

LVGL 移植

（基于 ST7789 芯片）



淘宝店铺：

PC 端：

<http://n-xytrt8gqu585po94mwj5atokcyd4.taobao.com/index.htm>

手机端：

https://shop.m.taobao.com/shop/shop_index.htm?sellerId=755668508&shopId=104493595&inShopPageId=423890608&pathInfo=shop/index2



资料下载地址：

链接：https://pan.baidu.com/s/1kCjD8yktZECsGmHomx_veg?pwd=q8er

提取码：q8er

源码下载地址：

<https://gitee.com/vi-iot/esp32-board.git>

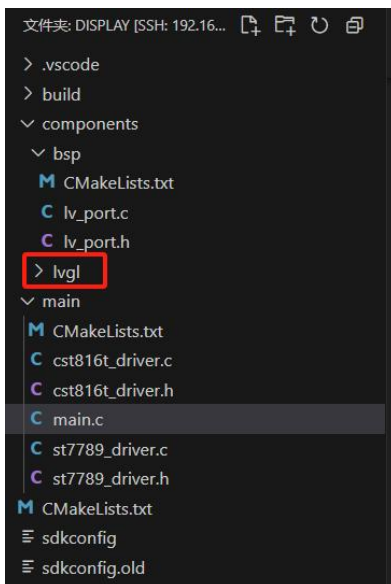
一、前言

本课内容涵盖了 SPI、LCD、LVGL 知识，每一个点单独来讲都是比较丰富的内容，尤其是 LVGL，LVGL（Light and Versatile Graphics Library）是一个开源的图形用户界面（GUI）库，专为嵌入式系统设计，旨在提供轻量级、高度可移植、灵活且易于使用的图形界面解决方案。该库支持在不同的操作系统、微控制器以及图形加速器上运行，非常适合资源有限的嵌入式设备，其源码较多，功能较为庞大，可以单独的开一门课程去讲。那么本节课只涉及到 LVGL 的移植、SPI 和 LCD 驱动接口的使用、以及 demo 演示。

本节课内容也不少，因为 SPI、LCD、LVGL 显示三者在这节课中关系密切，因此我这边合起来一起描述。

二、LVGL 源码获取和移植

大家可以从 <https://github.com/lvgl/lvgl.git> 上获取 LVGL 最新的源码，LVGL 目前最新的源码已更新到 9.X，不同的版本之间接口基本难以兼容，因此我们需要选定一个版本，在这个版本基础上再去开发我们的显示程序，在本例程中使用的是 8.3.10 版本。大家先看 esp32-board/display 目录结构是这样的



被我标红的就是 lvgl 仓库的源码，目前我用 git 子仓库的形式引入，里面的内容我没有改动，只是把版本切换到了 8.3.10，同时大家可能注意到，lvgl 是以组件形式引入到这个工程的（目录解析具体可看第 3 个教程），路径在 display/components/lvgl，另外我们也添加了一个 bsp 组件，bsp 组件中用于存放与板相关的代码，我们把 lvgl 移植相关的代码也放在这里，具体的移植代码位于 lv_port.c 中。现在我们看下 lv_port.c 中具体有什么内容。

先看一下与 LVGL 显示直接相关的初始化函数

```
/**
 * @brief 注册 LVGL 显示驱动
 *
 */
static void lv_port_disp_init(void)
{
    static lv_disp_draw_buf_t draw_buf_dsc;
    size_t disp_buf_height = 40;
```

```

/* 必须从内部 RAM 分配显存，这样刷新速度快 */
lv_color_t *p_disp_buf1 = heap_caps_malloc(LCD_WIDTH * disp_buf_height
* sizeof(lv_color_t), MALLOC_CAP_INTERNAL | MALLOC_CAP_DMA);
lv_color_t *p_disp_buf2 = heap_caps_malloc(LCD_WIDTH * disp_buf_height
* sizeof(lv_color_t), MALLOC_CAP_INTERNAL | MALLOC_CAP_DMA);
ESP_LOGI(TAG, "Try allocate two %u * %u display buffer, size:%u Byte",
LCD_WIDTH, disp_buf_height, LCD_WIDTH * disp_buf_height * sizeof(lv_color_t)
* 2);
if (NULL == p_disp_buf1 || NULL == p_disp_buf2) {
    ESP_LOGE(TAG, "No memory for LVGL display buffer");
    esp_system_abort("Memory allocation failed");
}
/* 初始化显示缓存 */
lv_disp_draw_buf_init(&draw_buf_dsc, p_disp_buf1, p_disp_buf2,
LCD_WIDTH * disp_buf_height);
/* 初始化显示驱动 */
lv_disp_drv_init(&disp_drv);
/*设置水平和垂直宽度*/
disp_drv.hor_res = LCD_WIDTH;
disp_drv.ver_res = LCD_HEIGHT;
/* 设置刷新数据函数 */
disp_drv.flush_cb = disp_flush;
/*设置显示缓存*/
disp_drv.draw_buf = &draw_buf_dsc;
/*注册显示驱动*/
lv_disp_drv_register(&disp_drv);
}

