AN1413 ATK-4.3’TFTLCD 电容触摸屏模块使用说明

本应用文档(AN1413,对应**战舰 STM32 开发板标准例程 实验 26/MiniSTM32 开发板(V3.0)标准例程 实验 21(同时适用 V3.0 之前的 Mini 板)**）将教大家如何在 ALIENTEK STM32 开发板上使用 ATK-4.3’ TFTLCD 电容触摸屏模块(**注意,本文档同时适用 ALIENTEK 战舰和 MiniSTM32 两款开发板**)。

本文档分为如下几部分：

1， ATK-4.3’ TFTLCD 电容触摸屏模块简介

2， 硬件连接

3， 软件实现

4， 验证

1、ATK-4.3’ TFTLCD 电容触摸屏模块简介

ATK-4.3’ TFTLCD 模块是 ALIENTEK 推出的一款高性能 4.3 寸电容触摸屏模块。该模块屏幕分辨率为 800*480，16 位真彩显示，采用 NT35510 驱动，该芯片直接自带 GRAM，无需外加驱动器，因而任何单片机，都可以轻易驱动。

模块采用电容触摸屏，最多支持 5 点同时触摸，具有非常好的操控效果。模块硬件接口与 ALIENTEK 其他液晶模块(2.4’/2.8’/3.5’/7’等)接口完全一致，因而原有产品，硬件上不需要任何变动，只需要稍微修改一下软件，就可以使用我们的 4.3’ TFTLCD 电容触摸屏模块。

1.1 模块引脚说明

ATK-4.3’ TFTLCD 电容触摸屏模块通过 2*17 的排针（2.54mm 间距）同外部连接，模块可以与 ALIENTEK 的 STM32 开发板直接对接,我们提供相应的例程,用户可以在 ALIENTEK STM32 开发板上直接测试。ATK-4.3’ TFTLCD 电容触摸屏模块外观如图 1.1.1 所示：

图 1.1.1-1 ATK-4.3’ TFTLCD 电容触摸屏模块正面图

图 1.1.1-2 ATK-4.3’ TFTLCD 电容触摸屏模块背面图

ATK-4.3’ TFTLCD 电容触摸屏模块通过 34（2*17）个引脚同外部连接，对外接口原理图如图 1.1.2 所示:

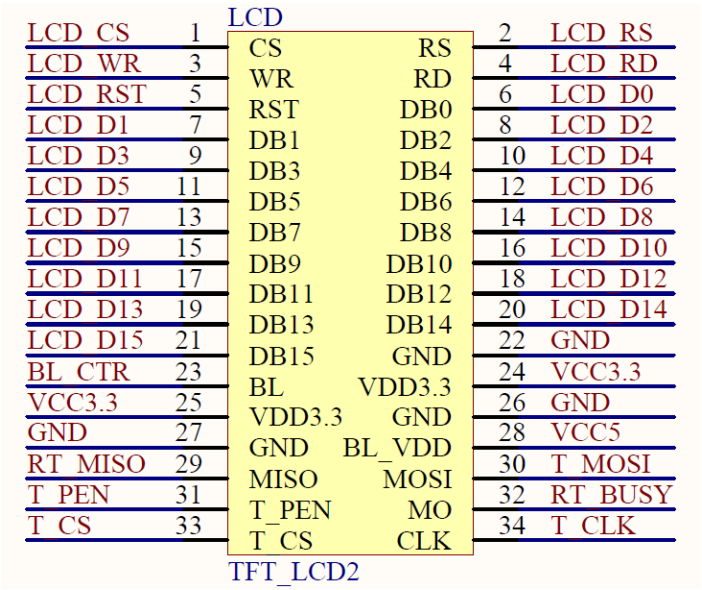

图 1.1.2 模块对外接口原理图

对应引脚功能详细描述如表 1.1.1 所示：

序号	名称	说明
1	CS	LCD 片选信号（低电平有效）
2	RS	命令/数据控制信号（0，命令；1，数据；）
3	WR	写使能信号（低电平有效）
4	RD	读使能信号（低电平有效）
5	RST	复位信号（低电平有效）
6~21	D0~D15	双向数据总线
22,26,27	GND	地线
23	BL_CTR	背光控制引脚（高电平点亮背光，低电平关闭）

24,25	VCC3.3	主电源供电引脚（3.3V）
28	VCC5	背光供电引脚（5V）
29	MISO	NC，电容触摸屏未用到
30	MOSI	电容触摸屏 IIC_SDA 信号(CT_SDA)
31	PEN	电容触摸屏中断信号(CT_INT)
32	BUSY	NC，电容触摸屏未用到
33	CS	电容触摸屏复位信号(CT_RST)
34	CLK	电容触摸屏 IIC_SCL 信号(CT_SCL)

表 1.1.1 ATK-4.3’ TFTLCD 模块引脚说明

从上表可以看出，LCD 控制器总共需要 21 个 IO 口，背光控制需要 1 个 IO 口，电容触摸屏需要 4 个 IO 口，这样整个模块需要 26 个 IO 口驱动。

