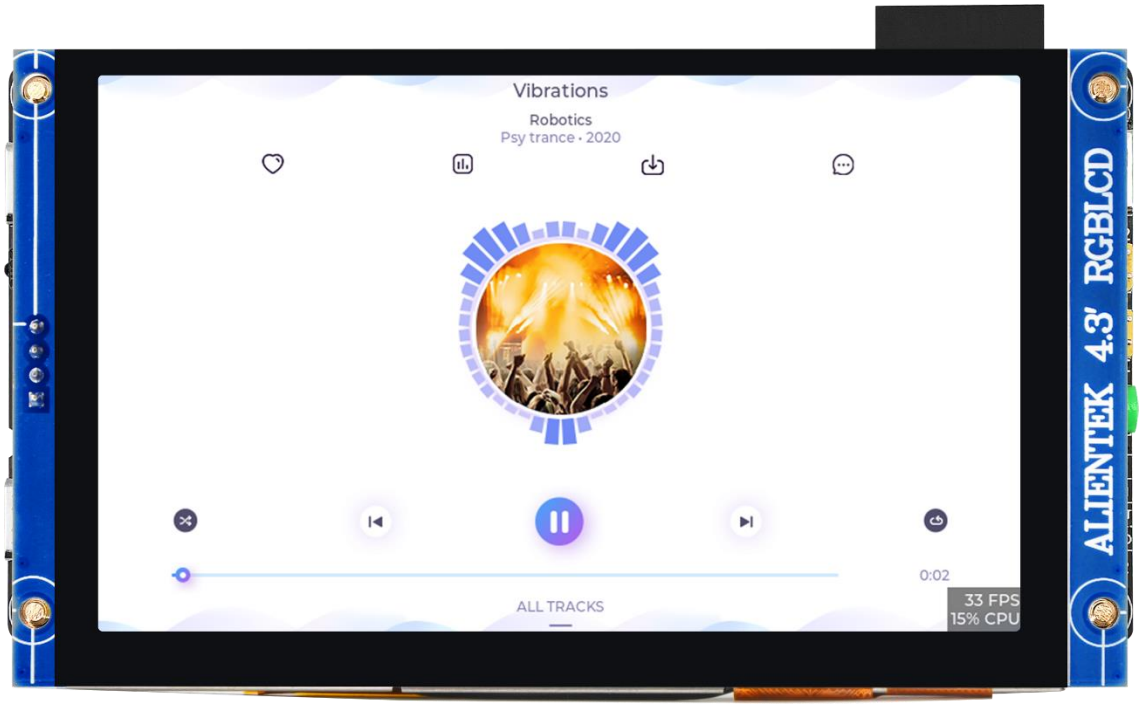
LVGL 移植教程

ESP32-S3 开发板



本教程适用于正点原子 DNESP32S3 开发板

修订历史：

版本	日期	修改内容
V1.0	2024/2/29	第一次发布

正点原子公司名称 ：广州市星翼电子科技有限公司

原子哥在线教学平台：www.yuanzige.com

开源电子网 / 论坛：www.openedv.com/forum.php

正点原子官方网站 ：www.alientek.com

正点原子淘宝店铺 ：https://openedv.taobao.com

正点原子 B 站视频 ：https://space.bilibili.com/394620890

电话：020-38271790 传真：020-36773971

请下载原子哥 APP，数千讲视频免费学习，更快更流畅。

请关注正点原子公众号，资料发布更新我们会通知。

扫码下载“原子哥”APP

扫码关注正点原子公众号

LVGL 系统移植

本章主要讲解 ESP32-S3 开发板的 LVGL 移植，乐鑫 SDK 自带 FreeRTOS，LVGL 也是支持操作系统的。

本章节将分为以下 6 个小节：

1.1 LVGL 源码下载
1.2 拷贝 LVGL 文件夹到工程
1.3 编写 LVGL 相关代码
1.4 调用接口函数
1.5 修改 LVGL Configuration 配置
1.6 下载验证

1.1 LVGL 源码下载

LVGL 相关的源码和工程都是存放在 GitHub 远程仓库中，该 GitHub 远程仓库地址为 https://github.com/lvgl/lvgl/，用户可以该仓库中下载 LVGL 图形库的源码。由于 GitHub 仓库的服务器在国外，如果用户在国内访问该服务器，可能登录不成功，此时，我们可以从正点原子光盘资料中获取 LVGL 的 V8.3 版本源码，具体路径为：“A 盘→6，软件资料→7，LVGL 学习资料→ lvgl-release-v8.3.zip”，如下图所示：

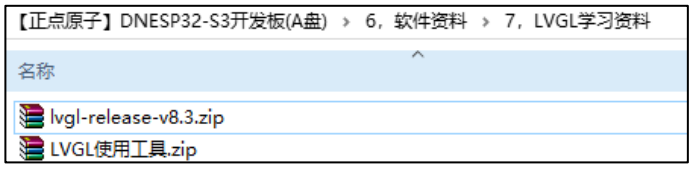


图 1.1.1 LVGL 源码

上图中，“LVGL 使用工具.zip”压缩包里存放了 LVGL 相关的离线转换工具，这些离线工具是广大爱好者根据 LVGL 的字库定义规则和图片的处理特性而编写的软件；“lvgl-release-v8.3.zip”压缩包里存放了 LVGL 图形库的 V8.3 版本源码，其解压后如下图所示：

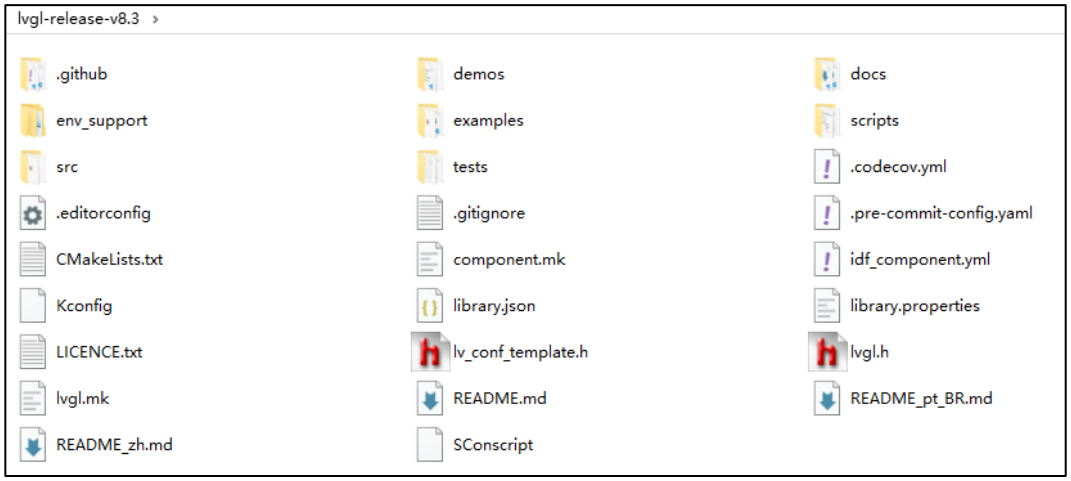


图 1.1.2 LVGL 源码文件

由上图可知，LVGL 源码的目录下有很多文件和文件夹，但用户并不需要完全了解它们，我们只需要了解与移植相关的部分即可。各文件夹和文件的功能如下表所示：

文件	说明
demos	LVGL 提供的综合演示源码
docs	LVGL 文献，主要说明 LVGL 每个部件的使用方法
env_support	环境的支持（MDK、ESP、RTThread）
examples	LVGL 例程源码和 LVGL 输入设备驱动，显示屏驱动文件
scripts	LVGL 手稿（与 MicroPython 有关）
src	LVGL 源文件（LVGL 部件源码、第三方库）
tests	官方人员的测试代码，该文件夹用户无需了解
lv_conf_template.h	LVGL 的剪裁文件
lvgl.h	LVGL 包含的头文件

表 1.1.1 lvgl-release-v8.3 文件说明

上表中，与 LVGL 移植相关的有 examples 文件夹、src 文件夹、lv_conf_template.h 和 lvgl.h 文件，其他的部分均与移植无关，用户可以选择忽略。

接下来我们分别看一下 examples、src 这两个文件夹的文件结构：

1. examples 文件夹

该文件夹主要包含 LVGL 部件实例、动画实例、其他第三方库实例以及输入设备和显示器驱动文件等内容，具体如表 1.1.2 所示：

文件	描述
anim	LVGL 动画例程实例
arduino	开源电子平台
assets	图片资源
event	LVGL 事件机制实例
get_started	LVGL 获取状态实例
layouts	LVGL 布局实例
libs	LVGL 移植第三方库实例
others	LVGL 其他测试
porting	LVGL 输入设备驱动、文件系统驱动以及显示器驱动
scroll	LVGL 滚动实例
styles	LVGL 对象样式实例
widgets	LVGL 部件实例

