次鼠标的位置坐标有关。而触摸屏则是一种绝对坐标系统，要选哪就直接点哪，与相对定位系统有着本质的区别。绝对坐标系统的特点是每一次定位坐标与上一次定位坐标没有关系，每次触摸的数据通过校准转为屏幕上的坐标，不管在什么情况下，触摸屏这套坐标在同一点的输出数据是稳定的。不过由于技术原理的原因，并不能保证同一点触摸每一次采样数据相同，不能保证绝对坐标定位，点不准，这就是触摸屏最怕出现的问题：漂移。对于性能质量好的触摸屏来说，漂移的情况出现并不是很严重。所以很多应用触摸屏的系统启动后，进入应用程序前，先要执行校准程序。通常应用程序中使用的 LCD 坐标是以像素为单位的。比如说：左上角的坐标是一组非 0 的数值，比如（20，20），而右下角的坐标为（220，300）。这些点的坐标都是以像素为单位的，而从触摸屏中读出的是点的物理坐标，其坐标轴的方向、XY 值的比例因子、偏移量都与

LCD 坐标不同，所以，需要在程序中把物理坐标首先转换为像素坐标，然后再赋给 POS 结构，达到坐标转换的目的。

校正思路：在了解了校正原理之后，我们可以得出下面的一个从物理坐标到像素坐标的转换关系式：

LCDx=xfac*Px+xoff；

LCDy=yfac*Py+yoff；

其中(LCDx,LCDy)是在 LCD 上的像素坐标，(Px,Py)是从触摸屏读到的物理坐标。xfac，yfac 分别是 X 轴方向和 Y 轴方向的比例因子，而 xoff 和 yoff 则是这两个方向的偏移量。

这样我们只要事先在屏幕上面显示 4 个点（这四个点的坐标是已知的），分别按这四个点就可以从触摸屏读到 4 个物理坐标，这样就可以通过待定系数法求出 xfac、yfac、xoff、yoff 这四个参数。我们保存好这四个参数，在以后的使用中，我们把所有得到的物理坐标都按照这个关系式来计算，得到的就是准确的屏幕坐标。达到了触摸屏校准的目的。

TP_Adjust 就是根据上面的原理设计的校准函数，注意该函数里面多次使用了 lcddev.width 和 lcddev.height，用于坐标设置，主要是为了兼容不同尺寸的 LCD（比如 320*240、480*320 和 800*480 的屏都可以兼容）。

接下来看看触摸屏初始化函数：TP_Init，该函数根据 LCD 的 ID（即 lcddev.id）判别是电阻屏还是电容屏，执行不同的初始化，该函数代码如下：

```
//触摸屏初始化
//返回值:0,没有进行校准  1,进行过校准
u8 TP_Init(void)
{
    if(lcddev.id==0X5510)       //电容触摸屏
    {
        if(GT9147_Init()==0)    //是 GT9147?
        {
            tp_dev.scan=GT9147_Scan;  //扫描函数指向 GT9147 触摸屏扫描
        }else
        {
            OTT2001A_Init();
            tp_dev.scan=OTT2001A_Scan;//扫描函数指向 OTT2001A 触摸屏扫描
        }
        tp_dev.touchtype|=0X80;      //电容屏
        tp_dev.touchtype|=lcddev.dir&0X01;//横屏还是竖屏
        return 0;
    } else if(lcddev.id==0X1963)
    {
        FT5206_Init();
        tp_dev.scan=FT5206_Scan;        //扫描函数指向 GT9147 触摸屏扫描
        tp_dev.touchtype|=0X80;                 //电容屏
        tp_dev.touchtype|=lcddev.dir&0X01;//横屏还是竖屏
        return 0;
    } else
    {
```

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC|
                       RCC_AHB1Periph_GPIOF, ENABLE);//使能 GPIOB,C,F 时钟
            //GPIOB1,2 初始化设置
            GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;//PB1/2 设置为上拉
输入
            GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//输入模式
            GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
            GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
            GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
            GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

            GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;//PB0 设置为推挽输出
            GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
            GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

            GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;//PC13 设置为推挽输出
            GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
            GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化

            GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;//PF11 设置推挽输出
            GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
            GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化
            TP_Read_XY(&tp_dev.x[0],&tp_dev.y[0]);//第一次读取初始化
            AT24CXX_Init();         //初始化 24CXX
            if(TP_Get_Adjdata())return 0;//已经校准
            else                    //未校准?
            {
                LCD_Clear(WHITE);//清屏
                TP_Adjust();        //屏幕校准
                TP_Save_Adjdata();
            }
            TP_Get_Adjdata();
        }
        return 1;
    }
```

该函数比较简单，重点说一下：tp_dev.scan，这个结构体函数指针，默认是指向 TP_Scan 的，如果是电阻屏则用默认的即可，如果是电容屏，则指向新的扫描函数 GT9147_Scan、OTT2001A_Scan 或 FT5206_Scan（根据芯片 ID 判断到底指向那个），执行电容触摸屏的扫描函数，这几个函数在后续会介绍。

其他的函数我们这里就不多介绍了，接下来打开 touch.h 文件，代码如下：

```
#define TP_PRES_DOWN    0x80    //触屏被按下
#define TP_CATH_PRES    0x40    //有按键按下了
#define CT_MAX_TOUCH    5       //电容屏支持的点数,固定为 5 点
//触摸屏控制器
```

```
typedef struct
{
    u8 (*init)(void);                   //初始化触摸屏控制器
    u8 (*scan)(u8);                     //扫描触摸屏.0,屏幕扫描;1,物理坐标;
    void (*adjust)(void);               //触摸屏校准
    u16 x[CT_MAX_TOUCH];                //当前坐标
    u16 y[CT_MAX_TOUCH];                //电容屏有最多 5 组坐标,电阻屏则用 x[0],y[0]代表: 此次
                                        //扫描时触屏的坐标,用 x[4],y[4]存储第一次按下时的坐标.
    u8    sta;                          //笔的状态
                                        //b7:按下 1/松开 0;
                                        //b6:0,没有按键按下;1,有按键按下.
                                        //b5:保留
                                        //b4~b0:电容触摸屏按下的点数(0,表示未按下,1 表示按下)
//////////////////////触摸屏校准参数(电容屏不需要校准)//////////////////////
    float xfac;
    float yfac;
    short xoff;
    short yoff;
//新增的参数,当触摸屏的左右上下完全颠倒时需要用到.
//b0:0,竖屏(适合左右为 X 坐标,上下为 Y 坐标的 TP)
//     1,横屏(适合左右为 Y 坐标,上下为 X 坐标的 TP)
//b1~6:保留.
//b7:0,电阻屏
//     1,电容屏
    u8 touchtype;
}_m_tp_dev;
extern _m_tp_dev tp_dev;            //触屏控制器在 touch.c 里面定义
//电阻屏芯片连接引脚
#define PEN             PBin(1)     //T_PEN
#define DOUT            PBin(2)     //T_MISO
#define TDIN            PFout(11)   //T_MOSI
#define TCLK            PBout(0)    //T_SCK
#define TCS             PCout(13)   //T_CS
//电阻屏函数
void TP_Write_Byte(u8 num);                     //向控制芯片写入一个数据
u16 TP_Read_AD(u8 CMD);                         //读取 AD 转换值
u16 TP_Read_XOY(u8 xy);                         //带滤波的坐标读取(X/Y)
……(//省略部分代码)
u8 TP_Scan(u8 tp);                              //扫描
u8 TP_Init(void);                               //初始化
#endif
```

上述代码，我们重点看看_m_tp_dev 结构体，改结构体用于管理和记录触摸屏（包括电阻触摸屏与电容触摸屏）相关信息。通过结构体，在使用的时候，我们一般直接调用 tp_dev 的相关成员函数/变量屏即可达到需要的效果，这种设计简化了接口，且方便管理和维护，大家可以效仿一下。

ctiic.c 和 ctiic.h 是电容触摸屏的 IIC 接口部分代码，与第二十九章的 myiic.c 和 myiic.h 基本一样，这里就不单独介绍了。接下来看看文件 ott2001a.c 代码如下：