特别注意：模块需要双电源供电：5V 和 3.3V，都必须接上，才可以正常工作，5V 电源用于背光供电，3.3V 用于除背光外的其他电源部分供电。

1.2 LCD 接口时序

ATK-4.3’ TFTLCD 模块采用 16 位 8080 总线接口，总线读写时序如图 1.2.1 所示：

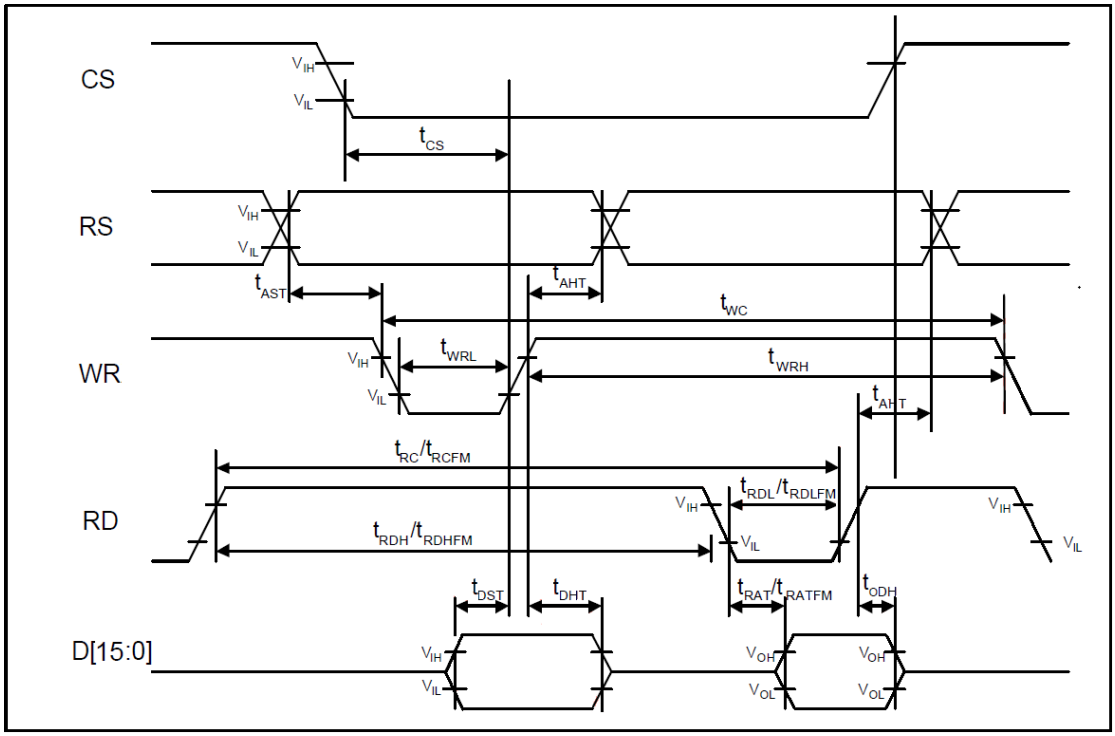

图 1.2.1 总线读写时序

图中各时间参数见表 1.2.1 所示：

Signal	Symbol	Parameter	MIN	MAX	Unit	Description
WR	t_{WC}	Write cycle	33	-	ns	
	t_{WRH}	Control pulse “H” duration	15	-	ns	
	t_{WRL}	Control pulse “L” duration	15	-	ns	
RD(ID)	t_{RC}	Read cycle (ID)	160	-	ns	When read ID data
	t_{RDH}	Control pulse “H” duration (ID)	90	-	ns	
	t_{RDL}	Control pulse “L” duration (ID)	45	-	ns	
RD(FM)	t_{RCFM}	Read cycle (FM)	400	-	ns	When read from frame memory
	t_{RDHFM}	Control pulse “H” duration (FM)	250	-	ns	
	t_{RDLFM}	Control pulse “L” duration (FM)	150	-	ns	
RS	t_{AST}	Address setup time (Write)	0	-	ns	
		Address setup time (Read)	10	-	ns	
	t_{AHT}	Address hole time	2	-	ns	
D[15:0]	t_{DST}	Data setup time	15	-	ns	
	t_{DHT}	Data hold time	10	-	ns	
	t_{RAT}	Read access time (ID)	-	40	ns	
	t_{RATFM}	Read access time (FM)	-	150	ns	
	t_{ODH}	Output disable time	5	-	ns	

表 1.2.1 16 位 8080 并口读写时间参数

从上表可以看出，模块的写周期是非常快的，只需要 33ns 即可，理论上最大速度可以达到：3030W 像素每秒，即刷屏速度可以达到每秒钟 78.9 帧。模块的读取速度相对较慢：读 ID（RD(ID)）周期是 160ns，读显存周期是 400ns（RD(FM)）。

LCD 详细的读写时序，请看 NT35510 数据手册第 28 页。

1.3 LCD 驱动说明

ATK-4.3’ TFTLCD 模块采用 NT35510 作为 LCD 驱动器，该驱动器自带 LCD GRAM，无需外加独立驱动器，并且，在指令上，基本兼容 ILI9341，使用非常方便。模块采用 16 位 8080 并口与外部连接（不支持其他接口方式，仅支持 16 位 8080 并口），在 8080 并口模式下，LCD 驱动需要用到的信号线如下：

CS：LCD 片选信号。

WR：向 LCD 写入数据。

RD：从 LCD 读取数据。

D[15：0]：16 位双向数据线。

RST：硬复位 LCD。

RS：命令/数据标志（0，读写命令；1，读写数据）。

除了以上信号，我们一般还需要用到这 2 个信号：RST 和 BL_CTR，其中 RST 是液晶的硬复位脚，低电平有效，用于复位 NT35510 芯片，实现液晶复位，在每次初始化之前，我们强烈建议大家先执行硬复位，再做初始化。BL_CTR 则是背光控制引脚，高电平有效，BL_CTR 自带了 100K 下拉电阻，所以如果这个引脚悬空，背光是不会亮的。必须接高电平，背光才会亮，另外可以用 PWM 控制 BL_CTR 脚，从而控制背光的亮度。

NT35510 自带 LCD GRAM（480*864*3 字节），并且最高支持 24 位颜色深度（1600 万色），不过，我们一般使用 16 位颜色深度（65K 色），RGB565 格式，这样，在 16 位模式下，可以达到最快的速度。

在 16 位模式下，NT35510 采用 RGB565 格式存储颜色数据，此时 NT35510 的低 16 位数据总线（高 8 位没有用到）与 MCU 的 16 位数据线以及 24 位 LCD GRAM 的对应关系如表 1.3.1 所示：

35510总线（16位）	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				D10	D9	D8	D7	D6	D5			D4	D3	D2	D1	D0			
LCD GRAM（24位）	R[4]	R[3]	R[2]	R[1]	R[0]	R[4]	R[3]	R[2]	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	G[5]	G[4]	B[4]	B[3]	B[2]	B[1]	B[0]	B[4]	B[3]	B[2]

表 1.3.1 16 位总线与 24 位 GRAM 对应关系

从上表可以看出，NT35510 的 24 位 GRAM 与 16 位 RGB565 的对应关系，其实就是分别将高位的 R、G、B 数据，搬运到低位做填充，“凑成”24 位，再显示。

