高通汉显®

GTHMI

GT-GUI LCD 1.9 寸液晶模组

GT-GL170320T19-XXXX

数据手册

V1.0

www.hmi.gaotongfont.cn

GUI-LCD 数据手册导读——开启液晶显示开发的极简之旅

■ 硬件篇：

1) **LCD 显示屏基本规格：**包含尺寸、分辨率、亮度、接口类型、驱动 IC 等；研发工程师可参考该规格初步确定该 LCD 显示屏是否符合技术要求；
2) **LCD 显示屏参考图纸：**显示屏结构，包括 LCD 显示屏外形尺寸， AA 尺寸，FPC 外形尺寸，FPC PIN 参数等；参考图纸可用于 PCB 布局和封装设计；
3) **LCD 屏接口原理图设计/PCB 布线：**
   参考规格书的目录 4 接口定义部分、目录 6 屏模块->8080 通信/SPI 通信->引脚介绍部分、 目录 7 GUI 芯片/字库芯片->引脚介绍部分、电容触摸模块->引脚介绍部分，注意若使用型号为搭载了 GUI 芯片的液晶模组时，为方便开发过程中烧录到 GUI 芯片修改内容，您在原理图设计时最好预留 8PIN 接口方便连接到 GUI 芯片

■ 软件篇：

1) **LCD 显示屏 SPI/8080 接口驱动开发：**根据 LCD 显示屏的示例代码和接口定义，开发屏 SPI/8080 屏驱动代码。参考目录 4 接口定义部分、目录 6 屏模块->8080 通信/SPI 通信->引脚部分；
2) **LCD 显示屏初始化设置：**根据 LCD 显示屏的示例代码完成显示屏的初始化配置，初始化后一般会有乱点出现代表初始化正常。参考目录 6 屏模块->8080/spi 驱动程序；
3) **LCD 显示屏图片/文字显示功能开发：**根据目录 6 屏模块->8080/spi 驱动程序下的 lcd_show_image 函数编写图形显示函数。根据目录 6 屏模块->8080/spi 驱动程序下 show_ch 函数编写文字显示函数；
4) **LCD 显示屏高通 GT-HMI 工具开发：**使用 GT-HMI Designer 和 GT-HMI Engine 工具,进行 LCD 显示屏 GUI 开发和移植，参考目录 9 HMI 移植；
5) **电容触摸驱动开发：**根据电容触摸屏示例代码及接口定义编写触摸驱动，参考目录 4 接口定义，目录 8 电容触摸模块下面的驱动例程；
6) **GUI 芯片/字库芯片驱动开发：**
   根据 GUI 芯片/字库芯片示例代码和接口定义开发芯片读写驱动；
   参考目录 7 GUI 芯片/字库芯片下面的 SPI 和 QSPI 例程。

修订记录

Rev.	Contents	Date
V1.0	First release	2023/09/11

目录

1.产品型号

GT- GL　170320　T/O　+19 —— X + X/G + X + X

GT-GL=高通 GUI-LCD

屏幕分辨率

T：TFT 彩屏
O: OLDE

屏幕尺寸

产品代码

消费类：X
工控类：G

C：电容触摸
R：电阻触摸
N：无触摸

容量：
4Mb
16Mb
32Mb
64Mb
128Mb

描述	型号
+液晶模组	GT-GL170320T19-S0XN
+GUI 芯片	GT-GL170320T19-S0XN64
+电容触摸	GT-GL170320T19-S0XC
+电容触摸+GUI 芯片	GT-GL170320T19-S0XC64

表 1-1

2.概述

GT-GUI LCD 1.9 寸液晶模组是高通开发的 GUI-LCD 轻量级嵌入式交互系统显示屏，使用 1.9 寸显示屏，分辨率为 170*320，搭配 GT-HMI 嵌入式 GUI-LCD 开发板可快速搭建起嵌入式 GUI 交互系统，驱动芯片为 ST7789V 芯片，支持 8080 端口通信和 SPI 通信,并搭载有高通字库芯片，本显示屏配合 GT-HMI 嵌入式 GUI 开发套件可支持高通全系列字库，例如中文、拉丁文、日文、韩文、希腊文、西里尔文、希伯来文、阿拉伯文、泰文等，客户可根据自身需求选择使用。

3.基本规格

No	Items	Parameter	Unit
1	LCD size	1.9	Inch
2	Number of Dots	170*(RGB)*320	Dots
3	Active Area	22.695 (W) x 42.720(H)	mm
4	Dimensional Outline（不含 TP）	25.8(W) *49.72(H)*1.35(T)	mm
5	Number of Pixels	170 (H)×320 (V)	pixels
6	Pixel Pitch	0.1335(W) × 0.1335 H)	mm
7	Pixel Arrangement	RGB Vertical Stripe	
8	Display Mode	Normally Black	
9	Weight	TBD	gram
10	display driver IC	ST7789V	
11	touch IC	BL6133	
12	GUI IC	64	Mb
13	Back Light	White LED	
14	Screen backlight voltage	3.0-3.4	V
15	Screen backlight current	80	mA
16	Screen communication mode	8080/SPI	
17	Flash communication mode	SPI/QSPI	
18	operating temperature	-20- +50℃	℃
19	Storage Temperature	-30- +60℃	℃

表 3-1

4.接口定义

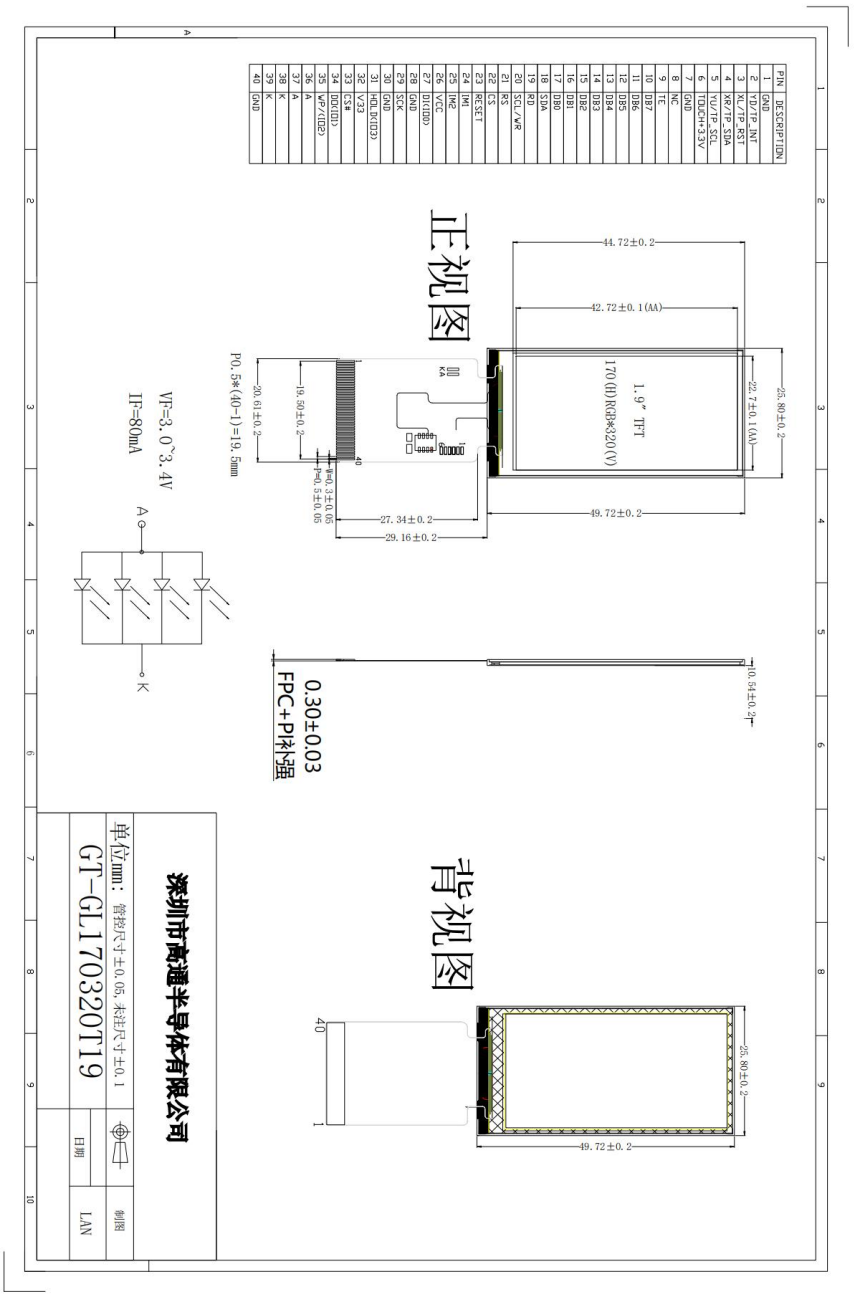
引脚	符号/名称	功能说明
1	GND	接地 0V
2	YD/TP_INT	触摸屏中断
3	XL/TP_RST	触摸屏复位
4	XR/TP_SDA	触摸屏数据
5	YU/TP_SCL	触摸屏时钟
6	TOUCH+3.3V	触摸屏供电(+3.3V)
7	GND	接地 0V
8	NC	空
9	TE	空
10	DB7	数据总线 DB7
11	DB6	数据总线 DB6
12	DB5	数据总线 DB5
13	DB4	数据总线 DB4
14	DB3	数据总线 DB3
15	DB2	数据总线 DB2
16	DB1	数据总线 DB1
17	DB0	数据总线 DB0
18	SDA	串口：串行数据 ； 并口：空
19	RD	串口：并口：读功能
20	WR/RS	并口： 写功能 串口： 寄存器选择信号， H/数据寄存器； L/指令寄存器
21	RS/SCL	并口： 寄存器选择信号， H/数据寄存器； L/指令寄存器 串口： 串行时
22	CS	片选，低电平选
23	RESET	复位，低电平复位；高电平正常工作
24	IM1	并/串口设置：IM1=1/ IM2=1(SPI)
25	IM2	并/串口设置：IM1=0/IM2=0(8080/8-bit)
26	VCC	显示屏供电（+3.3V)
27	DI(IO0)	字库芯片（接收数据）
28	GND	字库芯片地 0V
29	CLK	字库芯片（时钟信号）
30	GND	字库芯片地 0V
31	HOLD(IO3)	字库芯片数据
32	V33	字库芯片供电（+3.3V)
33	CS#	字库芯片(片选)
34	DO(IO1)	字库芯片（发送数据)
35	WP(IO2)	字库芯片写保护
36/37	BL_A	显示屏背光正极（3.0-3.4V）
38/39	BL_K	显示屏背光负极