```

在 LVGL 中，要想显示内容，首先得告诉 LVGL 你的显示驱动信息，显示驱动信息包含显示刷新缓存、显示屏的宽高、显示输出函数这三个信息。

`lv_disp_draw_buf_init` 函数用于初始化显示缓存，把用户定义的缓存设置到 `draw_buf_dsc` 显示缓存描述结构体上，这里要注意，我们使用 `esp-idf` 内存管理接口 `heap_caps_malloc` 申请的缓存一定要用 `MALLOC_CAP_INTERNAL | MALLOC_CAP_DMA` 修饰，因为 SPI 传输 RGB 数据用的是 DMA 方式，这种方式无法应用于外部 PSRAM，只能从内部 IRAM 进行申请。

`lv_disp_drv_init` 函数用于按默认配置初始化一个显示驱动 `disp_drv`。

之后我们对 `disp_drv` 这个驱动进行设置，包括显示宽高、缓存、显示数据输出函数，显示数据输出函数 `disp_flush` 是需要我们实现的，当 LVGL 进行完界面的绘画后，最终是要调用这个函数将 RGB 显示数据输出到 LCD 屏上，实现如下

```

/**
 * @brief 写入显示数据
 *
 * @param disp_drv 对应的显示器
 * @param area 显示区域
 * @param color_p 显示数据
 */

```

```
static void disp_flush(lv_disp_drv_t *disp_drv, const lv_area_t *area,
lv_color_t *color_p)
{
    (void) disp_drv;
    st7789_flush(area->x1, area->x2 + 1, area->y1, area->y2 + 1, color_p);
}
```

在 disp_flush 函数中，实际是调用了 st7789 驱动里面的刷新函数，将数据写入到 st7789 中，关于 st7789 的驱动我们在后面一节会讲到。

最后调用 `lv_disp_drv_register` 把这个显示驱动注册到 LVGL 中。

除了要设置显示驱动之外，还需要给 LVGL 设置一个定时器，这个定时器的功能是给 LVGL 做一些动画效果的，比如平移、缩放等，都需要用到这个定时器来进行计时操作。

```
/**
 * @brief LVGL 定时器时钟
 *
 * @param pvParam 无用
 */
static void lv_tick_inc_cb(void *data)
{
    uint32_t tick_inc_period_ms = *((uint32_t *) data);
    lv_tick_inc(tick_inc_period_ms);
}

/**
 * @brief 创建 LVGL 定时器
 *
 * @return esp_err_t
 */
static esp_err_t lv_port_tick_init(void)
{
    static uint32_t tick_inc_period_ms = 5;
    const esp_timer_create_args_t periodic_timer_args = {
        .callback = lv_tick_inc_cb,
        .name = "",
        .arg = &tick_inc_period_ms,
        .dispatch_method = ESP_TIMER_TASK,
        .skip_unhandled_events = true,
    };

    esp_timer_handle_t periodic_timer;
    ESP_ERROR_CHECK(esp_timer_create(&periodic_timer_args,
    &periodic_timer));
    ESP_ERROR_CHECK(esp_timer_start_periodic(periodic_timer,
    tick_inc_period_ms * 1000));
    return ESP_OK;
}
```

