



百问网 LVGL 系列教程 www.100ask.net

发布 1.0

www.100ask.net

三月 08, 2021

1	什么是 Lvgl ?	3
1.1	获取离线教程文档	3
1.2	主要特性	4
1.3	硬件要求	4
1.4	源码布局	5
1.5	LVGL 更新发行规则	5
1.6	LVGL 仓库分支说明	6
1.7	发布周期	6
1.8	版本标签	6
1.9	变更日志	6
1.10	版本兼容	6
2	系统框架	7
3	🔧 建立一个 lvgl 项目	9
3.1	配置文件	10
3.2	初始化 lvgl	11
4	显示接口	15
4.1	显示缓冲区	15
4.2	显示驱动器	16
4.3	相关 APIs	19
5	输入设备接口	21
5.1	输入设备的类型	21
5.2	触摸板, 鼠标或任何指针	22
5.3	触摸板或键盘	22
5.4	编码器	23

5.5	按键	24
5.6	其它功能	25
5.7	相关 API	26
6	心跳	27
7	任务处理器 (Task Handler)	29
8	睡眠管理	31
9	操作系统和中断	33
9.1	任务和线程	33
9.2	中断	33
10	日志记录	35
10.1	日志级别	35
10.2	使用 printf 记录	36
10.3	自定义日志功能	36
10.4	添加日志	37
11	对象 (Objects)	39
11.1	对象的属性 (Attributes)	39
11.2	对象的工作机制	40
11.3	追随原则	40
11.4	屏幕对象	43
11.5	零件 (Parts)	44
11.6	状态-States	44
12	对象层级 (Layers)	45
12.1	创建对象层级顺序	45
12.2	将图层设到前台 (foreground) 展示	46
12.3	顶层和系统层	47
13	事件 (Events)	49
13.1	事件类型	50
13.2	自定义事件包含的数据	52
13.3	手动发送事件	52
14	输入设备 (Input devices)	55
14.1	指针	55
14.2	键盘和编码器	56
14.3	相关 API	57
15	显示 (Displays)	59
15.1	多种显示支持	59

15.2 相关 API	64
16 字体 (Fonts)	65
16.1 Unicode 支持	65
16.2 内置字体	66
16.3 特殊功能	67
16.4 添加新字体	70
16.5 添加新符号	71
16.6 在运行时加载字体	71
16.7 添加新的字体引擎	72
16.8 相关 API	73
17 图片 (Images)	75
17.1 储存图片	75
17.2 变量	75
17.3 文件	76
17.4 色彩格式	76
17.5 添加和使用图像	77
17.6 图像缓存	82
17.7 相关 API	83
18 文件系统 (File system)	85
18.1 添加驱动程序	85
18.2 使用范例	86
18.3 使用图像驱动程序	87
18.4 相关 API	88
19 动画效果 (Animations)	89
19.1 创建动画	89
19.2 动画路径 (path)	91
19.3 速度与时间	91
19.4 删除动画	92
19.5 相关 API	92
20 任务 (Task)	93
20.1 创建一个任务	93
20.2 准备并重置	94
20.3 设定参数	94
20.4 一次行的任务	94
20.5 测量空闲时间	94
20.6 异步调用	95
20.7 相关 API	95
21 画画 (Drawing)	97

21.1	缓冲类型	97
21.2	屏幕刷新机制	98
21.3	蒙版 (遮罩)	99
22	基础对象 (lv_obj)	105
22.1	概述	105
22.2	坐标	106
22.3	父子关系	108
22.4	屏幕	108
22.5	层次	108
22.6	事件处理	109
22.7	部件	109
22.8	状态	109
22.9	风格	110
22.10	属性	110
22.11	保护	111
22.12	组	111
22.13	扩展点击区域	111
22.14	事件	112
22.15	按键	112
22.16	范例	112
22.17	相关 API	113
23	弧 (lv_arc)	145
23.1	概述	145
23.2	零件和样式	145
23.3	用法	146
23.4	事件	146
23.5	按键	147
23.6	范例	147
23.7	相关 API	149
24	进度条 (lv_bar)	155
24.1	概述	155
24.2	零件和样式	155
24.3	用法	156
24.4	事件	156
24.5	按键	156
24.6	范例	156
24.7	相关 API	157
25	按钮 (lv_btn)	159
25.1	概述	159

