LVGL 开发指南

V1.0

本教程适用于正点原子所有 STM32 开发板

修订历史：

版本	日期	修改内容
V1.0	2022/06/18	第一次发布
V1.1	2022/07/6	整理第二章的内容



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

正点原子官方网站 ：www.正点原子.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

基础知识篇 23

第一章 初识 LVGL 24

1.1 认识 LVGL 24
1.2 LVGL 移植需求 25
1.3 LVGL 源码下载 26

第二章 LVGL 无操作系统移植 29

2.1 移植准备工作 29
2.2 向工程添加文件 30
2.3 修改工程文件 36
2.4 移植官方例程 49
2.5 下载验证 51

第三章 LVGL 带操作系统移植 52

3.1 移植准备工作 52
3.2 添加应用程序 52
3.3 编写综合测试程序 61
3.3.1 硬件设计 61
3.3.2 软件设计 61
3.3.3 下载验证 64

第四章 PC 模拟器的使用 65

3.1 LVGL 模拟工程的下载 65
3.2 LVGL 例程模拟 67
3.3 工程文件的解析 68

第五章 LVGL 初探 72

5.1 系统总览 72
5.2 lv_conf.h 文件配置解析 73
5.3 LVGL 初始化流程 82
5.4 显示接口 82

5.4.1 绘制缓冲区 82
5.4.2 屏幕旋转 85
5.4.3 显示接口 API 函数 86
5.5 输入设备 87
5.5.1 触摸设备 88
5.5.2 光标设备 88
5.5.3 输入设备相关 API 90
5.6 LVGL 滴答 91
5.6.2 LVGL 任务处理 92
5.6.3 LVGL 的睡眠管理 92
5.7 操作系统和中断 93
5.8 LVGL 日志 94

第六章 LVGL 基础知识 95

6.1 LVGL 对象介绍 95
6.1.1 对象的基本属性(公用属性) 95
6.1.2 对象的具体属性(私有属性) 95
6.1.3 LVGL 工作机制 96
6.2 LVGL 布局介绍 99
6.2.1 对象坐标的设置 99
6.2.2 对象尺寸设置 99
6.2.3 对象的对齐设置 99
6.3 LVGL 样式属性 101
6.3.1 LVGL 样式属性设置 101
6.3.2 部件组成部分 102
6.3.3 部件的状态 103
6.3.4 LVGL 样式属性 105
6.4 LVGL 滚动属性 124
6.4.1 滚动的类型： 124
6.4.2 滚动条模式： 124
6.4.3 滚动条的回调事件 125
6.4.4 控制滚动条 125
6.4.5 滚动条其他特性 126
6.5 LVGL 动画属性 127

6.5.1 创建动画 127
6.5.2 设置动画路径 129
6.5.3 设置动画速度与时间 129
6.5.4 删除动画 130
6.5.5 动画实例 130
6.5.6 动画时间线 131
6.5.7 动画相关 API 函数 136
6.6 LVGL 定时器 137
6.6.1 创建定时器 137
6.6.2 重设/就绪定时器 138
6.6.3 定时器 API 函数 139
6.7 LVGL 事件 140
6.7.1 事件简介 140
6.7.2 添加事件 141
6.7.3 删除事件 141
6.7.4 事件类型 142
6.7.5 获取事件字段 144

第七章 LVGL 文件系统 145

7.1 LVGL 文件系统移植 145
7.2 LVGL 文件系统原理解析 147
7.2.1 LVGL 文件系统相关 API 函数 147
7.3 LVGL 文件系统实验 150
7.3.1 硬件设计 150
7.3.2 软件设计 150
7.3.3 下载验证 152

第八章 LVGL 字库使用 153

8.1 如何启用 UTF-8 编码 153
8.2 如何使用 LVGL 内置图标字体 153
8.3 如何使用 LVGL 内部字库 156
8.4 如何自定义字库 157
8.4.1 内部字库添加 157
8.4.2 外部字库添加 160

8.4.3 内部读取 TTF 字库 170
8.4.4 外部读取 TTF 字库 172

部件篇 177

第九章 基础对象 178

9.1 基础对象的角色定位 178
9.2 基础对象的相关知识 181
9.3 基础对象的 API 函数 182
9.4 基础对象部件的实验 183
9.4.1 硬件设计 183
9.4.2 软件设计 183
9.4.3 下载验证 187

第十章 圆弧部件(lv_arc) 188

10.1 圆弧部件的组成 188
10.2 圆弧部件的相关知识 188
10.2.1 圆弧的值和范围设置 188
10.2.2 圆弧部件角度设置 189
10.2.3 圆弧部件旋转设置 190
10.2.4 圆弧的模式选择 191
10.2.5 圆弧部件的变化率设置 191
10.2.6 移除旋钮 191
10.2.7 圆弧部件事件 191
10.3 圆弧部件的 API 函数 191
10.4 圆弧部件的实验 194
10.4.1 硬件设计 194
10.4.2 软件设计 195
10.4.3 下载验证 198

第十一章 进度条部件(lv_bar) 200

11.1 进度条部件的组成 200
11.2 进度条部件的相关知识 200
11.2.1 进度条部件的值和范围 200

11.2.2 进度条类型 202
11.2.3 进度条事件 202
11.3 进度条部件的 API 函数 202
11.4 进度条部件的实验 204
11.4.1 硬件设计 204
11.4.2 软件设计 204
11.4.3 下载验证 208

第十二章 按键部件(lv_btn) 209

12.1 按键部件的组成 209
12.2 按键部件的相关知识 209
12.3 按键部件的 API 函数 209
12.4 按键部件的实验 210
12.4.1 硬件设计 210
12.4.2 软件设计 210
12.4.3 下载验证 215

第十三章 按键矩阵部件(lv_ btnmatrix) 216

13.1 按键矩阵部件的组成 216
13.2 按键矩阵部件的相关知识 217
13.2.1 按键矩阵部件的文本显示 217
13.2.2 按键矩阵部件控制按键 218
13.2.3 按键矩阵部件的单次点击 219
13.2.4 按键矩阵部件着色设置 220
13.2.5 按键矩阵部件的事件 221
13.3 按键矩阵部件的 API 函数 221
13.4 按键矩阵部件的实验 224
13.4.1 硬件设计 224
13.4.2 软件设计 224
13.4.3 下载验证 227

第十四章 画布部件(lv_canvas) 228

14.1 画布部件的组成 228
14.2 画布部件的相关知识 228

14.2.1 画布缓冲区定义 228
14.2.2 画布调色板设置 229
14.2.3 画布部件的绘画 229
14.2.4 画布部件的旋转 231
14.2.5 画布部件的模糊处理 233
14.3 画布部件的 API 函数 235
14.4 画布部件的实验 240
14.4.1 硬件设计 240
14.4.2 软件设计 240
14.4.3 下载验证 243

第十五章 复选框部件(lv_checkbox) 244

15.1 复选框部件的组成 244
15.2 复选框部件的相关知识 244
15.2.1 设置复选框部件文本 244
15.2.2 复选框部件状态设置 245
15.2.3 复选框事件 245
15.3 复选框部件 API 函数 245
15.4 复选框部件实验 246
15.4.1 硬件设计 246
15.4.2 软件设计 247
15.4.3 下载验证 253

第十六章 下拉列表部件(lv_dropdown) 254

16.1 下拉列表部件的组成 254
16.2 下拉列表部件的相关知识 255
16.2.1 下拉列表部件添加/设置表项 255
16.2.2 下拉列表部件获取选择的选项 256
16.2.3 下拉列表部件设置方向 256
16.2.4 下拉列表部件符号设置 257
16.2.5 下拉列表部件设置显示项 258
16.2.6 下拉列表部件手动开闭 259
16.3 下拉列表部件的 API 函数 259
16.4 下拉列表部件的实验 262

16.4.1 硬件设计 262
16.4.2 软件设计 262
16.4.3 下载验证 266

第十七章 图片部件(lv_img) 268

17.1 图片部件的组成 268
17.2 图片部件的相关知识 268
17.2.1 图像源选择 268
17.2.2 图像重新着色 269
17.2.3 图像自动大小 269
17.2.4 图像偏移 269
17.2.5 图像其他属性的设置 270
17.3 图片部件的 API 函数 271
17.4 图片部件的实验 274
17.4.1 硬件设计 274
17.4.2 软件设计 274
17.4.3 下载验证 277

第十八章 标签部件(lv_label) 279

18.1 标签部件的组成 279
18.2 标签部件的相关知识 279
18.2.1 文本设置 279
18.2.2 换行符设置 280
18.2.3 文本长模式 280
18.2.4 文本着色 281
18.2.6 长文本设置 281
18.3 标签部件的 API 函数 281
18.4 标签部件的实验 283
18.4.1 硬件设计 283
18.4.2 软件设计 283
18.4.3 下载验证 286

第十九章 线部件(lv_line) 287

19.1 线部件的组成 287

19.2 线部件的相关知识 287
19.2.1 设置连接点 287
19.2.2 自适应大小 288
19.2.3 倒 Y 操作 289
19.3 线部件的 API 函数 289
19.4 线部件的实验 290
19.4.1 硬件设计 290
19.4.2 软件设计 291
19.4.3 下载验证 293

第二十章 滚筒部件(lv_roller) 295

20.1 滚筒部件的组成 295
20.2 滚筒部件的相关知识 296
20.2.1 添加选项及滚筒模式 296
20.2.2 获取选项文本 297
20.2.3 选项可见数量 297
20.3 滚筒部件的 API 函数 298
20.4 滚筒部件的实验 300
20.4.1 硬件设计 300
20.4.2 程序设计 300
20.4.3 下载验证 304

第二十一章 滑块控件(lv_slider) 305

21.1 滑块部件的组成 305
21.2 滑块部件的相关知识 305
21.2.1 设置滑块部件的数值范围 305
21.2.2 设置滑块部件的模式 306
21.2.3 Knob-only 模式 307
21.3 滑块部件的 API 函数 307
21.4 滑块部件的实验 309
21.4.1 硬件设计 309
21.4.2 软件设计 309
21.4.3 下载验证 315

第二十二章 开关控件(lv_switch) 317

22.1 开关部件的组成 317
22.2 开关部件的相关知识 317
22.2.1 获取开关部件状态 317
22.2.2 添加开关部件的状态 318
22.3 开关部件的 API 函数 318
22.4 开关部件的实验 318
22.4.1 硬件设计 318
22.4.2 软件设计 318
22.4.2.1 程序流程图 318
22.4.2.2 程序解析 319
22.4.3 下载验证 321

第二十三章 表格图控件(lv_table) 323

23.1 表格部件的组成 323
23.2 表格部件的相关知识 323
23.2.1 设置单元格的值 323
23.2.2 行和列的设置 324
23.2.3 宽度和高度的设置 325
23.2.4 合并单元格 325
23.3 表格部件的 API 函数 325
23.4 表格部件的实验 327
23.4.1 硬件设计 327
23.4.2 软件设计 328
23.4.3 下载验证 331

第二十四章 文本区域控件(lv_textarea) 333

24.1 文本区域部件的组成 333
24.2 文本区域部件的相关知识 333
24.2.1 创建文本区域部件 333
24.2.2 添加与删除字符 333
24.2.3 占位符文本 334
24.2.4 移动或者隐藏光标 335

24.2.5 文本区域部件的模式 335
24.2.6 限制字符输入 336
24.3 文本区域部件的 API 函数 337
24.4 文本区域部件的实验 340
24.4.1 硬件设计 340
24.4.2 软件设计 340
24.4.3 下载验证 345

第二十五章 日历部件(lv_calendar) 347

25.1 日历部件的组成 347
25.2 日历部件的相关知识 347
25.2.1 创建日历部件 347
25.2.2 日期的设置/显示 347
25.2.3 设置日期高亮 349
25.2.4 设置日名 350
25.3 日历部件的 API 函数 352
25.4 日历部件的实验 354
25.4.1 硬件设计 354
25.4.2 软件设计 355
25.4.3 下载验证 357

第二十六章 图表部件(lv_chart) 359

26.1 图表部件的组成 359
26.2 图表部件的相关知识 359
26.2.1 图表部件的主要功能 359
26.2.2 图表部件的辅助功能 364
26.3 图表部件 API 函数 367
26.4 图表控件的实验 369
26.4.1 硬件设计 369
26.4.2 软件设计 369
26.4.3 下载验证 373

第二十七章 色环部件(lv_colorwheel) 374

27.1 色环部件的组成 374

27.2 色环部件的相关知识 374
27.2.1 创建色环部件 374
27.2.2 设置色环部件的模式 375
27.2.3 色环部件的事件 376
27.3 色环部件 API 函数 376
27.4 色环控件的实验 377
27.4.1 硬件设计 377
27.4.2 软件设计 378
27.4.3 下载验证 380

第二十八章 图片按键部件(lv_imgbtn) 381

28.1 图片按键部件的组成 381
28.2 图片按键部件的相关知识 381
28.2.1 图像的来源 381
28.2.2 添加/清除的状态 382
28.3 图片按键部件 API 函数 383
28.4 图片按键控件实验 384
28.4.1 硬件设计 384
28.4.2 软件设计 384
28.4.3 下载验证 386

第二十九章 键盘部件(lv_keyboard) 388

29.1 键盘部件的组成 388
29.2 键盘部件的相关知识 389
29.2.1 键盘部件模式 389
29.2.2 指定文本区域 389
29.2.3 键弹窗设置 390
29.3 键盘部件 API 函数 390
29.4 键盘控件实验 392
29.4.1 硬件设计 392
29.4.2 软件设计 393
29.4.3 下载验证 395

第三十章 LED 部件(lv_led) 397

30.1 LED 部件的组成 397
30.2 LED 部件的相关知识 397
30.2.1 设置 LED 部件颜色 397
30.2.2 设置 LED 部件亮度 397
30.2.3 状态切换 398
30.3 LED 部件 API 函数 398
30.4 LED 控件实验 400
30.4.1 硬件设计 400
30.4.2 软件设计 400
30.4.3 下载验证 404

第三十一章 列表部件(lv_list) 406

31.1 列表部件的组成 406
31.2 列表部件的相关知识 406
31.2.1 添加列表按键 406
31.2.2 设置列表文本 407
31.3 列表部件 API 函数 408
31.4 列表控件实验 409
31.4.1 硬件设计 409
31.4.2 软件设计 409
31.4.3 下载验证 412

第三十二章 仪表部件(lv_meter) 414

32.1 仪表部件的组成 414
32.2 仪表部件的相关知识 415
32.2.1 仪表部件主要功能 415
32.2.2 仪表部件次要功能 421
32.3 仪表部件 API 函数 428
32.4 仪表控件实验 429
32.4.1 硬件设计 429
32.4.2 软件设计 429
32.4.3 下载验证 435

第三十三章 消息框部件(lv_msgbox) 436

33.1 消息框部件的组成 436
33.2 消息框部件的相关知识 436
33.2.1 创建消息框部件 436
33.2.2 获取消息框的组成部件 437
33.2.3 关闭消息框部件 437
33.2.4 消息框部件事件 437
33.3 消息框部件 API 函数 438
33.4 消息框控件实验 439
33.4.1 硬件设计 439
33.4.2 软件设计 439
33.4.3 下载验证 443

第三十四章 跨度部件(lv_span) 446

34.1 Span 部件的组成 446
34.2 Span 部件的相关知识 446
34.2.1 设置文本和样式 446
34.2.2 检索 span 组的子对象 447
34.2.3 Span 部件文本对齐 447
34.2.4 Span 组的模式选择 448
34.2.5 文本溢出 448
34.2.6 文本首行缩进 449
34.3 Span 部件 API 函数 449
34.4 Span 控件实验 449
34.4.1 硬件设计 449
34.4.2 软件设计 450
34.4.3 下载验证 453

第三十五章 微调器部件(lv_spinbox) 454

35.1 微调器部件的组成 454
35.2 微调器部件的相关知识 454
35.2.1 微调器部件属性设置 454
35.2.2 微调器部件翻转模式 455
35.3 微调器部件 API 函数 455
35.4 微调器控件实验 458

35.4.1 硬件设计 458
35.4.2 软件设计 458
35.4.3 下载验证 462

第三十六章 加载器部件(lv_spinner) 463

36.1 加载器部件的组成 463
36.2 加载器部件的相关知识 463
36.4 加载器部件 API 函数 463
36.5 加载器控件实验 464
36.4.1 硬件设计 464
36.4.2 软件设计 464
36.4.3 下载验证 465

第三十七章 选项卡视图部件(lv_tabview) 467

37.1 选项卡视图部件的组成 467
37.2 选项卡视图部件的相关知识 467
37.2.1 创建 Tab 视图对象 467
37.2.2 添加选项卡 468
37.2.3 切换选项卡 469
37.2.4 获取部件 469
37.3 选项卡视图部件的 API 函数 469
37.4 选项卡视图部件的实验 470
37.4.1 硬件设计 470
37.4.2 软件设计 471
37.4.3 下载验证 473

第三十八章 平铺视图部件(lv_tileview) 475

38.1 平铺视图部件的组成 475
38.2 平铺视图部件的相关知识 475
38.2.1 添加一个 Tile 视图 475
38.2.2 切换 Tile 视图 476
38.3 平铺视图部件的 API 函数 476
38.4 平铺视图部件的实验 478
38.4.1 硬件设计 478

38.4.2 软件设计 478
38.4.3 下载验证 480

第三十九章 窗口部件(lv_win) 481

39.1 窗口部件的组成 481
39.2 窗口部件的相关知识 481
39.2.1 创建窗口 481
39.2.2 标题与按键的设置 481
39.2.3 获取窗口部件区域 482
39.3 窗口部件的 API 函数 482
39.4 窗口部件的实验 483
39.4.1 硬件设计 483
39.4.2 软件设计 484
39.4.3 下载验证 487

第四十章 动画图像(lv_animimg) 488

40.1 动画图像部件的组成 488
40.2 动画图像部件的相关知识 488
40.3 动画图像部件的 API 函数 488
40.4 动画图像部件的实验 490

第四十一章 菜单部件(lv_menu) 491

41.1 菜单部件的组成 491
41.2 菜单部件的相关知识 491
41.2.1 创建菜单部件 491
41.2.2 标题模式 492
41.2.3 根返回按钮模式 492
41.2.4 创建菜单页面 492
41.2.5 设置主容器页面 492
41.2.6 设置侧边栏容器页面 494
41.2.7 菜单页面的连接 495
41.2.8 创建一个菜单容器、空区域和分隔符 496
41.3 菜单部件的 API 函数 496
41.4 菜单部件的实验 497

组件篇 498

第四十二章 LVGL BMP 图片 499

42.1 LVGL 的 BMP 解码库概述 499
42.2 LVGL 的 BMP 解码库实验 502
42.2.1 硬件设计 502
42.2.2 软件设计 503
42.2.3 下载验证 504

第四十三章 LVGL PNG 图片 505

43.1 LVGL PNG 解码库的移植 505
43.2 LVGL PNG 图片显示实验 506
43.2.1 硬件设计 506
43.2.2 软件设计 506
43.2.3 下载验证 508

第四十四章 LVGL JPEG 图片 510

44.1 LVGL JPEG 解码库的移植 510
44.1.1 JPG 转换 SJPG 图片格式 511
44.2 LVGL JPEG 图片显示实验 513
44.2.1 硬件设计 513
44.2.2 软件设计 513
44.2.3 下载验证 515

第四十五章 LVGL GIF 读取 516

45.1 LVGL GIF 解码库的移植 516
45.2 LVGL GIF 图形显示实验 516
45.2.1 硬件设计 516
45.2.2 软件设计 517
45.2.3 下载验证 519

第四十六章 LVGL 二维码库 520

46.1 LVGL 二维码库移植 520
46.1.1 LVGL 二维码库 API 函数 520

46.2 LVGL 二维码实验 521
46.2.1 硬件设计 521
46.2.2 软件设计 521
46.2.3 下载验证 523

第四十七章 SquareLine Studio 使用 524

47.1 SquareLine Studio 初探 524
47.2 SquareLine Studio 实验 532
47.3 SquareLine Studio 移植到工程 535

布局篇 537

第四十八章 Flex 布局 538

48.1 Flex 初探 538
48.2 Flex 相关知识 538
48.2.1 启用 Flex 布局 538
48.2.2 Flex 条文 538
48.2.3 对象如何使用 Flex 布局 542
48.2.4 Flex 对齐 543
48.2.5 flex-grow 属性 545
48.2.6 Flex 条文的样式函数 545
48.2.7 Flex 间隙 545
48.3 Flex 布局的实验 546

第四十九章 Grid 布局 547

49.1 Grid 初探 547
49.2 Grid 相关知识 547
49.2.1 启用 Grid 布局 547
49.2.2 Grid 条文 547
49.2.3 对象如何使用 Grid 布局 547
49.2.3 Grid 描述符 547
49.2.4 添加 Grid 对象 548
49.2.5 Grid 对齐 550
49.2.6 Grid 条文的样式函数 550

49.2.7 Grid 间隙 550
49.3 Grid 布局的实验 550

声明

本教程中关于 LVGL 的原理性知识均参考自 LVGL 官方手册：https://docs.lvgl.io/latest/en/html/index.html，其中大多数的理论知识都是翻译这本在线手册，有关 Cortex-M 的内核知识参考自《The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition》这本书。本教程作为 LVGL 学习、参考资料全部免费公开，包括教程中的实验源码！

基础知识篇

LVGL（Light and Versatile Graphics Library）是一个免费的开源图形库，提供创建具有易于使用的图形元素、漂亮的视觉效果以及低内存占用的嵌入式 GUI。

本教程分为 4 个部分讲解，第一部分将带领大家认识 LVGL，包括：LVGL 基础知识及工程搭建。第二部分将详细介绍 LVGL 相关部件的使用，包括：obj、btn、arc 等部件知识及部件的 API 函数使用。第三部分将详细介绍 LVGL 第三方库的使用，包括：BMP、PNG 等解码库。第四部分将详细介绍 LVGL 的 Flex 和 Grid 布局，这两个布局可以简便、完整、响应式地实现各种页面布局。

第一章 初识 LVGL

图形用户界面是一种人与计算机通信的界面显示格式，允许用户使用鼠标等输入设备操纵屏幕上的图标或菜单选项，以选择命令、调用文件、启动程序或执行其它一些日常任务。通过键盘输入文本或字符命令来完成例行任务的字符界面相比，图形用户界面有许多优点。图形用户界面由窗口、下拉菜单、对话框及其相应的控制机制构成，在各种新式应用程序中都是标准化的，即相同的操作总是以同样的方式来完成，在图形用户界面，用户看到和操作的都是图形对象，应用的是计算机图形学的技术。

在实际应用中，我们时常需要制作 UI 界面来实现人机交互，简单的 UI 可以直接编写代码，但对于那些复杂的交互界面，自己编写 UI 界面可能就力不从心了。因此可使用第三方的 GUI 库来设计 UI 界面，例如 LVGL、emWin、QT 等图形用户库，它们都可以设计漂亮的 UI 界面。本教程主要讲解 LVGL 下图形用户界面的开发与使用。

本章分为如下几个部分讲解:

1.1 认识 LVGL

1.2 LVGL 移植需求

1.3 LVGL 源码下载

1.1 认识 LVGL

LVGL（Light and Versatile Graphics Library）是一个免费的开源图形库，提供创建具有易于使用的图形元素、漂亮的视觉效果和低内存占用的嵌入式 GUI。接下来我们来看一下 LVGL 图形用户库的主要特征有哪些：

1. 强大的构建块：按钮、图表、列表、滑块、图像等部件。
2. 具有高级图形属性：具有动画、抗锯齿、不透明度、平滑滚动的高级图形。
3. 支持各种输入设备：如触摸、鼠标、键盘、编码器。
4. 支持多语言：UTF-8 编码。
5. 支持多显示器：它可以同时使用多个 TFT 或者单色显示器。
6. 支持多种样式属性：它具有类 CSS 样式的完全可定制的图形元素。
7. 独立于硬件之外：它与任何微控制器或显示器一起使用。
8. 可扩展性：它能够以小内存运行(最低 64 kB 闪存，16 kB RAM 的 MCU)。
9. 支持操作系统、外部存储器和 GPU(不是必需的)。
10. 具有高级图形效果：可进行单帧缓冲区操作。
11. 纯 C 编写： C 语言编写以获得最大的兼容性。

从上述特征可知：LVGL 是一款具有丰富的部件，具备高级图形特性，支持多种输入设备，多国语言和独立于硬件之外等免费的开源图形库。LVGL 官方地址为：https://lvgl.io/，该网页主要包含用户文档以及图片和字体转换器，该网页打开后如图 1.1.1 所示：



图 1.1.1 LVGL 官方网页

上图中的“Github”图标可点击进入下载 LVGL 相关源码的网页；上图中 的“Docs”图标可点击进入 LVGL 用户文档，该文档是纯英文编写的，主要讲解 LVGL 移植方式和各个部件的使用；上图右上角的“Tools”选项主要提供用户图片和字库在线转换器，这些转换器是以某种规则把目标文件转换成嵌入式可读取的文件。

点击上图中的“Docs”图标进入 LVGL 在线用户文档，该文档是为了用户能快速上手 LVGL 图形库并且它还提供了初学者学习的源码和资料，该在线用户文档网页如下所示：

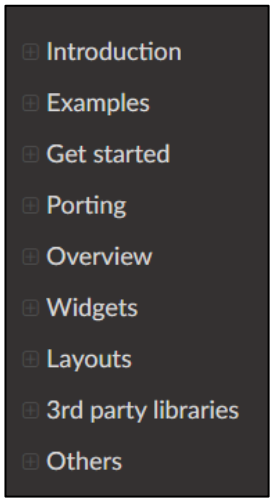


图 1.1.2 LVGL “在线文档教文档”

从上图可知：LVGL 的在线文档教程是英文文档，如果读者对英文很了解，笔者建议看官方的在线教程；如果读者对英文很敏感，则笔者建议看本教程的内容，因为本教程是根据 LVGL 在线教程翻译而编写的，它能让读者快速掌握 LVGL 下图形用户界面的开发与使用。在以后的章节中，笔者重点讲解上图的 Widgets 和 3rd party libraries 章节的内容，这些内容占据了 LVGL 百分之 70 的内容，所以笔者希望读者能掌握这些内容。

1.2 LVGL 移植需求

在市面上具有众多的 MCU 微处理器，这些处理器并不是每一个都适合移植 LVGL 图形库，如传统的 51 或者 52 单片机等微处理器，它们并不具备 LVGL 图形库的移植条件，因为 LVGL

图形库无论对 MCU 还是 LCD 显示屏都是具备相应的要求才能移植，下面我们分别地讲解它们的硬件需求有哪些。

1. MCU 的需求：

LVGL 图形库移植对 MCU 是有要求的，每一个能够驱动显示器的 MCU 是否能运行 LVGL 图形库，取决于 MCU 是否具备 LVGL 的最低移植需求。该最低移植需求如表 1.2.1 所示：

要求	说明
微控制器	16、32 或 64 位微控制器或处理器
主控频率(Hz)	> 16 MHz 时钟速度
Flash/ROM	> 64 kB，如果使用非常多的部件（推荐 > 180 kB）
内存(RAM)	静态 SRAM：~2 kB，取决于使用的功能和对象类型
	堆栈：> 2kB（推荐> 8 kB）
	动态数据（堆）：> 4 KB（如果使用多个对象，建议 > 32 kB）
	显示缓冲区：>“水平分辨率”像素（推荐 > 10 *“水平分辨率”）
	MCU (RAM)或外部存储器分配的一个帧缓冲区

表 1.2.1 MCU 移植 LVGL 的最低要求

从上表可知：微处理器至少是 16 位以上的，显然有些微处理器并不满足这个最低移植需求，如宏晶科技 STC89C52 和高性能 HCS08 系列等处理器，它们都是 8 位微处理器单元。接下来，笔者来对比一下正点原子 Mini 板是否满足 LVGL 最低移植需求，如果 Mini 板满足 LVGL 最低移植需求，则证明正点原子其他 STM32 开发板都满足 LVGL 最低移植要求，因为 Mini 开发板在正点原子 STM32 系列开发板中是处于最低配置的开发板，所以笔者使用这款开发板的资源与 LVGL 最低移植需求做比较，如果它能满足 LVGL 最低移植需求，则正点原子其他 STM32 系列开发板都能满足。Mini 开发板的主控 MCU 是 32 位微处理器单元，它的主频率为 72MHz，内嵌 Flash 为 256K 以及内嵌 SRAM 为 64K 等资源信息，这些资源信息比表 1.2.1 的 LVGL 最低移植要求是远远大于后者的，所以正点原子的 Mini 开发板可移植 LVGL 图形库。

2. 显示屏的要求：

LVGL 只需要一个简单的驱动程序函数即可将像素阵列复制到显示器的给定区域中，这就是 LVGL 调用显示驱动绘画原理，下面笔者来讲解 一下 LVGL 对显示屏的需求有哪些，如下所示：

① 具有 8/16 /24/ 32 位色深的显示屏。
② HDMI 端口的显示器。
③ 小型单色显示器。
④ 灰度显示。
⑤ LED 矩阵。
⑥ 其他可以控制像素颜色/状态的显示器。

从上述要求可知：我们正点原子的 2.8/3.5/4.3/7/10.1 寸 TFTLCD 模块以及 RGBLCD 模块都是 16 位深的显示屏，这些显示屏可满足 LVGL 的最低显示屏的要求。

1.3 LVGL 源码下载

前面笔者讲解到，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/，读者可以在浏览器打开并下载 LVGL 图形库最新版本源码。由于 GitHub 服务器是在国外的，所以在国内访问该服务器，可能登录不成功，我们可以在正点原子光盘资料下获取 LVGL 图形库的最新版本源码，该获取路径为：“6.软件资料→14/15.LVGL 学习资料”，如图 1.3.2 所示：

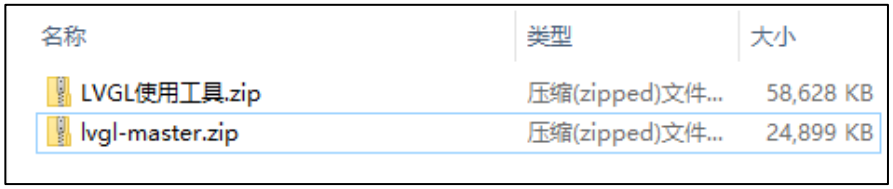


图 1.3.2 LVGL 学习资料

上图的“LVGL 使用工具”文件夹保存的是 LVGL 相关的离线的转换工具，这些离线工具是广大博主根据 LVGL 的字库和图片的处理特性编写的软件，而 lvgl-maser.zip 压缩包是 LVGL 图形库的源码，接下来笔者分别地介绍 lvgl-maser.zip 压缩包下的相关文件，首先我们解压图 1.3.2 中的 lvgl-master.zip 压缩包，解压后的 lvgl-master 文件夹下的内容如下图所示：



图 1.3.3 lvgl-master 图形库文件

在上图中，我们会发现 lvgl-master 目录下有很多文件和文件夹，这些文件和文件夹并不需要用户全部都要了解的，这里我们了解与移植相关的文件和文件夹即可。LVGL 移植所需要的文件夹和文件有 examples 文件夹、src 文件夹、lv_conf_template.h 和 lvgl.h 文件，其他的文件与移植无关。下面我们分别介绍图 1.3.3 的重要文件和文件夹如表 1.3.1 所示：

文件	说明
demos	LVGL 提供的综合例程源码
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，这个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 1.3.1 lvgl-master 文件说明

对于上述的文件夹，建议读者了解 examples 文件夹和 src 文件夹下的内容即可，其他文件夹与移植无关，接下来我们分别地讲解这两个文件夹的文件结构如下所示：

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，如表 1.3.2 所示：

文件	描述
anim	LVGL 动画例程实例
arduino	开源电子平台(不需要了解)
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(flex/grid)
libs	LVGL 移植第三方库实例
others	LVGL 其他测试
porting	LVGL 输入设备驱动和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 1.3.2 examples 文件夹下文件的描述

上表的 porting 文件夹和移植相关的，其他的文件夹是提供用户学习的实例。

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)，如表 1.3.3 所示：

文件	描述
core	LVGL 核心源码(事件、组、对象、坐标、样式、主题等)
draw	LVGL 绘画驱动(图片，解码、DMA2D、圆、线、圆弧、和文本等)
extra	LVGL 额外的文件(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(该文件夹的内容在 draw 文件夹内定义)
hal	硬件抽象层(显示驱动程序、输入设备程序以及提供 LVGL 系统滴答)
misc	主要描述 LVGL 其他定义(动画、内存管理、日志等等文件)
widgets	LVGL 基础部件

表 1.3.3 src 文件夹下文件的描述

在以后的学习中，我们重点讲解上表的 widgets 和 extra 文件夹下的文件内的接口使用。

第二章 LVGL 无操作系统移植

在上一个章节中，我们已经了解 LVGL 移植需求以及 LVGL 图形库下载路径。本章主要讲解裸机移植 LVGL 到正点原子 STM32 开发板上。首先声明一下，本教程适用于正点原子的所有 STM32 开发板，支持正点原子的全系列显示屏，当然不同的开发板对显示屏型号的支持力度大小可能会不同，这个会在相应开发板的配套例程代码中进行说明，为了方便读者，本教程中的图片和素材都是通用的，如有区别，笔者将会在教程中以表格的方式指出。

本章分为如下几个部分讲解：

2.1 移植准备工作
2.2 向工程添加文件
2.3 修改工程文件
2.4 移植官方例程
2.5 下载验证

2.1 移植准备工作

在移植 LVGL 之前，用户必须在裸机例程中找到一个移植基础例程，笔者建议使用正点原子 STM32 开发板的裸机例程中的内存管理实验(注意：该实验必须包含 LCD 显示驱动以及 touch 触摸驱动文件)，有学过正点原子 STM32 开发板的资料的用户都知道，内存管理实验的内存算法是正点原子独有的一套算法，该算法主要管理内部 SRAM 以及外部 SRAM 的内存。因为 LVGL 库内含自己内存管理算法，该管理算法的管理内存是由用户分配给它的，所以用户分配内存方式有两种，一种是内部 SRAM 分配方式，而另一种是外部 SRAM 分配方式。下面我们开始讲解移植准备工作的流程，如下所示：

1. 准备 LVGL 源码

LVGL 源码可从 LVGL 官方 GitHub 网址(https://github.com/lvgl/lvgl/)下载。当然我们也可以在正点原子光盘资料下获取，该获取路径为：“6.软件资料→14.LVGL 学习资料→lvgl-maser.zip”。该压缩包解压后得到下图的文件和文件夹。



图 2.1.1 LVGL 源码目录

上图中的文件和文件夹我们在第一章中已经介绍过了，这里笔者无需重复介绍，

2. 把 lv_conf_template.h 文件名修改成 lv_conf.h 文件名。

3. 打开 lv_conf.h 文件，修改条件编译指令，如下源码所示。

修改：

```
#if 0 /*Set it to "1" to enable content*/
```

修改成：

```
#if 1 /* 把#if 0 修改成#if 1 */
```

前面我们也讲解过，LVGL 移植所需要的文件夹和文件有 examples 文件夹、src 文件夹、lv_conf_template.h 和 lvgl.h 文件，其他文件和文件夹与移植无关，我们可以删除其他文件和文件夹。

4. 打开 examples 文件夹，除了 porting 文件夹外，用户可以删除其他文件和文件夹，如图 2.1.1 所示。



图 2.1.1 保留 porting 文件夹

根据上述步骤的操作，我们可得到 LVGL 移植的简洁源码，如下图所示。

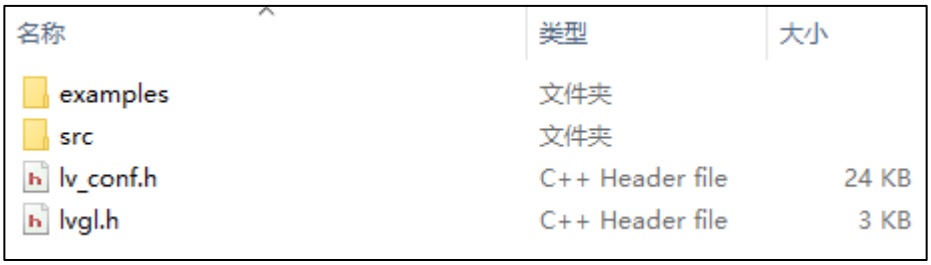


图 2.1.2 LVGL 移植简洁源码

2.2 向工程添加文件

1. 准备基础工程

在移植之前我们需要一个基础工程，这里我们使用裸机例程中的内存管理实验作为基础工程，我们在这个工程的基础上完成本章的 LVGL 移植。

首先我们把内存管理实验重命名为“LVGL 例程 1 无操作系统移植”工程，然后在该工程的 Middlewares 目录下新建 LVGL 文件夹，其次在该文件夹下新建 GUI 文件夹和 GUI_APP 文件夹，最后在 GUI 文件夹下新建 lvgl 文件夹，该文件夹保存图 2.1.2 中的文件和文件夹。

根据上述所述，可得到一个文件树形图，如下图所示：

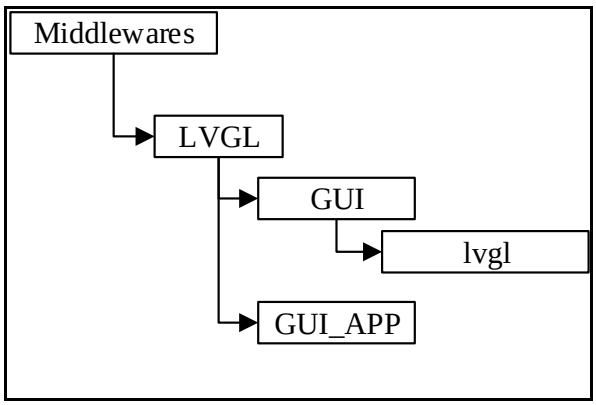


图 2.2.1 Middlewares 目录下文件夹树形图

我们的工程为什么使用这样的文件结构？因为 LVGL 的源文件内声明 lvgl.h 文件都是以“#include “../../lvgl.h”” 这样的形式声明的，显然这个文件的声明是放在第二层路径下，所以笔者把 LVGL 源码放在上图中的 lvgl 文件夹下，这样就符合了 LVGL 声明路径的风格。

2. 把 LVGL 源码移植到工程中

根据上图所示，把图 2.1.2 中的文件和文件夹复制到 Middlewaes/LVGL/GUI/lvgl 路径下，如下图所示：

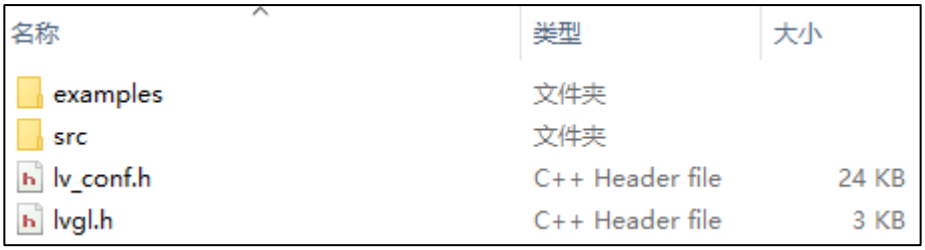


图 2.2.2 移植 LVGL 源码

3. 打开工程，添加工程管理项组

打开《LVGL 例程 1 无操作系统移植》工程，点击图标，添加下图的分组：

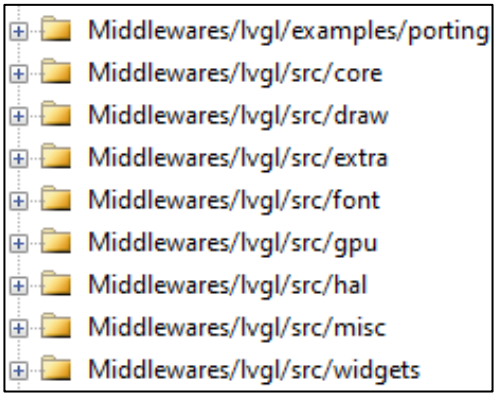


图 2.2.3 添加分组

打开工程下的 Middlewares/LVGL/GUI/lvgl/src 文件夹，我们会发现该文件夹下的文件夹名称与上图的分组名称相似，如图 2.2.4 所示：

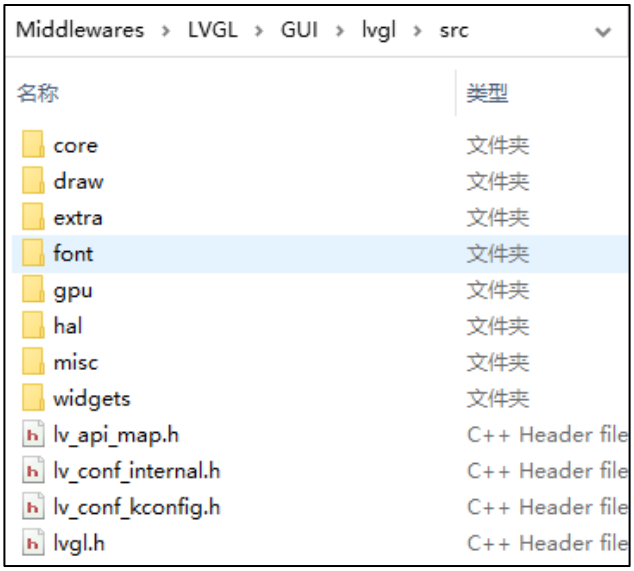


图 2.2.4 src 文件夹下的目录

根据图 2.2.3 和图 2.2.4 所示，这些分组需要添加的文件如下所示：

(1) Middlewares/lvgl/src/core 组添加 core 文件夹下的全部.c 文件，如下图所示：

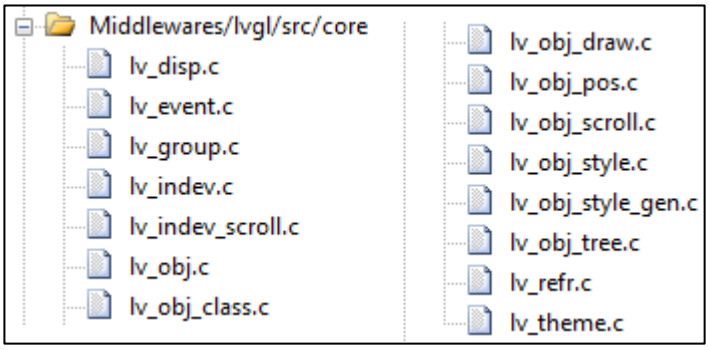


图 2.2.5 Middlewares/lvgl/src/core 组添加的文件

(2) Middlewares/lvgl/src/draw 组添加 draw 文件夹下除 nxp_pxp、nxp_vglite、sdl 和 stm32_dma2d 文件夹之外的全部.c 文件，如下图所示：

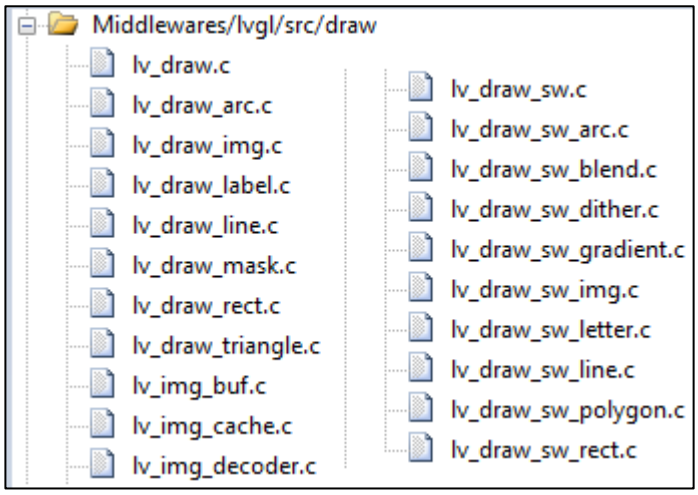


图 2.2.6 Middlewares/lvgl/src/draw 组添加的文件

(3) Middlewares/lvgl/src/extra 组添加 extra 文件夹下除 libs 文件夹之外的全部.c 文件，如下图所示：

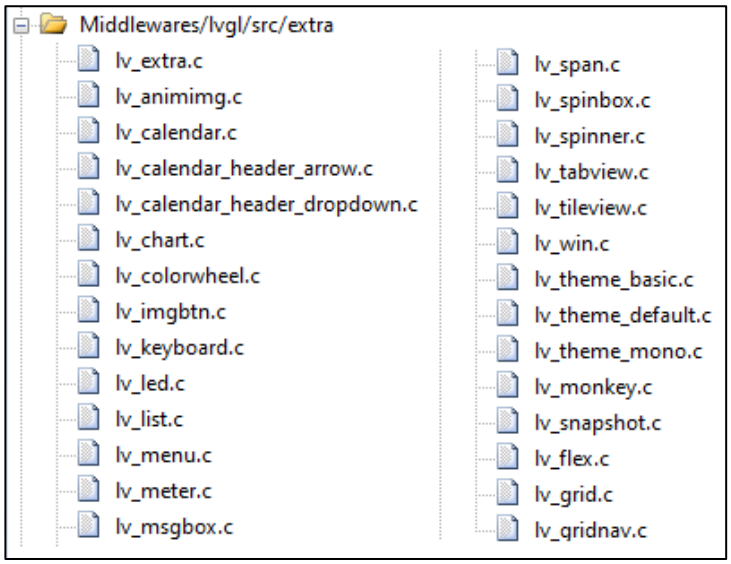


图 2.2.7 Middlewares/lvgl/src/extra 组添加的文件

(4) Middlewares/lvgl/src/font 组添加 font 文件夹下的全部.c 文件，如图 2.2.8 所示：

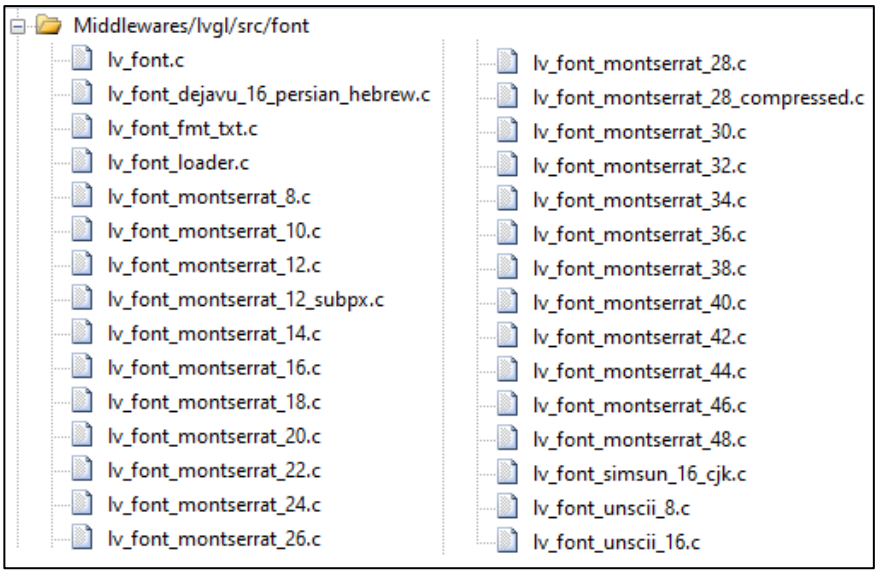


图 2.2.8Middlewares/lvgl/src/font 组添加的文件

(5) Middlewares/lvgl/src/gpu 组添加 draw/stm32_dma2d 和 draw/sdl 文件夹下的全部.c 文件，如图 2.2.9 所示：

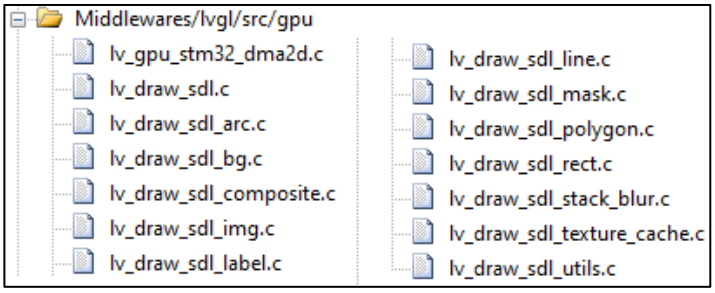


图 2.2.9 Middlewares/lvgl/src/gpu 组添加的文件

(6) Middlewares/lvgl/src/hal 组添加 hal 文件夹下的全部.c 文件，如图 2.2.10 所示：

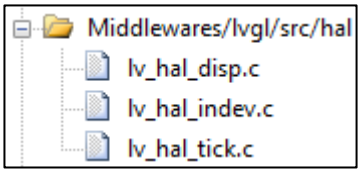


图 2.2.10 Middlewares/lvgl/src/hal 组添加的文件

(7) Middlewares/lvgl/src/misc 组添加 misc 文件夹下的全部.c 文件，如图 2.2.11 所示：

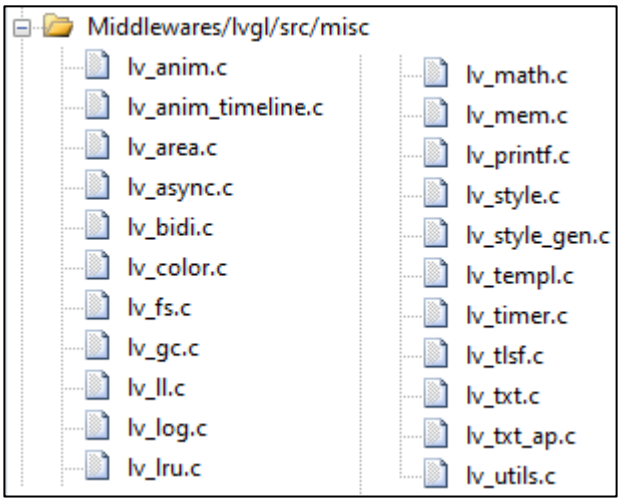


图 2.2.11 Middlewares/lvgl/src/misc 组添加的文件

(8) Middlewares/lvgl/src/widgets 组添加 widgets 文件夹下的全部.c 文件，如图 2.2.12 所示：

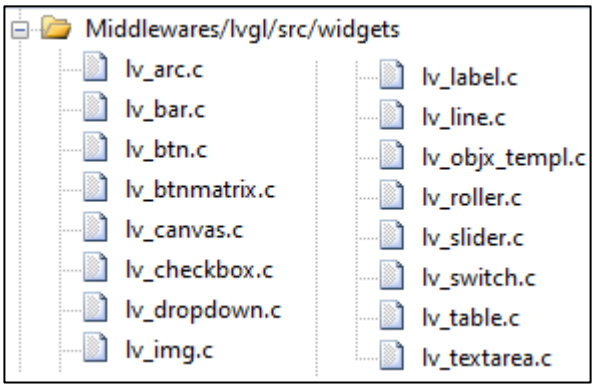


图 2.2.12 Middlewares/lvgl/src/widgets 组添加的文件

(9) Middlewares/lvgl/examples/porting 组添加 Middlewares/LVGL/GUI/lvgl/examples/porting 目录下的 lv_port_disp_template.c 和 lv_port_indev_template.c 文件，如图 2.2.13 所示：

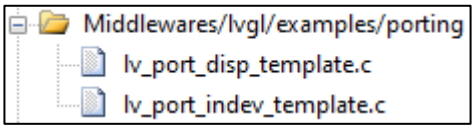


图 2.2.13 Middlewares/lvgl/examples/porting 组添加的文件

上图中的.c 文件与 LCD 和触摸驱动相关，这些文件我们在后面章节中讲解。

4. 添加文件路径

看到上述的步骤，很多人以为移植 LVGL 需要添加很多文件路径，其实不然，我们只添加关键路径即可，因为 lvgl.h 文件已经为我们省去这部分的操作，如图 2.2.14 所示：

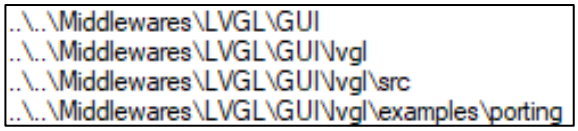


图 2.2.14 添加文件路径

5. 添加 touch 触摸驱动

该移植例程是没有添加 touch 触摸驱动的，虽然 LVGL 不强制使用触摸输入设备，但是正点原子的显示屏全都具备触摸功能的，因此我们在移植工程中添加 touch 触摸驱动文件，这个触摸驱动文件主要获取正点原子全系列显示屏的触摸坐标并把获取的坐标提供给 LVGL 输入设备，这样该输入设备就可以控制 UI 的部件了，如图 2.2.15 所示：

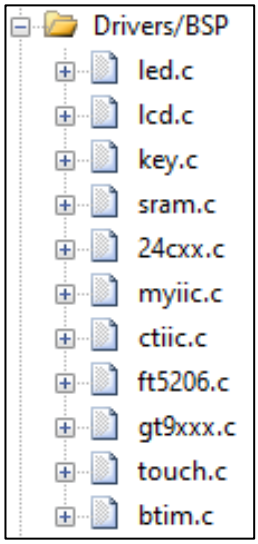


图 2.2.15 添加 touch 触摸驱动

6. 修改工程目标名称

本教程是以标准例程-HAL 库版本的内存管理实验工程为基础工程，内存管理实验工程的工程名为“MALLOC”，为了规范工程，笔者建议将工程的目标名修改为“LVGL”或根据读者的实际场景进行修改，修改如下图所示：

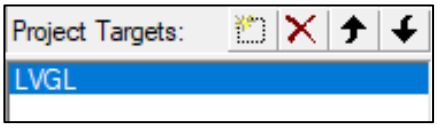


图 2.2.16 修改工程目标名

7. Keil5 忽略特定的警告

在 Misc Controls 中填入--diag_suppress=68 --diag_suppress=111 --diag_suppress=188 --diag_suppress=223 --diag_suppress=546 --diag_suppress=1295。这个方法有效消除 LVGL 带来的警告。如果我们不加这些命令，可能编译代码时会出现 30 个警告，笔者看到警告就浑身不舒服，就想着把警告解决的人，所以希望读者严格要求自己的工程不要出现过多的警告，

到了这一步，我们可以编译工程，显然工程编译是没有报错的，这样我们移植完成了吗，显然不是，我们还没有编写显示屏和触摸驱动与 LVGL 图形库衔接，更没有提供 LVGL 时基等操作，万事俱备，只欠东风。

2.3 修改工程文件

这里我们只需要“一加两改”。“一加”是指添加定时器提供 LVGL 的时基，“两改”是指修改 LVGL 显示驱动文件(lv_port_disp_templ.c/h)和输入设备文件(lv_port_indev_templ.c/h)。

1.“一加”：在 Drivers/BSP 分组下添加定时器驱动文件，如图 2.3.1 所示：

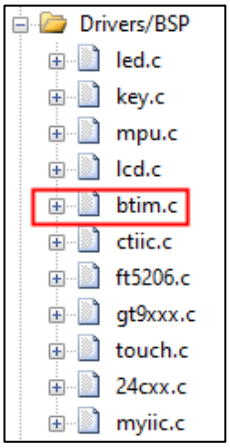


图 2.3.1 添加定时器文件

打开上图的 btim.c 文件，声明 LVGL 的头文件，如下源码所示：

```
#include "lvgl.h"
```

在该文件下的 HAL_TIM_PeriodElapsedCallback()函数添加以下源码：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim == (&g_timx_handle))
    {
        lv_tick_inc(1);    /* lvgl 的 1ms 心跳 */
    }
}
```

上述源码中，定时器服务函数调用了 LVGL 的 lv_tick_inc 函数，该函数让 LVGL 内部一个时基参数加 1。在 main 函数中，对定时器进行初始化操作，注意：根据自己的 MCU 开启定时器，定时器周期 1ms，如下表所示：

正点原子 STM32 系列开发板	定时器定时 1ms
Mini、精英、战舰	btim_timx_int_init(10-1,7200-1)
探索者、DMF407	btim_timx_int_init(10-1, 8400-1)
阿波罗 429、水星 429	btim_timx_int_init(10-1, 8900-1)
阿波罗 H743、水星 H743	btim_timx_int_init(10-1, 20000-1)
阿波罗 767、水星 767	btim_timx_int_init(10-1, 10700-1)
北极星 F750、水星 F750	
北极星 H750、Mini PRO H750	btim_timx_int_init(10-1, 24000-1)

表 2.3.1 不同的开发板定时器定时 1ms

2. “两改”：在文件 lv_port_disp_template.c/h 和 lv_port_indev_template.c/h 的条件编译指令#if 0 都修改成#if 1，如下源码所示：

修改：

```
#if 0   /* lv_port_disp_template.c 或者.h 和 lv_port_indev_template.c 或者.h */
```

修改为：

```
#if 1 /* 把#if 0 修改成#if 1 */
```

① lv_port_disp_template.c 文件修改

该文件作用是让自己的显示屏驱动文件与 LVGL 显示驱动做衔接操作。首先打开官方提供的 lv_port_disp_template.c 文件，我们会发现官方提供三种方法定义绘图缓冲区，该绘画缓存区主要渲染屏幕内容，定义缓冲区越大，刷新帧率越快。如下源码所示：

```
/*-----------------------------
 * Create a buffer for drawing
 *----------------------------*/

/* LVGL requires a buffer where it internally draws the widgets.
 * Later this buffer will passed your display drivers `flush_cb`
   to copy its content to your display.
 * The buffer has to be greater than 1 display row
 *
 * There are three buffering configurations:
 * 1. Create ONE buffer with some rows:
 *      LVGL will draw the display's content here and writes it to your display
 *
 * 2. Create TWO buffer with some rows:
 *    LVGL will draw the display's content to a buffer and writes it your display.
 *    You should use DMA to write the buffer's content to the display.
 *    It will enable LVGL to draw the next part of the screen to the
       other buffer while
 *    the data is being sent form the first buffer.
       It makes rendering and flushing parallel.
 *
 * 3. Create TWO screen-sized buffer:
 *      Similar to 2) but the buffer have to be screen sized.
        When LVGL is ready it will give the
 *      whole frame to display. This way you only need to change
        the frame buffer's address instead of
 *      copying the pixels.
 * */


/*-----------------------------
  * 创建绘图缓冲区
  *----------------------------*/
```

```
/* LVGL 需要一个内部绘制小部件的缓冲区。
    然后这个缓冲区会传递给你的显示驱动' flush_cb '来复制它的内容到你的显示。
    缓冲区必须大于 1 显示行
 *
 * 有三种缓冲配置:
 * 1. 创建一个缓冲区:
 *      LVGL 将在这里绘制显示的内容，并将其写入到显示
 *
 * 2. 创建两个缓冲区:
 *      LVGL 将绘制显示的内容到缓冲区，并写入显示.
 *      应该使用 DMA 将缓冲区的内容写入显示.
 *      它将使 LVGL 绘制下一部分的屏幕到另一个缓冲区，而同时
 *      数据正在从第一个缓冲区发送。它使呈现和刷新并行.
 *
 * 3.创建两个屏幕大小的缓冲区:
 *      类似于第二种，但缓冲区必须是屏幕大小。当 LVGL 准备好了，它会给整帧显示。
        这样你只需要改变帧缓冲区的地址而不是复制像素
 * */
```

根据上述所示，LVGL 定义绘图缓冲区具有三种类型，而正点原子 LVGL 的例程都是采用第一种方式定义绘图缓冲区，如下源码所示：

```
#include "lv_port_disp_template.h"
#include "../../lvgl.h"
/* 导入 lcd 驱动头文件 */
#include "./BSP/LCD/lcd.h"

/**********************
 *      DEFINES
 **********************/
#define USE_SRAM        0       /* 使用外部 sram 为 1，否则为 0 */
#ifdef USE_SRAM
#include "./MALLOC/malloc.h"
#endif

#define MY_DISP_HOR_RES (800)    /* 屏幕宽度 */
#define MY_DISP_VER_RES (480)    /* 屏幕高度 */

/* 显示设备初始化函数 */
static void disp_init(void);

/* 显示设备刷新函数 */
static void disp_flush(lv_disp_drv_t * disp_drv,
                       const lv_area_t * area,
```

```
                        lv_color_t * color_p);

/**
 * @brief        初始化并注册显示设备
 * @param        无
 * @retval       无
 */
void lv_port_disp_init(void)
{
    /*-------------------------
     * 初始化显示设备
     * -----------------------*/
    disp_init();

    /*-----------------------------
     * 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 中缓冲配置:
     * 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。
     *      LVGL 将始终以 'flush_cb' 的形式提供整个渲染屏幕，只需更改帧缓冲区的地址。
     */
/*  ..... (1) ..... */
/* 初始化 LCD */
disp_init();
/* 单缓冲区示例) */
static lv_disp_draw_buf_t draw_buf_dsc_1;
#if USE_SRAM
```

```
/* 设置缓冲区的大小为屏幕的全尺寸大小 */
void * buf_1 = mymalloc(SRAMEX, MY_DISP_HOR_RES * MY_DISP_VER_RES);
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&draw_buf_dsc_1,
                      buf_1, NULL,
                      MY_DISP_HOR_RES * MY_DISP_VER_RES);
#else
/* 设置缓冲区的大小为 10 行屏幕的大小 */
static lv_color_t buf_1[MY_DISP_HOR_RES * 10];
/*  ..................(2).................. */
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&draw_buf_dsc_1, buf_1, NULL, MY_DISP_HOR_RES * 10);
#endif
    /*-----------------------------------
     * 在 LVGL 中注册显示设备
     *-----------------------------------*/

    static lv_disp_drv_t disp_drv;                  /* 显示设备的描述符 */
    /*  ..................(3).................. */
    lv_disp_drv_init(&disp_drv);                    /* 初始化为默认值 */
    /* 建立访问显示设备的函数  */
    /* 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 */
    /*  ..................(4).................. */
    disp_drv.hor_res = lcddev.width;
    disp_drv.ver_res = lcddev.height;
    /*  ..................(5).................. */
    /* 用来将缓冲区的内容复制到显示设备 */
    disp_drv.flush_cb = disp_flush;
    /*  ..................(6).................. */
    /* 设置显示缓冲区 */
    disp_drv.draw_buf = &draw_buf_dsc_1;
    /*  ..................(7).................. */
    /* 注册显示设备 */
    lv_disp_drv_register(&disp_drv);
}

/**
 * @brief       初始化显示设备和必要的外围设备
 * @param       无
 * @retval      无
 */
```

```
static void disp_init(void)
{
    /*You code here*/
    lcd_init();          /* 初始化 LCD */
    lcd_display_dir(1); /* 设置横屏 */
}

/**
 * @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
 *   @arg       area        : 要刷新的区域，包含了填充矩形的对角坐标
 *   @arg       color_p     : 颜色数组
 *
 * @retval      无
 */
static void disp_flush(lv_disp_drv_t * disp_drv,
                       const lv_area_t * area,
                       lv_color_t * color_p)
{
    /* 重要！！！LCD 驱动函数，在指定区域内填充指定颜色块 */
    lcd_color_fill(area->x1,area->y1,area->x2,area->y2,(uint16_t*)color_p);
    /* 重要!!!* 通知图形库，已经刷新完毕了 */
    lv_disp_flush_ready(disp_drv);
}
```

从上述源码可知，我们知道配置 LVGL 显示屏驱动步骤如下：

根据 USE_SRAM 配置项选择内存类型，内存类型分别选择外部 SRAM 和内部 SRAM(笔者建议使用内部 SRAM 分配)。

(1) 调用函数 disp_init 初始化 LCD 驱动。

(2) 调用函数 lv_disp_draw_buf_init 初始化缓冲区(最低标准：显示屏的宽度分配率*10)。

(3) 调用函数 lv_disp_drv_init 初始化显示驱动。

(4) 设置显示的高度与宽度。

(5) 注册显示驱动回调。

(6) 设置显示驱动的绘画缓冲区。

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

上述的步骤笔者在程序中以(1)~(7)标号标出了，它们的作用就是把 LVGL 显示屏驱动与自己的 LCD 驱动衔接起来，在这个文件中我们只提供两个函数，它们分别为 LCD 初始化函数和 LCD 填充函数。

注意：如果用户使用 DMA2D 外设，请大家在 lv_conf.h 文件中的 LV_USE_GPU_STM32_DMA2D 宏定义置 1 并在 LV_GPU_DMA2D_CMSIS_INCLUDE 添加 MCU 的头文件路径，如下表所示：

正点原子具备 RGBLCD 接口的 STM32 开发板	LV_GPU_DMA2D_CMSIS_INCLUDE
阿波罗 429、水星 429	“#include stm32f429xx.h”
阿波罗 H743、水星 H743	“#include stm32h743xx.h”
阿波罗 767、水星 767	“#include stm32h767xx.h”
北极星 F750、水星 F750	“#include stm32f750xx.h”
北极星 H750	“#include stm32h750xx.h”

表 2.3.2 不同的开发板所设置的文件路径

上表的 LV_GPU_DMA2D_CMSIS_INCLUDE 是根据自己的 MCU 来设置头文件路径，注意： disp_flush 函数下的 lcd_color_fill 函数可以判断显示屏的类型，如果显示屏是 TFTLCD，则程序调用 lcd_color_fill 函数绘制图形；如果显示屏是 RGBLCD，则程序执行 ltdc_color_fill 函数，该函数使用 DMA2D 的等待方式实现的。当然用户可以使用 DMA2D 的中断方法来刷新显示屏，这种方式需要在 lv_port_disp_template 文件下修改 disp_init 和 disp_flush 函数，这些函数修改方式以下所示：

```
volatile uint8_t lv_gpu_state = 0;

/**
 * @brief       dma2d 中断服务函数
 * @param       无
 * @retval      无
 */
void DMA2D_IRQHandler(void)
{

  if ((DMA2D->ISR & DMA2D_FLAG_TC) != 0U)
  {
    if ((DMA2D->CR & DMA2D_IT_TC) != 0U)
    {
            /* 清除标志位 */
            DMA2D->CR &= ~DMA2D_IT_TC;
            DMA2D->IFCR = DMA2D_FLAG_TC;
            /* 刷新显示设备 */
            if(lv_gpu_state==1){
                 lv_gpu_state = 0;
                 lv_disp_flush_ready(&disp_drv);
             }
    }
  }
}

/**
 * @brief       dma2d 采用寄存器初始化
 * @param       无
 * @retval      无
 */
```

```
static void dma2d_reg_init(void)
{
    /* 设置 DMA2D 中断优先级 */
    HAL_NVIC_SetPriority(DMA2D_IRQn, 3, 0);
    /* 使能 DMA2D 中断*/
    HAL_NVIC_EnableIRQ(DMA2D_IRQn);
    /* 使能 DMA2D 外设时钟*/
    __HAL_RCC_DMA2D_CLK_ENABLE();
}

/**
 * @brief       初始化显示设备和必要的外围设备
 * @param       无
 * @retval      无
 */
static void disp_init(void)
{
    /*You code here*/
    lcd_init();          /* 初始化 LCD */
    lcd_display_dir(1); /* 设置横屏 */
    dma2d_reg_init();    /* 初始化 DMA2D */
}

static void disp_flush(lv_disp_drv_t * disp_drv,
                       const lv_area_t * area, lv_color_t * color_p)
{
    /* LVGL 官方给出的一个打点刷新屏幕的例子，但这个效率是最低效的 */

    uint32_t OffLineSrc = lcddev.width - (area->x2 - area->x1 + 1);
    uint32_t addr = LCD_FRAME_BUF_ADDR + 2*(lcddev.width * area->y1+area->x1);
    /* 中断传输 */
    /* 内存到内存模式 */
    DMA2D->CR      = 0x00000000UL | (1 << 9);
    /* 源地址 */
    DMA2D->FGMAR   = (uint32_t)(uint16_t*)(color_p);
    /* 目标地址 */
    DMA2D->OMAR    = (uint32_t)addr;

    /* 输入偏移 */
    DMA2D->FGOR    = 0;
    /* 输出偏移 */
    DMA2D->OOR     = OffLineSrc;
```

```
    /* 前景层和输出区域都采用的 RGB565 颜色格式 */
    DMA2D->FGPFCCR = DMA2D_OUTPUT_RGB565;
    DMA2D->OPFCCR  = DMA2D_OUTPUT_RGB565;

    /* 多少行 */
    DMA2D->NLR     = (area->y2 - area->y1 + 1)|((area->x2 - area->x1 +1)<< 16);

    /* 开启中断 */
    DMA2D->CR |= DMA2D_IT_TC|DMA2D_IT_TE|DMA2D_IT_CE;

    /* 启动传输 */
    DMA2D->CR   |= DMA2D_CR_START;
    lv_gpu_state = 1;
}
```

上述的源码就是采用 DMA2D 的中断方式来刷新显示屏，值得注意的是：使用 DMA2D 刷新显示屏可能在竖屏状态下乱码，这个问题可能是 DMA2D 刷新方向有关，这里笔者没有很好的解决方案，大家可自行研究，如果用户想使用 RGB 屏并且以竖屏的方式显示 UI，为此笔者编写了一个以画点方式刷新显示屏的函数，该函数如下源码所示：

```
#define LV_FIND_MIN(a,b) (a < b ? a : b)


void ltdc_color_fill( uint16_t sx,uint16_t sy,uint16_t ex,
                      uint16_t ey,uint16_t *color)
{
  uint16_t fillw,fillh;
  uint16_t x,y,i,j;
  fillw = ((ex - sx) > 0) ? (ex - sx + 1) : (sx - ex + 1);
  fillh = ((ey - sy) > 0) ? (ey - sy + 1) : (sy - ey + 1);
  x = LV_FIND_MIN(sx,ex);
  y = LV_FIND_MIN(sy,ey);

  for(i = y ; i < y + fillh ; i ++)
  {
    for(j = x ; j < x + fillw ; j ++)
    {
      ltdc_draw_point(j,i,(uint16_t)*(color));
      color ++;
    }
  }
}
```

这个方式是使用画点的方式来刷新显示屏，这种方法可能导致帧率过低。

② lv_port_indev_template.c 文件修改

该文件的作用是添加输入设备，例如触摸设备、鼠标、键盘、编码器、按键等输入设备，针对正点原子的例程来讲：只添加触摸输入设备即可，如下源码所示：

```
/*********************
 *      INCLUDES
 *********************/
#include "lv_port_indev_template.h"
#include "../../lvgl.h"
/* 导入屏幕触摸驱动头文件 */
#include "./BSP/TOUCH/touch.h"

/* 触摸屏 */
static void touchpad_init(void);
static void touchpad_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
static bool touchpad_is_pressed(void);
static void touchpad_get_xy(lv_coord_t * x, lv_coord_t * y);
lv_indev_t * indev_touchpad;     /* 触摸屏 */

/**
 * @brief       初始化并注册输入设备
 * @param       无
 * @retval      无
 */
void lv_port_indev_init(void)
{
    /**
     *
     * 在这里你可以找到 LittlevGL 支持的出入设备的实现示例:
     *  - 触摸屏
     *  - 鼠标 (支持光标)
     *  - 键盘 (仅支持按键的 GUI 用法)
     *  - 编码器 (支持的 GUI 用法仅包括: 左, 右, 按下)
     *  - 按钮 (按下屏幕上指定点的外部按钮)
     *
     *  函数 `..._read()` 只是示例
     *  你需要根据具体的硬件来完成这些函数
     */

    static lv_indev_drv_t indev_drv;

    /*------------------
     * 触摸屏
     * ------------------*/
    /* ..... (1) ..... */
```

```
    /* 初始化触摸屏(如果有) */
    touchpad_init();
    /* ……(2)…… */
    /* 注册触摸屏输入设备 */
    lv_indev_drv_init(&indev_drv);
    /* ……(3)…… */
    indev_drv.type = LV_INDEV_TYPE_POINTER;
    /* ……(4)…… */
    indev_drv.read_cb = touchpad_read;
    /* ……(5)…… */
    indev_touchpad = lv_indev_drv_register(&indev_drv);
}

/*------------------
 * 触摸屏
 * ------------------*/

/**
 * @brief       初始化触摸屏
 * @param       无
 * @retval      无
 */
static void touchpad_init(void)
{
    /*Your code comes here*/
    tp_dev.init();
    /* 电阻屏如果发现显示屏 XY 镜像现象，需要坐标矫正 */
    if (0 == (tp_dev.touchtype & 0x80)) {
        tp_adjust();
        tp_save_adjust_data();
    }
}

/**
 * @brief       图形库的触摸屏读取回调函数
 * @param       indev_drv   : 触摸屏设备
 *   @arg       data        : 输入设备数据结构体
 * @retval      无
 */
static 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);
        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;
}

/**
 * @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)
{
    /*Your code comes here*/
    tp_dev.scan(0);
    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
 *   @arg       y   : y 坐标的指针
 * @retval      无
 */
static void touchpad_get_xy(lv_coord_t * x, lv_coord_t * y)
{
    /*Your code comes here*/
    (*x) = tp_dev.x[0];
```

```
    (*y) = tp_dev.y[0];
}
```

从上述源码，我们知道配置 LVGL 的 touch 触摸驱动实现步骤，如下所示：

(1) 调用 touchpad_init 函数初始化 touch 驱动。

(2) 调用函数 lv_indev_drv_init 初始化初始化输入设备。

(3) 设置设备的的类型，因为是 touch 触摸所以修改成 LV_INDEV_TYPE_POINTER 为指示器类型。

(4) 设置触摸回调函数为 touchpad_read，该函数是获取触摸屏的坐标。

(5) 调用函数 lv_indev_drv_register 输入设备的登记。

上述的步骤笔者已经在程序中以(1)~(7)的标号注释了。LVGL 官方提供获取触摸的函数是 touchpad_get_xy，该函数的*x 和*y 的值需要用户提供，由于正点原子已经拥有了 touch 触摸驱动，所以我们可调用 tp_dev.x[0]和 tp_dev.y[0]取 XY 轴的触摸值并传入*x 和*y 参数中。

编译工程，工程可能出现图 2.3.2 的错误。

```
No space in execution regions with .ANY selector matching
No space in execution regions with .ANY selector matching
No space in execution regions with .ANY selector matching
No space in execution regions with .ANY selector matching
```

图 2.3.2 内存不足编译错误

遇到图 2.3.2 中的错误，可在 malloc.h 文件中修改 MEM1_MAX_SIZE 宏定义的大小。

注意：移植 LVGL 必须开启 C99 模式，如图 2.3.3 所示：

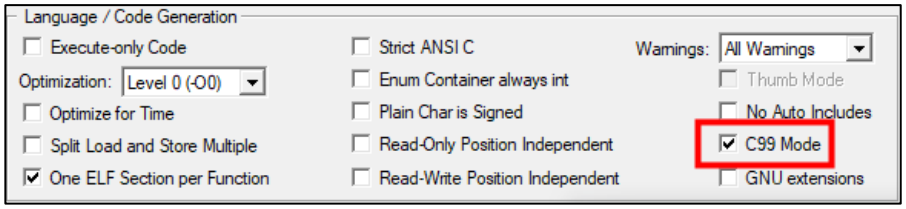


图 2.3.3 开启 C99 模式

好了，到了这里，我们测试一下 LVGL 是否移植成功，在工程中添加以下源码：

```
#include "lvgl.h"
#include "lv_port_indev_template.h"
#include "lv_port_disp_template.h"

int main(void)
{
    /* 系统其他硬件初始化信息，这里忽略 */
    btim_timx_int_init(10-1,7200-1);    /* 根据自己的开发板 MCU 定义定时器周期为 1ms */
    lv_init();                          /* lvgl 系统初始化 */
    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 */
    lv_obj_t *label = lv_label_create(lv_scr_act());
    lv_obj_set_style_text_color(label, lv_palette_main(LV_PALETTE_RED),
                                LV_STATE_DEFAULT);
    lv_obj_set_style_bg_color(lv_scr_act(),lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
```

```
    lv_obj_set_style_text_font(label,&lv_font_montserrat_14, LV_STATE_DEFAULT);
    lv_label_set_text(label,"Hello Alientek!!!");
    while(1)
    {
        lv_timer_handler();     /* LVGL 管理函数相当于 RTOS 触发任务调度函数 */
    }
}
```

根据上述的源码运行效果，如图 2.3.4 所示：



图 2.3.4 显示”Hello Alientek”字体

2.4 移植官方例程

在上一章节中，笔者已经说过，LVGL 的官方例程是在 demos 文件夹下保存着，接下来笔者介绍一下官方例程的移植流程。

(1) 把 demos 文件夹复制到 Middlewares/LVGL/GUI_APP 路径下，如图 2.4.1 所示：



图 2.4.1 移植官方例程

(2) 添加文件路径

由于 demos 文件夹有多个官方例程，不同的例程所添加的路径都不一样，这里笔者演示的官方例程 music，这个例程需要添加以下路径，如图 2.4.2 所示：

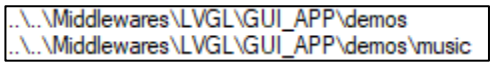


图 2.4.2 添加文件路径

(3) 打开 lv_conf.h 文件，找到宏定义 LV_USE_DEMO_MUSIC 并设置为 1 开启该实验。

(4) Middlewares/LVGL/GUI_APP 工程管理项组添加 demos/music 路径下的全部.c 文件，如图 2.4.3 所示：

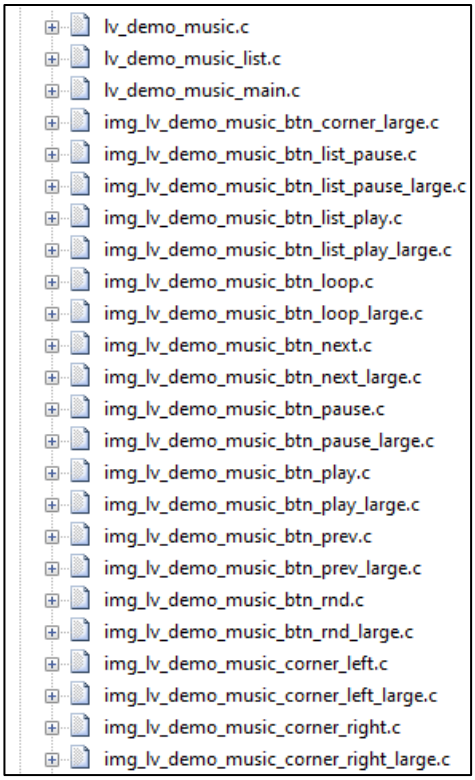


图 2.4.3 官方例程文件

(5) 编译工程，工程可能出现图 2.4.4 的错误：

```
error:  #20: identifier "lv_font_montserrat_12" is undefined

error:  #20: identifier "lv_font_montserrat_16" is undefined
```

图 2.4.4 字体没有定义

遇到上图的错误，打开 lv_conf.h 文件找到相应的字库并设置为 1，如图 2.4.5 所示：

```
/* 蒙特塞拉特字体的ASCII范围和一些符号使用bpp = 4
 * https://fonts.google.com/specimen/Montserrat */
#define LV_FONT_MONTSERRAT_8  0
#define LV_FONT_MONTSERRAT_10  0
#define LV_FONT_MONTSERRAT_12  1
#define LV_FONT_MONTSERRAT_14  1
#define LV_FONT_MONTSERRAT_16  1
#define LV_FONT_MONTSERRAT_18  0
#define LV_FONT_MONTSERRAT_20  0
#define LV_FONT_MONTSERRAT_22  1
#define LV_FONT_MONTSERRAT_24  0
#define LV_FONT_MONTSERRAT_26  0
#define LV_FONT_MONTSERRAT_28  0
#define LV_FONT_MONTSERRAT_30  0
#define LV_FONT_MONTSERRAT_32  1
#define LV_FONT_MONTSERRAT_34  0
#define LV_FONT_MONTSERRAT_36  0
#define LV_FONT_MONTSERRAT_38  0
#define LV_FONT_MONTSERRAT_40  0
#define LV_FONT_MONTSERRAT_42  0
#define LV_FONT_MONTSERRAT_44  0
#define LV_FONT_MONTSERRAT_46  0
#define LV_FONT_MONTSERRAT_48  0
```

图 2.4.5 定义字体

再一次编译就没有报错了，在 main 函数添加 lv_demo_music 函数，如下源码所示：

```
int main(void)
{
    /* 各种初始化/时钟/驱动等等，省略 */
    btim_timx_int_init(10-1,7200-1);            /* 这里以 F103 为例初始化定时器  */
    lv_init();                                  /* lvgl 系统初始化 */
    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 */
    lv_demo_music();

    while(1)
    {
        lv_timer_handler();     /* LVGL 管理 */
        HAL_Delay(5);
    }
}
```

2.5 下载验证

请参考“LVGL 例程 1 无操作系统移植”实验。工程编译并下载到开发板中，注意：请使用 800*480 分辨率的显示屏，因为这个官方例程规定是 800*480 分辨率的，如图 2.5.1 所示：

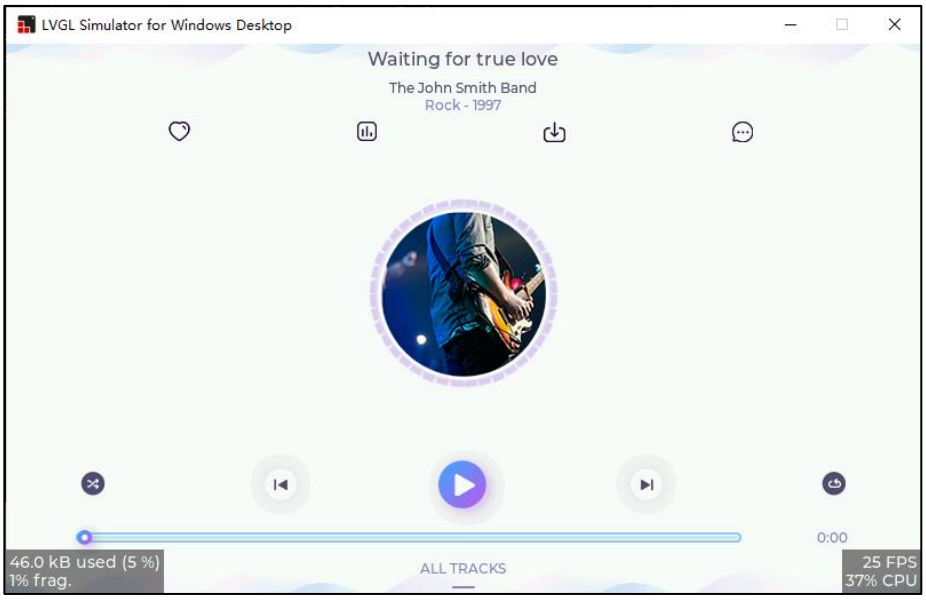


图 2.5.1 运行官方例程

第三章 LVGL 带操作系统移植

本章主要讲解操作系统的 LVGL 移植，上面笔者也讲解过，虽然 LVGL 是支持操作系统的，但是并不是必需的，因为 LVGL 内部实现机制和 RTOS 实现机制是类似的，所以该工程可能存在线程不安全的问题。如果我们单单涉及纯软件的项目，则建议不使用 RTOS；如果开发项目涉及其他硬件传感器，则建议使用 RTOS 来管理。

本章分为如下几个部分讲解：

3.1 移植准备工作

3.2 添加应用程序

3.3 下载验证

3.1 移植准备工作

LVGL 移植操作系统并不需要像 emWin 一样修改内部代码和替换兼容操作系统文件，我们只需要把上一章中移植工程换成任务或者线程的形式，关于 FreeRTOS 的移植，请参考正点原子《FreeRTOS 开发指南》。

在裸机移植时候，我们使用定时器来提供 LVGL 时基，使用该方法能准确的提供时基。LVGL 作者为了用户考虑，他提供了第二种方法，那就是使用 RTOS 来提供时基，首先打开 lv_conf.h 文件，在这个文件下把 LV_TICK_CUSTOM 宏定义置 1，然后我们设置 LV_TICK_CUSTOM_INCLUDE 和 LV_TICK_CUSTOM_SYS_TIME_EXPR 配置项，如下源码所示：

```
/* 使用自定义 tick 源，以毫秒为单位告诉运行时间。它不需要手动更新 `lv_tick_inc()函数` */
#define LV_TICK_CUSTOM                          1
#if LV_TICK_CUSTOM
#define LV_TICK_CUSTOM_INCLUDE                "FreeRTOS.h"      /* 系统时间函数头 */
/* 计算系统当前时间的表达式(以毫秒为单位) */
    #define LV_TICK_CUSTOM_SYS_TIME_EXPR     (xTaskGetTickCount())
#endif    /*LV_TICK_CUSTOM*/
```

上述源码可知：我们使用 FreeRTOS 的 xTaskGetTickCount 函数来提供 LVGL 时钟节拍。注意：在 FreeRTOSConfig.h 文件中，必须把 configTICK_RATE_HZ 宏定义设置为 1000。

3.2 添加应用程序

下面我们来讲解一下 LVGL 带操作系统的工程结构，该工程在 LVGL 裸机工程上一共需要修改 1 个文件并添加 4 个文件，修改的 1 个文件为 main.c，添加的 4 个文件为 lvgl_demo.c、lvgl_demo.h、lv_mainstart.c 和 lv_mainstart.h。对于 main.c 主要是在 main 函数中完成一些硬件的初始化，最后调用 lvgl_demo.c 文件中的 lvgl_demo 函数。lvgl_demo.c 则是用于编写 FreeRTOS 的相关应用程序代码。而 lv_mainstart.c 则是用于编写 LVGL 的相关应用程序代码。

注意：lvgl_demo.c、lvgl_demo.h 文件保存在 User 文件夹下，而 lv_mainstart.c、lv_mainstart.h 文件保存在 Middlewares\LVGL\GUI_APP 文件夹下。

1. main.c

由于正点原子的 STM32 系列开发板众多，这里以正点原子战舰开发板为例为读者进行演示，读者可以根据自己所移植的目标开发板在本教程的配套例程源码的《LVGL 例程 5 LVGL 文件系统使用》中查看对应的 main.c 文件。正点原子战舰开发板《LVGL 例程 5 LVGL 文件系

统使用》中的 main.c 文件如下所示：

```
/**

****************************************************************************
 * @file        main.c
 * @author      正点原子团队(ALIENTEK)
 * @version     V1.4
 * @date        2022-01-04
 * @brief       LVGL 文件系统使用 实验
 * @license     Copyright (c) 2020-2032, 广州市星翼电子科技有限公司

****************************************************************************
 * @attention
 *
 * 实验平台:正点原子 STM32F103 开发板
 * 在线视频:www.yuanzige.com
 * 技术论坛:www.openedv.com
 * 公司网址:www.alientek.com
 * 购买地址:openedv.taobao.com
 *

****************************************************************************
 */

#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"
#include "./BSP/LED/led.h"
#include "./BSP/LCD/lcd.h"
#include "./BSP/KEY/key.h"
#include "./BSP/SRAM/sram.h"
#include "./MALLOC/malloc.h"
#include "./BSP/TOUCH/touch.h"
#include "lvgl_demo.h"

int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72);                         /* 延时初始化 */
    usart_init(115200);                 /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
```

```
    key_init();                             /* 初始化按键 */
    sram_init();                            /* SRAM 初始化 */
    tp_dev.init();                          /* 触摸屏初始化 */
    my_mem_init(SRAMIN);                    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX);                    /* 初始化外部 SRAM 内存池 */

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

可以看到，在 main.c 文件中只包含了一个 main 函数，该函数主要就是完成了一些外设初始化，如串口、LED、LCD、按键等，并在最后调用了函数 lvgl_demo。

2. lvgl_demo.c

```
/**
 ****************************************************************************
 * @file        lvgl_demo.c
 * @author      正点原子团队(ALIENTEK)
 * @version     V1.0
 * @date        2022-01-11
 * @brief       LVGL 文件系统使用 实验
 * @license     Copyright (c) 2020-2032, 广州市星翼电子科技有限公司

****************************************************************************
 * @attention
 *
 * 实验平台:正点原子 STM32F103 开发板
 * 在线视频:www.yuanzige.com
 * 技术论坛:www.openedv.com
 * 公司网址:www.alientek.com
 * 购买地址:openedv.taobao.com
 *

****************************************************************************/

#include "lvgl_demo.h"
#include "./BSP/LED/led.h"
#include "FreeRTOS.h"
#include "task.h"

#include "lvgl.h"
#include "lv_port_disp_template.h"
#include "lv_port_indev_template.h"
#include "LVGL/GUI_APP/lv_mainstart.h"
```

```
/******************************************************************************/
/*FreeRTOS 配置*/

/* START_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define START_TASK_PRIO     1           /* 任务优先级 */
#define START_STK_SIZE      128         /* 任务堆栈大小 */
TaskHandle_t StartTask_Handler;         /* 任务句柄 */
void start_task(void *pvParameters);    /* 任务函数 */

/* LV_DEMO_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define LV_DEMO_TASK_PRIO   3           /* 任务优先级 */
#define LV_DEMO_STK_SIZE    1024        /* 任务堆栈大小 */
TaskHandle_t LV_DEMOTask_Handler;       /* 任务句柄 */
void lv_demo_task(void *pvParameters);  /* 任务函数 */

/* LED_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define LED_TASK_PRIO       4           /* 任务优先级 */
#define LED_STK_SIZE        128         /* 任务堆栈大小 */
TaskHandle_t LEDTask_Handler;           /* 任务句柄 */
void led_task(void *pvParameters);      /* 任务函数 */
/******************************************************************************/


void lvgl_demo(void)
{
    lv_init();                  /* lvgl 系统初始化 */
    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 */

    xTaskCreate((TaskFunction_t )start_task,            /* 任务函数 */
                (const char*    )"start_task",          /* 任务名称 */
                (uint16_t       )START_STK_SIZE,        /* 任务堆栈大小 */
                (void*          )NULL,                  /* 传递给任务函数的参数 */
                (UBaseType_t    )START_TASK_PRIO,       /* 任务优先级 */
                (TaskHandle_t*  )&StartTask_Handler);   /* 任务句柄 */

    vTaskStartScheduler();                              /* 开启任务调度 */
```

```
}

/**
 * @brief       start_task
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           /* 进入临界区 */

    /* 创建 LVGL 任务 */
    xTaskCreate((TaskFunction_t )lv_demo_task,
                (const char*    )"lv_demo_task",
                (uint16_t       )LV_DEMO_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )LV_DEMO_TASK_PRIO,
                (TaskHandle_t*  )&LV_DEMOTask_Handler);

    /* LED 测试任务 */
    xTaskCreate((TaskFunction_t )led_task,
                (const char*    )"led_task",
                (uint16_t       )LED_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )LED_TASK_PRIO,
                (TaskHandle_t*  )&LEDTask_Handler);

    taskEXIT_CRITICAL();            /* 退出临界区 */
    vTaskDelete(StartTask_Handler); /* 删除开始任务 */
}

/**
 * @brief       LVGL 运行例程
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void lv_demo_task(void *pvParameters)
{
    lv_mainstart();  /* 测试的 demo */

    while(1)
    {
        lv_timer_handler(); /* LVGL 计时器 */
```

```
        vTaskDelay(5);
    }
}

/**
 * @brief       系统再运行
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void led_task(void *pvParameters)
{
    while(1)
    {
        LED0_TOGGLE();
        vTaskDelay(1000);
    }
}
```

对于 lvgl_demo.c 文件，这里先简单的介绍一下这个文件的代码结构，这个文件的代码结构可分为 6 个部分，分别是包含头文件、FreeRTOS 相关配置、全局变量及自定义函数、应用程序入口函数、开始任务入口函数、其他任务入口函数，接下来分别地介绍以上这几个部分的代码。

(1) 包含头文件

包含的头文件分成两个部分，分别为 FreeRTOS 头文件、LVGL 头文件和其他头文件，如下所示：

```
#include "lvgl_demo.h"
#include "./BSP/LED/led.h"
#include "FreeRTOS.h"
#include "task.h"

#include "lvgl.h"
#include "lv_port_disp_template.h"
#include "lv_port_indev_template.h"
#include "LVGL/GUI_APP/lv_mainstart.h"
```

(2) FreeRTOS 相关配置

FreeRTOS 的配置主要包括所创建 FreeRTOS 任务的相关定义（任务优先级、任务堆栈大小、任务句柄、任务函数）以及 FreeRTOS 相关变量（信号量、事件、列表、软件定时器等）的定义，如下所示：

```
/**************************************************************************/
/*FreeRTOS 配置*/

/* START_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
```

```
#define START_TASK_PRIO      1              /* 任务优先级 */
#define START_STK_SIZE       128            /* 任务堆栈大小 */
TaskHandle_t StartTask_Handler;             /* 任务句柄 */
void start_task(void *pvParameters);        /* 任务函数 */

/* LV_DEMO_TASK 任务 配置
 * 包括：任务句柄 任务优先级 堆栈大小 创建任务
 */
#define LV_DEMO_TASK_PRIO    3              /* 任务优先级 */
#define LV_DEMO_STK_SIZE     1024           /* 任务堆栈大小 */
TaskHandle_t LV_DEMOTask_Handler;           /* 任务句柄 */
void lv_demo_task(void *pvParameters);      /* 任务函数 */

/* LED_TASK 任务 配置
 * 包括：任务句柄 任务优先级 堆栈大小 创建任务
 */
#define LED_TASK_PRIO        4              /* 任务优先级 */
#define LED_STK_SIZE         128            /* 任务堆栈大小 */
TaskHandle_t LEDTask_Handler;               /* 任务句柄 */
void led_task(void *pvParameters);          /* 任务函数 */
/************************************************************************************/
```

(3) 应用程序入口函数

这部分就是函数 lvgl_demo()，函数 lvgl_demo()一开始就是创建开始任务，最后开启 FreeRTOS 系统任务调度，如下所示：

```
void lvgl_demo(void)
{
    lv_init();                  /* lvgl 系统初始化 */
    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 */

    xTaskCreate((TaskFunction_t )start_task,            /* 任务函数 */
                (const char*    )"start_task",          /* 任务名称 */
                (uint16_t       )START_STK_SIZE,        /* 任务堆栈大小 */
                (void*          )NULL,                  /* 传递给任务函数的参数 */
                (UBaseType_t    )START_TASK_PRIO,       /* 任务优先级 */
                (TaskHandle_t*  )&StartTask_Handler);   /* 任务句柄 */

    vTaskStartScheduler();                              /* 开启任务调度 */
}
```

(4) 开始任务入口函数

这部分就是开始任务的入口函数，开始任务主要用于创建或初始化特定实验中使用到的一些硬件外设和 FreeRTOS 相关的软件（信号量、事件、列表、软件定时器等）以及创建其他用于实验演示的任务，如下所示：

```
/**
 * @brief       start_task
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           /* 进入临界区 */

    /* 创建 LVGL 任务 */
    xTaskCreate((TaskFunction_t )lv_demo_task,
                (const char*    )"lv_demo_task",
                (uint16_t       )LV_DEMO_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )LV_DEMO_TASK_PRIO,
                (TaskHandle_t*  )&LV_DEMOTask_Handler);

    /* LED 测试任务 */
    xTaskCreate((TaskFunction_t )led_task,
                (const char*    )"led_task",
                (uint16_t       )LED_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )LED_TASK_PRIO,
                (TaskHandle_t*  )&LEDTask_Handler);

    taskEXIT_CRITICAL();            /* 退出临界区 */
    vTaskDelete(StartTask_Handler); /* 删除开始任务 */
}
```

(6) 其他任务入口函数

这部分就包含了实验中用于演示的任务入口函数，如下所示：

```
/**
 * @brief       LVGL 运行例程
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void lv_demo_task(void *pvParameters)
{
    lv_mainstart();  /* 测试的 demo */

    while(1)
    {
        lv_timer_handler(); /* LVGL 计时器 */
        vTaskDelay(5);
```

```
    }
}

/**
 * @brief       系统再运行
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void led_task(void *pvParameters)
{
    while(1)
    {
        LED0_TOGGLE();
        vTaskDelay(1000);
    }
}
```

lv_demo_task 任务函数调用函数 lv_mainstart 执行 LVGL 例程，该函数是在 lv_mainstart.c 文件定义。而 led_task 任务函数主要表示系统是否再运行。本教程配套的实验例程都会按着这个代码结构来进行实验代码的编写，建议读者先熟悉这个代码结构。

3. lvgl_demo.h

```
#ifndef __LVGL_DEMO_H
#define __LVGL_DEMO_H

void lvgl_demo(void);

#endif
```

lvgl_demo.h 这个文件很简单，就是将函数 lvgl_demo 导出给其他 C 源文件调用。

4. lv_mainstart.c

该文件主要由用户编写的 LVGL 程序，如下所示：

```
/**
 * @brief       lvgl(lvgl 程序入口)
 * @param       无
 * @retval      无
 */
void lv_mainstart (void)
{
    /* 编写 lvgl 代码 */
}
```

5. lv_mainstart.h

```
#ifndef _LVGL_MAINSTART_H
#define _LVGL_MAINSTART_H
#include "./SYSTEM/sys/sys.h"
```

```
void lv_mainstart (void); /* lvgl 例程入口 */
#endif
```

lv_mainstart.h 这个文件很简单，就是将函数 lv_mainstart 导出给其他 C 源文件调用。

3.3 编写综合测试程序

3.3.1 硬件设计

1. 例程功能

本实验主要测试官方例程，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要执行官方例程。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 2 操作系统移植》。

3.3.2 软件设计

3.3.2.1 程序流程图

本实验的程序流程图，如下图 3.3.2.1.1 所示：

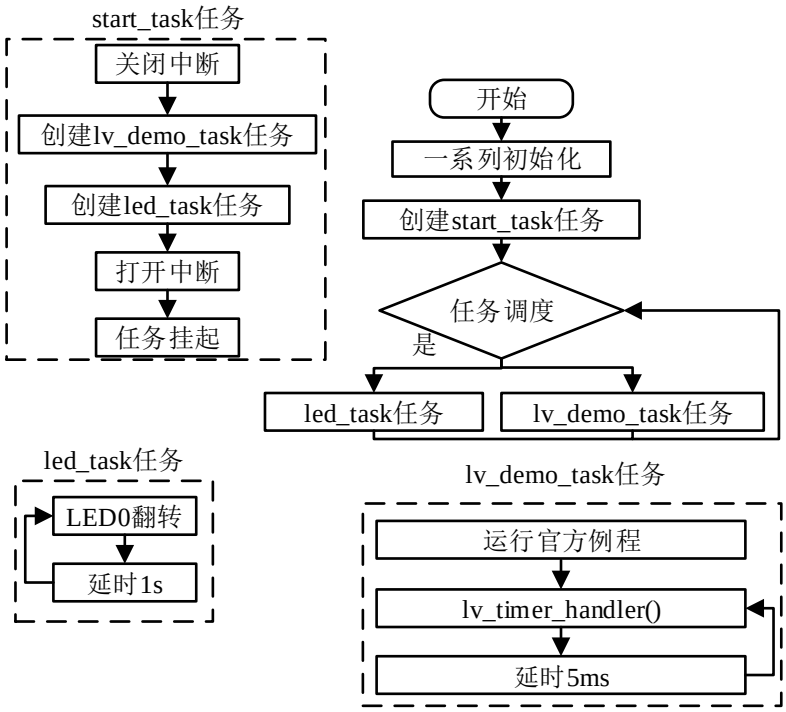


图 3.3.2.1.1 lvgl 操作系统流程图

3.3.2.2 程序解析

本小节着重讲解本实验相关的部分。如下源码所示：

1. 任务参数

```
/* START_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define START_TASK_PRIO    1                    /* 任务优先级 */
#define START_STK_SIZE     128                  /* 任务堆栈大小 */
TaskHandle_t StartTask_Handler;                 /* 任务句柄 */
void start_task(void *pvParameters);            /* 任务函数 */

/* LV_DEMO_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define LV_DEMO_TASK_PRIO     3                 /* 任务优先级 */
#define LV_DEMO_STK_SIZE      1024              /* 任务堆栈大小 */
TaskHandle_t LV_DEMOTask_Handler;               /* 任务句柄 */
void lv_demo_task(void *pvParameters);          /* 任务函数 */

/* LED_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define LED_TASK_PRIO    4                      /* 任务优先级 */
#define LED_STK_SIZE     128                    /* 任务堆栈大小 */
TaskHandle_t LEDTask_Handler;                   /* 任务句柄 */
void led_task(void *pvParameters);              /* 任务函数 */
```

注意：运行官方例程的任务的堆栈必须设置 512 大小以上。

2. 创建任务

```
/**
 * @brief       start_task
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           /* 进入临界区 */

    /* 创建 LVGL 任务 */
    xTaskCreate((TaskFunction_t     )lv_demo_task,
                (const char*        )"lv_demo_task",
                (uint16_t           )LV_DEMO_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )LV_DEMO_TASK_PRIO,
                (TaskHandle_t*      )&LV_DEMOTask_Handler);
```

```
    /* LED 测试任务 */
    xTaskCreate((TaskFunction_t     )led_task,
                (const char*        )"led_task",
                (uint16_t           )LED_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )LED_TASK_PRIO,
                (TaskHandle_t*      )&LEDTask_Handler);

    vTaskDelete(StartTask_Handler);     /* 删除开始任务 */
    taskEXIT_CRITICAL();                /* 退出临界区 */
}
```

从上述源码可知：start_task 任务函数主要创建了两个任务，分别为 lv_demo_task 和 led_task 任务。

3. 任务实现

```
/**
 * @brief       lv_demo_task
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void lv_demo_task(void *pvParameters)
{
    lv_demo_printer();  /* 测试的 demo */

    while(1)
    {
        lv_timer_handler();
        vTaskDelay(5);
    }
}

/**
 * @brief       led_task
 * @param       pvParameters : 传入参数(未用到)
 * @retval      无
 */
void led_task(void *pvParameters)
{
    while(1)
    {
        LED0_TOGGLE();
        vTaskDelay(1000);
    }
}
```

从上述源码可知：lv_demo_task 任务函数调用 lv_demo_music 函数执行官方例程，而它的 while(1)循环每 5ms 调用一次 lv_timer_handler 函数，该函数内部主要处理：绘制、获取输入等信息。

3.3.3 下载验证

工程编译并下载到开发板中。如图 3.3.3.1 所示：

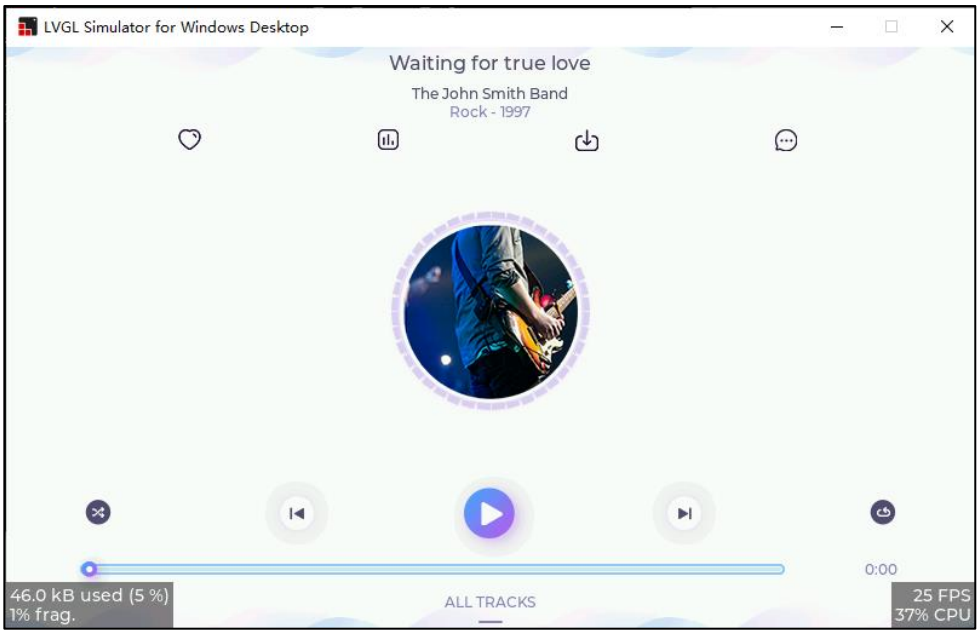


图 3.3.3.1 运行官方例程

第四章 PC 模拟器的使用

PC 模拟器是指那些可以在电脑上模拟其他平台的模拟器软件，所以在没有硬件的情况下，用户依然可以测试和调试 GUI 系统，另外 PC 模拟器对我们的前期界面迭代和后期的产品说明书制作都是可以起到非常大的作用。LVGL 作为一款优秀的 GUI 图形库，它肯定是支持模拟器功能的，LVGL 模拟器配套不同的 IDE(Integrated Development Environments)软件可运行在不同的操作系统上，比如 Windows 和 Linux 等。

本章主要分为如下几个部分讲解：

3.1 LVGL 模拟工程的下载

3.2 LVGL 例程模拟

3.3 工程文件的解析

3.1 LVGL 模拟工程的下载

这里我们打开 LVGL 官方文档并进入“Simulator on PC”页面，我们可以在这个页面下下载 LVGL 模拟工程，如下图所示：



图 3.1.1 模拟工程下载链接

这里我们下载 VisualStudio 模拟工程，点击上图中的“VisualStudio with SDL driver”链接进入下载页面，然后在这个页面下点击右上角 Code 复制“https://github.com/lvgl/lv_port_win_visual_studio.git”命令，这里笔者使用 Git 仓库克隆 GitHub 的工程，桌面新建“lvgl_simulator”文件夹（可选），如下图所示：



图 3.1.2 新建“lvgl_simulator”文件夹

我们在此文件夹右键打开“Git Bash Here”Git 命令行，在此命令行输入“git clone https://github.com/lvgl/lv_port_win_visual_studio.git” 克隆 LVGL 模拟工程的主仓库，如下图所示：

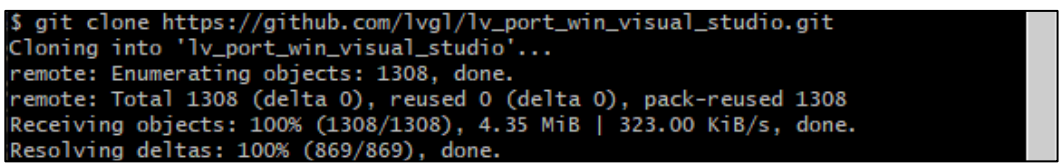


图 3.1.3 在 Github 远程仓库下载 LVGL 模拟工程主仓库

如果 Git 命令行显示“443: Timed out”失败，则下载超时，这个原因是 GitHub 服务器在国外的，所以在国内下载可能很慢，我们重复执行图 3.1.3 命令即可。下载完成之后我们在“lvgl_simulator”文件夹找到克隆的 LVGL 模拟器工程，如下图所示：

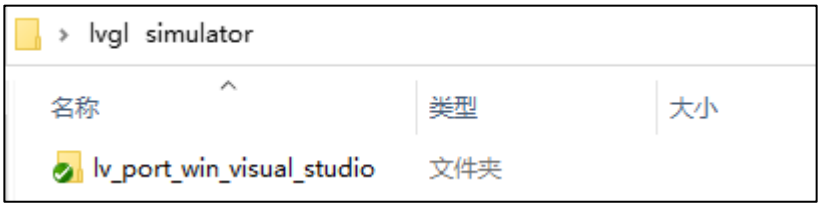


图 3.1.5 克隆 LVGL 模拟器主仓库完成

接下来就是同步子仓库，这里的子仓库是 lv_port_win_visual_studio\LVGL.Simulator 路径下的 freetype、lv_drivers 和 lvgl 等文件夹的内容，进入上图中的 lv_port_win_visual_studio 文件夹，在这个文件夹下右键打开“Git Bash Here”Git 命令行，在此命令行输入“git submodule update --init --recursive” 同步 LVGL 模拟工程的子仓库，如下图所示：

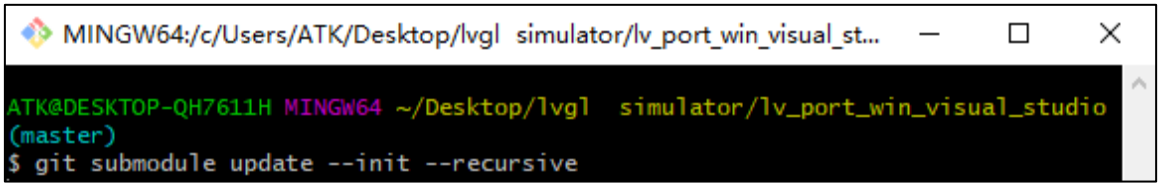


图 3.1.6 克隆 LVGL 模拟器子仓库

如果 LVGL 模拟器工程的子仓库同步失败，则重复执行上图的命令，当然我们也可以直接使用“git clone --recurse-submodules https://github.com/lvgl/lv_port_win_visual_studio.git”命令克隆整个工程包括主仓库和子仓库。至此，我们已经克隆远程仓库完成，接下来笔者带大家讲解 LVGL 模拟器工程的使用，如下步骤所示：

第一步：我们进入上图中的 lv_port_win_visual_studio 文件夹并点击 LVGL.Simulator.sln 文件启动 VS2019，如图 3.1.4 所示：

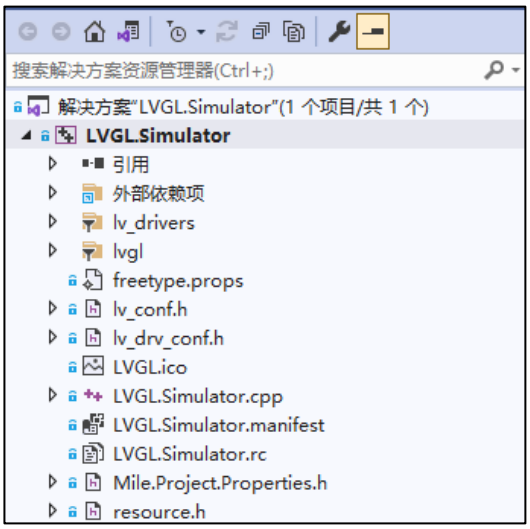


图 3.1.4 打开 LVGl 模拟项目

第二步：设置配置管理器，如图 3.1.5 所示：



图 3.1.5 配置管理器为 Debug x64

第三步：点击调试→开始调试(F5)，如图 3.1.6 所示：



图 3.1.6 开始调试

第四步：运行效果如下图 3.1.7 所示：

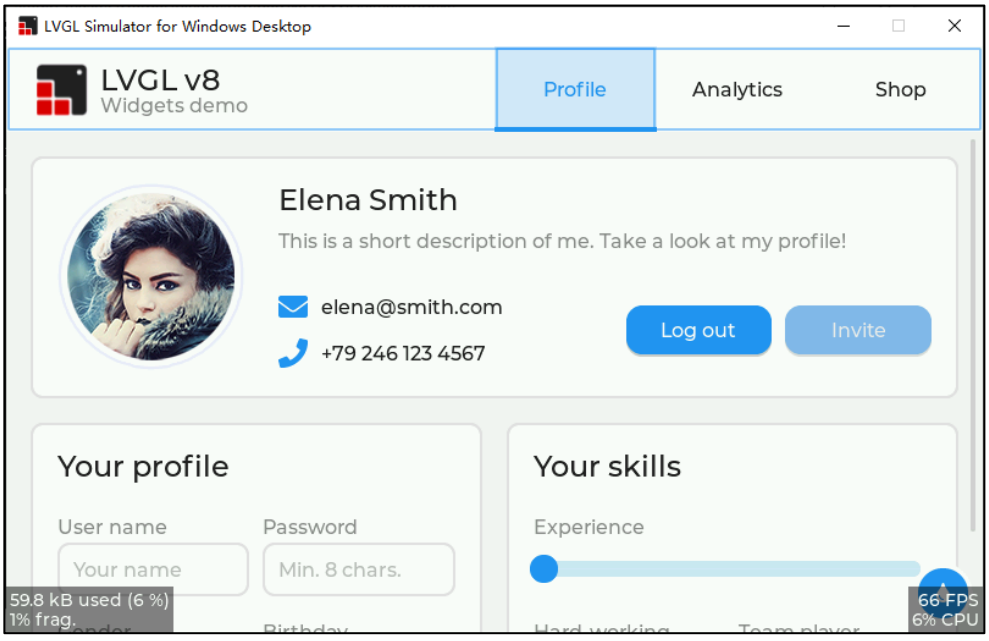


图 3.1.7 运行官方例程

3.2 LVGL 例程模拟

前面的章节中，笔者已经讲解过，LVGL 不仅仅提供综合例程还提供初学者学习的例程，这些学习的例程到底怎么在 PC 模拟器上运行的呢，首先我们在 LVGL 模拟工程打开 LVGL.Simulator.cpp 文件下找到 main 函数，如图 3.2.1 所示：

```
int main()
{
    // ---------------------------------
    // 综合例程
    // ---------------------------------

    lv_demo_widgets();           // ok
    // lv_demo_benchmark();
    // lv_demo_keypad_encoder();     // ok
    // lv_demo_music();              // removed from repository
    // lv_demo_printer();            // removed from repository
    // lv_demo_stress();             // ok

    // ---------------------------------
    // LVGL 例程
    // ---------------------------------

    // lv_ex_get_started_1();
    // lv_ex_get_started_2();
    // lv_ex_get_started_3();

    // lv_example_flex_1();
    // lv_example_flex_2();
    // lv_example_flex_3();
    // lv_example_flex_4();
    // lv_example_flex_5();
    // lv_example_flex_6();        // ok
```

图 3.2.1 main()函数定义

在图 3.1.7 中的实现效果是 lv_demo_widget 函数实现的，如果运行其他 LVGL 例程，先把函数 lv_demo_widget 注释，然后把运行的例程注销注释，最后开始调试即可。这里笔者重点强调一下模拟工程不能同时运行两个例程。模拟器的窗口大小请在 main 函数中设置，如下源码：

```
if (!lv_win32_init(
    GetModuleHandleW(NULL),
    SW_SHOW,
    800,
    480,
    LoadIconW(GetModuleHandleW(NULL), MAKEINTRESOURCE(IDI_LVGL))))
{
    return -1;
}
```

从上述源码可知：默认情况下，模拟器的窗口为 800x480 分辨率。

3.3 工程文件的解析

在 LVGL 的模拟项目中，最值得注意的文件是 lv_conf.h 文件，该文件负责裁剪以及设置等功能，这里我们分几个部分来讲解 lv_conf.h 文件如下所示：

①设置颜色深度：

```
/*颜色深度: 1 (1 byte per pixel), 8 (RGB332), 16 (RGB565), 32 (ARGB8888)*/
#define LV_COLOR_DEPTH     16
```

②设置内存大小

```
#define LV_MEM_SIZE (1024U * 1024U)          /*[字节]*/
```

③设置显示刷新周期和输入设备读取周期

```
/* 默认显示刷新周期。LVG 将在这段时间内重绘改变过的区域 */
#define LV_DISP_DEF_REFR_PERIOD 10      /*[ms]*/


/* 输入设备读取周期(毫秒) */
#define LV_INDEV_DEF_READ_PERIOD 10     /*[ms]*/
```

④设置日志功能

```
/* 启用日志模块进入翻译页面 */
#define LV_USE_LOG 1
```

⑤字库使用

```
/* 字库使用，设置 1 启用，设置 0 为失能 */
#define LV_FONT_MONTSERRAT_8  1
#define LV_FONT_MONTSERRAT_10 1
#define LV_FONT_MONTSERRAT_12 1
#define LV_FONT_MONTSERRAT_14 1
#define LV_FONT_MONTSERRAT_16 1
#define LV_FONT_MONTSERRAT_18 1
#define LV_FONT_MONTSERRAT_20 1
#define LV_FONT_MONTSERRAT_22 1
#define LV_FONT_MONTSERRAT_24 1
#define LV_FONT_MONTSERRAT_26 1
#define LV_FONT_MONTSERRAT_28 1
#define LV_FONT_MONTSERRAT_30 1
#define LV_FONT_MONTSERRAT_32 1
#define LV_FONT_MONTSERRAT_34 1
#define LV_FONT_MONTSERRAT_36 1
#define LV_FONT_MONTSERRAT_38 1
#define LV_FONT_MONTSERRAT_40 1
#define LV_FONT_MONTSERRAT_42 1
#define LV_FONT_MONTSERRAT_44 1
#define LV_FONT_MONTSERRAT_46 1
#define LV_FONT_MONTSERRAT_48 1
```

⑥设置字符串字符编码

```
/**
 * 为字符串选择字符编码，你的 IDE 或编辑器应该有相同的字符编码
 * - LV_TXT_ENC_UTF8
 * - LV_TXT_ENC_ASCII
 */
#define LV_TXT_ENC LV_TXT_ENC_UTF8
```

如果显示中文字符，宏定义 LV_TXT_ENC 必须选择 LV_TXT_ENC_UTF8。

⑦文件系统选择

```
#define LV_USE_FS_STDIO '\0'          /*Uses fopen, fread, etc*/
```

```
#define LV_USE_FS_POSIX '\0'           /*Uses open, read, etc*/

#define LV_USE_FS_WIN32 '/'            /*Uses CreateFile, ReadFile, etc*/

#define LV_USE_FS_FATFS '\0'           /*Uses f_open, f_read, etc*/
```

根据不同的接口选择文件系统，如果使用 SD 卡，就把 LV_USE_FS_FATFS 宏定义设置为'0'。

⑧选择第三方库(PNG\BMP\JPEG\GIF\QR)

```
/*PNG decoder library*/
#define LV_USE_PNG 0

/*BMP decoder library*/
#define LV_USE_BMP 0

/* JPG + split JPG decoder library.
 * Split JPG is a custom format optimized for embedded systems. */
#define LV_USE_SJPG 0

/*GIF decoder library*/
#define LV_USE_GIF 0

/*QR code library*/
#define LV_USE_QRCODE 0
```

上述源码可知：这些配置项表示使能/失能第三方解码库。

⑨选择演示实例

```
/*Show some widget*/
#define LV_USE_DEMO_WIDGETS        1
#if LV_USE_DEMO_WIDGETS
#define LV_DEMO_WIDGETS_SLIDESHOW  1
#endif

/*Demonstrate the usage of encoder and keyboard*/
#define LV_USE_DEMO_KEYPAD_AND_ENCODER     1

/*Benchmark your system*/
#define LV_USE_DEMO_BENCHMARK   1

/*Stress test for LVGL*/
#define LV_USE_DEMO_STRESS      1

/*Music player demo*/
#define LV_USE_DEMO_MUSIC       1
#if LV_USE_DEMO_MUSIC
```

```
# define LV_DEMO_MUSIC_SQUARE       1
# define LV_DEMO_MUSIC_LANDSCAPE    1
# define LV_DEMO_MUSIC_ROUND        1
# define LV_DEMO_MUSIC_LARGE        1
# define LV_DEMO_MUSIC_AUTO_PLAY    1
#endif
```

第五章 LVGL 初探

在前面的章节中，笔者相信读者已经对 LVGL 有一定的了解，本章主要讲解 LVGL 移植相关的知识。

本章分为以下几个部分讲解：

5.1 系统总览
5.2 lv_conf.h 文件配置解析
5.3 LVGL 初始化
5.4 显示设备
5.5 输入设备
5.6 LVGL 滴答
5.7 操作系统与中断
5.8 LVGL 日志

5.1 系统总览

在学习 LVGL 基础知识之前，我们首先了解 MCU 与 LVGL 之间是如何交互的，简单来说：它们的控制流程是如何实现的？根据官方提供的资料以及结合笔者的理解，我们可得到下图 5.1.1 的示意图：

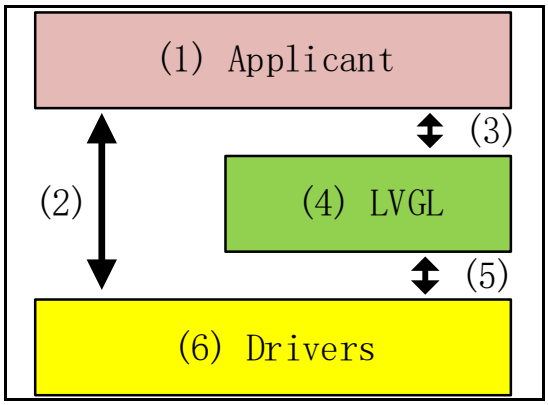


图 5.1.1 LVGL 与 MCU 交互流程

从图 5.1.1 可知，LVGL 与 MCU 交互流程一共分为六个部分，这六个部分不仅仅符合 LVGL 控制流程还符合其他的 GUI 控制流程，下面笔者分别地介绍这六个部分的作用，如下所示：

(1) **应用程序：**创建 GUI 并处理特定任务的应用程序，注意：这里不单单只包含 LVGL 的程序，因为应用程序可以包含其他的程序，例如一个任务处理摄像头数据，另一个任务处理 GUI。

(2) 应用程序可以直接驱动设备，例如控制开发板上的 LED 灯闪烁。

(3) 在应用程序调用 LVGL 函数时，系统不会直接执行(2)来驱动设备，而是经过 LVGL 图形库间接处理，比如界面的按键控制开发板上的 LED 亮灭。

(4) **LVGL 图形库：**应用程序可以与该库进行交互并制作相应的 UI 界面，例如在显示屏显示按键、菜单栏等部件。

(5) 经过 LVGL 图形库处理以后进行驱动设备，该库包含 HAL(硬件抽象层)的接口，用于注册显示和输入设备驱动程序等来控制设备。

(6) **驱动设备：**除特定的驱动程序外，它还包含一些功能，可驱动显示器(可选)到 GPU 并读取触摸板或按钮。根据 MCU，有两种典型的硬件设置。一个带有内置 LCD/TFT 驱动器外围设备，另一个不带外围设备。在这两种情况下，都需要一个帧缓冲区来存储屏幕的当前图像。

① 带有 TFT/LCD 驱动器 的 MCU ，如果你的 MCU 具有 TFT / LCD 驱动器外围设备，则可以直接通过 RGB 接口连接显示器。在这种情况下，帧缓冲区可以位于内部 RAM 中（如果 MCU 有足够的 RAM），也可以位于外部 RAM 中(如果 MCU 具有存储器接口)。

② 外部显示控制器 如果 MCU 不具有 TFT/LCD 驱动程序接口，则必须使用外部显示控制器(例如 SSD1963，SSD1306，ILI9341)。在这种情况下，MCU 可以通过并行端口，SPI 或有时通过 I2C 与显示控制器进行通信。帧缓冲区通常位于显示控制器中，从而为 MCU 节省了大量 RAM。

5.2 lv_conf.h 文件配置解析

首先，我们需要明确一个问题，lv_conf.h 是一个用户级别的文件，不属于内核的文件。每个用户可以有不同的 lv_conf.h。 lv_conf.h 文件具有两大功能：

(1) 具有配置功能：内存配置、屏幕刷新周期、输入设备的读取周期等相关配置

(2) 具有裁剪功能：使能/失能某些功能，有效减少 Flash 占用率。

lv_conf.h 文件划分为几个板块，如表 5.2.1 所示：

板块介绍	描述
颜色设置	颜色深度、屏幕透明等设置
内存设置	内存存储器(内/外)、分配内存大小
HAL(硬件抽象层)设置	刷新周期、输入设备读取周期以及 tick 时钟来源
特征设置	绘画、加速刷新、日志、断言以及帧率计算功能
编译器设置	关于编译器设置
字体设置	选择系统字体以及声明自定义字体
文本设置	字符编码、文本特性
部件设置	使能/失能核心部件
特别功能	额外的部件、主题、布局、第三方库等设置
实例设置	LVGL 演示实例设置

表 5.2.1 lv_conf.h 文件内容描述

接下来，笔者重点讲解 lv_conf.h 文件几个重要板块的信息，如以下所示：

1. 颜色设置

```
/*
*颜色设置
*/

/* 颜色深度: 1(每像素 1 字节), 8(RGB332), 16(RGB565), 32(ARGB8888) */
#define LV_COLOR_DEPTH                      16

/* 交换 2 字节的 RGB565 颜色。如果显示有 8 位接口(例如 SPI) */
#define LV_COLOR_16_SWAP                    0
```

```
/* 1: 启用屏幕透明.
 * 对 OSD 或其他有重叠的 gui 很有用.
 * 要求' LV_COLOR_DEPTH = 32 '颜色和屏幕的样式应该被修改: `style.body.opa = ...`*/
#define LV_COLOR_SCREEN_TRANSP              0

/* 调整颜色混合功能四舍五入。gpu 可能会以不同的方式计算颜色混合。
 * 0:取整，64:从 x.75 取整，128:从 half 取整，192:从 x.25 取整，254:从 half 取整 */
#define LV_COLOR_MIX_ROUND_OFS              (LV_COLOR_DEPTH == 32 ? 0: 128)

/* 如果使用色度键，将不会绘制这种颜色的图像像素） */
#define LV_COLOR_CHROMA_KEY                 lv_color_hex(0x00ff00)    /* 纯绿 */
```

从上述源码可知：我们设置 LV_COLOR_DEPTH 宏定义为 16 位颜色深度，因为正点原子的显示屏颜色深度为 16 位，其他宏定义我们选择默认即可。

2. 内存设置

```
/*
*内存设置
*/

/* 0: 使用内置的 `lv_mem_alloc()` 和 `lv_mem_free()`*/
#define LV_MEM_CUSTOM                       0
#if LV_MEM_CUSTOM == 0
    /* `lv_mem_alloc()`可获得的内存大小(以字节为单位)(>= 2kB) */
    #define LV_MEM_SIZE                     (100U * 1024U)/*[字节]*/

    /* 为内存池设置一个地址，而不是将其作为普通数组分配。也可以在外部 SRAM 中。 */
    #define LV_MEM_ADR                      0              /*0: 未使用*/
    /* 给内存分配器而不是地址，它将被调用来获得 LVGL 的内存池。例如 my_malloc */
    #if LV_MEM_ADR == 0
        //#define LV_MEM_POOL_INCLUDE your_alloc_library
        //#define LV_MEM_POOL_ALLOC   your_alloc
    #endif

#else       /*LV_MEM_CUSTOM*/
    #define LV_MEM_CUSTOM_INCLUDE <stdlib.h>   /* 动态内存函数的头 */
    #define LV_MEM_CUSTOM_ALLOC   malloc
    #define LV_MEM_CUSTOM_FREE    free
    #define LV_MEM_CUSTOM_REALLOC realloc
#endif     /*LV_MEM_CUSTOM*/

/* 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。
 * 如果没有足够的缓冲区，你会看到一个错误日志信息. */
#define LV_MEM_BUF_MAX_NUM                  16
```

```
/* 使用标准的 `memcpy` 和 `memset` 代替 LVGL 自己的函数。(可能更快，也可能不会更快) */
#define LV_MEMCPY_MEMSET_STD
```

从上述源码可知：如果 LV_MEM_CUSTOM 宏定义为 1，则不使用 LVGL 内存管理算法，用户自行添加内存管理算法。 LV_MEM_SIZE 宏定义表示申请内存大小，该宏定义受 LV_MEM_ADR 宏定义制约，如果 LV_MEM_ADR 设置为外部存储器首地址，那么内存大小是由外部存储器分配，否则在内部 SRAM 获取内存。

当然我们也可以把 LV_MEM_ADR 宏定义设置为 0 而选择外部存储器分配内存方式，然后把 LV_MEM_POOL_INCLUDE 和 LV_MEM_POOL_ALLOC 宏定义注销注释，最后自行添加你的申请内存函数以及头文件路径。

下面我们来讲解一下系统提供 LVGL 内存的方式有哪些，首先我们可以打开 lv_mem.c 文件找到 lv_mem_init 函数，如下源码所示：

```
void lv_mem_init(void)
{
#if LV_MEM_CUSTOM == 0

#if LV_MEM_ADR == 0
#ifdef LV_MEM_POOL_ALLOC
    /* 第一种方式：自己的内存管理申请内存 */
    tlsf = lv_tlsf_create_with_pool((void *)LV_MEM_POOL_ALLOC(LV_MEM_SIZE),
                                    LV_MEM_SIZE);
#else
    /* 第二种方式：分配一个大数组来存储动态分配的数据 */
static LV_ATTRIBUTE_LARGE_RAM_ARRAY MEM_UNIT work_mem_int[LV_MEM_SIZE /
                                                          sizeof(MEM_UNIT)];
    tlsf = lv_tlsf_create_with_pool((void *)work_mem_int, LV_MEM_SIZE);
#endif
#else
    /* 第三种方式：根据外部存储器地址分配内存 */
    tlsf = lv_tlsf_create_with_pool((void *)LV_MEM_ADR, LV_MEM_SIZE);
#endif
#endif
}
```

从上述源码可知：第二种方式是内部 SRAM 分配内存给 LVGL，而其他两种是由外部存储器分配内存。接下来我们来讲解这三种内存分配方式到底如何实现的。

(1) 第一种分配内存方式，在 lv_conf.h 文件中把 LV_MEM_CUSTOM 和 LV_MEM_ADR 配置项设置为 0，然后把 LV_MEM_POOL_INCLUDE 和 LV_MEM_POOL_ALLOC 配置项注销注释，如下源码所示：

```
/* 0: 使用内置的 `lv_mem_alloc()` 和 `lv_mem_free()`*/
#define LV_MEM_CUSTOM                       0
#if LV_MEM_CUSTOM == 0
    /* `lv_mem_alloc()`可获得的内存大小(以字节为单位)(>= 2kB) */
    #define LV_MEM_SIZE                     (50U * 1024U) /*[字节]*/
```

```
    /* 为内存池设置一个地址，而不是将其作为普通数组分配。也可以在外部 SRAM 中。 */
    #define LV_MEM_ADR                          0             /*0: 未使用*/
    /* 给内存分配器而不是地址，它将被调用来获得 LVGL 的内存池。例如 my_malloc */
    #if LV_MEM_ADR == 0
        #define LV_MEM_POOL_INCLUDE  "./MALLOC/malloc.h"
        #define LV_MEM_POOL_ALLOC    lv_mymalloc
    #endif

#else       /*LV_MEM_CUSTOM*/
    #define LV_MEM_CUSTOM_INCLUDE <stdlib.h>   /* 动态内存函数的头 */
    #define LV_MEM_CUSTOM_ALLOC   malloc
    #define LV_MEM_CUSTOM_FREE    free
    #define LV_MEM_CUSTOM_REALLOC realloc
#endif     /*LV_MEM_CUSTOM*/
```

此方式的分配内存是由LV_MEM_POOL_ALLOC配置项指向的lv_mymalloc函数分配的，LV_MEM_POOL_INCLUDE 配置项指向 lv_mymalloc 函数的声明路径，这里笔者在 malloc.c 文件中定义 lv_mymalloc 函数，所以该配置项指向"./MALLOC/malloc.h"声明路径。lv_mymalloc 函数编写方式如下所示：

```
/**
 * @brief       分配内存(外部调用)
 * @param       size : 要分配的内存大小(字节)
 * @retval      分配到的内存首地址.
 */
void * lv_mymalloc(uint32_t size)
{
    return  (void *)mymalloc(SRAMEX,size);/* 选择外部存储器，返回分配到的内存首地址 */
}
```

注意：lv_mymalloc 函数必须在 malloc.h 文件下声明，而该函数的申请内存大小由 LV_MEM_SIZE 配置项决定。

(2) 第二种分配内存方式，在 lv_config.h 文件中把 LV_MEM_CUSTOM、LV_MEM_ADR 和 LV_MEM_ADR 配置项设置为 0，然后不注销 LV_MEM_POOL_INCLUDE 和 LV_MEM_POOL_ALLOC 配置项注释，如下源码所示：

```
/* 0: 使用内置的 `lv_mem_alloc()` 和 `lv_mem_free()`*/
#define LV_MEM_CUSTOM                       0
#if LV_MEM_CUSTOM == 0
    /* `lv_mem_alloc()`可获得的内存大小(以字节为单位)(>= 2kB) */
    #define LV_MEM_SIZE                     (50U * 1024U)          /*[字节]*/
    /* 为内存池设置一个地址，而不是将其作为普通数组分配。也可以在外部 SRAM 中。 */
    #define LV_MEM_ADR                      0     /*0: 未使用*/
    /* 给内存分配器而不是地址，它将被调用来获得 LVGL 的内存池。例如 my_malloc */
    #if LV_MEM_ADR == 0
        //#define LV_MEM_POOL_INCLUDE your_alloc_library
        //#define LV_MEM_POOL_ALLOC   your_alloc
```

```
    #endif

#else       /*LV_MEM_CUSTOM*/
    #define LV_MEM_CUSTOM_INCLUDE <stdlib.h>   /* 动态内存函数的头 */
    #define LV_MEM_CUSTOM_ALLOC   malloc
    #define LV_MEM_CUSTOM_FREE    free
    #define LV_MEM_CUSTOM_REALLOC realloc
#endif     /*LV_MEM_CUSTOM*/
```

上述源码可知：使用此方式分配的内存是由一个大数组来存储动态分配的数据，申请内存大小由 LV_MEM_SIZE 配置项决定。

(3) 第三种分配内存方式，在 lv_config.h 文件中把 LV_MEM_CUSTOM 和 LV_MEM_ADR 配置项设置为 0，然后不注销 LV_MEM_POOL_INCLUDE 和 LV_MEM_POOL_ALLOC 配置项注释，最后设置 LV_MEM_ADR 设置为内存地址，如下源码所示：

```
/* 0: 使用内置的 `lv_mem_alloc()` 和 `lv_mem_free()`*/
#define LV_MEM_CUSTOM                       0
#if LV_MEM_CUSTOM == 0
    /* `lv_mem_alloc()`可获得的内存大小(以字节为单位)(>= 2kB) */
    #define LV_MEM_SIZE                     (50U * 1024U) /*[字节]*/

    /* 为内存池设置一个地址，而不是将其作为普通数组分配。也可以在外部 SRAM 中。 */
    #define LV_MEM_ADR                      0x68000000     /*0: 未使用*/
    /* 给内存分配器而不是地址，它将被调用来获得 LVGL 的内存池。例如 my_malloc */
    #if LV_MEM_ADR == 0
        //#define LV_MEM_POOL_INCLUDE your_alloc_library
        //#define LV_MEM_POOL_ALLOC   your_alloc
    #endif

#else       /*LV_MEM_CUSTOM*/
    #define LV_MEM_CUSTOM_INCLUDE <stdlib.h>   /* 动态内存函数的头 */
    #define LV_MEM_CUSTOM_ALLOC   malloc
    #define LV_MEM_CUSTOM_FREE    free
    #define LV_MEM_CUSTOM_REALLOC realloc
#endif     /*LV_MEM_CUSTOM*/
```

上述源码可知，我们设置宏定义 LV_MEM_ADR 为外部存储器的首地址，显然分配的内存是以外部存储器首地址开始，而配置项 LV_MEM_SIZE 设置分配内存的大小。

前面笔者也讲解过，如果设置 LV_MEM_CUSTOM 宏定义为 1，则系统不再使用 LVGL 内部的内存管理算法，而是选择自己的内存管理算法，如下源码所示：

```
/* 0: 使用内置的 `lv_mem_alloc()` 和 `lv_mem_free()`*/
#define LV_MEM_CUSTOM                       1
#if LV_MEM_CUSTOM == 0
    /* `lv_mem_alloc()`可获得的内存大小(以字节为单位)(>= 2kB) */
    #define LV_MEM_SIZE                     (100U * 1024U)     /*[字节]*/
```

```
    /* 为内存池设置一个地址，而不是将其作为普通数组分配。也可以在外部 SRAM 中。 */
    #define LV_MEM_ADR                          0                   /*0: 未使用*/
    /* 给内存分配器而不是地址，它将被调用来获得 LVGL 的内存池。例如 my_malloc */
    #if LV_MEM_ADR == 0
        //#define LV_MEM_POOL_INCLUDE your_alloc_library
        //#define LV_MEM_POOL_ALLOC   your_alloc
    #endif

#else       /*LV_MEM_CUSTOM*/
    #define LV_MEM_CUSTOM_INCLUDE "./MALLOC/malloc.h"    /* 动态内存函数的头 */
    #define LV_MEM_CUSTOM_ALLOC   lv_mymalloc
    #define LV_MEM_CUSTOM_FREE    lv_myfree
    #define LV_MEM_CUSTOM_REALLOC lv_myrealloc
#endif     /*LV_MEM_CUSTOM*/
```

上述源码可知：设置 LV_MEM_CUSTOM 宏定义为 1，不使用 LVGL 内部的内存管理算法，所以下表的宏定义指向的函数必须由用户定义，如表 5.2.2 所示：

宏定义	描述
LV_MEM_CUSTOM_INCLUDE	动态内存函数的头路径
LV_MEM_CUSTOM_ALLOC	申请内存函数
LV_MEM_CUSTOM_FREE	释放内存函数
LV_MEM_CUSTOM_REALLO	重新分配内存函数

表 5.2.2 设置内存管理算法函数

这里我们以正点原子的内存管理算法为例，大家可以使用其他的内存管理算法。打开正点原子 malloc.c 文件，添加以下源码：

```
/**
 * @brief       分配内存(外部调用)
 * @param       size : 要分配的内存大小(字节)
 * @retval      分配到的内存首地址.
 */
void * lv_mymalloc(uint32_t size)
{
    return (void *)mymalloc(SRAMIN,size);
}

/**
 * @brief       释放内存(外部调用)
 * @param       ptr  : 内存首地址
 * @retval      无
 */
void lv_myfree(void *ptr)
{
    myfree(SRAMIN,ptr);
}
```

```
/**
 * @brief       重新分配内存(外部调用)
 * @param       memx : 所属内存块
 * @param       *ptr : 旧内存首地址
 * @param       size : 要分配的内存大小(字节)
 * @retval      新分配到的内存首地址.
 */
void * lv_myrealloc(void *ptr, uint32_t size)
{
    return (void *)myrealloc(SRAMIN,ptr,size);
}
```

上述函数是以正点原子 malloc.c 文件的 mymalloc、myfree 和 myrealloc 函数基础上修改的，最后在 malloc.h 文件声明这些函数。

总结：

(1) LV_MEM_CUSTOM 宏定义为 0 时，提供 LVGL 内存具有三种方式，一种是内部 RAM 分配内存，而其他两种都是使用外部存储器分配，注意两者的区别。

(2) LV_MEM_CUSTOM 宏定义为 1 时，不选择 LVGL 内部的内存管理算法，可选择其他的内存管理算法，例如 FreeRTOS、lwIP 等的内存管理算法。

3. HAL(硬件抽象层)设置

```
/*
*HAL 设置
*/

/* 默认的显示刷新周期。LVGL 使用这个周期重绘修改过的区域 */
#define LV_DISP_DEF_REFR_PERIOD             4       /*[ms]*/

/* 输入设备的读取周期(以毫秒为单位) */
#define LV_INDEV_DEF_READ_PERIOD            4      /*[ms]*/

/* 使用自定义 tick 源，以毫秒为单位告诉运行时间。它不需要手动更新 `lv_tick_inc()` */
#define LV_TICK_CUSTOM                      1
#if LV_TICK_CUSTOM
#define LV_TICK_CUSTOM_INCLUDE          "FreeRTOS.h"       /* 系统时间函数头 */
/* 计算系统当前时间的表达式(以毫秒为单位) */
    #define LV_TICK_CUSTOM_SYS_TIME_EXPR    (xTaskGetTickCount())
#endif    /*LV_TICK_CUSTOM*/

/* 默认每英寸的点数量。用于初始化默认大小，例如小部件大小，样式填充。
 * (不是很重要，你可以调整它来修改默认大小和空格) */
#define LV_DPI_DEF                          130      /*[px/inch]*/
```

上述源码可知：LV_DISP_DEF_REFR_PERIOD 和 LV_INDEV_DEF_READ_PERIOD 宏定义表示显示刷新周期和输入设备读取周期(鼠标、触摸等输入设备)，而 LV_TICK_CUSTOM 宏

定义为 1 时，可使用 RTOS 提供 LVGL 时钟节拍，注意：FreeRTOSConfig.h 文件中的宏定义 configTICK_RATE_HZ 必须设置为 1000(1ms 时基)。

4. 特征设置

```
/*-------------
 * 3. 日志
 *-----------*/

/* 启用日志模块 */
#define LV_USE_LOG                          0
#if LV_USE_LOG

    /*应该添加多重要的日志:
    *LV_LOG_LEVEL_TRACE        大量的日志给出了详细的信息
    *LV_LOG_LEVEL_INFO         记录重要事件
    *LV_LOG_LEVEL_WARN         如果发生了一些不想要的事情但没有引起问题，则记录下来
    *LV_LOG_LEVEL_ERROR        只有在系统可能出现故障时才会出现关键问题
    *LV_LOG_LEVEL_USER         仅用户自己添加的日志
    *LV_LOG_LEVEL_NONE         不要记录任何内容*/
    #define LV_LOG_LEVEL LV_LOG_LEVEL_WARN

    /*1: 使用'printf'打印日志;
     *0: 用户需要用' lv_log_register_print_cb() '注册回调函数 */
    #define LV_LOG_PRINTF                   0

    /* 在产生大量日志的模块中启用/禁用 LV_LOG_TRACE */
    #define LV_LOG_TRACE_MEM                1
    #define LV_LOG_TRACE_TIMER              1
    #define LV_LOG_TRACE_INDEV              1
    #define LV_LOG_TRACE_DISP_REFR          1
    #define LV_LOG_TRACE_EVENT              1
    #define LV_LOG_TRACE_OBJ_CREATE         1
    #define LV_LOG_TRACE_LAYOUT             1
    #define LV_LOG_TRACE_ANIM               1

#endif  /*LV_USE_LOG*/
```

开启日志模块，主要用来调试程序是否执行正确。开启日志时，请把 LV_USE_LOG 宏定义设置为 1，然后设置 LV_LOG_LEVEL_WARN 的日志类型，最后设置 LV_LOG_PRINTF 宏定义为 1 表示打印日志信息。

5. 字体设置

```
/* 可以在这里声明您的自定义字体。
   你也可以使用这些字体作为默认字体
   它们将在全局范围内提供。如
 * #define LV_FONT_CUSTOM_DECLARE LV_FONT_DECLARE(my_font_1) \
```

```
 *                                LV_FONT_DECLARE(my_font_2)
 */
#define LV_FONT_CUSTOM_DECLARE
```

该宏定义主要声明自定义的字体。

6. 部件设置

```
/*-------------
 * 部件设置
 *-----------*/

#define LV_USE_ARC                      1

#define LV_USE_ANIMIMG                  1

#define LV_USE_BAR                      1

#define LV_USE_BTN                      1

#define LV_USE_BTNMATRIX                1

#define LV_USE_CANVAS                   1

#define LV_USE_CHECKBOX                 1

#define LV_USE_DROPDOWN                 1   /* 依赖: lv_label */

#define LV_USE_IMG                      1   /* 依赖: lv_label */

#define LV_USE_LABEL                    1
#if LV_USE_LABEL
#define LV_LABEL_TEXT_SELECTION         1  /* 启用标签的选择文本*/
/* 在标签中存储一些额外的信息，以加快绘制非常长的文本 */
    #define LV_LABEL_LONG_TXT_HINT      1
#endif

#define LV_USE_LINE                     1

#define LV_USE_ROLLER                   1  /* 依赖: lv_label */
#if LV_USE_ROLLER
    #define LV_ROLLER_INF_PAGES         7  /* 当滚筒无限时，额外的“页数” */
#endif

#define LV_USE_SLIDER                   1  /* 依赖: lv_bar*/
```

```
#define LV_USE_SWITCH                           1

#define LV_USE_TEXTAREA                         1   /* 依赖: lv_label*/
#if LV_USE_TEXTAREA != 0
    #define LV_TEXTAREA_DEF_PWD_SHOW_TIME    1500    /*ms*/
#endif

#define LV_USE_TABLE                            1
```

上述源码表示部件的使能与失能，如果设置为 1 时，使能该部件，否则失能该部件，这些配置项有效的解决 Flash/ROM 占用率的问题。

5.3 LVGL 初始化流程

官方文档中已经说明了 LVGL 初始化流程这一部分，如以下步骤所示：

第一步：首先调用 lv_init 初始化 LVGL 图形库。

第二步：lv_port_disp_init 注册显示驱动程序和 lv_port_indev_init 输入设备驱动程序。注意：注册显示驱动程序和输入设备驱动程序必须在函数 lv_init()初始化之后。

第三步：lv_tick_inc(x) 在 x 中断中每毫秒调用一次(例如：正点原子使用定时器定时刷新)，当然可以不使用定时器提供 LVGL 节拍，可使用 OS 操作系统的时钟函数来提供 LVGL 节拍，打开 lv_conf.h 文件，找到 LV_TICK_CUSTOM 宏定义设置为 1，这里笔者以 FreeRTOS 操作系统为例，如下源码所示：

```
/* 使用自定义 tick 源，以毫秒为单位告诉运行时间。它不需要手动更新 `lv_tick_inc()` */
#define LV_TICK_CUSTOM                  1
#if LV_TICK_CUSTOM
#define LV_TICK_CUSTOM_INCLUDE          "FreeRTOS.h"    /* 系统时间函数头 */
/* 计算系统当前时间的表达式(以毫秒为单位) */
#define LV_TICK_CUSTOM_SYS_TIME_EXPR    (xTaskGetTickCount())
#endif    /*LV_TICK_CUSTOM*/
```

第四步：lv_timer_handler 每隔几毫秒周期调用一次，处理与 LVGL 相关的任务(可以放在 while(1)循环中，但确保 5 毫秒以内 LVGL 系统得到响应)。

5.4 显示接口

在绘制 UI 之前，LVGL 必须要注册一个显示驱动，该显示驱动主要是把像素阵列复制到显示器的给定区域，显然最终调用了 LCD 驱动函数绘画图形，注册一个 LVGL 显示接口必须要初始化 lv_disp_draw_buf_t 和一个变量 lv_disp_drv_t，如下源码所示：

```
lv_disp_draw_buf_t      /*绘制缓冲区*/
lv_disp_drv_t           /* 包含回调函数以与显示交互并操作低级绘图行为 */
```

显示接口相关知识，我们到以后的章节会讲解，这里我们了解即可。

5.4.1 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 显示驱动程序中的设置函数发送到显示器，绘制缓冲区(lv_disp_draw_buf_t)初始化有三种方式，如下源码所示：

```
#define MY_DISP_HOR_RES (800)    /* 屏幕宽度 */
```

```
#define MY_DISP_VER_RES (480)    /* 屏幕高度 */

/* 第一种：单缓冲区示例 */
static lv_disp_draw_buf_t draw_buf_dsc_1;
/* 设置缓冲区的大小为 10 行屏幕的大小 */
static lv_color_t buf_1[MY_DISP_HOR_RES * 10];
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&draw_buf_dsc_1, buf_1, NULL, MY_DISP_HOR_RES * 10);

/* 第二种：双缓冲区示例 */
static lv_disp_draw_buf_t draw_buf_dsc_2;
/* 设置缓冲区的大小为 10 行屏幕的大小 */
static lv_color_t buf_2_1[MY_DISP_HOR_RES * 10];
/* 设置另一个缓冲区的大小为 10 行屏幕的大小 */
static lv_color_t buf_2_2[MY_DISP_HOR_RES * 10];
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&draw_buf_dsc_2, buf_2_1, buf_2_2, MY_DISP_HOR_RES * 10);

/* 第三种：全尺寸双缓冲区示例 并且在下面设置 disp_drv.full_refresh = 1 */
static lv_disp_draw_buf_t draw_buf_dsc_3;
/* 设置一个全尺寸的缓冲区 */
static lv_color_t buf_3_1[MY_DISP_HOR_RES * MY_DISP_VER_RES];
/* 设置另一个全尺寸的缓冲区 */
static lv_color_t buf_3_2[MY_DISP_HOR_RES * MY_DISP_VER_RES];
/* 初始化显示缓冲区 */
lv_disp_draw_buf_init(&draw_buf_dsc_3, buf_3_1, buf_3_2,
MY_DISP_HOR_RES * MY_DISP_VER_RES);
```

单缓冲区： LVGL 将屏幕的内容绘制到缓冲区当中，并将内容发送到显示器，缓冲区可以小于屏幕。在这种情况下，较大的区域将被重绘成多个部分。如果只有很小的区域发生变化（例如，按下按钮），则只有那些区域会被刷新。

双缓冲区：如果使用两个缓冲区，LVGL 可以在一个缓冲区中绘制，而另一个缓冲区的内容在后台发送到显示器。应使用 DMA 或其他硬件将数据传输到显示器，以便 MCU 可以继续绘图。这样，显示的渲染和刷新就变成了并行操作。说白了：就是显示屏刷新快，就像显示渲染和刷新时同步进行的。

全尺寸双缓冲区：在显示驱动程序 (lv_disp_drv_t) 中启用该 full_refresh 位将强制 LVGL 始终重绘整个屏幕。这适用于一个缓冲区和两个缓冲区模式。如果 full_refresh 启用并提供两个屏幕大小的绘制缓冲区，LVGL 的显示处理就像“传统”双缓冲一样工作。这意味着 flush_cb 回调只需要更新帧缓冲区（color_p 参数）的地址。如果 MCU 具有 LCD 控制器外设，而不是通过串行链路访问的外部显示控制器（例如 ILI9341 或 SSD1963），则应使用此配置。后者通常太慢而无法通过全屏重绘来保持高帧速率。注意：官方指出：使用全尺寸双缓冲区方式必须结合 DMA2D 来使用，如果不使用 DMA2D，那么刷新的速度提高不大。

1. 显示驱动程序编写：

缓冲区初始化完成后，lv_disp_drv_t 显示驱动程序编写步骤如下：

第一步：lv_disp_drv_init(&disp_drv) 初始化显示设备。

第二步：字段设置(显示设备分辨率、设置显示缓冲区等等设置)。

第三步：调用函数 lv_disp_drv_register(&disp_drv) 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。

```
/*-----------------------------------
* 在 LVGL 中注册显示设备
*-----------------------------------*/

static lv_disp_drv_t disp_drv;                  /* 显示设备的描述符 */
lv_disp_drv_init(&disp_drv);                    /* 初始化为默认值 */

/* 建立访问显示设备的函数  */

/* 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 = lcddev.width;
disp_drv.ver_res = lcddev.height;

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

/* 设置显示缓冲区 */
disp_drv.draw_buf = &draw_buf_dsc_1;
/* 注册显示设备 */
lv_disp_drv_register(&disp_drv);
```

从上述源码可知：如果缓冲区初始化完成后设置设备字段。LVGL 官方给出必须填写字段如下所示：

(1) draw_buf：指向已初始化 lv_disp_draw_buf_t 变量的指针。

(2) hor_res：显示器的水平分辨率（以像素为单位）。

(3) ver_res：显示器的垂直分辨率（以像素为单位）。

(4) flush_cb：一个回调函数，用于将缓冲区的内容复制到显示的特定区域。

2. lv_disp_drv_t 指针当中，有一些是可选的数据字段，如表 5.4.1.1 所示：

字段	描述
physical_hor_res	完整/物理显示器的水平分辨率（以像素为单位）
physical_ver_res	完整/物理显示器的垂直分辨率（以像素为单位）
offset_x	完整/物理显示的水平偏移（以像素为单位）
offset_y	完整/物理显示的垂直偏移（以像素为单位）
color_chroma_key	在 chrome 键控图像上将被绘制为透明的颜色(lv_conf.h 文件定义)
anti_aliasing	使用抗锯齿（边缘平滑）
rotated 和 sw_rotate	软件旋转
screen_transp	设置屏幕具有透明度。LV_COLOR_SCREEN_TRANSP 必须启用，LV_COLOR_DEPTH 必须为 32，这些宏定义在 lv_conf.h 文件定义
user_data	驱动程序的自定义 void 用户数据

full_refresh	重绘整个屏幕
direct_mode	直接绘制到帧缓冲区

表 5.4.1.1 可选的数据字段

3. 如果使用 GPU(图形处理器)，可以使用下面的函数：

(1) gpu_fill_cb ：用颜色填充内存中的区域。

(2) gpu_wait_cb： 如果任何 GPU 函数返回，而 GPU 则是仍在工作 LVGL，将使用此功能需要时请务必 GPU 渲染已准备就绪。

5.4.2 屏幕旋转

LVGL 屏幕旋转功能的意义是指当设计硬件显示器时，粗心把 X 和 Y(长和宽) 调转，为了不重新设计电路，可使用 LVGL 软件旋转补偿，如图 5.5.1 所示：

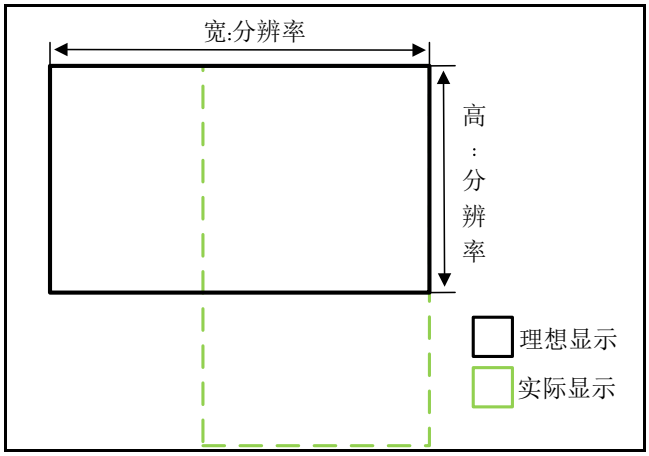


图 5.5.1 长宽调转

面对图 5.5.1 这种情况，可以使用 LVGL 软件旋转补偿，LVGL 支持以 90 度为增量旋转显示器。选择软件旋转时必须设置 sw_rotate 字段为 1 以及设置 rotated 字段为旋转角度，LVGL 将执行旋转。假设你执行旋转之后，屏幕的宽度和高度没有变化。只需像往常一样将像素刷新到显示器即可。使用软件旋转时，这个 flush_cb 回调函数不需要其他逻辑。

软件中执行旋转需要大量的内存开销，这就是为什么还可以使用硬件旋转的原因。在这种模式下，LVGL 进入缓冲区，就好像你的屏幕现在的宽度和高度都反转了一样。你有义务为旋转提供的像素。可以使用 rotated 标志设置显示器初始化时的默认旋转度，该标志可用选项如下所示：

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。

注意：旋转方向与沿顺时针方向旋转物理显示的方式有关。因此，这 LV_DISP_ROT_90 意味着将硬件顺时针旋转 90 度，显示屏将逆时针旋转 90 度进行补偿。也可以使用 API 在运行时更改显示旋转，LVGL 设置软件旋转函数如以下所示：

```
lv_disp_set_rotation(disp, rot)
```

屏幕旋转的步骤：

① 设置 sw_rotate 标志设置为 1。

② 使用旋转函数进行旋转。

```
lv_disp_set_rotation(lv_disp_get_default(), LV_DISP_ROT_180);
```

可以在注册显示设备时候进行旋转，该旋转值与 LCD 驱动代码绘画方向有关。

5.4.3 显示接口 API 函数

LVGL 官方提供显示接口相关 API，如表 5.4.3.1 所示：

函数	描述
注册显示设备相关函数	
lv_disp_drv_init()	初始化显示驱动程序
lv_disp_draw_buf_init()	初始化显示缓冲区
lv_disp_drv_register()	注册一个初始化的显示驱动程序
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()	获取显示器的完整/物理水平分辨率
lv_disp_get_physical_ver_res()	显示器的完整/物理垂直分辨率
lv_disp_get_offset_x()	获取完整/物理显示的水平偏移
lv_disp_get_offset_y()	从完整/物理显示获取垂直偏移
lv_disp_get_antialiasing()	获取是否为显示器启用了抗锯齿功能
lv_disp_get_dpi()	获取显示器的 DPI
屏幕旋转 API 函数	
lv_disp_set_rotation()	设置此显示的旋转
lv_disp_get_rotation()	获取此显示器的当前旋转
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 5.4.3.1 显示接口相关 API 函数

接下来我们来讲解 LVGL 显示接口常用的函数有哪些：

(1) lv_disp_drv_init 函数

初始化显示驱动程序，其函数原型如以下所示：

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

该函数的形参，如表 5.4.3.2 所示：

参数	描述
driver	指向要初始化的驱动变量的指针

表 5.4.3.2 函数 lv_disp_drv_init()形参描述

返回值：无。

(2) lv_disp_draw_buf_init 函数

初始化显示缓冲区，其函数原型如以下所示：

```
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，如表 5.4.3.3 所示：

参数	描述
draw_buf	要初始化的指针 lv_disp_draw_buf_t 变量
buf1	用来绘制图像的缓冲区。始终必须指定且不能为 NULL
buf2	指定第二个缓冲区以使图像渲染和图像刷新，可设置 NULL
size_in_px_cnt	buf1 和 buf2 像素大小

表 5.4.3.3 函数 lv_disp_draw_buf_init()形参描述

返回值：无。

(3) lv_disp_drv_register 函数

注册一个初始化的显示驱动程序，其函数原型如以下所示：

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

该函数的形参，如表 5.4.3.4 所示：

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

表 5.4.3.4 函数 lv_disp_drv_register()形参描述

返回值：指向新显示的指针或错误时为 NULL。

(4) lv_disp_remove 函数

移除显示器，其函数原型如以下所示：

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

该函数的形参，如表 5.4.3.5 所示：

参数	描述
disp	要显示的指针

表 5.4.3.5 函数 lv_disp_remove()形参描述

返回值：无。

(4) lv_disp_get_default 函数

获取默认显示设备，其函数原型如以下所示：

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

返回值：指向默认显示的指针。

(5) lv_disp_set_rotation 函数

设置此显示设备的旋转，其函数原型如以下所示：

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

该函数的形参，如表 5.4.3.6 所示：

参数	描述
disp	指向显示的指针（NULL 使用默认显示）
rotation	旋转角度

表 5.4.3.6 函数 lv_disp_set_rotation()形参描述

返回值：无。

5.5 输入设备

输入设备是用户和计算机系统之间进行信息交换的主要装置之一。例如触摸、键盘、鼠标输入装置等都属于输入设备。前面 LVGL 移植的时候，笔者已经说过，LVGL 是支持某种输入设备的，所以 LVGL 输入设备是在 lv_port_indev_template.c 文件中定义的。

在注册输入设备之前，我们务必至少注册一个显示器，接下来我们来讲解 LVGL 输入设备的注册流程有哪些，如下源码所示：

```
lv_disp_drv_register(&disp_drv);      /* 注册输入设备 */
```

```
static lv_indev_drv_t indev_drv;
lv_indev_drv_init(&indev_drv);          /* 输入设备的初始化 */
indev_drv.type =...                     /* 设置输入设备类型 */
indev_drv.read_cb =...                  /* 设置输入设备回调函数 */
/* 在 LVGL 中注册驱动程序，并保存创建的输入设备对象 */
lv_indev_t * my_indev = lv_indev_drv_register(&indev_drv);
```

indev_drv.type 参数指向输入设备类型，该设备类型如下所示：

① LV_INDEV_TYPE_POINTER：触摸板或鼠标。

② LV_INDEV_TYPE_KEYPAD：键盘或小键盘。

③ LV_INDEV_TYPE_ENCODER：具有左/右转和推动选项的编码器。

④ LV_INDEV_TYPE_BUTTON：外部按钮虚拟按下屏幕。

indev_drv.read_cb 是一个函数指针，它将定期调用以报告输入设备的当前状态。

5.5.1 触摸设备

在前面 LVGL 移植时候，我们已经讲述正点原子例程添加触摸设备的步骤和方法，这里我们就不必重复讲解了。

5.5.2 光标设备

光标是鼠标指示器，例如我们的电脑的鼠标显示在显示屏上，当鼠标移动时，光标也跟着移动，在正点原子例程中，因为电阻屏或者电容屏只有显示和触摸功能，所以光标想获取显示屏的坐标，必定与触摸相关，我们怎么添加光标设备呢？，LVGL 官方例程中也有说明，如下源码所示：