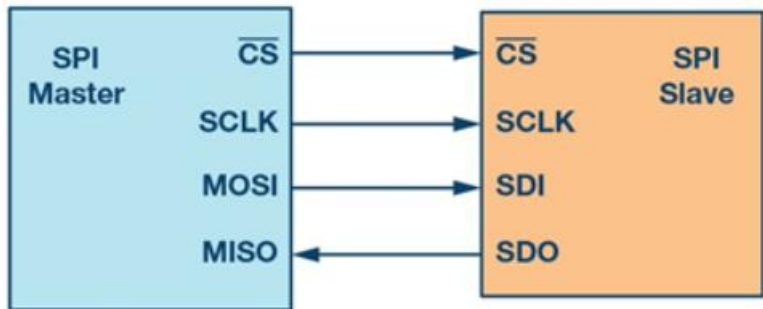
上述代码使用了 esp 定时器，定时周期是 5ms，我们只需要在定时器回调函数中调用 LVGL 的 `lv_tick_inc(tick_inc_period_ms)` 即可。`tick_inc_period_ms` 参数是定时器运行周期，单位是 ms，我们这里设置是 5ms。

另外，在 LVGL 调用完 `disp_flush` 函数往 LCD 写入数据时，会进入等待数据写入完成状态。我们需要手动通知 LVGL 写入数据完成，在写入数据完成后，需要调用 `lv_disp_flush_ready(&disp_drv)` 函数通知 LVGL，否则 LVGL 后续不会再写入数据。具体我们什么时候知道写入数据完成，这就需要我们驱动的支持了。

三、SPI 驱动 ST7789

ST7789 是一款广泛应用于小型至中型彩色 TFT（薄膜晶体管）显示屏的 LCD 控制器/驱动 IC，常见于智能手表、掌上游戏机及各类手持设备中。它支持 SPI（Serial Peripheral Interface）和 8080 并行接口通信，能够驱动分辨率较高的屏幕，如 240x320 像素的显示屏。ST7789 通过提供灵活的显示控制功能，如色彩格式转换、gamma 校正等，提升了显示效果和效率，是许多嵌入式系统和物联网(IoT)项目中进行图形界面设计的优选方案。在本课程中，我们使用 ESP32 的 SPI 接口驱动 ST7789。

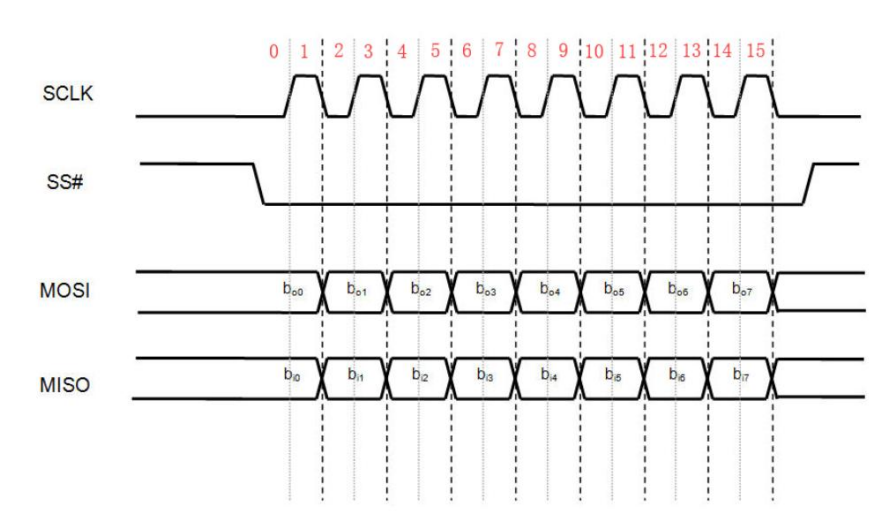
SPI 是一种高速的、全双工的、同步的通信总线，我们先来看一下 SPI 接口以及时序。



上图是 4 线 SPI 接口，线定义如下

引脚	定义
CS	片选（低有效）
SCLK	时钟信号
MOSI	主端输出，从端输入
MISO	主端输入，从端输出

时序如下图所示



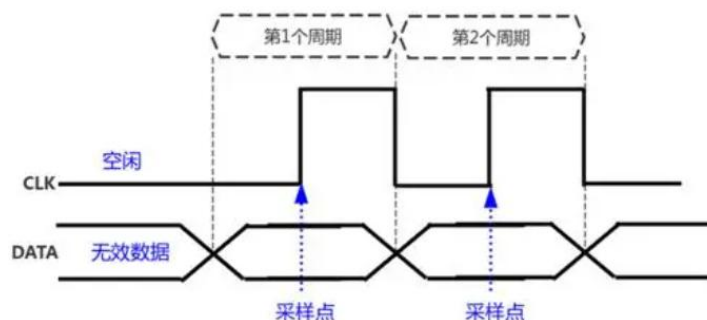
基本上可以描述为，SS 片选信号拉低后，开始工作，数据在 SCLK 的边沿被从端设备采样，那到底是在 SCLK 上升沿被采样还是下降沿被采样，这里就涉及到 4 个 SPI 的工作模式，和两个属性有关，CPOL 以及 CPHA。