25.2	零件和样式	159
25.3	用法	160
25.4	可检查	160
25.5	布局和适配	160
25.6	事件	160
25.7	按键	161
25.8	范例	161
25.9	相关 API	165
26	按钮矩阵 (lv_btnmatrix)	171
26.1	概述	171
26.2	零件和样式	171
26.3	用法	172
26.4	事件	173
26.5	按钮	173
26.6	范例	174
26.7	相关 API	175
27	日历 (lv_calendar)	181
27.1	概述	181
27.2	零件和样式	181
27.3	用法	182
27.4	事件	183
27.5	范例	183
27.6	相关 API	185
28	画布 (lv_canvas)	187
28.1	概述	187
28.2	零件和样式	187
28.3	用法	187
28.4	模糊效果	189
28.5	事件	189
28.6	按键	189
28.7	范例	190
28.8	相关 API	193
29	复选框 (lv_cb)	199
29.1	概述	199
29.2	零件和样式	199
29.3	用法	200
29.4	事件	200
29.5	按键	201
29.6	范例	201

29.7	相关 API	202
30	图表 (lv_chart)	203
30.1	概述	203
30.2	零件和样式	203
30.3	用法	204
30.4	事件	207
30.5	按键	207
30.6	范例	207
30.7	相关 API	210
31	容器 (lv_cont)	219
31.1	概述	219
31.2	零件和样式	219
31.3	用法	219
31.4	事件	221
31.5	按键	221
31.6	范例	221
31.7	相关 API	223
32	颜色选择器 (lv_cpicker)	225
32.1	概述	225
32.2	零件和样式	225
32.3	用法	226
32.4	按键	226
32.5	范例	227
32.6	相关 API	228
33	下拉列表 (lv_dropdown)	233
33.1	概述	233
33.2	小部件和样式	233
33.3	用法	234
33.4	事件	235
33.5	按键	235
33.6	范例	236
33.7	相关 API	238
34	量规 (lv_gauge)	239
34.1	概述	239
34.2	小部件和样式	239
34.3	用法	240
34.4	事件	240
34.5	按键	241

34.6	范例	241
34.7	相关 API	243
35	图片 (lv_img)	245
35.1	概述	245
35.2	零件和样式	245
35.3	用法	245
35.4	转换	247
35.5	旋转	247
35.6	事件	248
35.7	按键	248
35.8	范例	248
35.9	相关 API	251
36	图片按钮 (lv_imgbtn)	253
36.1	概述	253
36.2	零件和样式	253
36.3	用法	253
36.4	事件	254
36.5	按键	254
36.6	范例	254
36.7	相关 API	256
37	键盘 (lv_keyboard)	261
37.1	概述	261
37.2	零件和样式	261
37.3	用法	261
37.4	事件	262
37.5	按键	263
37.6	范例	263
37.7	相关 API	265
38	标签 (lv_label)	267
38.1	概述	267
38.2	零件和样式	267
38.3	用法	267
38.4	事件	269
38.5	按键	269
38.6	范例	270
38.7	相关 API	274
39	LED(lv_led)	275
39.1	概述	275

39.2	零件和样式	275
39.3	用法	275
39.4	事件	276
39.5	按钮处理	276
39.6	范例	276
39.7	相关 API	277
40	线 (lv_line)	279
40.1	概述	279
40.2	零件和样式	279
40.3	用法	279
40.4	事件	280
40.5	按钮处理	280
40.6	范例	280
40.7	相关 API	282
41	列表 (lv_list)	283
41.1	概述	283
41.2	零件和样式	283
41.3	用法	284
41.4	事件	285
41.5	按钮处理	285
41.6	范例	285
41.7	相关 API	287
42	仪表 (lv_lmeter)	289
42.1	概述	289
42.2	零件和样式	289
42.3	用法	289
42.4	事件	290
42.5	按钮处理	290
42.6	范例	290
42.7	相关 API	292
43	消息框 (lv_msdbox)	293
43.1	概述	293
43.2	零件和样式	293
43.3	用法	294
43.4	事件	294
43.5	按钮处理	294
43.6	范例	295
43.7	相关 API	299

44 对象蒙版 (lv_objmask)	301
44.1 概述	301
44.2 零件和样式	301
44.3 用法	301
44.4 事件	302
44.5 按键处理	302
44.6 范例	302
44.7 相关 API	307
45 页面 (lv_page)	309
45.1 概述	309
45.2 零件和样式	309
45.3 用法	310
45.4 事件	311
45.5 按键处理	312
45.6 范例	312
45.7 相关 API	313
46 滚筒 (lv_roller)	321
46.1 概述	321
46.2 零件和样式	321
46.3 用法	321
46.4 事件	322
46.5 按键处理	322
46.6 范例	323
46.7 相关 API	324
47 滑杆 (lv_slider)	325
47.1 概述	325
47.2 零件和样式	325
47.3 用法	326
47.4 事件	326
47.5 按键处理	327
47.6 范例	327
47.7 相关 API	330
48 数字调整框 (lv_spinbox)	331
48.1 概述	331
48.2 零件和样式	331
48.3 用法	331
48.4 事件	332
48.5 按键处理	332
48.6 范例	332

