高通汉显®

GTHMI

GT-GUI LCD 7.0 寸液晶模组

GT-GL800480T70-XXXX

数据手册

V1.1

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， VA 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 显示屏具体参数：**电气特性、光学参数……；
4) **LCD 屏接口原理图设计规格：**
   参考规格书的目录 4 接口定义部分、目录 9 屏模块->RGB888 通信->引脚介绍部分、 目录 10 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 RGB 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 RGB888 屏驱动代码。参考目录 4 接口定义部分、目录 9 屏模块->RGB 通信；
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 9 屏模块->RGB888 驱动程序；
3) **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 12GUI 工具使用-HMI；
4) **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 11 电容触摸模块下面的驱动例程；
5) **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 10 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	2023/08/30
V1.1	Modify module drawings	2024/01/30

目录

1.产品型号

GT- GL　800480　T/O　+70 —— 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-GL800480T70-S0GN
+GUI 芯片	GT-GL800480T70-S0GN128
+电容触摸	GT-GL800480T70-S0GC
+电容触摸+GUI 芯片	GT-GL800480T70-S0GC128

2.概述

GT-GUI LCD 7 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 7 寸显示屏，分辨率为 800*480，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 ILI6122+ILI5960，支持 RGB565/888,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	7.0 inch(Diagonal)	Inch
2	Resolution	800*3(RGB)*480	Dots
3	Active Area	154.08(H) ×85.92 (V)	mm
4	Dimensional Outline（不含 TP）	164.9(H)×100(V)	mm
5	Number of Pixels	800 (H)×480 (V)	pixels
6	Cell gap	3.13 ± 0.23	um
7	Pixel arrangement	RGB- island	
8	Dot pitch	0.0642(W) × 0.1790(H) mm	mm
9	View direction (Gray inversion)	6 o’clock	
10	Weight	TBD	gram
11	display driver IC	PEILI6122_SPEC+ILI5960_SC	
12	touch IC	GT911	
13	GUI IC	128	Mb
14	Back Light	White LED	
15	Screen backlight voltage	8.4-10.2	V
16	Screen backlight current	170-200	mA
17	Screen communication mode	RGB	
18	Flash communication mode	SPI/QSPI	
19	operating temperature	-20- +70℃	℃
20	Storage Temperature	-30- +80℃	℃

4.接口定义

Pin ON	Symbol	Function
1	WP(IO2)	Write Protect lnput(Data lnput Output2) ★
2	DO(IO1)	Sata Output lnput(Data lnput Output1) ★
3	CS#	Flash select ★
4	3V3	Powet Supply ★
5	HOLD(IO3)	Hold lnput(Data lnput Output3) ★
6	GND	Power ground
7	CLK	Serial clock lnput ★
8	GND	Power ground
9	DI(IO0)	Data lnput(Data lnput Output0) ★
10	CSX	NC
11	SCL	NC
12	SDA	NC
13	GND	Power ground
14	DITHB	Dithering function.
15	VCOM	Common voltage.
16	STBYB	Chip selection pin（Low enable.； High disable.）
17	RESET	Global reset pin.
18	AVDD	Power for Analog Circuit.
19	VGL	Gate OFF Voltage.
20	VGH	Gate ON Voltage.
21	UPDN	Up / Down selection.
22	SHLR	Left / right selection.
23	GND	Power ground
24	DCLK	Sample clock.
25	GND	Power ground
26	R0	Red data
27	R1	Red data
28	R2	Red data
29	R3	Red data
30	R4	Red data
31	R5	Red data
32	R6	Red data
33	R7	Red data
34	G0	Green data
35	G1	Green data

36	G2	Green data
37	G3	Green data
38	G4	Green data
39	G5	Green data
40	G6	Green data
41	G7	Green data
42	B0	Blue data
43	B1	Blue data
44	B2	Blue data
45	B3	Blue data
46	B4	Blue data
47	B5	Blue data
48	B6	Blue data
49	B7	Blue data
50	HSYNC	Horizontal Sync Input
51	VSYNC	Vertical Sync Input.
52	DE	Data Input Enable.
53	MODE	DE/SYNC mode select
54	VDD	Power for Digital Circuit
55	VCOM	Common voltage.
56	GND	Power ground
57	VLED-	Ground (Cathode).
58	VLED-	Ground (Cathode).
59	VLED+	LED Input Terminal l(Anode).
60	VLED+	LED Input Terminal l(Anode).

★：字库芯片通讯用

5.电气特性

5.1 极限参数

Item	Symbol	Values		Unit	Remark
		Min.	Max.		
Power voltage	DV_{DD}	-0.3	5.0	V	
	AV_{DD}	6.5	13.5	V	
	V_{GH}	-0.3	40.0	V	
	V_{GL}	-20.0	0.3	V	
	V_{GH}-V_{GL}	-	40.0	V	
Operation Temperature	T_{OP}	-20	70	℃	
Storage Temperature	T_{ST}	-30	80	℃	
LED Reverse Voltage	V_R	-	1.2	V	Each LED Note 2
LED Forward Current	I_F	-	25	mA	Each LED

Note 1: The absolute maximum rating values of this product are not allowed to be exceeded at any times. Should a module be used with any of the absolute maximum ratings exceeded, the characteristics of the module may not be recovered, or in an extreme case, the module may be permanently destroyed.随时都不允许超过本产品的绝对最大值。如果使用的模块超过了任何绝对最大额定值，则该模块的特性可能无法恢复，或者在极端情况下，该模块可能会被永久破坏。

Note 2: VR　Conditions: Zener Diode 20mA　VR 条件：齐纳二极管 20 mA

5.2 驱动 TFT 液晶面板

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Power voltage	DV_{DD}	3.0	3.3	3.6	V	Note 2
	AV_{DD}	10.2	10.4	10.6	V	
	V_{GH}	15.3	16.0	16.7	V	
	V_{GL}	-7.7	-7.0	-6.3	V	
Input signal voltage	V_{COM}	3.6	3.8	4.0	V	
Input logic high voltage	V_{IH}	0.7 DV_{DD}	-	DV_{DD}	V	Note 3
Input logic low voltage	V_{IL}	0	-	0.3 DV_{DD}	V	

Note 1: Be sure to apply DVDD and VGL to the LCD first, and then apply VGH.确保先在 LCD 上应用 DVDD 和 VGL，然后再应用 VGH。

Note 2: DVDD setting should match the signals output voltage (refer to Note 3) of customer's system board. DVDD 设置应与客户系统板的信号输出电压（见注 3）相匹配。

Note 3: DCLK,HS,VS,RESET,U/D, L/R,DE,R0~R7,G0~G7,B0~B7,MODE,DITHB.

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Current for Driver	I_{GH}	-	0.2	1.0	mA	V_{GH} =16.0V
	I_{GL}	-	0.2	1.0	mA	V_{GL} = -7.0V
	IDV_{DD}	-	4.0	10	mA	DV_{DD} =3.3V
	IAV_{DD}	-	20	50	mA	AV_{DD} =10.4V

5.3 背光规格

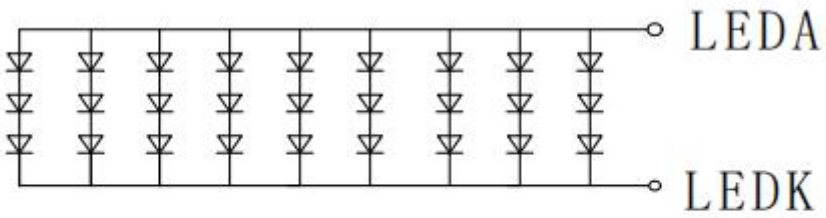
Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Voltage for LED backlight	V_L	8.4	9.3	10.2	V	Note 1
Current for LED backlight	I_L	170	180	200	mA	
LED life time	-	20,000	-	-	Hr	Note 2

Note 1: The LED Supply Voltage is defined by the number of LED at Ta=25℃ and IL =180mA.

LED 电源电压由 Ta=25℃和 IL =180mA 处的 LED 数量定义。

Note 2: The "LED life time" is defined as the module brightness decrease to 50% original brightness at Ta=25℃ and IL =180mA. The LED lifetime could be decreased if operating IL is lager than 180mA.

"LED 寿命"定义为在 Ta=25℃和 IL =180mA 时，模块亮度降低到原始亮度的 50%。如果工作 IL 小于 180 mA，LED 寿命可以降低。



LB:3S9P=27LEDs

IF=180mA, VF=8.4V~10.2V

6 光学参数

6.1 液晶模组光学特性

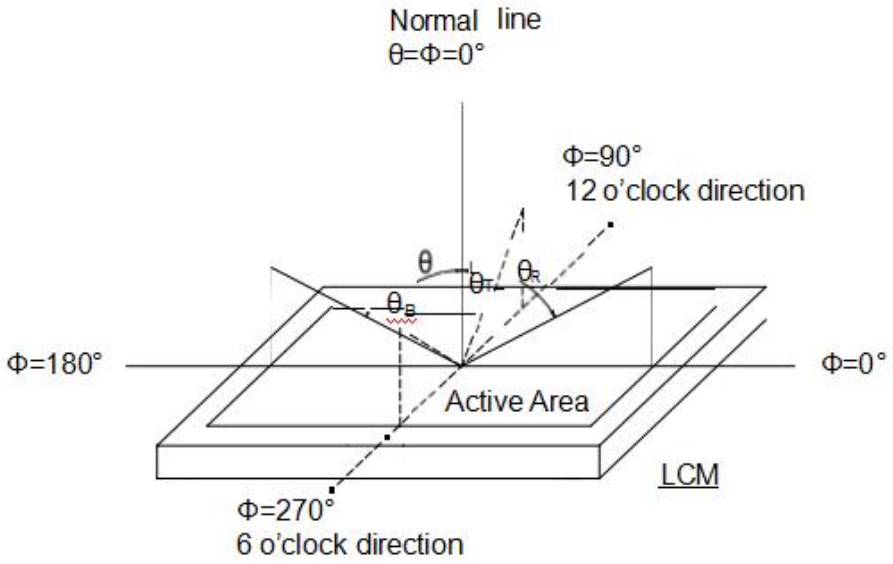
Ta=25ºC, DVDD=3.3V

Item	Symbol	Condition	Values			Unit	Remark
			Min.	Typ.	Max.		
Viewing angle (CR≥ 10)	θ_L	Φ=180° (9 o'clock)	60	70	-	degree	Note 1
	θ_R	Φ=0°(3 o'clock)	60	70	-		
	θ_T	Φ=90° (12 o'clock)	40	50	-		
	θ_B	Φ=270° (6 o'clock)	60	70	-		
Response time Rise+ Fall	T_{RT}	Normal θ=Φ=0°	-	25	50	msec	Note 3
Contrast ratio	CR		400	500	-	-	Note 4
Color chromaticity	R_X		Typ -0.03	0.31	Typ+ 0.03		Note 2 Note 5 Note 6
	R_Y			0.33			
	G_X			TBD			
	G_Y			TBD			
	B_X			TBD			
	B_Y			TBD			
	W_X			TBD			
	W_Y			TBD			
Luminance	L		200	250	-	cd/m²	Note 6
Uniformity	⊿Bp		70	75		%	

6.2 测量系统

6.2.1 LCM 观看角度及测量

Note 1: Definition of viewing angle range 视角范围的定义。

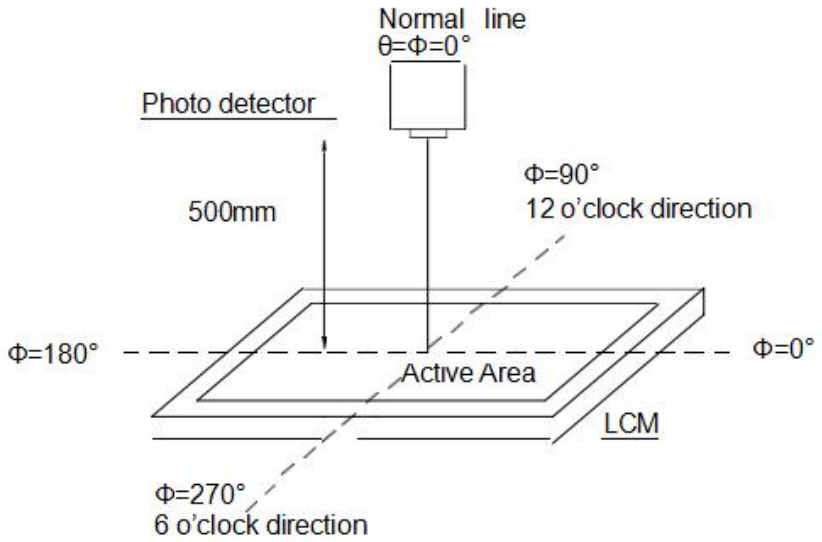


视角定义图

Note 2: Definition of optical measurement system.光学测量系统的定义。

The optical characteristics should be measured in dark room. After 30 minutes operation, the optical properties are measured at the center point of the LCD screen. (Viewing angle is measured by ELDIM-EZ contrast/Height :1.2mm,Response time is measured by Photo detector TOPCON BM-5A, other items are measured by BM-7A/Field of view: 1° /Height: 500mm.)