40	GND	接地 0V

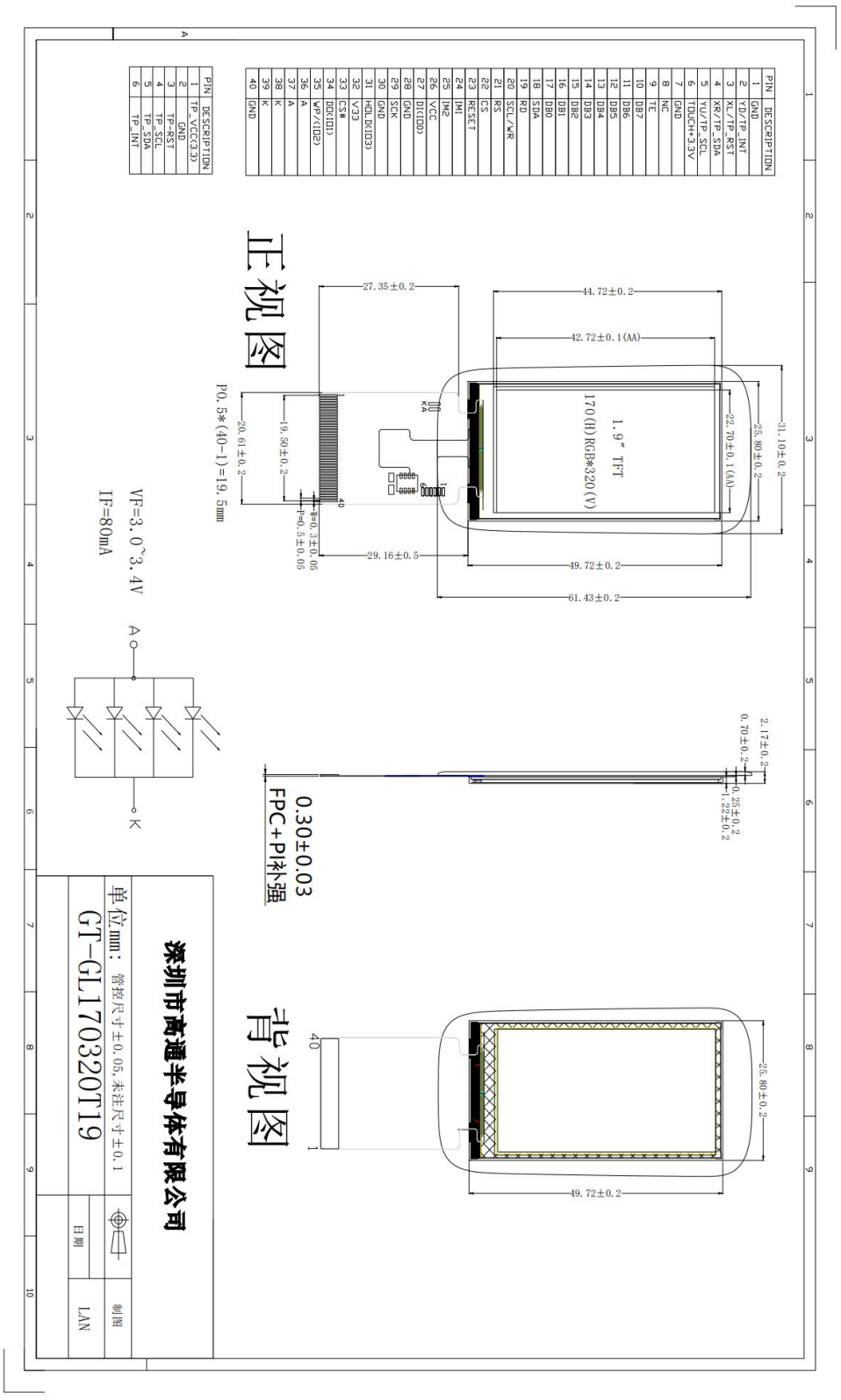
表 4-1

5.模组图纸

5.1 无 TP 模组图



5.2 电容 TP 模组图



PIN	DESCRIPTION
1	GND
2	YD/TP_INT
3	XL/TP_RST
4	XR/TP_SDA
5	YU/TP_SCL
6	TOUCH+3.3V
7	GND
8	NC
9	TE
10	DB7
11	DB6
12	DB5
13	DB4
14	DB3
15	DB2
16	DB1
17	DB0
18	SDA
19	RD
20	SCL/WR
21	RS
22	CS
23	RESET
24	IM1
25	IM2
26	VCC
27	DI(SD)
28	GND
29	SCK
30	GND
31	HDLDK(ID3)
32	V33
33	CS#
34	DDX(ID1)
35	WP/(ID2)
36	A
37	A
38	K
39	K
40	GND

PIN	DESCRIPTION
1	TP_VCC(3.3)
2	GND
3	TP-RST
4	TP_SCL
5	TP_SDA
6	TP_INT

6 屏模块

6.1 8080 通信

6.1.1 引脚介绍

开发板以 XMC 接口与 GT-GUI LCD 1.9 寸液晶模组 8080 通讯接口连接方式：8080 程序示例使用的即是此连接方式。

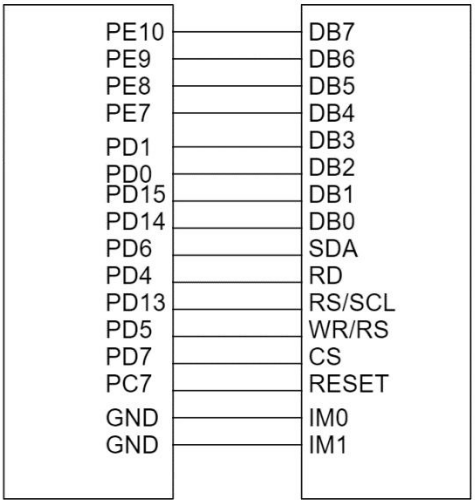


图 6-1

符号	名称	功能
D7	I/O	数据总线D7
D6	I/O	数据总线D6
D5	I/O	数据总线D5
D4	I/O	数据总线D4
D3	I/O	数据总线D3
D2	I/O	数据总线D2
D1	I/O	数据总线D1
D0	I/O	数据总线D0
RD	读	读使能
RS	寄存器选择信号	并口：H 写数据寄存器，L 写命令寄存器。 SPI:作为SCL时钟线
CS	片选	低电平片选
RST	屏复位	低电平复位，复位完成后，回到高电平，TFT 模块开始工作
WR	写	并口：写使能 SPI：H 写数据寄存器，L 写命令寄存器
IM1	IM1	IM1=0 选择并口, IM1=1 选择串口
IM2	IM2	IM2=0 选择并口, IM2=1 选择串口