48.7	相关 API	334
49	旋转器 (lv_spinner)	335
49.1	概述	335
49.2	零件和样式	335
49.3	用法	335
49.4	按键处理	336
49.5	范例	336
49.6	相关 API	337
50	开关 (lv_switch)	339
50.1	概述	339
50.2	零件和样式	339
50.3	用法	339
50.4	事件	340
50.5	按键处理	340
50.6	范例	340
50.7	相关 API	342
51	表格 (lv_table)	343
51.1	概述	343
51.2	零件和样式	343
51.3	用法	344
51.4	事件	345
51.5	按键处理	345
51.6	范例	345
51.7	相关 API	347
52	页签 (lv_tabview)	349
52.1	概述	349
52.2	零件和样式	349
52.3	用法	350
52.4	事件	351
52.5	按键处理	351
52.6	范例	351
52.7	相关 API	353
53	文本框 (lv_textarea)	355
53.1	概述	355
53.2	零件和样式	355
53.3	用法	355
53.4	事件	358
53.5	按键处理	359

53.6 范例	359
53.7 相关 API	365
54 平铺视图 (lv_tileview)	367
54.1 概述	367
54.2 零件和样式	367
54.3 用法	367
54.4 事件	369
54.5 按键处理	369
54.6 范例	369
54.7 相关 API	371
55 窗口 (lv_win)	373
55.1 概述	373
55.2 零件和样式	373
55.3 事件	374
55.4 按键处理	375
55.5 范例	375
55.6 相关 API	376
56 LVGL 项目实战 (Windows 模拟器)	377
57 LVGL 项目实战 (基于 Linux 开发板)	379
58 LVGL 项目实战 (基于 STM32F103)	381
59 公司简介	383
60 联系方式	385

本课程适用于对 LVGL (v7) 开发感兴趣的同学
视频课程在线学习: www.100ask.net

- [认准百问网 lvgl 官方在线站点](#)
 - [点我下载百问网 lvgl 系列教程配套资料](#)
 - [点我下载百问网 lvgl 系列教程 pdf 格式文档](#)
 - [点我下载百问网 lvgl 系列教程 html 格式文档](#)
 - [点我下载百问网 lvgl 系列教程 epub 格式文档](#)
-



LVGL(轻巧而多功能的图形库) 是一个免费的开放源代码图形库，它提供创建具有易于使用的图形元素，精美的视觉效果和低内存占用的嵌入式 GUI 所需的一切。

1.1 获取离线教程文档

- [认准百问网 lvgl 官方在线站点](#)
- [点我下载百问网 lvgl 系列教程配套资料](#)
- [点我下载百问网 lvgl 系列教程 pdf 格式文档](#)
- [点我下载百问网 lvgl 系列教程 html 格式文档](#)
- [点我下载百问网 lvgl 系列教程 epub 格式文档](#)

1.2 主要特性

1. 功能强大的构建块，例如按钮，图表，列表，滑块，图像等。
2. 带有动画，抗锯齿，不透明，平滑滚动的高级图形
3. 各种输入设备，例如触摸板，鼠标，键盘，编码器等
4. 支持 UTF-8 编码的多语言
5. 多显示器支持，如 TFT，单色显示器
6. 完全可定制的图形元素
7. 独立于任何微控制器或显示器使用的硬件
8. 可扩展以使用很少的内存 (64 kB 闪存，16 kB RAM) 进行操作
9. 操作系统，支持外部存储器和 GPU，但不是必需的
10. 单帧缓冲区操作，即使具有高级图形效果
11. 用 C 语言编写，以实现最大的兼容性 (与 C++ 兼容)
12. 模拟器可在没有嵌入式硬件的 PC 上进行嵌入式 GUI 设计
13. 可移植到 MicroPython
14. 可快速上手的教程、示例、主题
15. 丰富的文档教程
16. 在 MIT 许可下免费和开源