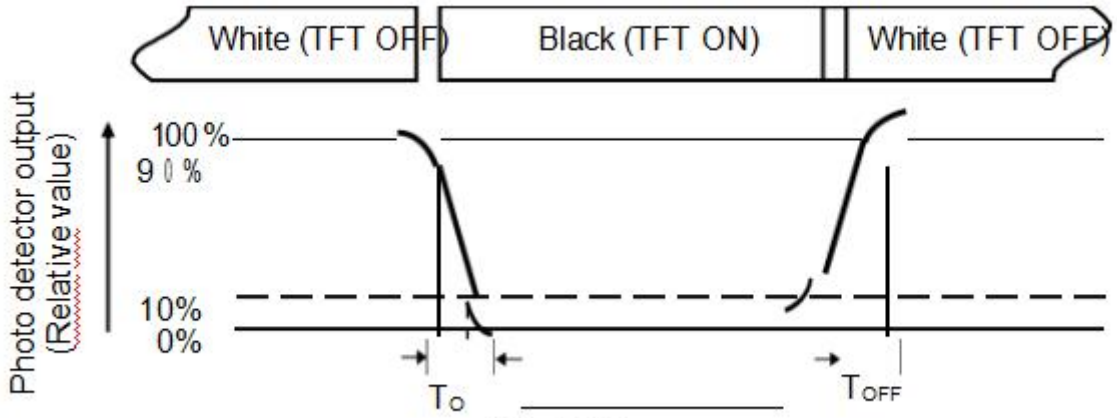
光学特性应在暗室中进行测量。操作 30 分钟后，在液晶屏幕的中心点测量光学性能。（视角采用 ELDIM-EZ 对比度/高度 1.2mm 测量，响应时间采用光探测器 TOPCON BM-5A 测量，其他项目采用 BM-7A/视场 1/高度 500mm 测量。）



6.2.2 反应时间

The response time is defined as the LCD optical switching time interval between “White” state and “ Black” state. Rise time (TON) is the time between photo detector output intensity changed from 90% to 10%. And fall time (TOFF) is the time between photo detector output intensity changed from 10% to 90%.

响应时间定义为“白”状态和“黑”状态之间的 LCD 光切换时间间隔。上升时间（TON）是指光电探测器输出强度从 90%变化到 10%之间的时间。下降时间（TOFF）是指光电探测器输出强度从 10%变化到 90%之间的时间。



响应时间图

6.2.3 对比度（CR）

Contrast Ratio (CR) is defined mathematically as:

$$\text{Contrast Ratio} = \frac{\text{Surface Lumi nance with all white pixels}}{\text{Surface Lumi nance with all black pixels}}$$

Surface lumi nance is the center point across the LCD surface 500mm from the surface with all pixels displaying white. （表面亮度是距离液晶显示器表面 500mm 的中心点，所有像素都显示为白色。）

Note 3: Definition of color chromaticity (CIE1931)
Color coordinates measured at center point of LCD.
在液晶显示器的中心点上测量的颜色坐标。

Note 4: All input terminals LCD panel must be ground while measuring the center area of the panel.在测量面板的中心面积时，所有输入端子液晶面板必须接地。

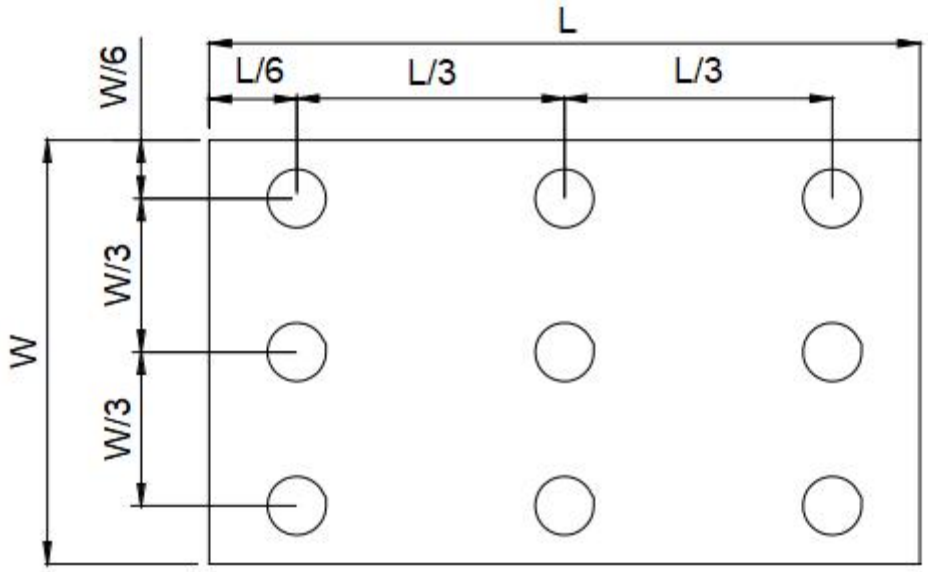
Note 5: Definition of Luminance Uniformity 亮度均匀性的定义
Active area is divided into 9 measuring areas (Refer to Fig. 4-4).Every

measuring point is placed at the center of each measuring area.活动区域分为 9 个测量区域（参见图 4-4)。每个测量点都位于每个测量区域的中心

$$Luminance\quad Uniformity\ (Yu) = \frac{B_{min}}{B_{max}}$$

W-------Active area length　　H----- Active area width

Bmax：所有测量位置的最大亮度。　Bmin：　所有测量位置测量的最小亮度。



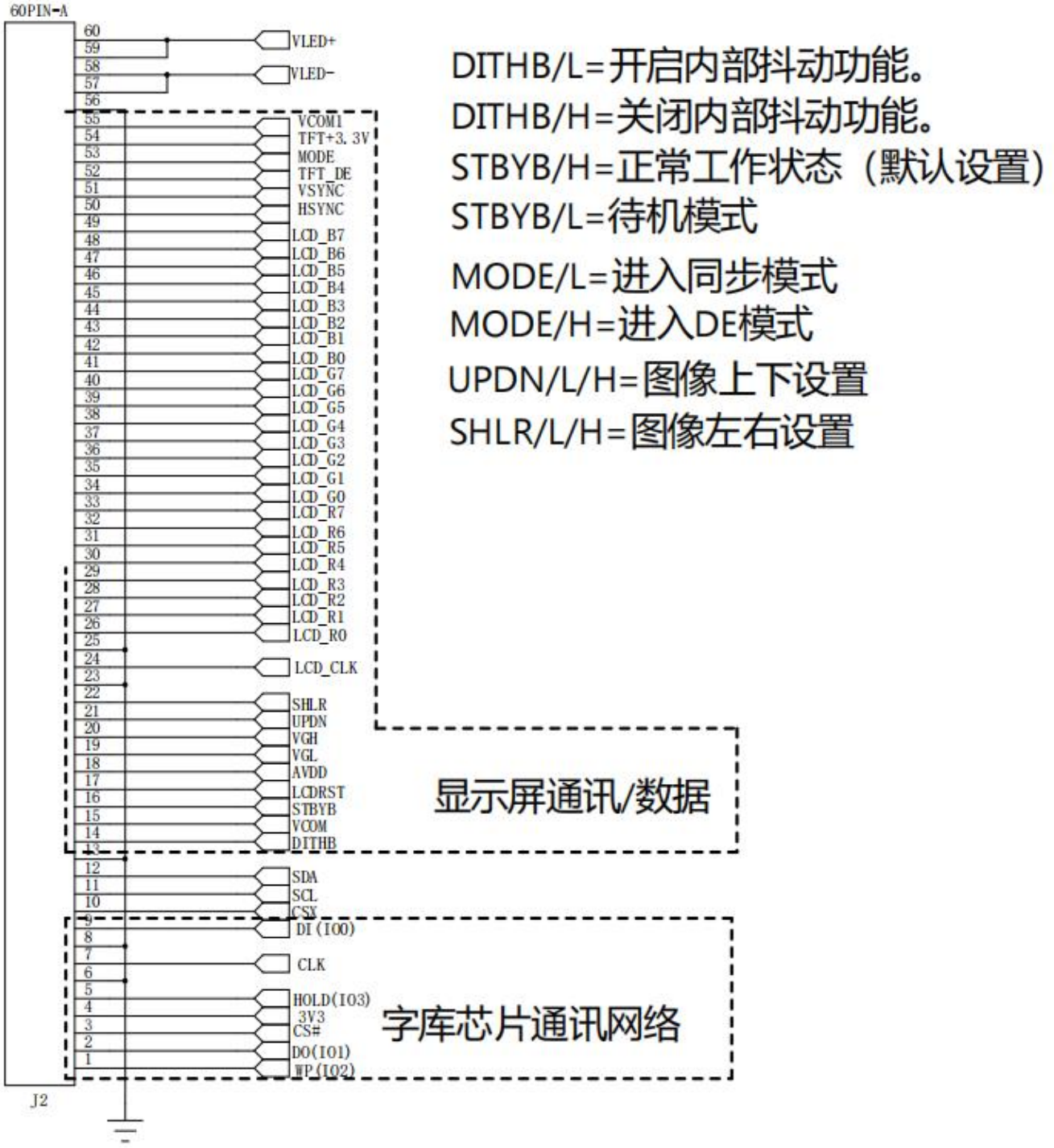
测量区域图

Note 8: Definition of Color of CIE Coordinate and NTSC Ratio.

注 8：CIE 坐标和 NTSC 比值的颜色定义。

$$S = \frac{\text{area of RGB triangle}}{\text{area of NTSC triangle}} \text{根} 100\%$$

7.FPC 原理图


60PIN-A
VLED+
VLED-
VCOM1
TFT+3.3V
MODE
TFT_DE
VSYNC
HSYNC
LCD_B7
LCD_B6
LCD_B5
LCD_B4
LCD_B3
LCD_B2
LCD_B1
LCD_B0
LCD_G7
LCD_G6
LCD_G5
LCD_G4
LCD_G3
LCD_G2
LCD_G1
LCD_G0
LCD_R7
LCD_R6
LCD_R5
LCD_R4
LCD_R3
LCD_R2
LCD_R1
LCD_R0
LCD_CLK
SHLR
UPDN
VGH
VGL
AVDD
LCDRST
STBYB
VCOM
DITHB
显示屏通讯/数据
SDA
SCL
CSX
DI(IO0)
CLK
HOLD(IO3)
3V3
CS#
DO(IO1)
WP(IO2)
字库芯片通讯网络
J2
DITHB/L=开启内部抖动功能。
DITHB/H=关闭内部抖动功能。
STBYB/H=正常工作状态（默认设置）
STBYB/L=待机模式
MODE/L=进入同步模式
MODE/H=进入DE模式
UPDN/L/H=图像上下设置
SHLR/L/H=图像左右设置