表 1.1.2 examples 文件夹的内容

上表中，只有 porting 文件夹与移植相关，其他文件夹中存放的是各种实例。

2. src 文件夹

该文件夹主要包含 LVGL 源文件（部件源码、多种解码库），具体如表 1.1.3 所示：

文件	描述
core	LVGL 核心源码（事件、组、对象、坐标、样式、主题）
draw	LVGL 绘画驱动（图片、解码、DMA2D、圆、线、圆弧、和文本）
extra	LVGL 的拓展内容（布局、第三方库、其他测试、主题以及部件）
font	LVGL 字库
gpu	LVGL 针对图形加速
hal	硬件抽象层（显示驱动程序、输入设备程序以及 LVGL 系统滴答）
misc	主要描述 LVGL 其他定义（动画、内存管理、日志）
widgets	LVGL 基础部件

表 1.1.3 src 文件夹的内容

上表中的内容都是与移植相关的，具体的移植方法我们后面将详细介绍，目前大家只需要对 LVGL 源码的文件结构有一定了解即可。

1.2 拷贝 LVGL 文件夹到工程

以 IDF 版本的 24_touch 例程作为移植 LVGL 的基础例程。

首先把“lvgl-release-v8.3.zip”在工程的 components 文件夹下解压出来，并重命名为“LVGL”，然后删除“lvgl-release-v8.3.zip”文件。

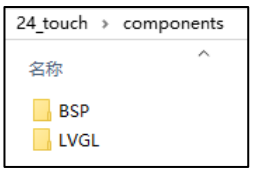


图 1.2.1 添加 LVGL 文件夹进工程中

1.3 编写 LVGL 相关代码

在 main 文件夹下，新建 APP 文件夹，并新建两个文件：lvgl_demo.c、lvgl_demo.h，如下图所示：

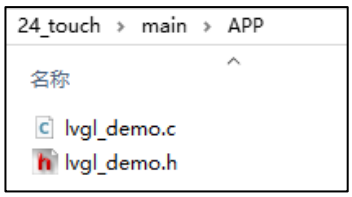


图 1.3.1 APP 文件夹下情况

这两个文件用于存放 LVGL 移植相关的代码，主要就是配置显示屏和触摸输入驱动。

1. lvgl_demo.c

首先看一下显示屏相关的代码，具体源码如下：

```
/**
 * @brief   初始化并注册显示设备
 * @param   无
 * @retval  无
 */
void lv_port_disp_init(void)
{
    void *buf1 = NULL;
    void *buf2 = NULL;

    /* 初始化显示设备 RGBLCD */
    ltdc_init();                    /* RGB 屏初始化 */
    ltdc_display_dir(1);            /* 设置横屏 */

    /*-----------------------------
     * 创建一个绘图缓冲区
     *-----------------------------*/
    /**
     * LVGL 需要一个缓冲区用来绘制小部件
     * 随后，这个缓冲区的内容会通过显示设备的 'flush_cb'(显示设备刷新函数) 复制到显示设备上
```

```
 * 这个缓冲区的大小需要大于显示设备一行的大小
 *
 * 这里有 3 种缓冲配置:
 * 1. 单缓冲区:
 *      LVGL 会将显示设备的内容绘制到这里，并将他写入显示设备。
 *
 * 2. 双缓冲区:
 *      LVGL 会将显示设备的内容绘制到其中一个缓冲区，并将他写入显示设备。
 *      需要使用 DMA 将要显示在显示设备的内容写入缓冲区。
 *      当数据从第一个缓冲区发送时，它将使 LVGL 能够将屏幕的下一部分绘制到另一个缓冲区。
 *      这样使得渲染和刷新可以并行执行。
 *
 * 3. 全尺寸双缓冲区
 *      设置两个屏幕大小的全尺寸缓冲区，并且设置 disp_drv.full_refresh = 1。
 *      这样，LVGL 将始终以 'flush_cb' 的形式提供整个渲染屏幕，只需更改帧缓冲区地址。
 */
/* 使用双缓冲 */
buf1 =heap_caps_malloc(ltdcdev.width*60*sizeof(lv_color_t),MALLOC_CAP_DMA);
buf2 =heap_caps_malloc(ltdcdev.width*60*sizeof(lv_color_t),MALLOC_CAP_DMA);

/* 初始化显示缓冲区 */
static lv_disp_draw_buf_t disp_buf;     /* 保存显示缓冲区信息的结构体 */
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&disp_buf, buf1, buf2, ltdcdev.width * 60);

/* 在 LVGL 中注册显示设备 */
static lv_disp_drv_t disp_drv;
/* 显示设备的描述符(要注册的显示驱动程序、与显示交互并处理与图形相关的结构体、回调函数)*/
lv_disp_drv_init(&disp_drv);            /* 初始化显示设备 */

/* 设置显示设备的分辨率
 * 这里为了适配正点原子的多款屏幕，采用了动态获取的方式，
 * 在实际项目中，通常所使用的屏幕大小是固定的，因此可以直接设置为屏幕的大小
 */
disp_drv.hor_res = ltdcdev.width;
disp_drv.ver_res = ltdcdev.height;

/* 用来将缓冲区的内容复制到显示设备 */
disp_drv.flush_cb = lvgl_disp_flush_cb;

/* 设置显示缓冲区 */
disp_drv.draw_buf = &disp_buf;
```

```
    disp_drv.user_data = panel_handle;

    /* 注册显示设备 */
    lv_disp_drv_register(&disp_drv);
}

/**
* @brief    将内部缓冲区的内容刷新到显示屏上的特定区域
* @note     可以使用 DMA 或者任何硬件在后台加速执行这个操作
*           但是，需要在刷新完成后调用函数 'lv_disp_flush_ready()'
* @param    disp_drv : 显示设备
* @param    area : 要刷新的区域，包含了填充矩形的对角坐标
* @param    color_map : 颜色数组
* @retval   无
*/
static void lvgl_disp_flush_cb(lv_disp_drv_t *drv, const lv_area_t *area,
lv_color_t *color_map)
{
    esp_lcd_panel_handle_t panel_handle=(esp_lcd_panel_handle_t)drv->user_data;

    /* 特定区域打点 */
    esp_lcd_panel_draw_bitmap(panel_handle, area->x1, area->y1, area->x2 + 1,
                              area->y2 + 1, color_map);

    /* 重要!!! 通知图形库，已经刷新完毕了 */
    lv_disp_flush_ready(drv);
}
```

将显示屏驱动与 LVGL 底层显示接口关联上的步骤如下：

（1）调用 ltdc_init 函数初始化 RGB 屏

（2）根据芯片实际情况选择缓冲配置，通过 heap_caps_malloc 函数去申请内存

（3）调用 lv_dis_draw_buf_init 函数初始化显示缓冲区

（4）调用 lv_disp_drv_init 函数初始化显示设备

（5）设置显示设备的分辨率（高度和宽度）

（6）注册显示驱动回调函数 lvgl_disp_flush_cb（回调函数的参数是固定的，内部主要调用打点函数实现）

（7）设置显示驱动的缓冲区 disp_buf

（8）调用 lv_disp_drv_register 函数注册显示驱动到 LVGL 列表中

在整个配置的过程中，实际上我们只需要提供两个函数：LCD 初始化函数和 LCD 填充函数。如果想要设置屏幕的方向，则调用 ltdc_display_dir 函数即可。

接下来看一下触摸屏相关的代码，具体源码如下：

```
/**
 * @brief   初始化并注册输入设备
 * @param   无
```

```
 * @retval  无
 */
void lv_port_indev_init(void)
{
    /* 初始化触摸屏 */
    tp_dev.init();

    /* 初始化输入设备 */
    static lv_indev_drv_t indev_drv;
    lv_indev_drv_init(&indev_drv);

    /* 配置输入设备类型 */
    indev_drv.type = LV_INDEV_TYPE_POINTER;

    /* 设置输入设备读取回调函数 */
    indev_drv.read_cb = touchpad_read;

    /* 在 LVGL 中注册驱动程序，并保存创建的输入设备对象 */
    lv_indev_t *indev_touchpad;
    indev_touchpad = lv_indev_drv_register(&indev_drv);
}

/**
 * @brief   获取触摸屏设备的状态
 * @param   无
 * @retval  返回触摸屏设备是否被按下
 */
static bool touchpad_is_pressed(void)
{
    tp_dev.scan(0);      /* 触摸按键扫描 */
    if (tp_dev.sta & TP_PRES_DOWN)
    {
        return true;
    }

    return false;
}

/**
 * @brief   在触摸屏被按下的时候读取 x、y 坐标
 * @param   x    : x 坐标的指针
 * @param   y    : y 坐标的指针
 * @retval  无
```

```
 */
static void touchpad_get_xy(lv_coord_t *x, lv_coord_t *y)
{
    (*x) = tp_dev.x[0];
    (*y) = tp_dev.y[0];
}

/**
 * @brief   图形库的触摸屏读取回调函数
 * @param   indev_drv   : 触摸屏设备
 * @param   data        : 输入设备数据结构体
 * @retval  无
 */
void touchpad_read(lv_indev_drv_t *indev_drv, lv_indev_data_t *data)
{
    static lv_coord_t last_x = 0;
    static lv_coord_t last_y = 0;

    /* 保存按下的坐标和状态 */
    if(touchpad_is_pressed())
    {
        touchpad_get_xy(&last_x, &last_y);  /* 在触摸屏被按下的时候读取 x、y 坐标 */
        data->state = LV_INDEV_STATE_PR;
    }
    else
    {
        data->state = LV_INDEV_STATE_REL;
    }

    /* 设置最后按下的坐标 */
    data->point.x = last_x;
    data->point.y = last_y;
}
```