```
/*------------------
 * 鼠标
 * ------------------*/

/**
 * @brief        初始化并注册输入设备
 * @param        无
 * @retval       无
 */
void lv_port_indev_init(void)
{
    /* 初始化鼠标(如果有) */
    mouse_init();

    /* 注册鼠标输入设备 */
    lv_indev_drv_init(&indev_drv);
    indev_drv.type = LV_INDEV_TYPE_POINTER;
    indev_drv.read_cb = mouse_read;
    indev_mouse = lv_indev_drv_register(&indev_drv);

    /* 设置光标，为了简单起见，现在设置为一个 HOME 符号 */
```

```
    lv_obj_t * mouse_cursor = lv_img_create(lv_scr_act());
    lv_img_set_src(mouse_cursor, LV_SYMBOL_HOME);
    lv_indev_set_cursor(indev_mouse, mouse_cursor);
}

/**
 * @brief       初始化鼠标
 * @param       无
 * @retval      无
 */
static void mouse_init(void)
{
    /*Your code comes here*/
    tp_dev.init();
    /* 电阻屏如果发现显示屏 XY 镜像现象，需要坐标矫正 */
    if (0 == (tp_dev.touchtype & 0x80))
    {
        tp_adjust();
        tp_save_adjust_data();
    }
}

/**
 * @brief       图形库的鼠标读取回调函数
 * @param       indev_drv   : 鼠标设备
 *   @arg       data        : 输入设备数据结构体
 * @retval      无
 */
static void mouse_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data)
{
    /* 获取当前的 x、y 坐标 */
    mouse_get_xy(&data->point.x, &data->point.y);

    /* 获取是否按下或释放鼠标按钮 */
    if(mouse_is_pressed()) {
        data->state = LV_INDEV_STATE_PR;
    } else {
        data->state = LV_INDEV_STATE_REL;
    }
}

/**
 * @brief       获取鼠标设备是否被按下
```

```
 * @param       无
 * @retval      返回鼠标设备是否被按下
 */
static bool mouse_is_pressed(void)
{
    /*Your code comes here*/
    tp_dev.scan(0);

    if (tp_dev.sta & TP_PRES_DOWN)
    {
        return true;
    }
    return false;
}

/**
 * @brief       当鼠标被按下时，获取鼠标的 x、y 坐标
 * @param       x    : x 坐标的指针
 *   @arg       y    : y 坐标的指针
 * @retval      无
 */
static void mouse_get_xy(lv_coord_t * x, lv_coord_t * y)
{
    /*Your code comes here*/
    (*x) = tp_dev.x[0];
    (*y) = tp_dev.y[0];
}
```

根据上述的源码，光标设备设置的步骤如下所示：

① 调用 mouse_init 函数初始化 touch 驱动。

② 调用 lv_indev_drv_register 函数注册输入设备。

③ 定义一张光标的图片部件。

④ 设置光标的图片。

⑤ 调用 lv_indev_set_cursor 函数设置光标。

5.5.3 输入设备相关 API

LVGL 官方提供输入设备相关 API，如表 5.5.3.1 所示：

函数	描述
lv_indev_drv_init()	使用默认值初始化输入设备驱动程序
lv_indev_drv_register()	注册一个初始化的输入设备驱动程序
lv_indev_drv_update()	在运行时更新驱动程序
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 5.5.3.1 输入设备相关 API 函数

接下来我们来讲解 LVGL 输入设备常用的函数有哪些：

(1) lv_indev_drv_init 函数

初始化输入设备驱动程序，其函数原型如以下所示：

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

该函数的形参，如表 5.5.3.2 所示：

参数	描述
driver	指向要初始化的驱动变量的指针

表 5.5.3.2 函数 lv_indev_drv_init()形参描述

返回值：无。

(2) lv_indev_drv_register 函数

注册一个初始化的输入设备驱动程序，其函数原型如以下所示：

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

该函数的形参，如表 5.5.3.3 所示：

参数	描述
driver	指向初始化的“lv_indev_drv_t”变量的指针（可以是局部变量）

表 5.5.3.3 函数 lv_indev_drv_register()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，其函数原型如以下所示：

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

该函数的形参，如表 5.5.3.4 所示：

参数	描述
indev	要删除的输入设备指针

表 5.5.3.4 函数 lv_indev_delete()形参描述

返回值：无。

(4) _lv_indev_read 函数

输入设备读取数据，其函数原型如以下所示：

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

该函数的形参，如表 5.5.3.5 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 5.5.3.5 函数_lv_indev_read()形参描述

返回值：无。

5.6 LVGL 滴答

有学习过操作系统的小伙伴都知道，RTOS 就是靠一个系统定时器以某个频率中断为操作系统提供调度(上下文切换)实现任务切换。显然地，LVGL 系统滴答也具有相同的操作，LVGL 需要一个系统时钟来知道动画和其他任务的经过时间。

在第二章和第三章中，我们了解到正点原子一般使用定时器或者 FreeRTOS 提供 LVGL 滴答时钟(lv_tick_inc())，请大家查看第三章第 3.1 小节。下面我们来讲解 LVGL 时钟相关的函数有哪些，如下所示：

(1) lv_tick_get 函数

该函数是获取自启动以来经过的毫秒数，该函数的原型如以下所示：

```
uint32_t lv_tick_get(void)
```

返回值：经过的毫秒数。

(2) lv_tick_elaps 函数

该函数是获取自上一个时间戳以来经过的毫秒数，该函数的原型如以下所示：

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

该函数具有一个参数，如表 5.6.1 所示：

参数	描述
prev_tick	自“ prev_tick”以来经过的毫秒数

表 5.6.1 lv_tick_elaps()函数描述

返回值：毫秒数。

5.6.2 LVGL 任务处理

什么是 LVGL 的任务处理程序，简单来说：LVGL 的任务处理程序(lv_timer_handler)类似于 RTOS 的切换任务的任务调度函数，该函数计时并不严格，但应保持大约 5 毫秒内，以保持系统响应速度即可。LVGL 任务处理并不是抢占式任务调度，而是轮询调度。

要处理 LVGL 的任务或者回调函数，需要 lv_timer_handler 定期调用，如以下方法所示：

① main 函数的 while(1)。

② 定时中断（优先级低于 lv_tick_inc）。

③ 定期执行 OS 任务。

5.6.3 LVGL 的睡眠管理

睡眠模式就是低功耗模式，当没有触发信号时，可以让系统进入睡眠模式，减少系统的功耗，如下源码所示：

```
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);
}
```

如果发生唤醒(按下，触摸或点击等)，应该在输入设备(touch)读取功能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 */
```

5.7 操作系统和中断

1. 操作系统

LVGL 默认是线程不安全的，LVGL 为什么是线程不安全，其实也很简单理解：主要原因是操作系统是抢占式的系统，如果任务 1 调用了 LVGL 的函数，而任务 2 也是调用 LVGL 的函数，如果任务 1 还没有执行完毕就被任务 2 抢占了，那么任务 1 的函数无法及时执行了。如果项目很小，可能没有出现这样的结果，一旦你的项目很大，完成依赖 LVGL，那么可能出现卡顿或者显示不全等问题。

在 LVGL 的事件回调和 Task 中，调用 LVGL 相关函数是有效的。而其他地方需要 IPC 通讯机制来解决，LVGL 作者建议使用实际任务或线程，则需要一个互斥锁(防止任务翻转问题)，该互斥锁应在调用之前被调用， 并在 lv_timer_handler 函数之后释放。此外，你还必须在与每个 LVGL(lv_...)相关的函数调用和代码周围的其他任务和线程中使用相同的互斥锁。这样，你就可以在真正的多任务环境中使用 LVGL。只需使用互斥锁避免同时调用 LVGL 函数。说了这么多，我们该怎么调用呢，以下是官方的模板，如下源码所示：

```
void lvgl_thread(void)
{
    while(1)
    {
        xSemaphoreTake(MutexSemaphore,portMAX_DELAY);
        lv_task_handler();
        xSemaphoreGive(MutexSemaphore);     /* 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 */
    xSemaphoreTake(MutexSemaphore,portMAX_DELAY);
    lv_obj_t *img = lv_img_create(lv_scr_act());
    xSemaphoreGive(MutexSemaphore); /* 释放互斥信号量 */

    while(1)
    {
        xSemaphoreTake(MutexSemaphore,portMAX_DELAY);
        /* 切换到下一个图像 */
        lv_img_set_src(img, next_image);
        xSemaphoreGive(MutexSemaphore); /* 释放互斥信号量 */
        thread_sleep(2000);
    }
}
```

2. 中断

尽量避免从中断(lv_tick_inc 和 lv_disp_flush_ready 函数除外)中调用 LVGL 函数。如果需要执行此操作，则必须在 lv_timer_handler 函数运行时，禁用 LVGL 功能的中断。

5.8 LVGL 日志

LVGL 具有内置的日志功能，用于通知用户知道当前 LVGL 库中正在发生的事情。

LVGL 默认是不开启 LVGL 日记，如果我们想开启 LVGL 日志功能，可以在 lv_conf.h 文件中找到宏定义 LV_USE_LOG 置 1 操作，LVGL 日志分为很多类型，如以下所示：

① LV_LOG_LEVEL_TRACE：很多日志提供详细信息。
② LV_LOG_LEVEL_INFO：记录重要事件。
③ LV_LOG_LEVEL_WARN：记录是否发生了不必要的事情，但没有引起问题。
④ LV_LOG_LEVEL_ERROR：唯一关键问题，系统可能会发生故障。
⑤ LV_LOG_LEVEL_USER：仅用户消息。
⑥ LV_LOG_LEVEL_NONE：不要记录任何内容级别高于设置的日志级别的事件也将记录。例如，如果是 LV_LOG_LEVEL_WARN，则错误也将被记录下来。

由于 LVGL 也是支持 printf，所以使用 printf 来输出日志信息，在 lv_conf.h 文件中找到宏定义 LV_LOG_PRINTF，把它置 1 表示开启。

如果无法使用 printf 或者不想使用 LVGL 内嵌日志函数进行日志记录，则自行注册“logger”回调 lv_log_register_print_cb，并自定义日志功能，如下源码所示：

```
void my_log_cb(const char * buf)
{
    printf(buf);
}
/* 可以在注册显示器中定义 */
lv_log_register_print_cb(my_log_cb);
```

注意：上述的函数必须在 lv_conf.h 文件的宏定义 LV_USE_LOG 置 1 而 LV_LOG_PRINTF 宏定义置 0 操作，然后在宏定义 LV_LOG_LEVEL 选择日志类型即可。

在自己的程序中，可以通过函数 LV_LOG_TRACE/INFO/WARN/ERROR(description) 功能使用日志模块。

第六章 LVGL 基础知识

工欲善其事必先利其器，在我们学习 LVGL 使用之前，我们必须了解 LVGL 的基础知识，例如 LVGL 的样式管理、布局管理以及动画的制作等相关知识，本章节讲解的知识非常重要，对于后面的学习起到承上启下的作用，希望读者能认真学习本章节的内容。

本章分为几个部分讲解：

6.1 LVGL 对象介绍
6.2 LVGL 布局介绍
6.3 LVGL 样式属性
6.4 LVGL 滚动属性
6.5 LVGL 动画
6.6 LVGL 定时器
6.7 LVGL 事件

6.1 LVGL 对象介绍

在 LVGL 中，用户界面的基本构建块是对象，也称为小部件。例如：按钮、标签、图片、列表、图表或文本区域等部件，都是简称对象，而每个对象都有公用属性也有它们的私有属性，本小节主要讲解对象的公用属性，例如你和你同桌的座位是学校的，是公用的座位，而书包是你自己的，所以它就是私人的东西。虽然 LVGL 图形库采用纯 c 语言开发，但是所编写的代码类似于 C++的面对对象思维。

6.1.1 对象的基本属性(公用属性)

在 LVGL 中，每个对象都有一些基本而相同的属性，例如：

① 大小
② 父类
③ 样式
④ 事件
⑤ 位置

对于上述的属性设置，LVGL 作者为了方便管理，设计了一套简单的函数来管理通用的属性，例如设置属性使用 lv_obj_set_size/pos(<widget>,xxx)函数设置，widget 是 LVGL 小部件名称：例如按键(button)、标签(label)、对象(obj)等小部件，如果我们获取部件的大小、父类、位置等属性，可使用 lv_obj_get_size/pos(<widget>,xxx)函数获取相关的属性，如下源码所示：

```
/* 设置基础对象属性 */
lv_obj_set_size(btn1, 100, 50);     /* 按键大小 */
lv_obj_set_pos(btn1, 20,30);        /* 按键位置 */
```

6.1.2 对象的具体属性(私有属性)

对象类型也具有特殊的属性。例如，滑块具有以下属性：

① 当前值
② 最小值和最高值

对于这些属性，每种对象都有唯一的 API 函数来设置。如下源码所示：

```
/* 设置滑块的特殊属性 */
lv_slider_set_range(slider1, 0, 100);           /* 设置滑块的最小、最大值 */
lv_slider_set_value(slider1, 40, LV_ANIM_ON);  /* 设置滑块当前值 */
```

6.1.3 LVGL 工作机制

1. LVGL 父类与子类的区别

父类可以视为子类的容器，它可以拥有多个子类，而子类就只有一个父类（屏幕除外，一般情况下，你只有一个爸爸，但爸爸有很多个儿子)。值得注意的是：子类可以当父类，（一般情况下，你只有一个爸爸，但爸爸有很多个儿子，你结婚生了一个儿子，那么你就是你儿子的父类，但是你还是你爸爸的子类)。

总结：

(1) 如果父类对象移动，则子类也会跟着父类移动，如图 6.1.3.1 所示：

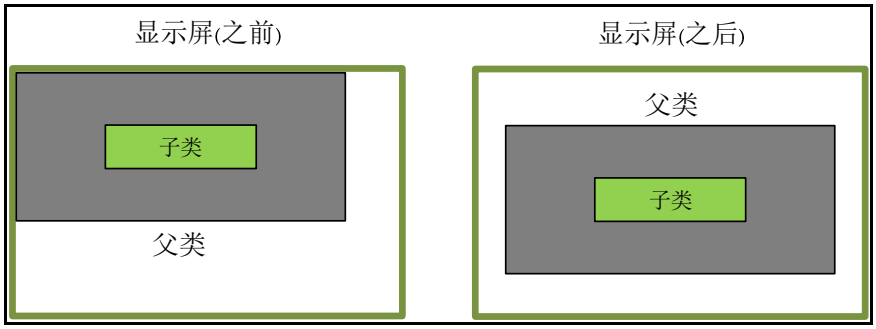


图 6.1.3.1 父移子随

(2) 如果子类移动，则父类不会跟着子类移动；如果子类移动的位置超过父类的范围，则子类超过的部分不能被显示，如图 6.1.3.2 所示：

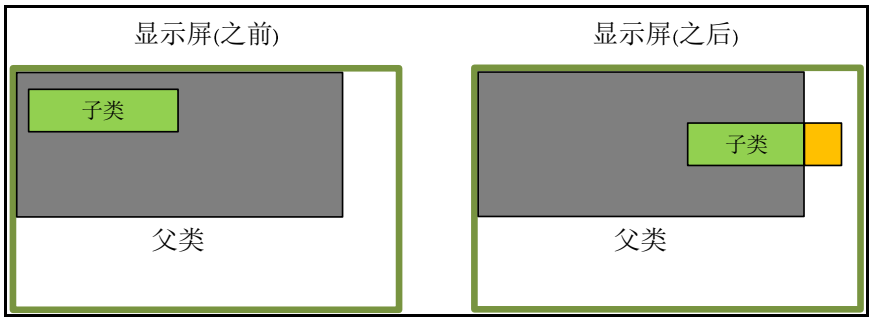


图 6.1.3.2 子移父不随

上图中的黄色区域就是子类移动时超出了父类的范围，显然这个区域不会在显示屏上绘制。

(3) 父类对象在显示屏中显示，它的位置以显示屏的左上角(0,0)点为起点，而子类相反，它以父类对象的左上角为(0,0)点，例如显示屏的分辨率为 800*480，父类创建的对象大小为 400*240 且对象位置在(400,240)坐标，那么对于父类显示范围来讲：显示屏左上角起点是(0,0)，终点为(800,480)，而子类显示范围的起点是父类的左上角，而终点是(400,240)，如图 6.1.3.3 所示：

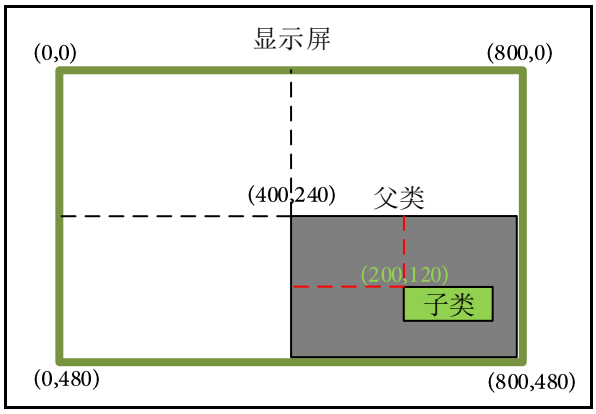


图 6.1.3.3 父类与子类的坐标定义

2. 创建对象与删除对象

在 LVGL 中，我们可以在程序运行中动态创建或者删除一个对象。这意味着仅当前创建的对象消耗 SRAM 内存被释放了。例如，在程序运行时创建一个部件，并在不可见或不必要时将其删除。每个对象都有自己的带有统一原型的创建功能。如下源码所示：

```
lv_obj_t * lv_<widget>_create(lv_obj_t * parent, <other parameters if any>);
```

注意：一般情况下，创建某个部件函数只有一个形参，那就是 parent，它指向父类，但是某些部件有两个或者两个以上的形参，例如 lv_win 窗口部件。所有对象均以 C 语言中的 lv_obj_t 指针作为句柄进行引用。并且可使用该指针设置或获取对象的属性。

如果删除一个对象，可使用以下几个函数删除：

① lv_obj_del(lv_obj_t * obj)：删除一个对象，那么该对象的子类一并删除(立刻删除)。

② lv_obj_del_async(lv_obj_t * obj)：等到下一次执行 lv_timer_handler 后删除对象(周期后删除)。

③ lv_obj_clean(lv_obj_t * obj)：删除一个对象的全部子类(立刻删除)。

④ lv_obj_del_delayed(lv_obj_t * obj, uint32_t delay_ms)：延时毫秒再删除对象(等待删除)。

3. LVGL 屏幕函数

(1) 创建屏幕:

前面笔者也讲解过屏幕是没有父对象的特殊对象，那么创建一个屏幕，可使用创建对象的方式：

```
lv_obj_t * scr1 = lv_obj_create(NULL);
```

注意：没有父类，所以填写 NULL 参数。

(2) 获取活动的屏幕:

在默认的情况下，LVGL 初始化时，系统已经帮用户创建一个活动屏幕，不需要我们再一次创建屏幕，获取该活动屏幕可以使用 lv_scr_act 函数获取。

(3) 加载屏幕:

如果要加载新的屏幕，可调用以下函数：

```
lv_scr_load(scr1)       /* 加载屏幕 */
```

(4) 动画加载屏幕:

可使用以下函数加载新屏幕的动画：

```
lv_scr_load_anim(scr, transition_type, time, delay, auto_del)
```

函数 lv_scr_load_anim 的 transition_type 存在以下转换类型，如表 6.1.3.4 所示：

参数	描述
LV_SCR_LOAD_ANIM_NONE	延时毫秒后立即切换
LV_SCR_LOAD_ANIM_OVER_LEFT/RIGHT/TOP/BOTTOM	将新屏幕移到给定方向上

LV_SCR_LOAD_ANIM_MOVE_LEFT/RIGHT/TOP/BOTTOM	将当前屏幕和新屏幕都移至给定方向
LV_SCR_LOAD_ANIM_FADE_ON	使新屏幕淡入旧屏幕

表 6.1.3.4 transition_type 形参描述

设置 auto_del 参数为 true 会在动画结束时自动删除旧屏幕，lv_scr_act 动画开始播放后，新屏幕将变为活动状态。

4. LVGL 图层

关于 LVGL 的层数问题，只了解三个问题即能理解本小节讲解的内容。第一问题：创建两个对象且这两个对象出现有交集，交集部分系统该如何绘制；第二个问题：两个对象出现交集，怎么样把底下的对象置上；第三个问题：LVGL 的图层有哪些。本小节，我们围绕着这三个问题来讲解 LVGL 图层的知识。

(1) 创建对象的顺序 (第一个问题)

LVGL 会在背景上绘制旧对象，在前景上绘制新对象。例如我们创建一个 button1 按钮对象，然后创建另一个名为 button2 按钮对象，如果 button1 和 button2 对象出现交集，则 button1 对象交集部分被 button2 对象交集部分覆盖。简单来讲：如果对象间出现交集，则谁先被创建谁被后一个覆盖，如图 6.1.3.5 所示：

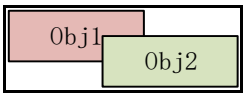


图 6.1.3.5 对象 2 覆盖对象 1

(2) 前台和后台 (第二个问题)

在(1)的讲述中，我们知道两个对象出现交集时，谁先被创建谁被后一个对象覆盖，那么我们可以把被覆盖的对象变成覆盖对象吗？答案是可以的，LVGL 作者为解决这个问题，提供了读者三个函数，如下所示：

① lv_obj_set_top 函数可以把对象置顶操作，如果 obj 或它的任何子对象被点击，那么 LVGL 将自动将该对象带到前台(置顶)。它的工作原理类似于 PC 上的典型 GUI，当点击背景中的一个窗口时，它会自动来到前台。如图 6.1.3.6 所示：

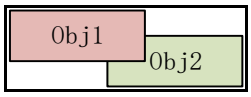


图 6.1.3.6 对象 1 覆盖对象 2

② lv_obj_move_foreground 函数，该函数告诉 LVGL 库将一个对象放到前台。

③ lv_obj_move_background 函数，该函数把对象移动到背景。

(3) LVGL 顶层和系统层 (第三个问题)

LVGL 具有两个特殊的层，分别名为 layer_top 和 layer_sys。两者在显示器的所有屏幕上都是可见的。但是，它们不能在多个物理显示器之间共享，layer_top 总是位于默认屏幕的顶部(lv_scr_act)，而 layer_sys 位于 layer_top 的顶部。

用户可在 layer_top 层创建一些随处可见的内容和部件，例如一个菜单栏，一个弹出窗口等部件。如果系统的单击属性是启用的，则 layer_top 层将吸收所有用户单击，并把它作为一个模态处理，点击属性设置方法如下源码所示：

```
lv_obj_add_flag(lv_layer_top(),LV_OBJ_FLAG_CLICKABLE);/* 设置可点击 */
```

注意：layer_sys 层在 layer_top 层上。例如鼠标的光标，系统以确保它总是可见的。

6.2 LVGL 布局介绍

LVGL 设置坐标的概念受到 CSS 的启发，但是 LVGL 绝不是 CSS 的完整实现，而是实现了一个可比较的子集。下面笔者带读者认识 LVGL 布局属性有哪些，如下所示：

6.2.1 对象坐标的设置

在 LVGL 中，设置对象的坐标有两种单位，如以下所示：

像素：整数填写。

```
lv_obj_set_x(obj,10)          /* 设置对象 x 轴坐标 */
lv_obj_set_y(obj,10)          /* 设置对象 y 轴坐标 */
lv_obj_set_pos(obj,10,10)     /* 设置对象 x,y 轴坐标 */
```

百分比：根据父级的内容区域大小计算，使用 lv_pct 函数将值转换为百分比。

```
lv_obj_set_x(obj,lv_pct(10))                /* 设置对象 x 轴坐标 */
lv_obj_set_y(obj,lv_pct(10))                /* 设置对象 y 轴坐标 */
lv_obj_set_pos(obj,lv_pct(10),lv_pct(10))   /* 设置对象 x,y 轴坐标 */
```

如果读者需要获取一个对象的坐标，而该坐标刚刚被修改了，那么我们可以调用 lv_obj_update_layout 函数强制 LVGL 重新计算该坐标，该函数会刷新所有对象的位置、大小和布局。

6.2.2 对象尺寸设置

在 LVGL 中，设置对象的尺寸有两种单位，如以下所示：

像素：整数填写。

```
lv_obj_set_width(obj, 200);          /* 设置对象的宽度*/
lv_obj_set_height(obj, 100);         /* 设置对象的高度*/
lv_obj_set_size(obj, 200, 100);      /* 设置对象的高度和宽度*/
```

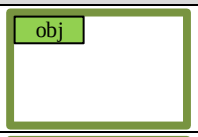
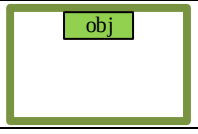
百分比：根据父级的内容区域大小计算，例如将对象的高度设置为屏幕高度。

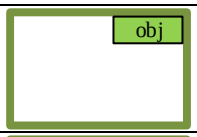
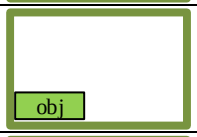
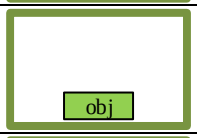
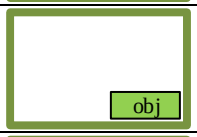
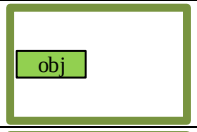
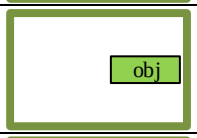
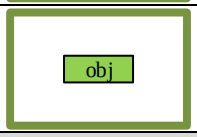
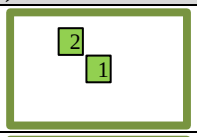
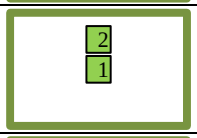
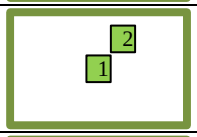
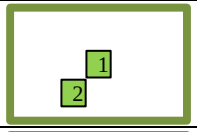
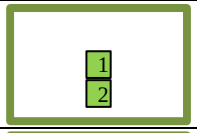
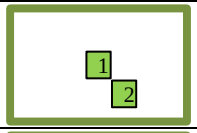
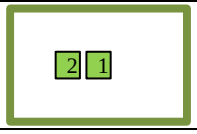
```
lv_obj_set_height(obj, lv_pct(100));/* 设置对象的高度为屏幕高度*/
```

6.2.3 对象的对齐设置

对齐在设计中是非常重要地。对齐的方式主要有：左对齐，右对齐，居中对齐，两端对齐等对齐方式，因为在 GUI 界面设计中，往往会用到很多小部件，如果我们手动一个个去调整部件的位置，那么开发效率会非常的慢，所以基于这样情况，对象对齐就登场了，它能解决手动调整部件位置的问题。

首先我们来看一下 LVGL 对齐模式有哪些，如表 6.2.3.1 所示：

对齐模式	描述	
内部对齐(用作于子类于父类对齐)		
LV_ALIGN_TOP_LEFT	顶部左边对齐	
LV_ALIGN_TOP_MID	顶部中间对齐	

LV_ALIGN_TOP_RIGHT	顶部右边对齐	
LV_ALIGN_BOTTOM_LEFT	底部左边对齐	
LV_ALIGN_BOTTOM_MID	底部中间对齐	
LV_ALIGN_BOTTOM_RIGHT	底部右边对齐	
LV_ALIGN_LEFT_MID	左边中间对齐	
LV_ALIGN_RIGHT_MID	右边中间对齐	
LV_ALIGN_CENTER	中间对齐	
外部对齐(作用于子类与子类或者父类与父类对齐)		
LV_ALIGN_OUT_TOP_LEFT	向外顶部左边对齐	
LV_ALIGN_OUT _TOP_MID	向外顶部中间对齐	
LV_ALIGN_OUT _TOP_RIGHT	向外顶部右边对齐	
LV_ALIGN_OUT _BOTTOM_LEFT	向外底部左边对齐	
LV_ALIGN_OUT _BOTTOM_MID	向外底部中间对齐	
LV_ALIGN_OUT _BOTTOM_RIGHT	向外底部右边对齐	
LV_ALIGN_OUT _LEFT_MID	向外左边中间对齐	

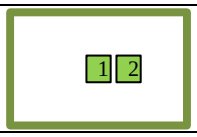
LV_ALIGN_OUT _RIGHT_MID	向外右边中间对齐	

表 6.2.3.1 LVGL 对齐模式

LVGL 提供了两个函数是用来设置对象的对齐，如以下函数所示：

```
void lv_obj_align(struct _lv_obj_t * obj,              /* 要对齐的对象 */
                  lv_align_t align,                    /* 要对齐的模式 */
                  lv_coord_t x_ofs,                    /* 要对齐后向 x 轴偏移量 */
                  lv_coord_t y_ofs);                   /* 要对齐后向 y 轴偏移量 */
```

该函数是子类与父类对齐，而父类对象是创建部件时的 parent 对象。

```
void lv_obj_align_to( struct _lv_obj_t * obj,          /* 要对齐的对象 */
                      const struct _lv_obj_t * base,   /* 向谁对齐 */
                      lv_align_t align,                /* 要对齐的模式 */
                      lv_coord_t x_ofs,                /* 要对齐后向 x 轴偏移量 */
                       lv_coord_t y_ofs);              /* 要对齐后向 y 轴偏移量 */
```

如果一个父类中创建了两个子类，则可使用上述函数对两个子类实施对齐 。

总结：

lv_obj_align 函数用作于子类在父类内的对齐，比如教室内的黑板和教室的对齐方式。

lv_obj_align_to 函数用作于两个对象的对齐，比如两个教室的对齐方式。

6.3 LVGL 样式属性

LVGL 样式属性是指用于设置对象的外观，LVGL 的样式深受 CSS 的启发。

① 样式是一个 lv_style_t 变量，它可以保存边框宽度、文本颜色等属性。它类似于 CSS 中的类。

② 可以将样式分配给对象以更改其外观。在赋值时，可以指定目标部分(CSS 中的伪元素)和目标状态(伪类)。例如，当一个滑块处于按下状态时，可以在它的旋钮上添加 style_blue 样式。

③ 任何数量的对象都可以使用相同的样式。

④ 样式可以级联，这意味着可以将多个样式分配给一个对象，并且每个样式可以具有不同的属性。因此，并非所有属性都必须在一个样式中指定。LVGL 将搜索一个属性，直到一个样式定义它，或者如果它没有被任何样式指定，则使用默认值。

⑤ 样式具有优先级。如果两个函数同时指向某个属性，则使用对象根据最后定义的函数描绘样式属性。

⑥ 如果对象中未指定某些属性（例如文本颜色），则可以从父对象继承。

⑦ 对象样式设置分为两种，一种为普通样式，另一种为本地样式，而本地样式比普通样式优先级高，所以优先执行本地样式。

6.3.1 LVGL 样式属性设置

在 LVGL 中，样式属性设置具有两个方式：

1. 普通样式设置，如下源码所示：

```
static lv_style_t style_btn;                                  /* 定义样式变量 */
lv_style_init(&style_btn);                                    /* 样式初始化 */
lv_style_set_bg_color(&style_btn, lv_color_hex(0x115588));/* 设置背景 */
```

```
lv_style_set_bg_opa(&style_btn, LV_OPA_50);                /* 设置背景透明度 */
lv_style_set_border_width(&style_btn, 2);                  /* 设置边框的宽度 */
lv_style_set_border_color(&style_btn, lv_color_black());   /* 设置边框的大小 */

lv_obj_t * obj = lv_obj_create(lv_scr_act());              /* 创建一个对象 */
lv_obj_add_style(obj, & style_btn, LV_STATE_DEFAULT);      /* 设置对象 1 的样式 */
lv_obj_t * obj1 = lv_obj_create(lv_scr_act());             /* 创建一个对象 1 */
lv_obj_add_style(obj1, & style_btn, LV_STATE_DEFAULT);     /* 设置对象 1 的样式 */
```

上述源码可知：首先定义样式变量 style_btn，然后我们调用 lv_style_init 函数初始化样式，最后调用样式属性函数设置对象的样式属性。值得注意的是：如果其他对象想使用 style_btn 样式可直接调用 lv_obj_add_style 函数添加该样式，因为普通样式可以用作于多个对象。

普通样式属性函数的通用结构如下所示：

```
lv_style_set_<property_name>(&style, <value>);
```

2. 本地样式设置，如下源码所示：

```
/* 创建一个对象 */
lv_obj_t * obj = lv_obj_create(lv_scr_act());
/* 设置对象背景颜色 */
lv_obj_set_style_bg_color(obj, lv_color_red(),LV_STATE_DEFAULT);
/* 创建一个对象 1 */
lv_obj_t * obj1 = lv_obj_create(lv_scr_act());
/* 设置对象 1 背景颜色 */
lv_obj_set_style_bg_color(obj1, lv_color_red(),LV_STATE_DEFAULT);
```

上述源码可知：我们必须创建一个部件，然后在该部件上使用本地样式，本地样式属性函数不能用作于多个对象，只能针对一个对象。

本地样式属性函数的通用结构如下所示：

```
lv_obj_set_style_<property_name>(obj, <value>, <selector>);
```

注意：本地样式的设置，它的形参<selector>的填写是根据下面的两个小节讲解的内容来填写的。

6.3.2 部件组成部分

我们知道绘画一个圆角矩阵可分为几个部分来绘画，首先我们绘画四条线，然后绘画圆弧，最后组成为圆角矩阵，如下图所示：

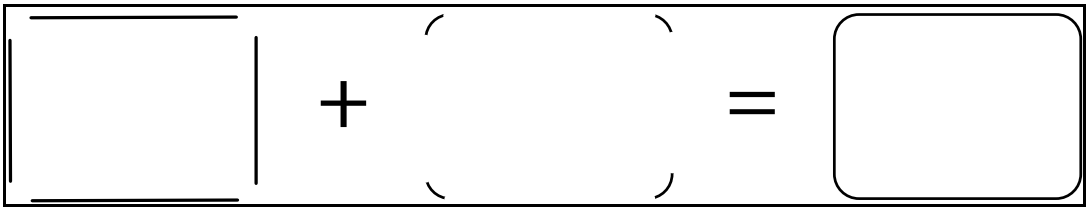


图 6.3.2.1 圆角矩形绘制示意图

从上图中，我们知道圆角矩形是由两个部分组成的，类似地，LVGL 小部件也是一样，每一个小部件都有自己的组成部分，例如，基础对象使用背景和滚动条部分组成的，而 Slider(进度条部件)使用背景、指示器和旋钮部分组成的。这些组成部分类似于 CSS 中的伪元素，如表 6.3.2.1 所示：

部件组成部分	描述
LV_PART_MAIN	像矩形一样的背景
LV_PART_SCROLLBAR	滚动条
LV_PART_INDICATOR	指示器，例如滑块、条形图、开关或复选框的复选框
LV_PART_KNOB	旋转纽
LV_PART_SELECTED	选择框
LV_PART_ITEMS	项(表格单元格)
LV_PART_TICKS	刻度(图表或仪表)
LV_PART_CURSOR	光标(文本区域或图表)
LV_PART_CUSTOM_FIRST	添加自定义部件

表 6.3.2.1 对象组成部分描述

从上表可知，虽然 LVGL 具有多种组成部分，但不是每一个小部件都必须包含这些部分，例如 Slider(进度条)部件是由三个部分组成：

① 背景(LV_PART_MAIN)。

② 指示器(LV_PART_INDICATOR)。

③ 旋钮(LV_PART_KNOB)。

下面笔者使用一个示意图来讲解 Slider(进度条)部件的部分组成，如图 6.3.2.1 所示：

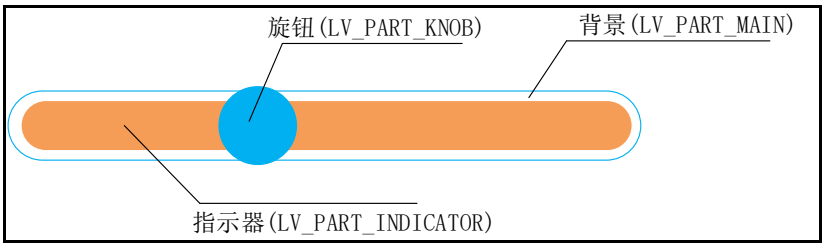


图 6.3.2.1 Slider(进度条)部件组成

前面我们已经讲解了样式属性的 知识，显然地，本小节的内容是对样式属性更深入的讲解，因为上图的 Slider(进度条)部件分为三个组成部分，所以我们使用样式属性分别设置这三个组成部分的样式就可以修改 Slider(进度条)部件样式了，对于 emWin 来讲就是换肤操作。

6.3.3 部件的状态

LVGL 所有部件可以是以下状态的组合，如表 6.3.3.1 所示：

状态	描述
LV_STATE_DEFAULT	正常状态
LV_STATE_CHECKED	切换或选中
LV_STATE_FOCUSED	通过键盘或编码器聚焦或通过触摸板/鼠标单击
LV_STATE_FOCUS_KEY	通过键盘或编码器而不是通过触摸板/鼠标聚焦
LV_STATE_EDITED	由编码器编辑
LV_STATE_HOVERED	鼠标悬停（现在不支持）
LV_STATE_PRESSED	已按下
LV_STATE_SCROLLED	滚动状态
LV_STATE_DISABLED	禁用状态
LV_STATE_USER_1	自定义状态
LV_STATE_USER_2	自定义状态
LV_STATE_USER_3	自定义状态
LV_STATE_USER_4	自定义状态

表 6.3.3.1 对象的状态

当用户与对象交互（按下、释放、聚焦等）时，状态通常由库自动更改，但是状态也可以手动更改。这里我们需要设置或清除给定状态（但保持其他状态不变）才能更改指定的状态，如下源码所示：

```
/* 添加/清除状态 */
lv_obj_add/clear_state(obj, part, LV_STATE_...)
/* 添加多个状态 */
lv_obj_add_state(obj, part, LV_STATE_PRESSED | LV_PRESSED_CHECKED)
```

如果我们使用普通样式来设置 Slider(进度条)部件的各个部分样式到底如何设置呢？如下源码所示：

```
static lv_style_t style_btn;                                    /* 定义样式变量 */
lv_style_init(&style_btn);                                      /* 样式初始化 */
/* 设置背景 */
lv_style_set_bg_color(&style_btn, lv_palette_main(LV_PALETTE_GREEN));

/* 创建进度条部件 */
lv_obj_t* lv_slider = lv_slider_create(lv_scr_act());
/* 居中设置 */
lv_obj_center(lv_slider);
/* 正常状态下设置指示器的样式属性 */
lv_obj_add_style(lv_slider, &style_btn, LV_PART_INDICATOR|LV_STATE_DEFAULT);

/* 创建对比部件 */
lv_obj_t* lv_obj1 = lv_slider_create(lv_scr_act());
/* 向外底下中间对齐 */
lv_obj_align_to(lv_obj1, lv_obj,LV_ALIGN_OUT_BOTTOM_MID,0,20);
```

上述源码可知：我们定义一个普通样式，该样式只设置背景属性为绿色，然后创建两个 Slider(进度条)部件，其中一个在 LV_STATE_DEFAULT 状态下设置部件指示器(LV_PART_INDICATOR)的背景颜色为绿色，而另一个使用系统默认样式属性，如图 6.3.3.2 所示：



图 6.3.3.2 设置指示器样式属性

从上图可知：我们把上方的 Slider(进度条)部件的指示器背景颜色设置为绿色，显然我们已经更改了该部件的指示器样式属性了。

如果我们使用本地样式来设置 Slider(进度条)部件的各个部分样式到底如何设置呢？如下源码所示：

```
/* 创建进度条部件 */
lv_obj_t* lv_slider = lv_slider_create(lv_scr_act());
/* 居中设置 */
lv_obj_center(lv_slider);
/* 正常状态下设置指示器样式 */
lv_obj_set_style_bg_color(lv_slider,lv_palette_main(LV_PALETTE_GREEN),
```

```
                        LV_PART_INDICATOR | LV_STATE_DEFAULT);

/* 创建对比部件 */
lv_obj_t* lv_obj1 = lv_slider_create(lv_scr_act());
/* 向外底下中间对齐 */
lv_obj_align_to(lv_obj1, lv_obj,LV_ALIGN_OUT_BOTTOM_MID,0,20);
```

上述源码实现功能和图 6.3.3.2 实现功能一样。

注意：上述源码是设置 Slider(进度条)部件的相关样式属性，在以后讲解的部件都是一样的设置方法，请根据这些部件的组成部分对应地设置。

关于样式的设置，笔者重点讲解一下“LV_PART_INDICATOR | LV_STATE_DEFAULT”形参，这个形参表示默认状态下更改指示器的样式，当然这个状态可以换成其他的状态，例如按下时更改指示器的样式属性。

6.3.4 LVGL 样式属性

6.3.4.1 大小与位置属性

该样式属性是与对象的大小、位置、对齐方式和布局相关的属性。

1. (width)宽度：

设置对象的宽度。单位可使用像素、百分比和 LV_SIZE_CONTENT，注意：百分比值是相对于父内容区域的宽度，如图 6.3.4.1.1 所示：

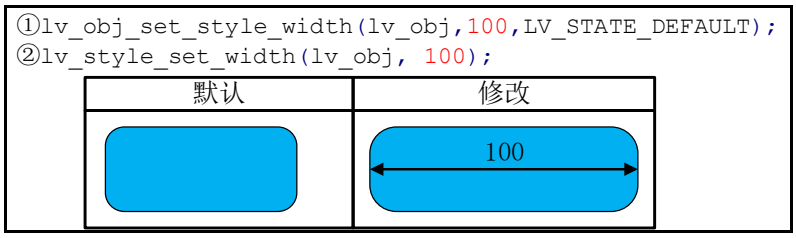


图 6.3.4.1.1 设置宽度属性

2. (min_width)最小宽度：

设置最小宽度，单位可使用像素、百分比。

注意：如果对象已经设置最小宽度，则调用 lv_obj_set_width 函数重新设置对象的宽度且该宽度小于最小宽度时，该函数不会得到响应。

3. (max_width)最大宽度：

设置最大宽度，可以使用像素和百分比值。

注意：如果对象已经设置最大宽度，则调用 lv_obj_set_width 函数重新设置对象的宽度且该宽度大于最大宽度时，该函数不会得到响应。

4. (height)高度：

设置对象的高度，单位可使用像素、百分比和LV_SIZE_CONTENT，如图6.3.4.1.2所示：

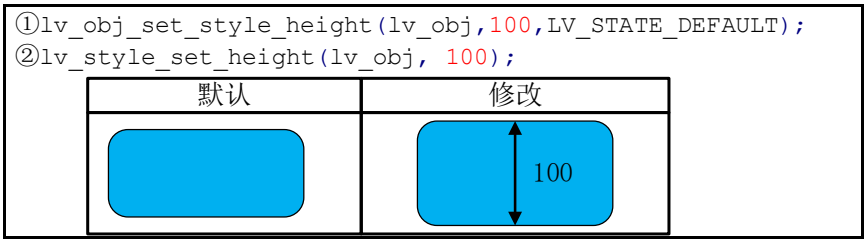


图 6.3.4.1.2 设置高度属性

5. (min_height)最小高度：

设置最小高度，单位可使用像素和百分比。

注意：如果对象已经设置最小高度，则调用 lv_obj_set_height()函数重新设置对象的高度且该高度小于最小高度时，该函数不会得到响应。

6. (max_height)最大高度：

设置最大高度，可以使用像素和百分比值。

注意：如果对象已经设置最大高度，则调用 lv_obj_set_height()函数重新设置对象的高度且该高度大于最大高度时，该函数不会得到响应。

7. X 坐标属性：

设置对象的 X 轴偏移量，单位可使用像素和百分比。如图 6.3.4.1.3 所示：

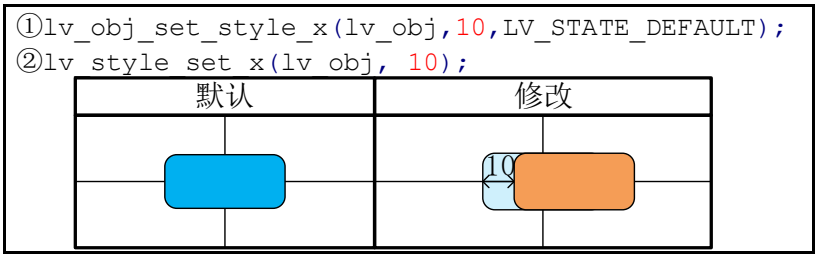


图 6.3.4.1.3 设置 X 坐标属性

8. Y 坐标属性：

设置对象的 Y 坐标，单位可使用像素和百分比，如图 6.3.4.1.4 所示：

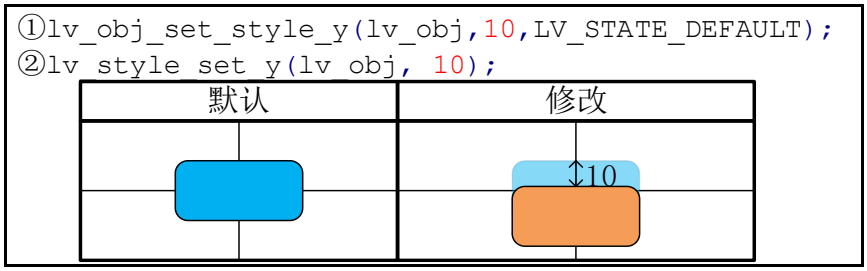


图 6.3.4.1.4 设置 Y 坐标属性

9. (Align)对齐属性：

设置对象对齐属性，如图 6.3.4.1.5 所示：

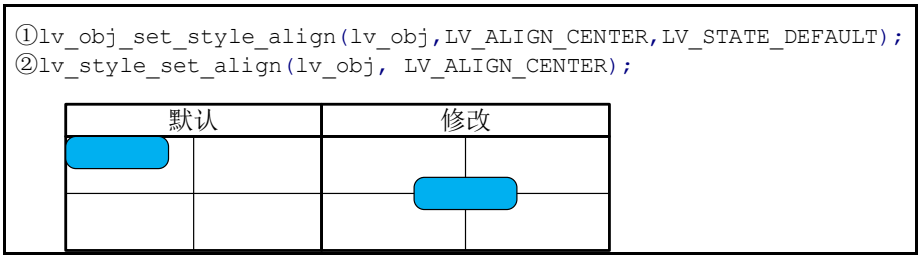


图 6.3.4.1.5 设置对齐属性

该对齐属性的对齐值如下所示：

- LV_ALIGN_DEFAULT。
- LV_ALIGN_TOP_LEFT/MID/RIGHT。
- LV_ALIGN_BOTTOM_LEFT/MID/RIGHT。
- LV_ALIGN_LEFT/RIGHT_MID。

- LV_ALIGN_CENTER。

10. (transform_width) 变换宽度属性：

转变宽度属性能使对象的两边变宽。单位可使用像素和百分比(使用 lv_pct(x))值，如图 6.3.4.1.6 所示：

```
①lv_obj_set_style_transform_width(lv_obj,100,LV_STATE_PRESSED);
②lv_style_set_transform_width(lv_obj, 100);
```

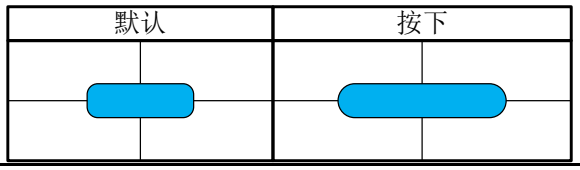


图 6.3.4.1.6 设置转变宽度属性

11. (transform_height) 变换高度属性：

转变高度属性能使对象的两边变高，单位可使用像素和百分比(使用 lv_pct(x))值，如图 6.3.4.1.7 所示：

```
①lv_obj_set_style_transform_height(lv_obj,100,LV_STATE_PRESSED);
②lv_style_set_transform_height(lv_obj, 100);
```

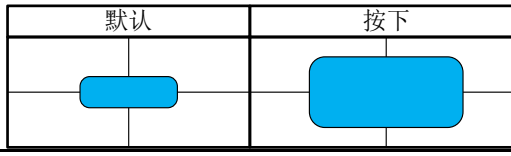


图 6.3.4.1.7 设置转变高度属性

12. (translate_x)偏移 X 坐标属性：

对象在 X 轴方向移动，单位可使用像素和百分比(使用 lv_pct(x))值，如图 6.3.4.1.8 所示：

```
①lv_obj_set_style_translate_x(lv_obj,10,LV_STATE_PRESSED);
②lv_style_set_translate_x(lv_obj, 10);
```

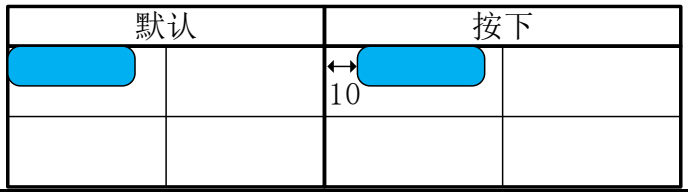


图 6.3.4.1.8 设置转变 X 坐标属性

13. (translate_y) 偏移 Y 坐标属性：

对象 Y 轴方向移动，单位可使用像素和百分比(使用 lv_pct(x))值，如图 6.3.4.1.9 所示：

```
①lv_obj_set_style_translate_x(lv_obj,10,LV_STATE_PRESSED);
②lv_style_set_translate_x(lv_obj, 10);
```

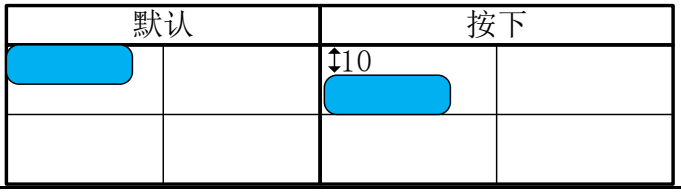


图 6.3.4.1.9 设置转变 Y 坐标属性

14. (transform_zoom)缩放属性：

缩放属性使图像放大和缩小，如果缩放值为 256(LV_IMG_ZOOM_NONE)，则表示图像正常大小；如果缩放值为 128。则表示图像缩小一半；如果缩放值为 512，则表示图像放大一倍，如图 6.3.4.1.10 所示：

```
①lv_obj_set_style_transform_zoom(lv_obj,128,LV_STATE_DEFAULT);
②lv_style_set_transform_zoom(lv_obj, 128);
```

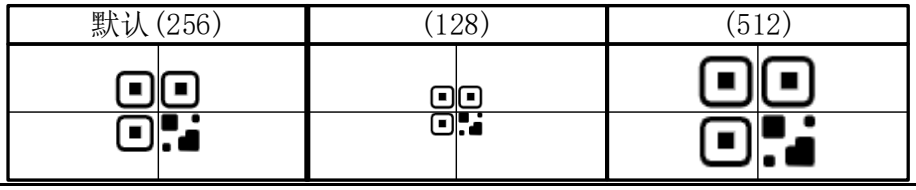


图 6.3.4.1.10 设置缩放属性

15. (transform_angle) 旋转属性：

旋转属性是指对图像在指定方向上旋转某个角度，该旋转值以 0.1 度为单位。例如 45 度=450，如图 6.3.4.1.11 所示：

```
①lv_obj_set_style_transform_angle(lv_obj,450,LV_STATE_DEFAULT);
②lv_style_set_transform_angle(lv_obj, 450);
```

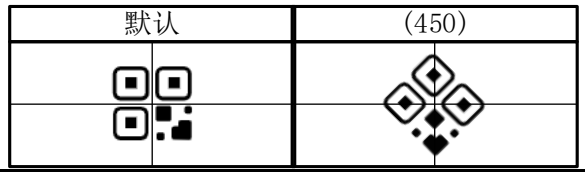


图 6.3.4.1.11 设置旋转属性

6.3.4.2 填充属性

填充属性是用来描述父类与子类之间以及子类之间的间距，这个属性非常类似于 HTML 中的填充属性。

1. (pad_top)顶部填充属性：

顶部的填充。它使内容区域的顶部方向上变小，如图 6.3.4.2.1 所示：

```
①lv_obj_set_style_pad_top(lv_obj,20,LV_STATE_DEFAULT);
②lv_style_set_pad_top(lv_obj, 20);
```

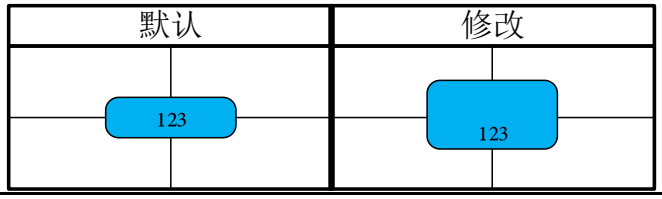


图 6.3.4.2.1 顶部填充

2. (pad_bottom)底部填充属性：

底部的填充，它使内容区域的底部方向上变小，如图 6.3.4.2.2 所示：

```
①lv_obj_set_style_pad_bottom(lv_obj,20,LV_STATE_DEFAULT);
②lv_style_set_pad_bottom(lv_obj, 20);
```

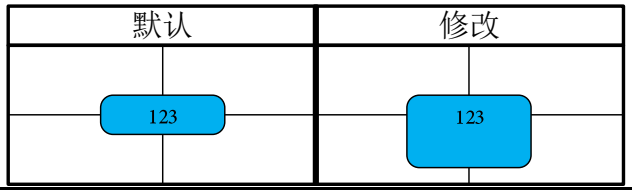


图 6.3.4.2.2 底部填充

3. (pad_left)左边填充属性：

左边的填充，它使内容区域的左边方向上变小，如图 6.2.4.2.3 所示“

```
①lv_obj_set_style_pad_left(lv_obj,20,LV_STATE_DEFAULT);
②lv_style_set_pad_left(lv_obj, 20);
```

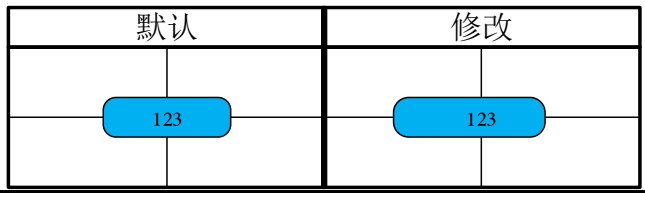


图 6.3.4.2.3 左边填充

4. (pad_right)右边填充属性：

右边的填充，它使内容区域的右边方向上变小，如图 6.3.4.2.4 所示：

```
①lv_obj_set_style_pad_right(lv_obj,20,LV_STATE_DEFAULT);
②lv_style_set_pad_right(lv_obj, 20);
```

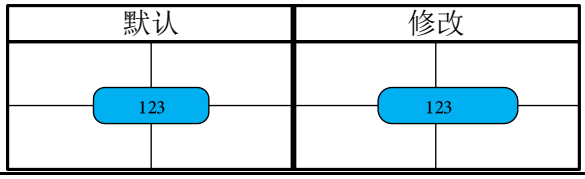


图 6.3.4.2.4 右边填充

5. (pad_row)行之间填充属性：

行之间的填充，由布局使用。

5. (pad_column)列之间填充属性：

列之间的填充，由布局使用。

6.3.4.3 背景属性

该属性来描述对象的背景颜色和图像。

1. (bg_color)背景属性：

设置对象的背景颜色，如图 6.3.4.3.1 所示：

```
①lv_obj_set_style_bg_color(   lv_obj,
                              lv_palette_main(LV_PALETTE_GREEN),
                              LV_STATE_DEFAULT);
②lv_style_set_bg_color(lv_obj, lv_palette_main(LV_PALETTE_GREEN));
```

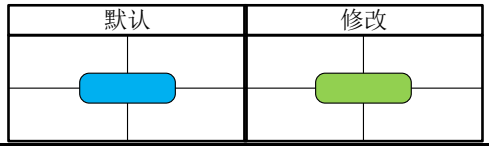


图 6.3.4.3.1 设置背景属性

2. (bg_opa)背景透明属性：

设置背景的不透明度，如果透明值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示对象或者图像不透明；如果透明值为 255、LV_OPA_100 和 LV_OPA_COVER，则表示对象或者图像完全透明；如果透明值为其他值如 LV_OPA_10、LV_OPA_20，则表示对象或者图像半透明，如图 6.3.4.3.2 所示：

```
①lv_obj_set_style_bg_opa(lv_obj,LV_OPA_50,LV_STATE_DEFAULT);
②lv_style_set_bg_opa(lv_obj, LV_OPA_50);
```

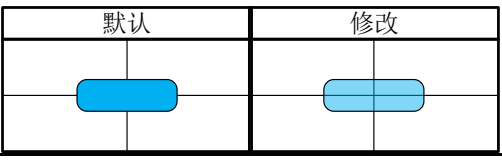


图 6.3.4.3.2 设置背景透明属性

3. (bg_grad_color)背景梯度颜色属性：

设置背景的渐变颜色，当 grad_dir 属性不是设置 LV_GRAD_DIR_NONE 时才有效，如图 6.3.4.3.3 所示：

```
①lv_obj_set_style_bg_grad_color(lv_obj,
                                 lv_palette_main(LV_PALETTE_GREEN),
                                 LV_STATE_DEFAULT);
②lv_style_set_bg_grad_color(lv_obj, lv_palette_main(LV_PALETTE_GREEN));
```

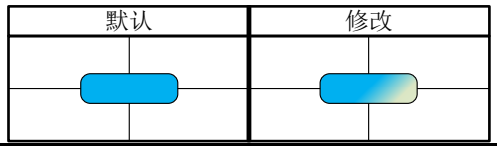


图 6.3.4.3.3 设置背景梯度颜色

4. (bg_grad_dir)背景梯度方向属性：

设置背景的渐变方向，该样式属性设置值为“LV_GRAD_DIR_NONE/HOR/VER”。

5. (bg_grad_stop) 背景梯度起点属性：

设置背景渐变颜色的起始点，如果渐变起始点为 0，则表示渐变是从顶部/左侧开始；如果渐变起始点为 255，则表示渐变是从底部/右侧开始；如果渐变起始点为 128，则表示渐变是从中心开始，如图 6.3.4.3.4 所示：

```
①lv_obj_set_style_bg_grad_stop(lv_obj,
                               value,
                               LV_STATE_DEFAULT);
②lv_style_set_bg_grad_stop(lv_obj, value);
```

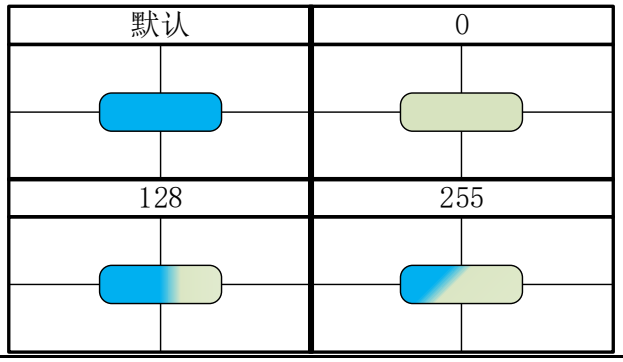


图 6.3.4.3.4 设置背景梯度起点

6. (bg_img_src)背景图片源属性：

设置背景图像源，该图像源可从 lv_img_dsc_t 的指针、文件路径或 LV_SYMBOL_…获取，如图 6.3.4.3.5 所示：

```
①lv_obj_set_style_bg_img_src( lv_obj,&image,LV_STATE_DEFAULT);
②lv_style_set_bg_img_src(lv_obj, &image);
```

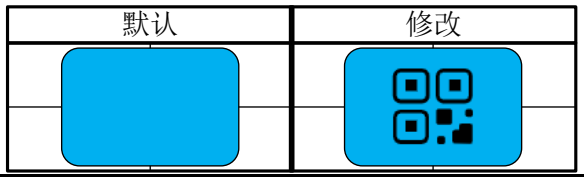


图 6.3.4.3.5 设置背景图片源

7. (bg_img_opa)背景图片透明度属性：

设置背景图像的不透明度，如果透明度数值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示背景图像为不透明；如果透明度数值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示背景图像为完全透明；如果透明度数值为其他值如 LV_OPA_10、LV_OPA_20，则表示背景图像为半透明。如图 6.3.4.3.6 所示：

```
①lv_obj_set_style_bg_img_opa( lv_obj,LV_OPA_100,LV_STATE_DEFAULT);
②lv_style_set_bg_img_opa(lv_obj, LV_OPA_100);
```

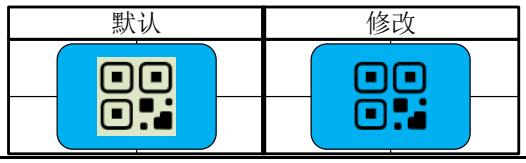


图 6.3.4.3.6 设置背景图片透明

8. (bg_img_recolor)背景图片重绘属性：

设置一个颜色混合到背景图像。

9. (bg_img_recolor_opa) 背景图片重绘透明属性：

设置背景图像重着色的强度，如果重绘透明值为0、LV_OPA_0和LV_OPA_TRANSP，则表示背景图像着色为不混合；如果重绘透明值为 255、LV_OPA_100 和 LV_OPA_COVER，则表示背景图像着色为完全重新着色；如果重绘透明值为其他值如 LV_OPA_10, LV_OPA_20，则背景图像着色按比例解析。如下源码所示：

```
/* 创建图片部件 */
lv_app_img = lv_img_create(lv_scr_act());
/* 设置图片源 */
lv_img_set_src(lv_app_img, &image);
/* 居中 */
lv_obj_center(lv_app_img);
/* 设置图像重着色的强度 */
lv_obj_set_style_bg_img_recolor_opa(lv_app_img, 255, LV_PART_MAIN);
/* 设置一个颜色混合到背景图像 */
lv_obj_set_style_bg_img_recolor( lv_app_img,
lv_color_make(255, 255, 255),
LV_STATE_DEFAULT);
```

使用上述源码可以重绘图片的背景颜色。

10. (bg_img_tiled)背景图片平铺属性：

如果该属性启用，就背景图像将被平铺。如图 6.3.4.3.7 所示：

```
①lv_obj_set_style_bg_img_tiled(lv_obj,true,LV_STATE_DEFAULT);
②lv_style_set_bg_img_tiled(lv_obj, true);
```

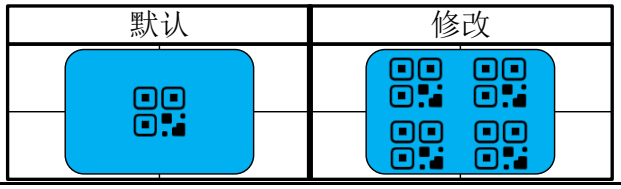


图 6.3.4.3.7 设置背景图片平铺

6.3.4.4 边框属性

该属性主要描述对象的边框。

1. (border_color)边框颜色：

设置边框的颜色属性，如图 6.3.4.4.1 所示：

```
①lv_obj_set_style_border_color( lv_obj,
                                lv_palette_main(LV_PALETTE_GREEN),
                                LV_STATE_DEFAULT);
②lv_style_set_border_color(lv_obj,lv_palette_main(LV_PALETTE_GREEN));
```

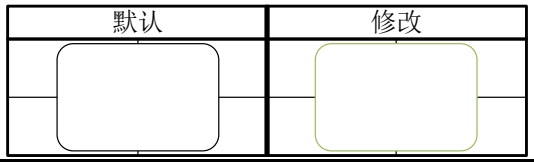


图 6.3.4.4.1 设置边框颜色

2. (border_opa)边框透明度：

设置边框的不透明度，如果透明值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示对象边框为不透明，如果透明值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示对象边框为完全透明；如果透明值为其他值如 LV_OPA_10、LV_OPA_20，则表示对象边框为半透明。如图 6.3.4.4.2 所示：

```
①lv_obj_set_style_border_opa( lv_obj,LV_OPA_20,LV_STATE_DEFAULT);
②lv_style_set_border_opa(lv_obj,LV_OPA_20);
```

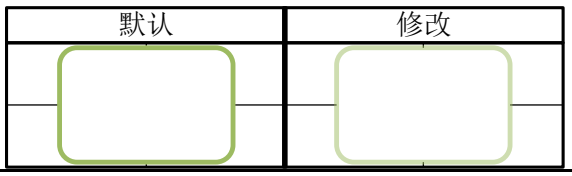


图 6.3.4.4.2 设置边框透明度

3. (border_width)边框宽度：

设置边框的宽度，注意：该样式属性只能使用像素值来设置，如图 6.3.4.4.3 所示：

```
①lv_obj_set_style_border_width( lv_obj,10,LV_STATE_DEFAULT);
②lv_style_set_border_width(lv_obj,10);
```

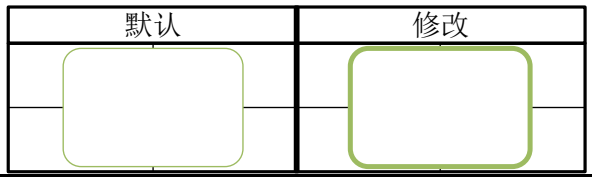


图 6.3.4.4.3 设置边框宽度

4. (border_side)边框内侧：

该属性是指绘制边界内边，形参值可能为：

① LV_BORDER_SIDE_NONE。
② LV_BORDER_SIDE_TOP。
③ LV_BORDER_SIDE_BOTTOM。
④ LV_BORDER_SIDE_LEFT。
⑤ LV_BORDER_SIDE_RIGHT。
⑥ LV_BORDER_SIDE_INTERNAL。

这些形参值可以组成传入该属性函数，如图 6.3.4.4.4 所示：

```
①lv_obj_set_style_border_side( lv_obj,
                               LV_BORDER_SIDE_TOP,
                               LV_STATE_DEFAULT);
②lv_style_set_border_side(lv_obj,LV_BORDER_SIDE_TOP);
```

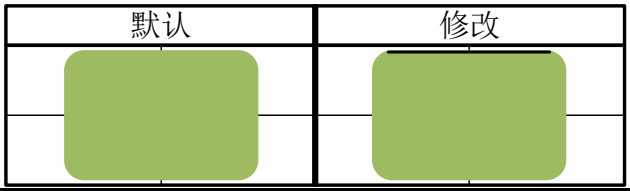


图 6.3.4.4.4 设置边框绘制方向

5. (border_post)边框绘制：

该属性是指设置对象边框的绘画时间，如果参数值为 true，则等子类绘画完，之后绘画对象的边框；如果为 flase，则首先绘画对象边框，之后绘画子类。

6.3.4.5 轮廓属性

该属性主要用来描述对象的轮廓，它就像一个边界一样，但该轮廓画在对象之外，如图 6.3.4.5.1 所示：

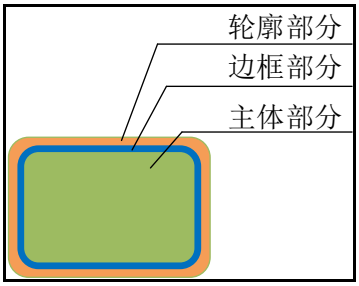


图 6.3.4.5.1 轮廓的位置

1. (outline_width) 轮廓宽度属性：

设置轮廓的宽度(以像素为单位)，如图 6.3.4.5.2 所示：

```
①lv_obj_set_style_outline_width( lv_obj,
                                 10,
                                 LV_STATE_DEFAULT);
②lv_style_set_outline_width(lv_obj,10);
```

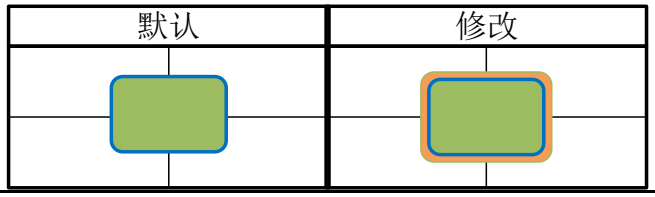


图 6.3.4.5.2 设置轮廓宽度

2. (outline_color) 轮廓颜色属性：

设置轮廓的颜色，如图 6.3.4.5.3 所示：

```
①lv_obj_set_style_outline_color( lv_obj,
                                 color_value,
                                 LV_STATE_DEFAULT);
②lv_style_set_outline_color(lv_obj,color_value);
```

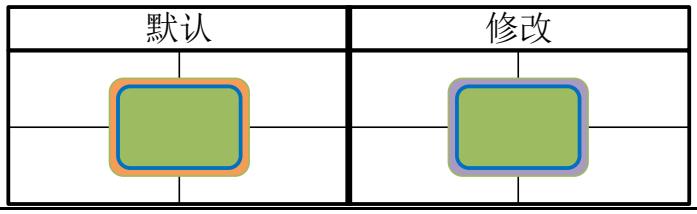


图 6.3.4.5.3 设置轮廓颜色

3. (outline_opa) 轮廓透明度属性：

设置轮廓的不透明度，如果轮廓透明度值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示对象轮廓为不透明；如果轮廓透明度值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示对

象轮廓为完全透明；如果轮廓透明度值为其他值如 LV_OPA_10、LV_OPA_20，则表示对象轮廓为半透明，如图 6.3.4.5.4 所示：

```
①lv_obj_set_style_outline_opa( lv_obj,
                               LV_OPA_20,
                               LV_STATE_DEFAULT);
②lv_style_set_outline_opa(lv_obj,LV_OPA_20);
```

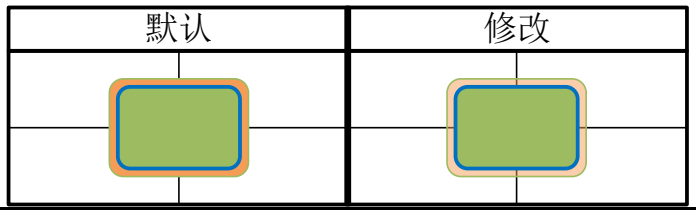


图 6.3.4.5.4 设置轮廓透明度

4. (outline_pad) 轮廓间隙属性：

设置轮廓线的填充，即对象和轮廓线之间的间隙，如图 6.3.4.5.5 所示：

```
①lv_obj_set_style_outline_pad( lv_obj,
                               10,
                               LV_STATE_DEFAULT);
②lv_style_set_outline_pad(lv_obj,10);
```

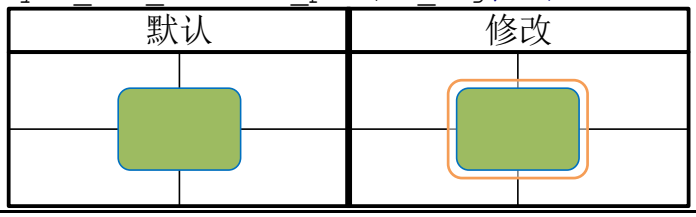


图 6.3.4.5.5 设置轮廓间隙

6.3.4.6 阴影属性

该属性主要描述对象的阴影。

1. (shadow_width) 阴影宽度属性：

以像素为单位设置阴影的宽度，注意：阴影宽度取值>= 0，如图 6.3.4.6.1 所示：

```
①lv_obj_set_style_shadow_width( lv_obj,
                                10,
                                LV_STATE_DEFAULT);
②lv_style_set_shadow_width(lv_obj,10);
```

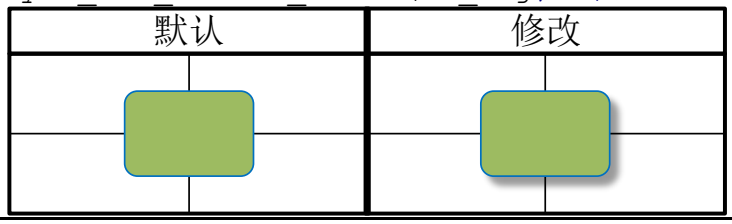


图 6.3.4.6.1 设置阴影宽度

2. (shadow_ofs_x) 阴影偏移 X 轴方向属性：

设置阴影在 X 方向上的像素偏移量，如图 6.3.4.6.2 所示：

```
①lv_obj_set_style_shadow_ofs_x( lv_obj,
                                10,
                                LV_STATE_DEFAULT);
②lv_style_set_shadow_ofs_x(lv_obj,10);
```

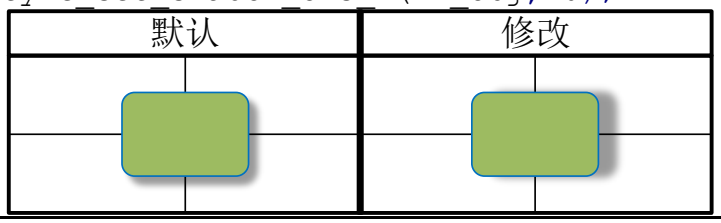


图 6.3.4.6.2 设置阴影 X 方向偏移

3. (shadow_ofs_y) 阴影偏移 Y 轴方向属性：

设置阴影 Y 方向的像素偏移量，如图 6.3.4.6.3 所示：

```
①lv_obj_set_style_shadow_ofs_y( lv_obj,
                                10,
                                LV_STATE_DEFAULT);
②lv_style_set_shadow_ofs_y(lv_obj,10);
```

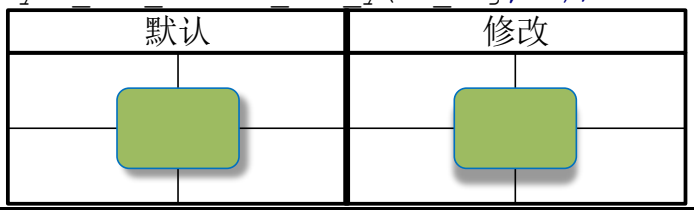


图 6.3.4.6.3 设置阴影 Y 方向偏移

4. (shadow_spread) 阴影范围属性：

对象的阴影计算较大或较小的主体为基础，如图 6.3.4.6.4 所示：

```
①lv_obj_set_style_shadow_spread( lv_obj,
                                 10,
                                 LV_STATE_DEFAULT);
②lv_style_set_shadow_spread(lv_obj,10);
```

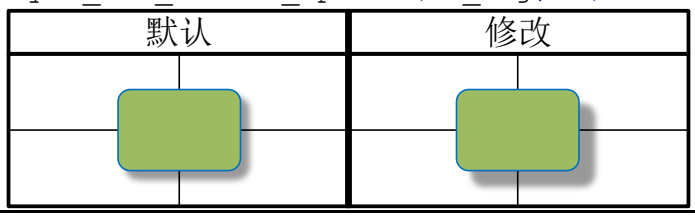


图 6.3.4.6.4 设置阴影范围

5. (shadow_color) 阴影颜色属性：

设置阴影的颜色，如图 6.3.4.6.5 所示：

```
①lv_obj_set_style_shadow_color( lv_obj,
                                 red,
                                 LV_STATE_DEFAULT);
②lv_style_set_shadow_color(lv_obj,red);
```

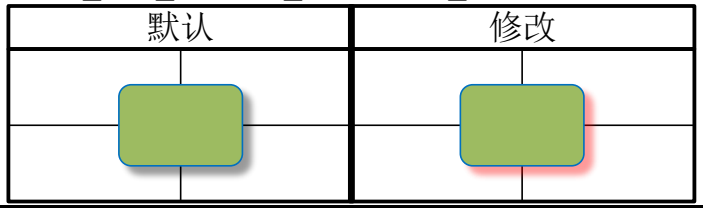


图 6.3.4.6.5 设置阴影颜色

6. (shadow_opa) 阴影透明属性：

设置阴影的不透明度，如果阴影透明值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示对象阴影为不透明；如果阴影透明值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示对象阴影为完全透明；如果阴影透明值为其他值如 LV_OPA_10、LV_OPA_20，则表示对象阴影为半透明，如图 6.3.4.6.6 所示：

```
①lv_obj_set_style_shadow_opa( lv_obj,
                               LV_OPA_20,
                               LV_STATE_DEFAULT);
②lv_style_set_shadow_opa(lv_obj,LV_OPA_20);
```



图 6.3.4.6.6 设置阴影透明度

6.3.4.7 图片属性

该属性主要描述图像部件，其他部件不适合使用该属性。

1. (img_opa)图片透明度属性：

该属性主要设置图像的不透明度，如果图片不透明度值为 0、LV_OPA_0 或 LV_OPA_TRANS，则表示图像部件为不透明；如果图片不透明度值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示图像部件为完全透明；如果图片不透明度值为其他值如 LV_OPA_10、LV_OPA_20，则表示图像部件为半透明，如图 6.3.4.7.1 所示：

```
①lv_obj_set_style_img_opa( lv_obj,
                            LV_OPA_20,
                            LV_STATE_DEFAULT);
②lv_style_set_img_opa(lv_obj,LV_OPA_20);
```

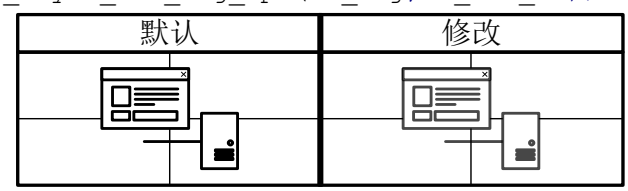


图 6.3.4.7.1 设置图片透明度

2. (img_recolor)图片重绘属性：

该属性主要设置颜色混合到图像。

3. (img_recolor_opa)图片重绘透明属性：

该属性主要设置颜色混合的强度，如果图片重绘透明值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示图片对象为不透明；如果图片重绘透明值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示图片对象为完全透明，如果图片重绘透明值为其他值如 LV_OPA_10、LV_OPA_20 则表示图片对象为半透明。如下源码所示：

```
/* 创建图片部件 */
lv_app_img = lv_img_create(lv_scr_act());
/* 设置图片源 */
lv_img_set_src(lv_app_img, &image);
/* 居中 */
lv_obj_center(lv_app_img);
/* 设置图像重着色的强度 */
lv_obj_set_style_img_recolor_opa(lv_app_img, 255, LV_PART_MAIN);
/* 设置一个颜色混合到背景图像 */
lv_obj_set_style_img_recolor( lv_app_img, lv_color_make(255, 255, 255),
                              LV_STATE_DEFAULT);
```

上述源码可知：我们把图像透明度设置为全透明，然后重新着色成白色，如图 6.3.4.7.2 所示：

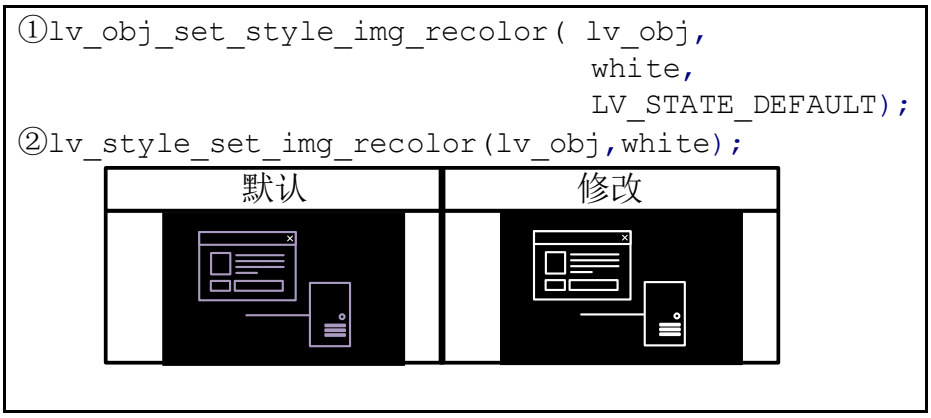


图 6.3.4.7.2 图像重新着色

6.3.4.8 线属性

该属性主要描述线部件，其他部件不适合使用该属性。

1. (line_width)线的宽度属性：

该属性是以像素为单位设置线条的宽度，如图 6.3.4.8.1 所示：

```
①lv_obj_set_style_line_width(lv_obj,
                             value,
                             LV_STATE_DEFAULT);
②lv_style_set_line_width(lv_obj, value);
```

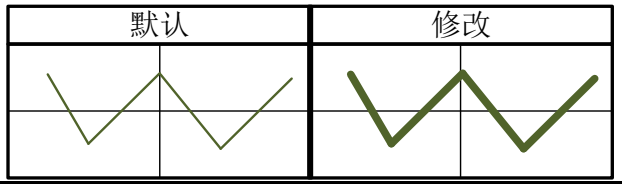


图 6.3.4.8.1 设置线部件宽度

2. (line_rounded)线的端点属性：

该属性主要设置线的端点形状，如果传入的参数为True，则线的端点为圆角；如果传入的参数为 false，则线的端点为垂线。如图 6.3.4.8.2 所示：

```
①lv_obj_set_style_line_rounded(lv_obj,
                               true,
                               LV_STATE_DEFAULT);
②lv_style_set_line_rounded(lv_obj, true);
```

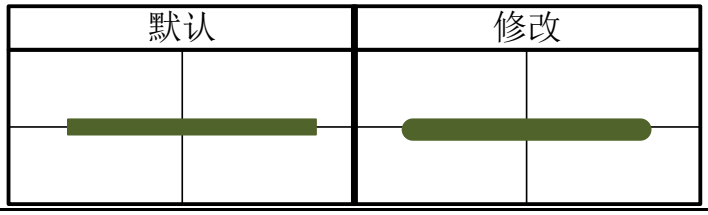


图 6.3.4.8.2 设置线部件端点

3. (line_color)线的颜色属性：

该属性主要设置线条的颜色，如图 6.3.4.8.3 所示：

```
①lv_obj_set_style_line_color(lv_obj,
                             vlaue,
                             LV_STATE_DEFAULT);
②lv_style_set_line_color(lv_obj, vlaue);
```

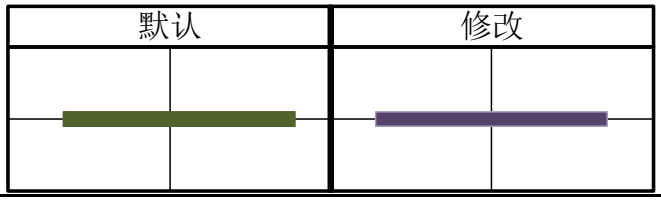


图 6.3.4.8.3 设置 线的颜色

4. (line_opa)线的透明度属性：

该属性主要设置线条的不透明度，如图 6.3.4.8.4 所示：

```
①lv_obj_set_style_line_opa(lv_obj,
                           LV_OPA_20,
                           LV_STATE_DEFAULT);
②lv_style_set_line_opa(lv_obj, LV_OPA_20);
```

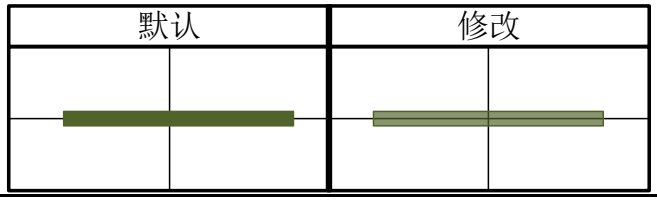


图 6.3.4.8.3 设置线的透明度

6.3.4.9 圆弧属性

该属性主要描述圆弧(arc)部件，其他部件不适合使用该属性。

1. (arc_width)圆弧宽度属性：

设置弧的宽度(厚度)，该属性的参数是以像素为单位，如图 6.3.4.9.1 所示：

```
①lv_obj_set_style_arc_width(lv_obj,
                            20,
                            LV_STATE_DEFAULT);
②lv_style_set_arc_width(lv_obj, 20);
```

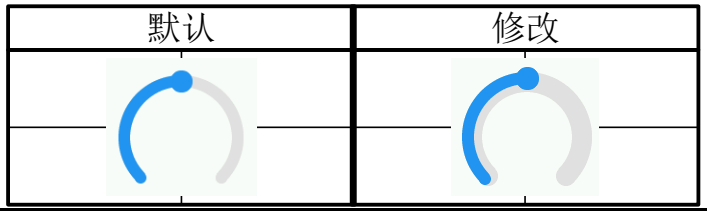


图 6.3.4.9.1 设置圆弧宽度

2. (arc_rounded)圆弧端点：

该属性主要设置圆弧端点形状，如果传入的参数为True，则线的端点为圆角；如果传入的参数为 false，则线的端点为垂线。如图 6.3.4.9.2 所示：

```
①lv_obj_set_style_arc_rounded(lv_obj,
                              false,
                              LV_STATE_DEFAULT);
②lv_style_set_arc_rounded(lv_obj, false);
```

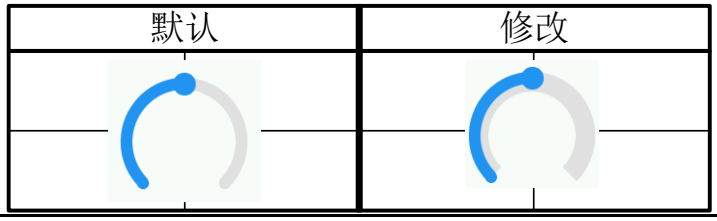


图 6.3.4.9.2 设置圆弧末端

3. (arc_color)圆弧颜色属性：

设置圆弧的颜色，如图 6.3.4.9.3 所示：

```
①lv_obj_set_style_arc_color(lv_obj,
                            color_value,
                            LV_STATE_DEFAULT);
②lv_style_set_arc_color(lv_obj, Color_value);
```

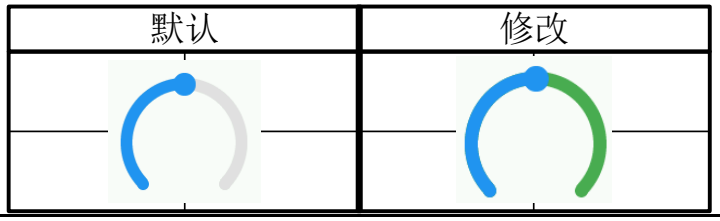


图 6.3.4.9.3 设置圆弧颜色

4. (arc_opa)圆弧透明度属性：

设置圆弧的透明度，如图 6.3.4.9.4 所示：

```
①lv_obj_set_style_arc_opa(lv_obj,
                          LV_OPA_20,
                          LV_STATE_DEFAULT);
②lv_style_set_arc_opa(lv_obj, LV_OPA_20);
```

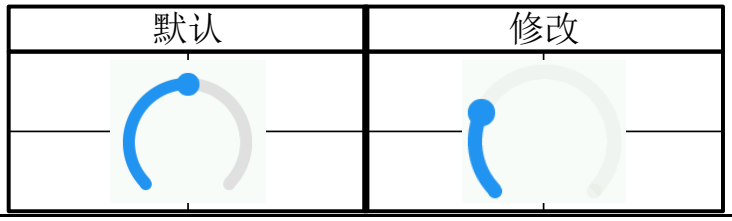


图 6.3.4.9.4 设置圆弧透明度

6.3.4.10 文本属性

该属性主要描述文本的属性。

1. (text_color)文本颜色属性：

设置文本的颜色，如图 6.3.4.10.1 所示：

```
①lv_obj_set_style_text_color(lv_obj,
                             color_value,
                             LV_STATE_DEFAULT);
②lv_style_set_text_color(lv_obj, Color_value);
```

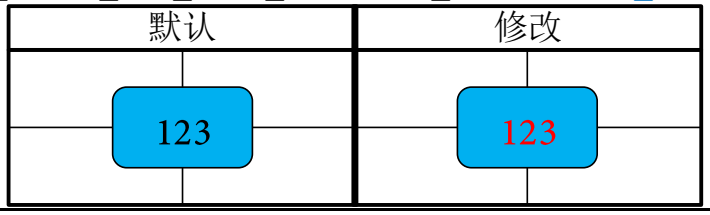


图 6.3.4.10.1 设置文本颜色

2. (text_opa)文本透明度属性：

该属性主要设置文本的不透明度。如果文本的不透明度的值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示该文本为不透明；如果文本的不透明度的值为 255、LV_OPA_100

或 LV_OPA_COVER，则表示该文本为完全透明；如果文本的不透明度的值为其他值如 LV_OPA_10、LV_OPA_20，则表示该文本为半透明，如图 6.3.4.10.2 所示：

```
①lv_obj_set_style_text_opa(lv_obj,
                           LV_OPA_20,
                           LV_STATE_DEFAULT);
②lv_style_set_text_opa(lv_obj, LV_OPA_20);
```

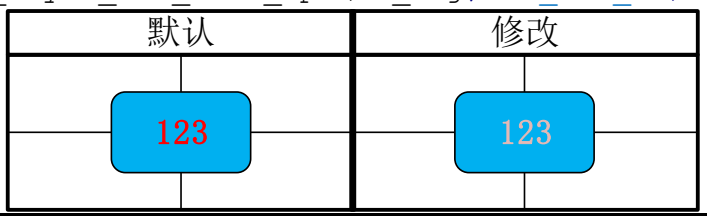


图 6.3.4.10.2 设置文本透明度

3. (text_font)文本字体属性：

该属性主要设置文本的字体，如图 6.3.4.10.3 所示：

```
①lv_obj_set_style_text_font(lv_obj,
                            &font_xx,
                            LV_STATE_DEFAULT);
②lv_style_set_text_font(lv_obj, &font_xx);
```

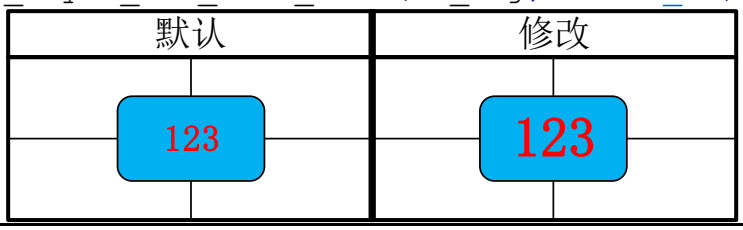


图 6.3.4.10.3 设置文本字体

4. (text_letter_space)文本字符空间属性：

设置文本的字符间的缩进，如图 6.3.4.10.4 所示：

```
①lv_obj_set_style_text_letter_space(lv_obj,
                                    5,
                                    LV_STATE_DEFAULT);
②lv_style_set_text_letter_space(lv_obj, 5);
```



图 6.3.4.10.4 设置文本字符空间

5. (text_decor)文本装饰属性：

该属性主要为文本内容装饰风格，该属性的传入形参如下所示：

① LV_TEXT_DECOR_NONE：正常文本。

② LV_TEXT_DECOR_UNDERLINE：文件添加下划线

③ LV_TEXT_DECOR_STRIKETHROUGH：文本添加删除线。

这些宏定义可组合形式传入该属性函数参数中，如图 6.3.4.10.5 所示：

```
①lv_obj_set_style_text_decor(lv_obj,
                             LV_TEXT_DECOR_UNDERLINE,
                             LV_STATE_DEFAULT);
②lv_style_set_text_decor(lv_obj,
                         LV_TEXT_DECOR_UNDERLINE);
```

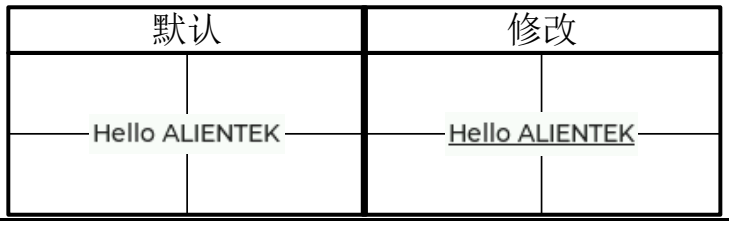


图 6.3.4.10.5 设置文本装饰

6. (text_align)文本对齐属性：

设置对象内的文本对齐。该属性函数所传入的参数如下：

① LV_TEXT_ALIGN_LEFT：对象内文本左对齐。

② LV_TEXT_ALIGN_CENTER：对象内文本中间对齐。

③ LV_TEXT_ALIGN_RIGHT：对象内文本右对齐。

④ LV_TEXT_ALIGN_AUTO：对象内文本自动对齐。

6.3.4.11 其他属性

1. (radius)设置圆角的半径属性：

设置对象边框的圆角的半径，如果圆角的半径为 0，则对象边框圆角为直角型；如果圆角的半径大于 0 且小于等于 LV_RADIUS_CIRCLE，则对象边框圆角为圆角型(圆角的半径根据设定值相关)，如图 6.3.4.11.1 所示：

```
①lv_obj_set_style_radius(lv_obj,
                         0,
                         LV_STATE_DEFAULT);
②lv_style_set_radius(lv_obj, 0);
```

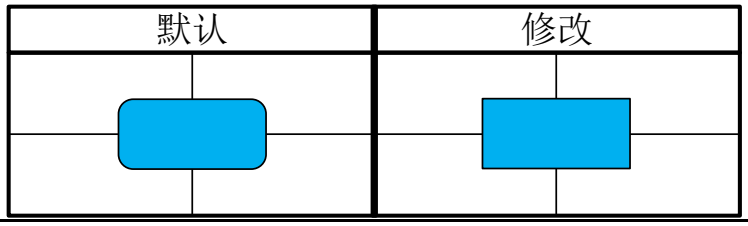


图 6.3.4.11.1 设置对象圆角

2. (opa)透明度：

设置对象的透明度，如果对象的透明度的值为 0、LV_OPA_0 或 LV_OPA_TRANSP，则表示该对象为不透明；如果对象的透明度的值为 255、LV_OPA_100 或 LV_OPA_COVER，则表示该对象为完全透明；如果对象的透明度的值为其他值如 LV_OPA_10、LV_OPA_20，则表示该对象为半透明，图 6.3.4.11.2 所示：

```
①lv_obj_set_style_opa(lv_obj,
                      LV_OPA_20,
                      LV_STATE_DEFAULT);
②lv_style_set_opa(lv_obj, LV_OPA_20);
```

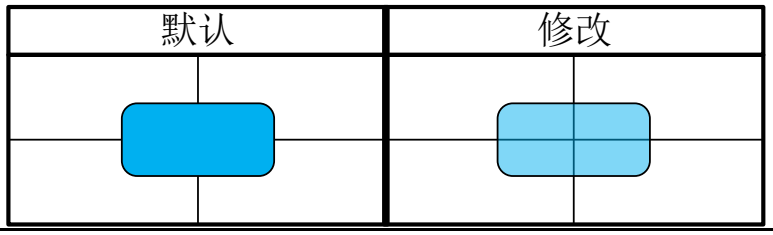


图 6.3.4.11.2 设置对象透明度

3. (anim)动画属性：

使对象具有动画效果。

4. (anim_time)动画时间属性：

设置动画时间，该动画时间是以毫秒为单位。

5. (anim_speed)动画速度属性：

设置动画速度。该动画速度是以像素/秒为单位。例如标签的滚动速度。

6. (layout)布局属性：

设置对象的布局。子元素将根据为布局设置的策略重新定位和调整大小。

7. (base_dir)对象方向属性：

设置对象的基本方向。取值为:LV_BIDI_DIR_LTR/RTL/AUTO。

6.4 LVGL 滚动属性

在 LVGL 中，滚动效果是非常直观，如果一个对象在其父内容区域之外，则该父对象变为可滚动并且出现滚动条，如图 6.4.1 所示：

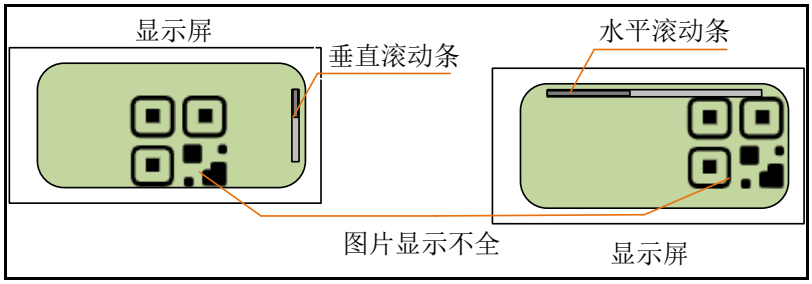


图 6.4.1 滚动条的创建

在 LVGL 中，任何对象都可以滚动，包括 lv_obj_t, lv_img, lv_btn,lv_meter 等部件。

6.4.1 滚动的类型：

从上图可知：LVGL 只支持水平和垂直滚动但它不支持对角滚动方式。

6.4.2 滚动条模式：

默认情况下，子类超出父类的范围区域，则该父类对象会自动开启滚动条。如果读者想使对象不开启滚动条，则使用函数 lv_obj_set_scrollbar_mode 设置滚动条模式，这些模式定义如表 6.4.2.1 所示：

滚动模式	描述
LV_SCROLLBAR_MODE_OFF	从不显示滚动条
LV_SCROLLBAR_MODE_ON	始终显示滚动条
LV_SCROLLBAR_MODE_ACTIVE	滚动对象时显示滚动条
LV_SCROLLBAR_MODE_AUTO	当内容大到可以滚动时显示滚动条

表 6.4.2.1 部件的滚动模式

lv_obj_set_scrollbar_mode 函数

设置滚动模式，其函数原型如以下所示：

```
void lv_obj_set_scrollbar_mode(lv_obj_t * obj, lv_scrollbar_mode_t mode)
```

该函数的形参，如表 6.4.2.1 所示：

参数	描述	
obj	设置的对象	
mode	滚动模式	
	LV_SCROLLBAR_MODE_OFF	从不显示滚动条
	LV_SCROLLBAR_MODE_ON	始终显示滚动条
	LV_SCROLLBAR_MODE_ACTIVE	滚动对象时显示滚动条
	LV_SCROLLBAR_MODE_AUTO	当内容大到可以滚动时显示滚动条

表 6.4.2.1 函数 lv_obj_set_scrollbar_mode()形参描述

前面大家已经了解了部件组成部分，小遍相信大家对它有一定的认识，其实滚动条可以归类于部件组成部分，由于部件的滚动部分可有可无，所以 LVGL 把滚动条部分独立归类。例如设置滚动条(LV_PART_SCROLLBAR)颜色，可以使用以下源码设置：

```
static lv_style_t style_red;
lv_style_init(&style_red);
/* 设置背景颜色 */
lv_style_set_bg_color(&style_red, lv_color_red());
/* 设置滚动条部分 */
lv_obj_add_style(obj, &style_red, LV_STATE_SCROLLED |LV_PART_SCROLLBAR);
```

上述源码可知：当对象滚动时会进入 LV_STATE_SCROLLED 状态，该状态会判断是否触发滚动，如果滚动状态被响应，则滚动条绘制成红色。

6.4.3 滚动条的回调事件

LV_EVENT_SCROLL_BEGIN：滚动开始。
LV_EVENT_SCROLL_END：滚动结束。
LV_EVENT_SCROLL：滚动发生了，每次位置变化时触发。
这些滚动的事件可以在对象回调函数中定义，这些事件主要判断滚动的期间。

6.4.4 控制滚动条

控制滚动条是指滚动的方向可设定，设定方向状态如下所示：

① LV_DIR_TOP：只向上滚。
② LV_DIR_LEFT：只向左滚动。
③ LV_DIR_BOTTOM：只向下滚动。
④ LV_DIR_RIGHT：只向右滚动。
⑤ LV_DIR_HOR：仅水平滚动。
⑥ LV_DIR_VER：仅垂直滚动。

⑦ LV_DIR_ALL：滚动任何方向。

这些状态可调用函数 lv_obj_set_scroll_dir 设置对象的滚动条方向，值得注意的是：这些状态允许组合的形式传入函数。

1. lv_obj_set_scroll_dir 函数

设置滚动方向，其函数原型如下所示：

```
void lv_obj_set_scroll_dir(lv_obj_t * obj, lv_dir_t dir)
```

该函数的形参，如表 6.4.4.1 所示：

参数	描述	
obj	目标对象	
dir	滚动方向	
	LV_DIR_TOP	只向上滚
	LV_DIR_LEFT	只向左滚动
	LV_DIR_BOTTOM	只向下滚动
	LV_DIR_RIGHT	只向右滚动
	LV_DIR_HOR	仅水平滚动
	LV_DIR_VER	仅垂直滚动
	LV_DIR_ALL	滚动任何方向

表 6.4.4.1 函数 lv_obj_set_scroll_dir()形参描述

如果用户不需要对象滚动具有滚动效果，请调用函数 lv_obj_clear_flag()清除滚动状态。

2. lv_obj_clear_flag 函数

清除状态，其函数原型如下所示：

```
void lv_obj_clear_flag(lv_obj_t * obj, lv_obj_flag_t f)
```

该函数的形参，如表 6.4.4.2 所示：

参数	描述
obj	目标对象
f	清除的状态

表 6.4.4.2 函数 lv_obj_clear_flag()形参描述

返回值：无。

lv_obj_flag_t结构体在lv_obj.h文件重定义，由于篇幅太大，这里笔者就不一一列举出来。

6.4.5 滚动条其他特性

1. 滚动链：

如果一个对象不能被进一步滚动（例如它的内容已经到达最底部的位置），则它额外的滚动被传播到它的父对象中；如果父级可以向这个方向滚动，则父类对象将被滚动。注意：它也继续传播给祖父类，由此类推，形成了滚动链。

滚动传播称为“滚动链接”，可以使用 LV_OBJ_FLAG_SCROLL_CHAIN_HOR/VER 标志启用和禁用。如果禁用链接，则对象上的传播将被停止，并且不会滚动父对象。

2. 滚动步量：

当用户滚动一个对象并释放它时，LVGL 可模拟滚动的惯性动量，这就像物体被抛出并且滚动平稳地减慢。滚动步量可使用 LV_OBJ_FLAG_SCROLL_MOMENTUM 标志启用和禁用滚动步量。

3. 弹性滚动：

通常一个对象不能滚动超过其内容的极限，即内容的顶部不能低于对象的顶部。但是，LV_OBJ_FLAG_SCROLL_ELASTIC 当用户“过度滚动”内容时会添加一种奇特的效果。滚动变慢，内容可以在对象内部滚动，当对象被释放时，其中滚动的内容将被动画化回到有效位置。

4. 滚动捕捉对象：

滚动捕捉对象是指滚动结束时，可根据特定规则捕捉对象的子对象。这些子类可以用 LV_OBJ_FLAG_SNAPPABLE 标志单独捕捉。对象可通过以下四种方式捕捉对象：

① LV_SCROLL_SNAP_NONE：捕捉被禁用（默认）。
② LV_SCROLL_SNAP_START：将子项与滚动对象的左侧/顶部对齐。
③ LV_SCROLL_SNAP_END：将子项与滚动对象的右侧/底部对齐。
④ LV_SCROLL_SNAP_CENTER：将子项与滚动对象的中心对齐。

这些状态可调用函数 lv_obj_set_scroll_snap_x/y 设置捕捉子项。

lv_obj_set_scroll_snap_x/y 函数

设置在水平/垂直滚动结束时捕捉子项的位置，其函数原型如以下所示：

```
void lv_obj_set_scroll_snap_x/y(struct _lv_obj_t * obj,
                                lv_scroll_snap_t align);
```

该函数的形参，如表 6.4.5.1 所示：

参数	描述	
obj	目标对象	
align	清除的状态	
	LV_SCROLL_SNAP_NONE	捕捉被禁用。（默认）
	LV_SCROLL_SNAP_START	将子项与滚动对象的左侧/顶部对齐
	LV_SCROLL_SNAP_END	将子项与滚动对象的右侧/底部对齐
	LV_SCROLL_SNAP_CENTER	将子项与滚动对象的中心对齐

表 6.4.5.1 函数 lv_obj_set_scroll_snap_x/y()形参描述

返回值：无。

6.5 LVGL 动画属性

动画是指将对象的连续动作、变化的画面、以一定的速度放映、给人以动态和变化的印象。LVGL 可使用动画在开始值和结束值之间自动更改变量的值，这动画将通过定期调用动画回调。函数刷新变量的值，该函数通用原型如下：

```
void func(void * var, lv_anim_var_t value);
```

6.5.1 创建动画

LVGL 要创建一个动画，则必须使用初始化动画和配置动画特性。LVGL 设置动画分为两个部分，一个是必须设置，另一个为可选设置。

1. 必须设置：

```
/* 初始化一个动画
 *-----------------------*/
lv_anim_t a;
lv_anim_init(&a);
/* 强制设置
 *------------------*/
/* 设置“动画”函数 */
lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t) lv_obj_set_x);
```

```
/* 设置动画的目标 */
lv_anim_set_var(&a, obj);
/* 动画长度[ms] */
lv_anim_set_time(&a, duration);
/* 设置起始值和结束值。例如 0,150 */
lv_anim_set_values(&a, start, end);
/* 开始动画
 *------------------*/
lv_anim_start(&a);
```

从上述源码可知：我们知道 LVGL 创建一个动画的流程如下：

① 定义一个 lv_anim_t 变量。

② 调用函数 lv_anim_init 初始化动画。

③ 调用函数 lv_anim_set_exec_cb 设置动画回调函数。

④ 调用函数 lv_anim_set_var 设置动画执行的目标。

⑤ 调用函数 lv_anim_set_time 设置动画时间长度。

⑥ 调用函数 lv_anim_set_values 设置起始值和结束值。

⑦ 调用函数 lv_anim_start 开始动画。

2. 可选设置：

```
/* 可选设置
 *------------------*/


/* 启动动画前的等待时间[ms] */
lv_anim_set_delay(&a, delay);
/* 设置路径(曲线)。默认是线性的 */
lv_anim_set_path(&a, lv_anim_path_ease_in);
/* 设置一个回调函数来指示动画何时准备好(空闲) */
lv_anim_set_ready_cb(&a, ready_cb);
/* 设置一个回调函数来指示动画何时启动(延迟之后) */
lv_anim_set_start_cb(&a, start_cb);
/* 准备好后，按此持续时间倒放动画。默认为 0(禁用)[ms] */
lv_anim_set_playback_time(&a, time);
/* 延迟在回放。默认为 0(禁用)[ms] */
lv_anim_set_playback_delay(&a, delay);
/* 重复的数量。默认值为 1。LV_ANIM_REPEAT_INFINITE 用于无限重复 */
lv_anim_set_repeat_count(&a, cnt);
/* 重复前延迟。默认为 0(禁用)[ms] */
lv_anim_set_repeat_delay(&a, delay);
/* True(默认):立即应用起始值，false:延迟后应用起始值。真正的开始  */
lv_anim_set_early_apply(&a, true/false);
```

注意：动画可选设置函数调用之前，必须先写动画必须设置的函数。

6.5.2 设置动画路径

LVGL 可控制动画的路径。最简单的情况是线性的，这意味着开始和结束之间的当前值以固定的步长变化，该路径是一个函数，它根据动画的当前状态计算要设置的下一个值。目前，LVGL 有以下内置路径函数：

① lv_anim_path_linear：线性动画。
② lv_anim_path_step：最后一步改变。
③ lv_anim_path_ease_in：一开始很慢。
④ lv_anim_path_ease_out：最后慢。
⑤ lv_anim_path_ease_in_out：开始和结束都很慢。
⑥ lv_anim_path_overshoot：超出最终值。
⑦ lv_anim_path_bounce：从最终值反弹一点。

上述动画路径根据函数 lv_anim_set_path 来设置的。

lv_anim_set_path 函数

设置动画路径，其函数原型如以下所示：

```
static inline void lv_anim_set_path_cb( lv_anim_t * a,
                                        lv_anim_path_cb_t path_cb)
```

该函数的形参，如表 6.5.2.1 所示：

参数	描述	
a	指向初始化的' lv_anim_t '变量的指针	
path_cb	设置动画路径	
	lv_anim_path_linear	线性动画
	lv_anim_path_step	最后一步改变
	lv_anim_path_ease_in	一开始很慢
	lv_anim_path_ease_out	最后慢
	lv_anim_path_ease_in_out	开始和结束都很慢
	lv_anim_path_overshoot	超出最终值
	lv_anim_path_bounce	从最终值反弹一点（如撞墙）

表 6.5.2.1 函数 lv_anim_set_path_cb()形参描述

返回值：无。

6.5.3 设置动画速度与时间

默认情况下，我们可直接设置动画的时间，但在某些情况下，设置动画速度更加实用。设置动画速度可调用函数 lv_anim_speed_to_time 计算给定速度的开始值到结束值所需的毫秒时间，该动画速度以单位/秒来解析。例如，lv_anim_speed_to_time(20,0,100)将产生 100/20 = 5 秒(5000 毫秒)。再例如，在 lv_obj_set_x 的例子中，因为该函数的参数单位是像素，所以 20 意味着 20 像素/秒的速度。

lv_anim_speed_to_time 函数

设置动画速度与时间，其函数原型如以下所示：

```
uint32_t lv_anim_speed_to_time(uint32_t speed, int32_t start, int32_t end);
```

该函数的形参，如表 6.5.3.1 所示：

参数	描述
speed	动画速度(单位/秒)
start	动画的起始值

end	动画的结束值

表 6.5.3.1 函数 lv_anim_set_path_cb()形参描述

返回值：给定参数的动画所需的时间[ms]。

6.5.4 删除动画

如果用户在 LVGL 运行当中，不需要某个动画可使用函数 lv_anim_del 删除这个动画。该函数具有两个形参，一个是要删除的动画，而另一个是删除动画的回调函数指针。

lv_anim_del 函数

用给定的动画回调函数和删除一个变量的动画，其函数原型如以下所示：

```
bool lv_anim_del(void * var, lv_anim_exec_xcb_t exec_cb);
```

该函数的形参，如表 6.5.3.1 所示：

参数	描述
var	变量指针
exec_cb	一个函数指针

表 6.5.3.1 函数 lv_anim_set_path_cb()形参描述

返回值：True:至少删除 1 个动画，false:不删除动画。

6.5.5 动画实例

下面笔者以一个简单的实例来讲解 LVGL 动画使用流程，如图 6.5.5.1 所示：

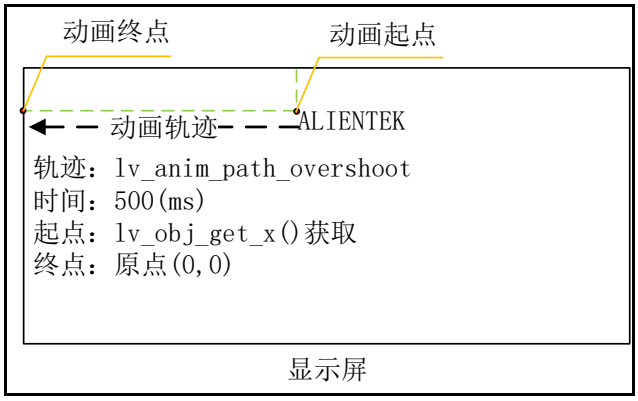


图 6.5.5.1 实例功能示例图

从 6.5.5.1 中，标签部件(ALIENTEK)向左边移动并移动到(0,0)坐标上。首先笔者设置动画路径为 lv_anim_path_overshoot(超出最终值)，然后设置动画时间(移动时间)为 500ms，标签的起点可使用函数 lv_obj_get_x 获取，最后设置动画的终点是 X 轴为 0 坐标，如下源码所示：

```
/**
  * @brief  动画回调函数
  * @param  var：对象
  * @param  v：数值
  * @retval 无
  */

static void anim_x_cb(void* var, int32_t v)
{
    lv_obj_set_x(var, v);
```

```
}

void lv_example(void)
{
    /* 第一步：创建标签 */
    lv_obj_t* lv_obj = lv_label_create(lv_scr_act());
    lv_label_set_text(lv_obj,"ALIENTEK");
    lv_obj_set_pos(lv_obj, 100, 10);

    /* 第二步：动画初始化 */
    lv_anim_t a;
    lv_anim_init(&a);
    /* 第三步：设置动画目标为标签对象 */
    lv_anim_set_var(&a, lv_obj);
    /* 第四步：设置动画起点和终点 */
    lv_anim_set_values(&a, lv_obj_get_x(lv_obj), 0);
    /* 第五步：设置动画时间 */
    lv_anim_set_time(&a, 500);
    /* 第六步：设置动画回调函数 */
    lv_anim_set_exec_cb(&a, anim_x_cb);
    /* 第七步：设置动画轨道 */
    lv_anim_set_path_cb(&a, lv_anim_path_overshoot);
    /* 第八步：开启动画 */
    lv_anim_start(&a);
}
```

根据上述的源码和图 6.5.5.1 的要求设计，我们会发现标签(ALIENTEK)会向做移动，该移动时间为 500ms，如图 6.5.5.2 所示：

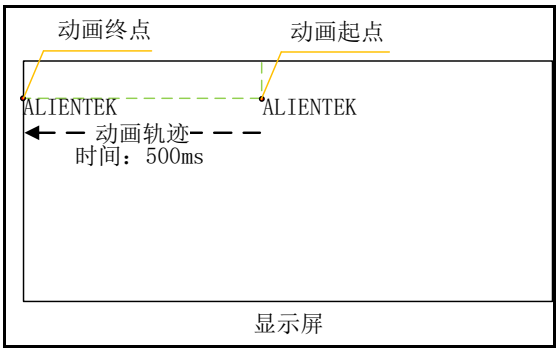


图 6.5.5.2 动画实例示意图

6.5.6 动画时间线

时间线表示时间段。它提供让用户可指定该时间段的长度、开始时间、重复次数、该时间段内时间进度的快慢等属性。LVGL 动画时间线是多个动画的集合，可以轻松创建复杂的复合动画，如图 6.5.6.1 所示：

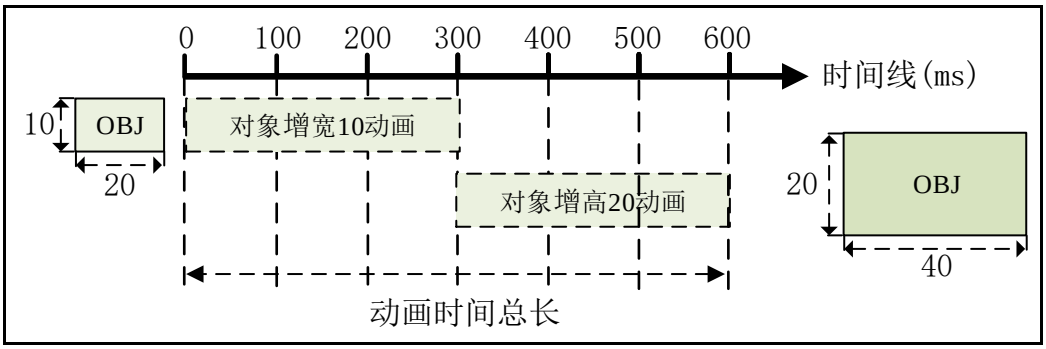


图 6.5.6.1 设置一个对象多个动画示意图

可设置多个对象和多个动画，LVGL 动画时间线类似于制作动画软件的时间轴的设置。

根据图 6.5.6.1 制作动画时间线的实例，如下源码所示：

```
static lv_anim_timeline_t* anim_timeline = NULL;

static lv_obj_t* obj1 = NULL;

static const lv_coord_t obj_width = 90;
static const lv_coord_t obj_height = 70;

/**
 * @brief       a1 动画回调函数
 * @param       var:对象
 * @param       v:数值
 * @retval      无
 */
static void set_width(void* var, int32_t v)
{
    /* 设置 obj1 对象宽度 */
    lv_obj_set_width((lv_obj_t*)var, v);
}

/**
 * @brief       a2 动画回调函数
 * @param       var:对象
 * @param       v:数值
 * @retval      无
 */
static void set_height(void* var, int32_t v)
{
    /* 设置 obj1 对象高度 */
    lv_obj_set_height((lv_obj_t*)var, v);
}
```

```
/**
 * @brief       创建时间线
 * @param       无
 * @retval      无
 */
static void anim_timeline_create(void)
{
    /* obj1 对象增宽动画 */
    lv_anim_t a1;
    lv_anim_init(&a1);
    lv_anim_set_var(&a1, obj1);
    lv_anim_set_values(&a1, obj_width, obj_width + 10 );
    lv_anim_set_early_apply(&a1, false);
    lv_anim_set_exec_cb(&a1, (lv_anim_exec_xcb_t)set_width);
    lv_anim_set_path_cb(&a1, lv_anim_path_overshoot);
    lv_anim_set_time(&a1, 300);

    /* obj1 对象增高动画 */
    lv_anim_t a2;
    lv_anim_init(&a2);
    lv_anim_set_var(&a2, obj1);
    lv_anim_set_values(&a2, obj_height, obj_height + 20 );
    lv_anim_set_early_apply(&a2, false);
    lv_anim_set_exec_cb(&a2, (lv_anim_exec_xcb_t)set_height);
    lv_anim_set_path_cb(&a2, lv_anim_path_ease_out);
    lv_anim_set_time(&a2, 300);

    /* 创建动画时间线 */
    anim_timeline = lv_anim_timeline_create();
    /* 把 a1 动画添加到时间线中 */
    lv_anim_timeline_add(anim_timeline, 0, &a1);
    /* 把 a2 动画添加到时间线中 */
    lv_anim_timeline_add(anim_timeline, 300, &a2);
}

/**
 * @brief       按键回调函数
 * @param       e：事件
 * @retval      无
 */
static void btn_start_event_handler(lv_event_t* e)
{
    lv_obj_t* btn = lv_event_get_target(e);
```

```
    if (!anim_timeline) {
        anim_timeline_create();
    }
    /* 获取按下的点击状态 */
    bool reverse = lv_obj_has_state(btn, LV_STATE_CHECKED);
    /* 支持整个动画组的向前和向后播放 */
    lv_anim_timeline_set_reverse(anim_timeline, reverse);
    /* 启动动画时间轴 */
    lv_anim_timeline_start(anim_timeline);
}

/**
 * @brief       LVGL 入口
 * @param       无
 * @retval      无
 */
void lv_main(void)
{
    lv_obj_t* par = lv_scr_act();
    /* 创建按键部件 */
    lv_obj_t* btn_start = lv_btn_create(par);
    /* 设置回调函数 */
    lv_obj_add_event_cb(btn_start, btn_start_event_handler,
                        LV_EVENT_VALUE_CHANGED, NULL);
    lv_obj_align(btn_start, LV_ALIGN_TOP_MID, -100, 20);
    /* 按键文本 */
    lv_obj_t* label_start = lv_label_create(btn_start);
    lv_label_set_text(label_start, "Start");
    lv_obj_center(label_start);

    /* 创建动画操作对象 */
    obj1 = lv_obj_create(par);
    lv_obj_set_size(obj1, obj_width, obj_height);
}
```

上述源码可知：当我们按下 btn_start 按键时创建 a1 和 a2 动画，这两个动画的动画时间都是 300ms，它们的功能分别为 obj1 增宽和 obj1 增高，然后调用函数 lv_anim_timeline_create 创建动画时间线并调用函数 lv_anim_timeline_add 把 a1 和 a2 动画挂载到动画时间线上，最后我们调用函数 lv_anim_timeline_start 开启动画时间线。

6.5.6.1 动画时间线相关 API 函数

1. lv_anim_timeline_create 函数

创建动画时间线，其函数原型如以下所示：

```
lv_anim_timeline_t * lv_anim_timeline_create(void)
```

返回值：动画时间线指针。

2. lv_anim_timeline_add 函数

把动画挂载时间线上，其函数原型如以下所示：

```
void lv_anim_timeline_add(lv_anim_timeline_t * at,
                          uint32_t start_time,
                          lv_anim_t * a)
```

该函数的形参，如表 6.5.6.1.1 所示：

参数	描述
at	时间线指针
start_time	开始时间
a	挂载的动画

表 6.5.6.1.1 函数 lv_anim_timeline_add()形参描述

返回值：无。

3. lv_anim_timeline_start 函数

开启动画时间线，其函数原型如以下所示：

```
uint32_t lv_anim_timeline_start(lv_anim_timeline_t * at)
```

该函数的形参，如表 6.5.6.1.2 所示：

参数	描述
at	时间线指针

表 6.5.6.1.2 函数 lv_anim_timeline_start()形参描述

返回值：活动时间。

4. lv_anim_timeline_set_reverse 函数

支持整个动画组的向前和向后播放，其函数原型如以下所示：

```
void lv_anim_timeline_set_reverse(lv_anim_timeline_t * at, bool reverse)
```

该函数的形参，如表 6.5.6.1.3 所示：

参数	描述
at	时间线指针
reverse	Ture:支持，false:不支持

表 6.5.6.1.3 函数 lv_anim_timeline_set_reverse()形参描述

返回值：无。

5. lv_anim_timeline_stop 函数

清除状态，其函数原型如以下所示：

```
void lv_anim_timeline_stop(lv_anim_timeline_t * at)
```

该函数的形参，如表 6.5.6.1.4 所示：

参数	描述
at	时间线指针

表 6.5.6.1.4 函数 lv_anim_timeline_stop()形参描述

返回值：无。

6. lv_anim_timeline_set_progress 函数

设置对象的状态对应时间轴的进度，其函数原型如以下所示：

```
void lv_anim_timeline_set_progress(lv_anim_timeline_t * at, uint16_t progress)
```

该函数的形参，如表 6.5.6.1.5 所示：

参数	描述

at	时间线指针
progress	进度数值

表 6.5.6.1.5 函数 lv_anim_timeline_set_progress()形参描述

返回值：无。

7. lv_anim_timeline_get_playtime 函数

函数获取整个动画时间线的总时长，其函数原型如以下所示：

```
uint32_t lv_anim_timeline_get_playtime(lv_anim_timeline_t * at)
```

该函数的形参，如表 6.5.6.1.6 所示：

参数	描述
at	时间线指针

表 6.5.6.1.6 函数 lv_anim_timeline_get_playtime()形参描述

返回值：时间线的总时长。

8. lv_anim_timeline_get_reverse 函数

获取是否反转动画时间线，其函数原型如以下所示：

```
bool lv_anim_timeline_get_reverse(lv_anim_timeline_t * at)
```

该函数的形参，如表 6.5.6.1.7 所示：

参数	描述
at	时间线指针

表 6.5.6.1.7 函数 lv_anim_timeline_get_reverse()形参描述

返回值：ture:获取成功，false:获取失败。

9. lv_anim_timeline_del 函数

清除状态，其函数原型如以下所示：

```
void lv_anim_timeline_del(lv_anim_timeline_t * at)
```

该函数的形参，如表 6.5.6.1.8 所示：

参数	描述
at	时间线指针

表 6.5.6.1.8 函数 lv_obj_clear_flag()形参描述

返回值：无。

6.5.7 动画相关 API 函数

LVGL 官方提供动画相关 API 函数，如表 6.5.7.1 所示：

函数	描述
lv_anim_init()	初始化一个动画变量
lv_anim_set_var()	设置一个变量来动画
lv_anim_set_exec_cb()	设置动画回调函数
lv_anim_set_time()	设置动画的持续时间
lv_anim_set_delay()	在开始动画之前设置延迟
lv_anim_set_values()	设置动画的开始和结束值
lv_anim_set_path_cb	设置动画的路径（曲线）
lv_anim_set_start_cb()	在动画真正开始时设置函数调用
lv_anim_set_get_value_cb()	设置动画回调函数以使用变量的当前值
lv_anim_set_ready_cb()	在动画准备好时设置函数调用
lv_anim_set_playback_time()	使动画在前进方向准备好时播放
lv_anim_set_playback_delay()	使动画在前进方向准备好时播放
lv_anim_set_repeat_count()	让动画自己重复次数

lv_anim_set_repeat_delay()	在重复动画之前设置延迟
lv_anim_set_early_apply()	设置一个动画是应该立即应用还是仅在延迟到期时应用
lv_anim_set_user_data()	设置动画的自定义用户数据字段
lv_anim_start()	开启动画
lv_anim_get_delay()	在开始动画之前获得延迟
lv_anim_get_playtime()	获取用于播放动画的时间
lv_anim_get_user_data()	获取动画的 user_data 字段
lv_anim_del()	使用给定的动画器函数删除变量的动画
lv_anim_del_all()	删除所有动画
lv_anim_count_running()	获取当前正在运行的动画数量
lv_anim_speed_to_time()	计算具有给定速度的动画的时间以及开始和结束值
lv_anim_refr_now()	手动刷新动画的状态
lv_anim_path_linear()	计算应用线性特征的动画的当前值
lv_anim_path_ease_in()	计算减缓开始阶段的动画的当前值
lv_anim_path_ease_out()	计算减缓结束阶段的动画的当前值
lv_anim_path_ease_in_out()	计算应用“S”特征（余弦）的动画的当前值
lv_anim_path_overshoot()	计算结束时有过冲的动画的当前值
lv_anim_path_bounce()	计算具有 3 次反弹的动画的当前值
lv_anim_path_step()	计算应用步进特性的动画的当前值

表 6.5.7.1 动画相关 API 函数描述

上述表中的函数是 LVGL 提供用户使用的动画相关 API 函数，这些函数非常简单，大家可以在官方文档查看它们的函数原型，这里笔者就不必再去讲解了。

6.6 LVGL 定时器

LVGL 有一个内置的软件定时器系统，该 LVGL 定时器系统是有误差的，前面笔者已经讲解过，由于 lv_timer_handler 函数并不准时的调用，所以导致 LVGL 软件定时器也是不准时的。

6.6.1 创建定时器

LVGL 定时器，它构建在硬件定时器基础之上，使系统能够提供不受硬件定时器资源限制的定时器服务，它实现的功能与硬件定时器也是类似的。

LVGL 定时器可调用函数 lv_timer_create 或者 lv_timer_create_basic 创建。注意：函数 lv_timer_create_basic 不指定任何参数的情况下创建一个新的定时器。

1. lv_timer_create 函数

创建一个定时器，其函数原型如以下所示：

```
lv_timer_t * lv_timer_create(timer_cb, period_ms, user_data)
```

该函数的形参，如表 6.6.1.1 所示：

参数	描述
timer_cb	定时器回调函数
period_ms	定时周期
user_data	传入参数

表 6.6.1.1 函数 lv_timer_create()形参描述

返回值：定时器控制块。

定时器回调函数原型如下所示：

```
void (*lv_timer_cb_t)(lv_timer_t *);
```

2. lv_timer_create_basic 函数

创建一个定时器，其函数原型如以下所示：

```
lv_timer_t * lv_timer_create_basic(void)
{
    return lv_timer_create(NULL, DEF_PERIOD, NULL);
}
```

返回值：定时器控制块。

注意：使用该函数创建 LVGL 定时器时必须调用函数 lv_timer_set_cb 自定义定时回调函数，默认定时周期为 DEF_PERIOD，读者也可以调用 lv_timer_set_period 设置定时周期。

6.6.2 重设/就绪定时器

lv_timer_ready (timer)在下一次调用 lv_timer_handler 时运行这个定时器。

lv_timer_reset (timer)重置定时器的周期。它将在定义的毫秒周期结束后再次被调用。

1. 设置定时器参数

lv_timer_set_cb 函数

设置定时器回调函数，其函数原型如以下所示：

```
void lv_timer_set_cb(lv_timer_t * timer, lv_timer_cb_t timer_cb)
```

该函数的形参，如表 6.6.2.1 所示：

参数	描述
timer	定时器控制块
timer_cb	定时回调函数

表 6.6.2.1 函数 lv_timer_set_cb()形参描述

返回值：无。

定时器回调函数原型如下所示：

```
void (*lv_timer_cb_t)(lv_timer_t *);
```

lv_timer_set_period 函数

设置定时器定时周期，其函数原型如以下所示：

```
void lv_timer_set_period(lv_timer_t * timer, uint32_t period)
```

该函数的形参，如表 6.6.2.2 所示：

参数	描述
timer	定时器控制块
new_period	定时周期

表 6.6.2.2 函数 lv_timer_set_period()形参描述

返回值：无。

2. 重复计数

可以使用 lv_timer_set_repeat_count(timer, count)使定时器只重复给定的次数。计时器将自动删除后，它被调用的定义次数。将计数设置为-1 以无限期地重复。

lv_timer_set_repeat_count 函数

设置 timer 将重复的次数，其函数原型如以下所示：

```
void lv_timer_set_repeat_count(lv_timer_t *timer, int32_t repeat_count )
```

该函数的形参，如表 6.6.3.3 所示：

参数	描述
timer	定时器控制块
repeat_count	-1：无穷大；0：停止；n>0：剩余次数

表 6.6.3.3 函数 lv_timer_set_repeat_count()形参描述

返回值：无。

3. 测量空闲时间

测量 LVGL 空闲时间可通过函数 lv_timer_get_idle 获得 lv_timer_handler 的空闲时间百分比。注意：该函数不测量整个系统的空闲时间，只测量 lv_timer_handler 的空闲时间。如果用户使用一个操作系统并在定时器中调用 lv_timer_handler，这可能会产生误导，因为它不会实际测量操作系统在空闲线程中花费的时间。

lv_timer_get_idle 函数

获得 lv_timer_handler 的空闲时间百分比，其函数原型如以下所示：

```
uint8_t lv_timer_get_idle(void)
```

返回值：返回空闲时间百分比。

6.6.3 定时器 API 函数

LVGL 官方提供输入设备相关 API，如表 6.6.3.1 所示：

函数	描述
lv_timer_del()	删除一个定时器
lv_timer_pause()	暂停定时器
lv_timer_resume()	恢复定时器
lv_timer_enable()	启用或禁用整个 lv_timer 处理
lv_timer_get_next()	获取下一个定时器控制块

表 6.6.3.1 定时器相关 API 函数描述

1. lv_timer_del 函数

删除一个定时器，其函数原型如以下所示：

```
void lv_timer_del ( lv_timer_t *timer)
```

该函数的形参，如表 6.6.3.2 所示：

参数	描述
timer	定时器控制块

表 6.6.3.2 函数 lv_timer_del()形参描述

返回值：无。

2. lv_timer_pause()函数

暂停定时器，其函数原型如以下所示：

```
void lv_timer_pause ( lv_timer_t *timer)
```

该函数的形参，如表 6.6.3.3 所示：

参数	描述
timer	定时器控制块

表 6.6.3.3 函数 lv_timer_pause()形参描述

返回值：无。

3. lv_timer_resume()函数

恢复定时器，其函数原型如以下所示：

```
void lv_timer_resume ( lv_timer_t *timer)
```

该函数的形参，如表 6.6.3.4 所示：

参数	描述
timer	定时器控制块

表 6.6.3.4 函数 lv_timer_resume()形参描述

返回值：无。

4. lv_timer_enable 函数

启用或禁用整个定时器系统处理，其函数原型如以下所示：

```
void lv_timer_enable ( bool en )
```

该函数的形参，如表 6.6.3.5 所示：

参数	描述
en	true：lv_timer 处理正在运行，false：lv_timer 处理已暂停

表 6.6.3.5 函数 lv_timer_enable()形参描述

返回值：无。

5. lv_timer_get_next 函数

获取下一个定时器控制块，其函数原型如以下所示：

```
lv_timer_t * lv_timer_get_next ( lv_timer_t * timer )
```

该函数的形参，如表 6.6.3.6 所示：

参数	描述
timer	定时器控制块

表 6.6.3.6 函数 lv_timer_get_next()形参描述

返回值：返回下一个定时器控制块。

6.7 LVGL 事件

6.7.1 事件简介

“事件(Event)：可以理解为对一个部件的某种同类型操作动作的集合。例如，单击一个按钮、在文本框中输入一个字符串、选择一个菜单选项、选中一个单选按钮等都可以认为是一个操作动作。LVGL 提供完善的事件处理机制，能够监听事件，识别事件源，并完成事件处理，如图 6.7.1.1 所示：

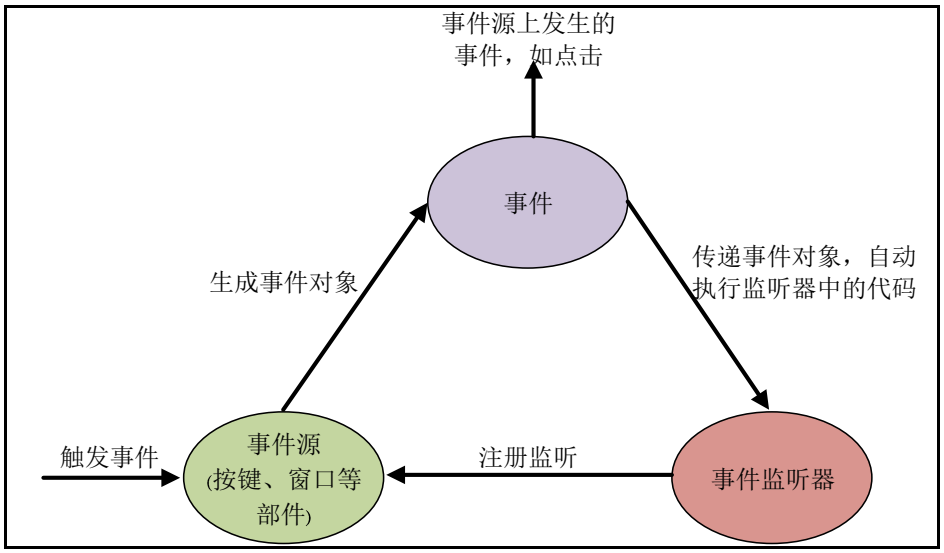


图 6.7.1.1 LVGL 事件处理机制

LVGL 事件处理机制分为三部分：

① 事件源：能够产生事件的部件，在 LVGL 中，每一个部件都可以触发事件。

② 事件：用户对部件的一个操作(点击，长按、短按、重绘、数值发生改变等事件)。

③ 事件监听器：接收事件、解释事件并处理用户交互的方法处理事件源发生的事件。

6.7.2 添加事件

在 LVGL 中，部件添加一个事件可调用函数 lv_obj_add_event_cb 添加相应的事件，如下源码所示：

```
lv_obj_t * btn = lv_btn_create(lv_scr_act());
lv_obj_add_event_cb(btn, my_event_cb, LV_EVENT_CLICKED, NULL);



static void my_event_cb(lv_event_t * event)
{
    printf("Clicked\n");
}
```

从上述源码可知：如果 btn 按键点击时，触发事件执行相应的回调函数。

lv_obj_add_event_cb 函数

部件添加一个事件，其函数原型如以下所示：

```
struct _lv_event_dsc_t * lv_obj_add_event_cb(lv_obj_t * obj,
                                             lv_event_cb_t event_cb,
                                             lv_event_code_t filter,
                                             void * user_data)
```

该函数的形参，如表 6.7.2.1 所示：

参数	描述
obj	指向对象的指针
event_cb	事件回调函数
filter	事件类型
user_data	事件参数

表 6.7.2.1 函数 lv_obj_add_event_cb()形参描述

返回值：返回事件描述符。

注意：事件类型，我们在 6.7.4 小节会讲解。

6.7.3 删除事件

事件可通过函数 lv_obj_remove_event_cb 或函数 lv_obj_remove_event_dsc 从对象中移除。event_dsc 是 lv_obj_add_event_cb 返回的指针。

1. lv_obj_remove_event_cb 函数

删除对象的事件处理函数，其函数原型如以下所示：

```
bool lv_obj_remove_event_cb(struct _lv_obj_t * obj, lv_event_cb_t event_cb);
```

该函数的形参，如表 6.7.3.1 所示：

参数	描述
obj	指向对象的指针
event_cb	事件回调函数

表 6.7.3.1 函数 lv_obj_remove_event_cb()形参描述

返回值：true：任何事件处理程序被移除。

2. lv_obj_remove_event_dsc 函数

删除一个对象的事件处理函数(已弃用)，其函数原型如以下所示：

```
bool lv_obj_remove_event_dsc( struct _lv_obj_t * obj,
```

```
struct _lv_event_dsc_t * event_dsc);
```

该函数的形参，如表 6.7.3.2 所示：

参数	描述
obj	指向对象的指针
event_dsc	指向要删除的事件描述符的指针(由 lv_obj_add_event_cb 返回)

表 6.7.3.2 函数 lv_obj_remove_event_dsc()形参描述

返回值：true：任何事件处理程序被移除。

6.7.4 事件类型

事件类型可以分为以下几类：

① **输入设备事件**
② **绘图事件**
③ **其他事件**
④ **特别事件**
⑤ **自定义事件**

所有对象（例如按钮/标签/滑块等），无论其类型如何，都会接收输入设备、绘图和其他事件。注意：自定义事件由用户添加，LVGL 从不发送自定义事件。

接下来我们分别地讲解这几个事件类型的作用。

1. 输入设备事件:主要描述触摸、按键等输入设备触发的事件，如表 6.7.4.1 所示：

事件类型	描述
LV_EVENT_PRESSED	按下状态
LV_EVENT_PRESSING	按下时连续调用
LV_EVENT_PRESS_LOST	对象仍被按下，但光标/手指滑离对象
LV_EVENT_SHORT_CLICKED	短按，滚动不调用
LV_EVENT_LONG_PRESSED	长按(long_press_time 指定长按时间)，滚动不调用
LV_EVENT_LONG_PRESSED_REPEAT	长按，只调用一次。滚动则不调用。
LV_EVENT_CLICKED	如果对象没有滚动（不管长按还是短按），则在释放时调用
LV_EVENT_RELEASED	每次释放对象时调用
LV_EVENT_SCROLL_BEGIN	滚动开始
LV_EVENT_SCROLL_END	滚动结束
LV_EVENT_SCROLL	滚动了一个对象
LV_EVENT_GESTURE	检测到手势
LV_EVENT_KEY	KEY 被发送到对象(按键输入设备有关)
LV_EVENT_FOCUSED	一个对象被聚焦
LV_EVENT_DEFOCUSED	对象未聚焦
LV_EVENT_LEAVE	对象未聚焦但仍被选中
LV_EVENT_HIT_TEST	执行高级命中测试。使用并检查是否可以单击对象

表 6.7.4.1 输入设备事件类型

2. 绘图事件：主要描述部件绘画的事件流程，如表 6.7.4.2 所示：

事件类型	描述
LV_EVENT_COVER_CHECK	检查一个物体是否完全覆盖了一个区域
LV_EVENT_REFR_EXT_DRAW_SIZE	获取对象周围所需的额外绘制区域（例如阴影）
LV_EVENT_DRAW_MAIN_BEGI	开始主要绘图阶段

LV_EVENT_DRAW_MAIN	执行主图
LV_EVENT_DRAW_MAIN_END	完成主要绘图阶段
LV_EVENT_DRAW_POST_BEGIN	开始绘制后阶段（当所有子类都被绘制时）
LV_EVENT_DRAW_POST	执行绘制后阶段（当所有子类都被绘制时）
LV_EVENT_DRAW_POST_END	完成后期绘制阶段（当所有子类都被绘制时）
LV_EVENT_DRAW_PART_BEGIN	开始画一个部分
LV_EVENT_DRAW_PART_END	完成绘制一个部分

表 6.7.4.2 部件的绘画事件描述

注意：在部件绘画事件中，不能设置大小等属性，可以获取部件的属性。

3. 其他事件：主要描述删除、大小、样式等事件，如表 6.7.4.3 所示：

事件类型	描述
LV_EVENT_DELETE	正在删除对象
LV_EVENT_CHILD_CHANGED	子类被删除/添加
LV_EVENT_CHILD_CREATED	子类被创造出来
LV_EVENT_CHILD_DELETED	子类被删除
LV_EVENT_SIZE_CHANGED	对象坐标/大小已更改
LV_EVENT_STYLE_CHANGED	对象的样式已更改
LV_EVENT_BASE_DIR_CHANGED	基本目录已更改
LV_EVENT_GET_SELF_SIZE	获取小部件的内部大小
LV_EVENT_SCREEN_UNLOAD_START	屏幕卸载开始
LV_EVENT_SCREEN_LOAD_STAR	屏幕加载开始
LV_EVENT_SCREEN_LOADED	加载了一个屏幕
LV_EVENT_SCREEN_UNLOADED	卸载屏幕

表 6.7.4.3 其他事件描述

4. 特别事件：主要描述对象数值被修改等事件，如表 6.7.4.4 所示：

事件类型	描述
LV_EVENT_VALUE_CHANGED	对象的值已更改（即滑块部件移动）
LV_EVENT_INSERT	文本被插入到对象中
LV_EVENT_REFRESH	通知对象刷新它上面的东西
LV_EVENT_READY	一个过程已经完成
LV_EVENT_CANCEL	一个进程已被取消

表 6.7.4.4 特别事件描述

注意：上面的事件类型和 6.7.2 小节添加事件函数的事件类型相关。

5. 自定义事件：

任何自定义事件代码都可通过以下方式注册：

```
uint32_t MY_EVENT_1 = lv_event_register_id();
```

上述函数是获取事件等级表的 ID，然后调用函数 lv_event_send()发送事件到任何事件回调函数当中。

lv_event_send 函数

发送一个事件，其函数原型如以下所示：

```
lv_res_t lv_event_send(lv_obj_t * obj,
                       lv_event_code_t event_code,
                       void * param)
```

该函数的形参，如表 6.7.4.5 所示：

参数	描述
obj	指向对象的指针

event_dsc	发送自定义事件类型
param	发送事件的参数

表 6.7.4.5 函数 lv_event_send()形参描述

返回值：LV_RES_OK：发送成功，否则失败。

6.7.5 获取事件字段

在事件回调函数中，如果多个对象共用一个事件回调或者获取事件回调函数传入的参数以及事件触发的类型等信息，那么用户如何获取这些信息呢？例如，用户如何区分那个对象触发了事件以及该对象触发的事件类型等问题。为了解决这些问题，LVGL 提供了事件字段函数来获取事件相关的信息，如下表 6.7.5.1 所示：

函数	描述
lv_event_get_code(e)	获取事件类型(按下、滑动、长按等类型)
lv_event_get_current_target(e)	获取向其发送事件的对象(触发这个事件的对象)
lv_event_get_target(e)	获取最初触发事件的对象
lv_event_get_user_data(e)	获取传入参数
lv_event_get_param(e)	获取由函数 lv_event_send()传入的参数

表 6.7.5.1 获取事件字段函数

第七章 LVGL 文件系统

LVGL 文件系统对于 LVGL 来说是非常重要地，因为我们设计复杂的 UI 界面时，可使用内部 flash 存储数据，如果 MCU 内部 flash 不足时，那么从外部存储设备读取是非常必要地。LVGL 作者为用户提供了一套 LVGL 的文件系统的接口，正所谓万宗不离其一，其实 FatFS 和 LVGL 文件系统很相似，因为 LVGL 文件系统是介绍操作 FatFS 文件系统函数。换句话来说，LVGL 文件系统 API 接口是从 FatFS API 函数进行封装而来。在 FatFS 文件操作时候，无非就是三、四步，打开/读取/写入/关闭等操作，而 LVGL 文件系统也是一样的操作，本章笔者来为大家讲解 LVGL 文件系统 API 函数的使用。

本章分为几个部分讲解：

7.1 LVGL 文件系统移植

7.2 LVGL 文件系统原理解析

7.3 LVGL 文件系统实验

7.1 LVGL 文件系统移植

在 LVGL 中，它支持 POSIX、WIN32、STDIO 和 FATFS 文件系统接口，在我们学习嵌入式时，FATFS 文件系统库是常用的接口，移植 FATFS 文件系统相关知识请大家参考正点原子裸机教程，该教程已经很详细讲解 FATFS 相关知识及使用，本教程只教大家如何使用 LVGL 文件系统 API 函数。

首先在正点原子相应的板卡找到 FATFS 裸机例程，然后找到 Middlewares 目录下复制 FATFS 以及 Drivers\BSP 目录下复制 NORFLASH、SPI、SDMMC 文件夹到 LVGL 工程当中。

注意：请根据自己的开发板找到裸机例程进行复制，因为不同开发板的文件夹命名是不一样，所以请根据大家仔细复制文件。

接下来我们分为几个部分来讲解文件系统的移植过程。

1. 在工程中，创建 Middlewares/FATFS 分组，添加 Middlewares/FATFS 文件夹下的文件，如图 7.1.1 所示：

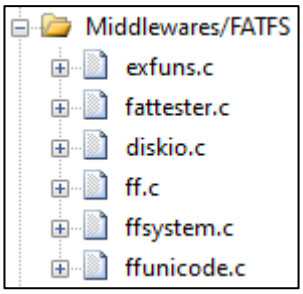


图 7.1.1 Middlewares/FATFS 分组添加文件

2. 添加 Middlewares/lvgl/src/fadrv 分组，添加 \lvgl\src\extra\libs\fsdrv 文件夹下的 lv_fs_fasfs.c 文件，如图 7.1.2 所示：

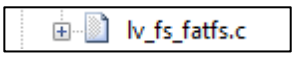


图 7.1.2 Middlewares/LVGL/GUI_FATFS 分组添加文件

3. Drivers/BSP 分组添加 SD 卡驱动文件，请查看裸机教程如何添加，这里不讲解。

4. 添加头文件路径，如图 7.1.3 所示：

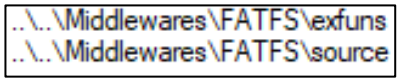


图 7.1.3 添加头文件路径

5. 打开 lv_conf.h 文件使能 fatfs 文件系统，如图 7.1.4 所示：

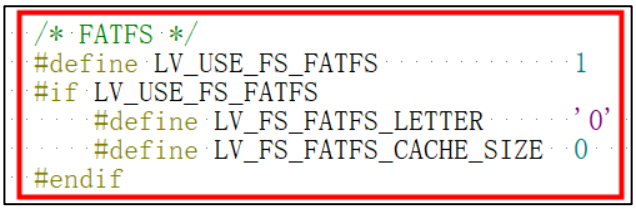


图 7.1.4 使能 fatfs 文件系统

6. 打开 lv_fs_fatfs.c 文件，修改 fs_init()函数，如下源码所示：

```
/**
 * @brief       初始化存储设备和文件系统
 * @param       无
 * @retval      无
 */
static void fs_init(void)
{
    uint8_t res;

    /* 初始化 SD 卡和 FatFS 本身
     * 最好在自己的库中完成，一遍以后更新 */
    while (sd_init())                /* 初始化 SD 卡 */
    {
        printf("SD Card Error, Please Check!\r\n");
        LED0_TOGGLE();
        HAL_Delay(200);
    }

    LED0(0);

    exfuns_init();                   /* 为 fatfs 相关变量申请内存 */
    res = f_mount(fs[0], "0:", 1);   /* 挂载 SD 卡 */

    if (0 != res)
    {
        printf("SD Card Mount Fail, Please Check!\r\n");
        LED0_TOGGLE();
        HAL_Delay(200);
    }
}
```

到了这里，我们已经把 LVGL 文件系统移植完成。

7.2 LVGL 文件系统原理解析

前面笔者已经讲解过，LVGL 文件系统和 FatFS 文件系统是类似的，只不过 LVGL 文件系统把 FatFS 相关接口进行封装，最终底层还是调用 FatFS 文件系统相关接口，如图 7.2 所示：

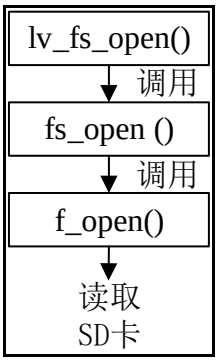


图 7.2 顶层函数调用流程

从上述图 7.2 所示，如果我们在应用层调用函数 lv_fs_open 时，该函数会调用 lv_fs_fatfs.c 文件中的函数 fs_open，该函数实际调用 FatFS 文件系统接口，所以其他应用层 LVGL 文件系统接口函数也是这样的流程。

7.2.1 LVGL 文件系统相关 API 函数

LVGL 官方提供 LVGL 文件系统相关 API，如表 7.2.1.1 所示：

函数	描述
操作文件 API	
lv_fs_open()	打开一个文件
lv_fs_close()	关闭一个文件
lv_fs_read()	读取文件内容
lv_fs_write()	写入数据到文件中
lv_fs_seek()	设置'cursor'(读/写指针)在文件中的位置
lv_fs_tell()	获取读写指针的位置
操作文件夹 API	
lv_fs_dir_open()	打开一个目录
lv_fs_dir_read()	读取下一个文件名形成一个目录
lv_fs_dir_close()	关闭目录读取

表 7.2.1.1 LVGL 文件系统相关 API 函数

1. lv_fs_open 函数

打开一个文件，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_open(lv_fs_file_t * file_p,
                       const char * path,
                       lv_fs_mode_t mode);
```

该函数的形参，如表 7.2.1.2 所示：

参数	描述	
file_p	指向 lv_fs_file_t 变量的指针	
path	打开文件路径	
mode	模式选择	
	读:FS_MODE_RD	读模式
	写:FS_MODE_WR	写模式

	FS_MODE_RD | FS_MODE_WR	读写模式

表 7.2.1.2 函数 lv_fs_open()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

2. lv_fs_close 函数

关闭一个文件，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_close(lv_fs_file_t * file_p);
```

该函数的形参，如表 7.2.1.3 所示：

参数	描述
file_p	指向 lv_fs_file_t 变量的指针

表 7.2.1.3 函数 lv_fs_close()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

3. lv_fs_read 函数

读取文件内容，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_read(lv_fs_file_t * file_p,
                       void * buf,
                       uint32_t btr,
                       uint32_t * br);
```

该函数的形参，如表 7.2.1.4 所示：

参数	描述
file_p	指向 lv_fs_file_t 变量的指针
buf	指向存储读字节的缓冲区的指针
btr	读取字节
br	实际读字节数(bytes read)。如果未使用 NULL。

表 7.2.1.4 函数 lv_fs_read()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

4. lv_fs_write 函数

写入文件内容，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_write(lv_fs_file_t * file_p,
                        const void * buf,
                        uint32_t btw,
                        uint32_t * bw);
```

该函数的形参，如表 7.2.1.5 所示：

参数	描述
file_p	指向 lv_fs_file_t 变量的指针
buf	指向要写入字节的缓冲区的指针
btw	字节写
bw	实际写入字节数(bytes written)。如果未使用 NULL。

表 7.2.1.5 函数 lv_fs_write()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

5. lv_fs_seek 函数

设置'cursor'(读/写指针)在文件中的位置，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_seek(lv_fs_file_t * file_p,
                       uint32_t pos,
                       lv_fs_whence_t whence);
```

该函数的形参，如表 7.2.1.6 所示：

参数	描述
file_p	指向 lv_fs_file_t 变量的指针
pos	索引(0:文件的开始)
whence	告诉从哪里设置的位置

表 7.2.1.6 函数 lv_fs_seek()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

6. lv_fs_tell 函数

给出读写指针的位置，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_tell(lv_fs_file_t * file_p, uint32_t * pos);
```

该函数的形参，如表 7.2.1.7 所示：

参数	描述
file_p	指向 lv_fs_file_t 变量的指针
pos	指针用于存储读、写指针的位置

表 7.2.1.7 函数 lv_fs_tell()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

7. lv_fs_dir_open 函数

为目录读取初始化一个'fs_dir_t'变量，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_dir_open(lv_fs_dir_t * rddir_p, const char * path);
```

该函数的形参，如表 7.2.1.8 所示：

参数	描述
rddir_p	指向'lv_fs_dir_t'变量的指针
path	目录路径

表 7.2.1.8 函数 lv_fs_dir_open()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

8. lv_fs_dir_read 函数

读取下一个文件名形成一个目录，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_dir_read(lv_fs_dir_t * rddir_p, char * fn);
```

该函数的形参，如表 7.2.1.9 所示：

参数	描述
rddir_p	指向'lv_fs_dir_t'变量的指针
fn	指向存储文件名的缓冲区的指针

表 7.2.1.9 函数 lv_fs_dir_read()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

9. lv_fs_dir_close 函数

关闭目录读取，其函数原型如以下所示：

```
lv_fs_res_t lv_fs_dir_close(lv_fs_dir_t * rddir_p);
```

该函数的形参，如表 7.2.1.10 所示：

参数	描述
rddir_p	指向'lv_fs_dir_t'变量的指针

表 7.2.1.10 函数 lv_fs_dir_read()形参描述

返回值：LV_FS_RES_OK：成功，其他：失败。

7.3 LVGL 文件系统实验

7.3.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要根据路径打开 SD 卡上的文件并读取该文件的内容输出到串口当中。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 5 LVGL 文件系统的使用》。注意：DMF407 和 MiniSTM32 不支持本实验。

7.3.2 软件设计

7.3.2.1 程序流程图

本实验的程序流程图，如下图 7.3.2.1 所示：

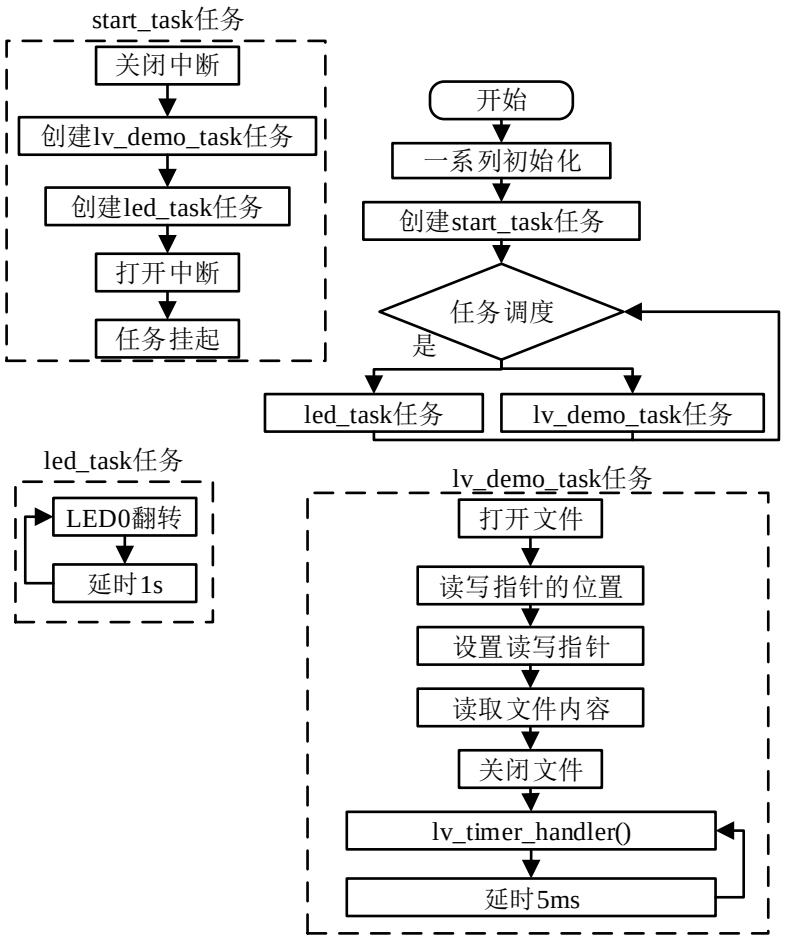


图 7.3.2.1 LVGL 文件系统实验流程图

7.3.2.2 程序解析

打开 lv_mainstart.c 文件，我们的代码都是在该文件下编写的，如下源码所示：

```
/**
 * @brief  文件系统测试
 * @param  fd：文件指针
 * @return 返回名称
 */
long lv_tell(lv_fs_file_t *fd)
{
    uint32_t pos = 0;
    lv_fs_tell(fd, &pos);
    return pos;
}

/**
 * @brief  文件系统测试
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    char rbuf[30] = {0};
    uint32_t rsize = 0;
    lv_fs_file_t fd;
    lv_fs_res_t res;

    res = lv_fs_open(&fd, "0:/SYSTEM/LV_FATFS/Fatfs_test.txt", LV_FS_MODE_RD);

    if (res != LV_FS_RES_OK)
    {
        printf("open 0:/Fatfs_test.txt ERROR\n");
        return ;
    }

    lv_tell(&fd);

    lv_fs_seek(&fd, 0,LV_FS_SEEK_SET);
    lv_tell(&fd);

    res = lv_fs_read(&fd, rbuf, 100, &rsize);

    if (res != LV_FS_RES_OK)
    {
```

```
        printf("read ERROR\n");
        return ;
    }

    lv_tell(&fd);
    printf("READ(%d): %s",rsize , rbuf);

    lv_fs_close(&fd);
}
```

上述源码可知：首先调用函数 lv_fs_open 打开文件，然后调用 lv_fs_tell 获取指针的位置，其次调用函数 lv_fs_read 读取文件内容，最后调用函数 lv_fs_close 关闭文件。

7.3.3 下载验证

工程进行编译并下载到我们的开发板中并插入 SD 卡，如图 7.3.3.1 所示：

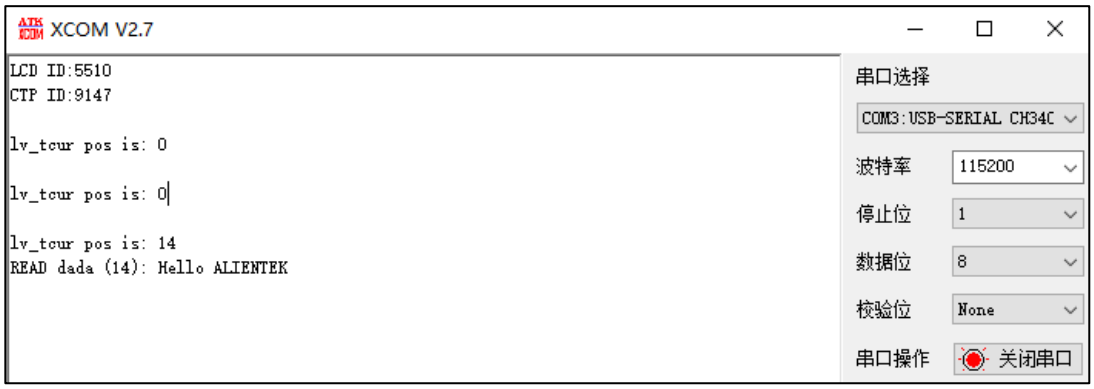


图 7.3.3.1 文件系统验证

第八章 LVGL 字库使用

在 LVGL 中的字体功能也是非常强大的，它支持最高 8bpp 的抗锯齿,另外还有 1bpp，2bpp，4bpp 三个值可选，实现灵活性配置，选择越大的 bpp 值时，要求的 flash 存储资源也是成倍的增加的，比如 4bpp 的存储消耗是 1bpp 存储消耗的 4 倍，除了存储消耗变大之外，bpp 值越大，在界面上进行字体渲染时，绘制速度也会越慢，当然了，bpp 值越大，也是有好处的，那就是绘制出来的字体边缘越平滑，没有毛刺，使我们产品的 UI 界面看上去更高大上，所以在实际项目中，我们应该根据硬件平台的实际性能来选出一个比较合适的 bpp 值，一般的项目都是默认采用 4bpp 就可以了，LVGL 的字体还支持 unicode 编码显示，注意：这里所谓的 unicode 编码是一种泛指能显示所有字符的编码概念，并不是指实际的 unicode 编码，而是指 UTF-8 编码，所以大家不要搞混淆了，LVGL 在 UTF-8 编码的支持下，可以实现显示全球所有的字符，最后要说的一个功能那就是它还支持”图标字体”,可能大家一听,有点懵,这是在”web 前端”中经常用到的一种技术，可以看出来 LVGL 的作者涉及范围很广呀,下面让我们开始更进一步的探究它的细节。

本章分为以下几个部分：

8.1 如何启用 UTF-8 编码

8.2 如何使用 LVGL 内置图标字体

8.3 如何使用 LVGL 内部字库

8.4 如何自定义字库

8.1 如何启用 UTF-8 编码

LVGL 本身支持 2 种编码方式，一种是 ASCII 编码，这种编码只支持英文字符的显示，另外一种编码方式就是 UTF-8 编码，这种编码可以支持全球所有字符的显示。在工程中必须选择 UTF-8 编码格式，至于选择哪一个编码是由 lv_conf.h 文件下 LV_TXT_ENC 配置项定义，如下源码所示：

```
/* 为字符串选择字符编码.
 * IDE 或编辑器应该具有相同的字符编码
 * 1. - LV_TXT_ENC_UTF8
 * 2. - LV_TXT_ENC_ASCII
 * */
#define LV_TXT_ENC LV_TXT_ENC_UTF8
```