图 6-2

图 6-2 为引脚功能图介绍，需要注意 IM1 和 IM2 引脚，RS 和 WR 引脚设置，这四个引脚在使用并口和串口功能时设置不一样。

6.1.2 8080 原理图

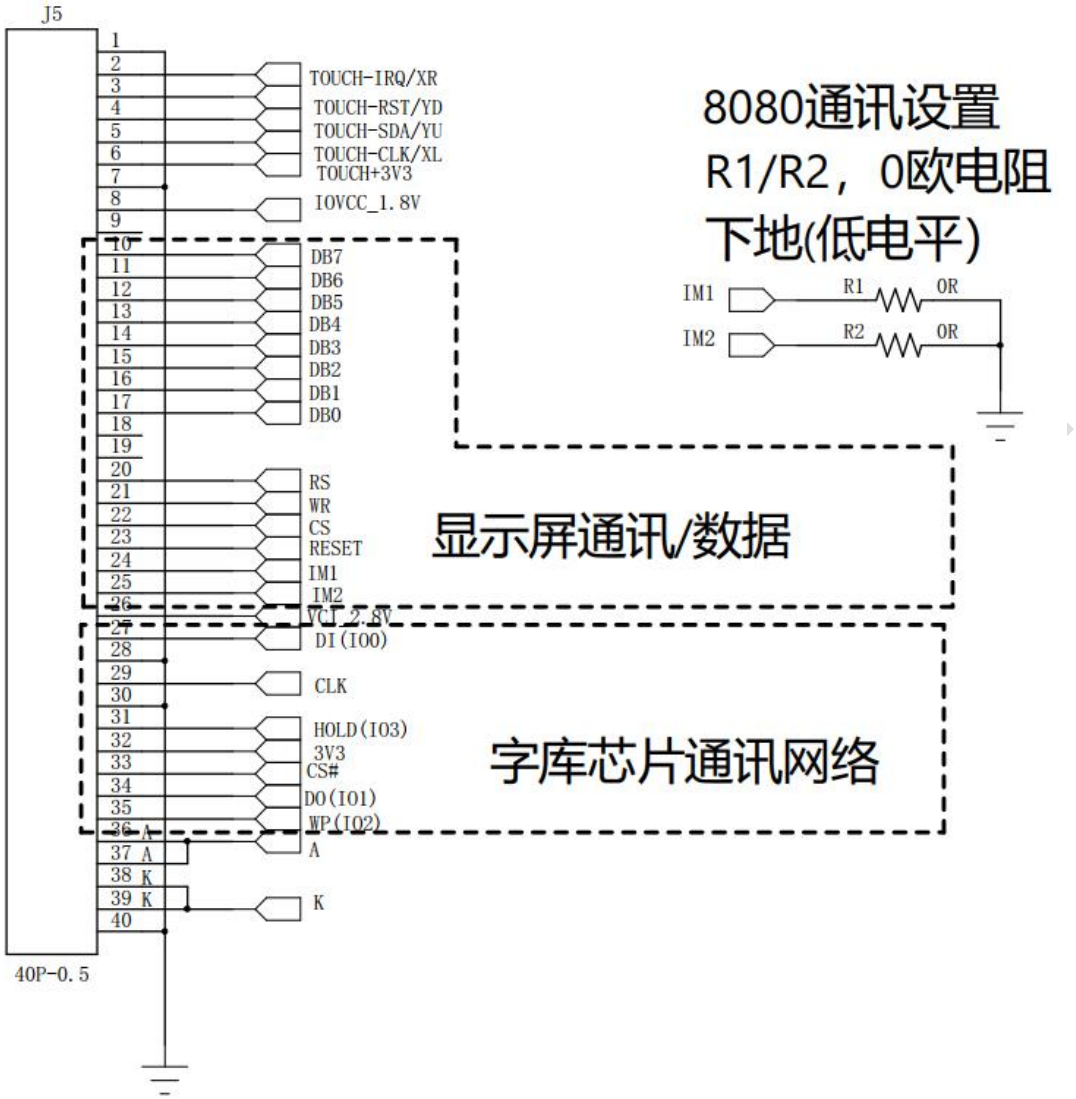


图 6-3

6.1.3 点亮屏流程

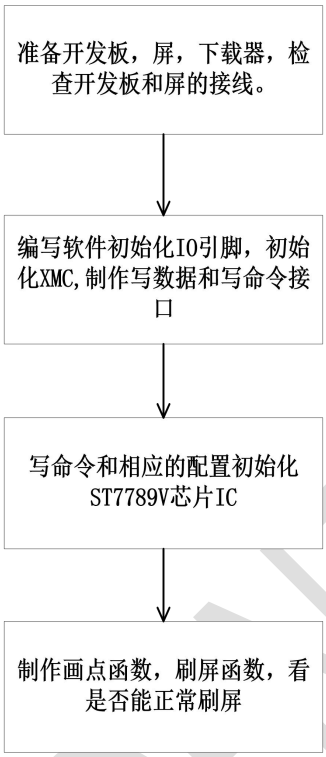


图 2.1.3

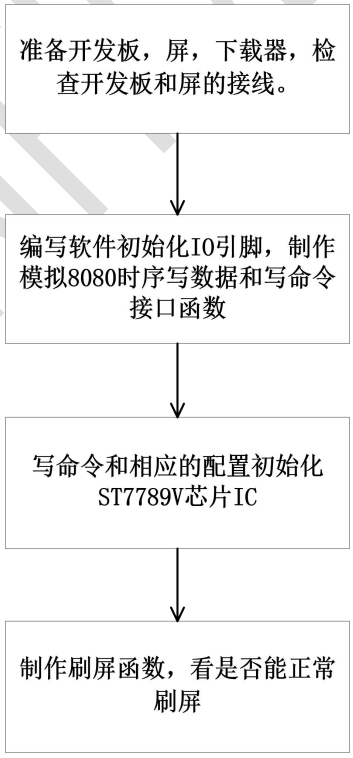


图 6-4

屏需要初始化屏 IC 才能正常显示，xmc 点屏操作步骤可参考图 6-3,模拟 8080 点屏参考图 6-4。

6.1.4 XMC 驱动程序

```
/* the address of write data & command (xmc_a0) */
#define   XMC_LCD_COMMAND                   0x60000000
#define   XMC_LCD_DATA                      (0x60000000|(1<<18)+1)
#define XMC_LCD_OFFSET_X   (0)    //屏幕 x 偏移
#define XMC_LCD_OFFSET_Y   (35)   //屏幕 y 偏移
#define XMC_LCD_WIDTH      (320)  //  屏幕横置  宽为 320
#define XMC_LCD_HEIGHT     (170)  //  屏幕横置  高为 170

/**
  * @brief   this function is write command to lcd.
  * @param   command : the command to write
  * @retval none
  */
void lcd_wr_command(uint8_t command)
{
  *(__IO uint8_t *) XMC_LCD_COMMAND = command;
}
/**
  * @brief   this function is write data to lcd.
  * @param   data : the data to write.
  * @retval none
  */
void lcd_wr_data(uint8_t data)
{
  *(__IO uint8_t *) XMC_LCD_DATA = data;
}
/**
  * @brief   configures the lcd.
  *          this function must be called before any write/read operation
  *          on the lcd.
  * @param   none
  * @retval none
  */
void lcd_xmc_init(void)
{
  /* init xmc */
  xmc_init(); //xmc 初始化
  /* reset lcd */
  LCD_RESET_HIGH; //复位引脚拉高
  delay_ms(200);
  LCD_RESET_LOW; //复位引脚拉低
  delay_ms(200);
```

```
LCD_RESET_HIGH;
delay_ms(800);

lcd_wr_command( 0x11); //退出睡眠

delay_ms(120);              //Delay 120ms

lcd_wr_command( 0x36); //行列扫描顺序及 RGB/BGR,横放/竖放控制
lcd_wr_data(0x60); //屏翻转方向

lcd_wr_command(0x3A); //定义 RGB 图像数据格式
lcd_wr_data( 0x55);

lcd_wr_command( 0xB2); //Porch Setting
lcd_wr_data(0x1F);
lcd_wr_data(0x1F);
lcd_wr_data(0x00);
lcd_wr_data( 0x33);
lcd_wr_data( 0x33);

lcd_wr_command( 0xB7); //Gate Control
lcd_wr_data(0x73);

lcd_wr_command( 0xBB); //VCOM Setting
lcd_wr_data( 0x1E);

lcd_wr_command(0xC0); //LCM Control
lcd_wr_data( 0x2C);

lcd_wr_command(0xC2); //VDV and VRH Command Enable
lcd_wr_data( 0x01);

lcd_wr_command(0xC3); //VRH Set
lcd_wr_data(0x13);

lcd_wr_command(0xC4); //VDV Set
lcd_wr_data(0x20);

lcd_wr_command( 0xC6);//Frame Rate Control in Normal Mode
lcd_wr_data( 0x0F);

lcd_wr_command( 0xD0); //Power Control
lcd_wr_data( 0xA4);
lcd_wr_data(0xA1);
```

```
lcd_wr_command(0xE0); //Positive Voltage Gamma Control
lcd_wr_data(0xF0);
lcd_wr_data(0x0C);
lcd_wr_data(0x15);
lcd_wr_data(0x09);
lcd_wr_data(0x09);

lcd_wr_data(0x07);
lcd_wr_data(0x3B);
lcd_wr_data(0x44);
lcd_wr_data(0x50);
lcd_wr_data(0x36);
lcd_wr_data(0x11);
lcd_wr_data(0x10);
lcd_wr_data(0x2F);
lcd_wr_data(0x35);

lcd_wr_command(0xE1); //Negative Voltage Gamma Control
lcd_wr_data(0xF0);
lcd_wr_data(0x17);
lcd_wr_data(0x1A);
lcd_wr_data(0x0C);
lcd_wr_data(0x0B);
lcd_wr_data(0x25);
lcd_wr_data(0x3A);
lcd_wr_data(0x43);
lcd_wr_data(0x4F);
lcd_wr_data(0x19);
lcd_wr_data(0x15);
lcd_wr_data(0x16);
lcd_wr_data(0x30);
lcd_wr_data(0x37);

lcd_wr_command( 0xe9); //Equalize time control
lcd_wr_data(0x07);
lcd_wr_data(0x07);
lcd_wr_data(0x03);
lcd_wr_command(0xE7); //SPI2 Enable
lcd_wr_data(0x10);
lcd_wr_command(0x21); //Display Inversion On 颜色翻转

lcd_wr_command(0x29); //Display on 开启显示

lcd_clear(WHITE); //清屏为白色
LCD_BL_LOW;    //点亮背光
```

```
}
/**
  * @brief  设置刷屏范围
  * @param  xstart : x 起始坐标值
  * @param  ystart : y 起始坐标值
  * @param  xend : x 结束坐标值
  * @param  yend : y 结束坐标值
  * @retval 无
  */
void lcd_setblock(uint16_t xs, uint16_t ys, uint16_t xe, uint16_t ye)
{
  xs += XMC_LCD_OFFSET_X;//添加偏移
  xe += XMC_LCD_OFFSET_X;
  ys += XMC_LCD_OFFSET_Y;
  ye += XMC_LCD_OFFSET_Y;
  /* 设置行范围 */
  lcd_wr_command(0x2a); //设置行范围命令
  lcd_wr_data(xs >> 8); //发送 x 起始坐标
  lcd_wr_data(xs);
  lcd_wr_data(xe >> 8); //发送 x 结束坐标
  lcd_wr_data(xe);

  /*设置列范围 */
  lcd_wr_command(0x2b); //设置列范围命令
  lcd_wr_data(ys >> 8); //发送 y 起始坐标
  lcd_wr_data(ys);
  lcd_wr_data(ye >> 8); //发送 y 起始坐标
  lcd_wr_data(ye);

  lcd_wr_command(0x2c); //写寄存器
}

/**
  * @brief  画点函数
  * @param  x :需要画点的 x 坐标值
  * @param  y :需要画点的 y 坐标值
  * @param  color :颜色值
  * @retval none
  */
void lcd_drawpoint(uint16_t x, uint16_t y, uint16_t color)
{
  lcd_setblock(x, y, x, y);  //设置刷屏区域

  lcd_wr_data(color >> 8);  //写颜色值
  lcd_wr_data(color);
```

```
}

/**
  * @brief  清屏函数.
  * @param  color : 清屏颜色值.
  * @retval 无
  */
void lcd_clear(uint16_t color)
{
  uint32_t i;

  lcd_setblock(0, 0, XMC_LCD_WIDTH-1, XMC_LCD_HEIGHT-1); //设置全屏区域

  for(i = 0; i < XMC_LCD_WIDTH * XMC_LCD_HEIGHT; i++)//发送 320x170 颜色值
  {
    lcd_wr_data(color >> 8);  //写颜色值
    lcd_wr_data(color);
  }
}
/**
* @brief  this function is display image on the lcd.
* @param  address:图片存在 flash 起始地址
* @param  x:图片显示 x 轴起始坐标
* @param  y:图片显示 y 轴起始坐标
* @param  w:图片宽度
* @param  h:图片高度
* @retval none
*/
void lcd_display_image(uint32_t address,uint16_t x,uint16_t y,uint16_t w,uint16_t h)
{
    uint8_t *dz_buff = buffer_write;
    uint32_t len = w * h << 1,i = 0;
    lcd_setblock(x,y,w-1,h-1);//设置区域
    spiflash_read(dz_buff,address,len);//从 flash 读图片数据
    for(i=0; i<len; i++)
    {
        lcd_wr_data(dz_buff[i]); //发送图片数据给屏幕
    }
}
//24x24 '啊'字
const u8 ch[72] ={
0x00,0x00,0x00,0x00,0x88,0x00,0x00,0xFC,0x06,0x44,0xCB,0xFF,
0x7E,0xC8,0x0C,0x64,0xC8,0x0C,0x64,0xC8,0x0C,0x64,0xD2,0x4C,
0x64,0xD3,0xEC,0x64,0xE2,0x4C,0x64,0xD2,0x4C,0x64,0xD2,0x4C,
0x64,0xCA,0x4C,0x64,0xCA,0x4C,0x7C,0xCB,0xCC,0x64,0xCA,0x4C,
```

```
0x64,0xFA,0x0C,0x40,0xD0,0x0C,0x00,0xC0,0x0C,0x00,0xC0,0x0C,
0x00,0xC0,0x0C,0x00,0xC0,0x7C,0x00,0xC0,0x18,0x00,0x80,0x10,
};

/**
* @brief   this function is display ch on the lcd.
* @param   x:汉字显示 x 轴起始坐标
* @param   y:汉字显示 y 轴起始坐标
* @param   w:汉字宽度
* @param   h:汉字高度
* @retval none
*/
void show_ch(u16 x,u16 y,u16 w,u16 h)
{
    unsigned int i,j,k,n;
    unsigned char temp;
    n = 0;
    lcd_setblock(x, y,x+w-1,y+h-1);//设置区域
    for( j = 0;j < ((w+7)>> 3);j++)//列数
    {
        for( i = 0;i < h; i++)//行数
        {
            temp = ch[n++];
            for(k = 0;k < 8;k++)
            {
                if(((temp << k)& 0x80) == 0 ){
                    /* 显示一个像素点 */
                    lcd_wr_data(0xff);
                    Lcd_wr_data(0xff);
                }else{
                    /* 显示一个像素点 */
                    lcd_wr_data(0x0);
                    lcd_wr_data(0x0);
                }
            }
        }
    }
}
```

6.1.5 模拟 8080 驱动程序

```
/**
* @brief   初始化 8080 用到的引脚.
```

```
* @param   none
* @retval none
*/
void lcd_port_8080_init(void)
{
    gpio_init_type  gpio_init_struct = {0};
    crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);//使能时钟
    crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_GPIOD_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_GPIOE_PERIPH_CLOCK, TRUE);

    gpio_init_struct.gpio_pins = GPIO_PINS_13 | GPIO_PINS_5 | GPIO_PINS_7 |GPIO_PINS_4;//配置 DC、WR、CS、
RD 引脚
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;//输出模式
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;//推挽
    gpio_init_struct.gpio_pull = GPIO_PULL_UP;//上拉
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(GPIOD, &gpio_init_struct);
    gpio_bits_set(GPIOD,GPIO_PINS_4|GPIO_PINS_5);//拉高 RD、WR 脚

    gpio_init_struct.gpio_pins =   GPIO_PINS_0 | GPIO_PINS_1 | GPIO_PINS_14 | GPIO_PINS_15;//配置 D0、D1、D2、
D3
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(GPIOD, &gpio_init_struct);

    gpio_init_struct.gpio_pins = GPIO_PINS_7 | GPIO_PINS_8 | GPIO_PINS_9 | GPIO_PINS_10;//配置 D4、D5、D6、
D7
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(GPIOE, &gpio_init_struct);

    /* lcd reset lines configuration */
    gpio_init_struct.gpio_pins = GPIO_PINS_7;//配置 RESER 引脚
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_UP;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(GPIOC, &gpio_init_struct);

    /* lcd BL lines configuration */
```

```
    gpio_init_struct.gpio_pins = GPIO_PINS_6;
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_UP;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(GPIOA, &gpio_init_struct);
    LCD_BL_HIGH;
}

/**
* @brief  8080 模拟发送一个字节数据函数.
* @param dat:发送的字节
* @retval none
*/
void  SendData(unsigned char dat)
{
    if((dat&0x01)==0x01) LCD_D0_HIGH;//一个引脚发送一位数据
    else LCD_D0_LOW;
    if( (dat&0x02)==0x02) LCD_D1_HIGH;
    else LCD_D1_LOW;
    if((dat&0x04)==0x04) LCD_D2_HIGH;
    else LCD_D2_LOW;
    if( (dat&0x08)==0x08 ) LCD_D3_HIGH;
    else LCD_D3_LOW;
    if((dat&0x10)==0x10) LCD_D4_HIGH;
    else LCD_D4_LOW;
    if( (dat&0x20)==0x20 ) LCD_D5_HIGH;
    else LCD_D5_LOW;
    if( (dat&0x40)==0x40) LCD_D6_HIGH;
    else LCD_D6_LOW;
    if( (dat&0x80)==0x80) LCD_D7_HIGH;
    else LCD_D7_LOW;
}
/**
* @brief  写命令函数.
* @param i:要发送的命令
* @retval none
*/
void WriteComm(unsigned int i)
{
    LCD_CS_LOW;//片选引脚拉低
    LCD_DC_LOW; //DC 脚拉低，发送命令

    SendData(i);//发送命令
    LCD_WR_LOW;//WR 脚拉低
```

```
    LCD_WR_HIGH;//WR 脚拉高上升沿写使能
    LCD_CS_HIGH;
}
void WriteData(unsigned int i)
{
    LCD_CS_LOW;//片选引脚拉低
    LCD_DC_HIGH;//DC 脚拉高，发送数据

    SendData(i);//发送数据
    LCD_WR_LOW;//WR 脚拉上升沿写使能
    LCD_WR_HIGH;
    LCD_CS_HIGH;
}

void lcd_8080_init(void)
{
    lcd_port_8080_init();//初始化 8080 引脚

    LCD_RESET_HIGH; //复位引脚拉高
    delay_ms(200);
    LCD_RESET_LOW;
    delay_ms(200);
    LCD_RESET_HIGH;
    WriteComm( 0x11); //退出睡眠

    delay_ms(120);          //Delay 120ms

    WriteComm( 0x36); //行列扫描顺序及 RGB/BGR,横放/竖放控制
    WriteData(0x60); //屏翻转方向

    WriteComm(0x3A); //定义 RGB 图像数据格式
    WriteData( 0x55);

    WriteComm( 0xB2); //Porch Setting
    WriteData(0x1F);
    WriteData(0x1F);
    WriteData(0x00);
    WriteData( 0x33);
    WriteData( 0x33);

    WriteComm( 0xB7); //Gate Control
    WriteData(0x73);

    WriteComm( 0xBB); //VCOM Setting
    WriteData( 0x1E);
```

```
WriteComm(0xC0); //LCM Control
WriteData( 0x2C);

WriteComm(0xC2); //VDV and VRH Command Enable
WriteData( 0x01);

WriteComm(0xC3); //VRH Set
WriteData(0x13);

WriteComm(0xC4); //VDV Set
WriteData(0x20);

WriteComm( 0xC6);//Frame Rate Control in Normal Mode
WriteData( 0x0F);

WriteComm( 0xD0); //Power Control
WriteData( 0xA4);
WriteData(0xA1);

WriteComm(0xE0); //Positive Voltage Gamma Control
WriteData(0xF0);
WriteData(0x0C);
WriteData(0x15);
WriteData(0x09);
WriteData(0x09);

WriteData(0x07);
WriteData(0x44);
WriteData(0x50);
WriteData(0x36);
WriteData(0x11);
WriteData(0x10);
WriteData(0x2F);
WriteData(0x35);

WriteComm(0xE1); //Negative Voltage Gamma Control
WriteData(0xF0);
WriteData(0x17);
WriteData(0x1A);
WriteData(0x0C);
WriteData(0x0B);
WriteData(0x25);
WriteData(0x3A);
WriteData(0x43);
```

```
    WriteData(0x4F);
    WriteData(0x19);
    WriteData(0x15);
    WriteData(0x16);
    WriteData(0x30);
    WriteData(0x37);

    WriteComm( 0xe9); //Equalize time control
    WriteData(0x07);
    WriteData(0x07);
    WriteData(0x03);
    WriteComm(0xE7); //SPI2 Enable
    WriteData(0x10);
    WriteComm(0x21); //Display Inversion On 颜色翻转

    WriteComm(0x29); //Display on 开启显示

    lcd_clear(WHITE); //清屏为白色
    LCD_BL_LOW;    //点亮背光
}
```