MCU 的 16 位数据中，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 NT35510 的指令是 16 位宽，数据除了 GRAM 读写的时候是 16 位宽，其他都是 8 位宽的（高 8 位无效），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 NT35510 的几个重要命令，因为 NT35510 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 NT35510 的 datasheet 看看。里面对这些命令有详细的介绍。我们将介绍：0XDA00，0XDB00，0XDC00，0X3600，0X2A00~0X2A03，0X2B00~0X2B03，0X2C00，0X2E00 等 14 条指令。

首先来看指令：0XDA00，0XDB00，0XDC00，这三条指令是读 ID1，ID2，ID3 指令，也就是用于读取 LCD 控制器的 ID，该指令如表 1.3.2 所示：

顺序	控制			各位描述									HEX
	RS	RD	WR	D15	D14	D13	D12	D11	D10	D9	D8	D7~D0	
指令 1	0	1	↑	1	1	0	1	1	0	1	0	00H	DA00H
参数 1	1	↑	1	0	0	0	0	0	0	0	0	00H	00H
指令 2	0	1	↑	1	1	0	1	1	0	1	1	00H	DB00H
参数 2	1	↑	1	0	0	0	0	0	0	0	0	80H	80H
指令 3	0	1	↑	1	1	0	1	1	1	0	0	00H	DC00H
参数 3	1	↑	1	0	0	0	0	0	0	0	0	00H	00H

表 1.3.2 读 ID 指令描述

从上表可以看出，LCD 读 ID，总共由 3 个指令（0XDA00、0XDB00 和 0XDC00）构成，每个指令输出一个参数，每个 ID 以 8 位数据（即指令后的参数）的形式输出（高 8 位固定为 0），不过这里输出的 ID，并不包含 5510 这样的字样，仅有指令 0XDB00 会输出 ID：0X80，其他两个指令读到的 ID 都是 0。将 3 个指令的输出，组合在一起，可以得到 NT35510 的 ID 为：0X8000。

通过这个 ID，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：0X3600，这是存储访问控制指令，可以控制 NT35510 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 1.3.3 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	↑	36H	0	0	0	0	0	0	0	0	3600H
参数	1	1	↑	00H	MY	MX	MV	ML	BGR	MH	RSMX	RSMY	00XXH

表 1.3.3 0X3600 指令描述

从上表可以看出，0X3600 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 NT35510 的全部扫描方向，如表

1.3.4 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 1.3.4 MY、MX、MV 设置与 LCD 扫描方向关系表

这样，我们在利用 NT35510 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：0X2A00~0X2A03，这几个是列地址设置指令，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 1.3.5 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 1	0	1	↑	2AH	0	0	0	0	0	0	0	0	2A00H
参数 1	1	1	↑	00H	SC15	SC14	SC13	SC12	SC11	SC10	SC9	SC8	SC[15:8]
指令 2	0	1	↑	2AH	0	0	0	0	0	0	0	1	2A01H
参数 2	1	1	↑	00H	SC7	SC6	SC5	SC4	SC3	SC2	SC1	SC0	SC[7:0]
指令 3	0	1	↑	2AH	0	0	0	0	0	0	1	0	2A02H
参数 3	1	1	↑	00H	EC15	EC14	EC13	EC12	EC11	EC10	EC9	EC8	EC[15:8]
指令 4	0	1	↑	2AH	0	0	0	0	0	0	1	1	2A03H
参数 4	1	1	↑	00H	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0	EC[7:0]

表 1.3.5 0X2A00~0X2A03 指令描述

在默认扫描方式时，这 4 个指令用于设置 x 坐标，每条指令带有 1 个参数，实际上总共就是 2 个坐标值：SC 和 EC（SC 和 EC 都是 16 位的，由 2 个 8 位组成），即列地址的起始值和结束值，SC 必须小于等于 EC，且 0≤SC/EC≤479。一般在设置 x 坐标的时候，我们只需要 0X2A00 和 0X2A01 两条指令即可，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 NT35510 的时候设置），从而提高速度。

与 0X2A00~0X2A03 指令类似，指令：0X2B00~0X2B03，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 1.3.6 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 1	0	1	↑	2BH	0	0	0	0	0	0	0	0	2B00H
参数 1	1	1	↑	00H	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SP[15:8]
指令 2	0	1	↑	2BH	0	0	0	0	0	0	0	1	2B01H
参数 2	1	1	↑	00H	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SP[7:0]

指令 3	0	1	↑	2BH	0	0	0	0	0	0	1	0	2B02H
参数 3	1	1	↑	00H	EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	EP[15:8]
指令 4	0	1	↑	2BH	0	0	0	0	0	0	1	1	2B03H
参数 4	1	1	↑	00H	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0	EP[7:0]

表 1.3.6 0X2B00~0X2B03 指令描述

在默认扫描方式时，这 4 个指令用于设置 y 坐标，每条指令带有 1 个参数，实际上总共就是 2 个坐标值：SP 和 EP（SP 和 EP 都是 16 位的，由 2 个 8 位组成），即页地址的起始值和结束值，SP 必须小于等于 EP，且 0≤SP/EP≤799。一般在设置 y 坐标的时候，我们只需要带 0X2B00 和 0X2B01 两条指令即可，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 NT35510 的时候设置），从而提高速度。

接下来看指令：0X2C00，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 1.3.7 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	↑	2CH	0	0	0	0	0	0	0	0	2C00H
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 1.3.7 0X2C 指令描述

从上表可知，在收到指令 0X2C00 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：0X2E00，该指令是读 GRAM 指令，用于读取 NT35510 的显存（GRAM），该指令在 NT35510 的数据手册上面的描述是有误的，真实的输出情况如表 1.3.8 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	↑	2EH				0	0	0	0	0	0	0	0	2E00H
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 1.3.8 0X2E00 指令描述

该指令用于读取 GRAM，如表 1.3.8 所示，NT35510 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 NT35510 常用的几个指令，通过这几个指令，我们便可以很好的控制