上述的代码可知，该工程已经设置支持 UTF8 编码。注意：我们的 Keil 最好也设置 Chinese GB2312 编码，这样我们的开发集成工具就可以支持中文显示了。

8.2 如何使用 LVGL 内置图标字体

什么叫做图标字体呢？图标字体是 web 前端中流行起来的一种技术，它是一个图标，按照我们正常的逻辑，那么图标给我第一反应就是图片，但是它在 LVGL 中使用时，既可以当成字体来显示，也可以当成图片来显示，俩者之间都可以的，这种图标唯一的缺点就是它只能显示单色，果断继承了字体的特性，在 LVGL 系统给用户自带了许多常用小图标，我们可以打开工程找到 lv_symbol_def.h 文件，如下源码所示：

```
#define LV_SYMBOL_AUDIO             "\xef\x80\x81" /*61441, 0xF001*/
```

```
#define LV_SYMBOL_VIDEO          "\xef\x80\x88" /*61448, 0xF008*/
#define LV_SYMBOL_LIST           "\xef\x80\x8b" /*61451, 0xF00B*/
#define LV_SYMBOL_OK             "\xef\x80\x8c" /*61452, 0xF00C*/
#define LV_SYMBOL_CLOSE          "\xef\x80\x8d" /*61453, 0xF00D*/
#define LV_SYMBOL_POWER          "\xef\x80\x91" /*61457, 0xF011*/
#define LV_SYMBOL_SETTINGS       "\xef\x80\x93" /*61459, 0xF013*/
#define LV_SYMBOL_HOME           "\xef\x80\x95" /*61461, 0xF015*/
#define LV_SYMBOL_DOWNLOAD       "\xef\x80\x99" /*61465, 0xF019*/
#define LV_SYMBOL_DRIVE          "\xef\x80\x9c" /*61468, 0xF01C*/
#define LV_SYMBOL_REFRESH        "\xef\x80\xa1" /*61473, 0xF021*/
#define LV_SYMBOL_MUTE           "\xef\x80\xa6" /*61478, 0xF026*/
#define LV_SYMBOL_VOLUME_MID     "\xef\x80\xa7" /*61479, 0xF027*/
#define LV_SYMBOL_VOLUME_MAX     "\xef\x80\xa8" /*61480, 0xF028*/
#define LV_SYMBOL_IMAGE          "\xef\x80\xbe" /*61502, 0xF03E*/
#define LV_SYMBOL_EDIT           "\xef\x8C\x84" /*62212, 0xF304*/
#define LV_SYMBOL_PREV           "\xef\x81\x88" /*61512, 0xF048*/
#define LV_SYMBOL_PLAY           "\xef\x81\x8b" /*61515, 0xF04B*/
#define LV_SYMBOL_PAUSE          "\xef\x81\x8c" /*61516, 0xF04C*/
#define LV_SYMBOL_STOP           "\xef\x81\x8d" /*61517, 0xF04D*/
#define LV_SYMBOL_NEXT           "\xef\x81\x91" /*61521, 0xF051*/
#define LV_SYMBOL_EJECT          "\xef\x81\x92" /*61522, 0xF052*/
#define LV_SYMBOL_LEFT           "\xef\x81\x93" /*61523, 0xF053*/
#define LV_SYMBOL_RIGHT          "\xef\x81\x94" /*61524, 0xF054*/
#define LV_SYMBOL_PLUS           "\xef\x81\xa7" /*61543, 0xF067*/
#define LV_SYMBOL_MINUS          "\xef\x81\xa8" /*61544, 0xF068*/
#define LV_SYMBOL_EYE_OPEN       "\xef\x81\xae" /*61550, 0xF06E*/
#define LV_SYMBOL_EYE_CLOSE      "\xef\x81\xb0" /*61552, 0xF070*/
#define LV_SYMBOL_WARNING        "\xef\x81\xb1" /*61553, 0xF071*/
#define LV_SYMBOL_SHUFFLE        "\xef\x81\xb4" /*61556, 0xF074*/
#define LV_SYMBOL_UP             "\xef\x81\xb7" /*61559, 0xF077*/
#define LV_SYMBOL_DOWN           "\xef\x81\xb8" /*61560, 0xF078*/
#define LV_SYMBOL_LOOP           "\xef\x81\xb9" /*61561, 0xF079*/
#define LV_SYMBOL_DIRECTORY      "\xef\x81\xbb" /*61563, 0xF07B*/
#define LV_SYMBOL_UPLOAD         "\xef\x82\x93" /*61587, 0xF093*/
#define LV_SYMBOL_CALL           "\xef\x82\x95" /*61589, 0xF095*/
#define LV_SYMBOL_CUT            "\xef\x83\x84" /*61636, 0xF0C4*/
#define LV_SYMBOL_COPY           "\xef\x83\x85" /*61637, 0xF0C5*/
#define LV_SYMBOL_SAVE           "\xef\x83\x87" /*61639, 0xF0C7*/
#define LV_SYMBOL_CHARGE         "\xef\x83\xa7" /*61671, 0xF0E7*/
#define LV_SYMBOL_PASTE          "\xef\x83\xAA" /*61674, 0xF0EA*/
#define LV_SYMBOL_BELL           "\xef\x83\xb3" /*61683, 0xF0F3*/
#define LV_SYMBOL_KEYBOARD       "\xef\x84\x9c" /*61724, 0xF11C*/
#define LV_SYMBOL_GPS            "\xef\x84\xa4" /*61732, 0xF124*/
```

```
#define LV_SYMBOL_FILE            "\xef\x85\x9b" /*61787, 0xF158*/
#define LV_SYMBOL_WIFI            "\xef\x87\xab" /*61931, 0xF1EB*/
#define LV_SYMBOL_BATTERY_FULL    "\xef\x89\x80" /*62016, 0xF240*/
#define LV_SYMBOL_BATTERY_3       "\xef\x89\x81" /*62017, 0xF241*/
#define LV_SYMBOL_BATTERY_2       "\xef\x89\x82" /*62018, 0xF242*/
#define LV_SYMBOL_BATTERY_1       "\xef\x89\x83" /*62019, 0xF243*/
#define LV_SYMBOL_BATTERY_EMPTY   "\xef\x89\x84" /*62020, 0xF244*/
#define LV_SYMBOL_USB             "\xef\x8a\x87" /*62087, 0xF287*/
#define LV_SYMBOL_BLUETOOTH       "\xef\x8a\x93" /*62099, 0xF293*/
#define LV_SYMBOL_TRASH           "\xef\x8B\xAD" /*62189, 0xF2ED*/
#define LV_SYMBOL_BACKSPACE       "\xef\x95\x9A" /*62810, 0xF55A*/
#define LV_SYMBOL_SD_CARD         "\xef\x9F\x82" /*63426, 0xF7C2*/
#define LV_SYMBOL_NEW_LINE        "\xef\xA2\xA2" /*63650, 0xF8A2*/
#define LV_SYMBOL_DUMMY           "\xEF\xA3\xBF"
#define LV_SYMBOL_BULLET          "\xE2\x80\xA2" /*20042, 0x2022*/
```

上述的图标字体是 LVGL 内置的字体，它们分别显示成下图的字体，如图 8.2.1 所示：

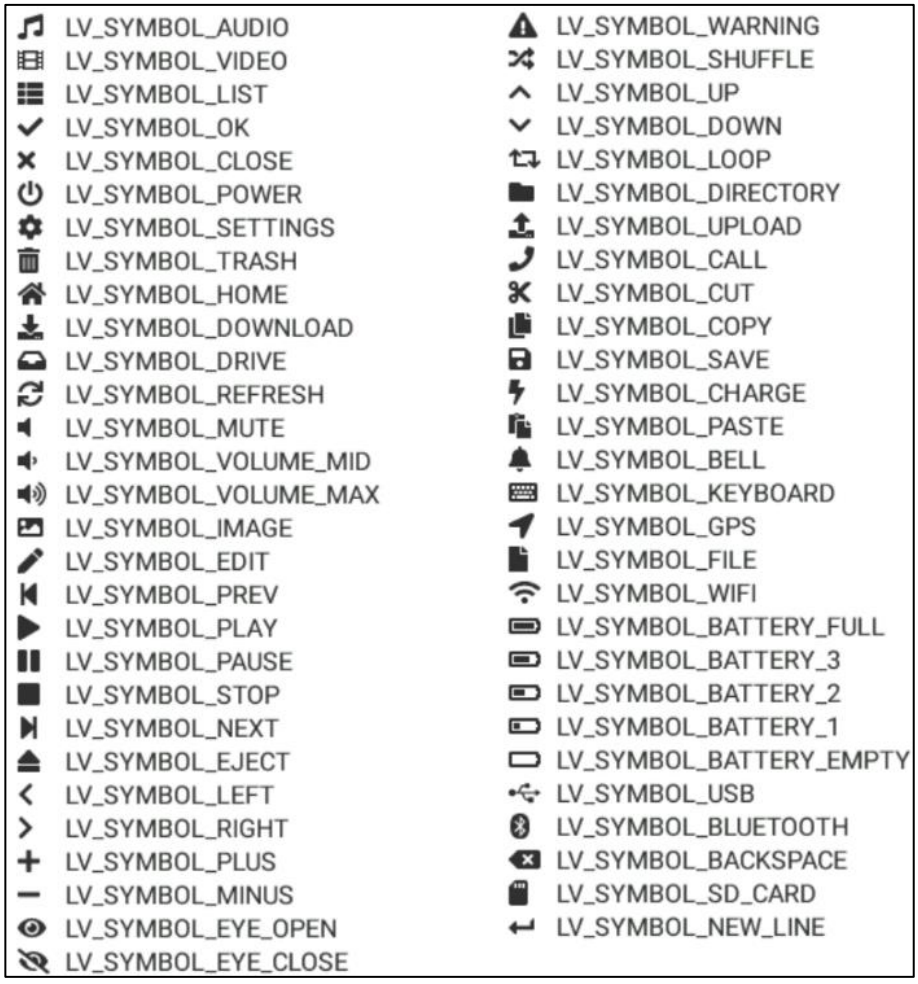


图 8.2.1 LVGL 内置图标字体

图标字体的使用方法，如下源码所示：

```
void lv_mainstart(void)
```

```
{
    lv_obj_t* label = lv_label_create(lv_scr_act());
    lv_label_set_text(label, LV_SYMBOL_AUDIO"AUDIO");
}
```

据上述的代码可知，图标字体可以与字符串连接在一起，如图 8.2.2 所示：



图 8.2.2 显示 LVGL 图标字体

8.3 如何使用 LVGL 内部字库

LVGL 具有内置的字库，这些字库根据 lv_conf.h 文件下的配置项而选择使用。这些字库已经添加到工程当中，打开 Middlewares/lvgl/src/font 分组，如图 8.3.1 所示：

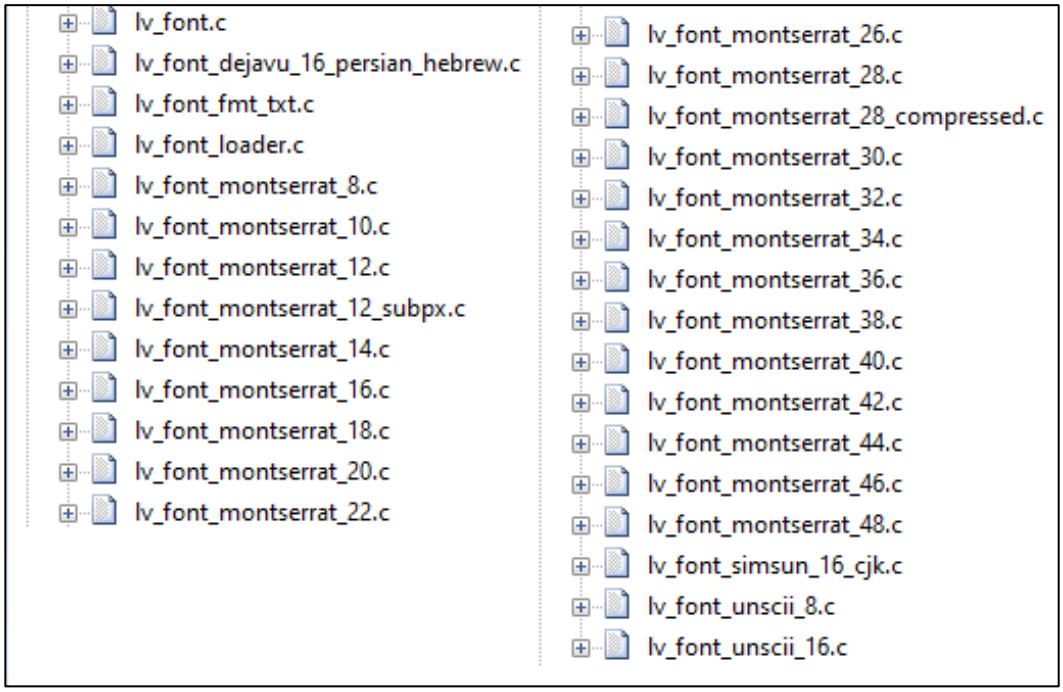


图 8.3.1 LVGL 内置字库

内部字库使用流程，如下所示：

(1) 在 lv_conf.h 中，使能字库，如图 8.3.2 所示：

```
#define LV_FONT_MONTSERRAT_8              0
#define LV_FONT_MONTSERRAT_10             0
#define LV_FONT_MONTSERRAT_12             1
#define LV_FONT_MONTSERRAT_14             1
#define LV_FONT_MONTSERRAT_16             1
#define LV_FONT_MONTSERRAT_18             0
#define LV_FONT_MONTSERRAT_20             0
#define LV_FONT_MONTSERRAT_22             1
#define LV_FONT_MONTSERRAT_24             0
#define LV_FONT_MONTSERRAT_26             0
#define LV_FONT_MONTSERRAT_28             0
#define LV_FONT_MONTSERRAT_30             0
#define LV_FONT_MONTSERRAT_32             1
#define LV_FONT_MONTSERRAT_34             0
#define LV_FONT_MONTSERRAT_36             0
#define LV_FONT_MONTSERRAT_38             0
#define LV_FONT_MONTSERRAT_40             0
#define LV_FONT_MONTSERRAT_42             0
#define LV_FONT_MONTSERRAT_44             0
#define LV_FONT_MONTSERRAT_46             0
#define LV_FONT_MONTSERRAT_48             0
```

图 8.3.2 内部字库使能/失能

(2) 创建 label 部件，调用函数 lv_obj_set_style _text_font/lv_style_set_text_font 指向字库，如下源码所示:

```
void lv_mainstart(void)
{
    lv_obj_t* label = lv_label_create(lv_scr_act());
    lv_obj_set_style_text_font(label,&lv_font_montserrat_16,LV_STATE_DEFAULT);
    lv_label_set_text(label,"Hello ALIENTEK!!!!");
}
```

根据上述的源码可知，如果对象不设置字体，则 LVGL 自动为该对象选择默认字库，如图 8.3.3 所示：



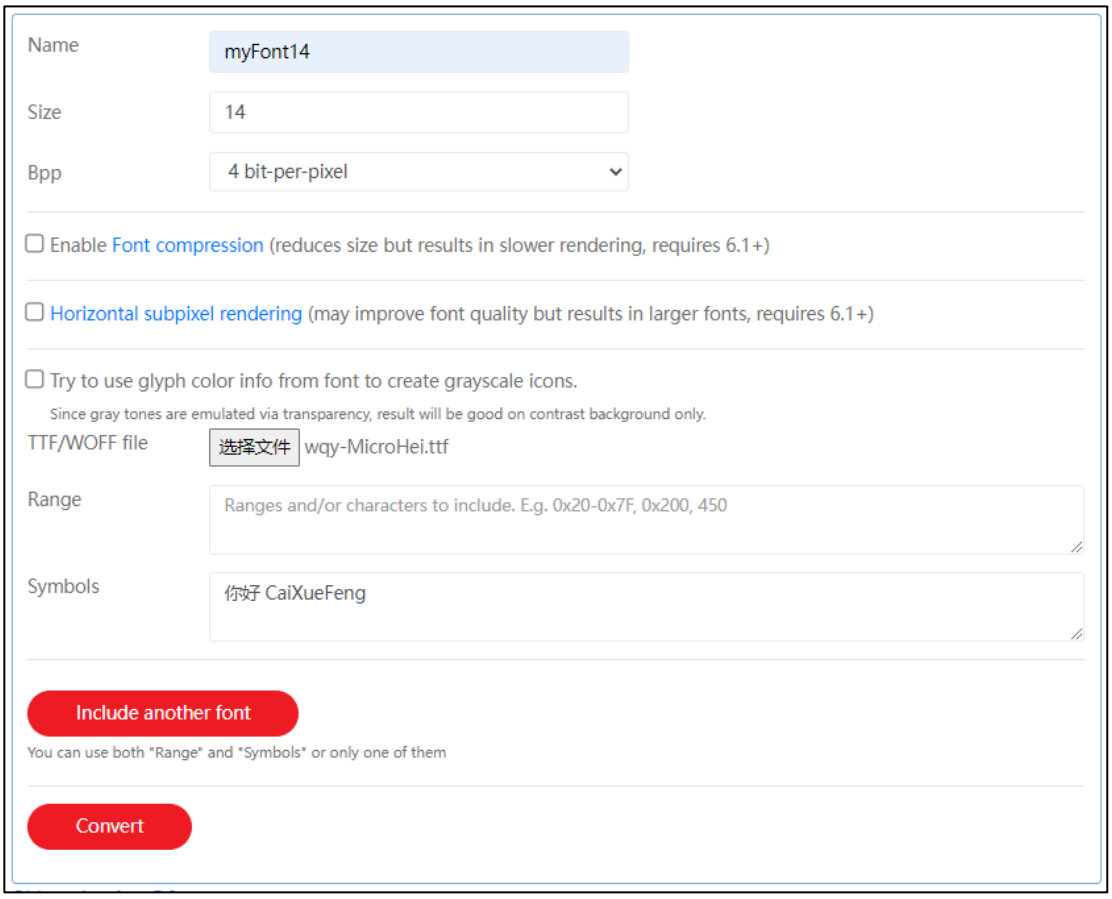
图 8.3.3 应用 LVGL 内置字库

8.4 如何自定义字库

8.4.1 内部字库添加

在上一个小节中，我们了解了 LVGL 内置图标字库和 LVGL 的内置字库等相关知识，这些都是属于 LVGL 内置的字库，如果读者想要添加自己的字库，该如何添加？，下面笔者来介绍一下添加字库的两种方式：

第一种方法：官方提供在线字体转换，在浏览器上传自己的 TTF 字库文件并可以选择范围和文字，然后该软件生成 c 字体文件并添加到工程中，由于该软件是国外的网址，可能需要更多的时间转换。首先我们打开 LVGL 在线字体转换软件，该软件的地址为：https://lvgl.io/tools/fontconverter，如下图所示：



8.4.1.1 设置转换文字

下面笔者分别地讲解上图各个选项的作用，如下所示：

① “Name”：输出字体命名。例如“myfont14”。
② “Size”：字体的高度。
③ “Bpp”设置 bpp (bit-per-piel)，值越高，字体越平滑（抗锯齿）。
④ 启用字体压缩（减小大小但导致渲染速度变慢，需要 6.1+）。
⑤ 水平子像素渲染（可能会提高字体质量，但会产生更大的字体，需要 6.1+）。
⑥ 尝试使用字体中的字形颜色信息来创建灰度图标。
⑦ 选择 TTF 或 WOFF 格式字体文件。
⑧ “Range”：可以同时选择多个字体文件转换，并为其指定范围。
⑨ “Symbols”：转换的符号和文字。
⑩ “Convert”：单击转换按钮。

这里笔者重点讲解一下⑧和⑨的作用，“Range”选项是选择 unicode 的范围，一般是从 0x00 开始，如果我们生成的字库包含常用的中文文字，必须在⑦选项中选择包含中文的字库，然后在⑧选项填入“0x00-0x9fa0”,最后点击“Convert”生成字库(可能需要很长的时间转换)；“Symbols”选项就是指定生成的文字，例如上图中的配置，这里笔者生成了“你好，CaiXueFeng”中文文字，点击“Convert”生成字库即可。

注意：④~⑥是可选择的选项。

第二种方法：首先我们感谢网友 【阿里】对 LVGL 学习提供了很大得帮助，我们可以在他的博客网址下载 http://dz.lfly.xyz/forum.php 字库生成软件，利用网友 【阿里】 提供的免费字体转换软件将中文字库转化为 C 数组文件、 XBF 、 bin 等文件格式。

第一种方法添加字库，笔者就不必多讲了，笔者主要讲解第二种方法，如下步骤所示：

1. 打开 LvglFontTool V0.5 软件，如图 8.4.1.2 所示：



图 8.4.1.2 LvglFontTool0.5 软件主界面

2. 选择字体，如图 8.4.1.3 所示：

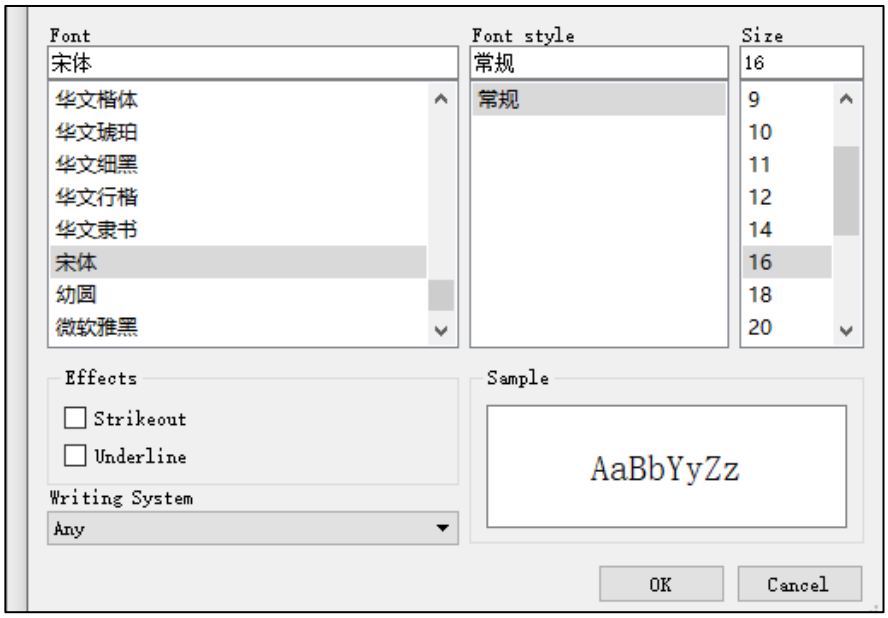


图 8.4.1.3 选择字库

3. 加入汉字，如图 8.4.1.4 所示：

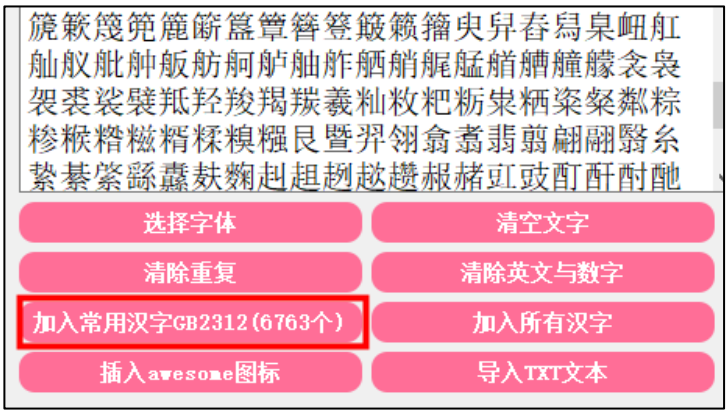


图 8.4.1.4 添加字体

4. 选择版本和类型，如图 8.4.1.5 所示：



图 8.4.1.5 设置转换参数

5. 开始转换和保存文件，我们会得到一个.c 文件，并把该文件移植到我们的工程中，如下图 8.4.1.6 所示：



图 8.4.1.6 移植到工程中

在 LVGL\GUI_APP 路径下创建 GUI_FONT 文件来保存这些.c 文件，然后我们在工程添加一个分组，该分组名称为 Middlewares/LVGL/GUI_FONT。

6. 在 lv_conf.h 文件声明该字库，找到 LV_FONT_CUSTOM_DECLARE 配置项，如下源码所示：

```
#define LV_FONT_CUSTOM_DECLARE    LV_FONT_DECLARE(myFont24)
```

经过上述步骤，我们可以使用自定义字库了，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* label = lv_label_create(lv_scr_act());
    lv_obj_set_style_text_font(label,&myFont14,LV_STATE_DEFAULT);
    lv_label_set_text(label,"Hello ALIENTEK!!!!");
}
```

根据上述的源码可知，如果对象不设置字库，则该对象选择 LVGL 的默认字库，运行效果如下图所示：



图 8.4.1.5 应用自定义字库

请参考“LVGL 例程 6 LVGL 内部字库读取”实验。

8.4.2 外部字库添加

上小节中，笔者已经讲述了 LVGl 自定义内部字库的方法，而现在笔者将讲解 LVGL 外部添加字库的方法。下面来看一下外部字库添加的方式如下步骤所示：

1. 打开 LvglFontTool 软件，如图 8.4.2.1 所示：



图 8.4.2.1 LvglFontTool 软件主界面

2. 选择字体，如图 8.4.2.2 所示：

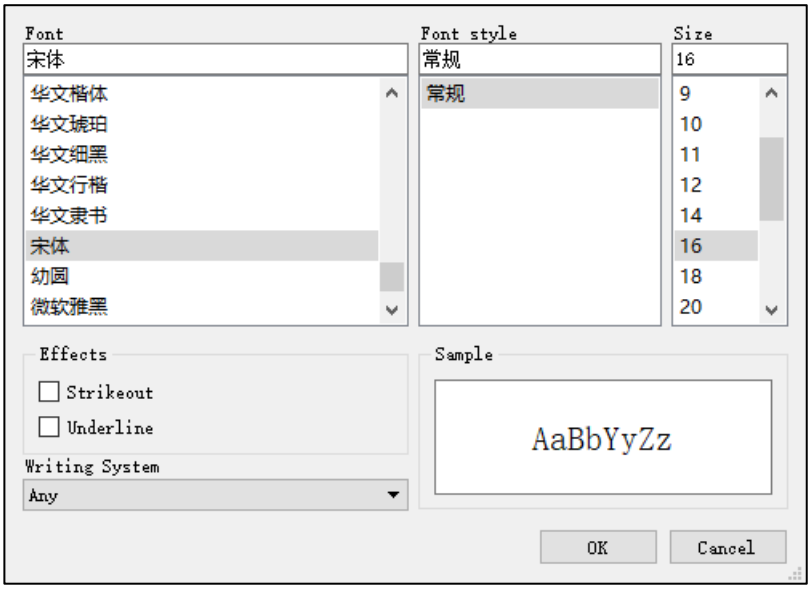


图 8.4.2.2 选择字库

3. 加入汉字，如图 8.4.2.3 所示：

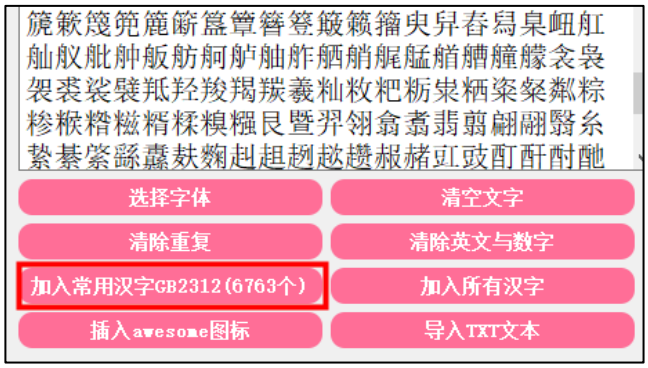


图 8.4.2.3 添加字体

4. 设置字库的参数，如图 8.4.2.4 所示：

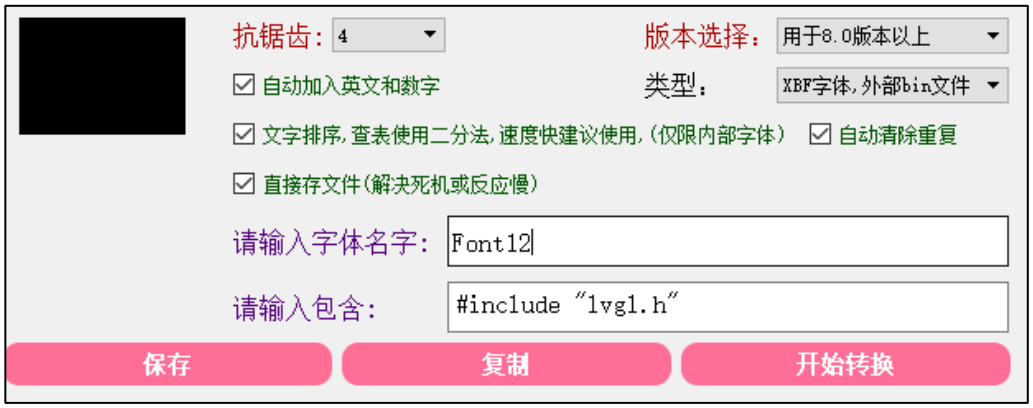


图 8.4.2.4 添加外部字库文件

5. 开始转换，我们会得到两个文件，一个文件是.c 文件，另一个是.bin 文件，我们把.c 文件移植到我们的工程当中，而.bin 文件放到我们的 SD 卡上。如图 8.4.2.5 所示：

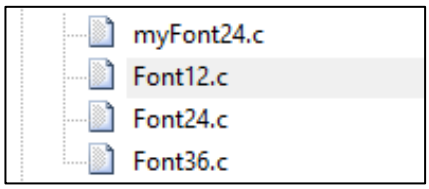


图 8.4.2.5 工程添加文件

在 LVGL\GUI_APP 路径下创建 GUI_FONT 文件来保存这些.c 文件，然后我们在工程添加一个分组，该分组名称为 Middlewares/LVGL/GUI_FONT。

6. 打开 Font12.c 文件，修改函数__user_font_getdata()，如下源码所示：

```
static uint8_t *__user_font_getdata(int offset, int size){
 /* 如字模保存在 SPI FLASH, SPIFLASH_Read(__g_font_buf,offset,size);
如字模已加载到 SDRAM,直接返回偏移地址即可如:return (uint8_t*)(sdram_fontddr+offset); */
    norflash_ex_read(__g_font_buf,ftinfo.lvgl_12addr +offset,size);
    return __g_font_buf;
}
```

注意：上述的源码是以 Mini Pro H750 开发板为例的，所以请用户根据自己的开发板例程来修改。

7. 添加正点原子的 TEXT 文件，如图 8.4.2.6 所示：

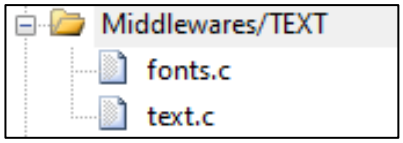


图 8.4.2.6 添加 TEXT 文件

8. 修改 fonts.h 文件，如下源码所示：

```
/* 字体信息保存首地址
 * 占 41 个字节,第 1 个字节用于标记字库是否存在.后续每 8 个字节一组,分别保存起始地址和文件大小
 */
extern uint32_t FONTINFOADDR;

/* 字库信息结构体定义
```

```
 * 用来保存字库基本信息，地址，大小等
 */
typedef __PACKED_STRUCT
{
    uint8_t fontok;                 /* 字库存在标志，0XAA，字库正常；其他，字库不存在 */
    uint32_t ugbkaddr;              /* unigbk 的地址 */
    uint32_t ugbksize;              /* unigbk 的大小 */
    uint32_t f12addr;               /* gbk12 地址 */
    uint32_t gbk12size;             /* gbk12 的大小 */
    uint32_t f16addr;               /* gbk16 地址 */
    uint32_t gbk16size;             /* gbk16 的大小 */
    uint32_t f24addr;               /* gbk24 地址 */
    uint32_t gbk24size;             /* gbk24 的大小 */
    uint32_t f32addr;               /* gbk32 地址 */
    uint32_t gbk32size;             /* gbk32 的大小 */

    uint32_t lvgl_12addr;           /* LVGL12 地址 */
    uint32_t lvgl_12size;           /* LVGL12 的大小 */
    uint32_t lvgl_24addr;           /* LVGL24 地址 */
    uint32_t lvgl_24size;           /* LVGL24 的大小 */
    uint32_t lvgl_36addr;           /* LVGL36 地址 */
    uint32_t lvgl_36size;           /* LVGL36 的大小 */

} _font_info;
```

这里笔者在_font_info 结构体添加了 lvgl_12addr、lvgl_24addr、lvgl_36addr 成员变量。

9. 修改 fonts.c 文件中的 FONT_GBK_PATH 和 FONT_UPDATE_REMIND_TBL 数组，如下源码所示：

```
/* 字库存放在磁盘中的路径 */
char *const FONT_GBK_PATH[8] =
{
    "/SYSTEM/FONT/UNIGBK.BIN",          /* UNIGBK.BIN 的存放位置 */
    "/SYSTEM/FONT/GBK12.FON",           /* GBK12 的存放位置 */
    "/SYSTEM/FONT/GBK16.FON",           /* GBK16 的存放位置 */
    "/SYSTEM/FONT/GBK24.FON",           /* GBK24 的存放位置 */
    "/SYSTEM/FONT/GBK32.FON",           /* GBK32 的存放位置 */
    "/SYSTEM/LVFONT/Font12.BIN",        /* Font12 的存放位置 */
    "/SYSTEM/LVFONT/Font24.BIN",        /* Font24 的存放位置 */
    "/SYSTEM/LVFONT/Font36.BIN",        /* Font36 的存放位置 */
};

/* 更新时的提示信息 */
char *const FONT_UPDATE_REMIND_TBL[8] =
{
```

```
    "Updating UNIGBK.BIN",              /* 提示正在更新 UNIGBK.bin */
    "Updating GBK12.FON ",              /* 提示正在更新 GBK12 */
    "Updating GBK16.FON ",              /* 提示正在更新 GBK16 */
    "Updating GBK24.FON ",              /* 提示正在更新 GBK24 */
    "Updating GBK32.FON ",              /* 提示正在更新 GBK32 */
    "Updating Font12.BIN",              /* 提示正在更新 Font12 */
    "Updating Font24.BIN",              /* 提示正在更新 Font24 */
    "Updating Font36.BIN",              /* 提示正在更新 Font36 */
};
```

笔者在这两个数组上只添加 Font12.BIN、Font26.BIN、Font36.BIN 字符串。

10. 修改 fonts.c 文件中的函数 fonts_update_fontx，如下源码所示：

```
/**
 * @brief       更新某一个字库
 * @param       x, y    : 提示信息的显示地址
 * @param       size    : 提示信息字体大小
 * @param       fpath   : 字体路径
 * @param       fx      : 更新的内容
 *   @arg                 0, ungbk;
 *   @Arg                 1, gbk12;
 *   @arg                 2, gbk16;
 *   @arg                 3, gbk24;
 *   @arg                 4, gbk32;
 * @param       color   : 字体颜色
 * @retval      0, 成功; 其他, 错误代码;
 */
static uint8_t fonts_update_fontx( uint16_t x,
                                   uint16_t y,
                                   uint8_t size,
                                   uint8_t *fpath,
                                   uint8_t fx,
                                   uint16_t color)
{
    uint32_t flashaddr = 0;
    FIL *fftemp;
    uint8_t *tempbuf;
    uint8_t res;
    uint16_t bread;
    uint32_t offx = 0;
    uint8_t rval = 0;
    fftemp = (FIL *)mymalloc(SRAMIN, sizeof(FIL));  /* 分配内存 */

    if (fftemp == NULL)rval = 1;
```

```
    tempbuf = mymalloc(SRAMIN, 4096);                   /* 分配 4096 个字节空间 */

    if (tempbuf == NULL)rval = 1;

    res = f_open(fftemp, (const TCHAR *)fpath, FA_READ);

    if (res)rval = 2;   /* 打开文件失败 */

    if (rval == 0)
    {
        switch (fx)
        {
            case 0: /* 更新 UNIGBK.BIN */
                 /* 信息头之后，紧跟 UNIGBK 转换码表 */
                ftinfo.ugbkaddr = FONTINFOADDR + sizeof(ftinfo);
                ftinfo.ugbksize = fftemp->obj.objsize;      /* UNIGBK 大小 */
                flashaddr = ftinfo.ugbkaddr;
                break;

            case 1: /* 更新 GBK12.BIN */
                /* UNIGBK 之后，紧跟 GBK12 字库 */
                ftinfo.f12addr = ftinfo.ugbkaddr + ftinfo.ugbksize;
                ftinfo.gbk12size = fftemp->obj.objsize;    /* GBK12 字库大小 */
                flashaddr = ftinfo.f12addr;                /* GBK12 的起始地址 */
                break;

            case 2: /* 更新 GBK16.BIN */
                 /* GBK12 之后，紧跟 GBK16 字库 */
                ftinfo.f16addr = ftinfo.f12addr + ftinfo.gbk12size;
                ftinfo.gbk16size = fftemp->obj.objsize;    /* GBK16 字库大小 */
                flashaddr = ftinfo.f16addr;                /* GBK16 的起始地址 */
                break;

            case 3: /* 更新 GBK24.BIN */
                 /* GBK16 之后，紧跟 GBK24 字库 */
                ftinfo.f24addr = ftinfo.f16addr + ftinfo.gbk16size;
                ftinfo.gbk24size = fftemp->obj.objsize;    /* GBK24 字库大小 */
                flashaddr = ftinfo.f24addr;                /* GBK24 的起始地址 */
                break;

            case 4: /* 更新 GBK32.BIN */
                 /* GBK24 之后，紧跟 GBK32 字库 */
                ftinfo.f32addr = ftinfo.f24addr + ftinfo.gbk24size;
```

```
                ftinfo.gbk32size = fftemp->obj.objsize;     /* GBK32 字库大小 */
                flashaddr = ftinfo.f32addr;                 /* GBK32 的起始地址 */
                break;
            case 5:/* 更新 LVGL12.BIN */                          (1)
                ftinfo.lvgl_12addr=ftinfo.f32addr+ftinfo.gbk32size;
                ftinfo.lvgl_12size=fftemp->obj.objsize;
                flashaddr=ftinfo.lvgl_12addr;
                break;
            case 6:/* 更新 LVGL24.BIN */                          (2)
                ftinfo.lvgl_24addr=ftinfo.lvgl_12addr+ftinfo.lvgl_12size;
                ftinfo.lvgl_24size=fftemp->obj.objsize;
                flashaddr=ftinfo.lvgl_24addr;
                break;
            case 7:/* 更新 LVGL36.BIN */                          (3)
                ftinfo.lvgl_36addr=ftinfo.lvgl_24addr+ftinfo.lvgl_24size;
                ftinfo.lvgl_36size=fftemp->obj.objsize;
                flashaddr=ftinfo.lvgl_36addr;
                break;
        }

        while (res == FR_OK)    /* 死循环执行 */
        {
            res = f_read(fftemp, tempbuf, 4096, (UINT *)&bread); /* 读取数据 */

            if (res != FR_OK)break;      /* 执行错误 */
            /* 从 0 开始写入 bread 个数据 */
            norflash_ex_write(tempbuf, offx + flashaddr, bread);
            offx += bread;
            /* 进度显示 */
            fonts_progress_show(x, y, size, fftemp->obj.objsize, offx, color);

            if (bread != 4096)break;     /* 读完了. */
        }

        f_close(fftemp);
    }

    myfree(SRAMIN, fftemp);      /* 释放内存 */
    myfree(SRAMIN, tempbuf);     /* 释放内存 */
    return res;
}
```

请看上述源码中的 (1)~(3)程序，这些程序是笔者添加上去的。

10. 修改 fonts.c 文件中的函数 fonts_update_font，如下源码所示：

```
/**
 * @brief       更新字体文件
 *   @note      所有字库一起更新(UNIGBK,GBK12,GBK16,GBK24,GBK32)
 * @param       x, y    : 提示信息的显示地址
 * @param       size    : 提示信息字体大小
 * @param       src     : 字库来源磁盘
 *   @arg                 "0:", SD 卡;
 *   @Arg                 "1:", FLASH 盘
 *   @arg                 "2:", U 盘
 * @param       color   : 字体颜色
 * @retval      0, 成功; 其他, 错误代码;
 */
uint8_t fonts_update_font( uint16_t x,
                           uint16_t y,
                           uint8_t size,
                           uint8_t *src,
                           uint16_t color)
{
    uint8_t *pname;
    uint32_t *buf;
    uint8_t res = 0;
    uint16_t i, j;
    FIL *fftemp;
    uint8_t rval = 0;
    res = 0XFF;
    ftinfo.fontok = 0XFF;
    pname = mymalloc(SRAMIN, 100);  /* 申请 100 字节内存 */
    buf = mymalloc(SRAMIN, 4096);   /* 申请 4K 字节内存 */
    fftemp = (FIL *)mymalloc(SRAMIN, sizeof(FIL));  /* 分配内存 */

    if (buf == NULL || pname == NULL || fftemp == NULL)
    {
        myfree(SRAMIN, fftemp);
        myfree(SRAMIN, pname);
        myfree(SRAMIN, buf);
        return 5;   /* 内存申请失败 */
    }
   /* 先查找文件 UNIGBK,GBK12,GBK16,GBK24,GBK32,
      LVGL12.BIN,LVGL24.BIN,LVGL36BIN 是否正常 */
    for (i = 0; i < 8; i++)                     (1)
    {
        strcpy((char *)pname, (char *)src);          /* copy src 内容到 pname */
        strcat((char *)pname, (char *)FONT_GBK_PATH[i]); /* 追加具体文件路径 */
```

```
        res = f_open(fftemp, (const TCHAR *)pname, FA_READ);/* 尝试打开 */

        if (res)
        {
            rval |= 1 << 7;   /* 标记打开文件失败 */
            break;            /* 出错了,直接退出 */
        }
    }

    myfree(SRAMIN, fftemp); /* 释放内存 */

    if (rval == 0)          /* 字库文件都存在. */
    {
        /* 提示正在擦除扇区 */
        lcd_show_string(x, y, 240, 320, size, "Erasing sectors... ", color);

        for (i = 0; i < FONTSECSIZE; i++)    /* 先擦除字库区域,提高写入速度 */
        {
          /* 进度显示 */
          fonts_progress_show(x + 20 * size / 2, y, size, FONTSECSIZE,
                              i, color);
          /* 读出整个扇区的内容 */
          norflash_ex_read((uint8_t *)buf, ((FONTINFOADDR /
                           4096)+i) * 4096, 4096);

            for (j = 0; j < 1024; j++)           /* 校验数据 */
            {
                if (buf[j] != 0XFFFFFFFF)break; /* 需要擦除 */
            }

            if (j != 1024)
            {
                 /* 需要擦除的扇区 */
                norflash_ex_erase_sector((FONTINFOADDR / 4096) + i);
            }
        }
        /* 依次更新 UNIGBK,GBK12,GBK16,GBK24,GBK32,
           LVGL12.BIN,LVGL24.BIN,LVGL36BIN */
        for (i = 0; i < 8; i++)
        {
            lcd_show_string(x, y, 240, 320, size,
                            FONT_UPDATE_REMIND_TBL[i], color);
                           strcpy((char *)pname, (char *)src);
```

```
            strcat((char *)pname, (char *)FONT_GBK_PATH[i]);
            res = fonts_update_fontx(x + 20 * size / 2, y, size,
                                     pname, i, color);

            if (res)
            {
                myfree(SRAMIN, buf);
                myfree(SRAMIN, pname);
                return 1 + i;
            }
        }

        /* 全部更新好了 */
        ftinfo.fontok = 0XAA;
        /* 保存字库信息 */
        norflash_ex_write((uint8_t *)&ftinfo, FONTINFOADDR, sizeof(ftinfo));
    }

    myfree(SRAMIN, pname);  /* 释放内存 */
    myfree(SRAMIN, buf);
    return rval;            /* 无错误. */
}
```

请看上述源码中的(1)程序，这里就是笔者修改的位置。

11. 在 lv_conf.h 文件声明该文件，找到 LV_FONT_CUSTOM_DECLARE 配置项，如下源码所示：

```
#define LV_FONT_CUSTOM_DECLARE  LV_FONT_DECLARE(Font12)
```

12. 编写测试代码：

```
void lv_mainstart(void)
{
    lv_obj_t* label = lv_label_create(lv_scr_act());
    lv_obj_set_style_text_font(label,& Font12,LV_STATE_DEFAULT);
    lv_label_set_text(label,"Hello ALIENTEK!!!!");
}
```

根据上述的源码可知，如果对象不设置字库，则该对象选择 LVGL 的默认字库，如图 8.4.2.7 所示：



图 8.4.2 应用外部字库

请参考“LVGL 例程 7 LVGL SPI Flash 读取 XBF 字库”实验。注意：DMF407、Mini、战舰、探索者以及精英不支持本实验。

8.4.3 内部读取 TTF 字库

内部读取 TTF 字库，无非就是把 TTF 字体转成 C 数组的形式，然后系统进行读取。下面笔者讲解一下内部 TTF 字库的生成步骤如下所示：

1. 打开软件 LvglFontTool_V0.4，该软件也是 LvglFontTool_V0.5 软件的研发者，这个软件可以生成 TTF 字库，如图 8.4.3.1 所示：

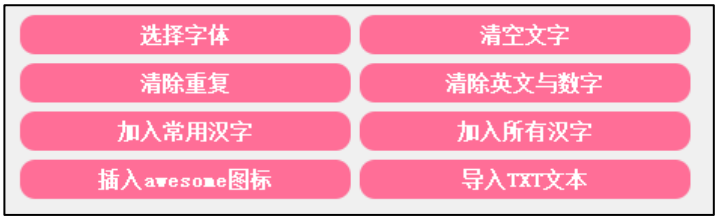


图 8.4.3.1 LvglFontTool_V0.4 主界面

2. 选择字体，选择 TTF 字库，如图 8.4.3.2 所示：

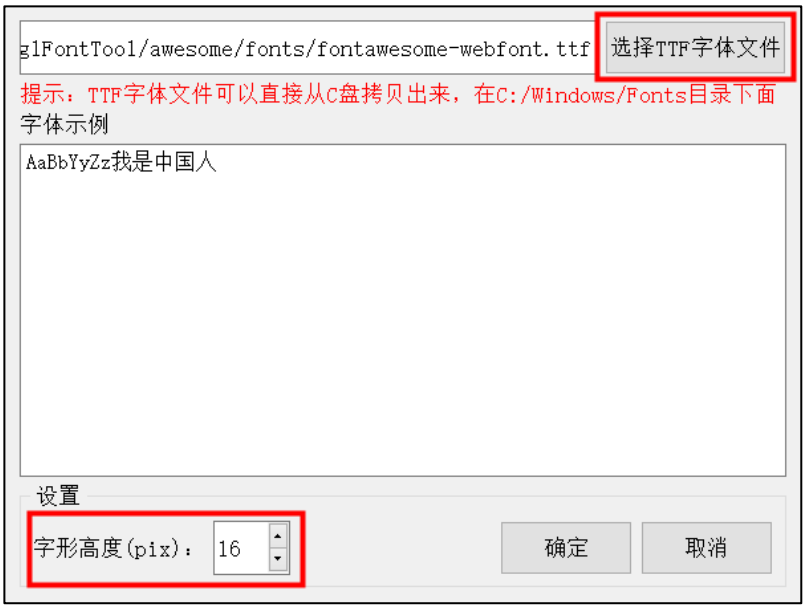


图 8.4.3.2 选择字体大小和字库

3. 添加常用的汉字，如图 8.4.3.3 所示：



图 8.4.3.3 加入字库

4. 开始转换并保存文件，如图 8.4.3.4 所示：



图 8.4.3.4 开始转换字体

5. 工程实现步骤，这里笔者就不必添加了，和内部字库添加的方法是一样的。

8.4.4 外部读取 TTF 字库

外部添加 TTF 字库是使用文件系统读取 SD 卡上的 TTF 字库，所以我们必须把 LVGL 文件系统开启，首先我们需要移植 FreeType 的库，Freeteye 移植库在网址 https://www.freetype.org/ 下载，笔者使用正点原子的 emWin 的 TTF 字体显示实验中的 TRUETYPE 文件夹中的 Freeteye 移植库，我们只需把该文件夹里面的 GUI_TTF.c 文件去除即可。

接下来我们分别地介绍 TTF 字库的移植过程，如下步骤所示：

1. 把 TRUETYPE 文件夹修改成 GUI_TRUETYPE 文件名并保存到 Middlewares/LVGL/GUI_APP 目录下，如图 8.4.4.1 所示：



图 8.4.4.1 添加文件 GUI_TRUETYPE

上图中 GUI_freetype 文件夹里面的文件 lv_font_ttf.c 和.h 文件，这两个文件并没有使用官方提供的 V8 版本的 lv_font_ttf.c 和.h 文件，这两个文件是 V7 版本的 TTF 字库读取接口文件，V8 所提供的 TTF 字库读取接口文件，笔者试过，并不成功，所以笔者使用 V7 版本改编过来的文件。

2. 添加 Middlewares/LVGL/GUI_TRUEPYTE 分组添加以下文件，如图 8.4.4.2 所示：

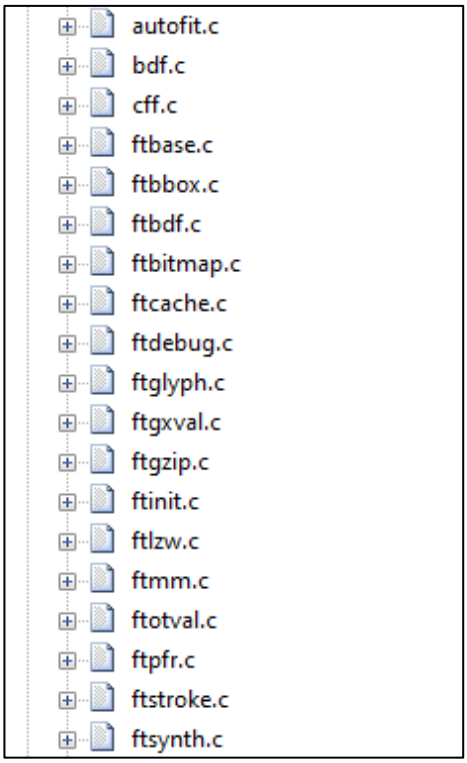


图 8.4.4.2 Middlewares/LVGL_GUI_TRUEPYTE 工程管理项添加文件

3. 添加 Middlewares/LVGL/GUI_FREEPYTE 分组添加 lv_font_ttf.c 文件，如图 8.4.4.3 所示：



图 8.4.4.3 Middlewares/LVGL/GUI_FREEPYTE 工程管理项添加文件

4. 添加头文件路径，如图 8.4.4.4 所示：

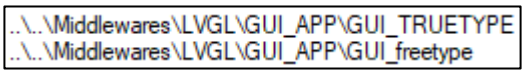


图 8.4.4.4 添加文件路径

5. 修改 lv_font_ttf.c 和 lv_font_ttf.h 文件，如以下步骤所示：

(1) 在 lv_font_ttf.h 文件定义 ttf 字体文件的大小(字节)，如下源码所示：

```
/**
 * ttf 字体文件的大小(字节)
 */
#define TTF_SIZE_MEMORY 30*1024
```

上述的宏定义是根据你的 TTF 字库文件定义的，必须大于或者等于 TTF 字库的大小。

(2) 在 lv_font_ttf.c 文件定义 ttf 字体文件的存储空间，如下源码所示：

```
/**
 * ttf 字体文件的存储空间
 */
uint8_t  ttf_memory[TTF_SIZE_MEMORY];
```

(3) 编写加载 TTF 字库程序，如下源码所示：

```
/**
 * @brief       加载 TTF 字库
 * @param       pash : 字库路径
 * @retval      无
 */
void lv_load_ttffont(const char * pash)
{
    int result;
    lv_fs_file_t TTFFontFile;
    uint32_t bread;
    result = lv_fs_open(&TTFFontFile, pash, LV_FS_MODE_RD);  /* 打开字库文件 */

    if (result != LV_FS_RES_OK)
    {
        printf("打开字库文件失败\r\n");
    }
     /* 读取数据 */
     result = lv_fs_read(&TTFFontFile,(char *)ttf_memory,
     sizeof(ttf_memory), (unsigned int *)&bread);

    if (result != LV_FS_RES_OK)
    {
```

```
        printf("文件读取失败\r\n"); /* 文件打开失败 */
    }

    lv_fs_close(&TTFFontFile); /* 关闭 TTFFointFile 文件*/
}
```

(4) 修改 lv_freetype_font_init 函数，如下源码所示：

```
/**
* @brief 初始化字体结构体
* @param font 指向字体的指针
* @param font_path 字体路径
* @param font_size 字体高度
* @retval FT_Error
*/
int lv_freetype_font_init(lv_font_t * font,
                          const char * font_path,
                          uint16_t font_size)
{
    FT_Error error;
    lv_load_ttffont(font_path);

    lv_font_fmt_freetype_dsc_t * dsc =
                              lv_mem_alloc(sizeof(lv_font_fmt_freetype_dsc_t));
    LV_ASSERT_MEM(dsc);
    if(dsc == NULL) return FT_Err_Out_Of_Memory;

    dsc->font_size = font_size;

//    error = FT_New_Face(library, font_path, 0, &dsc->face);
//
//    if ( error )
//    {
//        printf("Error in FT_New_Face: %d\n", error);
//        return error;
//    }
//    error = FT_Set_Pixel_Sizes(dsc->face, 0,font_size);
//    if ( error )
//    {
//        printf("Error in FT_Set_Char_Size: %d\n", error);
//        return error;
//    }

    error = FT_Init_FreeType(&library);
```

```
    if(error)
    {
        printf("Failed to init FreeType library: %d\n", error);
        goto err1;
    }

    error = FT_New_Memory_Face(library, ttf_memory, TTF_SIZE_MEMORY, 0,
                               &dsc->face);
    if(error)
    {
        printf("Failed to open font file: %d\n", error);
        goto err2;
    }

    error = FT_Set_Pixel_Sizes(dsc->face, 0, font_size);

    if(error)
    {
        printf("Failed to Set Pixel Sizes: %d\n", error);
        goto err3;
}



#if USE_CACHE_MANGER
    /* 设置回调以获取关于字形的信息 */
    font->get_glyph_dsc = get_glyph_dsc_cache_cb;
    /* 设置回调函数来获取 glyp 的位图 */
    font->get_glyph_bitmap = get_glyph_bitmap_cache_cb;
#else
    font->get_glyph_dsc = get_glyph_dsc_cb;       /* 设置回调以获取关于字形的信息 */
    font->get_glyph_bitmap = get_glyph_bitmap_cb;/* 设置回调函数来获取 glyp 的位图 */
#endif




#ifndef LV_USE_USER_DATA
#error "lv_freetype : user_data is required.Enable it
lv_conf.h(LV_USE_USER_DATA 1)"
#endif
    font->user_data = dsc;
    font->line_height = (dsc->face->size->metrics.height >> 6);
```

```
    /* 基线从 line - height 的顶部开始测量 */
    font->base_line = -(dsc->face->size->metrics.descender >> 6);
    font->subpx = LV_FONT_SUBPX_NONE;

    return FT_Err_Ok;




err4:
    FTC_Manager_Done(cache_manager);
err3:
    FT_Done_Face(dsc->face);
err2:
    FT_Done_FreeType(library);
err1:
    return -1;
}
```

(5) 在 lv_font_ttf.h 文件中找到宏定义 USE_CACHE_MANGER，把这个宏定义设置为 1。

经过上述的五步，我们已经完成 TTF 字库的移植，下面我们来讲解如何使用 LVGL 的 Freetype 的函数，如下源码所示：

```
#include "lv_font_ttf.h"
lv_font_t lv_font_ttf_36;

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_freetype_font_init(&lv_font_ttf_36,
                          "0:/SYSTEM/EMWINFONT/TTF/comesinhandy.ttf", 36);
    lv_obj_t * label_36 = lv_label_create(lv_scr_act());
    lv_obj_set_style_text_font(label_36, &lv_font_ttf_36, 0);
    lv_obj_set_style_text_color(label_36, lv_color_black(), 0);
    lv_label_set_text(label_36, "ALIENTEK LVGL TTF Font comesinhandy.ttf!");
    lv_obj_set_pos(label_36, 0, 0);
}
```

上述的源码请参考工程“LVGL 例程 8 TTF 字库”，实验，注意：DMF407、Mini、战舰、探索者以及精英不支持本实验。

部件篇

经过前面的学习，我们已经对 LVGL 基础知识具有一定的了解，接下来，我们开始学习 LVGL 部件的使用，部件是一个具有图形表示能力的对象，可在屏幕上显示，并可与用户进行交互。比如典型图形用户界面中的按钮、复选框和滚动条等都是部件示例。

在 LVGL 中常用的部件如 lv_label、lv_btn、lv_checkbox、lv_textarea 等三十多种部件，接下来，我们开始学习 LVGL 部件的知识且使用。

第九章 基础对象

“基础对象”是实现屏幕上小部件的基本属性，例如位置、大小和父类与子类等属性，它是按照面向对象思维方式继承了 LVGL 所有部件的基类，简单来说：设置所有部件的位置、大小等属性都是归基础对象管理的。

本章分为以下几个部分：

9.1 基础对象的角色定位
9.2 基础对象的相关知识
9.3 基础对象的 API 函数
9.4 基础对象部件的实验

9.1 基础对象的角色定位

基础对象(lv_obj)是调用函数 lv_obj_create 创建的，在第六章节中我们已经了解过这个函数，这里我们无需重复讲解了。基础对象被创建时，它是以一个圆角矩阵展示在屏幕上，如图 9.1.1 所示：

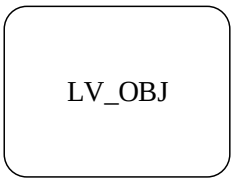


图 9.1.1 创建基础对象

1. 基础对象(lv_obj)在 UI 的作用：

在 UI 设计中，基础对象(lv_obj)一般作为容器使用，当然不排除其他使用，例如制作简单的正方形等形状，这些需要根据自己的 UI 而定，笔者一般使用基础对象作为容器来使用。例如，在多界面切换时，每一个界面都是一个容器，该容器是所有部件的父类。切换界面时，只需切换容器显示即可，如图 9.1.2 所示：

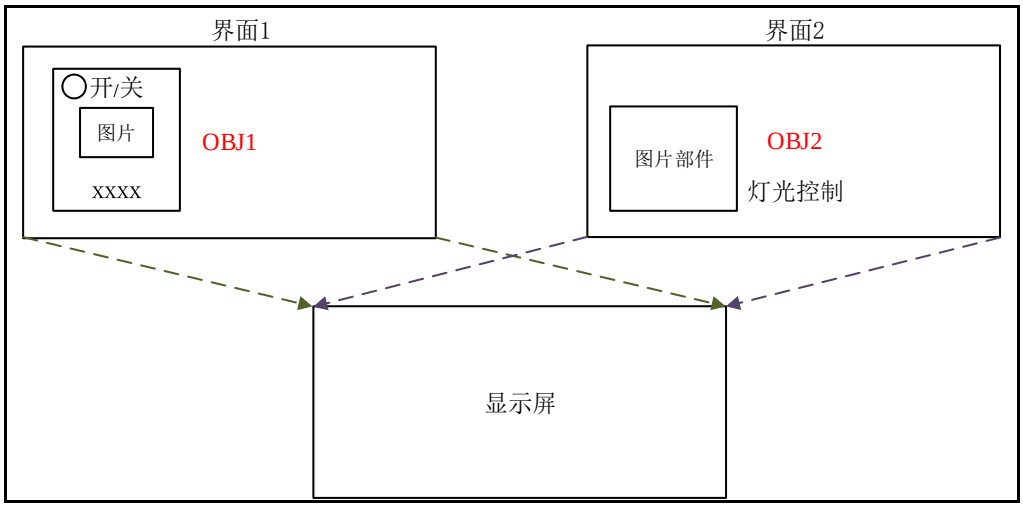


图 9.1.2 多界面切换原理

2. 界面切换原理：

前面笔者已经讲解了，基础对象(lv_obj)可以用作于容器使用，那么涉及多界面切换的方法有哪些呢？，如以下所示：

1. 删除法：

在第六章中我们已经讲解过，如果删除一个对象，则调用函数 lv_obj_del 删除对象，该函数可以把对象的子类一并删除，所以切换界面时，可以使用这样的方式，如图 9.1.3 所示：

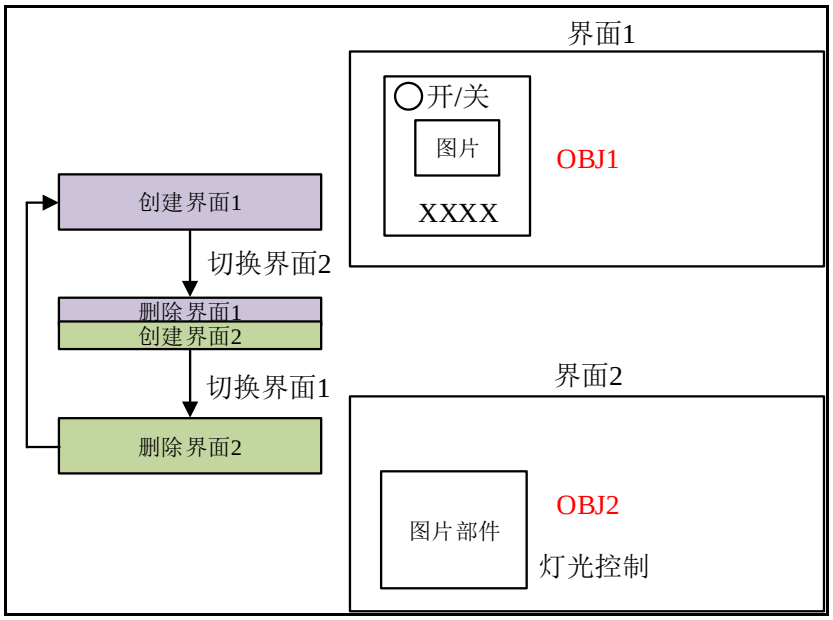


图 9.1.3 删除法切换界面

2. 隐蔽法：

隐蔽法主要是调用 lv_obj_add_flag 和 lv_obj_clear_flag 函数完成，如图 9.1.4 所示：

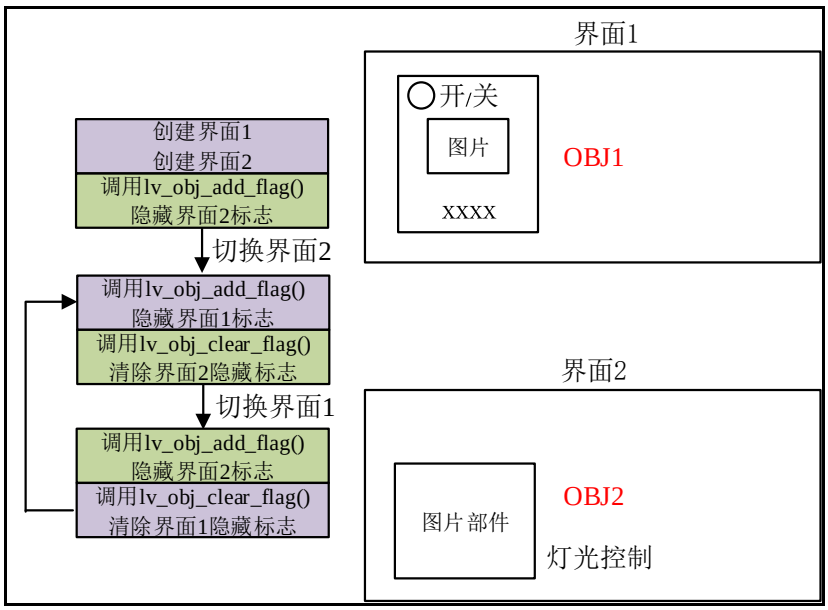


图 9.1.4 隐蔽法切换界面

隐蔽法会消耗更多的内存空间，因为这些界面一直占用着内存空间，所以隐蔽法消耗的内存会更多。

1. lv_obj_add_flag 函数

设置一个或多个标志，其函数原型如以下所示：

```
void lv_obj_add_flag(lv_obj_t *obj, lv_obj_flag_t f)
```

该函数的形参，如表 9.1.5 所示：

参数	描述
obj	指向对象的指针
f	R-ed 值从 lv_obj_flag_t 设置

表 9.1.5 函数 lv_obj_add_flag()形参描述

返回值：无。

2. lv_obj_clear_flag 函数

清除一个或多个标志，其函数原型如以下所示：

```
void lv_obj_clear_flag(lv_obj_t *obj, lv_obj_flag_t f)
```

该函数的形参，如表 9.1.6 所示：

参数	描述
obj	指向对象的指针
f	R-ed 值从 lv_obj_flag_t 设置

表 9.1.6 函数 lv_obj_clear_flag()形参描述

返回值：无。

lv_obj_flag_t 结构体如下源码所示：

```
enum {
    LV_OBJ_FLAG_HIDDEN          = (1L << 0),
    LV_OBJ_FLAG_CLICKABLE       = (1L << 1),
    LV_OBJ_FLAG_CLICK_FOCUSABLE = (1L << 2),
    LV_OBJ_FLAG_CHECKABLE       = (1L << 3),
    LV_OBJ_FLAG_SCROLLABLE      = (1L << 4),
    LV_OBJ_FLAG_SCROLL_ELASTIC  = (1L << 5),
    LV_OBJ_FLAG_SCROLL_MOMENTUM = (1L << 6),
    LV_OBJ_FLAG_SCROLL_ONE      = (1L << 7),
    LV_OBJ_FLAG_SCROLL_CHAIN_HOR = (1L << 8),
    LV_OBJ_FLAG_SCROLL_CHAIN_VER = (1L << 9),
    LV_OBJ_FLAG_SCROLL_CHAIN     = (LV_OBJ_FLAG_SCROLL_CHAIN_HOR |
                                    LV_OBJ_FLAG_SCROLL_CHAIN_VER),
    LV_OBJ_FLAG_SCROLL_ON_FOCUS = (1L << 10),
    LV_OBJ_FLAG_SCROLL_WITH_ARROW  = (1L << 11),
    LV_OBJ_FLAG_SNAPPABLE       = (1L << 12),
    LV_OBJ_FLAG_PRESS_LOCK      = (1L << 13),
    LV_OBJ_FLAG_EVENT_BUBBLE    = (1L << 14),
    LV_OBJ_FLAG_GESTURE_BUBBLE  = (1L << 15),
    LV_OBJ_FLAG_ADV_HITTEST     = (1L << 16),
    LV_OBJ_FLAG_IGNORE_LAYOUT   = (1L << 17),
    LV_OBJ_FLAG_FLOATING        = (1L << 18),
    LV_OBJ_FLAG_OVERFLOW_VISIBLE = (1L << 19),

    LV_OBJ_FLAG_LAYOUT_1        = (1L << 23),
    LV_OBJ_FLAG_LAYOUT_2        = (1L << 24),
```

```
    LV_OBJ_FLAG_WIDGET_1        = (1L << 25),
    LV_OBJ_FLAG_WIDGET_2        = (1L << 26),
    LV_OBJ_FLAG_USER_1          = (1L << 27),
    LV_OBJ_FLAG_USER_2          = (1L << 28),
    LV_OBJ_FLAG_USER_3          = (1L << 29),
    LV_OBJ_FLAG_USER_4          = (1L << 30),
};
typedef uint32_t lv_obj_flag_t;
```

9.2 基础对象的相关知识

该部件的相关知识，我们已经在第六章 6.1 小节详细介绍了，这里不做讲解。下面我们来讲解 一下基础对象的特殊功能，如下所示：

1. 扩展点击区域

在 LVGL 中，点击一个对象必须在该对象的边界区域内点击，如果我们想扩展点击区域，则调用函数 lv_obj_set_ext_click_area 设置。

lv_obj_set_ext_click_area 函数

扩展点击区域，其函数原型如以下所示：

```
void lv_obj_set_ext_click_area(lv_obj_t * obj, lv_coord_t size)
```

该函数的形参，如表 9.2.1 所示：

参数	描述
obj	指向对象的指针
size	设置点击区域大小

表 9.2.1 函数 lv_obj_set_ext_click_area()形参描述

返回值：无。

2. 组的定义

组是 LVGL 中特有的一个功能，其意义在于能快速的影响动作，如果一个 Slider 部件被聚焦并且移动时，则滑块的值将被更改。

(1) lv_group_create 函数

创建一个组，其函数原型如以下所示：

```
lv_group_t * lv_group_create(void)
```

返回值：返回组的控制块。

(2) lv_group_add_obj 函数

添加对象到组中，其函数原型如以下所示：

```
void lv_group_add_obj(lv_group_t * group, struct _lv_obj_t * obj);
```

该函数的形参，如表 9.2.2 所示：

参数	描述
group	指向组的指针
obj	添加对象

表 9.2.2 函数 lv_group_add_obj()形参描述

返回值：无。

9.3 基础对象的 API 函数

LVGL 官方提供基础对象相关 API 函数有很多，因为 obj 基础对象不仅仅可以作为独立的小部件还是所有小部件的基类，所以基础对象的 API 函数非常多，下面我们讲解只与基础对象相关的 API 函数，如表 9.3.1 所示：

函数	描述
lv_obj_create ()	创建基础对象（矩形）
lv_obj_set_user_data()	设置对象的 user_data 字段
lv_obj_has_flag()	检查是否在对象上设置了给定标志或所有给定标志
lv_obj_has_flag_any()	检查是否在对象上设置了给定标志或任何标志
lv_obj_get_state()	获取对象的状态
lv_obj_has_state()	检查对象是否处于给定状态
lv_obj_get_group()	获取对象的组
lv_obj_get_user_data()	获取对象的 user_data 字段
lv_obj_allocate_spec_attr()	如果尚未分配，则为对象分配特殊数据
lv_obj_check_type()	检查 obj 的类型
lv_obj_has_class()	检查是否有任何对象具有给定的类（类型）
lv_obj_get_class()	获取对象的类（类型）
lv_obj_is_valid()	检查是否有任何对象仍然“活着”

表 9.3.1 基础对象相关 API 函数

接下来我们来讲解 LVGL 基础对象常用的函数有哪些：

1. lv_obj_create 函数

创建基础对象，其函数原型如以下所示：

```
lv_obj_t * lv_obj_create(lv_obj_t * parent)
```

该函数的形参，如表 9.3.2 所示：

参数	描述
parent	部件的父类

表 9.3.2 函数 lv_obj_create()形参描述

返回值：返回对象控制块。

2. lv_obj_get_state 函数

获取对象的状态，其函数原型如以下所示：

```
lv_state_t lv_obj_get_state(const lv_obj_t * obj)
```

该函数的形参，如表 9.3.3 所示：

参数	描述
obj	对象指针

表 9.3.3 函数 lv_obj_get_state()形参描述

返回值：返回对象状态。

9.4 基础对象部件的实验

9.4.1 硬件设计

1. 例程功能

本实验主要测试基础对象 API 函数的使用，在 lvgl_demo.c 文件中设计了三个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要测试父类与子类的关系。实验现象：如果父类对象移动，则子类对象会跟着父类移动；如果子类移动，则父类不会跟着子类移动。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 9 lv_obj(基础对象)》。

9.4.2 软件设计

9.4.2.1 程序流程图

本实验的程序流程图，如下图 9.4.2.1 所示：

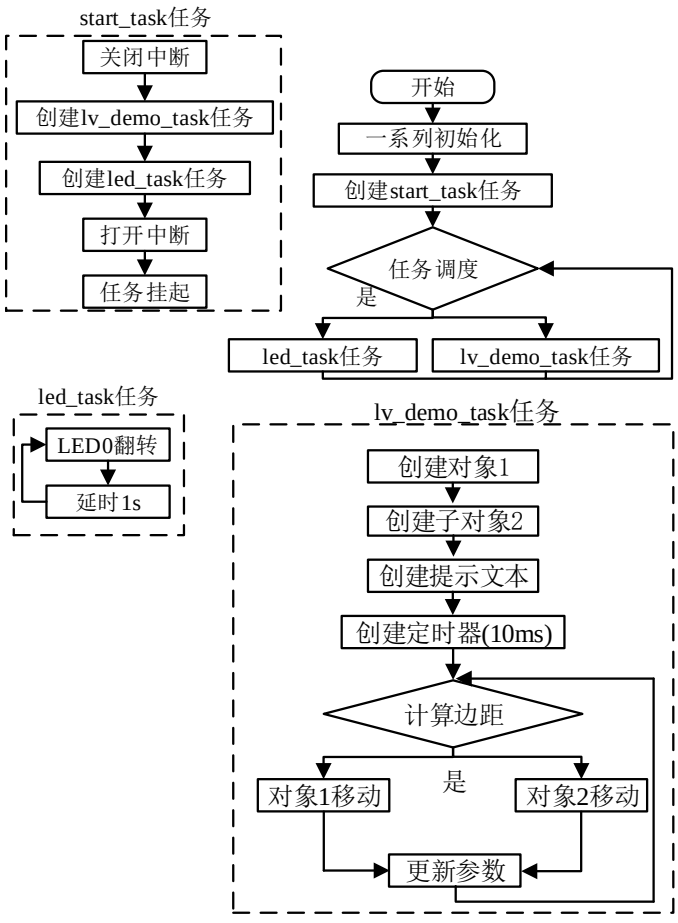


图 9.4.2.1 lv_obj 实验流程图

9.4.2.2 程序解析

关于 LVGL 实现程序在 lv_mainstart.c 文件中编写，如下源码所示：

(1) 创建对象并开启定时器

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    /* 初始化父对象，父对象的父对象为当前活动的屏幕 */
    parent_obj = lv_obj_create(lv_scr_act());
    /* 设置父对象的大小 */
    lv_obj_set_size(parent_obj, scr_act_width() / 2, scr_act_height() / 2);
    /* 设置父对象的背景色 */
    lv_obj_set_style_bg_color(parent_obj, lv_color_make(0x99, 0xCC, 0xFF), 0);
    /* 设置父对象的位置 */
    lv_obj_align(parent_obj, LV_ALIGN_TOP_LEFT, 0, 0);
    /* 手动更新父对象的参数 */
    lv_obj_update_layout(parent_obj);
    /* 获取父对象的宽度 */
    parent_width = lv_obj_get_width(parent_obj);
    /* 获取父对象的高度 */
    parent_height = lv_obj_get_height(parent_obj);
    /* 设置父对象内容区域宽度 */
    lv_obj_set_content_width(parent_obj, parent_width);
    /* 设置父对象内容区域高度 */
    lv_obj_set_content_height(parent_obj, parent_height);
    /* 初始化子对象 *//* 初始化子对象，子对象的父对象为父对象 */
    child_obj = lv_obj_create(parent_obj);
    /* 设置子对象的大小 */
    lv_obj_set_size(child_obj, parent_width / 2, parent_height / 2);
    /* 设置子对象的背景色 */
    lv_obj_set_style_bg_color(child_obj, lv_color_make(0x00, 0x33, 0x66), 0);
    /* 设置子对象的位置 */
    lv_obj_align_to(child_obj, parent_obj, LV_ALIGN_TOP_LEFT, 0, 0);
    /* 手动更新子对象的参数 */
    lv_obj_update_layout(child_obj);
    /* 获取子对象的宽度 */
    child_width = lv_obj_get_width(child_obj);
    /* 获取子对象的高度 */
    child_height = lv_obj_get_height(child_obj);
    /* 创建提示文本 */
```

```
    static lv_style_t font_style;  /* 定义字体风格 */
    lv_style_init(&font_style);    /* 初始化字体风格 */
    lv_style_set_text_font(&font_style, &myFont14);    /* 设置风格字体 */
    /* 设置风格颜色 */
    lv_style_set_text_color(&font_style, lv_color_make(0x33, 0x66, 0x99));
    static lv_obj_t* font_label_1;                              /* 定义标签物体 */
    font_label_1 = lv_label_create(lv_scr_act());               /* 初始化标签 */
    lv_obj_add_style(font_label_1, &font_style, 0);             /* 为标签添加风格 */
    lv_label_set_text(font_label_1, "1.子对象与父对象一起移动"); /* 添加标签文本 */
    lv_obj_align(font_label_1, LV_ALIGN_TOP_LEFT, 0, 0);        /* 设置标签位置 */
    static lv_obj_t* font_label_2;                              /* 定义标签物体 */
    font_label_2 = lv_label_create(lv_scr_act());               /* 初始化标签 */
    lv_obj_add_style(font_label_2, &font_style, 0);             /* 为标签添加风格 */
    /* 添加标签文本 */
    lv_label_set_text(font_label_2, "2.子对象超出父对象的部分将不被显示");
    /* 设置标签位置 */
    lv_obj_align_to(font_label_2, font_label_1,
    LV_ALIGN_OUT_BOTTOM_LEFT, 0, 0);
    /* 初始化定时器 */
    lv_timer_create(my_timer_cb, 10, NULL);
}
```

从上述源码可知：我们首先创建两个对象，一个为父类对象而另一个为子类对象，然后设置它们的大小和位置等属性，最后调用 lv_timer_create 函数创建 LVGL 软件定时器，定时时间为 10ms，该定时器的回调函数如下所示。

(2) 定时器回调函数

```
/**
 * @brief  定时器回调函数
 * @param  无
 * @return 无
 */
void my_timer_cb(lv_timer_t* timer)
{
    static uint8_t p_ud = 0, p_lr = 0;
    static lv_coord_t p_top_dis, p_bottom_dis, p_left_dis, p_right_dis;
    static uint8_t c_ud = 0, c_lr = 0;
    static lv_coord_t c_top_dis, c_bottom_dis, c_left_dis, c_right_dis;

    /* 计算边距 */
    p_top_dis = lv_obj_get_y(parent_obj);
    p_bottom_dis = scr_act_height() - lv_obj_get_y2(parent_obj);
    p_left_dis = lv_obj_get_x(parent_obj);
    p_right_dis = scr_act_width() - lv_obj_get_x2(parent_obj);
    c_top_dis = lv_obj_get_y(child_obj);
```

```
    c_bottom_dis = parent_height - lv_obj_get_y2(child_obj);
    c_left_dis = lv_obj_get_x(child_obj);
    c_right_dis = parent_width - lv_obj_get_x2(child_obj);
    /* 判断移动方向 */
    if (0 == p_top_dis)
        p_ud = 1;
    if (0 == p_bottom_dis)
        p_ud = 0;
    if (0 == p_left_dis)
        p_lr = 1;
    if (0 == p_right_dis)
        p_lr = 0;
    if (-child_height/2 == c_top_dis)
        c_ud = 1;
    if (-child_height / 2 == c_bottom_dis)
        c_ud = 0;
    if (-child_width / 2 == c_left_dis)
        c_lr = 1;
    if (-child_width / 2 == c_right_dis)
        c_lr = 0;
    /* 更新位置 */
    if (0 == p_ud)
        lv_obj_set_y(parent_obj, p_top_dis - 1);
    if (1 == p_ud)
        lv_obj_set_y(parent_obj, p_top_dis + 1);
    if (0 == p_lr)
        lv_obj_set_x(parent_obj, p_left_dis - 1);
    if (1 == p_lr)
        lv_obj_set_x(parent_obj, p_left_dis + 1);
    if (0 == c_ud)
        lv_obj_set_y(child_obj, c_top_dis - 1);
    if (1 == c_ud)
        lv_obj_set_y(child_obj, c_top_dis + 1);
    if (0 == c_lr)
        lv_obj_set_x(child_obj, c_left_dis - 1);
    if (1 == c_lr)
        lv_obj_set_x(child_obj, c_left_dis + 1);
    /* 手动更新参数 */
    lv_obj_update_layout(parent_obj);
    lv_obj_update_layout(child_obj);
}
```

从上述源码可知：程序首先获取父类对象和子类对象的方向，然后它根据移动方向来调用 lv_obj_set_x/y 函数设置父类对象和子类对象相对位置，最后程序调用 lv_obj_update_layout 函数更新对象的属性。

9.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 9.4.3.1 所示：

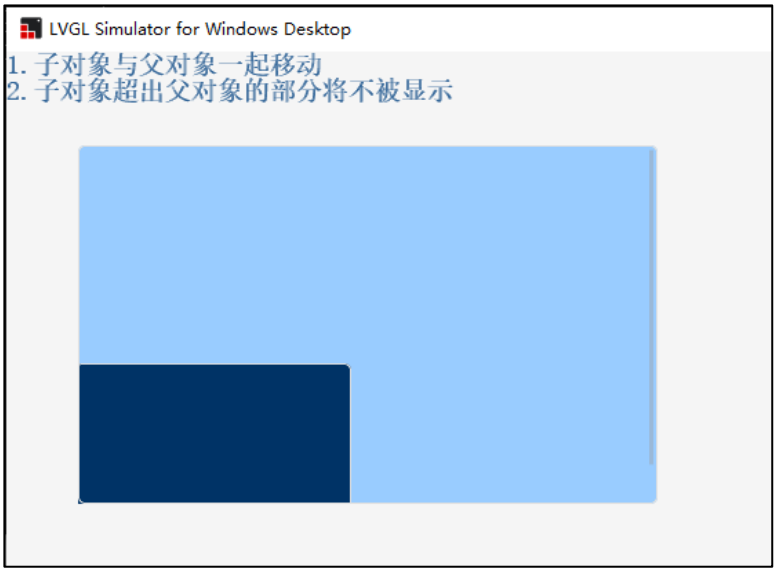


图 9.4.3.1 父类对象和子类对象移动

第十章 圆弧部件(lv_arc)

圆弧(arc)部件可以在项目中作为加载器或者进度条功能，其的结构非常简单，我们只需要控制圆弧的角度和厚度。圆弧是由背景弧和前景弧组成，它们两者都具有起始角度和终止角度以及厚度，可以简单说：背景弧设置灰色，而前景弧设置红色，当前景弧不断的延申，那么红色的前景弧会覆盖背景弧，这样圆弧的效果就是这样得来的。

本章分为如下几部分：

10.1 圆弧部件的组成

10.2 圆弧部件的相关知识

10.3 圆弧部件的 API 函数

10.4 圆弧部件的实验

10.1 圆弧部件的组成

圆弧(lv_arc)部件由背景弧、前景弧和指示器组成。指示器可以触摸调整前景弧，如图 10.1.1 所示：

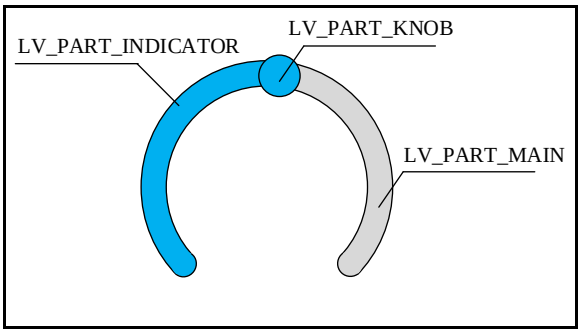


图 10.1.1 圆弧(lv_arc)组成部分

从上图可知：圆弧部件由三个部分组成，如下所示：

1. LV_PART_MAIN： 使用典型的背景样式属性绘制背景。
2. LV_PART_INDICATOR：使用圆弧样式属性绘制另一条圆弧。
3. LV_PART_KNOB：指示器。

如何设置圆弧部件各个部分的样式，请大家参考第六章 6.3 小节，这小节中，笔者已经很详细讲解部件各个部分的样式属性修改方法。

10.2 圆弧部件的相关知识

10.2.1 圆弧的值和范围设置

圆弧部件的值和范围是指设置圆弧的滑动范围以及当前滑动值，如果我们设置整个起始到终止的范围为 0 到 100 且圆弧当前值为 50(单位：像素)，则前景弧不断的延申并覆盖背景弧直到滑动值为 50 的地方才停止延申，下面笔者使用一个简单实例来讲解本小节的知识如下源码所示：

```
void lv_mainstart()
{
    lv_obj_t* arc = lv_arc_create(lv_scr_act());
```

```
    lv_arc_set_range(arc, 0, 100);      /* 设置 arc 的范围 */
    lv_arc_set_value(arc, 50);          /* 设置 arc 的值 */
    lv_obj_set_size(arc, 200, 200);
    lv_obj_align_to(arc, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

上述的源码可以在 PC 模拟器中运行，该效果图如下图 10.2.1.1 所示：

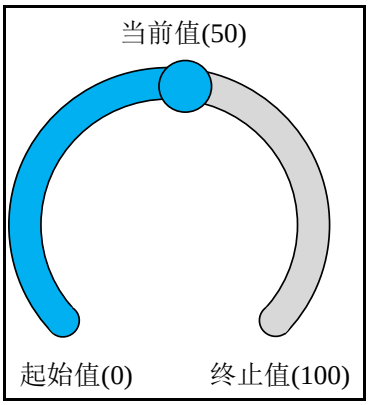


图 10.2.1.1 圆弧的值和范围联系图

从上图可以看出：我们设置圆弧的范围为 0 到 100，而设置圆弧的当前值为 50，所以圆弧的前景弧延申到一半就停止了。

10.2.2 圆弧部件角度设置

圆弧部件的 0 度位于对象右中间(3 点钟方向)，并且度数沿顺时针方向增加，这个旋转角度应在[0.360]范围内。前面我们已经了解到圆弧是由背景弧和前景弧组成的，所以用户设置圆弧的角度必须把这两者都设置成相同的起始角度和终止角度，如果背景弧比前景弧的角度大，则前景弧不能完全覆盖背景弧，反之前景弧比背景弧的角度越大，则前景弧会覆盖并超过背景弧的范围。

圆弧部件的角度设置具有六个函数，这六个函数可以分为两类，一类为背景弧的角度设置，另一类为前景弧的角度设置，这六个函数如下所示：

```
lv_arc_set_bg_angles(arc, start_angle, end_angle)  /* 背景起始角度和终止角度 */
lv_arc_set_bg_start/end_angle(arc, start_angle)    /* 背景设置起始角度/终止角度 */
lv_arc_set_angles(arc, start_angle, end_angle)     /* 设置前景弧起始角度和终止角度 */
lv_arc_set_start/end_angle(arc, start_angle)       /* 设置前景弧起始角度/终止角度 */
```

上述第一和第二个函数实现的原理类似，他们唯一的区别是第一个函数可以一起设置起始角度和终止角度，而第二个函数只能单个设置，第三和第四个函数也是一样的道理。

下面笔者使用一个简单实例来讲解本小节的知识如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* arc = lv_arc_create(lv_scr_act());
    lv_arc_set_bg_angles(arc, 135, 45);    /* 设置背景弧的起始角度和终止角度 */
    lv_arc_set_angles(arc, 135, 270);      /* 设置前景弧的起始角度和终止角度 */
    lv_obj_set_size(arc, 200, 200);
    lv_obj_align_to(arc, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

上述的函数中，我们首先设置背景弧的起始角度和终止角度为 135° 到 45° ，然后设置前景弧的起始角度和终止角度为 135° 到 270° ，至此我们就可以得到了下面的效果图，如图 10.2.2.1 所示：

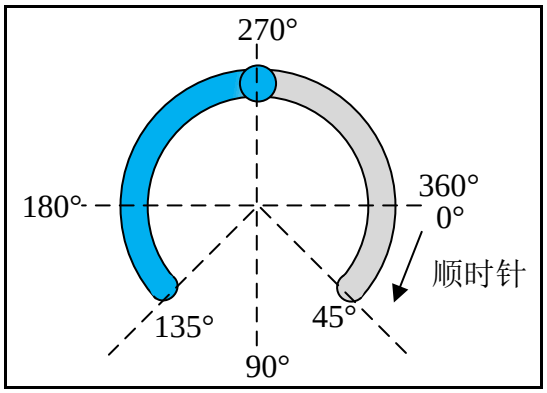


图 10.2.2.1 弧的角度解析

10.2.3 圆弧部件旋转设置

圆弧部件旋转是指对整个圆弧对象以顺时针旋转某个角度，注意：该角度的范围是[0~360]。LVGL 提供了 lv_arc_set_rotation 函数给用户对圆弧旋转，该函数非常简单，用户只需把该函数的第二个形参传入旋转值即可。

下面我们也是使用一个简单实例来讲解本小节的知识，如下源码所示：

```
void lv_mainstart()
{
    lv_obj_t* arc = lv_arc_create(lv_scr_act());
    lv_arc_set_bg_angles(arc, 135, 45);
    lv_arc_set_angles(arc, 135, 270);
    lv_arc_set_rotation(arc, 180);
    lv_obj_set_size(arc, 200, 200);
    lv_obj_align_to(arc, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

根据上述的源码，我们调用函数 lv_arc_set_rotation 对圆弧部件进行旋转，该旋转角度为 180° ，至此我们可以得到 如下示意图如图 10.2.3.1 所示：

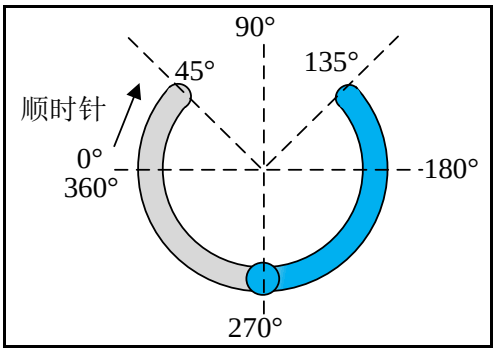


图 10.2.3.1 设置 arc 旋转角度

10.2.4 圆弧的模式选择

默认情况下，圆弧部件的方向是顺时针加载的，但是这个方向可以人为修改，如以下所示：

1. LV_ARC_MODE_NORMAL：指示器弧顺时针绘制。
2. LV_ARC_MODE_REVERSE：指示器弧沿逆时针方向绘制。
3. LV_ARC_MODE_SYMMETRIC：从中间点到当前值的指示弧线。

上述模式中，用户可调用函数 lv_arc_set_mode 设置。

10.2.5 圆弧部件的变化率设置

如果圆弧的指示器被滑动时，滑动的值将根据变化率以有限的速度设置。变化率以度/秒为单位定义，用户可调用 lv_arc_set_change_rage 设置变化率。

10.2.6 移除旋钮

前面我们知道圆弧是由三个部分组成的，用户可以通过滑动圆弧的指示器来调整前景弧的滑动数值，如果用户想把指示器移除并使圆弧不可调节，可使用一下源码设置。

```
lv_obj_remove_style(arc, NULL, LV_PART_KNOB);
lv_obj_clear_flag(arc, LV_OBJ_FLAG_CLICKABLE);
```

10.2.7 圆弧部件事件

1. 输入设备事件：

圆弧部件除了基本的事件之外，弧部件具有特殊属性是 LV_EVENT_VALUE_CHANGED。

2. 绘图事件：

(1) LV_ARC_DRAW_PART_BACKGROUND：绘制背景弧线。

(2) LV_ARC_DRAW_PART_FOREGROUND：绘制前景弧线。

(3) LV_ARC_DRAW_PART_KNOB ：绘制旋钮。

10.3 圆弧部件的 API 函数

LVGL 官方提供圆弧(lv_arc)部件相关 API 函数，如表 10.3.1 所示：

函数	描述
lv_arc_create()	创建圆弧对象
lv_arc_set_start_angle()	设置圆弧的起始角度
lv_arc_set_end_angle()	设置圆弧的结束角度
lv_arc_set_angles()	设置开始和结束角度
lv_arc_set_bg_start_angle()	设置圆弧背景的起始角度
lv_arc_set_bg_end_angle()	设置圆弧背景的结束角度
lv_arc_set_bg_angles()	设置圆弧背景的起止和结束角度
lv_arc_set_rotation()	设置整个圆弧的旋转
lv_arc_set_mode()	设置圆弧的类型
lv_arc_set_value()	在圆弧上设置新值
lv_arc_set_range()	设置弧的最小值和最大值
lv_arc_set_change_rate()	设置一个变化率来限制圆弧到达按压点的速度
lv_arc_get_angle_start()	获取圆弧的起始角度
lv_arc_get_angle_end()	获取圆弧的结束角度
lv_arc_get_bg_angle_start()	获取圆弧背景的起始角度

lv_arc_get_bg_angle_end()	获取圆弧背景的结束角度
lv_arc_get_value()	获取圆弧的值
lv_arc_get_min_value()	获取弧的最小值
lv_arc_get_max_value()	获取弧的最大值
lv_arc_get_mode()	获取弧线是否为类型

表 10.3.1 圆弧部件相关 API 函数

接下来我们来讲解 LVGL 圆弧部件常用的函数有哪些：

1. lv_arc_create 函数

创建圆弧对象，其函数原型如以下所示：

```
lv_obj_t * lv_arc_create ( lv_obj_t *parent)
```

该函数的形参，如表 10.3.2 所示：

参数	描述
parent	部件的父类

表 10.3.2 函数 lv_arc_create()形参描述

返回值：返回对象控制块。

2. lv_arc_set_start_angle 函数

设置圆弧的起始角度，其函数原型如以下所示：

```
void lv_arc_set_start_angle ( lv_obj_t * arc , uint16_t start )
```

该函数的形参，如表 10.3.3 所示：

参数	描述
arc	指向弧对象的指针
start	起始角度

表 10.3.3 函数 lv_arc_set_start_angle()形参描述

返回值：无。

3. lv_arc_set_end_angle 函数

设置圆弧的结束角度，其函数原型如以下所示：

```
void lv_arc_set_end_angle ( lv_obj_t * arc , uint16_t end )
```

该函数的形参，如表 10.3.4 所示：

参数	描述
arc	指向弧对象的指针
end	结束角度

表 10.3.4 函数 lv_arc_set_end_angle()形参描述

返回值：无。

4. lv_arc_set_angles 函数

设置开始和结束角度，其函数原型如以下所示：

```
void lv_arc_set_angles ( lv_obj_t * arc , uint16_t start, uint16_t end)
```

该函数的形参，如表 10.3.5 所示：

参数	描述
arc	指向弧对象的指针
start	起始角度
end	结束角度

表 10.3.5 函数 lv_arc_set_angles()形参描述

返回值：无。

5. lv_arc_set_bg_start_angle 函数

设置圆弧背景的起始角度，其函数原型如以下所示：

```
void lv_arc_set_bg_start_angle ( lv_obj_t * arc , uint16_t start )
```

该函数的形参，如表 10.3.6 所示：

参数	描述
arc	指向弧对象的指针
start	起始角度

表 10.3.6 函数 lv_arc_set_bg_start_angle()形参描述

返回值：无。

6. lv_arc_set_bg_end_angle 函数

设置圆弧的结束角度，其函数原型如以下所示：

```
void lv_arc_set_bg_end_angle ( lv_obj_t * arc , uint16_t end )
```

该函数的形参，如表 10.3.7 所示：

参数	描述
arc	指向弧对象的指针
end	结束角度

表 10.3.7 函数 lv_arc_set_bg_end_angle()形参描述

返回值：无。

7. lv_arc_set_bg_angles 函数

设置圆弧背景的起止角度，其函数原型如以下所示：

```
void lv_arc_set_bg_angles ( lv_obj_t * arc , uint16_t start, uint16_t end)
```

该函数的形参，如表 10.3.8 所示：

参数	描述
arc	指向弧对象的指针
start	起始角度
end	结束角度

表 10.3.8 函数 lv_arc_set_bg_angles()形参描述

返回值：无。

8. lv_arc_set_rotation 函数

设置整个圆弧的旋转，其函数原型如以下所示：

```
void lv_arc_set_rotation ( lv_obj_t * arc , uint16_t rotation)
```

该函数的形参，如表 10.3.9 所示：

参数	描述
arc	指向弧对象的指针
rotation	旋转角度

表 10.3.9 函数 lv_arc_set_rotation()形参描述

返回值：无。

9. lv_arc_set_mode 函数

设置圆弧的类型，其函数原型如以下所示：

```
void lv_arc_set_mode ( lv_obj_t * arc , lv_arc_mode_t type)
```

该函数的形参，如表 10.3.10 所示：

参数	描述
arc	指向弧对象的指针
type	弧的模式

表 10.3.10 函数 lv_arc_set_mode()形参描述

返回值：无。

10. lv_arc_set_value 函数

在圆弧上设置新值，其函数原型如以下所示：

```
void lv_arc_set_value ( lv_obj_t * arc , int16_t value)
```

该函数的形参，如表 10.3.11 所示：

参数	描述
arc	指向弧对象的指针
value	新值

表 10.3.11 函数 lv_arc_set_value()形参描述

返回值：无。

11. lv_arc_set_range 函数

设置弧的最小值和最大值，其函数原型如以下所示：

```
void lv_arc_set_range ( lv_obj_t * arc , int16_t min , int16_t max )
```

该函数的形参，如表 10.3.12 所示：

参数	描述
arc	指向弧对象的指针
min	最小值
max	最大值

表 10.3.12 函数 lv_arc_set_range()形参描述

返回值：无。

12. lv_arc_set_change_rate 函数

设置一个变化率来限制圆弧到达按压点的速度，其函数原型如以下所示：

```
void lv_arc_set_change_rate ( lv_obj_t * arc , uint16_t rate)
```

该函数的形参，如表 10.3.13 所示：

参数	描述
arc	指向弧对象的指针
rate	变化率

表 10.3.13 函数 lv_arc_set_change_rate()形参描述

返回值：无。

10.4 圆弧部件的实验

10.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数测试三种圆弧部件的效果。实验现象：实验一：用户可触摸调节圆弧数值，实验二和实验三：以动画的形式让圆弧部件数值加一。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 10 arc(弧)》。

10.4.2 软件设计

10.4.2.1 程序流程图

本实验的程序流程图，如下图 10.4.2.1 所示：

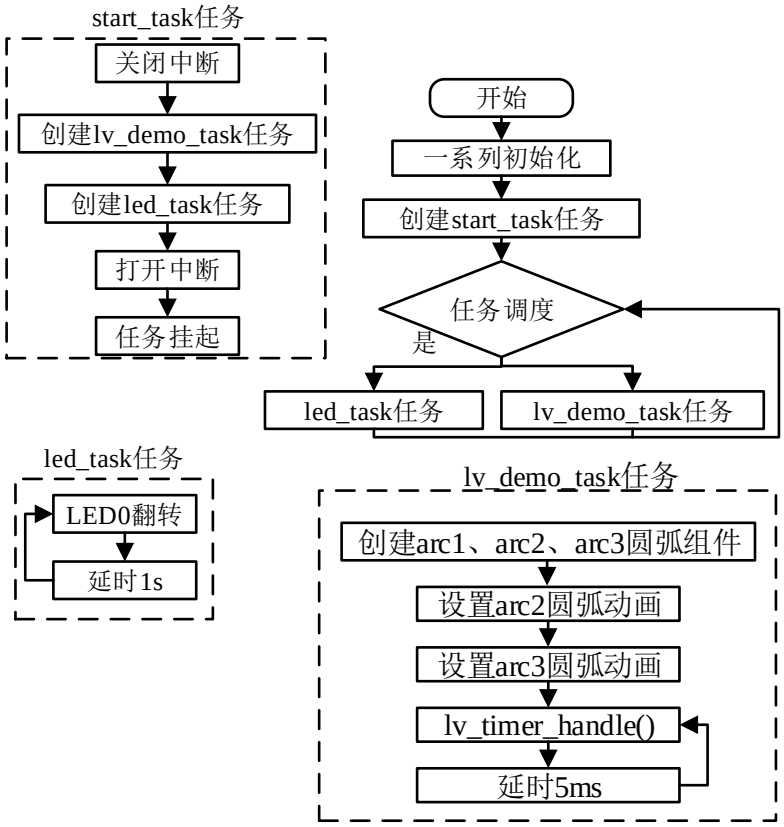


图 10.4.2.1 圆弧实验流程图

10.4.2.2 程序解析

关于 LVGL 实现程序在 lv_mainstart.c 文件中编写，如下源码所示：

(1) 创建三个圆弧部件，如下源码所示：

```
/**
 * @brief   LVGL 演示
 * @param   无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_arc_1();
    lv_example_arc_2();
    lv_example_arc_3();
}
```

(2) arc1 实现源码如下：

```
/**
 * @brief   例 1
```

```
 * @param  无
 * @return 无
 */
static void lv_example_arc_1(void)
{
    lv_obj_t* arc = lv_arc_create(lv_scr_act());   /* 定义弧 */
    /* 设置弧的大小 */
    lv_obj_set_size(arc, scr_act_height() / 3, scr_act_height() / 3);
    lv_arc_set_rotation(arc, 135);              /* 旋转弧 */
    lv_arc_set_bg_angles(arc, 0, 270);          /* 设置弧的始末角度 */
    lv_arc_set_value(arc, 40);                  /* 这是弧的角度 */
    /* 设置弧的位置 */
    lv_obj_align(arc, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);
}
```

上述源码没什么好讲解的，前面笔者已经讲解过了。

(3) arc2 实现源码如下：

```
/**
 * @brief  例 2a 动画回调函数
 * @param  无
 * @return 无
 */
static void anim_2_a_cb(lv_obj_t* obj, int32_t v)
{
    lv_arc_set_end_angle(obj, v);
}

/**
 * @brief  例 2b 动画回调函数
 * @param  无
 * @return 无
 */
static void anim_2_b_cb(lv_obj_t* obj, int32_t v)
{
    lv_arc_set_start_angle(obj, v);
}

/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_arc_2(void)
{
```

```
    lv_obj_t* arc = lv_arc_create(lv_scr_act());        /* 定义弧 */
    /* 设置弧的大小 */
    lv_obj_set_size(arc, scr_act_height() / 3, scr_act_height() / 3);
    lv_arc_set_rotation(arc, 270);                      /* 旋转弧 */
    lv_arc_set_bg_angles(arc, 0, 360);                  /* 设置弧的始末角度 */
    lv_arc_set_value(arc, 0);                           /* 这是弧的角度 */
    lv_obj_remove_style(arc, NULL, LV_PART_KNOB);       /* 移除手柄 */
    lv_obj_clear_flag(arc, LV_OBJ_FLAG_CLICKABLE);      /* 设置不可点击 */
    lv_obj_align(arc, LV_ALIGN_CENTER, 0, 0);           /* 设置弧的位置 */

    lv_anim_t a;                                        /* 定义动画 */
    lv_anim_init(&a);                                   /* 初始化动画 */
    lv_anim_set_var(&a, arc);                           /* 绑定物体 */
    lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t)anim_2_a_cb); /* 设置回调函数 */
    lv_anim_set_time(&a, 1000);                                 /* 设置动画时间 */
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);      /* 设置无限循环 */
    lv_anim_set_repeat_delay(&a, 0);                            /* 设置延时 */
    lv_anim_set_values(&a, 60, 420);                            /* 设置动画始末值 */
    lv_anim_start(&a);                                          /* 开启动画 */
    lv_anim_t b;                                                /* 定义动画 */
    lv_anim_init(&b);                                           /* 初始化动画 */
    lv_anim_set_var(&b, arc);                                   /* 绑定物体 */
    lv_anim_set_exec_cb(&b, (lv_anim_exec_xcb_t)anim_2_b_cb); /* 设置回调函数 */
    lv_anim_set_time(&b, 1000);                                 /* 设置动画时间 */
    lv_anim_set_repeat_count(&b, LV_ANIM_REPEAT_INFINITE);      /* 设置无限循环 */
    lv_anim_set_repeat_delay(&b, 0);                            /* 设置延时 */
    lv_anim_set_values(&b, 0, 360);                             /* 设置动画始末值 */
    lv_anim_start(&b);                                          /* 开启动画 */
}
```

从上述源码可知：我们给 arc2 圆弧部件设置了两个动画，它们动画回调函数分别设置该圆弧的结束值和起始值。注意：它们的动画时间都是 1s 中，动画 a 的数值为 60 到 420，这些数值范围会在该动画回调函数中设置圆弧的结束值，而动画 b 的数值范围为 0 到 360，这些数值范围会在该动画回调函数中设置圆弧的起始值。

(4) arc3 实现源码如下：

```
/**
 * @brief   例 3a 动画回调函数
 * @param   无
 * @return 无
 */
static void anim_3_a_cb(lv_obj_t* obj, int32_t v)
{
    lv_arc_set_value(obj, v);
}
```

```
/**
 * @brief  例 3
 * @param  无
 * @return 无
 */
static void lv_example_arc_3(void)
{
    lv_obj_t* arc = lv_arc_create(lv_scr_act());  /* 定义弧 */
    /* 设置弧的大小 */
    lv_obj_set_size(arc, scr_act_height() / 3, scr_act_height() / 3);
    lv_arc_set_rotation(arc, 270);                    /* 旋转弧 */
    lv_arc_set_bg_angles(arc, 0, 360);                /* 设置弧的始末角度 */
    lv_arc_set_value(arc, 0);                         /* 这是弧的角度 */
    lv_obj_remove_style(arc, NULL, LV_PART_KNOB);     /* 移除手柄 */
    lv_obj_clear_flag(arc, LV_OBJ_FLAG_CLICKABLE);    /* 设置不可点击 */
    /* 设置弧的位置 */
    lv_obj_align(arc, LV_ALIGN_CENTER, scr_act_width() / 3, 0);
    lv_anim_t a;                                      /* 定义动画 */
    lv_anim_init(&a);                                 /* 初始化动画 */
    lv_anim_set_var(&a, arc);                         /* 绑定物体 */
    lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t)anim_3_a_cb); /* 设置回调函数 */
    lv_anim_set_time(&a, 1000);                       /* 设置动画时间 */
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);    /* 设置无限循环 */
    lv_anim_set_repeat_delay(&a, 500);                /* 设置延时 */
    lv_anim_set_values(&a, 0, 100);                   /* 设置动画始末值 */
    lv_anim_start(&a);                                /* 开启动画 */
}
```

上述源码可知：我们给 arc3 圆弧部件设置一个动画效果，该动画的回调函数是设置圆弧的当前值，注意：这个动画是无限循环的，也就是说动画的数值以 1s 中之内从 0 开始累加到 100 结束，然后该动画的数值又从 0 开始累加。

10.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 10.4.3.1 所示：

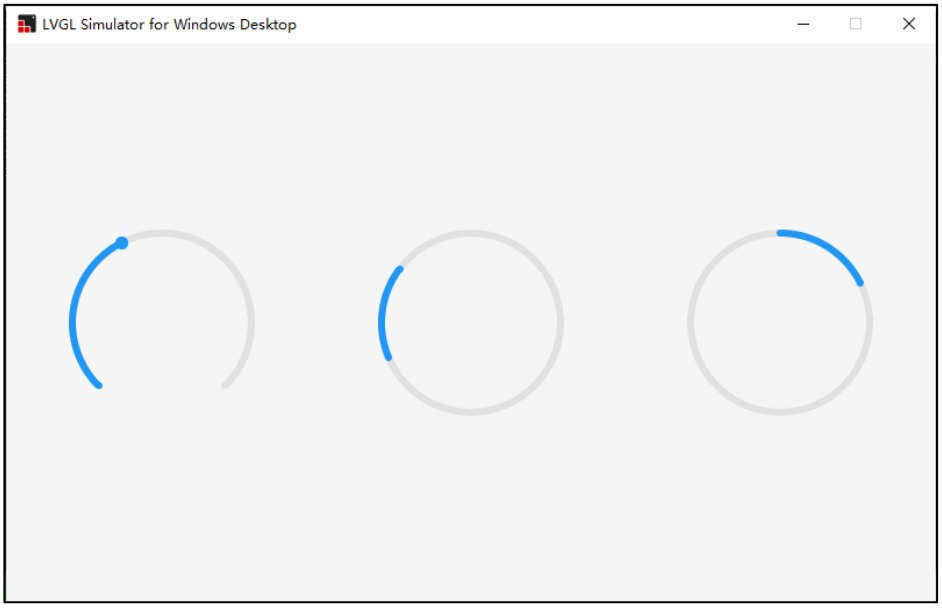


图 10.4.3.1 圆弧实验

第十一章 进度条部件(lv_bar)

进度条被设计来显示进度的当前完成百分比。进度条通过 CSS 编码灵活调整大小，默认会缩放到适应父容器的大小，它会从左向右填充，然后循环回到原点。

本章分为如下几部分：

11.1 进度条部件的组成

11.2 进度条部件的相关知识

11.3 进度条部件的 API 函数

11.4 进度条部件的实验

11.1 进度条部件的组成

进度条(lv_bar)部件由背景和指示器组成。它可以设置成水平进度条和垂直进度条，如果对象的宽度小于其高度为垂直进度条，否则为水平进度条，如图 11.1.1 所示：

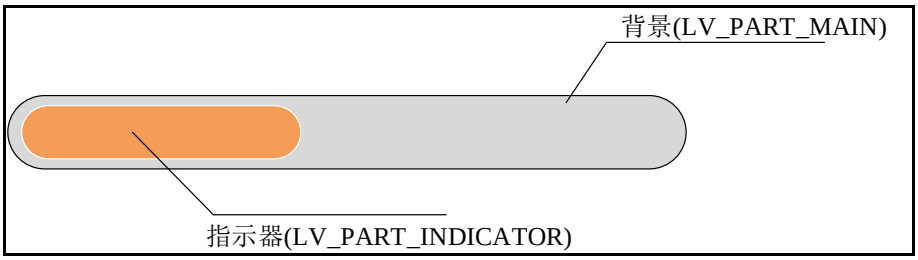


图 11.1.1 进度条的组成

1. LV_PART_MAIN：横栏的背景和它使用典型的背景样式属性，它可以设置 anim_time 样式属性的动画时间属性 LV_ANIM_ON。

2. LV_PART_INDICATOR：指示器本身，它可以使用所有典型的背景属性。

上述的组成部分可以设置该部件的样式，样式的设置方法请参考第六章 6.3 小节。

11.2 进度条部件的相关知识

11.2.1 进度条部件的值和范围

进度条部件可像圆弧部件一样设置当前数值和数值范围，例如设置进度条的数值范围可以调用以下函数。

```
lv_bar_set_range(bar, min, max) /* 设置进度条部件范围 */
```

此函数的默认范围是[1：100]，这里的最大值可根据用户而定。

进度条部件的当前数值可调用以下函数设置

```
lv_bar_set_value(bar, new_value, LV_ANIM_ON/OFF) /* 设置进度条当前值 */
```

该函数的第二个形参表示该进度条的当前数值，而第三个形参表示是否使用动画的形式加载，如果该形参设置为 LV_ANIM_OFF，则该进度条不使用动画加载，否则该进度条使用动画的形式加载。

注意：进度条可以不从 0 开始，这样的话我们可以从任意的位置开始，用户可调用以下函数设置起始地方，

```
lv_bar_set_start_value(bar, new_value, LV_ANIM_ON/OFF) /* 设置进度条开始值 */
```

其中函数 lv_bar_set_value 和函数 lv_bar_set_start_value 的最后一个形参表示是否开启动画的形式，如果不开启动画请传入 LV_ANIM_OFF，否则该形参传入 LV_ANIM_ON。

老规矩，我们使用一个简单实例来讲解本小节的知识，如下源码所示：

```
void lv_mainstart()
{
    lv_obj_t* lv_bar1 = lv_bar_create(lv_scr_act());        /* 创建进度条 */
    lv_obj_set_style_bg_color(lv_bar1, lv_color_black(),
                              LV_PART_MAIN |LV_STATE_DEFAULT);
    lv_obj_set_style_bg_color(lv_bar1, lv_palette_main(LV_PALETTE_RED),
    LV_PART_INDICATOR |LV_STATE_DEFAULT);
    lv_obj_set_size(lv_bar1, 100, 20);                      /* 设置进度条的大小 */
    lv_obj_align(lv_bar1, LV_ALIGN_CENTER, 0, 0);           /* 设置对齐模式 */
    lv_bar_set_range(lv_bar1, 0, 100);
    lv_bar_set_value(lv_bar1, 100, LV_ANIM_OFF);            /* 设置进度条的值 */


    lv_obj_t* lv_bar2 = lv_bar_create(lv_scr_act());        /* 创建进度条 */
    lv_obj_set_style_bg_color(lv_bar2, lv_color_black(),LV_STATE_DEFAULT);
    lv_obj_set_style_bg_color(lv_bar2, lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
    lv_obj_set_size(lv_bar2, 20, 100);                      /* 设置进度条的大小 */
    /* 设置对齐模式 */
    lv_obj_align_to(lv_bar2, lv_bar1, LV_ALIGN_OUT_BOTTOM_MID, 0, 20);
    lv_bar_set_range(lv_bar2, 0, 100);
    /* 填充满的时间 */
    lv_obj_set_style_anim_time(lv_bar2, 5000, LV_STATE_DEFAULT);
    lv_bar_set_value(lv_bar2, 100, LV_ANIM_ON);             /* 设置进度条的值 */
}
```

上述的源码表示：我们创建了两个进度条，它们不同的是 lv_bar1 进度条被设置为垂直进度条且不开启动画，而 lv_bar2 进度条设置为水平进度条且开启动画，其中 lv_bar2 进度条的动画时间为 5s，也就是说 lv_bar2 进度条经过 5s 中才能填充完毕，而 lv_bar1 进度条瞬间就填充完毕，如图 11.2.1.1 所示：

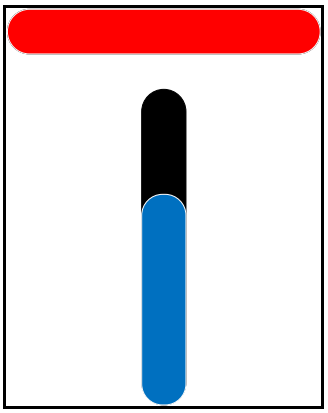


图 11.2.1.1 开启动画效果示意图

注意：开启动画效果的函数 lv_obj_set_style_anim_time 必须在 lv_bar_set_value 函数之前调用，否则不会出现效果。

11.2.2 进度条类型

进度条部件具有三种类型，如以下所示：

1. LV_BAR_MODE_NORMAL：普通进度条

2. LV_BAR_MODE_SYMMETRICAL：将指标从零值绘制到当前值。需要负的最小范围和正的最大范围。

3. LV_BAR_MODE_RANGE：也允许设置起始值，该起始值必须始终小于结束值。

上述的类型可调用函数 lv_bar_set_mode 设置，下面我们重点讲解一下 LV_BAR_MODE_SYMMETRICAL 类型，该类型可称为对称类型，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* bar2 = lv_bar_create(lv_scr_act());      /* 创建 bar 部件 */
    lv_obj_set_style_bg_color(bar2, lv_color_black(), LV_STATE_DEFAULT);
    lv_obj_set_style_bg_color(bar2, lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
    lv_bar_set_mode(bar2, LV_BAR_MODE_SYMMETRICAL);    /* 设置 bar 为对称模式 */
    lv_bar_set_range(bar2, -100, 100);                 /* 设置范围 */
    lv_obj_set_style_anim_time(bar2, 5000, LV_STATE_DEFAULT); /* 填充满的时间 */
    lv_bar_set_value(bar2, 100, LV_ANIM_ON);     /* 设置 bar 的值,动画显示打开 */
}
```

如果我们把进度条的范围设置为-100 到 100 的范围，并使用 LV_BAR_MODE_SYMMETRICAL 对称类型且设置指示器从 0 的位置执行，如图 11.2.2.1 所示，如果我们不设置类型为对称，则-100 到 0 的进度条位置被覆盖，如图 11.2.2.2 所示：

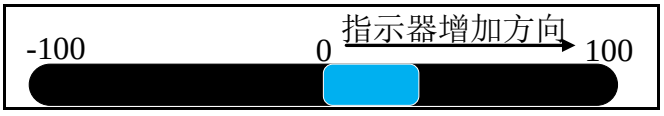


图 11.2.2.1 开启对称模式

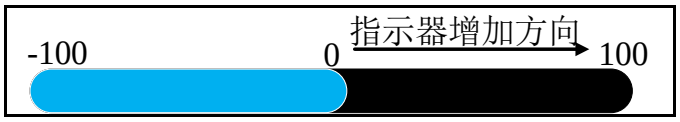


图 11.2.2.2 不开启对称模式

11.2.3 进度条事件

绘画事件：

LV_BAR_DRAW_PART_INDICATOR：条形图指示器。

11.3 进度条部件的 API 函数

LVGL 官方提供进度条(lv_bar)部件相关 API，如表 11.3.1 所示：

函数	描述
lv_bar_create()	创建进度条形对象
lv_bar_set_value()	在进度条设置新值

lv_bar_set_start_value()	在进度条设置新的起始值
lv_bar_set_range()	设置进度条最小值和最大值
lv_bar_set_mode()	设置进度条类型
lv_bar_get_value()	获取进度条的值
lv_bar_get_start_value()	获取进度条起始值
lv_bar_get_min_value()	获取进度条最小值
lv_bar_get_max_value()	获取进度条最大值
lv_bar_get_mode()	获取进度条类型

表 11.3.1 进度条相关 API 函数

接下来我们来讲解 LVGL 进度条部件常用的函数有哪些：

lv_bar_create()函数

创建进度条对象，其函数原型如以下所示：

```
lv_obj_t * lv_bar_create ( lv_obj_t *parent)
```

该函数的形参，如表 11.3.2 所示：

参数	描述
parent	部件的父类

表 11.3.2 函数 lv_bar_create()形参描述

返回值：返回对象控制块。

lv_bar_set_value()函数

在进度条设置新值，其函数原型如以下所示：

```
void lv_bar_set_value ( lv_obj_t * obj , int32_t value, lv_anim_enable_t anim )
```

该函数的形参，如表 11.3.3 所示：

参数	描述
obj	指向 bar 对象的指针
value	新值
anim	LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：不开启

表 11.3.3 函数 lv_bar_set_value()形参描述

返回值：无。

lv_bar_set_start_value()函数

在进度条设置新的起始值，其函数原型如以下所示：

```
void lv_bar_set_start_value ( lv_obj_t * obj ,
                              int32_t start_value ,
                              lv_anim_enable_t anim)
```

该函数的形参，如表 11.3.4 所示：

参数	描述
obj	指向 bar 对象的指针
start_value	新的起始值
anim	LV_ANIM_ON：用动画设置值；LV_ANIM_OFF：不开启

表 11.3.4 函数 lv_bar_set_start_value()形参描述

返回值：无。

lv_bar_set_range()函数

设置进度条最小值和最大值，其函数原型如以下所示：

```
void lv_bar_set_range ( lv_obj_t * obj , int32_t min , int32_t max )
```

该函数的形参，如表 11.3.5 所示：

参数	描述

obj	指向 bar 对象的指针
min	最小值
max	最大值

表 11.3.5 函数 lv_bar_set_range()形参描述

返回值：无。

lv_bar_set_mode()函数

设置进度条类型，其函数原型如以下所示：

```
void lv_bar_set_mode ( lv_obj_t * obj , lv_bar_mode_t mode)
```

该函数的形参，如表 11.3.6 所示：

参数	描述
obj	指向 bar 对象的指针
mode	lv_bar_mode_t 类型

表 11.3.6 函数 lv_bar_set_mode()形参描述

返回值：无。

11.4 进度条部件的实验

11.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要测试进度条以动画的形式加载。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 11 bar(进度条)》。

11.4.2 软件设计

11.4.2.1 程序流程图

本实验的程序流程图，如下图 11.4.2.1 所示：

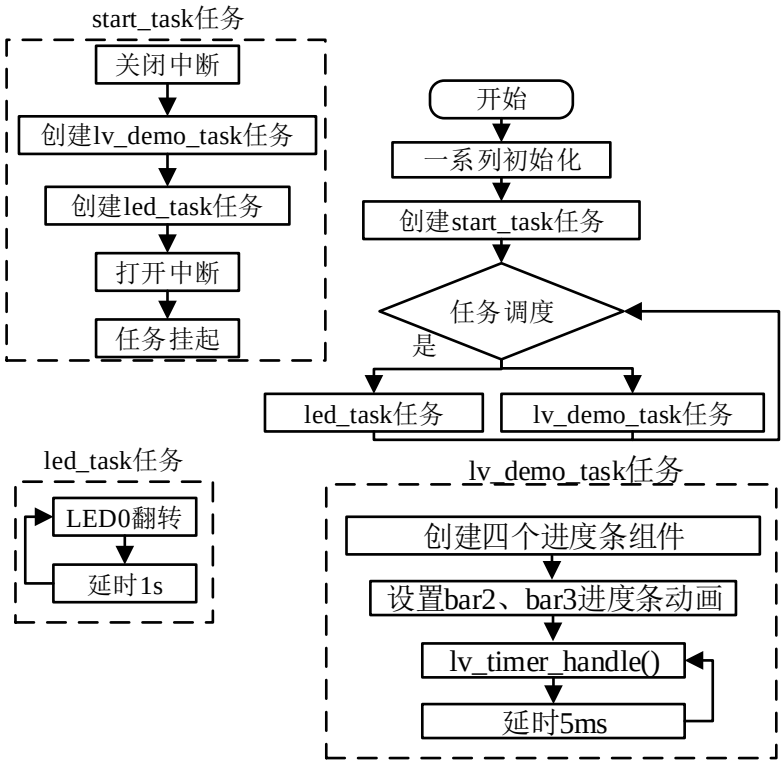


图 11.4.2.1 bar 部件实验流程图

11.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如以下所示：

(1) bar2 实现源码如下：

```
/**
 * @brief  例 2a 动画回调函数
 * @param  无
 * @return 无
 */
static void anim_2_a_cb(lv_obj_t* obj, int32_t v)
{
    lv_bar_set_value(obj, v, LV_ANIM_ON);
}

/**
 * @brief  例 2
 * @param  无
 * @return 无
*/
static void lv_example_bar_2(void)
{
    lv_obj_t* bar;                              /* 定义进度条 */
```

```
    bar = lv_bar_create(lv_scr_act());          /* 初始化进度条 */
    /* 设置进度条大小 */
    lv_obj_set_size(bar, scr_act_height() / 24, scr_act_width() / 4);
    lv_obj_align(bar, LV_ALIGN_CENTER, 0, 0); /* 设置进度条位置 */

    lv_anim_t a;                                /* 定义动画 */
    lv_anim_init(&a);                           /* 初始化动画 */
    lv_anim_set_var(&a, bar);                   /* 绑定物体 */
    lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t)anim_2_a_cb); /* 设置回调函数 */
    lv_anim_set_time(&a, 1000);                                /* 设置动画时间 */
    lv_anim_set_playback_time(&a, 1000);       /* 设置动画反方向时间 */
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);     /* 设置无限循环 */
    lv_anim_set_values(&a, 0, 100);                            /* 设置动画始末值 */
    lv_anim_start(&a);                                         /* 开启动画 */
}
```

上述源码可知：我们开启 bar2 进度条的动画加载，该动画的回调函数为 anim_2_a_cb，该回调函数 hi 根据动画的时间来设置 bar2 进度条当前值。

(2) bar3 实现源码如下：

```
/**
 * @brief  例 3 进度条时间回调
 * @param  无
 * @return 无
*/
static void event_3_cb(lv_event_t* e)
{
    lv_obj_draw_part_dsc_t* dsc =
                                (lv_obj_draw_part_dsc_t *)lv_event_get_param(e);
    if (LV_PART_INDICATOR != dsc->part)
        return;

    lv_obj_t* obj = lv_event_get_target(e);

    lv_draw_label_dsc_t label_dsc;
    lv_draw_label_dsc_init(&label_dsc);
    if (scr_act_width() <= 320)
        label_dsc.font = &lv_font_montserrat_8;
    else if (scr_act_width() <= 480)
        label_dsc.font = &lv_font_montserrat_10;
    else
        label_dsc.font = &lv_font_montserrat_14;

    char buf[8];
    lv_snprintf(buf, sizeof(buf), "%d", lv_bar_get_value(obj));
```

```
    lv_point_t txt_size;
    lv_txt_get_size(&txt_size, buf, label_dsc.font,
                    label_dsc.letter_space, label_dsc.line_space,
                    LV_COORD_MAX, label_dsc.flag);

    lv_area_t txt_area;
    if (lv_area_get_height(dsc->draw_area) > txt_size.y + 5) {
        txt_area.y1 = dsc->draw_area->y1 + 5;
        txt_area.y2 = dsc->draw_area->y1 + txt_size.y + 1;
        label_dsc.color = lv_color_white();
    } else {
        txt_area.y1 = dsc->draw_area->y1 - txt_size.y - 5;
        txt_area.y2 = dsc->draw_area->y2 - 1;
        label_dsc.color = lv_obj_get_style_bg_color(obj, 0);
    }

    txt_area.x1 = dsc->draw_area->x1 +
                  (lv_area_get_width(dsc->draw_area) - txt_size.y) / 2;
    txt_area.x2 = txt_area.x1 + txt_size.x - 1;

    lv_draw_label(dsc->draw_ctx, &label_dsc, &txt_area, buf, NULL);
}

/**
 * @brief  例 3
 * @param  无
 * @return 无
*/
static void lv_example_bar_3(void)
{
    lv_obj_t* bar = lv_bar_create(lv_scr_act());/* 定义并初始化进度条 */
    /* 设置进度条大小 */
    lv_obj_set_size(bar, scr_act_height() / 24, scr_act_width() / 4);
    /* 设置进度条位置 */
    lv_obj_align(bar, LV_ALIGN_CENTER, scr_act_width() / 3, 0);
    /* 设置进度条事件回调 */
    lv_obj_add_event_cb(bar, event_3_cb, LV_EVENT_DRAW_PART_END, NULL);

    lv_anim_t a;                                /* 定义动画 */
    lv_anim_init(&a);                           /* 初始化动画 */
    lv_anim_set_var(&a, bar);                   /* 绑定物体 */
    lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t)anim_3_a_cb); /* 设置动画回调 */
```

```
    lv_anim_set_time(&a, 2000);                                    /* 设置动画时间 */
    lv_anim_set_playback_time(&a, 2000);    /* 设置动画返回时间 */
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);  /* 设置动画无限循环 */
    lv_anim_set_values(&a, 0, 100);                                /* 设置动画始末值 */
    lv_anim_start(&a);                                             /* 开启动画 */
}
```

上述源码可知：我们开启了 bar3 进度条动画加载，然后设置动画回调函数为 event_3_cb，该回调函数是根据动画的时间来设置 bar3 进度条当前值。

11.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 11.4.3.1 所示：

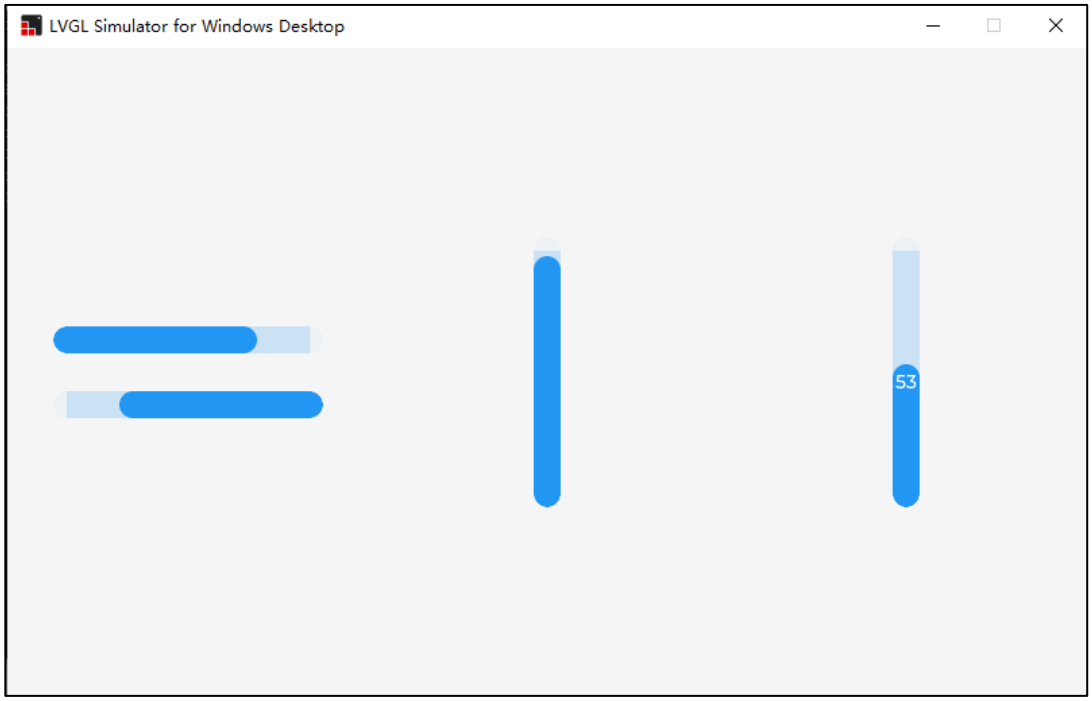


图 11.4.3.1 进度条实验

第十二章 按键部件(lv_btn)

在上几个章节中，我们已经了解了 LVGL 基础对象、arc 圆弧部件以及 bar 进度条部件的知识，本章我们来讲解 UI 常用的 button 按键部件，按钮部件是一个通用的输入或者输出部件，在我们实际开发中，该部件是必不可少的。 UI 按钮一般情况下是一个圆角矩形， 拥有按下或者释放的不同状态。注意： LVGL 的按钮并不像其他 GUI 一样可以设置文本， 而需要用户添加 label 作为文本显示。

本章分为如下几部分：

12.1 按键部件的组成

12.2 按键部件的相关知识

12.3 按键部件的 API 函数

12.4 按键部件的实验

12.1 按键部件的组成

按键部件(lv_btn)由背景组成，如图 12.1.1 所示：

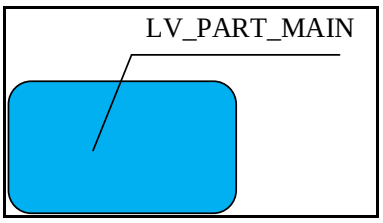


图 12.1.1 按键部件组成部分

从上图 12.1.1 所示：按键部件组成部分和基础对象的组成部分是类似的，那么按键部件和基础对象部件有什么区别呢，如以下所示：

(1) 默认情况下，不可滚动。

(2) 默认情况下，按键部件添加到默认组中。

(3) 按键部件默认高度和宽度设置为 LV_SIZE_CONTENT。

注意：上述的组成部分可以设置相关样式，样式的设置，请参考第六章 6.3 小节。

12.2 按键部件的相关知识

按键部件和基础对象部件类似，请大家查看第九章基础对象部件的讲解。

12.3 按键部件的 API 函数

lv_btn_create 函数

创建按键对象，其函数原型如以下所示：

```
lv_obj_t * lv_btn_create ( lv_obj_t *parent)
```

该函数的形参，如表 12.3.1 所示：

参数	描述
parent	部件的父类

表 12.3.1 函数 lv_btn_create()形参描述

返回值：返回对象控制块。

12.4 按键部件的实验

12.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要介绍 btn 部件的使用。实现现象：当按下左边的按键时，该按键中间的文本加 1；当按下中间的按键时，该按键主体发生改变；当按下右边的按键时，该按键主体发送抖动。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 12 btn(按键)》。

12.4.2 软件设计

12.4.2.1 程序流程图

本实验的程序流程图，如下图 12.4.2.1 所示：

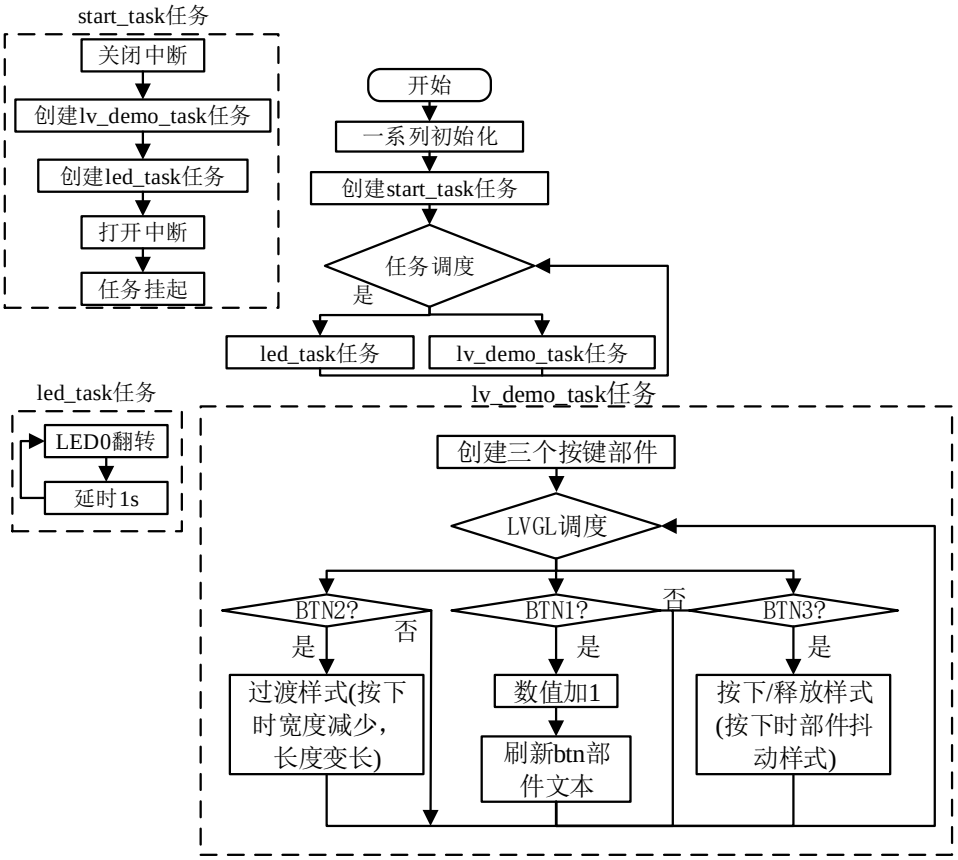


图 12.4.2.1 按键部件实验流程

12.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

(1) lv_mainstart 函数

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_btn_1();
    lv_example_btn_2();
    lv_example_btn_3();
}
```

(2) lv_example_btn_1 函数：创建 btn1 部件

```
/**
 * @brief  例 1 按键回调
 * @param  无
 * @return 无
 */
static void event_1_btn_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);

    static uint8_t pressed_cnt = 0;
    static uint8_t pressing_cnt = 0;
    char buf[15];

    if (LV_EVENT_PRESSED == code) {
        pressed_cnt++;
    }
    else if (LV_EVENT_PRESSING == code) {
        pressing_cnt++;
    }
    lv_snprintf(buf, sizeof(buf), "Buttom %d %d", pressed_cnt, pressing_cnt);
    lv_label_set_text((lv_obj_t*)e->user_data, buf);
}

/**
 * @brief  例 1
 * @param  无
 * @return 无
```

```
 */
static void lv_example_btn_1(void)
{
    lv_obj_t* btn;  /* 定义按钮 */
    btn = lv_btn_create(lv_scr_act());/* 初始化按钮 */
    /* 设置按钮大小 */
    lv_obj_set_size(btn, scr_act_width() / 4, scr_act_height() / 6);
    /* 设置按钮位置 */
    lv_obj_align(btn, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);

    lv_obj_t* label;                /* 定义标签 */
    label = lv_label_create(btn);   /* 初始化标签 */
    if (scr_act_width() <= 320)     /* 根据屏幕大小设置字体 */
        lv_obj_set_style_text_font(label, &lv_font_montserrat_10,
                                   LV_PART_MAIN);
    else if (scr_act_width() <= 480)
        lv_obj_set_style_text_font(label, &lv_font_montserrat_12,
                                   LV_PART_MAIN);
    else
        lv_obj_set_style_text_font(label, &lv_font_montserrat_14,
                                   LV_PART_MAIN);
    lv_label_set_text(label, "Buttom 0 0");     /* 设置标签文本 */
    lv_obj_center(label);                       /* 设置标签位置 */
    lv_obj_update_layout(label);                /* 手动更新参数 */
    /* 设置按钮回调 */
    lv_obj_add_event_cb(btn, event_1_btn_cb, LV_EVENT_ALL, label);
}
```

此函数主要创建 btn 部件并设置该部件相关属性和添加回调函数 event_1_btn_cb。该回调函数主要判断 btn1 是否被按下，如果按键被按下，则刷新 btn1 部件文本数值。

(3) lv_example_btn_2 函数：设置过渡特效

```
/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_btn_2(void)
{
    static lv_style_prop_t props[] = {
            LV_STYLE_TRANSFORM_WIDTH, LV_STYLE_TRANSFORM_HEIGHT,
            LV_STYLE_TEXT_LETTER_SPACE, LV_STYLE_PROP_INV
    };

    static lv_style_transition_dsc_t transition_dsc_def; /* 定义释放过渡特效 */
```

```
/* 初始化过渡特效 */
lv_style_transition_dsc_init(&transition_dsc_def, props,
                             lv_anim_path_bounce, 100, 0, NULL);

static lv_style_transition_dsc_t transition_dsc_pre; /* 定义按下过渡特效 */
/* 初始化过渡特效 */
lv_style_transition_dsc_init(&transition_dsc_pre, props,
                             lv_anim_path_ease_out, 300, 0, NULL);

static lv_style_t style_def;            /* 定义按钮释放时风格 */
lv_style_init(&style_def);              /* 初始化风格 */
/* 设置风格的过渡特效 */
lv_style_set_transition(&style_def, &transition_dsc_def);
static lv_style_t style_pre;            /* 定义按钮按下时风格 */

lv_style_init(&style_pre);              /* 初始化风格 */
/* 设置风格的过渡特效 */
lv_style_set_transition(&style_pre, &transition_dsc_pre);

lv_obj_t* btn;                          /* 定义按钮 */
btn = lv_btn_create(lv_scr_act());      /* 初始化按钮 */
/* 设置按钮大小 */
lv_obj_set_size(btn, scr_act_width() / 4, scr_act_height() / 6);
lv_obj_center(btn);                     /* 设置按钮位置 */
/* 按钮添加释放时风格 */
lv_obj_add_style(btn, &style_def, LV_STATE_DEFAULT);
/* 按钮添加按下时风格 */
lv_obj_add_style(btn, &style_pre, LV_STATE_PRESSED);
lv_obj_update_layout(btn);              /* 手动更新参数 */
/* 设置风格过渡后的相对宽度 */
lv_obj_set_style_transform_width(btn, lv_obj_get_width(btn) / 10,
                                 LV_STATE_PRESSED);
/* 设置风格过渡后的相对高度 */
lv_obj_set_style_transform_height(btn, -lv_obj_get_height(btn) / 10,
                                  LV_STATE_PRESSED);
/* 设置风格过渡后的相对字间距 */
lv_obj_set_style_text_letter_space(btn, lv_obj_get_width(btn) / 16,
                                   LV_STATE_PRESSED);
lv_obj_t* label; /* 定义标签 */


label = lv_label_create(btn);       /* 初始化标签 */
lv_obj_center(label);               /* 设置标签位置 */
```

```
    lv_label_set_text(label, "Buttom");     /* 这是标签文本 */
}
```

此函数创建一个 btn 部件，然后设置该部件的属性以及过渡样式，当按下时，btn 部件高度减少，宽度增大，释放 btn 部件时，恢复原样。

(3) lv_example_btn_3 函数：设置按下或者释放的样式

```
/**
 * @brief  例 3
 * @param  无
 * @return 无
 */
static void lv_example_btn_3(void)
{
    lv_obj_t* btn;                      /* 定义按钮 */
    btn = lv_btn_create(lv_scr_act()); /* 初始化按钮 */
    /* 设置按钮释放时风格 */
    lv_obj_set_style_radius(btn, 3, LV_STATE_DEFAULT);
    lv_obj_set_style_bg_opa(btn, LV_OPA_100, LV_STATE_DEFAULT);
    lv_obj_set_style_bg_color(btn, lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
    lv_obj_set_style_bg_grad_color(btn, lv_palette_darken(LV_PALETTE_BLUE, 2),
                               LV_STATE_DEFAULT);
    lv_obj_set_style_bg_grad_dir(btn, LV_GRAD_DIR_VER, LV_STATE_DEFAULT);
    lv_obj_set_style_border_opa(btn, LV_OPA_40, LV_STATE_DEFAULT);
    lv_obj_set_style_border_width(btn, 2, LV_STATE_DEFAULT);
    lv_obj_set_style_border_color(btn, lv_palette_main(LV_PALETTE_GREY),
                              LV_STATE_DEFAULT);
    lv_obj_set_style_shadow_width(btn, 8, LV_STATE_DEFAULT);
    lv_obj_set_style_shadow_color(btn, lv_palette_main(LV_PALETTE_GREY),
                              LV_STATE_DEFAULT);
    lv_obj_set_style_shadow_ofs_y(btn, 8, LV_STATE_DEFAULT);
    lv_obj_set_style_outline_opa(btn, LV_OPA_COVER, LV_STATE_DEFAULT);
    lv_obj_set_style_outline_color(btn, lv_color_white(), LV_STATE_DEFAULT);
    lv_obj_set_style_text_color(btn, lv_color_white(), LV_STATE_DEFAULT);
    lv_obj_set_style_pad_all(btn, 10, LV_STATE_DEFAULT);
    /* 设置按钮按下时风格 */
    lv_obj_set_style_outline_width(btn, 30, LV_STATE_PRESSED);
    lv_obj_set_style_outline_opa(btn, LV_OPA_TRANSP, LV_STATE_PRESSED);
    lv_obj_set_style_translate_y(btn, 5, LV_STATE_PRESSED);
    lv_obj_set_style_shadow_ofs_y(btn, 3, LV_STATE_PRESSED);
    lv_obj_set_style_bg_color(btn, lv_palette_darken(LV_PALETTE_BLUE, 1),
                              LV_STATE_PRESSED);
    lv_obj_set_style_bg_grad_color(btn, lv_palette_darken(LV_PALETTE_BLUE, 4),
                                   LV_STATE_PRESSED);
```

```
    static lv_style_transition_dsc_t trans_pre;
    static lv_style_prop_t props[] = { LV_STYLE_OUTLINE_WIDTH,
                                       LV_STYLE_OUTLINE_OPA, LV_STYLE_PROP_INV };
    lv_style_transition_dsc_init(&trans_pre, props, lv_anim_path_bounce,
                                 300, 0, NULL);
    lv_obj_set_style_transition(btn, &trans_pre, LV_STATE_PRESSED);

    lv_obj_set_width(btn, scr_act_width() / 4);    /* 设置按钮宽度 */
    lv_obj_set_height(btn, scr_act_height() / 6);  /* 设置按钮高度 */
    /* 设置按钮位置 */
    lv_obj_align(btn, LV_ALIGN_CENTER, scr_act_width() / 3, 0);

    lv_obj_t* label;                          /* 定义标签 */
    label = lv_label_create(btn);             /* 初始化标签 */
    lv_label_set_text(label, "Bottom");       /* 设置标签文本 */
    lv_obj_center(label);                     /* 设置标签位置 */
}
```

此函数创建一个 btn 部件，然后设置该部件的属性以及按下和释放的样式，这里非常简单，大家可以看一下实验效果。

12.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 12.4.3.1 所示：

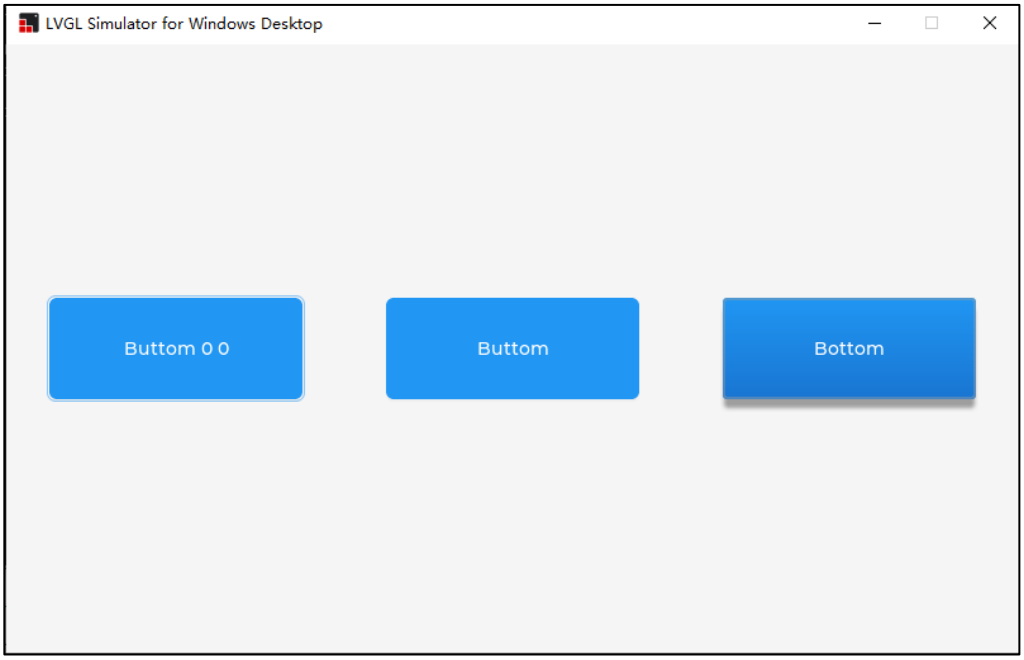


图 12.4.3.1 按键部件实验

第十三章 按键矩阵部件(lv_ btnmatrix)

LVGL 的 lv_btnmatrix 矩阵按钮对象可以看作是一系列伪按钮的集合，只不过它是以行和列有序的方式来排列这些子按钮，注意：这里故意加了一个伪字来说明它不是真正的 lv_btn 按钮对象，而是 lv_ btnmatrix 内部纯绘制出来的具有按钮外观的图形，而且这个图形具有和 lv_btn 按钮一样的点击效果，这种伪按钮的好处是它基本不占内存消耗(一个伪按钮大概需要 8 个字节)，这属于 LVGL 的一种优化操作，如果有多个按钮集合的应用场景的话，请最好使用 lv_ btnmatrix 部件，而不要使用 lv_btn 按钮的方案。

本章分为如下几部分：

13.1 按键矩阵部件的组成

13.2 按键矩阵部件的相关知识

13.3 按键矩阵部件的 API 函数

13.4 按键矩阵部件的实验

13.1 按键矩阵部件的组成

Button Matrix 对象是一种在行和列中显示多个按钮的轻量级方式。轻巧，因为按钮不是实际创建的，而是动态绘制的。这样，一个按钮只使用了 8 个额外字节的内存，而不是普通 Button 对象的约 100-150 个字节加上 label 对象的 100 个左右字节。

Button 矩阵被添加到默认组（如果设置了一个）。除了按钮矩阵是一个可编辑的对象，它也允许选择和单击带有编码器导航的按钮。

Button 矩阵部件有两个部分组成，如图 13.1.1 所示：

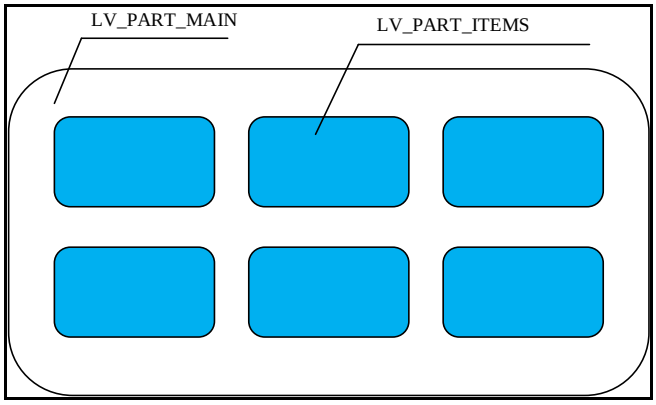


图 13.1.1 按键矩阵组成部分

1. LV_PART_MAIN：按钮矩阵的背景(典型的背景样式属性)，它可以设置 pad_row 和 pad_column 相关样式。

2. LV_PART_ITEMS：按键项：按钮的文本及背景样式。

注意：上述的组成部分可以设置相关样式，样式的设置，请参考第六章 6.3 小节。

13.2 按键矩阵部件的相关知识

13.2.1 按键矩阵部件的文本显示

在 LVGL 的 btnmatrix 部件的每个按钮上都有一个文本，如果我们要指定它们，则需要使用一个称为 map 的描述符字符串数组，该数组用来存放矩阵按键的文本如下源码所示：

```
const char * map[] = { "btn1", "btn2","btn3", "" };

void lv_mainstart()
{
    lv_obj_t * btnm1 = lv_btnmatrix_create(lv_scr_act());
    lv_btnmatrix_set_map(btnm1, map);
}
```

上述的源码可知，首先我们需要定义字符串数组，然后调用函数 lv_btnmatrix_create 创建矩阵按键，最后使用函数 lv_btnmatrix_set_map 把字符串数组映射到矩阵按键当中，如图 13.2.1.1 所示：

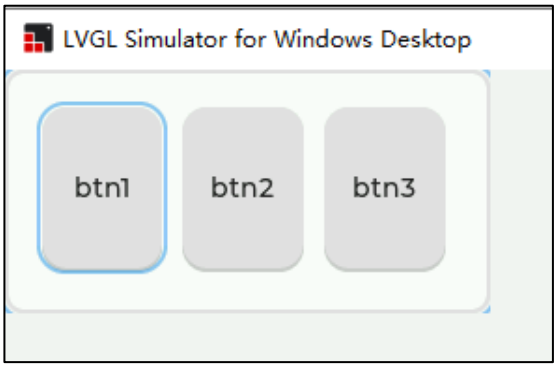


图 13.2.1.1 创建矩阵按键

注意：最后一个元素必须是空字符串!。

在字符串数组中使用“\n”表示换行，例如{"btn1"， "btn2"， "\n"， "btn3"， ""}，这样的数组不在局限于一行显示了，而是两行显示这些按键，注意：矩阵按键的每一行按钮宽度都是自动计算的，如下源码所示：

```
const char * map[] = { "btn1","\n", "btn2","btn3", "" };

void lv_mainstart()
{
    lv_obj_t * btnm1 = lv_btnmatrix_create(lv_scr_act());
    lv_btnmatrix_set_map(btnm1, map);
}
```

根据上述源码，我可以在 PC 模拟该代码，如图 13.2.1.2 所示：

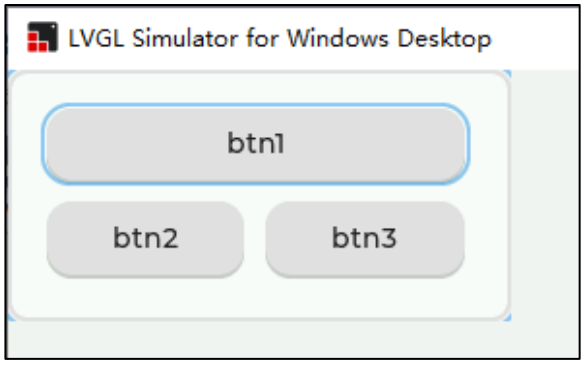


图 13.2.1.2 矩阵按键换行操作

当然我们所添加按键的编号跟数组一样，都是从 0 开始，所以数组有 3 个按键，分别编号是 0,1,2。

13.2.2 按键矩阵部件控制按键

按钮宽度可以相对于与同一行被设置为其他按钮中的线，例如两个按钮：btnA：宽度= 1 和，btnB：宽度= 2，btnA 将有 33%的宽度和 btnB 将有 66%的宽度。也就是说：有 3 个按钮，如果前两个按钮的宽度设置 1， 而最后一个按钮的宽度设置 2，则最后一个按钮的宽度将会是前面两个按钮的 2 倍。这样我们就可以设置矩阵按键的某按键宽度，该设置方法可调用以下函数设置：

```
lv_btnmatrix_set_btn_width(btnm1, 2, 2); /* 设置 id 为 2 的 子项设置 2 倍*/
```

根据上述的函数，我们可以得到图 13.2.2.1：

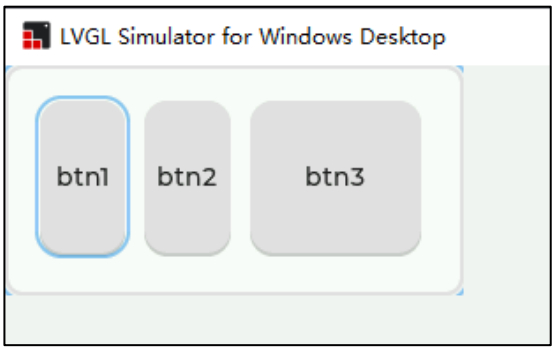


图 13.2.2.1 设置按键的宽度

由上图可知，btn3 按键的宽度大于 btn2 和 btn1 的宽度，而 btn1 和 btn2 的宽度相等。

另外，我们还可以通过函数 lv_btnmatrix_set_btn_ctrl 设置按键的类型，这些类型如下所示：

① LV_BTNMATRIX_CTRL_HIDDEN：将按钮隐藏（隐藏的按钮仍占据布局中的空间，它们不可见或不可单击）。

② LV_BTNMATRIX_CTRL_NO_REPEAT：长按按钮时禁用重复。

③ LV_BTNMATRIX_CTRL_DISABLED：禁用按钮。

④ LV_BTNMATRIX_CTRL_CHECKABLE：启用按钮切换。

⑤ LV_BTNMATRIX_CTRL_CHECKED：选中按钮。它将使用 LV_STATE_CHECHKED 样式

⑥ LV_BTNMATRIX_CTRL_POPOVER 按下此键时在弹出窗口中显示按钮标签。

⑦ LV_BTNMATRIX_CTRL_RECOLOR 使用 启用按钮文本的重新着色#。例如" #ff0000 red#"。

示例：

```
const char * map[] = { "btn1", "btn2","btn3", "" };

void lv_mainstart()
{
    lv_obj_t * btnm1 = lv_btnmatrix_create(lv_scr_act());
    lv_btnmatrix_set_map(btnm1, map);
    lv_obj_set_size(btnm1, 800, 480 / 2);
    lv_btnmatrix_set_btn_ctrl(btnm1, 0, LV_BTNMATRIX_CTRL_HIDDEN);
    lv_btnmatrix_set_btn_ctrl(btnm1, 1, LV_BTNMATRIX_CTRL_DISABLED);
    lv_btnmatrix_set_btn_ctrl(btnm1, 2, LV_BTNMATRIX_CTRL_CHECKABLE);
    lv_btnmatrix_set_btn_width(btnm1, 2, 2);
}
```

上述源码，我们使用函数 lv_btnmatrix_set_btn_ctrl 设置 btn1 为隐藏状态，而设置 btn2 为禁用状态(不能点击)，如图 13.2.2.2 所示：

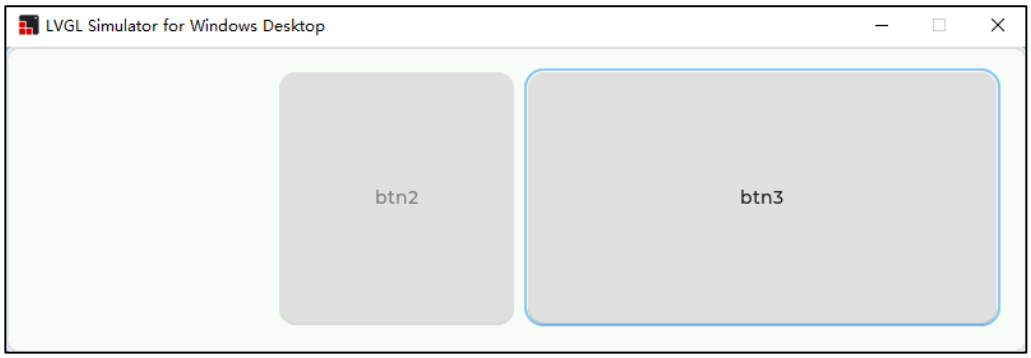


图 13.2.2.2 设置按键的状态

如果我们想要清除按键属性，可以使用函数 lv_btnmatrix_clear_btn_ctrl 清除按键类型。

13.2.3 按键矩阵部件的单次点击

单次点击是指只允许同一时间只有一个按键处于按下不弹起状态（Toggle）,简单来说：当按下 btn1 时时高亮状态，如图 13.2.3.1 和 13.2.3.2 所示，单次点击的设置可使用以下函数：

```
lv_btnmatrix_set_one_checked(btnm, true)
```

示例：

```
const char* map[] = { "btn1", "btn2","btn3", "" };

void lv_mainstart()
{
    lv_obj_t* btnm1 = lv_btnmatrix_create(lv_scr_act());
    lv_btnmatrix_set_map(btnm1, map);
    lv_btnmatrix_set_btn_ctrl(btnm1, 0, LV_BTNMATRIX_CTRL_CHECKABLE);
    lv_btnmatrix_set_btn_ctrl(btnm1, 1, LV_BTNMATRIX_CTRL_CHECKABLE);
    lv_btnmatrix_set_btn_ctrl(btnm1, 2, LV_BTNMATRIX_CTRL_CHECKABLE);
    lv_btnmatrix_set_btn_width(btnm1, 2, 2);
    lv_btnmatrix_set_one_checked(btnm1, true);
```

```
}
```

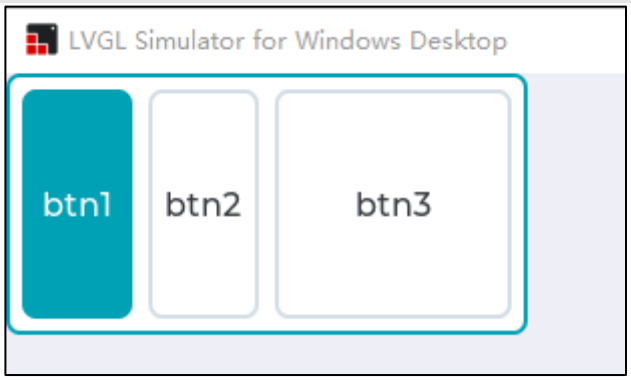


图 13.2.3.1 btn1 按键高亮

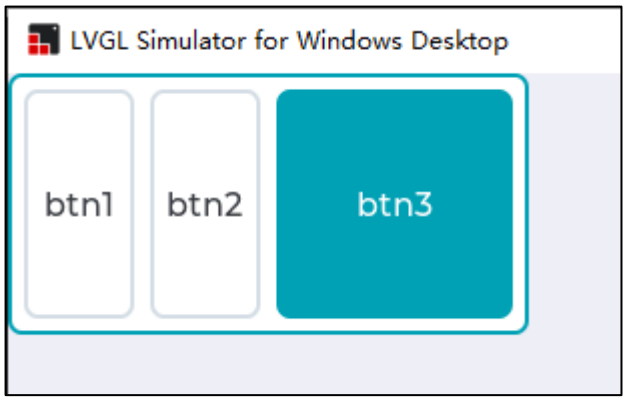


图 13.2.3.2 btn3 按键高亮

13.2.4 按键矩阵部件着色设置

默认情况下，矩阵按键中的按键文本都是黑色字体为主，当然我们可以设置这些文本为其他颜色，这个设置方法成为按键矩阵文本重新着色，它类似于 label 对象的重新着色功能，在着色之前必须调用以下函数开启矩阵按键重新着色功能，如下所示：

```
/* 设置 id 为 0， 该子项允许着色 */
lv_btnmatrix_set_btn_ctrl(btnm1, 0, LV_BTNMATRIX_CTRL_RECOLOR)
```