```
//向 OTT2001A 写入一次数据
//reg:起始寄存器地址
//buf:数据缓缓存区
//len:写数据长度
//返回值:0,成功;1,失败.
u8 OTT2001A_WR_Reg(u16 reg,u8 *buf,u8 len)
{
    u8 i; u8 ret=0;
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_WR);CT_IIC_Wait_Ack();//发送写命令
    CT_IIC_Send_Byte(reg>>8); CT_IIC_Wait_Ack();        //发送高 8 位地址
    CT_IIC_Send_Byte(reg&0XFF); CT_IIC_Wait_Ack();      //发送低 8 位地址
    for(i=0;i<len;i++)
    {
        CT_IIC_Send_Byte(buf[i]); ret=CT_IIC_Wait_Ack(); //发数据
        if(ret)break;
    }
    CT_IIC_Stop();                          //产生一个停止条件
    return ret;
}
//从 OTT2001A 读出一次数据
//reg:起始寄存器地址
//buf:数据缓缓存区
//len:读数据长度
void OTT2001A_RD_Reg(u16 reg,u8 *buf,u8 len)
{
    u8 i;
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_WR); CT_IIC_Wait_Ack();//发送写命令
    CT_IIC_Send_Byte(reg>>8); CT_IIC_Wait_Ack();        //发送高 8 位地址
    CT_IIC_Send_Byte(reg&0XFF); CT_IIC_Wait_Ack();      //发送低 8 位地址
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_RD); CT_IIC_Wait_Ack();//发送读命令
    for(i=0;i<len;i++) buf[i]=CT_IIC_Read_Byte(i==(len-1)?0:1); //发数据
    CT_IIC_Stop();//产生一个停止条件
}
//传感器打开/关闭操作
//cmd:1,打开传感器;0,关闭传感器
```

```
void OTT2001A_SensorControl(u8 cmd)
{
    u8 regval=0X00;
    if(cmd)regval=0X80;
    OTT2001A_WR_Reg(OTT_CTRL_REG,&regval,1);
}
//初始化触摸屏
//返回值:0,初始化成功;1,初始化失败
u8 OTT2001A_Init(void)
{
    u8 regval=0;
    GPIO_InitTypeDef    GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC,
                                                ENABLE);//使能 GPIOB,C 时钟
    //GPIOB1 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;//PB1 设置为上拉输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//输入模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;//PC13 设置为推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
    GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化

    CT_IIC_Init();          //初始化电容屏的 I2C 总线
    OTT_RST=0;              //复位
    delay_ms(100);
    OTT_RST=1;              //释放复位
    delay_ms(100);
    OTT2001A_SensorControl(1);//打开传感器
    OTT2001A_RD_Reg(OTT_CTRL_REG,&regval,1);//读取传感器运行寄存器的值来
判断
                                                //I2C 通信是否正常
    printf("CTP ID:%x\r\n",regval);
    if(regval==0x80)return 0;
    return 1;
}
const                                                   u16
OTT_TPX_TBL[5]={OTT_TP1_REG,OTT_TP2_REG,OTT_TP3_REG,OTT_TP4
                                    _REG,OTT_TP5_REG};
//扫描触摸屏(采用查询方式)
//mode:0,正常扫描.
```