屏模拟 8080 驱动与 xmc 驱动主要区别在于底层写数据函数，其他屏发送初始化指令都一样，其他画点等函数可参考 xmc 驱动代码。

6.2 SPI 通信

6.2.1 SPI 通信连接图

本 SPI 通信例程使用模拟 SPI，如客户使用硬件 SPI 需要修改 SPI 驱动。图 6-5 为 SPI 通信连接图，IM1 和 IM0 脚没有标出来，使用 SPI 通信是需要将这两个引脚置高的，在使用 SPI 接口时，RS 引脚是和并口写使能引脚是同一个引脚，SPI 中 SCL 引脚和并口 RS 脚是同一个引脚，在编写驱动程序需要注意，图 6-5 已经标出引脚功能。

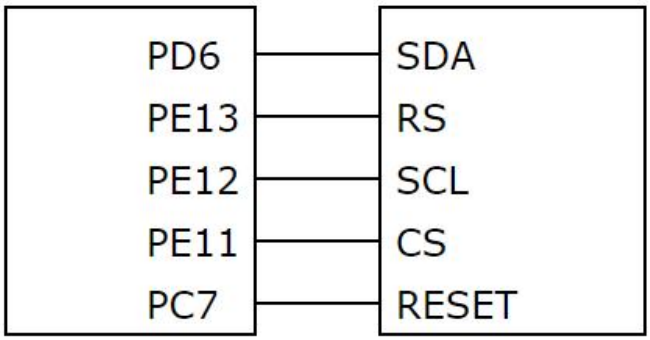


图 6-5

6.2.2 SPI 通信原理图

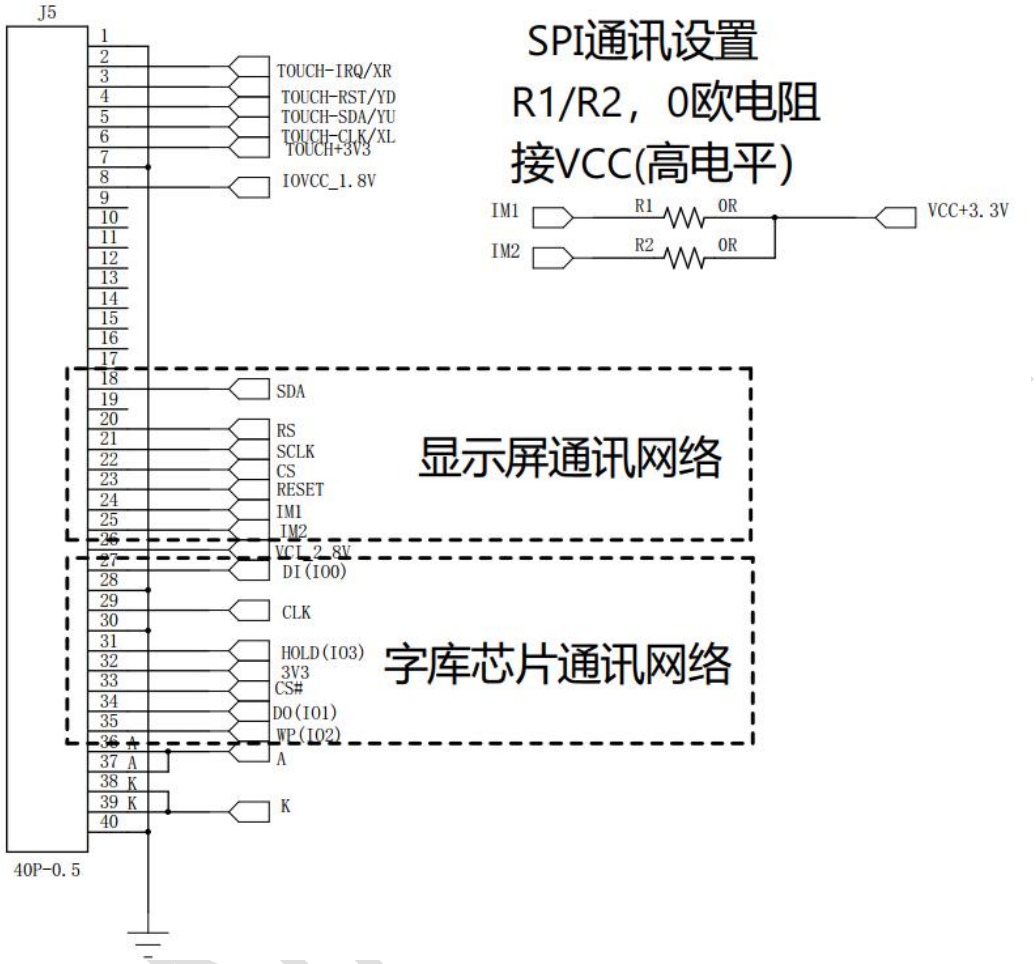


图 6-6

6.2.3 点亮屏流程

图 6-7

图 6-7 是 spi 驱动屏幕的步骤，与 XMC 通信相比，第二步不一样，还有第三步发送的指令数据也有所不同。

6.2.4 SPI 驱动程序

```
/**
* @brief SPI send data/command
* @param dat
*/
void   SendDataSPI(unsigned char dat)
{
    unsigned char i;

    for(i=0; i<8; i++)
    {
        LCD_SCL_LOW;
    if( (dat&0x80)!=0 ){
```

```
        LCD_SDA_HIGH;
    }else{
        LCD_SDA_LOW;
    }
        dat <<= 1;
        LCD_SCL_HIGH;
    }
}
/**
* @brief   this function is write command to lcd.
* @param   command :  写进去的命令值.
* @retval none
*/
void WriteComm(unsigned char command)
{
    LCD_CS_LOW;
    LCD_RS_LOW;//拉低选择发送命令
    SendDataSPI(command);
    LCD_CS_HIGH;
}
/**
* @brief   this function is write data to lcd.
* @param   data : the data to write.
* @retval none
*/
void WriteData(unsigned char data)
{
    LCD_CS_LOW;
    LCD_RS_HIGH;//拉高选择发送数据
    SendDataSPI(data);
    LCD_CS_HIGH;
}
/**
* @brief   configures the lcd.
*          this function must be called before any write/read operation
*          on the lcd.
*/
void lcd_soft_init(void)
{
    //软件 SPI 驱动  参数设置  颜色相反
    lcd_port_init(); //引脚初始化

    LCD_RESET_HIGH;  //复位引脚拉高
    delay_ms(1);
    LCD_RESET_LOW;  //复位引脚拉低
```

```
delay_ms(10);
LCD_RESET_HIGH;
delay_ms(120);

WriteComm(0x36); //行列扫描顺序及 RGB/BGR,横放/竖放控制
WriteData(0x60); //屏翻转方向

WriteComm(0x3A); //定义 RGB 图像数据格式
WriteData(0x05);

WriteComm(0xB2); //Porch Setting
WriteData(0x0C);
WriteData(0x0C);
WriteData(0x00);
WriteData(0x33);
WriteData(0x33);

WriteComm(0xB7); //Gate Control
WriteData(0x35);

WriteComm(0xBB); //VCOM Setting
WriteData(0x19);

WriteComm(0xC0); //LCM Control
WriteData(0x2C);

WriteComm(0xC2); //VDV and VRH Command Enable
WriteData(0x01);

WriteComm(0xC3); //VRH Set
WriteData(0x12);

WriteComm(0xC4); //VDV Set
WriteData(0x20);

WriteComm(0xC6); //Frame Rate Control in Normal Mode
WriteData(0x0F);

WriteComm(0xD0); //Power Control
WriteData(0xA4);
WriteData(0xA1);

WriteComm(0xE0); //Positive Voltage Gamma Control
WriteData(0xD0);
WriteData(0x04);
```

```
    WriteData(0x0D);
    WriteData(0x11);
    WriteData(0x13);
    WriteData(0x2B);
    WriteData(0x3F);
    WriteData(0x54);
    WriteData(0x4C);
    WriteData(0x18);
    WriteData(0x0D);
    WriteData(0x0B);
    WriteData(0x1F);
    WriteData(0x23);

    WriteComm(0xE1); //Negative Voltage Gamma Control
    WriteData(0xD0);
    WriteData(0x04);
    WriteData(0x0C);
    WriteData(0x11);
    WriteData(0x13);
    WriteData(0x2C);
    WriteData(0x3F);
    WriteData(0x44);
    WriteData(0x51);
    WriteData(0x2F);
    WriteData(0x1F);
    WriteData(0x1F);
    WriteData(0x20);
    WriteData(0x23);

    WriteComm(0x21); //Display Inversion On 颜色翻转

    WriteComm(0x11); //退出睡眠
    delay_ms(120);

    WriteComm(0x29); //Display on 开启显示
    DispColor(WHITE); //清屏为白色
    LCD_BL_LOW; //点亮背光
}
```

显示函数可以参考 8080 通信，这里不再赘述。

7 GUI 芯片模块

7.1 SPI 通信

7.1.1 引脚介绍

图 7-1 是普通 SPI 连接图，在使用 SPI 通信，HOLD 引脚和 WP 脚需要接 2K 电阻到 VCC。

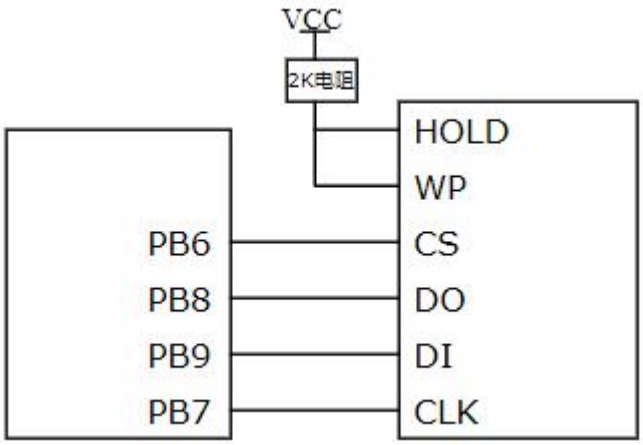


图 7-1

7.1.2 GUI 芯片指令

指令名称	字节 1	字节2	字节3	字节4	字节5	字节6	下一个字节
写使能	06h						
写禁能	04h						
读状态寄存器	05h	(S7-S0)					
写状态寄存器	01h	S7-S0					
读数据	03h	A23-A16	A15-A8	A7-A0	D7-D0	下一个字节	继续
快读	0Bh	A23-A16	A15-A8	A7-A0	伪字节	D7-D0	下一个字节
快读双输出	3Bh	A23-A16	A15-A8	A7-A0	伪字节	D7-D0	每四个时钟一个字节
页编程	02h	A23-A16	A15-A8	A7-A0	D7-D0	下一个字节	直到256个字节
块擦除(64k)	D8h	A23-A16	A15-A8	A7-A0			
扇区擦除(4k)	20h	A23-A16	A15-A8	A7-A0			
芯片擦除	C7h						
制造/器件ID	90h	伪字节	伪字节	00h	M7-M0		