CPOL 表示在空闲时 SCLK 的电平

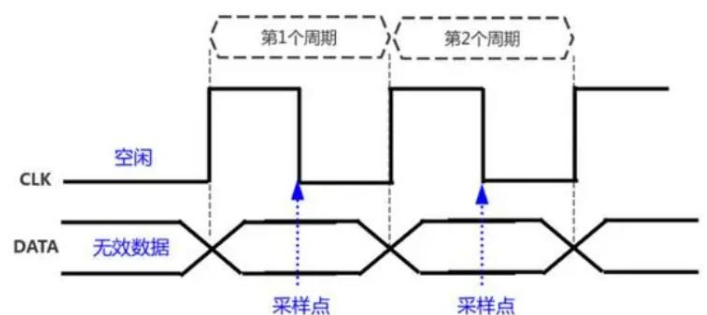
CPHA 表示数据在第几个跳变沿采样（0 是第一个，1 是第二个）

具体的模式请看如下几个图，不同的值时序上会稍有差异

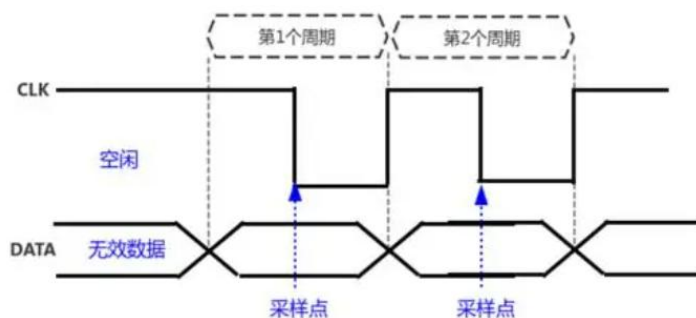
模式 0: (CPOL=0, CPHA=0)



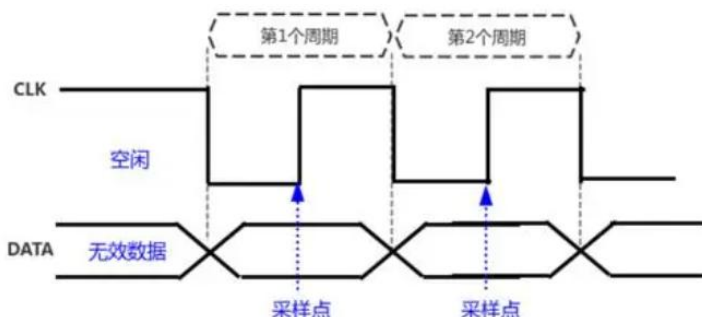
模式 1: (CPOL=0, CPHA=1)



模式 2: (CPOL=1, CPHA=0)



模式 3: (CPOL=1, CPHA=1)



驱动 ST7789 时，ESP32 作为主端，ST7789 为从端，我们不需要用到 MISO 引脚，因为我们不用从 ST7789 中读取数据，只需要往 ST7789 发送数据即可。但需要多增加 3 个引脚来驱动，一个是 DC 引脚，一个是 RST，还有一个是 BL。

引脚	定义
DC	数据/命令选择，0: 命令，1: 数据
RST	ST7789 硬件复位，低有效
BL	背光控制

在介绍完基本的理论知识后，我们要看看 st7789 的驱动怎么写，请查看 esp32-board/display/main/st7789_driver.c，这里不得不吐槽一下有些人写的 ESP32 LCD 驱动，一大堆无注释的代码，有几百个宏定义，然后又是考虑兼容所有屏型号、总线接口和尺寸，代码写得巨复杂，结果拿过来用都没法用，要改也无从下手，可读性非常差，明明是一个非常简单的点屏驱动，不用 200 行代码可以完事的工作。大家在做项目的时候一定要注意适用性，不要总想着考虑所有情况，把所有 LCD 驱动一个.c 文件做完，不现实也无法维护。