```
//返回值:当前触屏状态.
//0,触屏无触摸;1,触屏有触摸
u8 OTT2001A_Scan(u8 mode)
{
    u8 buf[4], i=0, res=0;
    static u8 t=0;//控制查询间隔,从而降低 CPU 占用率
    t++;
    if((t%10)==0||t<10)//空闲时,每进入 10 次,才检测 1 次,从而节省 CPU 使用率
    {
        OTT2001A_RD_Reg(OTT_GSTID_REG,&mode,1);//读取触摸点的状态
        if(mode&0X1F)
        {
            tp_dev.sta=(mode&0X1F)|TP_PRES_DOWN|TP_CATH_PRES;
            for(i=0;i<5;i++)
            {
                if(tp_dev.sta&(1<<i))    //触摸有效?
                {
                    OTT2001A_RD_Reg(OTT_TPX_TBL[i],buf,4); //读取 XY 坐标值
                    if(tp_dev.touchtype&0X01)//横屏
                    {
                        tp_dev.y[i]=(((u16)buf[2]<<8)+buf[3])*OTT_SCAL_Y;
                        tp_dev.x[i]=800-((((u16)buf[0]<<8)+buf[1])*OTT_SCAL_X);
                    }else
                    {
                        tp_dev.x[i]=(((u16)buf[2]<<8)+buf[3])*OTT_SCAL_Y;
                        tp_dev.y[i]=(((u16)buf[0]<<8)+buf[1])*OTT_SCAL_X;
                    }
                    //printf("x[%d]:%d,y[%d]:%d\r\n",i,tp_dev.x[i],i,tp_dev.y[i]);
                }
            }
            res=1;
            if(tp_dev.x[0]==0 && tp_dev.y[0]==0)mode=0;  //数据全 0,则忽略此次数据
            t=0;        //触发一次,则会最少连续监测 10 次,从而提高命中率
        }
    }
    if((mode&0X1F)==0)//无触摸点按下
    {
        if(tp_dev.sta&TP_PRES_DOWN) tp_dev.sta&=~(1<<7);//之前是按下，标记松开
        else //之前就没有被按下
        {
            tp_dev.x[0]=0xffff;    tp_dev.y[0]=0xffff;
            tp_dev.sta&=0XE0;//清除点有效标记
        }
    }
```

```
        if(t>240)t=10;//重新从 10 开始计数
        return res;
    }
```

此部分总共 5 个函数，其中 OTT2001A_WR_Reg 和 OTT2001A_RD_Reg 分别用于读写 OTT2001A 芯片，这里特别注意寄存器地址是 16 位的，与 OTT2001A 手册介绍的是有出入的，必须 16 位才能正常操作。另外，重点介绍下 OTT2001A_Scan 函数，OTT2001A_Scan 函数用于扫描电容触摸屏是否有按键按下，由于我们不是用的中断方式来读取 OTT2001A 的数据的，而是采用查询的方式，所以这里使用了一个静态变量来提高效率，当无触摸的时候，尽量减少对 CPU 的占用，当有触摸的时候，又保证能迅速检测到。至于对 OTT2001A 数据的读取，则完全是我们在上面介绍的方法，先读取手势 ID 寄存器（OTT_GSTID_REG），判断是不是有有效数据，如果有，则读取，否则直接忽略，继续后面的处理。

其他的函数我们这里就不多介绍了，接下来看下 gt9147.c 里面的代码，这里我们仅介绍 GT9147_Init 和 GT9147_Scan 两个函数，代码如下：

```
//初始化 GT9147 触摸屏
//返回值:0,初始化成功;1,初始化失败
u8 GT9147_Init(void)
{
    u8 temp[5];
    GPIO_InitTypeDef    GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC,
                                                    ENABLE);//使能 GPIOB,C 时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 ;//PB1 设置为上拉输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//输入模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;//PC13 设置为推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
    GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化
    CT_IIC_Init();                  //初始化电容屏的 I2C 总线
    GT_RST=0; delay_ms(10);   //复位
    GT_RST=1; delay_ms(10);   //释放复位
    GPIO_Set(GPIOB,PIN1,GPIO_MODE_IN,0,0,GPIO_PUPD_NONE);//PB1 浮空输入
    delay_ms(100);
    GT9147_RD_Reg(GT_PID_REG,temp,4);//读取产品 ID
    temp[4]=0;
    printf("CTP ID:%s\r\n",temp);           //打印 ID
    if(strcmp((char*)temp,"9147")==0)      //ID==9147
    {
        temp[0]=0X02;
        GT9147_WR_Reg(GT_CTRL_REG,temp,1);//软复位 GT9147
        GT9147_RD_Reg(GT_CFGS_REG,temp,1);//读取 GT_CFGS_REG 寄存器
```