图 7-2 操作存储芯片指令，可供客户写驱动做参考。

7.1.3 SPI 驱动程序

```
/**
* @brief   write a byte to flash
* @param   data: data to write
* @retval flash return data
*/
uint8_t spi_byte_write(uint8_t data)
{
    uint8_t brxbuff;
    spi_i2s_dma_transmitter_enable(SPI4, FALSE);
    spi_i2s_dma_receiver_enable(SPI4, FALSE);
    spi_i2s_data_transmit(SPI4, data);
    while(spi_i2s_flag_get(SPI4, SPI_I2S_RDBF_FLAG) == RESET);
    brxbuff = spi_i2s_data_receive(SPI4);
    while(spi_i2s_flag_get(SPI4, SPI_I2S_BF_FLAG) != RESET);
    return brxbuff;
}

/**
* @brief   read a byte to flash
* @param   none
* @retval flash return data
*/
uint8_t spi_byte_read(void)
{
    return (spi_byte_write(FLASH_SPI_DUMMY_BYTE));
}
/**
* @brief   read data from flash
* @param   pbuffer: the pointer for data buffer
* @param   read_addr: the address where the data is read
* @param   length: buffer length
* @retval none
*/
void spiflash_read(uint8_t *pbuffer, uint32_t read_addr, uint32_t length)
{
    FLASH_CS_LOW();
    spi_byte_write(0x03); /* send instruction 0x03 普通读取*/
    spi_byte_write((uint8_t)((read_addr) >> 16)); /* send 24-bit address */
    spi_byte_write((uint8_t)((read_addr) >> 8));
    spi_byte_write((uint8_t)read_addr);
    //spi_byte_write((uint8_t)0xff);//使用 0x0B 快速读取时需额外发送一个字节
    spi_bytes_read(pbuffer, length);
```

```
    FLASH_CS_HIGH();
}
/**
* @brief   erase a sector data
* @param   erase_addr: sector address to erase
* @retval none
*/
void spiflash_sector_erase(uint32_t erase_addr)
{
    spiflash_write_enable();

    spiflash_write_enable();
    spiflash_wait_busy();
    FLASH_CS_LOW();
    spi_byte_write(0x20); //擦除指令
    spi_byte_write((uint8_t)((erase_addr) >> 16));
    spi_byte_write((uint8_t)((erase_addr) >> 8));
    spi_byte_write((uint8_t)erase_addr);
    FLASH_CS_HIGH();
    spiflash_wait_busy();
}
/**
* @brief   read data continuously
* @param   pbuffer: buffer to save data
* @param   length: buffer length
* @retval none
*/
void spi_bytes_read(uint8_t *pbuffer, uint32_t length)
{
    while(length--)
    {
        while(spi_i2s_flag_get(SPI4, SPI_I2S_TDBE_FLAG) == RESET);
        spi_i2s_data_transmit(SPI4, 0xa5);//随意值皆可
        while(spi_i2s_flag_get(SPI4, SPI_I2S_RDBF_FLAG) == RESET);
        *pbuffer = spi_i2s_data_receive(SPI4);
        pbuffer++;
    }
}
/**
* @brief   read device id
* @param   none
* @retval device id
*/
uint16_t spiflash_read_id(void)
{
```

```
    uint16_t wreceivedata = 0;
    FLASH_CS_LOW();
    spi_byte_write(0x90); //0x90 读取 id 指令
    spi_byte_write(0x00);
    spi_byte_write(0x00);
    spi_byte_write(0x00);
    wreceivedata |= spi_byte_read() << 8;
    wreceivedata |= spi_byte_read();
    FLASH_CS_HIGH();
    return wreceivedata;
}
```

7.2 QSPI 通信

7.2.1 引脚介绍

图 7-3 是字库芯片连接图，IO0~IO3 是 QSPI 数据线，传输数据用，QSPI 是四线并口传输，速度比普通 SPI 快。

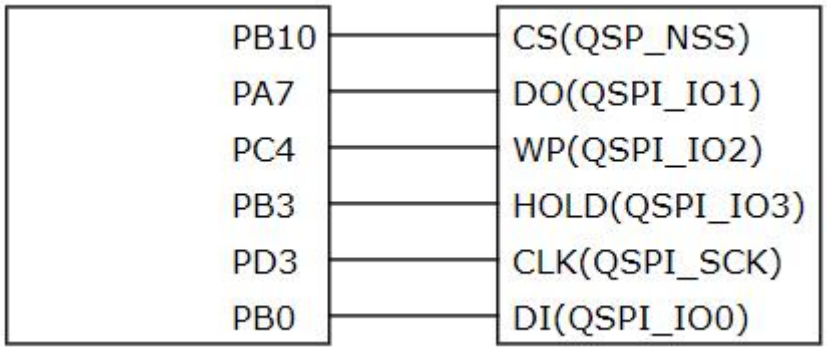


图 7-3

7.2.2 QSPI 驱动程序

```
qspi_cmd_type qspi_flash_cmd_config;
#define QSPI_FLASH_FIFO_DEPTH            (32*4)
#define QSPI_FLASH_PAGE_SIZE             256
#define QSPI_FLASH_QSPIx                 QSPI1
/**
  * @brief   qspi flash write enable
  * @param   none
  * @retval none
  */
```

```
void qspi_flash_write_enable(void)
{
  qspi_flash_cmd_wren_config(&qspi_flash_cmd_config);//写使能配置
  qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

  /* wait command completed */
  while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
  qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
}
/**
  * @brief  qspi_flash cmd erase config
  * @param  qspi_cmd_struct: the pointer for qspi_cmd_type parameter
  * @param  addr: erase address
  * @retval none
  */
void qspi_flash_cmd_erase_config(qspi_cmd_type *qspi_cmd_struct, uint32_t addr)
{
  qspi_cmd_struct->pe_mode_enable = FALSE;//性能增强模式关闭
  qspi_cmd_struct->pe_mode_operate_code = 0;
  qspi_cmd_struct->instruction_code = 0x20;//擦除指令
  qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE;//命令长度
  qspi_cmd_struct->address_code = addr;//擦除地址
  qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_BYTE;//地址长度 3 字节
  qspi_cmd_struct->data_counter = 0;
  qspi_cmd_struct->second_dummy_cycle_num = 0;
  qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_111;
  qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO;
  qspi_cmd_struct->read_status_enable = FALSE;
  qspi_cmd_struct->write_data_enable = TRUE;
}
/**
  * @brief  qspi_flash cmd rdsr config
  * @param  qspi_cmd_struct: the pointer for qspi_cmd_type parameter
  * @retval none
  */
void qspi_flash_cmd_rdsr_config(qspi_cmd_type *qspi_cmd_struct)
{
  qspi_cmd_struct->pe_mode_enable = FALSE;
  qspi_cmd_struct->pe_mode_operate_code = 0;
  qspi_cmd_struct->instruction_code = 0x05;//读状态寄存器指令
  qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE;//命令长度
  qspi_cmd_struct->address_code = 0;
  qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_0_BYTE;
  qspi_cmd_struct->data_counter = 0;
  qspi_cmd_struct->second_dummy_cycle_num = 0;
```

```
  qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_111;
  qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO;
  qspi_cmd_struct->read_status_enable = TRUE;
  qspi_cmd_struct->write_data_enable = FALSE;
}
/**
  * @brief   qspi flash check busy
  * @param   none
  * @retval none
  */
void qspi_flash_busy_check(void)
{
  qspi_flash_cmd_rdsr_config(&qspi_flash_cmd_config);
  qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

  /* wait command completed */
  while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
  qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
}
/**
  * @brief   qspi flash erase data
  * @param   sec_addr: the sector address for erase
  * @retval none
  */
void qspi_flash_erase(uint32_t sec_addr)
{
  qspi_flash_write_enable();//写使能

  qspi_flash_cmd_erase_config(&qspi_flash_cmd_config, sec_addr);//擦除结构体配置
  qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);//执行擦除操作

  /* wait command completed */
  while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
  qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);

  qspi_flash_busy_check();//判忙
}

/**
  * @brief   qspi_flash cmd read config
  * @param   qspi_cmd_struct: the pointer for qspi_cmd_type parameter
  * @param   addr: read start address
  * @param   counter: read data counter
  * @retval none
  */
```

```
void qspi_flash_cmd_read_config(qspi_cmd_type *qspi_cmd_struct, uint32_t addr, uint32_t counter)
{
  qspi_cmd_struct->pe_mode_enable = FALSE;//性能增强模式关闭
  qspi_cmd_struct->pe_mode_operate_code = 0;
  qspi_cmd_struct->instruction_code = 0xEB;//四线读指令
  qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE;//命令长度
  qspi_cmd_struct->address_code = addr;//读取地址
  qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_BYTE;//地址长度 3 字节
  qspi_cmd_struct->data_counter = counter;//读取数据长度
  qspi_cmd_struct->second_dummy_cycle_num = 6;//第二个虚拟状态周期数 6
  qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_144;//四线输入输出模式
  qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO;//硬件读取
  qspi_cmd_struct->read_status_enable = FALSE;
  qspi_cmd_struct->write_data_enable = FALSE;
}
/**
  * @brief   qspi flash read data
  * @param   addr: the address for read
  * @param   total_len: the length for read
  * @param   buf: the pointer for read data
  * @retval none
  */
void qspi_flash_data_read(uint32_t addr, uint8_t* buf, uint32_t total_len)
{
  uint32_t i, len = total_len;

  qspi_flash_cmd_read_config(&qspi_flash_cmd_config, addr, total_len);
  qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

  /* read data */
  do{
    if(total_len >= QSPI_FLASH_FIFO_DEPTH)
    {
      len = QSPI_FLASH_FIFO_DEPTH;
    }else{
      len = total_len;
    }
    while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_RXFIFORDY_FLAG) == RESET);
    for(i = 0; i < len; ++i)
    {
      *buf++ = qspi_byte_read(QSPI_FLASH_QSPIx);
    }
    total_len -= len;
  }while(total_len);
  /* wait command completed */
```

```
  while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
  qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
}
/**
  * @brief   qspi flash write data
  * @param   addr: the address for write
  * @param   total_len: the length for write
  * @param   buf: the pointer for write data
  * @retval none
  */
void qspi_flash_data_write(uint32_t addr, uint8_t* buf, uint32_t total_len)
{
  uint32_t i, len;

  do{
    qspi_flash_write_enable();
    /* send up to 256 bytes at one time, and only one page */
    len = (addr / QSPI_FLASH_PAGE_SIZE + 1) * QSPI_FLASH_PAGE_SIZE - addr;
    if(total_len < len)
      len = total_len;

    qspi_flash_cmd_write_config(&qspi_flash_cmd_config, addr, len);
    qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

    for(i = 0; i < len; ++i)
    {
      while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_TXFIFORDY_FLAG) == RESET);
      qspi_byte_write(QSPI_FLASH_QSPIx, *buf++);
    }
    total_len -= len;
    addr += len;

    /* wait command completed */
    while(qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
    qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);

    qspi_flash_busy_check();

  }while(total_len);
}
```