现在我们回到 st7789 驱动，驱动里面只做了 3 个事情：

- 1) 使用传入的 GPIO 配置，初始化 SPI 接口
- 2) 使用 SPI 接口向 ST7789 发送一些初始化命令
- 3) 向 LVGL 提供写入 RGB 数据接口

以下是初始化函数

```

/** st7789 初始化
 * @param st7789_cfg_t 接口参数
 * @return 成功或失败
 */
esp_err_t st7789_driver_hw_init(st7789_cfg_t* cfg)

```

```

{
    //初始化 SPI
    spi_bus_config_t buscfg = {
        .sclk_io_num = cfg->clk,          //SCLK 引脚
        .mosi_io_num = cfg->mosi,         //MOSI 引脚
        .miso_io_num = -1,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .flags = SPICOMMON_BUSFLAG_MASTER, //SPI 主模式
        .max_transfer_sz = cfg->width * 40 * sizeof(uint16_t), //DMA 单次
传输最大字节, 最大 32768
    };
    ESP_ERROR_CHECK(spi_bus_initialize(LCD_SPI_HOST, &buscfg,
SPI_DMA_CH_AUTO));

    s_flush_done_cb = cfg->done_cb; //设置刷新完成回调函数
    s_bl_gpio = cfg->bl;           //设置背光 GPIO
    //初始化 GPIO(BL)
    gpio_config_t bl_gpio_cfg =
    {
        .pull_up_en = GPIO_PULLUP_DISABLE,          //禁止上拉
        .pull_down_en = GPIO_PULLDOWN_DISABLE,      //禁止下拉
        .mode = GPIO_MODE_OUTPUT,                   //输出模式
        .intr_type = GPIO_INTR_DISABLE,              //禁止中断
        .pin_bit_mask = (1<<cfg->bl)                //GPIO 脚
    };
    gpio_config(&bl_gpio_cfg);
    //初始化复位脚
    if(cfg->rst > 0)
    {
        gpio_config_t rst_gpio_cfg =
        {
            .pull_up_en = GPIO_PULLUP_DISABLE,          //禁止上拉
            .pull_down_en = GPIO_PULLDOWN_DISABLE,      //禁止下拉
            .mode = GPIO_MODE_OUTPUT,                   //输出模式
            .intr_type = GPIO_INTR_DISABLE,              //禁止中断
            .pin_bit_mask = (1<<cfg->rst)                //GPIO 脚
        };
        gpio_config(&rst_gpio_cfg);
    }
    //创建基于 spi 的 lcd 操作句柄
    esp_lcd_panel_io_spi_config_t io_config = {
        .dc_gpio_num = cfg->dc,          //DC 引脚
        .cs_gpio_num = cfg->cs,          //CS 引脚

```



```

    .pclk_hz = cfg->spi_fre,           //SPI 时钟频率
    .lcd_cmd_bits = 8,                 //命令长度
    .lcd_param_bits = 8,               //参数长度
    .spi_mode = 0,                     //使用 SPI0 模式
    .trans_queue_depth = 10,           //表示可以缓存的 spi 传输事务队列深度
    .on_color_trans_done = notify_flush_ready, //刷新完成回调函数
    .user_ctx = cfg->cb_param,          //回调函数参数

    .flags = { // 以下为 SPI 时序的相关参数，需根据 LCD 驱动 IC 的数据手册以及硬件的配置确定
        .sio_mode = 0, // 通过一根数据线（MOSI）读写数据，0: Interface I 型，1: Interface II 型
    },
};
// Attach the LCD to the SPI bus
ESP_LOGI(TAG, "create esp_lcd_new_panel");
ESP_ERROR_CHECK(esp_lcd_new_panel_io_spi((esp_lcd_spi_bus_handle_t)LCD_SPI_HOST, &io_config, &lcd_io_handle));

//硬件复位
if(cfg->rst > 0)
{
    gpio_set_level(cfg->rst, 0);
    vTaskDelay(pdMS_TO_TICKS(20));
    gpio_set_level(cfg->rst, 1);
    vTaskDelay(pdMS_TO_TICKS(20));
}
/*向 LCD 写入初始化命令*/
esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_SWRESET, NULL, 0);
//软件复位
vTaskDelay(pdMS_TO_TICKS(150));
esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_SLPOUT, NULL, 0);
//退出休眠模式
vTaskDelay(pdMS_TO_TICKS(200));
esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_COLMOD, (uint8_t[]) {0x55,}, 1); //选择 RGB 数据格式，0x55:RGB565,0x66:RGB666
esp_lcd_panel_io_tx_param(lcd_io_handle, 0xb0, (uint8_t[]) {0x00, 0xf0}, 2);
esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_INVON, NULL, 0); //颜色翻转
esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_NORON, NULL, 0); //普通显示模式
uint8_t spin_type = 0;
switch(cfg->spin)