8.模组图纸

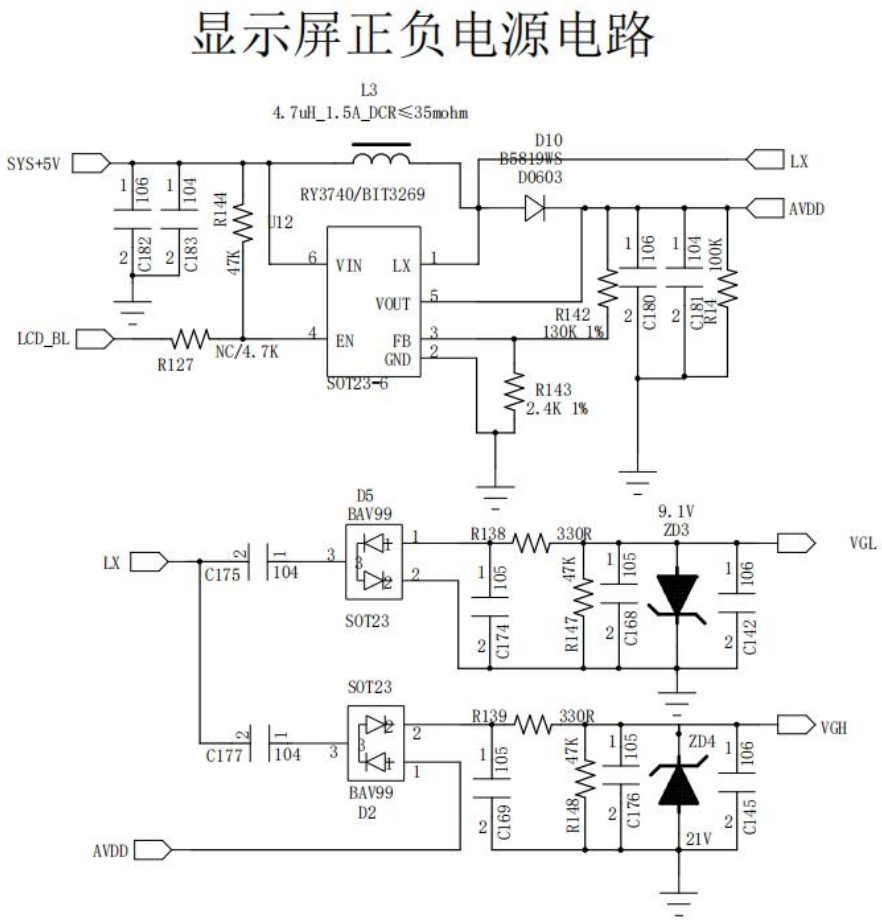
8.1 无 TP 模组图

8.2 电容 TP 模组图

9 屏模块

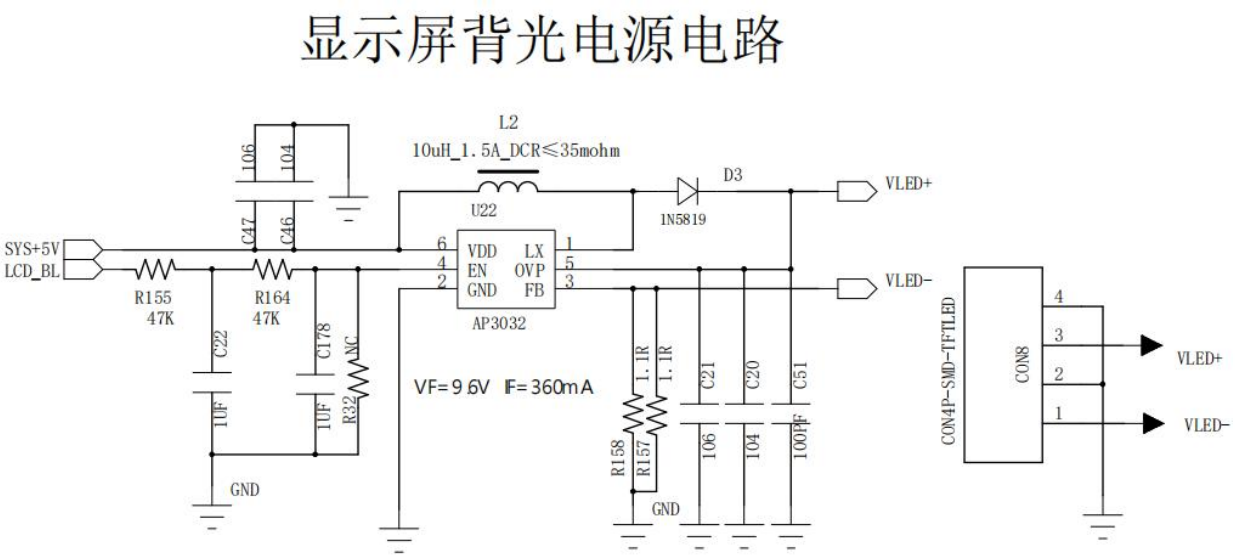
9.1 屏外围参考电路

9.1.1 显示屏正负电源电路

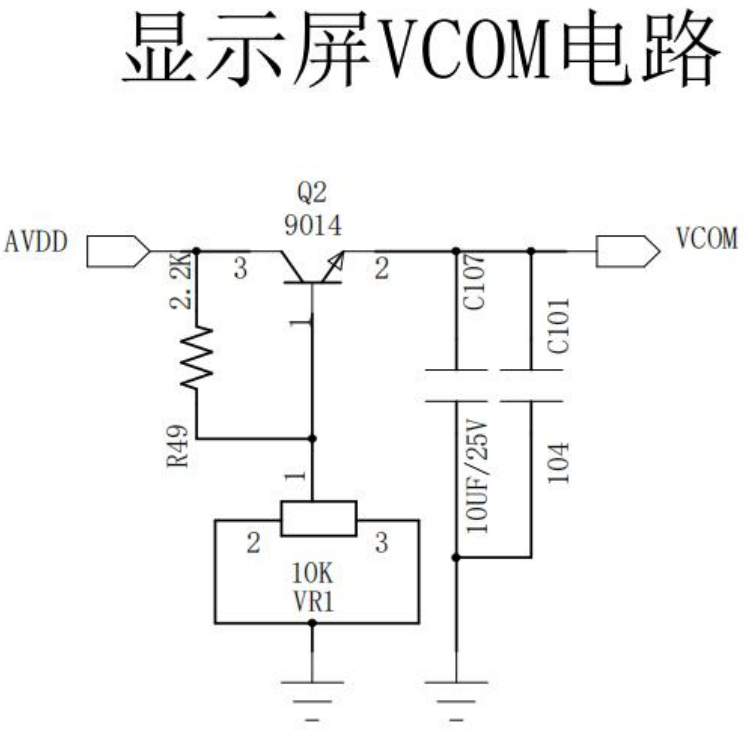


Vout=0.2(1+R142/R143)

9.1.2 显示屏背光电源电路



9.1.3 显示屏 VOCM 电路



9.2 RGB 通信

9.2.1 引脚介绍

开发板主控 MCU 与 GT-GUI LCD 7 寸液晶模组 RGB 通讯接口引脚连接方式如下图 2.1.1：RGB888 程序示例使用的即是此连接方式。

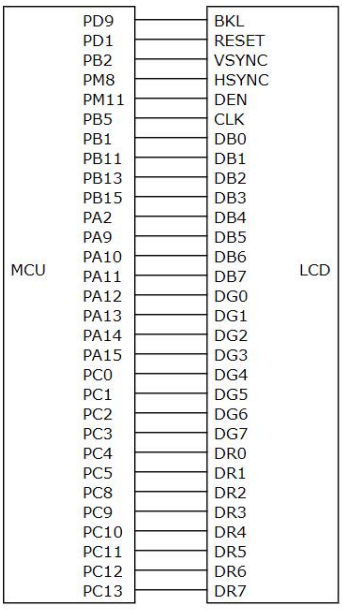


图 2.1.1

符号	名称	功能
BKL	背光	屏幕背光控制
RESET	复位	全局复位引脚，低电平有效进入复位状态
VSYNC	垂直同步	垂直同步输入
HSYNC	水平同步	水平同步输入
DEN	使能	数据使能信号线
CLK	时钟	时钟信号线
DB0-DB7	数据引脚	8根蓝色数据信号线
DG0-DG7	数据引脚	8根绿色数据信号线
DR0-DR7	数据引脚	8根红色数据信号线

图 2.1.2

图 2.1.2 为各引脚功能图介绍。

9.2.2 点亮屏流程

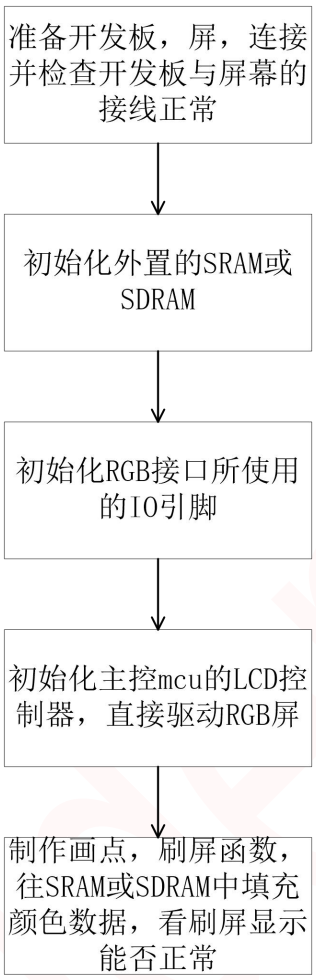


图 2.1.3

屏幕初始化点亮屏幕操作步骤流程可参考图 2.1.3。

9.2.3 RGB888 驱动程序

```
/**
 * @brief lcd  引脚配置及端口初始化
 */
static void _bsp_lcd_gpio_config(void){
    uint32_t i;

    GPIO_Init(LCD_BKL_PORT, LCD_BKL_PIN, 1, 0, 0, 0);//初始化背光引脚
    GPIO_SetBit(LCD_BKL_PORT, LCD_BKL_PIN);//背光引脚拉高
    GPIO_Init(LCD_RST_PORT, LCD_RST_PIN, 1, 0, 0, 0);//初始化复位引脚
    GPIO_ClrBit(LCD_RST_PORT, LCD_RST_PIN);//复位引脚拉低
    for(i = 0; i < 1000000; i++) __NOP();
    GPIO_SetBit(LCD_RST_PORT, LCD_RST_PIN);//复位引脚拉高
    for(i = 0; i < 1000000; i++) __NOP();
```

```
    PORT_Init(LCD_B0_PORT, LCD_B0_PIN, LCD_B0_SEL, 0);//初始化 RGB888 24 根数据端口
    PORT_Init(LCD_B1_PORT, LCD_B1_PIN, LCD_B1_SEL, 0);
    PORT_Init(LCD_B2_PORT, LCD_B2_PIN, LCD_B2_SEL, 0);
    PORT_Init(LCD_B3_PORT, LCD_B3_PIN, LCD_B3_SEL, 0);
    PORT_Init(LCD_B4_PORT, LCD_B4_PIN, LCD_B4_SEL, 0);
    PORT_Init(LCD_B5_PORT, LCD_B5_PIN, LCD_B5_SEL, 0);
    PORT_Init(LCD_B6_PORT, LCD_B6_PIN, LCD_B6_SEL, 0);
    PORT_Init(LCD_B7_PORT, LCD_B7_PIN, LCD_B7_SEL, 0);
    PORT_Init(LCD_G0_PORT, LCD_G0_PIN, LCD_G0_SEL, 0);
    PORT_Init(LCD_G1_PORT, LCD_G1_PIN, LCD_G1_SEL, 0);
    PORT_Init(LCD_G2_PORT, LCD_G2_PIN, LCD_G2_SEL, 0);
    PORT_Init(LCD_G3_PORT, LCD_G3_PIN, LCD_G3_SEL, 0);
    PORT_Init(LCD_G4_PORT, LCD_G4_PIN, LCD_G4_SEL, 0);
    PORT_Init(LCD_G5_PORT, LCD_G5_PIN, LCD_G5_SEL, 0);
    PORT_Init(LCD_G6_PORT, LCD_G6_PIN, LCD_G6_SEL, 0);
    PORT_Init(LCD_G7_PORT, LCD_G7_PIN, LCD_G7_SEL, 0);
    PORT_Init(LCD_R0_PORT, LCD_R0_PIN, LCD_R0_SEL, 0);
    PORT_Init(LCD_R1_PORT, LCD_R1_PIN, LCD_R1_SEL, 0);
    PORT_Init(LCD_R2_PORT, LCD_R2_PIN, LCD_R2_SEL, 0);
    PORT_Init(LCD_R3_PORT, LCD_R3_PIN, LCD_R3_SEL, 0);
    PORT_Init(LCD_R4_PORT, LCD_R4_PIN, LCD_R4_SEL, 0);
    PORT_Init(LCD_R5_PORT, LCD_R5_PIN, LCD_R5_SEL, 0);
    PORT_Init(LCD_R6_PORT, LCD_R6_PIN, LCD_R6_SEL, 0);
    PORT_Init(LCD_R7_PORT, LCD_R7_PIN, LCD_R7_SEL, 0);

    PORT_Init(LCD_VSYNC_PORT, LCD_VSYNC_PIN, LCD_VSYNC_SEL, 0);//初始化垂直同步
    PORT_Init(LCD_HSYNC_PORT, LCD_HSYNC_PIN, LCD_HSYNC_SEL, 0);//初始化水平同步
    PORT_Init(LCD_DEN_PORT  , LCD_DEN_PIN  , LCD_DEN_SEL  , 0);//初始化数据使能
    PORT_Init(LCD_DCLK_PORT , LCD_DCLK_PIN , LCD_DCLK_SEL , 0);//初始化时钟端口
}
#define LCD_BUF_UT        uint32_t
#define LCD_WIDTH         (800)
#define LCD_HEIGTH        (480)
#define LCD_ClkDiv        (4)
#define LCD_Format        (LCD_FMT_RGB888)
#define LCD_Hfp           (40)
#define LCD_Hbp           (40)
#define LCD_Vfp           (13)
#define LCD_Vbp           (29)
#define LCD_HsyncWidth    (48)
#define LCD_VsyncWidth    (48)
LCD_BUF_UT *_LCD_Buffer = (LCD_BUF_UT *)SDRAMM_BASE;

/**
```

```
* @brief lcd 初始化
*/
void _lcd_init_sturct(){
    LCD_InitStructure LCD_initStruct;

    LCD_initStruct.ClkDiv = LCD_ClkDiv;//时钟分频 4 分频
    LCD_initStruct.Format = LCD_Format;//数据格式 RGB888
    LCD_initStruct.HnPixel = LCD_WIDTH;//水平方向像素点 宽度 800
    LCD_initStruct.VnPixel = LCD_HEIGTH;//垂直方向像素点 高度 480
    LCD_initStruct.Hfp = LCD_Hfp;//水平同步信号的前肩
    LCD_initStruct.Hbp = LCD_Hbp;//水平同步信号的后肩
    LCD_initStruct.Vfp = LCD_Vfp;//垂直同步信号的前肩
    LCD_initStruct.Vbp = LCD_Vbp;//垂直同步信号的后肩
    LCD_initStruct.HsyncWidth = LCD_HsyncWidth;//表示水平同步信号的宽度
    LCD_initStruct.VsyncWidth = LCD_VsyncWidth;//表示垂直同步脉冲的宽度
    LCD_initStruct.DataSource = (uint32_t)_LCD_Buffer;//数据地址
    LCD_initStruct.Background = 0x0;//背景色
    LCD_initStruct.SampleEdge = LCD_SAMPLE_RISE;     // 上升沿采样
    LCD_initStruct.IntEOTEn = 1;     //End of Transter（传输完成）中断使能
    LCD_Init(LCD, &LCD_initStruct);
}

/**
 * @brief    rgb888 lcd init
 */
void bsp_lcd_rgb_init(void)
{
    _MemoryInit();//SDRAM init
    _bsp_lcd_gpio_config();//lcd gpio init
    _lcd_init_sturct();//lcd init
    LCD_Start(LCD);//start lcd
}
/**
* @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
*/
void lcd_draw_point(uint16_t x, uint16_t y, uint32_t color)
{
    _LCD_Buffer[y * LCD_WIDTH + x] = color;
}
```