```
            if(temp[0]<0X60)//默认版本比较低,需要更新 flash 配置
            {
                printf("Default Ver:%d\r\n",temp[0]);
                GT9147_Send_Cfg(1);//更新并保存配置
            }
            delay_ms(10);
            temp[0]=0X00;
            GT9147_WR_Reg(GT_CTRL_REG,temp,1);//结束复位
            return 0;
        }
        return 1;
    }
    const                                                                    u16
GT9147_TPX_TBL[5]={GT_TP1_REG,GT_TP2_REG,GT_TP3_REG,GT_TP4_REG,
                                        GT_TP5_REG};
    //扫描触摸屏(采用查询方式)
    //mode:0,正常扫描.
    //返回值:当前触屏状态.
    //0,触屏无触摸;1,触屏有触摸
    u8 GT9147_Scan(u8 mode)
    {
        u8 buf[4]; u8 i=0; u8 res=0; u8 temp;
        static u8 t=0;//控制查询间隔,从而降低 CPU 占用率
        t++;
        if((t%10)==0||t<10)//空闲时,每进入 10 次,函数才检测 1 次,从而节省 CPU 使用率
        {
            GT9147_RD_Reg(GT_GSTID_REG,&mode,1);//读取触摸点的状态
            if((mode&0XF)&&((mode&0XF)<6))
            {
                temp=0XFF<<(mode&0XF);//将点的个数转换为 1 的位数,匹配 tp_dev.sta
定义
                tp_dev.sta=(~temp)|TP_PRES_DOWN|TP_CATH_PRES;
                for(i=0;i<5;i++)
                {
                    if(tp_dev.sta&(1<<i))    //触摸有效?
                    {
                        GT9147_RD_Reg(GT9147_TPX_TBL[i],buf,4); //读取 XY 坐标值
                        if(tp_dev.touchtype&0X01)//横屏
                        {
                            tp_dev.y[i]=((u16)buf[1]<<8)+buf[0];
                            tp_dev.x[i]=800-(((u16)buf[3]<<8)+buf[2]);
                        }else
                        {
                            tp_dev.x[i]=((u16)buf[1]<<8)+buf[0];
```

```
                                tp_dev.y[i]=((u16)buf[3]<<8)+buf[2];
                            }
                        }
                    }
                    res=1;
                    if(tp_dev.x[0]==0 && tp_dev.y[0]==0)mode=0;  //数据全 0,则忽略此次数据
                    t=0;        //触发一次,则会最少连续监测 10 次,从而提高命中率
                }
                if(mode&0X80&&((mode&0XF)<6)) //清标志?
                { temp=0; GT9147_WR_Reg(GT_GSTID_REG,&temp,1);}
            }
            if((mode&0X8F)==0X80)//无触摸点按下
            {
                if(tp_dev.sta&TP_PRES_DOWN) tp_dev.sta&=~(1<<7);//之前是按下，标记松开
                else //之前就没有被按下
                {
                    tp_dev.x[0]=0xffff;
                    tp_dev.y[0]=0xffff;
                    tp_dev.sta&=0XE0;//清除点有效标记
                }
            }
            if(t>240)t=10;//重新从 10 开始计数
            return res;
        }
```

以上代码，GT9147_Init 用于初始化 GT9147，该函数通过读取 0X8140~0X8143 这 4 个寄存器，并判断是否是：“9147”,来确定是不是 GT9147 芯片，在读取到正确的 ID 后，软复位 GT9147，然后根据当前芯片版本号，确定是否需要更新配置，通过 GT9147_Send_Cfg 函数，发送配置信息（一个数组），配置完后，结束软复位，即完成 GT9147 初始化。GT9147_Scan 函数,用于读取触摸屏坐标数据,这个和前面的 OTT2001A_Scan 大同小异,大家看源码即可。

最后我们打开 main.c,修改部分代码,这里就不全部贴出来了,仅介绍三个重要的函数：