1.3 硬件要求

基本上，每个现代控制器 (肯定必须要能够驱动显示器) 都适合运行 LVGL。LVGL 的最低运行要求很低：

- 16、32 或 64 位微控制器或处理器
- 最低 16 MHz 时钟频率
- Flash/ROM: 对于非常重要的组件要求 >64 kB(建议 > 180 kB)
- RAM
 - 静态 RAM 使用量: ~2 kB，取决于所使用的功能和对象类型
 - 堆栈: > 2kB(建议 > 8 kB)
 - 动态数据 (堆): > 2 KB(如果使用多个对象，则建议 > 16 kB)。由 lv_conf.h 中的 LV_MEM_SIZE 宏进行设置。
 - 显示缓冲区: > “水平分辨率” 像素 (建议 > 10× “水平分辨率”)

- MCU 或外部显示控制器中的一帧缓冲区
- C99 或更高版本的编译器
- 具备基本的 C(或 C++) 知识: 指针, 结构, 回调...

请注意, 内存使用情况可能会因具体的体系结构、编译器和构建选项而异。

1.4 源码布局

- `./lvgl` 库本身
- `./lv_drivers` 显示和输入设备驱动程序
- `./lv_examples` 示例和演示
- lvgl 官方文档网站 (<https://docs.lvgl.io>)
- lvgl 官方博客博客站点 (<https://blog.lvgl.io>)
- sim 在线模拟器网站 (<https://sim.lvgl.io>)
- `lv_sim_...` 适用于各种 IDE 和平台的模拟器项目
- `lv_port_...` 移植到其他开发板
- `lv_binding_...` 绑定到其他语言
- `lv_...` 移植到其他平台

其中, `lvgl`, `lv_examples` 和 `lv_drivers` 是最受维护、关注的核心存储库。



1.5 LVGL 更新发行规则

- lvgl 核心存储库遵循语义版本控制规则:
 - 不兼容的 API 的主要版本更改。例如。v5.0.0, v6.0.0
 - 次要版本, 用于新的但向后兼容的功能。例如。v6.1.0, v6.2.0
 - 修补程序版本, 用于向后兼容的错误修复。例如。v6.1.1, v6.1.2

1.6 LVGL 仓库分支说明

核心存储库至少具有以下分支：

- master 分支，最新版本，补丁直接在这里合并。
- dev 分支，开发人员在此处合并新功能，直到将它们合并到 master 分支为止。
- release/vX 分支，主要版本的稳定版本

1.7 发布周期

LVGL 有 2 周的发布周期。在每月的第一个和第三个 星期二：

1. (基于新功能) 从 master 分支创建主要、次要或错误修复的版本
2. 将 master 分支合并到 release/vX 中
3. 发布后立即将 dev 分支合并到 master 分支
4. 在接下来的 2 周内，测试 master 分支的新功能
5. 错误修复直接合并到 master 中
6. 2 周后，再从第一步重新开始迭代