```

```

{
    case 0:
        spin_type = 0x00;    //不旋转
        break;
    case 1:
        spin_type = 0x60;    //顺时针 90
        break;
    case 2:
        spin_type = 0xC0;    //180
        break;
    case 3:
        spin_type = 0xA0;    //顺时针 270, (逆时针 90)
        break;
    default:break;
}
esp_lcd_panel_io_tx_param(lcd_io_handle,LCD_CMD_MADCTL,(uint8_t[])
{spin_type,}, 1);    //屏旋转方向
vTaskDelay(pdMS_TO_TICKS(150));
esp_lcd_panel_io_tx_param(lcd_io_handle,LCD_CMD_DISPON,NULL,0);    /
/打开显示
vTaskDelay(pdMS_TO_TICKS(300));
return ESP_OK;
}

```

因为大部分代码均有注释，这里简单介绍一下内容。esp-idf 提供了丰富的 LCD 底层驱动，我们在初始化的时候进行正确的配置，在我们调用 LCD 接口时，esp-idf 会帮我们自动的切换 DC，准备 buffer，然后通过 DMA 将数据传输到 LCD 上，具体的代码 esp-idf 中已开源，有兴趣的朋友可以自行查看，实现的比较复杂。

首先需要用 `spi_bus_config_t` 配置来初始化 SPI 总线，需要指定 SCLK 和 MOSI 管脚、SPI 模式、DMA 传输字节，通过 `spi_bus_initialize` 函数执行初始化，然后我们初始化 BL 和 RST 引脚 GPIO 模式为输出，这两个引脚完全由我们应用自己控制，尤其是 BL，我们在应用中应当适时关屏，节省功耗。接下来要配置 `esp_lcd_panel_io_spi_config_t` 结构体，设定 DC 和 CS 引脚、SPI 时钟频率、SPI 模式、命令位宽和参数位宽（一般设为 8），比较关键的是 `on_color_trans_done` 成员，这个是回调函数，当底层 LCD 驱动完成 RGB 数据输出时，会调用这个回调函数，说到这里大家是不是联想到了 LVGL 的数据传输完成通知？没错，我们就是要在这个函数上通知到 LVGL 数据传输完成。配置完成后通过 `esp_lcd_new_panel_io_spi` 函数把 spi 和 LCD 驱动关联起来，并且返回一个 LCD 句柄，我们就可以通过这个句柄使用 LCD 的驱动函数了，其实无外乎使用两个函数

1) 传输命令数据

```

esp_err_t esp_lcd_panel_io_tx_param(esp_lcd_panel_io_handle_t io, int
lcd_cmd, const void *param, size_t param_size)

```

2) 传输 RGB 数据

```

esp_err_t esp_lcd_panel_io_tx_color(esp_lcd_panel_io_handle_t io, int
lcd_cmd, const void *color, size_t color_size);

```

接下来就是向 ST7789 发送初始化命令了，再发送命令之前，需要先硬件复位一下，操作 RST 引脚拉低再拉高就可以了，初始化命令包含软复位、退出休眠、选择显示模式（RGB565）、颜色翻转、普通显示模式、屏旋转方向、启动显示，具体初始化需要发送什么命令，一般来说供应商会给出代码示例，我们参照即可，我这里发送的这些命令主要是参考 st7789 驱动 IC 的 Datasheet 来写的，用于驱动开发板上的 1.69 寸 ips 屏没有问题。