```
//5 个触控点的颜色(电容触摸屏用)

const u16 POINT_COLOR_TBL[5]={RED,GREEN,BLUE,BROWN,GRED};
//电阻触摸屏测试函数
void rtp_test(void)
{
    u8 key; u8 i=0;
    while(1)
    {
        key=KEY_Scan(0);
        tp_dev.scan(0);
        if(tp_dev.sta&TP_PRES_DOWN)            //触摸屏被按下
        {
            if(tp_dev.x[0]<lcddev.width&&tp_dev.y[0]<lcddev.height)
```

```
                {

if(tp_dev.x[0]>(lcddev.width-24)&&tp_dev.y[0]<16)Load_Drow_Dialog();
                    else TP_Draw_Big_Point(tp_dev.x[0],tp_dev.y[0],RED);/画图

                }
            }else delay_ms(10);      //没有按键按下的时候
            if(key==KEY0_PRES)  //KEY0 按下,则执行校准程序
            {
                LCD_Clear(WHITE);    //清屏
                TP_Adjust();                //屏幕校准
                TP_Save_Adjdata();
                Load_Drow_Dialog();
            }
            i++;
            if(i%20==0)LED0=!LED0;
        }
}
//电容触摸屏测试函数
void ctp_test(void)
{
        u8 t=0; u8 i=0;
        u16 lastpos[5][2];          //最后一次的数据
        while(1)
        {
            tp_dev.scan(0);
            for(t=0;t<5;t++)
            {
                if((tp_dev.sta)&(1<<t))
                {
                    if(tp_dev.x[t]<lcddev.width&&tp_dev.y[t]<lcddev.height)
                    {
                        if(lastpos[t][0]==0XFFFF)
                        {
                            lastpos[t][0] = tp_dev.x[t];
                            lastpos[t][1] = tp_dev.y[t];
                        }
                        lcd_draw_bline(lastpos[t][0],lastpos[t][1],tp_dev.x[t],tp_dev.y[t],2,
                                       POINT_COLOR_TBL[t]);//画线
                        lastpos[t][0]=tp_dev.x[t];
                        lastpos[t][1]=tp_dev.y[t];
                        if(tp_dev.x[t]>(lcddev.width-24)&&tp_dev.y[t]<20)
                        {
                            Load_Drow_Dialog();//清除
```

```
                                }
                            }
                        }else lastpos[t][0]=0XFFFF;
                    }
                    delay_ms(5);i++;
                    if(i%20==0)LED0=!LED0;
                }
            }
            int main(void)
            {
                NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
                delay_init(168);  //初始化延时函数
                uart_init(115200);      //初始化串口波特率为 115200
                LED_Init();                     //初始化 LED
                LCD_Init();                     //LCD 初始化
                KEY_Init();                 //按键初始化
                tp_dev.init();              //触摸屏初始化
                POINT_COLOR=RED;//设置字体为红色
                LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
                LCD_ShowString(30,70,200,16,16,"TOUCH TEST");
                LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
                LCD_ShowString(30,110,200,16,16,"2014/5/7");
                if(tp_dev.touchtype!=0XFF)LCD_ShowString(30,130,200,16,16,"Press KEY0 to
Adjust");
                                                                    //电阻屏才显示
                delay_ms(1500);
                Load_Drow_Dialog();