1.8 版本标签

每个版本都会创建 vx.Y.Z 之类的标签，如：v7.9.0。

1.9 变更日志

版本更改记录在 ./lvgl/CHANGELOG.md 中。

1.10 版本兼容

在核心存储库中，每个主要版本都有一个分支 (例如 release/v6)。该主要版本的所有次要版本和修补程序版本都在此处合并

这样就可以添加稳定的较旧版本，而无需打扰较新的版本

所有主要版本的官方支持周期为 1 年。

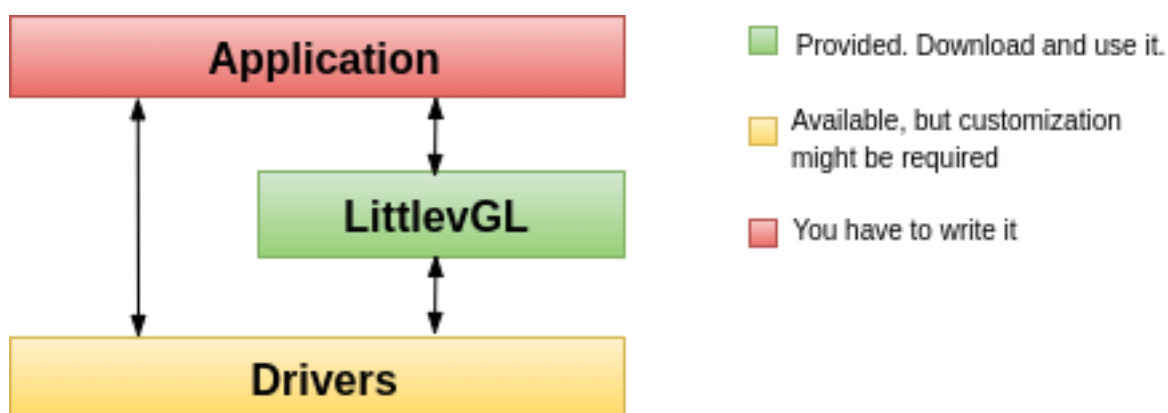


图 1: lvgl 系统框架

应用程序创建 GUI 并处理特定任务的应用程序。

LVGL 本身是一个图形库。我们的应用程序通过调用 LVGL 库来创建 GUI。它包含一个 HAL（硬件抽象层）接口，用于注册显示和输入设备驱动程序。

驱动程序除特定的驱动程序外，它还有其他的功能，可驱动显示器到 GPU (可选)、读取触摸板或按钮的输入。

根据 MCU，有两种典型的硬件设置。一个带有内置 LCD/TFT 驱动器的外围设备，而另一种是没有内置 LCD/TFT 驱动器的外围设备。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. 集成了 TFT/LCD 驱动器的 MCU 如果 MCU 集成了 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（如果 MCU 有足够的 RAM）中，也可以位于外

部 RAM (如果 MCU 具有存储器接口) 中。

2. 如果 MCU 没有集成 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。

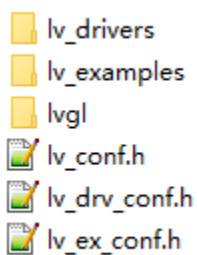
🔗 建立一个 lvgl 项目

要在我们的项目中使用 lvgl，我们起码需要获取到官方的这两个库：

- lvgl(lvgl) 核心图形库的官方 GitHub 仓库地址：<https://github.com/lvgl/lvgl>。
- lvgl(lv_drivers) 输入输出设备驱动官方 GitHub 仓库地址：https://github.com/lvgl/lv_drivers

我们可以克隆或下载这两个库的最新版本，将它们复制到我们的项目中，然后进行适配。

- 目录 lvgl 就是 lvgl 的官方图形库
- 目录 lv_drivers 是 lvgl 输入输出设备驱动官方示例配置
- 目录 lv_examples 是 lvgl 的官方 demo(可选，但不要直接使用到实际项目中)



3.1 配置文件

上面的三个库中有一个类似名为 `lv_conf_template.h` 的配置头文件 (template 就是模板的意思)。通过它可以设置库的基本行为, 裁剪不需要模块和功能, 在编译时调整内存缓冲区的大小等等。

1. 将 `lvgl/lv_conf_template.h` 复制到 `lvgl` 同级目录下, 并将其重命名为 `lv_drv_conf.h`。打开文件并将开头的 `#if 0` 更改为 `#if 1` 以使其内容。
2. 将 `lv_drivers/lv_drv_conf_template.h` 复制到 `lv_drivers` 同级目录下, 并将其重命名为 `lv_conf.h`。打开文件并将开头的 `#if 0` 更改为 `#if 1` 以使其内容。
3. (可选) 将 `lv_examples/lv_ex_conf_template.h` 复制到 `lv_examples` 同级目录下, 并将其重命名为 `lv_ex_conf.h`。打开文件并将开头的 `#if 0` 更改为 `#if 1` 以使其内容。