开启按键矩阵重新着色功能之后在字符串数组中定义着色字符，如带有“#FF0000 红色#”字符串，该字符串会把文本设置为红色，如下源码所示：

```
const char * map[] = { "#FF0000 btn1#", "btn2","btn3", "" };

void lv_mainstart()
{
    lv_obj_t * btnm1 = lv_btnmatrix_create(lv_scr_act(), NULL);
    lv_btnmatrix_set_map(btnm1, map);
lv_btnmatrix_set_btn_ctrl(btnm1, 0, LV_BTNMATRIX_CTRL_RECOLOR);
}
```

根据上述源码，我们得到图 13.2.4.1 所示：



图 13.2.4.1 为 btn1 着色

13.2.5 按键矩阵部件的事件

1. 输入设备事件

LV_EVENT_VALUE_CHANGED：当一个按钮被按下/释放或长按后重复时发送。

2. 绘画事件

LV_BTNMATRIX_DRAW_PART_BTN：单个按钮。

13.3 按键矩阵部件的 API 函数

LVGL 官方提供矩阵按键(lv_btnmatrix)部件相关 API，如表 13.3.1 所示：

函数	描述
lv_btnmatrix_create()	创建一个按钮矩阵对象
lv_btnmatrix_set_map()	设置子项按键
lv_btnmatrix_set_ctrl_map()	设置按钮矩阵的按钮控制图
lv_btnmatrix_set_selected_btn()	设置选中的按钮
lv_btnmatrix_set_btn_ctrl()	设置按钮矩阵的一个按钮的属性
lv_btnmatrix_clear_btn_ctrl()	清除按钮矩阵的某个按钮的属性
lv_btnmatrix_set_btn_ctrl_all()	设置按钮矩阵的所有按钮的属性
lv_btnmatrix_clear_btn_ctrl_all()	清除按钮矩阵的所有按钮的属性
lv_btnmatrix_set_btn_width()	设置单个按钮的相对宽度
lv_btnmatrix_set_one_checked()	使按钮矩阵像一个选择器小部件
lv_btnmatrix_get_map()	获取按钮矩阵的当前映射
lv_btnmatrix_get_selected_btn()	获取用户最后“激活”按钮的索引
lv_btnmatrix_get_btn_text()	获取按钮的文本
lv_btnmatrix_has_btn_ctrl()	获取按钮矩阵的按钮是否启用或禁用部件值
lv_btnmatrix_get_one_checked()	是否启用了“一次检查”模式

表 13.3.1 矩阵按键(lv_btnmatrix)部件相关 API 函数

接下来我们来讲解 LVGL 矩阵按键部件常用的函数有哪些：

1. lv_btnmatrix_create 函数

创建矩阵按键对象，其函数原型如以下所示：

```
lv_obj_t * lv_ btnmatrix_create ( lv_obj_t *parent)
```

该函数的形参，如表 13.3.2 所示：

参数	描述
parent	部件的父类

表 13.3.2 函数 lv_ btnmatrix_create()形参描述

返回值：返回对象控制块。

2. lv_btnmatrix_set_map 函数

设置子项按键，其函数原型如以下所示：

```
void lv_btnmatrix_set_map ( lv_obj_t * obj , const char * map [ ] )
```

该函数的形参，如表 13.3.3 所示：

参数	描述
obj	指向按钮矩阵对象的指针
map	指针一个字符串数组,最后一个字符串必须是：“”,使用“\n”进行换行

表 13.3.3 函数 lv_btnmatrix_set_map()形参描述

返回值：无。

3. lv_btnmatrix_set_ctrl_map 函数

设置按钮矩阵的按钮控制图，其函数原型如以下所示：

```
void lv_btnmatrix_set_ctrl_map ( lv_obj_t * obj ,
                                 const lv_btnmatrix_ctrl_t ctrl_map [ ] )
```

该函数的形参，如表 13.3.4 所示：

参数	描述
obj	指向按钮矩阵对象的指针
ctrl_map	指向控制字节数组的指针

表 13.3.4 函数 lv_btnmatrix_set_ctrl_map()形参描述

返回值：无。

4. lv_btnmatrix_set_selected_btn 函数

设置选中的按钮，其函数原型如以下所示：

```
void lv_btnmatrix_set_selected_btn ( lv_obj_t * obj , uint16_t btn_id )
```

该函数的形参，如表 13.3.5 所示：

参数	描述
obj	指向按钮矩阵对象的指针
btn_id	要修改的按钮的基于 0 的索引。（不计算新行）

表 13.3.5 函数 lv_btnmatrix_set_selected_btn()形参描述

返回值：无。

5. lv_btnmatrix_set_btn_ctrl 函数

设置按钮矩阵的一个按钮的属性，其函数原型如以下所示：

```
void lv_btnmatrix_set_btn_ctrl ( lv_obj_t * obj ,
                                 uint16_t btn_id ,
                                 lv_btnmatrix_ctrl_t ctrl )
```

该函数的形参，如表 13.3.6 所示：

参数	描述
obj	指向按钮矩阵对象的指针
btn_id	要修改的按钮的基于 0 的索引。（不计算新行）
ctrl	OR-ed 属性，请参考本教程 13.2.2 小节

表 13.3.6 函数 lv_btnmatrix_set_btn_ctrl()形参描述

返回值：无。

6. lv_btnmatrix_clear_btn_ctrl 函数

清除按钮矩阵的某个按钮的属性，其函数原型如以下所示：

```
void lv_btnmatrix_clear_btn_ctrl ( lv_obj_t * obj ,
                                   uint16_t btn_id ,
```

```
                                    lv_btnmatrix_ctrl_t ctrl )
```

该函数的形参，如表 13.3.7 所示：

参数	描述
obj	指向按钮矩阵对象的指针
btn_id	要修改的按钮的基于 0 的索引。（不计算新行）
ctrl	OR-ed 属性，请参考本教程 13.2.2 小节

表 13.3.7 函数 lv_btnmatrix_clear_btn_ctrl()形参描述

返回值：无。

7. lv_btnmatrix_set_btn_ctrl_all 函数

设置按钮矩阵的所有按钮的属性，其函数原型如以下所示：

```
void lv_btnmatrix_set_btn_ctrl_all (  lv_obj_t * obj ,
                                      lv_btnmatrix_ctrl_t ctrl )
```

该函数的形参，如表 13.3.8 所示：

参数	描述
obj	指向按钮矩阵对象的指针
ctrl	OR-ed 属性，请参考本教程 13.2.2 小节

表 13.3.8 函数 lv_btnmatrix_set_btn_ctrl_all()形参描述

返回值：无。

8. lv_btnmatrix_clear_btn_ctrl_all 函数

清除按钮矩阵的所有按钮的属性，其函数原型如以下所示：

```
void lv_btnmatrix_clear_btn_ctrl_all (  lv_obj_t * obj ,
                                        lv_btnmatrix_ctrl_t ctrl )
```

该函数的形参，如表 13.3.9 所示：

参数	描述
obj	指向按钮矩阵对象的指针
ctrl	OR-ed 属性，请参考本教程 13.2.2 小节

表 13.3.9 函数 lv_btnmatrix_clear_btn_ctrl_all()形参描述

返回值：无。

9. lv_btnmatrix_set_btn_width 函数

设置单个按钮的相对宽度，其函数原型如以下所示：

```
void lv_btnmatrix_set_btn_width (  lv_obj_t * obj ,
                                   uint16_t btn_id ,
                                   uint8_t width)
```

该函数的形参，如表 13.3.10 所示：

参数	描述
obj	指向按钮矩阵对象的指针
btn_id	要修改的按钮的基于 0 的索引
width	与同一行中的按钮相比的相对宽度[1..7]

表 13.3.10 函数 lv_btnmatrix_set_btn_width()形参描述

返回值：无。

10. lv_btnmatrix_set_one_checked 函数

一次只能检查一个按钮，其函数原型如以下所示：

```
void lv_btnmatrix_set_one_checked ( lv_obj_t * obj , bool en )
```

该函数的形参，如表 13.3.11 所示：

参数	描述
obj	指向按钮矩阵对象的指针
en	是否启用“一次检查”模式

表 13.3.11 函数 lv_btnmatrix_set_one_checked()形参描述

返回值：无。

13.4 按键矩阵部件的实验

13.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数使用矩阵按键构建多种键盘，如数字键盘、大小写键盘等。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 13 btnmatrix (按键)》。

13.4.2 软件设计

13.4.2.1 程序流程图

本实验的程序流程图，如下图 13.4.2.1 所示：

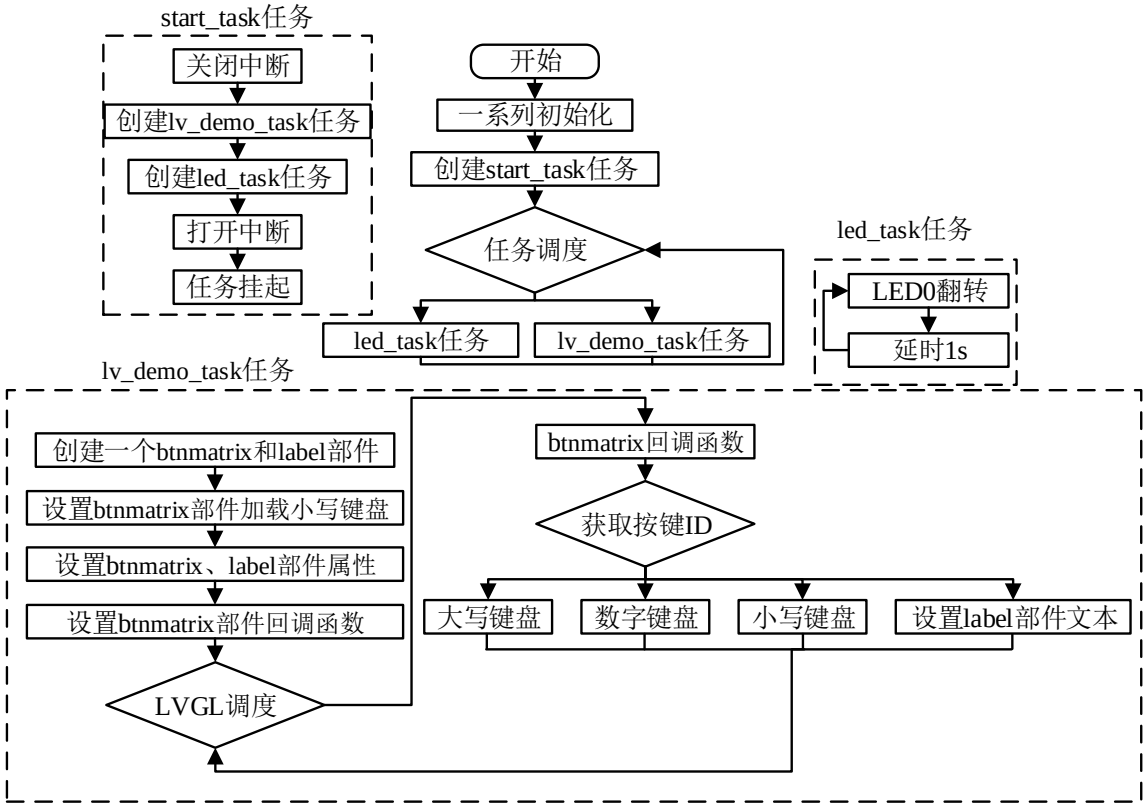


图 13.4.2.1 btnmatrix 部件实验流程图

13.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

(1) lv_mainstart 函数

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_btnmatrix(void)
{
    /* 定义并初始化按钮矩阵 */
    lv_obj_t* keyboard = lv_btnmatrix_create(lv_scr_act());
    /* 设置按钮矩阵大小 */
    lv_obj_set_size(keyboard, scr_act_width(), scr_act_height() / 2);
    /* 加载小写键盘 */
    load_small_keymap(keyboard);
    /* 设置按钮矩阵位置 */
    lv_obj_align(keyboard, LV_ALIGN_BOTTOM_MID, 0, 0);
    /* 手动更新按钮矩阵参数 */
    lv_obj_update_layout(keyboard);
    /* 定义并初始化标签 */
    lv_obj_t* label = lv_label_create(lv_scr_act());
    /* 设置文本居中对齐 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 设置标签位置 */
    lv_obj_align(label, LV_ALIGN_TOP_MID, 0, lv_obj_get_y(keyboard) / 2);
    /* 设置标签文本 */
    lv_label_set_text(label, "Wiat input...");
    /* 设置按钮矩阵回调 */
    lv_obj_add_event_cb(keyboard, event_keyboard_cb, LV_EVENT_ALL, label);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_btnmatrix();
}
```

此函数主要调用 lv_example_btnmatrix 创建 btnmatrix 部件以及 label 部件并设置它们相关属性，最后调用 lv_obj_add_event_cb 设置 btnmatrix 部件的回调函数。

(2) event_keyboard_cb 回调函数

```
/**
 * @brief  矩阵按钮回调
 * @param  e:事件
 * @return 无
 */
static void event_keyboard_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* keyboard = lv_event_get_target(e);
    uint32_t id;

    if (LV_EVENT_PRESSED == code) {
        /* 获取按键 ID */
        id = lv_btnmatrix_get_selected_btn(keyboard)
        if (((0 == keyboard_state) || (1 == keyboard_state)) && (0 == id)) {
            /* 字母键盘切换到数字键盘 */
            load_num_keymap(keyboard);
        }
        else if ((2 == keyboard_state) && (11 == id)) {
            /* 数字键盘切换到小写键盘 */
            load_small_keymap(keyboard);
        }
        else if ((0 == keyboard_state) && (12 == id)) {
            /* 小写键盘切换到大写键盘 */
            load_big_keymap(keyboard);
        }
        else if ((1 == keyboard_state) && (12 == id)) {
            /* 大写键盘切换到小写键盘 */
            load_small_keymap(keyboard);
        }
        else if (id <= 39){
            char buf[12];
            lv_snprintf(buf, sizeof(buf), "%s",
            lv_btnmatrix_get_btn_text(keyboard, id));
            lv_label_set_text((lv_obj_t*)e->user_data, buf);
        }
    }
}
```

此函数是 btnmatrix 部件回调函数，当我们按下“1#”按键时字母键盘切换到数字键盘；当我们按下“ABC” 按键时小写键盘切换到大写键盘。

13.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 13.4.3.1 所示：

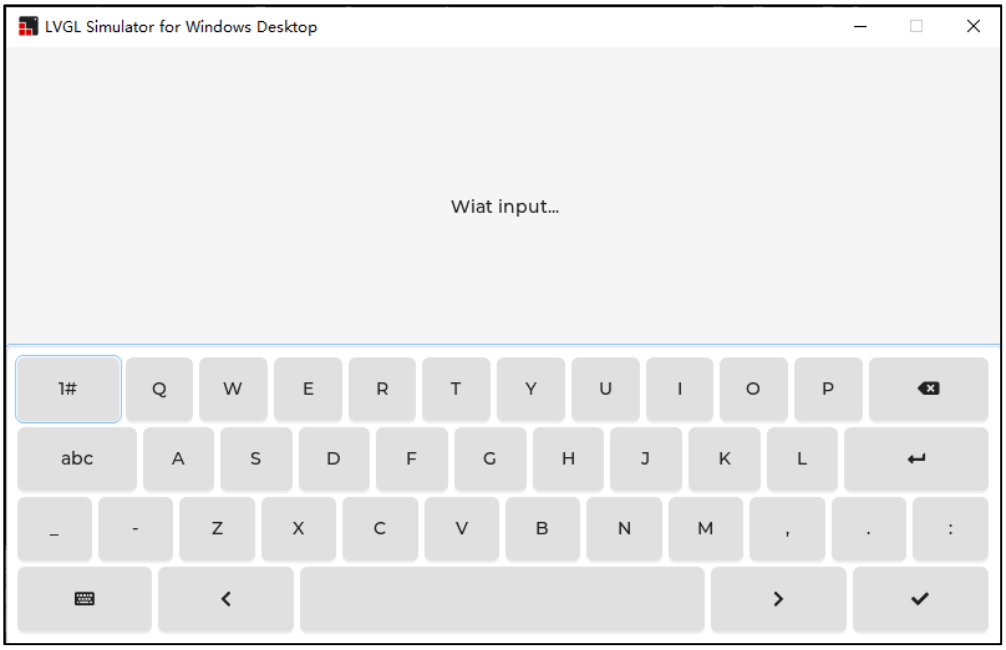


图 13.4.3.1 btnmatrix 部件实验

第十四章 画布部件(lv_canvas)

LVGL 的画布部件继承 image 图像，用户可以在其中绘制任何内容。它可以使用 LVGL 的绘图引擎在此处绘制矩形，文本，图像，圆弧。除了一些“效果”，还可以应用旋转，缩放和模糊等效果图。

本章分为如下几部分：

14.1 画布部件的组成

14.2 画布部件的相关知识

14.3 画布部件的 API 函数

14.4 画布部件的实验

14.1 画布部件的组成

画布部件的组成只有一个典型的背景样式和图片样式属性(LV_PART_MAIN)。

注意：上述的组成部分可以设置相关样式，样式的设置，请参考第六章 6.3 小节。

14.2 画布部件的相关知识

14.2.1 画布缓冲区定义

画布需要一个缓冲区来存储绘制的图像，我们要将缓冲区分配给画布，才能在画布中绘画图下，画布缓冲区分配方法可调用以下函数定义：

```
lv_canvas_set_buffer(canvas, buffer, width, height, LV_IMG_CF_...)
```

其中 buffer 是一个静态缓冲区，它是用来保存画布的图像的。该静态缓冲区定义方式如下所示：

```
static lv_color_t buffer[LV_CANVAS_BUF_SIZE_TRUE_COLOR(width, height)]
```

上述源码 LV_CANVAS_BUF_SIZE_……宏是帮助确定不同颜色格式的缓冲区的大小，画布支持所有内置的颜色格式，如 LV_IMG_CF_TRUE_COLOR 或 LV_IMG_CF_INDEXED_2BIT。在颜色格式部分中可以看到完整的列表。

接下来我们分几个步骤来讲述画布部件的使用流程，如以下步骤所示：

第一步：为画布申请缓冲区内存，如下源码所示：

```
#define CANVAS_WIDTH    200
#define CANVAS_HEIGHT   150

/* 第一步：为画布申请缓冲区内存 */
static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(CANVAS_WIDTH,
                       CANVAS_HEIGHT)];
```

第二步：创建一个画布并初始化它的调色板，如下源码所示：

```
/* 第二步：创建一个画布*/
lv_obj_t * canvas = lv_canvas_create(lv_scr_act());
```

第三步：为画布设置缓冲区，如下源码所示：

```
/* 第三步：为画布设置缓冲区 */
lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH,
```

```
CANVAS_HEIGHT, LV_IMG_CF_TRUE_COLOR);
```

经过上述的步骤，我们就可以创建一个画布部件，如图 14.2.1.1 所示：

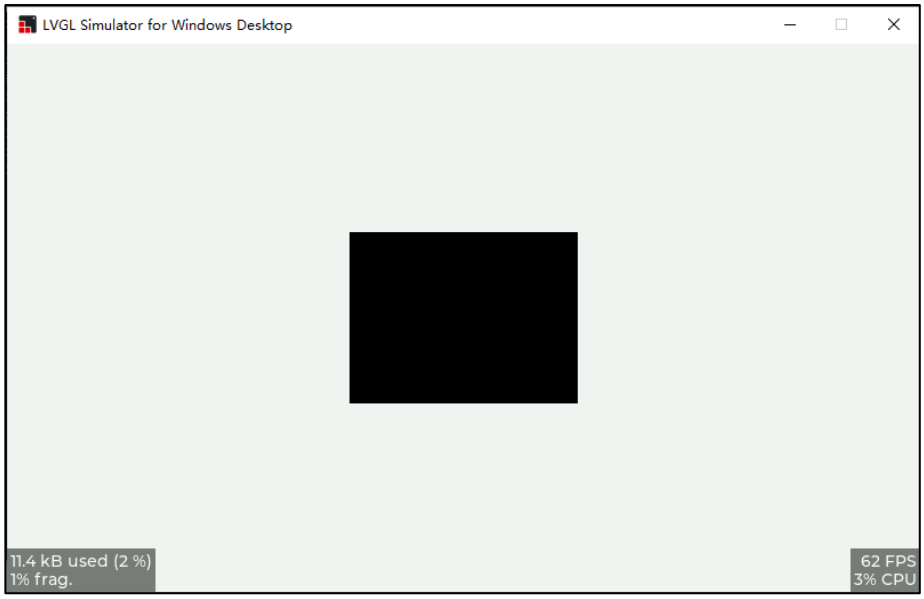


图 14.2.1.1 创建画布

注意：使用画布部件时，上述的步骤是必须的，它们缺一不可。

14.2.2 画布调色板设置

调色板初始化，可调用以下函数初始化：

```
lv_canvas_set_palette(canvas, 3, LV_COLOR_RED)
```

上述源码表示它将 index = 3 的像素设置为红色。

14.2.3 画布部件的绘画

要在画布上设置一个像素，使用以下函数设置：

```
lv_canvas_set_px _color(canvas, x, y, LV_COLOR_RED)
```

与 LV_IMG_CF_INDEXED_…或 LV_IMG_CF_ALPHA_…，颜色的索引或 alpha 值需要作为颜色传递。例如 lv_color_t c ; c.full = 3;

也可以使用以下函数来填充画布的背景颜色：

```
lv_canvas_fill_bg(canvas, LV_COLOR_BLUE, LV_OPA_50)
```

上述的函数表示：用 50%的不透明度将整个画布填充成蓝色。注意：如果当前的颜色格式不支持颜色(例如 LV_IMG_CF_ALPHA_2BIT)，该颜色将被忽略。类似地，如果不支持不透明度(例如 LV_IMG_CF_TRUE_COLOR)，它也将被忽略。

如果一个像素数组可以被复制到画布，请使用一些函数定义：

```
lv_canvas_copy_buf(canvas, buffer_to_copy, x, y, width, height)
```

注意：缓冲区和画布的颜色格式需要匹配。

在前面，笔者已经说过，画布部件可以绘画矩形，文本，图像等图片，如以下函数所示：

```
/* 绘制矩形 */
lv_canvas_draw_rect(canvas, x, y, width, heigth, &draw_dsc)
/* 绘制文本 */
lv_canvas_draw_text(canvas, x, y, max_width, &draw_dsc, txt,
```

```
                    LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)
/* 绘制图像 */
lv_canvas_draw_img(canvas, x, y, &img_src, &draw_dsc)
/* 绘制线 */
lv_canvas_draw_line(canvas, point_array, point_cnt, &draw_dsc)
/* 绘制多边形 */
lv_canvas_draw_polygon(canvas, points_array, point_cnt, &draw_dsc)
/* 绘制圆弧 */
lv_canvas_draw_arc(canvas, x, y, radius, start_angle, end_angle, &draw_dsc)
```

上述函数的 Draw_dsc 形参是一个 lv_draw_rect/label/img/line_dsc_t 变量，它应该首先被初始化，请使用以下函数初始化：

```
lv_draw_rect/label/img/line_dsc_init()
```

如以下步骤所示：

(1) 定义一个描述符，例如 lv_draw_rect_dsc_t rect_dsc。

(2) 调用函数 lv_draw_rect_dsc_init 初始化函数。

(3) 描述符定义属性，如下源码所示：

```
#define CANVAS_WIDTH        200
#define CANVAS_HEIGHT       150

void lv_mainstart(void)
{
    lv_draw_rect_dsc_t rect_dsc;
    /* 第一步：为画布申请缓冲区内存 */
    static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(CANVAS_WIDTH,
                                                         CANVAS_HEIGHT)];
    /* 第二步：创建一个画布并初始化它的调色板 */
    lv_obj_t* canvas = lv_canvas_create(lv_scr_act());
    lv_obj_align(canvas,LV_ALIGN_CENTER,0,0);
    /* 第三步：为画布设置缓冲区 */
    lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH,
    CANVAS_HEIGHT, LV_IMG_CF_TRUE_COLOR);
    lv_draw_rect_dsc_init(&rect_dsc);
    rect_dsc.radius = 10;                       /* 设置圆角属性为 10 */
    rect_dsc.bg_opa = LV_OPA_COVER;             /* 设置透明覆盖 */
    rect_dsc.bg_color = lv_palette_main(LV_PALETTE_RED);  /* 设置背景延时为红色 */
    rect_dsc.border_width = 2;                  /* 设置边框厚度为 2 */
    rect_dsc.border_opa = LV_OPA_90;            /* 设置边框透明 */
    rect_dsc.border_color = lv_color_white();   /* 设置边框颜色 */
    rect_dsc.shadow_width = 5;                  /* 设置阴影 */
    rect_dsc.shadow_ofs_x = 5;                  /* 设置阴影偏移 x */
    rect_dsc.shadow_ofs_y = 5;                  /* 设置阴影偏移 y */
    lv_canvas_draw_rect(canvas, 70, 60, 100, 70, &rect_dsc);
}
```

经过上述步骤所示，其他图形绘画步骤相似，如图 14.2.3.1 所示：

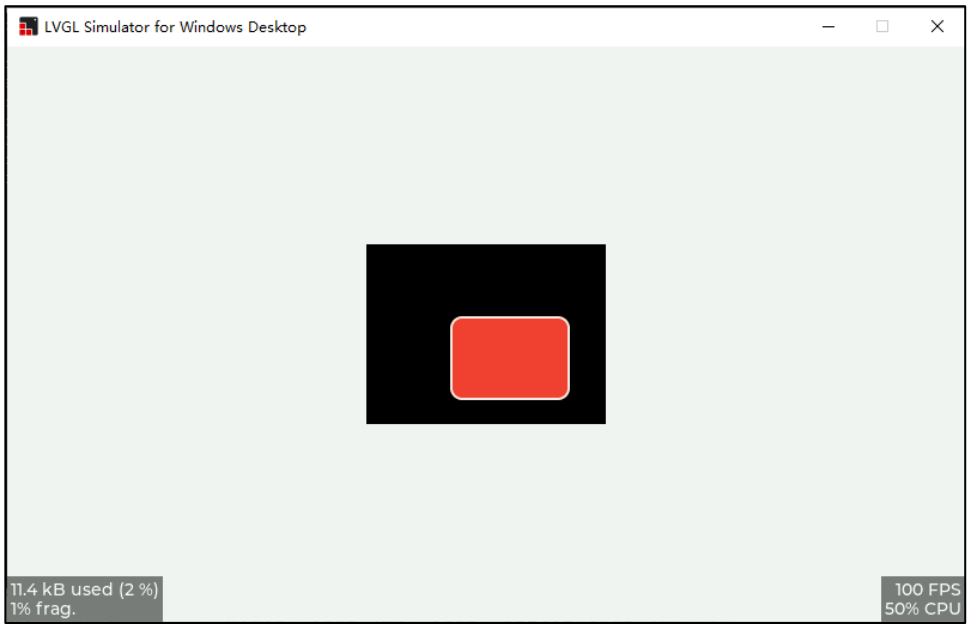


图 14.2.3.1 绘画矩形

14.2.4 画布部件的旋转

canvas 部件的旋转可以使用函数 lv_canvas_transform 可用于旋转和/或缩放图像的图像，并将结果存储在画布上。该函数需要参数如表 14.2.4.1 所示：

参数	描述
canvas	指向画布对象
img_pointer	转换为图像描述符
angle	旋转角度(0..3600)，以 0.1 度为分辨率
zoom	缩放系数(256 不缩放，512 倍大，128 缩小一倍)
offset_x	偏移量 X
offset_y	偏移量 Y
pivot_x	旋转的 X 轴坐标
pivot_y	旋转的枢轴 Y 坐标
antialias	true:在转换过程中应用抗锯齿

表 14.2.4.1 lv_canvas_transform()函数形参描述

注意：画布不能自己旋转。需要一个源和目标画布或图像。

上述笔者说过，画布不能自转，它需要一个源目标和图像来旋转，简单来说：它需要另一个存储原始图像的缓冲区，然后复制当前图像到缓冲区，并将其旋转到画布当中。下面我们来讲解图像旋转的步骤，如以下所示：

第一步：创建一个数组，如下源码所示：

```
static lv_color_t cbuf_tmp[CANVAS_WIDTH * CANVAS_HEIGHT];
```

第二步：复制源目标数据到该数组中，如下源码所示：

```
memcpy(cbuf_tmp, cbuf, sizeof(cbuf_tmp));
```

第三步：定义一个 lv_img_dsc_t 描述符，如下源码所示：

```
lv_img_dsc_t img;
```

第四步：描述符定义属性，如下源码所示：

```
img.data = (void *)cbuf_tmp;            /* 设置图片数据 */
img.header.cf = LV_IMG_CF_TRUE_COLOR;   /* 设置不透明 */
img.header.w = CANVAS_WIDTH;            /* 设置宽度 */
img.header.h = CANVAS_HEIGHT;           /* 设置高度 */
```

第五步：调用函数 lv_canvas_transform()旋转，如下源码所示：

```
lv_canvas_transform(canvas, &img, 30, LV_IMG_ZOOM_NONE,
0, 0, CANVAS_WIDTH / 2, CANVAS_HEIGHT / 2, true);
```

代码实例：

```
#define CANVAS_WIDTH        200
#define CANVAS_HEIGHT       150

void lv_mainstart(void)
{
    lv_draw_rect_dsc_t rect_dsc;
    /* 第一步：为画布申请缓冲区内存 */
    static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(CANVAS_WIDTH,
        CANVAS_HEIGHT)];
    /* 第二步：创建一个画布并初始化它的调色板 */
    lv_obj_t* canvas = lv_canvas_create(lv_scr_act());
    lv_obj_align(canvas,LV_ALIGN_CENTER,0,0);
    /* 第三步：为画布设置缓冲区 */
    lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH,
        CANVAS_HEIGHT, LV_IMG_CF_TRUE_COLOR);
    lv_draw_rect_dsc_init(&rect_dsc);
    rect_dsc.radius = 10;                       /* 设置圆角属性为 10 */
    rect_dsc.bg_opa = LV_OPA_COVER;             /* 设置透明覆盖 */
    rect_dsc.bg_color = lv_palette_main(LV_PALETTE_RED);  /* 设置背景延时为红色 */
    rect_dsc.border_width = 2;                  /* 设置边框厚度为 2 */
    rect_dsc.border_opa = LV_OPA_90;            /* 设置边框透明 */
    rect_dsc.border_color = lv_color_white();   /* 设置边框颜色 */
    rect_dsc.shadow_width = 5;                  /* 设置阴影 */
    rect_dsc.shadow_ofs_x = 5;                  /* 设置阴影偏移 x */
    rect_dsc.shadow_ofs_y = 5;                  /* 设置阴影偏移 y */
    lv_canvas_draw_rect(canvas, 70, 60, 100, 70, &rect_dsc);

    static lv_color_t cbuf_tmp[CANVAS_WIDTH * CANVAS_HEIGHT];
    memcpy(cbuf_tmp, cbuf, sizeof(cbuf_tmp));
    lv_img_dsc_t img;
    img.data = (const uint8_t *)cbuf_tmp;           /* 设置图片数据 */
    img.header.cf = LV_IMG_CF_TRUE_COLOR;           /* 设置不透明 */
    img.header.w = CANVAS_WIDTH;                    /* 设置宽度 */
    img.header.h = CANVAS_HEIGHT;                   /* 设置高度 */
    lv_canvas_transform(canvas, &img, 30, LV_IMG_ZOOM_NONE, 0, 0,
```

```
                        CANVAS_WIDTH / 2, CANVAS_HEIGHT / 2, true);
}
```

上述源码可知：首先创建画布部件，然后在画布部件内绘画矩形以及设置其属性，然后复制源目标数据到该数组(cbuf_tmp)，最后该目标源进行旋转，如图 14.2.4.1 所示：

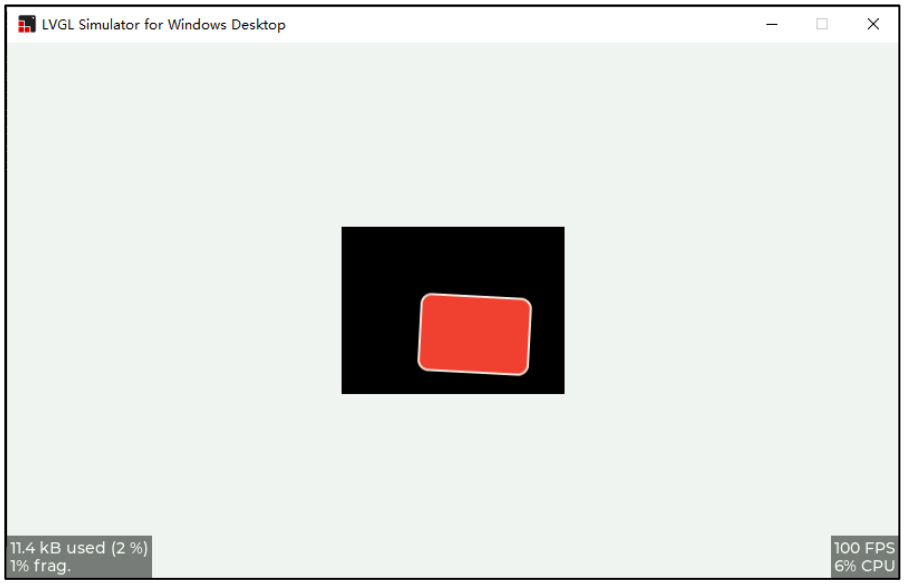


图 14.2.4.1 设置目标源旋转

14.2.5 画布部件的模糊处理

如果画布的给定区域可以被水平模糊，如以下函数所示：

```
lv_canvas_blur_hor(canvas, &area, r)
```

垂直模糊请使用以下函数处理：

```
lv_canvas_blur_ver(canvas, &area, r)
```

此函数的 r 形参是模糊的半径(越大意味着越强烈的毛刺)。区域是应该应用模糊的区域(相对于画布解释)。

代码实例：

```
#define CANVAS_WIDTH    200
#define CANVAS_HEIGHT   150
void lv_mainstart(void)
{
    lv_draw_rect_dsc_t rect_dsc;
    /* 第一步：为画布申请缓冲区内存 */
    static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(CANVAS_WIDTH,
        CANVAS_HEIGHT)];
    /* 第二步：创建一个画布并初始化它的调色板 */
    lv_obj_t* canvas = lv_canvas_create(lv_scr_act());
    lv_obj_align(canvas,LV_ALIGN_CENTER,0,0);
    /* 第三步：为画布设置缓冲区 */
    lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH,
                         CANVAS_HEIGHT, LV_IMG_CF_TRUE_COLOR);
    lv_draw_rect_dsc_init(&rect_dsc);
```

```
    rect_dsc.radius = 10;                           /* 设置圆角属性为 10 */
    rect_dsc.bg_opa = LV_OPA_COVER;                 /* 设置透明覆盖 */
    rect_dsc.bg_color = lv_palette_main(LV_PALETTE_RED);  /* 设置背景延时为红色 */
    rect_dsc.border_width = 2;                      /* 设置边框厚度为 2 */
    rect_dsc.border_opa = LV_OPA_90;                /* 设置边框透明 */
    rect_dsc.border_color = lv_color_white();       /* 设置边框颜色 */
    rect_dsc.shadow_width = 5;                      /* 设置阴影 */
    rect_dsc.shadow_ofs_x = 5;                      /* 设置阴影偏移 x */
    rect_dsc.shadow_ofs_y = 5;                      /* 设置阴影偏移 y */
    lv_canvas_draw_rect(canvas, 70, 60, 100, 70, &rect_dsc);
    static lv_color_t cbuf_tmp[CANVAS_WIDTH * CANVAS_HEIGHT];
    memcpy(cbuf_tmp, cbuf, sizeof(cbuf_tmp));
    lv_img_dsc_t img;
    img.data = (const uint8_t *)cbuf_tmp;           /* 设置图片数据 */
    img.header.cf = LV_IMG_CF_TRUE_COLOR;           /* 设置不透明 */
    img.header.w = CANVAS_WIDTH;                    /* 设置宽度 */
    img.header.h = CANVAS_HEIGHT;                   /* 设置高度 */
    lv_canvas_transform(canvas, &img, 30, LV_IMG_ZOOM_NONE, 0, 0,
                        CANVAS_WIDTH / 2, CANVAS_HEIGHT / 2, true);
    lv_area_t area;
    area.x1 = 70;       /* 设置区域 x1 */
    area.y1 = 60;       /* 设置区域 y1 */
    area.x2 = 100;      /* 设置区域 x2 */
    area.y2 = 100;      /* 设置区域 y2 */
    lv_canvas_blur_hor(canvas, &area, 40);/* 设置水平模糊 */
}
```

此函数在前面以及讲解，这里我们只讲解迷糊设置的函数使用，首先设置模糊区域，然后调用 lv_canvas_blur_hor 函数设置在模糊区域执行水平模糊处理。如图 14.2.5.1 所示：

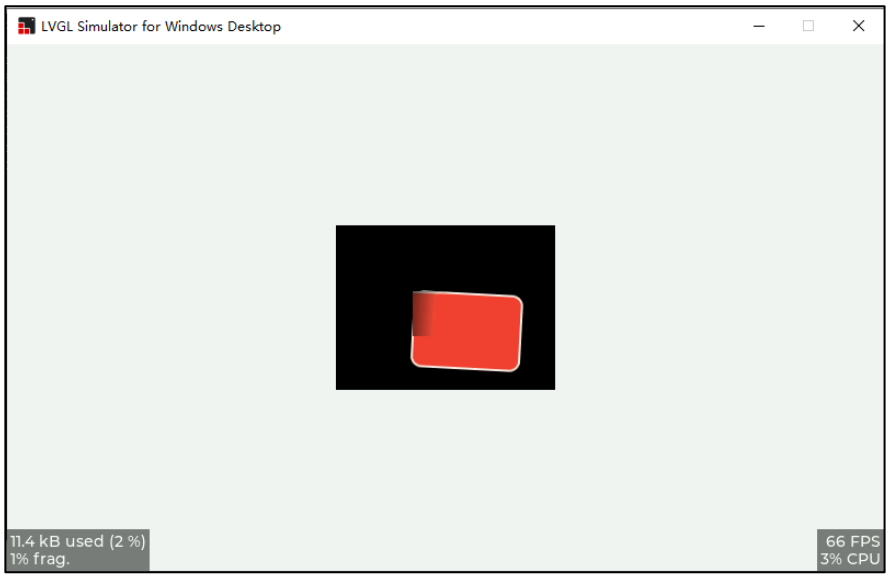


图 14.2.5.1 设置模糊区域

14.3 画布部件的 API 函数

下面笔者来讲解以下 canvas 部件提供的函数作用，如表 14.3.1 所示：

函数	描述
canvas 部件设置函数	
lv_canvas_create()	创建一个画布对象
lv_canvas_set_buffer()	为画布设置缓冲区
lv_canvas_set_px_color()	设置画布上一个像素的颜色
lv_canvas_set_px()	设置画布上一个像素的颜色(已弃用)
lv_canvas_set_palette()	使用索引格式设置画布的调色板颜色
lv_canvas_set_px_opa()	设置画布上一个像素的不透明度
canvas 部件获取函数	
lv_canvas_get_px()	获取画布上像素的颜色
lv_canvas_get_img()	获取画布的图像作为指向 lv_img_dsc_t 变量的指针
canvas 部件绘画函数	
lv_canvas_copy_buf()	将缓冲区复制到画布
lv_canvas_transform()	转换并成像并将结果存储在画布上
lv_canvas_blur_hor()	在画布上应用水平模糊
lv_canvas_blur_ver()	在画布上应用垂直模糊
lv_canvas_fill_bg()	用颜色填充画布
lv_canvas_draw_rect()	在画布上画一个矩形
lv_canvas_draw_text()	在画布上绘制文本
lv_canvas_draw_img()	在画布上绘制图像
lv_canvas_draw_line()	在画布上画一条线
lv_canvas_draw_polygon()	在画布上绘制多边形
lv_canvas_draw_arc()	在画布上画圆弧

表 14.3.1 canvas 部件函数描述

接下来我们来讲解 LVGL 画布部件常用的函数有哪些：

1. lv_ canvas _create 函数

该函数的作用是创建一个画布对象，其函数原型如下所示：

```
lv_obj_t * lv_canvas_create(lv_obj_t * parent)
```

该函数的形参描述如表 14.3.2 所示：

参数	描述
parent	指向对象的指针, 它将是新画布的父对象

表 14.3.2 函数 lv_canvas_create()形参描述

返回值：指向创建的画布的指针。

2. lv_canvas_set_buffer ()函数

该函数的作用是为画布设置缓冲区，其函数原型如下所示：

```
void lv_canvas_set_buffer( lv_obj_t * canvas,void * buf,
                           lv_coord_t w,lv_coord_t h,lv_img_cf_t cf )
```

该函数的形参描述如表 14.3.3 所示：

参数	描述
canvas	指向对象的指针,
buf	画布内容所在的缓冲区。所需的大小为(lv_img_color_format_get_px_size(cf)* w)/ 8 * h) 可以使用它分配 lv_mem_alloc()或可以是静态分配的数组(例如，静态 lv_color_t buf [100 * 50])，

	也可以是 RAM 中的地址或外部地址 SRAM
w	画布的宽度
h	画布的高度
cf	颜色格式。LV_IMG_CF_...

表 14.3.3 函数 lv_canvas_set_buffer()形参描述

返回值：无。

3. lv_canvas_transform 函数

该函数的作用是转换并成像并将结果存储在画布上，其函数原型如下所示：

```
void lv_canvas_transform(lv_obj_t * canvas,lv_img_dsc_t * img,
                         int16_t angle,uint16_t zoom,
                         lv_coord_t offset_x,lv_coord_t offset_y,
                         int32_t ivot_x,int32_t ivot_y,bool antialias )
```

该函数的形参描述如表 14.3.4 所示：

参数	描述
canvas	指向对象的指针,
img	指向要转换的图像描述符的指针。也可以是其他画布的图像描述符(lv_canvas_get_img())。
angle	旋转角度(0..3600),0.1 度分辨率
zoom	变焦倍数(256 无变焦)
offset_x	偏移 X 以告知将结果数据放在目标画布上的位置
offset_y	偏移 Y 以告知将结果数据放在目标画布上的位置
ivot_x	旋转 X 轴。相对于源画布设置为围绕中心旋转 source width / 2
ivot_y	旋转的枢轴 Y。相对于源画布设置为围绕中心旋转 source height / 2
antialias	在转换过程中应用抗锯齿。看起来更好,但速度较慢。

表 14.3.4 函数 lv_canvas_get_img()形参描述

返回值：无。

4. lv_canvas_blur_hor 函数

该函数的作用是在画布上应用水平模糊，其函数原型如下所示：

```
void lv_canvas_blur_hor(lv_obj_t *canvas,constlv_area_t * area,uint16_t r )
```

该函数的形参描述如表 14.3.5 所示：

参数	描述
canvas	指向对象的指针,
area	要模糊的区域。如果 NULL 整个画布会模糊
r	模糊半径

表 14.3.5 函数 lv_canvas_blur_hor()形参描述

返回值：无。

5. lv_canvas_blur_ver 函数

该函数的作用是在画布上应用垂直模糊，其函数原型如下所示：

```
void lv_canvas_blur_ver (lv_obj_t *canvas,constlv_area_t * area,uint16_t r )
```

该函数的形参描述如表 14.3.6 所示：

参数	描述
canvas	指向对象的指针,
area	要模糊的区域。如果 NULL 整个画布会模糊
r	模糊半径

表 14.3.6 函数 lv_canvas_blur_ver()形参描述

返回值：无。

6. lv_canvas_fill_bg 函数

该函数的作用是用颜色填充画布，其函数原型如下所示：

```
void lv_canvas_fill_bg(lv_obj_t *canvas,lv_color_t color,lv_opa_t opa )
```

该函数的形参描述如表 14.3.7 所示：

参数	描述
canvas	指向对象的指针,
color	背景色
opa	所需的不透明度

表 14.3.7 函数 lv_canvas_fill_bg()形参描述

返回值：无。

7. lv_canvas_draw_rect 函数

该函数的作用是在画布上画一个矩形，其函数原型如下所示：

```
void lv_canvas_draw_rect(lv_obj_t * canvas,lv_coord_t x,
                         lv_coord_t y,lv_coord_t w,
                         lv_coord_t h,
                         const lv_draw_rect_dsc_t * rect_dsc )
```

该函数的形参描述如表 14.3.8 所示：

参数	描述
canvas	指向对象的指针,
x	矩形的左坐标
y	矩形的顶部坐标
w	矩形的宽度
h	矩形的高度
rect_dsc	矩形的描述符

表 14.3.8 函数 lv_canvas_draw_rect()形参描述

返回值：无。

8. lv_canvas_draw_text 函数

该函数的作用是在画布上绘制文本，其函数原型如下所示：

```
void lv_canvas_draw_text(lv_obj_t *canvas,lv_coord_t x,
                         lv_coord_t y,
                         lv_coord_t max_w,
                         lv_draw_label_dsc_t * label_draw_dsc,
                         const char * txt,
                         lv_label_align_t align )
```

该函数的形参描述如表 14.3.9 所示：

参数	描述
canvas	指向对象的指针,
x	文本的左坐标
y	文本的顶部坐标
w	文字的最大宽度
label_draw_dsc	指向有效标签描述符的指针 lv_draw_label_dsc_t
txt	要显示的文字
align	对齐文字(LV_LABEL_ALIGN_LEFT/RIGHT/CENTER)

表 14.3.9 函数 lv_canvas_draw_text()形参描述

返回值：无。

9. lv_canvas_draw_img 函数

该函数的作用是在画布上绘制图像，其函数原型如下所示：

```
void lv_canvas_draw_img( lv_obj_t * canvas,lv_coord_t x,
                         lv_coord_t y,const void * src,
                         const lv_draw_img_dsc_t * img_draw_dsc )
```

该函数的形参描述如表 14.3.10 所示：

参数	描述
canvas	指向对象的指针,
x	图像的左坐标
y	图像的右坐标
src	图像源，可以是 lv_img_dsc_t 变量的指针或图像的路径。
img_draw_dsc	指向有效标签描述符的指针 lv_draw_img_dsc_t

表 14.3.10 函数 lv_canvas_draw_img()形参描述

返回值：无。

代码实例：

画布部件绘画图片如以下步骤所示：

1. 定义图片描述符。
2. 初始化图片描述符。
3. 设置图片的属性。
4. 调用函数 lv_canvas_draw_img 绘画图片。

```
lv_draw_img_dsc_t image_dsc; /* 第一步：定义图片描述符 */
lv_obj_t* canvas;

void lv_mainstart(void)
{
    /* 初始化图片描述符 */
    lv_draw_img_dsc_init(&image_dsc);
    /* 第三步：设置描述符属性 */
    image_dsc.angle = 0;                /* 角度 */
    image_dsc.zoom = 128;               /* 缩放 */
    image_dsc.opa = LV_OPA_90;          /* 设置透明度 */
    image_dsc.recolor = LV_COLOR_RED;   /* 设置重新着色 */
    static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR
                           (CANVAS_WIDTH, CANVAS_HEIGHT)];

    canvas = lv_canvas_create(lv_scr_act());
    lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH,
                         CANVAS_HEIGHT, LV_IMG_CF_TRUE_COLOR);
    lv_obj_align(canvas, LV_ALIGN_CENTER, 0, 0);
    lv_canvas_fill_bg(canvas, LV_COLOR_SILVER, LV_OPA_COVER);
   /* 第四步：绘画图片 */
    lv_canvas_draw_img(canvas, 0, 0, "S:./lv_fatfs/atk02.bin", &image_dsc);
}
```

注意：lv_canvas_draw_img 函数的第二、第三的形参是坐标，它是以画布的左上角为原点，然后第四个形参是图片的路径或者图片的数据。

10. lv_canvas_draw_line 函数

该函数的作用是在画布上画一条线，其函数原型如下所示：

```
void lv_canvas_draw_line(lv_obj_t *canvas,const lv_point_t points[],
                         uint32_t point_cnt,
                         const lv_draw_line_dsc_t * line_draw_dsc )
```

该函数的形参描述如表 14.3.11 所示：

参数	描述
canvas	指向对象的指针
points[]	线的点
point_cnt	点数
line_draw_dsc	指向初始化 lv_draw_line_dsc_t 变量的指针

表 14.3.11 函数 lv_canvas_draw_line()形参描述

返回值：无。

11. lv_canvas_draw_polygon 函数

该函数的作用是在画布上绘制多边形，其函数原型如下所示：

```
void lv_canvas_draw_polygon(lv_obj_t *canvas,const lv_point_t points [],
                            uint32_t point_cnt,
                            const lv_draw_rect_dsc_t * poly_draw_dsc )
```

该函数的形参描述如表 14.3.12 所示：

参数	描述
canvas	指向对象的指针
points[]	线的点
point_cnt	点数
poly_draw_dsc	指向初始化 lv_draw_line_dsc_t 变量的指针

表 14.3.12 函数 lv_canvas_draw_polygon()形参描述

返回值：无。

12. lv_canvas_draw_arc 函数

该函数的作用是在画布上画圆弧，其函数原型如下所示：

```
void lv_canvas_draw_arc( lv_obj_t * canvas,lv_coord_t x,lv_coord_t y,
                         lv_coord_t r,int32_t start_angle,
                         int32_t end_angle,
                         const lv_draw_line_dsc_t * arc_draw_dsc )
```

该函数的形参描述如表 14.3.13 所示：

参数	描述
canvas	指向对象的指针
x	弧的起点 x
y	弧度的起点 y
r	圆弧半径
start_angle	起始角度(度)
end_angle	结束角度(度)
arc_draw_dsc	指向初始化 lv_draw_line_dsc_t 变量的指针

表 14.3.13 函数 lv_canvas_draw_arc()形参描述

返回值：无。

14.4 画布部件的实验

14.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数运行 LVGL 官方的画布程序，该画布部件先被创建，然后在它主体中绘画矩形，其次绘画文本，最后旋转矩形。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 14 canvas(画布)》。注意：DMF407、MiniSTM32、战舰、探索者以及精英不支持本实验。

14.4.2 软件设计

14.4.2.1 程序流程图

本实验的程序流程图，如下图 14.4.2.1 所示：

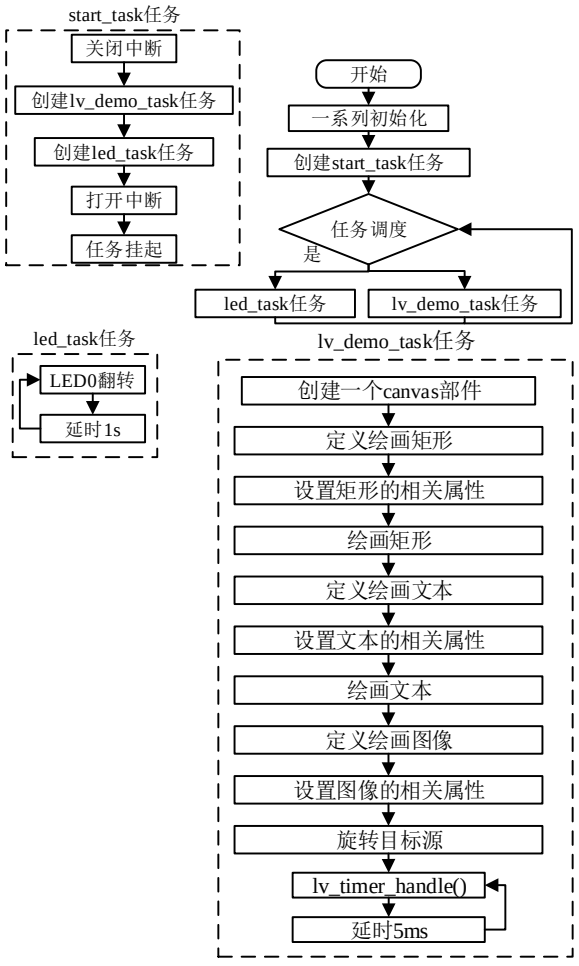


图 14.4.2.1 canvas 部件实验流程图

14.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_canvas(void)
{
    /* 定义画布缓冲区 */
    static lv_color_t canvas_buf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(canvas_width(),
                                 canvas_height())];
    /* 定义并初始化画布 */
    lv_obj_t* canvas = lv_canvas_create(lv_scr_act());
    /* 设置画布缓冲区 */
    lv_canvas_set_buffer(canvas, canvas_buf, canvas_width(),
                         canvas_height(), LV_IMG_CF_TRUE_COLOR);
    /* 设置画布位置 */
    lv_obj_center(canvas);
    /* 设置画布背景颜色 */
    lv_canvas_fill_bg(canvas, lv_palette_lighten(LV_PALETTE_GREY, 3),
                      LV_OPA_COVER);
    /* 定义绘画矩形 */
    lv_draw_rect_dsc_t rect_dsc;
    /* 初始化绘画矩形 */
    lv_draw_rect_dsc_init(&rect_dsc);
    /* 设置圆角 */
    rect_dsc.radius = 10;
    /* 设置透明度 */
    rect_dsc.bg_opa = LV_OPA_COVER;
    /* 设置颜色渐变方向 */
    rect_dsc.bg_grad.dir = LV_GRAD_DIR_HOR;
    /* 设置开始颜色 */
    rect_dsc.bg_grad.stops[0].color = lv_palette_main(LV_PALETTE_RED);
    /* 设置结束颜色 */
    rect_dsc.bg_grad.stops[1].color = lv_palette_main(LV_PALETTE_BLUE);
    /* 设置边缘宽度 */
    rect_dsc.border_width = 2;
    /* 设置边缘透明度 */
    rect_dsc.border_opa = LV_OPA_90;
    /* 设置边缘颜色 */
    rect_dsc.border_color = lv_color_white();
```

```
/* 在画布上绘制矩形 */
lv_canvas_draw_rect(canvas,
                    (canvas_width() - img_width()) / 2,
                    (canvas_height() - img_height()) / 2,
                    img_width(),
                    img_height(),
                    &rect_dsc);
/* 定义绘制标签 */
lv_draw_label_dsc_t label_dsc;
/* 初始化绘制标签 */
lv_draw_label_dsc_init(&label_dsc);
/* 设置标签颜色 */
label_dsc.color = lv_color_black();
/* 在画布上绘制标签 */
lv_canvas_draw_text(canvas,
                    canvas_width() / 8,
                    canvas_height() / 8,
                    100,
                    &label_dsc,
                    "Some text on text canvas");
/* 定义图片缓冲区 */
static lv_color_t canvas_buf_temp[LV_CANVAS_BUF_SIZE_TRUE_COLOR(
                                  canvas_width(), canvas_height())];
/* 复制旧缓冲区 */
lv_memcpy(canvas_buf_temp, canvas_buf, sizeof(canvas_buf_temp));
/* 定义图片 */
lv_img_dsc_t img;
/* 设置图片数据 */
img.data = (uint8_t*)canvas_buf_temp;
/* 图片颜色格式 */
img.header.cf = LV_IMG_CF_TRUE_COLOR;
/* 宽度 */
img.header.w = canvas_width();
/* 高度 */
img.header.h = canvas_height();
/* 旋转画布 */
lv_canvas_transform(canvas,
                    &img, 30,
                    LV_IMG_ZOOM_NONE,
                    0, 0,
                    canvas_width() / 2,
                    canvas_height() / 2,
                    true);
```

```
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_canvas();
}
```

此函数主要调用 lv_example_canvas 函数创建 canvas 画布部件，在画布部件重绘画矩形和文本并且把矩形旋转 45°。

14.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 14.4.3.1 所示：

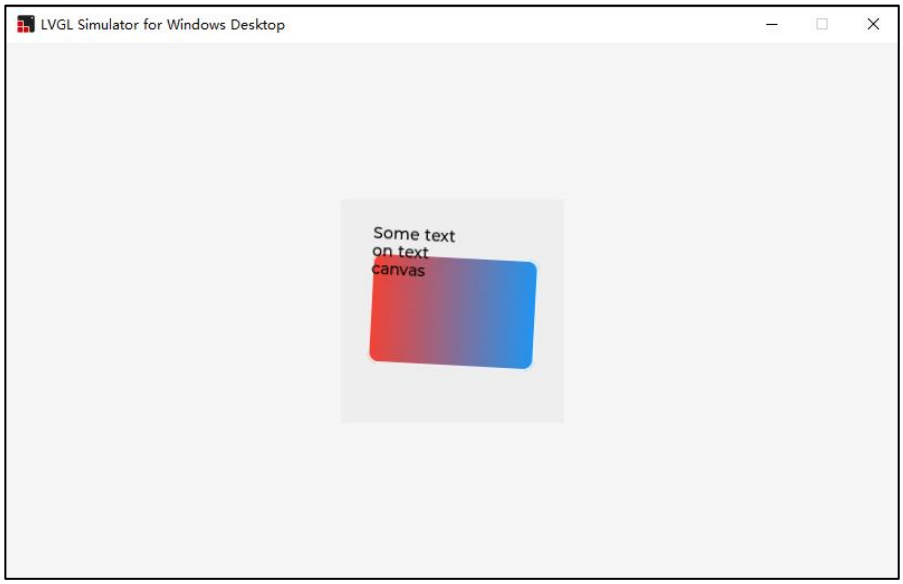


图 14.4.3.1 canvas 画布实验

第十五章 复选框部件(lv_checkbox)

复选框部件类似我们平常填写表格时，用于非选和选择的，该部件很像按键 btn 的使能了 toggle，当按下时，处于选择状态，当再一次按下切换成非选择状态，LVGL 中 lv_cb 部件是由 btn 部件基础上构建而来，主要是 btn 背景重绘并使能了 toggle 状态。

本章分为如下几部分：

15.1 复选框部件的组成

15.2 复选框部件的相关知识

15.3 复选框部件 API 函数

15.4 复选框部件实验

15.1 复选框部件的组成

LVGL 的 checkbox 对象复选框是由 3 部分组成的，第一部分是最外层的背景，第二部分是内部左侧的小方块，这个小方块其实就是一个 lv_btn 按钮，所以这个小方块也具有 5 种状态和相应的 5 个样式，第三部分是内部右侧的文本，该右则的文本的控件是 label，可以使用函数 lv_checkbox_set_text 设置该文本，如图 15.1.1 所示：

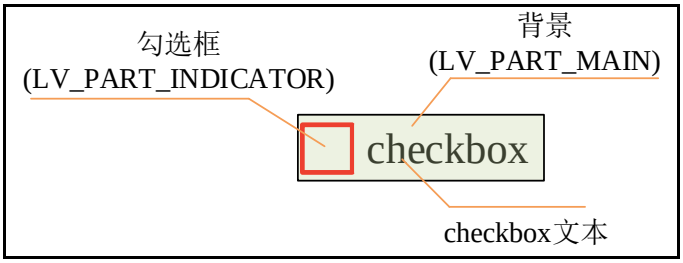


图 15.1.1 lv_cb 复选框的组成

LV_PART_MAI：复选框的背景，它使用了文本和所有典型的背景样式属性。

LV_PART_INDICATOR：“勾选框”是一个使用了所有典型背景样式属性的正方形。 默认情况下，它的大小等于主体部分字体的高度。

注意：上述的组成部分可以设置相关样式，样式的设置，请参考第六章 6.3 小节。

15.2 复选框部件的相关知识

15.2.1 设置复选框部件文本

Checkbox 部件的文本可以通过 lv_checkbox_set_text(cb， "New text")函数进行修改。该函数会动态分配文本，如果使用静态文本，请使用 lv_checkbox_set_text_static(cb, txt)函数设置。下面我们使用一个简单实例来讲解本小节的知识如下源码所示：

```
void lv_mainstart()
{
    lv_obj_t* cb1 = lv_checkbox_create(lv_scr_act());
    lv_checkbox_set_text(cb1, "CheckBox1");/* 动态设置复选框的文本 */
    lv_obj_align(cb1, LV_ALIGN_CENTER, 0,0);
```

```
    lv_obj_t* cb2 = lv_checkbox_create(lv_scr_act());
    lv_checkbox_set_text_static(cb2, "CheckBox2");/* 静态设置复选框的文本 */
    lv_obj_align_to(cb2, cb1,LV_ALIGN_LEFT_MID, 0, 50);
}
```

上述源码表示：我们调用函数 lv_checkbox_set_text 和函数 lv_checkbox_set_text_static 动态和静态设置复选框的文本，如图 15.2.1.1 所示：

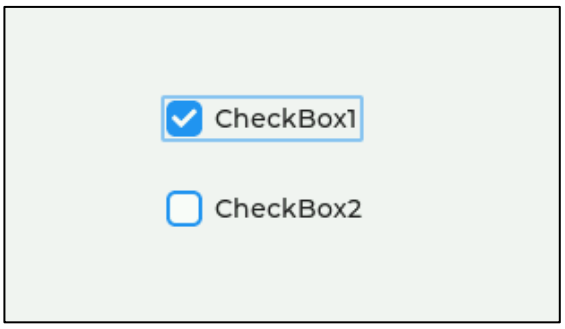


图 15.2.1.1 设置复选框的文本

15.2.2 复选框部件状态设置

复选框的状态一般具有三种，它们分别为选中、未选中以及禁用，这三种状态由以下函数手动设置。

```
lv_obj_add_state(cb, LV_STATE_CHECKED);                    /* 选中复选框 */
lv_obj_clear_state(cb, LV_STATE_CHECKED);                  /* 清除复选框 */
lv_obj_add_state(cb, LV_STATE_CHECKED | LV_STATE_DISABLED);  /* 禁用复选框 */
```

15.2.3 复选框事件

1. 输入设备事件

LV_EVENT_VALUE_CHANGED：当复选框被切换时发送。

2. 绘画事件

LV_CHECKBOX_DRAW_PART_BOX：复选框的复选框。

15.3 复选框部件 API 函数

LVGL 提供给我们的 checkbox 控件函数不多，如表 15.3.1 所示：

函数	描述
lv_checkbox_create()	创建一个复选框对象
lv_checkbox_set_text()	设置复选框的文本。txt 将被复制,并且在此函数返回后可能会被释放。
lv_checkbox_set_text_static()	设置复选框的文本。txt 在此复选框有效期内不得释放
checkbox 控件获取函数	
lv_checkbox_get_text()	获取复选框的文本

表 15.3.1 checkbox 控件函数描述

接下来我们来讲解 LVGL 复选框部件常用的函数有哪些：

1. lv_checkbox_create 函数

该函数的作用是创建一个复选框对象，其函数原型如下所示：

```
lv_obj_t * lv_checkbox_create(lv_obj_t * parent)
```

该函数的形参描述如表 15.3.2 所示：

参数	描述
parent	指向对象的指针,它将是复选框的父对象

表 15.3.2 函数 lv_checkbox_create()形参描述

返回值：指向已创建复选框的指针。

2. lv_checkbox_set_text 函数

该函数的作用是动态设置复选框的文本，其函数原型如下所示：

```
void lv_checkbox_set_text(lv_obj_t * obj,const char* txt)
```

该函数的形参描述如表 15.3.3 所示：

参数	描述
obj	指向复选框的指针
txt	复选框的文本。使用 NULL 刷新当前文本

表 15.3.3 函数 lv_checkbox_set_text()形参描述

返回值：无。

3. lv_checkbox_set_text_static 函数

该函数的作用是静态设置复选框的文本，其函数原型如下所示：

```
void lv_checkbox_set_text_static(lv_obj_t * obj,const char* txt)
```

该函数的形参描述如表 15.3.4 所示：

参数	描述
obj	指向复选框的指针
txt	复选框的文本。使用 NULL 刷新当前文本

表 15.3.4 函数 lv_checkbox_set_text_static()形参描述

返回值：无。

4. lv_checkbox_get_text 函数

该函数的作用是获取复选框的文本，其函数原型如下所示：

```
const char* lv_checkbox_get_text(const lv_obj_t * obj)
```

该函数的形参描述如表 15.3.5 所示：

参数	描述
obj	指向复选框的指针

表 15.3.5 函数 lv_checkbox_get_text()形参描述

返回值：指向复选框文本的指针。

15.4 复选框部件实验

15.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建了一个 checkbox 部件以及多个 btn 按键，这个 checkbox 部件的相关样式属性由多个 btn 按键来控制，例如修改 checkbox 部件大小，选中/禁用等属性。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。该实验的实验工程，请参考《LVGL 例程 15 checkbox(复选框)》。

15.4.2 软件设计

15.4.2.1 程序流程图

本实验的程序流程图，如下图 15.4.2.1 所示：

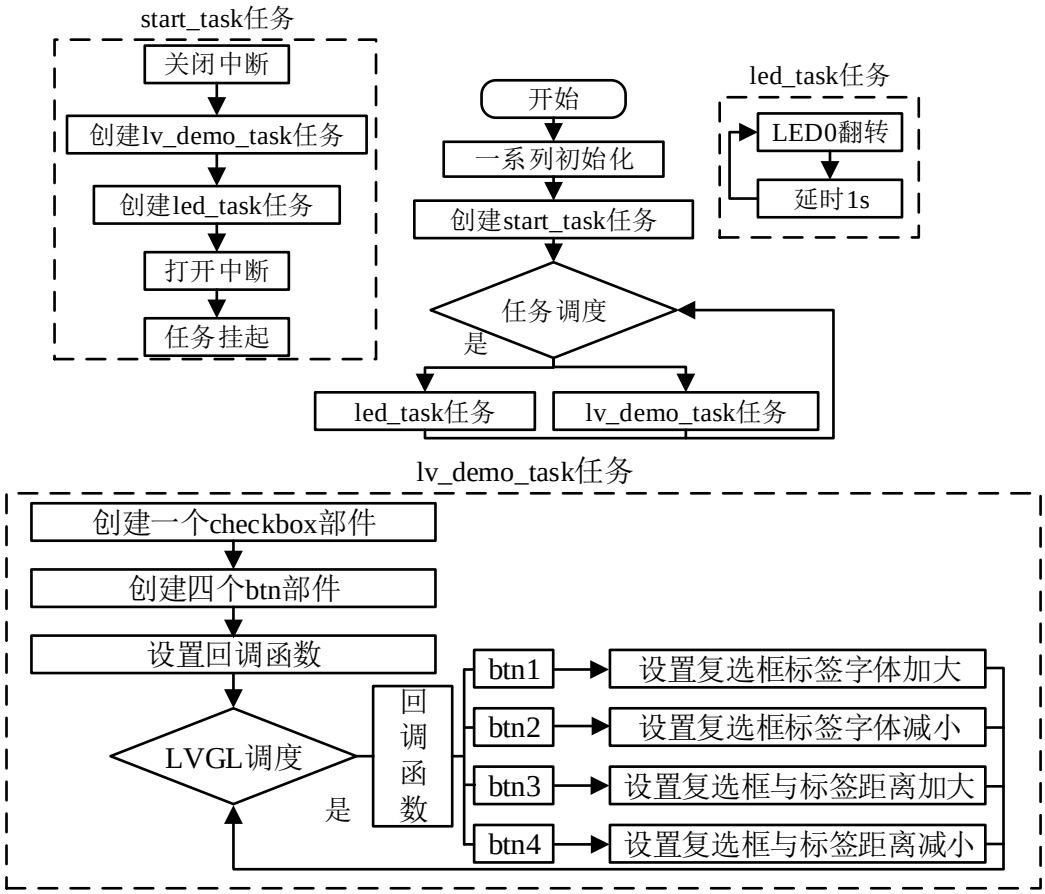


图 15.4.2.1 checkbox 部件实验流程图

15.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

(1) lv_mainstart 函数

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_checkbox(void)
{
    /* 定义并创建复选框 */
```

```
checkbox = lv_checkbox_create(lv_scr_act());
/* 设置位置 */
lv_obj_align(checkbox, LV_ALIGN_CENTER, 0, -scr_act_height() / 4);
/* 添加标签文本 */
lv_checkbox_set_text(checkbox, "CheckBox");
/* 设置标签文本字体 */
lv_obj_set_style_text_font(checkbox, &lv_font_montserrat_8, LV_PART_MAIN);
/* 设置标签文本与复选框的距离 */
lv_obj_set_style_pad_column(checkbox, 5, LV_PART_MAIN);/* 设置复选框状态 */
lv_obj_add_state(checkbox, LV_STATE_DEFAULT);

char buf[20];
lv_obj_t* btn;
lv_obj_t* label;

/* 按钮：设置复选框标签字体加大 *//* 定义并创建按钮 */
btn = lv_btn_create(lv_scr_act());
/* 设置按钮宽度 */
lv_obj_set_width(btn, scr_act_width() / 3 - 8);
/* 设置按钮的高度 */
lv_obj_set_height(btn, scr_act_height() / 4 - 6);
/* 设置按钮位置 */
lv_obj_align(btn, LV_ALIGN_BOTTOM_LEFT, 6, -scr_act_height() / 4 - 6);
/* 定义并创建标签 */
label = lv_label_create(btn);
/* 设置标签位置 */
lv_obj_center(label);
lv_snprintf(buf, sizeof(buf), "Font +");
/* 设置标签文本 */
lv_label_set_text(label, buf);
/* 设置标签文本对齐方式 */
lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
/* 添加按钮事件回调 */
lv_obj_add_event_cb(btn, event_btn_cb, LV_EVENT_CLICKED, label);

/* 按钮：设置复选框标签字体减小 */
btn = lv_btn_create(lv_scr_act());/* 定义并创建按钮 */
lv_obj_set_width(btn, scr_act_width() / 3 - 8);        /* 设置按钮宽度 */
lv_obj_set_height(btn, scr_act_height() / 4 - 6);      /* 设置按钮高度 */
lv_obj_align(btn, LV_ALIGN_BOTTOM_LEFT, 6, -6);        /* 设置按钮位置 */
label = lv_label_create(btn);                          /* 定义并创建标签 */
lv_obj_center(label);                                  /* 设置标签位置 */
lv_snprintf(buf, sizeof(buf), "Font -");
```

```
    lv_label_set_text(label, buf);                      /* 设置标签文本 */
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 添加按钮事件回调 */
    lv_obj_add_event_cb(btn, event_btn_cb, LV_EVENT_CLICKED, label);

    /* 按钮：设置复选框与标签距离加大 *//* 定义并创建按钮 */
    btn = lv_btn_create(lv_scr_act());
    lv_obj_set_width(btn, scr_act_width() / 3 - 8);    /* 设置按钮宽度 */
    lv_obj_set_height(btn, scr_act_height() / 4 - 6);  /* 设置按钮的高度 */
    /* 设置按钮位置 */
    lv_obj_align(btn, LV_ALIGN_BOTTOM_MID, 0, -scr_act_height() / 4 - 6);
    /* 定义并创建标签 */
    label = lv_label_create(btn);
    lv_obj_center(label);               /* 设置标签位置 */
    lv_snprintf(buf, sizeof(buf), "Label <>");
    lv_label_set_text(label, buf); /* 设置标签文本 */
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 添加按钮事件回调 */
    lv_obj_add_event_cb(btn, event_btn_cb, LV_EVENT_CLICKED, label);
    /* 按钮：设置复选框与标签距离减小 */
    btn = lv_btn_create(lv_scr_act());                  /* 定义并创建按钮 */
    lv_obj_set_width(btn, scr_act_width() / 3 - 8);    /* 设置按钮宽度 */
    lv_obj_set_height(btn, scr_act_height() / 4 - 6);  /* 设置按钮高度 */
    lv_obj_align(btn, LV_ALIGN_BOTTOM_MID, 0, -6);     /* 设置按钮位置 */
    label = lv_label_create(btn);                       /* 定义并创建标签 */
    lv_obj_center(label);               /* 设置标签位置 */
    lv_snprintf(buf, sizeof(buf), "Label ><");
    lv_label_set_text(label, buf);      /* 设置标签文本 */
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 添加按钮事件回调 */
    lv_obj_add_event_cb(btn, event_btn_cb, LV_EVENT_CLICKED, label);
    /* 按钮：设置复选框状态 */
    btn = lv_btn_create(lv_scr_act());/* 定义并创建按钮 */
    lv_obj_set_width(btn, scr_act_width() / 3 - 8);    /* 设置按钮宽度 */
    lv_obj_set_height(btn, scr_act_height() / 2 - 6); /* 设置按钮高度 */
    lv_obj_align(btn, LV_ALIGN_BOTTOM_RIGHT, -6, -6); /* 设置按钮位置 */
    label = lv_label_create(btn); /* 定义并创建标签 */
    lv_obj_center(label); /* 设置标签位置 */
    lv_snprintf(buf, sizeof(buf), "State\nChange");
    lv_label_set_text(label, buf); /* 设置标签文本 */
```

```
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 添加按钮事件回调 */
    lv_obj_add_event_cb(btn, event_btn_cb, LV_EVENT_CLICKED, label);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_checkbox();
}
```

此函数主要调用 lv_example_checkbox 函数创建 checkbox 部件、label 部件和四个 btn 部件并且设置这些部件的相关属性，最后调用 lv_obj_add_event_cb 函数除了 label 部件外的所有部件添加到同一个回调函数 event_btn_cb。注意：该回调函数的形参不为空，而是指向 label 部件控制块。

(2) event_btn_cb 回调函数

```
/**
 * @brief  按钮事件回调
 * @param  e:触发的事件
 * @return 无
 */
static void event_btn_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* label = (lv_obj_t*)e->user_data;

    char* buf;
    /* 获取 label 部件文本 */
    buf = lv_label_get_text(label);
    if (LV_EVENT_CLICKED == code)
    {
    /* 根据文本所对应的 btn 部件来设置 checkbox 相关属性 */
        if (0 == strcmp(buf, "Font +")) {
            const lv_font_t* old_font = lv_obj_get_style_text_font(checkbox,
                                                                   LV_PART_MAIN);
            const lv_font_t* new_font = NULL;
            if (&lv_font_montserrat_8 == old_font)
                new_font = &lv_font_montserrat_10;
            if (&lv_font_montserrat_10 == old_font)
```

```
            new_font = &lv_font_montserrat_12;
        if (&lv_font_montserrat_12 == old_font)
            new_font = &lv_font_montserrat_14;
        if (&lv_font_montserrat_14 == old_font)
            new_font = &lv_font_montserrat_16;
        if (&lv_font_montserrat_16 == old_font)
            new_font = &lv_font_montserrat_18;
        if (&lv_font_montserrat_18 == old_font)
            new_font = &lv_font_montserrat_20;
        if (&lv_font_montserrat_20 == old_font)
            new_font = &lv_font_montserrat_20;
        /* 根据文本所对应的 btn 部件来设置 checkbox 字体属性 */
        lv_obj_set_style_text_font(checkbox, new_font, LV_PART_MAIN);
    }
    if (0 == strcmp(buf, "Font -")) {
        const lv_font_t* old_font = lv_obj_get_style_text_font(checkbox,
                                                               LV_PART_MAIN);
        const lv_font_t* new_font = NULL;
        if (&lv_font_montserrat_8 == old_font)
            new_font = &lv_font_montserrat_8;
        if (&lv_font_montserrat_10 == old_font)
            new_font = &lv_font_montserrat_8;
        if (&lv_font_montserrat_12 == old_font)
            new_font = &lv_font_montserrat_10;
        if (&lv_font_montserrat_14 == old_font)
            new_font = &lv_font_montserrat_12;
        if (&lv_font_montserrat_16 == old_font)
            new_font = &lv_font_montserrat_14;
        if (&lv_font_montserrat_18 == old_font)
            new_font = &lv_font_montserrat_16;
        if (&lv_font_montserrat_20 == old_font)
            new_font = &lv_font_montserrat_18;
        /* 根据文本所对应的 btn 部件来设置 checkbox 字体属性 */
        lv_obj_set_style_text_font(checkbox, new_font, LV_PART_MAIN);
    }
    if (0 == strcmp(buf, "Label <>")) {
        lv_coord_t old_dis = lv_obj_get_style_pad_column(checkbox,
                                                         LV_PART_MAIN);
        lv_coord_t new_dis = 0;
        new_dis = old_dis += 5;
        if (new_dis >= 35)
            new_dis = 35;
        /* 根据文本所对应的 btn 部件来设置 checkbox 与 label 间距 */
```

```
            lv_obj_set_style_pad_column(checkbox, new_dis, LV_PART_MAIN);
        }
        if (0 == strcmp(buf, "Label ><")) {
            lv_coord_t old_dis = lv_obj_get_style_pad_column(checkbox,
                                                             LV_PART_MAIN);
            lv_coord_t new_dis = 0;
            new_dis = old_dis -= 5;
            if (new_dis <= 5)
                new_dis = 5;
            /* 根据文本所对应的 btn 部件来设置 checkbox 与 label 间距 */
            lv_obj_set_style_pad_column(checkbox, new_dis, LV_PART_MAIN);
        }
        if (0 == strcmp(buf, "State\nChange")) {
            static uint8_t state = 1;
            /* 根据文本所对应的 btn 部件来设置 checkbox 状态 */
            switch (state) {
            case(0):
                lv_obj_add_state(checkbox, LV_STATE_CHECKED);
                lv_obj_add_state(checkbox, LV_STATE_DISABLED);
                state = 1;
                break;
            case(1):
                lv_obj_add_state(checkbox, LV_STATE_CHECKED);
                lv_obj_clear_state(checkbox, LV_STATE_DISABLED);
                state = 2;
                break;
            case(2):
                lv_obj_clear_state(checkbox, LV_STATE_CHECKED);
                lv_obj_add_state(checkbox, LV_STATE_DISABLED);
                state = 3;
                break;
            case(3):
                lv_obj_clear_state(checkbox, LV_STATE_CHECKED);
                lv_obj_clear_state(checkbox, LV_STATE_DISABLED);
                state = 0;
                break;
            }
        }
    }
}
```

此回调函数主要根据 btn 部件的 label 文本来做触发源，当按下不同的 btn 部件来设置 checkbox 相关属性。

15.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 15.4.3.1 所示：

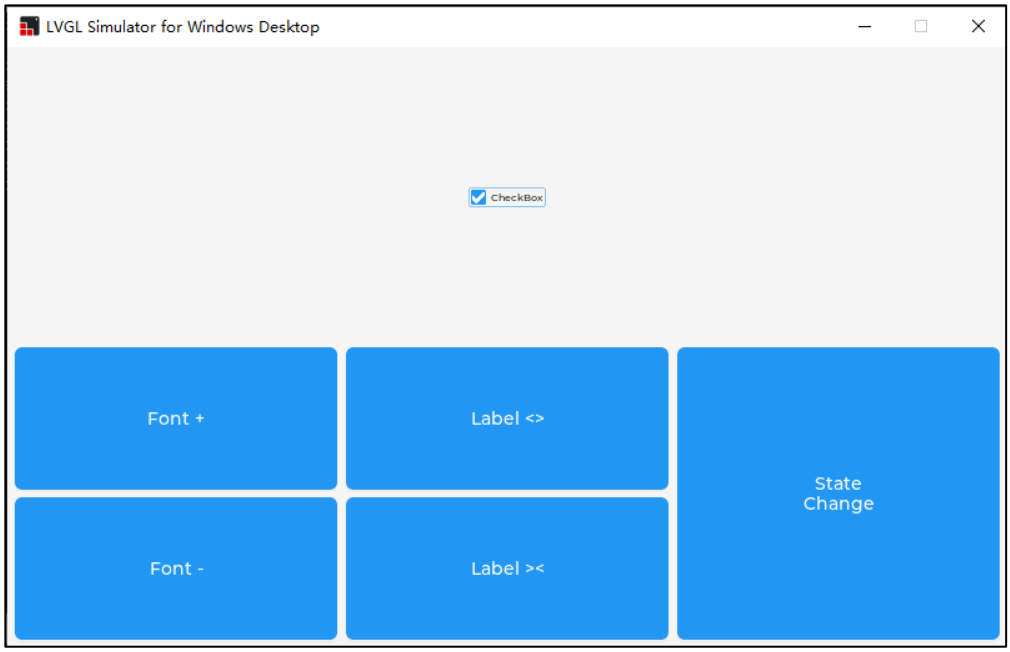


图 15.4.3.1 checkbox 复选框实验

第十六章 下拉列表部件(lv_dropdown)

lv_dropdown 是一个下拉列表部件，它允许用户从多个选项值中去任意选择一个值，它具有收缩和展开两个状态，默认情况下，它是处于收缩状态的，在收缩状态下，它会将当前选择的值给显示出来，然后用户可以从中选择一个想要的值，一旦选择好之后,下拉列表部件会自动从展开状态回到收缩状态。

本章分为如下几部分：

16.1 下拉列表部件的组成

16.2 下拉列表部件的相关知识

16.3 下拉列表部件的 API 函数

16.4 下拉列表部件的实验

16.1 下拉列表部件的组成

下拉列表部件是由“button”和“list”元素构建 (注意：这两者与按钮和列表小部件无关)，如图 16.1.1 所示：

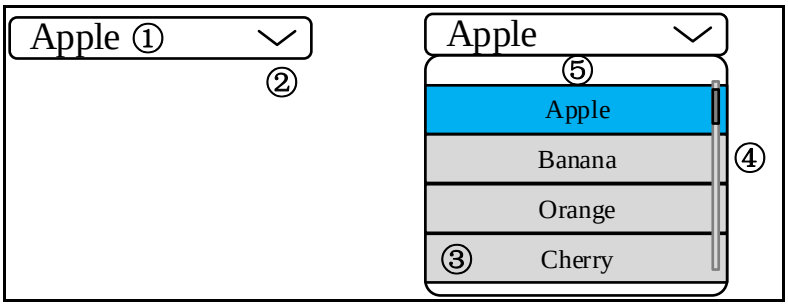


图 16.1.1 下拉列表组成部分

上图左边的下拉列表是收缩状态下的，而右边是展开状态下的下拉列表，这里笔者使用标签符号①~⑤描述，①②标签表示“button” 元素的基础部分，它们分别为 LV_PART_MAIN 和 LV_PART_INDICATOR 基础部分，而③~⑤标签表示“list” 元素的基础部分，它们分别为 LV_PART_MAIN、LV_PART_SCROLLBAR 和 LV_PART_SELECTED 基础部分，下面我们逐一介绍它们的功能，如下所示：

Button 元素：

① LV_PART_MAIN：按钮的背景(背景属性和文本属性)，它对应上图中的①。

② LV_PART_INDICATOR：指示器，它对应上图中的②。

List 元素：

① LV_PART_MAIN：列表本身(背景属性)，它对应上图中的⑤。

② LV_PART_SCROLLBAR：滚动条，它对应上图中的④。

③ LV_PART_SELECTED：当前按下、选中的选项，它对应上图中的③。

注意：上述的组成部分可以设置相关样式，样式的设置，请参考第六章 6.3 小节的内容。

16.2 下拉列表部件的相关知识

16.2.1 下拉列表部件添加/设置表项

前面我们知道下拉列表中的 list 元素可以添加多种表项，这些选项是由专门的函数添加的，下面我们来讲解一下 LVGL 下拉列表如何添加选项。下拉列表部件添加选项具有三种方式，如下所示：

第一种：调用 lv_dropdown_set_options 函数添加表项，该函数如下源码所示：

```
lv_dropdown_set_options(dropdown, options) /*添加下拉列表项 */
```

使用上述的函数把选项的字符串传递给下拉列表，这些选项字符串之间通过‘\n’ 进行分隔，如下源码所示：

```
"First \n Second \n Third"
```

上述源码的字符串将保存在下拉列表中，因此该函数可以添加多个选项且保存在本地变量中。

第二种：调用 lv_dropdown_add_option 函数添加表项，该函数如下源码所示：

```
lv_dropdown_add_option(dropdown, "New option", pos)
```

注意：每次调用只添加一个选项，pos 表示添加到那个位置，注意：0 表示最上面的位置，由此类推。

第三种：调用 lv_dropdown_set_static_options 函数添加选项，该函数如下源码所示：

```
lv_dropdown_set _options_static (dropdown, options)
```

上述函数是从静态(常量)字符串设置表项。在这种情况下，我们不能使用 lv_dropdown_add_option 函数添加选项，因为这个函数只能作用于动态添加选项的函数。注意：使用第三种方式添加选项主要为了节省内存的使用。

下面笔者以简单的实例分别地讲解这三种添加方式如下源码所示：

```
void lv_mainstart()
{
    /* 创建一个下拉列表 */
    lv_obj_t* lv_ddlist1 = lv_dropdown_create(lv_scr_act());
    /*添加下拉列表项 */
    lv_dropdown_set_options(lv_ddlist1, "a\nb\nc\nd");
    lv_obj_set_pos(lv_ddlist1, 100, 100);
    /* 默认显示哪一个下拉列表项 */
    lv_dropdown_set_selected(lv_ddlist1, 0);

    lv_obj_t* lv_ddlist2 = lv_dropdown_create(lv_scr_act());
    /*添加下拉列表项 */
    lv_dropdown_add_option(lv_ddlist2, "0", 0);
    lv_dropdown_add_option(lv_ddlist2, "1", 1);
    lv_dropdown_add_option(lv_ddlist2, "2", 2);
    lv_dropdown_add_option(lv_ddlist2, "3", 3);
    /* 默认显示哪一个下拉列表项 */
    lv_dropdown_set_selected(lv_ddlist2, 1);
    lv_obj_set_pos(lv_ddlist2, 300, 100);
```

```
    lv_obj_t* lv_ddlist3 = lv_dropdown_create(lv_scr_act());
    /*添加下拉列表项 */
    lv_dropdown_set _options_static (lv_ddlist3, "1");
    lv_obj_set_pos(lv_ddlist3, 500, 100);
}
```

上述源码可知，我们创建三个下拉列表并使用上述的三种方式添加表项，如图 16.2.1.1 所示：



图 16.2.1.1 不同的函数添加下拉列表项

16.2.2 下拉列表部件获取选择的选项

对于下拉列表的使用我相信大家并不陌生，在日常生活中会经常用到，我们知道当按下某个表项时可以触发一个动作，例如选择性别、年龄等操作，显然 LVGL 的下拉列表也有类似的功能，当用户按下某个表项时，系统根据表项的内容触发动作，这个动作是由用户来实现。对于这个触发源，LVGL 提供了两个函数获取，这两个函数如下所示：

```
lv_dropdown_get_selected(dropdown) /* 获取选中的选项 */
lv_dropdown_get_selected_str(dropdown, buf, buf_size)/* 选中的选项字符串保存到 buf */
```

第一个函数就是获取表项的索引，而第二个函数是获取表项的文本，这两个函数都可以作为触发源来使用。

16.2.3 下拉列表部件设置方向

默认情况下，当我们展开下拉列表时都是往下展开的，这样导致展开的表项可能遮住其他的部件，至此 LVGL 为下拉列表添加了展开方向设置，例如往下/左/右/上展开，具体展开方向是由以下函数设置。

```
lv_dropdown_set_dir(dropdown, LV_DIR_LEFT/RIGHT/UP/BOTTOM)/* 设置下拉方向 */
```

老规矩，我们使用一个简单实例来讲解本小节的知识，如下源码所示：

代码实例：

```
void lv_mainstart(void)
{
    /* 创建一个下拉列表 */
    lv_obj_t* lv_ddlist1 = lv_dropdown_create(lv_scr_act());
    /*添加下拉列表项 */
    lv_dropdown_set_options(lv_ddlist1, "a\nb\nc\nd");
    lv_obj_set_pos(lv_ddlist1, 100, 100);
    /* 默认显示哪一个下拉列表项 */
    lv_dropdown_set_selected(lv_ddlist1, 0);

    lv_obj_t* lv_ddlist2 = lv_dropdown_create(lv_scr_act());
```

```
    lv_dropdown_set_dir(lv_ddlist2,LV_DIR_LEFT);/* 设置下拉方向 */
    /*添加下拉列表项 */
    lv_dropdown_add_option(lv_ddlist2, "0", 0);
    lv_dropdown_add_option(lv_ddlist2, "1", 1);
    lv_dropdown_add_option(lv_ddlist2, "2", 2);
    lv_dropdown_add_option(lv_ddlist2, "3", 3);
    /* 默认显示哪一个下拉列表项 */
    lv_dropdown_set_selected(lv_ddlist2, 1);
    lv_obj_set_pos(lv_ddlist2, 300, 100);

    lv_obj_t* lv_ddlist3 = lv_dropdown_create(lv_scr_act());
    /*添加下拉列表项 */
    lv_dropdown_set_options_static(lv_ddlist3, "1");
    lv_obj_set_pos(lv_ddlist3, 500, 100);
}
```

上述函数可知：我们创建三个 dropdown 部件，前面我们已经讲解过，这里我们重点讲解 lv_dropdown_set_dir 函数，该函数会根据用户传入的形参而设置展开方向，例如 lv_ddlist2 下拉列表设置展开方向 LV_DIR_LEFT，如图 16.2.3.1 所示：

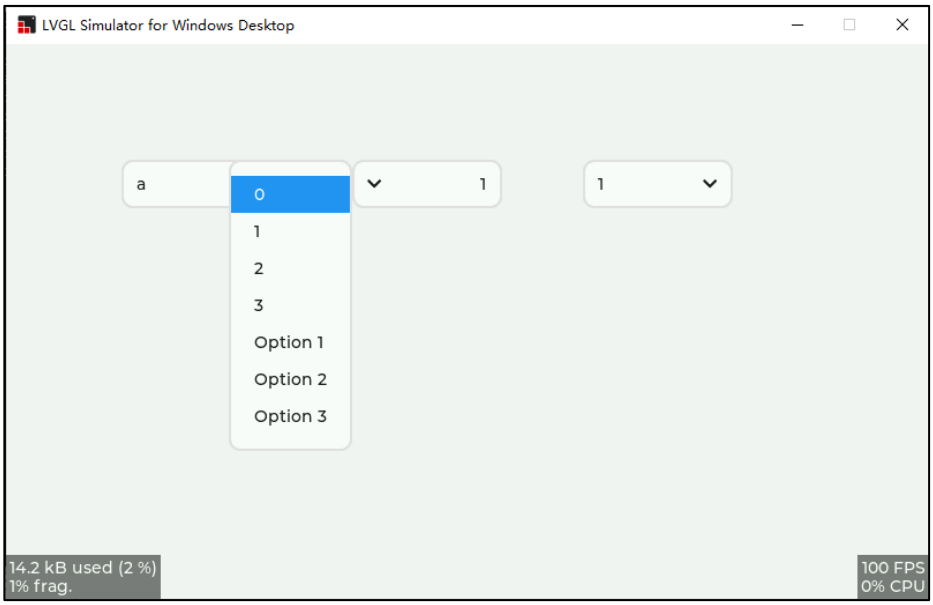


图 16.2.3.1 设置 dropdown 部件的方向

当我们按下中间的下拉列表时，该下拉列表的展开方向是从左边展开的。

16.2.4 下拉列表部件符号设置

从前面的内容来看，下拉列表创建时它所附带的图标，该图标的箭头是往下的，如果我们设置表项展开方向为往上方展开，那么该图标和展开方向就不应景了，至此 LVGL 为下拉列表添加了图标符号设置功能，这个功能需要用户调用以下函数设置：

```
lv_dropdown_set_symbol(dropdown, LV_SYMBOL_…) /* 设置符号 */
```

上述函数的第二个形参就是图标，我们在第八章也讲过，这里笔者不必讲解。当然默认情况的下拉列表的图标都是箭头形式，如图 16.2.4.1 所示：



图 16.2.4.1 设置下拉列表的符号

16.2.5 下拉列表部件设置显示项

默认情况下，当我们按下下拉列表的表项时，该表项的文本会显示在“button”元素的文本当中，这样我们可以看到上一次选择的表项内容。在 UI 设计时我们往往在下拉列表的顶部添加一个 label 标签来声明这个下拉列表的作用，这样无疑多使用了一个 label 部件，LVGL 为了节省内存，它提供了一个函数来设置“button”元素的文本，这个文本不会被按下的表项文本而改变，它放弃了显示上一次选项表项的内容功能，设置显示项方式如下所示：

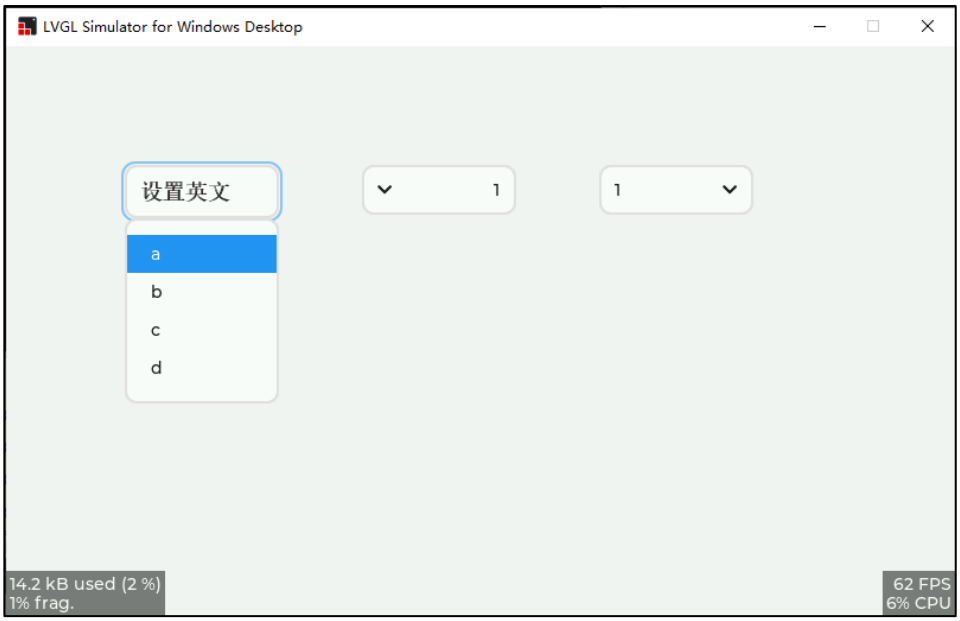
```
lv_dropdown_set_text(dropdown,"text") /* 设置文本 */
```

下面我们还是使用一个简单实例来讲解本小节的内容，如下源码所示：

```
LV_FONT_DECLARE(myFont14)

void lv_mainstart(void)
{
    /* 创建一个下拉列表 */
    lv_obj_t* lv_ddlist1 = lv_dropdown_create(lv_scr_act());
    lv_obj_set_style_text_font(lv_ddlist1, &myFont14, LV_STATE_DEFAULT);
    lv_dropdown_set_text(lv_ddlist1, "设置英文");
    /*添加下拉列表项 */
    lv_dropdown_set_options(lv_ddlist1, "a\nb\nc\nd");
    lv_obj_set_pos(lv_ddlist1, 100, 100);
    /* 默认显示哪一个下拉列表项 */
    lv_dropdown_set_selected(lv_ddlist1, 0);
    lv_obj_t* lv_ddlist2 = lv_dropdown_create(lv_scr_act());
    lv_dropdown_set_dir(lv_ddlist2,LV_DIR_LEFT);/* 设置下拉方向 */
    /*添加下拉列表项 */
    lv_dropdown_add_option(lv_ddlist2, "0", 0);
    lv_dropdown_add_option(lv_ddlist2, "1", 1);
    lv_dropdown_add_option(lv_ddlist2, "2", 2);
    lv_dropdown_add_option(lv_ddlist2, "3", 3);
    /* 默认显示哪一个下拉列表项 */
    lv_dropdown_set_selected(lv_ddlist2, 1);
    lv_obj_set_pos(lv_ddlist2, 300, 100);
    lv_obj_t* lv_ddlist3 = lv_dropdown_create(lv_scr_act());
    /*添加下拉列表项 */
    lv_dropdown_set_options_static(lv_ddlist3, "1");
    lv_obj_set_pos(lv_ddlist3, 500, 100);
}
```

此函数前面笔者已经讲解过，我们重点讲解 lv_dropdown_set_tex 函数的使用，这里我们把“button”元素的文本设置为“设置英文”字符串如图 16.2.5.1 所示：



16.2.5.1 设置 dropdown 部件显示文本

注意：设置了显示文本之后，点击下拉列表中的表项不会覆盖显示文本。

16.2.6 下拉列表部件手动开闭

按下下拉列表时，往往需要用户点击某一个表项才使下拉列表从展开状态转换成收缩状态，如果用户不选择表项，则下拉列表一直处于展开状态，这样不利于智能化，LVGL 为了解决这个问题，它提供一个函数让用户手动打开或者关闭下拉列表，该设置函数如下所示：

```
lv_dropdown_open/close(dropdown) /* 打开/关闭下拉列表 */
```

用户可以使用实体按键来控制 dropdown 部件的打开与关闭。

16.3 下拉列表部件的 API 函数

LVGL 提供给用户的 dropdown 部件函数不多，如表 16.3.1 所示：

函数	描述
lv_dropdown_create()	创建下拉列表对象
lv_dropdown_set_text()	设置下拉列表按钮的文本
lv_dropdown_set_options()	从字符串中设置下拉列表中的选项
lv_dropdown_set_options_static()	从静态字符串(全局、静态或动态分配)中设置下拉列表中的选项
lv_dropdown_add_option()	从字符串向下拉列表中添加选项
lv_dropdown_clear_options()	清除下拉列表中的所有选项。适用于静态和动态选项
lv_dropdown_set_selected()	设置所选选项
lv_dropdown_set_dir()	设置一个下拉列表框的方向
lv_dropdown_set_symbol()	设置下拉列表按钮上显示的箭头或其他符号。通常是向下的插入符号或箭头
lv_dropdown_set_selected_highlight()	设置列表中选择的选项是否应该高亮显示
lv_dropdown_get_list()	获取下拉列表，以允许样式化或其他修改
lv_dropdown_get_text()	获取下拉列表按钮的文本

lv_dropdown_get_options()	获取下拉列表的选项
lv_dropdown_get_selected()	获取所选选项的索引
lv_dropdown_get_option_cnt()	获取选项的总数
lv_dropdown_get_selected_str()	以字符串形式获取当前选定的选项
lv_dropdown_get_symbol()	获取下拉列表中的符号。通常是向下的插入符号或箭头
lv_dropdown_get_selected_highlight()	获取列表中所选选项是否应该高亮显示
lv_dropdown_get_dir()	获取下拉列表的方向
lv_dropdown_open()	打开下拉列表
lv_dropdown_close()	关闭下拉列表
lv_dropdown_is_open()	说明列表是否已打开

表 16.3.1 dropdown 部件函数描述

接下来我们来讲解 LVGL 下拉列表部件常用的函数有哪些：

1. lv_dropdown_create 函数

该函数的作用是创建一个下拉列表对象，其函数原型如下所示：

```
lv_obj_t * lv_ dropdown _create (lv_obj_t * parent,)
```

该函数的形参描述如表 16.3.2 所示：

参数	描述
parent	指向对象的指针,它将是新下拉列表的父对象

表 16.3.2 函数 lv_ dropdown _create()形参描述

返回值：指向创建的下拉列表的指针。

2. lv_dropdown_set_text 函数

该函数的作用是设置 ddlist 的文本，其函数原型如下所示：

```
void lv_dropdown_set_text (lv_obj_t * obj,const char* txt)
```

该函数的形参描述如表 16.3.3 所示：

参数	描述
obj	指向下拉列表对象的指针
txt	文本作为字符串(仅保存其指针)

表 16.3.3 函数 lv_dropdown_set_text()形参描述

返回值：无。

3. lv_dropdown_set_options 函数

该函数的作用是在字符串的下拉列表中设置选项(动态)，其函数原型如下所示：

```
void lv_dropdown_set_options(lv_obj_t * obj,const char * options)
```

该函数的形参描述如表 16.3.4 所示：

参数	描述
obj	指向下拉列表对象的指针
options	带有'的字符串'分开的选项

表 16.3.4 函数 lv_dropdown_set_options()形参描述

返回值：无。

4. lv_dropdown_set_options_static 函数

该函数的作用是在字符串的下拉列表中设置选项(静态)，其函数原型如下所示：

```
void lv_dropdown_set_options_static(lv_obj_t * obj,const char * options)
```

该函数的形参描述如表 16.3.5 所示：

参数	描述
obj	指向下拉列表对象的指针

options	带有'的字符串'分开的选项

表 16.3.5 函数 lv_dropdown_set_options_static()形参描述

返回值：无。

5. lv_dropdown_add_option 函数

该函数的作用是将选项从字符串添加到下拉列表中，其函数原型如下所示：

```
void lv_dropdown_add_option(lv_obj_t * obj,constchar * option,uint32_t pos)
```

该函数的形参描述如表 16.3.6 所示：

参数	描述
obj	指向下拉列表对象的指针
option	带有'的字符串'分开的选项
pos	插入位置

表 16.3.6 函数 lv_dropdown_add_option()形参描述

返回值：无。

7. lv_dropdown_set_selected 函数

该函数的作用是设置选择的选项，其函数原型如下所示：

```
void lv_dropdown_set_selected(lv_obj_t * obj,uint16_t sel_opt)
```

该函数的形参描述如表 16.3.7 所示：

参数	描述
obj	指向下拉列表对象的指针
sel_opt	所选选项的 ID(0 ...选项编号-1);

表 16.3.7 函数 lv_dropdown_set_selected()形参描述

返回值：无。

8. lv_dropdown_set_dir 函数

该函数的作用是设置下拉列表的方向，其函数原型如下所示：

```
oid lv_dropdown_set_dir(lv_obj_t * obj,lv_dropdown_dir_t dir)
```

该函数的形参描述如表 16.3.9 所示：

参数	描述
obj	指向下拉列表对象的指针
dir	LV_DIR_LEFT/RIGHT/TOP/BOTTOM

表 16.3.9 函数 lv_dropdown_set_dir()形参描述

返回值：无。

9. lv_dropdown_set_symbol 函数

该函数的作用是设置关闭下拉列表时显示的箭头或其他符号，其函数原型如下所示：

```
void lv_dropdown_set_symbol(lv_obj_t * obj,const char*symbol)
```

该函数的形参描述如表 16.3.10 所示：

参数	描述
obj	指向下拉列表对象的指针
symbol	像 LV_SYMBOL_DOWN 或 NULL 这样的文本不会绘制图标

表 16.3.10 函数 lv_dropdown_set_symbol()形参描述

返回值：无。

注意：如果符号是图像，可以应用角度和缩放变换。例如，当下拉被选中(打开)时，将符号旋转 180 度。

16.4 下拉列表部件的实验

16.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建了多个 dropdown 部件以及一个 label 部件，本实验主要测试 dropdown 部件的符号设置以及下拉方向的设置。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 16 dropdown (下拉列表)》。

16.4.2 软件设计

16.4.2.1 程序流程图

本实验的程序流程图，如下图 16.4.2.1 所示：

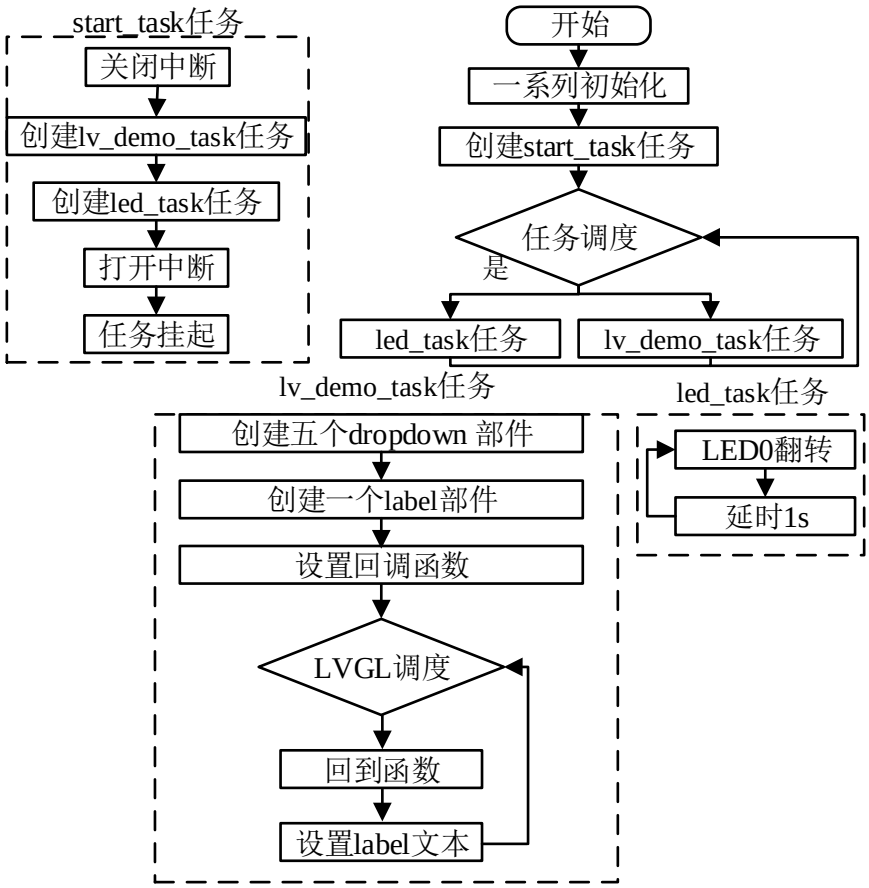


图 16.4.2.1 dropdown 部件实验流程图

16.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

(1) lv_mainstart 函数

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_dropdown_1();
    lv_example_dropdown_2();
}
```

此函数创建两个 dropdown 实验。

(2) lv_example_dropdown_1 函数

```
/**
 * @brief  例 1
 * @param  无
 * @return 无
 */
static void lv_example_dropdown_1(void)
{
    const lv_font_t *dropdown_font;
    lv_coord_t dropdown_width;

    if (scr_act_width() <= 320) {
        dropdown_font = &lv_font_montserrat_14;
        dropdown_width = 90;
    }
    else if (scr_act_width() <= 480) {
        dropdown_font = &lv_font_montserrat_18;
        dropdown_width = 120;
    }
    else {
        dropdown_font = &lv_font_montserrat_22;
        dropdown_width = 150;
    }
    /* 定义并创建下拉列表 */
    lv_obj_t* dropdown = lv_dropdown_create(lv_scr_act());
    /* 添加下拉列表选项 */
    lv_dropdown_set_options(dropdown, options);
    /* 设置下拉列表字体 */
```

```
    lv_obj_set_style_text_font(dropdown, dropdown_font, LV_PART_MAIN);
    /* 设置下拉列表宽度 */
    lv_obj_set_width(dropdown, dropdown_width);
    /* 设置下拉列表位置 */
    lv_obj_align(dropdown, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);
    /* 定义并创建标签 */
    lv_obj_t* label = lv_label_create(lv_scr_act());
    /* 设置标签表字体 */
    lv_obj_set_style_text_font(label, dropdown_font, LV_PART_MAIN);
    /* 设置标签宽度 */
    lv_obj_set_width(label, dropdown_width);
    /* 设置标签位置 */
    lv_obj_align_to(label, dropdown, LV_ALIGN_OUT_TOP_MID, 15,
                    -scr_act_height() / 8);
    /* 设置标签文本 */
    lv_label_set_text(label, "option 1");
    /* 添加下拉列表回调 */
    lv_obj_add_event_cb(dropdown, event_dropdown_cb, LV_EVENT_ALL, label);
}
```

此函数主要创建一个 dropdown 部件和 label 部件，然后设置部件相关的属性，最后调用 lv_obj_add_event_cb 函数添加回调函数 event_dropdown_cb，该回调函数如下所示：

(3) event_dropdown_cb 回调函数

```
/**
 * @brief  下拉列表事件回调
 * @param  无
 * @return 无
 */
static void event_dropdown_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* dropdown = lv_event_get_target(e);
    lv_obj_t* label = (lv_obj_t*)e->user_data;
    /* 定义 label 文本 */
    if (LV_EVENT_VALUE_CHANGED == code) {
        char buf[10];
        lv_dropdown_get_selected_str(dropdown, buf, sizeof(buf));
        lv_label_set_text(label, buf);
    }
}
```

此函数主要作用是获取下拉列表部件按下项的文本，然后更新 label 部件文本并显示按下表项的文本。

(3) lv_example_dropdown_2 函数

```
/**
```

```
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_dropdown_2(void)
{
    const lv_font_t* dropdown_font;
    lv_coord_t dropdown_width;

    if (scr_act_width() <= 320) {
        dropdown_font = &lv_font_montserrat_14;
        dropdown_width = 90;
    }
    else if (scr_act_width() <= 480) {
        dropdown_font = &lv_font_montserrat_18;
        dropdown_width = 120;
    }
    else {
        dropdown_font = &lv_font_montserrat_22;
        dropdown_width = 150;
    }

    lv_obj_t* dropdown; /* 定义下拉列表 */

    dropdown = lv_dropdown_create(lv_scr_act());            /* 创建下拉列表 */
    lv_dropdown_set_options_static(dropdown, options);      /* 添加下拉列表选项 */
    /* 设置下拉列表字体 */
    lv_obj_set_style_text_font(dropdown, dropdown_font, LV_PART_MAIN);
    /* 设置下拉列表宽度 */
    lv_obj_set_width(dropdown, dropdown_width);
    lv_dropdown_set_dir(dropdown, LV_DIR_BOTTOM);           /* 设置下拉列表方向 */
    lv_dropdown_set_symbol(dropdown, LV_SYMBOL_DOWN);       /* 设置下拉列表符号 */
    /* 设置下拉列表位置 */
    lv_obj_align(dropdown, LV_ALIGN_CENTER, scr_act_width() / 8,
                 -3 * scr_act_height() / 8);

    dropdown = lv_dropdown_create(lv_scr_act());            /* 创建下拉列表 */
    lv_dropdown_set_options_static(dropdown, options);      /* 添加下拉列表选项 */
    /* 设置下拉列表字体 */
    lv_obj_set_style_text_font(dropdown, dropdown_font, LV_PART_MAIN);
    /* 设置下拉列表宽度 */
    lv_obj_set_width(dropdown, dropdown_width);
    /* 设置下拉列表方向 */
```

```
    lv_dropdown_set_dir(dropdown, LV_DIR_LEFT);
    /* 设置下拉列表符号 */
    lv_dropdown_set_symbol(dropdown, LV_SYMBOL_LEFT);
    /* 设置下拉列表位置 */
    lv_obj_align(dropdown, LV_ALIGN_CENTER, scr_act_width() / 8,
                 -1 * scr_act_height() / 8);

    dropdown = lv_dropdown_create(lv_scr_act());           /* 创建下拉列表 */
    lv_dropdown_set_options_static(dropdown, options);     /* 添加下拉列表选项 */
    /* 设置下拉列表字体 */
    lv_obj_set_style_text_font(dropdown, dropdown_font, LV_PART_MAIN);
    /* 设置下拉列表宽度 */
    lv_obj_set_width(dropdown, dropdown_width);
    /* 设置下拉列表方向 */
    lv_dropdown_set_dir(dropdown, LV_DIR_RIGHT);
    /* 设置下拉列表符号 */
    lv_dropdown_set_symbol(dropdown, LV_SYMBOL_RIGHT);
    /* 设置下拉列表位置 */
    lv_obj_align(dropdown, LV_ALIGN_CENTER, scr_act_width() / 8,
                 1 * scr_act_height() / 8);

    dropdown = lv_dropdown_create(lv_scr_act());           /* 创建下拉列表 */
    /* 添加下拉列表选项 */
    lv_dropdown_set_options_static(dropdown, options);
    /* 设置下拉列表字体 */
    lv_obj_set_style_text_font(dropdown, dropdown_font, LV_PART_MAIN);
    /* 设置下拉列表宽度 */
    lv_obj_set_width(dropdown, dropdown_width);
    /* 设置下拉列表方向 */
    lv_dropdown_set_dir(dropdown, LV_DIR_TOP);
    /* 设置下拉列表符号 */
    lv_dropdown_set_symbol(dropdown, LV_SYMBOL_UP);
    /* 设置下拉列表位置 */
    lv_obj_align(dropdown, LV_ALIGN_CENTER, scr_act_width() / 8,
                 3 * scr_act_height() / 8);
}
```

此函数主要创建四个下拉列表部件，然后设置这些部件的下拉方向以及下拉符号。

16.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 16.4.3.1 所示：

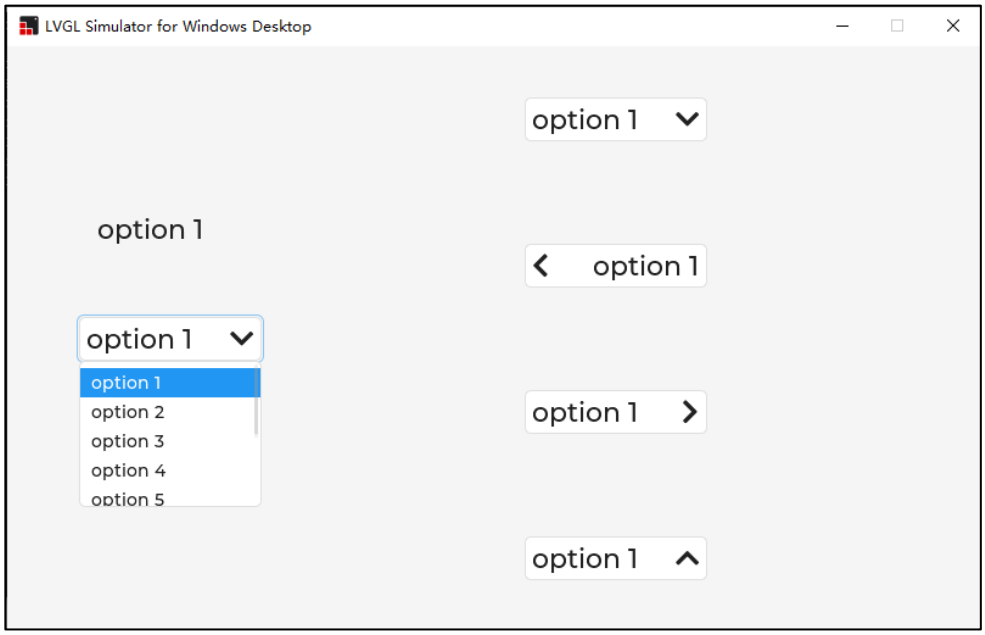


图 16.4.3.1 dropdown 复选框实验

第十七章 图片部件(lv_img)

lv_img 是一个图片部件，它根据传入的图片源来显示图片，因为 LVGL 为了提高图片源的灵活性，所以图片部件与 BMP/JPEG/PNG 图片解码库紧密相连。注意：图片部件可以读取 C 数组格式的文件、图片二进制的.bin 文件以及图标字体。如果图片部件直接显示 BMP/JPEG 等图片，则经过解码才可以显示，关于这些解码库的使用，我们会在后面章节讲解。

本章分为如下几部分：

17.1 图片部件的组成
17.2 图片部件的相关知识
17.3 图片部件的 API 函数
17.4 图片部件的实验

17.1 图片部件的组成

图片部件是由 LV_PART_MAIN 部分组成的，使用了典型背景样式属性和图像样式属性绘画成背景矩形。

17.2 图片部件的相关知识

17.2.1 图像源选择

图片部件的图像源来源分为三种：

① 图片源 C 数组
② 存储在外部(如 SD 卡)的文件
③ 带有符号的文本(请参考第八章 8.2 小节)

图片部件设置图像源的函数为 lv_img_set_src(img, src)，该函数的第一个形参为图片部件的控制块，而第二个形参为图像源。下面我们分别地讲解这些图像源的生成和使用。

1. 图片源 C 数组生成与使用

(1) 如何生成图片的 C 数组

如果我们想把 PNG、JPG 和 BMP 图像格式转换成 C 数组，可以使用官方在线转换工具进行转换操作，该在线转换工具地址为：https://lvgl.io/tools/imageconverter。如图 17.2.1.1 所示：

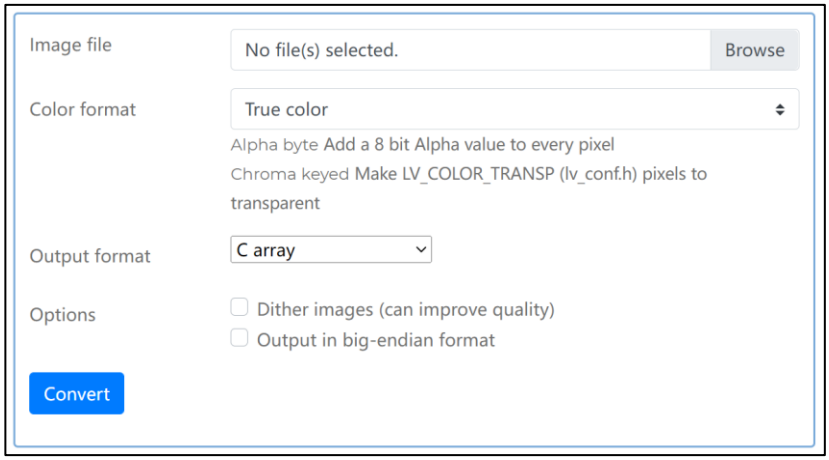


图 17.2.1.1 在线转换工具

首先点击上图中 Browse 按键导入图片，然后点击图中 Color format 选择颜色格式(RAW)，其次点击图中 Output format 输出格式(C array)，最后点击 Convert 生成。

注意：上图中，Options 的两个选项分为抖动图像和大端格式。

(2)图片的 C 数组的使用步骤

第一步：C 数组文件导入工程当中，在工程文件中调用 LV_IMG_DECLARE(xxx)宏定义对图片源进行声明。

第二步：调用 lv_img_create 函数创建图片部件。

第三步：调用 lv_img_set_src(img1,&xxx)函数设置图片源。

2. 外部图片源文件的生成和使用

(1) 外部图片源文件生成

打开在线转换工具，如上图 17.2.1.1 所示，然后选择颜色格式为 True color，其次点击 Output format 输出格式为 Binary RGB565，最后点击 Convert 生成。

(2) 外部图片源文件的使用步骤

第一步：使能 LVGL 的文件系统，请参考第七章节的内容。

第二步：拷贝文件到指定的 SD 中。

第三步：调用 lv_img_create 函数创建图片部件。

第四步：调用 lv_img_set_src(img1, "0:APP/my_img.bin")函数设置图片源。

3. 带有符号的文本作为图像源

在第八章中，大家已经了解了 LVGL 内置图标字体的使用，这些图标字体可以作为图片部件的图像源，它们可以调用 lv_img_set_src(img1, LV_SYMBOL_XXX)函数进行设置，当然也可以像 label 部件一样，图标字体和字符串组成显示，如下源码所示：

```
lv_img_set_src(img, LV_SYMBOL_DUMMY "Some text")
```

17.2.2 图像重新着色

图像重新着色是指一种颜色可以与图像的每个像素混合并具有给定的强度。这可以用于显示图像的不同状态，例如选中、未激活、按下等状态，简单来讲：有一张图片的颜色为黑色，当用户按下时，可以把该图片重新着色成其他的颜色。

图像的重新着色的方法是设置 img_recolor_opa 和 img_recolor 样式属性，注意：img_recolor_opa 属性范围为 0-255 的数值，如果 img_recolor_opa 设置为 LV_OPA_TRANSP(数值：0)，则表示无透明度，如果设置 LV_OPA_COVER(数值：255)，则表示全透明。

17.2.3 图像自动大小

如果图像对象的宽度或高度设置为 LV_SIZE_CONTENT，则对象的大小将根据图像源在各自方向上的大小来设置。

17.2.4 图像偏移

图像偏移是指图片部件的区域所显示的图像进行偏移，如果图像某区域偏移出图片部件的区域，则该区域的图像绘画在图片部件区域的左边，如图 17.2.4 所示：

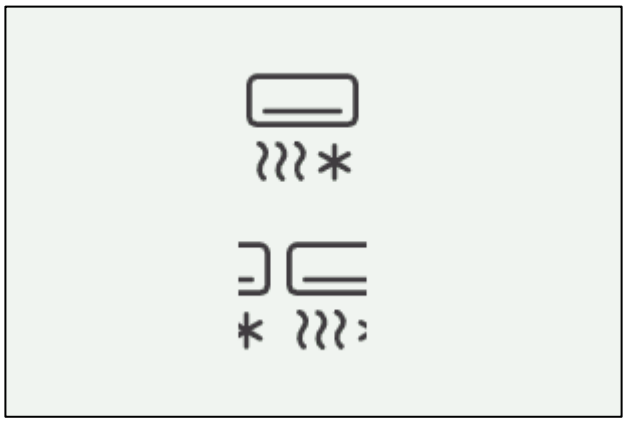


图 17.2.4 图像偏移

设置图像偏移可使用以下函数设置：

```
lv_img_set_offset_x(img, x_ofs)/* 图像往 X 轴偏移 */
lv_img_set_offset_y(img, y_ofs)/* 图像往 Y 轴偏移 */
```

17.2.5 图像其他属性的设置

1. 图像的缩放

图像缩放是指对数字图像的大小进行调整的过程。图像缩放是一种非平凡的过程，需要在处理效率以及结果的平滑度和清晰度上做一个权衡。当一个图像的大小增加之后，组成图像的像素的可见度将会变得更高，从而使得图像表现得“软”。相反地，缩小一个图像将会增强它的平滑度和清晰度。

在 LVGL 中，图像缩放可调用 lv_img_set_zoom 函数设置。该函数具有两个形参，第一个形参为图片部件的控制块，而第二个形参为缩放的比例。如果缩放的比例设置为 256 或 LV_IMG_ZOOM_NONE，则表示禁用缩放。如果缩放的比例设置为 128，则表示缩放最小。如果缩放的比例设置为 512，则表示缩放最大。如图 17.2.5.1 所示：

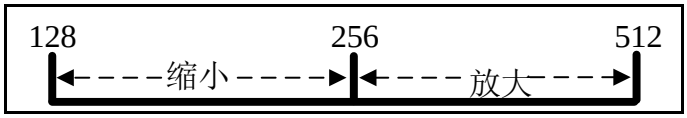


图 17.2.5.1 图像缩放比例

注意：如果缩放比例设置为 255 和 257，那么 255 的比例图像比原图像缩小 10%，而 257 的比例图像比原图像放大 10%，由此类推。

2. 图像旋转

图像旋转是指图像以某一点为中心旋转一定的角度，形成一幅新的图像的过程。当然这个点通常就是图像的中心。既然是按照中心旋转，自然会有这样一个属性：旋转前和旋转后的点离中心的位置不变。

在 LVGL 中，图像旋转可调用 lv_img_set_angle 函数设置，该函数的第二个形参表示旋转的角度。如果用户想把图像旋转 45°，则设置旋转角度为 450。

注意：因为 LVGL 设置旋转角度是以 0.1 度为单位。

3. 设置图像中心

在 LVGL 中，图像中间的坐标默认为该图像的中心点，如图 17.2.5.2 所示：

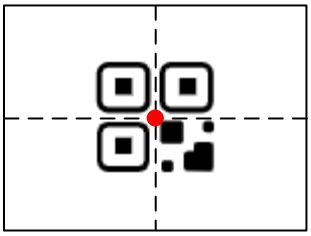


图 17.2.5.2 图像默认中心

如果设置图像旋转时，则该图像围绕着中心旋转。如图 17.2.5.3 所示：

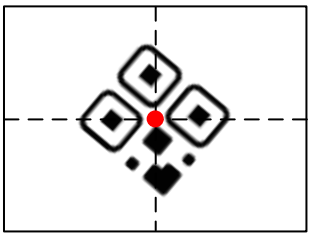


图 17.2.5.3 旋转 45°

注意：如果用户设置图像的中心为其他位置，则该图像围绕着该坐标旋转。

LVGL 提供图像旋转的函数为 lv_img_set_pivot(img, pivot_x, pivot_y)，该函数的第二和第三个形参表示图像中心坐标的设置。

注意：设置（0, 0）是左上角。例如屏幕显示就以屏幕左上角为原点，如果图片定义在其他部件上，就以该控件的左上角为原点。

4. 图像大小模式设置

默认情况下，图像被缩放或旋转时，图像对象的实际坐标不会改变。较大的内容会溢出对象的边界。这也意味着布局不受转换的影响。由于 LVGL 的图像模式默认为 LV_IMG_SIZE_MODE_VIRTUAL，在该模式下，图像旋转的宽度/高度将被改变，这可能导致旋转完的图像比原来图像的大小要大。为了解决这一状况，用户可设置 LVGL 图像模式为 LV_IMG_SIZE_MODE_REAL。

注意：LVGL 图像模式可调用 lv_img_set_size_mode 函数设置。

17.3 图片部件的 API 函数

LVGL 提供给用户的图片部件相关函数，如表 17.3.1 所示：

函数	描述
lv_img_create()	创建一个图像对象
lv_img_set_src()	设置图像源显示
lv_img_set_offset_x()	设置图像源的 x 偏移量
lv_img_set_offset_y()	设置图像源的 y 偏移量
lv_img_set_pivot()	设置图像的旋转中心
lv_img_set_angle()	设置图像的旋转角度
lv_img_set_zoom()	设置图像的缩放倍数
lv_img_set_antialias()	启用/禁用转换的抗锯齿功能(旋转,缩放)
lv_img_set_size_mode()	设置图像的模式
lv_img_get_src()	获取图像的来源
lv_img_get_offset_x()	获取 img 对象的 offset.x 属性
lv_img_get_offset_y()	获取 img 对象的 offset.y 属性
lv_img_get_angle()	获取图像的旋转角度

lv_img_get_pivot()	获取图像的旋转中心
lv_img_get_zoom()	获取图像的缩放系数
lv_img_get_antialias()	获取转换(旋转,缩放)是否抗锯齿
lv_img_get_size_mode()	获取图像模式

表 17.3.1 img 部件函数描述

接下来我们来讲解 LVGL 的 img 部件常用的函数有哪些：

1. lv_img_create 函数

该函数的作用是创建一个图像对象，其函数原型如下所示：

```
lv_obj_t * lv_img_create(lv_obj_t * parent)
```

该函数的形参描述如表 17.3.2 所示：

参数	描述
parent	指向对象的指针,它将是图像的父对象

表 17.3.2 函数 lv_img_create()形参描述

返回值：指向创建的图像的指针。

2. lv_img_set_src 函数

该函数的作用是设置像素图以按图像显示，其函数原型如下所示：

```
void lv_img_set_src(lv_obj_t * obj,const void * src_img)
```

该函数的形参描述如表 17.3.3 所示：

参数	描述
obj	指向图像对象的指针
src_img	图像数据

表 17.3.3 函数 lv_img_set_src()形参描述

返回值：无。

3. lv_img_set_offset_x 函数

该函数的作用是设置图像源的 x 偏移量。因此将以新的原点显示图像。，其函数原型如下所示：

```
void lv_img_set_offset_x(lv_obj_t * obj,lv_coord_t x)
```

该函数的形参描述如表 17.3.4 所示：

参数	描述
obj	指向图像对象的指针
x	沿 x 轴的新偏移量

表 17.3.4 函数 lv_img_set_offset_x()形参描述

返回值：无。

4. lv_img_set_offset_y 函数

该函数的作用是设置图像源的 y 偏移量。因此将以新的原点显示图像。，其函数原型如下所示：

```
void lv_img_set_offset_y(lv_obj_t * obj,
                         lv_coord_t y)
```

该函数的形参描述如表 17.3.5 所示：

参数	描述
obj	指向图像对象的指针
y	沿 y 轴的新偏移量

表 17.3.5 函数 lv_img_set_offset_y()形参描述

返回值：无。

5. lv_img_set_pivot 函数

该函数的作用是设置图像的旋转中心。图像将围绕此点旋转，其函数原型如下所示：

```
void lv_img_set_pivot(lv_obj_t * obj,
                      lv_coord_t ivot_x,
                      lv_coord_t ivot_y)
```

该函数的形参描述如表 17.3.6 所示：

参数	描述
obj	指向图像对象的指针
x	图像的旋转中心 x
y	图像的旋转中心 y

表 17.3.6 函数 lv_img_set_pivot()形参描述

返回值：无。

6. lv_img_set_angle 函数

该函数的作用是设置图像的旋转角度。图像将围绕由 lv_img_set_pivot()决定，其函数原型如下所示：

```
void lv_img_set_angle(lv_obj_t * obj,int16_t  angle)
```

该函数的形参描述如表 17.3.7 所示：

参数	描述
obj	指向图像对象的指针
angle	旋转角度,以度为单位,分辨率为 0.1 度(0..3600：顺时针方向)

表 17.3.7 函数 lv_img_set_angle()形参描述

返回值：无。

7. lv_img_set_zoom 函数

该函数的作用是设置图像的缩放倍数，其函数原型如下所示：

```
void lv_img_set_zoom(lv_obj_t * obj,uint16_t zoom)
```

该函数的形参描述如表 17.3.8 所示：

参数	描述
obj	指向图像对象的指针
zoom	缩放系数

表 17.3.8 函数 lv_img_set_zoom()形参描述

返回值：无。

8. lv_img_set_antialias 函数

该函数的作用是启用/禁用转换的抗锯齿功能(旋转,缩放)，其函数原型如下所示：

```
void lv_img_set_antialias(lv_obj_t * obj,bool antialias)
```

该函数的形参描述如表 17.3.9 所示：

参数	描述
obj	指向图像对象的指针
antialias	true：抗锯齿; flase：不抗锯齿

表 17.3.9 函数 lv_img_set_antialias()形参描述

返回值：无。

17.4 图片部件的实验

17.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建了六个 Slider 部件以及一个 img 部件，当我们滑动这些 Slider 部件时，系统会根据这些滑动值修改 img 部件的样式属性。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 17 lv_img(图像)》。

17.4.2 软件设计

17.4.2.1 程序流程图

本实验的程序流程图，如下图 17.4.2.1 所示：

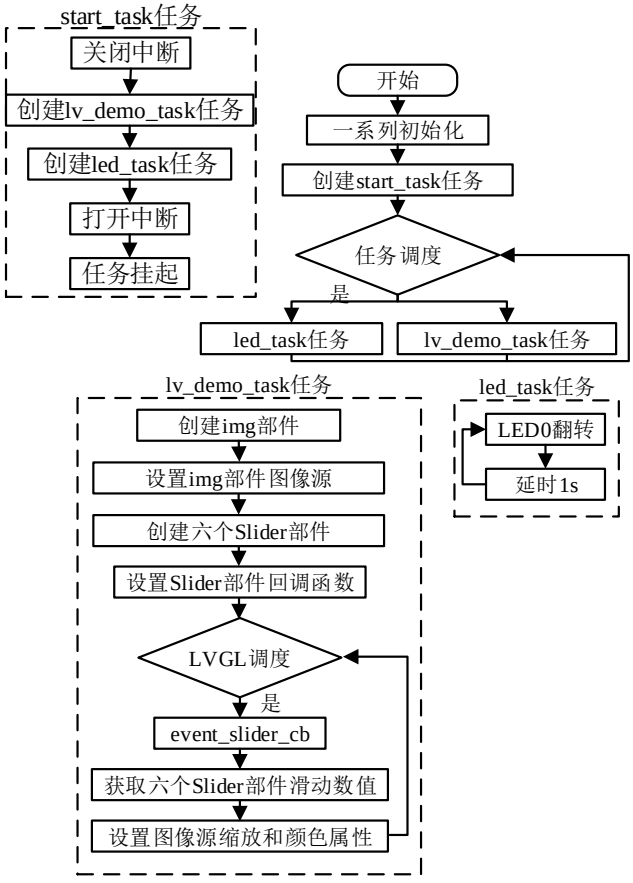


图 17.4.2.1 img 部件实验流程图

17.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  自定义创建滑杆
 * @param  color:颜色值
 * @return 无
 */
static lv_obj_t* my_slider_create(lv_color_t color)
{
    lv_obj_t* slider = lv_slider_create(lv_scr_act());     /* 定义并创建滑杆 */
    lv_obj_set_width(slider, scr_act_height() / 12);       /* 设置滑杆的宽度 */
    /* 设置滑杆的高度 */
    lv_obj_set_height(slider, scr_act_height() - 3.5 * scr_act_height() / 10);
    lv_obj_set_style_bg_color(slider, color, LV_PART_KNOB); /* 设置滑杆把手颜色 */
    /* 设置滑杆颜色 */
    lv_obj_set_style_bg_color(slider,
                              lv_color_darken(color, LV_OPA_40),
                              LV_PART_INDICATOR);
    /* 设置滑杆背景颜色 */
    lv_obj_set_style_bg_color(slider,
                                lv_color_darken(color, LV_OPA_40),
                                LV_PART_MAIN);
    /* 设置滑杆回调 */
    lv_obj_add_event_cb(slider, event_slider_cb, LV_EVENT_VALUE_CHANGED, NULL);
    return slider;
}

/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_img(void)
{
    img = lv_img_create(lv_scr_act());      /* 定义并创建图像 */
    lv_img_set_src(img, &img_gear);         /* 填充图像 */
    /* 设置图像位置 */
    lv_obj_align(img, LV_ALIGN_CENTER, -scr_act_width() / 4, 0);
    /* 设置图像旋转的中心点 */
    lv_img_set_pivot(img, IMG_WIDTH / 2, IMG_HEIGHT / 2);
    lv_obj_update_layout(img);              /* 手动更新图像参数 */
    /* 缩放控制滑杆 *//* 创建缩放滑杆 */
```

```
    slider_zoom = my_slider_create(lv_palette_main(LV_PALETTE_GREY));
    lv_slider_set_range(slider_zoom, 128, 512);             /* 设置滑杆的范围 */
    lv_slider_set_value(slider_zoom, 256, LV_ANIM_OFF);     /* 设置滑杆的值 */
    lv_obj_align(slider_zoom, LV_ALIGN_CENTER, 0, 0);       /* 设置滑杆位置 */
    /* 旋转控制滑杆 *//* 创建缩放滑杆 */
    slider_angle = my_slider_create(lv_palette_main(LV_PALETTE_GREY));
    lv_slider_set_range(slider_angle, 0, 7200);             /* 设置滑杆的范围 */
    lv_slider_set_value(slider_angle, 3600, LV_ANIM_OFF);  /* 设置滑杆的值 */
    /* 设置滑杆位置 */
    lv_obj_align_to(slider_angle, slider_zoom, LV_ALIGN_OUT_LEFT_MID,
                    scr_act_height() / 5, 0);
    /* 红色通道控制滑杆 *//* 创建缩放滑杆 */
    slider_r = my_slider_create(lv_palette_main(LV_PALETTE_RED));
    lv_slider_set_range(slider_r, LV_OPA_0, LV_OPA_100);/* 设置滑杆的范围 */
    lv_slider_set_value(slider_r, LV_OPA_20, LV_ANIM_OFF);/* 设置滑杆的值 */
    /* 设置滑杆位置 */
    lv_obj_align_to(slider_r, slider_angle, LV_ALIGN_OUT_LEFT_MID,
                    scr_act_height() / 5, 0);
    /* 绿色通道控制滑杆 *//* 创建缩放滑杆 */
    slider_g = my_slider_create(lv_palette_main(LV_PALETTE_GREEN));
    lv_slider_set_range(slider_g, LV_OPA_0, LV_OPA_100);   /* 设置滑杆的范围 */
    lv_slider_set_value(slider_g, LV_OPA_30, LV_ANIM_OFF); /* 设置滑杆的值 */
    /* 设置滑杆位置 */
    lv_obj_align_to(slider_g, slider_r, LV_ALIGN_OUT_LEFT_MID,
    scr_act_height() / 5, 0);
    /* 蓝色通道控制滑杆 *//* 创建缩放滑杆 */
    slider_b = my_slider_create(lv_palette_main(LV_PALETTE_BLUE));
    lv_slider_set_range(slider_b, LV_OPA_0, LV_OPA_100);   /* 设置滑杆的范围 */
    lv_slider_set_value(slider_b, LV_OPA_40, LV_ANIM_OFF); /* 设置滑杆的值 */
    /* 设置滑杆位置 */
    lv_obj_align_to(slider_b, slider_g, LV_ALIGN_OUT_LEFT_MID,
                    scr_act_height() / 5, 0);
    /* 颜色填充控制滑杆 */
    slider_intense = my_slider_create(lv_color_black());       /* 创建缩放滑杆 */
    /* 设置滑杆的范围 */
    lv_slider_set_range(slider_intense, LV_OPA_0, LV_OPA_100);
    /* 设置滑杆的值 */
    lv_slider_set_value(slider_intense, LV_OPA_50, LV_ANIM_OFF);
    /* 设置滑杆位置 */
    lv_obj_align_to(slider_intense, slider_b,
    LV_ALIGN_OUT_LEFT_MID, scr_act_height() / 5, 0);
}
```

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_img();
}
```

从上述源码可知：首先创建 img 部件，其次设置 img 图像源，然后创建六个 Slider 部件，最后设置 event_slider_cb()回调函数，该回调函数如下所示：

```
/**
 * @brief  滑杆事件回调
 * @param  e：事件描述
 * @return 无
 */
static void event_slider_cb(lv_event_t* e)
{
    lv_img_set_zoom(img, lv_slider_get_value(slider_zoom)); /* 设置图像缩放 */
    /* 设置图像旋转角度 */
    lv_img_set_angle(img, lv_slider_get_value(slider_angle) - 3600);
    /* 设置图像颜色 */
    lv_obj_set_style_img_recolor(  img,

    lv_color_make(lv_slider_get_value(slider_r),
                  lv_slider_get_value(slider_g),
                   lv_slider_get_value(slider_b)),
                   LV_PART_MAIN);
    /* 设置图像颜色透明度 */
    lv_obj_set_style_img_recolor_opa(img, lv_slider_get_value(slider_intense),
                                     LV_PART_MAIN);
}
```

从上述源码可知：首先图像的缩放根据 slider_zoom 部件的数值而定，其次图像的旋转角度根据 slider_angle 部件数值的而定，然后图像的颜色根据 slider_r、slider_g 和 slider_b 部件的数值而定，最后图像的透明度属性设置根据 slider_intense 部件的数值而定。

17.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 17.4.3.1 所示：

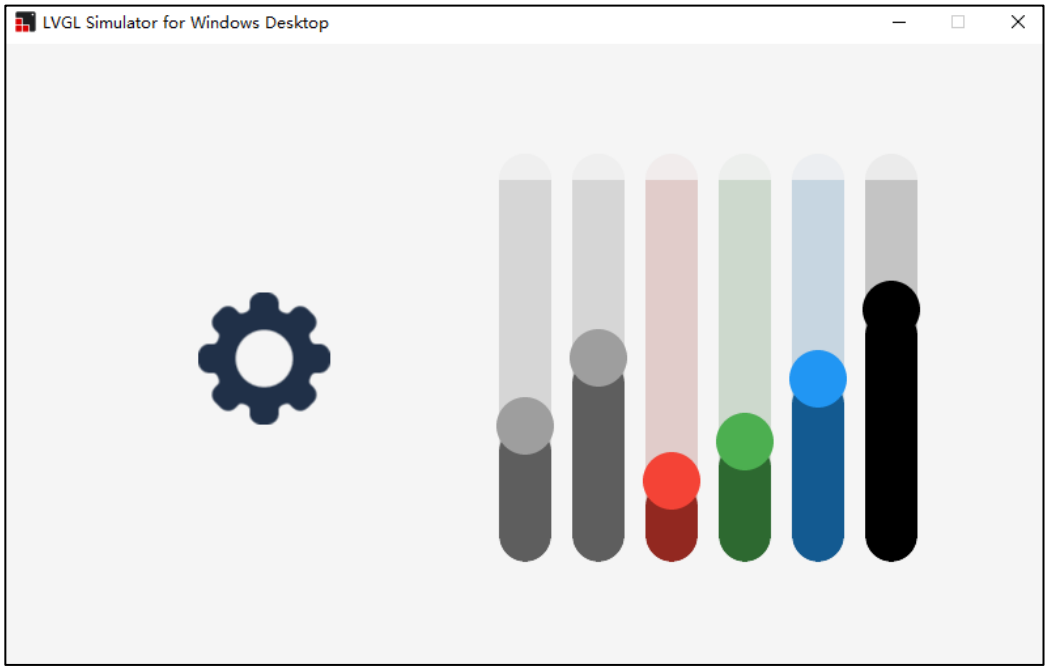


图 17.4.3.1 img 实验

第十八章 标签部件(lv_label)

在 LVGL 中，label 部件作为文本显示的作用，例如标题、显示字体等。在前面学习 btn 部件时，因为该部件并没有内置文本，所以需要用户去创建 label 部件为 btn 部件的子类。label 部件算是一个比较常用的部件，希望大家重点掌握该部件的相关知识。

本章分为如下几部分：

18.1 标签部件的组成

18.2 标签部件的相关知识

18.3 标签部件的 API 函数

18.4 标签部件的实验

18.1 标签部件的组成

在 LVGL 中，label 部件由三个部分组成，如下所示：

① LV_PART_MAIN：使用所有典型的背景属性和文本属性。

② LV_PART_SCROLLBAR：当文本大于小部件大小时，显示滚动条。

③ LV_PART_SELECTED：所选文本的样式(text_color 和 bg_color)。

根据上述的基础部分的描述，我们可以使用一个示意图来讲解 label 部件组成，如下图 18.1.1 所示：



图 18.1.1 label 部件组成部分

上图的标签和上述基础部分的标签是一一对应的，大家可根据 label 基础部分设置标签部件的样式属性。

18.2 标签部件的相关知识

18.2.1 文本设置

标签部件的文本设置方法有三种，这三种方式有哪些不一样呢，下面我们来讲解一下它们的不同之处。

第一种：lv_label_set_text 函数

该函数是 label 部件最常用的文本设置函数，它的存储文本的内存由动态内存分配的。

第二种：lv_label_set_text_fmt 函数

该函数为式样化输出函数，它的存储文本的内存也是由动态内存分配的。

注意：该函数与 C 语言的 sprintf 式样化输出函数类似，都可以设置“式样化字符串”和“参数表”，如下源码所示：

```
lv_label_set_text_fmt(label, "Value: %d", 15)
```

第三种：label_set_text_static 函数

该函数为静态文本设置。在这种情况下，文本不存储在动态内存中，而是直接使用给定的缓冲区，这意味着当函数退出时，数组不能是超出作用域的局部变量。

18.2.2 换行符设置

如果用户设置文本太长或者想把文本分两段显示，则可在内容部分添加“\n”表示换行。

老规矩，我们使用简单的实例来讲解本小节的内容，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* lv_label1 = lv_label_create(lv_scr_act());
    lv_label_set_text(lv_label1, "ALIENTEK \nLVGL \nDemo text");

    lv_obj_t* lv_label2 = lv_label_create(lv_scr_act());
    lv_obj_align_to(lv_label2, lv_label1,LV_ALIGN_OUT_BOTTOM_MID,0,0);
    /* 设置标签文本及偏移 */
    lv_label_set_text_fmt(lv_label2, "Label value %d", 20);
}
```

上述源码可知：我们创建两个 label 部件，label1 部件调用 lv_label_set_text 函数显示文本，该文本使用了“\n”换行，而 label2 部件调用 lv_label_set_text_fmt 式样化输出函数显示文本。如图 18.2.1.1 所示：

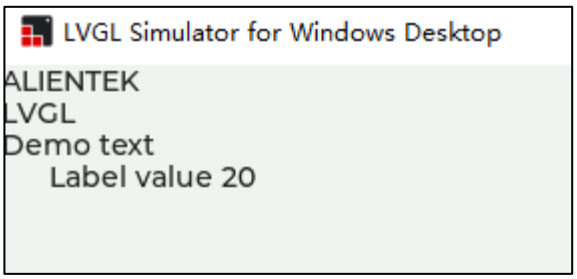


图 18.2.1.1 label 部件文本显示

18.2.3 文本长模式

默认情况下，label 部件的宽度和高度都设置为 LV_SIZE_CONTENT。因此，标签的大小自动扩展为文本大小。如果用户调用属性函数修改 label 部件的大小，则可以根据几个长模式策略来操作 label 宽度的行。

注意：如果文本的高度大于标签的高度，则可以应用策略。

长模式策略可调用 lv_label_set_long_mode 函数设置，该函数的第二个形参表示如下：

(1) LV_LABEL_LONG_WRAP (默认)：如果 label 部件的高度是 LV_SIZE_CONTENT，则该部件高度将被扩展，否则文本将被剪切。

(2) LV_LABEL_LONG_DOT：将 label 右下角的最后 3 个字符替换为点(.)。

(3) LV_LABEL_LONG_SCROLL：如果文本比 label 部件宽，则水平前后滚动。如果文本比 label 部件高，垂直滚动。注意：只滚动一个方向，水平滚动具有更高的优先级。

(4) LV_LABEL_LONG_SCROLL_CIRCULAR：如果文本比 label 部件宽，则水平滚动；如果文本比 label 部件高，垂直滚动。注意：只滚动一个方向，水平滚动具有更高的优先级。

(5) LV_LABEL_LONG_CLIP：简单的剪辑 label 部件外面的文本部分。

18.2.4 文本着色

在文本中，可使用命令为文本部分重新着色。这个特性可以通过 lv_label_set_recolor 函数为每个 label 部件单独启用。

下面我们还是使用一个简单实例来讲解本小节的内容，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* lv_label1 = lv_label_create(lv_scr_act());
    lv_label_set_text(lv_label1, "ALIENTEK #ff0000 LVGL# Demo text");
    /* 使能重新着色 */
    lv_label_set_recolor(lv_label1,true);
}
```

从上述源码可知：我们创建一个 label 部件，然后设置文本中的“LVGL”字符串颜色为红色(ff0000)，最后使能文本重新着色功能。如图 18.2.4.1 所示：

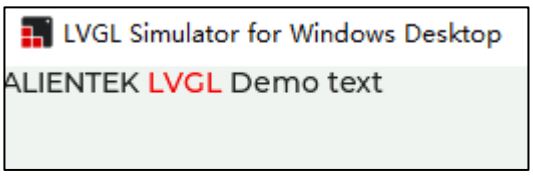


图 18.2.4.1 设置文本颜色

18.2.6 长文本设置

如果 label 部件的文本超出 LVGL 默认有效承受范围(>40K 字节)，则必须启用长文本模式，该模式启用方式是把 LV_LABEL_LONG_TXT_HINT(lv_conf.h)宏定义设置为 1。

18.3 标签部件的 API 函数

LVGL 提供给用户的标签部件相关函数，如表 18.3.1 所示：

函数	描述
label 部件设置函数	
lv_label_create()	创建标签对象
lv_label_set_text()	设置标签的新文本
lv_label_set_text_fmt()	设置标签的新格式的文本
lv_label_set_text_static()	设置一个静态文本
lv_label_set_long_mode()	使用更长的文本然后设置对象大小来设置标签
lv_label_set_recolor()	启用重新着色
lv_label_set_text_sel_start()	设置选择开始索引
lv_label_set_text_sel_end()	设置选择结束索引
lable 部件获取函数	
lv_label_get_text()	获取标签文本
lv_label_get_long_mode()	获取标签的长模式
lv_label_get_recolor()	获取重新着色属性
lv_label_get_letter_pos()	获取字母的相对 x 和 y 坐标
lv_label_get_letter_on()	获取标签相对点上的字母索引
lv_label_is_char_under_pos()	检查是否在一个点下绘制了一个字符
lv_label_get_text_selection_start()	获取选择开始索引

lv_label_get_text_selection_end()	获取选择结束索引
lv_label_ins_text()	在标签上插入文本。标签文本不能是静态的
lv_label_cut_text()	从标签中删除字符。标签文本不能是静态的

表 18.3.1 label 部件 API 函数

接下来我们来讲解 LVGL 标签部件常用的函数有哪些：

1. lv_label_create 函数

该函数的作用是创建一个 label 部件，该函数原型如下所示：

```
lv_obj_t * lv_label_create(lv_obj_t * parent)
```

该函数的形参，如表 18.3.2 所示：

参数	描述
parent	指向对象的指针

表 18.3.2 函数 lv_label_create()形参描述

返回值：无

2. lv_label_set_text 函数

该函数的作用是设置 label 部件的新文本，该函数原型如下所示：

```
void lv_label_set_text(lv_obj_t * obj, const char* text)
```

该函数的形参，如表 18.3.3 所示：

参数	描述
obj	指向对象的指针
text	'\ 0'终止的字符串。使用 NULL 刷新当前文本

表 18.3.3 函数 lv_label_set_text()形参描述

返回值：无

3. lv_label_set_text_fmt 函数

该函数的作用是设置 label 部件的新格式文本，该函数原型如下所示：

```
void lv_label_set_text_fmt(lv_obj_t * obj, const char * fmt,  ... )
```

该函数的形参，如表 18.3.4 所示：

参数	描述
obj	指向对象的指针
fmt	printf -类似格式

表 18.3.4 函数 lv_label_set_text_fmt()形参描述

返回值：无

4. lv_label_set_long_mode 函数

该函数的作用是将 label 部件的行为设置为比对象大小更长的文本，该函数原型如下所示：

```
void lv_label_set_long_mode(lv_obj_t * obj,lv_label_long_mode_t long_mode )
```

该函数的形参，如表 18.3.5 所示：

参数	描述
obj	指向对象的指针
long_mode	长模式

表 18.3.5 函数 lv_label_set_long_mode()形参描述

返回值：无

5. lv_label_set_recolor 函数

该函数的作用是启用重新着色，该函数原型如下所示：

```
void lv_label_set_recolor(lv_obj_t * obj, bool en )
```

该函数的形参，如表 18.3.6 所示：

参数	描述

obj	指向对象的指针
en	true：启用重新着色， false：禁用

表 18.3.6 函数 lv_label_set_recolor()形参描述

返回值：无

18.4 标签部件的实验

18.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建了 3 个 label 部件，它们分别设置可着色、长模式以及其他样式属性。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 18 lv_label(标签)》。

18.4.2 软件设计

18.4.2.1 程序流程图

本实验的程序流程图，如下图 18.4.2.1 所示：

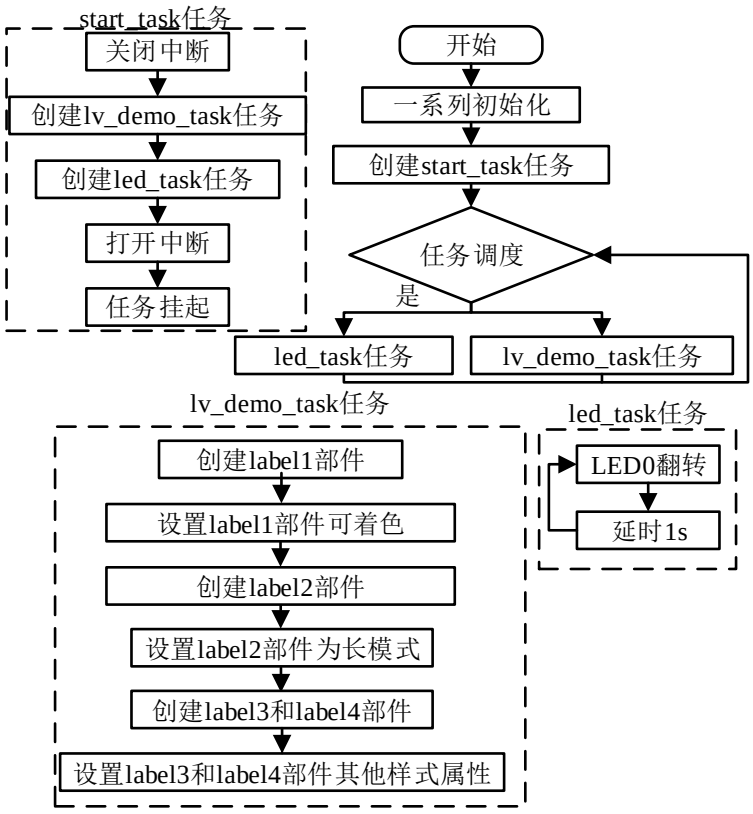


图 18.4.2.1 label 部件实验流程图

18.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例 1
 * @param  无
 * @return 无
 */
static void lv_example_label_1(void)
{
    if (scr_act_width() <= 320)
        font = &lv_font_montserrat_10;
    else if (scr_act_width() <= 480)
        font = &lv_font_montserrat_14;
    else
        font = &lv_font_montserrat_20;

    lv_obj_t* label = lv_label_create(lv_scr_act());       /* 定义并创建标签 */
    lv_label_set_long_mode(label, LV_LABEL_LONG_WRAP);     /* 设置标签长模式 */
    lv_obj_set_width(label, scr_act_width() / 3);          /* 设置标签宽度 */
    lv_obj_set_style_text_font(label, font, LV_PART_MAIN); /* 设置标签文本字体 */
    lv_obj_set_height(label, LV_SIZE_CONTENT); /* 标签长文本自动扩展高度 */
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    lv_label_set_recolor(label, true);                 /* 启用标签文本重新着色 */
    /* 设置标签位置 */
    lv_obj_align(label, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);
    /* 设置标签文本 */
    lv_label_set_text(label, "#0000ff Re-color# #ff00ff words# #ff0000 of a# "
                             "label, align the lines to the center"
                             "and  wrap long text automatically.");
}

/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_label_2(void)
{
    lv_obj_t* label = lv_label_create(lv_scr_act());       /* 定义并创建标签 */
    /* 设置标签长模式 */
    lv_label_set_long_mode(label, LV_LABEL_LONG_SCROLL_CIRCULAR);
```

```
    lv_obj_set_width(label, scr_act_width() / 3);              /* 设置标签宽度 */
    lv_obj_set_style_text_font(label, font, LV_PART_MAIN);     /* 设置文本字体 */
    lv_obj_align(label, LV_ALIGN_CENTER, 0, 0);                /* 设置标签位置 */
    /* 设置标签文本 */
    lv_label_set_text(label, "It is a circularly scrolling text. ");
}

/**
 * @brief  例 3
 * @param  无
 * @return 无
 */
static void lv_example_label_3(void)
{
    lv_obj_t* label = lv_label_create(lv_scr_act());   /* 定义并创建标签 */
    lv_obj_set_width(label, scr_act_width() / 3);      /* 设置标签宽度 */
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 设置标签文本字体 */
    lv_obj_set_style_text_font(label, font, LV_PART_MAIN);
    /* 设置标签位置 */
    lv_obj_align(label, LV_ALIGN_CENTER, scr_act_width() / 3, 0);
    /* 设置标签文本 */
    lv_label_set_text_fmt(label, "Label can set text like %s", "printf");
    /* 定义并创建阴影标签 */
    lv_obj_t* label_shadow = lv_label_create(lv_scr_act());
    /* 设置标签文本透明度 */
    lv_obj_set_style_text_opa(label_shadow, LV_OPA_30, LV_PART_MAIN);
    /* 设置标签文本颜色 */
    lv_obj_set_style_text_color(label_shadow, lv_color_black(), LV_PART_MAIN);
    /* 设置标签宽度 */
    lv_obj_set_width(label_shadow, scr_act_width() / 3);
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label_shadow,
    LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 设置标签文本字体 */
    lv_obj_set_style_text_font(label_shadow, font, LV_PART_MAIN);
    /* 设置标签位置 */
    lv_obj_align_to(label_shadow, label, LV_ALIGN_TOP_LEFT, 2, 2);
    /* 设置标签文本 */
    lv_label_set_text(label_shadow, lv_label_get_text(label));
}
```

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_label_1();
    lv_example_label_2();
    lv_example_label_3();
}
```

从上述源码可知：首先 lv_example_label_1 函数主要设置 label 部件的文本可着色，其次 lv_example_label_2 函数主要设置 label 部件的文本为长模式，最后 lv_example_label_3 函数主要设置 label 部件其他样式属性。

18.4.3 下载验证

把工程编译并下载到我们的开发板中， 如图 18.4.3.1 所示：

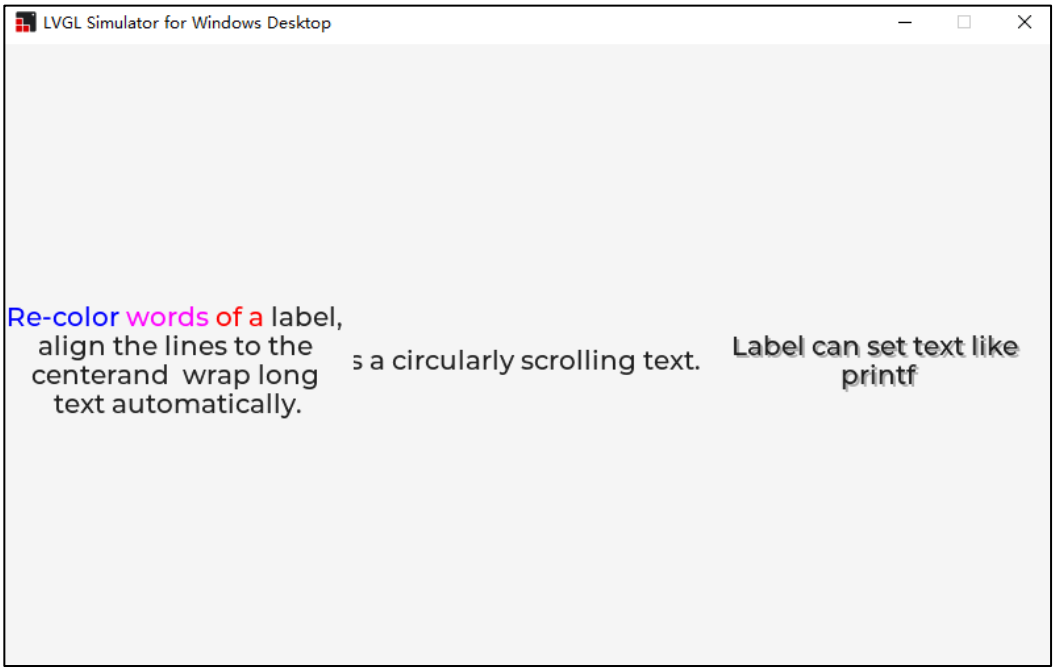


图 18.4.3.1 label 部件实验

第十九章 线部件(lv_line)

lv_line 线条简单来说是由多个点连接而成的对象,它可以通过多个接口来设置对象的大小、颜色、翻转 y 轴等属性。

本章分为如下几部分讲解：

19.1 线部件的组成

19.2 线部件的相关知识

19.3 线部件的 API 函数

19.4 线部件的实验

19.1 线部件的组成

在 LVGL 中，线部件是由 LV_PART_MAIN 典型的背景属性和线条样式属性组成。

19.2 线部件的相关知识

19.2.1 设置连接点

lv_line 线条是由多个点连接而成的对象，这些点使用 lv_point_t 类型的数组存储，并调用函数 lv_line_set_points 把该数据传递给线对象，最后由对象把点连接起来绘制成线条。

下面我们使用一个简单实例来讲解本小节的内容，如下源码所示：

```
void lv_mainstart(void)
{
    /* 为直线的点创建一个数组 */
    static lv_point_t line_points[] = { {5, 5}, {70, 70}, {120, 10},
                                        {180, 60}, {240, 10} };
    lv_obj_t* line1;
    /* 创建 line 部件 */
    line1 = lv_line_create(lv_scr_act());
    /* 设置线部件的点数组 */
    lv_line_set_points(line1, line_points, 5);
    /* 居中 */
    lv_obj_center(line1);
}
```