NT35510 显示我们所要显示的内容了。

一般 TFTLCD 模块的使用流程如图 1.3.1：

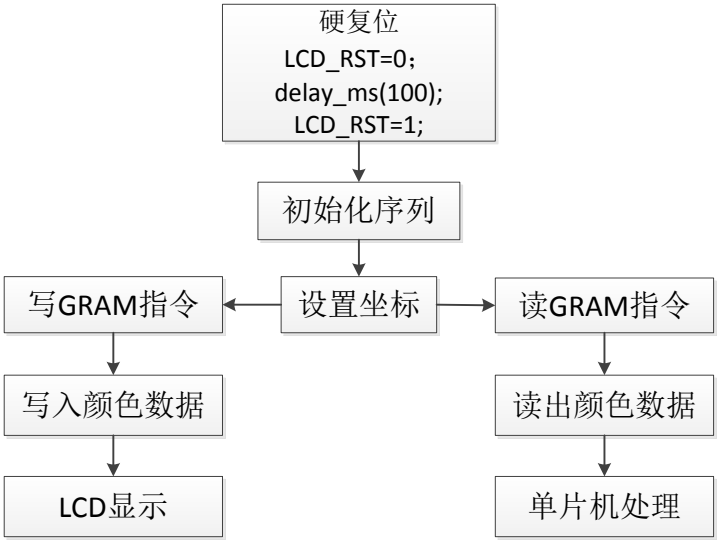

图 1.3.1 TFTLCD 使用流程

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，我们看看要利用 STM32 驱动模块并显示字符/数字的操作步骤：通过以上介绍，我们可以得出 TFTLCD 显示字符/数字需要的相关设置步骤如下：

1）设置 STM32 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。这里需要根据连接电路以及 TFTLCD 模块的设置来确定。

2）初始化 TFTLCD 模块。

即图 1.3.1 的初始化序列，这里我们例程里面没有硬复位 LCD 的操作，因我们 STM32 开发板的 LCD 接口，将 TFTLCD 的 RST 同 STM32 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，所以这步直接由 MCU 的硬复位替代了。

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 1.3.1 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/数字的目标，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。

1.4 电容触摸屏接口说明

1.4.1 电容式触摸屏简介

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 2.5.1.1 所示：

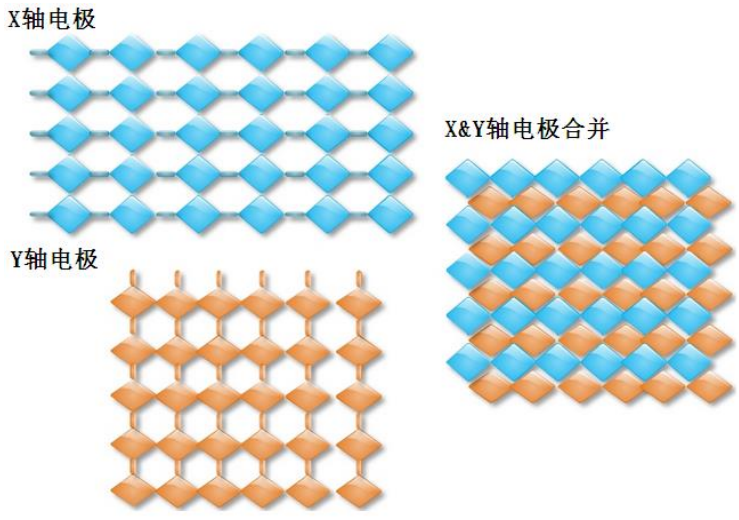

图 2.5.1.1 投射式电容屏电极矩阵示意图

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 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，在本例程，仅支持 ALIENTEK 4.3 寸 TFTLCD 电容触摸屏模块，所以这里介绍仅 OTT2001A 和 GT9147，GT811 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。

1.4.2 OTT2001A 简介

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，请参考开发板 IIC 实验。

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 1.4.2.1 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 1.4.2.1 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 1.4.2.2 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 1.4.2.2 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。

1.4.3 GT9147 简介

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 1.4.3.1 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 1.4.3.1 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 1.4.3.2 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 1.4.3.2 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。

2、硬件连接

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 初始化代码，如果不是，则执行 OTT2001A 的初始化代码，初始化电容触摸屏，随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

ATK-4.3’ TFTLCD 模块的接口同 ALIENTEK 的 2.4’/2.8’/3.5’ TFTLCD 模块接口一模一样，所以可以直接插在 ALIENTEK STM32 开发板上（还是靠右插哦！）。在硬件上，ATK-4.3’TFTLCD 模块与战舰 STM32 开发板的 IO 口对应关系如下：

RST 对应开发板的复位引脚 RESET，通过开发板复位键复位 LCD 控制器;

NCE 对应 PG12 即 FSMC_NE4;

RS 对应 PG0 即 FSMC_A10;

WR 对应 PD5 即 FSMC_NWE;

RD 对应 PD4 即 FSMC_NOE;

D[15:0]则直接连接在 FSMC_D15~FSMC_D0;

MOSI(CT_SDA)连接 PF9;

CLK(CT_SCL)连接 PB1;

PEN(CT_INT)连接 PF10;

CS(CT_RST)连接 PB2;

最后提醒大家两点注意事项：

1，这里我们仅针对战舰 STM32 的例程进行说明，MiniSTM32 开发板的例程，同战舰 STM32 的基本一样，请参考本文档学习即可。

2，如果使用 ALIENTEK MiniSTM32 开发板，且版本在 V2.0 之前的，由于 Mini 板在 PEN 信号上加入了 RC 滤波，需要将 C32（在 LCD 插座的右下角）去掉，否则 RC 滤波电路会将电容触摸屏的中断信号滤掉，导致电容触摸屏失效！！这个电容去掉后，并不会对我们后续使用 Mini 板造成不便，所以大家大可放心去掉它。

3、软件实现

本实验（注：这里仅以战舰板代码为例进行介绍，MiniSTM32 开发板对应代码与之相似，详见 MiniSTM32 开发板（V3.0 版本）标准例程 实验 21），本例程在原来战舰板的触摸屏实验（老版本，不支持电容触摸屏）的基础上修改，