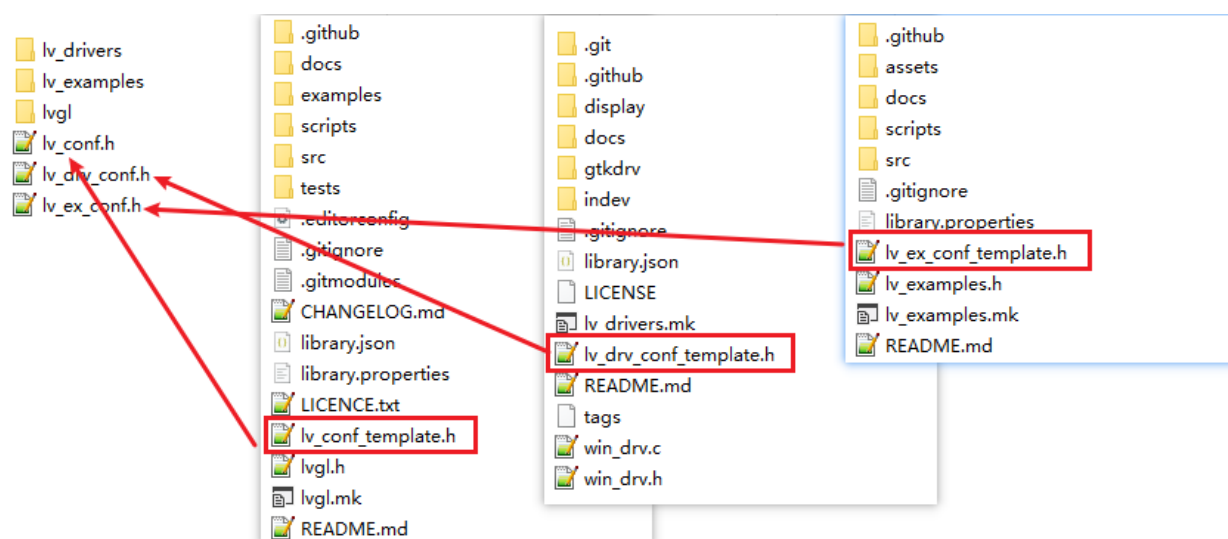


图 1: 准备 lvgl 配置文件

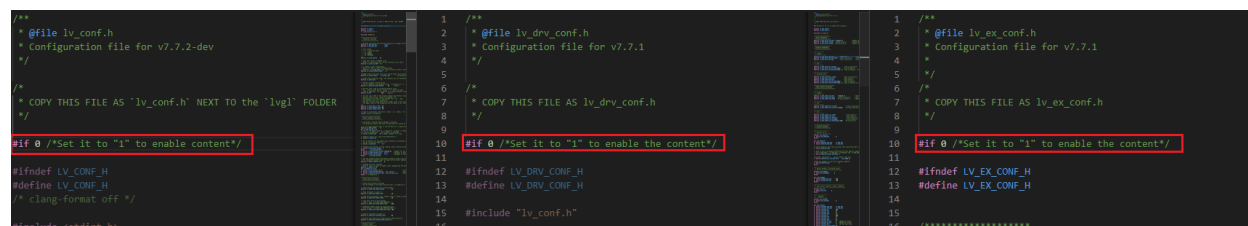


图 2: 使能配置文件

`lv_conf.h` 也可以复制到其他位置, 但是应该在编译器选项中添加 ``LV_CONF_INCLUDE_SIMPLE`` 定义 (例如, 对于 `gcc` 编译器为 ``-DLV_CONF_INCLUDE_SIMPLE``) 并手动设置包含路径。

在配置文件中, 注释说明了各个选项的含义。我们在移植时至少要检查以下三个配置选项, 其他配置根据具体的需要进行修改:

- LV_HOR_RES_MAX 显示器的水平分辨率。
- LV_VER_RES_MAX 显示器的垂直分辨率。
- LV_COLOR_DEPTH 颜色深度，其可以是：
 - 8 - RG332
 - 16 - RGB565
 - 32 - (RGB888 和 ARGB8888)

3.2 初始化 lvgl

准备好这三个库：lvgl、lv_drivers、lv_examples 后，我们就要开始使用 lvgl 带给我们的功能了。使用 lvgl 图形库之前，我们还必须初始化 lvgl 以及相关其他组件。初始化的顺序为：

1. 调用 lv_init() 初始化 lvgl 库；
2. 初始化驱动程序；
3. 在 LVGL 中注册显示和输入设备驱动程序；
4. 在中断中每隔 x 毫秒调用 lv_tick_inc(x) 用以告知 lvgl 经过的时间；
5. 每隔 x 毫秒定期调用 lv_task_handler() 用以处理与 lvgl 相关的任务。

3.2.1 Windows 初始化示例 (Cdoe::Blocks)

如果你是基于 windows 上的 IDE 模拟器 (推荐) 进行学习，请先 [点击这里](#) 下载配置好的项目工程 及 windows 上的 IDE 模拟器 (Cdoe::Blocks) 用于后面的学习。

```

1      #if WIN32
2      int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR_
↪szCmdLine, int nCmdShow)
3      #else
4      int main(int argc, char** argv)
5      {
6          /*Initialize LittlevGL*/
7          lv_init();
8
9          /*Initialize the HAL for LittlevGL*/
10         hal_init();
11
12         /*Check the themes too*/
13         lv_disp_set_default(lv_windows_disp);
14     }
  
```

(下页继续)

(续上页)

```

15         /*Run your APP here */
16
17
18     #if WIN32
19         while(!lv_win_exit_flag) {
20     #else
21         while(1) {
22     #endif // WIN32
23
24         /* Periodically call the lv_task handler.
25          * It could be done in a timer interrupt or an OS task too.*/
26         lv_task_handler();
27         usleep(5*1000);      /*Just to let the system breath*/
28         lv_tick_inc(5*1000)
29     }
30     return 0;
31 }