上述源码可知：我们首先创建点数组，然后创建一个 line 部件，最后该部件将点数组存储的点连接起来。如图 19.2.1.1 所示：

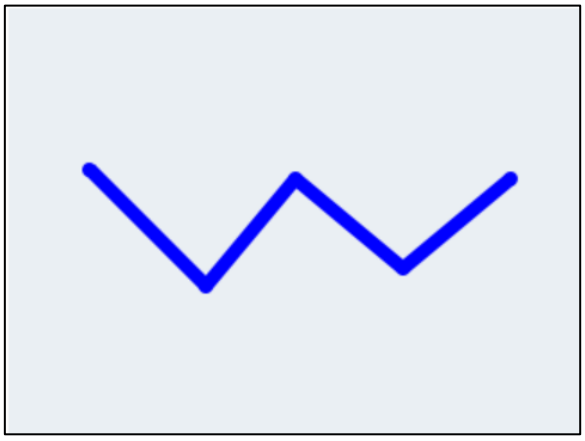


图 19.2.1.1 line 实验示意图

这里我们了解一下点数组的点到底如何绘画在指定的坐标地。上图中，因为 line 部件已经被居中对齐，所以点数组的点是以 line 部件居中后的左上方的点为原点坐标，如下图所示：

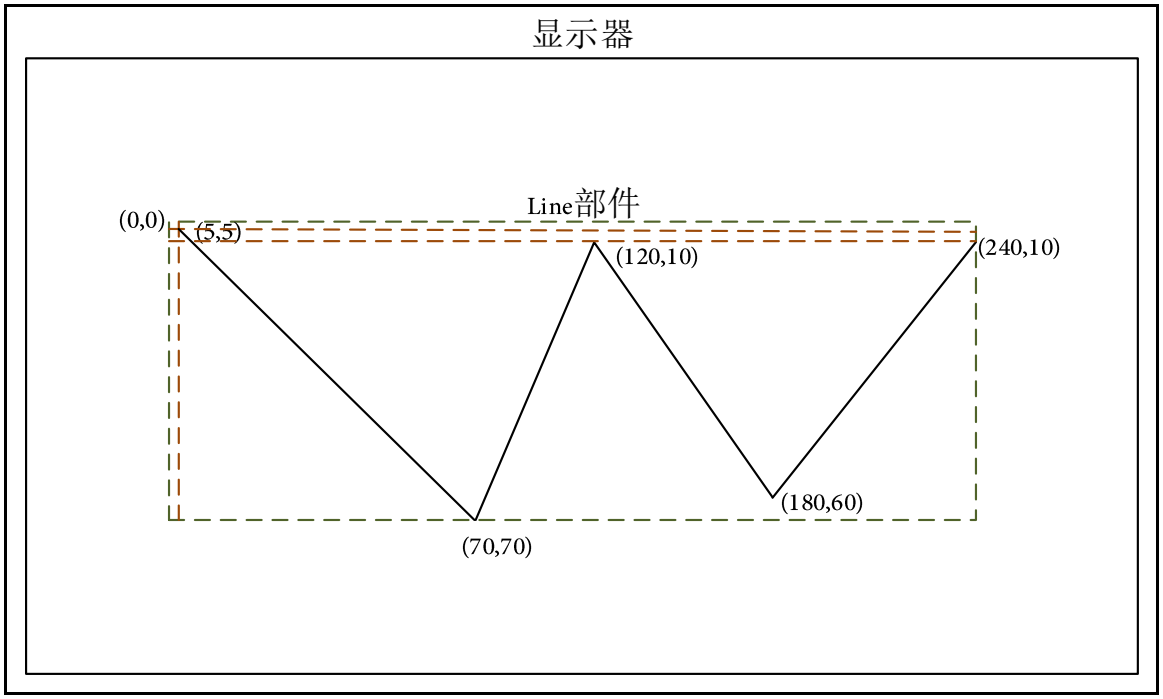


图 19.2.1.2 点数组的点绘制坐标示意图

从上图可知：点数组的点是根据 line 部件的位置而确认原点(0,0)，显然该数组并不是以显示屏的原点作为参考目标。

19.2.2 自适应大小

默认情况下，line 部件的宽度和高度都设置为 LV_SIZE_CONTENT。这意味着它将自动设置自身的大小以适应所有的点(请看上图 19.2.1.2 的线部件)。如果用户显式设置了尺寸，则线上的部分可能不可见。

19.2.3 倒 Y 操作

默认情况下，点数组的点是以 line 部件左上角为参考原点，而倒 Y 操作是把 line 部件的参考原点设置到左下角。line 部件倒 Y 操作可调用函数 lv_line_set_y_invert 启用，如图 19.2.3.1 所示：

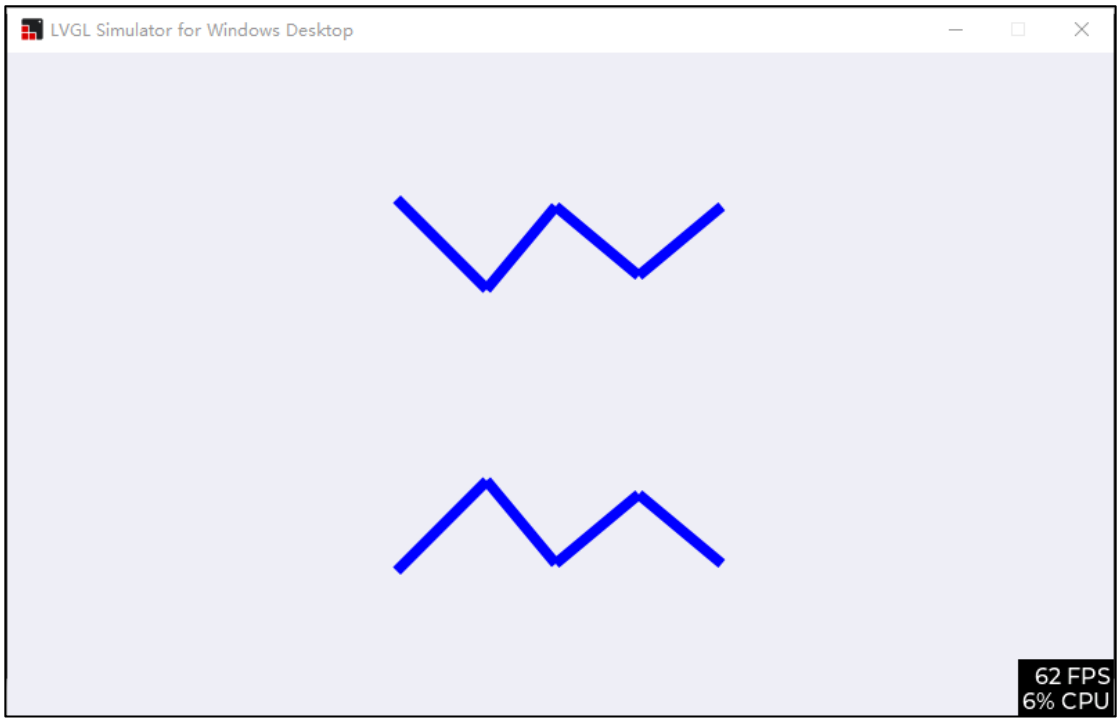


图 19.2.3.1 设置 line 部件倒 Y 形式

19.3 线部件的 API 函数

LVGL 提供给用户的 line 部件相关函数，如表 19.3.1 所示：

函数	描述
lv_line_create()	创建 line 部件
lv_line_set_points()	设置一个点数组。线对象将这些点连接起来。
lv_line_set_y_invert()	启用(或禁用)y 坐标反转
lv_line_get_y_invert()	获取 y 反转属性

表 19.3.1.1 line 部件 API 函数

接下来我们来讲解 LVGL 的 line 部件常用的函数有哪些：

1. lv_line_create 函数

该函数的作用是创建一个 line 部件，该函数原型如下所示：

```
lv_obj_t * lv_line_create(lv_obj_t * parent)
```

该函数的形参，如表 19.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 19.3.2 函数 lv_line_create()形参描述

返回值：无。

2. lv_line_set_points 函数

该函数的作用是设置一个点数组，该函数原型如下所示：

```
void lv_line_set_points( lv_obj_t *obj,
                         const lv_point_t points[],
                         uint16_t point_num)
```

该函数的形参，如表 19.3.3 所示：

参数	描述
obj	指向 line 对象的指针
points	一组点
point_num	点的数量

表 19.3.3 函数 lv_line_set_points()形参描述

返回值：无。

3. lv_line_set_y_invert 函数

该函数的作用是启用(或禁用)y 坐标反转，该函数原型如下所示：

```
void lv_line_set_y_invert(lv_obj_t *obj, bool en)
```

该函数的形参，如表 19.3.4 所示：

参数	描述
obj	指向 line 对象的指针
en	True:开启 y 倒转，false:关闭 y 倒转

表 19.3.4 函数 lv_line_set_y_invert()形参描述

返回值：无。

4. lv_line_get_y_invert 函数

该函数的作用是获取 y 反转属性，该函数原型如下所示：

```
bool lv_line_get_y_invert(const lv_obj_t *obj)
```

该函数的形参，如表 19.3.5 所示：

参数	描述
obj	指向 line 对象的指针

表 19.3.5 函数 lv_line_get_y_invert()形参描述

返回值：True:开启 y 反转，false:关闭 y 反转。

19.4 线部件的实验

19.4.1 硬件设计

19.4.1.1 硬件设计

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建了 1 个 line 部件，这个 line 部件的坐标是根据用户触摸来设定。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 19 lv_line(线)》。

19.4.2 软件设计

19.4.2.1 程序流程图

本实验的程序流程图，如下图 19.4.2.1 所示：

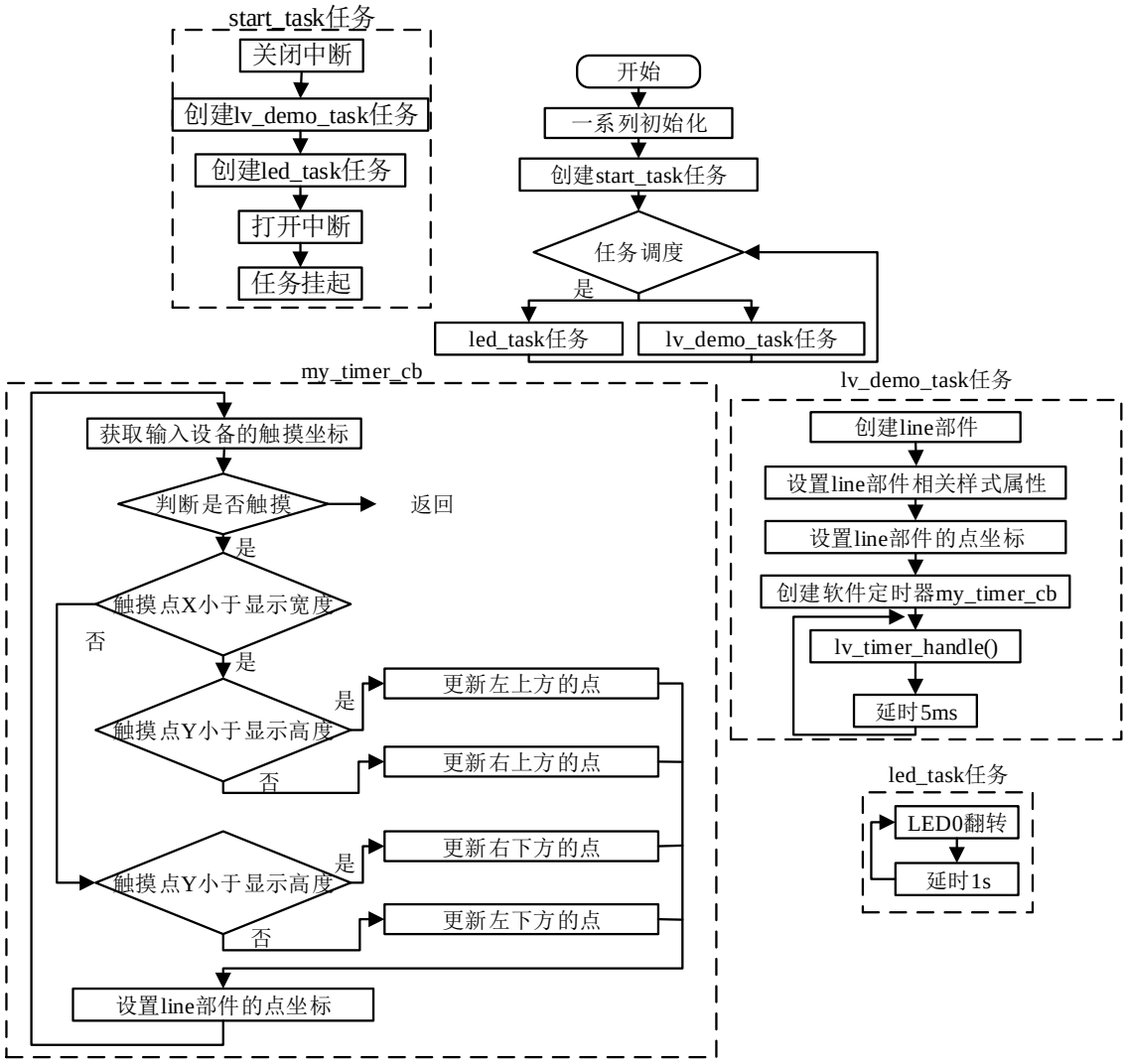


图 19.4.2.1 line 部件实验流程图

19.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_line(void)
{
    /* 根据屏幕大小设置点的位置 */
```

```
    line_points[0].x = scr_act_width()  / 4;
    line_points[0].y = scr_act_height() / 4;
    line_points[1].x = scr_act_width()  / 4;
    line_points[1].y = scr_act_height() / 4 * 3;
    line_points[2].x = scr_act_width()  / 4 * 3;
    line_points[2].y = scr_act_height() / 4 * 3;
    line_points[3].x = scr_act_width()  / 4 * 3;
    line_points[3].y = scr_act_height() / 4;
    line_points[4].x = scr_act_width()  / 4;
    line_points[4].y = scr_act_height() / 4;
    /* 定义并创建线 */
    lv_obj_t* line = lv_line_create(lv_scr_act());
    /* 设置线上点 */
    lv_line_set_points(line, line_points, sizeof(line_points)
                       / sizeof(lv_point_t));
    /* 设置线的宽度 */
    lv_obj_set_style_line_width(line, 8, LV_PART_MAIN);
    /* 设置线的颜色 */
    lv_obj_set_style_line_color(line, lv_palette_main(LV_PALETTE_BLUE),
                            LV_PART_MAIN);
    /* 设置拐点圆角 */
    lv_obj_set_style_line_rounded(line, true, LV_PART_MAIN);
    /* 创建定时任务 */
    lv_timer_create(my_timer_cb, 10, line);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_line();
}
```

上述源码可知：首先我们设置 line_points 数组的点坐标，这些点坐标是根据 LCD 显示屏高度与宽度设置，然后设置 line 部件的相关属性，最后创建 10ms 定时器，该定时器回调函数如下所示：

```
static void my_timer_cb(lv_timer_t* timer)
{
    lv_obj_t* line = (lv_obj_t*)timer->user_data;
    lv_point_t point;
    /* 获取输入设备坐标点 */
```

```
    lv_indev_get_point(indev_touchpad, &point);
    /* 如果没有触摸，则返回 */
    if (NULL == point.x && NULL == point.y)
        return;
    /* 判断触摸 x 点坐标是否小于屏幕宽度 */
    if (point.x < scr_act_width() >> 1) {
    /* 判断触摸 y 点坐标是否小于屏幕高度 */
        if (point.y < scr_act_height() >> 1) {
            /* 更新左上角坐标 */
            line_points[0].x = point.x;
            line_points[0].y = point.y;
            line_points[4].x = point.x;
            line_points[4].y = point.y;
        }
        else {
            /* 更新右上角坐标 */
            line_points[1].x = point.x;
            line_points[1].y = point.y;
        }
    }
    else {/* 判断触摸 x 点坐标是大于屏幕宽度 */
        /* 判断触摸 y 点坐标是否小于屏幕高度 */
        if (point.y < scr_act_height() >> 1) {
            /* 更新右下角坐标 */
            line_points[3].x = point.x;
            line_points[3].y = point.y;
        }
        else {
            /* 更新左下角坐标 */
            line_points[2].x = point.x;
            line_points[2].y = point.y;
        }
    }
    /* 更新 line 部件点坐标 */
    lv_line_set_points(line, line_points, sizeof(line_points)
                       / sizeof(lv_point_t));
}
```

从上述可知：首先我们调用函数 lv_indev_get_point 获取输入设备的坐标值，然后根据坐标值判断触摸方向，最后根据触摸方向更新点数组的坐标点，如果触摸方向是屏幕的左上角，则更新 line_points[0].x 和 line_points[0].y 的数值。

19.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 19.4.3.1 所示：

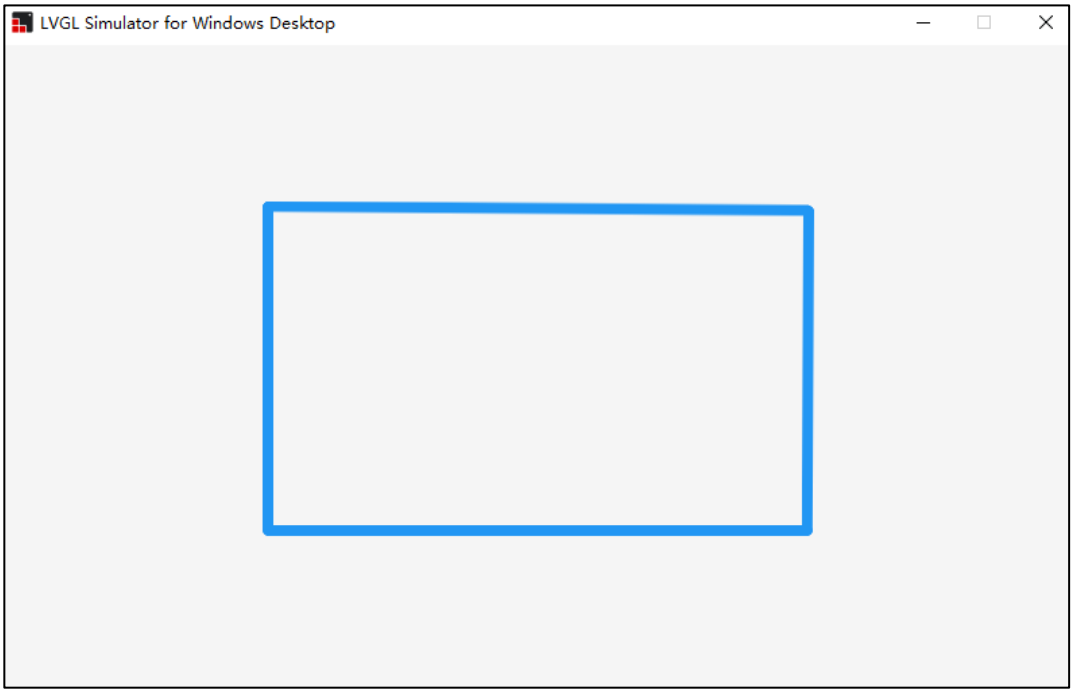


图 19.4.3.1 line 部件实验

上述图中为程序开启默认主界面，当我们触摸屏幕左上方时，点数组会根据触摸方向更新数据，如图 19.4.3.2 所示：

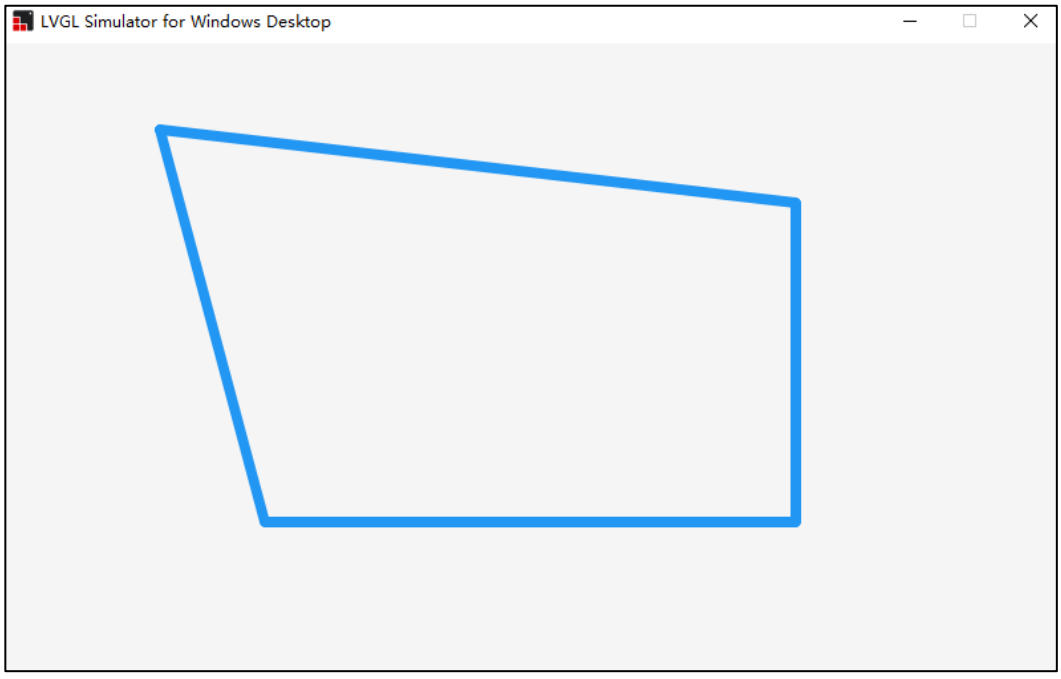


图 19.4.3.2 更新点数组数据

第二十章 滚筒部件(lv_roller)

lv_roller 是一个滚轮部件，它和我们前面学习的 lv_ddlist 下拉列表框部件非常的相似，它们最大的区别在于 lv_ddlist 是以展开和收缩的方式来选择选项值，而 lv_roller 是以滚动的方式来选择选项值。在 UI 中，该部件通常用于设置时间/日期等作用。

本章分为如下几部分讲解：

20.1 滚筒部件的组成

20.2 滚筒部件的相关知识

20.3 滚筒部件的 API 函数

20.4 滚筒部件的实验

20.1 滚筒部件的组成

滚筒部件是由 LV_PART_MAIN 和 LV_PART_SELECTED 两部分构建组成。LV_PART_MAIN 部分主要设置 Roller 部件背景属性和文本样式属性(文本的颜色、文本的字体和文本透明度样式等属性)，而 LV_PART_SELECTED 部分主要设置文本样式属性(选择区域内文本的外观)。下面笔者使用一个示意图来讲解滚筒部件的基础部分，如图 20.1.1 所示：

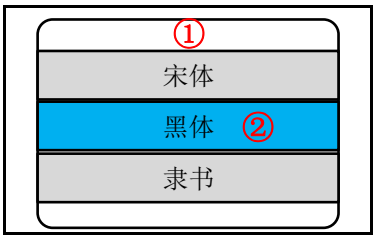


图 20.1.1 滚筒部件的基础部分

上图中的①是由 LV_PART_MAIN 基础部分来设置样式属性，而上图中的②是由 LV_PART_SELECTED 基础部分来设置样式属性的。

接下来我们使用简单的实例来讲解 Roller 部件的样式属性使用，如下源码所示：

```
/* 滚轮选项 */
static const char* roller_options = "option 0\n"
                                    "option 1\n"
                                    "option 2\n"
                                    "option 3\n"
                                    "option 4\n"
                                    "option 5\n"
                                    "option 6\n"
                                    "option 7";
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
```

```
{
    /* 定义并创建滚轮 */
    lv_obj_t* roller = lv_roller_create(lv_scr_act());
    /* 设置滚轮宽度 */
    lv_obj_set_width(roller, 110);
    /* 设置滚轮字体样式属性 */
    lv_obj_set_style_text_font(roller, &lv_font_montserrat_20, LV_PART_MAIN);
    /* 设置选择滚轮文本区域样式属性 */
    lv_obj_set_style_text_letter_space(roller, 5, LV_PART_SELECTED);
}
```

从上述源码可知：首先我们创建 Roller 部件，然后调用函数 lv_obj_set_style_text_font 设置 Roller 部件的文本字体，由于该函数的形参设置为 LV_PART_MAIN，所以 Roller 部件的文本的字体都是 20 号字体，最后调用函数 lv_obj_set_style_text_letter_space 设置文本字符的空间为 5，由于该函数的形参设置为 LV_PART_SELECTED，所以 Roller 部件选择的文本字符的间距为 5，如图 20.1.1 所示：

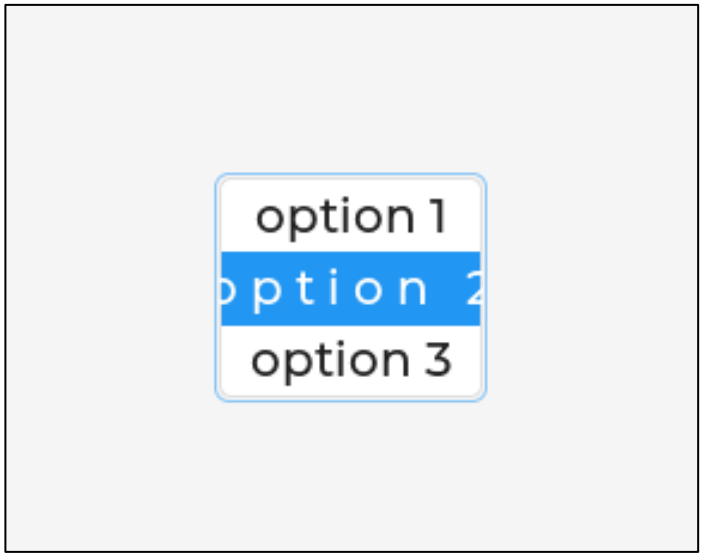


图 20.1.1 设置 Roller 部件样式属性示意图

从上图可知：选择 Roller 部件某个选项时，该选项的文本字符间距为 5，而选项的文本都是 20 号字体。

总结：

LV_PART_MAIN：对 Roller 部件全部选项及背景的属性。

LV_PART_SELECTED：对 Roller 部件选择的选项属性。

20.2 滚筒部件的相关知识

20.2.1 添加选项及滚筒模式

1. 添加选项

默认情况下， Roller 部件创建时并不具备任何选项，而这些选项必须由用户来提供。添加这些选项可调用函数 lv_roller_set_options，该函数具有三个形参，第一个形参表示指向 Roller 部件对象；第二个形参表示要加入的选项；第三个形参表示 Roller 部件模式。

2. 滚筒模式：

滚筒部件具有两种设置模式，不同的模式可改变滚筒的显示功能，接下来我们讲解 Roller 部件模式有哪些，如下所示：

(1) LV_ROLLER_MODE_INFINITE：设置 Roller 部件滚轮为正常模式圆形。

(2) LV_ROLLER_MODE_NORMAL：设置 Roller 部件滚轮为正常模式。

上述模式表示 Roller 部件临界区的状态。如果 Roller 部件设置模式为 LV_ROLLER_MODE_INFINITE，则 Roller 部件的最后一个和第一个选项形成环形，简单来讲：滚动到最后一个选项时，可从该选项滚动到第一个选项；如果 Roller 部件设置模式为 LV_ROLLER_MODE_NORMAL，则 Roller 部件的最后一个和第一个选项不能切换，简单来讲：滚动到最后一个选项时，只能向下滚动切换选项，而不能往上滚动切换到最后一个选项。如图 20.2.1.1 所示：

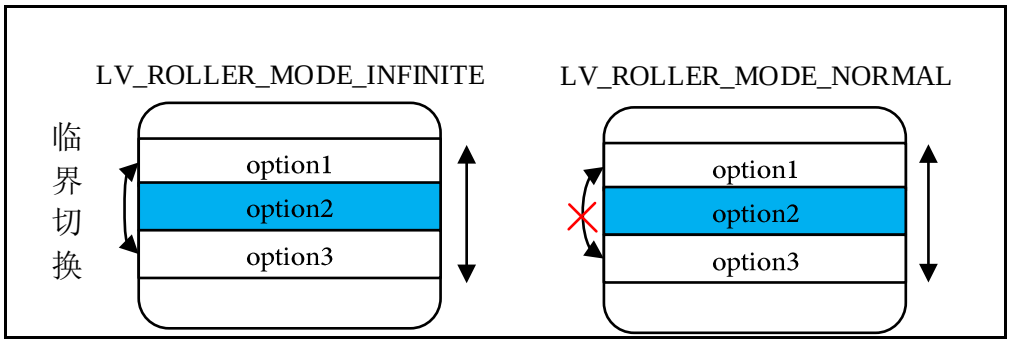


图 20.2.1.1 Roller 部件模式示意图

默认情况下，Roller 部件显示的选项为第一个添加的选项，如果用户想设置某个选项作为第一个显示项，则可调用函数 lv_roller_set_selected 手动设置。

20.2.2 获取选项文本

如果用户想获取获取当前选定选项的索引，可调用函数 lv_roller_get_selected 获取；如果用户想获取当前选定选项的文本，可调用函数 lv_roller_get_selected_str 获取。

20.2.3 选项可见数量

选项可见数量是指 Roller 部件的选项可显示数量。默认情况下，Roller 部件默认只显示一个选项，用户可调用函数 lv_roller_set_visible_row_count 设置滚轮可见选项个数。接下来我们以简单实例来讲解本小节的知识。

```
/* 滚轮选项 */
static const char* roller_options = "option 0\n"
                                    "option 1\n"
                                    "option 2\n"
                                    "option 3\n"
                                    "option 4\n"
                                    "option 5\n"
                                    "option 6\n"
```

```
                                    "option 7";
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    /* 定义并创建滚轮 */
    lv_obj_t* roller = lv_roller_create(lv_scr_act());
    /* 滚轮添加选项并设置无限模式 */
    lv_roller_set_options(roller, roller_options, LV_ROLLER_MODE_NORMAL);
    /* 设置滚轮宽度 */
    lv_obj_set_width(roller, 110);
    /* 设置滚轮可见选项个数 */
    lv_roller_set_visible_row_count(roller, 3);
}
```

上述源码可知：我们调用函数 lv_roller_set_visible_row_count 设置滚轮可见选项个数为 3，如图 20.2.3.1 所示：

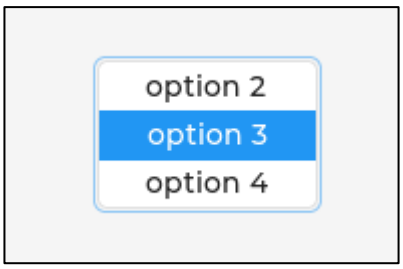


图 20.2.3.1 滚轮可见选项个数

上图中，Roller 部件可见选项个数为 3。

20.3 滚筒部件的 API 函数

LVGL 提供给用户的 Roller 部件相关函数，如表 20.3.1 所示：

函数	描述
lv_roller_create()	创建 roller 部件
lv_roller_set_options()	设置 roller 部件选项
lv_roller_set_selected()	设置所选选项
lv_roller_set_visible_row_count()	设置可见选项数量
lv_roller_get_selected()	获取所选选项 ID
lv_roller_get_selected_str()	获取所选选项文本
lv_roller_get_options()	获取一个滚轮的选择
lv_roller_get_option_cnt()	获取获取选项的总数

表 20.3.1.1 roller 部件 API 函数

接下来我们来讲解 LVGL 的 Roller 部件常用的函数有哪些：

1. lv_roller_create 函数

该函数的作用是创建一个 roller 部件，该函数原型如下所示：

```
lv_obj_t * lv_roller_create(lv_obj_t * parent)
```

该函数的形参，如表 20.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 20.3.2 函数 lv_roller_create()形参描述

返回值：返回 roller 部件对象。

2. lv_roller_set_options 函数

该函数的作用是设置滚轮上的选项，该函数原型如下所示：

```
void lv_roller_set_options(lv_obj_t *obj,
                           const char *options,
                           lv_roller_mode_t mode)
```

该函数的形参，如表 20.3.3 所示：

参数	描述
obj	指向对象的指针
options	添加的选项，E.g. "One\nTwo\nThree"
mode	LV_ROLLER_MODE_NORMAL 和 LV_ROLLER_MODE_INFINITE

表 20.3.3 函数 lv_roller_set_options()形参描述

返回值：无。

3. lv_roller_set_selected 函数

该函数的作用是设置所选的选项，该函数原型如下所示：

```
void lv_roller_set_selected(lv_obj_t *obj,
                            uint16_t sel_opt,
                            lv_anim_enable_t anim)
```

该函数的形参，如表 20.3.4 所示：

参数	描述
obj	指向对象的指针
sel_opt	选定选项的索引(0…选项数- 1)
anim	LV_ANIM_ON：设置动画；LV_ANOM_OFF：无

表 20.3.4 函数 lv_roller_set_selected()形参描述

返回值：无。

4. lv_roller_set_visible_row_count 函数

该函数的作用是获取所选选项的索引，该函数原型如下所示：

```
void lv_roller_set_visible_row_count(lv_obj_t *obj, uint8_t row_cnt)
```

该函数的形参，如表 20.3.5 所示：

参数	描述
obj	指向对象的指针
row_cnt	所需的可见行数

表 20.3.5 函数 lv_roller_set_visible_row_count()形参描述

返回值：无。

5. lv_roller_get_selected 函数

该函数的作用是获取所选选项的索引，该函数原型如下所示：

```
uint16_t lv_roller_get_selected(const lv_obj_t *obj)
```

该函数的形参，如表 20.3.6 所示：

参数	描述
obj	指向对象的指针

表 20.3.6 函数 lv_roller_get_selected()形参描述

返回值：无。

6. lv_roller_get_selected_str 函数

该函数的作用是以字符串形式获取当前选定的选项，该函数原型如下所示：

```
void lv_roller_get_selected_str(const lv_obj_t *obj,
                                char *buf,
                                uint32_t buf_size)
```

该函数的形参，如表 20.3.7 所示：

参数	描述
obj	指向对象的指针
buf	指向存储字符串的数组的指针
buf_size	buf 的大小，以字节为单位。0:表示忽略。

表 20.3.7 函数 lv_roller_get_selected_str()形参描述

返回值：无。

7. lv_roller_get_options 函数

该函数的作用是获取一个滚轮的选择，该函数原型如下所示：

```
const char *lv_roller_get_options(const lv_obj_t *obj)
```

该函数的形参，如表 20.3.8 所示：

参数	描述
obj	指向对象的指针

表 20.3.8 函数 lv_roller_get_options()形参描述

返回值：无。

20.4 滚筒部件的实验

20.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数由两个 roller 实验构成，第一个 roller 实验主要使用 btn 部件来控制 roller 部件的滚动方向，而第二个 roller 实验主要设置可见选项为 3 个。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 20 lv_roller(滚轮)》。

20.4.2 程序设计

20.4.2.1 程序流程图

本实验的程序流程图，如下图 20.4.2.1 所示：

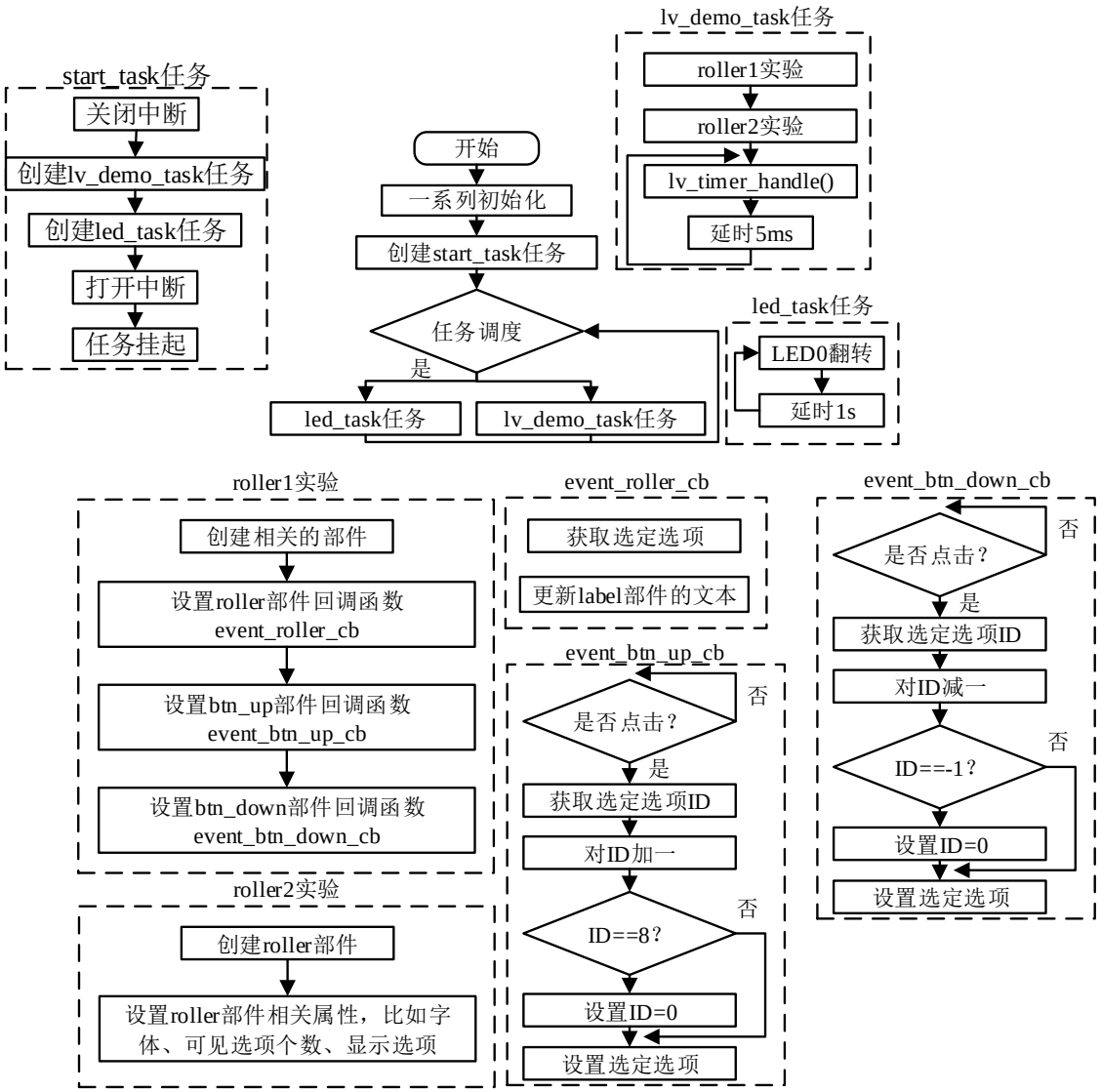


图 20.4.2.1 roller 部件实验流程图

20.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例 1
 * @param  无
 * @return 无
 */
static void lv_example_roller_1(void)
{
    /* 定义并创建滚轮 */
    lv_obj_t* roller = lv_roller_create(lv_scr_act());
    /* 滚轮添加选项并设置无限模式 */
    lv_roller_set_options(roller, roller_options, LV_ROLLER_MODE_INFINITE);
```

```
    /* 设置滚轮可见选项个数 */
    lv_roller_set_visible_row_count(roller, 4);
    /* 设置滚轮位置 */
    lv_obj_align(roller, LV_ALIGN_CENTER, -scr_act_width() / 4, 0);
    /* 设置滚轮宽度 */
    lv_obj_set_width(roller, 80);
    /* 定义并创建标签 */
    lv_obj_t* label_roller = lv_label_create(lv_scr_act());
    /* 设置标签宽度 */
    lv_obj_set_width(label_roller, 80);
    /* 设置标签位置 */
    lv_obj_align_to(label_roller, roller, LV_ALIGN_OUT_TOP_MID, 0,
                    -scr_act_height() / 10);
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label_roller, LV_TEXT_ALIGN_CENTER,
                            LV_PART_MAIN);
    /* 设置标签显示文本 */
    lv_label_set_text(label_roller, "option 0");
    /* 定义并创建按钮 */
    lv_obj_t* btn_up = lv_btn_create(lv_scr_act());
    /* 设置按钮宽度 */
    lv_obj_set_width(btn_up, 80);
    /* 设置按钮位置 */
    lv_obj_align_to(btn_up, roller, LV_ALIGN_OUT_BOTTOM_MID, 0,
                scr_act_height() / 20);
    /* 定义并创建按钮标签 */
    lv_obj_t* label_btn_up = lv_label_create(btn_up);
    /* 设置按钮标签位置 */
    lv_obj_center(label_btn_up);
    /* 设置按钮标签文本 */
    lv_label_set_text(label_btn_up, LV_SYMBOL_UP);
    /* 定义并创建按钮 */
    lv_obj_t* btn_down = lv_btn_create(lv_scr_act());
    /* 设置按钮宽度 */
    lv_obj_set_width(btn_down, 80);
    /* 设置按钮位置 */
    lv_obj_align_to(btn_down, btn_up, LV_ALIGN_OUT_BOTTOM_MID, 0,
                scr_act_height() / 40);
    /* 定义并创建按钮标签 */
    lv_obj_t* label_btn_down = lv_label_create(btn_down);
    /* 设置按钮标签位置 */
    lv_obj_center(label_btn_down);
    /* 设置按钮标签文本 */
```

```
    lv_label_set_text(label_btn_down, LV_SYMBOL_DOWN);
    /* 添加滚轮事件回调 */
    lv_obj_add_event_cb(roller, event_roller_cb, LV_EVENT_ALL, label_roller);
    /* 添加按钮回调 */
    lv_obj_add_event_cb(btn_up, event_btn_up_cb, LV_EVENT_CLICKED, roller);
    /* 添加按钮回调 */
    lv_obj_add_event_cb(btn_down, event_btn_down_cb, LV_EVENT_CLICKED, roller);
}
/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_roller_2(void)
{
    /* 定义并创建滚轮 */
    lv_obj_t* roller = lv_roller_create(lv_scr_act());
    /* 滚轮添加选项并设置无限模式 */
    lv_roller_set_options(roller, roller_options, LV_ROLLER_MODE_NORMAL);
    /* 设置滚轮位置 */
    lv_obj_align(roller, LV_ALIGN_CENTER, scr_act_width() / 4, 0);
    /* 设置滚轮宽度 */
    lv_obj_set_width(roller, 110);
    /* 设置滚轮字体 */
    lv_obj_set_style_text_font(roller, &lv_font_montserrat_20, LV_PART_MAIN);
    /* 设置滚轮字体 */
    lv_obj_set_style_text_letter_space(roller, 5, LV_PART_SELECTED);
    /* 设置滚轮可见选项个数 */
    lv_roller_set_visible_row_count(roller, 3);
    /* 设置滚轮显示的选项 */
    lv_roller_set_selected(roller, 3, LV_ANIM_OFF);
}
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_roller_1();
    lv_example_roller_2();
}
```

上述源码可知：本实验具有两个小实验，第一个实验主要讲解 btn 部件来控制 roller 部件滚动，而第二个实验主要讲解 roller 部件相关属性的设置。

20.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 20.4.3.1 所示：

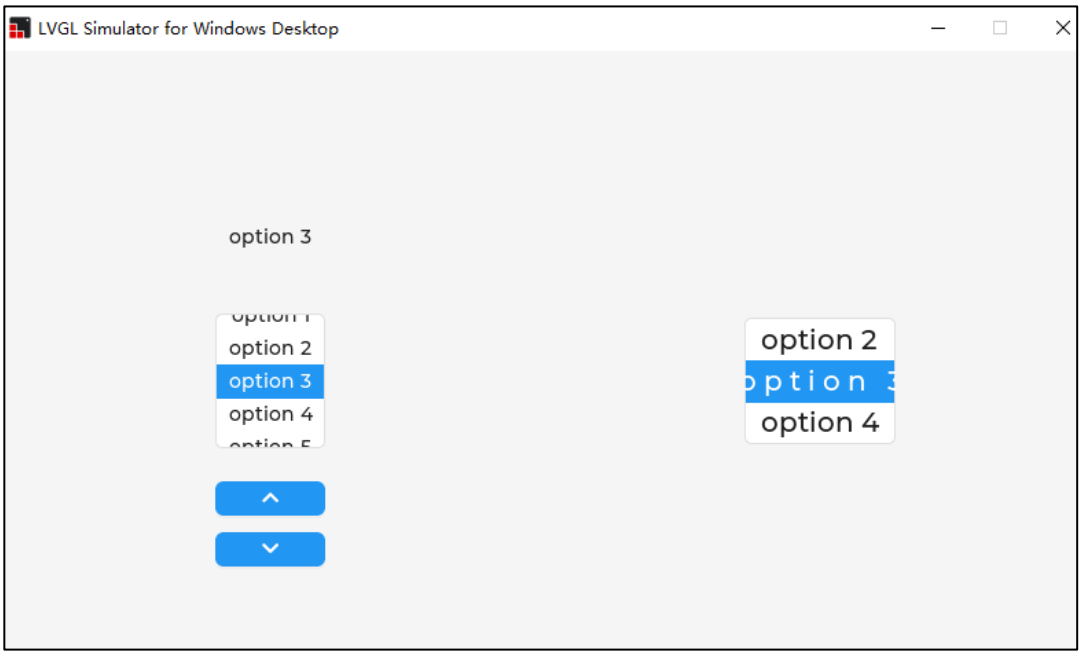


图 20.4.3.1 roller 部件实验

上图可知：当按下两个 btn 按键时，可使 roller 部件滚动。

第二十一章 滑块控件(lv_slider)

lv_slider 部件是由 lv_bar 进度条对象外加一个类似于旋钮的东西构建而成的，这个旋钮可以被拖拽来设置 lv_slider 部件的值，和 lv_bar 进度条一样，lv_slider 也可以被设置成是水平或者是垂直放置，不仅如此，lv_bar 对象上的大部分特性在 lv_slider 对象上基本都能找到，比如设置进度值、动画时间、设置最小最大范围等功能。

本章分为如下几部分讲解：

21.1 滑块部件的组成

21.2 滑块部件的相关知识

21.3 滑块部件的 API 函数

21.4 滑块部件的实验

21.1 滑块部件的组成

滑块部件是由 LV_PART_MAIN、LV_PART_INDICATOR、LV_PART_KNOB 三部分构建组成。LV_PART_MAIN 部分主要设置滑块部件背景属性， LV_PART_INDICATOR 部分主要设置指示器的背景属性，而 LV_PART_KNOB 部分主要设置旋钮的背景属性。

根据上面对滑块部件描述，我们可得到下面的示意图，如图 21.1.1 所示：

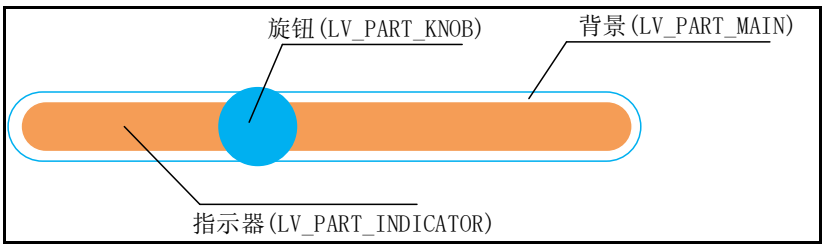


图 21.1.1 滑块部件组成部分

从上图可知：用户可根据滑块部件组成部分来设置相关样式属性。

21.2 滑块部件的相关知识

21.2.1 设置滑块部件的数值范围

默认情况下，创建滑块部件时，它的数值范围是 0~100，而它的当前值从 0 开始。如果用户指定滑块部件的某数值范围，可调用函数 lv_slider_set_range 设置，该函数的第二和第三形参分别表示最小值和最大值；如果用户指定滑块部件初始值，可调用函数 lv_slider_set_value 设置，该函数的第二和第三形参分别表示设定当前值和动画属性。

接下来我们以简单实例来讲解这两个函数的使用，如下所示：

```
void lv_mainstart(void)
{
    /* 创建 slider 部件 */
    lv_obj_t* slider = lv_slider_create(lv_scr_act());
    /* 设置 slider 部件范围 */
    lv_slider_set_range(slider,0,255);
    /* 设置 slider 部件当前值 */
```

```
    lv_slider_set_value(slider,100, LV_ANIM_ON);
    /* 设置 slider 部件居中 */
    lv_obj_center(slider);
}
```

从上述可知：首先笔者调用函数 lv_slider_create 创建滑块部件，其次调用函数 lv_slider_set_range 设置滑块部件的数值范围，最后调用函数 lv_slider_set_value 设置滑块部件的当前值。上述源码可在 PC 模拟器或者在开发板上运行，其效果图如图 21.2.1.1 所示：

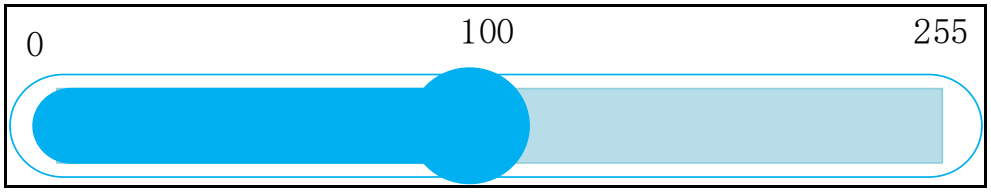


图 21.2.1.1 slider 部件示意图

从上图可知：笔者设置 slider 部件的数值范围为 0~255，而设置 slider 部件的当前数值为 100。

21.2.2 设置滑块部件的模式

除了普通模式外，滑块部件还可以配置为两种额外模式，这些模式如下所示：

(1) LV_SLIDER_MODE_SYMMETRICAL：将指示器从零绘制到当前值，该滑块部件需要设置负的最小范围和正的最大范围。

(2) LV_SLIDER_MODE_RANGE：系统允许调用 lv_bar_set_start_value 函数设置起始值，该起始值必须小于结束值。

滑块部件设置相关模式可通过函数定义：

```
lv_slider_set_mode(slider, LV_SLIDER_TYPE_…)/* 设置 slider 控件的模式*/
```

接下来我们分别地介绍上述额外模式的使用，如以下所示：

```
void lv_mainstart(void)
{
    /* 创建 slider 部件 */
    lv_obj_t* slider = lv_slider_create(lv_scr_act());
    /* 设置 slider 部件模式 */
    lv_slider_set_mode(slider, LV_SLIDER_MODE_SYMMETRICAL);
    /* 设置 slider 部件数值范围 */
    lv_slider_set_range(slider,-100,100);
    lv_obj_center(slider);

    /* 创建 slider1 部件 */
    lv_obj_t* slider1 = lv_slider_create(lv_scr_act());
    /* 创建 slider1 部件模式 */
    lv_slider_set_mode(slider1, LV_SLIDER_MODE_RANGE);
    /* 创建 slider1 部件起始值 */
    lv_bar_set_start_value(slider1,0,LV_ANIM_OFF);
    /* 创建 slider1 部件数值范围 */
    lv_slider_set_range(slider1, -100, 100);
```

```
    lv_obj_align_to(slider1, slider,LV_ALIGN_OUT_BOTTOM_MID,0,30);
}
```

从上述源码可知：首先我们创建了两个滑块部件分别为 slider 对象和 slider1 对象，然后这两个部件分别设置 LV_SLIDER_MODE_SYMMETRICAL 和 LV_SLIDER_MODE_RANGE 模式，其次 slider1 对象设置起始值为 0，最后这两个部件的数值范围为-100~100。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 21.2.2.1 所示：

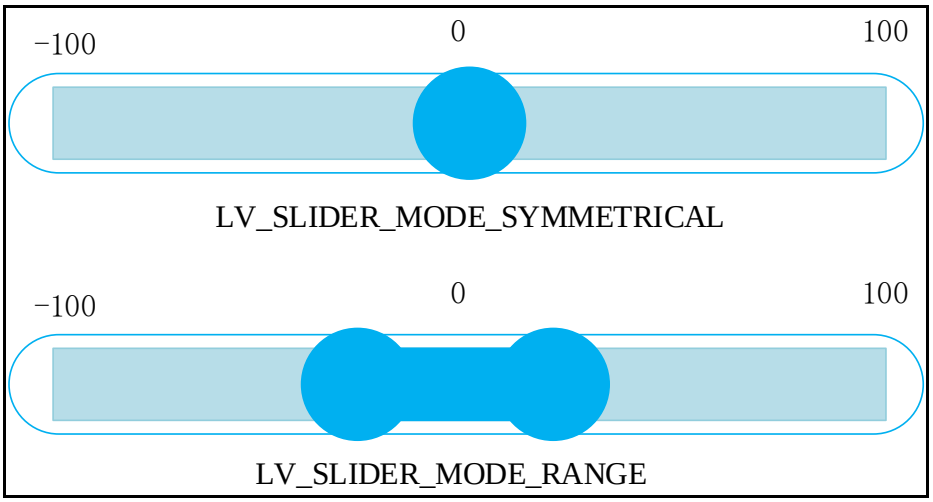


图 21.2.2.1 设置不同模式的 slider 部件

从上图可知：LV_SLIDER_MODE_SYMMETRICAL 模式的滑块部件是从 0 点开始绘制，可往左右滑动，而 LV_SLIDER_MODE_RANGE 模式的滑块部件可调用函数设置起始值，该部件可起始值两边可滑动。

21.2.3 Knob-only 模式

通常情况下，滑块部件可以通过拖动旋钮或者单击滑块来调整指示器位置。对于这两种方式，笔者更建议使用只在拖动旋钮时产生反应，因为点击滑块来调整指示器位置会导致数值以突变的方式增长，对于这个情景，拖动旋钮调整指示器位置方式更为妥当。

在默认情况下，滑块部件是支持两个调整指示器位置的方式，如果用户想设置滑块部件只支持一种拖动旋钮调整方式，可调用以下函数设置：

```
lv_obj_add_flag(slider, LV_OBJ_FLAG_ADV_HITTEST)/* 单击或者拖动滑块来调整滑块 */
```

21.3 滑块部件的 API 函数

LVGL 提供给用户的滑块部件相关函数，如表 21.3.1 所示：

函数	描述
lv_slider_create()	创建一个 slider 对象
lv_slider_set_value()	设置 slider 部件的当前值
lv_slider_set_left_value()	为 slider 部件的左边旋钮设置一个新值
lv_slider_set_range()	设置 slider 部件的数值范围
lv_slider_set_mode()	设置 slider 部件模式
lv_slider_get_value()	获取 slider 部件当前值
lv_slider_get_left_value()	获取 slider 部件的左边旋钮设置一个新值
lv_slider_get_min_value()	获取 slider 部件最小值
lv_slider_get_max_value()	获取 slider 部件最大值
lv_slider_is_dragged()	判断 slider 是否被拖动

lv_slider_get_mode()	获取 slider 部件模式

表 21.3.1.1 滑块部件 API 函数

接下来我们来讲解 LVGL 的滑块部件常用的函数有哪些：

1. lv_slider_create 函数

该函数的作用是创建一个 slider 部件，该函数原型如下所示：

```
lv_obj_t * lv_slider_create(lv_obj_t * parent)
```

该函数的形参，如表 21.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 21.3.2 函数 lv_slider_create()形参描述

返回值：返回 slider 部件对象。

2. lv_slider_set_value 函数

该函数的作用是设置 slider 部件的当前值，该函数原型如下所示：

```
static inline void lv_slider_set_value( lv_obj_t *obj,
int32_t value,
lv_anim_enable_t anim)
```

该函数的形参，如表 21.3.3 所示：

参数	描述
obj	指向 slider 对象的指针
value	设置 slider 当前值
anim	LV_ANIM_ON:用动画设置值，LV_ANIM_OFF:不使用动画

表 21.3.3 函数 lv_slider_set_value()形参描述

返回值：无。

3. lv_slider_set_range 函数

该函数的作用是设置 slider 部件的范围，该函数原型如下所示：

```
static inline void lv_slider_set_range(lv_obj_t *obj, int32_t min, int32_t max)
```

该函数的形参，如表 21.3.3 所示：

参数	描述
obj	指向 slider 对象的指针
min	设置 slider 最小值
max	设置 slider 最大值

表 21.3.3 函数 lv_slider_set_range()形参描述

返回值：无。

4. lv_slider_set_mode 函数

该函数的作用是设置 slider 的模式，该函数原型如下所示：

```
static inline void lv_slider_set_mode(lv_obj_t *obj, lv_slider_mode_t mode)
```

该函数的形参，如表 21.3.4 所示：

参数	描述
obj	指向 slider 对象的指针
mode	slider 的模式

表 21.3.4 函数 lv_slider_set_mode()形参描述

返回值：无。

21.4 滑块部件的实验

21.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数由三个 slider 实验构成，第一个 slider 实验主要显示滑块的当前值，当滑动滑块时，系统会把该实验的 label 部件显示滑块的数值，第二个 slider 实验主要设置它的样式属性，而第三个 slider 实验主要设置滑块的数值范围、模式和当前值属性，当滑动滑块时，系统会把该实验的 label 部件显示滑块的数值。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 21 lv_slider(滑杆)》。

21.4.2 软件设计

21.4.2.1 程序流程图

本实验的程序流程图，如下图 21.4.2.1 所示：

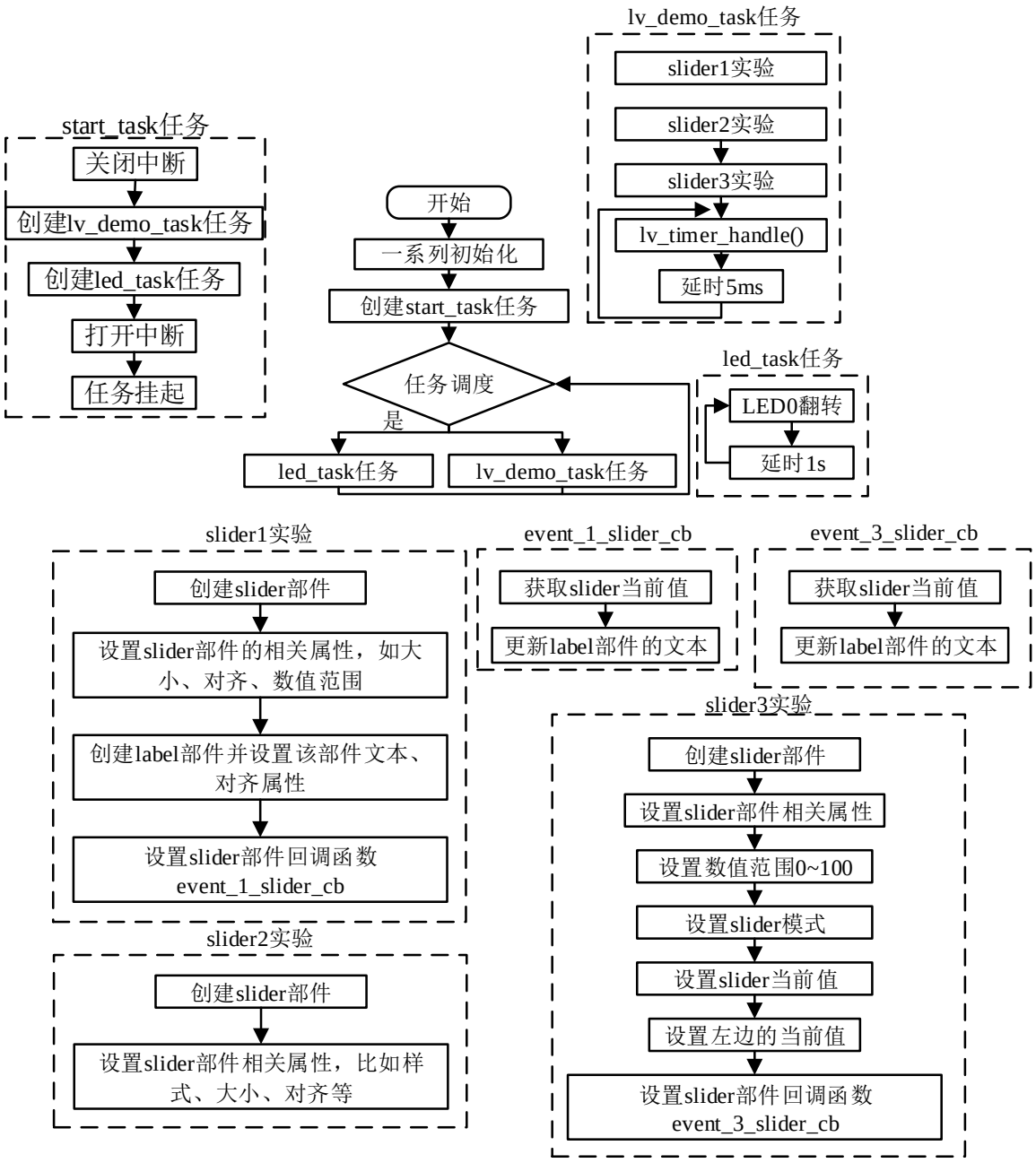


图 21.4.2.1 slider 部件实验流程图

21.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
```

```
{
    lv_example_slider_1();
    lv_example_slider_2();
    lv_example_slider_3();
}
```

此函数具有三个小实验，下面我们分别地介绍这些实验实现的功能。

(1) lv_example_slider_1 函数

```
/**
 * @brief  例 1 滑杆事件回调
 * @param  e：事件结构描述
 * @return 无
 */
static void event_1_slider_cb(lv_event_t* e)
{
    lv_obj_t* slider = lv_event_get_target(e);
    lv_obj_t* label = (lv_obj_t*)e->user_data;
    lv_event_code_t code = lv_event_get_code(e);

    if (LV_EVENT_VALUE_CHANGED == code) {
        lv_label_set_text_fmt(label, "%d%", lv_slider_get_value(slider));
    }
}

/**
 * @brief  例 1
 * @param  无
 * @return 无
 */
static void lv_example_slider_1(void)
{
    /* 定义并创建滑杆 */
    lv_obj_t* slider = lv_slider_create(lv_scr_act());
    /* 设置滑杆宽度 */
    lv_obj_set_width(slider, scr_act_width() / 4);
    /* 设置滑杆高度 */
    lv_obj_set_height(slider, scr_act_height() / 20);
    /* 设置滑杆位置 */
    lv_obj_align(slider, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);
    /* 设置滑杆范围 */
    lv_slider_set_range(slider, 0, 100);

    /* 定义并创建标签 */
    lv_obj_t* label = lv_label_create(lv_scr_act());
```

```
    /* 设置标签文本 */
    lv_label_set_text_fmt(label, "%d%", lv_slider_get_value(slider));
    /* 设置标签位置 */
    lv_obj_align_to(label, slider, LV_ALIGN_OUT_TOP_MID, 0,
                    -scr_act_height() / 15);

    /* 设置滑杆回调 */
    lv_obj_add_event_cb(slider, event_1_slider_cb,
    LV_EVENT_VALUE_CHANGED, label);
}
```

上述源码可知：此实验首先创建滑块部件，然后设置该部件相关属性，比如宽度，高度、对齐、滑块部件的数值范围等属性，其次创建 label 部件，该部件主要根据滑块当前值绘画文本，最后添加滑块部件的回调函数，该回调函数主要周期获取滑块部件的当前值并且刷新 label 部件的文本。

(2) lv_example_slider_2 函数

```
/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_slider_2(void)
{
    /* 创建特效 */
    static const lv_style_prop_t props[] = { LV_STYLE_BG_COLOR,
                                             LV_STYLE_PROP_INV};
    static lv_style_transition_dsc_t transition_dsc;
    lv_style_transition_dsc_init(&transition_dsc, props,
    lv_anim_path_linear, 300, 0, NULL);
    /* 创建风格 */
    static lv_style_t style_main;
    static lv_style_t style_indicator;
    static lv_style_t style_knob;
    static lv_style_t style_pressed_color;
    lv_style_init(&style_main);
    lv_style_set_bg_opa(&style_main, LV_OPA_COVER);
    lv_style_set_bg_color(&style_main, lv_color_hex3(0xbbb));
    lv_style_set_radius(&style_main, LV_RADIUS_CIRCLE);
    lv_style_set_pad_ver(&style_main, -2);

    lv_style_init(&style_indicator);
    lv_style_set_bg_opa(&style_indicator, LV_OPA_COVER);
    lv_style_set_bg_color(&style_indicator, lv_palette_main(LV_PALETTE_CYAN));
    lv_style_set_radius(&style_indicator, LV_RADIUS_CIRCLE);
```

```
    lv_style_set_transition(&style_indicator, &transition_dsc);

    lv_style_init(&style_knob);
    lv_style_set_bg_opa(&style_knob, LV_OPA_COVER);
    lv_style_set_bg_color(&style_knob, lv_palette_main(LV_PALETTE_CYAN));
    lv_style_set_border_color(&style_knob,
    lv_palette_darken(LV_PALETTE_CYAN, 3));
    lv_style_set_border_width(&style_knob, 2);
    lv_style_set_radius(&style_knob, LV_RADIUS_CIRCLE);
    lv_style_set_pad_all(&style_knob, 6);
    lv_style_set_transition(&style_knob, &transition_dsc);

    lv_style_init(&style_pressed_color);
    lv_style_set_bg_color(&style_pressed_color,
                          lv_palette_darken(LV_PALETTE_CYAN, 2));

    /* 定义并创建滑杆 */
    lv_obj_t* slider = lv_slider_create(lv_scr_act());
    /* 设置滑杆宽度 */
    lv_obj_set_width(slider, scr_act_height() / 20);
    /* 设置滑杆高度 */
    lv_obj_set_height(slider, scr_act_height() / 2);
    /* 设置滑杆位置 */
    lv_obj_center(slider);
    /* 添加滑杆风格 */
    lv_obj_add_style(slider, &style_main, LV_PART_MAIN);
    lv_obj_add_style(slider, &style_indicator, LV_PART_INDICATOR);
    lv_obj_add_style(slider, &style_pressed_color, LV_PART_INDICATOR
                     | LV_STATE_PRESSED);
    lv_obj_add_style(slider, &style_knob, LV_PART_KNOB);
    lv_obj_add_style(slider, &style_pressed_color, LV_PART_KNOB
                     | LV_STATE_PRESSED);
}
```

此实验主要设置滑块组成部分的相关属性，当按下旋钮时，该旋钮的背景颜色为蓝绿色；当滑动指示器时，该指示器的背景颜色为蓝绿色。

(3) lv_example_slider_3 函数

```
/**
 * @brief  例 3 滑杆事件回调
 * @param  e：事件结构描述
 * @return 无
 */
static void event_3_slider_cb(lv_event_t* e)
{
```

```
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* slider = lv_event_get_target(e);
    if (code == LV_EVENT_REFR_EXT_DRAW_SIZE) {
        lv_coord_t* size = (lv_coord_t*)lv_event_get_param(e);
        *size = LV_MAX(*size, 50);
    }
    else if (LV_EVENT_DRAW_PART_END == code) {
        lv_obj_draw_part_dsc_t* dsc = (lv_obj_draw_part_dsc_t*)
         lv_event_get_param(e);
        if (LV_PART_INDICATOR == dsc->part) {
            char buf[20];
            lv_snprintf(buf, sizeof(buf), "%d - %d",
                        lv_slider_get_left_value(slider),
            lv_slider_get_value(slider));

            lv_point_t label_size;
            lv_txt_get_size(&label_size, buf, LV_FONT_DEFAULT, 0, 0,
                            LV_COORD_MAX, LV_TEXT_FLAG_NONE);
            lv_area_t label_area;
            label_area.x1 = dsc->draw_area->x1 +
                            lv_area_get_width(dsc->draw_area) / 2
                            - label_size.x / 2;
            label_area.x2 = label_area.x1 + label_size.x;
            label_area.y2 = dsc->draw_area->y1 - 10;
            label_area.y1 = label_area.y2 - label_size.y;

            lv_draw_label_dsc_t label_draw_dsc;
            lv_draw_label_dsc_init(&label_draw_dsc);

            lv_draw_label(&label_area, dsc->clip_area,
                          &label_draw_dsc, buf, NULL);
        }
    }
}

/**
 * @brief  例 3
 * @param  无
 * @return 无
 */
static void lv_example_slider_3(void)
{
    /* 定义并创建滑杆 */
```

```
    lv_obj_t* slider = lv_slider_create(lv_scr_act());
    /* 设置滑杆宽度 */
    lv_obj_set_width(slider, scr_act_width() / 4);
    /* 设置滑杆高度 */
    lv_obj_set_height(slider, scr_act_height() / 20);
    /* 设置滑杆位置 */
    lv_obj_align(slider, LV_ALIGN_CENTER, scr_act_width() / 3, 0);
    /* 设置滑杆范围 */
    lv_slider_set_range(slider, 0, 100);
    /* 设置滑杆的模式 */
    lv_slider_set_mode(slider, LV_SLIDER_MODE_RANGE);
    /* 设置滑杆右侧的值 */
    lv_slider_set_value(slider, 70, LV_ANIM_OFF);
    /* 设置滑杆右侧的值 */
    lv_slider_set_left_value(slider, 20, LV_ANIM_OFF);
    /* 设置滑杆事件回调 */
    lv_obj_add_event_cb(slider, event_3_slider_cb, LV_EVENT_ALL, NULL);
    lv_obj_refresh_ext_draw_size(slider);
}
```

此实验首先创建一个滑块部件，然后设置该部件的相关属性，比如宽度、高度、位置、滑块部件数值范围和滑块模式等属性，最后添加滑块回调函数，该回调函数主要绘画滑块内部的文本内容，该文本内容时根据用户滑动的当前值相关。

21.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 21.4.3.1 所示：

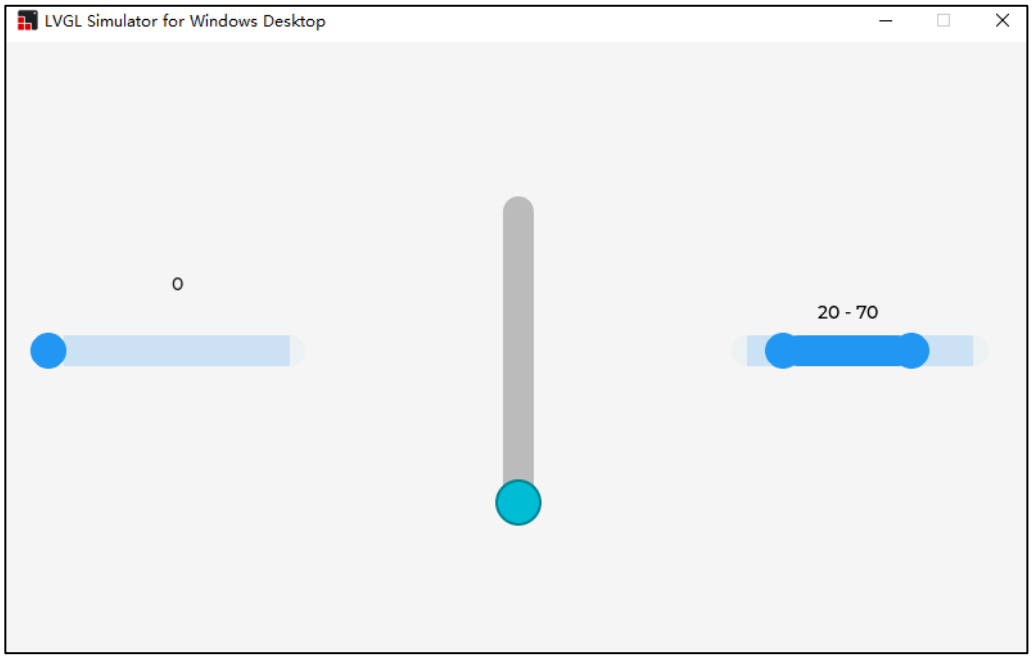


图 21.4.3.1 slider 部件实验

上图中，左边的滑块部件是由实验 1 绘画，中间的滑块部件是由实验 2 绘画，而右边的滑块部件是由实验 3 绘画。

第二十二章 开关控件(lv_switch)

switch 部件看起来像是一个缩小版的 slider 部件，很显然不是错觉，其实它的本质就是一个 slider 部件，只不过在 slider 部件上做一些小的升级，就变成了 switch 部件。

本章分为如下几部分讲解：

22.1 开关部件的组成

22.2 开关部件的相关知识

22.3 开关部件的 API 函数

22.4 开关部件的实验

22.1 开关部件的组成

前面笔者也讲解过，switch 部件是在 slider 部件上做了小升级，显然 switch 部件的组成部分与 slider 部件是相似的，如图 22.1.1 所示：

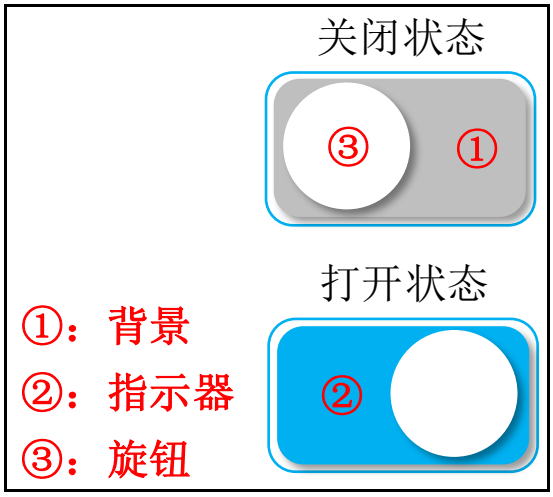


图 22.1.1 switch 部件组成部分

上图中的①表示 LV_PART_MAIN 基础部分；上图中的②表示 LV_PART_INDICATOR 基础部分；上图中的③表示 LV_PART_KNOB 基础部分。

从上图所示：switch 部件组成部分与 slider 部件相似，请大家可查看上一章节的内容。

关于开关部件的样式属性的设置，这里我们无需重复讲解了，前面已经在多个章节中提醒大家了。

22.2 开关部件的相关知识

22.2.1 获取开关部件状态

用户可使用函数 lv_obj_has_state(switch, LV_STATE_CHECKED)获取 switch 部件的当前状态，如果该函数返回为 true，则 switch 部件已打开。

22.2.2 添加开关部件的状态

在默认情况下，创建一个 switch 部件是可点击的，如果用户想设置该部件不可点击，可调用以下函数设置：

```
lv_obj_clear_state(switch, LV_STATE_CHECKED)
```

此函数主要清除 switch 部件的点击状态。如果用户想使不可点击的 switch 部件转换成可点击的部件，可调用以下函数设置：

```
lv_obj_add_state(switch, LV_STATE_CHECKED)
```

此函数主要添加 switch 部件的点击状态。

注意：一般设置 switch 部件的回调函数的事件类型是 LV_EVENT_VALUE_CHANGED。当然读者也可以设置其他的事件类型，比如点击、长按等事件类型。

22.3 开关部件的 API 函数

LVGL 提供用户 switch 部件相关函数只有一个，如下所示：

lv_switch_create 函数

该函数的作用是创建一个 switch 部件，该函数原型如下所示：

```
lv_obj_t * lv_switch_create(lv_obj_t * parent)
```

该函数的形参，如表 22.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 22.3.2 函数 lv_switch_create()形参描述

返回值：返回 switch 部件对象。

22.4 开关部件的实验

22.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建四个 switch 部件以及一个 led 部件，这四个 switch 部件位于显示屏右侧从上到下排列，第一和第二个 switch 部件设置为可点击，而第三和第四个 switch 设置为不可点击，当我们点击第一个 switch 部件时，LED 部件由关闭状态转成开启状态。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 22 lv_switch(开关)》。注意： MiniSTM32 不支持本实验。

22.4.2 软件设计

22.4.2.1 程序流程图

本实验的程序流程图，如下图 22.4.2.1 所示：

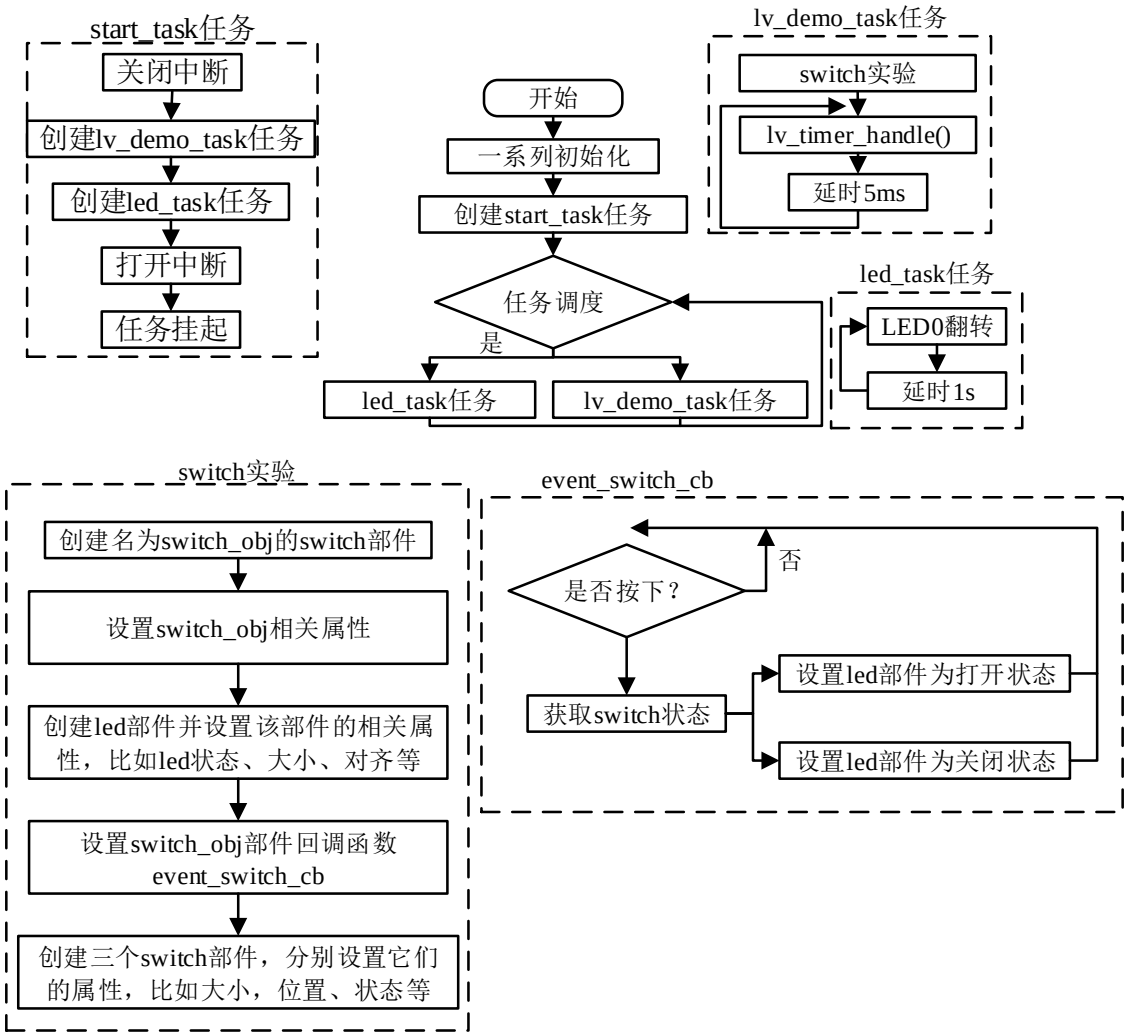


图 22.4.2.1 switch 部件实验流程图

22.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例 3 滑杆事件回调
 * @param  e：事件描述
 * @return 无
 */
static void event_switch_cb(lv_event_t* e)
{
    lv_obj_t* switch_obj = lv_event_get_target(e);
    lv_obj_t* led = (lv_obj_t*)e->user_data;
    lv_event_code_t code = lv_event_get_code(e);

    if (LV_EVENT_CLICKED == code) {
        lv_obj_has_state(switch_obj, LV_STATE_CHECKED) ?
```

```
lv_led_on(led) : lv_led_off(led);
    }
}

/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_switch(void)
{
    /* 定义并创建开关 */
    lv_obj_t* switch_obj = lv_switch_create(lv_scr_act());
    /* 设置开关宽度 */
    lv_obj_set_width(switch_obj, 90);
    /* 设置开关高度 */
    lv_obj_set_height(switch_obj, 45);
    /* 设置开关位置 */
    lv_obj_align(switch_obj, LV_ALIGN_CENTER, scr_act_width() / 4,
                 -scr_act_height() / 8 * 3);

    /* 定义并创建 LED */
    lv_obj_t* led = lv_led_create(lv_scr_act());
    /* 设置 LED 大小 */
    lv_obj_set_size(led, 100, 100);
    /* 设置 LED 位置 */
    lv_obj_align(led, LV_ALIGN_CENTER, -scr_act_width() / 4, 0);
    /* 设置 LED 状态 */
    lv_led_off(led);

    /* 设置开关回调函数 */
    lv_obj_add_event_cb(switch_obj, event_switch_cb, LV_EVENT_ALL, led);

    lv_obj_t* switch_temp;

    /* 定义并创建开关 */
    switch_temp = lv_switch_create(lv_scr_act());
    /* 设置开关位置 */
    lv_obj_align(switch_temp, LV_ALIGN_CENTER, scr_act_width() / 4,
                 -scr_act_height() / 8 * 1);
    /* 设置开关大小 */
    lv_obj_set_size(switch_temp, 90, 45);
    /* 设置为选中状态 */
```

```
    lv_obj_add_state(switch_temp, LV_STATE_CHECKED);

    /* 定义并创建开关 */
    switch_temp = lv_switch_create(lv_scr_act());
    /* 设置开关位置 */
    lv_obj_align(switch_temp, LV_ALIGN_CENTER, scr_act_width() / 4,
                 scr_act_height() / 8 * 1);
    /* 设置开关大小 */
    lv_obj_set_size(switch_temp, 90, 45);
    /* 设置为选中状态 */
    lv_obj_add_state(switch_temp, LV_STATE_DISABLED);

    /* 定义并创建开关 */
    switch_temp = lv_switch_create(lv_scr_act());
    /* 设置开关位置 */
    lv_obj_align(switch_temp, LV_ALIGN_CENTER, scr_act_width() / 4,
                 scr_act_height() / 8 * 3);
    /* 设置开关大小 */
    lv_obj_set_size(switch_temp, 90, 45);
    /* 设置为选中状态 */
    lv_obj_add_state(switch_temp, LV_STATE_CHECKED | LV_STATE_DISABLED);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_switch();
}
```

从上述源码可知：lv_mainstart 函数间接调用 lv_example_switch 函数，该函数首先创建了命名为 switch_obj 的 switch 部件，然后该部件设置了相关样式属性，其次创建 led 部件且设置相关的样式属性并添加 switch_obj 对象的 event_switch_cb 回调函数，该回调函数主要控制 led 部件的状态，最后创建三个 switch 部件，它们分别测试可点击与不可点击的功能。

22.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 22.4.3.1 所示：

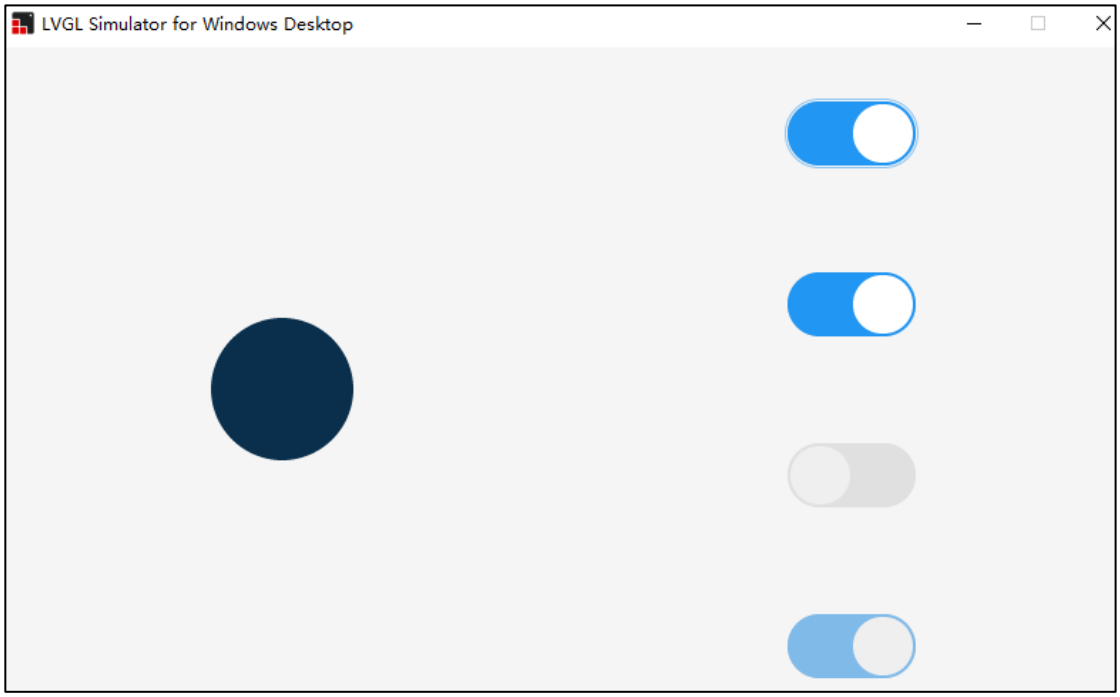


图 22.4.3.1 switch 部件实验

当按下右上角的 switch 按键时，可切换 led 部件的状态，如图 22.4.3.2 所示：

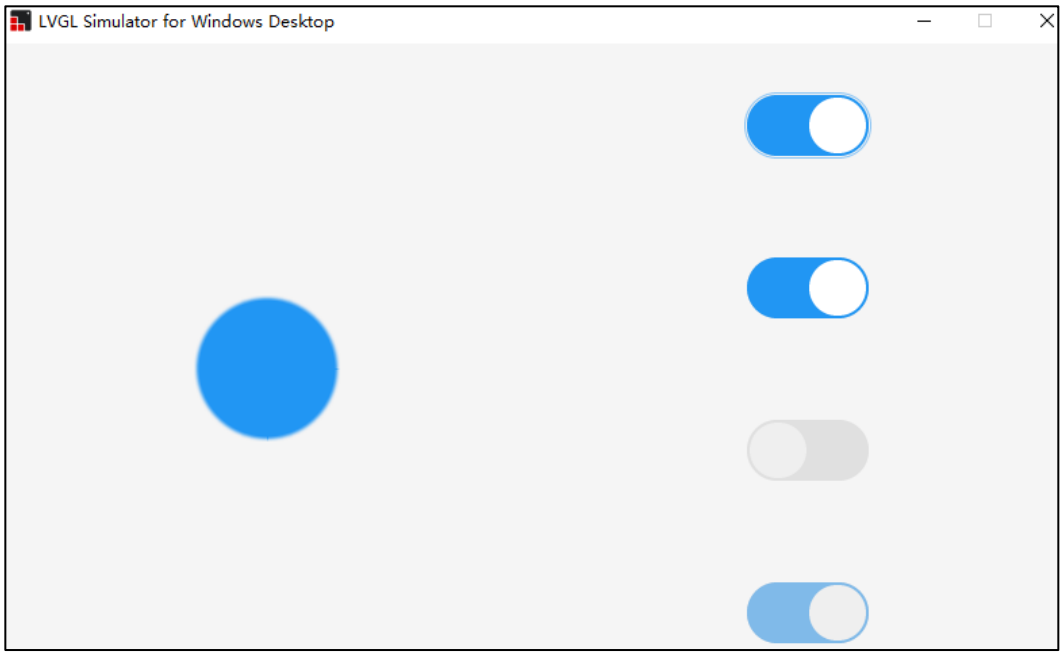


图 22.4.3.2 切换 led 部件状态

第二十三章 表格图控件(lv_table)

lv_table 是表格部件，它是由行(row)和列(col)构成的，说的再直白一点，它就是由一个一个的单元格(cell)组成的。在 LVGL 中，table 部件是非常轻量化的，经过了专门的优化，因为它的单元格(cell)只能存放文本形式的内容，不支持存放其他任何类型的对象或者部件，所以单元格中的文本内容是一个伪标签对象，该伪标签对象不是真正的 lv_label 标签部件，它是 table 部件内部通过绘图函数纯绘制出来的效果，我们可以认为每个单元格基本没有内存消耗。

本章分为如下几部分讲解：

23.1 表格部件的组成

22.2 表格部件的相关知识

22.3 表格部件的 API 函数

22.4 表格部件的实验

23.1 表格部件的组成

在 LVGL 中，table 部件是由 LV_PART_MAIN 和 LV_PART_ITEMS 两个部分组成，这两个部分分别表示 table 部件的典型的背景样式属性和 table 部件的单元格的背景样式属性和文本属性。

如果用户想设置 table 部件的相关属性，请参考第六章 6.3 小节，这小节已经很详细的讲解部件的组成部分样式属性的设置。

23.2 表格部件的相关知识

23.2.1 设置单元格的值

前面笔者已经讲解过，table 部件的单元格不支持存放其他任何类型的对象或者部件，该单元格只能存储文本，因此数字在单元格显示之前必须转换为文本。

默认情况下，调用函数 lv_table_create 创建 table 部件时，该部件是一个一行一列的单元格，如果用户想在这个单元格添加文本，则可调用函数 lv_table_set_cell_value 设置文本，该函数的第二和第三形参表示行和列，而第四个形参表示要传入的文本。

接下来我们以简单实例来讲解 table 部件的单元格添加文本，如下所示：

```
void lv_mainstart(void)
{
    /* 创建 table 部件 */
    lv_obj_t* table = lv_table_create(lv_scr_act());
    /* 设置 table 部件的单元格文本 */
    lv_table_set_cell_value(table, 0, 0, "123");
    lv_obj_center(table);
}
```

从上述可知：首先我们调用函数 lv_table_create 创建 table 部件，然后调用函数 lv_table_set_cell_value 设置指定单元格的文本，最后调用函数 lv_obj_center 对 table 部件中间对齐。由于 table 部件创建时，是一个一行一列的单元格，所以该 lv_table_set_cell_value 函数的第

二和第三个形参设置为 0 表示第 0 行和第 0 列。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 23.2.1.1 所示：



图 23.2.1.1 添加 table 部件的单元格文本

从上图可知：我们在 tabel 部件单元格设置“123”文本。

23.2.2 行和列的设置

前面大家也了解过，table 部件创建时，该部件只创建一个单元格，对于一个 table 部件来讲，它不单单只创建一个单元格，还可以创建多行多列的单元格部件。用户可调用 lv_table_set_row_cnt 和 lv_table_set_col_cnt 函数设置 table 部件行和列，也就是说：行*列等于 N 个单元格。

接下来我们以简单实例来讲解 table 部件行和列的设置，如下所示：

```
void lv_mainstart(void)
{
    /* 创建 table 部件 */
    lv_obj_t* table = lv_table_create(lv_scr_act());
    /* 设置 table 部件的单元格文本 */
    lv_table_set_cell_value(table, 0, 0, "123");
    /*设置行数 */
    lv_table_set_row_cnt(table,2);
    /*设置列数 */
    lv_table_set_col_cnt(table,2);
    lv_obj_center(table);
}
```

从上述源码可知：我们调用函数 lv_table_set_row_cnt 和 lv_table_set_col_cnt 分别设置 table 部件的行数为 2 和列数为 2，显然地，2 行*2 列等于 4 个单元格，如图 23.2.2.1 所示：

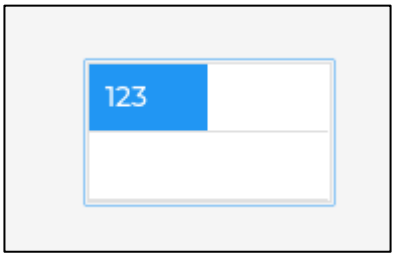


图 23.2.2.1 设置 table 部件的行列数

从上图可知：如果设置其他的单元格文本，则调用函数 lv_table_set_cell_value 指定行列数设置。接下来我们以一个示意图来讲解 table 部件的行列分布，如图 23.2.2.2 所示：

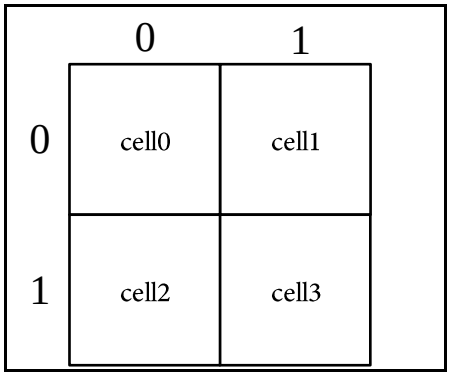


图 23.2.2.2 table 部件的行列分布

从上图可知：行和列的编号是从 0 开始的，如果设置 cell0 单元格的文本，则以第 0 行和第 0 列描述。

23.2.3 宽度和高度的设置

在 table 部件中，可通过函数 lv_table_set_col_width 设置某一列的宽度，而设置某一列的高度则根据单元格样式(字体、填充等)和行数自动计算出来的。

23.2.4 合并单元格

合并单元格是指将选定的连续单元区域合并为 1 个单元格，简单来说：上图 23.2.2.2 中的 cell0 和 cell1 合并成一个单元格。注意：在 LVGL 中，table 部件中的某一个单元格只能水平右边的一个单元格进行合并，不能垂直合并单元格。合并单元格可调用 lv_table_add_cell_ctrl 函数合并单元格。

接下来我们以一个示意图来讲解本小节的知识点，如图 23.2.4.1 所示：

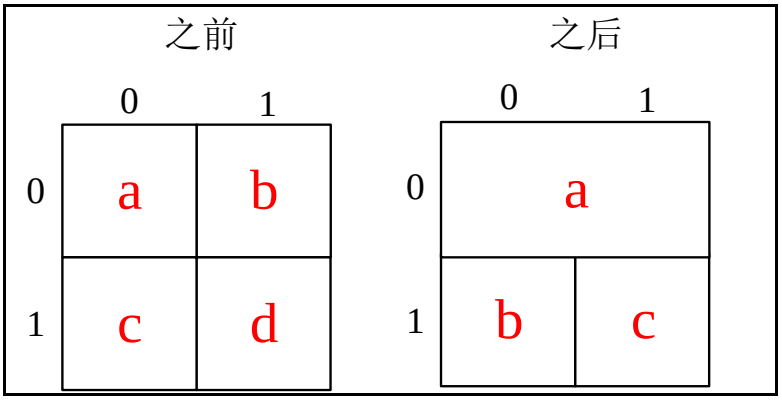


图 23.2.4.1 table 部件单元格合并示意图

从上图可知：如果 a 和 b 的单元格合并，则把 a 和 b 合并为 1 个单元格。该单元格只能从水平左到右合并，合并之后的单元格归 a 单元格所有，而 b 单元格被删除了。

23.3 表格部件的 API 函数

LVGL 提供给用户的 table 部件相关函数，如表 23.3.1 所示：

函数	描述
lv_table_create()	创建一个 tabel 对象
lv_table_set_cell_value()	设置单元格的值
lv_table_set_cell_value_fmt()	设置单元格的值，LVGL 分配内存来存储表中的文本

lv_table_set_row_cnt()	设置行数
lv_table_set_col_cnt()	设置列数
lv_table_set_col_width()	设置列的宽度
lv_table_add_cell_ctrl()	向单元格添加控制位
lv_table_clear_cell_ctrl()	清除单元格的控制位
lv_table_get_cell_value()	获取单元格的值
lv_table_get_row_cnt()	获取行数
lv_table_get_col_cnt()	获取列数
lv_table_get_col_width()	获取列的宽度
lv_table_has_cell_ctrl()	获取单元格是否具有控制位
lv_table_get_selected_cell()	获得选定的单元格

表 23.3.1.1 table 部件 API 函数

接下来我们来讲解 LVGL 的 table 部件常用的函数有哪些：

1. lv_table_create 函数

该函数的作用是创建一个 table 部件，该函数原型如下所示：

```
lv_obj_t * lv_table_create(lv_obj_t * parent)
```

该函数的形参，如表 23.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 23.3.2 函数 lv_table_create()形参描述

返回值：返回 table 部件对象。

2. lv_table_set_cell_value 函数

该函数的作用是设置单元格的值，该函数原型如下所示：

```
void lv_table_set_cell_value( lv_obj_t *obj,
                              uint16_t row,
                              uint16_t col,
                              const char *txt)
```

该函数的形参，如表 23.3.3 所示：

参数	描述
obj	指向 table 对象的指针
row	行 ID
col	列 ID
txt	要显示在单元格中的文本

表 23.3.3 函数 lv_table_set_cell_value()形参描述

返回值：无。

3. lv_table_set_cell_value_fmt 函数

该函数的作用是设置单元格的值，LVGL 分配内存来存储表中的文本，该函数原型如下所示：

```
void lv_table_set_cell_value_fmt(lv_obj_t *obj,
                                 uint16_t row,
                                 uint16_t col,
                                 const char *fmt, ...)
```

该函数的形参，如表 23.3.4 所示：

参数	描述
obj	指向 table 对象的指针

row	行 ID
col	列 ID
fmt	类似 printf 格式

表 23.3.4 函数 lv_table_set_cell_value_fmt()形参描述

返回值：无。

4. lv_table_set_row_cnt 函数

该函数的作用是设置行数，该函数原型如下所示：

```
void lv_table_set_row_cnt(lv_obj_t *obj, uint16_t row_cnt)
```

该函数的形参，如表 23.3.5 所示：

参数	描述
obj	指向 table 对象的指针
row_cnt	行数

表 23.3.5 函数 lv_table_set_row_cnt()形参描述

返回值：无。

5. lv_table_set_col_cnt 函数

该函数的作用是设置列数，该函数原型如下所示：

```
void lv_table_set_col_cnt(lv_obj_t *obj, uint16_t row_cnt)
```

该函数的形参，如表 23.3.6 所示：

参数	描述
obj	指向 table 对象的指针
row_cnt	列数

表 23.3.6 函数 lv_table_set_col_cnt()形参描述

返回值：无。

6. lv_table_set_col_width 函数

该函数的作用是设置列的宽度，该函数原型如下所示：

```
void lv_table_set_col_width(lv_obj_t *obj, uint16_t col_id, lv_coord_t w)
```

该函数的形参，如表 23.3.7 所示：

参数	描述
obj	指向 table 对象的指针
col_id	列数的 ID
w	宽度

表 23.3.7 函数 lv_table_set_col_width()形参描述

返回值：无。

23.4 表格部件的实验

23.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建 table 部件，然后在该部件的第一列上添加 100 个列单元格，其次在该部件的第二列上以奇偶数列添加

单元格的文本，最后设置第二列的奇数行添加控制位，当我们按下第二列的奇数行时，系统自动显示该行的文本。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 23 lv_table(表格)》。

23.4.2 软件设计

23.4.2.1 程序流程图

本实验的程序流程图，如下图 23.4.2.1 所示：

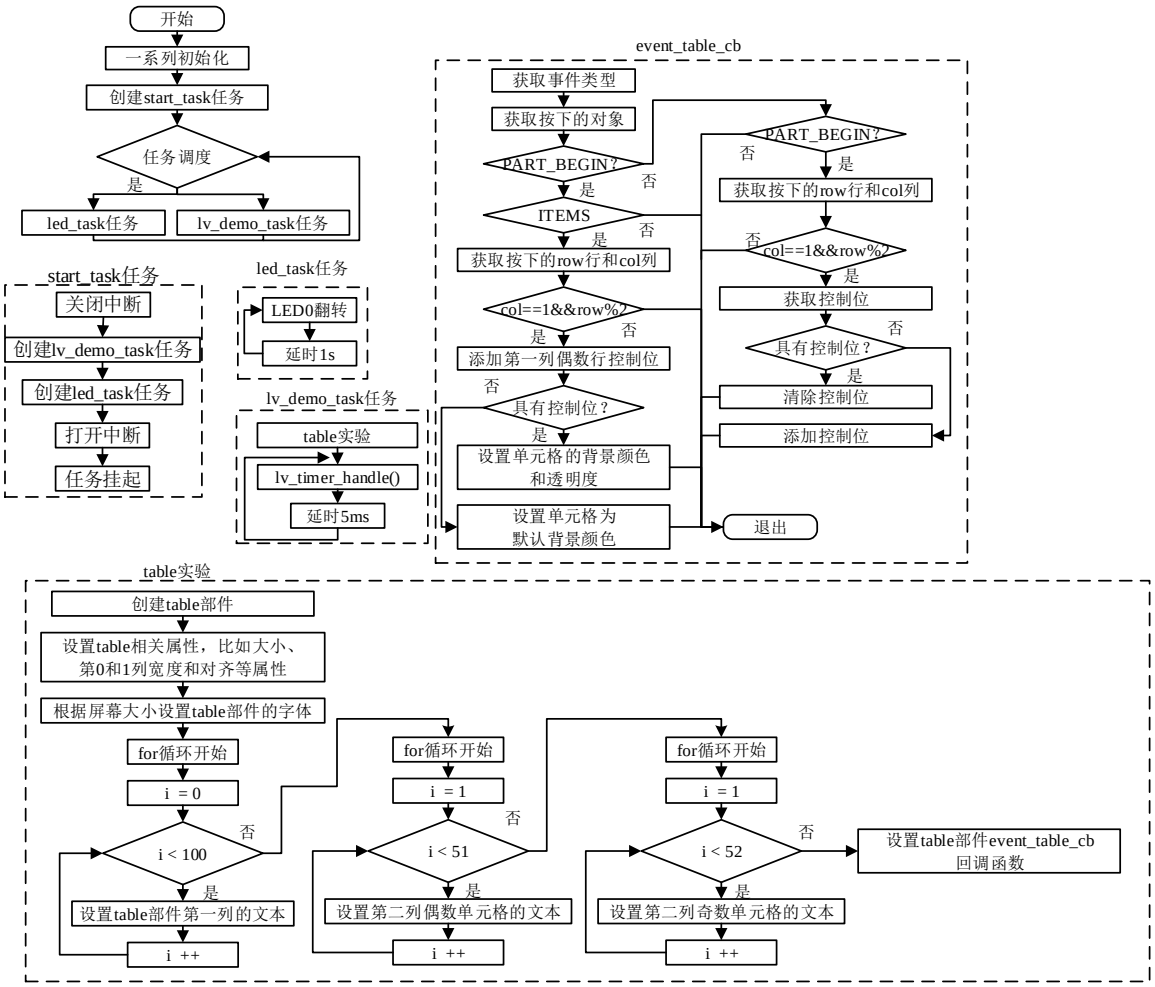


图 23.4.2.1 table 部件实验流程图

23.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
```

```
static void lv_example_table(void)
{
    lv_obj_t* table = lv_table_create(lv_scr_act());   /* 定义并创建表格 */
    lv_obj_set_width(table, scr_act_width() - 40);     /* 设置表格宽度 */
    lv_obj_set_height(table, scr_act_height() - 40);   /* 设置表格高度 */
    lv_obj_center(table);                              /* 设置表格位置 */
    /* 设置表格文本对齐方式 */
    lv_obj_set_style_text_align(table, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    lv_obj_update_layout(table);                       /* 手动更新表格参数 */
    /* 设置表格列宽度 */
    lv_table_set_col_width(table, 0, lv_obj_get_width(table) / 2 - 1);
    /* 设置表格列宽度 */
    lv_table_set_col_width(table, 1, lv_obj_get_width(table) / 2 - 1);
    /* 设置表格字体 */
    if (scr_act_width() <= 320)
        lv_obj_set_style_text_font(table, &lv_font_montserrat_14,
                                   LV_PART_MAIN);
    else if (scr_act_width() <= 480)
        lv_obj_set_style_text_font(table, &lv_font_montserrat_18,
                                   LV_PART_MAIN);
    else
        lv_obj_set_style_text_font(table, &lv_font_montserrat_20,
                                   LV_PART_MAIN);
    /* 设置第一列 */
    lv_table_set_cell_value(table, 0, 0, "Title 0");
    for (uint8_t i = 0; i <= 100; i++)
        lv_table_set_cell_value_fmt(table, i + 1, 0, "Item %d", i);
    /* 设置第二例 */
    lv_table_set_cell_value(table, 0, 1, "Title 1");
    for (uint8_t i = 1; i < (100 >> 1) + 1; i++)
        lv_table_set_cell_value_fmt(table, 2 * i, 1, "Value %d", i - 1);
    for (uint8_t i = 1; i < (100 >> 1) + 2; i++)
        lv_table_set_cell_value_fmt(table, 2 * i - 1, 1, "Hide %d", i - 1);
    /* 添加表格事件回调 */
    lv_obj_add_event_cb(table, event_table_cb, LV_EVENT_ALL, NULL);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
```

```
{
    lv_example_table();
}
```

从上述源码可知：首先我们创建 table 部件，然后设置该部件的相样式属性，其次设置该部件的单元格文本，最后设置 event_table_cb 回调函数，该回调函数主要判断用户是否按下第二列奇数单元格，如果是，则显示该单元格的文本；如果不是，则无任何效果。该回调函数如下所示：

```
/**
 * @brief  例 3table 事件回调
 * @param  e:事件描述
 * @return 无
 */
static void event_table_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* table = lv_event_get_target(e);
    lv_obj_draw_part_dsc_t* dsc = (lv_obj_draw_part_dsc_t*)
    lv_event_get_param(e);

    if (LV_EVENT_DRAW_PART_BEGIN == code) {
        if (LV_PART_ITEMS == dsc->part) {
            uint16_t row = dsc->id / lv_table_get_col_cnt(table);
            uint16_t col = dsc->id - row * lv_table_get_col_cnt(table);
        /* 判断是否是第二列奇数行 */
            if (col == 1 && row % 2 != 0) {
                /* 获取单元格是否具有控制位 */
                bool chk = lv_table_has_cell_ctrl(table, row, col,
                                                  LV_TABLE_CELL_CTRL_CUSTOM_1);
                if (true == chk) {
                    /* 设置该单元格的背景颜色和透明度 */
                    dsc->rect_dsc->bg_color = lv_color_mix(
                         lv_palette_main(LV_PALETTE_GREY),
                         dsc->rect_dsc->bg_color, LV_OPA_40);
                         dsc->rect_dsc->bg_opa = LV_OPA_COVER;
                }
                else {
                    /* 设置该单元格的文本颜色为 table 部件默认颜色 */
                    dsc->label_dsc->color = dsc->rect_dsc->bg_color;
                }
            }
        }
    }
    else if (LV_EVENT_VALUE_CHANGED == code) {
```

```
        uint16_t col, row;
        /* 获取选择的单元格 */
        lv_table_get_selected_cell(table, &row, &col);
        if (col == 1 && row % 2 != 0) {
            /* 获取单元格是否具有控制位 */
            bool chk = lv_table_has_cell_ctrl(table, row, col,
                                              LV_TABLE_CELL_CTRL_CUSTOM_1);

            if (true == chk)
                /* 如果有，则清除控制位 */
                lv_table_clear_cell_ctrl(table, row, col,
                                         LV_TABLE_CELL_CTRL_CUSTOM_1);
            else
                /* 如果没有，则添加控制位 */
                lv_table_add_cell_ctrl(table, row, col,
                                       LV_TABLE_CELL_CTRL_CUSTOM_1);
        }
    }
}
```

从上述源码可知：此函数主要判断用户按下第二列奇数行的某单元格是否有控制位，如果该单元格没有控制位，则添加控制位并设置这些单元格文本颜色为该部件背景颜色；如果该单元格有控制块，则设置该单元格背景颜色为灰色并清除控制位。

23.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 23.4.3.1 所示：

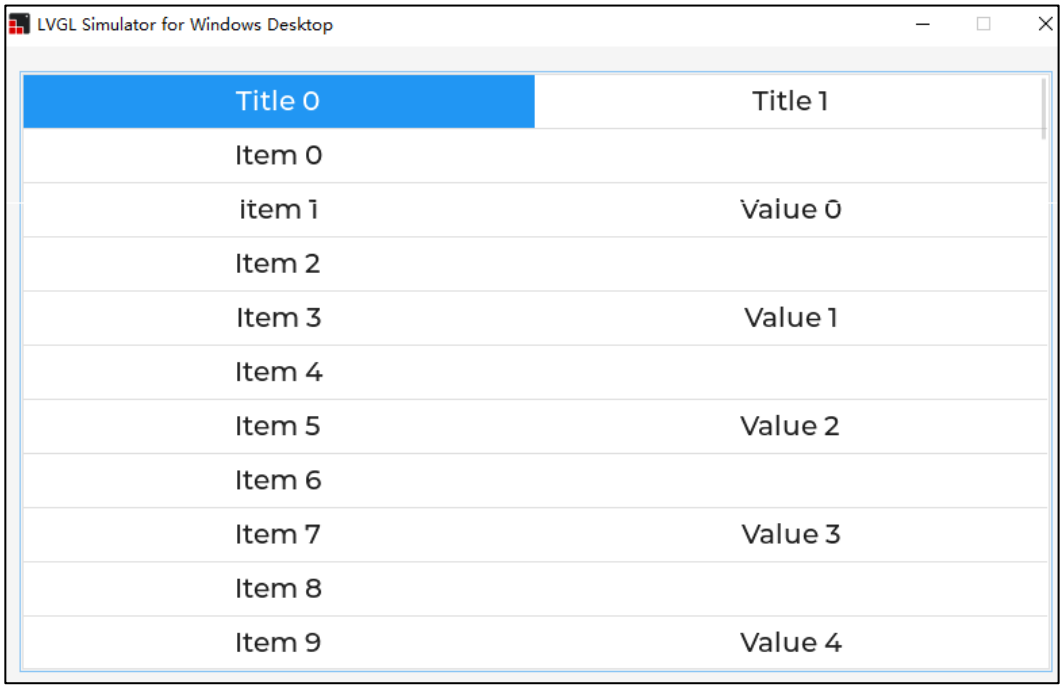


图 23.4.3.1 table 部件实验

当用户按下第二列奇数行的某单元格时，该单元格设置背景颜色为灰色，如图 23.4.3.2 所示：

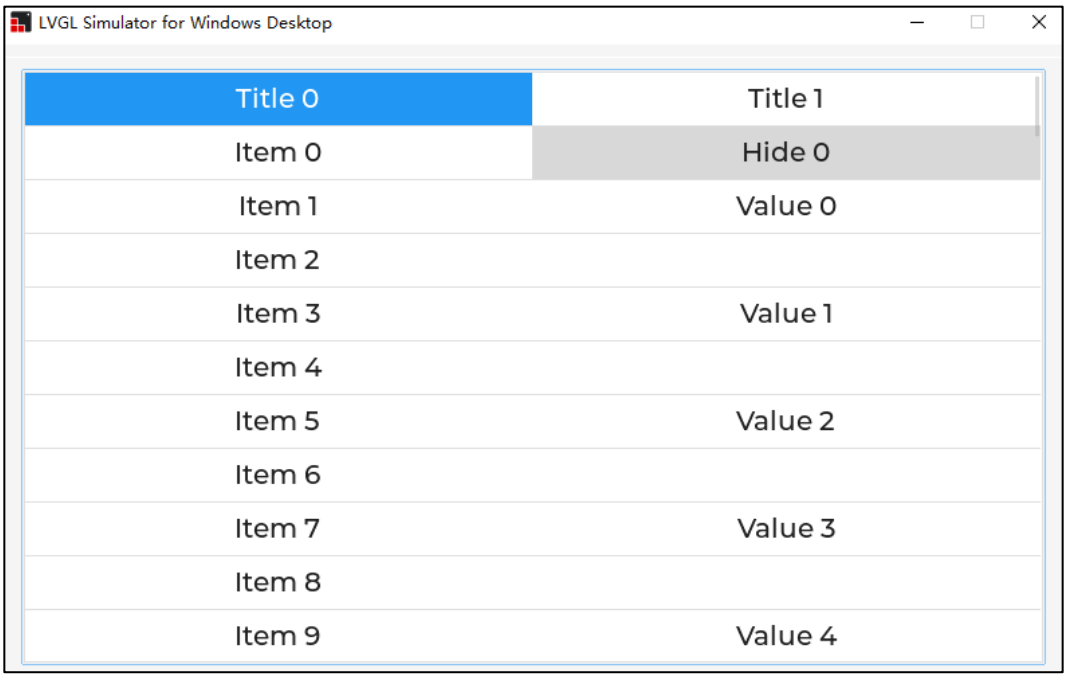


图 23.4.3.2 按下第二列奇数行的单元格效果图

第二十四章 文本区域控件(lv_textarea)

lv_textarea 文本域部件就是我们通常所看到的文本输入框，配合我们后面将要讲到的 lv_keyboard 键盘部件可以实现输入某些内容，lv_textarea 文本域部件是由一个 lv_obj 部件、一个 lv_label 标签部件和一个光标构成的。lv_textarea 文本域部件利用 lv_obj 部件的特性可以实现垂直和水平滚动，它还利用 lv_label 标签部件的特性可以实现显示文本的功能，而光标的作用就是用于标记当前所在的文本输入位置。

本章分为如下几部分讲解：

24.1 文本区域部件的组成

24.2 文本区域部件的相关知识

24.3 文本区域部件的 API 函数

24.4 文本区域部件的实验

24.1 文本区域部件的组成

在 LVGL 中，textarea 部件是由五个部分组成，它们的描述信息如下所示：

① LV_PART_MAIN：主要设置 textarea 部件背景属性以及文本样式属性，比如部件的背景颜色和文本的对齐等属性。

② LV_PART_SCROLLBAR：主要设置 textarea 部件滚动条，如果 textarea 区域显示的文本太长，则 textarea 区域显示滚动条。

③ LV_PART_SELECTED：主要设置 textarea 部件文本的样式。

④ LV_PART_CURSOR：主要设置 textarea 部件的游标位置及闪烁时间，用户可使用 anim_time 样式设置游标的样式属性。

⑤ LV_PART_TEXTAREA_PLACEHOLDER：主要设置 textarea 部件特殊区域。

在 LVGL 中，用户一般使用(1)、(2)和(4)组成部分设置相关样式属性，而(3)和(5)组成部分一般使用不到。

24.2 文本区域部件的相关知识

24.2.1 创建文本区域部件

在 LVGL 中，textarea 部件是一个文本输入框部件，如果用户使用 textarea 部件，则必须创建该部件，其次设置该部件的相关属性。创建 textarea 部件可调用以下函数：

```
lv_obj_t *lv_textarea_create(lv_obj_t *parent)
```

上述函数是 textarea 部件的创建函数，该函数只有一个形参，该形参指向该部件的父类。

24.2.2 添加与删除字符

前面我们知道 textarea 部件是一个文本输入框部件， 该部件的文本区域可输入字符和删除字符等操作，下面我们分别地讲解输入字符和删除字符的知识，如下所示：

1. 添加字符和文本

用户在文本区域中添加一个字符或者一段字符串，可使用 lv_textarea_add_char 和 lv_textarea_add_text 函数分别添加一个字符和一段字符串文本。

接下来我们以一个简单实例分别讲解以上两个函数的使用，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* textarea = lv_textarea_create(lv_scr_act());/* 定义并创建文本框 */
    lv_textarea_add_char(textarea, 'c');                  /* 添加一个字符 */
    lv_textarea_add_text(textarea, "insert this text");   /* 添加一个字符串 */
    lv_obj_center(textarea);                              /* 中间对齐 */
}
```

上述源码可知：首先我们创建 textarea 部件，然后调用 lv_textarea_add_char 函数在文本区域添加“c”字符，其次调用 lv_textarea_add_text 函数在文本区域添加一段“insert this text”字符串，最后把 textarea 部件中间对齐。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 24.2.2.1 所示：



图 24.2.2.1 添加文本区域的字符和字符串

从上图可知：首先添加“c”字符，然后添加一段“insert this text”字符串。

2. 删除字符

在 LVGL 中，textarea 部件删除文本区域的字符具有两个方式，这两种方式如下所示：

① 如果从光标位置的左边删除一个字符，则调用函数 lv_textarea_del_char 设置。

② 如果从光标位置的右边删除一个字符，则调用函数 lv_textarea_del_char_forward 设置。

上述的两个方式都是根据文本区域的光标位置来确定方向来删除字符的。

24.2.3 占位符文本

所谓占位符（Placeholder），是界面设计中常用到的一种元素，它可以包含具体的功能或是信息的组织结构。用户面对空白的电子表单等界面时，往往会产生不能理解交互方式的“认知摩擦”。占位符的主要作用就是在此类用户界面中占据一定的空间面积，既能调节或控制界面给用户的视觉传达效果，同时又可以实现对用户进行默认的提示、说明或引导等功能。

在 LVGL 中，textarea 部件的文本区域的占位符可调用 lv_textarea_set_placeholder_text 函数设置，该函数具有两个形参，值得读者注意的是第二个形参，该形参对文本区域起到说明和引导作用，接下来我们以一个简单实例来讲解上述函数的使用，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* textarea = lv_textarea_create(lv_scr_act());/* 定义并创建文本框 */
    /* 设置字体 */
    lv_obj_set_style_text_font(textarea,&myFont14,LV_STATE_DEFAULT);
    /* 设置占位符 */
    lv_textarea_set_placeholder_text(textarea, "请输入文本.....");
    lv_obj_center(textarea);
```

```
}
```

从上述源码可知：我们设置 textarea 部件字体为 myFont14，然后调用函数设置占位符，该占位符提示文本为“请输入文本.....”。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 24.2.3.1 所示：



图 24.2.3.1 设置占位符

从上图可知：占位符就是对用户进行提示和引导。

24.2.4 移动或者隐藏光标

1. 移动光标

在 LVGL 中，文本区域的光标是在左上角位置显示，当我们添加一个字符时，该光标从左往右移动，默认情况下，光标一直在文本最后一个字符后的位置。当然用户可以手动设置光标的位置，可调用函数 lv_textarea_set_cursor_pos 设置，该函数的第二个形参表示光标的移动位置，如果该形参设置为 0，则光标在第一个字符之前；如果该形参设置为 LV_TA_CURSOR_LAST，则光标在最后一个字符后。

如果用户不想使用以上的方式设置光标的位置，可使用第二种方式移动光标，这些方式如表 24.2.4.1 所示：

函数	描述
lv_textarea_cursor_right()	光标往右移动
lv_textarea_cursor_left()	光标往左移动
lv_textarea_cursor_up()	光标往上移动
lv_textarea_cursor_down()	光标往下移动

表 24.2.4.1 移动光标函数

上述的光标移动函数一般用作于按键手动控制光标，因为触摸控制光标的位置可通过点击某个位置，光标自动跳转到被点击的位置。在默认情况下，触摸控制光标位置是启用的，如果触摸不能使用，可调用函数 lv_textarea_set_cursor_click_pos (textarea, true) 启用触摸控制，该函数可将光标跳转到文本区域被点击的位置。

2. 隐藏光标

默认情况下，无论用户是否操作文件区域，该文本区域的光标都是可见的。在理想状态下，我们只想操作文本区域时，该光标是可见，而不操作文本区域时，该光标是隐藏状态的。对于这样的状态实现，用户可调用函数 lv_obj_add_flag 把文本区域添加 LV_STATE_FOCUSED 状态。

24.2.5 文本区域部件的模式

在 LVGl 中，textarea 部件的模式可设置两个模式，这些模式如下所示：

① 一行模式：默认情况下，文本区域的文本超出它的宽度时，该文本自动换行，如果文本区域设置一行模式，则文本超出它的宽度并不会自动换行，该超出的文本以水平滚动条滚动显示。一行模式可通过函数 lv_textarea_set_one_line(textarea, true)设置。

② 密码模式：为了保证输入文本的机密性，用户输入文本区域的文本以“*”字符代替。密码模式可通过函数 lv_textarea_set_password_mode(textarea, true)设置。

上述的模式可以共用于一个 textarea 部件当中，这个部件类似于银行 ATM 自动取款机的密码框，该密码框输入字符时，有大概有一秒钟的时间显示输入的字符，然后这些字符以“*”代替，这个显示的时间我们可以在 lv_conf.h 文件的 LV_TEXTAREA_DEF_PWD_SHOW_TIME 宏定义设置显示时间。

如果用户想获取密码框的文本内容，可调用函数 lv_textarea_get_text 获取文本，该文本并不是“*”字符，而是正常输入的文本内容。

24.2.6 限制字符输入

限制字符输入是指文本区域只允许规定的字符输入，例如银行 ATM 自动取款机的密码框，这个密码框只允许输入 0~9 字符且六位数密码。LVGL 的文本区域限制字符输入具有两类操作，这两类操作如下所示：

第一类操作：限制输入的字符，比如 ATM 的密码框，它只能输入数字 0~9。第一类操作方式主要调用函数 lv_textarea_set_accepted_chars 设置，该函数的第二个形参表示可接收的字符列表，例如"0123456789"字符串。

第二类操作：限制输入的字符长度，比如 ATM 的密码框，它只能输入 6 位密码。第一类操作方式主要调用函数 lv_textarea_set_max_length 设置，该函数的第二个形参表示输入字符的长度。

接下来我们以简单实例来讲解这两类操作到底如何使用，如下源码所示：

```
void lv_mainstart(void)
{
    /* 定义并创建文本框 */
    lv_obj_t* textarea = lv_textarea_create(lv_scr_act());
    /* 设置输入字符长度为 6 */
    lv_textarea_set_max_length(textarea,6);
    /* 设置接收输入字符列表 */
    lv_textarea_set_accepted_chars(textarea,"0123456789");
    /* 一行模式 */
    lv_textarea_set_one_line(textarea, true);
    /* 中间对齐 */
    lv_obj_center(textarea);
    /* 输入的字符 */
    lv_textarea_add_char(textarea, 'C');
    lv_textarea_add_char(textarea, 'a');
    lv_textarea_add_char(textarea, 'i');
    lv_textarea_add_char(textarea, 'X');
    lv_textarea_add_char(textarea, 'e');
    lv_textarea_add_char(textarea, 'u');
    lv_textarea_add_char(textarea, 'e');
    lv_textarea_add_char(textarea, 'F');
    lv_textarea_add_char(textarea, 'e');
    lv_textarea_add_char(textarea, 'n');
```

```
    lv_textarea_add_char(textarea, 'g');
    lv_textarea_add_char(textarea, '1');
    lv_textarea_add_char(textarea, '6');
    lv_textarea_add_char(textarea, '8');
    lv_textarea_add_char(textarea, '6');
    lv_textarea_add_char(textarea, '6');
    lv_textarea_add_char(textarea, '6');
}
```

从上述源码可知：首先我们调用函数 lv_textarea_set_max_length 设置输入字符长度为 6，然后调用函数 lv_textarea_set_accepted_chars 设置输入接收的字符列表为“0123456789”，最后调用多个函数 lv_textarea_add_char 添加文本，该文本为“CaiXueFeng168666”字符串。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 24.2.6.1 所示：

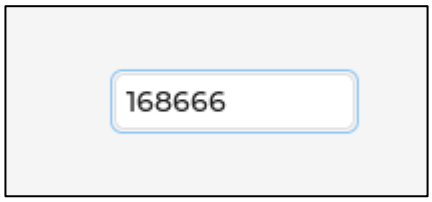


图 24.2.6.1 测试限制字符输入实验

从上图可知：我们输入的文本为“CaiXueFeng168666”，而最终显示的是“168666”，因为我们已经设置文本区域的接收字符列表为 0~9 数字，所以输入的字符必须符合 0~9 的数字。

总结：

如果文本区域中有很长的文本(例如> 20k 字符)，滚动和绘制可能会很慢。但是，通过在 lv_conf.h 中启用宏定义 LV_LABEL_LONG_TXT_HINT，可以极大地提高性能。这将保存一些关于标签的额外信息，以加快其绘制速度。使用 LV_LABEL_LONG_TXT_HINT 滚动和绘制将与“普通”短文本一样快。

24.3 文本区域部件的 API 函数

LVGL 提供给用户的 textarea 部件相关函数，如表 24.3.1 所示：

函数	描述
lv_textarea_create()	创建一个文本区域对象
lv_textarea_add_char()	插入一个字符到当前光标位置
lv_textarea_add_text()	将文本插入到当前光标位置
lv_textarea_del_char()	删除当前光标位置的左字符
lv_textarea_del_char_forward()	从当前光标位置删除右字符
lv_textarea_set_text()	设置文本区域的文本
lv_textarea_set_placeholder_text()	设置文本区域的占位符文本
lv_textarea_set_cursor_pos()	设置光标位置
lv_textarea_set_cursor_click_pos()	启用/禁用光标的位置
lv_textarea_set_password_mode()	启用/禁用密码模式
lv_textarea_set_one_line()	将文本区域配置为单行
lv_textarea_set_accepted_chars()	设置接收字符列表
lv_textarea_set_max_length()	设置文本区域的最大长度
lv_textarea_set_insert_replace()	文本区域添加自动格式
lv_textarea_set_text_selection()	启用/禁用选择模式
lv_textarea_set_password_show_time()	将密码更改为“*”之前显示多长时间

lv_textarea_set_align()	已弃用，设置标签的对齐方式
lv_textarea_get_text()	在密码模式下，获取文本区域的文本
lv_textarea_get_placeholder_text()	获取文本区域的占位符文本
lv_textarea_get_label()	获取文本区域的标签
lv_textarea_get_cursor_pos()	获取当前光标在字符索引中的位置
lv_textarea_get_cursor_click_pos()	获取是否启用光标单击定位
lv_textarea_get_password_mode()	获取密码模式属性
lv_textarea_get_one_line()	获取一行配置属性
lv_textarea_get_accepted_chars()	获取可接收字符的列表
lv_textarea_get_max_length()	获取文本区域的最大长度
lv_textarea_text_is_selected()	查找文本是否被选中
lv_textarea_get_text_selection()	查看是否启用了选择模式
lv_textarea_get_password_show_time()	获取密码更改为“*”之前显示的时间
lv_textarea_clear_selection()	清除文本区域上的选择
lv_textarea_cursor_right()	将光标向右移动一个字符
lv_textarea_cursor_left()	将光标向左移动一个字符
lv_textarea_cursor_down()	将光标向下移动一个字符
lv_textarea_cursor_up()	将光标向上移动一个字符

表 24.3.1.1 textarea 部件 API 函数

接下来我们来讲解 LVGL 的 textarea 部件常用的函数有哪些：

1. lv_textarea_create 函数

该函数的作用是创建一个 textarea 部件，该函数原型如下所示：

```
lv_obj_t * lv_textarea_create(lv_obj_t * parent)
```

该函数的形参，如表 24.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 24.3.2 函数 lv_textarea_create()形参描述

返回值：返回 textarea 部件对象。

2. lv_textarea_add_char 函数

该函数的作用是插入一个字符到当前光标位置，该函数原型如下所示：

```
void lv_textarea_add_char(lv_obj_t *obj, uint32_t c)
```

该函数的形参，如表 24.3.3 所示：

参数	描述
obj	指向文本区域对象的指针
c	输入的字符

表 24.3.3 函数 lv_textarea_add_char()形参描述

返回值：无。

3. lv_textarea_add_text 函数

该函数的作用是将文本插入到当前光标位置，该函数原型如下所示：

```
void lv_textarea_add_text(lv_obj_t *obj, const char *txt)
```

该函数的形参，如表 24.3.4 所示：

参数	描述
obj	指向文本区域对象的指针
txt	要插入的以'\0'结尾的字符串

表 24.3.4 函数 lv_textarea_add_text()形参描述

返回值：无。

4. lv_textarea_del_char 函数

该函数的作用是删除当前光标位置的左字符，该函数原型如下所示：

```
void lv_textarea_del_char(lv_obj_t *obj)
```

该函数的形参，如表 24.3.5 所示：

参数	描述
obj	指向文本区域对象的指针

表 24.3.5 函数 lv_textarea_del_char()形参描述

返回值：无。

5. lv_textarea_del_char_forward 函数

该函数的作用是从当前光标位置删除右字符，该函数原型如下所示：

```
void lv_textarea_del_char_forward(lv_obj_t *obj)
```

该函数的形参，如表 24.3.6 所示：

参数	描述
obj	指向文本区域对象的指针

表 24.3.6 函数 lv_textarea_del_char_forward()形参描述

返回值：无。

6. lv_textarea_set_text 函数

该函数的作用是插入一个字符到当前光标位置，该函数原型如下所示：

```
void lv_textarea_add_char(lv_obj_t *obj, uint32_t c)
```

该函数的形参，如表 24.3.7 所示：

参数	描述
obj	指向文本区域对象的指针
c	输入的字符

表 24.3.7 函数 lv_textarea_set_text()形参描述

返回值：无。

7. lv_textarea_set_password_mode 函数

该函数的作用是启用/禁用密码模式，该函数原型如下所示：

```
void lv_textarea_set_password_mode(lv_obj_t *obj, bool en)
```

该函数的形参，如表 24.3.8 所示：

参数	描述
obj	指向文本区域对象的指针
en	true:启用，false:不启用

表 24.3.8 函数 lv_textarea_set_password_mode()形参描述

返回值：无。

8. lv_textarea_set_one_line 函数

该函数的作用是将文本区域配置为单行，该函数原型如下所示：

```
void lv_textarea_set_one_line(lv_obj_t *obj, bool en)
```

该函数的形参，如表 24.3.9 所示：

参数	描述
obj	指向文本区域对象的指针
en	true:一行，false:正常

表 24.3.9 函数 lv_textarea_set_one_line()形参描述

返回值：无。

9. lv_textarea_set_accepted_chars 函数

该函数的作用是设置接收字符列表，该函数原型如下所示：

```
void lv_textarea_set_accepted_chars(lv_obj_t *obj, const char *list)
```

该函数的形参，如表 24.3.10 所示：

参数	描述
obj	指向文本区域对象的指针
list	字符的列表。只保存指针。如。“0123456789 + -。”

表 24.3.10 函数 lv_textarea_set_accepted_chars()形参描述

返回值：无。

10. lv_textarea_set_max_length 函数

该函数的作用是设置文本区域的最大长度，该函数原型如下所示：

```
void lv_textarea_set_max_length(lv_obj_t *obj, uint32_t num)
```

该函数的形参，如表 24.3.11 所示：

参数	描述
obj	指向文本区域对象的指针
num	最大字符数

表 24.3.11 函数 lv_textarea_set_max_length()形参描述

返回值：无。

24.4 文本区域部件的实验

24.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建多个部件，这些部件分别为文本框*2、label*2 以及 keyboard 部件。文本框分别为用户名文本框和密码文本框，它们都设置成一行类型，而密码文本框还设置密码模式，它所输入的字符都以“*”代替；label 部件分别提示用户文本框和密码文本框，它们分别设置"User name"和"Password"字符串，这些字符串都放置在对应的文本框的左则；keyboard 部件主要对用户名文本框和密码文本框输入字符。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 24 lv_textarea(文本框)》。

24.4.2 软件设计

24.4.2.1 程序流程图

本实验的程序流程图，如下图 24.4.2.1.1 所示：

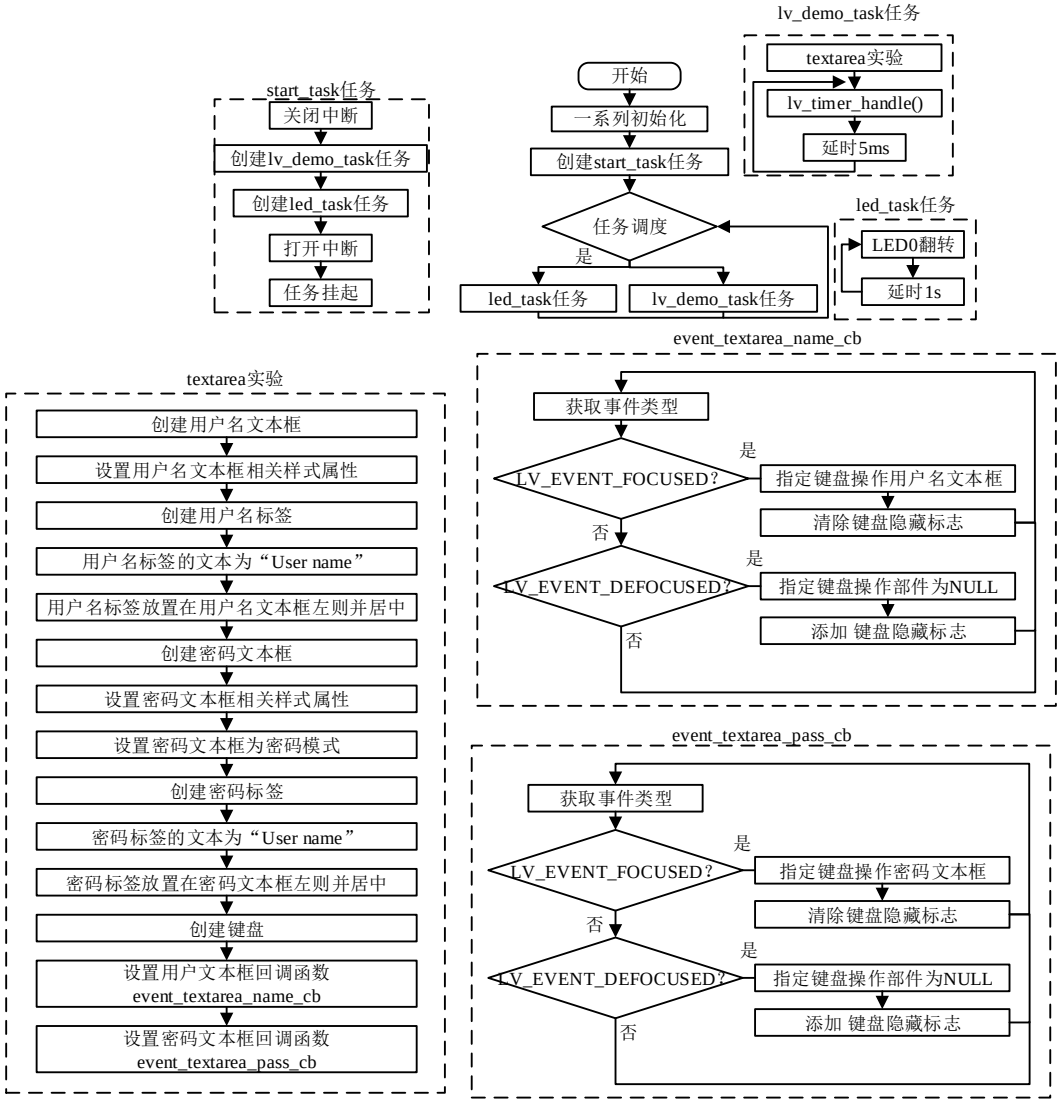


图 24.4.2.1.1 lv_textarea 实验流程图

24.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_textarea(void) {
    const lv_font_t* font;
    /* 根据屏幕大小设置字体 */
    if (scr_act_width() <= 320)
        font = &lv_font_montserrat_12;
    else if (scr_act_width() <= 480)
```

```
    font = &lv_font_montserrat_14;
else
    font = &lv_font_montserrat_22;

/* 用户名文本框 *//* 定义并创建文本框 */
lv_obj_t* textarea_name = lv_textarea_create(lv_scr_act());
/* 设置文本框的宽度 */
lv_obj_set_width(textarea_name, scr_act_width() / 2);
/* 设置文本框的高度 */
lv_obj_set_height(textarea_name, font->line_height << 1);
/* 设置文本框的字体 */
lv_obj_set_style_text_font(textarea_name, font, LV_PART_MAIN);
lv_obj_align(textarea_name, LV_ALIGN_CENTER, scr_act_width() >> 3,
             -scr_act_height() / 8 * 3);  /* 设置文本框的位置 */
lv_obj_align(lv_textarea_get_label(textarea_name),
             LV_ALIGN_LEFT_MID, 0, 0);     /* 设置文本框文本对齐方式 */
/* 设置文本框为单行模式 */
lv_textarea_set_one_line(textarea_name, true);
/* 设置文本框输入的最大长度 */
lv_textarea_set_max_length(textarea_name, 20);
/* 设置文本框选择文本 */
lv_textarea_set_text_selection(textarea_name, true);
/* 用户名标签 *//* 定义并创建标签 */
lv_obj_t* label_name = lv_label_create(lv_scr_act());
/* 设置标签的宽度 */
lv_obj_set_width(label_name, font->line_height * 9);
/* 设置标签的高度 */
lv_obj_set_height(label_name, font->line_height + 10);
/* 设置标签的位置 */
lv_obj_align_to(label_name, textarea_name, LV_ALIGN_OUT_LEFT_BOTTOM,
                -10, 0);
/* 设置标签的字体 */
lv_obj_set_style_text_font(label_name, font, LV_PART_MAIN);
/* 设置标签的文本 */
lv_label_set_text(label_name, "User name");
/* 设置标签的文本对齐方式 */
lv_obj_set_style_text_align(label_name, LV_TEXT_ALIGN_RIGHT, LV_PART_MAIN);
/* 密码文本框 *//* 定义并创建文本框 */
lv_obj_t* textarea_pass = lv_textarea_create(lv_scr_act());
/* 设置文本框的宽度 */
lv_obj_set_width(textarea_pass, scr_act_width() / 2);
/* 设置文本框的高度 */
lv_obj_set_height(textarea_pass, font->line_height << 1);
```

```
    /* 设置文本框的字体 */
    lv_obj_set_style_text_font(textarea_pass, font, LV_PART_MAIN);
    /* 设置文本框的位置 */
    lv_obj_align_to(textarea_pass, textarea_name, LV_ALIGN_OUT_BOTTOM_MID,
                    0, font->line_height);
    /* 设置文本框文本对齐方式 */
    lv_obj_align(lv_textarea_get_label(textarea_pass), LV_ALIGN_LEFT_MID,
                 0, 0);
    /* 设置文本框为单行模式 */
    lv_textarea_set_one_line(textarea_pass, true);
    /* 设置文本框为密码模式 */
    lv_textarea_set_password_mode(textarea_pass, true);
    /* 设置密码显示的时间 */
    lv_textarea_set_password_show_time(textarea_pass, 300);
    /* 设置文本框输入的最大长度 */
    lv_textarea_set_max_length(textarea_pass, 20);
    /* 密码标签 *//* 定义并创建标签 */
    lv_obj_t* label_pass = lv_label_create(lv_scr_act());
    /* 设置标签的宽度 */
    lv_obj_set_width(label_pass, font->line_height * 8);
    /* 设置标签的高度 */
    lv_obj_set_height(label_pass, font->line_height + 10);
    /* 设置标签的位置 */
    lv_obj_align_to(label_pass, textarea_pass, LV_ALIGN_OUT_LEFT_BOTTOM,
                    -10, 0);
    /* 设置标签的字体 */
    lv_obj_set_style_text_font(label_pass, font, LV_PART_MAIN);
    /* 设置标签的文本 */
    lv_label_set_text(label_pass, "password" );
    /* 设置标签的文本对齐方式 */
    lv_obj_set_style_text_align(label_pass, LV_TEXT_ALIGN_RIGHT, LV_PART_MAIN);
    /* 键盘 *//* 定义并创建键盘 */
    lv_obj_t* keyboard = lv_keyboard_create(lv_scr_act());
    lv_obj_add_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    /* 添加文本框事件回调 */
    lv_obj_add_event_cb(textarea_name, event_textarea_name_cb,
                        LV_EVENT_ALL, keyboard);
    /* 添加文本框事件回调 */
    lv_obj_add_event_cb(textarea_pass, event_textarea_pass_cb,
                        LV_EVENT_ALL, keyboard);
}

/**
```

```
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_textarea();
}
```

此函数实际调用 lv_example_textarea 函数编写 LVGL 程序，该函数首先判断 LCD 的类型，系统会根据 LCD 的类型选择字库大小，其次创建用户名文本框和密码文本框，它们分别设置各自的样式属性，密码文本框还启用了密码模式，该模式下的输入字符以“*”代替，然后创建两个 label 部件，它们分别设置“User name”和“password”字符串来提示用户名文本框和密码文本框，最后创建键盘部件，该键盘主要操作文本框的字符输入与删除，键盘操作的程序放在 event_textarea_name_cb 和 event_textarea_pass_cb 回调函数中，这些回调函数如下所示：

```
    /**
 * @brief  name 文本框事件回调
 * @param  无
 * @return 无
 */
static void event_textarea_name_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* textarea_name = lv_event_get_target(e);
    lv_obj_t* keyboard = (lv_obj_t*)e->user_data;

    if (LV_EVENT_FOCUSED == code) {
        lv_keyboard_set_textarea(keyboard, textarea_name);
        lv_obj_clear_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    }
    else if (LV_EVENT_DEFOCUSED == code) {
        lv_keyboard_set_textarea(keyboard, NULL);
        lv_obj_add_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    }
}

/**
 * @brief  pass 文本框事件回调
 * @param  无
 * @return 无
 */
static void event_textarea_pass_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
```

```
    lv_obj_t* textarea_pass = lv_event_get_target(e);
    lv_obj_t* keyboard = (lv_obj_t*)e->user_data;

    if (LV_EVENT_FOCUSED == code) {
        lv_keyboard_set_textarea(keyboard, textarea_pass);
        lv_obj_clear_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    }
    else if (LV_EVENT_DEFOCUSED == code) {
        lv_keyboard_set_textarea(keyboard, NULL);
        lv_obj_add_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    }
}
```

上述的两个回调函数都是类似操作，这两个回调函数首先判断文本框是否聚焦，如果文本框触发聚焦事件，则键盘操作该文本框并清除键盘隐藏标志；如果文本框非聚焦状态，则该键盘不操作该文本框并把键盘添加隐藏标志。

24.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 24.4.3.1 所示：

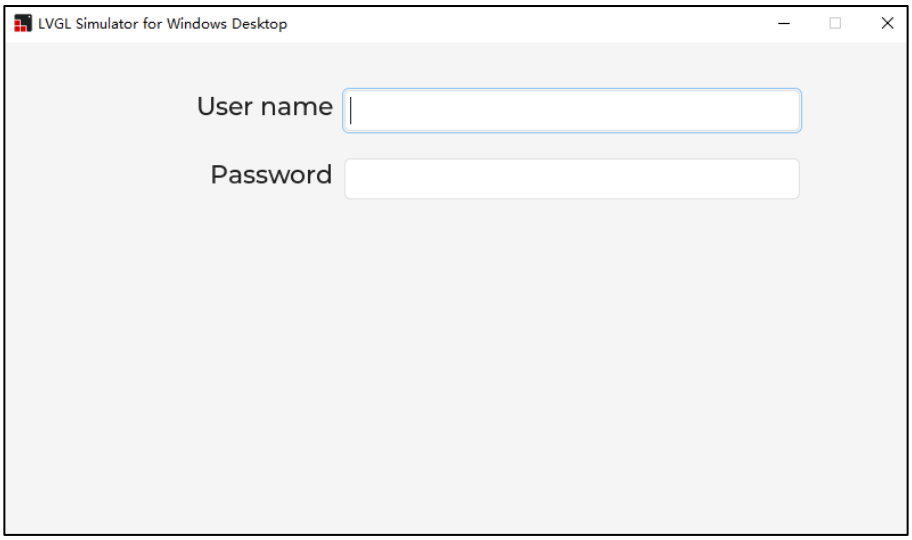


图 24.4.3.1 textarea 部件实验

当按下某个文本框时，键盘操作该文本框并清除隐藏标志，如图 24.4.3.2 所示：

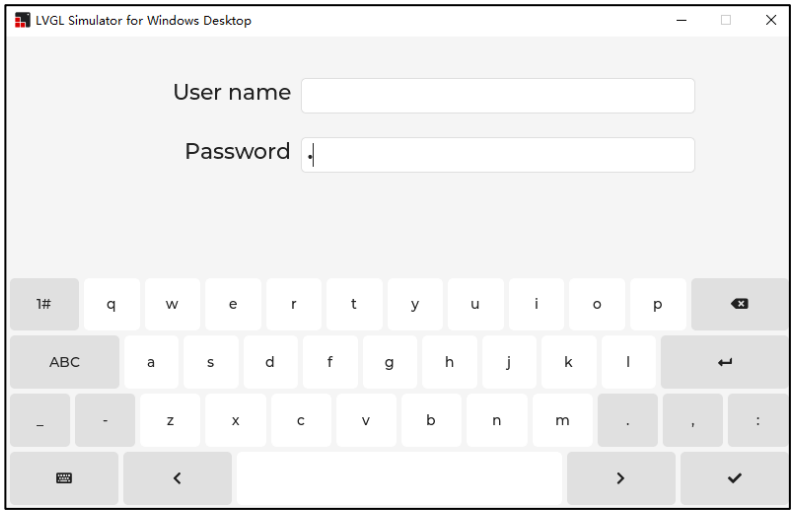


图 24.4.3.2 按下某个文本框

在密码文本框输入字符，该字符以“*”代替，如图 24.4.3.3 所示：

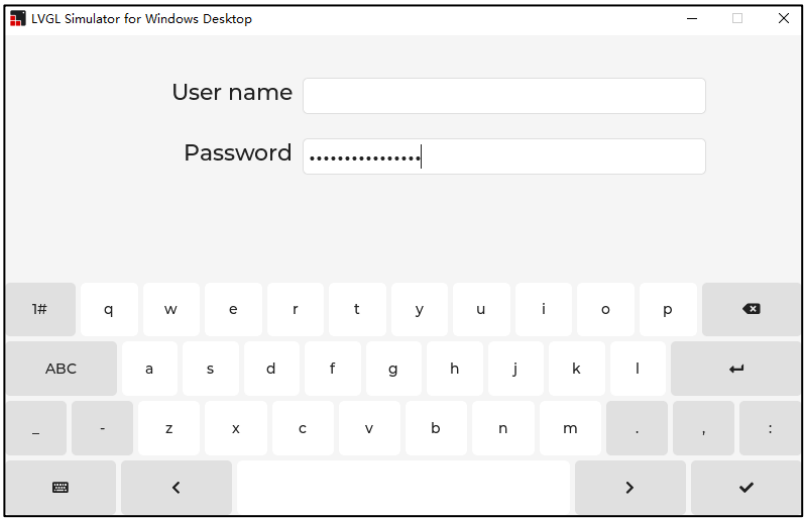


图 24.4.3.3 密码模式下的文本框

第二十五章 日历部件(lv_calendar)

Calendar 部件是一款类似桌面的日历类应用，提供快速浏览每月日程及当日日程安排，左右划动切换每月日历，可以突出当前日期(今天)和自定义日期。

本章分为如下几部分讲解：

25.1 日历部件的组成

25.2 日历部件的相关知识

25.3 日历部件的 API 函数

25.4 日历部件的实验

25.1 日历部件的组成

Calendar 部件使用 Button 矩阵对象将日期排列到一个矩阵中。显然地，Calendar 部件与 Button 矩阵对象息息相关，接下来我们分别地讲解 Calendar 部件组成部分到底有哪些：

(1) LV_PART_MAIN：设置日历的背景属性。

(2) LV_PART_ITEMS：指向日期和日名。例如设置按钮矩阵控制标志来区分按钮，并添加一个自定义抽屉事件，修改按钮的属性如下所示：

① 日名(星期一….)没有边框，没有背景，只能用灰色绘制。

② 上个月和下个月的天数使用 LV_BTNMATRIX_CTRL_DISABLED 标志。

③ 设置突出日期(今天)的边框与主题。

如果用户想设置 Calendar 部件的相关属性，请参考第六章 6.3 小节，这小节已经很详细的讲解部件的组成部分样式属性的设置。

25.2 日历部件的相关知识

25.2.1 创建日历部件

在 LVGL 中，Calendar 部件是一个日历部件，如果用户使用 Calendar 部件，则必须创建该部件并设置该部件的相关属性。创建 Calendar 部件可调用以下函数：

```
lv_obj_t *lv_Calendar_create(lv_obj_t *parent)
```

上述函数是 Calendar 部件的创建函数，该函数只有一个形参，该形参指向该部件的父类。

25.2.2 日期的设置/显示

1. 设置当前日期

当前日期是指用户设置某年月日为开始的日期，比如今天是 2022 年 4 月 7 号为当前日期(今天)。默认情况下，我们创建一个 Calendar 部件时，该部件的当前日期为 2020 年 1 月 1 号，显然地，它并不是当前准确的日期，所以 LVGL 作者提供了一个函数可设置当前日期，该函数为 lv_calendar_set_today_date。

2. 显示日期

前面我们讲解过 Calendar 部件默认显示当前日历为 2020 年 1 月，如果我们只设置当前日期为 2022 年 4 月 7 号，则该部件不会跳转到 2022 年 4 月的日历，所以我们必须手动跳转当前日历为 2022 年 4 月，该设置方法是调用函数 lv_calendar_set_shown_date 设置。

接下来我们以简单实例来讲解上述函数的使用流程，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())

void lv_mainstart(void)
{
    /* 定义并初始化日历 */
    lv_obj_t* calendar = lv_calendar_create(lv_scr_act());
    /* 设置日历的大小 */
    lv_obj_set_size(calendar, scr_act_height() * 0.85, scr_act_height()
                    * 0.85);
    lv_obj_center(calendar);
    /* 设置日历的日期 */
    lv_calendar_set_today_date(calendar, 2022, 4, 7);
    /* 设置日历显示的月份 */
    lv_calendar_set_showed_date(calendar, 2022, 4);
    /* 设置日历头 */
    lv_calendar_header_dropdown_create(calendar);
    /* 更新日历参数 */
    lv_obj_update_layout(calendar);
}
```

上述源码可知：首先我们创建一个 Calendar 部件，然后设置当前日期和显示日历月份，最后显示日历头和更新日历参数。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 25.2.2.1 所示：

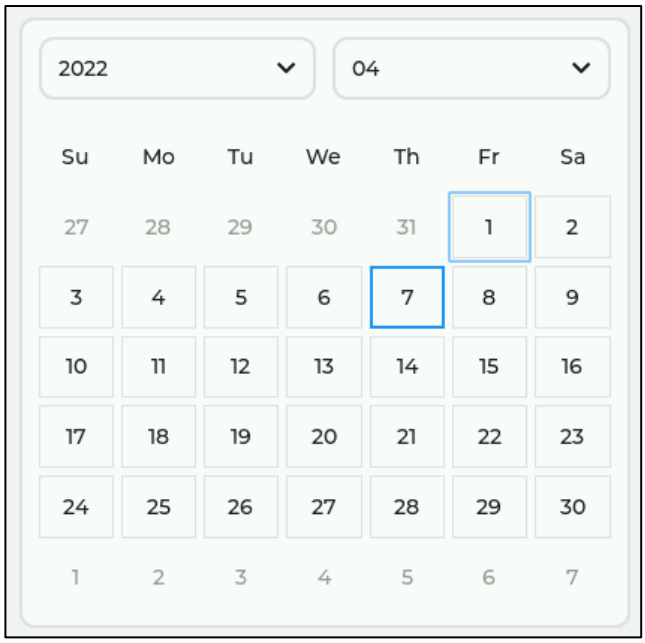


图 25.2.2.1 设置当前日期

从上图可知：首先我们调用函数 lv_calendar_header_dropdown_create 显示日历顶部的年份和月份，然后调用函数 lv_calendar_set_showed_date 显示 2022 年 4 月份的日历，最后调用函数 lv_calendar_set_today_date 设置当前日期(今天)。

总结：

Calendar 部件显示日历顶部的年份和月份具有两个方式，这些方式如下所示：

① 调用函数 lv_calendar_header_dropdown_create 设置。

② 调用函数 lv_calendar_header_arrow_create 设置。

上述两个方式都是可以显示日历的年份和月份，它们唯一不同的是年份和月份的选择，一个使用下拉列表选择年月份，另一个是使用按键选择月份但不能调整年份，该年份根据月份的进度来决定，比如年份为 2022 年，该年份切换到 2023 年必须由月份不断累加，直到月份跨过 12 月份。

25.2.3 设置日期高亮

在日常生活中，无论是手机日历还是实体日历，它们会把重要的节日以高亮的形式展现出来，例如我们国家的传统节日都是设置成高亮的形式，以表示这些日期非常重要。LVGL 的 Calendar 部件也是支持这样的功能，用户可调用函数 lv_calendar_set_highlighted_dates 来设置哪些日期高亮。下面我们以一个实例来讲解上述函数的使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())
static lv_calendar_date_t highlight_days[2]; /* 定义的日期,必须用全局或静态定义 */

void lv_mainstart(void)
{
    /* 定义并初始化日历 */
    lv_obj_t* calendar = lv_calendar_create(lv_scr_act());
    /* 设置日历的大小 */
    lv_obj_set_size(calendar, scr_act_height() * 0.85, scr_act_height()
                    * 0.85);
    lv_obj_center(calendar);
    /* 设置日历的日期 */
    lv_calendar_set_today_date(calendar, 2022, 4, 7);
    /* 设置日历显示的月份 */
    lv_calendar_set_showed_date(calendar, 2022, 4);
    /* 设置日历头 */
    lv_calendar_header_dropdown_create(calendar);

    /* 设置第一个日期 */
    highlight_days[0].year = 2022;
    highlight_days[0].month = 4;
    highlight_days[0].day = 5;
    /* 设置第二个日期 */
    highlight_days[1].year = 2022;
```

```
    highlight_days[1].month = 4;
    highlight_days[1].day = 6;

    lv_calendar_set_highlighted_dates(calendar, highlight_days,2);
    /* 更新日历参数 */
    lv_obj_update_layout(calendar);
}
```

从上述源码可知：首先定义一个静态或者全局变量 highlight_days，然后设置该结构体的成员变量分别为 2022.4.5 和 2022.4.6 两个日期，最后调用函数 lv_calendar_set_highlighted_dates 设置这两个日期为高亮状态。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 25.2.3.1 所示：

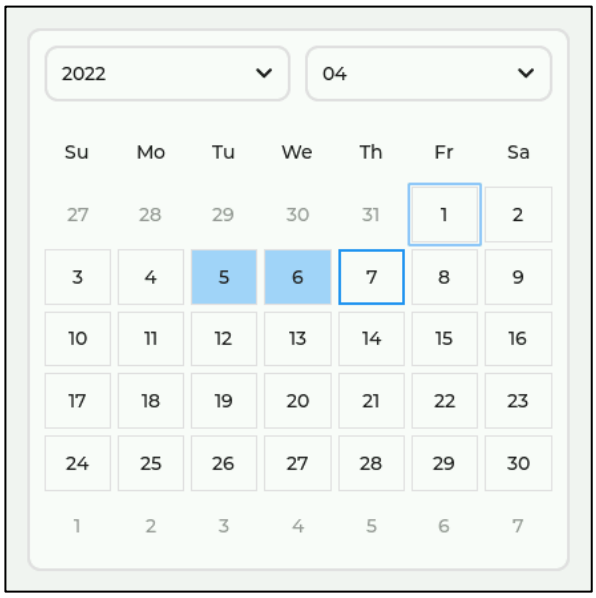


图 25.2.3.1 设置某个日期为高亮

上图可知：我们设置 2022 年 4 月 5 和 6 号为高亮。

25.2.4 设置日名

前面我们了解到 Calendar 部件初始化的时候它的日名都是以英文形式展示(Su、Mo、Tu、We、Th、Fr 和 Sa)，如果用户想设置成中文的日名，可调用 lv_calendar_set_day_names 设置，下面我们使用上述的函数来设置日历的中文日名，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())
LV_FONT_DECLARE(myFont14)
static lv_calendar_date_t highlight_days[2];/* 定义的日期,必须用全局或静态定义 */
const char* day_names[7] = { "一", "二","三","四","五","六","七", };
void lv_mainstart(void){
    /* 定义并初始化日历 */
    lv_obj_t* calendar = lv_calendar_create(lv_scr_act());
    /* 设置字体 */
```

```
    lv_obj_set_style_text_font(calendar, &myFont14, LV_STATE_DEFAULT);
    /* 设置日历的大小 */
    lv_obj_set_size(calendar, scr_act_height() * 0.85, scr_act_height()
                                                        * 0.85);
    lv_obj_center(calendar);
    /* 设置日历的日期 */
    lv_calendar_set_today_date(calendar, 2022, 4, 7);
    /* 设置日历显示的月份 */
    lv_calendar_set_showed_date(calendar, 2022, 4);
    /* 设置日历头 */
    lv_calendar_header_dropdown_create(calendar);
    highlight_days[0].year = 2022;  /* 设置第一个日期 */
    highlight_days[0].month = 4;
    highlight_days[0].day = 5;
    highlight_days[1].year = 2022;  /* 设置第二个日期 */
    highlight_days[1].month = 4;
    highlight_days[1].day = 6;
    lv_calendar_set_highlighted_dates(calendar, highlight_days, 3);
    /* 设置日名 */
    lv_calendar_set_day_names(calendar, day_names);
    /* 更新日历参数 */
    lv_obj_update_layout(calendar);
}
```

从上述源码可知：首先我们设置 Calendar 部件字库为 myFont14，该字库包含中文字体，然后设置 day_names 数组参数，这些参数都是中文字体，其次调用函数 lv_calendar_set_day_names 设置日历的日名，最后更新日历参数。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 25.2.4.1 所示：



图 25.2.4.1 设置日历的日名

从上图可知：我们把 Calendar 部件的日名为中文日名。值得注意的是：上方年份和月份的下拉列表的方向图标消失了，因为我们使用的是自定义的字库，该字库没有包含图标字体，所以该下拉列表的方向图标字体消失了。

25.3 日历部件的 API 函数

LVGL 提供给用户的 Calendar 部件相关函数，如表 25.3.1 所示：

函数	描述
lv_calendar_create()	创建日历对象
lv_calendar_set_today_date()	设置今天的日期
lv_calendar_set_showed_date()	设置当前显示日期
lv_calendar_set_highlighted_dates()	设置高亮显示的日期
lv_calendar_set_day_names()	设置日名的名称
lv_calendar_get_btnmatrix()	获取日历的按钮矩阵对象
lv_calendar_get_today_date()	获取今天的日期
lv_calendar_get_showed_date()	获取当前显示的日期
lv_calendar_get_highlighted_dates()	获取突出显示的日期
lv_calendar_get_highlighted_dates_num()	获取突出显示的日期的数量
lv_calendar_get_pressed_date()	获取当前紧迫的日期

表 25.3.1.1 Calendar 部件 API 函数

接下来我们来讲解 LVGL 的 Calendar 部件常用的函数有哪些：

1. lv_calendar_create 函数

该函数的作用是创建一个 Calendar 部件，该函数原型如下所示：

```
lv_obj_t * lv_Calendar_create(lv_obj_t * parent)
```

该函数的形参，如表 25.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 25.3.2 函数 lv_calendar_create()形参描述

返回值：返回 Calendar 部件对象。

2. lv_calendar_set_today_dat 函数

该函数的作用是设定今天的日期，该函数原型如下所示：

```
void lv_calendar_set_today_date(   lv_obj_t *obj,
                                   uint32_t year,
                                   uint32_t month,
                                   uint32_t day)
```

该函数的形参，如表 25.3.3 所示：

参数	描述
obj	指向日历对象的指针
year	年
month	月
day	日

表 25.3.3 函数 lv_calendar_set_today_date()形参描述

返回值：无。

3. lv_calendar_set_showed_date 函数

该函数的作用是设置当前显示的日期，该函数原型如下所示：

```
void lv_calendar_set_showed_date(  lv_obj_t *obj,
```

```
                                    uint32_t year,
                                    uint32_t month)
```

该函数的形参，如表 25.3.4 所示：

参数	描述
obj	指向日历对象的指针
year	年
month	月

表 25.3.4 函数 lv_calendar_set_showed_date()形参描述

返回值：无。

4. lv_calendar_set_highlighted_dates 函数

该函数的作用是设置高亮显示的日期，该函数原型如下所示：