10 GUI 芯片模块

10.1 SPI 通信（推荐使用 QSPI 或 SFC 通信）

10.1.1 引脚介绍

图 3.1.1 是普通 SPI 连接图，在使用 SPI 通信，HOLD 引脚需要上拉，WP 脚可以悬空。

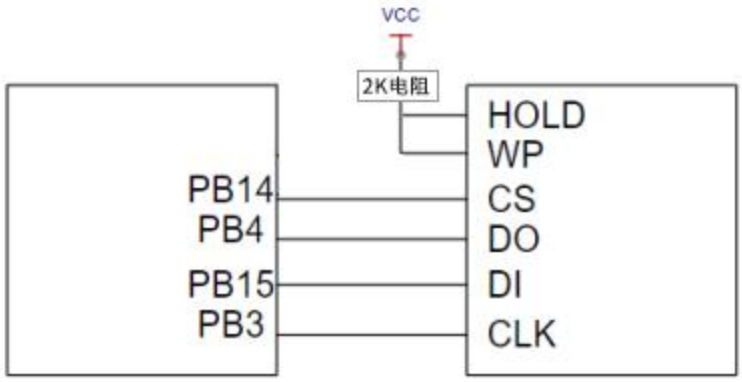


图 3.1.1

10.1.2 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 3.1.2 操作存储芯片指令，可供客户写驱动做参考。

10.1.3 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(SPI1, FALSE);
    spi_i2s_dma_receiver_enable(SPI1, FALSE);
    spi_i2s_data_transmit(SPI1, data);
    while(spi_i2s_flag_get(SPI1, SPI_I2S_RDBF_FLAG) == RESET);
    brxbuff = spi_i2s_data_receive(SPI1);
    while(spi_i2s_flag_get(SPI1, 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)
{
    erase_addr *= SPIF_SECTOR_SIZE; /* translate sector address to byte address */
    spiflash_write_enable();
    spiflash_wait_busy();
    FLASH_CS_LOW();
    spi_byte_write(0x20);
    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(SPI1, SPI_I2S_TDBE_FLAG) == RESET);
        spi_i2s_data_transmit(SPI1, 0xa5);//随意值皆可
        while(spi_i2s_flag_get(SPI1, SPI_I2S_RDBF_FLAG) == RESET);
        *pbuffer = spi_i2s_data_receive(SPI1);
        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);
    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;
}
```

10.2 QSPI 通信

10.2.1 引脚介绍

图 3.2.3 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。

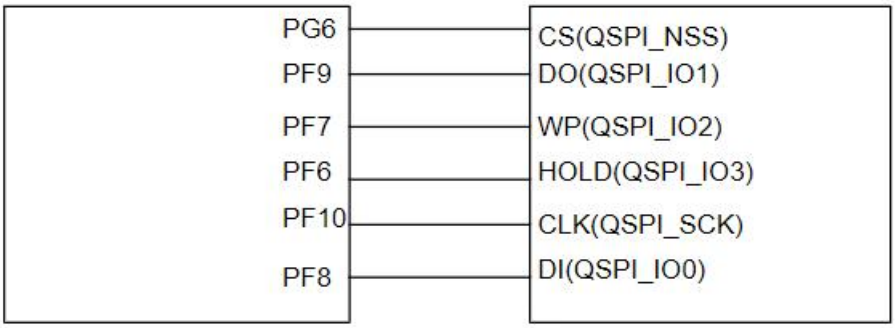


图 3.2.3

10.2.2 QSPI 驱动程序

```
qspi_cmd_type esmt32m_cmd_config;
/**
  * @brief   qspi write enable
  * @param   none
  * @retval none
  */
void qspi_write_enable(void)
{
    esmt32m_cmd_wren_config(&esmt32m_cmd_config);
    qspi_cmd_operation_kick(QSPI1, &esmt32m_cmd_config);

    /* wait command completed */
    while(qspi_flag_get(QSPI1, QSPI_CMDSTS_FLAG) == RESET);
```

```
    qspi_flag_clear(QSPI1, QSPI_CMDSTS_FLAG);
}
/**
  * @brief   esmt32m cmd rdsr config
  * @param   qspi_cmd_struct: the pointer for qspi_cmd_type parameter
  * @retval none
  */
void esmt32m_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;
    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_SW_ONCE;
    qspi_cmd_struct->read_status_enable = TRUE;
    qspi_cmd_struct->write_data_enable = FALSE;
}
/**
  * @brief   qspi check busy
  * @param   none
  * @retval none
  */
void qspi_busy_check(void)
{
    do
    {
        esmt32m_cmd_rdsr_config(&esmt32m_cmd_config);//结构体配置
        qspi_cmd_operation_kick(QSPI1, &esmt32m_cmd_config);

        /* wait command completed */
        while(qspi_flag_get(QSPI1, QSPI_CMDSTS_FLAG) == RESET);//等待操作完成
        qspi_flag_clear(QSPI1, QSPI_CMDSTS_FLAG);
        /* check wip status */
    }while(QSPI1->rsts & (1 << QSPI_BUSY_OFFSET_0));
}
/**
  * @brief   esmt32m cmd erase config
  * @param   qspi_cmd_struct: the pointer for qspi_cmd_type parameter
  * @param   addr: erase address
  * @retval none
```

```
 */
void esmt32m_cmd_erase_config(qspi_cmd_type *qspi_cmd_struct, uint32_t addr)
{
    qspi_cmd_struct->pe_mode_enable = FALSE;
    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;
    qspi_cmd_struct->address_code = addr;//擦除地址
    qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_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 = FALSE;
    qspi_cmd_struct->write_data_enable = TRUE;
}
/**
 * @brief read data from flash by qspi
 *
 * @param pbuff the pointer for data buffer
 * @param Addr the address where the data is read
 * @param rlen read buffer length
 */
void GT_QSPI_Read(u8* pbuff,u32 Addr,u16 rlen)
{
    u16 len = 0, i = 0;
    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 = 0x0B; //发送 0x0B 快速读取指令
    qspi_cmd_struct.instruction_length = QSPI_CMD_INSLEN_1_BYTE; //命令长度
    qspi_cmd_struct.address_code = ((Addr<<8) & 0xFFFFFF00) | 0xFF; // address|0xFF
    qspi_cmd_struct.address_length = QSPI_CMD_ADRLEN_4_BYTE; //  地址长度与 4
    qspi_cmd_struct.data_counter = rlen; //读数据长度
    qspi_cmd_struct.second_dummy_cycle_num = 0;
    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;
    qspi_cmd_struct.read_status_enable = FALSE;
    qspi_cmd_struct.write_data_enable = FALSE;
    qspi_cmd_operation_kick(QSPI1, &qspi_cmd_struct); //设置命令端口

    do
    {
        if(rlen>=128) //FIFO 最大 128 字节
```

```
            len=128;
        else
            len=rlen;

        while(qspi_flag_get(QSPI1, QSPI_RXFIFORDY_FLAG) == RESET);
        for(i = 0; i < len; i++)
        {
            *pbuff++ = qspi_byte_read(QSPI1);
        }
        rlen-=len;
    }while(rlen);

    while(qspi_flag_get(QSPI1, QSPI_CMDSTS_FLAG) == RESET){}
    qspi_flag_clear(QSPI1, QSPI_CMDSTS_FLAG);
}
/**
  * @brief   qspi erase data
  * @param   sec_addr: the sector address for erase
  * @retval none
  */
void qspi_erase(uint32_t sec_addr)
{
    qspi_write_enable();//写使能

    esmt32m_cmd_erase_config(&esmt32m_cmd_config, sec_addr);//擦除结构体配置
    qspi_cmd_operation_kick(QSPI1, &esmt32m_cmd_config);//执行擦除操作

    /* wait command completed */
    while(qspi_flag_get(QSPI1, QSPI_CMDSTS_FLAG) == RESET);//等待擦除完成
    qspi_flag_clear(QSPI1, QSPI_CMDSTS_FLAG);

    qspi_busy_check();
}
/**
  * @brief   esmt32m cmd write config
  * @param   qspi_cmd_struct: the pointer for qspi_cmd_type parameter
  * @param   addr: write start address
  * @param   counter: write data counter
  * @retval none
  */
void esmt32m_cmd_write_config(qspi_cmd_type *qspi_cmd_struct, uint32_t addr, uint32_t counter)
{
    qspi_cmd_struct->pe_mode_enable = FALSE;
    qspi_cmd_struct->pe_mode_operate_code = 0;
    qspi_cmd_struct->instruction_code = 0x32;//写指令
```

```
    qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE;
    qspi_cmd_struct->address_code = addr;//写地址
    qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_BYTE;
    qspi_cmd_struct->data_counter = counter;//写数据长度
    qspi_cmd_struct->second_dummy_cycle_num = 0;
    qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_114;
    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 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_data_write(uint32_t addr, uint32_t total_len, uint8_t* buf)
{
    uint32_t i, len = total_len;

    do{
        qspi_write_enable();//写使能·
        if(total_len >= FLASH_PAGE_PROGRAM_SIZE){
            len = FLASH_PAGE_PROGRAM_SIZE;
        }else{
            len = total_len;
        }
        esmt32m_cmd_write_config(&esmt32m_cmd_config, addr, len);//结构体配置
        qspi_cmd_operation_kick(QSPI1, &esmt32m_cmd_config);//执行操作

        for(i = 0; i < len; ++i){
            while(qspi_flag_get(QSPI1, QSPI_TXFIFORDY_FLAG) == RESET);
            qspi_byte_write(QSPI1, *buf++);
        }
        total_len -= len;
        addr += len;

        /* wait command completed */
        while(qspi_flag_get(QSPI1, QSPI_CMDSTS_FLAG) == RESET);
        qspi_flag_clear(QSPI1, QSPI_CMDSTS_FLAG);

        qspi_busy_check();//等待忙完成

    }while(total_len);
```

10.2.3 SFC 驱动程序

除上述常规 QSPI 驱动方式外，部分主控 MCU（如华芯微特的 SWM341 系列芯片）可能携带有 SFC(Serial Flash Controller)串行 Flash 控制器，可以使用这种控制器进行驱动以获取更快的读取速度。

```
/******************************************************************************************
* 函数名称: SFC_Init()
* 功能说明: SFC（Serial Flash Controller）初始化
* 输    入: SFC_InitStructure * initStruct    SFC 初始化配置值
* 输    出: 无
* 注意事项: 无
******************************************************************************************/
void SFC_Init(SFC_InitStructure * initStruct)
{
    SYS->CLKEN1 |= (1 << SYS_CLKEN1_SFC_Pos);

    *((__IO uint32_t *)((uint32_t )&SFC->CFG + 0x3F4)) = 7;