将触摸屏驱动与 LVGL 底层输入接口关联上的步骤如下：

（1）调用 tp_dev.init 函数实际上是调用 tp_init 函数初始化触摸设备

（2）调用 lv_indev_drv_init 函数初始化输入设备

（3）设置设备的类型为 LV_INDEV_TYPE_POINTER

（4）设置触摸回调函数 touchpad_read，该函数用于获取触摸屏的坐标

（5）调用 lv_indev_drv_register 注册输入设备

最后，编写 lvgl_demo 入口函数，源码如下：

```
/**
 * @brief   lvgl_demo 入口函数
 * @param   无
```

```
 * @retval  无
 */
void lvgl_demo(void)
{
    lv_init();                  /* 初始化 LVGL 图形库 */
    lv_port_disp_init();        /* lvgl 显示接口初始化,放在 lv_init()的后面 */
    lv_port_indev_init();       /* lvgl 输入接口初始化,放在 lv_init()的后面 */

    /* 为 LVGL 提供时基单元 */
    const esp_timer_create_args_t lvgl_tick_timer_args = {
        .callback = &increase_lvgl_tick,
        .name = "lvgl_tick"
    };
    esp_timer_handle_t lvgl_tick_timer = NULL;
    ESP_ERROR_CHECK(esp_timer_create(&lvgl_tick_timer_args, &lvgl_tick_timer));
    ESP_ERROR_CHECK(esp_timer_start_periodic(lvgl_tick_timer, 1 * 1000));

    /* 官方 demo,需要在 SDK Configuration 中开启对应 Demo */
    lv_demo_music();
    // lv_demo_benchmark();
    while (1)
    {
        lv_timer_handler();             /* LVGL 计时器 */
        vTaskDelay(pdMS_TO_TICKS(10));  /* 延时 10 毫秒 */
    }
}

/**
 * @brief   告诉 LVGL 运行时间
 * @param   arg : 传入参数(未用到)
 * @retval  无
 */
static void increase_lvgl_tick(void *arg)
{   /* 告诉 LVGL 已经过了多少毫秒 */
    lv_tick_inc(1);
}
```

该函数的作用是启动 LVGL，首先调用 lv_init 函数对 lvgl 系统初始化，调用 lv_port_disp_init 函数初始化 lvgl 显示接口，调用 lv_port_indev_init 初始化 lvgl 输入接口，然后为 LVGL 提供时基单元（创建一个定时器唯 LVGL 所用，定时周期为 1ms）。最后调用 lv_music_demo 函数运行音乐例程，在 while 循环中，调用 lv_timer_handler 函数和延时函数。lv_timer_handler 相当于 RTOS 触发任务调度函数。

注意：如果使用的是其他的官方例程，请根据实际的例程来包含头文件以及调用 demo 函数，上述代码中以音乐播放器为例。

2. lvgl_demo.h

```
#ifndef __LVGL_DEMO_H
#define __LVGL_DEMO_H

#include "lvgl.h"

/* 函数声明 */
void lvgl_demo(void);
/* lvgl_demo 入口函数 */
void lv_port_disp_init(void);
/* 初始化并注册显示设备 */
void lv_port_indev_init(void);
/* 初始化并注册输入设备 */
void touchpad_read(lv_indev_drv_t *indev_drv, lv_indev_data_t *data);
/* 图形库的触摸屏读取回调函数 */
static void increase_lvgl_tick(void *arg);
/* 告诉 LVGL 运行时间 */
static void lvgl_disp_flush_cb(lv_disp_drv_t *drv, const lv_area_t *area,
lv_color_t *color_map);    /* 将内部缓冲区的内容刷新到显示屏上的特定区域 */

#endif
```

这个文件很简单，只是声明了 LVGL 的接口函数。

1.4 调用接口函数

接下来只需要在 main.c 文件中调用 LVGL 的接口函数 lvgl_demo 即可，具体源码如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "nvs_flash.h"
#include "led.h"
#include "xl9555.h"
#include "lvgl_demo.h"

i2c_obj_t i2c0_master;

/**
 * @brief   程序入口
 * @param   无
 * @retval  无
 */
```

```
void app_main(void)
{
    esp_err_t ret;

    ret = nvs_flash_init();                 /* 初始化 NVS */

    if (ret==ESP_ERR_NVS_NO_FREE_PAGES || ret==ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    i2c0_master = iic_init(I2C_NUM_0);  /* 初始化 IIC0 */
    xl9555_init(i2c0_master);           /* 初始化 XL9555 */