8 电容触摸模块

8.1 引脚介绍

电容模块 IC 是用 BL6133，这个模块是用 IIC 通信，还有一个中断引脚，如有触摸按下中断引脚会有电平变化。图 7-4 是 BL6133 与主控连接图。

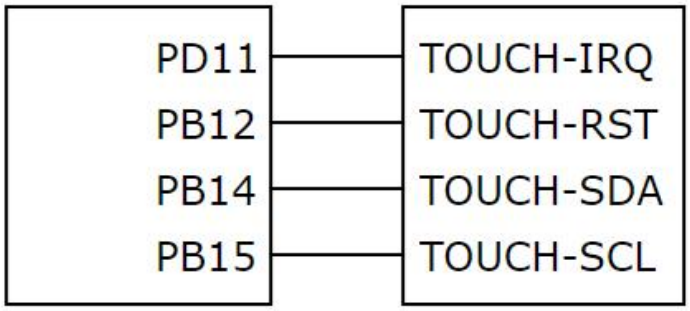


图 7-4

8.2 寄存器介绍

图 7-5 是介绍本次用到的寄存器功能。

寄存器	功能
0xAA	芯片ID
0x03	x轴坐标值高4bit
0x04	x轴坐标值低8bit
0x05	y轴坐标值高4bit
0x06	y轴坐标值低8bit

图 7-5

8.3 电容触摸驱动程序

```
#define BL6133_ADDR             0x2C    // 设备地址 7bit
#define BL6133_Chip_ID          0xE7    // chip ID
#define BL6133_Gesture          0x01    // Gesture Code
#define BL6133_Point_Numb       0x02    // Point Number
#define BL6133_Pos_XH           0x03    // X 高 6bit
```

```
#define BL6133_Pos_XL            0x04     // X 低 8bit
#define BL6133_Pos_YH            0x05     // Y 高 4bit
#define BL6133_Pos_YL            0x06     // Y 低 8bit

#define BL6133_X_OFFSET     (0)
#define BL6133_Y_OFFSET     (0)

/* static functions ---------------------------------------------------*/
/**
 * @brief write data to register
 * @param reg register address
 * @param buf need write data
 * @param len need write lenth
 * @return true write OK
 * @return false write err
 */
static bool _touch_write_reg(uint8_t ic_add , uint8_t reg, uint8_t *buf , uint32_t len)
{
    uint32_t i ;

    // start
    IIC_Start();

    // send addr
    IIC_Send_Byte( ic_add << 1 );
    if(IIC_Wait_Ack()){
        goto ret_fail;
    }

    // send reg
    IIC_Send_Byte( reg );
    if(IIC_Wait_Ack()){
        goto ret_fail;
    }

    // write data
    for(i = 0 ; i < len ; i++){
        IIC_Send_Byte( buf[i] );
        if(IIC_Wait_Ack()){
            goto ret_fail;
        }
    }

    // stop
    IIC_Stop();
```

```
    return true;

ret_fail:
    return false;
}
/**
* @brief read data from cst836u register
* @param reg register address
* @param buf read data buff
* @param len need read len
* @return true read OK
* @return false read err
*/
static bool _touch_read_reg(uint8_t ic_add , uint8_t reg, uint8_t *buf , uint32_t len)
{
    uint32_t i = 0;

    // start
    IIC_Start();

    // send addr
    IIC_Send_Byte( ic_add << 1 );
    if(IIC_Wait_Ack()){
        goto ret_fail;
    }
    // send reg
    IIC_Send_Byte( reg );
    if(IIC_Wait_Ack()){
        goto ret_fail;
    }
    IIC_Stop();
    IIC_Start();
    // read
    IIC_Send_Byte((ic_add << 1) | 0x01);
    if(IIC_Wait_Ack()){
        goto ret_fail;
    }

    // read data
    for(i = 0 ; i < len-1 ; i++){
        buf[i] = IIC_Read_Byte(1);
    }
    buf[i] = IIC_Read_Byte(0);

    // stop
```

```
    IIC_Stop();
    return true;

ret_fail:
    return false;
}
/**
* @brief   get one point data from cst836u
* @return true read ok
* @return false read err
*/
static bool bl6133_read_point(void)
{
    uint8_t temp[6];
    uint16_t x = 0xFFFF, y = 0xFFFF ;

    if(NULL == tp_dev_bl6133 || NULL == tp_dev_bl6133->read_reg){
        return false;
    }
    //读寄存器 读取 6 个字节
    tp_dev_bl6133->read_reg( BL6133_ADDR , BL6133_Gesture, &temp[0] , 6);

    x=(uint16_t)((temp[2]&0x3F)<<8)|temp[3];//高 6 位|低 8 位
    y=(uint16_t)((temp[4]&0x0F)<<8)|temp[5];//高 4 位|低 8 位

    tp_dev_bl6133->point.status = temp[2] >> 6 ;

    tp_dev_bl6133->point.x = x + BL6133_X_OFFSET;
    tp_dev_bl6133->point.y = y + BL6133_Y_OFFSET;
    return true;
}
/**
* @brief read chipID
*/
static bool bl6133_read_chipID(void)
{
    uint8_t chip_id[2] = {0};

    if(NULL == tp_dev_bl6133 || NULL == tp_dev_bl6133->read_reg || NULL == tp_dev_bl6133->write_reg){
        return false;
    }

    chip_id[0] = ~0x0A;
    chip_id[1] = BL6133_Chip_ID;
```

```
    if(!tp_dev_bl6133->write_reg(BL6133_ADDR , BL6133_Chip_ID , &chip_id[0] , 2))
    {
        return false;
    }

    chip_id[0] = 0x00;
    chip_id[1] = 0x00;
    if(!tp_dev_bl6133->read_reg( BL6133_ADDR , BL6133_Chip_ID, &chip_id[0] , 1))
    {
        return false;
    }

    tp_dev_bl6133->chipID = chip_id[1] << 8 | chip_id[0];
    return true;
}
/**
* @brief init TP
* @param dev
*/
void bl6133_init(tp_dev_t *dev)
{
    if(NULL == dev){
        return ;
    }

    dev->read_point = bl6133_read_point;//函数指针指向 read point 函数
    tp_dev_bl6133 = dev;

    bl6133_read_chipID();//读 ID
}
/**
 * @brief init touch pin and init cst836u tp ic
 */
void touch_init(void)
{
    gpio_init_type gpio_init_struct;
    exint_init_type exint_init_struct;

    crm_periph_clock_enable(TOUCH_RST_GPIO_CRM_CLK, TRUE);
    // irq
    crm_periph_clock_enable(CRM_SCFG_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(TOUCH_IRQ_GPIO_CRM_CLK, TRUE);

    scfg_exint_line_config(TOUCH_IRQ_GPIO, TOUCH_IRQ_PIN);
```

```
    exint_default_para_init(&exint_init_struct);
    exint_init_struct.line_enable = TRUE;
    exint_init_struct.line_mode = EXINT_LINE_INTERRUPUT;
    exint_init_struct.line_select = TOUCH_IRQ_EXINT_LINE;
    exint_init_struct.line_polarity = EXINT_TRIGGER_FALLING_EDGE;
    exint_init(&exint_init_struct);
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
    nvic_irq_enable(TOUCH_IRQ_EXINT_IRQn, 1, 0);

    /* configure the rst gpio */
    gpio_default_para_init(&gpio_init_struct);
    gpio_init_struct.gpio_pins = TOUCH_RST_PIN ;
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_UP;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(TOUCH_RST_GPIO, &gpio_init_struct);

    // iic io
    IIC_Init();
    // rst set 1
    TOUCH_RST_SET(1);
    delay_ms(5);
    TOUCH_RST_SET(0);
    delay_ms(5);
    TOUCH_RST_SET(1);
    delay_ms(100);
    //tp ic init
    tp_dev.write_reg = _touch_write_reg;
    tp_dev.read_reg = _touch_read_reg;
    bl6133_init(&tp_dev);
}
//中断读取坐标值
void TOUCH_IRQ_EXINT_IRQHandler(void)
{
    if(exint_flag_get(TOUCH_IRQ_EXINT_LINE) != RESET)
    {
        if(NULL != tp_dev.read_point){
            tp_dev.read_point();
        }

        exint_flag_clear(TOUCH_IRQ_EXINT_LINE);
    }
}
```

9 GUI 工具使用-HMI

9.1 GUI-HMI 概述

GT-HMI 是高通专为 GUI 用户开发的界面设计和编辑工具，包含 GT-HMI Designer 和 GT-HMI Engine 两款软件，上位机 GT-HMI Designer 自带图形库并包含多种控件供开发者使用，可以帮助用户快速创作出富有创意的 UI 图形交互效果。下位机 GT-HMI Engine 是一款高效的嵌入式 UI 引擎，可以帮助用户更快的构建、集成和优化 UI 应用程序；GT-HMI 工具为开发者提供了创新、高效的解决方案，让用户轻松实现自己的图形界面想法，有助于用户节省大量开发时间，提升开发效率，降低开发成本，从而增强产品的竞争力和用户体验，是开发者在使用 GUI 时的第一选择。

9.2 GT-GUI LCD 模组与 HMI 工具使用

GT-GUI LCD 1.9 寸模组配合我司 GUI-LCD 开发板和 GT-HMI Designer 上位机设计界面使用，可以直接在我们移植好的示例工程上进行开发，提高开发效率。也可以使用 GT-GUI LCD 1.9 寸模组配合客户的自己的单片机进行开发，后面介绍 HMI 界面移植。

获取 GUI-HMI 工具的方式及视频教程：

1. 软件下载链接：高通字库官方网站高通字库-首页 (gaotongfont.cn)
2. 说明书下载： 高通字库-GT-HMI Designer 用户手册 (gaotongfont.cn)
3. 软件视频教程：OPEN_GT 的个人空间_哔哩哔哩_bilibili

9.3 GT-HMI Designer 上位机代码移植

下位机配合 GT-HMI Designer 上位机设计界面非常快，本小节讲解下位机与上位机配合使用。

第一步：首先在 GT-HMI Designer 上新建工程设计界面，设计界面教程可参考“GT-HMI Designer 用户手册”。

第二步：GT-HMI Designer 工程目录下的 screen 目录是每个界面的程序源码，需要把这部分源码添加到 keil5 工程上，在 main 函数里面初始化 gt_ui_init()函数接口。这里还没完，还需要将 board 目录的 gt_port_vf.c，gt_gui_driver.lib 和 gt_gui_driver.h 替换工程 GT-HMI-Engine\driver 下的同名文件。