    SFC->CFG &= ~(SFC_CFG_CLKDIV_Msk | SFC_CFG_DATA4L_RD_Msk | SFC_CFG_DATA4L_PP_Msk);
    SFC->CFG |= (initStruct->ClkDiv << SFC_CFG_CLKDIV_Pos) |
                (initStruct->Width_Read << SFC_CFG_DATA4L_RD_Pos) |
                (initStruct->Width_PageProgram << SFC_CFG_DATA4L_PP_Pos);

    SFC->CFG |= (1 << SFC_CFG_CMDWREN_Pos);
    SFC->CMDAHB &= ~(SFC_CMDAHB_READ_Msk | SFC_CMDAHB_PP_Msk);
    SFC->CMDAHB |= (initStruct->Cmd_Read << SFC_CMDAHB_READ_Pos) |
                   (initStruct->Cmd_PageProgram << SFC_CMDAHB_PP_Pos);
    SFC->CFG &= ~(1 << SFC_CFG_CMDWREN_Pos);

    SFC->TIM &= ~(SFC_TIM_WIP_CHK_ITV_Msk | SFC_TIM_WIP_CHK_LMT_Msk);
    SFC->TIM |= ((CyclesPerUs / 10) << SFC_TIM_WIP_CHK_ITV_Pos) |   //2048 * (CyclesPerUs / 10)
/ CyclesPerUs us = 0.2 ms
                (255 << SFC_TIM_WIP_CHK_LMT_Pos);
}

void bsp_sfc_config(uint8_t width)
{
    SFC_InitStructure SFC_initStruct;

    /* SFC 使用专用的 FSPI（Flash SPI）接口连接 SPI Flash */
    PORT_Init(PORTD, PIN5, PORTD_PIN5_FSPI_SCLK,  0);
    PORT_Init(PORTD, PIN6, PORTD_PIN6_FSPI_SSEL,  0);
    PORT_Init(PORTD, PIN8, PORTD_PIN8_FSPI_MOSI,  1);
    PORT_Init(PORTD, PIN7, PORTD_PIN7_FSPI_MISO,  1);
    PORT_Init(PORTD, PIN3, PORTD_PIN3_FSPI_DATA2, 1);
```

```
    PORT_Init(PORTD, PIN4, PORTD_PIN4_FSPI_DATA3, 1);

    switch(width)
    {
    case 1:
    default:
        SFC_initStruct.ClkDiv = SFC_CLKDIV_4;
        SFC_initStruct.Cmd_Read = 0x03;
        SFC_initStruct.Width_Read = SFC_RDWIDTH_1;
        SFC_initStruct.Cmd_PageProgram = 0x02;
        SFC_initStruct.Width_PageProgram = SFC_PPWIDTH_1;
        break;

    case 2:
        SFC_initStruct.ClkDiv = SFC_CLKDIV_4;
        SFC_initStruct.Cmd_Read = 0xBB;
        SFC_initStruct.Width_Read = SFC_RDWIDTH_2;
        SFC_initStruct.Cmd_PageProgram = 0x02;
        SFC_initStruct.Width_PageProgram = SFC_PPWIDTH_1;
        break;

    case 4:
        SFC_initStruct.ClkDiv = SFC_CLKDIV_4;
        SFC_initStruct.Cmd_Read = 0xEB;
        SFC_initStruct.Width_Read = SFC_RDWIDTH_4;
        SFC_initStruct.Cmd_PageProgram = 0x32;
        SFC_initStruct.Width_PageProgram = SFC_PPWIDTH_4;
        break;
    }

    SFC_Init(&SFC_initStruct);
}

/*******************************************************************************
* 函数名称: SFC_Write()
* 功能说明: SPI Flash 数据写入
* 输    入: uint32_t addr     数据要写入到 Flash 中的地址，字对齐
*           uint32_t buff[]     要写入 Flash 中的数据
*           uint32_t cnt        要写的数据的个数，以字为单位，最大 64
* 输    出: 无
* 注意事项: 要写入的数据必须全部在同一页内，即 addr/256 == (addr+(cnt-1)*4)/256
*           当 cnt > 4 时，LCD_DCLK 输出可能出现间断（|__|‾|__|‾|__|‾|__|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|__|‾|__|‾|__|），这种情况下有些屏幕会显示异常，遇
*           到这种情况，可通过以 cnt = 4 多次调用 SFC_Write() 解决
*******************************************************************************/
```

```
void SFC_Write(uint32_t addr, uint32_t buff[], uint32_t cnt)
{
    SFC->CFG |= (1 << SFC_CFG_WREN_Pos);
    for(int i = 0; i < cnt; i++)
        *((volatile unsigned int *)(0x70000000+addr+i*4)) = buff[i];

    while(SFC->SR & SFC_SR_BUSY_Msk) __NOP();
    SFC->CFG &= ~SFC_CFG_WREN_Msk;
}

/*******************************************************************************
* 函数名称: SFC_Read()
* 功能说明: SPI Flash 数据读取
* 输    入: uint32_t addr     要读取的数据在 Flash 中的地址，字对齐
*           uint32_t buff[]     读取到的数据存入 buff 指向的内存
*           uint32_t cnt        要读取的数据的个数，以字为单位
* 输    出: 无
* 注意事项: 无
*******************************************************************************/
void SFC_Read(uint32_t addr, uint32_t buff[], uint32_t cnt)
{
    for(int i = 0; i < cnt; i++)
        buff[i] = *((volatile unsigned int *)(0x70000000+addr+i*4));
}

/*******************************************************************************
* 函数名称: SFC_Erase()
* 功能说明: SPI Flash 扇区擦除，每个扇区 4K 字节
* 输    入: uint32_t addr     要擦除扇区的地址，必须 4K 对齐，即 addr%4096 == 0
*           uint8_t wait        1 等待 Flash 完成擦除操作后再返回   0 发出擦除命令后立即返回
* 输    出: 无
* 注意事项: 无
*******************************************************************************/
void SFC_Erase(uint32_t addr, uint8_t wait)
{
    SFC_EraseEx(addr, SFC_CMD_ERASE_SECTOR, wait);
}

/*******************************************************************************
* 函数名称: SFC_EraseEx()
* 功能说明: SPI Flash 擦除，通过提供不同的命令码支持片擦、块擦、扇区擦
* 输    入: uint32_t addr     要擦除的块的地址，当 addr == 0xFFFFFFFF 时，执行片擦
*           uint8_t cmd         擦除命令码，有些 SPI Flash 支持多种大小的块的擦除，不同块大大小使
用不同命令码
*           uint8_t wait        1 等待 Flash 完成擦除操作后再返回   0 发出擦除命令后立即返回
```

```
* 输    出: 无
* 注意事项: 无
*******************************************************************************/
void SFC_EraseEx(uint32_t addr, uint8_t cmd, uint8_t wait)
{
    uint8_t type = (addr == 0xFFFFFFFF) ? 5 : 7;

    SFC->ADDR = addr;

    SFC->CFG &= ~SFC_CFG_CMDTYPE_Msk;
    SFC->CFG |= (1 << SFC_CFG_WREN_Pos) |
                (1 << SFC_CFG_CMDWREN_Pos) |
                (type << SFC_CFG_CMDTYPE_Pos);
    SFC->CMD = cmd;
    SFC->GO = 1;

    for(int i = 0; i < CyclesPerUs; i++) __NOP();   //等待命令发出

    SFC->CFG &= ~SFC_CFG_WREN_Msk;

    if(wait)
        while(SFC_FlashBusy()) __NOP();
}

/*******************************************************************************
* 函数名称: SFC_QuadSwitch()
* 功能说明: SPI Flash Quad 模式开关
* 输    入: uint8_t on           1 开启 Quad 模式   0 关闭 Quad 模式
* 输    出: 无
* 注意事项: 无
*******************************************************************************/
void SFC_QuadSwitch(uint8_t on)
{
    uint16_t reg = (SFC_ReadStatusReg(SFC_CMD_READ_STATUS_REG2) << 8) |
                   (SFC_ReadStatusReg(SFC_CMD_READ_STATUS_REG1) << 0);

    if(on)
        reg |= (1 << SFC_STATUS_REG_QUAD_Pos);
    else
        reg &=~(1 << SFC_STATUS_REG_QUAD_Pos);

    SFC_WriteStatusReg(SFC_CMD_WRITE_STATUS_REG1, reg);
}
```

11 电容触摸模块

11.1 引脚介绍

电容模块 IC 是用 GT911，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 4.1 是 GT911 与主控连接图。

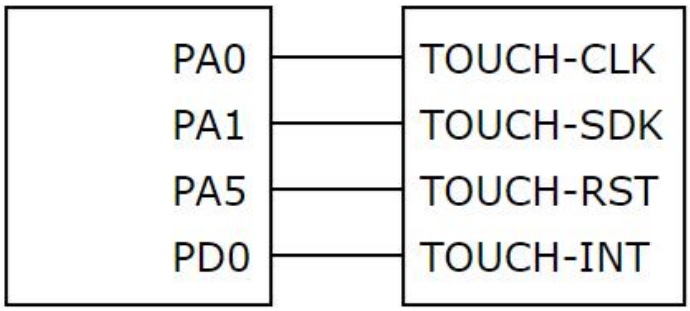


图 4.1

11.2 寄存器介绍

图 4.2 是介绍本次用到的寄存器功能，如想知道其他寄存器功能，请自行查询。读取芯片 ID 可测试硬件通信是否正常。

寄存器	功 能
0x8040	控制寄存器
0x8047	配置起始地址寄存器
0x80FF	校验和寄存器
0x8140	芯片ID寄存器
0x814E	当前触摸状态
0x8150	第一触摸点数据地址
0x8158	第二触摸点数据地址
0x8160	第三触摸点数据地址
0x8168	第四触摸点数据地址
0x8170	第五触摸点数据地址

图 4.2

11.3 电容触摸驱动程序

```
#define TP_GPIO_RST           GPIOA
#define TP_PIN_RST            PIN5

#define TP_GPIO_INT           GPIOD
#define TP_PIN_INT            PIN0

#define TP_PORT_SCL           PORTA
#define TP_PIN_SCL            PIN1
#define TP_PIN_SCL_FUN        PORTA_PIN1_I2C0_SCL

#define TP_PORT_SDA           PORTA
#define TP_PIN_SDA            PIN0
#define TP_PIN_SDA_FUN        PORTA_PIN0_I2C0_SDA

#define GT_CTRL_REG       0X8040       //GT911 控制寄存器
#define GT_CFGS_REG       0X8047       //GT911 配置起始地址寄存器
#define GT_CHECK_REG      0X80FF       //GT911 校验和寄存器
#define GT_PID_REG        0X8140       //GT911 产品 ID 寄存器

#define GT_GSTID_REG      0X814E       //GT911 当前检测到的触摸情况
#define GT_TP1_REG        0X8150       //第一个触摸点数据地址
#define GT_TP2_REG        0X8158       //第二个触摸点数据地址
#define GT_TP3_REG        0X8160       //第三个触摸点数据地址
#define GT_TP4_REG        0X8168       //第四个触摸点数据地址
#define GT_TP5_REG        0X8170       //第五个触摸点数据地址

/**
 * @brief 向 GT911 写入一次数据
 * @param reg 起始寄存器地址
 * @param buf 数据缓缓存区
 * @param len 写数据长度
 * @return 0,成功;1,失败.
 */
uint8_t GT911_WR_Reg(uint16_t reg, uint8_t * buf, uint8_t len)
{
    uint32_t i;
    uint8_t ack;

    ack = I2C_Start(I2C0, (GT9x_ADDR << 1) | 0, 1);
    if (ack == 0)
        goto wr_fail;

    ack = I2C_Write(I2C0, reg >> 8, 1);
    if (ack == 0)
        goto wr_fail;
```

```
    ack = I2C_Write(I2C0, reg & 0XFF, 1);
    if (ack == 0)
        goto wr_fail;

    for (i = 0; i < len; i++) {
        ack = I2C_Write(I2C0, buf[i], 1);
        if (ack == 0)
            goto wr_fail;
    }

    I2C_Stop(I2C0, 1);
    return 0;

wr_fail:
    I2C_Stop(I2C0, 1);
    return 1;
}
/**
 * @brief 从 GT911 读出一次数据
 * @param reg 起始寄存器地址
 * @param buf 数据缓缓存区
 * @param len 读数据长度
 * @return 0,成功;1,失败.
 */
uint8_t GT911_RD_Reg(uint16_t reg, uint8_t * buf, uint8_t len)
{
    uint8_t i;
    uint8_t ack;

    ack = I2C_Start(I2C0, (GT9x_ADDR << 1) | 0, 1);
    if (ack == 0) {
        goto rd_fail;
    }
    ack = I2C_Write(I2C0, reg >> 8, 1);
    if (ack == 0) {
        goto rd_fail;
    }
    ack = I2C_Write(I2C0, reg & 0XFF, 1);
    if (ack == 0) {
        goto rd_fail;
    }
    swm_delay_ms(5);//此处必须延时等待一会再启动

    ack = I2C_Start(I2C0, (GT9x_ADDR << 1) | 1, 1); //ReStart
```

```
    if (ack == 0) {
        goto rd_fail;
    }
    for (i = 0; i < len - 1; i++) {
        buf[i] = I2C_Read(I2C0, 1, 1);
    }
    buf[i] = I2C_Read(I2C0, 0, 1);