    lvgl_demo();                        /* 运行 LVGL 例程 */
}
```

这里需要注意的是相关头文件要包含进来，比如要#include “lvgl_demo.h”。然后点击编译，这时候就会提示出“12 号以及 16 号字体没有定义”，“lv_demo_music 函数找不到”，我们就需要打开“SDK Configuration”配置菜单，下滑到LVGL Configuration处对字体以及Demo使能。

1.5 修改 LVGL Configuration 配置

1．选择字体库

图 1.5.1 使能 Music_Demo 涉及到的字体

2．使能 music demo

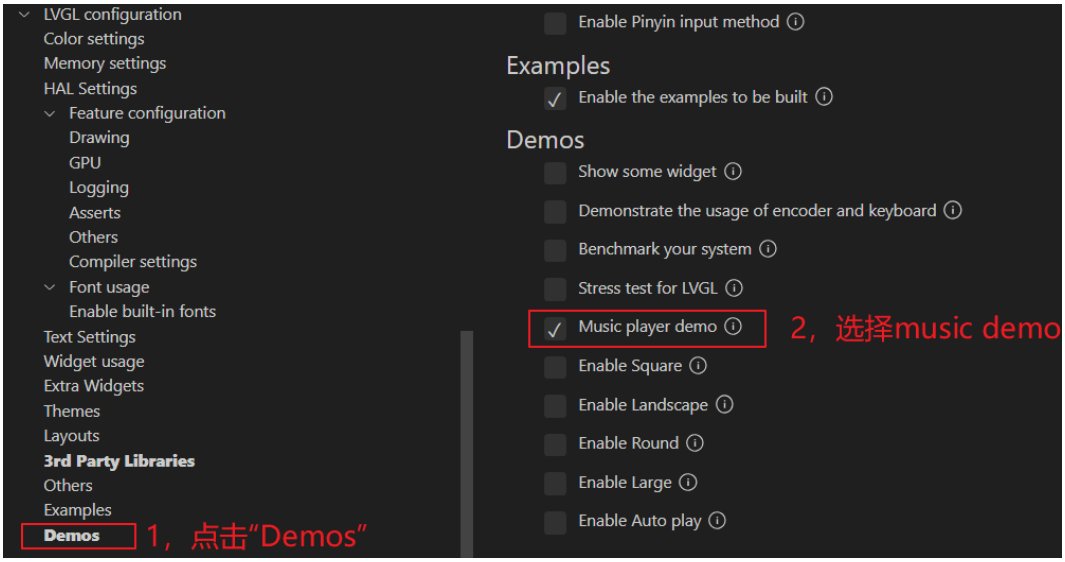


图 1.5.2 使能 Music_Demo

如果内存够，你也可以使能其他 Demo，其他 Demo 也有涉及到其他字号，勾选即可。

1.6 下载验证

编译工程并下载到开发板中，代码的运行界面如下图所示：

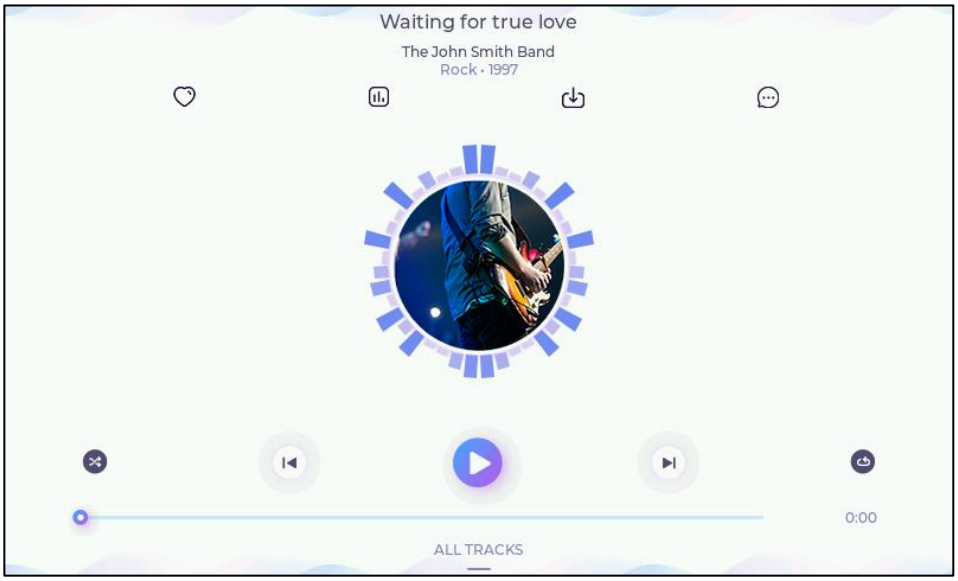


图 1.6.1 音乐播放器界面

LVGL 移植的相关知识

在前面的章节里面，我们已经成功地移植了 LVGL 到开发板当中，但并没有介绍移植过程中的一些知识点，所以本章主要讲解 LVGL 移植相关的一些知识点。

本章节将分为以下几个小节：

2.1 LVGL 初始化流程
2.2 LVGL_configuration 说明
2.3 显示设备
2.4 输入设备
2.5 LVGL 时基
2.6 LVGL 任务处理
2.7 拓展知识

2.1 LVGL 初始化流程

在学习 LVGL 移植相关知识之前，我们需要先简单地了解一下 LVGL 的初始化流程，这样即可知道整个初始化过程中所涉及的一些关键配置，而这些关键配置是在移植过程中需要重点关注的。LVGL 初始化流程分为以下几个步骤：

第一步：调用 lv_init 函数，初始化 LVGL 图形库。 在这一步的初始化中，涉及的内容非常的多，包括：内存管理、文件系统、定时器，等等。

第二步：调用 lv_port_disp_init 函数和 lv_port_indev_init 函数，注册显示设备和输入设备。

注意： 在注册显示设备和输入设备之前，必须先初始化 LVGL 图形库（调用 lv_init 函数）。

第三步：为 LVGL 提供时基。用户可以使用定时器，在其中断里面定时调用 lv_tick_inc 函数，为 LVGL 提供时基。如果工程中带有 OS 操作系统，则可以使用相应的时钟函数来为 LVGL 提供时基。

第四步：定时处理 LVGL 任务。用户需要每隔几毫秒调用一次 lv_timer_handler 函数，以处理 LVGL 相关的任务，该函数可以放在 while 循环中，但延时不宜过大，确保 5 毫秒以内。

2.2 LVGL_configuration 说明

LVGL_configuaration 已经在 SDK Configuration 页面中，它的作用就是对 LVGL 进行设置，在 STM32 开发 LVGL 时会使用到 lv_conf.h，而现在直接是页面的方式进行配置，更加人性化。

接下来，看一下 LVGL_configuaration 的配置项，如下图所示。

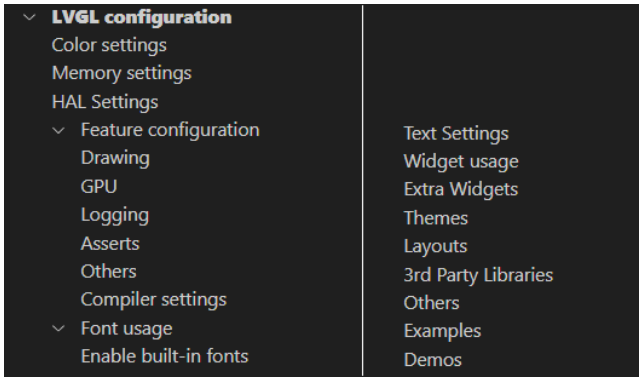


图 2.2.1 LVGL Configuration 配置项