第三步：将 board 目录下的 resource.bin 烧录到存储芯片里面去。

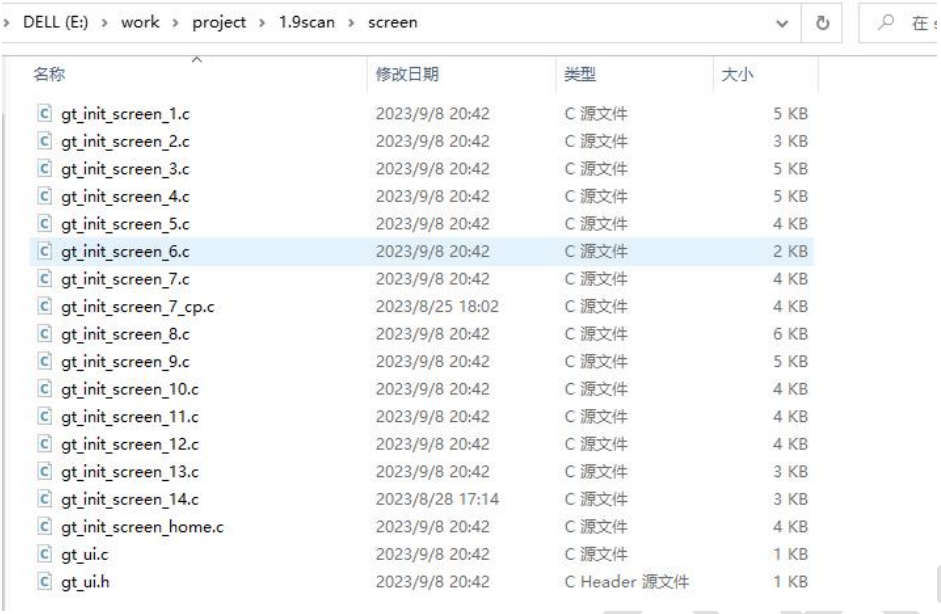


图 9-1 screen 目录下的文件

图 9-2 board 目录下的文件

烧录 bin 文件步骤如下：

第一步：备好如下烧录器，在烧录接的是最下面四行，图 9-3 右边表格表示烧录器接存储芯片的引脚号。

1	8
2	7
3	6
4	5

图 9-3

第二步：将烧录器和存储芯片引脚连接起来，图 9-4 为开发板上的存储芯片与预留座子图，右边表格

表示座子对应芯片的引脚号，若使用型号为搭载了 GUI 芯片的 1.9 寸液晶模组时，请连接在原理图设计及 PCB 布线时预留的 8PIN 接口。

1	2
3	4
5	6
7	8

图 9-4

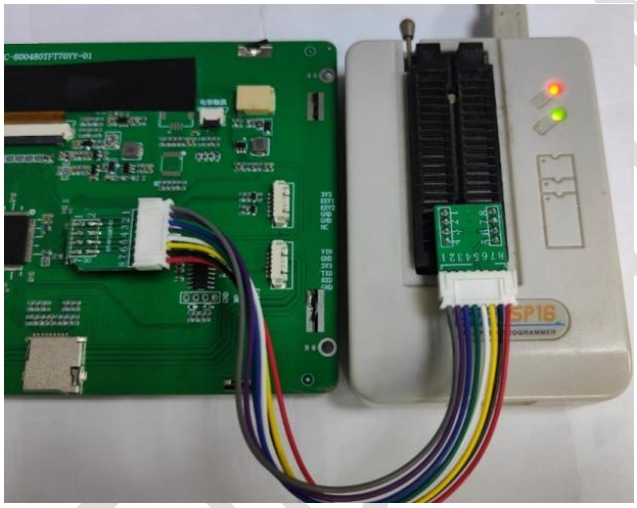


图 9-5 存储芯片与烧录器连接图

第三步：打开 FlyPRO 烧录软件，选择 FlyPRO 上方的芯片-检测芯片型号。

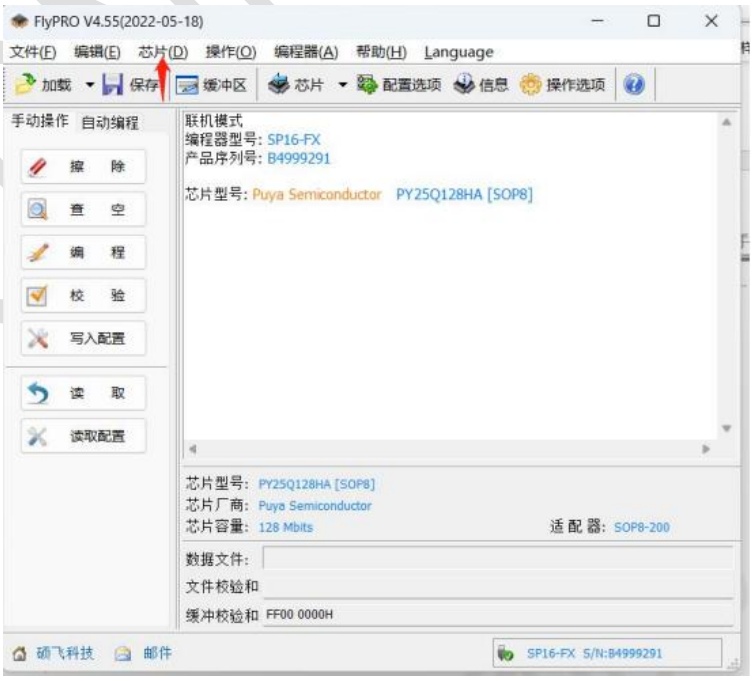


图 9-6

第四步：点击 FlyPRO 软件界面上方的加载，将所需要烧入的 bin 文件加载进去。

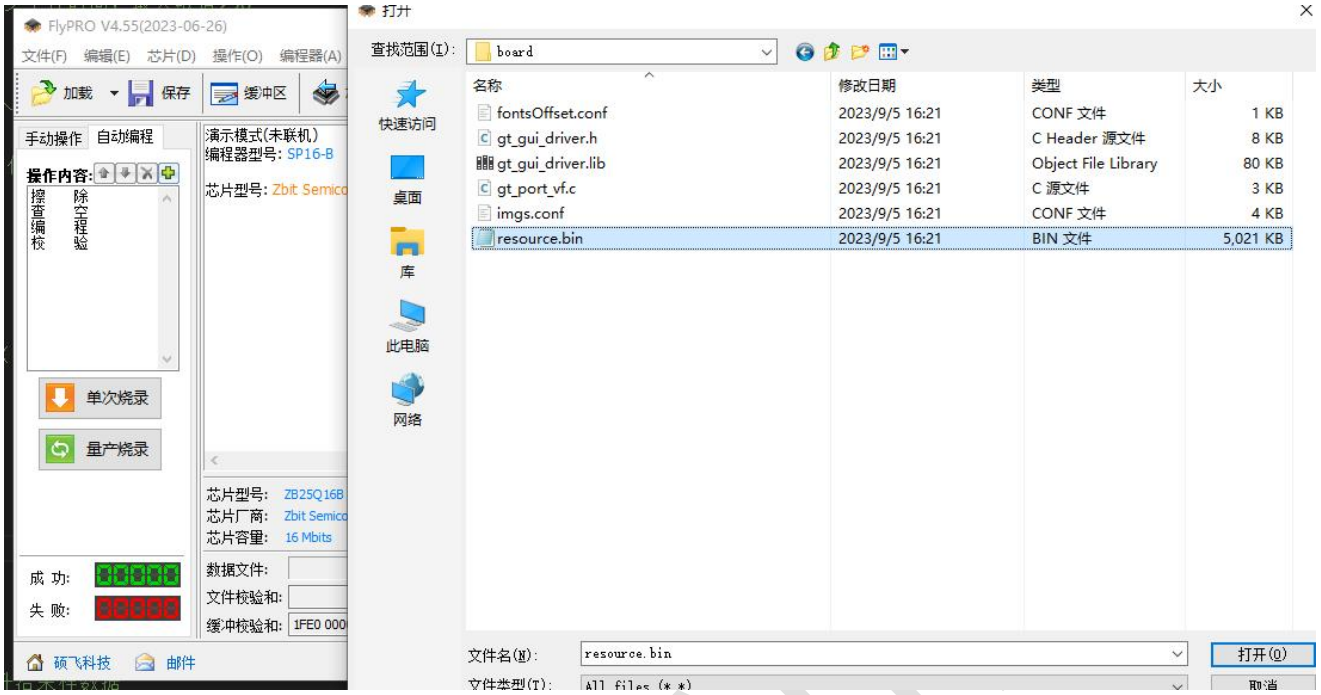


图 9-7

第五步：点击烧录软件的自动编程，选择单次烧录。等待程序将 bin 文件烧录进 flash。（这个过程必须全程按住 RST 按钮，否则烧录失败），图 9-8 为成功烧录的界面。

图 9-8

9.4 GT-HMI-Engine 代码移植

GT-HMI-Engine 代码移植很重要，在 hmi 上位机设计的界面需要依赖这些文件才能使用，下面介绍移植步骤（下面介绍的步骤是以雅特力的 AT32F437VGT7 为例，在 keil5 环境下完成的，其他芯片移植过程基本一致）。

第一步：首先用 git 的下载 GT-HMI-Engine 源代码。

在 git bash 上使用

"git clone git@gitee.com:genitop/GT-HMI-Engine.git -b develop"命令拉取代码。

```
$ git clone git@gitee.com:genitop/GT-HMI-Engine.git -b develop
Cloning into 'GT-HMI-Engine'...
remote: Enumerating objects: 4939, done.
remote: Counting objects: 100% (4939/4939), done.
remote: Compressing objects: 100% (3385/3385), done.
remote: Total 4939 (delta 3981), reused 1921 (delta 1546), pack-reused 0
Receiving objects: 100% (4939/4939), 1015.83 KiB | 5.58 MiB/s, done.
Resolving deltas: 100% (3981/3981), done.
```

图 9-9

第二步：将 GT-HMI-Engine 源码添加到功能目录里面去，并添加到 keil5 的工程里面去，

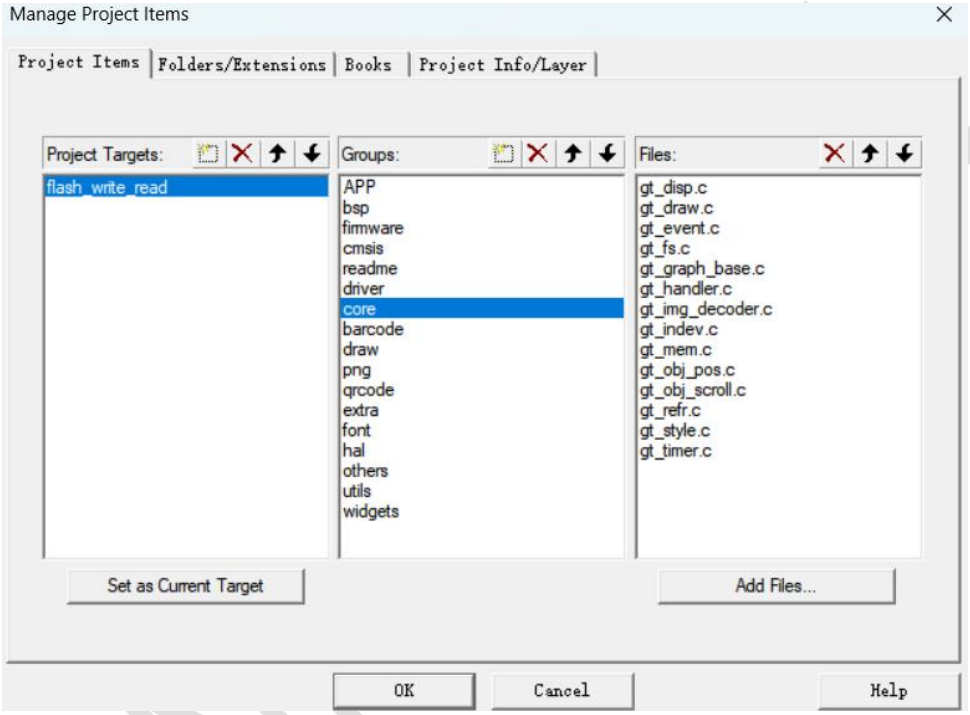


图 9-10

第三步：打开 gnu 配置进行编译，图 9-11 为参考的配置，不同的 keil5 版本配置 gnu 界面都不一样。这里 gnu 配置必须打开，不然会报很多错误。