接下来看下写入 RGB 数据的函数，给 LVGL 提供的一个刷入 RGB 的数据的接口

```
/** st7789 写入显示数据
 * @param x1,x2,y1,y2:显示区域
 * @return 无
 */
void st7789_flush(int x1,int x2,int y1,int y2,void *data)
{
    // define an area of frame memory where MCU can access
    if(x2 <= x1 || y2 <= y1)
    {
        if(s_flush_done_cb)
            s_flush_done_cb(NULL);
        return;
    }
    esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_CASET, (uint8_t[]) {
        (x1 >> 8) & 0xFF,
        x1 & 0xFF,
        ((x2 - 1) >> 8) & 0xFF,
        (x2 - 1) & 0xFF,
    }, 4);
    esp_lcd_panel_io_tx_param(lcd_io_handle, LCD_CMD_RASET, (uint8_t[]) {
        (y1 >> 8) & 0xFF,
        y1 & 0xFF,
        ((y2 - 1) >> 8) & 0xFF,
        (y2 - 1) & 0xFF,
    }, 4);
    // transfer frame buffer
    size_t len = (x2 - x1) * (y2 - y1) * 2;
    esp_lcd_panel_io_tx_color(lcd_io_handle, LCD_CMD_RAMWR, data, len);
    return ;
}
```

这个函数很简单，首先写入行列起始和结束坐标，也就是定义写入区域（通过命令接口），然后写入 RGB 数据即可。

最后我们看一下，写入完成回调函数

```
static bool notify_flush_ready(esp_lcd_panel_io_handle_t panel_io,
esp_lcd_panel_io_event_data_t *edata, void *user_ctx)
{
    if(s_flush_done_cb)
```

```
s_flush_done_cb(user_ctx);
return false;
}
```

当底层 LCD 驱动写入完成后，会调用 `notify_flush_ready` 函数，而我们可以在这个函数中通知 LVGL 刷入数据完成，在这里通过上层设置回调函数的方式去调用 LVGL 通知函数。

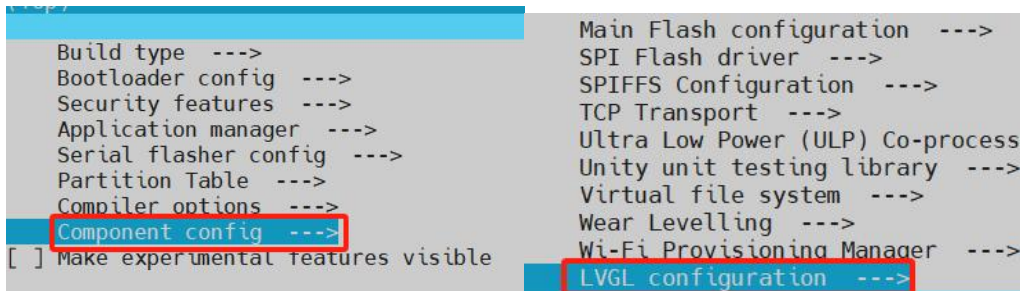
现在我们回到 `display/components/bsp/lv_port.c` 文件，看下 `st7789` 的驱动初始化是怎样的。

```
/**
 * @brief LCD 接口初始化
 *
 * @return NULL
 */
static void lcd_init(void)
{
    st7789_cfg_t st7789_config;
    st7789_config.mosi = GPIO_NUM_19;
    st7789_config.clk = GPIO_NUM_18;
    st7789_config.cs = GPIO_NUM_5;
    st7789_config.dc = GPIO_NUM_17;
    st7789_config.rst = GPIO_NUM_21;
    st7789_config.bl = GPIO_NUM_26;
    st7789_config.spi_fre = 40*1000*1000; //SPI 时钟频率
    st7789_config.width = LCD_WIDTH; //屏宽
    st7789_config.height = LCD_HEIGHT; //屏高
    st7789_config.spin = 1; //顺时针旋转 90 度
    st7789_config.done_cb = lv_port_flush_ready; //数据写入完成回调函数
    st7789_config.cb_param = &disp_drv; //回调函数参数

    st7789_driver_hw_init(&st7789_config);
}
```