为了更好了解这些配置项，在这里专门弄了一个表格进行说明，如下表所示。

板块介绍	描述
颜色设置	设置颜色深度、字节交换、屏幕透明等
内存设置	对内存管理算法选择、设置内存分配大小等
硬件抽象层设置	设置显示刷新周期、输入设备读取周期等
特征设置	设置绘图、日志、帧率显示等
字体设置	开启系统字体、配置自定义字体等
文本设置	设置字符编码、文本特性
部件设置	基础控件的支持与使能
额外部件	对额外的高级控件的支持与使能
主题设置	自带的一些主题
布局设置	Flexbox 和 Grid 布局设置
第三方库设置	开启第三方库
其他设置	快照、搞怪测试、拼音输入法等方法使能
例程构建	开启/关闭构建实例功能
实例设置	开启/关闭 LVGL 演示实例

表 2.2.1 lvgl_configuration 文件内容描述

接下来，我们重点讲解 lvgl_configuration 文件中几个比较重要板块内容。

1. 颜色设置

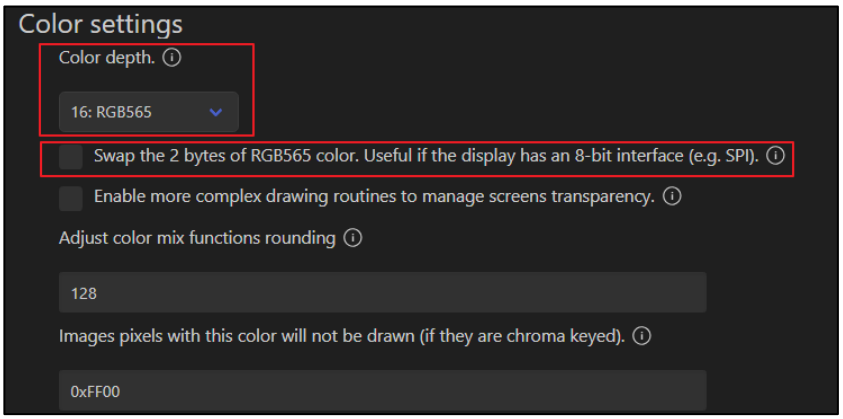


图 2.2.2 颜色设置界面

在颜色设置界面，我们一般情况下只会关注两个地方。

Color depth 可用于设置颜色的深度，用户根据实际的显示屏颜色深度进行配置即可

Swap the 2 bytes of RGB565 color 可用于交换 2 字节的 RGB565 颜色，如果用户使用的是 SPI 的屏幕，发现颜色出现异常，可以尝试将该宏置 1。

2. 内存设置

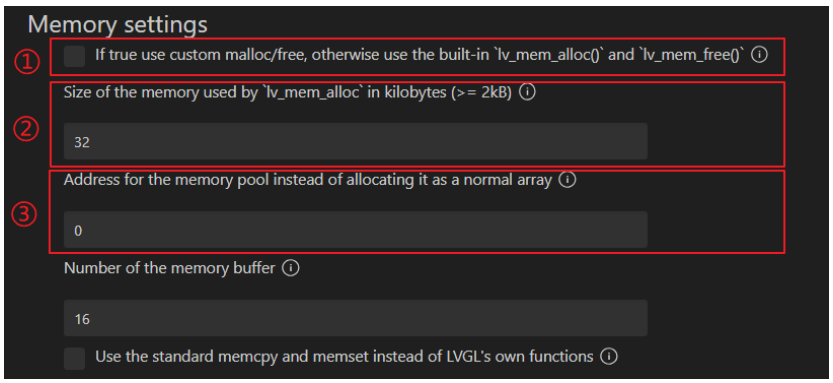


图 2.2.3 内存设置界面

上图这种的内存分配方式，选项①决定了申请内存是使用 lv_mem_alloc，释放内存是使用 lv_mem_free，而 lv_mem 的内存池大小就为选项②中的设置值 32KB。选项③决定了使用的是 LVGL 的内存池。

Number of the memory buffer 为在渲染和其他内部处理机制期间使用的中间内存缓冲区的数量。

3. HAL(硬件抽象层)设置

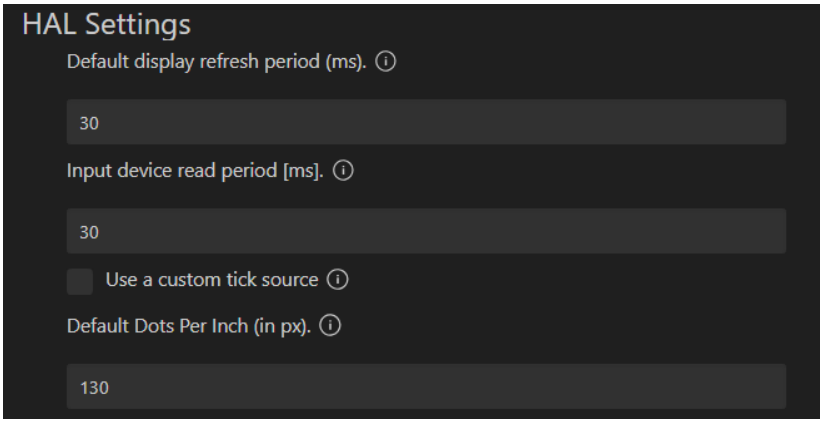


图 2.2.4 HAL(硬件抽象层)界面

在上图中，Default display refresh period 用于配置显示设备的刷新周期，Input device read period 用于配置输入设备读取周期。当 Use a custom tick source 选项打钩时，可使用 RTOS 为 LVGL 提供时基，用户只需要配置好头文件和节拍获取的函数即可。Default Dots Per Inch 顾名思义就是每英寸的点数量，用于初始化默认大小，例如小部件大小、样式填充。

4. 特征设置

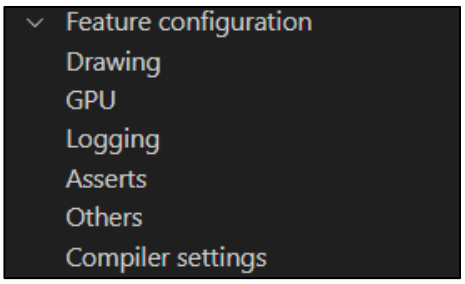


图 2.2.5 特征设置界面

特征设置分为 6 个子部分，分为为 Drawing 绘画效果、GPU 设置、Logging 日志功能、Assert 断言、Others 其他功能设置和 Compiler settings 编译器设置。

该板块中我们一般只用到日志部分，它可以在 LVGL 运行时，打印相关的信息，这对于用户调试程序，了解程序的运行情况非常有用。

5. 字体设置

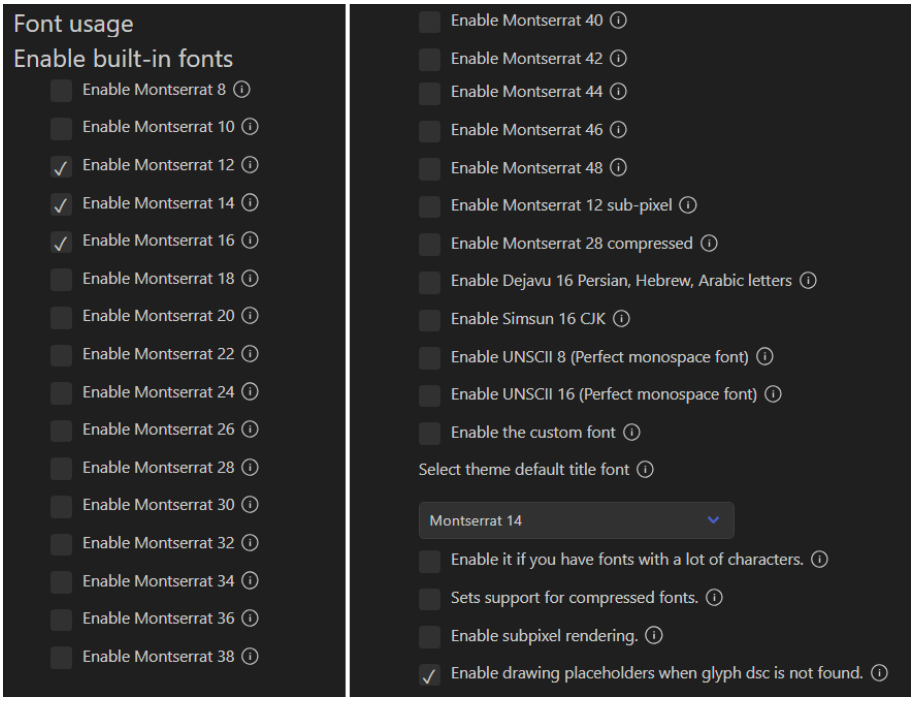


图 2.2.6 字体设置界面

用到什么字体就直接打钩即可。

6. 文本设置

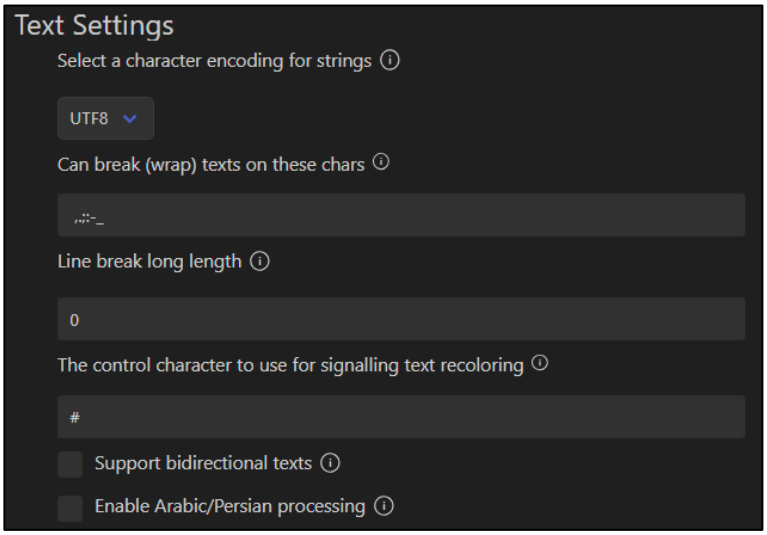