```
void lv_calendar_set_highlighted_dates(lv_obj_t *obj,
                                       lv_calendar_date_t highlighted[],
                                       uint16_t date_num)
```

该函数的形参，如表 25.3.5 所示：

参数	描述
obj	指向日历对象的指针
highlighted	年
date_num	月

表 25.3.5 函数 lv_calendar_set_highlighted_dates()形参描述

返回值：无。

5. lv_calendar_set_day_names 函数

该函数的作用是设置日名的名称，该函数原型如下所示：

```
void lv_calendar_set_day_names(lv_obj_t *obj, const char **day_names)
```

该函数的形参，如表 25.3.6 所示：

参数	描述
obj	指向日历对象的指针
day_names	指向具有名称的数组的指针

表 25.3.6 函数 lv_calendar_set_day_names()形参描述

返回值：无。

6. lv_calendar_get_btnmatrix 函数

该函数的作用是获取日历的按钮矩阵对象，该函数原型如下所示：

```
lv_obj_t *lv_calendar_get_btnmatrix(const lv_obj_t *obj)
```

该函数的形参，如表 25.3.7 所示：

参数	描述
obj	指向日历对象的指针

表 25.3.7 函数 lv_calendar_get_btnmatrix()形参描述

返回值：指向按钮矩阵的指针。

7. lv_calendar_get_today_date 函数

该函数的作用是获取今天的日期，该函数原型如下所示：

```
const lv_calendar_date_t *lv_calendar_get_today_date(const lv_obj_t *calendar)
```

该函数的形参，如表 25.3.8 所示：

参数	描述
calendar	指向日历对象的指针

表 25.3.8 函数 lv_calendar_get_today_date()形参描述

返回值：指向包含今天日期的 lv_calendar_date_t 变量的指针。

8. lv_calendar_get_showed_date 函数

该函数的作用是获取当前显示的日期，该函数原型如下所示：

```
const lv_calendar_date_t *lv_calendar_get_showed_date(const lv_obj_t *calendar)
```

该函数的形参，如表 25.3.9 所示：

参数	描述
calendar	指向日历对象的指针

表 25.3.9 函数 lv_calendar_get_showed_date()形参描述

返回值：指向包含日期的 lv_calendar_date_t 变量的指针将显示出来。

9. lv_calendar_get_highlighted_dates 函数

该函数的作用是获取突出显示的日期，该函数原型如下所示：

```
lv_calendar_date_t *lv_calendar_get_highlighted_dates(const lv_obj_t *calendar)
```

该函数的形参，如表 25.3.10 所示：

参数	描述
calendar	指向日历对象的指针

表 25.3.10 函数 lv_calendar_get_highlighted_dates()形参描述

返回值：指向包含日期的 lv_calendar_date_t 数组的指针。

10. lv_calendar_get_highlighted_dates_num 函数

该函数的作用是获取突出显示的天数，该函数原型如下所示：

```
uint16_t lv_calendar_get_highlighted_dates_num(const lv_obj_t *calendar)
```

该函数的形参，如表 25.3.11 所示：

参数	描述
calendar	指向日历对象的指针

表 25.3.11 函数 lv_calendar_get_highlighted_dates_num()形参描述

返回值：获取突出显示的天数。

11. lv_calendar_get_pressed_date 函数

该函数的作用是获取当前紧迫的一天，该函数原型如下所示：

```
lv_res_t lv_calendar_get_pressed_date( const lv_obj_t *calendar,
                                       lv_calendar_date_t *date)
```

该函数的形参，如表 25.3.11 所示：

参数	描述
calendar	指向日历对象的指针
date	把按下的日期存储在这里

表 25.3.11 函数 lv_calendar_get_pressed_date()形参描述

返回值：LV_RES_OK:有一个有效的按下日期；LV_RES_INV:没有按下的数据。

25.4 日历部件的实验

25.4.1 硬件设计

1. 例程功能

本实验主要测试 LVGL 文件系统，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建日历部件并在日历的头部，当我们按下某个日期时，系统会获取用户按下的日期并在指定的 lable 部件中显示该日期。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 25 lv_calendar(日历)》。

25.4.2 软件设计

25.4.2.1 程序流程图

本实验的程序流程图，如下图 25.4.2.1.1 所示：

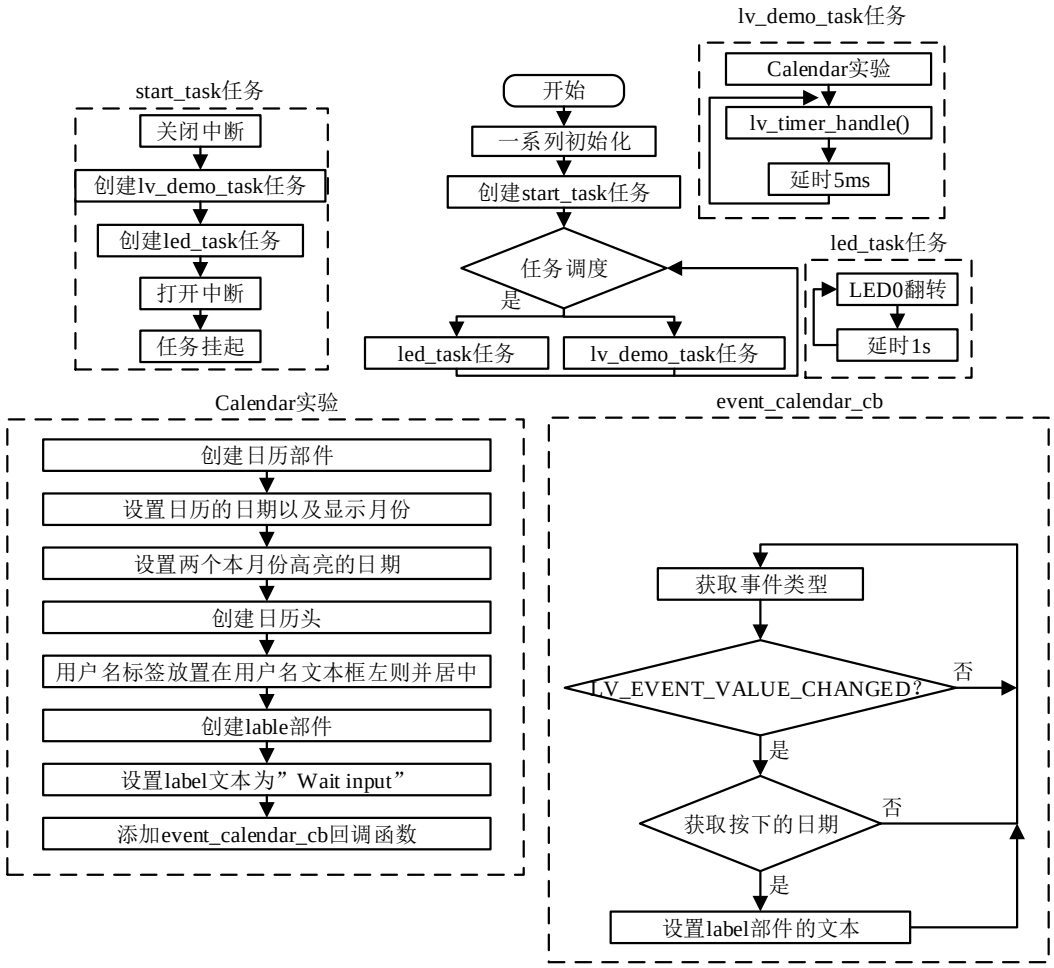


图 25.4.2.1.1 lv_calendar 实验流程图

25.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
```

```
 */
void lv_mainstart(void)
{
    lv_example_calendar();
}
```

此函数实际调用 lv_example_calendar 函数编写 LVGL 代码，该函数源码如下所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_calendar(void)
{
    /* 定义并初始化日历 */
    calendar = lv_calendar_create(lv_scr_act());
    /* 设置日历的大小 */
    lv_obj_set_size(calendar, scr_act_height() * 0.85, scr_act_height()* 0.85);
    lv_obj_center(calendar);
    /* 设置日历的日期 */
    lv_calendar_set_today_date(calendar, 2022, 4, 7);
    /* 设置日历显示的月份 */
    lv_calendar_set_showed_date(calendar, 2022, 4);
    /* 设置日历头 */
    lv_calendar_header_dropdown_create(calendar);


    highlight_days[0].year = 2022;  /* 设置第一个日期 */
    highlight_days[0].month = 4;
    highlight_days[0].day = 5;
    highlight_days[1].year = 2022;  /* 设置第二个日期 */
    highlight_days[1].month = 4;
    highlight_days[1].day = 6;

    lv_calendar_set_highlighted_dates(calendar, highlight_days, 3);
    /* 更新日历参数 */
    lv_obj_update_layout(calendar);


    lv_obj_t* label = lv_label_create(lv_scr_act());     /* 定义并创建标签 */
    lv_obj_set_width(label, lv_obj_get_x(calendar));     /* 设置标签宽度 */
    lv_obj_align(label, LV_ALIGN_LEFT_MID, 0, 0);        /* 设置标签位置 */
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
```

```
    lv_label_set_text(label, "Wait input...");          /* 设置标签文本 */
    /* 设置日历回调 */
    lv_obj_add_event_cb(calendar, event_calendar_cb, LV_EVENT_ALL, label);
}
```

此函数我们可以分为两个部分来讲解，第一部分：首先调用 lv_calendar_create 函数创建日历部件，然后调用函数 lv_calendar_set_today_date 设置当前的日期为 2022.4.7，其次调用函数 lv_calendar_set_showed_date 设置日历当前显示的年月为 2022.4，如果我们不设置当前显示的年月份，则系统自动显示的年月为 2020.1，最后创建日历头以及设置 2022.4.5 和 2022.4.6 日期为高亮日期；第二部分：首先调用函数 lv_label_create 创建 label 部件，其次设置 lable 部件的文本为“Wait input…”字符串，最后添加 calendar 部件的回调函数 event_calendar_cb。下面我们来讲解 event_calendar_cb 回调函数到底实现了什么功能，如下源码所示：

```
/**
 * @brief  日历回调
 * @param  无
 * @return 无
 */
static void event_calendar_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* obj = lv_event_get_target(e);
    lv_obj_t* label = (lv_obj_t*)e->user_data;
    lv_calendar_date_t date_temp;
    char buf[11];

    if (LV_EVENT_VALUE_CHANGED == code)
    {
        if (LV_RES_OK == lv_calendar_get_pressed_date(calendar, &date_temp))
         {
            lv_calendar_set_today_date(calendar, date_temp.year,
                                       date_temp.month, date_temp.day);
            lv_snprintf(buf, sizeof(buf), "%d/%02d/%02d", date_temp.year,
                        date_temp.month, date_temp.day);
            lv_label_set_text(label, buf);
        }
    }
}
```

此回调函数非常简单，主要判断日历部件的 LV_EVENT_VALUE_CHANGED 事件是否响应，如果该事件响应，则调用函数 lv_calendar_get_pressed_date 获取按下的日期并把获取的日期显示在 lable 部件当中。

25.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 25.4.3.1 所示：

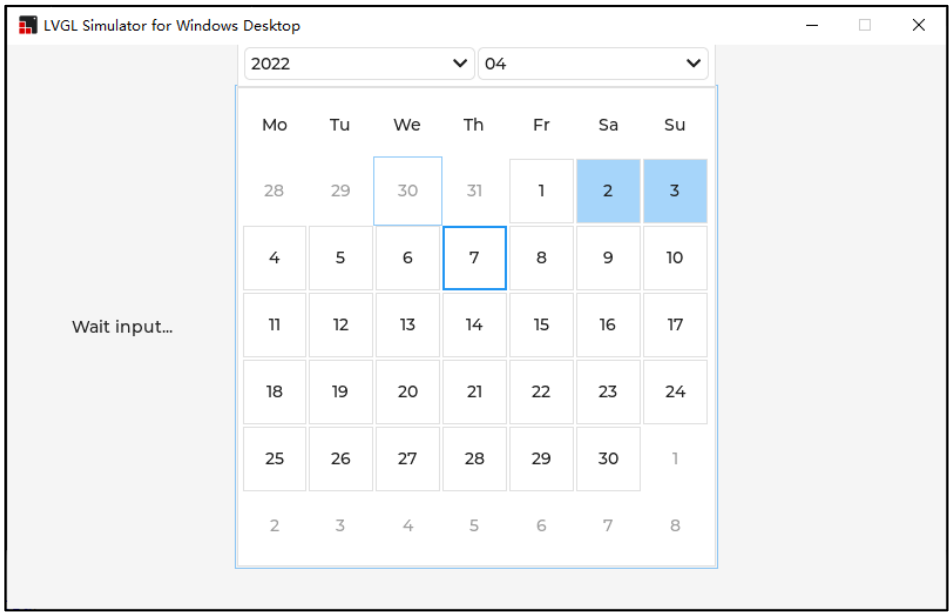


图 25.4.3.1 calendar 部件实验

从上图可知，我们就会发现 LVGL 的 Calendar 的 bug，在程序中，我们明明已经设置 4.5 和 4.6 的日期为高亮日期，而系统只把 4.2 和 4.3 日期为高亮日期，显然地，这个效果并不是我们所需要的。

第二十六章 图表部件(lv_chart)

图表部件，数据可视化，图形化显示是当前工业领域、商业领域、金融领域等不可或缺的元素，通常采用图表进行数据可视化展示，直观地显示数据、对比数据、分析数据。图表部件就是具有这一能力的部件，很多项目开发中都会需求到图表部件。LVGL 的图表部件支持折线图(点与线连接或者在折线上画点)和条形图这两种图表，本章主要讲解图表部件的相关知识以及接口的使用。

本章分为如下几部分：

26.1 图表部件的组成

26.2 图表部件的相关知识

26.3 图表部件 API 函数

26.4 图表控件的实验

26.1 图表部件的组成

LVGL 图表部件是以 lv_chart 字符串为命名，该部件所绘画的图表可以添加分割线、2 个 Y 轴、标志刻度和刻度文本、游标以及滚动和缩放功能，这些功能会在以后的章节会讲解，本小节主要讲解 chart 部件组成部分到底有哪些？。

chart 部件是由六个部分组成，这些部分如表 26.1.1 所示：

组成部分	描述
LV_PART_MAIN	主要设置背景属性和线条(用于分隔线)相关的样式属性
LV_PART_SCROLLBAR	主要设置缩放图表时使用的滚动条
LV_PART_ITEMS	根据图表类型(折线图和条形图)设置
	折线图：设置线属性，如宽度、高度、bg_color 和半径
	条形图：背景属性
LV_PART_INDICATOR	设置折线图和散点图上的点(小圆或小方)的属性
LV_PART_CURSOR	设置游标，比如它的宽度、高度、bg_color 和半径
LV_PART_TICKS	设置线条和文本样式属性用于设置刻度的样式

表 26.1.1 chart 部件组成部分描述

从上表可知：如果用户想设置图表的样式属性，则必须根据组成部分来设置相关样式属性。至于图表的样式属性的设置方法，请大家参考第六章 6.3 小节。

26.2 图表部件的相关知识

26.2.1 图表部件的主要功能

图表部件可分为两个部分来讲解，第一是图表的主要功能，该功能是图表部件不可或缺的功能，简单来说：图表部件必须设置的功能；第二个是图表的次要功能，该功能主要让图表更好的可视化，如标注刻度和刻度文本、缩放和游标等功能。下面我们开始讲解图表的主要功能有哪些，如下所示。

1. 图表的类型

前面我们已经讲解到图表的类型分为两种，这两种类型分为折线图和条形图，这些图表的类型是由函数 lv_chart_set_type 设置，该函数可设置四种模式，这些模式如表 26.2.1.1 所示：

模式	描述
LV_CHART_TYPE_NONE	隐藏数据显示
LV_CHART_TYPE_LINE	点与点连接成线(折线图)
LV_CHART_TYPE_BAR	条形图
LV_CHART_TYPE_SCATTER	X/Y 图画点和点之间的线

表 26.2.1.1 图表的模式

上表的模式主要配置图表数据绘画类型，一般常用的模式是 LV_CHART_TYPE_LINE 和 LV_CHART_TYPE_BAR。下面我们使用一个示意图来描述这两种模式的图表，如图 26.2.1.1 所示：

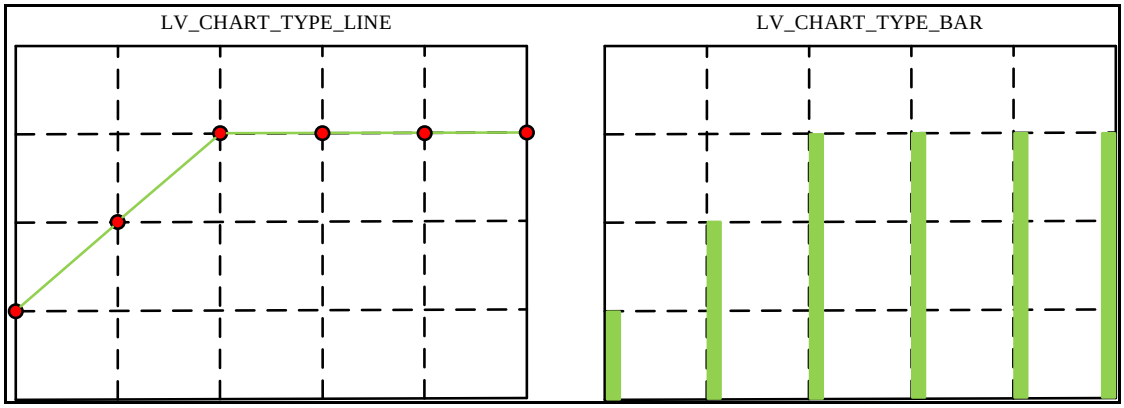


图 26.2.1.1 折线图和条形图

2. 添加数据序列

数据序列是指绘制在图表上的一组相关值。在不同类型的图表中，数据序列的表示方式各不相同：

(1) 条形图：数据序列由一系列具有相同填充（颜色或纹理）的条形表示。

(2) 折线图（也称为图形）：数据序列用单线表示。

图表设置成条形图或者折线图的方法，前面我们已经讲解过，这里笔者无需重复讲解，数据序列简单来讲就是图表具有多少通道绘制数值，如示波器一般可显示两条或者两条以上的通道。图表的数据序列可调用函数 lv_chart_add_series 添加，该函数具有三个形参，其中第一个形参指向图表对象；第二个形参是设置数据序列的颜色；而第三个形参表示数据缩放方向，这些方向如表 26.2.1.2 所示：

数据缩放方向	描述
LV_CHART_AXIS_PRIMARY_Y	左轴
LV_CHART_AXIS_SECONDARY_Y	右轴
LV_CHART_AXIS_PRIMARY_X	底部
LV_CHART_AXIS_SECONDARY_X	顶部

表 26.2.1.2 数据缩放方向

上述的宏定义与数据缩放有关，数据缩放功能我们以后会讲解到，这里我们了解即可。

3. 添加数据

默认情况下，图表部件只支持 10 个数据点，当然啦，我们也可以手动设置图表的数据点的数量，这个设置方法，我们以后会去讲解。图表部件添加数据的方法有五种，下面我们分别地讲解这五种添加数据的过程，如下所示：

第一种： lv_chart_set_ext_y_array/lv_chart_set_ext_x_array 函数

这两个函数都是添加数据到图表当中。下面我们分别讲解这两个函数的区别是什么？

函数 lv_chart_set_ext_y_array 是为 Y 轴数据点设置一个外部阵列，该函数一般用在折线图和条形图类型。函数 lv_chart_set_ext_x_array 是为 X 轴数据点设置一个外部阵列，该函数一般用在 LV_CHART_TYPE_SCATTER 类型上。

下面我们编写一个简单实例来讲解 lv_chart_set_ext_y_array 函数到底如何使用，如下源码所示：

```
/* 示例数据 */
static const lv_coord_t example_data[] = {
    25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
};
void lv_mainstart(void)
{
    lv_obj_t* chart;
    /* 创建图表部件 */
    chart = lv_chart_create(lv_scr_act());
    /* 设置图表大小 */
    lv_obj_set_size(chart, 200, 150);
    /* 中间对齐 */
    lv_obj_center(chart);
    /* 设置折线图类型 */
    lv_chart_set_type(chart, LV_CHART_TYPE_LINE);
    /* 添加数据序列 */
    lv_chart_series_t* ser1 = lv_chart_add_series(chart,
                                                  lv_palette_main(LV_PALETTE_RED),
                                                  LV_CHART_AXIS_SECONDARY_X);
    /* 为 Y 轴数据点设置一个外部阵列 */
    lv_chart_set_ext_y_array(chart, ser1, (lv_coord_t*)example_data);
    /* 更新图表 */
    lv_chart_refresh(chart);
}
```

从上述源码可知：我们首先创建图表，然后把图表设置图表的类型为折线图，其次图表添加数据序列，最后添加数据到图表当中。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 26.2.1.2 所示：

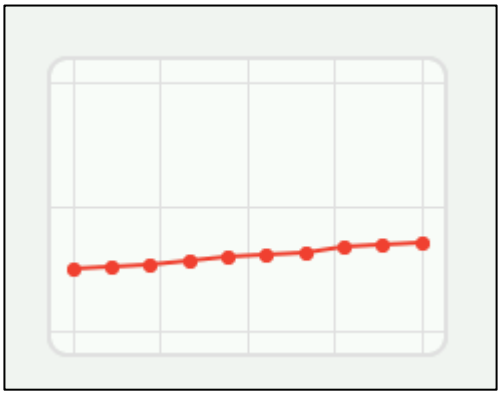


图 26.2.1.2 折线图示意图

上图的数据都是由 example_data 数组提供给图表部件。

第二种：lv_chart_set_next_value 函数

该函数主要设置图表的下一个数值，使用该函数添加数据到图表的模式具有两种，如表 26.2.1.3 所示：

模式	描述
LV_CHART_UPDATE_MODE_SHIFT	将旧数据移到左边，并将新数据添加到右边
LV_CHART_UPDATE_MODE_CIRCULAR	以循环的方式添加新数据，就像心电图一样

表 26.2.1.3 添加数据模式

上述的模式是由 lv_chart_set_update_mode 设置，图表会根据这些模式来设置添加数据的方式。接下来我们编写一个简单的实例来讲解这两个模式的作用，如下源码所示：

```
lv_obj_t* chart;
lv_obj_t* chart1;

static void add_data(lv_timer_t* timer)
{
    LV_UNUSED(timer);
    lv_chart_set_next_value(chart, timer->user_data, lv_rand(20, 90));
}

static void lv_add_data(lv_timer_t* timer)
{
    LV_UNUSED(timer);
    lv_chart_set_next_value(chart1, timer->user_data, lv_rand(20, 90));
}

void lv_mainstart(void)
{
    /* 创建图表部件 */
    chart = lv_chart_create(lv_scr_act());
    /* 设置图表大小 */
    lv_obj_set_size(chart, 200, 150);
    /* 中间对齐 */
    lv_obj_center(chart);
    /* 将旧数据移到左边 */
    lv_chart_set_update_mode(chart,LV_CHART_UPDATE_MODE_SHIFT);
    /* 添加数据序列 */
    lv_chart_series_t* ser1 = lv_chart_add_series(chart,
                              lv_palette_main(LV_PALETTE_RED),
                              LV_CHART_AXIS_SECONDARY_X);
    /* 创建定时器 */
    lv_timer_create(add_data, 200, ser1);
    /* 更新图表 */
    lv_chart_refresh(chart);
```

```
    chart1 = lv_chart_create(lv_scr_act());
    lv_obj_set_size(chart1, 200, 150);
    /* 对齐 */
    lv_obj_align_to(chart1, chart,LV_ALIGN_OUT_BOTTOM_MID,0,20);
    /* 添加数据序列 */
    lv_chart_series_t* ser2 = lv_chart_add_series(chart1,
                              lv_palette_main(LV_PALETTE_RED),
                              LV_CHART_AXIS_SECONDARY_X);
    /* 以循环的方式添加新数据 */
    lv_chart_set_update_mode(chart1, LV_CHART_UPDATE_MODE_CIRCULAR);
    /* 创建定时器 */
    lv_timer_create(lv_add_data, 200, ser2);
    /* 更新图表 */
    lv_chart_refresh(chart1);
}
```

此函数主要创建两个图表，它们分别设置不同的数据模式，然后创建两个定时器分别以 200ms 调用函数 lv_chart_set_next_value 更新数据。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 26.2.1.3 所示：

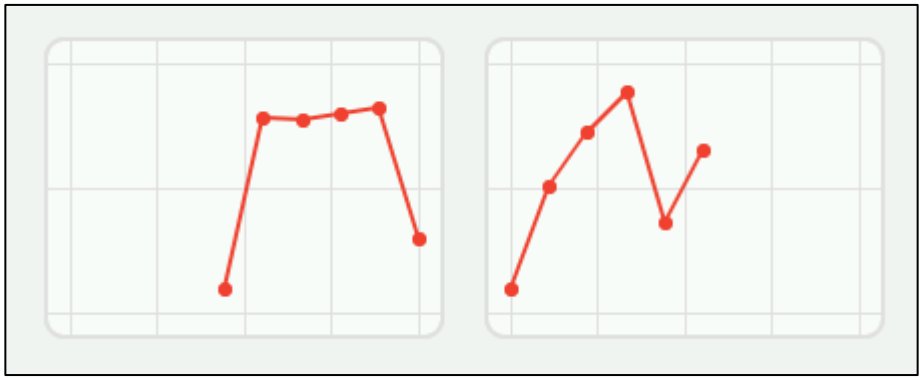


图 26.2.1.3 两种模式下添加数据

上图中，左边的图表是 LV_CHART_UPDATE_MODE_SHIFT 模式添加数据，该模式下的数据是从右边往左边移动，而右边的图表是 LV_CHART_UPDATE_MODE_CIRCULAR 模式添加数据是从左边往右边移动，如医院的心电图。

第三种：lv_chart_set_all_value 函数

该函数很简单理解，它是把所有的数据都初始化为一个数值。前面我们讲解到图表部件只支持 10 个数据点，如果我们使用这个函数初始化数据点，则这些数据点的数值都是一样的。

第四种：lv_chart_set_value_by_id 函数

此函数主要修改某个数据点的数值，前面我们讲到图表部件默认只支持 10 数据点，这些数据点是从 0 开始自增。

第五种：在数组中手动设置

使用 lv_chart_add_series 函数所返回的数据序列指针的成员变量 y_points 或者 x_points 数组设置数值，如 ser2->y_points[0] = 90。

上述五种添加数据的方式常用的是(1)和(2)方式，这些方式都必须在后面调用函数 lv_chart_refresh 更新数据。

4. 设置数据点数量

前面笔者也讲解过，图表部件默认只支持 10 个数据点，如果我们具有 11 个数据，那么图表先显示前 10 个数据，而第 11 个数据会将图表的 10 个数据逐一往左移位，最后把第一个数据点的数值去除了。下面我们使用一个示意图来讲解上述的过程，如图 26.2.1.4 所示：

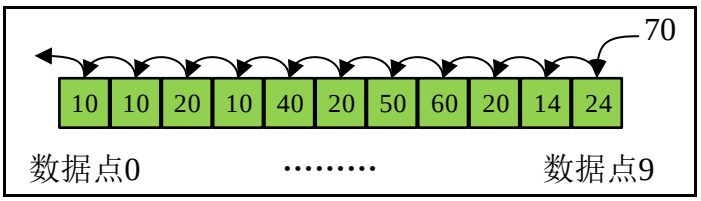


图 26.2.1.4 数据移位

结合我们前面讲解的知识，这些数据点类似于一个数组(其实它是一个连续内存的内存块)，该数组默认长度为 10，所以上图的数据点可以存放 10 个元素，如果第 11 给元素加载到数据点中，则该数组的数值往左移位，最后把原本第一位的元素除去，元素 70 插入该数组的最后一位。上述的过程就是图表部件的添加数据的原理。

虽然图表部件默认只支持 10 个数据点，但是我们可以手动设置图表可支持的数据点，如 5、6、7 等数据点，这些数据点的数量由用户决定。我们调用函数 lv_chart_set_point_count 来设置该图表所支持的数据点。注意：当一个外部缓冲区被分配给一个序列时，这也会影响处理的点数，所以你需要确保外部数组足够大。

26.2.2 图表部件的辅助功能

1. 垂直范围

垂直范围是指图表所添加的数据不能超出这个范围。图表默认的垂直范围为 0~100，我们可以手动调用函数 lv_chart_set_range 设置垂直范围。注意：设置垂直范围的数据序列必须是 LV_CHART_AXIS_PRIMARY 和 LV_CHART_AXIS_SECONDARY。下面我们使用一个示意图来讲解图表的垂直范围的知识，如图 26.2.2.1 所示：

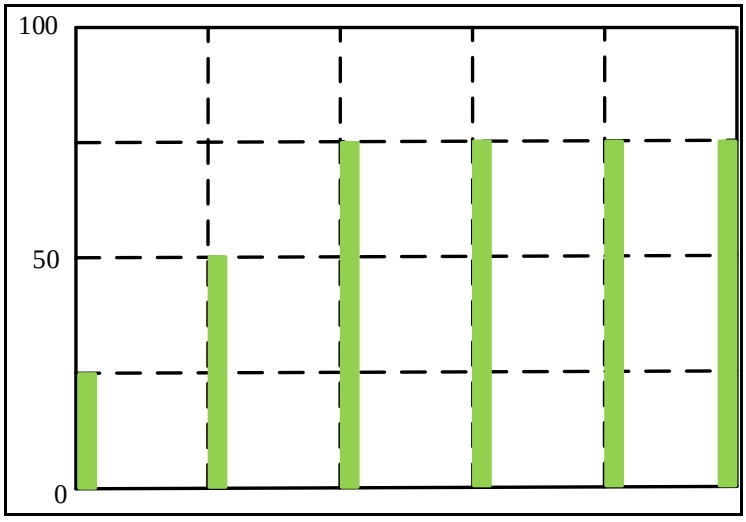


图 26.2.2.1 设置垂直范围

从上图可知：图表部件默认的垂直范围为 0~100。

2. 分隔线

分隔线是用于辅助数据图表提升投射关联的。应用分隔线能够提升数据信息的可阅读性，分隔线给予了二种作用：一是拓宽标值标尺至数据可视化目标中，便于观查数据信息值之尺寸；二是提升数据可视化目标中间的较为基本。

LVGL 的图表部件默认设置为 3 条水平分隔线和 5 条垂直分隔线，如果某边有可见的边框且该边没有填充，则分割线将绘制在边框的顶部。图表部件分割线是调用函数 lv_chart_set_div_line_count 设置，该函数的第二和第三个形参表示水平和垂直分割线的数量，下面我们使用一个示意图来讲解分割线到底是什么，如图 26.2.2.2 所示：

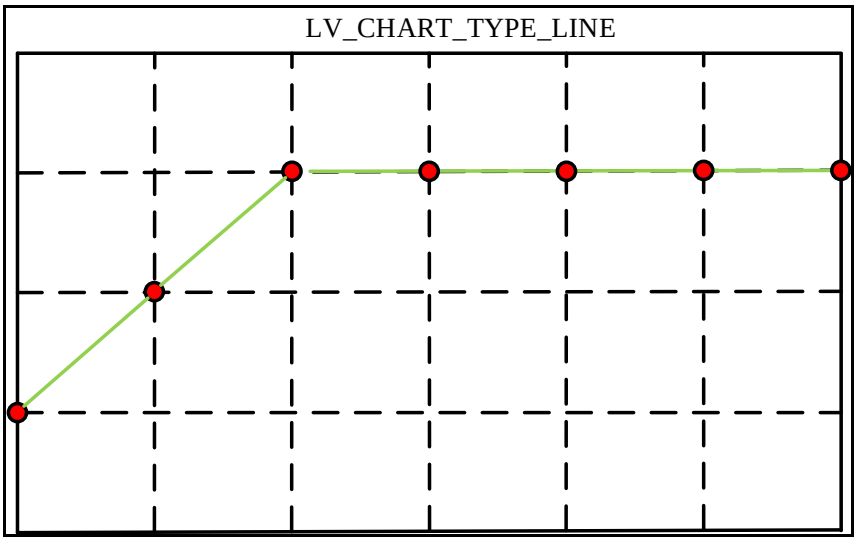


图 26.2.2.2 图表分割线示意图

从上图可知：图表默认设置为 3 条水平分隔线和 5 条垂直分隔线，这些分割线主要拓宽标值标尺至数据可视化目标。

3. 设置默认起始点

如果用户想让绘图从一个点开始而不是默认值 point [0]点开始，我们可以使用函数 lv_chart_set_x_start_point(chart, ser, id)设置一个替代索引，其中 id 是开始绘图的新索引位置。下面我们使用一个示意图来讲解 LV_CHART_UPDATE_MODE_SHIFT/CIRCULAR 这两种模式下设置起始点到底有何不同，如图 26.2.2.3 所示：

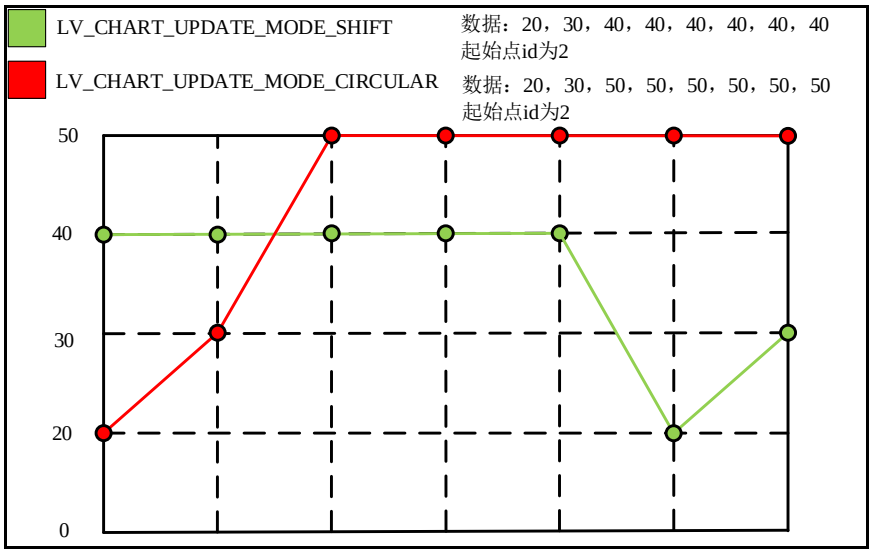


图 26.2.2.3 两种模式下的起始点设置

从上图可知：LV_CHART_UPDATE_MODE_SHIFT 图表是从 id 为 2 的数据点开始，而数据点 1 和数据点 0 是最后才添加到图表当中； LV_CHART_UPDATE_MODE_ CIRCULAR 图表也是一样。

4. 刻度线和刻度线标签

刻度由刻度标签和刻度线组成，若要进一步设置刻度样式，就需要引入刻度定位器(locator)和刻度格式器(formatter)的概念。刻度定位器用来设置刻度线的位置；刻度格式器用来设置刻度标签的显示样式。至于刻度格式器和刻度定位器如何实现，这里我们无需了解，因为 LVGL 作者已经帮我们写好了，我们只调用 API 接口实现即可。

刻度线和刻度线标签的需要用户调用函数 lv_chart_set_axis_tick 添加到当前图表中，该函数具有八个形参，这些形参主要设置刻度线和刻度线的标签。

lv_chart_set_axis_tick 函数

该函数的作用是设置刻度标签和刻度线，该函数原型如下所示：

```
void lv_chart_set_axis_tick(lv_obj_t * obj,
                            lv_chart_axis_t axis,
                            lv_coord_t major_len,
                            lv_coord_t minor_len,
                            lv_coord_t major_cnt,
                            lv_coord_t minor_cnt,
                            bool label_en,
                            lv_coord_t draw_size)
```

该函数的形参，如表 26.2.1.4 所示：

参数	描述
obj	指向图表对象的指针
axis	计数的轴:LV_CHART_AXIS_X / PRIMARY_Y / SECONDARY_Y
major_len	主刻度长度
minor_len	小刻度长度
major_cnt	主刻度的数量
minor_cnt	小刻度的数量
label_en	true:启用在主刻度上绘制标签
draw_size	绘制刻度和标签所需的额外大小

表 26.2.1.4 函数 lv_chart_set_axis_tick()形参描述

返回值：无。

下面我们编写一个简单实例来讲解这个函数到底如何添加刻度标签和刻度线，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* chart;
    /* 创建一个图表 */
    chart = lv_chart_create(lv_scr_act());
    lv_obj_set_size(chart, 200, 150);
    lv_obj_center(chart);
    /* 设置 Y 轴的刻度和刻度标签，主刻度长度为 10，小刻度为 5，
    主刻度数量为 6，小刻度数量为 5 */
    lv_chart_set_axis_tick(chart, LV_CHART_AXIS_PRIMARY_Y, 10, 5, 6, 5,
```

```
                            true, 40);
    /* 设置 x 轴的刻度和刻度标签，主刻度长度为 10，小刻度为 5，
    主刻度数量为 10，小刻度数量为 2 */
    lv_chart_set_axis_tick(chart, LV_CHART_AXIS_PRIMARY_X, 10, 5, 10, 1,
                            true, 30);
    lv_chart_refresh(chart);
}
```

从上述源码可知：我们设置图表的 Y 轴的主刻度数量为 6，小刻度数量为 5，然后设置 X 轴的主刻度数量为 10，小刻度数量为 1。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 26.2.2.4 所示：

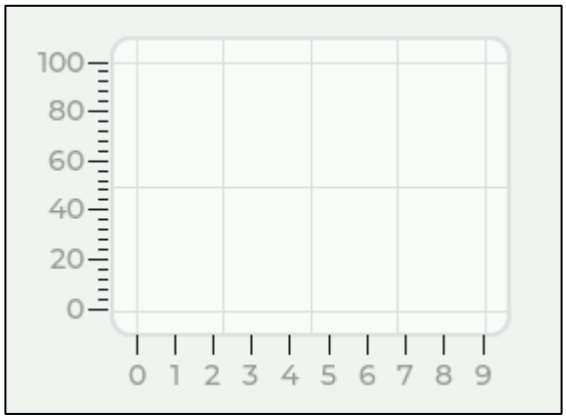


图 26.2.2.4 设置图表的 Y 和 X 轴刻度和刻度标签

由于图表默认的垂直范围为 0~100，我们设置主刻度为 6，该图表的 Y 轴具有五个间隔，所以 100/5 = 20，显然地，图表的 Y 轴增量以 20 步长设置；图表的 X 轴是以 1 步长为增量。

5. 缩放

图表缩放指的是局部放大或缩小图表，可以更方便的查看图表数据。LVGL 的图表缩放接口由两个函数设置，一个是 lv_chart_set_zoom_x 对 X 轴缩放；另一个是 lv_chart_set_zoom_y 对 Y 轴缩放。它们的的第二个形参表示缩放因子，如果缩放因子为 256，则图表没有缩放；如果缩放因子大于 256 且小于等于 512，则图表进行缩放。

6. 游标

图表游标可以查看数据是否溢出指定的范围。图表游标需要用户调用函数 lv_chart_add_cursor 添加，该函数具有三个形参，它们分别指向图表对象、游标颜色以及游标的方向。游标的方向形参可以设置 LV_DIR_NONE/RIGHT/UP/LEFT/DOWN/HOR/VER/ALL 配置项并告诉游标应该在哪个方向绘制。

设置图表游标的位置具有两个方式，如下所示：

① lv_chart_set_cursor_pos(chart, cursor， &point))函数的 Pos 形参是一个指向 lv_point_t 变量的指针。例如 lv_point_t point ={10,20};如果图表被滚动，游标将保持在相同的位置。

② lv_chart_set_cursor_point (chart, cursor, series, point_id)将游标粘在一个点上。 如果点的位置改变(滚动)，游标将随着点移动。

26.3 图表部件 API 函数

LVGL 提供给用户的 chart 部件相关函数，如表 26.3.1 所示：

函数	描述
创建图表函数	
lv_chart_create()	创建图表对象
图表设置及获取函数	
lv_chart_set_type()	为图表设置一个新的类型
lv_chart_set_point_count()	设置数据线上的点数
lv_chart_set_range()	设置轴上的最小和最大 y 值
lv_chart_set_update_mode()	设置图表对象的更新模式
lv_chart_set_div_line_count()	设置水平和垂直分隔线的数量
lv_chart_set_zoom_x()	按 X 方向放大图表
lv_chart_set_zoom_y()	按 Y 方向放大图表
lv_chart_get_zoom_x()	获取 X 缩放数值
lv_chart_get_zoom_y()	获取 Y 缩放数值
lv_chart_set_axis_tick()	设置轴线上的划线数
lv_chart_get_type()	获取图表的类型
lv_chart_get_point_count()	获取图表上每条数据线的数据点数字
lv_chart_get_x_start_point()	获取数据数组中 x 轴起点的当前索引
lv_chart_get_point_pos_by_id()	获取图表中一个点的位置
lv_chart_refresh()	如果图表的数据线发生了变化，请刷新图表
lv_chart_add_series()	添加数据序列到图表
lv_chart_remove_series()	从图表中释放并删除数据序列
lv_chart_hide_series()	隐藏/取消隐藏图表的单个数据系列
lv_chart_set_series_color()	改变一个数据系列的颜色
lv_chart_set_x_start_point()	设置数据数组中 x 轴起点的索引
lv_chart_get_series_next()	下一个数据系列
lv_chart_add_cursor()	添加一个给定颜色的游标
lv_chart_set_cursor_pos()	设置游标在点的坐标
lv_chart_set_cursor_point()	把游标固定在一个点上
lv_chart_get_cursor_point()	获取游标在点的坐标
lv_chart_set_all_value()	用一个值初始化序列中的所有数据点
lv_chart_set_next_value()	根据更新模式策略设置下一个点的 Y 值
lv_chart_set_next_value2()	根据更新模式策略设置下一个点的 X 和 Y 值
lv_chart_set_value_by_id()	直接根据索引设置图表序列中单个点的 y 值
lv_chart_set_value_by_id2()	根据索引直接设置图表系列的单个点的 x 和 y 值(只能在 LV_CHART_TYPE_SCATTER 类型图表设置)
lv_chart_set_ext_y_array()	设置用于图表的 y 数据点的外部数组
lv_chart_set_ext_x_array()	设置用于图表的 x 数据点的外部数组
lv_chart_get_y_array()	获取一个系列的 y 值的数组
lv_chart_get_x_array()	获取一个系列的 x 值的数组
lv_chart_get_pressed_point()	获取当前被压点的指数

表 26.3.1 chart 部件 API 函数

上表的 chart 部件 API 函数前面我们已经讲解了常用的函数，请大家对应官方的文档来学习剩下的函数。

26.4 图表控件的实验

26.4.1 硬件设计

26.4.1.1 例程功能

本实验主要测试 chart 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要测试图表如何添加数据、缩放以及添加刻度等功能。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 26 lv_chart(图表)》。

26.4.2 软件设计

26.4.2.1 程序流程图

本实验的程序流程图，如下图 26.4.2.1.1 所示：

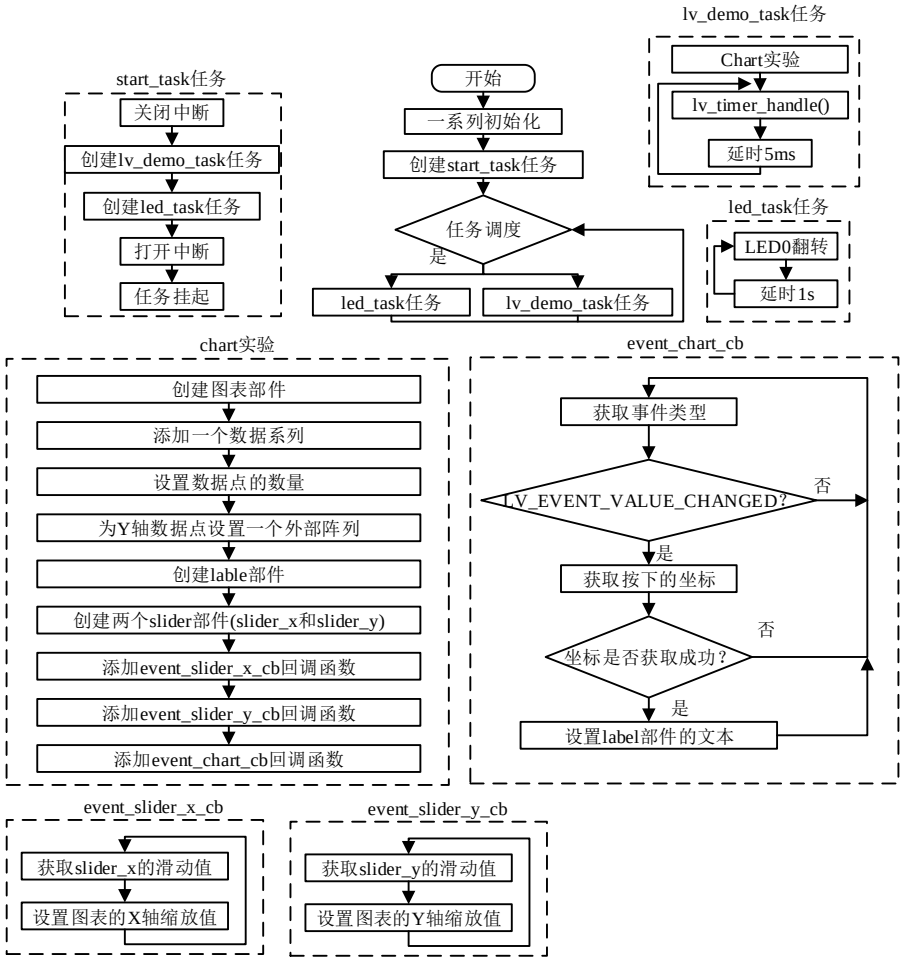


图 26.4.2.1.1 lv_chart 实验流程图

26.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_chart();
}
```

此函数实际调用 lv_example_chart 函数编写 LVGL 代码，该函数源码如下所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_chart(void)
{
    lv_obj_t* chart = lv_chart_create(lv_scr_act()); /* 定义并初始化图表 */
    lv_obj_set_size(chart, scr_act_width(), scr_act_height());/* 设置图表大小 */
    lv_obj_align(chart, LV_ALIGN_TOP_LEFT, 0, 0); /* 设置图表位置 */
    /* 设置左轴 Y 方向的最大最小值 */
    lv_chart_set_range(chart, LV_CHART_AXIS_PRIMARY_Y, -5, 55);
    lv_obj_set_style_size(chart, 0, LV_PART_INDICATOR); /* 不显示数据上的点 */
    /* 在图表中分配并添加数据系列 */
    lv_chart_series_t* ser = lv_chart_add_series(chart,
                                                 lv_palette_main(LV_PALETTE_RED),
                                                 LV_CHART_AXIS_PRIMARY_Y);
    /* 计算数据线上的新点数 */
    uint32_t pcnt = sizeof(example_data) / sizeof(example_data[0]);
    /* 在图表上设置数据线上的点数 */
    lv_chart_set_point_count(chart, pcnt);
    /* 为 Y 轴数据点设置一个外部阵列 */
    lv_chart_set_ext_y_array(chart, ser, (lv_coord_t*)example_data);
    /* 添加光标 */
    lv_chart_cursor_t* cursor = lv_chart_add_cursor(chart,
                                                    lv_palette_main(LV_PALETTE_BLUE),
                                                    LV_DIR_LEFT | LV_DIR_BOTTOM);
    lv_obj_update_layout(chart);                    /* 手动更新图表参数 */
    label = lv_label_create(lv_scr_act());          /* 定义并创建显示值的标签 */
    lv_obj_align(label, LV_ALIGN_TOP_LEFT, 5, 5);  /* 设置标签的位置 */
```

```
    /* 设置标签的字体 */
    lv_obj_set_style_text_font(label, &lv_font_montserrat_18, LV_PART_MAIN);
    lv_label_set_text(label, "");/* 设置标签的文本 */
    /* 定义并创建 X 轴缩放滑杆 */
    lv_obj_t* slider_x = lv_slider_create(lv_scr_act());
    /* 设置滑杆范围 */
    lv_slider_set_range(slider_x, LV_IMG_ZOOM_NONE, LV_IMG_ZOOM_NONE * 10);
    /* 设置滑杆回调 */
    lv_obj_add_event_cb(slider_x, event_slider_x_cb,
                        LV_EVENT_VALUE_CHANGED, chart);
    /* 设置滑杆宽度 */
    lv_obj_set_width(slider_x, lv_obj_get_width(chart)
                     - 3.5 * lv_obj_get_height(chart) / 10);
    /* 设置滑杆高度 */
    lv_obj_set_height(slider_x, lv_obj_get_height(chart) / 10);
    /* 设置滑杆位置 */
    lv_obj_align_to(slider_x, chart, LV_ALIGN_BOTTOM_LEFT,
                    lv_obj_get_height(chart) / 10,
                    -lv_obj_get_height(chart) / 20);

    /* 定义并创建 Y 轴缩放滑杆 */
    lv_obj_t* slider_y = lv_slider_create(lv_scr_act());
    /* 设置滑杆范围 */
    lv_slider_set_range(slider_y, LV_IMG_ZOOM_NONE, LV_IMG_ZOOM_NONE * 10);
    /* 设置滑杆回调 */
    lv_obj_add_event_cb(slider_y, event_slider_y_cb,
                        LV_EVENT_VALUE_CHANGED, chart);
    /* 设置滑杆宽度 */
    lv_obj_set_width(slider_y, lv_obj_get_height(chart) / 10);
    /* 设置滑杆高度 */
    lv_obj_set_height(slider_y, lv_obj_get_height(chart)
                      - 3.5 * lv_obj_get_height(chart) / 10);
    lv_obj_align_to(slider_y, chart, LV_ALIGN_TOP_RIGHT,
                    -lv_obj_get_height(chart) / 20,
                    lv_obj_get_height(chart) / 10);     /* 设置滑杆位置 */

    /* 设置图表回调 */
    lv_obj_add_event_cb(chart, event_chart_cb, LV_EVENT_VALUE_CHANGED, cursor);
}
```

此函数代码有点多，笔者重点讲解几个部分，首先我们添加到图表的数据是由外部阵列提供的，然后创建 label 部件主要为了显示按下图表的坐标，其次 slider_x 和 slider_y 滑动部件主要根据滑动值来缩放图表，最后添加 event_slider_x_cb、event_slider_y_cb 以及 event_chart_cb 回调函数。下面我们分别地讲解这三个回调函数的作用，如下源码所示：

```
/**
 * @brief  图表回调
 * @param  无
 * @return 无
 */
static void event_chart_cb(lv_event_t* e)
{
    lv_obj_t* chart = lv_event_get_target(e);
    lv_event_code_t code = lv_event_get_code(e);
    lv_chart_cursor_t* cursor = (lv_chart_cursor_t*)e->user_data;

    if (LV_EVENT_VALUE_CHANGED == code) {
        uint16_t point_id = lv_chart_get_pressed_point(chart);
        if (LV_CHART_POINT_NONE != point_id) {
            lv_chart_set_cursor_point(chart, cursor, NULL, point_id);
            char buf[20];
            lv_snprintf(buf, sizeof(buf), "(%d, %d)", point_id,
                        example_data[point_id]);
            lv_label_set_text(label, buf);
        }
    }
}

/**
 * @brief  X 轴滑杆回调
 * @param  无
 * @return 无
 */
static void event_slider_x_cb(lv_event_t* e)
{
    lv_obj_t* slider_x = lv_event_get_target(e);
    lv_obj_t* chart = (lv_obj_t*)e->user_data;
    lv_chart_set_zoom_x(chart, lv_slider_get_value(slider_x));
}

/**
 * @brief  Y 轴滑杆回调
 * @param  无
 * @return 无
 */
static void event_slider_y_cb(lv_event_t* e)
{
    lv_obj_t* slider_y = lv_event_get_target(e);
```

```
    lv_obj_t* chart = (lv_obj_t*)e->user_data;
    lv_chart_set_zoom_y(chart, lv_slider_get_value(slider_y));
}
```

从上述源码可知：event_chart_cb 回调函数主要获取图表的坐标并在 label 部件中显示；event_slider_x_cb 回调函数主要根据 slider_x 滑动值缩放图表 X 轴；event_slider_y_cb 回调函数主要根据 slider_y 滑动值缩放图表 Y 轴。

26.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 26.4.3.1 所示：

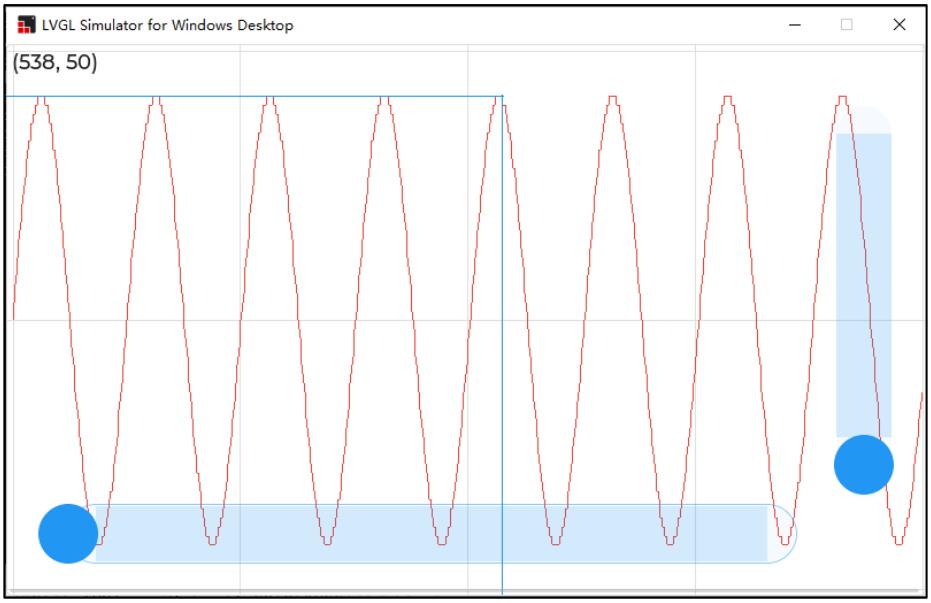


图 26.4.3.1 chart 部件实验

第二十七章 色环部件(lv_colorwheel)

顾名思义，色环部件是允许用户选择一种颜色。该颜色的色相、饱和度和值可以分别选择的。在 UI 设计中，色环部件一般充当颜色选择器的角色。

本章分为如下几部分：

27.1 色环部件的组成

27.2 色环部件的相关知识

27.3 色环部件 API 函数

27.4 色环控件的实验

27.1 色环部件的组成

色环部件使用 lv_colorwheel 来表示，该部件是由 LV_PART_MAIN 和 LV_PART_KNOB 部分组成，LV_PART_MAIN 主要设置 arc_width 色轮的宽度属性(只有这个唯一的属性可设置)，值得注意的是：它不能修改背景属性，因为色环背景有多种颜色组成，所以背景属性显然就失效了； LV_PART_KNOB 主要设置旋钮的矩阵/圆形属性和填充属性，如图 27.1.1 所示：

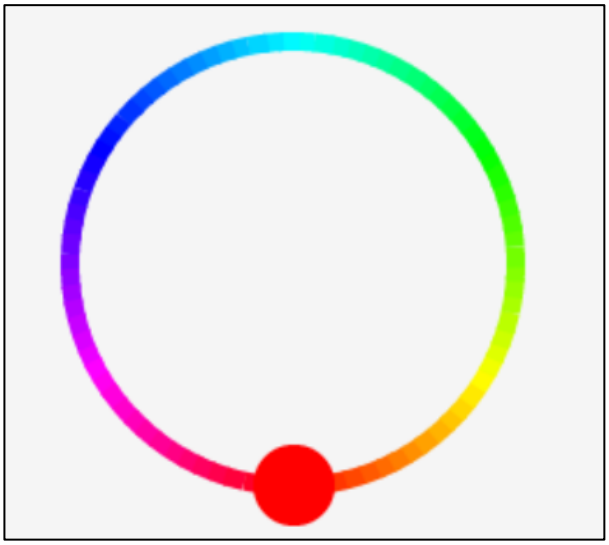


图 27.1.1 colorwheel 部件示意图

从上图可知：LV_PART_MAIN 部分主要修改色轮的宽度，而 LV_PART_KNOB 部分主要设置旋钮的填充属性以及矩阵/圆形属性。这些组成部分如何修改部件的属性，请大家参考第六章的 6.3.3 小节，这里我们不会重复讲解。

27.2 色环部件的相关知识

27.2.1 创建色环部件

前面我们已经学习了多个部件的使用，那么它们的唯一相同的是必须先创建部件，然后设置该部件的相关属性。colorwheel 部件可调用 lv_colorwheel_create(parent, knob_recolor)函数创建，细心的小伙伴可能发现该部件创建的函数具有两个形参，前面学习的部件创建都是一个形参，由于 LVGL 提供多款部件，每一个部件的创建函数的结构都是通用，它们唯一不同就是中

间部件名和形参，有些部件创建只有一个形参，而有些部件具有两个或者两个以上的形参，这类问题主要根据该部件来决定。

上面讲解这么多，主要表达的是每个部件创建的函数结构通用，它们唯一不同就是形参的数量(请参考第六章 6.1.3 小节的创建对象和删除对象)。接下来我们讲解 colorwheel 部件创建函数的形参到底如何设置，该函数具有两个形参，第一个形参指向父类对象，而第二个形参表示旋钮的颜色是否设置为选择的颜色，如果该形参为 true，则旋钮的背景颜色将被设置为当前颜色。

27.2.2 设置色环部件的模式

colorwheel 部件具有三种模式，这些模式主要设置色环的色相、饱和度和数值，这些模式如表 27.2.2.1 所示：

模式	描述
LV_COLORWHEEL_MODE_HUE	色相
LV_COLORWHEEL_MODE_SATURATION	饱和度
LV_COLORWHEEL_MODE_VALUE	数值

表 27.2.2.1 色环模式描述

上表中：色相是色彩的第一个特征，是区分各种不同色彩的最准确的标准，如日光通过三棱镜分解出的红、橙、黄、绿、青、紫六种色相(色环的默认模式)；饱和度是指颜色的纯度，如白色到红色的转变；数值是指黑色到某个颜色的数值，比如黑色到红色的过程就是从 0 到 255 的数值转变。

下面我们以一个简单实例分别地讲解这些模式到底如何使用，如下源码所示：

```
void lv_mainstart(void)
{
    /* 定义并创建色环 */
    lv_obj_t* colorwheel = lv_colorwheel_create(lv_scr_act(), true);
    lv_obj_set_size(colorwheel,100,100);
    lv_obj_center(colorwheel);              /* 设置色环位置 */
    lv_obj_update_layout(colorwheel);       /* 手动更新色环参数 */
    lv_colorwheel_set_mode(colorwheel, LV_COLORWHEEL_MODE_VALUE);

    /* 定义并创建色环 */
    lv_obj_t* colorwheel1 = lv_colorwheel_create(lv_scr_act(), true);
    lv_obj_set_size(colorwheel1, 100, 100);
    /* 设置色环位置 */
    lv_obj_align_to(colorwheel1, colorwheel,LV_ALIGN_OUT_LEFT_MID,-20,0);
    lv_obj_update_layout(colorwheel1);      /* 手动更新色环参数 */
    lv_colorwheel_set_mode(colorwheel1, LV_COLORWHEEL_MODE_HUE);


    /* 定义并创建色环 */
    lv_obj_t* colorwheel2 = lv_colorwheel_create(lv_scr_act(), true);
    lv_obj_set_size(colorwheel2, 100, 100);
    /* 设置色环位置 */
    lv_obj_align_to(colorwheel2, colorwheel, LV_ALIGN_OUT_RIGHT_MID, 20, 0);
```

```
    lv_obj_update_layout(colorwheel2);      /* 手动更新色环参数 */
    lv_colorwheel_set_mode(colorwheel2, LV_COLORWHEEL_MODE_SATURATION);
}
```

此函数主要创建三个色环部件，然后我们调用函数 lv_colorwheel_set_mode 设置三个色环部件的模式分别为数值、色相和饱和度。色相模式的色环在屏幕的左侧；数值模式的色环在屏幕的中间；饱和度的色环在屏幕的右侧。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 27.2.2.1 所示：

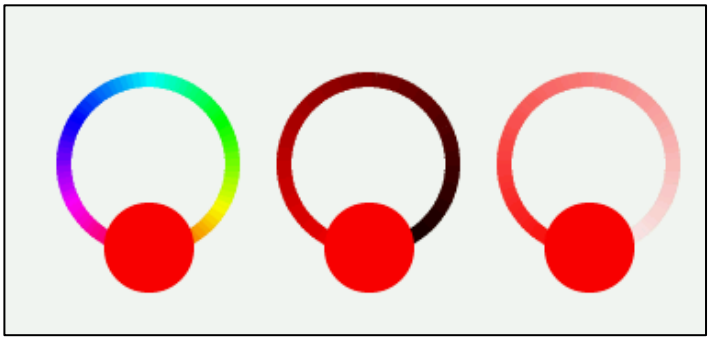


图 27.2.2.1 设置色环的模式

从上图可知：中间和右侧的色环都是红色为基准，如果用户想修改其他颜色作为基准，可调用函数 lv_colorwheel_set_rgb 来设置色环的基准颜色。

27.2.3 色环部件的事件

colorwheel 部件的事件回调一般设置事件类型为 LV_EVENT_VALUE_CHANGED。当然其他事件类型都可以触发。

27.3 色环部件 API 函数

LVGL 提供给用户的 colorwheel 部件相关函数，如表 27.3.1 所示：

函数	描述
lv_colorwheel_create()	创建色环部件
lv_colorwheel_set_hsv()	设置色环当前 hsv
lv_colorwheel_set_rgb()	设置色环的当前颜色
lv_colorwheel_set_mode()	设置当前的颜色模式
lv_colorwheel_get_hsv()	获取色轮当前选中的 hsv
lv_colorwheel_get_rgb()	获取色轮当前选定的颜色
lv_colorwheel_get_color_mode()	获取当前的颜色模式

表 27.3.1 colorwheel 部件 API 函数

接下来我们来讲解 LVGL 的 colorwheel 部件常用的函数有哪些：

1. lv_colorwheel_create 函数

该函数的作用是创建一个 colorwheel 部件，该函数原型如下所示：

```
lv_obj_t * lv_colorwheel_create(lv_obj_t * parent)
```

该函数的形参，如表 27.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 27.3.2 函数 lv_ colorwheel _create()形参描述

返回值：返回 colorwheel 部件对象。

2. lv_colorwheel_set_hsv 函数

该函数的作用是设置色环的当前 hsv，该函数原型如下所示：

```
bool lv_colorwheel_set_hsv(lv_obj_t *obj, lv_color_hsv_t hsv)
```

该函数的形参，如表 27.3.3 所示：

参数	描述
obj	指向色环对象的指针
hsv	当前选中的 hsv

表 27.3.3 函数 lv_colorwheel_set_hsv()形参描述

返回值： true：设置成功，false：设置失败。

3. lv_colorwheel_set_rgb 函数

该函数的作用是设置色环的当前颜色，该函数原型如下所示：

```
bool lv_colorwheel_set_rgb(lv_obj_t *obj, lv_color_t color)
```

该函数的形参，如表 27.3.4 所示：

参数	描述
obj	指向色环对象的指针
color	当前选中的颜色

表 27.3.4 函数 lv_colorwheel_set_rgb()形参描述

返回值： true：设置成功，false：设置失败。

4. lv_colorwheel_set_mode 函数

该函数的作用是设置当前的颜色模式，该函数原型如下所示：

```
void lv_colorwheel_set_mode(lv_obj_t *obj, lv_colorwheel_mode_t mode)
```

该函数的形参，如表 27.3.4 所示：

参数	描述
obj	指向色环对象的指针
mode	颜色模式

表 27.3.4 函数 lv_colorwheel_set_mode()形参描述

返回值：无。

27.4 色环控件的实验

27.4.1 硬件设计

1. 例程功能

本实验主要测试 colorwheel 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建了色环以及基础对象部件，当用户转动色环时，系统会调用函数 lv_colorwheel_get_rgb 获取色环的 RGB 颜色值并设置基础对象部件的背景颜色。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 27 lv_colorwheel(色环)》。

27.4.2 软件设计

27.4.2.1 程序流程图

本实验的程序流程图，如下图 27.4.2.1.1 所示：

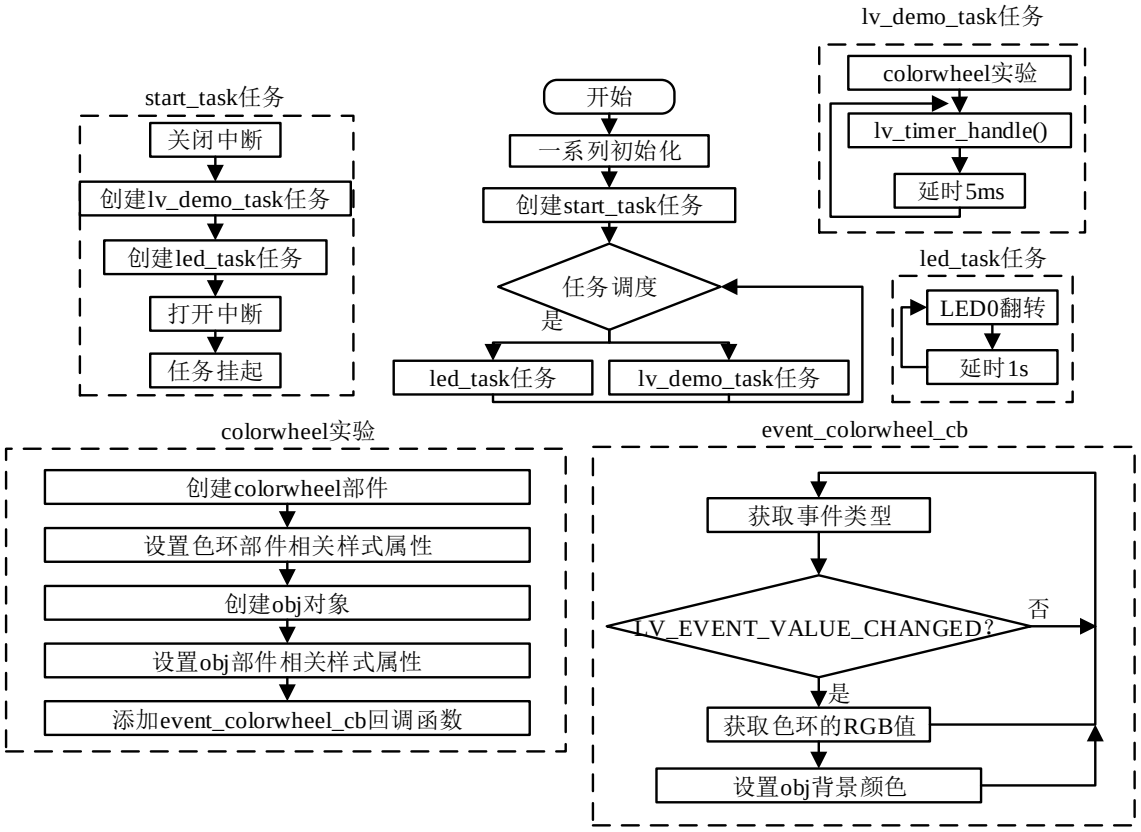


图 27.4.2.1.1 lv_colorwheel 实验流程图

27.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_colorwheel();
}
```

此函数实际调用 lv_example_colorwheel 函数编写 LVGL 代码，该函数源码如下所示：

```
    /**
 * @brief  例
 * @param  无
 * @return 无
```

```
 */
static void lv_example_colorwheel()
{
    /* 定义并创建色环 */
    lv_obj_t* colorwheel = lv_colorwheel_create(lv_scr_act(), true);
    /* 设置色环宽度 */
    lv_obj_set_width(colorwheel, LV_MIN(scr_act_width(),
                     scr_act_height()) * 0.8);
    /* 设置色环高度 */
    lv_obj_set_height(colorwheel, LV_MIN(scr_act_width(),
                     scr_act_height()) * 0.8);
    lv_obj_center(colorwheel); /* 设置色环位置 */
    /* 设置色环宽度 */
    lv_obj_set_style_arc_width(colorwheel,
                               scr_act_height() * 0.1, LV_PART_MAIN);
    lv_obj_update_layout(colorwheel); /* 手动更新色环参数 */

    lv_obj_t* obj = lv_obj_create(lv_scr_act()); /* 定义并创建物体 */
    /* 设置物体宽度 */
    lv_obj_set_width(obj, lv_obj_get_width(colorwheel) - scr_act_height() / 3);
    /* 设置物体高度 */
    lv_obj_set_height(obj, lv_obj_get_height(colorwheel) -
                     scr_act_height() / 3);
    /* 设置物体位置 */
    lv_obj_align_to(obj, colorwheel, LV_ALIGN_CENTER, 0, 0);
    /* 设置物体圆角 */
    lv_obj_set_style_radius(obj, LV_RADIUS_CIRCLE, LV_PART_MAIN);
    lv_obj_set_style_bg_color(obj, lv_colorwheel_get_rgb(colorwheel),
                              LV_PART_MAIN);

    lv_obj_add_event_cb(colorwheel, event_colorwheel_cb,
                        LV_EVENT_VALUE_CHANGED, obj);  /* 设置色环事件回调 */
}
```

此函数的代码非常简单，主要创建色环部件以及 obj 部件，然后添加 event_colorwheel_cb 回调函数，注意：该回调函数的响应事件是 LV_EVENT_VALUE_CHANGED 事件类型。

event_colorwheel_cb 回调函数实现功能如下：

```
/**
 * @brief  色环事件回调
 * @param  e:事件
 * @return 无
 */
static void event_colorwheel_cb(lv_event_t* e)
{
```

```
    lv_obj_t* colorwheel = lv_event_get_target(e);
    lv_obj_t* obj = (lv_obj_t*)e->user_data;

    lv_obj_set_style_bg_color(obj, lv_colorwheel_get_rgb(colorwheel),
                              LV_PART_MAIN);
}
```

此回调函数非常简单，当色环部件响应 LV_EVENT_VALUE_CHANGED 事件时，系统执行 event_colorwheel_cb 回调函数内的代码，这些代码主要作用是获取色环的 RGB 值，然后设置 obj 对象的背景颜色。

27.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 27.4.3.1 所示：

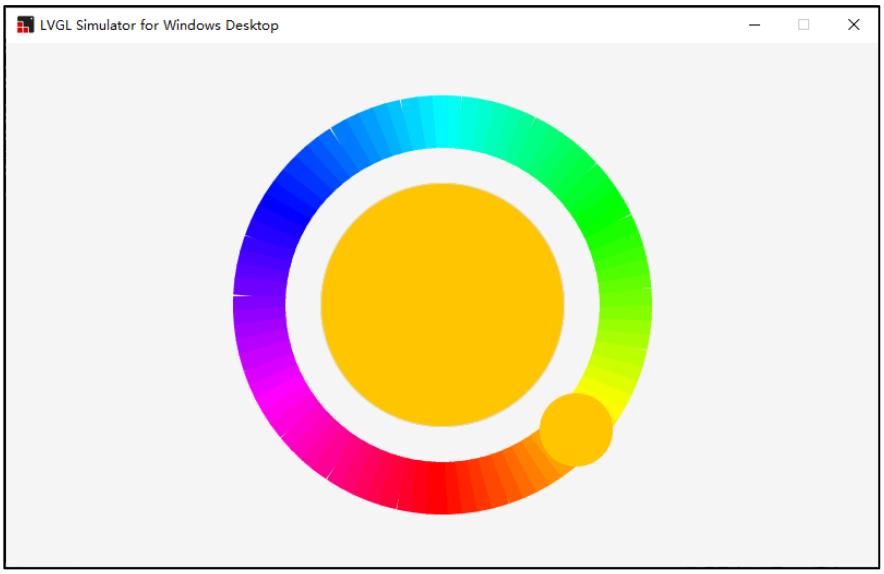


图 27.4.3.1 colorwheel 部件实验

当转动上图色环的旋钮时，中间的 obj 对象得背景颜色根据旋钮的颜色而设置。

第二十八章 图片按键部件(lv_imgbtn)

lv_imgbtn 部件跟 lv_btn 部件是非常相似，它只不过是在 lv_btn 部件的每一个状态基础上增加了图片显示的功能。

本章分为如下几部分：

28.1 图片按键部件的组成
28.2 图片按键部件的相关知识
28.3 图片按键部件 API 函数
28.4 图片按键控件实验

28.1 图片按键部件的组成

imgbtn 部件只有一个主要部分组成，该部分称为 LV_LED_PART_MAIN，它使用了所有典型的背景样式属性。

28.2 图片按键部件的相关知识

28.2.1 图像的来源

前面我们学习 img 部件时，该部件的图像源来源于三个渠道，第一个为 C 数组格式的图像源，第二个为文件系统读取 SD 卡上的图片，而第三个为图标字体。虽然 imgbtn 部件和 img 部件有着某种联系，但是 imgbtn 部件的图像源只支持 C 数组格式和外部存储器读取的方式，它并不支持图标字体显示的方式，这里我们必须要注意的。

imgbtn 部件使用方法：首先调用函数 lv_imgbtn_create 创建对象，然后调用函数 lv_imgbtn_set_src 设置图像源，最后设置该部件的相关属性即可。

下面我们就以一个实例来讲解上述的函数如何使用，如下所示：

```
#ifdef __cplusplus
extern "C" {
#endif
    LV_IMG_DECLARE(img_zdyz);
#ifdef __cplusplus
}
#endif

void lv_mainstart(void)
{
    /* 定义并创建图像按钮 */
    lv_obj_t* imgbtn = lv_imgbtn_create(lv_scr_act());
    /* 设置图像释放时的图像 */
    lv_imgbtn_set_src(imgbtn, LV_IMGBTN_STATE_RELEASED, NULL, &img_zdyz, NULL);
    /* 设置图像按钮大小 */
    lv_obj_set_size(imgbtn, 100, 100);
    /* 设置图像按钮位置 */
```

```
    lv_obj_center(imgbtn);
    lv_obj_set_style_img_recolor_opa(imgbtn, LV_OPA_30, LV_STATE_PRESSED);
    lv_obj_set_style_img_recolor(imgbtn, lv_color_black(), LV_STATE_PRESSED);
    lv_obj_set_style_translate_y(imgbtn, 5, LV_STATE_PRESSED);
}
```

从上述源码可知：首先调用函数 lv_imgbtn_create 创建 imgbtn 部件，然后调用函数 lv_imgbtn_set_src 设置图像源，该函数的第二个形参设置它的状态，而第三个和第五个形参一般设置为 NULL。最后设置按下时的样式属性。注意：图像源必须由 LV_IMG_DECLARE 函数声明才能使用，上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 28.2.1.1 所示：



图 28.2.1.1 imgbtn 部件设置图像源

上图中，当我们按下该图像时，该图像会重新着色并往 y 轴偏移 5 个像素，如图 28.2.1.2 所示：



图 28.2.1.2 imgbtn 部件按下时的状态

上图中，当我们释放 imgbtn 部件时，该图像恢复原样。

28.2.2 添加/清除的状态

默认情况下，imgbtn 部件具有六个状态可选，这些状态如下所示：

① LV_IMGBTN_STATE_RELEASED：释放状态。

② LV_IMGBTN_STATE_PRESSED：按下状态。

③ LV_IMGBTN_STATE_DISABLED：禁用状态。
④ LV_IMGBTN_STATE_CHECKED_RELEASED：点击释放状态。
⑤ LV_IMGBTN_STATE_CHECKED_PRESSED：点击按下状态。
⑥ LV_IMGBTN_STATE_CHECKED_DISABLED：点击禁用状态。

如果用户想清除或者添加上述的某一个状态，可调用函数 lv_obj_add/clear_state 手动设置。

28.3 图片按键部件 API 函数

LVGL 提供给用户的 imgbtn 部件相关函数，如表 28.3.1 所示：

函数	描述
lv_imgbtn_create()	创建 imgbtn 部件
lv_imgbtn_set_src()	为图像按钮的状态设置图像源
lv_imgbtn_set_state()	设置图像按钮的状态
lv_imgbtn_get_src_left()	获取给定状态下的左侧图像
lv_imgbtn_get_src_middle()	获取处于给定状态的中间图像
lv_imgbtn_get_src_right()	在给定的状态下获得右侧的图像

表 28.3.1 imgbtn 部件 API 函数

接下来我们来讲解 LVGL 的 imgbtn 部件常用的函数有哪些：

1. lv_imgbtn_create 函数

该函数的作用是创建一个 imgbtn 部件，该函数原型如下所示：

```
lv_obj_t * lv_imgbtn_create(lv_obj_t * parent)
```

该函数的形参，如表 28.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 28.3.2 函数 lv_imgbtn_create()形参描述

返回值：返回 imgbtn 部件对象。

2. lv_imgbtn_set_src 函数

该函数的作用是为图像按钮的状态设置图像源，该函数原型如下所示：

```
void lv_imgbtn_set_src(lv_obj_t *imgbtn,
                       lv_imgbtn_state_t state,
                       const void *src_left,
                       const void *src_mid,
                       const void *src_right)
```

该函数的形参，如表 28.3.3 所示：

参数	描述	
imgbtn	指向 imgbtn 对象的指针	
state	状态	
	LV_IMGBTN_STATE_RELEASED	释放状态
	LV_IMGBTN_STATE_PRESSED	按下状态
	LV_IMGBTN_STATE_DISABLED	禁用状态
	LV_IMGBTN_STATE_CHECKED_RELEASED	点击释放状态
	LV_IMGBTN_STATE_CHECKED_PRESSED	点击按下状态
	LV_IMGBTN_STATE_CHECKED_DISABLED	点击禁用状态
src_left	指向按钮左侧的图像源的指针(C 数组或文件路径)	
src_mid	指向按钮中间的图像源的指针(C 数组或文件路径)	
src_right	指向按钮右侧的图像源的指针(C 数组或文件路径)	

表 28.3.3 函数 lv_imgbtn_set_src()形参描述

返回值：无。

3. lv_imgbtn_set_state 函数

该函数的作用是设置 imgbtn 部件的状态，该函数原型如下所示：

```
void lv_imgbtn_set_state(lv_obj_t *imgbtn, lv_imgbtn_state_t state)
```

该函数的形参，如表 28.3.4 所示：

参数	描述
imgbtn	指向 imgbtn 对象的指针
state	设置 imgbtn 状态

表 28.3.4 函数 lv_imgbtn_set_state()形参描述

返回值：无。

28.4 图片按键控件实验

28.4.1 硬件设计

1. 例程功能

本实验主要测试 imgbtn 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建一个图片按键部件，当按下图片按键并不释放按键时，图片按键中得图像源由原本的颜色转变成灰色底；当释放图片按键时，恢复默认颜色。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 28 lv_imgbtn(图像按钮)》。

28.4.2 软件设计

28.4.2.1 程序流程图

本实验的程序流程图，如下图 28.4.2.1.1 所示：

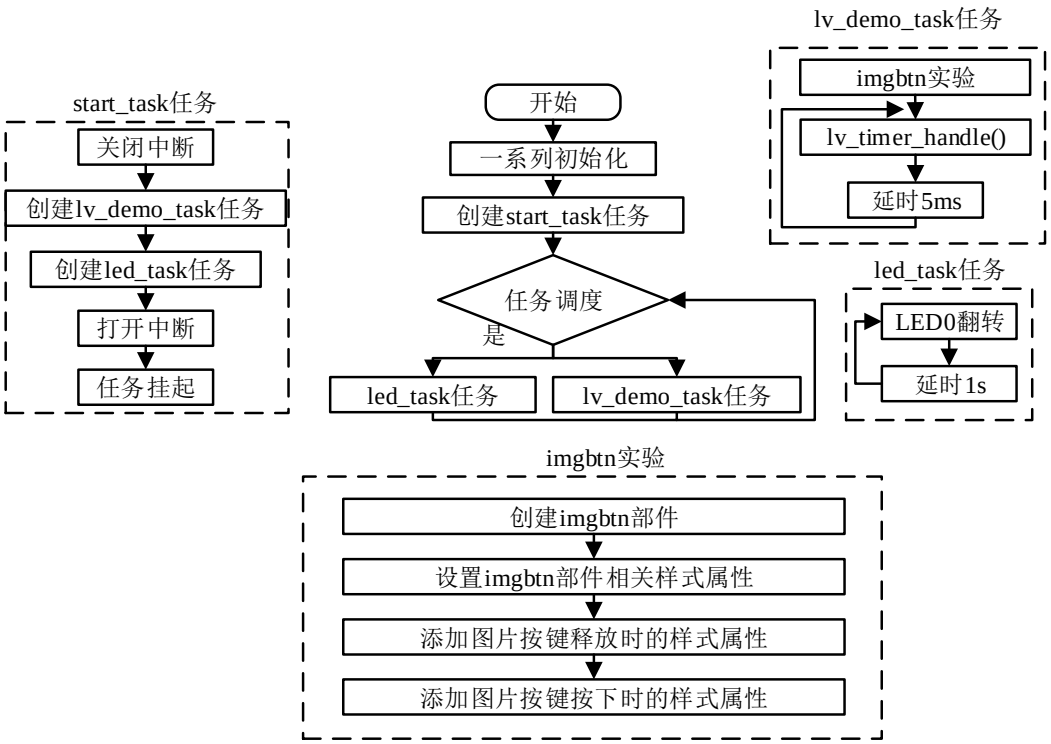


图 28.4.2.1.1 lv_imgbtn 实验流程图

28.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_imgbtn();
}
```

此函数实际调用 lv_example_imgbtn 函数编写 LVGL 代码，该函数源码如下所示：

```
    /**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_imgbtn(void)
{
    lv_obj_t* imgbtn = lv_imgbtn_create(lv_scr_act());/* 定义并创建图像按钮 */
    /* 设置图像按钮图像 */
    lv_imgbtn_set_src(imgbtn, LV_IMGBTN_STATE_RELEASED, NULL, &img_zdyz, NULL);
```

```
    lv_obj_set_size(imgbtn, 100, 100);      /* 设置图像按钮大小 */
    lv_obj_center(imgbtn);                  /* 设置图像按钮位置 */

    static lv_style_prop_t tr_prop[] = {LV_STYLE_TRANSLATE_Y,
                                        LV_STYLE_IMG_RECOLOR_OPA,
                                        LV_STYLE_PROP_INV};
    static lv_style_transition_dsc_t tr;
    lv_style_transition_dsc_init(&tr, tr_prop, lv_anim_path_linear,
                                 100, 0, NULL);
    /* 添加按钮释放风格 */
    lv_obj_set_style_transition(imgbtn, &tr, LV_STATE_DEFAULT);
    /* 添加按钮按下风格 */
    lv_obj_set_style_img_recolor_opa(imgbtn, LV_OPA_30, LV_STATE_PRESSED);
    lv_obj_set_style_img_recolor(imgbtn, lv_color_black(), LV_STATE_PRESSED);
    lv_obj_set_style_translate_y(imgbtn, 5, LV_STATE_PRESSED);
}
```

此函数主要设置图片按键释放时和按下时的样式，当按下图片按键时，图片按键部件的图像源的透明度为 30%，图像源着色为黑色并往 Y 方向移动 5 个像素。

28.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 28.4.3.1 所示：



图 28.4.3.1 imgbtn 部件实验

当按下上图的图片按键时，该部件重新着色为黑色并往 Y 轴移动五个像素，如图 28.4.3.2 所示：



图 28.4.3.2 按下 imgbtn 部件的示意图

第二十九章 键盘部件(lv_keyboard)

键盘部件是一个特殊的按钮矩阵，具有预定义的键映射和其他功能，以实现将文本写入文本区域的虚拟键盘。

本章分为如下几部分：

29.1 键盘部件的组成

29.2 键盘部件的相关知识

29.3 键盘部件 API 函数

29.4 键盘控件实验

29.1 键盘部件的组成

keyboard 部件与按钮矩阵类似，它是由两个部分组成:

- LV_PART_MAIN：设置键盘的主体背景属性。
- LV_PART_ITEMS：设置键盘的按键背景属性和文本属性。

接下来我们以一个实例来讲解 keyboard 部件的样式属性如何设置，如以下所示：

```
void lv_mainstart(void)
{
    /* 定义并创建键盘 */
    lv_obj_t* keyboard = lv_keyboard_create(lv_scr_act());
    /* 设置 LV_PART_MAIN 部分背景颜色 */
    lv_obj_set_style_bg_color(keyboard,lv_palette_main(LV_PALETTE_GREY),
                              LV_PART_MAIN|LV_STATE_DEFAULT);
    /* 设置 LV_PART_ITEMS 部分背景颜色 */
    lv_obj_set_style_bg_color(keyboard, lv_palette_main(LV_PALETTE_RED),
                              LV_PART_ITEMS | LV_STATE_DEFAULT);
}
```

从上述源码可知：我们分别地设置 keyboard 部件组成部分的背景样式，其效果图如图 29.1.1 所示：



如图 29.1.1 设置 keyboard 部件组成部分的背景样式

从上图可知：我们设置 keyboard 部件的背景颜色为灰色，而按键的背景颜色为红色。

29.2 键盘部件的相关知识

29.2.1 键盘部件模式

keyboard 部件是内嵌了多款输入法，用户可以触摸相应的按键进行切换。前面我们也讲解过，LVGL 的触摸输入设备并不是必须的，用户可以使用其他输入设备来控制 UI 界面，如果用户手动切换输入法，可调用函数 lv_keyboard_set_mode 进行切换，该函数的第二个形参表示要切换的输入法模式，这些模式如下所示：

① LV_KEYBOARD_MODE_TEXT_LOWER：显示小写字母。
② LV_KEYBOARD_MODE_TEXT_UPPER：显示大写字母。
③ LV_KEYBOARD_MODE_TEXT_SPECIAL：显示特殊字符。
④ LV_KEYBOARD_MODE_NUMBER：显示数字，+/-符号和十进制点。
⑤ LV_KEYBOARD_MODE_USER_1 ~ LV_KEYBOARD_MODE_USER_4：自定义模式。

默认情况下，keyboard 部件创建后默认选择 LV_KEYBOARD_MODE_TEXT_UPPER 模式，该模式表示键盘显示大写的输入法，如果用户想设置初始的输入法，可调用上面的函数设置它的模式。

29.2.2 指定文本区域

前面我们已经了解了 textarea 部件的相关知识，该部件主要显示输入的内容。如果用户想要把键盘的输入值在 textarea 部件中显示，可调用函数 lv_keyboard_set_textarea 设置。该函数主要把键盘按下的文本传递给 textarea 部件。

接下来我们以一个简单实例来讲解本小节的知识，如下所示：

```
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())

void lv_mainstart(void)
{
    /* 定义并创建键盘 */
    lv_obj_t* keyboard = lv_keyboard_create(lv_scr_act());
    /* 定义并创建文本框 */
    lv_obj_t* textarea = lv_textarea_create(lv_scr_act());
    /* 设置文本框宽度 */
    lv_obj_set_width(textarea, scr_act_width() - 10);
    /* 设置文本框高度 */
    lv_obj_set_height(textarea, (scr_act_height() >> 1) - 10);
    lv_obj_align(textarea,LV_ALIGN_TOP_LEFT,0,0);
    /* 为键盘指定一个文本区域 */
    lv_keyboard_set_textarea(keyboard, textarea);
}
```

从上述源码可知：首先创建 keyboard 和 textarea 部件，然后设置 textarea 部件的相关属性，最后调用函数 lv_keyboard_set_textarea 为键盘指定一个文本区域。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 29.2.2.1 所示：

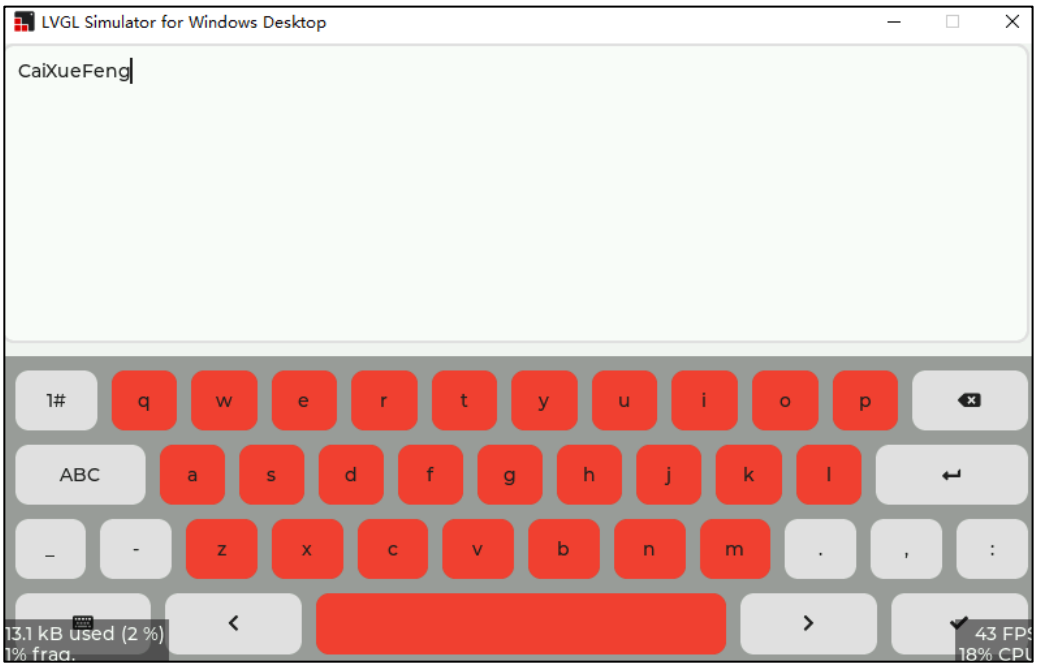


图 29.2.2.1 键盘指定一个文本区域

29.2.3 键弹窗设置

键弹窗是指当按下键盘的某一个按键时，该按键的文本弹出部件边界之外，如下图 29.2.3.1 所示：

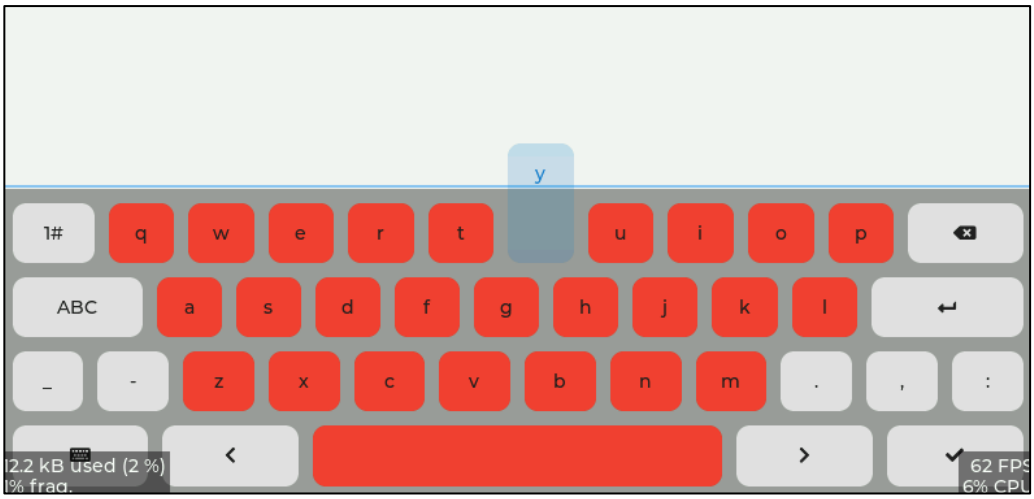


图 29.2.3.1 设置键盘键弹窗

上图中，设置键盘键弹出功能可调用函数 lv_keyboard_set_popovers(keyboard, true)设置，该函数的第二个形参表示是否使能键弹窗，如果参数为 true，则启用键弹窗。

29.3 键盘部件 API 函数

LVGL 提供给用户的 keyboard 部件相关函数，如表 29.3.1 所示：

函数	描述
创建键盘的函数	
lv_keyboard_create()	创建一个键盘对象
设置键盘部件的函数	

lv_keyboard_set_textarea()	分配一个文本区域，按下的字符将被放在这里
lv_keyboard_set_mode()	设置键盘模式
lv_keyboard_set_popovers()	按下的字符弹出窗口显示
lv_keyboard_set_map()	设置一个新 map
获取键盘部件属性的函数	
lv_keyboard_get_textarea()	获取键盘映射的文本区域对象
lv_keyboard_get_mode()	获取键盘模式
lv_btnmatrix_get_popovers()	弹窗”模式是否启用
lv_keyboard_get_map_array()	获得键盘的当前 map
lv_keyboard_get_selected_btn()	获取键盘选择的按键的索引
lv_keyboard_get_btn_text()	获取按钮的文本
lv_keyboard_def_event_cb()	将字符添加到文本区域并更改映射的默认键盘事件

表 29.3.1 keyboard 部件 API 函数

接下来我们来讲解 LVGL 的 keyboard 部件常用的函数有哪些：

1. lv_imgbtn_create 函数

该函数的作用是创建一个 keyboard 部件，该函数原型如下所示：

```
lv_obj_t * lv_keyboard_create(lv_obj_t * parent)
```

该函数的形参，如表 29.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 29.3.2 函数 lv_keyboard_create()形参描述

返回值：返回 keyboard 部件对象。

2. lv_keyboard_set_textarea 函数

该函数的作用是为键盘分配一个文本区域，该函数原型如下所示：

```
void lv_keyboard_set_textarea(lv_obj_t *kb, lv_obj_t *ta)
```

该函数的形参，如表 29.3.3 所示：

参数	描述
kb	指向键盘对象的指针
ta	指向要写入的文本区域对象的指针

表 29.3.3 函数 lv_keyboard_set_textarea()形参描述

返回值：无。

3. lv_keyboard_set_mode 函数

该函数的作用是设置键盘模式，该函数原型如下所示：

```
void lv_keyboard_set_mode(lv_obj_t *kb, lv_keyboard_mode_t mode)
```

该函数的形参，如表 29.3.4 所示：

参数	描述	
kb	指向键盘对象的指针	
mode	键盘的模式选择	
	LV_KEYBOARD_MODE_TEXT_LOWER	显示小写字母键盘
	LV_KEYBOARD_MODE_TEXT_UPPER	显示大写字母键盘
	LV_KEYBOARD_MODE_TEXT_SPECIAL	显示特殊字符键盘
	LV_KEYBOARD_MODE_NUMBER	显示数字键盘
	LV_KEYBOARD_MODE_USER_1 LV_KEYBOARD_MODE_USER_4	自定义模式

表 29.3.4 函数 lv_keyboard_set_mode()形参描述

返回值：无。

4. lv_keyboard_set_popovers 函数

该函数的作用是按下键盘的文本弹出窗口，该函数原型如下所示：

```
void lv_keyboard_set_popovers(lv_obj_t *kb, bool en)
```

该函数的形参，如表 29.3.5 所示：

参数	描述
kb	指向键盘对象的指针
en	true:启动弹窗；false：禁用弹窗

表 29.3.5 函数 lv_keyboard_set_popovers()形参描述

返回值：无。

5. lv_keyboard_set_map 函数

该函数的作用是创建新的 map，该函数原型如下所示：

```
void lv_keyboard_set_map(lv_obj_t *kb,
                         lv_keyboard_mode_t mode,
                         const char *map[],
                         const lv_btnmatrix_ctrl_t ctrl_map[])
```

该函数的形参，如表 29.3.6 所示：

参数	描述	
kb	指向键盘对象的指针	
mode	键盘的模式选择	
	LV_KEYBOARD_MODE_TEXT_LOWER	显示小写字母键盘
	LV_KEYBOARD_MODE_TEXT_UPPER	显示大写字母键盘
	LV_KEYBOARD_MODE_TEXT_SPECIAL	显示特殊字符键盘
	LV_KEYBOARD_MODE_NUMBER	显示数字键盘
	LV_KEYBOARD_MODE_USER_1 LV_KEYBOARD_MODE_USER_4	自定义模式
map	指向描述映射的字符串数组的指针	
ctrl_map	要设置新的 map	

表 29.3.6 函数 lv_keyboard_set_map()形参描述

返回值：无。

29.4 键盘控件实验

29.4.1 硬件设计

1. 例程功能

本实验主要测试 keyboaed 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建一个 keyboead 以及 textarea 部件，当按下 textarea 部件时，该部件往上偏移并显示键盘；当按下键盘的 btn 时，textarea 部件显示该 btn 的文本。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当。

该实验的实验工程，请参考《LVGL 例程 29 lv_keyboard(键盘)》。

29.4.2 软件设计

29.4.2.1 程序流程图

本实验的程序流程图，如下图 29.4.2.1.1 所示：

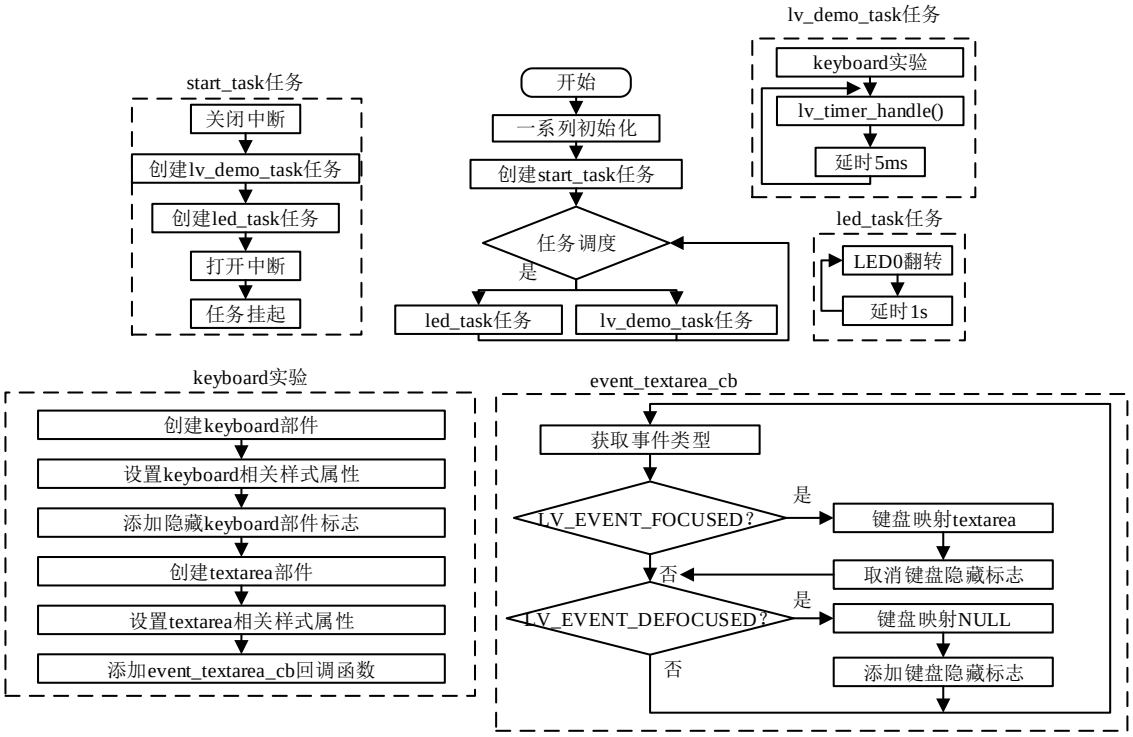


图 29.4.2.1.1 lv_keyboard 实验流程图

29.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_keyboard();
}
```

此函数实际调用 lv_example_ keyboard 函数编写 LVGL 代码，该函数源码如下所示：

```
    /**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_keyboard(void)
```

```
{
    /* 定义并创建键盘 */
    lv_obj_t* keyboard = lv_keyboard_create(lv_scr_act());
    lv_obj_add_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    /* 定义并创建文本框 */
    lv_obj_t* textarea = lv_textarea_create(lv_scr_act());
    /* 设置文本框宽度 */
    lv_obj_set_width(textarea, scr_act_width() - 10);
    /* 设置文本框高度 */
    lv_obj_set_height(textarea, (scr_act_height() >> 1) - 10);
    /* 设置文本框位置 */
    lv_obj_center(textarea);
    /* 设置文本框事件回调 */
    lv_obj_add_event_cb(textarea, event_textarea_cb, LV_EVENT_ALL, keyboard);

    static lv_style_prop_t tr_prop[] = {LV_STYLE_TRANSLATE_Y,
                                        LV_STYLE_PROP_INV};
    static lv_style_transition_dsc_t tr;
    lv_style_transition_dsc_init(&tr, tr_prop, lv_anim_path_ease_out,
                                 1000, 0, NULL);
    /* 设置文本框聚焦风格 */
    lv_obj_set_style_transition(textarea, &tr, LV_STATE_DEFAULT);
    /* 设置文本框按下时的聚焦风格 */
    lv_obj_set_style_translate_y(textarea, -((scr_act_height() >> 1) -
                                (scr_act_height() >> 2)), LV_STATE_FOCUSED);
}
```

此函数首先创建一个 keyboard 键盘部件并添加隐藏标志，其次创建 textarea 部件，该部件主要显示键盘按下的 btn 文本， 然后设置文本框的聚焦样式，当文本框按下时，文本框以 1000ms 时间往 Y 轴移动-((scr_act_height() >> 1) – (scr_act_height() >> 2))步长，最后添加文本框回调函数 event_textarea_cb，该回调函数源码如下所示：

```
/**
 * @brief  文本框事件回调
 * @param  无
 * @return 无
 */
static void event_textarea_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* textarea = lv_event_get_target(e);
    lv_obj_t* keyboard = (lv_obj_t*)e->user_data;

    if (code == LV_EVENT_FOCUSED) {
        lv_keyboard_set_textarea(keyboard, textarea);
```

```
        lv_obj_clear_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    }
    else if (code == LV_EVENT_DEFOCUSED) {
        lv_keyboard_set_textarea(keyboard, NULL);
        lv_obj_add_flag(keyboard, LV_OBJ_FLAG_HIDDEN);
    }
}
```

此回调函数主要做两个判断，当文本框聚焦时，设置键盘所按下的文本映射到文本框中并清除该键盘的隐藏标志；当文本框没有聚焦时，设置键盘所按下的文本映射为 NULL 中并添加该键盘的隐藏标志。

29.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 29.4.3.1 所示：

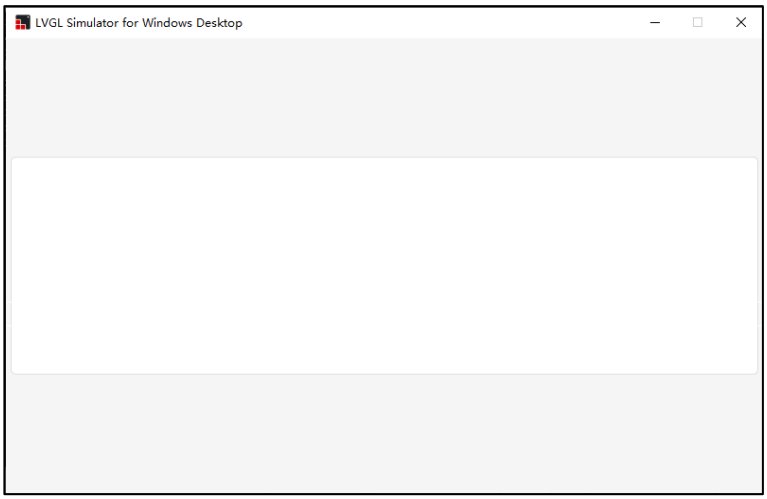


图 29.4.3.1 keyboard 部件实验开启界面

当按下上图中的文本框时，该文本框以动画的形式往 Y 轴移动并显示键盘如下图 29.4.3.2 所示：

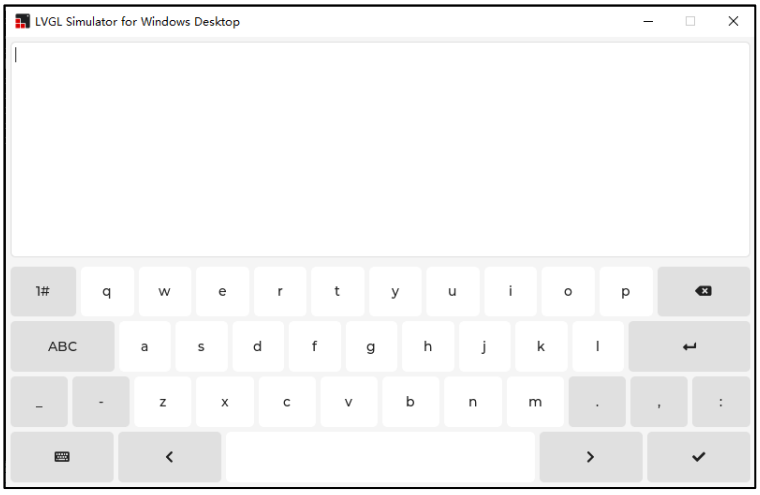


图 29.4.3.2 按下文本框后的界面

我们可以在上图的键盘输入字符到文本框中，如图 29.4.3.3 所示：

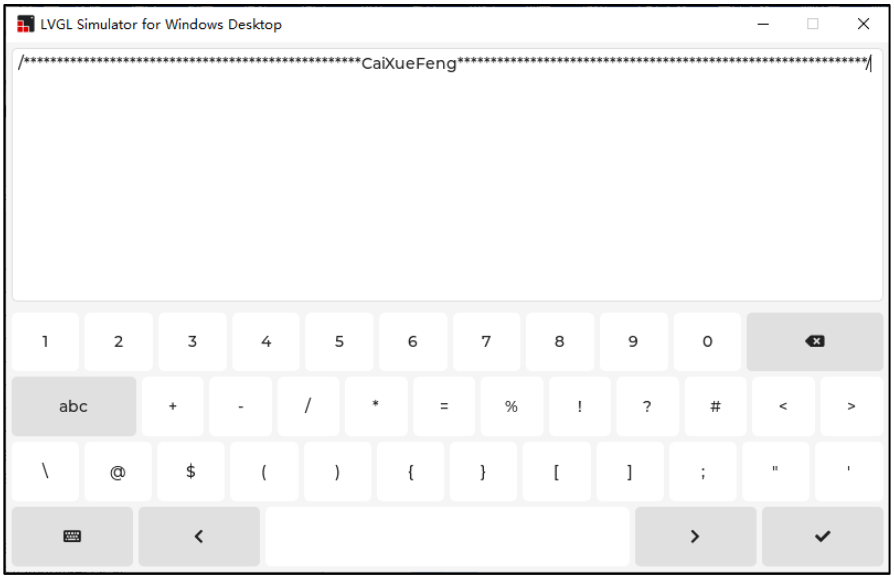


图 29.4.3.3 文本框显示输入的字符

第三十章 LED 部件(lv_led)

LED 部件是矩形(或圆形)的物体，它的亮度调节是根据背景颜色深度有关。LED 部件亮度越低，它的背景颜色越暗。在 UI 设计中，LED 部件一般描绘硬件 LED 闪烁状态。

本章分为如下几部分：

30.1 LED 部件的组成

30.2 LED 部件的相关知识

30.3 LED 部件 API 函数

30.4 LED 控件实验

30.1 LED 部件的组成

LED 部件只有一个主要部分组成，该部分称为 LV_LED_PART_MAIN，它使用了所有典型的背景样式属性。

30.2 LED 部件的相关知识

30.2.1 设置 LED 部件颜色

默认情况下，LED 部件的主体颜色为蓝色，如果用户设置该主体颜色，可调用函数 lv_led_set_color 设置，该函数具有两个形参，第一个形参表示要设置颜色的对象，而第二个形参表示要设置的对象主体颜色。如下图 30.2.1.1 所示：

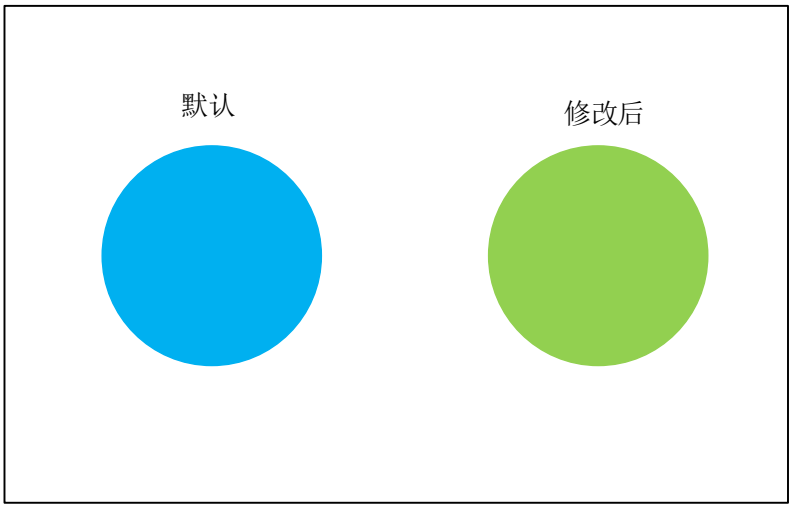


图 30.2.1.1 设置 LED 部件主体颜色

从上图可知：LED 部件默认主体颜色为蓝色，而调用函数 lv_led_set_color 可设置 LED 主体颜色。

30.2.2 设置 LED 部件亮度

前面笔者已经讲解过，LED 部件的亮度与颜色深度有关，比如白色的深度值逐渐自增，最后使白色逐渐变成灰色。LED 部件亮度与它相似，它们唯一不同的是深度值的变化，传统

的深度值是 0%~100%范围内的值，而 LED 部件的深度值是 0~255 的整数。如果 LED 部件的亮度设置为 0，则 LED 部件的主体为最暗；如果 LED 部件的亮度设置为 255，则 LED 部件的主体为最亮。

LED 部件默认设置为最亮的，如果用户想修改 LED 部件默认的亮度值，可调用函数 lv_led_set_brightness 设置。接下来我们就以一个示意图来讲解亮度的设置，如图 30.2.2.1 所示：

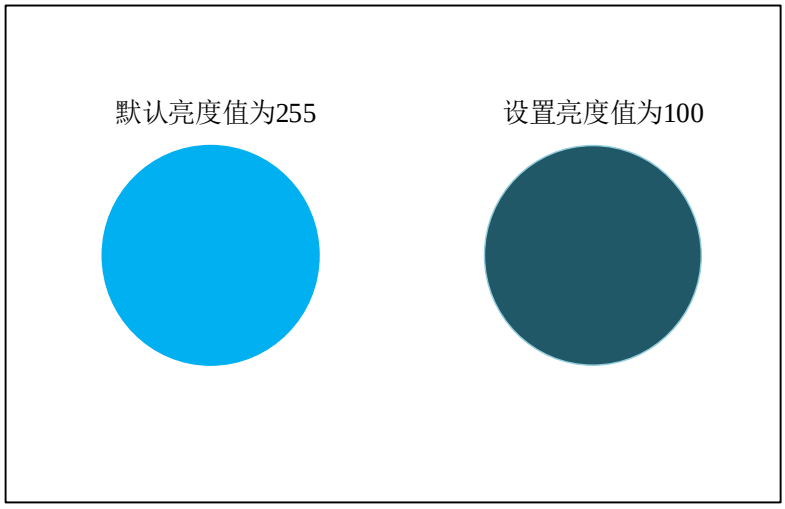


图 30.2.2.1 修改 LED 部件的亮度值

从上图可知：LED 部件设置的亮度值越低，则该部件的主体就越暗。

30.2.3 状态切换

在学习裸机时，如果我们点亮一个 LED 灯，可设置某个管脚输出高低电平来打开或者关闭它，这样的操作称之为状态切换。LED 部件也是具有它相似的功能，可调用 lv_led_on 和 lv_led_off 函数进行状态切换。注意：前面我们讲到，LED 部件默认亮度的值为 255，如果调用函数 lv_led_off 操作，则把 LED 部件的默认亮度的值设置为 0。

30.3 LED 部件 API 函数

LVGL 提供给用户的 LED 部件相关函数，如表 30.3.1 所示：

函数	描述
lv_led_create()	创建 LED 部件
lv_led_set_color()	设置 LED 部件主体颜色
lv_led_set_brightness()	设置 LED 部件的亮度
lv_led_on()	设置 LED 部件的亮度值为最大
lv_led_off()	设置 LED 部件的亮度值为最小
lv_led_toggle()	切换 LED 的状态
lv_led_get_brightness()	获取 LED 对象的亮度

表 30.3.1 LED 部件 API 函数

接下来我们来讲解 LVGL 的 LED 部件常用的函数有哪些：

1. lv_led_create 函数

该函数的作用是创建一个 LED 部件，该函数原型如下所示：

```
lv_obj_t * lv_led_create(lv_obj_t * parent)
```

该函数的形参，如表 30.3.2 所示：

参数	描述

parent	指向父类对象的指针

表 30.3.2 函数 lv_led_create()形参描述

返回值：返回 led 部件对象。

2. lv_led_set_color 函数

该函数的作用是设置 LED 部件的主体颜色，该函数原型如下所示：

```
void lv_led_set_color(lv_obj_t *led, lv_color_t color)
```

该函数的形参，如表 30.3.3 所示：

参数	描述
led	指向 LED 对象的指针
color	设置 LED 对象的主体颜色

表 30.3.3 函数 lv_led_set_color()形参描述

返回值：无。

3. lv_led_set_brightness 函数

该函数的作用是设置 LED 部件的亮度值，该函数原型如下所示：

```
void lv_led_set_brightness(lv_obj_t *led, uint8_t bright)
```

该函数的形参，如表 30.3.4 所示：

参数	描述
led	指向 LED 对象的指针
bright	设置 LED 对象的亮度值

表 30.3.4 函数 lv_led_set_brightness()形参描述

返回值：无。

4. lv_led_on 函数

该函数的作用是设置 LED 部件的亮度值为最大，该函数原型如下所示：

```
void lv_led_on(lv_obj_t *led)
```

该函数的形参，如表 30.3.5 所示：

参数	描述
led	指向 LED 对象的指针

表 30.3.5 函数 lv_led_on()形参描述

返回值：无。

5. lv_led_off 函数

该函数的作用是设置 LED 部件的亮度值为最小，该函数原型如下所示：

```
void lv_led_off(lv_obj_t *led)
```

该函数的形参，如表 30.3.6 所示：

参数	描述
led	指向 LED 对象的指针

表 30.3.6 函数 lv_led_off()形参描述

返回值：无。

6. lv_led_toggle 函数

该函数的作用是切换 LED 部件状态，该函数原型如下所示：

```
void lv_led_toggle(lv_obj_t *led)
```

该函数的形参，如表 30.3.7 所示：

参数	描述
led	指向 LED 对象的指针

表 30.3.7 函数 lv_led_toggle()形参描述

返回值：无。

7. lv_led_get_brightness 函数

该函数的作用是获取 LED 部件的亮度值，该函数原型如下所示：

```
uint8_t lv_led_get_brightness(const lv_obj_t *obj)
```

该函数的形参，如表 30.3.8 所示：

参数	描述
led	指向 LED 对象的指针

表 30.3.8 函数 lv_led_get_brightness()形参描述

返回值：无。

30.4 LED 控件实验

30.4.1 硬件设计

1. 例程功能

本实验主要测试 list 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数由三个实验组成，LED1 实验和 LED2 实验主要功能是按下 LED 部件切换状态，而 LED3 实验的 LED 部件是根据开发板上的 LED1 状态进行切换。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 30 lv_led(灯)》。

30.4.2 软件设计

30.4.2.1 程序流程图

本实验的程序流程图，如下图 30.4.2.1 所示：

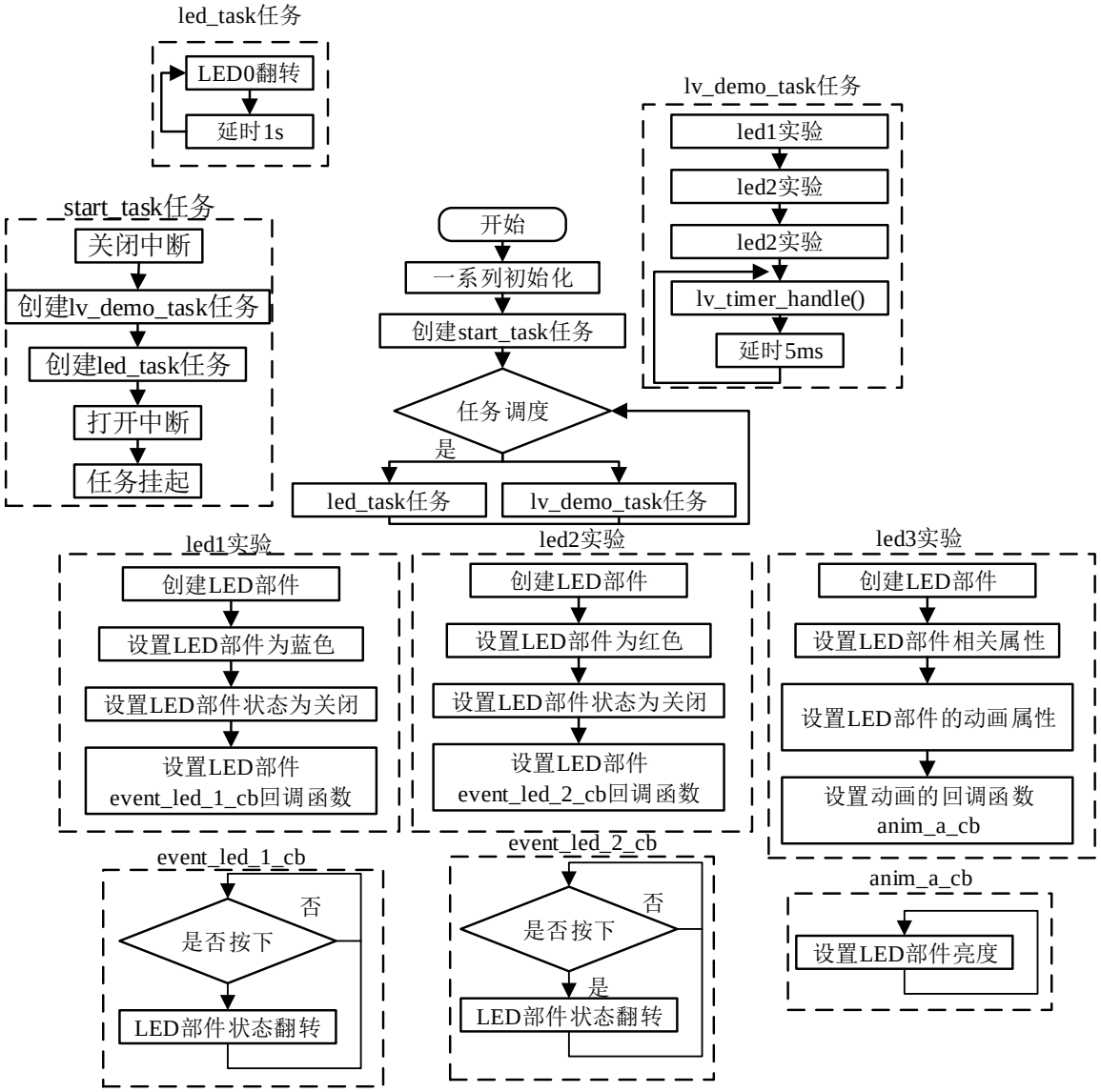


图 30.4.2.1 LED 部件实验流程图

30.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_led_1();
    lv_example_led_2();
    lv_example_led_3();
```

```
}
```

此函数具有三个小实验，下面我们分别地介绍这些实验实现的功能。

(1) lv_example_led_1 函数

```
/**
 * @brief  例 1LED 事件回调
 * @param  e:事件描述
 * @return 无
 */
static void event_led_1_cb(lv_event_t* e)
{
    lv_obj_t* led = lv_event_get_target(e);
    static bool is_led_on = false;

    is_led_on ? lv_led_off(led) : lv_led_on(led);
    is_led_on ? (is_led_on = false) : (is_led_on = true);
}

/**
 * @brief  例 1
 * @param  无
 * @return 无
 */
static void lv_example_led_1(void)
{
    lv_obj_t* led = lv_led_create(lv_scr_act());/* 定义并创建 LED */
    /* 设置 LED 宽度 */
    lv_obj_set_width(led, LV_MIN(scr_act_width(), scr_act_height()) / 5);
    /* 设置 LED 高度*/
    lv_obj_set_height(led, LV_MIN(scr_act_width(), scr_act_height()) / 5);
    /* 设置 LED 位置 */
    lv_obj_align(led, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);
    lv_led_off(led);/* 关闭 LED */
    /* 设置 LED 事件回调 */
    lv_obj_add_event_cb(led, event_led_1_cb, LV_EVENT_CLICKED, NULL);
}
```

从上述源码可知：首先创建 LED 部件，然后设置该部件的相关属性，其次设置该部件的状态为关闭态，最后设置该部件的回调函数，该回调函数主要切换 LED 部件的状态。**注意：**is_led_on 变量是 static 静态局部变量，所以该变量只初始化一次。

(2) lv_example_led_2 函数

```
/**
 * @brief  例 2LED 事件回调
 * @param  e:事件描述
 * @return 无
```

```
 */
static void event_led_2_cb(lv_event_t* e)
{
    lv_obj_t* led = lv_event_get_target(e);
    static bool is_led_on = false;

    is_led_on ? lv_led_off(led) : lv_led_on(led);
    is_led_on ? (is_led_on = false) : (is_led_on = true);
}

/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_led_2(void)
{
    lv_obj_t* led = lv_led_create(lv_scr_act());/* 定义并创建 LED */
    /* 设置 LED 宽度 */
    lv_obj_set_width(led, LV_MIN(scr_act_width(), scr_act_height()) / 5);
    /* 设置 LED 高度*/
    lv_obj_set_height(led, LV_MIN(scr_act_width(), scr_act_height()) / 5);
    lv_obj_align(led, LV_ALIGN_CENTER, 0, 0); /* 设置 LED 位置 */
    lv_led_off(led); /* 关闭 LED */
    lv_led_set_color(led, lv_palette_main(LV_PALETTE_RED)); /* 设置 LED 颜色 */
    /* 设置 LED 事件回调 */
    lv_obj_add_event_cb(led, event_led_2_cb, LV_EVENT_CLICKED, NULL);
}
```

上述源码和实验 1 一样实现原理，它们唯一不同的是部件的背景颜色。

(3) lv_example_led_3 函数

```
/**
 * @brief  例 3 动画 a 回调
 * @param  无
 * @return 无
 */
static void anim_a_cb(lv_obj_t* obj, int32_t v)
{
    lv_led_set_brightness(obj, v);
}

/**
 * @brief  例 3
 * @param  无
```

```
 * @return 无
 */
static void lv_example_led_3(void)
{
    lv_obj_t* led = lv_led_create(lv_scr_act());/* 定义并创建 LED */
    /* 设置 LED 宽度 */
    lv_obj_set_width(led, LV_MIN(scr_act_width(), scr_act_height()) / 5);
    /* 设置 LED 高度*/
    lv_obj_set_height(led, LV_MIN(scr_act_width(), scr_act_height()) / 5);
    /* 设置 LED 位置 */
    lv_obj_align(led, LV_ALIGN_CENTER, scr_act_width() / 3, 0);
    /* 设置 LED 颜色 */
    lv_led_set_color(led, lv_palette_main(LV_PALETTE_GREEN));
    /* 定义呼吸灯动画 */
    lv_anim_t a;
    lv_anim_init(&a);
    lv_anim_set_var(&a, led);
    lv_anim_set_delay(&a, 0);
    lv_anim_set_playback_delay(&a, 0);
    lv_anim_set_time(&a, 2000);
    lv_anim_set_playback_time(&a, 2000);
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);
    lv_anim_set_repeat_delay(&a, 0);
    lv_anim_set_values(&a, 80, 255);
    lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t)anim_a_cb);
    lv_anim_start(&a);
}
```

上述源码可知：我们使用动画属性函数对 LED 部件的亮度调节，该动画回调函数调用 LED 部件的 lv_led_set_brightness 函数，该函数是根据动画的时间值来调节 LED 部件的亮度。

30.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 30.4.3.1 所示：



图 30.4.3.1 LED 部件实验

当我们点击上图中的蓝色和红色 LED 部件时，可切换该部件的状态，如图 30.4.3.2 所示：

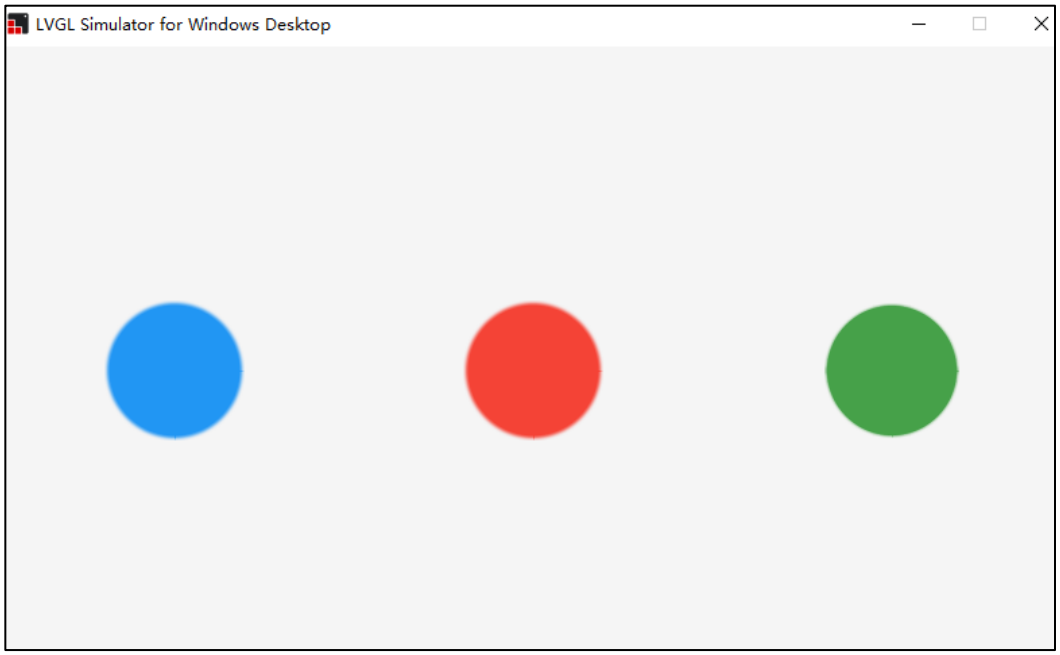


图 30.4.3.2 切换 LED 部件状态

第三十一章 列表部件(lv_list)

列表部件基本上是一个矩形的垂直布局，按钮和文本可以添加到其中。在 UI 设计中，list 部件应用广泛，例如文件管理系统、设置系统等 UI 界面都是使用 list 部件制作的。

本章分为如下几部分：

31.1 列表部件的组成

31.2 列表部件的相关知识

31.3 列表部件 API 函数

31.4 列表控件实验

31.1 列表部件的组成

list 部件是由 LV_PART_MAIN 和 LV_PART_SCROLLBAR 两部分构建组成。LV_PART_MAIN 部分主要设置 list 部件背景属性，而 LV_PART_SCROLLBAR 部分主要设置滚动条属性。下面我们来看一下 list 部件的示意图，如图 31.1.1 所示：

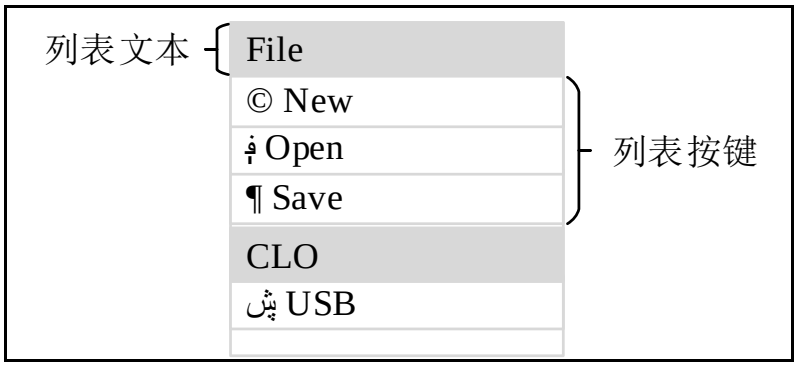


图 31.1.1 list 部件示意图

从上图可知：如果我们使用 list 部件制作 UI 界面，则 list 部件除了设置样式，还具备添加列表文本和列表按键。

31.2 列表部件的相关知识

31.2.1 添加列表按键

前面我们已经讲解过，用户使用部件就必须先创建该部件，然后再设置样式或者添加该部件的相关功能，部件如何创建，这里笔者就不需要讲解了，大家可以看下面的 list 部件的 API 函数。

上一节的图 31.1.1 中列表按键如何添加呢？，在默认情况下，list 部件被创建时，该部件只有一个背景矩形框，在背景矩形框中需要用户自行添加列表按键，该列表按键的添加方法就是调用函数 lv_list_add_btn 添加列表按键，该函数具有三个形参，这些形参最值得注意的是第二和第三个形参，第二个形参表示列表按键是否添加图标，而第三个形参表示该列表按键的文本。

下面我们就以一个简单是实例来讲解 list 部件如何添加列表按键，如下源码所示：

```
void lv_mainstart(void)
{
```

```
    lv_obj_t* list = lv_list_create(lv_scr_act());  /* 创建列表 */
    lv_obj_set_width(list, 200);                    /* 设置列表宽度 */
    lv_obj_set_height(list, 150);                   /* 设置列表高度 */
    lv_obj_center(list);

    lv_obj_t* btn;
    btn = lv_list_add_btn(list, LV_SYMBOL_FILE, "New");          /* 添加按钮 */
    btn = lv_list_add_btn(list, LV_SYMBOL_DIRECTORY, "Open");    /* 添加按钮 */
}
```

从上述源码可知：首先创建 list 部件，然后调用函数 lv_list_add_btn 添加列表按键，该列表按键的第二个形参指向图标，而第三个形参表示列表按键的文本。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 31.2.1.1 所示：

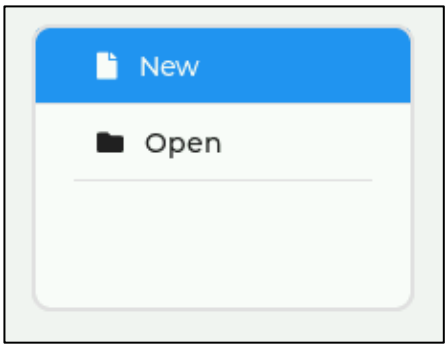


图 31.2.1.1 添加列表按键

从上图可知：我们添加了两个列表按键，这两个列表按键都是具有图标和文本的，如果用户不想显示图标或者文本，则在对应的形参填入 NULL。

31.2.2 设置列表文本

前面笔者已经讲解了列表按键的添加方式，那么本小节主要讲解列表文本的添加。顾名思义，列表文本在 list 部件中主要提示和引导列表按键，列表文本添加方式是调用函数 lv_list_add_text 添加列表文本，该函数具有两个形参，第一个形参指向列表对象，而第二个形参表示设置的文本。

下面我们就以一个简单是实例来讲解 list 部件如何添加列表文本，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t* list = lv_list_create(lv_scr_act());             /* 创建列表 */
    lv_obj_set_width(list, 200);                               /* 设置列表宽度 */
    lv_obj_set_height(list, 150);                              /* 设置列表高度 */
    lv_obj_center(list);

    lv_obj_t* btn;
    lv_list_add_text(list, "File");                            /* 列表添加标签 */
    btn = lv_list_add_btn(list, LV_SYMBOL_FILE, "New");        /* 添加按钮 */
    btn = lv_list_add_btn(list, LV_SYMBOL_DIRECTORY, "Open"); /* 添加按钮 */
}
```

从上述源码可知：我们调用函数 lv_list_add_text 添加列表文本，该文本主要提示和引导列表按键。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 31.2.1.2 所示：

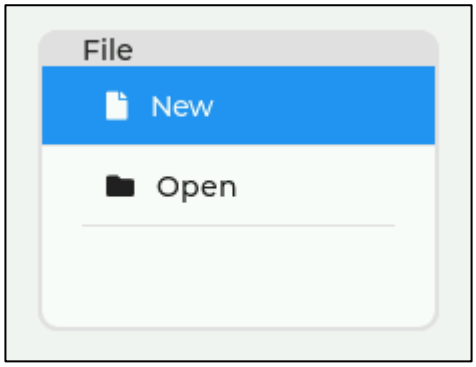


图 31.2.1.2 添加列表文本

31.3 列表部件 API 函数

LVGL 提供给用户的 list 部件相关函数，如表 31.3.1 所示：

函数	描述
lv_list_create()	创建 list 部件
lv_list_add_text()	添加列表文本
lv_list_add_btn()	添加列表按键
lv_list_get_btn_text()	获取按下的按键文本

表 31.3.1 list 部件 API 函数

接下来我们来讲解 LVGL 的 list 部件常用的函数有哪些：

1. lv_list_create 函数

该函数的作用是创建一个 list 部件，该函数原型如下所示：

```
lv_obj_t * lv_list_create(lv_obj_t * parent)
```

该函数的形参，如表 31.3.2 所示：

参数	描述
parent	指向父类对象的指针

表 31.3.2 函数 lv_list_create()形参描述

返回值：返回 list 部件对象。

2. lv_list_add_text 函数

该函数的作用是设置列表文本，该函数原型如下所示：

```
lv_obj_t *lv_list_add_text(lv_obj_t *list, const char *txt)
```

该函数的形参，如表 31.3.3 所示：

参数	描述
list	指向 list 对象的指针
txt	设置的文本

表 31.3.3 函数 lv_list_add_text()形参描述

返回值：返回按键对象。

3. lv_list_add_btn 函数

该函数的作用是设置列表按键，该函数原型如下所示：

```
lv_obj_t *lv_list_add_btn(lv_obj_t *list, const char *icon, const char *txt)
```

该函数的形参，如表 31.3.4 所示：

参数	描述

list	指向 list 对象的指针
icon	图标
txt	设置的文本

表 31.3.4 函数 lv_list_add_btn()形参描述

返回值：返回按键对象。

4. lv_list_get_btn_text 函数

该函数的作用是获取列表按键，该函数原型如下所示：

```
const char *lv_list_get_btn_text(lv_obj_t *list, lv_obj_t *btn)
```

该函数的形参，如表 31.3.5 所示：

参数	描述
list	指向 list 对象的指针
btn	图标

表 31.3.5 函数 lv_list_get_btn_text()形参描述

返回值：返回文本。

31.4 列表控件实验

31.4.1 硬件设计

1. 例程功能

本实验主要测试 list 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数主要创建一个 list 部件和 obj 部件，当按下 list 部件的某一个选项时，该选项的文本在 obj 对象中显示。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 31 lv_list(列表)》。

31.4.2 软件设计

31.4.2.1 程序流程图

本实验的程序流程图，如下图 31.4.2.1 所示：

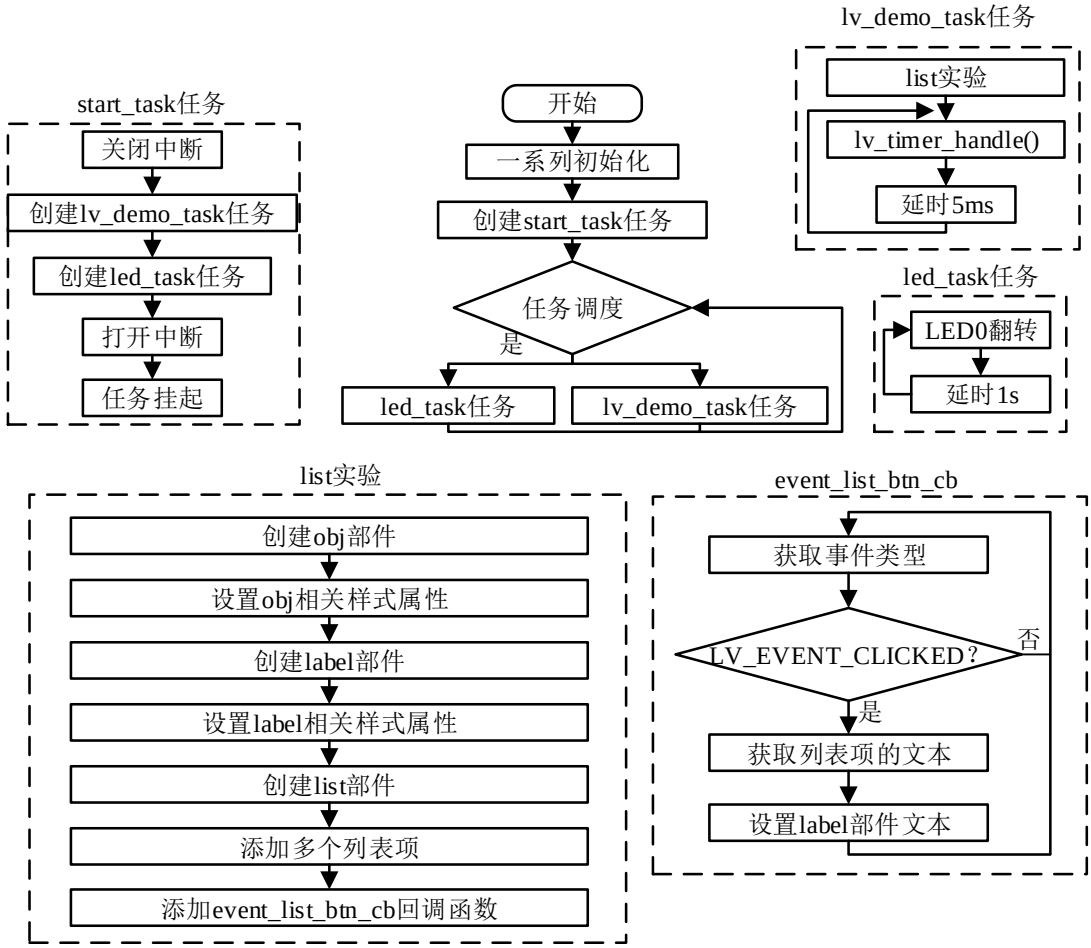


图 31.4.2.1 list 部件实验流程图

31.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_list();
}
```

此函数实际调用 lv_example_list 函数编写 LVGL 代码，该函数源码如下所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
```

```
static void lv_example_list(void)
{
    const lv_font_t* font;
    /* 根据屏幕大小设置字体 */
    if (scr_act_width() <= 320)
        font = &lv_font_montserrat_14;
    else if (scr_act_width() <= 480)
        font = &lv_font_montserrat_16;
    else
        font = &lv_font_montserrat_18;

    lv_obj_t* obj_left = lv_obj_create(lv_scr_act()); /* 定义并创建一个物体 */
    lv_obj_set_width(obj_left, scr_act_width() * 0.7); /* 设置物体的宽度 */
    lv_obj_set_height(obj_left, scr_act_height() * 0.9); /* 设置物体的高度 */
    lv_obj_align(obj_left, LV_ALIGN_LEFT_MID, 5, 0); /* 设置物体的位置 */
    lv_obj_update_layout(obj_left); /* 手动更新物体的参数 */
    lv_obj_t* obj_right = lv_obj_create(lv_scr_act()); /* 定义并创建一个物体 */
    /* 设置物体的宽度 */
    lv_obj_set_width(obj_right, scr_act_width() -
                     lv_obj_get_width(obj_left) - 15);
    /* 设置物体的高度 */
    lv_obj_set_height(obj_right, lv_obj_get_height(obj_left));
    /* 设置物体的位置 */
    lv_obj_align_to(obj_right, obj_left, LV_ALIGN_OUT_RIGHT_MID, 5, 0);
    /* 手动更新物体的参数 */
    lv_obj_update_layout(obj_right);
    /* 定义并创建标签 */
    lv_obj_t* label_list = lv_label_create(obj_right);
    /* 设置标签的宽度 */
    lv_obj_set_width(label_list, lv_obj_get_width(obj_right) - 13);
    /* 设置标签为位置 */
    lv_obj_align(label_list, LV_ALIGN_TOP_MID, 0, 5);
    /* 手动更新标签的参数 */
    lv_obj_update_layout(label_list);
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label_list,
                                LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 设置标签文本 */
    lv_label_set_text(label_list, "New");
    /* 设置标签文本字体 */
    lv_obj_set_style_text_font(label_list, font, LV_PART_MAIN);

    list = lv_list_create(obj_left); /* 创建列表 */
```

```
    lv_obj_set_width(list, lv_obj_get_width(obj_left) * 0.8); /* 设置列表宽度 */
    /* 设置列表高度 */
    lv_obj_set_height(list, lv_obj_get_height(obj_left) * 0.9);
    lv_obj_center(list);  /* 设置列表的位置 */
    lv_obj_set_style_text_font(list, font, LV_PART_MAIN); /* 为列表添加按钮 */

    lv_obj_t* btn;
    lv_list_add_text(list, "File");/* 列表添加标签 */
    btn = lv_list_add_btn(list, LV_SYMBOL_FILE, "New");/* 添加按钮 */
    /* 添加按钮回调 */
    lv_obj_add_event_cb(btn, event_list_btn_cb, LV_EVENT_CLICKED, label_list);
    /*********************省略一部分同样操作的代码*********************/

}
```

此函数代码篇幅过长，这里笔者就不一一的讲解了，下面我们来看一下 event_list_btn_cb 回调函数到底实现什么功能，如下源码所示：

```
    /**
 * @brief  列表按钮事件回调
 * @param  无
 * @return 无
 */
static void event_list_btn_cb(lv_event_t* e)
{
    lv_obj_t* list_label = (lv_obj_t*)e->user_data;
    lv_obj_t* list_btn = lv_event_get_target(e);

    lv_label_set_text(list_label, lv_list_get_btn_text(list, list_btn));
}
```

该回调函数非常简单，当按下列表项时，系统会进入该回调函数，该函数主要获取按下的列表项文本并在 lable 部件显示。

31.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 31.4.3.1 所示：

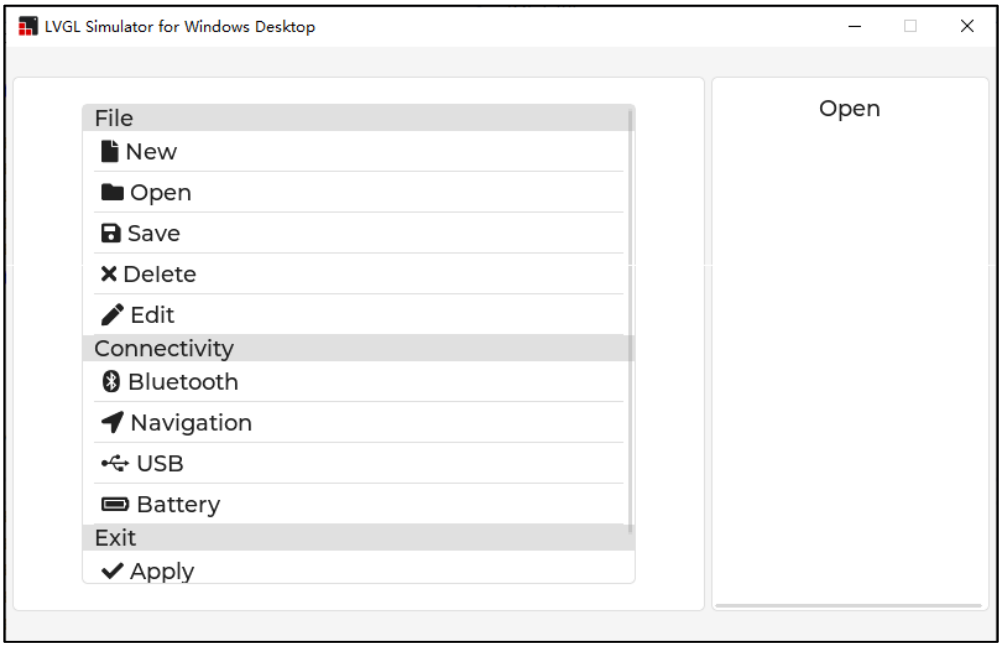


图 31.4.3.1 list 部件实验

第三十二章 仪表部件(lv_meter)

meter 部件可以以非常灵活的方式可视化数据。该部件可以显示弧线、指针、划线和标签等图形。例如汽车的油表就是使用该部件制作的。

本章分为如下几部分：

32.1 仪表部件的组成

32.2 仪表部件的相关知识

32.3 仪表部件 API 函数

32.4 仪表控件实验

32.1 仪表部件的组成

meter 部件是由 LV_PART_MAIN/TICK/INDICATOR/ITEMS 四个部分构建而成，它们的相关说明如以下所示：

① LV_PART_MAIN：仪表的背景。

② LV_PART_TICK：line 或者 text 样式属性，该属性主要绘画仪表的数值和刻度。

③ LV_PART_INDICATOR：line 或者 img 样式属性，该属性主要绘画仪表指针样式。

④ LV_PART_ITEMS：arc 属性。该属性主要绘画仪表的圆弧属性。

根据上述描述，可得到 meter 部件组成示意图。如图 32.1.1 所示：

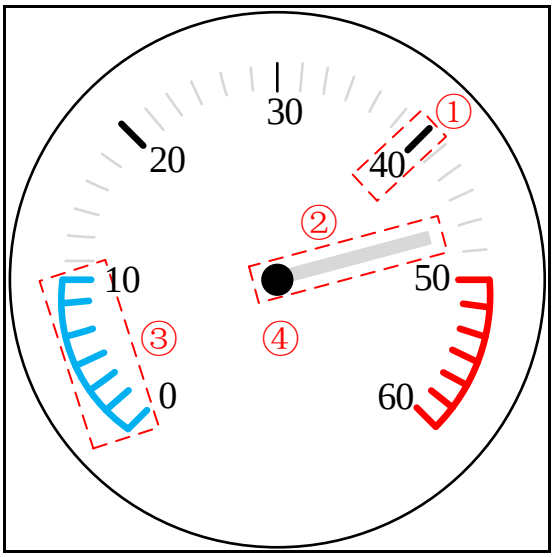


图 32.1.1 meter 组成部分

上图中①的样式属性是由 LV_PART_TICK 组成部分设置；图中②的样式属性是由 LV_PART_INDICATOR 组成部分设置；图中③的样式属性由 LV_PART_ITEMS 组成部分设置；图中④的样式属性由 LV_PART_MAIN 设置。

mater 部件相关的样式属性修改，请大家参考第六章 6.3 小节。

32.2 仪表部件的相关知识

32.2.1 仪表部件主要功能

在 LVGL 中，创建一个 mater 部件时，该部件在屏幕上显示圆形和圆点背景，显然地，该部件不具备仪表的角度、仪表的范围、仪表的刻度和仪表的指针等功能，这些功能都是由用户来设置的。接下来我们分别地讲解这些功能到底如何添加到仪表当中：

1. 添加刻度

从图 32.1.1 可知：刻度分为小刻度和主刻度，主刻度是图中加粗的线条，而小刻度是图中正常的线条。这些刻度的添加是由三个函数来完成，如表 32.2.1.1 所示：

函数	描述
lv_meter_add_scale()	添加刻度
lv_meter_set_scale_ticks()	设置小刻度
lv_meter_set_scale_major_ticks()	设置主刻度

表 32.2.1.1 仪表添加刻度函数描述

上表的函数调用是有一定的顺序的，首先调用第一个函数把刻度添加到仪表当中，该函数返回一个刻度的对象，然后我们根据这个刻度的对象分别设置小刻度和主刻度。

下面我们分别地介绍这些函数的原型，如下所示：

(1) lv_meter_add_scale 函数

该函数的作用是给仪表添加一个刻度，该函数原型如下所示：

```
lv_meter_scale_t *lv_meter_add_scale(lv_obj_t *obj)
```

该函数的形参，如表 32.3.1.2 所示：

参数	描述
obj	指向 mater 对象的指针

表 32.3.1.2 函数 lv_meter_add_scale()形参描述

返回值：返回刻度的对象。

(2) lv_meter_set_scale_ticks 函数

该函数的作用是给仪表添加一个小刻度，该函数原型如下所示：

```
void lv_meter_set_scale_ticks(lv_obj_t *obj, lv_meter_scale_t *scale,
                              uint16_t cnt, uint16_t width,
                              uint16_t len, lv_color_t color)
```

该函数的形参，如表 32.3.1.3 所示：

参数	描述
meter	指向 mater 对象的指针
scale	指向刻度对象
cnt	小刻度的数量
width	小刻度的宽度
len	小刻度的长度
color	小刻度的颜色

表 32.3.1.3 函数 lv_meter_set_scale_ticks()形参描述

返回值：无。

(3) lv_meter_set_scale_major_ticks 函数

该函数的作用是给仪表添加一个主刻度，该函数原型如下所示：

```
void lv_meter_set_scale_major_ticks(lv_obj_t *obj, lv_meter_scale_t *scale,
                                    uint16_t nth, uint16_t width, uint16_t len,
```

```
                                    lv_color_t color, int16_t label_gap)
```

该函数的形参，如表 32.3.1.3 所示：

参数	描述
meter	指向 mater 对象的指针
scale	指向刻度对象
nth	绘画主刻度的步长
width	主刻度的宽度
len	主刻度的长度
color	主刻度的颜色
label_gap	刻度与标签之间的间隙

表 32.3.1.3 函数 lv_meter_set_scale_major_ticks()形参描述

返回值：无。

下面我们以一个实例来讲解这些函数的使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())

void lv_mainstart(void)
{
    lv_obj_t* meter = lv_meter_create(lv_scr_act());   /* 定义并创建仪表 */
    lv_obj_set_width(meter, scr_act_height() * 0.4);   /* 设置仪表宽度 */
    lv_obj_set_height(meter, scr_act_height() * 0.4); /* 设置仪表高度 */
    /* 设置仪表位置 */
    lv_obj_center(meter);
    /* 设置仪表刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);   /* 定义并添加刻度 */
    /* 设置小刻度数量为 41，宽度为 1，长度为屏幕高度除以 80，颜色为灰色 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1, scr_act_height() / 80,
                             lv_palette_main(LV_PALETTE_GREY));
    /* 设置主刻度的步长为 8,宽度为 1，长度为屏幕高度除以 60，颜色为黑色，
    刻度与数值的间距为屏幕高度除以 30 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1, scr_act_height() / 60,
                                   lv_color_black(), scr_act_height() / 30);
}
```

从上述源码可知：首先创建 mater 部件，然后添加刻度到仪表当中，其次设置小刻度，该刻度的数量是 41 个，宽度为 1，而长度根据屏幕高度除以 80，最后设置主刻度，主刻度是根据小刻度的数量来计算的，它的计算公式是：主刻度数量=(小刻度数量-1)/(主刻度的步长) + 1，显然地，根据这条公式可以算出主刻度的数量为 6((41(小刻度数量)-1)/8(步长) +1)。

注意：第一个主刻度是从第一个小刻度开始绘画，然后根据小刻度以 8 为步长绘画第二个主刻度，由此类推。这些主刻度对应的数值是由仪表数值范围决定的，默认数值最大为 100，系统将该数值除以主刻度数量为增量，如第一个主刻度为 0，经过小刻度 8 步长绘画第二个主刻度，该刻度的对应数值为 16(100/6)，由此类推。

上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 32.2.1.1 所示：

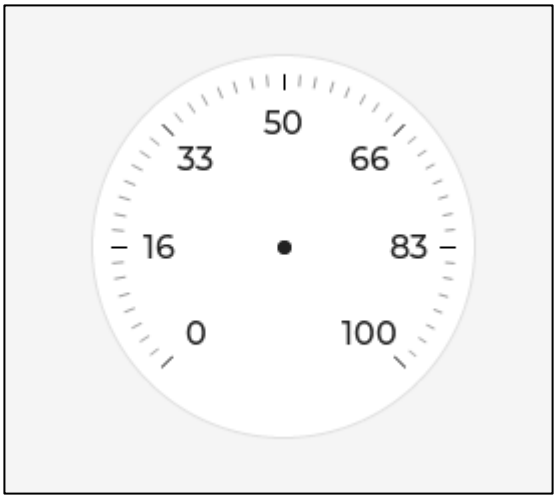


图 32.2.1.1 设置刻度到仪表当中

从上图可知：从第二个刻度的数值开始，该数值的偏差为正负 1。

2. 添加指针

仪表指针，是指用于仪表上指示数据的零部件。指针的功能就是以比较客观直接的方法指示复杂的数据结构。指针的结构一般是狭长的，一端尖，便于指示和读数。因为精确度是仪器仪表的主要性能之一，指针的这种设计和构造有利于提高仪器仪表的精度，减小误差。从上图 32.2.1.1 中，该仪表只添加了刻度功能，但它并没有指针的功能，如果该仪表不添加指针，那么我们很难读取仪表的数值，所以仪表指针对于仪表来说是非常重要的。

在 LVGL 中。关于仪表的指针具有两个函数，这两个函数描述如表 32.3.1.4 所示：

函数	描述
lv_meter_add_needle_line()	添加指针线指示刻度
lv_meter_set_indicator_value()	设置指针指向的数值

表 32.3.1.4 仪表指针相关函数

接下来我们分别地讲解上述的函数传入的形参有哪些，如下所示：

1. lv_meter_add_needle_line 函数

该函数的作用是给仪表添加一个指针，该函数原型如下所示：

```
lv_meter_indicator_t *lv_meter_add_needle_line(lv_obj_t *obj,
                                               lv_meter_scale_t *scale,
                                               uint16_t width,
                                               lv_color_t color,
                                               int16_t r_mod)
```

该函数的形参，如表 32.3.1.5 所示：

参数	描述
obj	指向 mater 对象的指针
scale	指向刻度对象
width	指针宽度
color	指针颜色
r_mod	修改半径长度(0：默认值；-10：默认长度-10)

表 32.3.1.5 函数 lv_meter_add_needle_line()形参描述

返回值：返回仪表指针的对象。

2. lv_meter_set_indicator_value 函数

该函数的作用是设置指针的数值，该函数原型如下所示：

```
void lv_meter_set_indicator_value( lv_obj_t * obj,
                                   lv_meter_indicator_t * indic,
                                   int32_t value)
```

该函数的形参，如表 32.3.1.5 所示：

参数	描述
obj	指向 mater 对象的指针
indic	指向指针对象
value	指向的数值

表 32.3.1.5 函数 lv_meter_set_indicator_value()形参描述

返回值：无。

上述函数是关于仪表指针的添加和指针指向的数值，下面我们以一个实例来讲解这两个函数的使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())

void lv_mainstart(void)
{
    lv_obj_t* meter = lv_meter_create(lv_scr_act());   /* 定义并创建仪表 */
    lv_obj_set_width(meter, scr_act_height() * 0.4);   /* 设置仪表宽度 */
    lv_obj_set_height(meter, scr_act_height() * 0.4); /* 设置仪表高度 */
    /* 设置仪表位置 */
    lv_obj_center(meter);
    /* 设置仪表刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);   /* 定义并添加刻度 */
    /* 设置小刻度 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1, scr_act_height() / 80,
                             lv_palette_main(LV_PALETTE_GREY));
    /* 设置主刻度 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1, scr_act_height() / 60,
                                   lv_color_black(), scr_act_height() / 30);

    lv_meter_indicator_t* indic;
    /* 添加仪表指针，该指针宽度为 4，颜色为灰色，长度-10 */
    indic = lv_meter_add_needle_line(meter, scale, 4,
                                     lv_palette_main(LV_PALETTE_GREY), -10);
    /* 设置指针指向的数值 */
    lv_meter_set_indicator_value(meter, indic,54);

}
```

上述有些源码前面笔者已经讲解过，这里不必重复讲解，我们主要关心仪表指针的相关函数。首先我们调用函数 lv_meter_add_needle_line 添加仪表指针，该指针的宽度为 4，颜色为灰色，而指针长度-10，然后该函数返回仪表指针对象，最后根据仪表指针对象设置仪表指针指

向的数值为 54。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 32.2.1.2 所示：

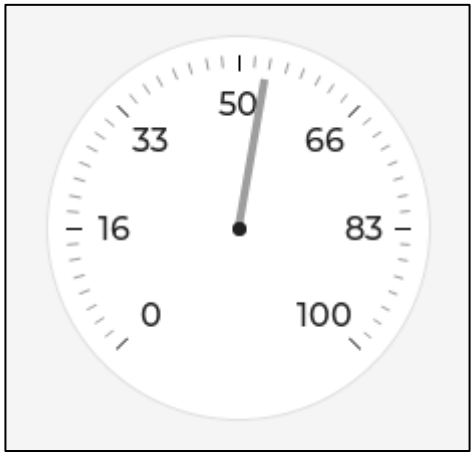


图 32.2.1.2 添加仪表指针并设置该仪表指针指向的数值

根据上图可知：仪表指针的顶端和刻度有一定的距离，该距离和函数 lv_meter_add_needle_line 的最后的形参传入的数值有关，如果该传入的数值为 0，则仪表指针覆盖到小刻度的顶端；如果该传入的数值大于 0，则仪表指针超出小刻度；如果该传入的数值小于 0，则该仪表指针默认长度减去该数值。

3. 设置仪表的角度和仪表的范围

仪表的角度主要设置仪表从顺时针旋转的角度以及有效的角度范围，比如图 32.2.1.2 中，该图中的仪表有效角度范围为 0~270°，如果我们使用仪表部件制作时钟 UI，显然 0~270°有效角度范围并不满足我们的需求，所以我们可手动设置仪表的有效角度范围为 0~360°。

仪表的范围是指仪表的数值范围，默认仪表的数值范围为 0~100 的数值，我们可以手动调整数值的范围。

上述仪表的三个功能是由一个 LVGL 函数 lv_meter_set_scale_range 统一设置，该函数具有六个形参，这些形参描述如下所示：

lv_meter_set_scale_range 函数

该函数的作用是设置仪表的角度和仪表的范围，该函数原型如下所示：

```
void lv_meter_set_scale_range(lv_obj_t * obj,
                              lv_meter_scale_t * scale,
                              int32_t min, int32_t max,
                              uint32_t angle_range, uint32_t rotation)
```

该函数的形参，如表 32.3.1.6 所示：

参数	描述
obj	指向 mater 对象的指针
scale	指向指针对象
仪表数值范围设置	
min	最小值
max	最大值
仪表的有效角度设置	
angle_range	最大的角度(默认是 270°，可设置 0~360°)
仪表的旋转角度设置	
rotation	旋转的角度(默认是 0°，可设置 0~360°)

表 32.3.1.6 函数 lv_meter_set_scale_range()形参描述

返回值：无。

上述函数是关于仪表的属性功能，下面我们以一个实例来讲解这个函数到底如何使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())

void lv_mainstart(void)
{
    lv_obj_t* meter = lv_meter_create(lv_scr_act());   /* 定义并创建仪表 */
    lv_obj_set_width(meter, scr_act_height() * 0.4);   /* 设置仪表宽度 */
    lv_obj_set_height(meter, scr_act_height() * 0.4); /* 设置仪表高度 */
    /* 设置仪表位置 */
    lv_obj_center(meter);
    /* 设置仪表刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);   /* 定义并添加刻度 */
    /* 设置小刻度 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1, scr_act_height() / 80,
                             lv_palette_main(LV_PALETTE_GREY));
    /* 设置主刻度 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1, scr_act_height() / 60,
                                   lv_color_black(), scr_act_height() / 30);

    lv_meter_indicator_t* indic;
    /* 添加仪表指针，该指针宽度为 4，颜色为灰色，长度-10 */
    indic = lv_meter_add_needle_line(meter, scale, 4,
                                     lv_palette_main(LV_PALETTE_GREY), -10);
    /* 设置指针指向的数值 */
    lv_meter_set_indicator_value(meter, indic,54);
    /* 设置仪表数值范围、有效角度和旋转角度 */
    lv_meter_set_scale_range(meter, scale, 0, 150, 360, 90);

}
```