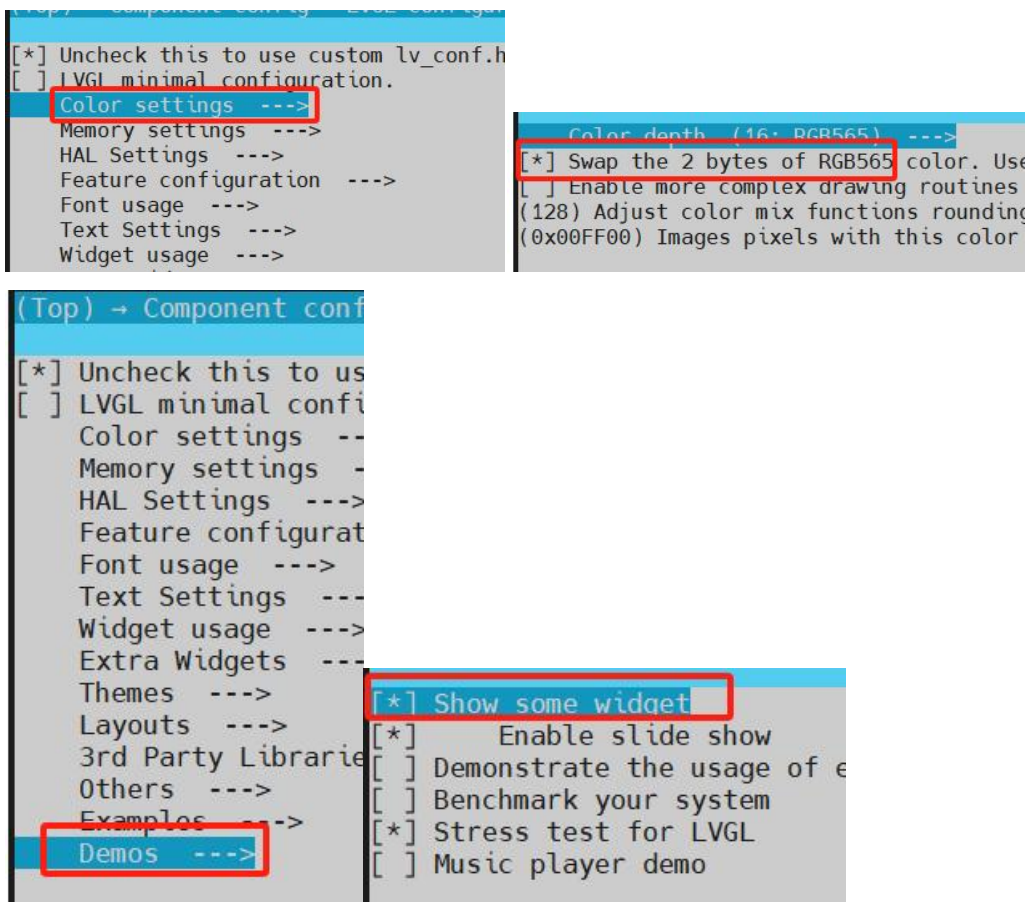
在这个函数中我们把 `lv_port_flush_ready` 函数设置为 LCD 底层驱动数据传输完成回调函数，这样就可以通知到 LVGL 数据传输完成了。

接下来我们还需要对 `lvgl` 组件进行配置，在 `display` 目录下，执行 `idf.py menuconfig`，配置如下几项。



```
Build type --->
Bootloader config --->
Security features --->
Application manager --->
Serial flasher config --->
Partition Table --->
Compiler options --->
Component config --->
[ ] Make experimental features visible

Main Flash configuration --->
SPI Flash driver --->
SPIFFS Configuration --->
TCP Transport --->
Ultra Low Power (ULP) Co-process
Unity unit testing library --->
Virtual file system --->
Wear Levelling --->
Wi-Fi Provisioning Manager --->
LVGL configuration --->
```



在 main.c 文件中，lvgl 示例代码如下：

```
// 主函数
void app_main(void)
{
    lv_port_init();           //初始化 LVGL
    lv_demo_widgets();        //初始化控件 demo 程序
    //lv_demo_stress();
    st7789_lcd_backlight(true); //打开背光
    while(1)
    {
        vTaskDelay(pdMS_TO_TICKS(10));
        lv_task_handler();    //LVGL 循环处理
    }
}
```

lv_task_handler 函数是 lvgl 专门用于处理各种事件、画面更新等操作的函数，需要放在一个无限循环中执行，具体的其他代码，大家请查看 esp32-board/display 工程。我们通过 idf.py build 编译，然后通过 idf.py flash 烧录到开发板上之后，就可以看见屏被点亮并且显示 lvgl 的一个 demos 了。

注意！！开发板需要短接背光跳针才能亮屏！