3.1 LCD 驱动代码

战舰板原来的触摸屏实验（老版本），LCD 驱动部分代码，并不支持 NT35510，所以需

要在原来的 ILI93xx.c 和 lcd.c 里面加入 NT35510 驱动。这里整个代码比较多，我们就不全部贴出来了，具体代码请看最新版本的战舰板标准例程 实验 26，本节，我们仅挑一些重点进行介绍。

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 A10 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
{
    u16 LCD_REG;
    u16 LCD_RAM;
} LCD_TypeDef;
//使用 NOR/SRAM 的  Bank1.sector4,地址位 HADDR[27,26]=11 A10 作为数据命令区分
//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 | 0x000007FE))
#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 0X000007FE，则是 A10 的偏移量。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 0X6C00,07FE，对应 A10 的状态为 0(即 RS=0)，而 LCD-> LCD_RAM 的地址就是 0X6C00,0800（结构体地址自增），对应 A10 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。我们再来看 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 14 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，下面我们介绍 ILI93xx.c 里面的一些重要函数。

先看 6 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(u16 regval)
{
    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(u16 data)
{
    LCD->LCD_RAM=data;
}
//读 LCD 数据
//返回值:读到的值
u16 LCD_RD_DATA(void)
{
    return LCD->LCD_RAM;
}
//写寄存器
//LCD_Reg:寄存器地址
//LCD_RegValue:要写入的数据
void LCD_WriteReg(u8 LCD_Reg, u16 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(u8 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;
}
```

因为 FSMC 自动控制了 WR/RD/CS 等这些信号，所以这 6 个函数实现起来都非常简单，我们就不多说，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)。其中可以看到：NT35510 的设置同我们 1.3 接介绍的一样，只设置了 SC 和 SP，可以加快速度。

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（对于我们的 4.3 屏，GRAM 就需要更多了），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)
{
    u16 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
    if(LCD->LCD_RAM)r=0;                           //dummy Read
    opt_delay(2);
    r=LCD->LCD_RAM;                                //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->LCD_RAM;
        g=r&0XFF;
        //对于 9341/5310/5510,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));    //这三个 IC 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 OLED 模块的字符显示函数（请参考 OLED 模块例程）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
//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;
    u16 colortemp=POINT_COLOR;
    //设置窗口
    num=num-' ';//得到偏移后的值
    if(!mode) //非叠加方式
    {
        for(t=0;t<size;t++)
        {
            if(size==12)temp=asc2_1206[num][t];  //调用 1206 字体
            else temp=asc2_1608[num][t];         //调用 1608 字体
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)POINT_COLOR=colortemp;
                else POINT_COLOR=BACK_COLOR;
                LCD_DrawPoint(x,y);
                temp<<=1;
                y++;
                if(x>=lcddev.width){POINT_COLOR=colortemp;return;}//超区域了
                if((y-y0)==size)
                {
                    y=y0; x++;
                    if(x>=lcddev.width){POINT_COLOR=colortemp;return;}//超区域
                    break;
                }
            }
        }
    }else//叠加方式
    {
        for(t=0;t<size;t++)
        {
            if(size==12)temp=asc2_1206[num][t];  //调用 1206 字体
            else temp=asc2_1608[num][t];         //调用 1608 字体
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)LCD_DrawPoint(x,y);
                temp<<=1;
                y++;
                if(x>=lcddev.height){POINT_COLOR=colortemp;return;}//超区域了
                if((y-y0)==size)
                {
                    y=y0; x++;
                    if(x>=lcddev.width){POINT_COLOR=colortemp;return;}//超区
```

```
                        break;
                    }
                }
            }
        }
        POINT_COLOR=colortemp;
    }
```

在 LCD_ShowChar 函数里面，我们采用画点函数来显示字符，虽然速度不如开辟窗口的方式，但是这样写可以有更好的兼容性，方便在不同 LCD 之间移植。该代码中我们用到了两个字符集点阵数据数组 asc2_1206 和 asc2_1608，这两个字符集的点阵数据的提取方式，同 OLED 实验章节介绍的提取方法是一模一样的。详细请参考 OLED 实验。