上述有些源码前面笔者已经讲解过，这里不必重复讲解，我们主要关心仪表属性函数的设置。该函数设置仪表的数值范围为 0 到 150，有效角度为 360°，而旋转角度为 90°。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 32.2.1.3 所示：

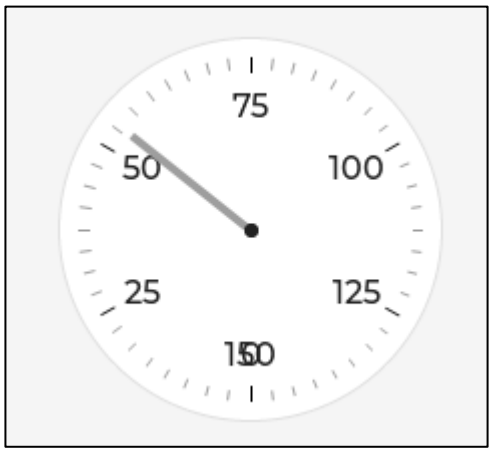


图 32.2.1.3 设置仪表属性

从上图可知：我们设置仪表有效角度为 0~360，数值范围为 0~150，而旋转角度为 90°(默认情况下，0 和 150° 的位置是在三点钟位置，经过旋转 90° 得到上图效果)。

细心的小伙伴可能会发现一个问题，上图的仪表的第二个刻度的数值为 25，为什么不是 16？，前面笔者也讲解过，这些数值与数值范围有关。

32.2.2 仪表部件次要功能

mater 部件的次要功能主要装饰仪表，如仪表指针使用图片替代、设置仪表的指示刻度以及设置仪表弧线指示器等次要功能。下面我们分别地讲解这三个次要功能到底如何使用，如下所示：

1. 仪表指针图片

从如 32.2.1.3 中，该仪表的指针是由线来绘画的，显然地，这样的绘画不能使仪表具有美观的效果，LVGL 作者为了提高仪表的美观性，他提供用户一个函数可把图片替代仪表的指针，这样的操作能使仪表更加生动。下面我们来看一下这个函数到底有哪些参数，如下所示：

lv_meter_add_needle_img 函数

该函数的作用是添加一个指针图像，该函数原型如下所示：

```
lv_meter_indicator_t *lv_meter_add_needle_img( lv_obj_t *obj,
                                               lv_meter_scale_t *scale,
                                               const void *src,
                                               lv_coord_t pivot_x,
                                               lv_coord_t pivot_y)
```

该函数的形参，如表 32.2.2.1 所示：

参数	描述
obj	指向 mater 对象的指针
scale	指向刻度对象
src	图像源(C 数组、路径等)
pivot_x	X 枢轴点
pivot_y	Y 枢轴点

表 32.2.2.1 函数 lv_meter_add_needle_img()形参描述

返回值：无。

下面我们以一个实例来讲解这个函数到底如何使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
```

```
#define scr_act_height() lv_obj_get_height(lv_scr_act())
LV_IMG_DECLARE(img_hand)

void lv_mainstart(void)
{
    lv_obj_t* meter = lv_meter_create(lv_scr_act());   /* 定义并创建仪表 */
    lv_obj_set_width(meter, scr_act_height() * 0.4);   /* 设置仪表宽度 */
    lv_obj_set_height(meter, scr_act_height() * 0.4);  /* 设置仪表高度 */
    /* 设置仪表位置 */
    lv_obj_center(meter);
    /* 设置仪表刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);   /* 定义并添加刻度 */
    /* 设置小刻度 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1, scr_act_height() / 80,
                             lv_palette_main(LV_PALETTE_GREY));
    /* 设置主刻度 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1, scr_act_height() / 60,
                                   lv_color_black(), scr_act_height() / 30);
    /* 添加指针图像 */
    lv_meter_indicator_t * indic = lv_meter_add_needle_img(meter,
                                                           scale,
                                                           &img_hand, 5, 5);
    /* 设置指针指向的数值 */
    lv_meter_set_indicator_value(meter, indic,54);
}
```

从上述源码可知：首先我们自定义一个图像源，然后程序调用函数 lv_meter_add_needle_img 设置该图像为仪表指针，最后程序调用函数 lv_meter_set_indicator_value 设置仪表指针指向数值。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 32.2.2.1 所示：

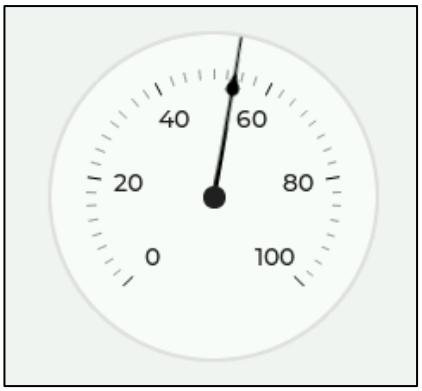


图 32.2.2.1 设置图像为仪表指针

注意：函数 lv_meter_add_needle_img 的最后两个形参根据实际情况设置，这两个形参默认为 0，如果该图像和原点稍微有点偏差，则大家根据实际情况来设置这两个形参的数值。

2. 仪表的指示刻度

仪表的指示刻度是指在仪表刻度上指定某个范围刻度填充颜色，前面我们学习过 lv_meter_set_scale_ticks 和 lv_meter_set_scale_major_ticks 这两个函数，它们都有一个形参表示刻度填充的颜色，显然地，这些形参只针对整体设置颜色，并不是指定某个刻度范围设置颜色。这个次要功能也是比较常见的，比如汽车油表，它的最低值和最大值的某范围都是使用明显的颜色提示和引导驾驶员。下面我们使用一个示意图来讲解这个次要功能，如图 32.2.2.2 所示：

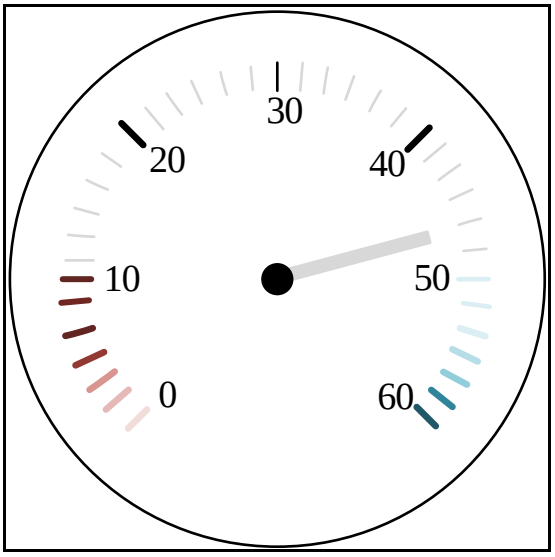


图 32.2.2.2 设置某个刻度范围颜色

从上图可知：我们设置仪表从 0 到 10 和 50 到 60 以某种颜色渐变，这样的次要功能在仪表部件中是比较常见的。关于这个次要功能 LVGL 官方提供了三个函数来完成，如表 32.2.2.2 所示：

函数	描述
lv_meter_add_scale_lines()	添加刻度线指示刻度
lv_meter_set_indicator_start_value()	设置指示刻度的起始值
lv_meter_set_indicator_end_value()	设置指示刻度的终止值

表 32.2.2.2 绘制指示刻度相关函数

上述表中的函数也是按照某些顺序执行，首先我们调用函数 lv_meter_add_scale_lines 在仪表上添加指示刻度，该函数返回指示刻度指针，然后程序根据指示刻度指针设置指示刻度的起始值和终止值，起始值和终止值无需顺序调用。注意：如果仪表不设置起始值，则指示刻度从 0 开始；如果仪表只添加指示刻度不设置起始值和终止值，则指示刻度不会被绘画出来。

下面我们分别地讲解这三个函数到底有哪些形参如下所示：

(1) lv_meter_add_scale_lines 函数

该函数的作用是添加刻度线指示刻度，该函数原型如下所示：

```
lv_meter_indicator_t *lv_meter_add_scale_lines(lv_obj_t *obj,
                                               lv_meter_scale_t *scale,
                                               lv_color_t color_start,
                                               lv_color_t color_end,
                                               bool local,
                                               int16_t width_mod)
```

该函数的形参，如表 32.2.2.3 所示：

参数	描述
obj	指向 mater 对象的指针

scale	指向刻度对象
color_start	开始颜色
color_end	终止颜色
local	true： 渐变，false： 渐变(注：这两种状态看不出有任何变化)
width_mod	指示刻度的宽度

表 32.2.2.3 函数 lv_meter_add_scale_lines()形参描述

返回值：返回指示刻度指针。

(2) lv_meter_set_indicator_start_value 函数

该函数的作用是设置指示刻度的起始值，该函数原型如下所示：

```
void lv_meter_set_indicator_start_value(lv_obj_t *obj,
                                        lv_meter_indicator_t *indic,
                                        int32_t value)
```

该函数的形参，如表 32.2.2.4 所示：

参数	描述
obj	指向 mater 对象的指针
indic	指向刻度对象
value	起始值

表 32.2.2.4 函数 lv_meter_set_indicator_start_value()形参描述

返回值：无。

(3) lv_meter_set_indicator_end_value 函数

该函数的作用是设置指示刻度的起始值，该函数原型如下所示：

```
void lv_meter_set_indicator_end_value(lv_obj_t *obj,
                                      lv_meter_indicator_t *indic,
                                      int32_t value)
```

该函数的形参，如表 32.2.2.4 所示：

参数	描述
obj	指向 mater 对象的指针
indic	指向刻度对象
value	起始值

表 32.2.2.4 函数 lv_meter_set_indicator_end_value()形参描述

返回值：无。

下面我们以一个实例来讲解这些函数到底如何使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())
LV_IMG_DECLARE(img_hand)

void lv_mainstart(void)
{
    lv_obj_t* meter = lv_meter_create(lv_scr_act());   /* 定义并创建仪表 */
    lv_obj_set_width(meter, scr_act_height());         /* 设置仪表宽度 */
    lv_obj_set_height(meter, scr_act_height());        /* 设置仪表高度 */
    /* 设置仪表位置 */
    lv_obj_center(meter);
    /* 设置仪表刻度 */
```

```
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);   /* 定义并添加刻度 */
    /* 设置小刻度 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1, 10,
                             lv_palette_main(LV_PALETTE_GREY));
    /* 设置主刻度 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1, 10,
                                   lv_color_black(), scr_act_height() / 30);
    /* 添加指针图像 */
    lv_meter_indicator_t * indic = lv_meter_add_needle_img(meter,
                                                           scale,
                                                           &img_hand, 5, 5);
    /* 设置指针指向的数值 */
    lv_meter_set_indicator_value(meter, indic,54);
    /* 添加指示刻度，起始颜色为浅蓝色，终止颜色为深蓝色，状态为 true，指示刻度为 10 */
    indic = lv_meter_add_scale_lines(meter, scale,
                                     lv_palette_darken(LV_PALETTE_BLUE,1),
                                     lv_palette_darken(LV_PALETTE_BLUE, 4),
                                     true, 10);
    /* 设置指示刻度起始值为 50 */
    lv_meter_set_indicator_start_value(meter, indic, 50);
    /* 设置指示刻度终止值为 100 */
    lv_meter_set_indicator_end_value(meter, indic, 100);
}
```

从上述源码可知：我们在仪表添加了指示刻度，该指示刻度由浅蓝色到深蓝色渐变，然后我们设置了指示刻度的起始值和终止值。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 32.2.2.3 所示：

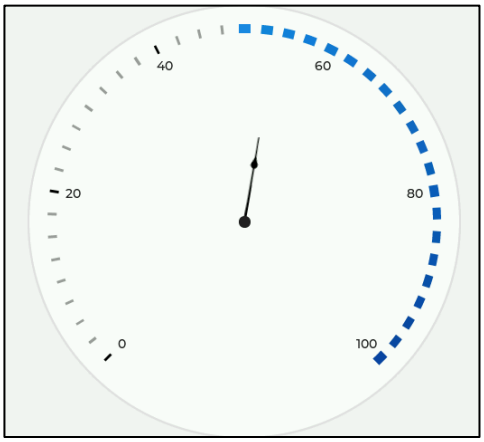


图 32.2.2.3 添加指示刻度

上图中，我们会发现刻度和指示刻度是不一样的，显然地，仪表设置指示刻度能使仪表本身非常美观。

3. 仪表弧线指示器

仪表弧线指示器也是一样比较常见的仪表功能，该次要功能和指示刻度具有相似的效果，主要也是引导和提示用户的功能。下面我们以一个示意图来讲解仪表弧线指示器到底是什么，如图 32.2.2.4 所示：

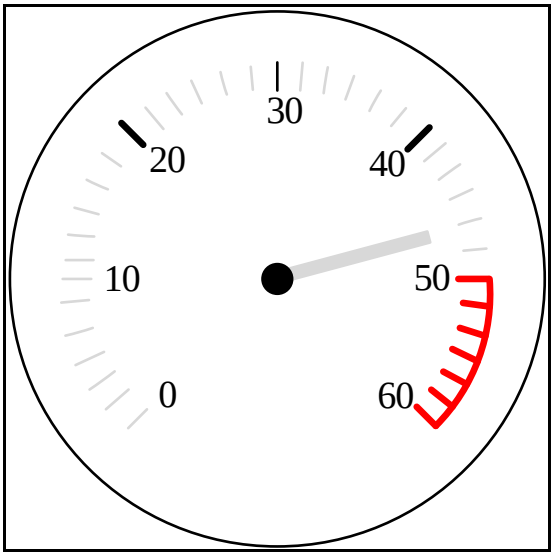


图 32.2.2.4 添加仪表弧线指示器

上图中，我们把仪表添加了指示刻度和弧线指示器的效果图，这个效果图类似与汽车油表的油量极限区域。关于这个次要功能 LVGL 官方提供三个函数来完成，这些函数如表 32.2.2.5 所示：

函数	描述
lv_meter_add_arc()	添加弧线指示器
lv_meter_set_indicator_start_value()	设置指示刻度的起始值
lv_meter_set_indicator_end_value()	设置指示刻度的终止值

表 32.2.2.5 绘制弧线指示器相关函数

上表中第二和第三个函数我们前面已经讲解了，这里我们无需重复讲解，接下来我们来看一下第一个函数到底有哪些形参，如下所示：

lv_meter_add_arc 函数

该函数的作用是添加弧线指示器，该函数原型如下所示：

```
lv_meter_indicator_t *lv_meter_add_arc(lv_obj_t *obj,
                                       lv_meter_scale_t *scale,
                                       uint16_t width,
                                       lv_color_t color,
                                       int16_t r_mod)
```

该函数的形参，如表 32.2.2.6 所示：

参数	描述
obj	指向 mater 对象的指针
scale	指向刻度对象
width	弧线宽度
color	弧线颜色
r_mod	偏移值

表 32.2.2.6 函数 lv_meter_add_arc()形参描述

返回值：无。

下面我们以一个实例来讲解这些函数到底如何使用，如下源码所示：

```
/* 获取当前活动屏幕的宽高 */
#define scr_act_width() lv_obj_get_width(lv_scr_act())
#define scr_act_height() lv_obj_get_height(lv_scr_act())
LV_IMG_DECLARE(img_hand)

void lv_mainstart(void)
{
    lv_obj_t* meter = lv_meter_create(lv_scr_act());   /* 定义并创建仪表 */
    lv_obj_set_width(meter, scr_act_height());         /* 设置仪表宽度 */
    lv_obj_set_height(meter, scr_act_height());        /* 设置仪表高度 */
    /* 设置仪表位置 */
    lv_obj_center(meter);
    /* 设置仪表刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);   /* 定义并添加刻度 */
    /* 设置小刻度 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1,10,
                             lv_palette_main(LV_PALETTE_GREY));
    /* 设置主刻度 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1, 10,
                                   lv_color_black(), scr_act_height() / 30);
    /* 添加指针图像 */
    lv_meter_indicator_t * indic = lv_meter_add_needle_img(meter,
                                                           scale,
                                                           &img_hand, 5, 5);
    /* 设置指针指向的数值 */
    lv_meter_set_indicator_value(meter, indic,54);
    /* 添加指示刻度，起始颜色为浅蓝色，终止颜色为深蓝色，状态为 true，指示刻度为 10 */
    indic = lv_meter_add_scale_lines(meter, scale,
                                     lv_palette_darken(LV_PALETTE_BLUE,1),
                                     lv_palette_darken(LV_PALETTE_BLUE, 4),
                                     true, 10);
    /* 设置指示刻度起始值为 50 */
    lv_meter_set_indicator_start_value(meter, indic, 50);
    /* 设置指示刻度终止值为 100 */
    lv_meter_set_indicator_end_value(meter, indic, 100);

    /* 添加弧线指示器，弧线宽度为 10，颜色为红色，偏移 10 */
    indic = lv_meter_add_arc(meter, scale, 10,
                             lv_palette_main(LV_PALETTE_RED), 10);
    lv_meter_set_indicator_start_value(meter, indic, 50);
    lv_meter_set_indicator_end_value(meter, indic, 100);
}
```

从上述源码可知：我们调用函数 lv_meter_add_arc 添加弧线指示器，该弧线指示器的宽度为 10，弧线颜色为红色，该弧线偏移 10。上述源码可在 PC 模拟器或者在开发板上运行，其程序效果图如下图 32.2.2.5 所示：

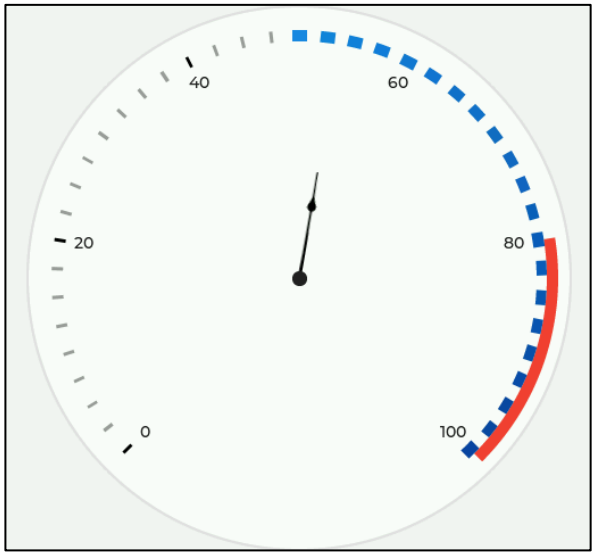


图 32.2.2.5 添加弧线指示器

从上图可知：弧线的指示器在 80~100 的刻度范围内，然后该弧线往外偏移 10 步长。

32.3 仪表部件 API 函数

LVGL 提供给用户的 meter 部件相关函数，如表 32.3.1 所示：

函数	描述
lv_meter_create()	创建一个 Meter 对象
lv_meter_add_scale()	给仪表加一个新刻度
lv_meter_set_scale_ticks()	设置小刻度的属性
lv_meter_set_scale_major_ticks()	设置主刻度的属性
lv_meter_set_scale_range()	设置比例的值和角度范围
lv_meter_add_needle_line()	在刻度处添加针线指示器
lv_meter_add_needle_img()	添加针形图像指示器刻度
lv_meter_add_arc()	添加一个圆弧指示器的比例
lv_meter_add_scale_lines()	添加刻度线指示器
lv_meter_set_indicator_value()	设置指标的值
lv_meter_set_indicator_start_value()	设置指标的起始值
lv_meter_set_indicator_end_value()	设置指标的终止值

表 32.3.1 meter 部件 API 函数

上述表中的函数笔者已经在上一个小节讲解过，这里我们就无需重复讲解了。

32.4 仪表控件实验

32.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数有三个实例，它们分别讲述三种类型的图表的作用。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行当中。

该实验的实验工程，请参考《LVGL 例程 32 lv_meter(仪表)》。

32.4.2 软件设计

32.4.2.1 程序流程图

本实验的程序流程图，如下图 32.4.2.1 所示：

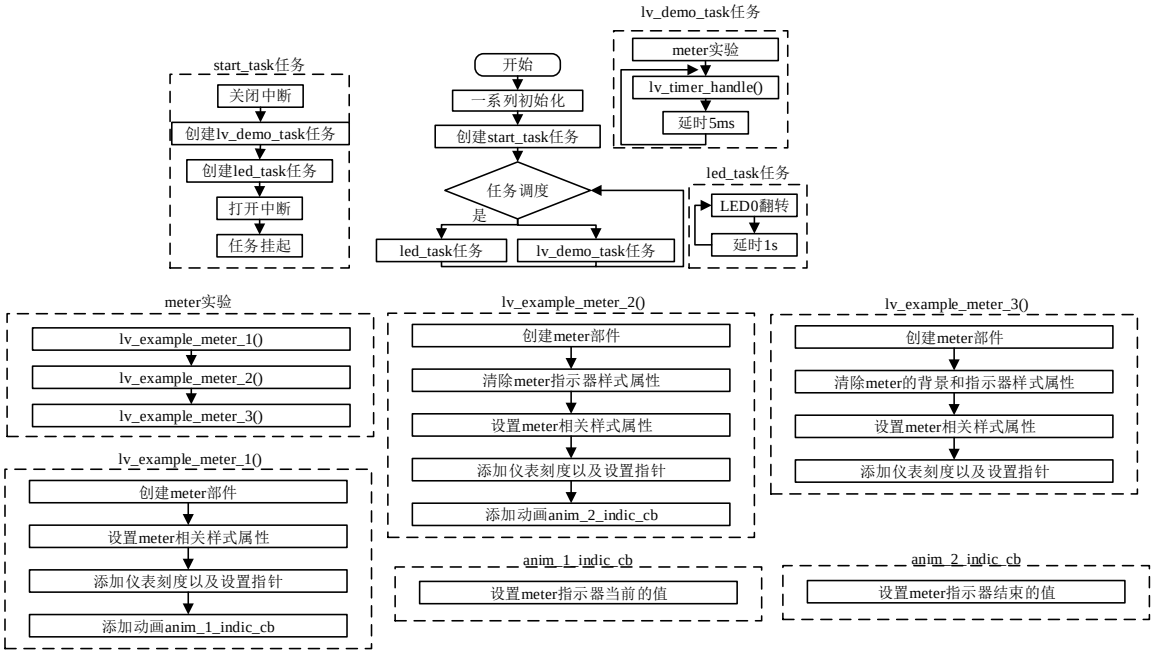


图 32.4.2.1 meter 部件实验流程图

32.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief   LVGL 演示
 * @param   无
```

```
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_meter_1();
    lv_example_meter_2();
    lv_example_meter_3();
}
```

此函数具有三个小实验，下面我们分别地介绍这些实验实现的功能。

(1) lv_example_meter_1 函数

```
/**
 * @brief  例 1
 * @param  无
 * @return 无
 */
static void lv_example_meter_1(void)
{
    const lv_font_t* font;
    /* 根据屏幕大小设置字体 */
    if (scr_act_width() <= 320)
        font = &lv_font_montserrat_10;
    else if (scr_act_width() <= 480)
        font = &lv_font_montserrat_14;
    else
        font = &lv_font_montserrat_16;

    /* 定义并创建仪表 */
    lv_obj_t* meter = lv_meter_create(lv_scr_act());
    /* 设置仪表宽度 */
    lv_obj_set_width(meter, scr_act_height() * 0.4);
    /* 设置仪表高度 */
    lv_obj_set_height(meter, scr_act_height() * 0.4);
    /* 设置仪表位置 */
    lv_obj_align(meter, LV_ALIGN_CENTER, -scr_act_width() / 3, 0);
    /* 设置仪表文字字体 */
    lv_obj_set_style_text_font(meter, font, LV_PART_MAIN);
    /* 设置仪表刻度 */
    /* 定义并添加刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);
    /* 设置小刻度 */
    lv_meter_set_scale_ticks(meter, scale, 41, 1,
                             scr_act_height() / 80,
                             lv_palette_main(LV_PALETTE_GREY));
```

```
    /* 设置主刻度 */
    lv_meter_set_scale_major_ticks(meter, scale, 8, 1,
                                   scr_act_height() / 60, lv_color_black(),
                                   scr_act_height() / 30);
    /* 设置指针 */
    lv_meter_indicator_t* indic;
    indic = lv_meter_add_arc(meter, scale, 2,
                             lv_palette_main(LV_PALETTE_BLUE), 0);
    lv_meter_set_indicator_start_value(meter, indic, 0);
    lv_meter_set_indicator_end_value(meter, indic, 20);
    indic = lv_meter_add_scale_lines(meter, scale,
                                   lv_palette_main(LV_PALETTE_BLUE),
                                   lv_palette_main(LV_PALETTE_BLUE), false, 0);
    lv_meter_set_indicator_start_value(meter, indic, 0);
    lv_meter_set_indicator_end_value(meter, indic, 20);
    indic = lv_meter_add_arc(meter, scale, 2,
                             lv_palette_main(LV_PALETTE_RED), 0);
    lv_meter_set_indicator_start_value(meter, indic, 80);
    lv_meter_set_indicator_end_value(meter, indic, 100);
    indic = lv_meter_add_scale_lines(meter, scale,
                              lv_palette_main(LV_PALETTE_RED),
                              lv_palette_main(LV_PALETTE_RED), false, 0);
    lv_meter_set_indicator_start_value(meter, indic, 80);
    lv_meter_set_indicator_end_value(meter, indic, 100);
    indic = lv_meter_add_needle_line(meter, scale, 4,
                                     lv_palette_main(LV_PALETTE_GREY), -10);
    lv_obj_set_user_data(meter, indic);
    /* 设置指针动画 */
    lv_anim_t a;
    lv_anim_init(&a);
    lv_anim_set_exec_cb(&a,(lv_anim_exec_xcb_t)anim_1_indic_cb);
    lv_anim_set_var(&a, meter);
    lv_anim_set_values(&a, 0, 100);
    lv_anim_set_time(&a, 2000);
    lv_anim_set_repeat_delay(&a, 100);
    lv_anim_set_playback_time(&a, 500);
    lv_anim_set_playback_delay(&a, 100);
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);
    lv_anim_start(&a);
}
```

此函数首先判断屏幕的大小来决定使用那个字库，然后程序创建 meter 部件并设置相关的样式属性，其次在 meter 部件添加指针等相关属性设置，最后我们使用动画的形式驱动指针旋转，驱动指针旋转是由 anim_1_indic_cb 动画回调函数实现，该回调函数如下所示：

```
/**
 * @brief  例 1 指针动画
 * @param  meter：对象
 * @param  value：数值
 * @return 无
 */
static void anim_1_indic_cb(lv_obj_t* meter, int32_t value)
{
    lv_meter_indicator_t* indic = (lv_meter_indicator_t*)meter->user_data;

    lv_meter_set_indicator_value(meter, indic, value);
}
```

此函数主要设置 meter 部件指针的当前数值。

(2) lv_example_meter_2 函数

```
/**
 * @brief  例 2
 * @param  无
 * @return 无
 */
static void lv_example_meter_2(void)
{
    const lv_font_t* font;
    /* 根据屏幕大小设置字体 */
    if (scr_act_width() <= 320)
        font = &lv_font_montserrat_8;
    else if (scr_act_width() <= 480)
        font = &lv_font_montserrat_10;
    else
        font = &lv_font_montserrat_14;

    /* 定义并创建仪表 */
    meter_2 = lv_meter_create(lv_scr_act());
    /* 设置仪表宽度 */
    lv_obj_set_width(meter_2, scr_act_height() * 0.4);
    /* 设置仪表高度 */
    lv_obj_set_height(meter_2, scr_act_height() * 0.4);
    /* 设置仪表位置 */
    lv_obj_center(meter_2);
    /* 设置仪表文字字体 */
    lv_obj_set_style_text_font(meter_2, font, LV_PART_MAIN);
    /* 删除仪表指针样式 */
    lv_obj_remove_style(meter_2, NULL, LV_PART_INDICATOR);
    /* 设置仪表刻度 */
```

```
    lv_meter_scale_t* scale = lv_meter_add_scale(meter_2);
    lv_meter_set_scale_ticks(meter_2, scale, 11, 2,
                             scr_act_height() / 15,
                             lv_palette_main(LV_PALETTE_GREY));
    lv_meter_set_scale_major_ticks(meter_2, scale, 1, 2,
                                   scr_act_height() / 15, lv_color_hex3(0xEEE),
                                   scr_act_height() / 40);
    lv_meter_set_scale_range(meter_2, scale, 0, 100, 270, 90);
    /* 设置指针 */
    lv_meter_indicator_t* indic1 = lv_meter_add_arc(meter_2, scale,
                                    scr_act_height() / 45,
                                    lv_palette_main(LV_PALETTE_RED), 0);
    lv_meter_indicator_t* indic2 = lv_meter_add_arc(meter_2, scale,
                                    scr_act_height() / 45,
                                    lv_palette_main(LV_PALETTE_GREEN),
                                    -scr_act_height() / 45);
    lv_meter_indicator_t* indic3 = lv_meter_add_arc(meter_2, scale,
                                    scr_act_height() / 45,
                                    lv_palette_main(LV_PALETTE_BLUE),
                                    -scr_act_height() / 22.5);
    /* 设置指针动画 */
    lv_anim_t a;
    lv_anim_init(&a);
    lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t)anim_2_indic_cb);
    lv_anim_set_values(&a, 0, 100);
    lv_anim_set_repeat_delay(&a, 100);
    lv_anim_set_playback_delay(&a, 100);
    lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);

    lv_anim_set_time(&a, 2000);
    lv_anim_set_playback_time(&a, 500);
    lv_anim_set_var(&a, indic1);
    lv_anim_start(&a);

    lv_anim_set_time(&a, 1000);
    lv_anim_set_playback_time(&a, 1000);
    lv_anim_set_var(&a, indic2);
    lv_anim_start(&a);

    lv_anim_set_time(&a, 1000);
    lv_anim_set_playback_time(&a, 2000);
    lv_anim_set_var(&a, indic3);
    lv_anim_start(&a);
```

```
}
```

此函数首先判断屏幕的大小来决定使用那个字库，然后程序创建 meter 部件并删除指示器的样式及设置相关样式属性，其次在 meter 部件添加指针等相关属性设置，最后我们使用动画的形式驱动指针旋转，驱动指针旋转是由 anim_2_indic_cb 动画回调函数实现，该回调函数如下所示：

```
/**
 * @brief  例 2 指针动画
 * @param  meter：对象
 * @param  value：数值
 * @return 无
 */
static void anim_2_indic_cb(lv_meter_indicator_t* indic, int32_t value)
{
    lv_meter_set_indicator_end_value(meter_2, indic, value);
}
```

此函数主要设置 meter 部件指针结束的数值。

(3) lv_example_meter_3 函数

```
/**
 * @brief  例 3
 * @param  无
 * @return 无
 */
static void lv_example_meter_3(void)
{
    /* 定义并创建仪表 */
    lv_obj_t* meter = lv_meter_create(lv_scr_act());
    /* 移除背景样式 */
    lv_obj_remove_style(meter, NULL, LV_PART_MAIN);
    /* 移除指针样式 */
    lv_obj_remove_style(meter, NULL, LV_PART_INDICATOR);
    /* 设置仪表宽度 */
    lv_obj_set_width(meter, scr_act_height() * 0.4);
    /* 设置仪表高度 */
    lv_obj_set_height(meter, scr_act_height() * 0.4);
    /* 设置仪表位置 */
    lv_obj_align(meter, LV_ALIGN_CENTER, scr_act_width() / 3, 0);
    /* 手动更新参数 */
    lv_obj_update_layout(meter);
    /* 设置仪表刻度 */
    lv_meter_scale_t* scale = lv_meter_add_scale(meter);
    lv_meter_set_scale_ticks(meter, scale, 0, 0, 0, lv_color_black());
    lv_meter_set_scale_range(meter, scale, 0, 100, 360, 0);
    /* 设置指针 */
```

```
    lv_coord_t indic_w = scr_act_height() * 0.05;
    lv_meter_indicator_t* indic1 = lv_meter_add_arc(meter, scale, indic_w,
                                   lv_palette_main(LV_PALETTE_ORANGE), 0);
    lv_meter_set_indicator_start_value(meter, indic1, 0);
    lv_meter_set_indicator_end_value(meter, indic1, 40);

    lv_meter_indicator_t* indic2 = lv_meter_add_arc(meter, scale, indic_w,
                                   lv_palette_main(LV_PALETTE_YELLOW), 0);
    /*Start from the previous*/
    lv_meter_set_indicator_start_value(meter, indic2, 40);
    lv_meter_set_indicator_end_value(meter, indic2, 80);

    lv_meter_indicator_t* indic3 = lv_meter_add_arc(meter, scale, indic_w,
                                   lv_palette_main(LV_PALETTE_DEEP_ORANGE), 0);
    /*Start from the previous*/
    lv_meter_set_indicator_start_value(meter, indic3, 80);
    lv_meter_set_indicator_end_value(meter, indic3, 100);
}
```

上述函数非常简单，主要创建 meter 部件，然后删除背景和指示器的样式属性，最后我们设置它的指针属性。

32.4.3 下载验证

把工程编译并下载到我们的开发板中，如图 32.4.3.1 所示：

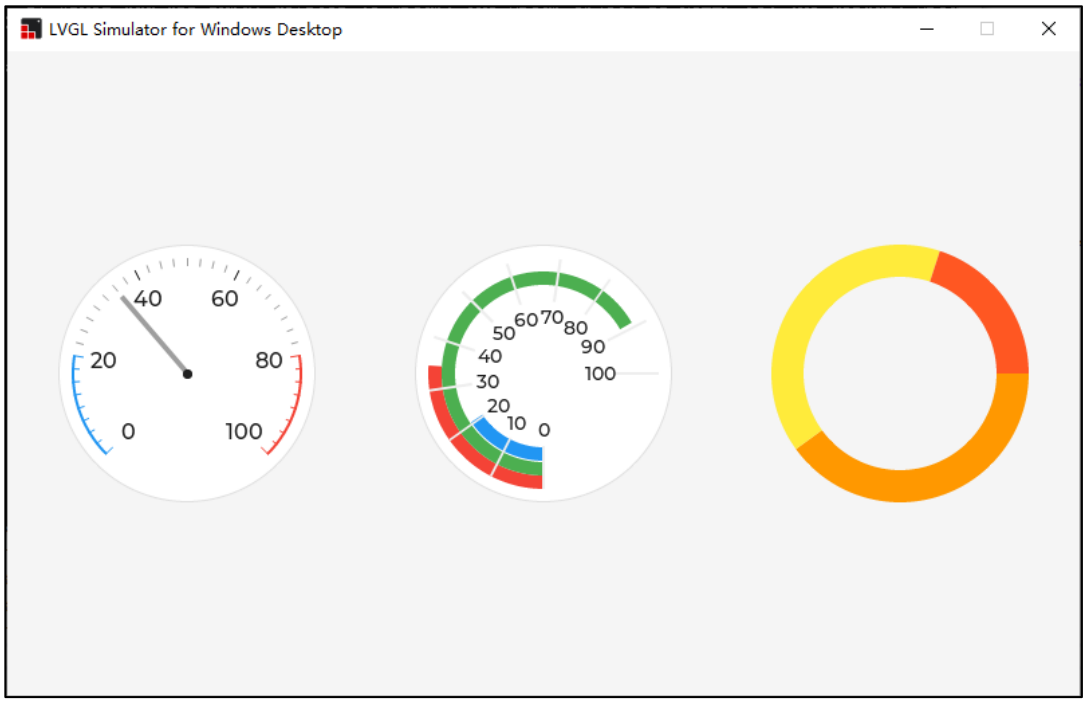


图 32.4.3.1 meter 部件实验

第三十三章 消息框部件(lv_msgbox)

“消息”框充当弹出窗口。它们由背景容器、标题、可选关闭按钮、文本和可选按钮构建而成。文本将自动分成多行，高度将自动设置为包括文本和按钮。消息框部件可以设置模态或者非模态形式，模态是指“消息”框弹出时，除了消息框区域可点击，其他区域不可点击。

本章分为如下几部分：

33.1 消息框部件的组成
33.2 消息框部件的相关知识
33.3 消息框部件 API 函数
33.4 消息框控件实验

33.1 消息框部件的组成

消息框部件是由其他小部件构建而成的，例如 lv_obj、lv_btn、lv_label 和 lv_btnmatrix 部件。这些部件如何构建消息框部件的呢，我们来看一下下面的示意图，如图 33.1.1 所示：

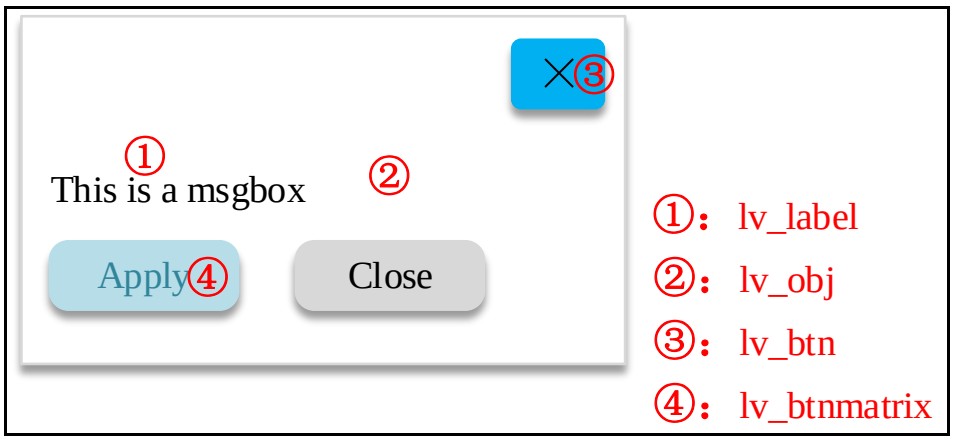


图 33.1.1 msgbox 部件组成

上图中，msgbox 部件由四个小部件组成，这些部件的知识可参考前面章节的内容。

33.2 消息框部件的相关知识

33.2.1 创建消息框部件

msgbox 部件只需调用函数 lv_msgbox_create 创建，该函数具有五个形参，如以下所示：

① parent：如果该形参为 NULL，则该 msgbox 部件为模态形式。
② title：设置 msgbox 消息框的标题。
③ txt：设置 msgbox 消息框的文本(提示文本)。
④ btn_txts[]：按钮文本的数组，该数组的设置请参考第十三章节。
⑤ add_close_btn：添加/不添加关闭按钮。

下面笔者使用一个实例来讲解 lv_msgbox_create 函数的使用，如下源码所示：

```
void lv_mainstart(void)
{
    static const char* btns[] = { "Apply", "Close", "" };
```

```
    lv_obj_t* lv_msgbox = lv_msgbox_create(NULL,"Hello","ALIENTEK", btns,true);
    lv_obj_center(lv_msgbox);
}
```

从上述源码可知：我们调用函数 lv_msgbox_create 创建一个消息框，该消息框标题为“Hello”,消息的内容为“ALIENTEK”，而消息按键添加了“Apply”和“Close”按键并使能关闭按钮。注意：如果该函数的第一个形参为 NULL，则该消息框模态形式，该形式不能点击消息框外的部件，只能点击消息框内的部件，如下图 33.2.1.1 所示：

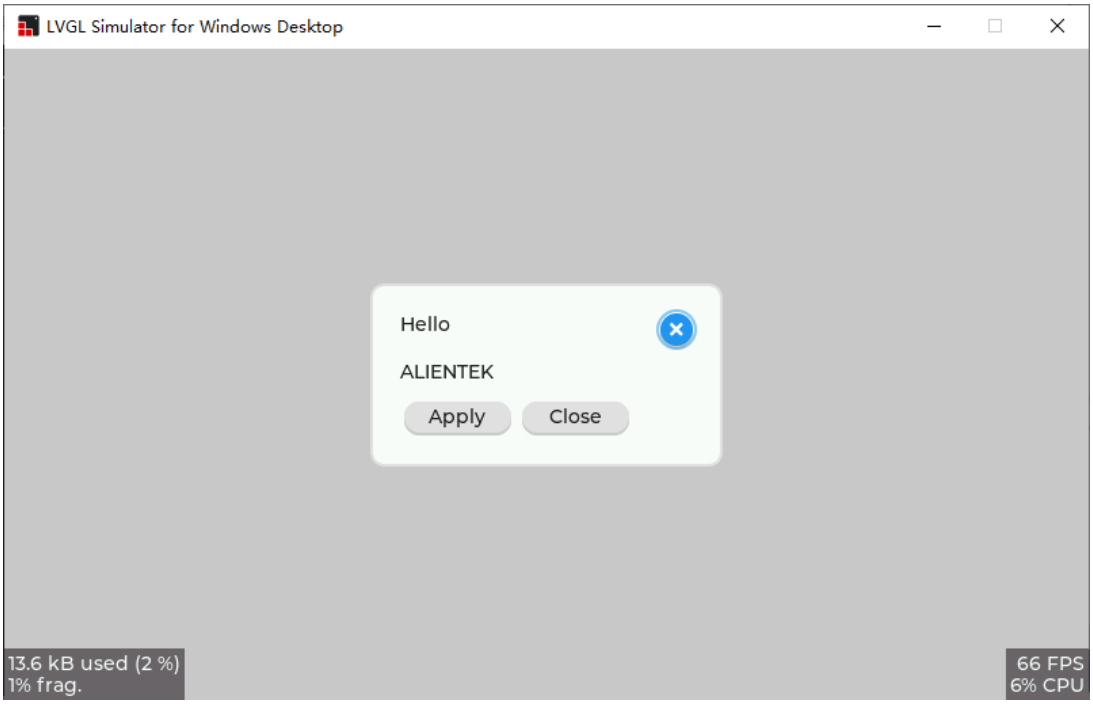


图 33.2.1.1 创建消息框部件

33.2.2 获取消息框的组成部件

前面我们已经了解了消息框部件是由四个小部件构建而成，这些部件获取方法可参考以下函数：

```
lv_obj_t * lv_msgbox_get_title(lv_obj_t * mbox);
lv_obj_t * lv_msgbox_get_close_btn(lv_obj_t * mbox);
lv_obj_t * lv_msgbox_get_text(lv_obj_t * mbox);
lv_obj_t * lv_msgbox_get_btns(lv_obj_t * mbox);
```

注意：这些函数获取的部件对象时，我们可根据这些对象设置相应的样式。例如修改消息框的 btn 部件的样式，可调用 lv_msgbox_get_close_btn 函数获取 btn 对象，然后根据对象设置样式。

33.2.3 关闭消息框部件

如果用户想手动关闭消息框，可调用 lv_msgbox_close 函数关闭消息框。

33.2.4 消息框部件事件

如果消息框部件设置了回调函数，则 LV_EVENT_VALUE_CHANGED 事件类型会被触发。该回调函数可调用 lv_event_get_target 函数获取点击的 btnmatrix 和 btn 按键对象。如果获取消

息部件对象，则调用 lv_event_get_current_target 函数获取；如果获取点击的按键文本和索引，则分别使用 lv_msgbox_get_active_btn_text 和 lv_msgbox_get_active_btn 函数获取。

33.3 消息框部件 API 函数

LVGL 提供给用户的 msgbox 部件相关函数，如表 33.3.1 所示：

函数	描述
lv_msgbox_create()	创建消息框部件
lv_msgbox_get_title()	获取消息框标题文本对象
lv_msgbox_get_close_btn()	获取 btn 部件对象
lv_msgbox_get_text()	获取消息框提示文本对象
lv_msgbox_get_content()	获取消息框内容对象
lv_msgbox_get_btns()	获取按键数组对象
lv_msgbox_get_active_btn()	获取按键对象索引
lv_msgbox_get_active_btn_text()	获取按键文本
lv_msgbox_close()	关闭消息框
lv_msgbox_close_async()	异步关闭消息框

表 33.3.1 msgbox 部件相关函数描述

接下来我们来讲解 LVGL lv_msgbox 部件常用的函数有哪些：

1. lv_msgbox_create 函数

该函数的作用是创建一个 msgbox 对象，其函数原型如下所示：

```
lv_obj_t *lv_msgbox_create(lv_obj_t *parent,
                           const char *title,
                           const char *txt,
                           const char *btn_txts[],
                           bool add_close_btn)
```

该函数的形参描述如表 33.3.2 所示：

参数	描述
parent	指向父类指针，如果设置为 NULL，则消息框为模态
title	消息框标题
txt	消息框文本
btn_txts	消息框按键文本数组
add_close_btn	true:添加关闭按钮

表 33.3.2 函数 lv_msgbox_create()形参描述

返回值：指向创建的 msgbox 指针。

2. lv_msgbox_get_title 函数

该函数的作用是获取 msgbox 标题文本对象，其函数原型如下所示：

```
lv_obj_t *lv_msgbox_get_title(lv_obj_t *obj)
```

该函数的形参描述如表 33.3.3 所示：

参数	描述
obj	指向对象指针

表 33.3.3 函数 lv_msgbox_get_title()形参描述

返回值：msgbox 标题文本对象。

3. lv_msgbox_get_close_btn()函数

该函数的作用是获取 msgbox 关闭按键对象，其函数原型如下所示：

```
lv_obj_t *lv_msgbox_get_close_btn(lv_obj_t *obj)
```

该函数的形参描述如表 33.3.4 所示：

参数	描述
obj	指向对象指针

表 33.3.4 函数 lv_msgbox_get_close_btn()形参描述

返回值：获取 msgbox 关闭按键对象。

4. lv_msgbox_get_text 函数

该函数的作用是获取 msgbox 提示文本对象，其函数原型如下所示：

```
lv_obj_t *lv_msgbox_get_text(lv_obj_t *obj)
```

该函数的形参描述如表 33.3.5 所示：

参数	描述
obj	指向对象指针

表 33.3.5 函数 lv_msgbox_get_text()形参描述

返回值：获取 msgbox 提示文本对象。

5. lv_msgbox_close 函数

该函数的作用是关闭消息框，其函数原型如下所示：

```
void lv_msgbox_close(lv_obj_t *mbox)
```

该函数的形参描述如表 33.3.6 所示：

参数	描述
mbox	指向消息框对象指针

表 33.3.6 函数 lv_msgbox_close()形参描述

返回值：无。

33.4 消息框控件实验

33.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 colorwheel 和 btn 部件，当用户滑动色环部件时并按下 btn 部件，该部件会自动创建 msgbox 部件，如果用户按下 msgbox 部件中的 apply 按键，则系统会根据色环选择的颜色设置屏幕背景颜色并关闭 msgbox 部件；如果用户按下 msgbox 部件中的 Close 按键，则系统会关闭 msgbox 部件。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 33 lv_msgbox(消息框)》。

33.4.2 软件设计

33.4.2.1 程序流程图

本实验的程序流程图，如下图 33.4.2.1.1 所示：

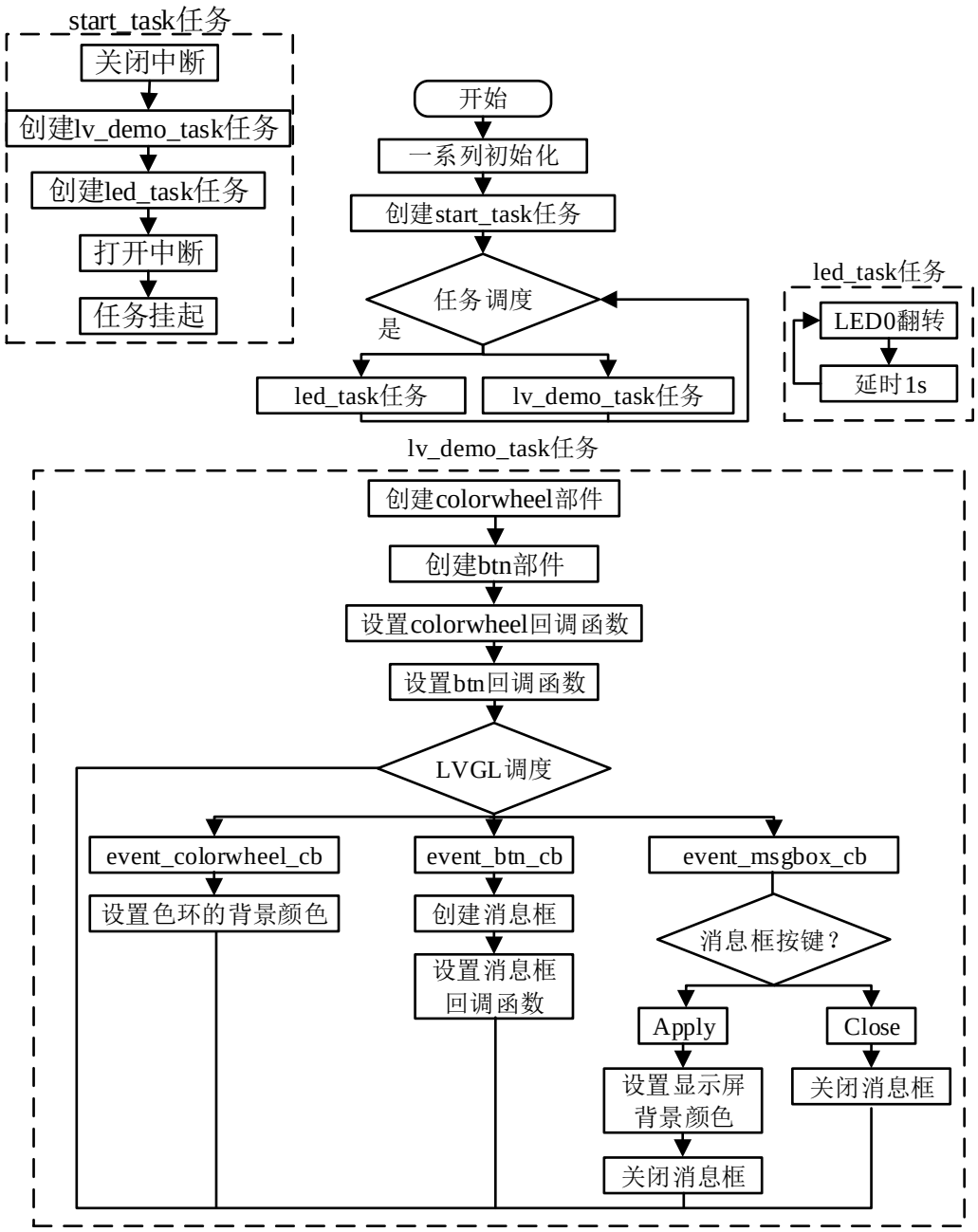


图 33.4.2.1.1 msgbox 实验流程图

33.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_msgbox(void)
```

```
{
    obj = lv_obj_create(lv_scr_act());
    lv_obj_set_width(obj, scr_act_width());
    lv_obj_set_height(obj, scr_act_height());
    lv_obj_center(obj);
    lv_obj_update_layout(obj);

    /* 创建色环部件 */
    colorwheel = lv_colorwheel_create(obj, true);
    /* 设置色环部件样式属性 */
    lv_obj_set_width(colorwheel, LV_MIN(lv_obj_get_width(obj),
                lv_obj_get_height(obj)) * 0.6);
    lv_obj_set_height(colorwheel, LV_MIN(lv_obj_get_width(obj),
                 lv_obj_get_height(obj)) * 0.6);
    lv_obj_align(colorwheel, LV_ALIGN_CENTER, -lv_obj_get_width(obj) / 5, 0);
    lv_obj_set_style_arc_width(colorwheel, lv_obj_get_width(obj) * 0.05,
                           LV_PART_MAIN);
    lv_obj_update_layout(colorwheel);

    lv_obj_t* obj_color = lv_obj_create(obj);
    lv_obj_set_width(obj_color, lv_obj_get_width(colorwheel)
                     - lv_obj_get_height(obj) / 4);
    lv_obj_set_height(obj_color, lv_obj_get_height(colorwheel)
                     - lv_obj_get_height(obj) / 4);
    lv_obj_align_to(obj_color, colorwheel, LV_ALIGN_CENTER, 0, 0);
    lv_obj_set_style_radius(obj_color, LV_RADIUS_CIRCLE, LV_PART_MAIN);
    lv_obj_set_style_bg_color(obj_color, lv_colorwheel_get_rgb(colorwheel),
                          LV_PART_MAIN);
    /* 设置色环部件回调函数 */
    lv_obj_add_event_cb(colorwheel, event_colorwheel_cb,
                    LV_EVENT_VALUE_CHANGED, obj_color);

    /* 创建 btn 部件 */
    lv_obj_t* btn = lv_btn_create(obj);
    /* 设置 btn 部件样式属性 */
    lv_obj_set_width(btn, lv_obj_get_width(obj) * 0.2);
    lv_obj_set_height(btn, lv_obj_get_height(obj) * 0.2);
    lv_obj_align(btn, LV_ALIGN_CENTER, lv_obj_get_width(obj) / 4, 0);
    lv_obj_t* label_btn = lv_label_create(btn);
    lv_obj_center(label_btn);
    lv_label_set_text(label_btn, "Set");
    lv_obj_set_style_text_align(label_btn, LV_TEXT_ALIGN_CENTER, LV_PART_MAIN);
    /* 设置 btn 部件回调函数 */
```

```
    lv_obj_add_event_cb(btn, event_btn_cb, LV_EVENT_CLICKED, NULL);
}


/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_msgbox();
}
```

上述源码主要创建 colorwheel 和 btn 部件，设置这些部件的样式属性，最后设置这些部件的回调函数 event_colorwheel_cb 和 event_btn_cb。

接下来分别地介绍以上这几个回调函数的代码。

(1) event_colorwheel_cb 函数

```
/**
 * @brief  色环事件回调
 * @param  无
 * @return 无
 */
static void event_colorwheel_cb(lv_event_t * e)
{
    lv_obj_t* colorwheel = lv_event_get_target(e);
    lv_obj_t* obj_color = (lv_obj_t*)e->user_data;

    lv_obj_set_style_bg_color(obj_color, lv_colorwheel_get_rgb(colorwheel),
                              LV_PART_MAIN);
}
```

此函数主要设置 colorwheel 部件的背景颜色。

(2) event_btn_cb 函数

```
/**
 * @brief  按钮事件回调
 * @param  无
 * @return 无
 */
static void event_btn_cb(lv_event_t* e)
{
    lv_obj_t* btn = lv_event_get_target(e);
    static const char* btns[] = { "Apply", "Close", "" };

    /* 定义并创建消息框 */
    msgbox = lv_msgbox_create(NULL,
```

```
                                "Set", "Set the background color as
                                  the color wheel color.", btns, true);
    /* 设置消息框宽度 */
    lv_obj_set_width(msgbox, lv_obj_get_width(obj) * 0.6);
    /* 设置消息框高度 */
    lv_obj_set_height(msgbox, lv_obj_get_height(obj) * 0.6);
    /* 设置消息框事件回调 */
    lv_obj_add_event_cb(msgbox, event_msgbox_cb, LV_EVENT_VALUE_CHANGED, NULL);
    /* 设置消息框位置 */
    lv_obj_center(msgbox);
}
```

此回调函数主要创建消息框部件以及设置该部件的样式属性，然后程序设置消息框部件的回调函数，该回调函数如下所示：

```
/**
 * @brief  消息框事件回调
 * @param  无
 * @return 无
 */
static void event_msgbox_cb(lv_event_t* e)
{
    /* 获取消息框按钮文本 */
    if (0 == strcmp("Apply", lv_msgbox_get_active_btn_text(msgbox)))
    {
        lv_obj_set_style_bg_color(obj, lv_colorwheel_get_rgb(colorwheel),
                                  LV_PART_MAIN);
        /* 关闭消息框 */
        lv_msgbox_close(msgbox);
    }
    else if (0 == strcmp("Close", lv_msgbox_get_active_btn_text(msgbox)))
    {
        /* 关闭消息框 */
        lv_msgbox_close(msgbox);
    }
}
```

此回调函数主要获取按下按键的文本，如果按下 Apple 按键，则设置 obj 背景颜色且关闭消息框；如果按下 Close 按键，则程序会关闭消息框。

33.4.3 下载验证

把工程编译并下载到开发板中， 如图 33.4.3.1 所示：

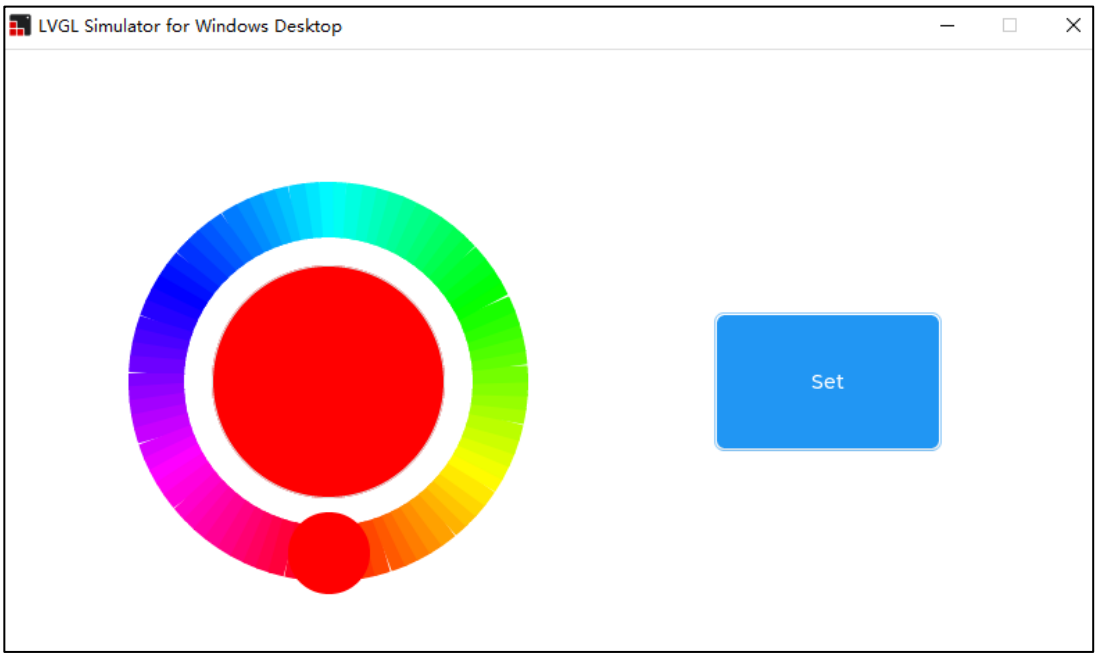


图 33.4.3.1 msgbox 实验主界面

当按下 set 按键时，创建 msgbox 消息框。如图 33.4.3.2 所示：

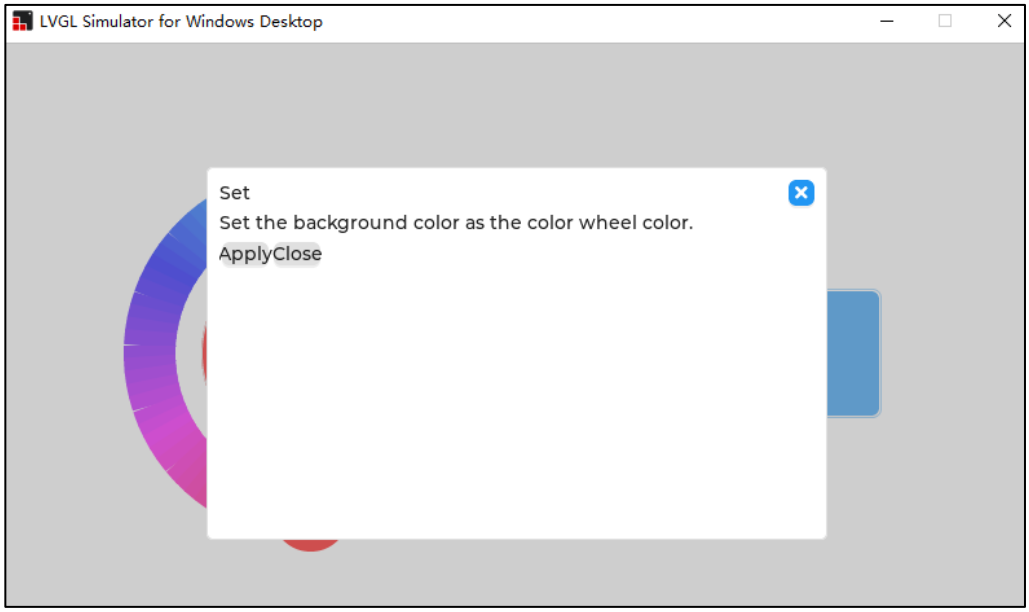


图 33.4.3.2 按下“set”按键

如果按下 Apple 按键，则设置 obj 背景颜色且删除消息框；如果按下 Close 按键，则删除消息框。如图 33.4.3.3 所示：

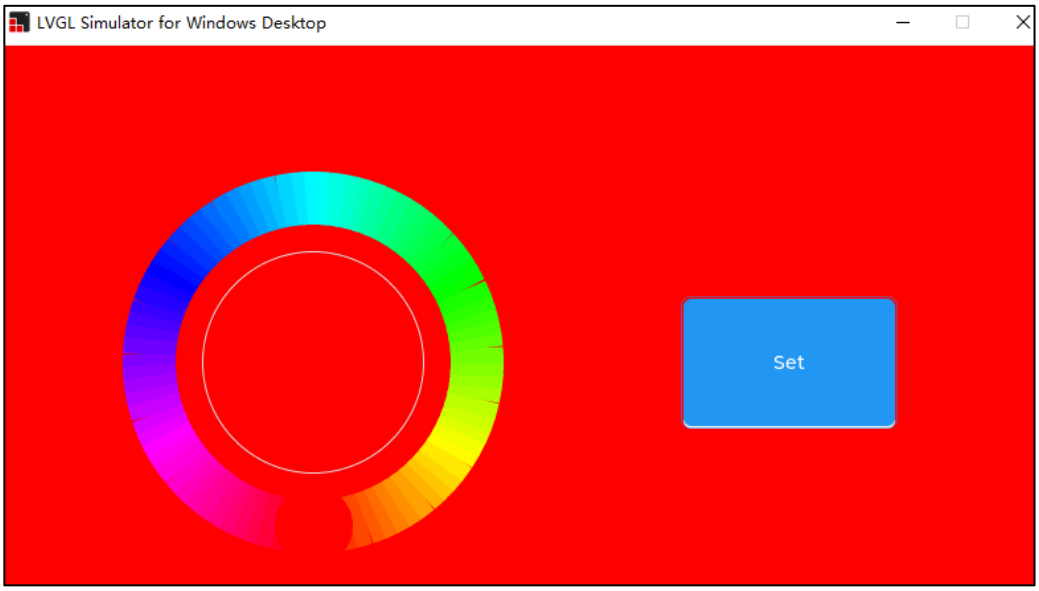


图 33.4.3.3 设置 obj 背景颜色

第三十四章 跨度部件(lv_span)

Span 部件是用来显示华丽文本的对象。它与 label 标签对象是不一样，Span 部件可以将具有不同字体、颜色和大小的文本呈现到 Span 部件对象当中。在 UI 设计中，Span 部件一般对文本进行装饰。

本章分为如下几部分：

34.1 Span 部件的组成

34.2 Span 部件的相关知识

34.3 Span 部件 API 函数

34.4 Span 控件实验

34.1 Span 部件的组成

span 部件组成与 lv_obj 部件一样，都是由 LV_PART_MAIN 部分构建而成的。至于 Span 部件属性设置，请大家参考第六章 6.3 小节。

34.2 Span 部件的相关知识

34.2.1 设置文本和样式

span 部件是用来描述文本以及修改文本样式的，在设置文本样式之前，我们必须创建 span 部件的描述符，然后设置它的文本样式属性，最后让 span 部件的样式通过其样式成员配置为普通样式，这样我们就可以使用跨度部件来修饰文本样式了。

下面我们使用一个实例来讲解本小节的知识点，首先我们列出跨度部件使用步骤，然后编写实例代码如下所示：

第一步：创建一个普通样式，如下源码所示：

```
static lv_style_t style;                    /* 定义普通样式 */
lv_style_init(&style);                      /* 通样式初始化 */
lv_style_set_border_width(&style, 1);       /* 设置边框宽度 */
/* 设置边框颜色 */
lv_style_set_border_color(&style, lv_palette_main(LV_PALETTE_ORANGE));
lv_style_set_pad_all(&style, 2);
```

注意：这一步可以不需要创建样式。

第二步：调用函数 lv_spangroup_create 创建 span 部件组，这个组用来添加 span 部件描述符的，如下源码所示：

```
lv_obj_t * spans = lv_spangroup_create(lv_scr_act());  /* 创建 spans 组 */
lv_obj_set_width(spans, 400);                          /* 设置该部件的宽度 */
lv_obj_set_height(spans, 300);                         /* 创建该部件的高度 */
lv_obj_center(spans);                                  /* 中间对齐 */
/* 把 spans 部件添加到普通样式当中 */
lv_obj_add_style(spans, &style, 0);
/* 设置该部件的文本对齐模式 */
lv_spangroup_set_align(spans, LV_TEXT_ALIGN_LEFT);
```

```
/* 设置该部件的文本缩进 */
lv_spangroup_set_indent(spans, 20);
/* 设置该部件的模式 */
lv_spangroup_set_mode(spans, LV_SPAN_MODE_BREAK);
```

从上述源码可知：我们调用函数 lv_obj_add_style 添加到普通样式当中，这样我们就可以调用样式属性函数修饰 span 部件上的文本了。至于文本缩进以及模式等设置，我们放到其他小节来讲解。

第三步：调用函数 lv_spangroup_new_span 创建 span 部件的描述符并添加到 span 组中，如下源码所示：

```
lv_span_t * span = lv_spangroup_new_span(spans);          /* 创建 spans 描述符 */
lv_span_set_text(span, "China is a beautiful country.");  /* 设置要显示的文本 */
/* 设置该文本的颜色 */
lv_style_set_text_color(&span->style, lv_palette_main(LV_PALETTE_RED));
/* 设置该文本的字体 */
lv_style_set_text_font(&span->style,  &lv_font_montserrat_24);
```

上述函数主要创建 span 描述符并添加到 span 组中，我们调用函数 lv_span_set_text 设置要显示的文本，最后调用样式属性函数设置文本的样式。注意：上述的源码可以在模拟器以及开发板上运行，该运行结果图下所示：

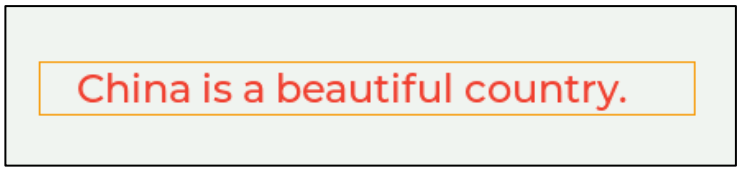


图 34.2.1.1 设置文本样式

注意：我们可以根据上面的第三步类似的操作修饰其他的文本。

34.2.2 检索 span 组的子对象

从上一个小节可知：span 组以不同于普通对象的方式存储它们的子类，因此用于获取子类的普通函数将不起作用，我们必须调用 span 部件相关的函数才可以获取 span 组的子类，下面我们来讲解跨度部件获取子类的函数。

lv_spangroup_get_child (spangroup, id)函数，该函数具有两个形参，它们分别指向 span 组指针以及指向索引 id，如果 id 为 0，则获取 span 组的第一个子类，如果 id 为-1，则获取 span 组的最后一个子类。当然我们可以调用 lv_spangroup_get_child_cnt 函数获取 span 组的子类个数。

34.2.3 Span 部件文本对齐

Span 部件与 label 对象一样，它们可以设置文本对齐模式，下面我们来列举一下文本对齐模式有哪些，如表 34.2.3.1 所示：

对齐模式	描述
LV_TEXT_ALIGN_LEFT	文本左对齐
LV_TEXT_ALIGN_CENTER	文本中间对齐
LV_TEXT_ALIGN_RIGHT	文本右对齐
LV_TEXT_ALIGN_AUTO	文本自动对齐

表 34.2.3.1 Span 部件文本对齐模式

这些对齐模式是由 lv_spangroup_set_align 函数设置的，前面我们也讲解过，这些我们无需重复讲解。

34.2.4 Span 组的模式选择

span 组的模式具有三种模式，这些模式如下表 34.2.4.1 所示：

文本模式	描述
LV_SPAN_MODE_FIXED	修复对象大小
LV_SPAN_MODE_EXPAND	将对象大小扩展为文本大小，但保持在一行上
LV_SPAN_MODE_BREAK	保持宽度，断开太长的行和自动扩大高度

表 34.2.3.1 Span 部件文本对齐模式

创建 span 组时，该组一般是 LV_SPAN_MODE_EXPAND 文本模式，该模式是将对象大小扩展为文本大小，简单来讲：所有文本绘制在一行上。这些文本模式可以调用函数 lv_spangroup_set_mode 修改。

注意：我们可以在 LV_SPAN_MODE_BREAK 模式下显示的最大行数，这个最大的行数由 lv_spangroup_set_lines 函数设置，如果该函数的第二个形参设置为负值，则表示无限制行数；如果该形参设置为 10，则表示最大行数为 10 行。

34.2.5 文本溢出

文本溢出是指文本的大小超过 span 组的区域范围，对于这种情况 LVGL 提供两种溢出模式来解决这个问题，这些模式如表 34.2.5.1 所示：

溢出模式	描述
LV_SPAN_OVERFLOW_CLIP	在区域限制处截断文本
LV_SPAN_OVERFLOW_ELLIPSIS	当文本溢出该区域时显示省略号(…)

表 34.2.5.1 溢出模式的描述

下面笔者使用一个示意图来讲解这两种溢出模式的作用，如图 34.2.5.1 所示：

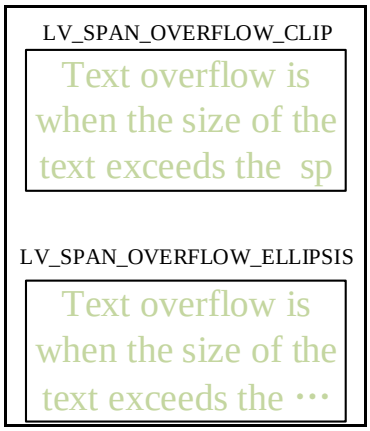


图 34.2.5.1 溢出模式的作用

从上图可知：溢出模式在文本大小超出 span 组的区域范围时才能起作用。这些溢出模式是由 lv_spangroup_set_overflow 函数设置的。

34.2.6 文本首行缩进

首行缩进是将段落的第一行从左向右缩进一定的距离，首行外的各行都保持不变。在 span 组的文本想实现缩进操作则必须调用 lv_spangroup_set_indent 函数设置，这个函数的第二个形参表示缩进的像素大小，当然我们也可以使用百分比形式来设置文本缩进。

34.3 Span 部件 API 函数

LVGL 提供给用户的 Span 部件相关函数，如表 33.3.1 所示：

函数	描述
Span 部件创建与删除函数	
lv_spangroup_create()	创建 span 组
lv_spangroup_new_span()	创建 span 描述符
lv_spangroup_del_span()	删除 span 描述符
Span 部件设置函数	
lv_span_set_text()	设置文本
lv_span_set_text_static()	静态设置文本
lv_spangroup_set_align()	设置文本对齐模式
lv_spangroup_set_overflow()	设置溢出模式
lv_spangroup_set_indent()	设置文本首项缩进
lv_spangroup_set_mode()	设置 span 组模式
lv_spangroup_set_lines()	设置显示行数
Span 部件获取函数	
lv_spangroup_get_child()	获取子类
lv_spangroup_get_child_cnt()	获取子类数量
lv_spangroup_get_align()	获取对齐模式
lv_spangroup_get_overflow()	获取溢出模式
lv_spangroup_get_indent()	获取首项缩进
lv_spangroup_get_mode()	获取 span 组模式
lv_spangroup_get_lines()	获取显示行数
lv_spangroup_get_max_line_h()	获取行的最大高度
lv_spangroup_get_expand_width()	获取扩大宽度
l lv_spangroup_get_expand_height()	获取扩大高度
lv_spangroup_refr_mode()	更新模式

表 33.3.1 span 部件相关函数描述

前面笔者已经讲解了常用函数作用以及使用方法，剩下的函数大家可以看一下官方的文档描述，这里我们不一一的讲解了。

34.4 Span 控件实验

34.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 span 组并在这个组添加多个 span 描述符，这些描述符可以根据设置样式来修饰文本。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 34 lv_span(跨度)》。

34.4.2 软件设计

34.4.2.1 程序流程图

本实验的程序流程图，如下图 34.4.2.1.1 所示：

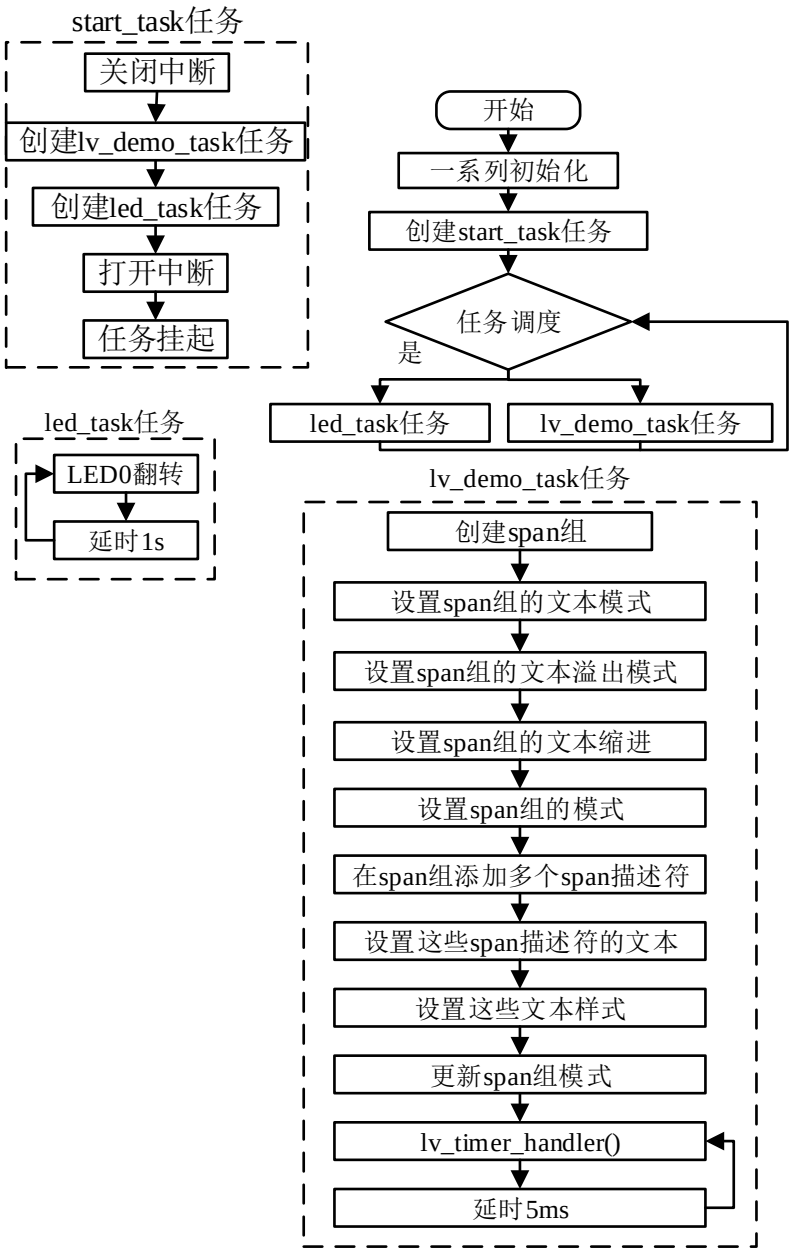


图 34.4.2.1.1 span 部件实验流程图

34.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_span();
}
```

此函数实际调用 lv_example_span 函数编写 LVGL 代码，该函数源码如下所示：

```
    /**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_span(void)
{
    /* 定义并创建 spangroup */
    lv_obj_t* spangroup = lv_spangroup_create(lv_scr_act());
    /* 设置 spangroup 宽度 */
    lv_obj_set_width(spangroup, 200);
    /* 设置 spangroup 高度 */
    lv_obj_set_height(spangroup, 200);
    /* 设置 spangroup 位置 */
    lv_obj_center(spangroup);
    /* 设置 spangroup 边框宽度 */
    lv_obj_set_style_border_width(spangroup, 1, LV_PART_MAIN);
    /* 设置 spangroup 边框颜色 */
    lv_obj_set_style_border_color(spangroup,
                                  lv_palette_main(LV_PALETTE_ORANGE),
                                  LV_PART_MAIN);
    /* 设置 spangroup 文本对齐方式 */
    lv_spangroup_set_align(spangroup, LV_TEXT_ALIGN_LEFT);
    /* 设置 spangroup 溢出 */
    lv_spangroup_set_overflow(spangroup, LV_SPAN_OVERFLOW_CLIP);
    /* 设置 spangroup 缩进 */
    lv_spangroup_set_indent(spangroup, 20);
    /* 设置 spangroup 模式 */
    lv_spangroup_set_mode(spangroup, LV_SPAN_MODE_BREAK);
```

```
    /* 创建并新建 span */
    lv_span_t* span = lv_spangroup_new_span(spangroup);
    /* 设置 span 文本 */
    lv_span_set_text(span, "This is span 1.");
    /* 设置 span 文本颜色 */
    lv_style_set_text_color(&span->style, lv_palette_main(LV_PALETTE_RED));
    /* 设置 span 文本格式 */
    lv_style_set_text_decor(&span->style, LV_TEXT_DECOR_STRIKETHROUGH
                            | LV_TEXT_DECOR_UNDERLINE);
    /* 设置 span 文本透明度 */
    lv_style_set_text_opa(&span->style, LV_OPA_30);

    /* 新建 span */
    span = lv_spangroup_new_span(spangroup);
    /* 设置 span 文本 */
    lv_span_set_text_static(span, "This is span 2.");
    /* 设置 span 文本字体 */
    lv_style_set_text_font(&span->style, &lv_font_montserrat_16);
    /* 设置 span 文本颜色 */
    lv_style_set_text_color(&span->style, lv_palette_main(LV_PALETTE_GREEN));

    /* 新建 span */
    span = lv_spangroup_new_span(spangroup);
    /* 设置 span 文本 */
    lv_span_set_text_static(span, "This is span 3.");
    /* 设置 span 文本颜色 */
    lv_style_set_text_color(&span->style, lv_palette_main(LV_PALETTE_BLUE));

    /* 新建 span */
    span = lv_spangroup_new_span(spangroup);
    /* 设置 span 文本 */
    lv_span_set_text_static(span, "This is span 4.");
    /* 设置 span 文本颜色 */
    lv_style_set_text_color(&span->style, lv_palette_main(LV_PALETTE_GREEN));
    /* 设置 span 文本字体 */
    lv_style_set_text_font(&span->style, &lv_font_montserrat_10);
    /* 设置 span 文本格式 */
    lv_style_set_text_decor(&span->style, LV_TEXT_DECOR_UNDERLINE);

    /* 新建 span */
    span = lv_spangroup_new_span(spangroup);
    /* 设置 span 文本 */
    lv_span_set_text(span, "This is span 5.");
```

```
    /* 更新 spangroup 模式 */
    lv_spangroup_refr_mode(spangroup);
}
```

此函数非常简单，大家可以看 34.2.1 小节的内容，这些内容和本实验的内容类似，这里笔者无需重复讲解了。

34.4.3 下载验证

把工程编译并下载到开发板中， 如图 34.4.3.1 所示：

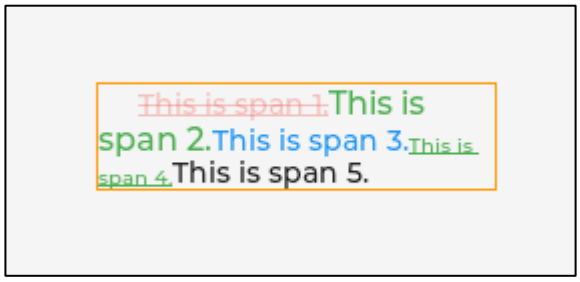


图 34.4.3.1 span 实验主界面

第三十五章 微调器部件(lv_spinbox)

lv_spinbox 部件本质上就是一个 lv_ta 文本域部件，只不过在其部件上做了某种延伸，主要是用于递增或者递减地调节某数值，它是可以按照某个 step 步值来精确调节的。在实际项目中，如果使用该部件微调某个数值，则使用该部件调整数值。

本章分为如下几部分：

35.1 微调器部件的组成

35.2 微调器部件的相关知识

35.3 微调器部件 API 函数

35.4 微调器控件实验

35.1 微调器部件的组成

因为 lv_spinbox 部件本质上就是一个 lv_ta 文本域部件，所以该部件与 lv_ta 文本域部件组成类似。

35.2 微调器部件的相关知识

35.2.1 微调器部件属性设置

1. 数值格式设置

默认情况下，Spinbox 部件创建时，它只能显示位数为 5，而没有小数点位数，如图 35.2.1.1 所示：

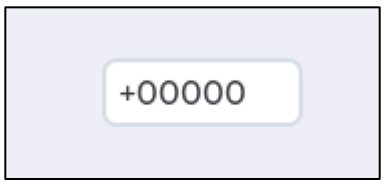


图 35.2.1.1 默认的 spinbox 部件

由于 Spinbox 部件是一个数值微调器，因此我们可设置微调器小数点的值，我们可以使用以下函数设置：

```
/* 设置数字的格式 */
lv_spinbox_set_digit_format(spinbox, digit_count, separator_position)
```

上述函数中 digit_count 形参表示位数，而 separator_position 形参表示小数点前的位数。

老规矩，下面我们还是使用一个实例来讲解本小节的内容，如下源码所示：

```
void lv_mainstart(void)
{
    lv_obj_t *spinbox = lv_spinbox_create(lv_scr_act()); /* 创建 spinbox 控件 */
    lv_spinbox_set_digit_format(spinbox, 5, 2);          /* 设置数字格式 */
    lv_obj_set_width(spinbox, 100);
    lv_obj_align_to(spinbox, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

从上述源码可知：创建一个 Spinbox 部件，然后调用 lv_spinbox_set_digit_format 设置数字

的格式。如图 35.2.1.2 所示：

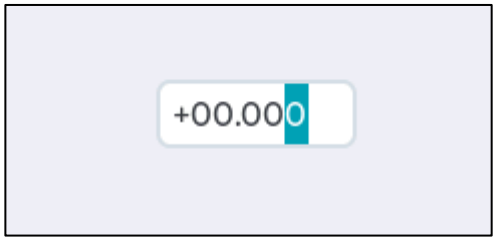


图 35.2.1.2 设置数字格式

从上图可知：我们调用函数 lv_spinbox_set_digit_format 设置微调器部件的数字位为 5 位，由于我们设置小数点前的位数为 2，所以小数点位数为 3 位。

2. 数值和范围

微调器的当前值和数值范围可调用以下函数设置：

```
lv_spinbox_set_range(spinbox, min, max)     /* 设置了 spinbox 的范围 */
lv_spinbox_set_value(spinbox, num)          /* 手动设置 spinbox 的值 */
lv_spinbox_increment(spinbox)               /* 增加 spinbox 的值 */
lv_spinbox_decrement(spinbox)               /* 减少 spinbox 的值 */
lv_spinbox_set_step(spinbox, step)          /* 将数量设置为自增递减 */
lv_spinbox_set_pos(spinbox, pos)            /* 设置光标位置 */
```

lv_spinbox_set_range 函数是设置 Spinbox 部件的数值范围。简单来说：数值不能超过这个范围并只能在这个范围活动。

lv_spinbox_set_value 函数是设置 Spinbox 部件的当前数值，如果我们设置当前值为 20，那么 spinbox 部件只能显示 00.020，而并不是 20.000 的数值。如图 35.2.1.3 所示：

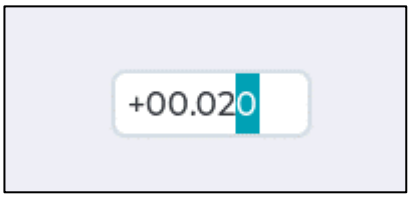


图 35.2.1.3 设置当前值

lv_spinbox_set_step 函数是设置自增递减的数量，默认情况下，Spinbox 控件自增递减的数量是为 1 的，如果我们设置为 5，那么每一次按下增按键或者减按键，就是以 5 的数值自增或者自减。剩下的函数非常简单，这里笔者无需讲解了。

35.2.2 微调器部件翻转模式

Spinbox 部件翻转模式是防止数值自增或者自减超过最小值或最大值，所以它提供了翻转模式，该模式需要调用 lv_spinbox_set_rollover(spinbox, true/false)函数设置。

35.3 微调器部件 API 函数

LVGL 提供给用户的 spinbox 部件相关函数，如表 35.3.1 所示：

函数	描述
lv_spinbox_create()	创建 spinbox 部件
lv_spinbox_set_value()	设置 spinbox 部件的数值
lv_spinbox_set_rollover()	设置翻转
lv_spinbox_set_digit_format()	设置数值格式

lv_spinbox_set_step()	设置步长
lv_spinbox_set_range()	设置范围
lv_spinbox_set_pos()	设置光标位置
lv_spinbox_set_digit_step_direction()	根据编码器数值设置数字步长方向
lv_spinbox_get_rollover()	获取翻转状态
lv_spinbox_get_value()	获取当前值
lv_spinbox_get_step()	获取步长值
lv_spinbox_step_next()	将步长除以 10，选择下一个较低的数字
lv_spinbox_step_prev()	将步长乘以 10，选择下一个更高的数字
lv_spinbox_increment()	数值自增
lv_spinbox_decrement()	数额之自减

表 35.3.1 spinbox 部件向函数描述

接下来我们来讲解 LVGL lv_spinbox 部件常用的函数有哪些：

1. lv_spinbox_create 函数

该函数的作用是创建一个 spinbox 对象，其函数原型如下所示：

```
lv_obj_t *lv_spinbox_create(lv_obj_t *parent)
```

该函数的形参描述如表 35.3.2 所示：

参数	描述
parent	父对象指针

表 35.3.2 函数 lv_spinbox_create()形参描述

返回值：指向创建的 spinbox 的指针。

2. lv_spinbox_set_value 函数

该函数的作用是设置 spinbbox 当前值，其函数原型如下所示：

```
void lv_spinbox_set_value(lv_obj_t *obj, int32_t i)
```

该函数的形参描述如表 35.3.3 所示：

参数	描述
obj	对象指针
i	设置当前值

表 35.3.3 函数 lv_spinbox_set_value()形参描述

返回值：无。

3. lv_spinbox_set_rollover 函数

该函数的作用是设置翻转模式，其函数原型如下所示：

```
void lv_spinbox_set_rollover(lv_obj_t *obj, bool b)
```

该函数的形参描述如表 35.3.4 所示：

参数	描述
obj	对象指针
b	True 或 false 启用或禁用(默认)

表 35.3.4 函数 lv_spinbox_set_rollover()形参描述

返回值：无。

4. lv_spinbox_set_digit_format 函数

该函数的作用是创建一个 spinbox 对象，其函数原型如下所示：

```
void lv_spinbox_set_digit_format(  lv_obj_t *obj,
                                   uint8_t digit_count,
                                   uint8_t separator_position)
```

该函数的形参描述如表 35.3.5 所示：

参数	描述
obj	对象指针
digit_count	位数，不包括小数点分隔符和符号
separator_position	小数点前的位数。如果为 0，则不显示小数点

表 35.3.5 函数 lv_spinbox_set_digit_format()形参描述

返回值：无。

5. lv_spinbox_set_step 函数

该函数的作用是设置步长，其函数原型如下所示：

```
void lv_spinbox_set_step(lv_obj_t *obj, uint32_t step)
```

该函数的形参描述如表 35.3.6 所示：

参数	描述
obj	对象指针
step	递增/递减的步长

表 35.3.6 函数 lv_spinbox_set_step()形参描述

返回值：无。

6. lv_spinbox_set_range 函数

该函数的作用是设置 spinbox 数值范围，其函数原型如下所示：

```
void lv_spinbox_set_range(lv_obj_t *obj, int32_t range_min, int32_t range_max)
```

该函数的形参描述如表 35.3.7 所示：

参数	描述
obj	对象指针
range_min	最小值
range_max	最大值

表 35.3.7 函数 lv_spinbox_set_range()形参描述

返回值：无。

7. lv_spinbox_set_pos 函数

该函数的作用是设置光标的位置，其函数原型如下所示：

```
void lv_spinbox_set_pos(lv_obj_t *obj, uint8_t pos)
```

该函数的形参描述如表 35.3.8 所示：

参数	描述
obj	父对象指针
pos	设置位置

表 35.3.8 函数 lv_spinbox_set_pos()形参描述

返回值：无。

8. lv_spinbox_increment 函数

该函数的作用是 spinbox 对象数值自增，其函数原型如下所示：

```
void lv_spinbox_increment(lv_obj_t *obj)
```

该函数的形参描述如表 35.3.9 所示：

参数	描述
obj	父对象指针

表 35.3.9 函数 lv_spinbox_increment()形参描述

返回值：无。

9. lv_spinbox_decrement 函数

该函数的作用是 spinbox 对象数值自减，其函数原型如下所示：

```
void lv_spinbox_decrement(lv_obj_t *obj)
```

该函数的形参描述如表 35.3.10 所示：

参数	描述
obj	父对象指针

表 35.3.10 函数 lv_spinbox_decrement()形参描述

返回值：无。

35.4 微调器控件实验

35.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 spinbox 以及两个 btn 部件，这两个 btn 部件可以使 spinbox 部件的指定位加 1 或者减 1 操作。。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 35 lv_spinbox(微调器)》。

35.4.2 软件设计

35.4.2.1 程序流程图

本实验的程序流程图，如下图 35.4.2.1.1 所示：

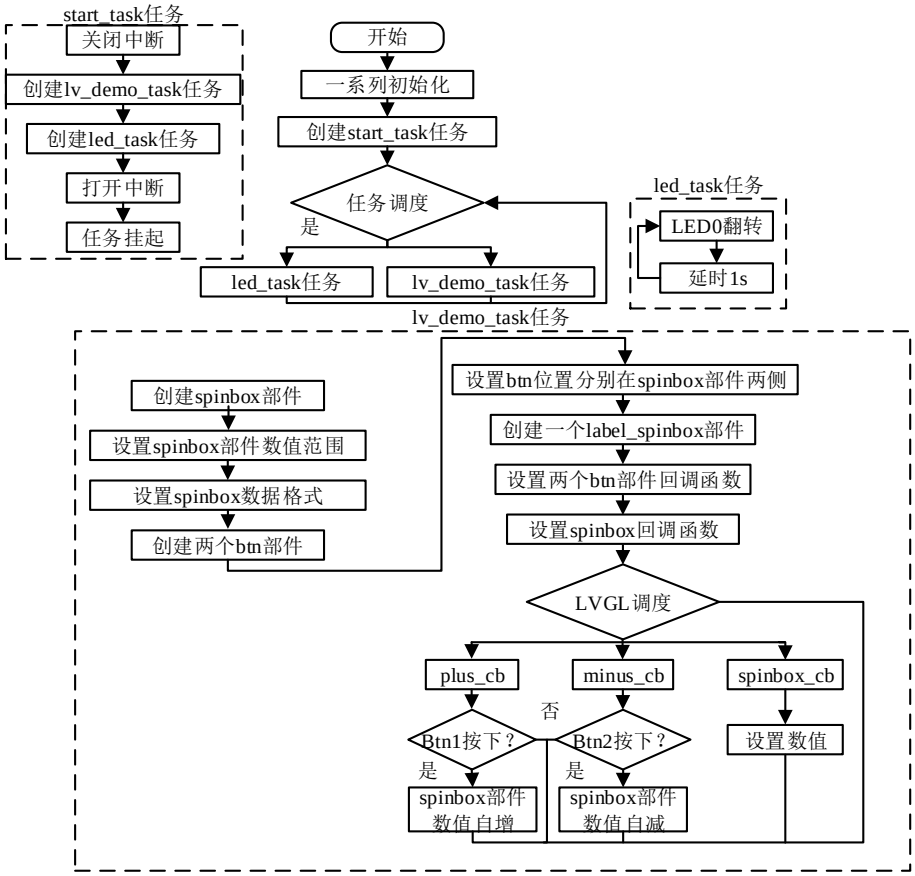


图 35.4.2.1.1 lv_ spinbox (加载器)实验流程图

35.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_spinbox(void)
{
    /* 定义并创建 spinbox */
    lv_obj_t* spinbox = lv_spinbox_create(lv_scr_act());
    /* 设置 spinbox 范围 */
    lv_spinbox_set_range(spinbox, -10000, 10000);
    /* 设置 spinbox 格式 */
    lv_spinbox_set_digit_format(spinbox, 5, 3);
    /* 通过将步长乘以 10 为版本选择下一个更高的数字 */
    lv_spinbox_step_prev(spinbox);
    /* 设置 spinbox 宽度 */
    lv_obj_set_width(spinbox, 100);
```

```
    /* 设置 spinbox 位置 */
    lv_obj_center(spinbox);
    /* 手动更新参数 */
    lv_obj_update_layout(spinbox);
    /* 定义并创建按钮 */
    lv_obj_t* btn = lv_btn_create(lv_scr_act());
    /* 设置按钮的宽度 */
    lv_obj_set_width(btn, lv_obj_get_height(spinbox));
    /* 设置按钮的高度 */
    lv_obj_set_height(btn, lv_obj_get_height(spinbox));
    /* 设置按钮的位置 */
    lv_obj_align_to(btn, spinbox, LV_ALIGN_OUT_LEFT_MID, -10, 0);
    /* 设置按钮的背景 */
    lv_obj_set_style_bg_img_src(btn, LV_SYMBOL_PLUS, LV_PART_MAIN);
    /* 设置按钮事件回调 */
    lv_obj_add_event_cb(btn, event_btn_plus_cb, LV_EVENT_ALL, spinbox);
    /* 定义并创建按钮 */
    btn = lv_btn_create(lv_scr_act());
    /* 设置按钮的宽度 */
    lv_obj_set_width(btn, lv_obj_get_height(spinbox));
    /* 设置按钮的高度 */
    lv_obj_set_height(btn, lv_obj_get_height(spinbox));
    /* 设置按钮的位置 */
    lv_obj_align_to(btn, spinbox, LV_ALIGN_OUT_RIGHT_MID, 10, 0);
    /* 设置按钮的背景 */
    lv_obj_set_style_bg_img_src(btn, LV_SYMBOL_MINUS, LV_PART_MAIN);
    /* 设置按钮事件回调 */
    lv_obj_add_event_cb(btn, event_btn_minus_cb, LV_EVENT_ALL, spinbox);
    /* 定义并创建标签 */
    lv_obj_t* label_spinbox = lv_label_create(lv_scr_act());
    /* 设置标签的位置 */
    lv_obj_align_to(label_spinbox, spinbox, LV_ALIGN_OUT_TOP_MID, 0, -10);
    /* 设置标签文本对齐方式 */
    lv_obj_set_style_text_align(label_spinbox, LV_TEXT_ALIGN_CENTER,
                                LV_PART_MAIN);
    /* 设置标签文本 */
    lv_label_set_text(label_spinbox, "0.0");

    /* spinbox 添加事件回调 */
    lv_obj_add_event_cb(spinbox, event_spinbox_cb, LV_EVENT_VALUE_CHANGED,
                        label_spinbox);
}
```

```
/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_spinbox();
}
```

从上述源码可知：首先创建 spinbox、btn 和 label 部件，然后设置这些部件相关样式属性，最后设置 spinbox 和 btn 部件回调函数。这些回到函数如下所示“

```
/**
 * @brief  增加按钮事件回调
 * @param  e:事件描述
 * @return 无
 */
static void event_btn_plus_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* spinbox = (lv_obj_t*)e->user_data;

    if (code == LV_EVENT_SHORT_CLICKED || code == LV_EVENT_LONG_PRESSED_REPEAT)
    {
        /* 增加 spinbox 的值 */
        lv_spinbox_increment(spinbox);
    }
}

 /**
  * @brief  减少按钮事件回调
  * @param  e:事件描述
  * @return 无
  */
static void event_btn_minus_cb(lv_event_t* e)
{
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* spinbox = (lv_obj_t*)e->user_data;

    if (code == LV_EVENT_SHORT_CLICKED || code == LV_EVENT_LONG_PRESSED_REPEAT)
    {
        /* 减少 spinbox 的值 */
        lv_spinbox_decrement(spinbox);
    }
```

```
}
/**
 * @brief  spinbox 添加事件回调
 * @param  e:事件描述
 * @return 无
 */
static void event_spinbox_cb(lv_event_t* e)
{
    lv_obj_t* label_spinbox = (lv_obj_t*)e->user_data;
    lv_obj_t* spinbox = lv_event_get_target(e);
    int32_t spinbox_value = lv_spinbox_get_value(spinbox);
    lv_label_set_text_fmt(label_spinbox, "%s%d.%d",
        spinbox_value < 0 ? "-" : "",
        spinbox_value / 100 < 0 ? -spinbox_value / 100 : spinbox_value / 100,
        spinbox_value % 100 < 0 ? -spinbox_value % 100 : spinbox_value % 100);
}
```

从上述源码可知：event_btn_plus_cb()回调函数主要判断自增按键是否按下，如果按键按下，则 spinbox 部件数值自增。event_btn_minus_cb()回调函数主要判断自减按键是否按下，如果按键按下，则 spinbox 部件数值自减。event_spinbox_cb()函数主要刷新 label_spinbox 部件的文本。

35.4.3 下载验证

把工程编译并下载到开发板中， 如图 35.4.3.1 所示：

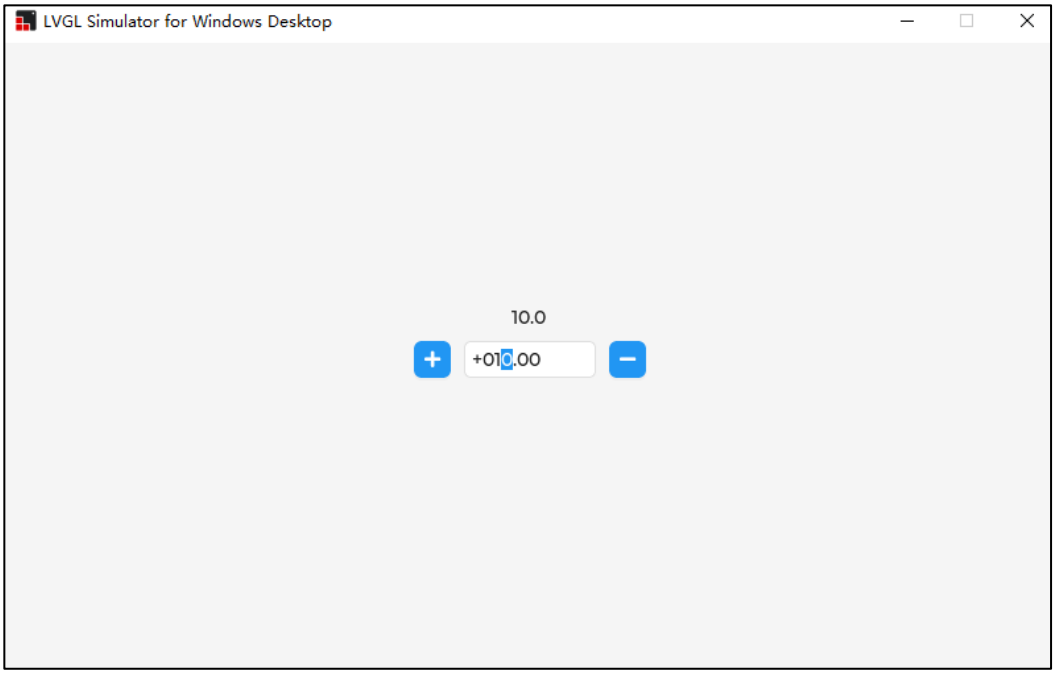


图 35.4.3.1 spinbox 部件实验示意图

第三十六章 加载器部件(lv_spinner)

加载器在各种 UI 系统中也是常见的， 例如在我们的生活中，使用手机加载一个文件时，一般有一个弧形在旋转，表示文件在加载中。LVGL 的 Spinner 部件是基于 arc 弧形部件进行构建的，在其基础上增加了与加载相关的操作接口。

本章分为如下几部分：

36.1 加载器部件的组成

36.2 加载器部件的相关知识

36.4 加载器部件 API 函数

36.5 加载器控件实验

36.1 加载器部件的组成

Spinner 部件是由封闭的弧形背景以及指示器组成，如图 36.1.1 所示：

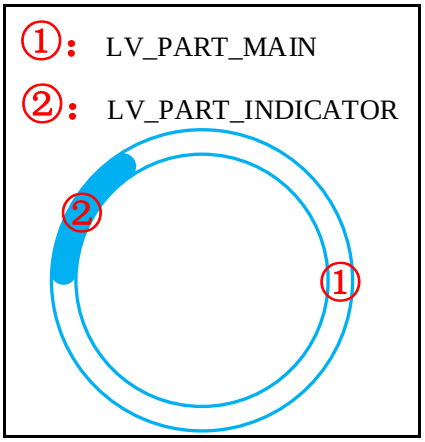


图 36.1.1 Spinner 部件组成部分

该部件的样式属性设置，请参考第六章的内容，这里我们无需重复讲解。

36.2 加载器部件的相关知识

1. 创建 Spinner 部件

创建 Spinner 部件可调用 lv_spinner_create(parent, spin_time, arc_length)函数，该函数具有三个形参，parent 形参表示部件的父类，spin_time 形参表示自旋的时间，而 arc_length 形参表示指示器的长度。

注意：自旋的时间是以毫秒为单位。如果设置 1000 毫秒，则指示器以 1 秒钟旋转一圈。

36.4 加载器部件 API 函数

LVGL 的 Spinner 部件只提供用户一个函数，如下所示：

lv_spinner_create 函数

该函数的作用是创建一个 spinner 对象，其函数原型如下所示：

```
lv_obj_t *lv_spinner_create(lv_obj_t *parent)
```

该函数的形参描述如表 36.3.2 所示：

参数	描述
parent	父对象指针

表 36.3.2 函数 lv_spinner_create()形参描述

返回值：指向创建的 spinner 的指针。

36.5 加载器控件实验

36.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 Spinner 部件，该部件以 1000ms 为周期旋转。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 36 lv_spinner(加载器)》。

36.4.2 软件设计

36.4.2.1 程序流程图

本实验的程序流程图，如下图 36.4.2.1.1 所示：

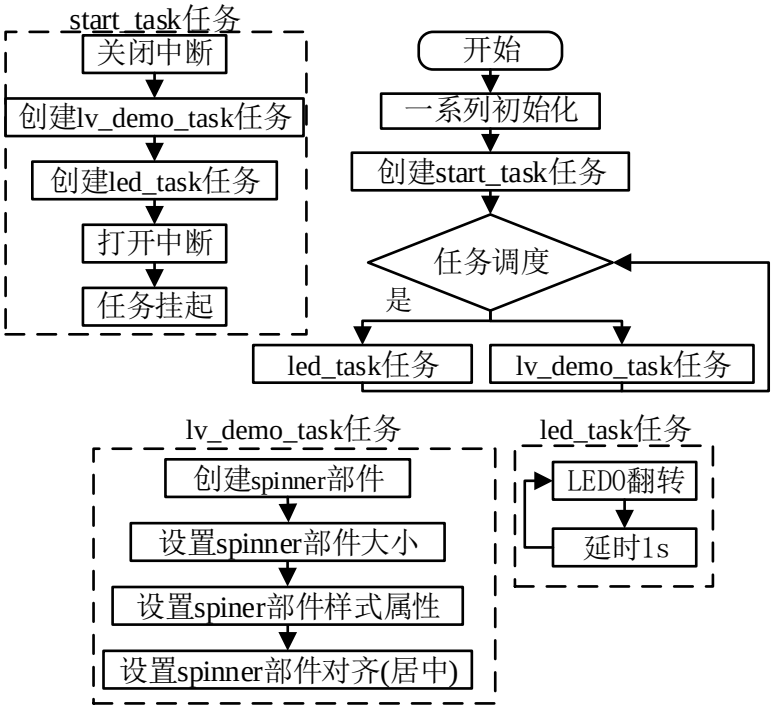


图 36.4.2.1.1 lv_spinner(加载器)实验流程图

36.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_spinner(void)
{
    /* 定义并创建 spinner */
    lv_obj_t* spinner = lv_spinner_create(lv_scr_act(), 1000, 60);
    /* 设置 spinner 宽度 */
    lv_obj_set_width(spinner, LV_MIN(scr_act_width(), scr_act_height()) >> 1);
    /* 设置 spinner 高度 */
    lv_obj_set_height(spinner, LV_MIN(scr_act_width(), scr_act_height()) >> 1);
    /* 设置 spinner 位置 */
    lv_obj_center(spinner);
    /* 设置背景宽度 */
    lv_obj_set_style_arc_width(spinner, scr_act_height() * 0.05, LV_PART_MAIN);
    /* 设置条宽度 */
    lv_obj_set_style_arc_width(spinner, scr_act_height() * 0.05,
                               LV_PART_INDICATOR);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_spinner();
}
```

从上述源码可知：首先我们创建 Spinner 部件并设置部件的宽度和旋转时间，然后程序设置 Spinner 部件的样式属性，最后根据组成部分设置指示器和背景的样式。

36.4.3 下载验证

把工程编译并下载到开发板中， 如图 36.4.3.1 所示：

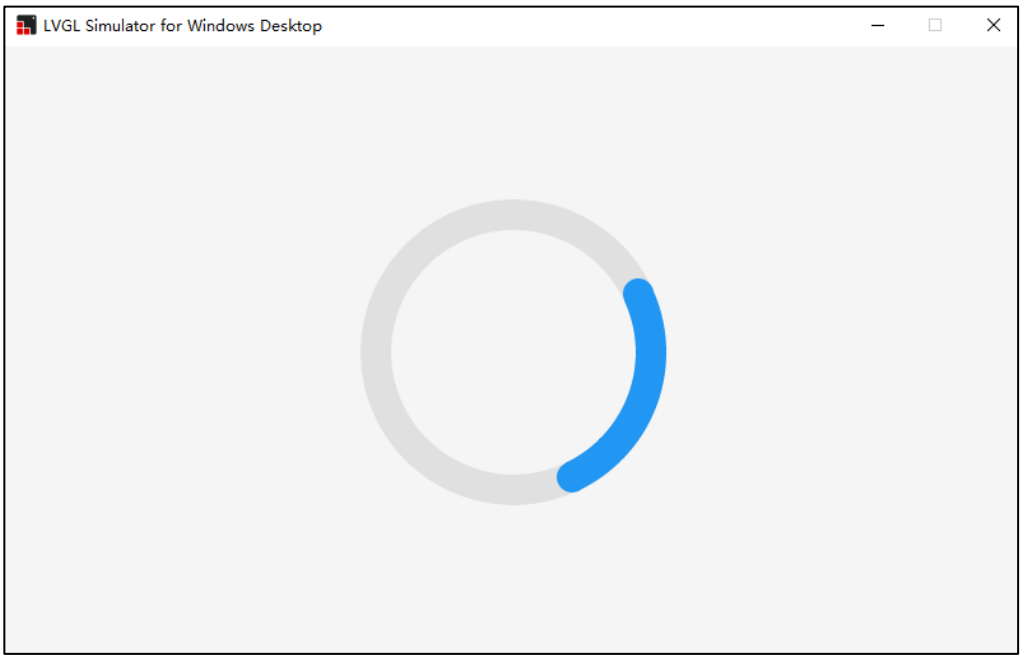


图 36.4.3.1 Spinner 部件实验示意图

第三十七章 选项卡视图部件(lv_tabview)

lv_tabview 部件可实现多页面切换，该切换方式可通过点击选项卡 tab 按键。

本章分为如下几部分：

37.1 选项卡视图部件的组成

37.2 选项卡视图部件的相关知识

37.3 选项卡视图部件的 API 函数

37.4 选项卡视图部件的实验

37.1 选项卡视图部件的组成

Tab 视图对象可以用来组织 Tab 中的内容，它是由其他小部件构建而成的，这些小部件如下所示：

① 主要的容器、标签的容器、Tab 内容：lv_obj。

② 选项卡按钮：lv_btnmatrix。

根据上述可知：Tab 视图对象由两个部分组成，如图 37.1.1 所示：

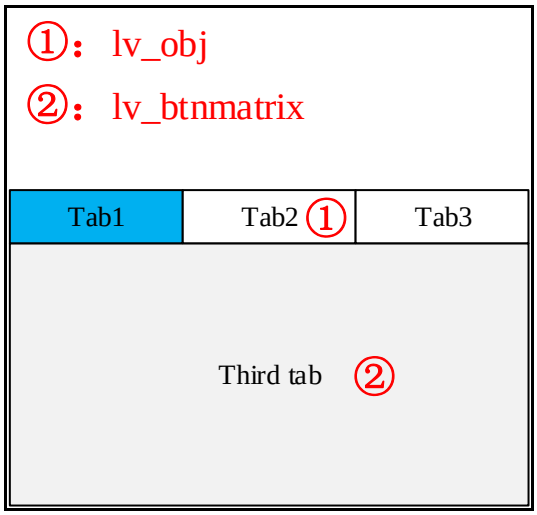


图 37.1.1 选项卡视图对象组成部分

从上图可知：在 Tab 视图中没有特殊的部分，但是 lv_obj 和 lv_btnnmatrix 小部件用于创建 Tab 视图的。

37.2 选项卡视图部件的相关知识

37.2.1 创建 Tab 视图对象

Tab 视图对象可调用 lv_tabview_create(parent, tab_pos, tab_size)函数创建，该函数具有三个形参，parent 形参表示 Tab 视图对象的父类，tab_pos 形参表示选项卡的方向，而 tab_size 形参代表选项卡的大小。如图 37.2.1.1 所示：

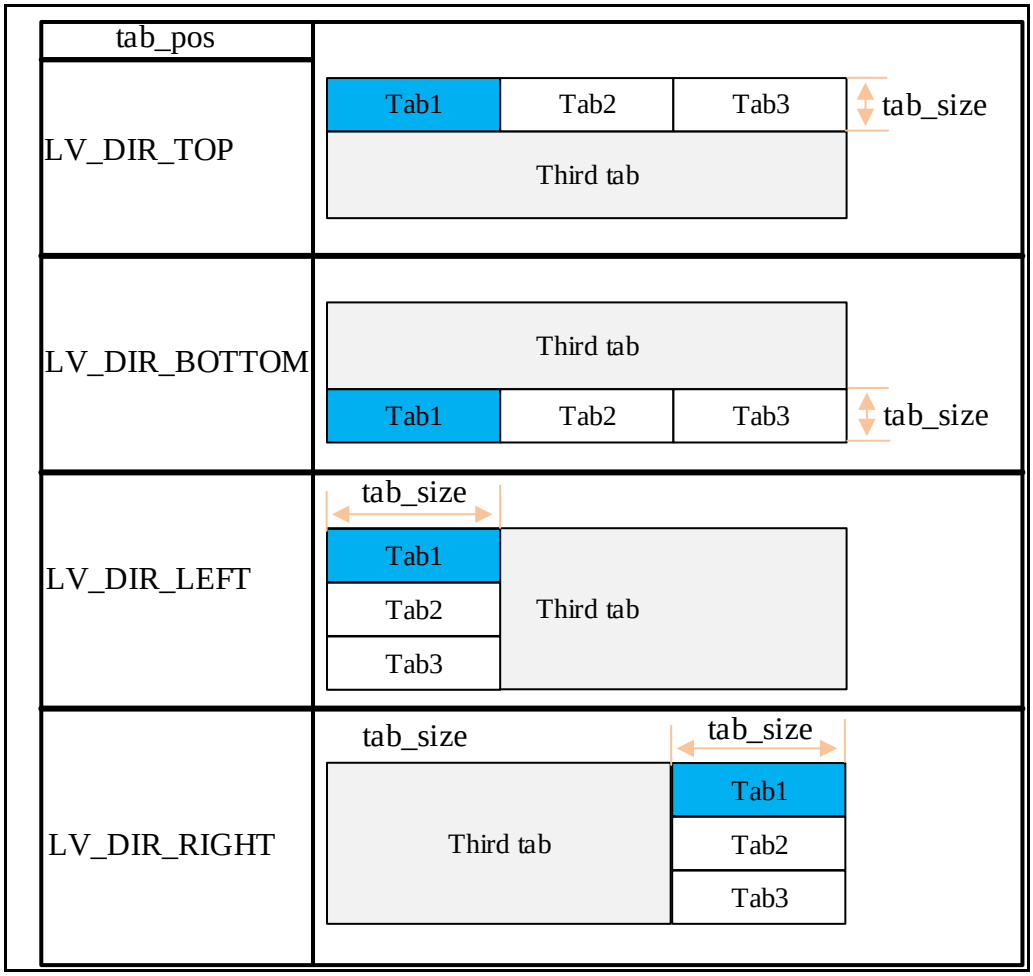


图 37.2.1.1 创建 Tab 视图对象

从图 37.2.1.1 可知：tab_pos 形参主要定义选项卡 label 的方向，而 tab_size 形参是根据 tab_pos 参数而定。

注意：Tabview 部件创建时是没有选项卡的。

37.2.2 添加选项卡

Tabview 控件默认情况下是没有选项卡的，所以用户必须调用以下函数添加选项卡：

```
lv_tabview_add_tab(tabview, "tab_name")/* 添加新标签 */
```

该函数将返回一个指向 lv_obj 对象指针，我们可根据该对象添加选项卡的内容，下面我们使用一个实例来讲解本小节的内容，如下源码所示：

```
void lv_mainstart(void)
{
    /* 创建一个 Tab 视图对象 */
    lv_obj_t* tabview;
    tabview = lv_tabview_create(lv_scr_act());
    lv_obj_t* tab1 = lv_tabview_add_tab(tabview, "Tab 1");/* 添加选项卡 1 */
    lv_obj_t* tab2 = lv_tabview_add_tab(tabview, "Tab 2");/* 添加选项卡 2 */
}
```

上述源码可知：首先我们创建一个 Tab 视图部件，然后程序调用 lv_tabview_add_tab 函数

添加两个选项卡，添加完毕之后就可以在显示屏上显示了，如图 37.2.2.1 所示：

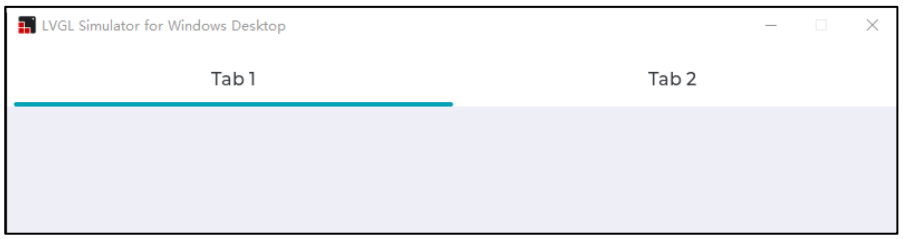


图 37.2.2.1 添加选项卡

37.2.3 切换选项卡

根据上图 37.2.2.1 中，怎么切换到 Tab2 的选项卡呢？，LVGL 提供了三种方式切换选项卡，如以下所示：

① 触摸点击 Tab2。

② 往左滑动。

③ 手动切换(lv_tabview_set_act(tabview, id, LV_ANIM_ON/OFF))。

注意：lv_tabview_set_tab_act 函数的 id 形参是以 0 开始，也就是说：0 就是第一个选项卡，1 为第二个选项卡，由此类推。

37.2.4 获取部件

前面我们也讲到，Tab 视图部件是由 lv_obj 和 lv_btnnmatrix 组成，如果修改 Tab 视图样式，则必须根据 lv_obj 以及 lv_btnnmatrix 对象来设置样式属性。获取 Tab 视图部件的组成部分可调用 lv_tabview_get_content(tabview)和 lv_tabview_get_tab_btns(tabview)函数，它们分别获取选项卡的容器和 Tab 按钮对象。

37.3 选项卡视图部件的 API 函数

LVGL 提供给用户的 Tabview 部件相关函数，如表 37.3.1 所示：

函数	描述
lv_tabview_create()	创建 Tabview 部件
lv_tabview_add_tab()	添加 Tab 选项卡
lv_tabview_get_content()	获取选项卡容器对象
lv_tabview_get_tab_btns()	获取 Tab 按键对象
lv_tabview_set_act()	设置显示的选项卡
lv_tabview_get_tab_act()	获取当前的选项卡

表 37.3.1 tileview 部件相关函数

接下来我们来讲解 LVGL 的 lv_tabview 部件常用的函数有哪些：

1. lv_tabview_create 函数

该函数的作用是创建一个 tileview 对象，其函数原型如下所示：

```
lv_obj_t *lv_tabview_create(    lv_obj_t *parent,
                                lv_dir_t tab_pos,
                                lv_coord_t tab_size)
```

该函数的形参描述如表 37.3.2 所示：

参数	描述
parent	父对象指针

tab_pos	选项卡方向
tab_size	选项卡大小

表 37.3.2 函数 lv_tabview_create()形参描述

返回值：指向创建的 tabview 对象指针。

2. lv_tabview_add_tab 函数

该函数的作用是添加一个选项卡对象，其函数原型如下所示：

```
lv_obj_t *lv_tabview_add_tab(lv_obj_t *tv, const char *name)
```

该函数的形参描述如表 37.3.3 所示：

参数	描述
tv	tabview 对象指针
name	选项卡名称

表 37.3.3 函数 lv_tabview_add_tab()形参描述

返回值：指向选项卡容器对象的指针。

3. lv_tabview_get_content 函数

该函数的作用是获取选项卡容器，其函数原型如下所示：

```
lv_obj_t *lv_tabview_get_content(lv_obj_t *tv)
```

该函数的形参描述如表 37.3.4 所示：

参数	描述
tv	tabview 对象指针

表 37.3.4 函数 lv_tabview_get_content()形参描述

返回值：获取选项卡容器对象。

4. lv_tabview_get_tab_btns 函数

该函数的作用是获取选项卡按键对象，其函数原型如下所示：

```
lv_obj_t *lv_tabview_get_tab_btns(lv_obj_t *tv)
```

该函数的形参描述如表 37.3.5 所示：

参数	描述
tv	tabview 对象指针

表 37.3.5 函数 lv_tabview_get_tab_btns()形参描述

返回值：获取选项卡按键对象。

37.4 选项卡视图部件的实验

37.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 tabview 部件，在该部件上添加三个 tab 视图，用户可以在视图中设置自己的 UI。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 37 lv_tabview(选项卡视图)》。

37.4.2 软件设计

37.4.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 37.4.2.1.1 所示：

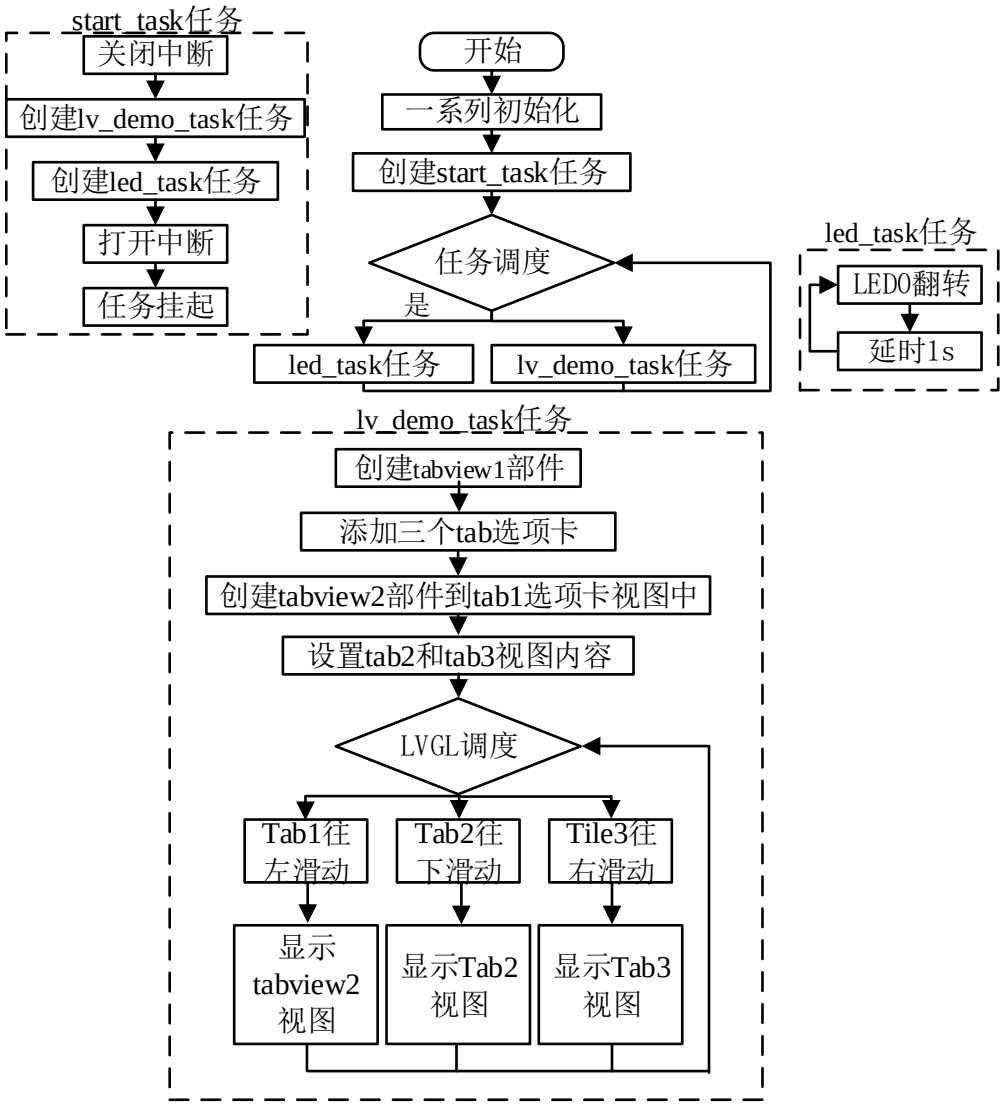


图 37.4.2.1.1 lv_tabview(选项卡视图)实验流程图

37.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart()函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_tabview(void)
{
```

```
/* 定义并创建 tabview */
lv_obj_t* tabview_top = lv_tabview_create(lv_scr_act(), LV_DIR_TOP,
                                          scr_act_height() / 5);
/* 为 tabview 添加 tab */
lv_obj_t* tab1_top = lv_tabview_add_tab(tabview_top, "Tab 1");
/* 为 tabview 添加 tab */
lv_obj_t* tab2_top = lv_tabview_add_tab(tabview_top, "Tab 2");
/* 为 tabview 添加 tab */
lv_obj_t* tab3_top = lv_tabview_add_tab(tabview_top, "Tab 3");
/* 在 tab1_top 视图创建创建 tabview */
lv_obj_t* tabview_left = lv_tabview_create(tab1_top, LV_DIR_LEFT,
                                           scr_act_width() / 5);
/* 为 tabview 添加 tab */
lv_obj_t* tab1_left = lv_tabview_add_tab(tabview_left, "Tab 1");
/* 为 tabview 添加 tab */
lv_obj_t* tab2_left = lv_tabview_add_tab(tabview_left, "Tab 2");
/* 为 tabview 添加 tab */
lv_obj_t* tab3_left = lv_tabview_add_tab(tabview_left, "Tab 3");
/* 为 tabview 添加 tab */
lv_obj_t* tab4_left = lv_tabview_add_tab(tabview_left, "Tab 4");
/* 为 tabview 添加 tab */
lv_obj_t* tab5_left = lv_tabview_add_tab(tabview_left, "Tab 5");

/* 在 tab1_left 视图中添加 label 文本*/
lv_obj_t* label = lv_label_create(tab1_left);
lv_label_set_text(label, "tabview can be nested.");

/* 在 tab2_top 视图中添加 label 文本*/
label = lv_label_create(tab2_top);
lv_label_set_text(label, "This the second tab\n\n"
    "If the content\n"
    "of a tab\n"
    "becomes too\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
    "long..\n"
```

```
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "long..\n"
        "longer\n"
        "than the\n"
        "container\n"
        "then it\n"
        "automatically\n"
        "becomes\n"
        "scrollable.\n"
        "\n"
        "\n"
        "\n"
        "Can you see it?");
    /* 在 tab3_top 视图中添加 label 文本*/
    label = lv_label_create(tab3_top);
    lv_label_set_text(label, "This is third tab.");
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_tabview();
}
```

从上述源码可知：首先调用 lv_tabview_create()创建 tabview 部件，然后添加 tab1、tab2 和 tab1 三个 tab 视图，最后设置这些 tab 视图的内容。

37.4.3 下载验证

把工程编译并下载到开发板中， 如图 37.4.3.1 所示：

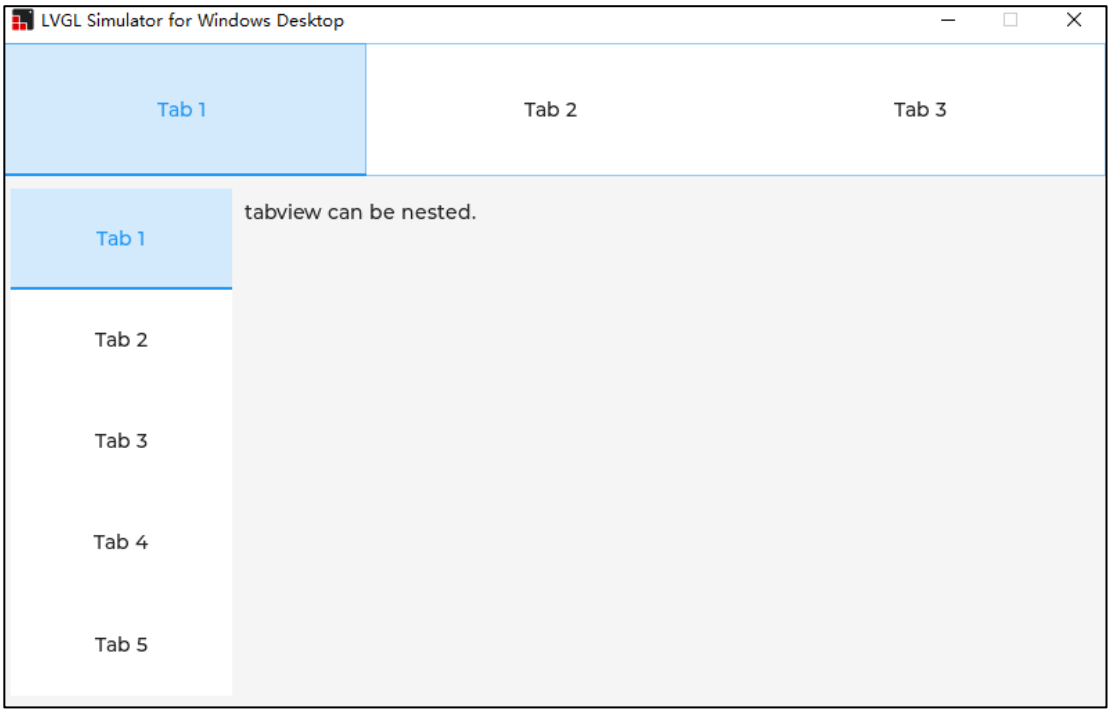


图 37.4.3.1 tabview 部件实验示意图

第三十八章 平铺视图部件(lv_tileview)

Tile 视图是一个容器对象，它的元素(称为 Tile)可以以网格形式排列。用户可以通过滑动在 Tile 之间导航。任何方向的滑动都可以单独在贴图上被禁用，简单来说：不允许从一个贴图移动到另一个贴图。

本章分为如下几部分：

38.1 平铺视图部件的组成

38.2 平铺视图部件的相关知识

38.3 平铺视图部件的 API 函数

38.4 平铺视图部件的实验

38.1 平铺视图部件的组成

Tile 视图是由 lv_obj 容器构建而成的。它类似于家里的地板一样，它们都是多个瓷砖组成在一起，而 Tile 视图与瓷砖组成方式类似，如图 38.1.1 所示：

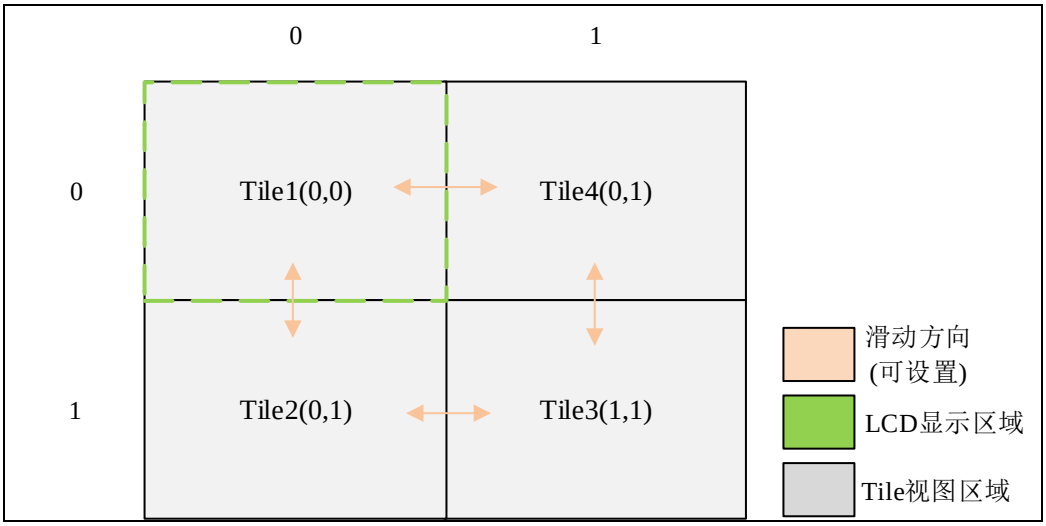


图 38.1.1 tileview 部件组成

从上图可知：每一个 Tile 相当一块瓷砖，Tile 视图都可以作为一个界面，可设置某个方向滑动切换。注意：绿色的框框为 LCD 显示区域，而金色箭头表示滑动方向。

38.2 平铺视图部件的相关知识

38.2.1 添加一个 Tile 视图

平铺视图部件是以“lv_tileview”标识符来表示，用户可调用 lv_tileview_create 函数创建平铺视图部件。创建 tileview 部件之后，必须调用 lv_tileview_add_tile 函数创建 Tile 视图，该函数第二和第三形参与上图 38.1.1 坐标相关，主要表示 Tile 视图的位置，而第四个形参代表滑动方向，该滑动方向的参数如下所示：

① LV_DIR_LEFT：往左滑动。

② LV_DIR_RIGHT：往右滑动。

③ LV_DIR_TOP：往上滑动。

④ LV_DIR_BOTTOM：往下滑动。
⑤ LV_DIR_HOR：水平滑动。
⑥ LV_DIR_VER：垂直滑动。
⑦ LV_DIR_ALL：全部方向。
注意：该函数的返回值为该 Tile 视图控制块。

38.2.2 切换 Tile 视图

切换 Tile 视图的方法有两种，这两种分别微触摸切换和硬件切换。下面我们来讲解一下这两种切换到底有何不同。

第一种：触摸滑动：LCD 显示屏是否内嵌触摸屏。

第二种：硬件切换：硬件按键或者其他输入设备切换，可调用以下函数切换：

```
/* 根据 Tile 视图切换 */
lv_obj_set_tile(tileview, tile_obj, LV_ANIM_ON/OFF)
/* 根据 Tile 视图坐标切换 */
lv_obj_set_tile_id(tileviewv, col_id, row_id, LV_ANIM_ON/OFF);
```

38.3 平铺视图部件的 API 函数

LVGL 提供给用户的 tileview 部件相关函数，如表 38.3.1 所示：

函数	描述
lv_tileview_create()	创建 Tileview 部件
lv_tileview_add_tile()	添加 Tile 视图
lv_obj_set_tile()	设置 Tile 视图
lv_obj_set_tile_id()	根据 id 设置 Tile 视图
lv_tileview_get_tile_act()	获取当前的 Tile 视图

表 38.3.1 tileview 部件相关函数

接下来我们来讲解 LVGL 的 lv_tileview 部件常用的函数有哪些：

1. lv_tileview_create 函数

该函数的作用是创建一个 tileview 对象，其函数原型如下所示：

```
lv_obj_t *lv_tileview_create(lv_obj_t *parent)
```

该函数的形参描述如表 38.3.2 所示：

参数	描述
parent	父对象指针

表 38.3.2 函数 lv_tileview_create()形参描述

返回值：指向创建的 tileview 的指针。

2. lv_tileview_add_tile 函数

该函数的作用是添加 Tile 视图对象，其函数原型如下所示：

```
lv_obj_t *lv_tileview_add_tile(lv_obj_t *tv,
                               uint8_t col_id,
                               uint8_t row_id,
                               lv_dir_t dir)
```

该函数的形参描述如表 38.3.3 所示：

参数	描述
tv	对象指针
col_id	列 ID

row_id	行 ID
dir	滑动方向

表 38.3.3 函数 lv_tileview_create()形参描述

返回值：指向 Tile 视图的指针。

3. lv_obj_set_tile 函数

该函数的作用是设置显示 Tile 视图，其函数原型如下所示：

```
void lv_obj_set_tile(  lv_obj_t *tv,
                       lv_obj_t *tile_obj,
                        lv_anim_enable_t anim_en)
```

该函数的形参描述如表 38.3.4 所示：

参数	描述
tv	对象指针
tile_obj	Tile 视图对象
anim_en	是否启用动画效果

表 38.3.4 函数 lv_obj_set_tile()形参描述

返回值：无。

4. lv_obj_set_tile_id 函数

该函数的作用是设置显示 Tile 视图，其函数原型如下所示：

```
void lv_obj_set_tile_id( lv_obj_t *tv,
                         uint32_t col_id,
                         uint32_t row_id,
                         lv_anim_enable_t anim_en)
```

该函数的形参描述如表 38.3.5 所示：

参数	描述
tv	对象指针
col_id	列 ID
row_id	行 ID
anim_en	是否启用动画效果

表 38.3.5 函数 lv_obj_set_tile_id()形参描述

返回值：无。

5. lv_tileview_get_tile_act 函数

该函数的作用是获取当前 Tile 视图，其函数原型如下所示：

```
lv_obj_t *lv_tileview_get_tile_act(lv_obj_t *obj)
```

该函数的形参描述如表 38.3.6 所示：

参数	描述
obj	对象指针

表 38.3.6 函数 lv_tileview_get_tile_act()形参描述

返回值：无。

38.4 平铺视图部件的实验

38.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 tileview 部件，在该部件上添加四个 Tille 视图，这些 Tile 视图根据用户设置的方向切换 Tile 视图，比如有 Tile 视图可以往下或者往左移动切换视图，其他 Tile 视图类似。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 38 lv_tileview(平铺视图)》。

38.4.2 软件设计

38.4.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 38.4.2.1.1 所示：

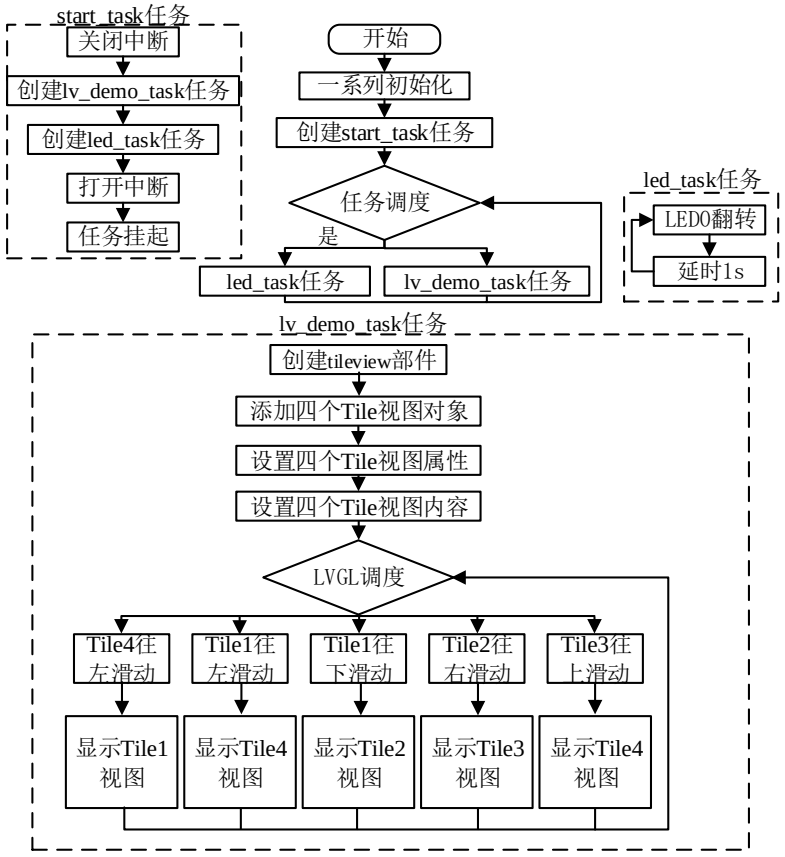


图 38.4.2.1.1 lv_tileview(平铺视图)实验流程图

38.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_tileview(void)
{
    /* 定义并创建 tileview */
    lv_obj_t* tileview = lv_tileview_create(lv_scr_act());
    /* 定义并创建一个能下滑和右滑的 tile */
    lv_obj_t* tile_bottom = lv_tileview_add_tile(tileview, 0, 0,
                                                 LV_DIR_BOTTOM | LV_DIR_RIGHT);
    lv_obj_t* label = lv_label_create(tile_bottom);
    lv_label_set_text_fmt(label, "%s    %s", LV_SYMBOL_DOWN, LV_SYMBOL_RIGHT);
    lv_obj_center(label);
    /* 定义并创建一个只能右滑的 tile */
    lv_obj_t* tile_right = lv_tileview_add_tile(tileview, 0, 1, LV_DIR_RIGHT);
    label = lv_label_create(tile_right);
    lv_label_set_text(label, LV_SYMBOL_RIGHT);
    lv_obj_center(label);
    /* 定义并创建一个只能上滑的 tile */
    lv_obj_t* tile_top = lv_tileview_add_tile(tileview, 1, 1, LV_DIR_TOP);
    label = lv_label_create(tile_top);
    lv_label_set_text(label, LV_SYMBOL_UP);
    lv_obj_center(label);
    /* 定义并创建一个只能左滑的 tile */
    lv_obj_t* tile_left = lv_tileview_add_tile(tileview, 1, 0, LV_DIR_LEFT);
    label = lv_label_create(tile_left);
    lv_label_set_text(label, LV_SYMBOL_LEFT);
    lv_obj_center(label);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_tileview();
```

```
}
```

上述源码可知：首先创建 tileview 部件，然后调用 lv_tileview_add_tile()函数添加四个 Tile 视图和设置滑动方向，最后添加 Tile 视图对象内容。

38.4.3 下载验证

把工程编译并下载到开发板中， 如图 38.4.3.1 所示：

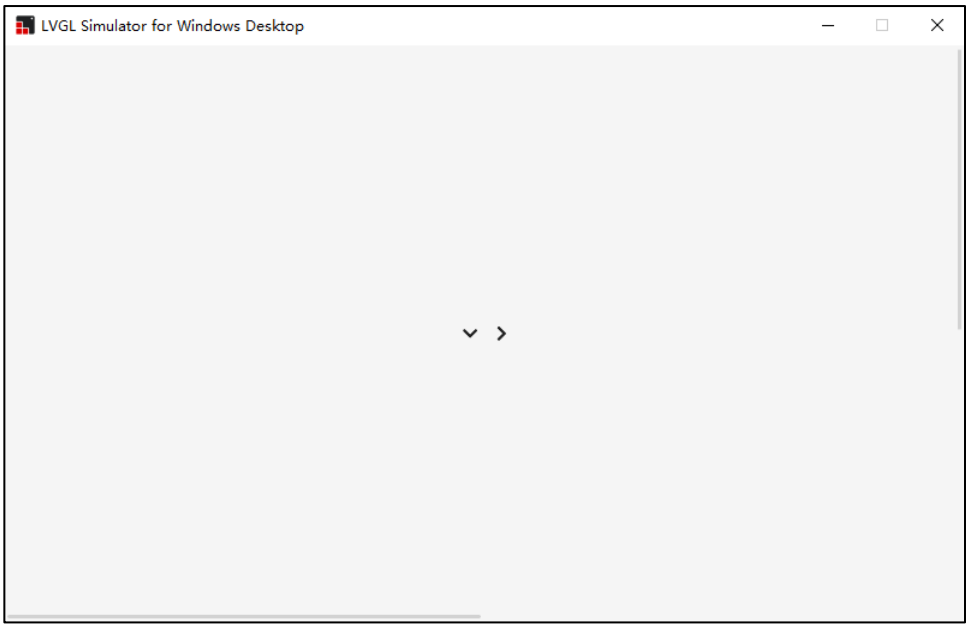


图 38.4.3.1 tileview 部件实验示意图

上图中，Tile1 视图可往左移和往下移切换 Tile 视图。

第三十九章 窗口部件(lv_win)

win 部件是一个容器状的对象，它带有标题、按钮和内容区域的头部构建而成。

本章分为如下几部分：

39.1 窗口部件的组成

39.2 窗口部件的相关知识

39.3 窗口部件的 API 函数

39.4 窗口部件的实验

39.1 窗口部件的组成

窗口部件是由其他小部件构建而成，这些小部件如以下所示：

① Header 区域的背景：lv_obj。

② Header 区域上的标题：lv_label。

③ Header 区域上的按钮：lv_btn。

④ 背景上的内容区域：lv_obj。

根据上述描述可知：窗口控件可以归类于两个部分组成，这三个部分如下图 39.1.1 所示：

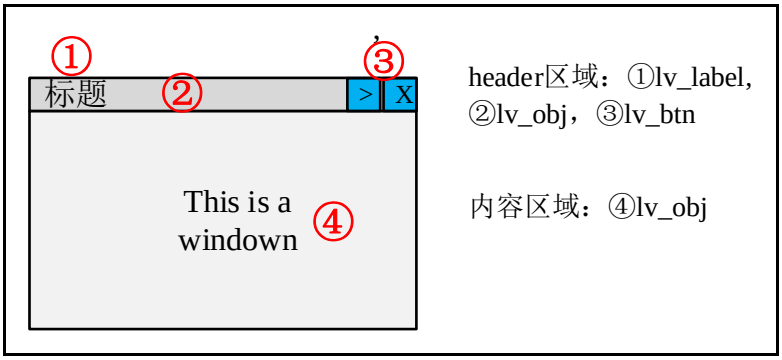


图 39.1.1 窗口部件组成部分

设置窗口部件样式属性，可根据组成的部件来设置样式属性。

39.2 窗口部件的相关知识

39.2.1 创建窗口

窗口部件是以“lv_win”标识符来表示，用户只需调用 lv_win_create 函数创建窗口，该函数具有两个形参，第一个形参为指向窗体的父类，而第二个形参为指向 header 区域的高度。

39.2.2 标题与按键的设置

窗体的标题可调用 lv_win_add_title 函数设置标题，该函数设置的文本会添加到图 39.1.1 中 header 区域当中。

按钮部件可使用 lv_win_add_btn 函数添加到窗口 header 区域当中。该函数的 icon 形参是图像源，btn_width 形参是设置按钮的宽度。

注意：标题和按钮将按照调用函数的顺序添加。因此，添加一个按钮、一个文本和两个其他按钮将导致左边有一个按钮、一个标题和右边有两个按钮。标题的宽度被设置为占据头部的所有剩余空间。换句话说:它向右推送所有添加在标题之后的按钮。

下面我们使用一个示意图来讲解本小节的知识，该示意图如下图 39.2.2.1 所示：

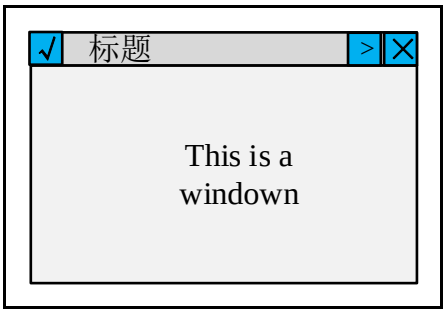


图 39.2.2.1 窗口部件

从上图可知：谁先创建谁排在最左边，所以上图的 header 区域是按键、标题、按键、按键顺序排列的。

39.2.3 获取窗口部件区域

从上图 39.1.1 可知：窗口部件区域主要分为两个区域，一个 header 区域，而另一个为内容区域。用户想获取它们所在的区域是需要调用不同的函数，如 header 区域只需调用 lv_win_get_header 函数获取，而内容区域只需调用 lv_win_get_content 函数获取。

39.3 窗口部件的 API 函数

LVGL 提供给用户的窗口部件相关函数，如表 39.3.1 所示：

函数	描述
lv_win_create()	创建窗口部件
lv_win_add_title()	添加标题
lv_win_add_btn()	添加按键
lv_win_get_header()	获取 header 区域
lv_win_get_content()	获取内容区域

表 39.3.1 窗口部件相关函数

接下来我们来讲解 LVGL 的 lv_win 部件常用的函数有哪些：

1. lv_win_create 函数

该函数的作用是创建一个窗口对象，其函数原型如下所示：

```
lv_obj_t *lv_win_create(lv_obj_t *parent, lv_coord_t header_height)
```

该函数的形参描述如表 39.3.2 所示：

参数	描述
parent	父对象指针
header_height	Header 区域的高度

表 39.3.2 函数 lv_win_create()形参描述

返回值：指向创建的窗口的指针。

2. lv_win_add_title 函数

该函数的作用是设置窗口标题，其函数原型如下所示：

```
lv_obj_t *lv_win_add_title(lv_obj_t *win, const char *txt)
```

该函数的形参描述如表 39.3.3 所示：

参数	描述
win	指向对象的指针
txt	设置的文本

表 39.3.3 函数 lv_win_add_title()形参描述

返回值：指向标题 label 的指针。

3. lv_win_add_btn 函数

该函数的作用是设置窗口按键，其函数原型如下所示：

```
lv_obj_t *lv_win_add_btn(lv_obj_t *win, const void *icon, lv_coord_t btn_w)
```

该函数的形参描述如表 39.3.4 所示：

参数	描述
win	指向对象的指针
icon	设置图像源
btn_w	按键的宽度

表 39.3.4 函数 lv_win_add_btn()形参描述

返回值：指向按键 lv_btn 的指针。

4. lv_win_get_header 函数

该函数的作用是获取 header 区域，其函数原型如下所示：

```
lv_obj_t *lv_win_get_header(lv_obj_t *win)
```

该函数的形参描述如表 39.3.5 所示：

参数	描述
win	指向对象的指针

表 39.3.5 函数 lv_win_get_header()形参描述

返回值：指向 header 区域 lv_obj 指针。

5. lv_win_get_content 函数

该函数的作用是获取 content 区域，其函数原型如下所示：

```
lv_obj_t *lv_win_get_content (lv_obj_t *win)
```

该函数的形参描述如表 39.3.6 所示：

参数	描述
win	指向对象的指针

表 39.3.6 函数 lv_win_get_content()形参描述

返回值：指向 content 区域 lv_obj 指针。

39.4 窗口部件的实验

39.4.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数创建一个 win 部件，在该部件的 hread 区域添加三个 btn 和一个标题以及在 content 区域创建一个图片按键，当用户按下 hread 区域 btn 按键时，content 区域中的图片按键显示用户按下的 btn 按键内容。

led_task 任务：主要测试 LED0 是否在闪烁，如果 LED0 在运行，则代表系统再运行中。

该实验的实验工程，请参考《LVGL 例程 39 lv_win(窗口)》。

39.4.2 软件设计

39.4.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 39.4.2.1.1 所示：

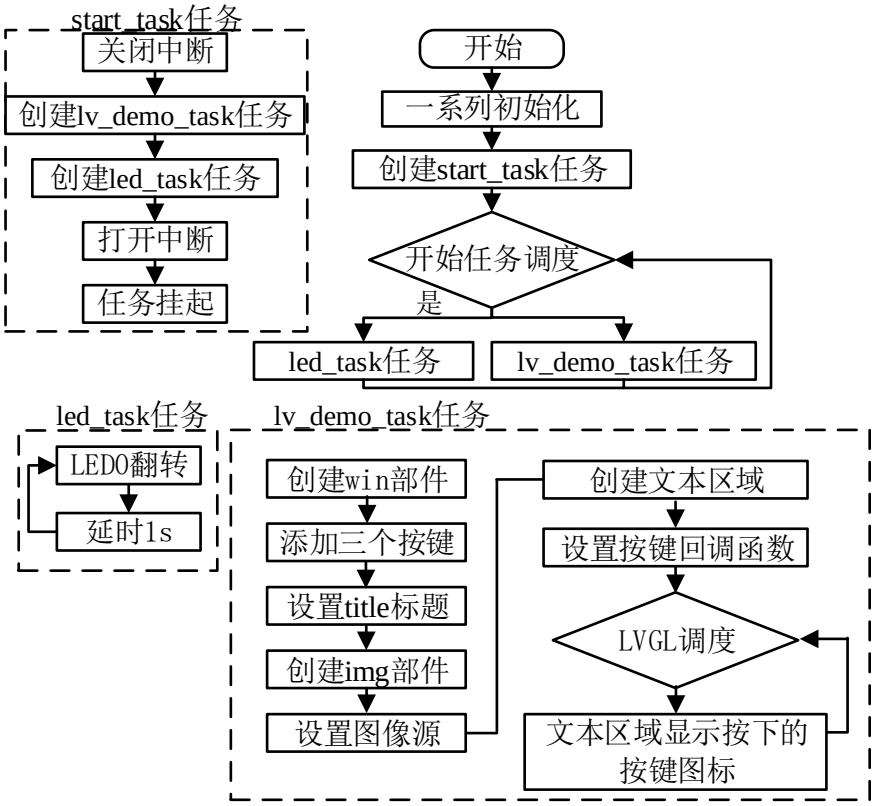


图 39.4.2.1.1 LVGL lv_win(窗口)实验流程图

39.4.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief  例
 * @param  无
 * @return 无
 */
static void lv_example_win(void)
{
    const lv_font_t* font;
    /* 根据屏幕大小设置字体 */
    if (scr_act_width() <= 320)
        font = &lv_font_montserrat_14;
```

```
    else if (scr_act_width() <= 480)
        font = &lv_font_montserrat_18;
    else
        font = &lv_font_montserrat_22;

    lv_coord_t dev_num = scr_act_width() <= 480 ? 6 : 7;
    /* 定义并创建 win */
    lv_obj_t* win = lv_win_create(lv_scr_act(), scr_act_height() / dev_num);
    /* win 添加按钮 */
    lv_obj_t* btn_back = lv_win_add_btn(win, LV_SYMBOL_LEFT,
    scr_act_height() / dev_num);
    /* win 添加按钮 */
    lv_obj_t* btn_forward = lv_win_add_btn(win, LV_SYMBOL_RIGHT,
                                           scr_act_height() / dev_num);
    /* win 添加按钮 */
    lv_obj_t* btn_refresh = lv_win_add_btn(win, LV_SYMBOL_REFRESH,
                                       scr_act_height() / dev_num);
    /* win 添加标题 */
    lv_obj_t* title = lv_win_add_title(win, "http://www.openedv.com/");
    /* 获取 win 的内容物体 */
    lv_obj_t* content = lv_win_get_content(win);
    lv_obj_t* textarea = lv_textarea_create(content);
    lv_obj_center(textarea);
    lv_obj_set_width(textarea, scr_act_width() * 0.7);
    lv_obj_align(lv_textarea_get_label(textarea), LV_ALIGN_CENTER, 0, 0);
    lv_textarea_set_one_line(textarea, true);
    lv_obj_set_style_text_font(textarea, font, LV_PART_MAIN);
    lv_obj_update_layout(textarea);

    lv_obj_t* img = lv_img_create(content);
    lv_img_set_src(img, &img_zdyz);
    if (scr_act_width() <= 320)
        lv_img_set_zoom(img, 160);
    else if (scr_act_width() <= 480)
        lv_img_set_zoom(img, 256);
    else
        lv_img_set_zoom(img, 300);
    lv_obj_align_to(img, lv_scr_act(), LV_ALIGN_CENTER, 0,
                    -lv_obj_get_height(textarea) * 1.5);

    lv_obj_add_event_cb(btn_back, event_btn_back_cb, LV_EVENT_CLICKED,
                        textarea);
    lv_obj_add_event_cb(btn_forward, event_btn_forward_cb, LV_EVENT_CLICKED,
```

```
                        textarea);
    lv_obj_add_event_cb(btn_refresh, event_btn_refresh_cb, LV_EVENT_CLICKED,
                        textarea);
}

/**
 * @brief  LVGL 演示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    lv_example_win();
}
```

上述源码可知：创建 win、img 和 textarea 部件，然后添加 win 窗口按键和设置标题 Tile，最后设置窗口按键回调函数。当按下窗口按键时，触发相应回调函数，如下源码所示：

```
/**
 * @brief  后退按钮事件回调
 * @param  e:事件结构体
 * @return 无
 */
static void event_btn_back_cb(lv_event_t* e)
{
    lv_obj_t* textarea = (lv_obj_t*)e->user_data;

    lv_textarea_set_text(textarea, LV_SYMBOL_LEFT);
}

/**
 * @brief  前进按钮事件回调
 * @param  e:事件结构体
 * @return 无
 */
static void event_btn_forward_cb(lv_event_t* e)
{
    lv_obj_t* textarea = (lv_obj_t*)e->user_data;

    lv_textarea_set_text(textarea, LV_SYMBOL_RIGHT);
}

/**
 * @brief  刷新按钮事件回调
 * @param  e:事件结构体
```

```
 * @return 无
 */
static void event_btn_refresh_cb(lv_event_t* e)
{
    lv_obj_t* textarea = (lv_obj_t*)e->user_data;

    lv_textarea_set_text(textarea, LV_SYMBOL_REFRESH);
}
```

上述源码可知：当按下窗口按键时，根据按键对象触发相应的回调函数，该回调函数主要设置 textarea 文本区域的内容。

39.4.3 下载验证

把工程编译并下载到开发板中， 如图 39.4.3.1 所示：

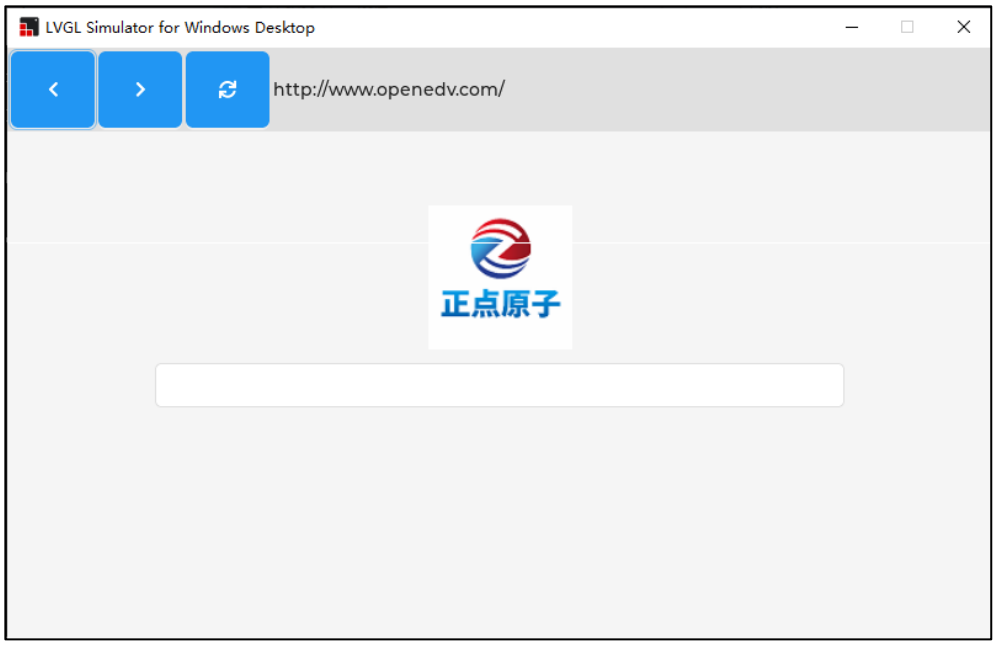


图 39.4.3.1 win 窗口实验示意图

第四十章 动画图像(lv_animimg)

动画图像类似于普通的“图像”对象。唯一的区别是，不是设置一个源图像，而是设置多个源图像的数组。用户可以指定持续时间和重复次数调用动画图像。

本章分为如下几部分：

40.1 动画图像部件的组成
40.2 动画图像部件的相关知识
40.3 动画图像部件的 API 函数
40.4 动画图像部件的实验

40.1 动画图像部件的组成

动画图像部件是由 LV_PART_MAIN 部分组成，LV_PART_MAIN 一个使用典型背景样式属性的背景矩形，使用图像样式属性可以修改图像本身的样式。

40.2 动画图像部件的相关知识

动画图像部件是以“lv_animimg”标识符来表示，前面我们已经讲解到，动画图像类似于多个图像源的数组，严格来说我们遍历这个图像源数组就可以使图片以动画的形式播放。下面我们以一个图来讲解动画图像部件的作用，如下图 40.2.1 所示：

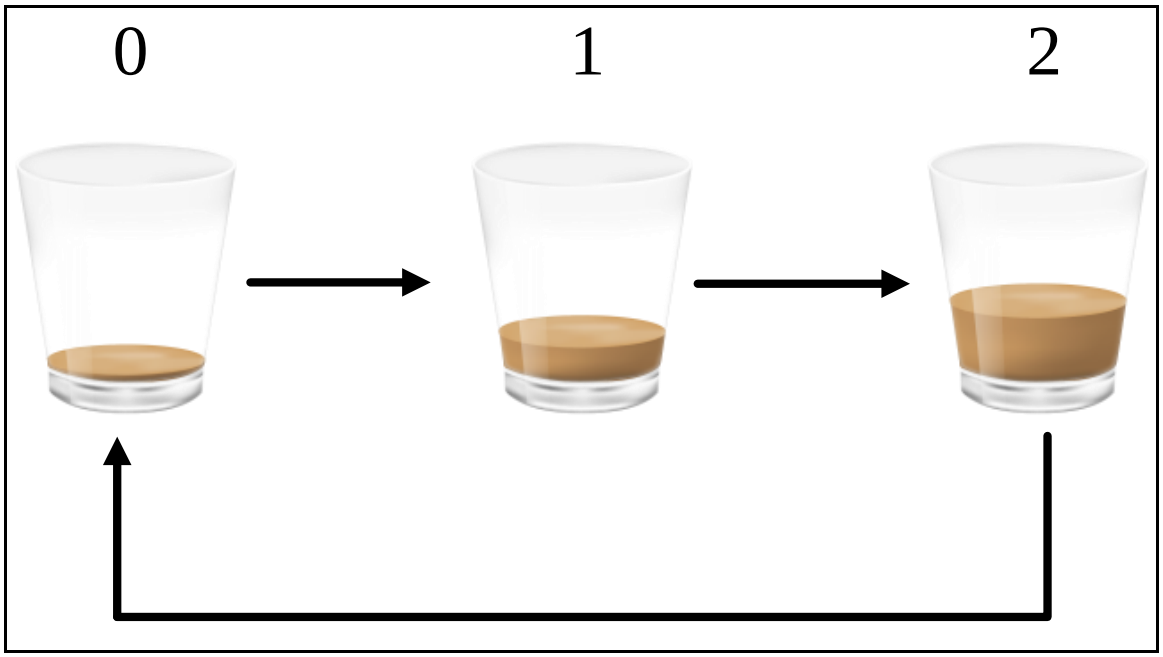


图 40.2.1 动画图像的作用

从上图可知：我们把 0、1 和 2 图像源加载到动画图像数组中，然后调用 lv_animimg_start 函数开启动画图像部件，这样我们可以看到图像从 0~2 转变。

40.3 动画图像部件的 API 函数

LVGL 提供给用户的动画图像部件相关函数，如表 40.3.1 所示：

函数	描述
lv_animimg_create ()	创建动画图像部件
lv_animimg_set_src ()	设置图像源
lv_animimg_start ()	开启动画图像
lv_animimg_set_duration ()	设置动画图像的时间
lv_animimg_set_repeat_count ()	设置动画图像重复次数

表 40.3.1 动画图像部件相关函数

接下来我们来讲解 LVGL 的 lv_ animimg 部件常用的函数有哪些：

1. lv_animimg_create 函数

该函数的作用是创建一个动画图像对象，其函数原型如下所示：