    I2C_Stop(I2C0, 1);
    return 0;

rd_fail:
    I2C_Stop(I2C0, 1);
    return 1;
}
/**
 * @brief G911 软复位操作
 */
void GT911_Soft_Reset(void)
{
    uint8_t buf[1];
    buf[0] = 0x01;
    GT911_WR_Reg(GT911_COMMAND_REG, (uint8_t *)buf, 1);
}
/**
 * @brief GT911 硬复位操作,RST 为低电平时，INT 持续为低电平，1ms 后 RST 置为高电平，10ms 后 INT 设置为输入，
 *  使 GT911 地址设定为 0xBA/0x28。
 */
void GT911_Reset_Sequence(uint8_t ucAddr) //未使用
{
    //GT911_RST_INT_GPIO_Init();
    switch (ucAddr) {
    case 0xBA:
        GT911_RST_0(); //RST 引脚低电平
        GT911_INT_0(); //INT 引脚低电平
        swm_delay_ms(30); //延时 30ms，最短 1
        GT911_RST_1(); //RST 引脚高电平
        GT911_INT_0(); //INT 引脚低电平
        swm_delay_ms(30); //延时 30ms，最短 20
        GT911_INT_0();
        swm_delay_ms(30); //延时 30ms，最短 20
        GT911_INT_1();
        break;
    case 0x28:
```

```
        GT911_RST_0(); //RST 引脚低电平
        GT911_INT_1(); //INT 引脚高电平
        swm_delay_ms(30); //延时 30ms，最短 1
        GT911_RST_1(); //RST 引脚高电平
        GT911_INT_1(); //INT 引脚高电平
        swm_delay_ms(30); //延时 30ms，最短 20
        GT911_INT_0();
        swm_delay_ms(30); //延时 30ms，最短 20
        GT911_INT_1();
        break;
    default: //其他默认为 0xBA
        GT911_RST_0(); //RST 引脚低电平
        GT911_INT_0(); //INT 引脚低电平
        swm_delay_ms(30); //延时 30ms，最短 1
        GT911_RST_1(); //RST 引脚高电平
        GT911_INT_0(); //INT 引脚低电平
        swm_delay_ms(30); //延时 30ms，最短 20
        GT911_INT_0();
        swm_delay_ms(30); //延时 30ms，最短 20
        GT911_INT_1();
        break;
    }
}

/**
 * @brief GT911 初始化
 * @return 0 初始化成功 1 初始化失败非 GT911
 */
uint8_t GT911_Init(void)
{
    uint8_t temp[4]; uint16_t i = 0;
    I2C_InitStructure I2C_initStruct;

    PORT_Init(TP_PORT_SCL, TP_PIN_SCL, TP_PIN_SCL_FUN, 1); //配置为 I2C0 SCL 引脚
    TP_PORT_SCL->PULLU |= (1 << TP_PIN_SCL);                //必须使能上拉
    TP_PORT_SCL->OPEND |= (1 << TP_PIN_SCL);                //开漏
    PORT_Init(TP_PORT_SDA, TP_PIN_SDA, TP_PIN_SDA_FUN, 1); //配置为 I2C0 SDA 引脚
    TP_PORT_SDA->PULLU |= (1 << TP_PIN_SDA);
    TP_PORT_SDA->OPEND |= (1 << TP_PIN_SDA);

    I2C_initStruct.Master = 1;
    I2C_initStruct.Addr10b = 0;
    I2C_initStruct.MstClk = 100000;
    I2C_initStruct.TXEmptyIEn = 0;    //发送寄存器空中断使能
    I2C_initStruct.RXNotEmptyIEn = 0;//接收寄存器非空中断使能
```

```
    I2C_Init(I2C0, &I2C_initStruct);
    I2C_Open(I2C0);

    GPIO_Init(TP_GPIO_INT, TP_PIN_INT, 0, 0, 1, 0); // 输入，开启下拉。复位时 INT 为低，选择 0xBA 作为地址
    GPIO_Init(TP_GPIO_RST, TP_PIN_RST, 1, 1, 0, 0); //复位引脚初始化
    GPIO_ClrBit(TP_GPIO_RST, TP_PIN_RST);//复位拉低
    swm_delay_ms(10);
    GPIO_SetBit(TP_GPIO_RST, TP_PIN_RST); //复位拉高
    swm_delay_ms(10);
    temp[0] = 0x02;
    GT911_WR_Reg(GT_CTRL_REG, temp, 1); // 软复位
    swm_delay_ms(100);

    GT911_Reset_Sequence(0xBA);//硬复位 设置复位后地址 0xBA
    GT911_RD_Reg(GT_PID_REG, temp, 4);//读取产品 ID
    if (strcmp((char *)temp, "911") == 0)//ID==911
    {
        temp[0] = 0X02;
        GT911_WR_Reg(GT_CTRL_REG, temp, 1);//软复位 GT911
        GT911_RD_Reg(GT_CFGS_REG, temp, 1);//读取 GT_CFGS_REG 寄存器
        swm_delay_ms(10);
        temp[0] = 0X00;
        GT911_WR_Reg(GT_CTRL_REG, temp, 1);//结束复位
        return 0;
    }
    return 1;
}
/**
 * @brief 触摸扫描
 * @param mode 0 正常扫描
 * @return 0 无触摸 1 有触摸
 */
uint8_t GT911_Scan(uint8_t mode)
{
    uint8_t buf[4];
    uint8_t i = 0, res = 0;
    uint8_t temp, temp_sta;
    uint8_t cnt = 0;
    uint8_t status = 0;
    uint8_t ret_reset = 0;

    //读取触摸点的状态
    if (GT911_RD_Reg(GT_GSTID_REG, &status, 1)) {
        ret_reset = 1;
        goto reset_lb;
```

```
    }
    if (TP_Dev.status & TP_PRES_DOWN) {
        switch (status) {
        case 0x00: {
            res = 1;
            goto ret_lb;
        }
        case 0xff: {
            res = 0;
            TP_Dev.status = 0x00;
            goto ret_lb;
        }
        default:
            break;
        }
    }
    cnt = status & 0X0f;

    if (status & 0X80 && cnt < 6) {
        temp = 0;
        //清标志
        if (GT911_WR_Reg(GT_GSTID_REG, &temp, 1)) {
            ret_reset = 1;
            goto reset_lb;
        }
    }
    if (!cnt) {
        goto ret_lb;
    }
    temp = 0XFF << cnt;       //将点的个数转换为 1 的位数,匹配 TP_Dev.sta 定义
    temp_sta = TP_Dev.status;              //保存当前的 TP_Dev.sta 值
    TP_Dev.status = (~temp) | TP_PRES_DOWN | TP_CATH_PRES;

    TP_Dev.x[4] = TP_Dev.x[0];   //保存触点 0 的数据
    TP_Dev.y[4] = TP_Dev.y[0];
    for (i = 0; i < 5; i++) {
        if (TP_Dev.status & (1 << i)) {
            //读取 XY 坐标值
            if (GT911_RD_Reg(GT911_TPX_TBL[i], buf, 4)) {
                ret_reset = 1;
                goto reset_lb;
            }
            TP_Dev.x[i] = ((uint16_t)(buf[1] & 0X0F) << 8) + buf[0];
            TP_Dev.y[i] = ((uint16_t)(buf[3] & 0X0F) << 8) + buf[2];
        }
```

```
    }
    res = 1;
    if (TP_Dev.x[0] < LCD_HDOT && TP_Dev.y[0] < LCD_VDOT) {
        goto ret_lb;
    }
    /** 非法范围的值 */
    if (cnt > 1) {
        TP_Dev.x[0] = TP_Dev.x[1];
        TP_Dev.y[0] = TP_Dev.y[1];
    } else {
        TP_Dev.x[0] = TP_Dev.x[4];
        TP_Dev.y[0] = TP_Dev.y[4];
        status = 0X80;
        TP_Dev.status = temp_sta;    //恢复 TP_Dev.sta
    }
ret_lb:
    if ((status & 0X8F) == 0X80)//无触摸点按下
    {
        if (TP_Dev.status & TP_PRES_DOWN)    //之前是被按下的
        {
            TP_Dev.status &= ~(1 << 7); //标记按键松开
        } else {
            TP_Dev.x[0] = 0xffff;
            TP_Dev.y[0] = 0xffff;
            TP_Dev.status &= 0XE0;  //清除点有效标记
        }
    }
    return res;