```

3.2.2 Linux 初始化示例

如果你是基于 Linux 开发板进行学习，请先 [点击这里](#) 下载配置好的项目工程 用于后面的学习。

```

1  int main(void)
2  {
3      /* LittlevGL init */
4      lv_init();
5
6      /* Linux frame buffer device init */
7      fbdev_init();
8
9      /* A small buffer for LittlevGL to draw the screen's content */
10     static lv_color_t buf[DISP_BUF_SIZE];
11
12     /* Initialize a descriptor for the buffer */
13     static lv_disp_buf_t disp_buf;
14     lv_disp_buf_init(&disp_buf, buf, NULL, DISP_BUF_SIZE);
15
16     /* Initialize and register a display driver */
17     lv_disp_drv_t disp_drv;
18     lv_disp_drv_init(&disp_drv);
19     disp_drv.buffer = &disp_buf;
20     disp_drv.flush_cb = fbdev_flush;
21     lv_disp_drv_register(&disp_drv);

```

(下页继续)

(续上页)

```
22
23     //hal_init
24     lv_disp_set_default lv_windows_disp
25
26
27     /* Linux input device init */
28     evdev_init();
29
30     /* Initialize and register a display input driver */
31     lv_indev_drv_t indev_drv;
32     lv_indev_drv_init(&indev_drv);      /*Basic initialization*/
33
34     indev_drv.type = LV_INDEV_TYPE_POINTER;
35     indev_drv.read_cb = evdev_read;    //lv_gesture_dir_t lv_indev_get_gesture_
↪dir(const lv_indev_t * indev)
36     lv_indev_t * my_indev = lv_indev_drv_register(&indev_drv);
37
38
39
40     /*Run your APP here */
41
42
43     /*Handle LittlevGL tasks (tickless mode)*/
44     while(1) {
45         lv_task_handler();
46         usleep(5000);
47         lv_tick_inc(5*1000);
48     }
49
50     return 0;
51 }
```

3.2.3 STM32F103 初始化示例

如果你是基于 STM32F103 进行学习, 请先 [点击这里](#) 下载配置好的项目工程 用于后面的学习。

TODO

要设置显示，必须初始化 `lv_disp_buf_t` 和 `lv_disp_drv_t` 变量。

- **lv_disp_buf_t** 保存显示缓冲区信息的结构体
- **lv_disp_drv_t** HAL 要注册的显示驱动程序、与显示交互并处理与图形相关的结构体、回调函数。

4.1 显示缓冲区

`lv_disp_buf_t` 初始化示例：

```
1  /*A static or global variable to store the buffers*/
2  static lv_disp_buf_t disp_buf;
3
4  /*Static or global buffer(s). The second buffer is optional*/
5  static lv_color_t buf_1[MY_DISP_HOR_RES * 10];
6  static lv_color_t buf_2[MY_DISP_HOR_RES * 10];
7
8  /*Initialize `disp_buf` with the buffer(s) */
9  lv_disp_buf_init(&disp_buf, buf_1, buf_2, MY_DISP_HOR_RES*10);
```

关于缓冲区大小，有 3 种情况：

1. 一个缓冲区 LVGL 将屏幕的内保存到缓冲区中并将其发送到显示器。缓冲区可以小于屏幕。在这种情况下，较大的区域将被重画成多个部分。如果只有很小的区域发生变化 (例如按下按钮)，则只会刷新该部分的区域。

2. **两个非屏幕大小的缓冲区**具有两个缓冲区的 LVGL 可以将其中一个作为显示缓冲区，而另一缓冲区的内容发送到后台显示。应该使用 DMA 或其他硬件将数据传输到显示器，以让 CPU 同时绘图。这样，渲染和刷新并行处理。与 **一个缓冲区**的情况类似，如果缓冲区小于要刷新的区域，LVGL 将按块绘制显示内容
3. **两个屏幕大小的缓冲区**与两个非屏幕大小的缓冲区相反，LVGL 将始终提供整个屏幕的内容，而不仅仅是块。这样，驱动程序可以简单地将帧缓冲区的地址更改为从 LVGL 接收的缓冲区。因此，当 MCU 具有 LCD/TFT 接口且帧缓冲区只是 RAM 中的一个位置时，这种方法的效果很好。