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：

```
//初始化 lcd
void LCD_Init(void)
{
    RCC->AHBENR|=1<<8;          //使能 FSMC 时钟
    RCC->APB2ENR|=1<<3;         //使能 PORTB 时钟
    RCC->APB2ENR|=1<<5;         //使能 PORTD 时钟
    RCC->APB2ENR|=1<<6;         //使能 PORTE 时钟
    RCC->APB2ENR|=1<<8;         //使能 PORTG 时钟
    RCC->APB2ENR|=1<<0;         //使能 AFIO 时钟
    GPIOB->CRL&=0XFFFFFFF0;//PB0 推挽输出 背光
    GPIOB->CRL|=0X00000003;
    //PORTD 复用推挽输出
    GPIOD->CRH&=0X00FFF000;
    GPIOD->CRH|=0XBB000BBB;
    GPIOD->CRL&=0XFF00FF00;
    GPIOD->CRL|=0X00BB00BB;
    //PORTE 复用推挽输出
    GPIOE->CRH&=0X00000000;
    GPIOE->CRH|=0XBBBBBBBB;
    GPIOE->CRL&=0X0FFFFFFF;
    GPIOE->CRL|=0XB0000000;
    //PORTG12 复用推挽输出 A0
    GPIOG->CRH&=0XFFF0FFFF;
    GPIOG->CRH|=0X000B0000;
```

```
GPIOG->CRL&=0XFFFFFFF0;//PG0->RS
GPIOG->CRL|=0X0000000B;
//寄存器清零
//bank1 有 NE1~4,每一个有一个 BCR+TCR，所以总共八个寄存器。
//这里我们使用 NE4 ，也就对应 BTCR[6],[7]。
FSMC_Bank1->BTCR[6]=0X00000000;
FSMC_Bank1->BTCR[7]=0X00000000;
FSMC_Bank1E->BWTR[6]=0X00000000;
//操作 BCR 寄存器      使用异步模式
FSMC_Bank1->BTCR[6]|=1<<12;           //存储器写使能
FSMC_Bank1->BTCR[6]|=1<<14;           //读写使用不同的时序
FSMC_Bank1->BTCR[6]|=1<<4;            //存储器数据宽度为 16bit
//操作 BTR 寄存器
//读时序控制寄存器
FSMC_Bank1->BTCR[7]|=0<<28;     //模式 A

FSMC_Bank1->BTCR[7]|=1<<0;      //地址建立时间（ADDSET）为 2 个 HCLK
//因为液晶驱动 IC 的读数据的时候，速度不能太快，尤其对 1289 这个 IC。
FSMC_Bank1->BTCR[7]|=0XF<<8;    //数据保存时间为 16 个 HCLK
//写时序控制寄存器
FSMC_Bank1E->BWTR[6]|=0<<28;    //模式 A
FSMC_Bank1E->BWTR[6]|=0<<0;     //地址建立时间（ADDSET）为 1 个 HCLK
//4 个 HCLK（HCLK=72M）因为液晶驱动 IC 的写信号脉宽，
//最少也得 50ns。72M/4=24M=55ns
FSMC_Bank1E->BWTR[6]|=3<<8;     //数据保存时间为 4 个 HCLK
//使能 BANK1,区域 4
FSMC_Bank1->BTCR[6]|=1<<0;      //使能 BANK1，区域 4
delay_ms(50); // delay 50 ms
LCD_WriteReg(0x0000,0x0001);
delay_ms(50); // delay 50 ms
lcddev.id = LCD_ReadReg(0x0000);
if(lcddev.id <0XFF||lcddev.id==0XFFFF)//读到 ID 不正确
{
    //尝试 9341 ID 的读取
    LCD_WR_REG(0XD3);
    LCD_RD_DATA();                  //dummy read
```

```
        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);
            LCD_RD_DATA();              //dummy read
            LCD_RD_DATA();              //读回 0X01
            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);
                LCD_RD_DATA();              //dummy read
                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)       //不是 NT35310,尝试是不是 NT35510
                {
                    LCD_WR_REG(0XDA00);
                    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 ID 是 8000H,为方便区分,这里强制设置为 5510
                }
            }
        }
    }
```

```
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==0X6804)             //6804 初始化
{
    ……//6804 初始化代码
} else if(lcddev.id==0X5310)             //5310 初始化
{
    ……//5310 初始化代码
} else if(lcddev.id==0X5510)             //5510 初始化
{
    ……//5510 初始化代码
}else if(lcddev.id==0x9325)//9325
{
    ……//9325 初始化代码
}else if(lcddev.id==0x9328)              //ILI9328     OK
{
    ……//9328 初始化代码
}else if(lcddev.id==0x9320||lcddev.id==0x9300)//未测试.
{
    ……//9300 初始化代码
}else if(lcddev.id==0X9331)
{
    ……//9331 初始化代码
}else if(lcddev.id==0x5408)
{
    ……//5408 初始化代码
}
else if(lcddev.id==0x1505)//OK
{
    ……//1505 初始化代码
}else if(lcddev.id==0xB505)
{
    ……//B505 初始化代码
}else if(lcddev.id==0xC505)
```

```
    {
        ……//C505 初始化代码
    }else if(lcddev.id==0x8989)
    {
        ……//8989 初始化代码
    }else if(lcddev.id==0x4531)
    {
        ……//4531 初始化代码
    }else if(lcddev.id==0x4535)
    {
        ……//4535 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);
}
```

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。

3.2 电容触摸屏驱动代码

在原来的 HARDWARE\TOUCH 文件夹下面新建：ctiic.c、ctiic.h、ott2001a.c、ott2001a.h、gt9147.c 和 gt9147.h 等 6 个文件，这四个文件用于驱动电容触摸屏。另外，原来 touch.c 和 touch.h 也要进行少量修改（注意：函数名有变化），以同时支持电阻和电容触摸屏。

首先，我们看看 touch.c 里面的代码，由于代码比较多，我们就不一一介绍了，这里我们仅介绍 TP_Init 函数，该函数根据 LCD 的 ID（即 lcddev.id）判别是电阻屏还是电容屏，然后执行不同的初始化，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
        {
            //注意,时钟使能之后,对 GPIO 的操作才有效
            //所以上拉之前,必须使能时钟.才能实现真正的上拉输出
            RCC->APB2ENR|=1<<3;      //PB 时钟使能
            RCC->APB2ENR|=1<<7;      //PF 时钟使能
            GPIOB->CRL&=0XFFFFF00F;//PB1   2
            GPIOB->CRL|=0X00000330;
            GPIOB->ODR|=3<<1;          //PB1 2  推挽输出
            GPIOF->CRH&=0XFFFFF000;
            GPIOF->CRH|=0X00000838;
            GPIOF->ODR|=7<<8;          //PF8,9,10  全部上拉
            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 或 OTT2001A_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，这两个函数在后续会介绍。

touch.c 里面的其他的函数我们这里就不多介绍了（请参考战舰板原来触摸屏实验的教程）。接下来打开 touch.h 文件，看看该文件的代码：

```
#ifndef __TOUCH_H__
#define __TOUCH_H__
#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)
    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          PFin(10)     //PF10 INT
#define DOUT         PFin(8)      //PF8  MISO
#define TDIN         PFout(9)     //PF9  MOSI
#define TCLK         PBout(1)     //PB1  SCLK
#define TCS          PBout(2)     //PB2  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)
u8 TP_Read_XY(u16 *x,u16 *y);                        //双方向读取(X+Y)
u8 TP_Read_XY2(u16 *x,u16 *y);                       //带加强滤波的双方向坐标读取
void TP_Drow_Touch_Point(u16 x,u16 y,u16 color);//画一个坐标校准点
void TP_Draw_Big_Point(u16 x,u16 y,u16 color);       //画一个大点
void TP_Save_Adjdata(void);                          //保存校准参数
u8 TP_Get_Adjdata(void);                             //读取校准参数
void TP_Adjust(void);                                //触摸屏校准
void TP_Adj_Info_Show(u16 x0,u16 y0,u16 x1,u16 y1,u16 x2,u16 y2,u16 x3,u16 y3,
                      u16 fac);//显示校准信息
//电阻屏/电容屏 共用函数
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 IIC 实验章节的 myiic.c 和 myiic.h 基本一样，这里就不单独介绍了，记得把 ctiic.c 加入 HARDWARE 组下。接下来看看：ott2001a.c，在该文件输入如下代码：