reset_lb:
    if (ret_reset) {
        GT911_Soft_Reset();
        // GT911_Init();
    }
    return res;
}
```

12 GUI 工具使用-HMI

12.1 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。

12.2 GT-GUI LCD 模组与 HMI 工具使用

GT-GUI LCD 7 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 7 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

12.3 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”。然后准备一个已经移植好 GT-HMI Engine 的工程，（使用 GUI-LCD 开发板可以省略移植 Engine 的过程，直接使用提供的示例工程即可），将 Designer 上位机代码移植到工程运行

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。

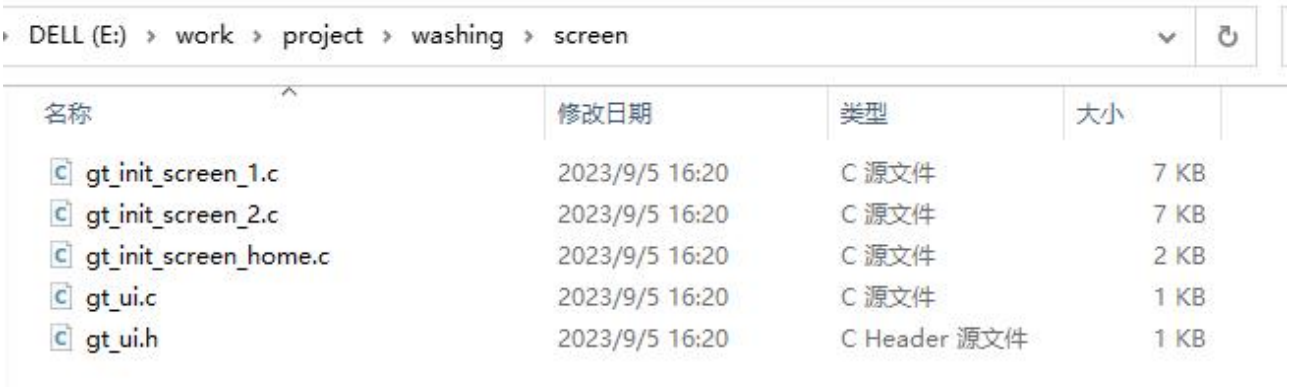


图 11.1 screen 目录下的文件

图 11.2 board 目录下的文件

烧录 bin 文件步骤如下

注:强烈推荐使用 TF 卡升级方法进行升级！详见（12.7 HMI TF/SD 卡升级方法）章节.

第一步：准备好如下烧录器，烧录器连接的是最下面四行，图 11.3 右边表格表示烧录器连接存储芯片的引脚号。

1	8
2	7
3	6
4	5

图 11.3

第二步：将烧录器和存储芯片引脚连接起来，图 11.4 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 7 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 11.4

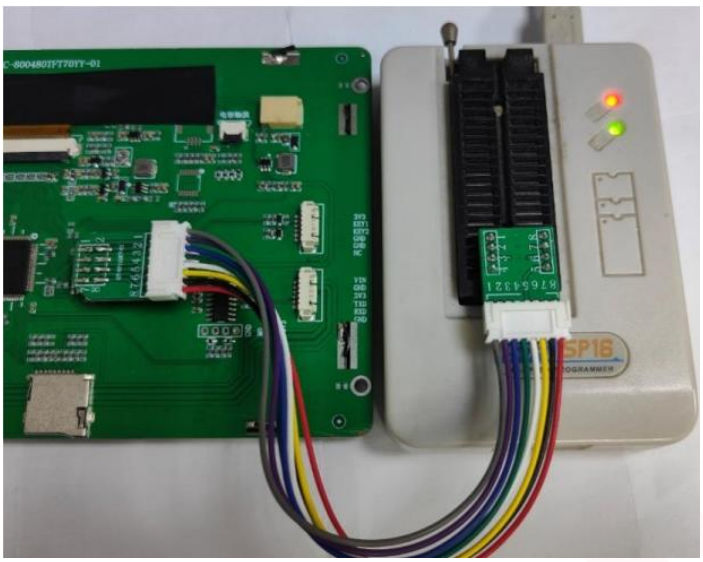


图 11.5 存储芯片与烧录器连接图

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

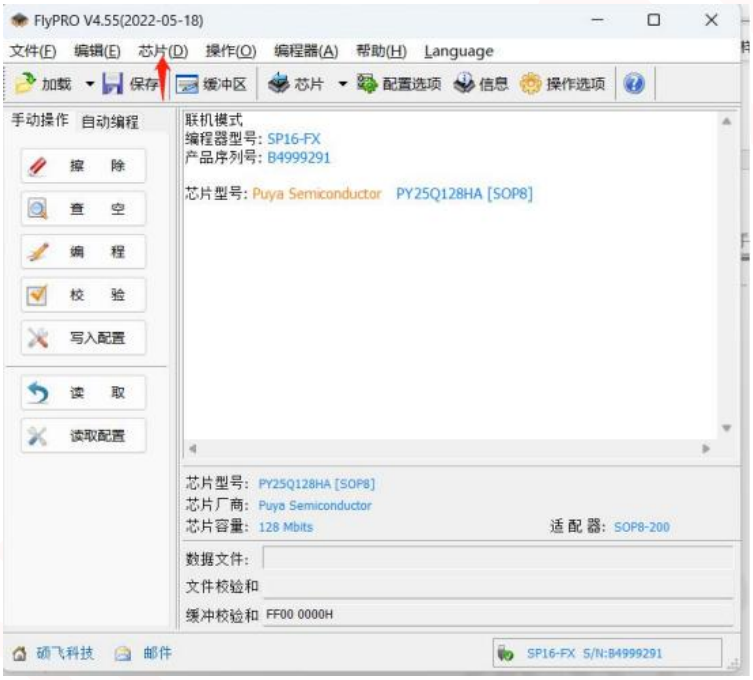


图 11.6

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。

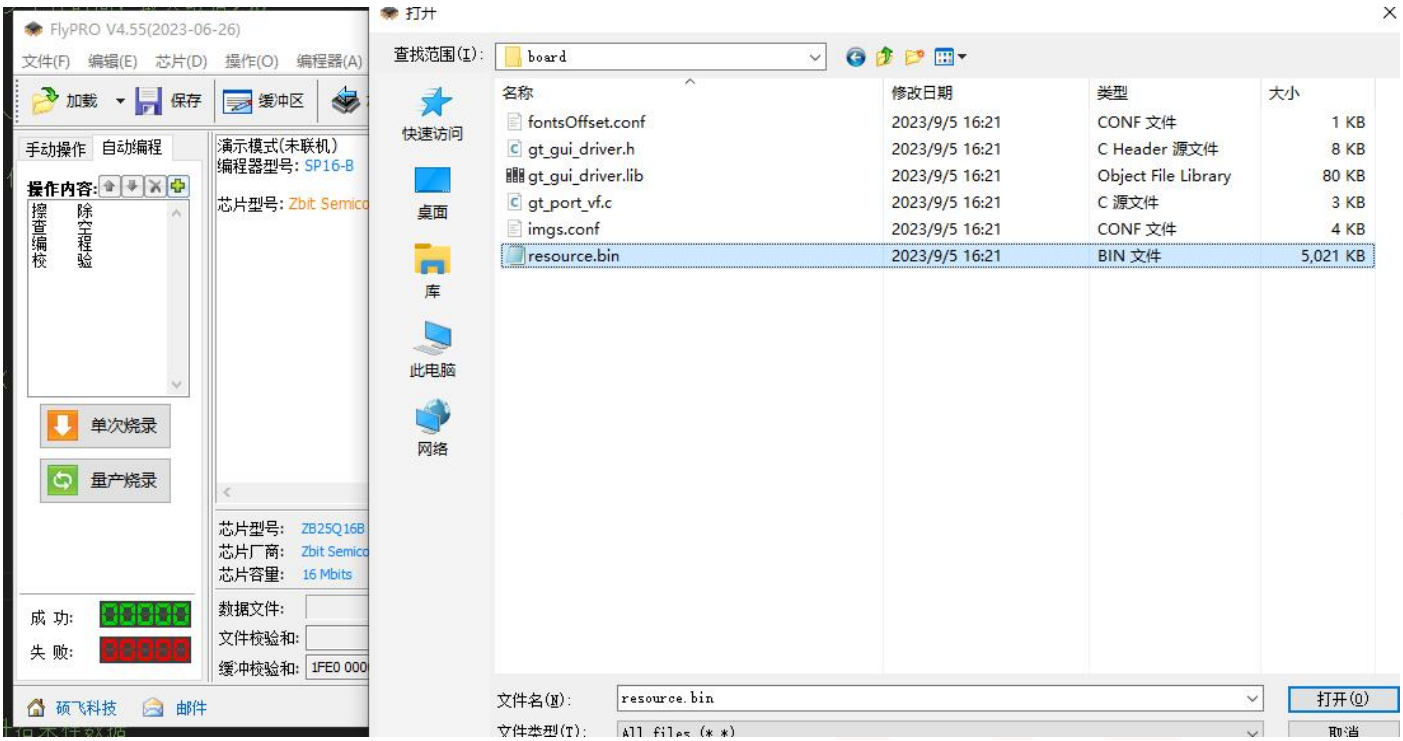


图 11.7

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 11.8 为成功烧录的界面。

图 11.8

12.4 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 11.9

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，

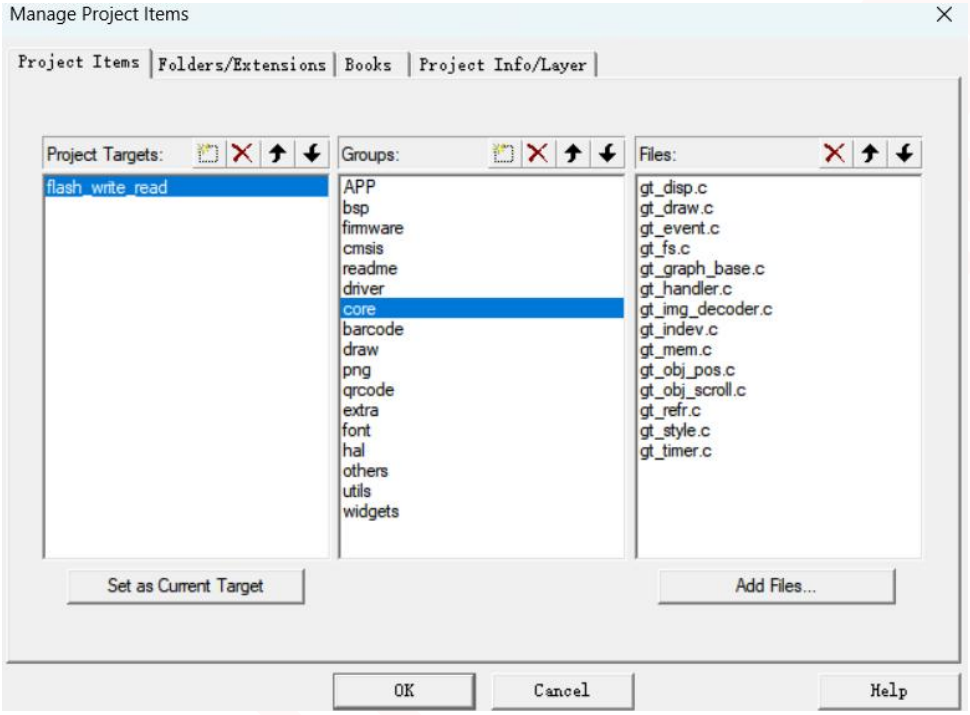


图 11.10

第三步：打开 gnu 配置进行编译，图 11.11 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。

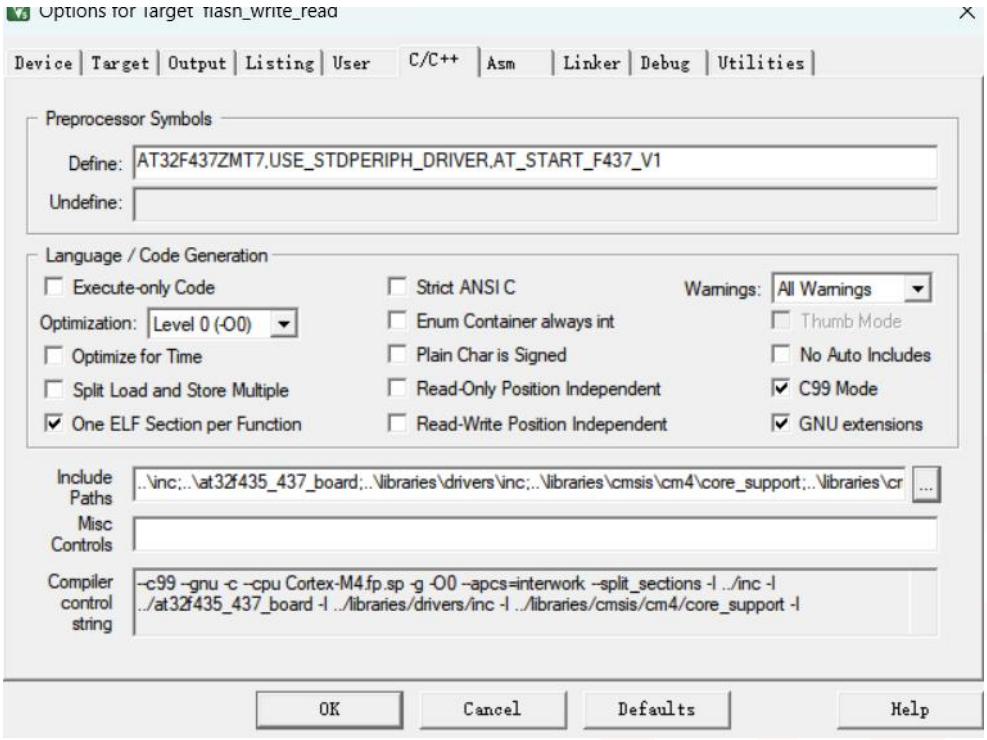


图 11.11

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

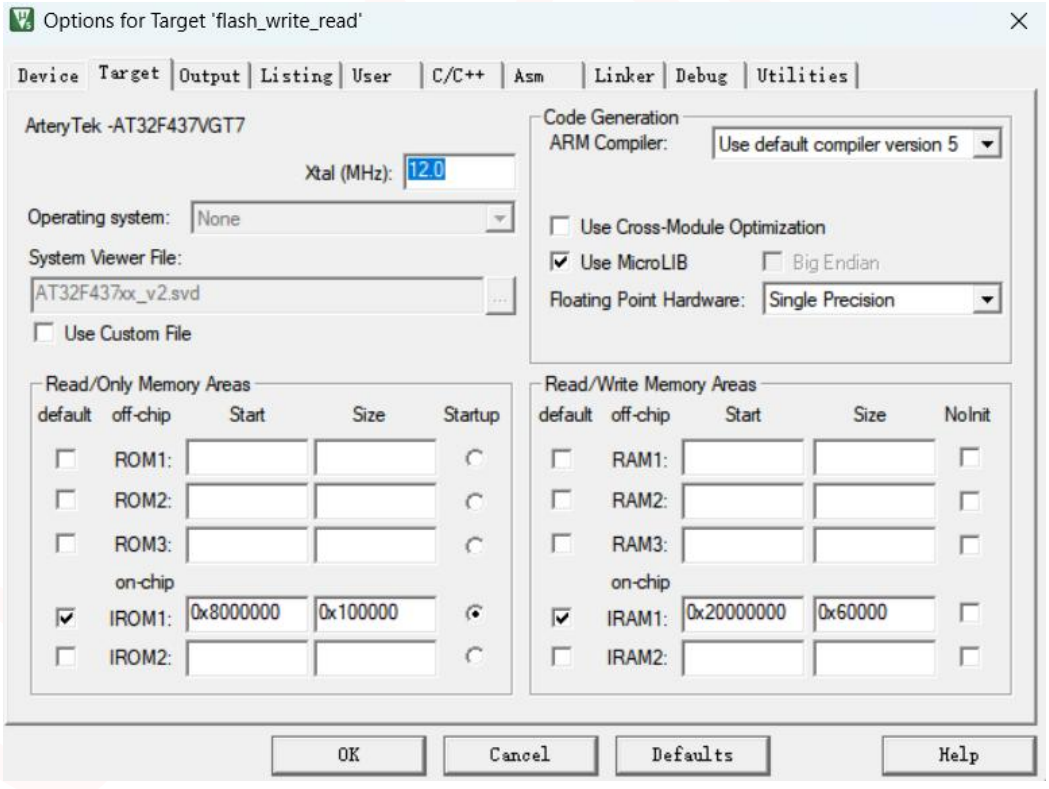


图 11.12

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 11.13

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。

12.5 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;
    uint8_t cnt = 0;
    uint32_t x, j = 0;

    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

    cnt = ReadAddr % 4;
    if (cnt == 0) {
        addr = ReadAddr;
    }
    else {
        addr = ReadAddr - cnt;
    }

    len = len_read + cnt;
    if (len % 4 != 0) {
        len = len + 4 - (len % 4);
    }

    memcpy((volatile unsigned int *)0x80368000, (volatile unsigned int *)(0x70000000 + addr), len);
    memcpy(data_read, (volatile unsigned int *)(0x80368000 + cnt), len_read);

    return 1;
}
void _flush_cb(struct _gt_disp_drv_s * drv, gt_area_st * area, gt_color_t * color) {
```