```
lv_obj_t * lv_animimg_create(lv_obj_t * parent)
```

该函数的形参描述如表 40.3.2 所示：

参数	描述
parent	父对象指针

表 40.3.2 函数 lv_animimg_create()形参描述

返回值：指向创建动画图像的指针。

2. lv_animimg_set_src 函数

该函数的作用是设置动画图像的图像源，其函数原型如下所示：

```
void lv_animimg_set_src(lv_obj_t * obj,  lv_img_dsc_t * dsc[], uint8_t num)
```

该函数的形参描述如表 40.3.3 所示：

参数	描述
obj	指向动画图像对象
dsc[]	图像描述符数组
num	图像源数量

表 40.3.3 函数 lv_animimg_set_src()形参描述

返回值：无。

3. lv_animimg_start 函数

该函数的作用是开启动画图像，其函数原型如下所示：

```
void lv_animimg_start(lv_obj_t * obj)
```

该函数的形参描述如表 40.3.4 所示：

参数	描述
obj	指向动画图像对象

表 40.3.4 函数 lv_animimg_start()形参描述

返回值：无。

4. lv_animimg_set_duration 函数

该函数的作用是设置动画图像时间，其函数原型如下所示：

```
void lv_animimg_set_duration(lv_obj_t * obj, uint32_t duration)
```

该函数的形参描述如表 40.3.5 所示：

参数	描述
obj	指向动画图像对象
duration	动画图像时间

表 40.3.5 函数 lv_animimg_set_duration()形参描述

返回值：无。

5. lv_animimg_set_repeat_count 函数

该函数的作用是设置动画图像重复次数，其函数原型如下所示：

```
void lv_animimg_set_repeat_count(lv_obj_t * obj, uint16_t count)
```

该函数的形参描述如表 40.3.5 所示：

参数	描述
obj	指向动画图像对象
count	重复次数

表 40.3.5 函数 lv_animimg_set_repeat_count()形参描述

返回值：无。

40.4 动画图像部件的实验

关于菜单部件的实验，大家可以参考 LVGL 官方提供的实验，该实验的路径为 lvgl-release-v8.2\examples\widgets\animimg 文件夹下，该文件夹具有一个实例提供给读者学习。关于该实例所用到的图像文件可在 lvgl-release-v8.2\examples\assets 路径下找到。

第四十一章 菜单部件(lv_menu)

菜单小部件可轻松创建多级菜单。命令、子菜单或者分隔条都可包括在菜单之中。每一个创建的菜单至多有四级子菜单。菜单部件是一组通常在功能上相关的命令或控件的容器。提供特殊的布局行为，并支持用户启动的工具栏大小调整和排列。

本章分为如下几部分：

41.1 菜单部件的组成

41.2 菜单部件的相关知识

41.3 菜单部件的 API 函数

41.4 菜单部件的实验

41.1 菜单部件的组成

菜单小部件是由两个大类构建而成，第一个大类为主容器，第二个大类为侧边栏容器，它们各自也是由不同的部分构建而成的，下面我们就以一个示意图来讲解菜单部件，如图 41.1.1 所示：

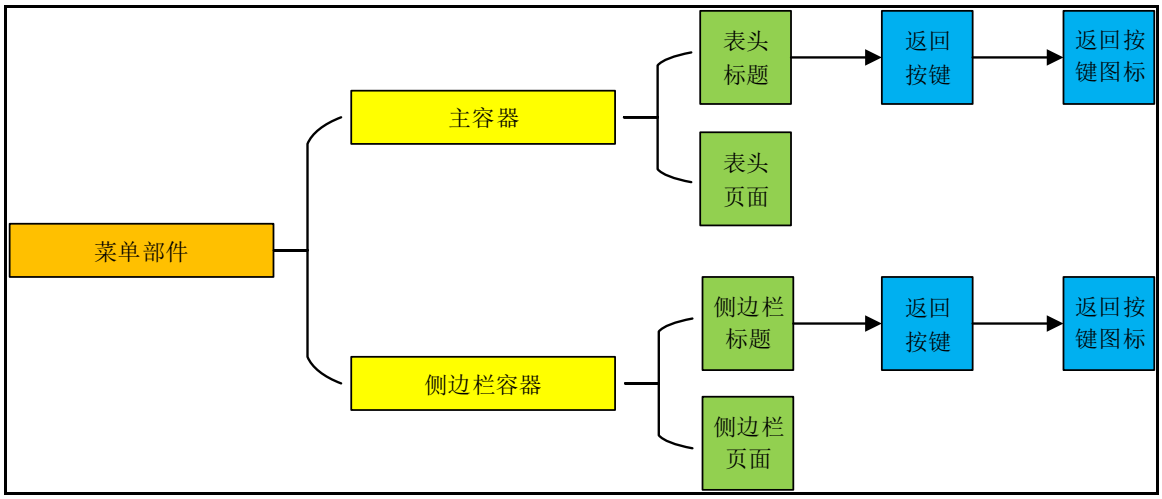


图 41.1.1 菜单部件的构建部分

从上图可知：菜单部件主要包含两个容器，它们分别为主容器和侧边栏容器，主容器是由标题和页面组成，而侧边栏容器也是类似部分组成。

41.2 菜单部件的相关知识

41.2.1 创建菜单部件

创建菜单部件非常简单，我们只需调用 lv_menu_create 函数创建菜单部件。一开始创建菜单部件时，它只包含主容器和根返回按键，如图 41.2.1.1 所示：

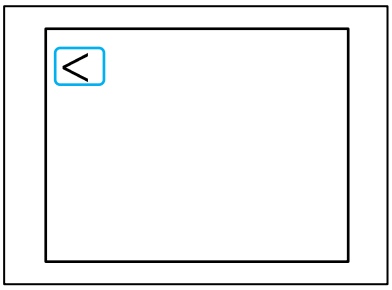


图 41.2.1.1 初始菜单部件示意图

从上图可知：菜单部件创建时，它只单单创建主容器以及根返回按键，其他部分需由用户自己添加。

41.2.2 标题模式

标题模式可以设置在表头标题和侧边栏标题，用户可调用函数 lv_menu_set_mode_header 设置它们的标题模式，该函数具有两个形参，第一个形参指向菜单部件对象，而第二个形参表示要设置的标题模式，标题模式具有三种类型，这些类型如下所示：

① LV_MENU_HEADER_TOP_FIXED：标题位于顶部。
② LV_MENU_HEADER_TOP_UNFIXED：标题位于顶部，可以滚动出视图。
③ LV_MENU_HEADER_TOP_UNFIXED：标题位于底部。

41.2.3 根返回按钮模式

根返回按键模式具有两种，这些模式如下所示：

① LV_MENU_ROOT_BACK_BTN_DISABLED：菜单根按键禁用。
② LV_MENU_ROOT_BACK_BTN_ENABLED：菜单根按键启用。

上述模式主要设置根返回按键的模式，用户可调用函数 lv_menu_set_mode_root_back_btn 设置。默认情况下，这个根按键是启用的。

41.2.4 创建菜单页面

创建一个新的空菜单页。我们可以向该页面添加任何小部件，创建新的菜单页面的函数是 lv_menu_page_create，该函数具有两个形参，第一个形参指向菜单对象，而第二个形参表示菜单的标题。注意：这个函数不能单独使用，可以结合下面的设置主容器页面和设置侧边栏容器页面小节的内容。

41.2.5 设置主容器页面

创建菜单页面之后，用户可以使用 lv_menu_set_page 将其设置为主要区域。该函数具有两个形参，第一个形参指向菜单的对象，而第二个形参指向菜单页面，如果该形参为 NULL，则该函数是用来清除主菜单和清除菜单历史。当我们调用该函数把菜单页面设置成主容器页面时，菜单部件的主体会生成自动生成主容器和侧边栏容器，下面我们就以一个示意图来讲解这个知识点，如下图 41.2.4.1 所示：

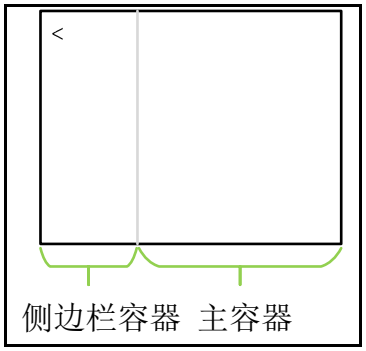


图 41.2.4.1 创建菜单页面

从上图可知：当我们使用函数 lv_menu_set_page 设置菜单页面为主容器页面，系统自动把菜单部件的主体划分为两个部分，第一个部分为侧边栏容器，另一个是主容器。

接下来，我们以一个简单实例来讲解这小节的知识，如下源码所示：

```
void lv_mainstart()
{
    /* 第一步：创建菜单部件 */
    lv_obj_t* menu = lv_menu_create(lv_scr_act());
    /* 设置根返回按键模式 */
    lv_menu_set_mode_root_back_btn(menu, LV_MENU_ROOT_BACK_BTN_ENABLED);
    /* 设置菜单部件大小 */
    lv_obj_set_size(menu, lv_disp_get_hor_res(NULL),
                    lv_disp_get_ver_res(NULL));
    /* 设置菜单部件中间对齐 */
    lv_obj_center(menu);
    /* 第二步：创建菜单页面 */
    lv_obj_t* sub_mechanics_page = lv_menu_page_create(menu,"ATK");
    /* 第三步：设置菜单页面样式 */
    lv_obj_set_style_pad_hor(sub_mechanics_page,
                        lv_obj_get_style_pad_left(lv_menu_get_main_header(menu),
                        0), 0);
    /* 第四步：设置菜单页面为主容器页面 */
    lv_menu_set_page(menu, sub_mechanics_page);
}
```

此函数主要把创建的菜单页面设置为主容器的页面，该页面的标题是“ATK”字符串，下面我们就以一个示意图来模拟上述源码功能，大家可以把这些代码在 PC 模拟器或者开发板上运行，该示意图如下图 41.2.4.2 所示：

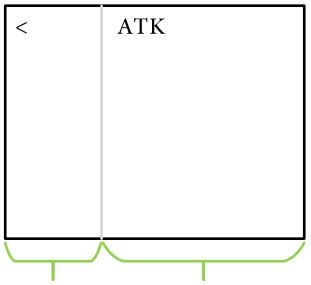


图 41.2.4.2 主容器页面的添加

从上图可知：我们已经在主容器种添加了一个页面，大家可以在这个页面实现自己的 UI。

41.2.6 设置侧边栏容器页面

创建菜单页面之后，用户可以使用 lv_menu_set_sidebar_page 函数将其设置到侧边栏。该函数具有两个形参，第一个形参指向菜单的对象，而第二个形参指向菜单页面，如果该形参为 NULL，则该函数是用来清除侧边栏。本小节的知识和上小节知识类似，主要讲解菜单页面添加带侧边栏容器中。

接下来，我们以一个简单实例来讲解这小节的知识，如下源码所示：

```
void lv_mainstart()
{
    /* 第一步：创建菜单部件 */
    lv_obj_t* menu = lv_menu_create(lv_scr_act());
    /* 设置根返回按键模式 */
    lv_menu_set_mode_root_back_btn(menu, LV_MENU_ROOT_BACK_BTN_ENABLED);
    /* 设置菜单部件大小 */
    lv_obj_set_size(menu, lv_disp_get_hor_res(NULL),
                    lv_disp_get_ver_res(NULL));
    /* 设置菜单部件中间对齐 */
    lv_obj_center(menu);
    /* 第二步：创建菜单页面 */
    lv_obj_t* sub_mechanics_page = lv_menu_page_create(menu,"ATK");
    /* 第三步：设置菜单页面样式 */
    lv_obj_set_style_pad_hor(sub_mechanics_page,
                        lv_obj_get_style_pad_left(lv_menu_get_main_header(menu),
                        0), 0);
    /* 第四步：设置菜单页面为侧边栏容器页面 */
    lv_menu_set_sidebar_page(menu, sub_mechanics_page);
}
```

此函数主要把创建的菜单页面设置为侧边栏容器的页面，该页面的标题是“ATK”字符串，下面我们就以一个示意图来模拟上述源码功能，大家可以把这些代码在 PC 模拟器或者开发板上运行，该示意图如下图 41.2.4.2 所示：

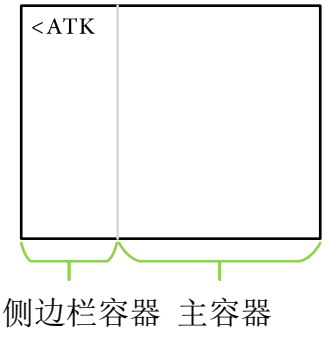


图 41.2.4.2 主容器页面的添加

从上图可知：我们已经在侧边栏容器种添加了一个页面，大家可以在这个页面实现自己的 UI。

41.2.7 菜单页面的连接

菜单页面的连接是指我们在页面上创建了一个可点击的对象，我们希望点击该对象之后系统自动打开我们已经创建好的页面，这叫做菜单页面的连接，这个操作类似于界面切换原理。菜单页面连接的使用可调用 lv_menu_set_load_page_event 函数设置连接关系，该函数具有三个形参，第一个形参指向菜单部件，第二个形参表示连接源(点击对象)，而第三个形参指向连接的页面，这个页面也是调用 lv_menu_page_create 函数创建的。下面我们使用一个示意图来讲解本小节的知识，如下图 41.2.7.1 所示：

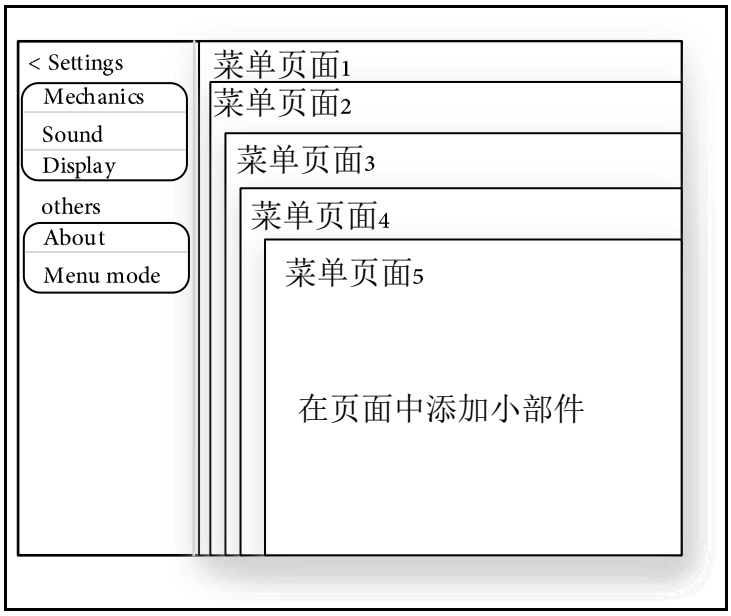


图 41.2.7.1 菜单页面的连接示意图

从上图可知：左边的侧边栏也是菜单页面，这个设置方法是调用 lv_menu_set_sidebar_page 函数设置，该菜单页面可以添加各种小部件来作为连接源(点击对象)，上图右边的是连接的页面，当用户点击侧边栏中的“Mechanics”标签可切换右边的任意菜单页面，这个菜单页面需要用户调用 lv_menu_set_load_page_event 进行连接。

41.2.8 创建一个菜单容器、空区域和分隔符

LVGL 的菜单部件提供用户三个好用的函数，它们分别为菜单容器的创建、创建空区域以及创建分隔符，这些函数是用来辅助菜单部件的开发的，这些函数如下表 41.2.8 所示：

函数	描述
lv_menu_cont_create()	创建一个新的容器
lv_menu_section_create()	创建一个空的区域
lv_menu_separator_create()	创建一个分隔符

表 41.2.8.1 菜单部件辅助函数

下面我们使用一个示意图来讲解这些菜单部件辅助函数的使用，如下图 41.2.8.1 所示：

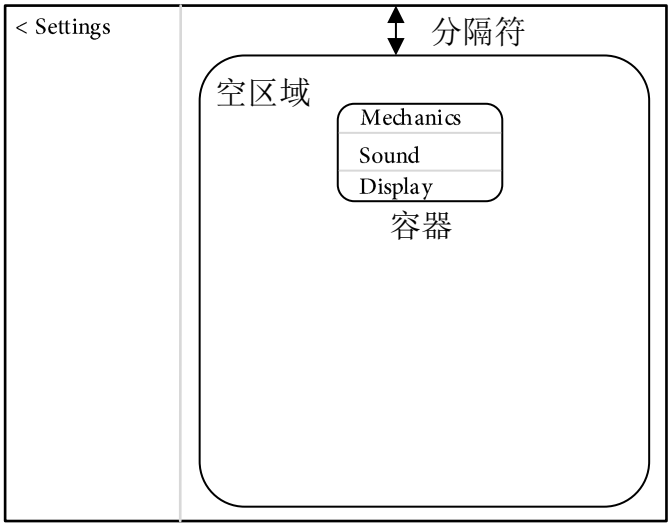


图 41.2.8.1 菜单辅助函数的作用

从上述可知：设置分隔符是指空区域与主容器顶部的相隔位置，一般我们创建菜单页面时会调用函数 lv_menu_section_create 空区域，然后在这个空区域添加我们的小部件，当然我们可以调用函数 lv_menu_cont_create 创建一个容器并在这个容器上添加小部件。

41.3 菜单部件的 API 函数

LVGL 提供给用户的菜单部件相关函数，如表 41.3.1 所示：

函数	描述
菜单部件设置函数	
lv_menu_create()	创建菜单部件
lv_menu_page_create()	创建菜单页面
lv_menu_cont_create()	创建菜单容器
lv_menu_section_create()	创建空区域
lv_menu_separator_create()	创建分隔符
lv_menu_set_page()	设置菜单页面为主容器
lv_menu_set_sidebar_page()	设置菜单页面为侧边栏容器
lv_menu_set_mode_header()	设置标题模式
lv_menu_set_mode_root_back_btn()	设置根返回按键
lv_menu_set_load_page_event()	菜单页面连接
菜单部件获取函数	
lv_menu_get_cur_main_page()	返回指向当前在 main 中显示的菜单页的指针

lv_menu_get_cur_sidebar_page()	返回指向当前在侧边栏中显示的菜单页的指针
lv_menu_get_main_header()	获取主标题
lv_menu_get_main_header_back_btn()	获取主标题返回按键
lv_menu_get_sidebar_header()	获取侧边栏标题
lv_menu_get_sidebar_header_back_btn()	获取侧边栏返回按键
lv_menu_back_btn_is_root()	设置返回根按键
lv_menu_clear_history()	清除菜单

表 41.3.1 菜单部件相关函数

41.4 菜单部件的实验

关于菜单部件的实验，大家可以参考LVGL官方提供的实验，该实验的路径为lvgl-release-v8.2\examples\widgets\menu 文件夹下，该文件夹具有五个实例提供给读者学习。

组件篇

经过前面的学习，我们已经对 LVGL 基础知识以及 LVGL 的部件知识具有深刻的认识，接下来，我们学习一下 LVGL 提供的组件库到底如何使用，LVGL 提供的组件库非常多，例如 BMP、PNG、JPEG、GIF、QR 等组件库，这些组件库相关的源码文件存放的路径是\lvgl\src\extra\libs 目录下，其中文件系统也属于组件库的一种，而 ffmpeg 和 rlottie 组件库本教程并没有涉及，大家可以在 linux 环境下学习这两个组件库。

第四十二章 LVGL BMP 图片

BMP 是英文 Bitmap（位图）的简写，它是 Windows 操作系统中的标准图像文件格式，能够被多种 Windows 应用程序所支持。随着 Windows 操作系统的流行与丰富的 Windows 应用程序的开发，BMP 位图格式理所当然地被广泛应用。这种格式的特点是包含的图像信息较丰富，几乎不进行压缩，但由此导致了它与生俱来的缺点--占用磁盘空间过大。所以，BMP 在单机上比较流行。本章主要讲解 LVGL BMP 集成解码库的使用，该 BMP 解码库是按需读取像素(不会加载整个图像)，因此使用 BMP 图像需要很少的 RAM 消耗。

本章我们分为以下介个部分讲解：

42.1 LVGL 的 BMP 解码库概述

42.2 LVGL 的 BMP 解码库实验

42.1 LVGL 的 BMP 解码库概述

复制 “LVGL 例程 5 LVGL 文件系统使用”工程并重命名为“LVGL 例程 41 BMP 图片读取”。在该工程中，创建 Middlewares/lvgl/src/bmp 工程管理项分组，该组添加 lv_bmp.c 文件。这文件在 LVGL/GUI/lvgl/src/extra/libs/bmp 路径下定义的。如图 42.1.1 所示：

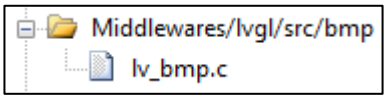


图 42.1.1 工程管理项添加文件

注意：使用 LVGL BMP 解码库之前，必须在 lv_conf.h 文件启用 LV_USE_BMP。如果该宏定义设置为 1，则启用 BMP 解码库，否则不启用该解码库。

LVGL BMP 解码库的局限性：

(1) BMP 文件只能从文件加载。如果想将它们存储在闪存中，最好使用 LVGL 的图像转换器将它们转换为 C 数组。

(2) BMP 文件的颜色格式必须与 LV_COLOR_DEPTH 匹配。使用 GIMP 以所需的格式保存图像，例如我们正点原子的 tft 显示屏支持 16 位深度的，所以 LV_COLOR_DEPTH 设置为 16。

(3) 不支持调色板，简单来讲：不支持图像修改颜色。

注意：由于官方的 BMP 解码库是 32 位颜色深度，而正点原子的 TFT 显示屏只能处理 16 位颜色深度，所以需要修改 BMP 源代码，修改过程如下所示：

1. 修改 bmp_dsc_t 结构体，如下源码所示：

```
typedef struct {
    uint8_t blue;
    uint8_t green;
    uint8_t red;
    uint8_t reserved;
}rgbquad_t;

typedef struct {
    lv_fs_file_t f;
    unsigned int px_offset;
```

```
    int px_width;
    int px_height;
    unsigned int bpp;
    int row_size_bytes;
    rgbquad_t palette[256];
} bmp_dsc_t;
```

上述源码可知：我们定义了一个 rgbquad_t 结构体，该结构体主要描述 RGB 颜色值。

2. 修改函数 decoder_read_line ，如下源码所示：

```
static lv_res_t decoder_read_line( lv_img_decoder_t * decoder,
                                   lv_img_decoder_dsc_t * dsc,
                                   lv_coord_t x,
                                   lv_coord_t y,
                                   lv_coord_t len,
                                   uint8_t * buf)
{

    bmp_dsc_t * b = dsc->user_data;
    y = (b->px_height - 1) - y; /*BMP images are stored upside down*/
    uint32_t p = b->px_offset + b->row_size_bytes * y;
    p += x * (b->bpp / 8);
    lv_fs_seek(&b->f, p, LV_FS_SEEK_SET);
    lv_fs_read(&b->f, buf, len * (b->bpp / 8), NULL);

    if((b->bpp >= 16) && (b->bpp == (sizeof(lv_color_t) * 8))) {
        lv_fs_read(&b->f, buf, len * (b->bpp / 8), NULL);
#if LV_COLOR_DEPTH == 32
        lv_color32_t * bufc = (lv_color32_t*)buf;
        for(uint32_t i = 0; i < len; i++) {
            /* Byte0:blue, byte1:green, byte2:red, byte3:alpha */
            uint8_t *p = (uint8_t *)&bufc[i];
            bufc[i].ch.red = p[2];
            bufc[i].ch.green = p[1];
            bufc[i].ch.blue = p[0];
            bufc[i].ch.alpha = 0xff; /*Amazing*/
        }
#endif
    }
    else {
        lv_color_t *color_p = (lv_color_t *)buf;
        uint8_t color_buf[4];
        switch(b->bpp)
        {
            case 24:
```

```
            for(int i = 0; i < len; i++)
            {
                lv_fs_read(&b->f, color_buf, 3, NULL);
                color_p[i] = lv_color_make(color_buf[2],
                                           color_buf[1], color_buf[0]);
            }
            break;
        case 16:
            for(int i = 0; i < len; i++)
            {
                lv_fs_read(&b->f, color_buf, 2, NULL);
                uint16_t color16 = *(uint16_t *)color_buf;

                #define RGB565_R5(rgb565) ((rgb565 >> 11) & 0x1f)
                #define RGB565_G6(rgb565) ((rgb565 >> 5) & 0x3f)
                #define RGB565_B5(rgb565) ((rgb565 >> 0) & 0x1f)
                #define R5_2_R8(r5) (r5 << 3)
                #define G6_2_G8(g6) (g6 << 2)
                #define B5_2_B8(b5) (b5 << 3)

                color_p[i] = lv_color_make(R5_2_R8(RGB565_R5(color16)),
                G6_2_G8(RGB565_G6(color16)), B5_2_B8(RGB565_B5(color16)));
            }
            break;
        case 8:
            for(int i = 0; i < len; i++)
            {
                lv_fs_read(&b->f, color_buf, 1, NULL);

                uint8_t index = color_buf[0];
                rgbquad_t color = b->palette[index];
                color_p[i] = lv_color_make( color.red,
                                            color.green,
                                            color.blue);
            }
            break;
        case 4:
            for(int i = 0; i < len;)
            {
                lv_fs_read(&b->f, color_buf, 1, NULL);

                for(int j = 0; (j < 8) && (i < len); j += 4, i++) {
                    uint8_t index = (color_buf[0] >> (4 - j)) & 0x0f;
```

```
                        rgbquad_t color = b->palette[index];
                        color_p[i] = lv_color_make(color.red,
                                                   color.green,
                                                   color.blue);
                    }
                }
                break;
            case 1:
                for(int i = 0; i < len;) {
                    lv_fs_read(&b->f, color_buf, 1, NULL);

                    for(int j = 0; (j < 8) && (i < len); j++, i++) {
                        uint8_t index = (color_buf[0] >> (7 - j)) & 0x01;
                        rgbquad_t color = b->palette[index];
                        color_p[i] = lv_color_make(    color.red,
                                                       color.green,
                                                       color.blue);
                    }
                }
                break;
            default:
                break;
        }
    }

    return LV_RES_OK;
}
```

上述源码可知：颜色深度不再局限于 32 位，而是根据 b->bpp 参数选择颜色深度。

42.2 LVGL 的 BMP 解码库实验

42.2.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数调用 LVGL 的 BMP 解码库函数读取 SD 卡上的 BMP 并在显示屏上显示。

该实验的实验工程，请参考《LVGL 例程 41 BMP 图片读取》。注意：DMF407、MiniSTM32 不支持本实验。

42.2.2 软件设计

42.2.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 42.2.2.1.1 所示：

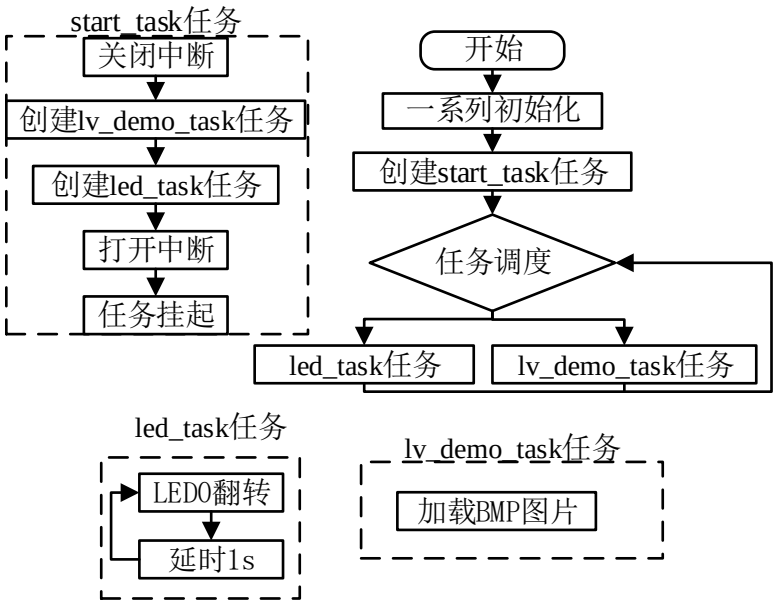


图 42.2.2.1.1 LVGL BMP 图片读取实验流程图

42.2.2.3 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
typedef struct
{
    char *img_path;
    char *label_text;
}img_info_t;

static const img_info_t draw_img[] =
{
    {"0:/PICTURE/BMP/camera.bmp", "camera"},
    {"0:/PICTURE/BMP/ALIENTEKLOGO.bmp", "ALIENTEKLOGO"},
};

#define img_Num sizeof(draw_img) / sizeof(draw_img[0])

lv_obj_t *img;

/**
```

```
 * @brief       lv_mainstart
 * @param       无
 * @retval      无
 */
void lv_mainstart()
{
    lv_obj_t *label = lv_label_create(lv_scr_act());
    lv_label_set_text(label,"BMP_Decoder");
    lv_obj_set_style_text_color(label,lv_palette_main(LV_PALETTE_RED),
                                LV_STATE_DEFAULT);
    lv_obj_set_style_text_font(label,&lv_font_montserrat_32,LV_STATE_DEFAULT);
    lv_obj_align_to(label,NULL,LV_ALIGN_TOP_MID,0,0);
    lv_obj_set_style_bg_color(lv_scr_act(),lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
    img = lv_img_create(lv_scr_act());                  /* 创建图像 */
    lv_img_set_src(img, draw_img[0].img_path);          /* 设置图像源 */
    lv_obj_align_to(img, NULL, LV_ALIGN_CENTER, 0, 0);/* 设置对齐模式 */
}
```

上述源码可知：我们主要创建 img 部件，然后调用 lv_img_set_src()设置图像源，最后设置对齐模式。

42.2.3 下载验证

把工程编译并下载到开发板中， 如图 42.2.3.1 所示：

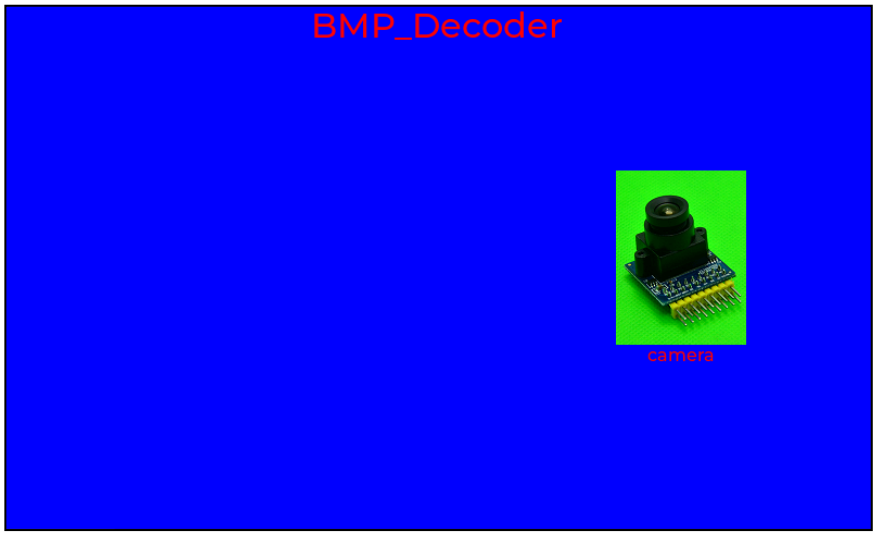


图 42.2.3.1 BMP 图片读取实验示意图

第四十三章 LVGL PNG 图片

PNG 是一种采用无损压缩算法的位图格式，其设计目的是试图替代 GIF 和 TIFF 文件格式，同时增加一些 GIF 文件格式所不具备的特性。PNG 使用从 LZ77 派生的无损数据压缩算法。本章主要讲解 LVGL PNG 集成解码库的使用。

本章我们分为以下几个部分讲解：

43.1 LVGL PNG 解码库的移植

43.2 LVGL PNG 图片显示实验

43.1 LVGL PNG 解码库的移植

复制 “LVGL 例程 5 LVGL 文件系统使用”工程并重命名为“LVGL 例程 42 PNG 图片读取”。打开工程并在工程中创建 Middlewares/lvgl/src/png 工程管理项分组，在该组添加 lodepng.c 和 lv_png.c 文件，这些文件在 LVGL/GUI/lvgl/src/extra/libs/png 路径下定义的。如图 43.1.1 所示：

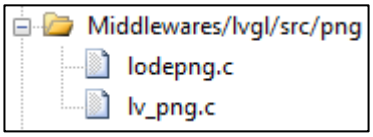


图 43.1.1 在工程管理项组添加文件

注意：使用 LVGL PNG 解码库之前，必须在 lv_conf.h 文件启用 LV_USE_PNG 宏定义。如果该宏定义设置为 1，则启用 PNG 解码库，否则不启用该解码库。

小知识：

PNG 图像被解码时，在解码过程中所需要 RAM 等于图像宽度*图像高度* 4 个字节。例如一张 800*480 的 PNG 图片，它所需要的 RAM 内存为 800*480*4 (约等于 1.4M)内存。

注意：下面的操作只针对 Mini Pro H750 和北极星开发板，其他开发板可忽略以下内容。

首先打开工程，然后点击 Linker 选项卡，最后打开分散加载文件，如图 43.1.2 所示：

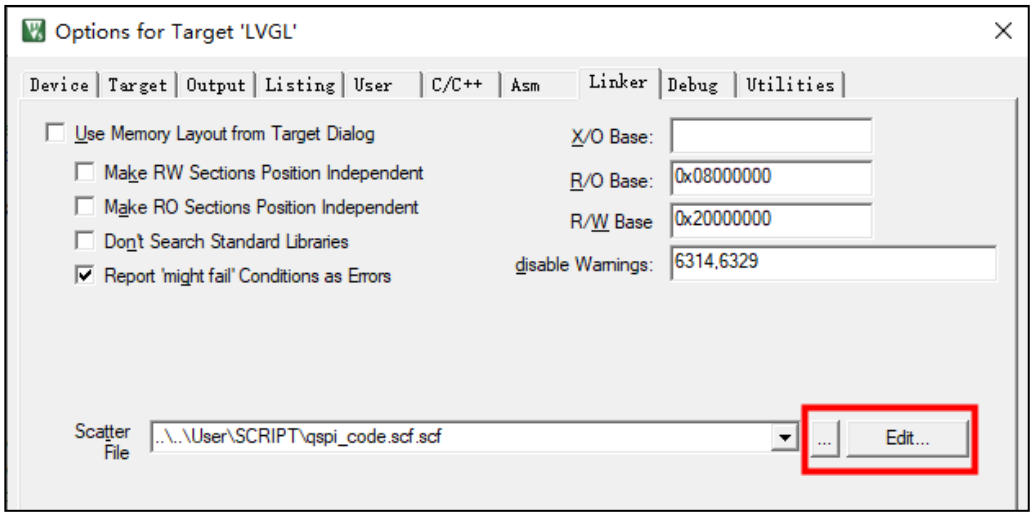


图 43.1.2 打开分散加载文件

在该文件添加图 43.1.3 的源码：

```
        rand.o
        edit.o
        hl_free.o
        hl_init.o
        hl_extend.o
        hl_alloc.o
        init_alloc.o
        rt_ctype_table.o
        rt_heap_descriptor_intlibspace.o
        maybetermalloc1.o
        initio.o
        rt_memclr_w.o
        fopen.o
        fclose.o
        sys_io.o
        strlen.o
        setvbuf.o
        __dczerorl2.o
    }
    RW_m_stmsram m_stmsram_start m_stmsram_size {
        .ANY (+RW + ZI)
    }
}
```

图 43.1.3 把这些文件放在内部 flash

注意：如果添加了图 43.1.3 的源码，程序还是无法进入 main 函数，则开启“Use MicroLIB”微库。

43.2 LVGL PNG 图片显示实验

43.2.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数调用 LVGL 的 PNG 解码库函数读取 SD 卡上的 PNG 并在显示屏上显示。

该实验的实验工程，请参考《LVGL 例程 42 PNG 图片读取》。注意：DMF407、MiniSTM32 和精英不支持本实验。

43.2.2 软件设计

43.2.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 43.2.2.1.1 所示：

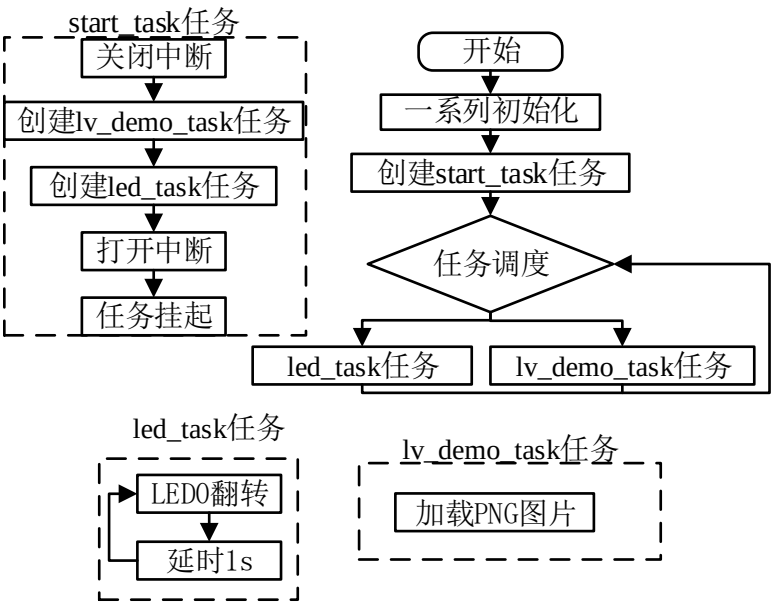


图 43.2.2.1.1 LVGL BMP 图片读取实验流程图

43.2.2.3 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/* PNG 图片结构体 */
typedef struct
{
    char *img_path;     /* 图片路径 */
    char *label_text;   /* 图片名称 */
}img_info_t;

/* 定义 PNG 路径 */
const img_info_t PNG_PATH[] =
{
    {"0:/PICTURE/PNG/mlljt.png", "xiaomao.png"},
    {"0:/PICTURE/PNG/laji.png", "laji.png"},
};
/* 获取路径的个数 */
#define image_mun (int)(sizeof(PNG_PATH)/sizeof(PNG_PATH[0]))

lv_obj_t *img;

/**
 * @brief       创建 PNG 图片文件
 * @param       parent: 父类
```

```
 * @param       path:   图片路径
 * @retval      返回图片控件
 */
lv_obj_t * lv_png_create_from_file(lv_obj_t * parent, const char * path)
{
    lv_obj_t * img = lv_img_create(parent);
    lv_img_set_src(img, path);
    lv_obj_align_to(img,NULL,LV_ALIGN_CENTER,0,0);

    return img;
}



/**
 * @brief       LVGL 程序入口
 * @param       无
 * @retval      无
 */
void lv_mainstart()
{
    lv_png_init(); /* 初始化 PNG 解码库 */
    lv_obj_t *label = lv_label_create(lv_scr_act());
    lv_label_set_text(label,"PNG_Decoder");
    lv_obj_align_to(label,NULL,LV_ALIGN_TOP_MID,0,0);
    /* 创建 PNG 文件 */
    img = lv_png_create_from_file(lv_scr_act(),PNG_PATH[0].img_path);
}
```

上述源码可知：首先创建 img 部件，然后调用 lv_img_set_src 设置图像源，最后设置对齐模式。

43.2.3 下载验证

把工程编译并下载到开发板中， 如图 43.2.3.1 所示：

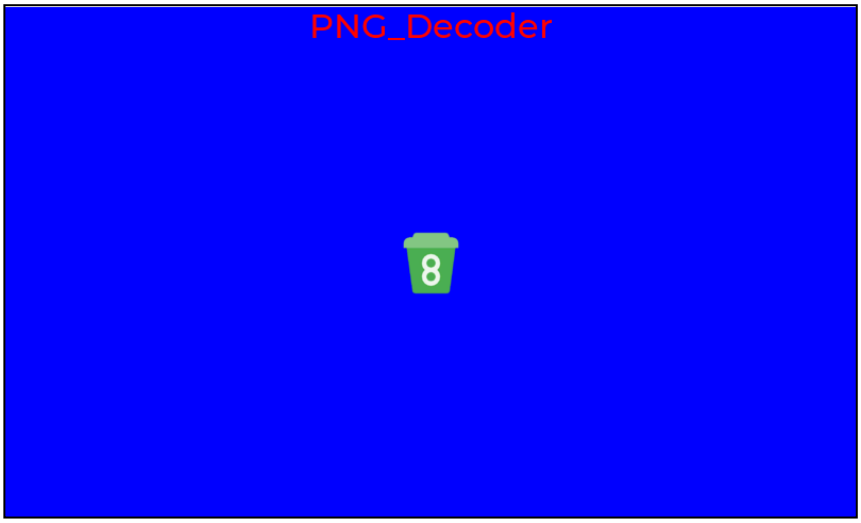


图 43.2.3.1 PNG 图片读取实验示意图

第四十四章 LVGL JPEG 图片

本章主要讲解 LVGL 的 JPEG 集成解码库的使用，JPEG 是全彩和灰度图像的标准压缩方法，JPEG 用于压缩 “真实世界”的景象、线条画、卡通，其他非现实图像并不是其强项。JPEG 会有损耗，意指输出图像与输入图像并不完全相同，因此，如果需要达到完全相同的输出位，则不能使用 JPEG。不过，对于常见的照片图像，可以得到非常好的压缩级别。LVGL 的 JPEG 解码库的原理就是将图片如何压缩为字节流以及重新解码为图片的过程。

本章我们分为以下部分讲解：

44.1 LVGL JPEG 解码库的移植

44.2 LVGL JPEG 图片显示实验

44.1 LVGL JPEG 解码库的移植

复制 “LVGL 例程 5 LVGL 文件系统使用”工程并重命名为“LVGL 例程 45 jpeg 图片读取”，在该工程中创建工程管理项分组 Middlewares/lvgl/src/jepg，在该组添加 lv_sjpg.c 和 tjpgd.c 文件，这些文件在 LVGL/GUI/lvgl/src/extra/libs/sjpg 路径下定义的。

用户如何使用该解码库呢，LVGL 作者已经提供一个方案给我们了，如图 44.1.1 所示：

17 lines (14 sloc) | 376 Bytes

```
#include "../../lv_examples.h"
#if LV_USE_SJPG && LV_BUILD_EXAMPLES

/**
 * Load an SJPG image
 */
void lv_example_sjpg_1(void)
{
    lv_obj_t * wp;

    wp = lv_img_create(lv_scr_act());
    /* Assuming a File system is attached to letter 'A'
     * E.g. set LV_USE_FS_STDIO 'A' in lv_conf.h */
    lv_img_set_src(wp, "A:lvgl/examples/libs/sjpg/small_image.sjpg");
}

#endif
```

图 44.1.1 官方解码库的示例

首先打开 lv_conf.h 文件，然后在该文件找到 LV_USE_SJPG 宏定义并设置为 1 启用 JPEG 解码库，该解码库的初始化在 lv_init/lv_extra_init 函数下调用 lv_split_jpeg_init 函数。

官方对 LVGL 的 JPEG 解码器概述：

① 该解码库支持常规 jpg 和自定义 sjpg 格式。

② 解码普通 jpg 会占用整个未压缩图像内存（建议用于具有更多 RAM 的设备）。

③ sjpg 是基于“普通” JPG 的自定义格式，是为 lvgl 专门制作的。

④ sjpg 是“ split-jpeg”，它是一堆带有 sjpg 标头的小 jpeg 片段。

⑤ sjpg 大小将几乎与 jpg 文件相当，或者可能稍大。
⑥ 从磁盘（读取）和 c 数组读取实现。
⑦ 如果在缓存中可用，则 SJPEG 帧片段缓存可实现行的快速获取。
⑧ 默认情况下，sjpg 图像缓存为图像宽度 * 2 * 16 字节（可以修改）。
⑨ 当前仅支持 16 位图像格式（可做）。
⑩ JPG 和 SJPG 图像只解码所需的部分，因此不能缩放或旋转

LVGL 解码库支持三种读取 jpeg 方式，这三种方式如下所示：

1. 将 JPG 转换为 C 数组，使用在线图片软件转换即可，注意：Color format = RAW, output format = C Array 选项配置。
2. SJPG 图片格式读取，这类不是我们涉及的领域，但是我们 MCU 可以读取这类型的图片格式，SJPG 图片格式使用 JPG 图片经过 python3 和 PIL 库进行转换。
3. 直接使用 LVGL 文件系统读取 SD 卡路径下的 jpeg 图片。

44.1.1 JPG 转换 SJPG 图片格式

在上面我们已经介绍过，JPG 转换 SJPG 图片格式需要 python3 和 PIL 库，首先我们下载 python 软件安装，打开网址：https://www.python.org/downloads/，然后点击下载，如图 44.1.1.1 所示：



图 44.1.1.1 下载 python

从上图中我们选择 windowns 版本下载，python 的安装请大家自行安装。安装完成之后，按下键盘 win+r 快捷键，输入 CMD 进入命令行，在该命令行下输入“python”字段，测试安装是否成功，图 44.1.1.2 所示：



图 44.1.1.2 安装 python 成功

如果命令行出现上图的信息，则表示 python 安装成功。

安装 PIL 库，请在命令行输入” easy_install Pillow”，如图 44.1.1.3 所示：

```
C:\Users\ALIENTEK>easy_install Pillow
WARNING: The easy_install command is deprecated and will be removed in a future version.
Searching for Pillow
Best match: pillow 8.2.0
Processing pillow-8.2.0-py3.9-win-amd64.egg
pillow 8.2.0 is already the active version in easy-install.pth

Using c:\python39\lib\site-packages\pillow-8.2.0-py3.9-win-amd64.egg
Processing dependencies for Pillow
Finished processing dependencies for Pillow

C:\Users\ALIENTEK>
```

图 44.1.1.3 安装 PIL 库

如果安装 PIL 库失败，则把 python 安装路径加载到电脑环境变量中，然后重新安装 PIL 库。

在桌面新建一个文件夹，把 lvgl-release-v8.2/ scripts/路径下的文件 jpg_to_sjpg.py 脚本复制到新建的文件夹下，最后放入转换的 jpeg 图片到该文件夹中，如图 44.1.1.4 所示：

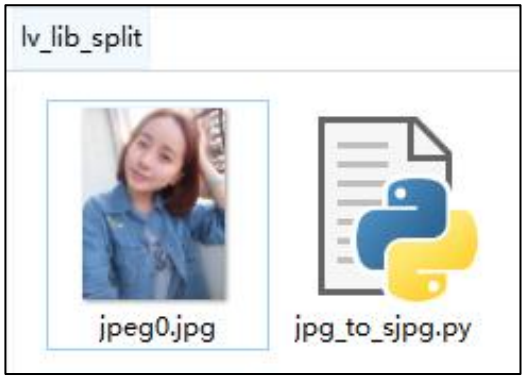


图 44.1.1.4 文件夹下的文件

笔者不知道大家有没有学过 python，它运行文件夹下的.py 文件可以使用 pycharm、vscobe 等软件，如果不使用这些软件，可以使用命令行执行.py 文件，这些命令如以下所示：

```
cd C:\Users\ATK\Desktop\lv_lib_split
python jpg_to_sjpg.py jpeg0.jpg
```

首先我们使用 cd 命令跳到新建的文件夹下，然后执行.py 文件，如图 44.1.1.4 所示：

```
C:\Users\ATK>cd C:\Users\ATK\Desktop\lv_lib_split

C:\Users\ATK\Desktop\lv_lib_split>python jpg_to_sjpg.py jpeg0.jpg

Conversion started...

Input:
        jpeg0.jpg
        RES = 240 x 320

Output:
        Time taken = 0.26 sec
        bin size = 35.2 KB
        jpeg0.sjpg      (bin file)
        jpeg0.c         (c array)

All good!
```

图 44.1.1.4 执行 jpg_to_sjpg.py 脚本

当执行完毕时，在新建文件夹下得到图 44.1.1.5 的文件：



图 44.1.1.5 生成 sjpg 格式图片

上述图 44.1.1.5 中，jpeg0.sjpg 图片格式可使用文件系统读取显示，而 jpeg0.c 文件可直接加载到 flash 中。

注意：如果使用 Python3.10 以上版本，则使用“pip3 install pillow”命令安装 PIL 库，然后输入拷贝 jpg_to_sjpg.py 脚本和转换图片到文件夹中，最后我们进入转换的文件夹，并在这个路径下输入“python jpg_to_sjpg.py jpeg0.jpg”命令转换。

44.2 LVGL JPEG 图片显示实验

44.2.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数调用 LVGL 的 JPEG 解码库函数读取 SD 卡上的 JPEG 并在显示屏上显示。

该实验的实验工程，请参考《LVGL 例程 45 jpeg 图片读取》。注意：DMF407、MiniSTM32 和精英不支持本实验。

44.2.2 软件设计

44.2.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 44.2.2.1.1 所示：

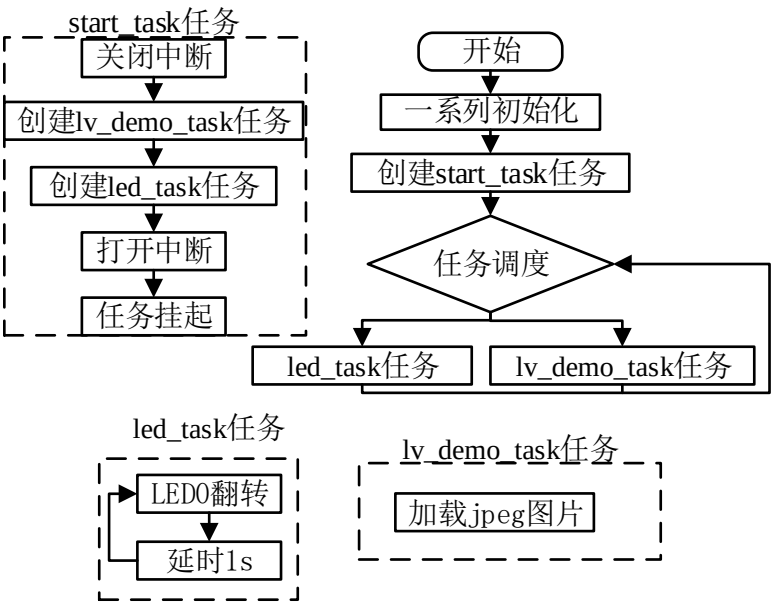


图 44.2.2.1.1 LVGL jpeg 图片读取实验流程图

44.2.2.2 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
/**
 * @brief       lvgl 程序入口
 * @param       无
 * @retval      无
 */
void lv_mainstart()
{
    /* 创建标签 */
    lv_obj_t *label = lv_label_create(lv_scr_act());
    /* 设置标签颜色 */
    lv_label_set_text(label,"JPEG_Decoder");
    /* 设置文本颜色 */
    lv_obj_set_style_text_color(label,lv_palette_main(LV_PALETTE_RED),
                                LV_STATE_DEFAULT);
    /* 设置文本字体 */
    lv_obj_set_style_text_font(label,&lv_font_montserrat_32,LV_STATE_DEFAULT);
    /* 设置顶部中间对齐 */
    lv_obj_align(label,LV_ALIGN_TOP_MID,0,0);
    /* 设置背景颜色 */
    lv_obj_set_style_bg_color(lv_scr_act(),lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
```

```
    /* 创建 image 控件 */
    lv_obj_t * img = lv_img_create(lv_scr_act());
    /* 设置图像源 */
    lv_img_set_src(img, "0:/PICTURE/JPEG/SIM900A.jpg");
    /* 中间对齐 */
    lv_obj_align(img,LV_ALIGN_CENTER,0,0);
}
```

上述源码可知：创建一个 img 部件，然后调用 lv_img_set_src()设置图像源，最后中间对齐。注意：必须在 lv_cong.h 使能 LVGL 文件系统。

44.2.3 下载验证

把工程编译并下载到我们的开发板中， 如图 44.2.3.1 所示：

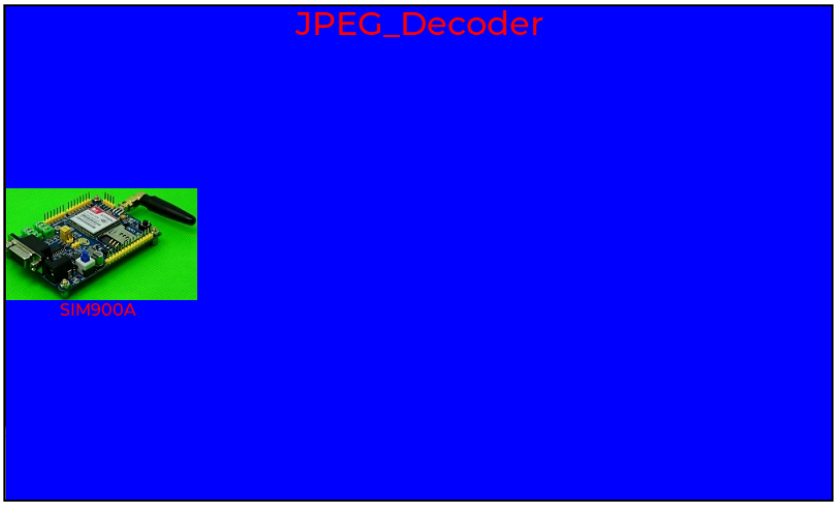


图 44.2.3.1 jpeg 图片读取实验示意图

第四十五章 LVGL GIF 读取

上一章已经给大家讲解了 jpeg 图片读取和 jpeg 图片转换成 sjpg 格式方法，本章主要讲解 LVGL 提供 GIF 解码库的使用。

本章我们分为以下几个部分讲解：

45.1 LVGLGIF 解码库的移植

45.2 LVGL GIF 图形交换显示实验

45.1 LVGL GIF 解码库的移植

复制“LVGL 例程 5 LVGL 文件系统使用”工程并重命名为“LVGL 例程 46 GIF 读取”，在该工程中创建 Middlewares/lvgl/src/gif 工程管理项组，该组添加 gifdec.c 和 lv_gif.c 文件，这些文件在 LVGL/GUI/lvgl/src/extra/libs/gif 路径下定义的。如图 45.1.1 所示：



图 45.1.1 工程管理项添加文件

注意：使用 LVGL GIF 解码库之前，必须在 lv_conf.h 文件启用 LV_USE_GIF。如果该宏定义设置为 1，则启用 GIF 解码库，否则不启用该解码库。

小知识：

GIF 图像被解码时，在解码过程中所需要 RAM 是以颜色深度决定的。

(1) LV_COLOR_DEPTH 为 8，则所需要的 RAM 为宽度*高度*3 内存。

(2) LV_COLOR_DEPTH 为 16，则所需要的 RAM 为宽度*高度*4 内存。

(3) LV_COLOR_DEPTH 为 32，则所需要的 RAM 为宽度*高度*5 内存。

例如，颜色深度为 16 的一个 800*480 的 GIF 图像，它所需要的 RAM 为 800*480*4(约等于 1.4M)内存。

GIF 读取方案：

LVGL GIF 解码库读取具有两种方案，第一个是把 GIF 转成 C 数组格式文件，而另一种就是文件系统读取。

注意：如果 GIF 图像使用官方在线转换工具，则有 Color format 选择颜色格式(RAW)。

45.2 LVGL GIF 图形显示实验

45.2.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数调用 LVGL 的 GIF 解码库函数读取 SD 卡上的 gif 并在显示屏上显示。

该实验的实验工程，请参考《LVGL 例程 43 GIF 图片读取》。注意：DMF407、MiniSTM32 和精英不支持本实验。

45.2.2 软件设计

45.2.2.1 程序流程图

根据上述的例程功能，我们得到以下流程图，如图 45.2.2.1.1 所示：

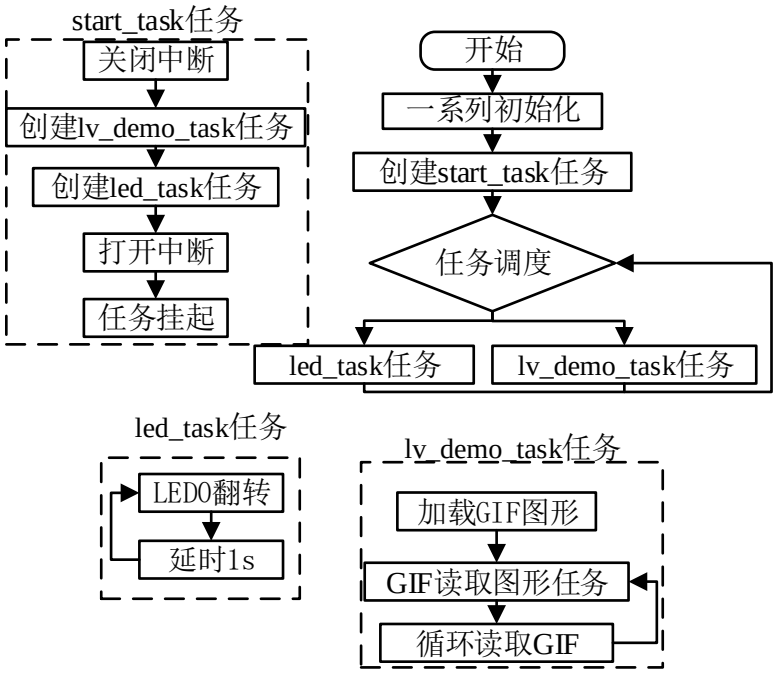


图 45.2.2.1.1 LVGL GIF 图片读取实验流程图

45.2.2.3 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
typedef struct
{
    char *img_path;
    char *label_text;
}img_info_t;

const img_info_t GIF_PATH[] =
{
    {"0:/PICTURE/GIF/alientek.gif", "alientek.gif"},
};

#define image_mun (int)(sizeof(GIF_PATH)/sizeof(GIF_PATH[0]))
```

```
int image = 0;
lv_obj_t *img;
static lv_style_t img_style;
extern int has_next;

void lv_mainstart(void)
{
    lv_obj_t *label;
    label = lv_label_create(lv_scr_act());
    lv_label_set_text(label,"GIF_Decoder");
    lv_obj_set_style_text_color(label, lv_palette_main(LV_PALETTE_RED),
                                LV_STATE_DEFAULT);
    lv_obj_set_style_text_font(label,&lv_font_montserrat_32,LV_STATE_DEFAULT);
    lv_obj_align_to(label,NULL,LV_ALIGN_TOP_MID,0,0);
    lv_obj_set_style_bg_color(lv_scr_act(),lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
    lv_obj_set_style_bg_color(lv_scr_act(),lv_palette_main(LV_PALETTE_BLUE),
                              LV_STATE_DEFAULT);
    img = lv_gif_create(lv_scr_act());
    lv_gif_set_src(img, GIF_PATH[image].img_path);
    lv_obj_align_to(img,NULL,LV_ALIGN_CENTER,0,0);

    lv_timer_create(lv_my_timer,10,img);
}
```

从上述源码可知：调用 lv_gif_create 函数创建 GIF 控制块，然后调用 lv_gif_set_src 函数显示 GIF 图像，最后调用 lv_timer_create 创建 10ms 的定时器并设置 lv_my_timer 定时器回调函数。该定时器回调函数如下：

```
void lv_my_timer(lv_timer_t *timer)
{
    if (has_next == 0)          /* 如果是最后一帧，那么切换 GIF */
    {
        image++;
        lv_obj_del(img);        /* 删除前面的 GIF */

        if (image >= image_mun) /* 判断 GIF 库包含的个数是否最大 */
        {
            image = 0;          /* 重新开始展示 */
            img = lv_gif_create(lv_scr_act());
            lv_gif_set_src(img, GIF_PATH[image].img_path);
        }
        else                    /* 如果不是最后的 GIF */
        {
            img = lv_gif_create(lv_scr_act());
```

```
            lv_gif_set_src(img, GIF_PATH[image].img_path);
        }
        timer->user_data = img; /* 设置任务数据等于获取的图片数据 */
        lv_obj_align_to(img,NULL,LV_ALIGN_CENTER,0,0);
    }
}
```

上述源码可知：如果显示 GIF 最后帧数时，则调用 lv_obj_del 删除 GIF 控制块并重新调用 lv_gif_create 创建 GIF 控制块。**注意：**has_next 参数在 lv_gif.c 定义的，该参数主要获取 GIF 的最后帧数。由于该参数原本为局部变量，所以笔者修改它为全局变量。如果为 0，则表示最后帧数。

45.2.3 下载验证

把工程编译并下载到我们的开发板中， 如图 45.2.3.1 所示：

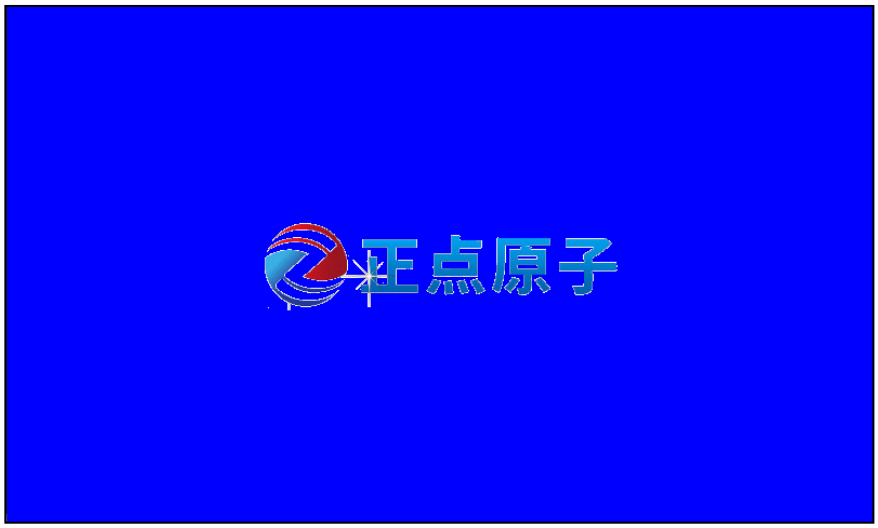


图 45.2.3.1GIF 图形读取实验示意图

第四十六章 LVGL 二维码库

二维码又称二维条码，常见的二维码为 QR Code，QR 全称 Quick Response，是一个近几年来移动设备上超流行的一种编码方式，它比传统的 Bar Code 条形码能存更多的信息，也能表示更多的数据类型。本章主要讲解 LVGL QR Code 二维码库的使用。

本章我们分为几个小节来讲解：

46.1 LVGL 二维码库移植

46.2 LVGL 二维码实验

46.1 LVGL 二维码库移植

复制“LVGL 例程 5 LVGL 文件系统使用”工程并重命名为“LVGL 例程 44 二维码显示”，在该工程中，创建 Middlewares/lvgl/src/qrcode 工程管理项分组，该组主要添加 lv_arcode.c 和 qrcodegen.c 文件，这些文件在 LVGL/GUI/lvgl/src/extra/libs/qrcode 路径下定义的。如图 46.1.1 所示：

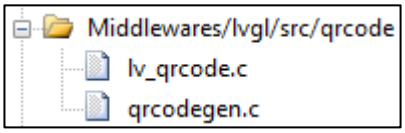


图 46.1.1 把 LVGL 二维码库添加到工程

46.1.1 LVGL 二维码库 API 函数

LVGL 的二维码库只提供用户三个函数使用，如表 46.1.1.1 所示：

函数	描述
lv_qrcode_create()	创建一个 QR 码
lv_qrcode_update()	设置二维码对象的数据
lv_qrcode_delete()	删除一个 QR 码对象

图 46.1.1.1 LVGL 二维码库的 API 函数

1. lv_qrcode_create 函数

该函数的作用是创建一个空的 QR 码，其函数原型如下所示：

```
lv_obj_t * lv_qrcode_create(lv_obj_t * parent,
                            lv_coord_t size,
                            lv_color_t dark_color,
                            lv_color_t light_color)
```

该函数的形参描述如表 46.1.1.2 所示：

参数	描述
parent	指向一个创建 QR 码的对象
size	二维码的宽度和高度
dark_color	二维码的暗颜色
light_color	二维码的高亮色

表 46.1.1.2 函数 lv_qrcode_create()形参描述

返回值：指向创建的 QR code 对象的指针。

2. lv_qrcode_update 函数

该函数的作用是设置二维码对象的数据，其函数原型如下所示：

```
lv_res_t lv_qrcode_update( lv_obj_t * qrcode,
                           const void * data,
                           uint32_t data_len)
```

该函数的形参描述如表 46.1.1.3 所示：

参数	描述
qrcode	指向 QR 代码对象的指针
data	数据显示
data_len	以字节为单位的数据长度

表 46.1.1.3 函数 lv_qrcode_update()形参描述

返回值：LV_RES_OK: 没有错误;LV_RES_INV:错误。

3. lv_qrcode_delete 函数

该函数的作用是删除一个 QR 码对象，其函数原型如下所示：

```
void lv_qrcode_delete(lv_obj_t * qrcode)
```

该函数的形参描述如表 46.1.1.4 所示：

参数	描述
qrcode	指向 QR 代码对象的指针

表 46.1.1.4 函数 lv_qrcode_delete()形参描述

返回值：无

46.2 LVGL 二维码实验

46.2.1 硬件设计

1. 例程功能

本实验主要测试 meter 部件相关 API 函数的使用，在 lvgl_demo.c 文件中设计了四个任务，它们分别为 start_task、lv_demo_task 和 led_task 任务，下面我们分别地讲解这些任务到底实现什么功能。

start_task 任务：主要负责创建 user_task、touch_task 和 led_task 任务，最后删除自身。

lv_demo_task 任务：主要调用 lv_mainstart 函数执行 lvgl 程序，该函数调用 LVGL 二维码库的 API 函数绘画二维码。

该实验的实验工程，请参考《LVGL 例程 44 二维码显示》。注意：DMF407、MiniSTM32 和精英不支持本实验。

46.2.2 软件设计

46.2.2 程序流程图

根据上述例程功能分析，我们得到以下流程图，如图 46.2.2.1 所示：

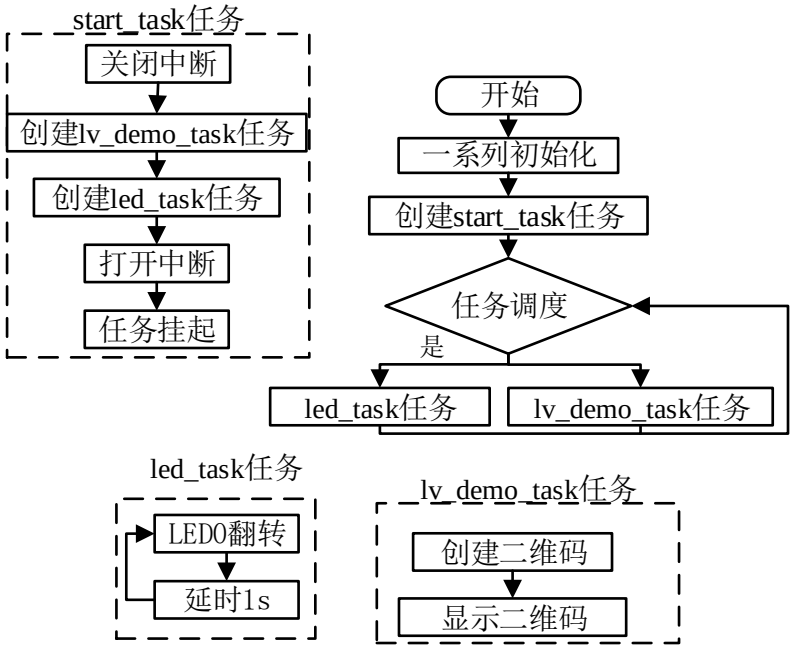


图 46.2.2.1 二维码显示实验流程图

46.2.3 程序解析

关于 LVGL 程序主要在 lv_mainstart.c 文件定义，首先我们看 lv_mainstart 函数，该函数是 LVGL 的程序入口，如下源码所示：

```
const char * data = "Hello ALIENTEK(正点原子) ";

/**
 * @brief  二维码显示
 * @param  无
 * @return 无
 */
void lv_mainstart(void)
{
    /* 创建一个标签 */
    lv_obj_t *label_time = lv_label_create(lv_scr_act());
    /* 设置标签的文本 */
    lv_label_set_text(label_time,"ALIENTEK_QR");
    /* 设置标签的文本字体颜色 */
    lv_obj_set_style_text_color(label_time,lv_palette_main(LV_PALETTE_RED),
                                LV_STATE_DEFAULT);
    /* 设置标签的文本字体 */
    lv_obj_set_style_text_font(label_time,&lv_font_montserrat_32,
                               LV_STATE_DEFAULT);
    /* 设置标签的顶部中间对齐 */
    lv_obj_align(label_time,LV_ALIGN_TOP_MID,0,0);
    /* 创建一个 lcddev.width/2 的二维码 */
```

```
    lv_obj_t * qr = lv_qrcode_create(lv_scr_act(),
                                     lv_obj_get_width(lv_scr_act())/2,
                                     lv_color_hex3(0x33f),
                                     lv_color_hex3(0xeef));
    /* 设置数据 */
    lv_qrcode_update(qr, data, strlen(data));
    /* 二维码中间对齐 */
    lv_obj_align(qr,LV_ALIGN_CENTER,0,0);
}
```

上述源码可知：首先调用 lv_qrcode_create 函数创建二维码部件，然后调用 lv_qrcode_update 函数设置二维码显示的数据。

46.2.3 下载验证

把工程编译并下载到我们的开发板中，如图 46.2.3.1 所示：



图 46.2.3.1 二维码显示示意图

第四十七章 SquareLine Studio 使用

使用 SquareLine Studio 的设计师可以快速有效地将他们的想法付诸实践。它极大地减少了程序员的负担，并产生了更优化的工作流程。SquareLine Studio 是基于开源的 LVGL 图形库。由于此导出的代码可以完全开放源代码，无需任何专有的预构建库文件。LVGL 还确保即使在低功率设备上的高性能。使用 SquareLine Studio 和学习如何将 UI 组合在一起是非常直观的。

本章我们分为几个小节来讲解：

47.1 SquareLine Studio 初探

47.2 SquareLine Studio 实验

47.3 SquareLine Studio 移植到工程

47.1 SquareLine Studio 初探

Squareline Studio 是一个多平台的设计工具，可以帮助设计师和开发人员快速高效地工作。我们的目标是帮助设计师和开发人员使用相同的文件格式，SquareLine Studio 使之成为可能，通过创建一个完美的代码到您的项目。无论您使用的是 C 语言还是 Python 语言，都可以导出这两种语言的代码。这些代码可以移植到我们工程当中。下面我们分几个部分来介绍 SquareLine Studio 设计器整体框架。

1. SquareLine Studio 设计器的安装

SquareLine Studio 设计器的下载地址为：https://squareline.io/downloads，该设计器可分为三种安装包，它们分别为 window、linux 以及 MAC OS 安装包，这里我们选择 window 版本安装包，如图 47.1.1 所示：

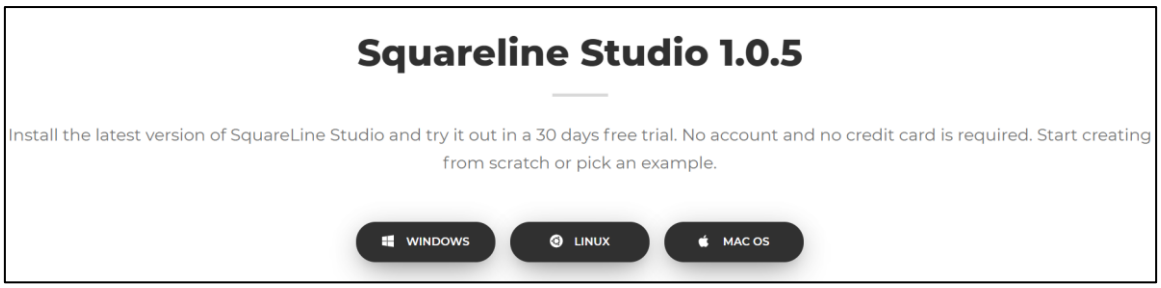


图 47.1.1 SquareLine Studio 设计器安装包

上图中，我们选择 windows 版本安装包即可，关于该设计器的介绍文档可在这个路径下查看：https://docs.squareline.io/docs/introduction/tipycal_dev，该介绍文档都是纯英文的，如果大家对英文不太熟悉，可看本章节的内容，笔者尽量讲解全面一些。

根据图 47.1.1 所示，我们点击“WINDOWS”下载 SquareLine Studio 设计器，下载完成之后我们就会得到 SquareLine_Studio_Windows_v1_0_5.zip 压缩包。在这个压缩包里面有一个 SquareLine_Studio_Setup.exe 文件，双击运行该文件如图 47.1.2 所示：



图 47.1.2 SquareLine_Studio 安装界面

点击上图中的 Options 选项选择安装路径，这里我们默认选择 C 盘的路径，然后点击“install”安装该软件，直到弹出完成界面并点击“finish”选项即可。

2. SquareLine Studio 设计器创建工程

SquareLine Studio 设计器创建工程，这里笔者分为几个步骤来讲解：

第一步：在桌面上双击 SquareLine Studio 打开设计器，进入后的界面如图 47.1.3 所示：

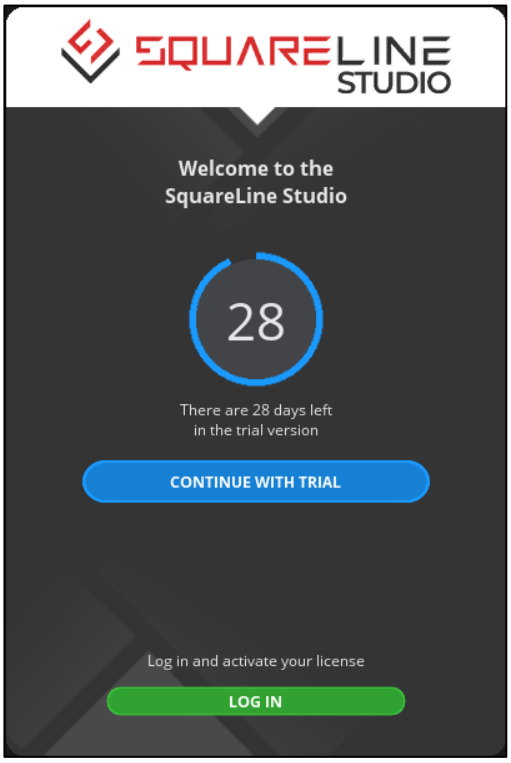


图 47.1.3 SquareLine Studio 登录界面

从上图可知：SquareLine Studio 设计器是一个付费软件，大家可以在它的官方购买使用时间，购买地址为：https://squareline.io/pricing/licenses，这里只有三个购买项，如图 47.1.4 所示：

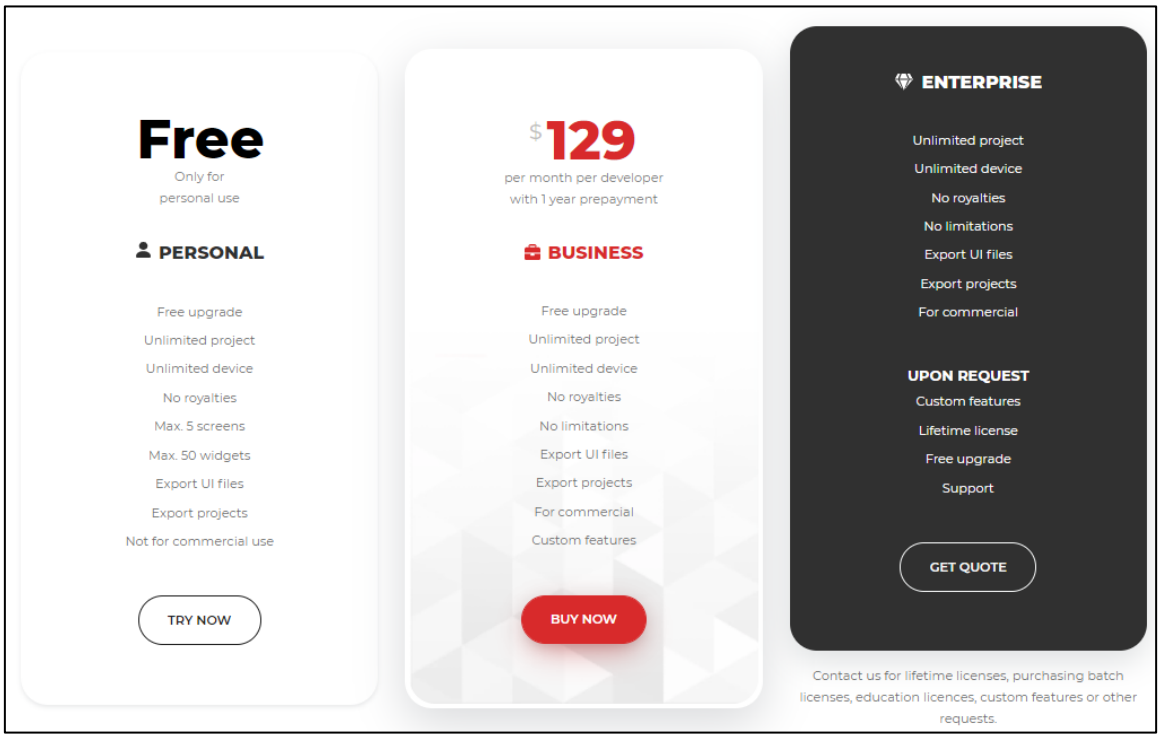


图 47.1.4 SquareLine Studio 设计器购买方式

从图 47.1.3 中，我们选择“CONTINUE WITH TRIAL”选项，这个选项表示不够买并试用该软件，试用期为 30 天。

第二步：点击“CONTINUE WITH TRIAL”选项进入构建工程界面，如图 47.1.5 所示：

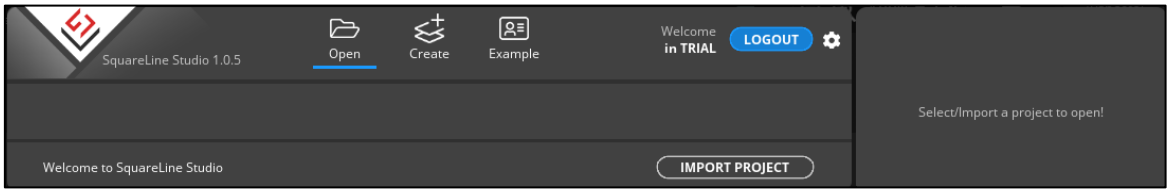


图 47.1.5 构建工程界面

上图中，“Open”以及“IMPORT PROJECT”选项都是可以导入以创建的工程，“Create”选项为创建 SquareLine Studio 设计器工程，“Example”选项是官方提供的工程 demo，这些 demo 非常漂亮，大家可以打开它们并运行，如图 47.1.6 所示：

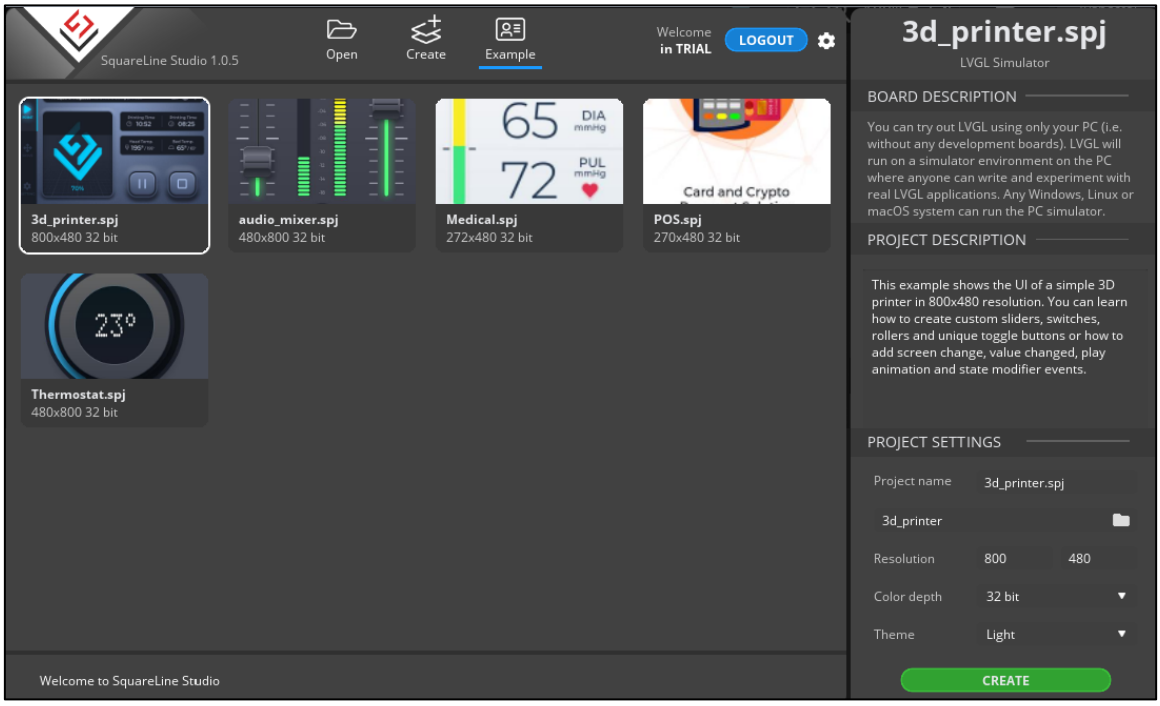


图 47.1.6 官方提供的工程 demo

大家可以选择自己喜欢的 demo，然后点击右下方的“CREATE”选项创建工程。

第三步：点击上图 47.1.5 的“Create”选项构建自己的工程，如图 47.1.7 所示：

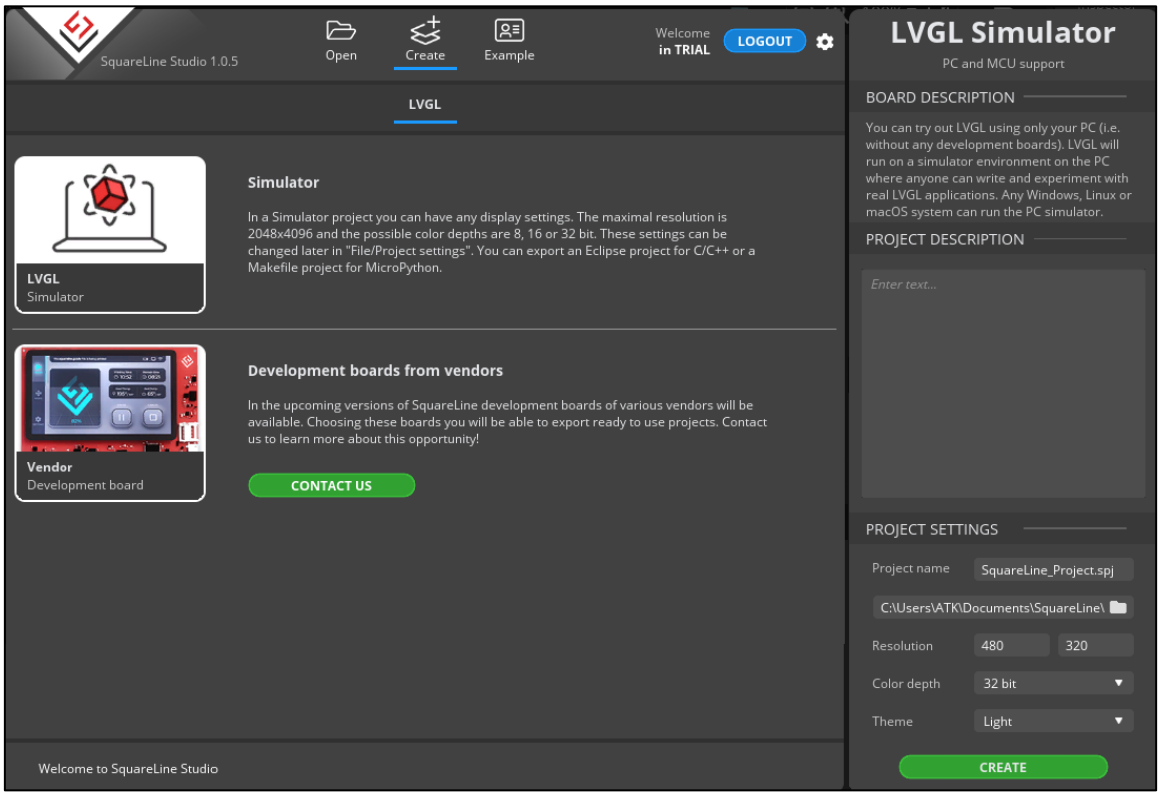


图 47.1.7 创建工程界面

我们可以在上图右下角的“PROJECT SETTINGS”框架填写工程相关的参数，首先“Project name”为工程的名称，它下方的区域是设置工程保存的路径，“Resolution”选项是设置 UI 界面的宽度和高度，“Color depth”为 UI 的颜色深度，这里我们选择 16bit，因为我们

正点原子提供的 LCD 都是 16 位颜色深度，“Theme”选项是 UI 的主题，最后点击“CREATE”选项创建工程。

3. SquareLine Studio 框架介绍

SquareLine Studio 框架一共分为两个大板块，这些板块分别为：表头菜单、面板菜单，下面我们分别介绍它们内部的架构组成。

(1) 表头菜单：

表头菜单分为三个子菜单，这些子菜单分别为 File Menu、Export Menu、Help Menu 三个子菜单，下面分别介绍这些子菜单包含的内容，如图 45.1.8 所示：

- **File Menu**

New：创建工程。

Open：打开工程。

Save：保存工程。

Save As：另存为。

Preferences： 查看软件信息。

Project settings： 当前项目的设置。

Exit：退出软件 。

- **Export Menu**

Export File：导出文本。

Export Project：导出工程。

- **Help Menu**

SquareLine Studio docs：软件的文档网页。

LVGL Website： LVGL 官网。

YouTube channel：未实现选项。

EULA：用户许可协议(EULA)网页。

(2) 面板菜单

面板菜单是由 6 个板块组成，它们分别为层次和动画体系板块、小部件板块、可编辑视图板块、资源和控制台板块、事件板块、样式属性及历史和字库管理板块。

下面我们分别地讲解这些板块的作用，如下所示：

- **层次和动画体系板块**

这个板块主要分为两个部分，一个是层次体系和动画体系，层次体系主要选择并放置添加到屏幕上的元素(小部件)，而动画体系就是添加动画效果的部分，这个板块位于 SquareLine Studio 软件的左上角，如图 47.1.8 所示：

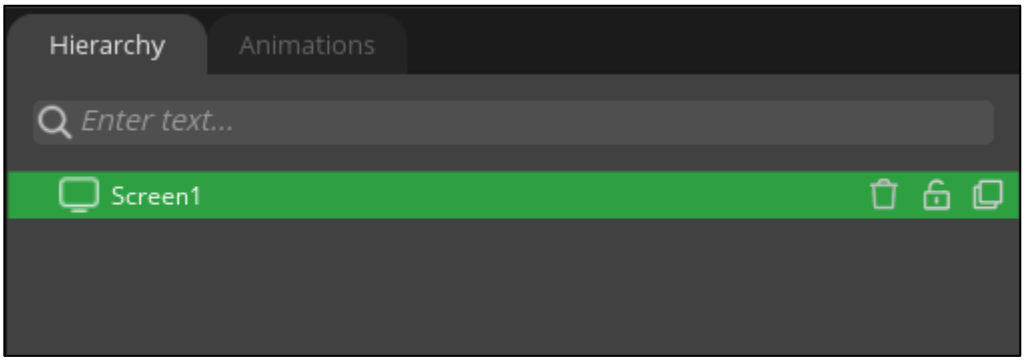


图 47.1.8 层次和动画体系板块

- **小部件板块**

这个板块主要在屏幕上放置小部件，这些小部件前面的章节中有讲解到，这里我们就无需讲解了，这个板块位于 SquareLine Studio 软件的左下角区域，如图 47.1.9 所示：

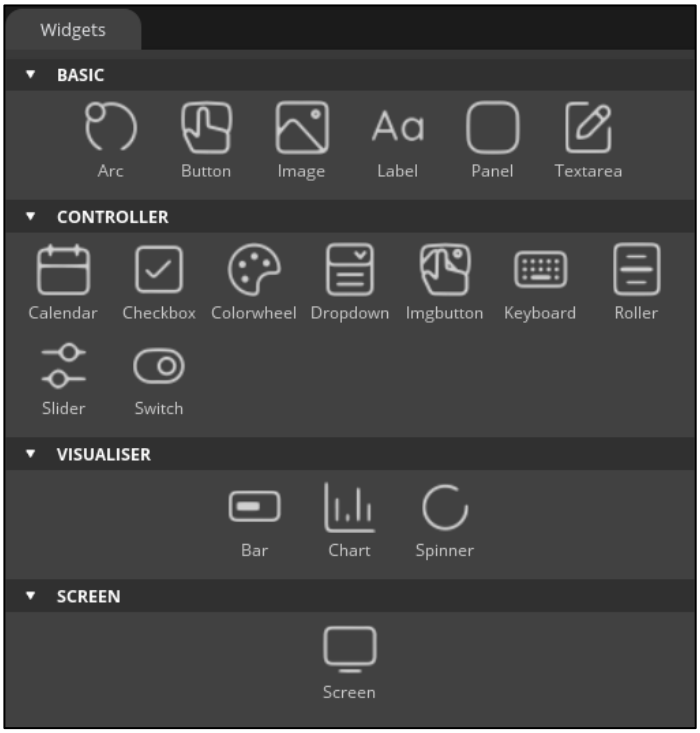


图 47.1.9 小部件板块

从上图可知，SquareLine Studio 软件只有支持大部分的小部件。

- **可编辑视图板块**

可编辑视图类似于 LCD 屏幕。可通过滚动滚轮方式放大或缩小可编辑视图。我们一般点击小部件板块的小部件就可以加载到可编辑视图当中，用户可以使用鼠标移动小部件的位置属性，该板块位于 SquareLine Studio 软件顶部中间位置，如图 47.1.10 所示：

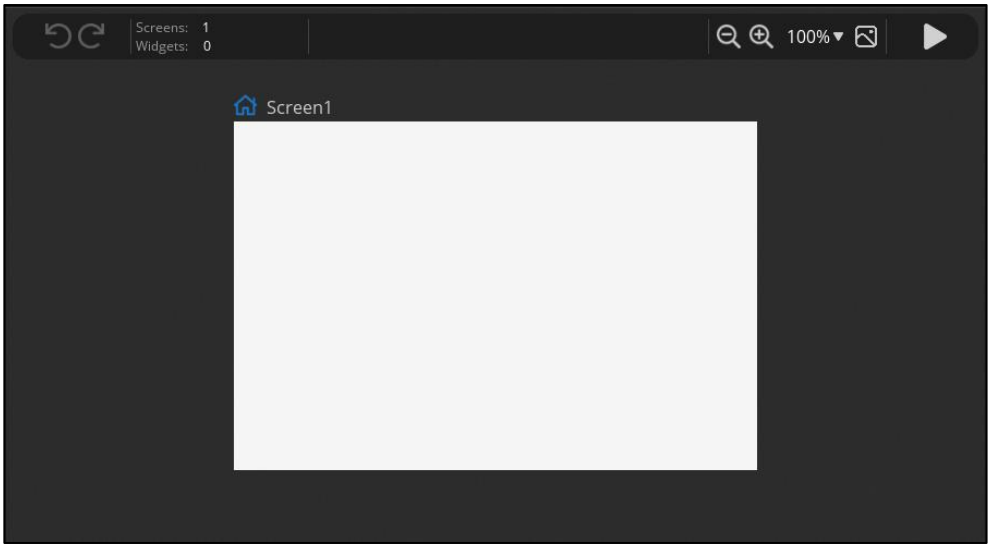


图 47.1.10 可编辑视图板块

注意：上图的右上角的图标为模拟启动按键。

- **资源和控制台板块**

资源和控制台板块是由资源管理和控制台管理部分组成，资源管理主要添加 UI 所需的图片等资源，而控制台管理部分主要与 LVGL python 有关，这里我们就不需要讲解控制台管理部分，该板块位于 SquareLine Studio 软件底部中间位置，如图 47.1.11 所示：



图 47.1.11 资源和控制台板块

从上图可知，在资源管理界面下可点击右下角的“ADD FILE INIO ASSETS”选项，当按下这个选项时，可添加相应的图片文件到资源管理。

- **事件板块**

事件板块是针对所有小部件的板块，如果一个小部件触发事件时，可在事件回调函数执行相应的动作，比如一个 BUTTON 按键部件按下时切换屏幕背景颜色。该板块位于 SquareLine Studio 软件右下角区域中，如图 47.1.12 所示：

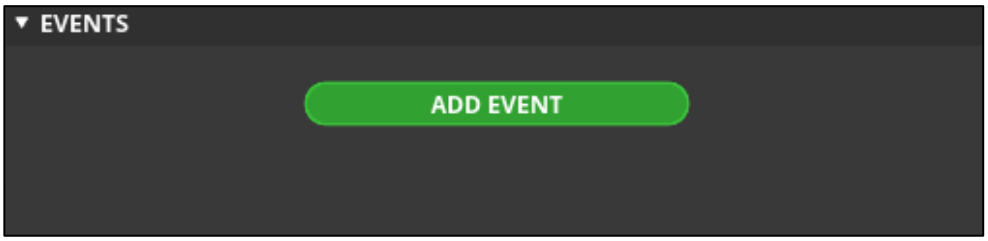


图 47.1.12 事件板块

当用户在可编辑视图中点击一个小部件，然后点击上图中的“ADD EVENT”添加该部件的事件，如图 47.1.13 所示：

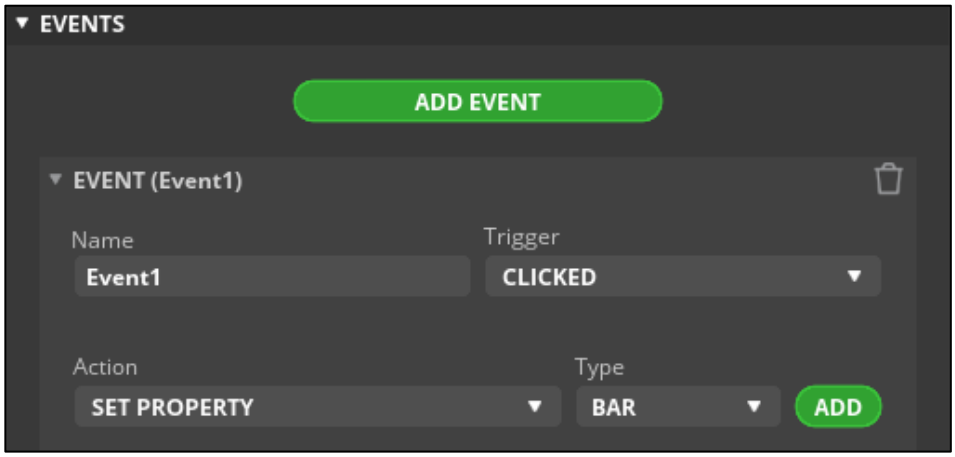


图 47.1.13 添加一个事件

下面我们来讲解一下上图的包含的内容，“Name”标签下的文本框是设置事件的名称，也就是说：生成代码时的事件函数名称；“Trigger”标签下的文本框是设置事件的类型，例如点击、释放、长按或者短按等事件类型；“Action”标签下的文本框是设置活动类型，比如改变屏幕、添加圆弧、设置透明度等类型，我们一般设置“SET PROPERTY”表示包含所有类型；“Type”标签下的文本框是设置触发部件，比如操作圆弧、下拉列表等部件。

如果我们选择了“Action”和“Type”文本框的内容且按下“ADD”选项，则弹出一个界面让用户添加相应的参数，如图 47.1.14 所示：

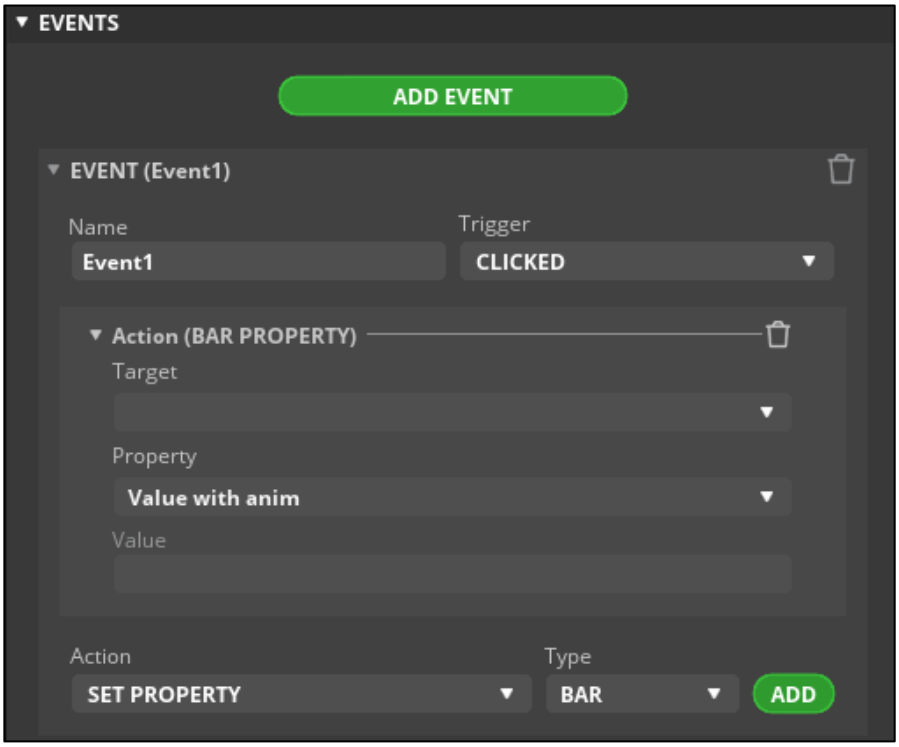


图 47.1.14 添加一个活动

“Target”标签下的文本框表示需要动作的进度条部件，“Property” 标签下的文本框表示修改该部件的数值是否使用动画形式，这里只有两个选项，一个是没有动画形式，另一个是动画形式，“Value” 标签下的文本框表示数值添加的步长大小。

- **样式属性及历史和字库管理板块**

样式属性及历史和字库管理板块是由样式属性、历史以及字库管理组成。样式属性：主要设置小部件的样式属性，每一个部件所需要设置的样式都不一样，大家可以参考官方网址：https://docs.squareline.io/docs/dev_env/widgets 定义相关样式属性；历史记录：主要表示添加操作历史记录，这个历史记录可以返回到过去的操作，这个功能类似于 PS 的历史记录；字库管理主要创建自定义的字库提供给 UI 界面使用。这个板块位于 SquareLine Studio 软件右上角区域中，如图 47.1.15 所示：

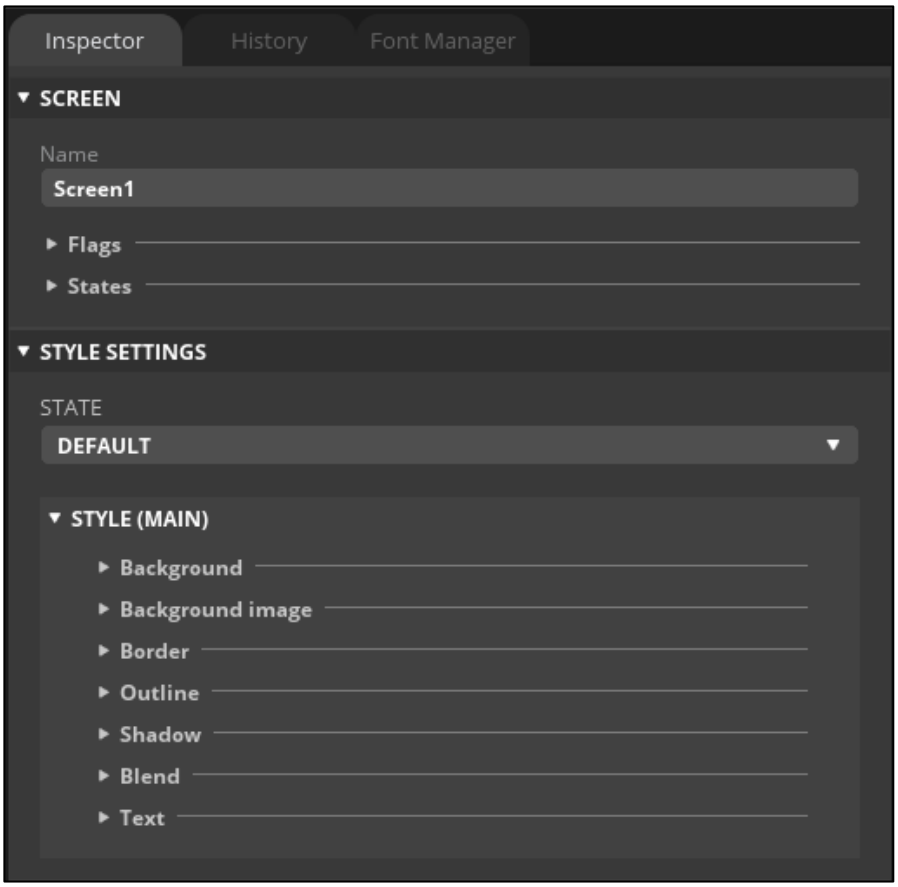


图 47.1.15 样式属性及历史和字库管理板块

上图中，“STYLE SETTINGS”下的内容请参考官方的网址，因为每一个部件设置的样式属性都不一样，该网址为：https://docs.squareline.io/docs/dev_env/widgets；上图的 SCREEN 下的内容是所有部件都是一样的，“Name”文本框表示部件的名称；“Flags”表示要添加还是清除标志；“States”表示对部件添加哪些状态。

47.2 SquareLine Studio 实验

前面的章节我们已经很讲解 SquareLine Studio 软件各个板块的作用以及相关知识，下面我们就以一个简单的实例来使用 SquareLine Studio 软件，这个实例的实现功能是创建两个屏幕，然后这两个屏幕都创建一个 button 按键部件，其中它们自己的按键可切换到另一个屏幕当中。

下面我们就分为价格步骤来实现这个功能，这些步骤如下所示：

第一步：在 Screen1 屏幕下添加 Button 按键，该部件添加方法直接在小部件板块点击 Button 部件即可，如图 47.2.1 所示：

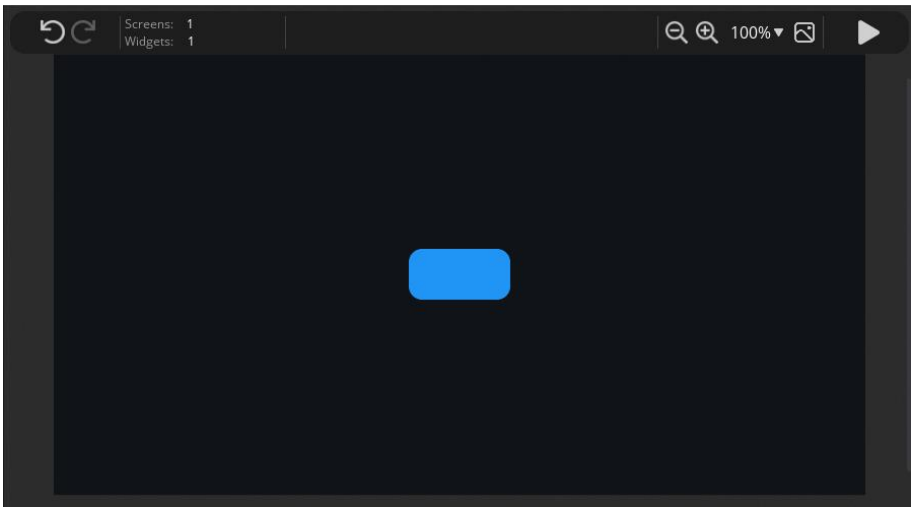


图 47.2.1 在 Screen1 添加一个 Button 部件

第二步：点击小部件板块添加另一个屏幕 Screen2，如图 45.2.2 所示：



图 47.2.2 添加 Screen2 屏幕

为了区别，我们把 Screen2 屏幕设置背景颜色为红色，如图 47.2.3 所示：

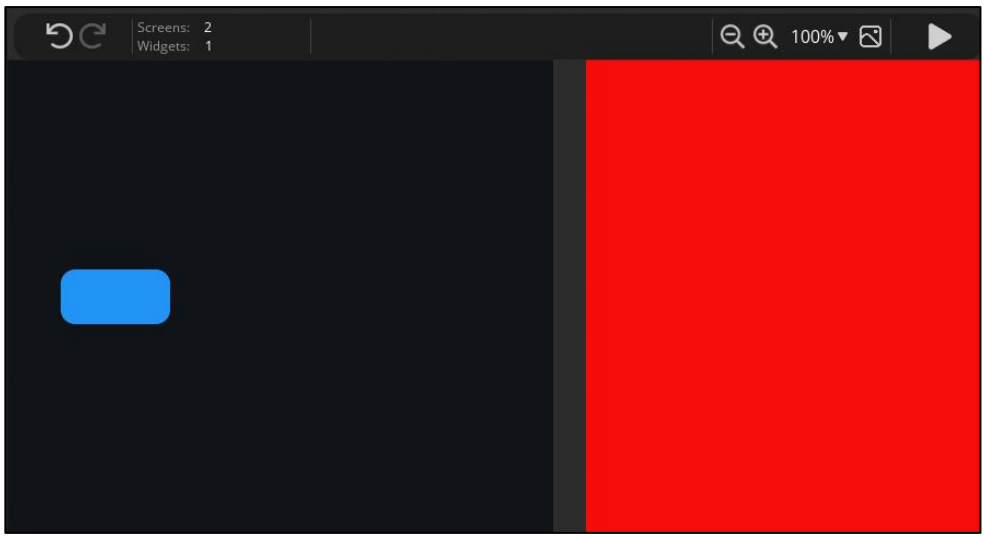


图 47.2.3 设置 Screen2 屏幕背景为红色

Screen2 屏幕背景为红色的设置方法这里我们就无需讲解了，直接操作样式属性管理板块即可。

第三步：在 Screen2 屏幕添加 Button 按键部件，如图 47.2.4 所示：

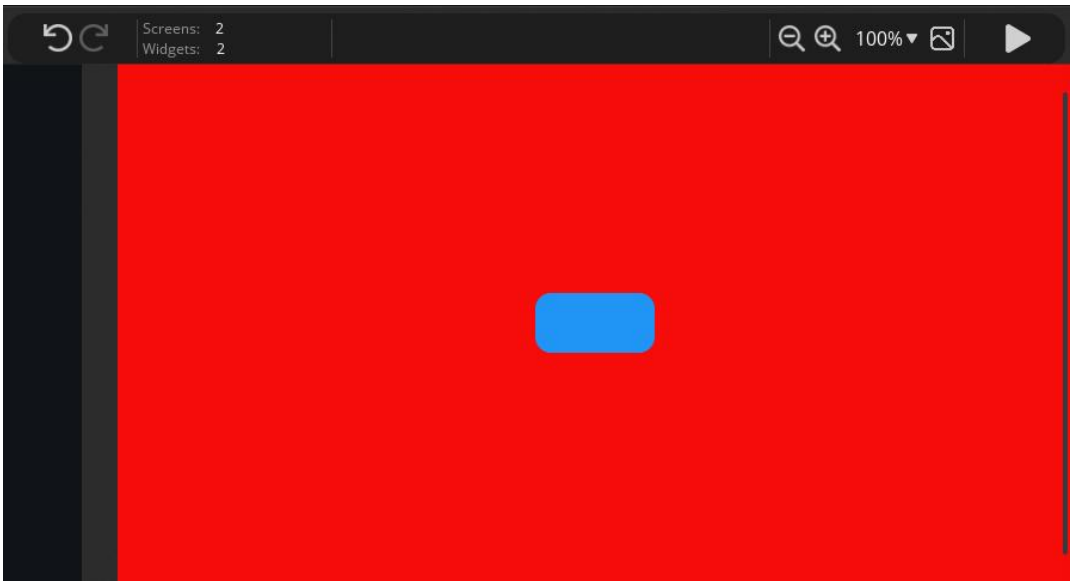


图 47.2.4 在 Screen2 屏幕上添加 Button 按键

第四步：添加 Screen1 的 Button 按键的事件，该事件类型(Trigger)为 CLICKED，活动(Action)为 CHANGE SCREEN，如图 47.2.5 所示：

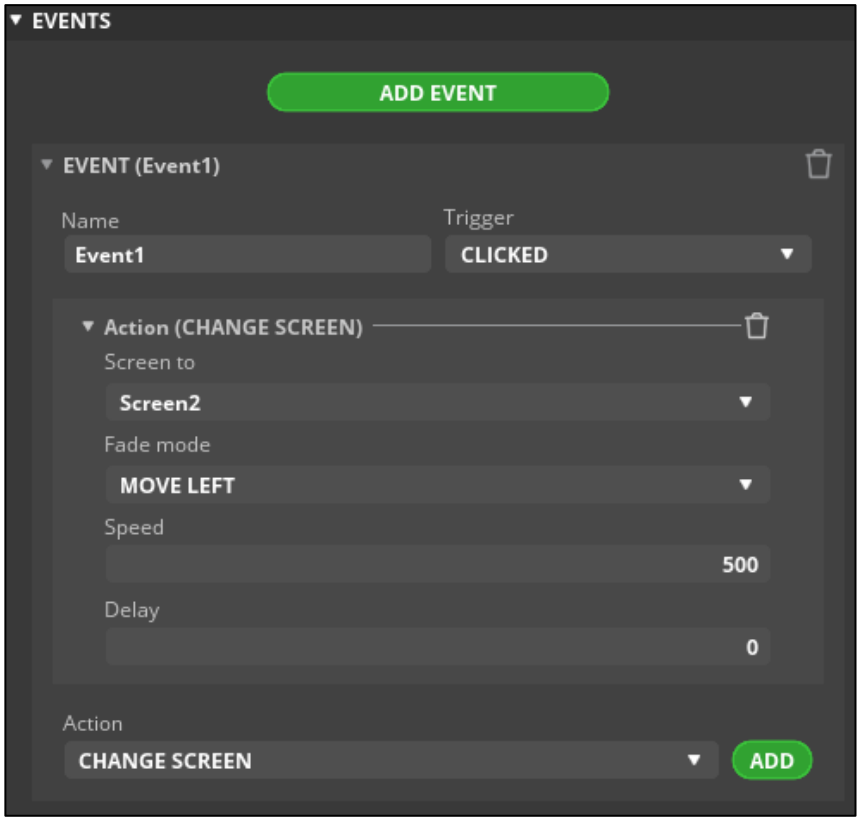


图 47.2.5 添加事件

从上图可知：“Screen to”标签下的文本框表示要切换的目标屏幕，这里我们选择 Screen2，“Fade mode” 标签下的文本框表示切换的方式，这里我们选择往左移动，“Speed” 标签下的文本框表示切换的速度，“Delay” 标签下的文本框表示切换的时间。

第五步：添加 Screen2 的 Button 按键的事件，添加方法与第四步一样，注意先点击 Buttonb 部件才点击添加事件，我们在图 47.2.5 中的“Screen to”标签下的文本框选择 Screen1 即可。

第六步：点击可编辑视图右上角的启动图标，如图 47.2.6 所示：

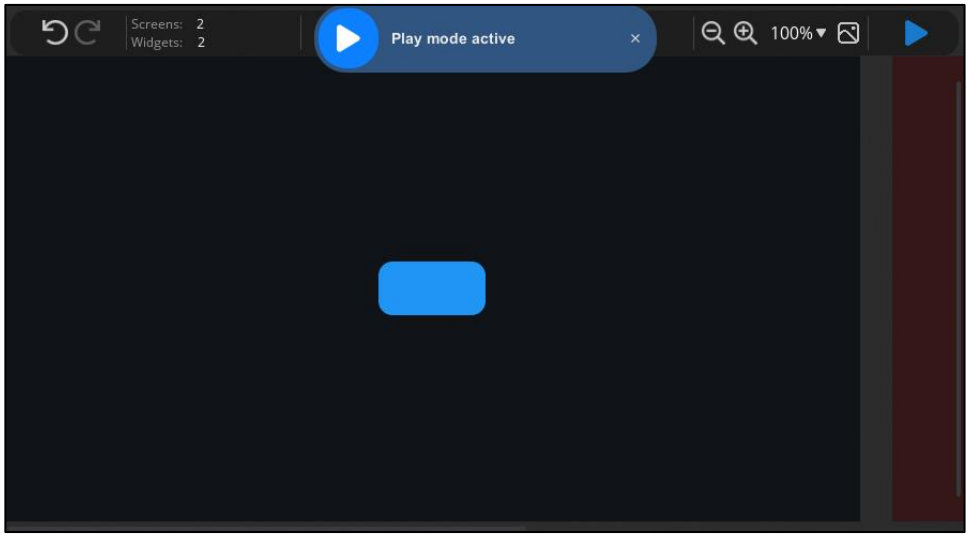


图 47.2.6 启动模拟

当我们点击 Screen1 中的 Button 按键时，系统自动往左切换到 Screen2 屏幕中，当我们点击我们点击 Screen2 中的 Button 按键时，系统自动往左切换到 Screen1 屏幕中。

47.3 SquareLine Studio 移植到工程

前面我们已经讲解了 SquareLine Studio 软件可以导出 C 代码，这些 C 代码可以移植到我们工程当中，下面我们分几个步骤来讲解：

第一步：点击表头菜单下的 Export 菜单，找到“Export file”选项并点击生成代码。

第二步：打开工程保存的路径，我们会发现四个文件，这些文件就是移植到工程中的文件，如图 47.3.1 所示：

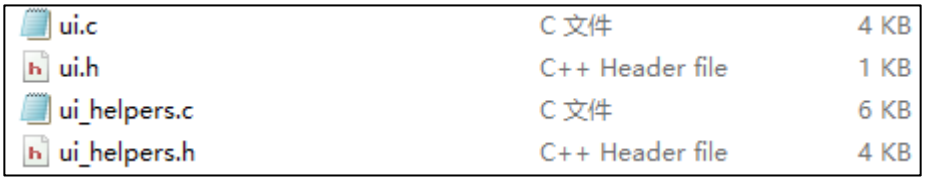


图 47.3.1 导出的代码

第三步：把这些文件复制并拷贝到目标工程/ Middlewares/LVGL/GUI_APP 路径下。

第四步，打开工程并添加到工程当中，如图 47.3.2 所示：

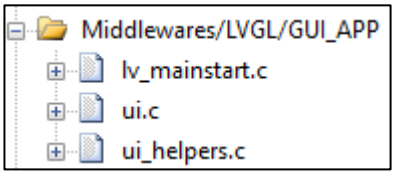


图 47.3.2 添加 UI 源码

第五步：在 ui_helpers.h 以及 ui.h 文件把#if __has_include("lvgl.h")去除，如下所示：

```
#if __has_include("lvgl.h")
    #include "lvgl.h"
#else
    #include "lvgl/lvgl.h"
#endif
```

我们把上述源码的条件编译语句去除，修改为如下源码：

```
#include "lvgl.h"
```

第六步：在 lv_mainstart 函数下调用 ui_init 函数初始化 UI 界面即可。

经过上面的六步，我们就可以完成 SquareLine Studio 设置的 UI 并导出代码移植到工程当中，大家也可以尝试一下 SquareLine Studio 软件，不得不说这个软件能提高我们的开发效率。

布局篇

经过前面的学习，笔者相信读者已经对 LVGL 的基础知识以及部件的使用有一定的了解，下面笔者开始介绍布局篇的内容，本来这些内容会在 LVGL 基础知识章节中讲解的，由于布局篇的内容会涉及到部件的使用，所以笔者把这一篇的内容放在最后讲解。

第四十八章 Flex 布局

Flexible Box 模型，通常被称为 flexbox，是一种一维的布局模型。它给 flexbox 的子元素之间提供了强大的空间分布和对齐能力。

本章我们分为几个小节来讲解：

48.1 Flex 初探

48.2 Flex 相关知识

48.3 Flex 布局的实验

48.1 Flex 初探

Flex 布局，是一种可以简便、完整、响应式地实现各种页面布局，它是 CSS 的一个重点应用。LVGL 从 V8 版本开始已经支持 CSS 的 Flex 和 Grid 布局，前面笔者也说过： flexbox 是一种一维的布局，是因为一个 flexbox 一次只能处理一个维度上的元素布局，一行或者一列，但是 Grid 布局是一种二维布局，它可以同时处理行和列上的布局，本章主要讲解 Flex 布局知识，而 Grid 布局知识笔者会在下一个章节讲解。

简单来说：它可以将对象排列成行或列，并处理环绕，调整对象和行/列之间的间距，处理增长以使对象填充剩余空间的最小/最大宽度和高度。

48.2 Flex 相关知识

48.2.1 启用 Flex 布局

在 lv_conf.h 文件中把 LV_USE_FLEX 配置项置 1 即可启用 Flex 布局。

48.2.2 Flex 条文

① 轨道：行或列。
② 主轴：行或列，对象放置的方向。
③ 交叉轴：垂直于主轴。
④ 换行/换列：如果轨道中没有更多空间，则开始新轨道
⑤ 增长：如果设置在一个项目上，它将增长以填充轨道上的剩余空间。
⑥ 间隙：行和列或轨道上的对象之间的空间。

当使用 flex 布局时，首先想到的是两根轴线，它们分别为主轴和交叉轴。主轴是 flex 方向，另一根轴垂直于它。我们使用 flexbox 的所有属性都跟这两根轴线有关，所以有必要在一开始首先理解它。下面笔者带读者来讲解一下主轴和交叉轴的区别：

1. 主轴：是定义对象的放置方向的，在 LVGL 中，它一般可取八个值，如下表所示：

主轴的定义	描述
LV_FLEX_FLOW_ROW	将子类们排成一行并不换行
LV_FLEX_FLOW_ROW_WRAP	将子类们排成一行并换行
LV_FLEX_FLOW_ROW_REVERSE	将子类们排成一行并不换行且顺序相反
LV_FLEX_FLOW_ROW_WRAP_REVERSE	将子类们排成一行并换行且顺序相反
LV_FLEX_FLOW_COLUMN	将子类们排成一列并不换行
LV_FLEX_FLOW_COLUMN_WRAP	将子类们排成一列并换行

LV_FLEX_FLOW_COLUMN_REVERSE	将子类们排成一列并不换行且顺序相反
LV_FLEX_FLOW_COLUMN_WRAP_REVERSE	将子类们排成一列并换行且顺序相反

表 48.2.2.1 Flex 布局主轴配置项

从上表可知：如果我们设置 LV_FLEX_FLOW_ROW~ LV_FLEX_FLOW_ROW_WRAP_REVERSE 为主轴配置项时，容器的主轴将沿着 inline 方向延伸，如下图所示：

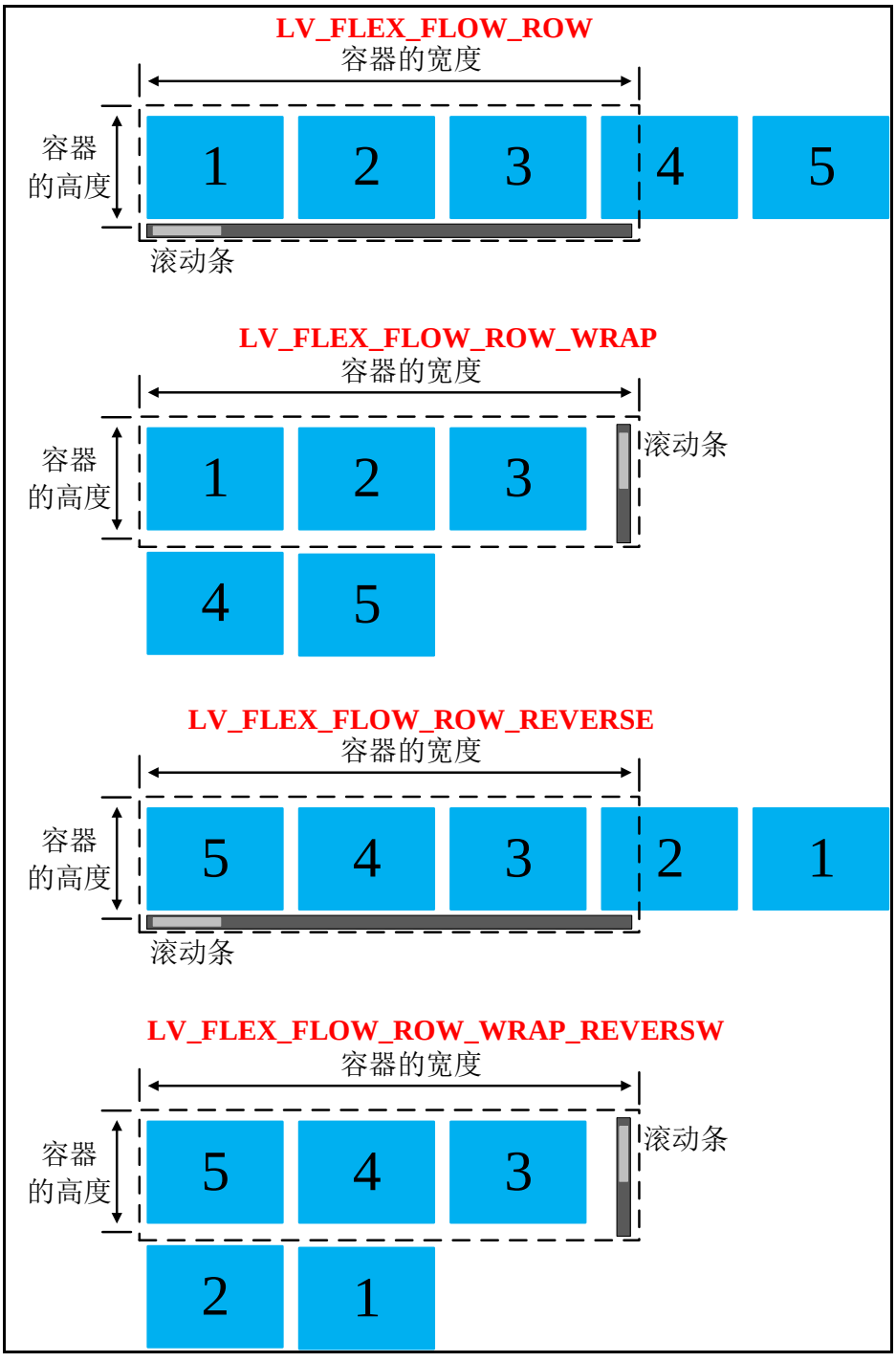


图 48.2.2.1 主轴(行)设置示意图

从上图可知：容器就是绘画在显示屏上的部件，如果有些子类超出了容器范围，则系统自动使能滚动条。值得注意的是：带有 WRAP 的配置项会把超出容器的子类以换行的形式添加在第二行中，例如上图中的第二和第四的示意图；带有 REVEESW 的配置项会把所有子类以翻转的顺序排列，例如上图中的第三和第四的示意图。

从上表可知：如果我们设置 LV_FLEX_FLOW_COLUMN ~ LV_FLEX_FLOW_COLUMN_WRAP_REVERSE 为主轴配置项时，容器的主轴会沿着上下方向延伸--也就是 block 排列的方向，如下图所示：

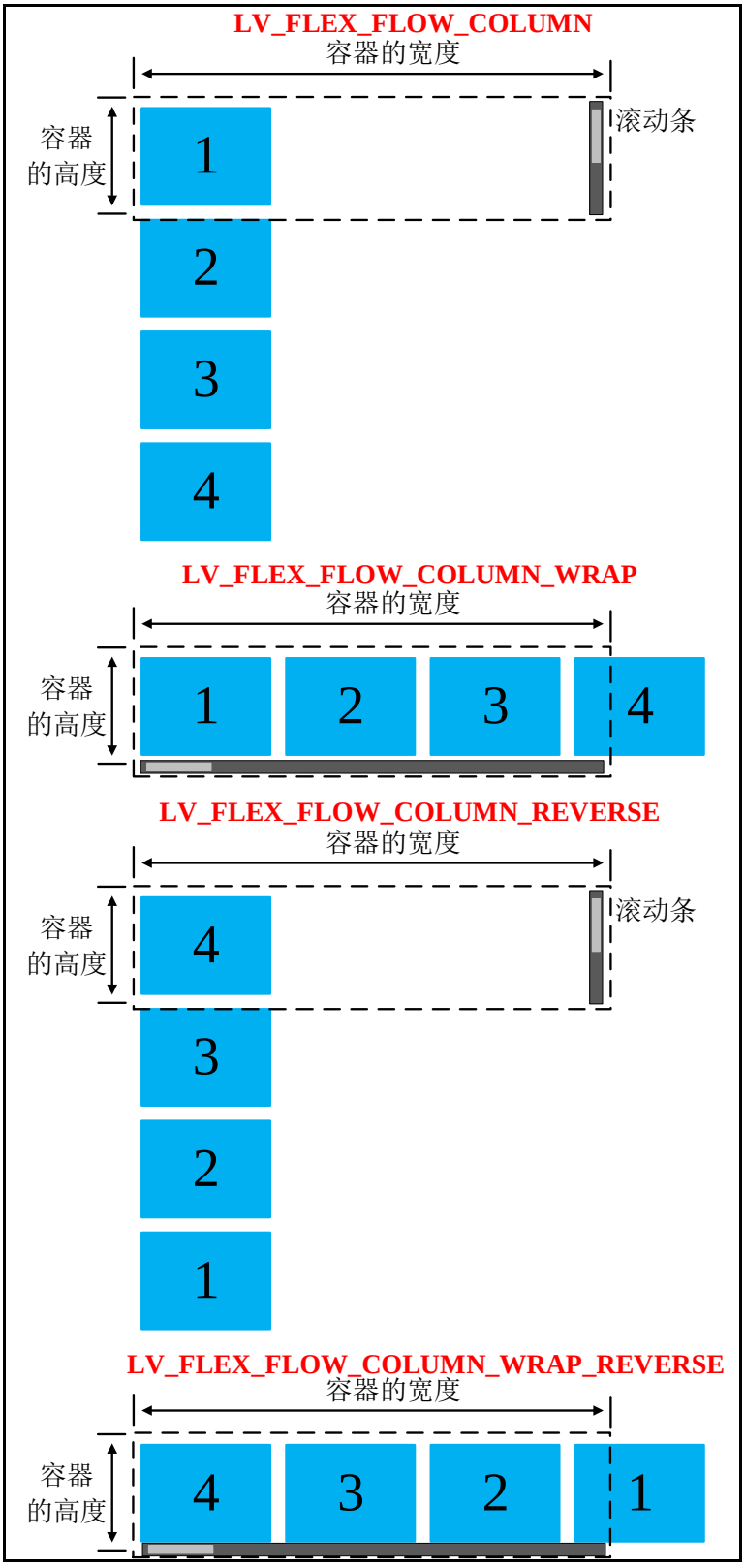


图 48.2.2.2 主轴(列)设置示意图

从上图可知：容器就是绘画在显示屏上的部件，如果有些子类超出了容器范围，则系统自动使能滚动条。值得注意的是：带有 WRAP 的配置项会把超出容器的子类以换列的形式添加在第二列中，例如上图中的第二和第四的示意图；带有 REVEESW 的配置项会把所有子类以翻转的顺序排列，例如上图中的第三和第四的示意图。

2. 交叉轴：交叉轴垂直于主轴，所以如果你的 flex-direction (主轴) 设成了 ROW 的话，交叉轴的方向就是沿着列向下的，如下图所示：

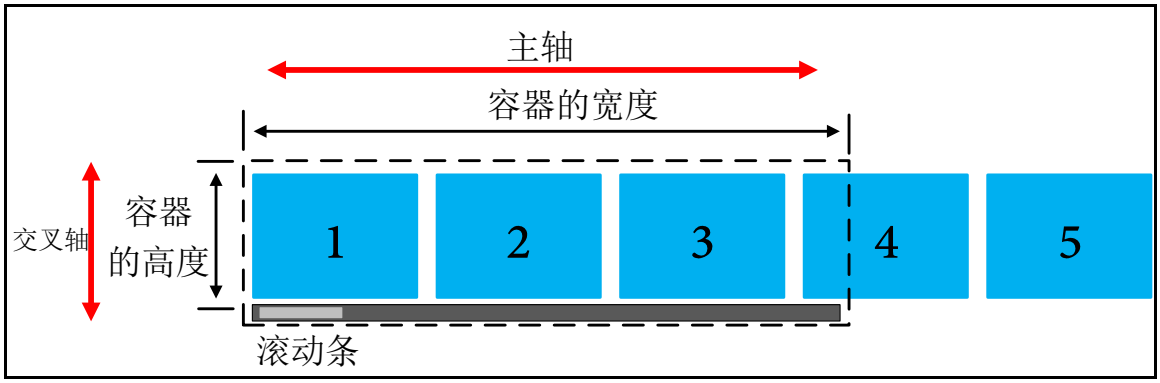


图 48.2.2.3 列的交叉主轴

如果你的 flex-direction (主轴) 设成了 COLUMN 的话，交叉轴就是水平方向，如下图所示：

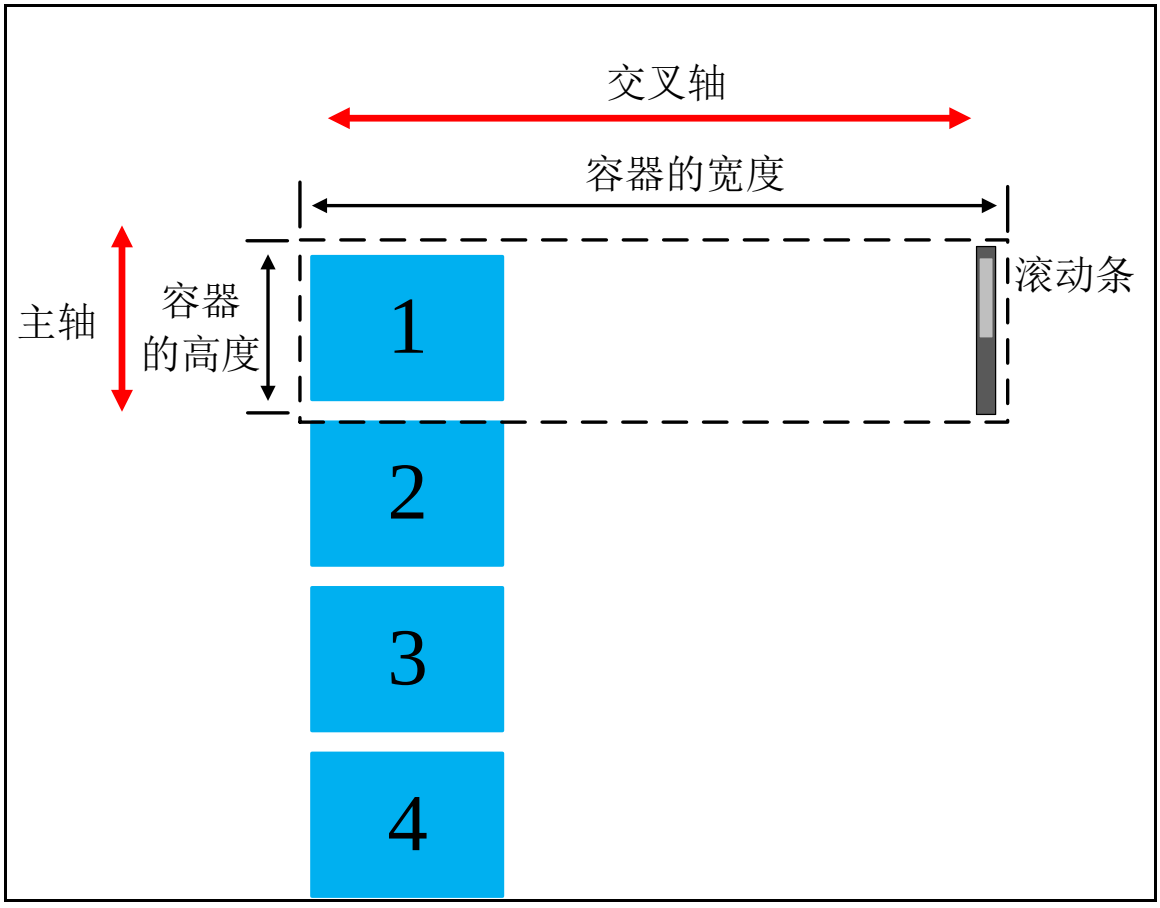


图 48.2.2.4 行的交叉主轴

至此，笔者已经讲解了 Flex 多个条文的作用，例如上面的主轴和交叉主轴以及换行/换列等知识，剩下的条文我们会在下面的小节来讲解。

48.2.3 对象如何使用 Flex 布局

在 LVGL 中，使用 Flex 布局有两种方式，第一种是使用 lv_obj_set_layout 函数方法设置对象启用 Flex 布局，第二种是使用 lv_obj_set_flex_flow 函数启用 Flex 布局。下面笔者以一个简单实例来讲解本小节的内容，如下所示：

```
void lv_mainstart(void)
{
    /* 第一种方式 */
    static lv_style_t style;
    lv_style_init(&style);
    lv_style_set_flex_flow(&style, LV_FLEX_FLOW_ROW_WRAP);
    /* 设置 Flex 布局 */
    lv_style_set_layout(&style, LV_LAYOUT_FLEX);

    lv_obj_t* cont = lv_obj_create(lv_scr_act());
    lv_obj_set_size(cont, 300, 220);
    lv_obj_add_style(cont, &style, 0);

    /* 第二种方式 */
    lv_obj_t* cont_row = lv_obj_create(lv_scr_act());
    lv_obj_set_size(cont_row, 300, 220);
    lv_obj_align(cont_row, LV_FLEX_FLOW_ROW, 0, 5);
    /* 设置 Flex 布局 */
    lv_obj_set_flex_flow(cont_row, LV_FLEX_FLOW_ROW_WRAP);
    lv_obj_align_to(cont_row, cont, LV_ALIGN_OUT_RIGHT_TOP, 20, 0);

    uint32_t i;

    for (i = 0; i < 10; i++) {
        lv_obj_t* obj;
        lv_obj_t* label;

        /* 在 cont 容器中添加 btn 对象 */
        obj = lv_btn_create(cont);
        lv_obj_set_size(obj, 100, LV_PCT(100));

        label = lv_label_create(obj);
        lv_label_set_text_fmt(label, "Item: %u", i);
        lv_obj_center(label);

        /* 在 cont_row 容器中添加 btn 对象 */
        obj = lv_btn_create(cont_row);
        lv_obj_set_size(obj, 100, LV_PCT(100));
```

```
        label = lv_label_create(obj);
        lv_label_set_text_fmt(label, "Item: %u", i);
        lv_obj_center(label);
    }
}
```

上述源码非常简单，它们分别使用不同的方式启用 Flex 布局，而它们的布局类型都是 LV_FLEX_FLOW_ROW_WRAP 配置项，这个配置项笔者在前面也讲解过，这里我们无需重复讲解。把代码下载到开发板中或者在 PC 机模拟可得到以下效果图：

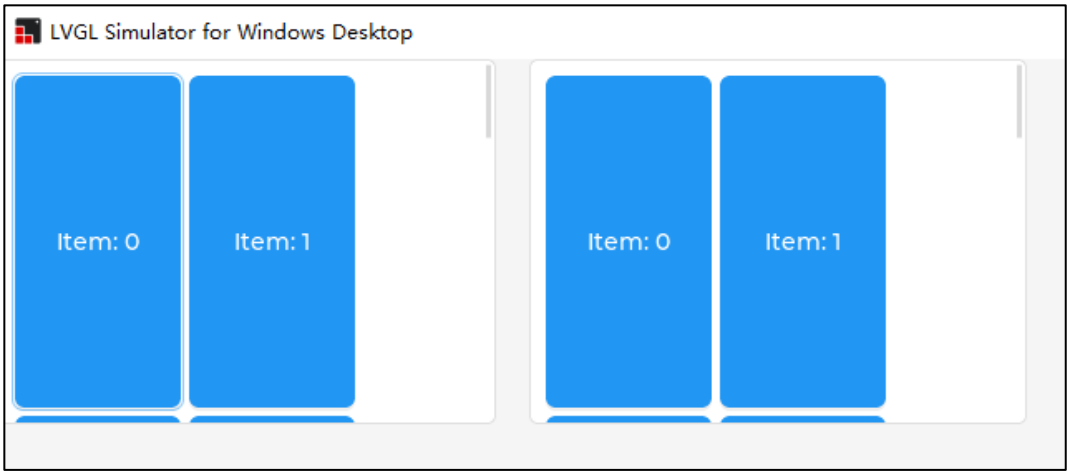


图 48.2.3.1 启用 Flex 的方式

48.2.4 Flex 对齐

Flexbox 的一个关键特性是能够设置 flex 元素沿主轴方向和交叉轴方向的对齐方式以及它们之间的空间分配。这里不得不提起 LVGL 提供的一个函数，该函数为 lv_obj_set_flex_align(obj, main_place, cross_place, track_cross_place)，这个函数具有四个形参，下面笔者来讲解一下这四个形参的作用，如下表所示：

形参	描述
obj	指向设置对齐的对象
main_place	主轴上的对象对齐
cross_place	交叉轴上的对象对齐
track_cross_place	行/列交叉轴上的对象对齐

表 48.2.4.1 v_obj_set_flex_align()函数的形参描述

main_place、cross_place 和 track_cross_place 形参的转入值如下表所示：

配置项	描述
LV_FLEX_ALIGN_START	水平方向左上和垂直上方向上（默认）
LV_FLEX_ALIGN_END	水平方向右侧和垂直底部
LV_FLEX_ALIGN_CENTER	居中
LV_FLEX_ALIGN_SPACE_EVENLY	任何两个对象之间的间距（它们的边缘空间相等），但不适用于 track_cross_place 形参
LV_FLEX_ALIGN_SPACE_AROUND	对象在轨道上均匀分布，周围空间相等，但不适用于 track_cross_place 形参
LV_FLEX_ALIGN_SPACE_BETWEEN	对象在轨道上均匀分布：第一个对象在开始行，最后一个项目在结束行，但不适用于 track_cross_place 形参

表 48.2.4.2 对齐配置项

如果 main_place 形参分别传入这六个形参，那么容器中的子对象如何布局呢？下面笔者使用一个示意图来描述这个功能，如下所示：

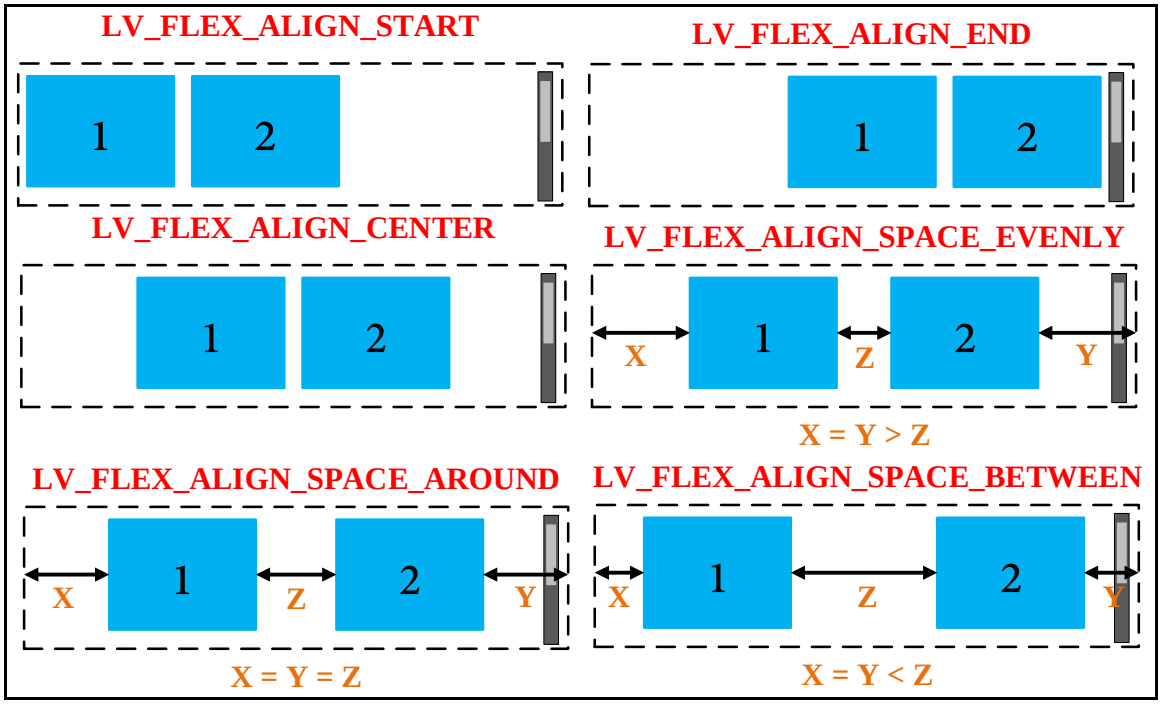


图 48.2.4.1 main_place 形参传入的配置项示意图

注意：上图的效果图是以 LV_FLEX_FLOW_ROW_WRAP 类型来绘画的，其他类型请读者自行研究。

track_cross_place 形参只能传入 LV_FLEX_ALIGN_START、LV_FLEX_ALIGN_END 和 LV_FLEX_ALIGN_CENTER 配置项，它们分别代表显示的位置。为了更好的理解这个知识，笔者花费一点点时间绘画一个示意图来让读者了解这个知识，如下图所示：

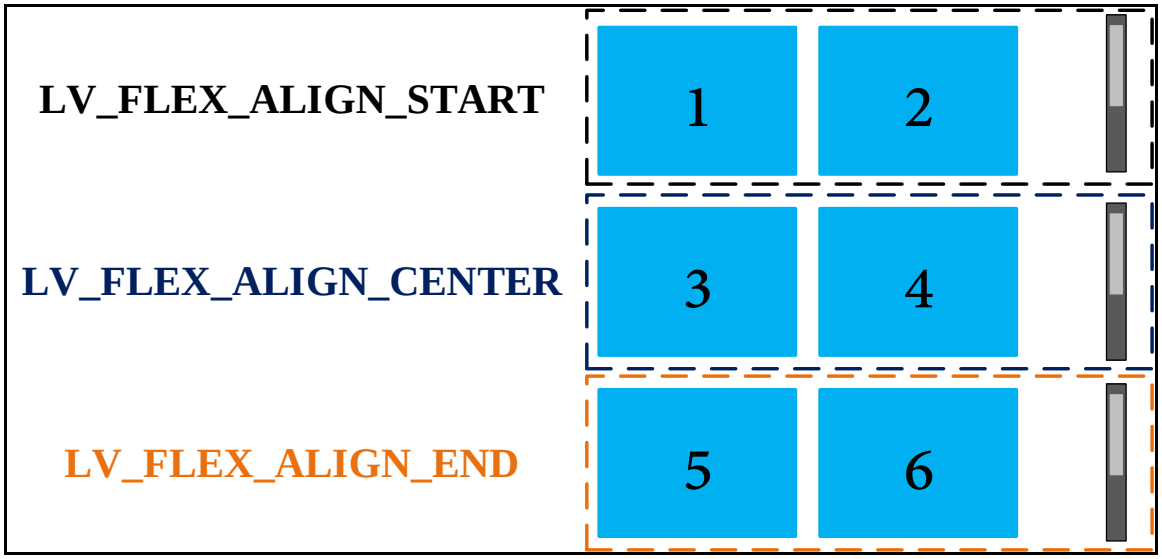


图 48.2.4.2 track_cross_place 形参传入的配置项示意图

从上图可知：如果 track_cross_place 形参设置为 LV_FLEX_ALIGN_START，则该容器显示子类 1 和 2；如果 track_cross_place 形参设置为 LV_FLEX_ALIGN_END，则该容器显示子类

5 和 6；如果 track_cross_place 形参设置为 LV_FLEX_ALIGN_CENTER，则该容器显示子类 3 和 4。

cross_place 形参一般在子类对象具有不同的高度时会起作用。

48.2.5 flex-grow 属性

flex-grow 若被赋值为一个正整数，它沿主轴方向增长尺寸。这会使该元素延展，并占据此方向轴上的可用空间（available space）。如果有其他元素也被允许延展，那么它们会各自占据可用空间的一部分。关于这个 grow 属性，LVGL 提供了 lv_obj_set_flex_grow 函数来设置，该函数具有两个形参，第一个形参指向要增长的对象，而第二个形参表示增长的倍数，例如有 400 像素剩余空间和 3 个对象增长，它们设置分别为 1、1 和 2，所以它们的宽度分别为 100，100 和 200 像素，如下图所示：

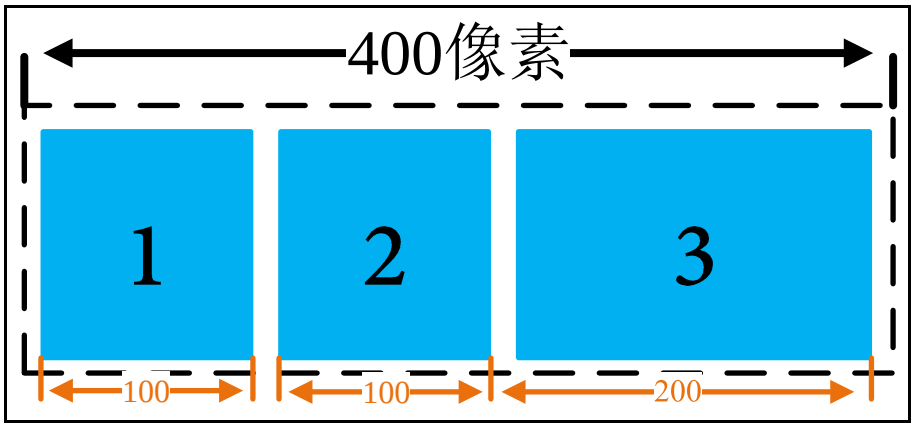


图 48.2.5 grow 属性的设置

注意：lv_obj_set_flex_grow 函数的第二个形参为 0 时禁用 grow 属性。

48.2.6 Flex 条文的样式函数

在样式中，所有与 Flex 相关的值都是底层的样式属性，这些与 Flex 相关的样式属性函数如下表所示：

函数	描述
lv_style_set_flex_flow()	设置 Flex 流动布局
lv_style_set_flex_main_place()	设置主轴
lv_style_set_flex_cross_place()	设置交叉轴
lv_style_set_flex_track_place()	设置轨道轴
lv_style_set_flex_grow()	设置增长

表 48.2.6.1 与 Flex 相关样式属性函数

这些函数一般用在样式启动 Flex 布局程序当中。注意：也可以使用本地样式属性函数设置。

48.2.7 Flex 间隙

Flex-gap 属性用来设置元素列之间的间隔（gutter）大小，例如修改对象之间的最小空间，可以在 flex 容器样式上设置以下属性：

① pad_row：设置行之间的填充。

② pad_column：设置列之间的填充。

如果读者不希望对象之间有任何填充，可调用 lv_style_set_pad_column(&row_container_style,0)函数设置。

48.3 Flex 布局的实验

关于 Flex 布局的实验，大家可以参考 LVGL 官方提供的实验，该实验的路径为 lvgl-release-v8.2\examples\ layouts\flex 文件夹下，该文件夹具有六个实例提供给读者学习。

第四十九章 Grid 布局

网格布局引入了二维网格布局系统，可用于布局页面主要的区域布局或小型组件。本章主要介绍 LVGL 网格布局的使用。

本章我们分为几个小节来讲解：

49.1 Grid 初探

49.2 Grid 相关知识

49.3 Grid 布局的实验

49.1 Grid 初探

Grid 网格是一组相交的水平线和垂直线，它定义了网格的列和行。我们可以将网格元素放置在与这些行和列相关的位置上。简单来说：Grid 网格可以将对象排列到具有行或列（轨道）的二维“表”中，该对象可以跨越多个列或行。轨道的大小可以设置为像素、LV_GRID_CONTENT 或“空闲单元”(FR) 以按比例分配空闲空间。

49.2 Grid 相关知识

49.2.1 启用 Grid 布局

在 lv_conf.h 文件中把 LV_USE_GRID 配置项置 1 即可启用 Flex 布局。

49.2.2 Grid 条文

① 轨道(tracks)：行或列。
② 空闲单元(FR)：如果在轨道上设置了大小，FR 它将增长以填充父级的剩余空间。
③ 间隙(gap)：行和列或轨道上的项目之间的空间。

49.2.3 对象如何使用 Grid 布局

在 LVGL 中，需要读者使用 lv_obj_set_layout 函数设置对象启用 Grid 布局。

49.2.3 Grid 描述符

Grid 描述符是用来描述行/列的数量以及单元格的像素，它一般使用两个数组来描述网格的行数和列数，例如声明 2 个数组，它们分别描述行和列的个数以及大小，注意：这些数组的最后一个元素必须是 LV_GRID_TEMPLATE_LAST 配置项，如下源码所示：

```
/* 2 列 100 和 400 ps 宽度 */
static lv_coord_t column_dsc[] = {100, 400, LV_GRID_TEMPLATE_LAST};
/* 3 个 100 像素高的行*/
static lv_coord_t row_dsc[] = {100, 100, 100, LV_GRID_TEMPLATE_LAST};
```

从上述源码可知：column_dsc 和 row_dsc 数组创建了 3 行 2 列的网格，网格内的单元格大小由这些数组内的数值决定。下面笔者使用一个示意图来讲解上述的内容，如下所示：

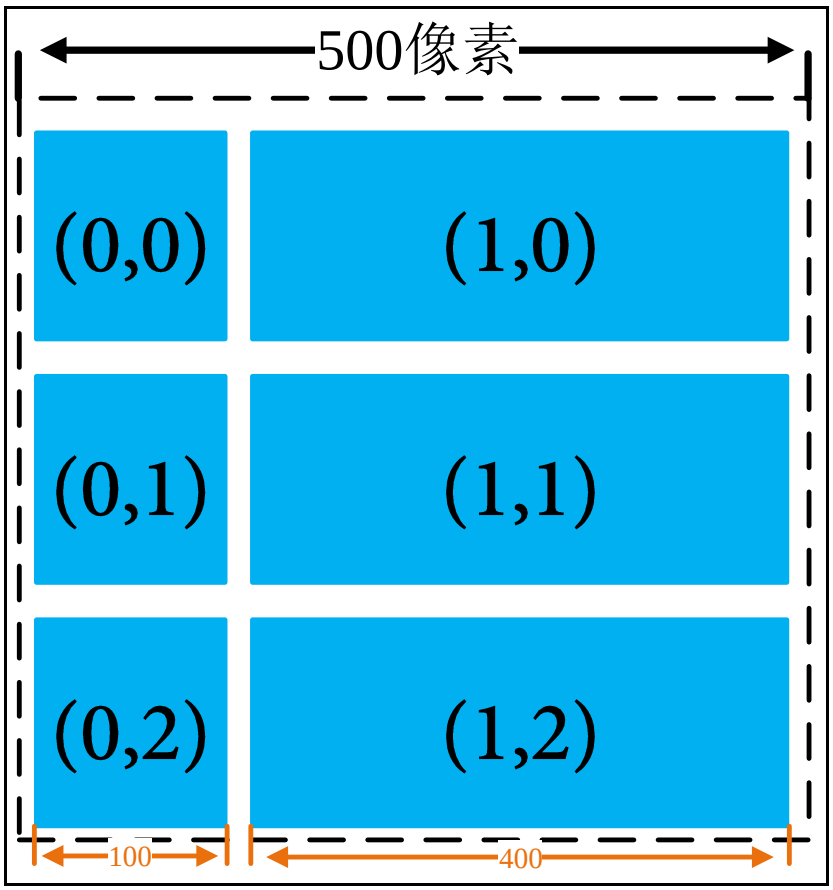


图 49.2.3 创建 3 行 2 列的 Grid 网格

上述的数组是调用了 lv_obj_set_style_grid_column_dsc_array 和 lv_obj_set_style_grid_row_dsc_array 函数创建网格的。

当然除了以像素为单位的简单设置之外，我们还可以使用两个特殊值：

① LV_GRID_CONTENT：将宽度设置为此轨道上最大的子类。

② LV_GRID_FR(X)：告诉该轨道应使用剩余空间的哪一部分，数值越大，空间越大。

49.2.4 添加 Grid 对象

默认情况下，我们的子项不会自动添加到网格中。这些子类需要读者手动添加到单元格中，这个添加方式需要读者调用 lv_obj_set_grid_cell 把子类对象添加到指定网格位置，下面笔者分别讲解一下这个函数的形参到底如何传入数值，如下表所示：

形参	描述	
child	添加的对象	
column_pos	列索引	
column_align row_align	列对齐/行对齐	
	对齐的配置项	
	LV_GRID_ALIGN_START	水平方向左上和垂直上方向上（默认）
	LV_GRID_ALIGN_END	水平方向右侧和垂直底部
	LV_GRID_ALIGN_CENTER	居中
row_pos	行索引	
column_span row_span	起始单元格开始涉及多少个轨道(默认设置为 1)	

表 49.2.4.1 lv_obj_set_grid_cell()函数形参描述

下面笔者使用一个简单的实例来讲解本小节的内容，如下所示：

```
void lv_mainstart(void){
    static lv_coord_t col_dsc[] = { 70, 70, 70, LV_GRID_TEMPLATE_LAST };
    static lv_coord_t row_dsc[] = { 50, 50, 50, LV_GRID_TEMPLATE_LAST };
    /* 创建一个容器 */
    lv_obj_t* cont = lv_obj_create(lv_scr_act());
    /* 设置网格的列数量 */
    lv_obj_set_style_grid_column_dsc_array(cont, col_dsc, 0);
    /* 设置网格的行数量 */
    lv_obj_set_style_grid_row_dsc_array(cont, row_dsc, 0);
    /* 设置容器大小 */
    lv_obj_set_size(cont, 300, 220);
    lv_obj_center(cont);
    /* 开启 GRID 网格 */
    lv_obj_set_layout(cont, LV_LAYOUT_GRID);
    lv_obj_t* obj;
    /* 创建一个 btn 子类 */
    obj = lv_btn_create(cont);
    /* 把 btn 对象添加到网格(0,0)位置 */
    lv_obj_set_grid_cell(obj, LV_GRID_ALIGN_STRETCH, 0,
                         1,LV_GRID_ALIGN_STRETCH, 0, 1);
    /* 创建一个 switch 子类 */
    obj = lv_switch_create(cont);
    /* 把 btn 对象添加到网格(0,1)位置 */
    lv_obj_set_grid_cell(obj, LV_GRID_ALIGN_STRETCH, 0, 1,
                         LV_GRID_ALIGN_STRETCH, 1, 1);}
```

从上述源码可知：笔者创建了一个名为 cont 容器，在该容器内根据 col_dsc 和 row_dsc 数组创建 3 行 3 列的网格，至此我们调用 lv_obj_set_layout 函数开启 GRID，然后我们在网格的(0,0)位置添加 btn 对象，在网格的(0,1)添加 switch 对象，最后编译代码并把代码下载到开发板中，如图 49.2.4.1 所示：

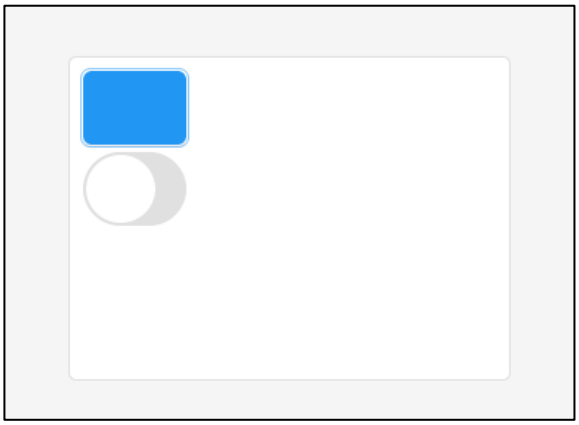


图 49.2.4.1 在指定位置添加对象

49.2.5 Grid 对齐

Grid 对齐与 flex 容器对齐标准是类似的，请参考第四十八章的 48.2.4 小节，注意：Grid 对齐使用的是 lv_obj_set_grid_align 函数并不是 lv_obj_set_flex_align 函数。

49.2.6 Grid 条文的样式函数

在样式中，所有与 Grid 相关的值都是底层的样式属性，这些与 Grid 相关的样式属性函数如下所示：

① GRID_COLUMN_DSC_ARRAY：设置列的描述符数组。
② GRID_ROW_DSC_ARRAY：设置行的描述符数组。
③ GRID_COLUMN_ALIGN：设置列的对象对齐。
④ GRID_ROW_ALIGN：设置列的对象对齐。
⑤ GRID_CELL_X_ALIGN：设置单元格的 X 轴方向对齐。
⑥ GRID_CELL_COLUMN_POS：设置列的单元格位置。
⑦ GRID_CELL_COLUMN_SPAN：设置列的单元格宽度。
⑧ GRID_CELL_Y_ALIGN：设置单元格的 Y 轴方向对齐。
⑨ GRID_CELL_ROW_POS：设置行的单元格位置。
⑩ GRID_CELL_ROW_SPAN：设置行的单元格宽度。

这些函数是使用本地样式来设置 Grid 布局属性。

49.2.7 Grid 间隙

Grid-gap 属性用来设置元素列之间的间隔（gutter）大小，例如修改对象之间的最小空间，可以在 Grid 容器样式上设置以下属性：

① pad_row：设置行之间的填充。
② pad_column：设置列之间的填充。

49.3 Grid 布局的实验

关于 Grid 布局的实验，大家可以参考 LVGL 官方提供的实验，该实验的路径为 lvgl-release-v8.2\examples\ layouts\grid 文件夹下，该文件夹具有六个实例提供给读者学习。