                if(tp_dev.touchtype&0X80)ctp_test();//电容屏测试
                else rtp_test();                    //电阻屏测试
            }
```

下面分别介绍一下这三个函数。

rtp_test，该函数用于电阻触摸屏的测试，该函数代码比较简单，就是扫描按键和触摸屏，如果触摸屏有按下，则在触摸屏上面划线，如果按中“RST”区域，则执行清屏。如果按键 KEY0 按下，则执行触摸屏校准。

ctp_test，该函数用于电容触摸屏的测试，由于我们采用 tp_dev.sta 来标记当前按下的触摸屏点数，所以判断是否有电容触摸屏按下，也就是判断 tp_dev.sta 的最低 5 位，如果有数据，则划线，如果没数据则忽略，且 5 个点划线的颜色各不一样，方便区分。另外，电容触摸屏不需要校准，所以没有校准程序。

main 函数，则比较简单，初始化相关外设，然后根据触摸屏类型，去选择执行 ctp_test 还是 rtp_test。

软件部分就介绍到这里，接下来看看下载验证。

33.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 探索者 STM32F4 开发板上，电阻触摸屏测试如图 33.4.1 所示界面：

图 33.4.1 电阻触摸屏测试程序运行效果

图中我们在电阻屏上画了一些内容，右上角的 RST 可以用来清屏，点击该区域，即可清屏重画。另外，按 KEY0 可以进入校准模式，如果发现触摸屏不准，则可以按 KEY0，进入校准，重新校准一下，即可正常使用。

如果是电容触摸屏，测试界面如图 33.4.2 所示：

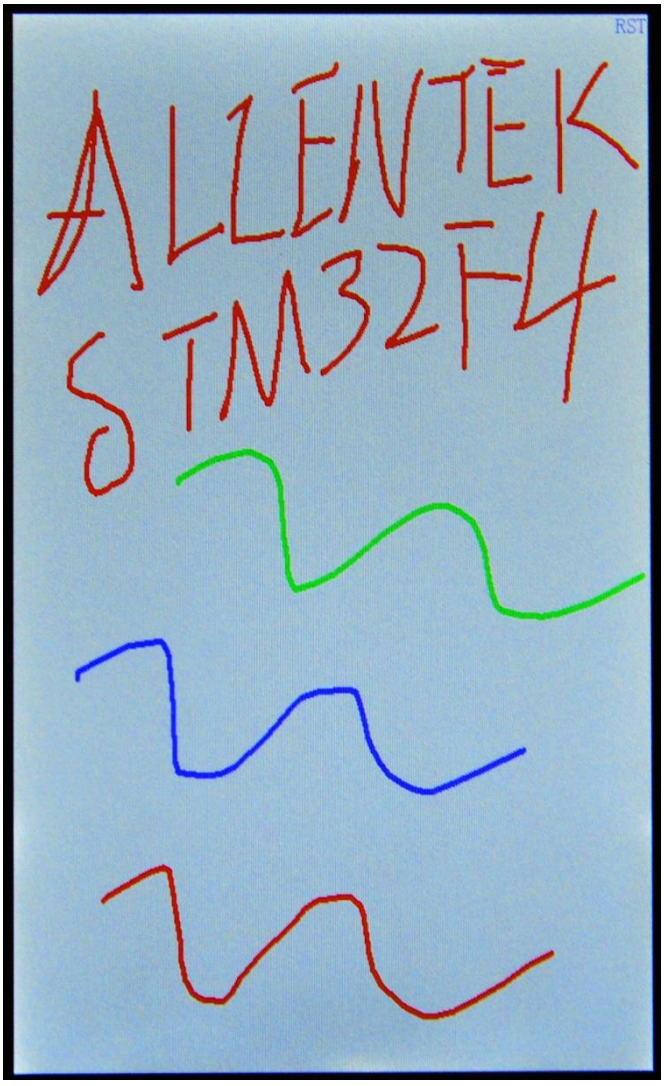

图 33.4.2 电容触摸屏测试界面

图中，同样输入了一些内容。电容屏支持多点触摸，每个点的颜色都不一样，图中的波浪线就是三点触摸画出来的，最多可以 5 点触摸。注意：电容触摸屏支持：ALIENTEK 4.3 寸电容触摸屏模块或者 ALIENTEK 新款 7 寸电容触摸屏模块（SSD1963+FT5206 方案），老款的 7 寸电容触摸屏模块（CPLD+GT811 方案）本例程不支持!!

同样，按右上角的 RST 标志，可以清屏。电容屏无需校准，所以按 KEY0 无效。KEY0 校准仅对电阻屏有效。