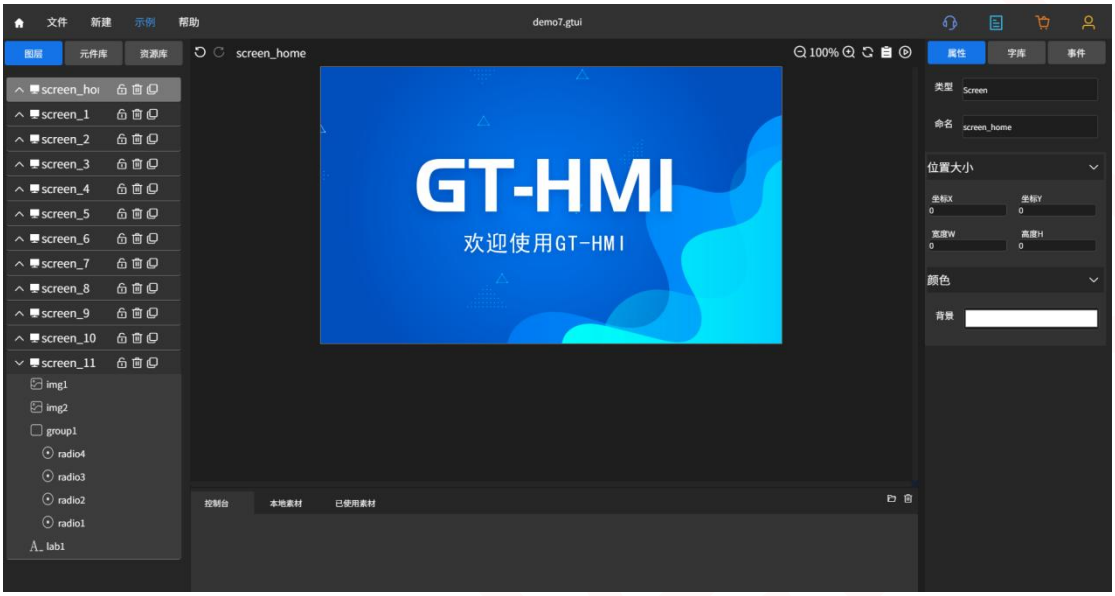
```
    uint32_t x = 0, y, j = 0;
    uint32_t indx = 0;
    uint16_t _color;
    uint16_t * p = &_LCD_Buffer[(area->y) * LCD_WIDTH + (area->x)];
    uint16_t step = LCD_WIDTH - area->w;

    if (area->w == LCD_WIDTH)
        memcpy(p, color, area->w * area->h * 2);
    else
        for (y = 0; y < area->h; y++) {
            memcpy(p, color, area->w * 2);
            p += (step + area->w);
            color += (area->w);
        }
}
void read_cb(struct _gt_indev_drv_s * indev_drv, gt_indev_data_st * data) {
    if (!GT911_Scan(0)) {
        data->state = GT_INDEV_STATE_RELEASED;
        return;
    }
    ;
    data->point.x = TP_Dev.x[0];
    data->point.y = TP_Dev.y[0];
    data->state = GT_INDEV_STATE_PRESSED;
}
```

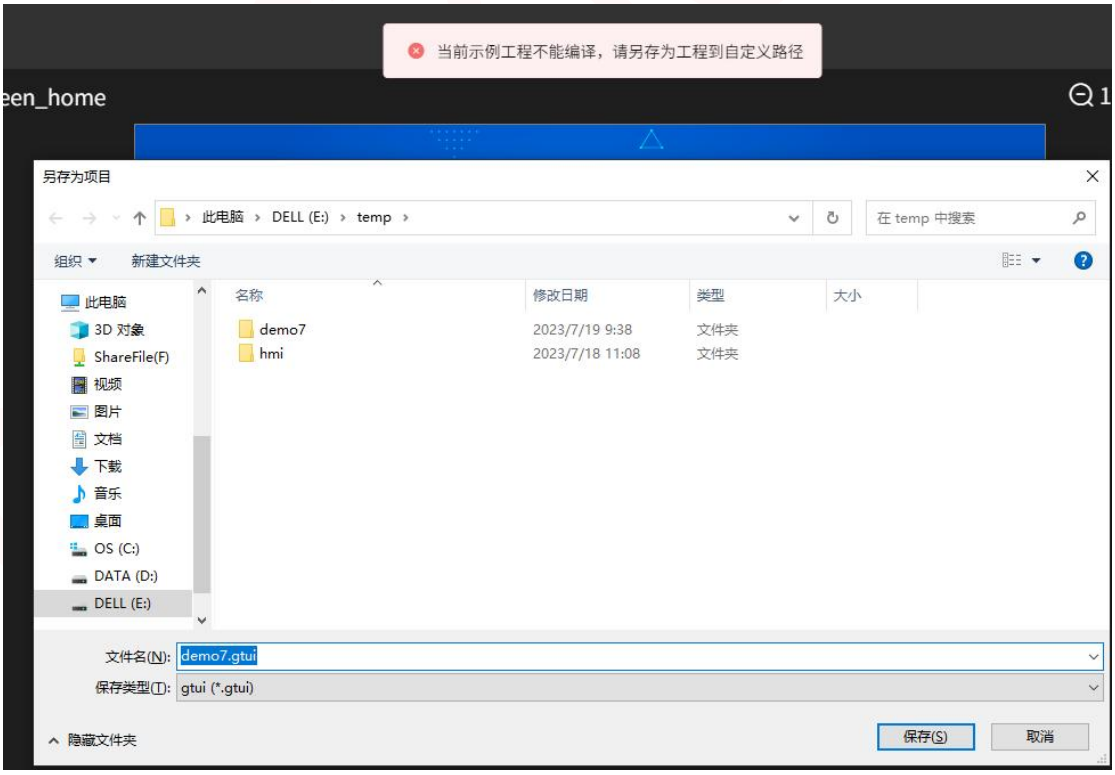
注：read_cb_btn 接口函数，如有需求可联系我司技术人员提供。

12.6 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 7 寸示例来说明移植过程。首先打开 7 寸示例工程。

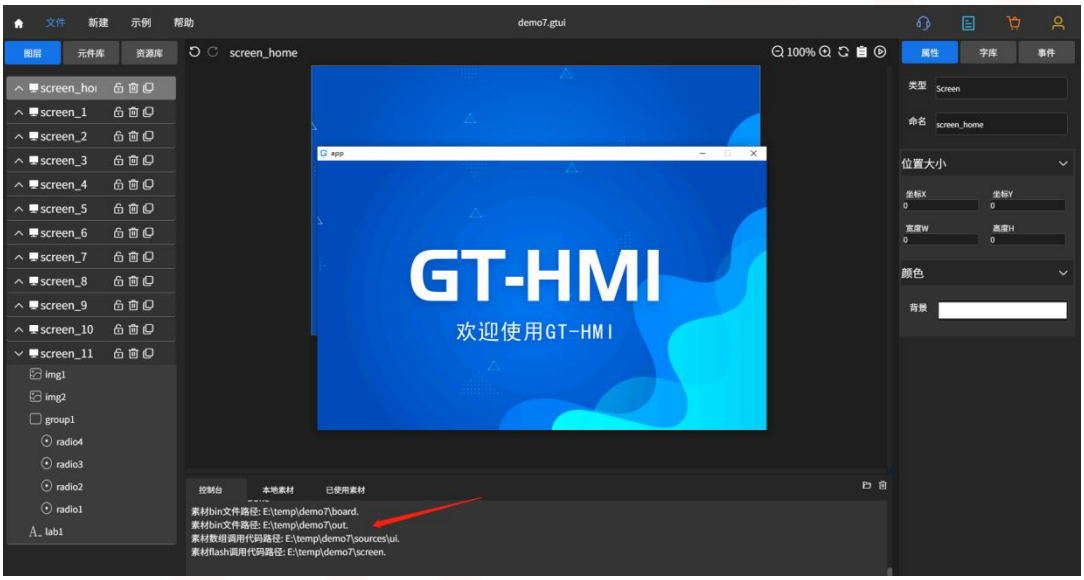


可以看到该示例中使用并展示了多种控件的使用和交互方法，我们在点击仿真运行前，需要先另存为到其它路径。



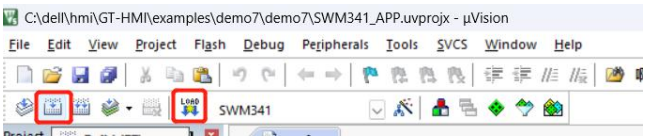
不勾选下次询问时，点击仿真会弹出编译环境设置，我们设置好开发板卡设置与字库配置。

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。



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 7 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。

我们按照 12.3 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。



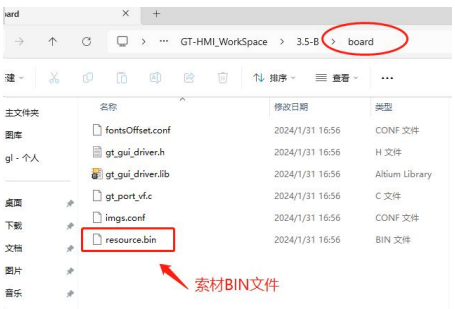
打开 out 文件夹中的 resource.bin 资源文件，使用烧录器将其烧录到板子上的 flash 中。烧录完成后，即可在板子上运行。

12.7 HMI TF/SD 卡升级方法（推荐使用）

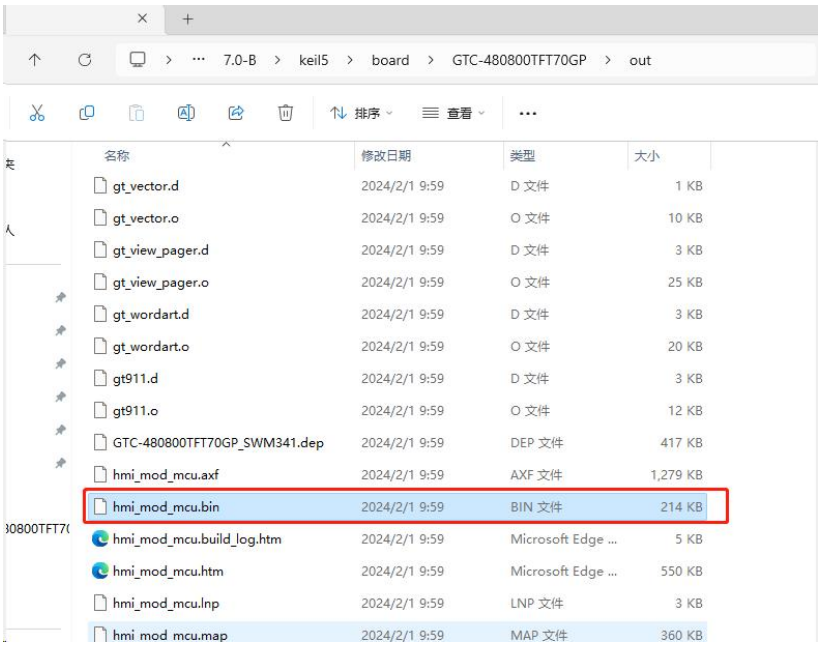
升级步骤：

1: 准备 1 张 Mirco sd 卡。

2：在 SD 卡根目录中新建 gt_loader 文件夹，



3：将工程所在目录\board\resource.bin 和 KEIL 工程所生成的.bin 一起拷贝到 gt_loader 文件夹下，resource.bin 文件是图片资源文件，升级到板载字库芯片当中，KEIL 生成的.bin 文件是升级到板载 MCU 当中。也可以根据需要单独升级其中一个文件。



4：将 keil 工程所生成的.bin 文件：hmi_mod_mcu.bin 一同拷贝到 SD 卡 gt_loader 文件夹

5：将 SD 卡，插入板子 TF 卡槽中，按板子上面的复位按键（reset 键）或重新上电，即可进入升级状态。

6：板子上面的灯开始闪烁代表正在升级。灯闪烁停止表示升级已完成，结束后拔出 Mirco sd 卡。

13 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。

14 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：