图 2.2.7 文本设置界面

选择字符编码。

7. 部件设置

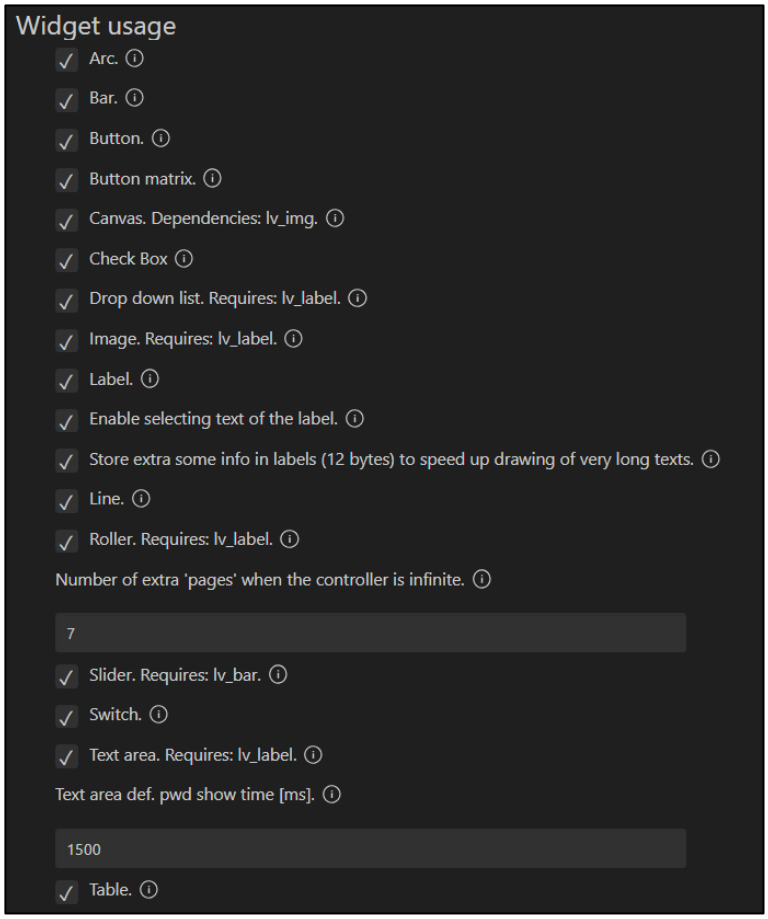


图 2.2.8 部件设置界面

上图中的选项框是用于控制相应部件的使能与失能，当某个部件打钩时则使能该部件；不勾选该部件即失能部件。这些配置项有效地优化 flash 的分配。

8. 额外部件设置

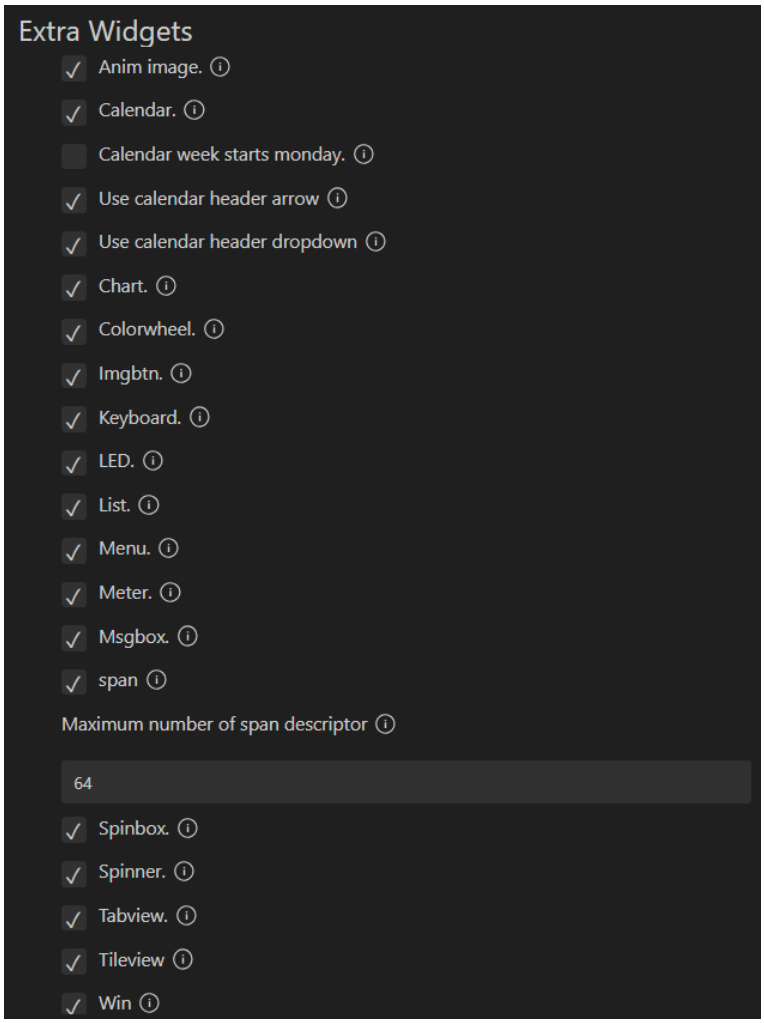


图 2.2.9 额外部件设置界面

上图中，还看到比较多的控件，LVGL 开发指南中也有对这些控件进行使用。若要使用控件，在控件前面的框打钩即可。

9. 主题设置

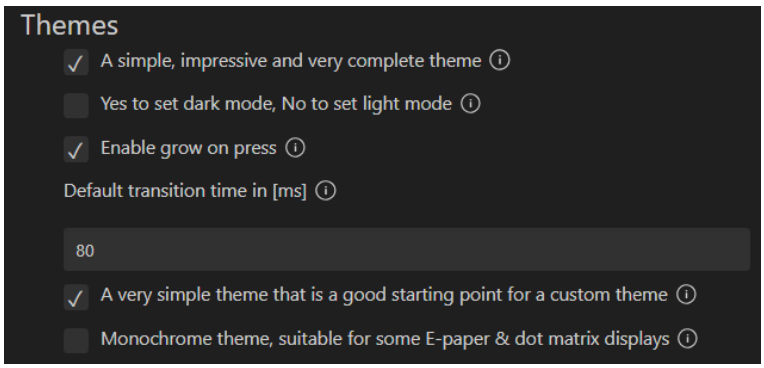


图 2.2.10 主题设置界面

主题是风格的集合，可应用于不同的物件，以统一界面的视觉效果。默认设置选择一个简单的主题。

10. 布局设置

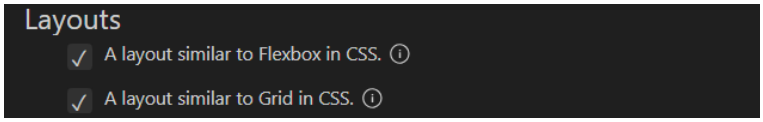


图 2.2.11 布局界面

LVGL 提供两种布局选择：Flexblox 弹性布局和 Grid 网格布局。要使用哪种布局就勾选哪个即可。

11. 第三方库设置

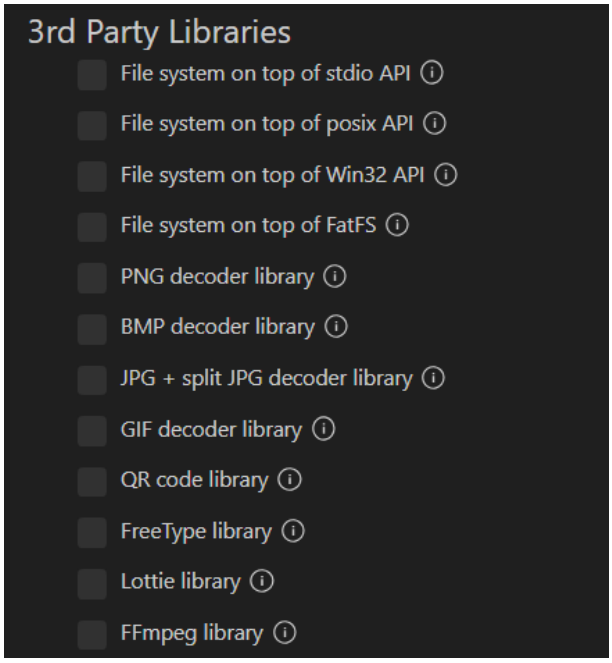


图 2.2.11 第三方库设置界面

LVGL 提供的第三方库很丰富，有文件系统接口、图片编解码、FreeType 和 FFmpeg。

12. 其他设置

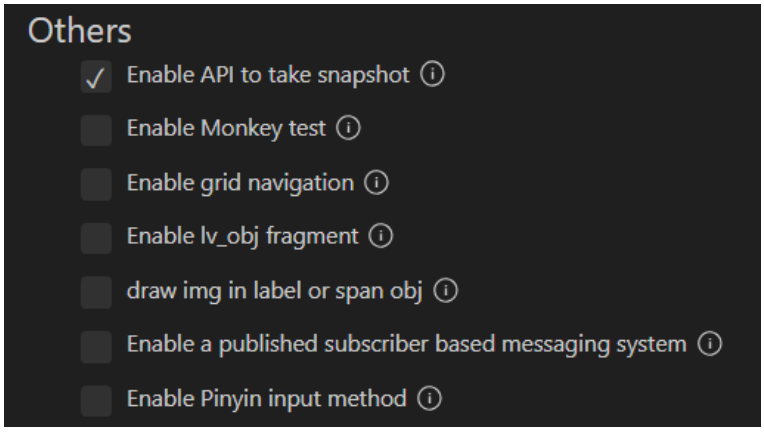


图 2.2.12 其他设置界面

13. 例程构建设置

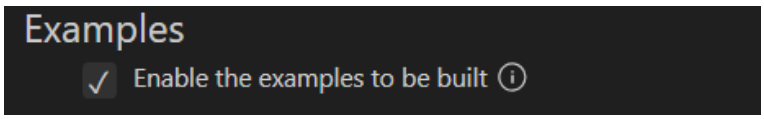


图 2.2.13 例程构建设置界面

14. 示例设置

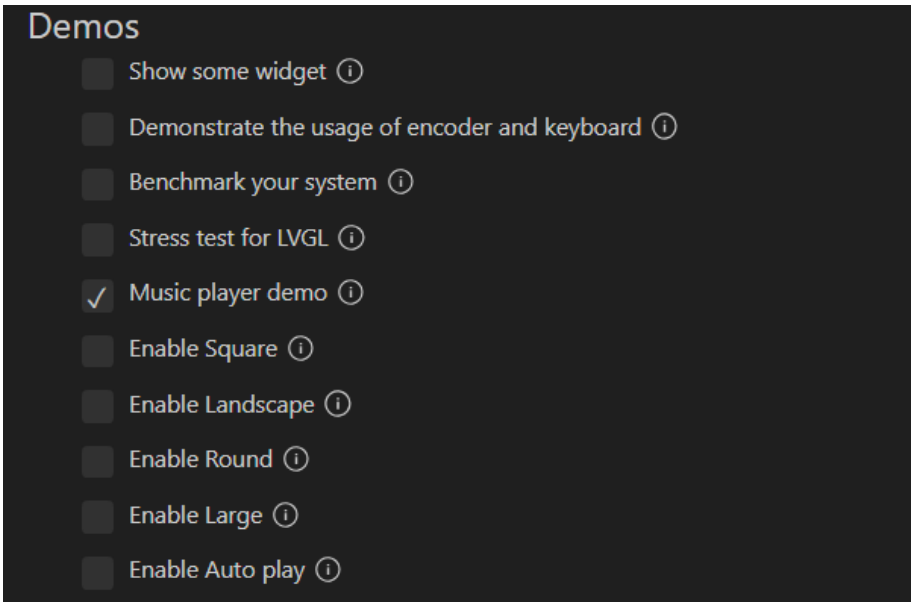


图 2.2.14 示例设置界面

2.3 显示接口

在绘制 UI 之前，LVGL 必须要注册一个绘制缓冲区以及显示驱动，该显示驱动可以把像素阵列（在缓冲区中）复制到显示器的指定区域。

2.3.1 绘制缓冲区

绘制缓冲区是 LVGL 用来渲染屏幕内容的简单数组。一旦渲染好，绘制缓冲区的内容就会使用 flush_cb 中的回调函数发送到显示器。

接下来，我们结合源码分析绘制缓冲区的三种初始化方式。

①单缓冲区：此时只有一个绘制缓冲区，其大小可以小于屏幕的总像素，但不建议小于“屏幕水平像素*10”。如果缓冲区较小，不足以一次性刷新全部内容，在这种情况下，整块区域将被重绘成多个部分，此时的屏幕刷新速度会变得不理想。而 LVGL 为了尽可能地提高刷新速度，当屏幕只有很小的区域发生变化时，它只会刷新那部分的区域。

注意：绘制缓冲区越大，对内存的占用就越多，所以在设置缓冲区大小的时候，需要考虑实际的内存是否够用。

②双缓冲区：当我们拥有两个缓冲区时，LVGL 可以在一个缓冲区中进行绘制，与此同时，将另一个缓冲区的内容从后台发送到显示器，这样即可实现渲染和刷新的并行。

③全尺寸双缓冲区：全尺寸的双缓冲区对内存的占用非常高，以上述代码为例，屏幕的分辨率为 800*480，颜色深度为 16 位（2 字节），此时全尺寸双缓冲区所占的内存为：800*480*2*2= 1536000 字节，而绝大多数 MCU 的内部 SRAM 不足以开辟如此大的空间。

2.3.2 注册显示驱动

缓冲区初始化完成后，LVGL 将会注册显示驱动，具体源码如下所示：