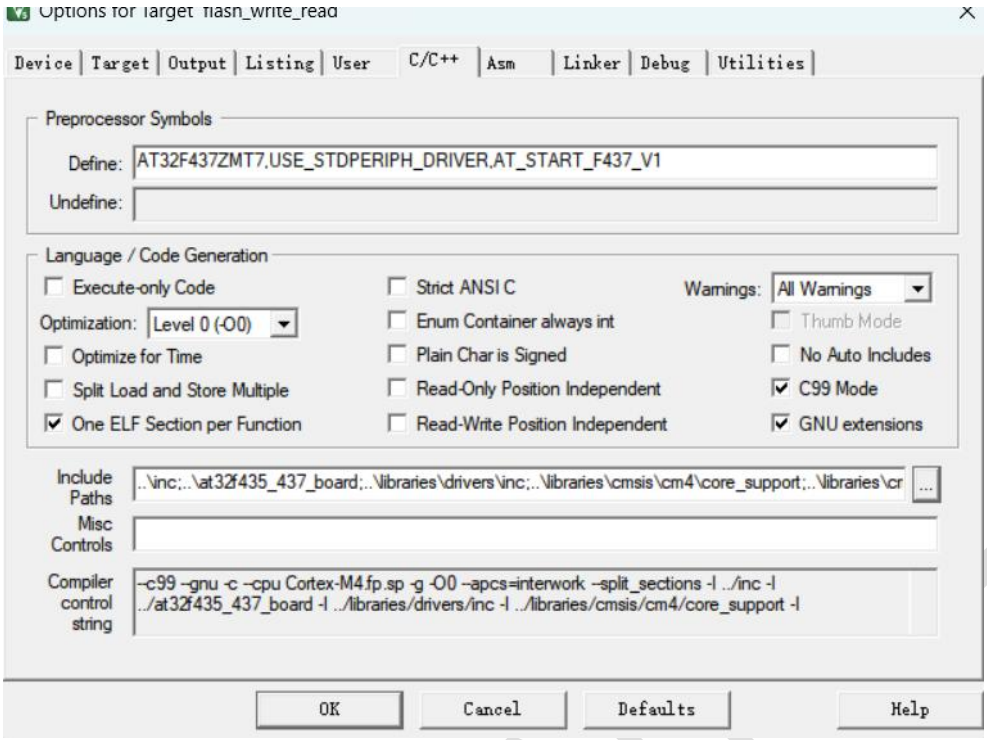


图 9-11

图 7.4 为 Target 配置，Use MicroLIB 也需要打开，不然也会报错。

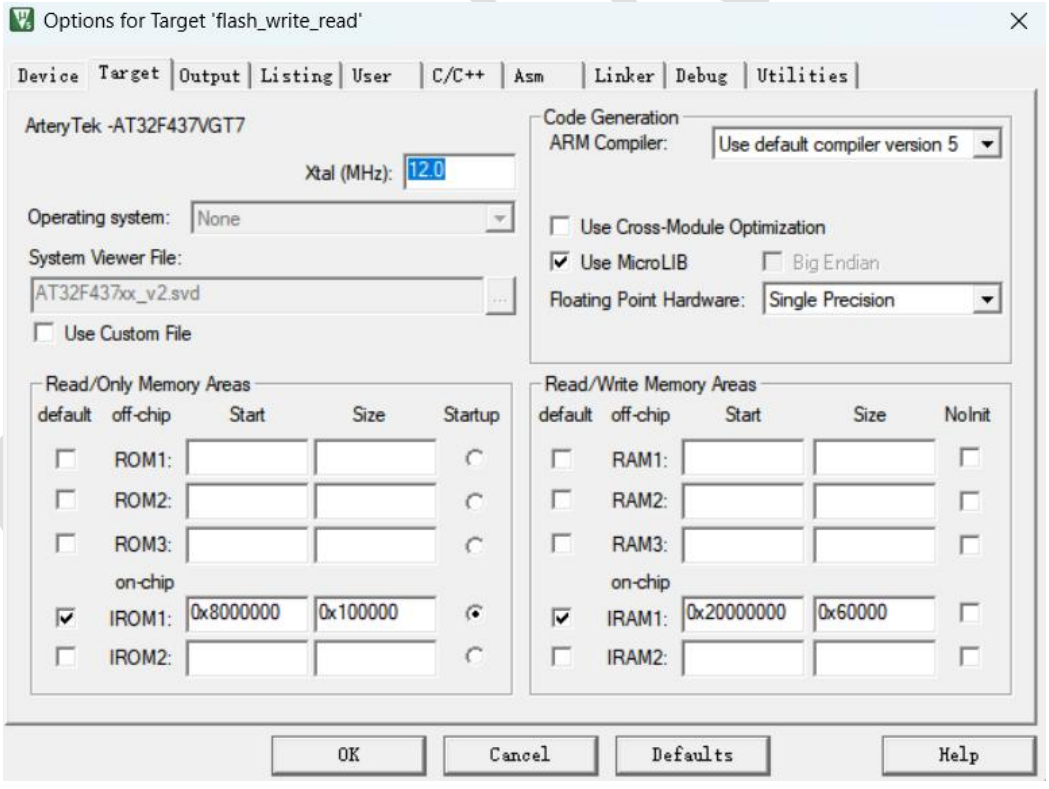


图 9-12

第四步：编译通过之后添加 spi_wr、_flush_cb、read_cb、read_cb_btn 四个函数，spi_wr 是读取素材图片的，支持 flash、SD 卡读取，_flush_cb 是刷屏函数接口，read_cb 是触摸上报接口，read_cb_btn 是按键上报接口。read_cb_btn 接口根据客户的要求添加，如果用不到，可定义空函数，就是这函数里面什么也没有，防止编译报错。

第五步：初始化 gt_init 函数，while(1)循环添加如下图操作，gt_tick_inc 为 GUI 系统提供 1ms 的

心跳。

```
while(1)
{
    gt_tick_inc(1);
    gt_task_handler();
    delay_ms(1);
}
```

图 9-13

做完上面操作可显示一个按键控件看是否能正常显示，如显示不正常请检查_flush_cb 刷屏函数是否正确。

控件能正常使用并能正常触摸，表明 GT-HMI-Engine 已移植成功。

9.5 接口函数代码

```
uint32_t spi_wr(uint8_t * data_write, uint32_t len_write, uint8_t * data_read, uint32_t len_read)
{
    unsigned long ReadAddr;
    unsigned long addr, len;
    ReadAddr = *(data_write + 1) << 16;                 //高八位地址
    ReadAddr += *(data_write + 2) << 8;                 //中八位地址
    ReadAddr += *(data_write + 3);                      //低八位地址

    spiflash_read(data_read,ReadAddr,len_read);

    return 1;
}

void _flush_cb(struct _gt_disp_drv_s * drv, gt_area_st * area, gt_color_t * color) {
    gt_size_t x=area->x,y=area->y;
    uint16_t w = area->w,h = area->h;
    int i=0;

    lcd_setblock(x,y,x+w-1,y+h-1);
    for(i=0;i<w*h;i++)
    {
        lcd_writeonepoint(color->full);
        color++;
    }

}

void read_cb(struct _gt_indev_drv_s * indev_drv, gt_indev_data_st * data) {
    if (!touch_status) {
        data->state = GT_INDEV_STATE_RELEASED;
```

```
        return;
    }
    touch_status = 0;
    data->point.x = tp_dev.point.x;
    data->point.y = tp_dev.point.y;
    data->state = GT_INDEV_STATE_PRESSED;
}
```

注：read_cb_btn 接口函数，如有需求可联系我司技术人员提供。

9.6 HMI 示例移植

GT-HMI-Engine 移植成功后，就可以实现将 HMI 自行设计的界面移植到板子上，移植过程查看 GT-HMI Designer 上位机代码移植章节或参考“GT-HMI Engine 用户手册”，下面讲解一下 HMI 示例移植，以 GT-HMI Designer 软件示例界面中的 1.9 寸示例来说明移植过程。首先打开 1.9 寸示例工程。

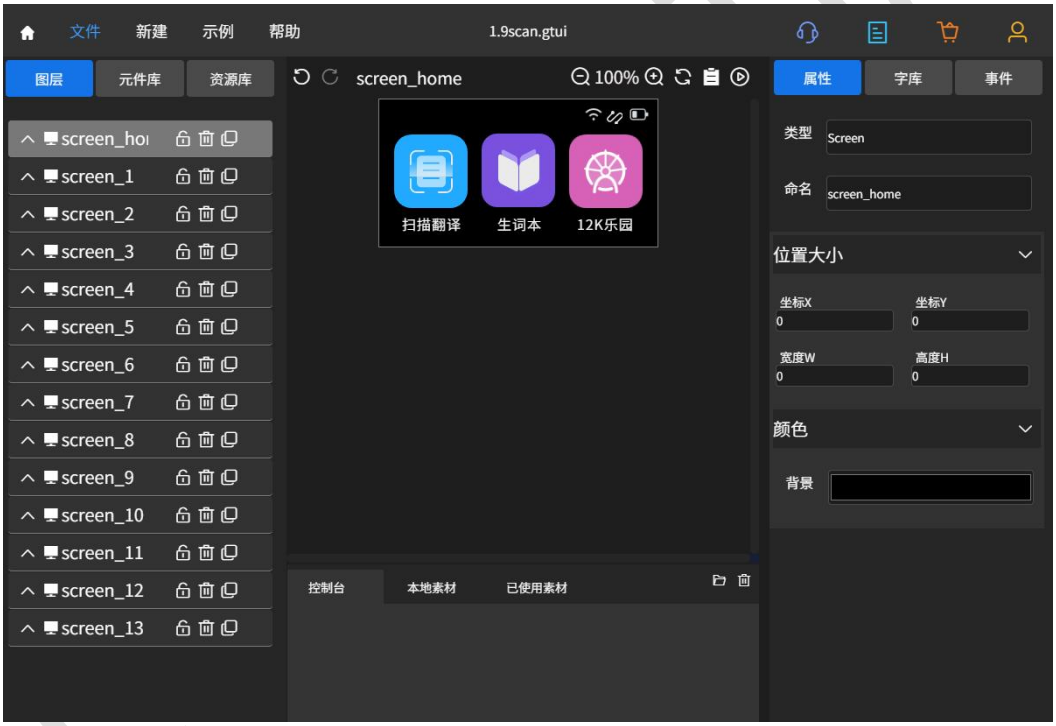


图 9-14

可以看到该示例中使用并展示了多种控件的使用和交互方法，假如使用软件中的示例工程，我们在点击仿真运行前，需要先另存为到其它路径。

不勾选下次询问时，点击仿真会弹出编译环境设置，我们设置好开发板卡设置与字库配置。

图 9-15

可以看到仿真完成后，软件控制台打印出仿真编译产物的路径信息，路径为另存为时设置的工程路径。

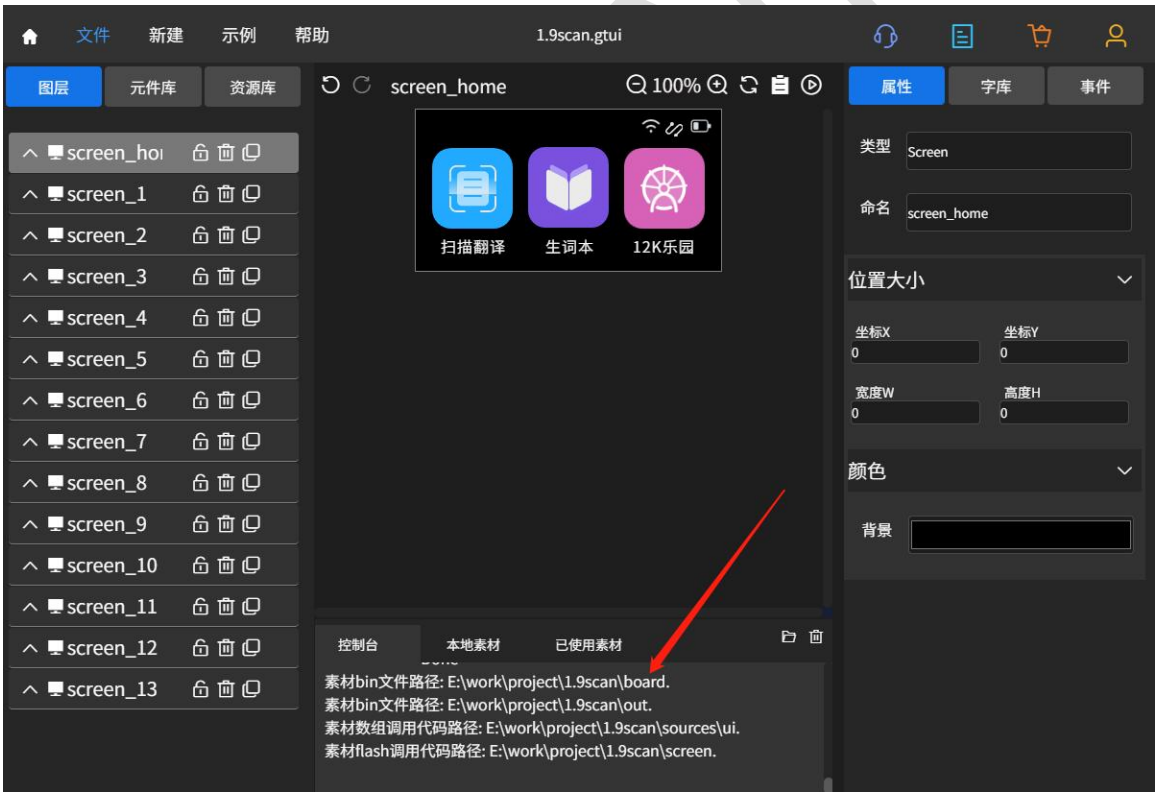


图 9-16

我们打开工程目录，可以看到有多个子文件夹，其中 board 文件夹是提供给 GT-HMI 模块使用的，out 文件夹则是提供给非 GT-HMI 模块使用的，两者都是资源文件，包括生成的素材 bin 文件，图片排布顺序以及字库调用库等。Keil5 文件夹中是编译自动生成的对应 GT-HMI 模块 keil 工程（与编译环境设置中开发板设置对应，即 1.9 寸模块工程），screen 文件夹则是工程各页面的调用代码，sources 文件夹中是个图片素材及图片的数组调用代码。

图 9-17

我们按照 9.3 章的内容将非 GT-HMI 模块使用的 out 文件夹中的 gt_port_vf.c，gt_gui_driver.lib 和 gt_gui_driver.h 替换掉我们自身移植好 Engine 工程的 GT-HMI-Engine\driver 下的同名文件，点击 keil5 工程的编译按钮开始编译程序文件，编译完成后使用 J-LINK 与模块板子相连，点击右边的下载按钮，将程序代码下载到板子中。

图 9-18

打开 out 文件夹中的 resource.bin 资源文件，使用烧录器将其烧录到板子上的 flash 中。烧录完成后，即可在板子上运行。

图 9-19

10 注意事项

1.请勿拆卸液晶显示模块。

2.不要在印制电路板上钻额外的孔 ， 修改形状或更改印制线路板上元件的位置。

3.除焊接接口外 ， 不要用烙铁做任何更改；焊接温度保证在 320°C-350°C ， 焊接时间控制在 10S 以内 ， 焊接时注意不要在同一处停留时间太久以免烫伤 FPC。

4.其他事项在不清楚使用之前 ， 请联系我司人员指导进行。

11 联系信息

深圳高通半导体有限公司
地址：深圳市福田区车公庙泰然九路金润大厦 12C
电话： 0755-83453881 83453855
技术支持： www.hmi.gaotongfont.cn
Call：0755-83453881

QQ 技术频道：

企业微信客服：