```
//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); //发送写命令
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg>>8);          //发送高 8 位地址
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg&0XFF);        //发送低 8 位地址
    CT_IIC_Wait_Ack();
    for(i=0;i<len;i++)
    {
        CT_IIC_Send_Byte(buf[i]);      //发数据
        ret=CT_IIC_Wait_Ack();
        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);    //发送写命令
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg>>8);        //发送高 8 位地址
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg&0XFF);      //发送低 8 位地址
    CT_IIC_Wait_Ack();
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_RD);    //发送读命令
```

```
        CT_IIC_Wait_Ack();
        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;
        RCC->APB2ENR|=1<<3;             //先使能外设 IO PORTB 时钟
        RCC->APB2ENR|=1<<7;             //先使能外设 IO PORTF 时钟
        GPIOB->CRL&=0XFFFFF0FF;   //PB2  推挽输出
        GPIOB->CRL|=0X00000300;
        GPIOB->ODR|=1<<2;           //PB2  输出高
        GPIOF->CRH&=0XFFFFF0FF;   //PF10  输入
        GPIOF->CRH|=0X00000800;
        GPIOF->ODR|=1<<10;          //PF10  上拉
        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];
    u8 i=0;
    u8 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 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;  //数据是 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) //之前是被按下的
            {
                tp_dev.sta&=~(1<<7);    //标记按键松开
            }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 CTP_Scan 函数，CTP_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，保存 ott2001a.c 文件，并把该文件加入到 HARDWARE 组下。接下来打开 ott2001a.h 文件，在该文件里面输入如下代码：

```
#ifndef __OTT2001A_H
#define __OTT2001A_H
#include "sys.h"
//IO 操作函数
#define OTT_RST         PBout(2) //OTT2001A 复位引脚
#define OTT_INT         PFin(10)  //OTT2001A 中断引脚
```

```
//通过 OTT_SET_REG 指令,可以查询到这个信息
//注意,这里的 X,Y 和屏幕的坐标系刚好是反的.
#define OTT_MAX_X       2700        //TP X 方向的最大值(竖方向)
#define OTT_MAX_Y       1500        //TP Y 方向的最大值(横方向)
//缩放因子
#define OTT_SCAL_X      0.2963      //屏幕的 纵坐标/OTT_MAX_X
#define OTT_SCAL_Y      0.32        //屏幕的 横坐标/OTT_MAX_Y
//I2C 读写命令
#define OTT_CMD_WR          0XB2    //写命令
#define OTT_CMD_RD          0XB3    //读命令
//寄存器地址
#define OTT_GSTID_REG   0X0000      //OTT2001A 当前检测到的触摸情况
#define OTT_TP1_REG     0X0100      //第一个触摸点数据地址
#define OTT_TP2_REG     0X0500      //第二个触摸点数据地址
#define OTT_TP3_REG     0X1000      //第三个触摸点数据地址
#define OTT_TP4_REG     0X1400      //第四个触摸点数据地址
#define OTT_TP5_REG     0X1800      //第五个触摸点数据地址
#define OTT_SET_REG     0X0900      //分辨率设置寄存器地址
#define OTT_CTRL_REG    0X0D00      //传感器控制(开/关)
u8 OTT2001A_WR_Reg(u16 reg,u8 *buf,u8 len);     //写寄存器(实际无用)
void OTT2001A_RD_Reg(u16 reg,u8 *buf,u8 len);   //读寄存器
void OTT2001A_SensorControl(u8 cmd);//传感器打开/关闭操作
u8 OTT2001A_Init(void);             //4.3 电容触摸屏始化函数
u8 OTT2001A_Scan(u8 mode);          //电容触摸屏扫描函数
#endif
```

这段代码比较简单，重点注意一下 OTT_SCAL_X 和 OTT_SCAL_Y 的由来，前面说了，OTT2001A 输出 X 范围固定为 0~2700，Y 范围固定为：0~1500，所以，要根据我们屏幕的分辨率(4.3 寸电容屏触摸屏分辨率为：800*480)进行一次换算，得到 LCD 坐标与 OTT2001A 坐标的比例关系：

OTT_SCAL_X=800/2700=0.2963

OTT_SCAL_Y=480/1500=0.32

这样，我们只需要将 OTT2001A 的输出坐标乘以比例因子，就可以得到真实的 LCD 坐标。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];
    RCC->APB2ENR|=1<<3;          //先使能外设 IO PORTB 时钟
    RCC->APB2ENR|=1<<7;          //先使能外设 IO PORTF 时钟
    GPIOB->CRL&=0XFFFFF0FF;  //PB2 推挽输出
    GPIOB->CRL|=0X00000300;
    GPIOB->ODR|=1<<2;        //PB2 输出高
    GPIOF->CRH&=0XFFFFF0FF;  //PF10 推挽输出
    GPIOF->CRH|=0X00000300;
    GPIOF->ODR|=1<<10;       //PF10 输出 1
    CT_IIC_Init();               //初始化电容屏的 I2C 总线
    GT_RST=0;                    //复位
    delay_ms(10);
    GT_RST=1;                    //释放复位
    delay_ms(10);
    GPIOF->CRH&=0XFFFFF0FF;  //清除原来设置
    GPIOF->CRH|=0X00000800;//PF10 下拉输入
    GPIOF->ODR&=~(1<<10);  //PF10 下拉
    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);//点个数转换为 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];
                    }
                    //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&0X80&&((mode&0XF)<6))
        {
            temp=0;
            GT9147_WR_Reg(GT_GSTID_REG,&temp,1);//清标志
        }
    }
    if((mode&0X8F)==0X80)//无触摸点按下
    {
```

```
            if(tp_dev.sta&TP_PRES_DOWN) //之前是被按下的
            {
                tp_dev.sta&=~(1<<7);   //标记按键松开
            }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。