```
    void *buf1 = NULL;
void *buf2 = NULL;
```

```
/* 初始化显示设备 RGBLCD */
ltdc_init();                    /* RGB 屏初始化 */
ltdc_display_dir(1);            /* 设置横屏 */

/*-----------------------------
 * 创建一个绘图缓冲区（双缓冲区）
 *-----------------------------*/
buf1 =heap_caps_malloc(ltdcdev.width*60*sizeof(lv_color_t),MALLOC_CAP_DMA);
buf2 =heap_caps_malloc(ltdcdev.width*60*sizeof(lv_color_t),MALLOC_CAP_DMA);

/* 初始化显示缓冲区 */
static lv_disp_draw_buf_t disp_buf;     /* 保存显示缓冲区信息的结构体 */
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&disp_buf, buf1, buf2, ltdcdev.width * 60);

/* 在 LVGL 中注册显示设备 */
static lv_disp_drv_t disp_drv;
lv_disp_drv_init(&disp_drv);            /* 初始化显示设备 */

/* 设置显示设备的分辨率
 * 这里为了适配正点原子的多款屏幕，采用了动态获取的方式，
 * 在实际项目中，通常所使用的屏幕大小是固定的，因此可以直接设置为屏幕的大小
 */
disp_drv.hor_res = ltdcdev.width;
disp_drv.ver_res = ltdcdev.height;

/* 用来将缓冲区的内容复制到显示设备 */
disp_drv.flush_cb = lvgl_disp_flush_cb;

/* 设置显示缓冲区 */
disp_drv.draw_buf = &disp_buf;
disp_drv.user_data = panel_handle;

/* 注册显示设备 */
lv_disp_drv_register(&disp_drv);
```

从上述源码可知，显示驱动的注册流程分为以下几个步骤：

第一步：调用 lv_disp_drv_init 函数，初始化显示设备。

第二步：设置显示设备的分辨率、显示缓冲区以及刷屏函数。

第三步：调用 lv_disp_drv_register 函数注册显示设备。

注意：lv_disp_drv_t 需要是静态、全局或动态分配的变量。

2.3.3 屏幕旋转

当我们硬件显示器的 X 和 Y 轴方向调转且无法通过硬件方式来改变时，为了不重新设计硬件电路，可使用 LVGL 的屏幕旋转功能（纯软件），示意效果如下图所示：

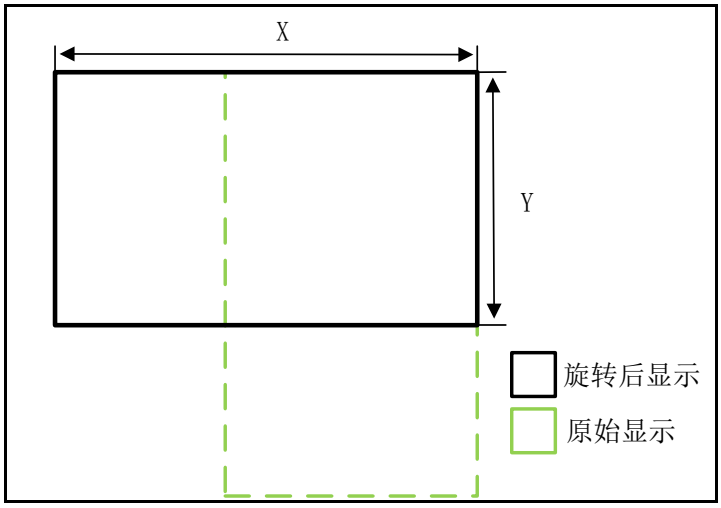


图 2.3.3.1 X 与 Y 轴调转

需要注意的是，纯软件实现屏幕旋转需要大量的内存开销，并且刷新效果会大打折扣。接下来，我们介绍屏幕旋转的配置步骤：

```
/* 第一步：设置 sw_rotate 标志设置为 1 */
disp_drv.sw_rotate = 1;
/* 第二步：调用旋转函数，对屏幕方向进行旋转 */
lv_disp_set_rotation(lv_disp_get_default(), LV_DISP_ROT_180);
```

用户可以在注册显示设备的时候进行屏幕旋转，旋转角度相关的枚举如下：

① LV_DISP_ROT_NONE：无需旋转；

② LV_DISP_ROT_90：旋转 90 度；

③ LV_DISP_ROT_180：旋转 180 度；

④ LV_DISP_ROT_270：旋转 270 度；

2.3.4 显示接口 API 函数

LVGL 官方提供了很多与显示接口相关 API，如下表所示：

函数	描述
注册显示设备相关函数	
lv_disp_drv_init()	初始化显示驱动
lv_disp_draw_buf_init()	初始化显示缓冲区
lv_disp_drv_register()	注册一个显示驱动
lv_disp_drv_update()	在运行时更新驱动程序
lv_disp_remove()	移除显示器
lv_disp_set_default()	设置默认显示设备
lv_disp_get_default()	获取默认显示设备
lv_disp_get_hor_res()	获取显示器的水平分辨率
lv_disp_get_ver_res()	获取显示器的垂直分辨率
lv_disp_get_physical_hor_res()	获取显示器的物理水平分辨率
lv_disp_get_physical_ver_res()	获取显示器的物理垂直分辨率
lv_disp_get_offset_x()	获取物理显示的水平偏移

lv_disp_get_offset_y()	获取物理显示的垂直偏移
lv_disp_get_antialiasing()	获取是否启用了抗锯齿功能
lv_disp_get_dpi()	获取显示器的 DPI
屏幕旋转 API 函数	
lv_disp_set_rotation()	设置旋转
lv_disp_get_rotation()	获取当前旋转参数
其他显示接口 API 函数	
lv_disp_get_next()	获取下一个显示设备
lv_disp_get_draw_buf	获取显示器的内部缓冲区

表 2.3.4.1 显示接口相关 API 函数

接下来，我们介绍一些 LVGL 显示接口的常用函数：

(1) lv_disp_drv_init 函数

初始化显示驱动，其函数原型如下所示：

```
void lv_disp_drv_init(lv_disp_drv_t *driver)
```

该函数的形参，如下表所示：

参数	描述
driver	指向要初始化的驱动变量的指针

表 2.3.4.2 lv_disp_drv_init 函数形参描述

返回值：无。

(2) lv_disp_draw_buf_init 函数

初始化显示缓冲区，其函数原型如下所示：

```
void lv_disp_draw_buf_init(lv_disp_draw_buf_t * draw_buf ,
                           void * buf1 ,
                           void * buf2 ,
                           uint32_t size_in_px_cnt)
```

该函数的形参，如下表所示：

参数	描述
draw_buf	要初始化的指针变量
buf1	用来绘制图像的缓冲区。必须有，不能为 NULL
buf2	第二个缓冲区，可设置 NULL
size_in_px_cnt	buf1 和 buf2 像素大小

表 2.3.4.3 lv_disp_draw_buf_init 函数形参描述

返回值：无。

(3) lv_disp_drv_register 函数

注册一个显示设备，其函数原型如下所示：

```
lv_disp_t * lv_disp_drv_register(lv_disp_drv_t *driver)
```

该函数的形参，如下表所示：

参数	描述
driver	指向已初始化的“lv_disp_drv_t”变量的指针

表 2.3.4.4 lv_disp_drv_register 函数形参描述

返回值：成功时返回相应的指针，错误时为 NULL。

(4) lv_disp_remove 函数

移除显示器，其函数原型如下所示：

```
void lv_disp_remove(lv_disp_t * disp)
```

该函数的形参，如下表所示：

参数	描述
disp	显示设备的指针

表 2.3.4.5 lv_disp_remove 函数形参描述

返回值：无。

(5) lv_disp_get_default 函数

获取默认显示设备，其函数原型如下所示：

```
lv_disp_t * lv_disp_get_default(void)
```

返回值：指向默认显示设备的指针。

(6) lv_disp_set_rotation 函数

设置屏幕的旋转，其函数原型如下所示：

```
void lv_disp_set_rotation(lv_disp_t * disp , lv_disp_rot_t rotation)
```

该函数的形参，如下表所示：

参数	描述
disp	指向显示设备的指针（NULL 表示使用默认显示设备）
rotation	旋转角度

表 2.3.4.6 lv_disp_set_rotation 函数形参描述

返回值：无。

2.4 输入设备

输入设备可以让用户和计算机系统之间进行信息交换，在 LVGL 中，支持的输入设备包括触摸屏、键盘、鼠标、编码器以及按钮。我们需要成功地驱动自己的输入设备，就必须在 LVGL 中将其注册。

2.4.1 注册输入设备

这里我们结合源码来介绍 LVGL 输入设备的注册流程，源码如下所示（以触摸屏为例）：

```
/* 初始化触摸屏 */
tp_dev.init();

/* 初始化输入设备 */
static lv_indev_drv_t indev_drv;
lv_indev_drv_init(&indev_drv);

/* 配置输入设备类型 */
indev_drv.type = LV_INDEV_TYPE_POINTER;

/* 设置输入设备读取回调函数 */
indev_drv.read_cb = touchpad_read;

/* 在 LVGL 中注册驱动程序，并保存创建的输入设备对象 */
lv_indev_t *indev_touchpad;
indev_touchpad = lv_indev_drv_register(&indev_drv);
```

由上述源码可知，输入设备的注册流程一共有五个步骤，不同类型输入设备的配置方法有所区别，但注册的流程都是相同的。

2.4.2 输入设备相关 API

LVGL 官方提供输入设备相关 API，如下表所示：

函数	描述
lv_indev_drv_init()	初始化输入设备
lv_indev_drv_register()	注册输入设备
lv_indev_drv_update()	在运行时更新驱动
lv_indev_delete()	移除输入设备
lv_indev_get_next()	获取下一个输入设备
lv_indev_read()	从输入设备读取数据

表 2.4.2.1 输入设备相关 API 函数

接下来，我们介绍 LVGL 输入设备的一些常用函数：

(1) lv_indev_drv_init 函数

初始化输入设备，其函数原型如下所示：

```
void lv_indev_drv_init(struct _lv_indev_drv_t *driver)
```

该函数的形参，如下表所示：

参数	描述
driver	指向要初始化的设备指针

表 2.4.2.2 lv_indev_drv_init 函数形参描述

返回值：无。

(2) lv_indev_drv_register 函数

注册输入设备，其函数原型如下所示：

```
lv_indev_t * lv_indev_drv_register ( struct _lv_indev_drv_t *driver)
```

该函数的形参，如下表所示：

参数	描述
driver	指向注册的设备指针

表 2.4.2.3 lv_indev_drv_register 函数形参描述

返回值：指向新输入设备的指针或 NULL 错误。

(3) lv_indev_delete 函数

移除输入设备，其函数原型如下所示：

```
void lv_indev_delete(lv_indev_t * indev)
```

该函数的形参，如下表所示：

参数	描述
indev	要删除的输入设备指针

表 2.4.2.4 lv_indev_delete 函数形参描述

返回值：无。

(4) _lv_indev_read 函数

读取输入设备数据，其函数原型如下所示：

```
void _lv_indev_read（lv_indev_t * indev， lv_indev_data_t *driver）
```

该函数的形参，如下表所示：

参数	描述
indev	指向输入设备的指针
driver	指向驱动程序的指针

表 2.4.2.5 _lv_indev_read 函数形参描述

返回值：无。

2.5 LVGL 时基

在 RTOS 中，任务的切换需要依赖系统定时器，而 LVGL 同样也需要一个时基，这样它才可以知道动画以及任务所经过的时间。

关于 LVGL 的时基配置，请大家回顾 LVGL 系统移植章节。下面我们来讲解 LVGL 时钟相关的函数：

(1) lv_tick_get 函数

该函数可以获取自启动以来经过的毫秒数，其原型如下所示：

```
uint32_t lv_tick_get(void)
```

返回值：经过的毫秒数。

(2) lv_tick_elaps 函数

该函数可以获取自上一个时间戳以来经过的毫秒数，其原型如下所示：

```
uint32_t lv_tick_elaps(uint32_t prev_tick)
```

该函数具有一个参数，如下表所示：

参数	描述
prev_tick	上一个时间戳

表 2.5.1 lv_tick_elaps 函数描述

返回值：经过的毫秒数。

2.6 LVGL 任务处理

LVGL 的任务处理函数（lv_timer_handler）类似于 RTOS 切换任务的任务调度函数。lv_timer_handler 函数对于定时的要求并不严格，一般来说，用户只需要确保在 5 毫秒以内调用一次该函数即可，以保持系统响应的速度。值得注意的是，LVGL 的任务处理并不是抢占式的，而是采用轮询的方式。

要处理 LVGL 的任务或者回调函数，用户只需要将 lv_timer_handler 函数定时调用即可，该函数可放在以下地方：

① main 函数的循环。

② 定时器的中断（优先级需要低于 lv_tick_inc）。

③ 定时执行的 OS 任务。

2.7 拓展知识

1. LVGL 睡眠模式

睡眠模式即低功耗模式。当没有触发信号时，我们可以让系统进入睡眠模式，以减少系统的功耗，具体的代码实现如下：

```
while(1)
{
    /* 正常操作(无睡眠)< 1 秒不活动 */
    if(lv_disp_get_inactive_time(NULL) < 1000)
    {
        lv_task_handler();
    }
    /* 静止 1 秒后睡觉 */
    else
```

```
    {
        timer_stop();   /* 停止调用 lv_tick_inc()的定时器 */
        sleep();        /* 调用自己的 MCU 睡眠函数*/
    }
    delay_ms(5);
}
```

注意：如果需要使用输入唤醒（按下、触摸等），应该在输入设备的读取中添加以下几行代码，如下所示：

```
lv_tick_inc(LV_DISP_DEF_REFR_PERIOD);  /* 强制唤醒任务执行 */
timer_start();                         /* 在调用 lv_tick_inc()时重新启动计时器 */
lv_task_handler();                     /* 手动调用' lv_task_handler() '来处理唤醒事件 */
```

2. 操作系统

在默认情况下，LVGL 不是线程安全的，其主要原因是：操作系统是抢占式的，而不同的任务之间，可能会发生资源的抢占，这将导致 LVGL 的运行出现异常，出现卡顿或者显示不全等问题。值得注意的是，在 LVGL 事件和定时器回调中部分，它们是安全的。

如果用户需要使用实际任务或线程，则需要一个互斥锁，以防止任务翻转，该互斥锁应在 lv_timer_handler 函数之前被调用，并在它之后释放，除此之外，用户在调用 LVGL 相关的函数和代码时，其相关的任务和线程中也需要使用相同的互斥锁。

以下是官方所提供的模板，源码如下：

```
static mutex_t lvgl_mutex;

void lvgl_thread(void)
{
    while(1)
    {
        mutex_lock(&lvgl_mutex);
        lv_task_handler();
        mutex_unlock(&lvgl_mutex); /* 释放互斥信号量 */
        thread_sleep(10);          /* sleep for 10 ms */
    }
}

void other_thread(void)
{
    /* 使用 LVGL api 时，你必须始终持有互斥对象 */
    mutex_lock(&lvgl_mutex);
    lv_obj_t *img = lv_img_create(lv_scr_act());
    mutex_unlock(&lvgl_mutex);      /* 释放互斥信号量 */

    while(1)
    {
        mutex_lock(&lvgl_mutex);
        /* 切换到下一个图像 */
```

```
        lv_img_set_src(img, next_image);
        mutex_unlock(&lvgl_mutex);      /* 释放互斥信号量 */
        thread_sleep(2000);
    }
}
```

3. 中断

用户应该尽量避免在中断里调用 LVGL 函数（lv_tick_inc 和 lv_disp_flush_ready 函数除外），如果必须执行此操作，则需要在 lv_timer_handler 函数运行时，禁用相关的中断。