保存 gt9147.c 文件，并把该文件加入到 HARDWARE 组下。其他代码，我们就不再介绍了，请大家参考光盘本例程源码。

最后我们打开 test.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
const u16 POINT_COLOR_TBL[CT_MAX_TOUCH]={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();//清除
                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==KEY_RIGHT)  //KEY_RIGHT 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<CT_MAX_TOUCH;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]<16)
                        {
                            Load_Drow_Dialog();//清除
                        }
                    }
                }else lastpos[t][0]=0XFFFF;
            }
            delay_ms(5);i++;
            if(i%20==0)LED0=!LED0;
        }
    }
    int main(void)
    {
        Stm32_Clock_Init(9);    //系统时钟设置
        uart_init(72,9600);     //串口初始化为 9600
        delay_init(72);         //延时初始化
        LED_Init();             //初始化与 LED 连接的硬件接口
        LCD_Init();             //初始化 LCD
        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(60,50,200,16,16,"WarShip STM32");
        LCD_ShowString(60,70,200,16,16,"TOUCH TEST");
        LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(60,110,200,16,16,"2014/2/12");
        if(tp_dev.touchtype!=0XFF)LCD_ShowString(60,130,200,16,16,"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。

4、验证

在代码编译成功之后，我们下载代码到我们的 STM32 开发板上，ATK-4.3’TFTLCD 电容触摸屏模块测试效果如图 4.1 和 4.2 所示：

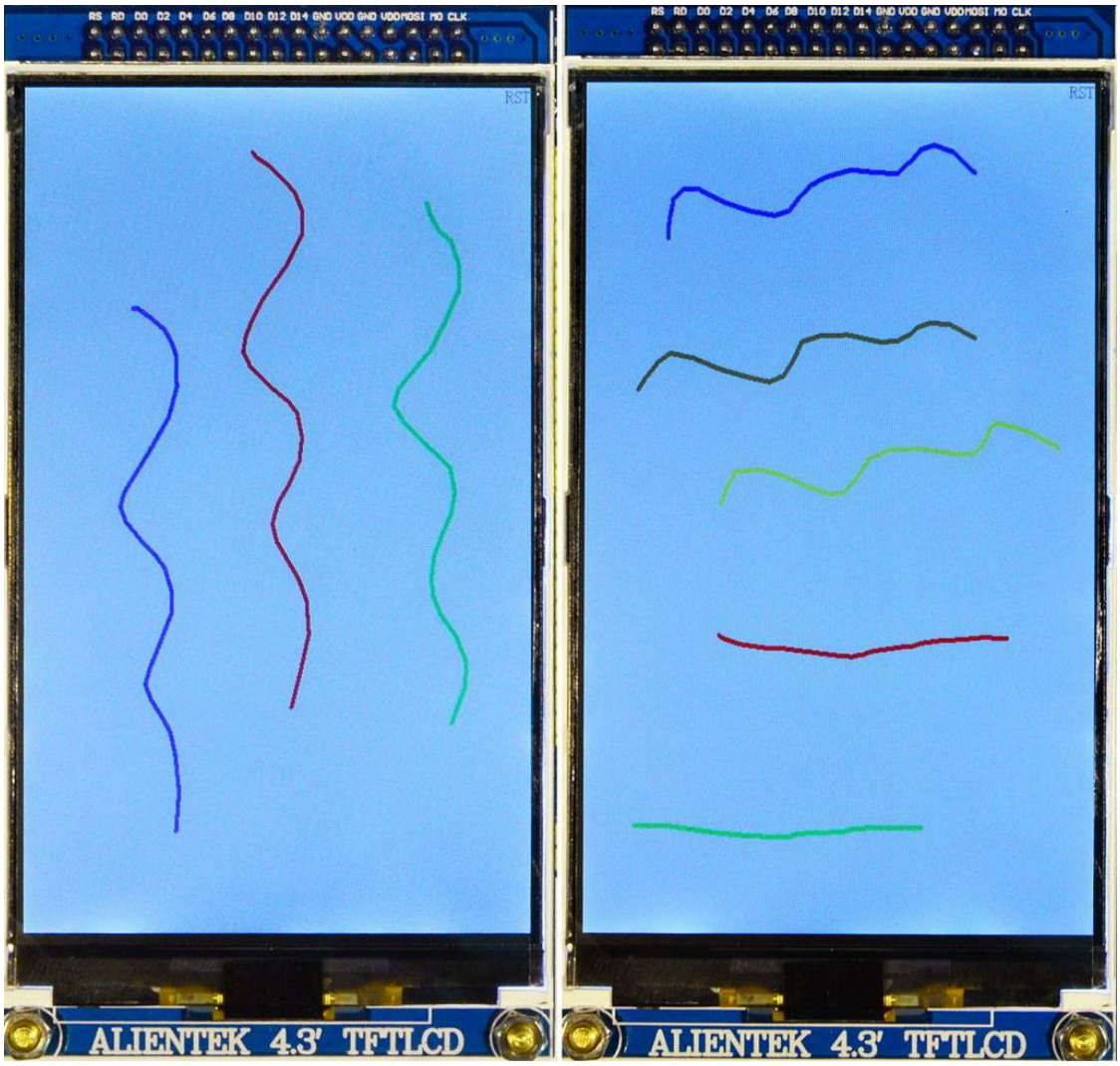

图 4.1 ATK-4.3’TFTLCD 电容触摸屏 3 点与 5 点画图效果

图 4.1 是 ATK-4.3’TFTLCD 电容触摸屏 3 点与 5 点画图效果，左侧图片是 3 点触摸的效果，右侧图片是 5 点触摸的效果。

这里提醒大家，多点触摸的时候，手指尖间隔不要太近，否则触摸效果不好，甚至无法实现多点触摸。

图 4.2 手写效果

图 4.2 是 ATK-4.3’TFTLCD 电容触摸屏的手写效果。

至此，ATK-4.3’ TFTLCD 电容触摸屏模块的使用就介绍完了，希望通过本文档，大家可以快速掌握 ATK-4.3’ TFTLCD 电容触摸屏模块的使用。

正点原子@ALIENTEK
公司网址：www.alientek.com
技术论坛：www.openedv.com
电话：020-38271790
传真：020-36773971