4.2 显示驱动器

一旦缓冲区初始化准备就绪，就需要初始化显示驱动程序。在最简单的情况下，仅需要设置 `lv_disp_drv_t` 的以下两个字段：

- **buffer** 指向已初始化的 `lv_disp_buf_t` 变量的指针。
- **flush_cb** 回调函数，用于将缓冲区的内容复制到显示的特定区域。刷新准备就绪后，需要调用 `lv_disp_flush_ready()`。LVGL 可能会以多个块呈现屏幕，因此多次调用 `flush_cb`。使用 `lv_disp_flush_is_last()` 可以查看哪块是最后渲染的。

其中，有一些可选的数据字段：

- **hor_res** 显示器的水平分辨率。(默认为 `lv_conf.h` 中的 `LV_HOR_RES_MAX`)
- **ver_res** 显示器的垂直分辨率。(默认为 `lv_conf.h` 中的 `LV_VER_RES_MAX`)
- **color_chroma_key** 在 chrome 键控图像上将被绘制为透明的颜色。(默认为 `lv_conf.h` 中的 `LV_COLOR_TRANSP`)
- **user_data** 驱动程序的自定义用户数据。可以在 `lv_conf.h` 中修改其类型。
- **anti-aliasing** 使用抗锯齿 (anti-aliasing)(边缘平滑)。缺省情况下默认为 `lv_conf.h` 中的 `LV_ANTIALIAS`。
- **rotated** 如果 1 交换 `hor_res` 和 `ver_res`。两种情况下 LVGL 的绘制方向相同 (从上到下的线条)，因此还需要重新配置驱动程序以更改显示器的填充方向。
- **screen_transp** 如果为 1，则屏幕可以具有透明或不透明的样式。需要在 `lv_conf.h` 中启用 `LV_COLOR_SCREEN_TRANSP`。

要使用 GPU，可以使用以下回调：

- **gpu_fill_cb** 用颜色填充内存中的区域。
- **gpu_blend_cb** 使用不透明度混合两个内存缓冲区。
- **gpu_wait_cb** 如果在 GPU 仍在运行 LVGL 的情况下返回了任何 GPU 函数，则在需要确保 GPU 渲染就绪时将使用此函数。

注意，这些功能需要绘制到内存 (RAM) 中，而不是直接显示在屏幕上。

其他一些可选的回调，使单色、灰度或其他非标准 RGB 显示一起使用时更轻松、优化：

- **rounder_cb** 四舍五入要重绘的区域的坐标。例如。2x2 像素可以转换为 2x8。如果显示控制器只能刷新特定高度或宽度的区域（对于单色显示器，通常为 8 px 高），则可以使用它。
- **set_px_cb** 编写显示缓冲区的自定义函数。如果显示器具有特殊的颜色格式，则可用于更紧凑地存储像素。（例如 1 位单色，2 位灰度等）。这样，lv_disp_buf_t 中使用的缓冲区可以较小，以仅保留给定区域大小所需的位数。set_px_cb 不能与两个屏幕大小的缓冲区一起显示缓冲区配置。
- **monitor_cb** 回调函数告诉在多少时间内刷新了多少像素。
- **clean_dcache_cb** 清除与显示相关的所有缓存的回调

要设置 lv_disp_drv_t 变量的字段，需要使用 lv_disp_drv_init(& disp_drv) 进行初始化。最后，要为 LVGL 注册显示设备，需要调用 lv_disp_drv_register(& disp_drv)。

代码示例：

```

1      lv_disp_drv_t disp_drv;                /*A variable to hold the drivers. Can be
↪local variable*/
2      lv_disp_drv_init(&disp_drv);           /*Basic initialization*/
3      disp_drv.buffer = &disp_buf;           /*Set an initialized buffer*/
4      disp_drv.flush_cb = my_flush_cb;        /*Set a flush callback to draw to the
↪display*/
5      lv_disp_t * disp;
6      disp = lv_disp_drv_register(&disp_drv); /*Register the driver and save the
↪created display objects*/

```

回调的一些简单示例：

```

1      void my_flush_cb(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t *
↪color_p)
2      {
3          /*The most simple case (but also the slowest) to put all pixels to the
↪screen one-by-one*/
4          int32_t x, y;
5          for(y = area->y1; y <= area->y2; y++) {
6              for(x = area->x1; x <= area->x2; x++) {
7                  put_px(x, y, *color_p)
8                  color_p++;
9              }
10         }
11
12         /* IMPORTANT!!!
13          * Inform the graphics library that you are ready with the flushing*/
14         lv_disp_flush_ready(disp);
15     }

```

(下页继续)

(续上页)

```

16
17 void my_gpu_fill_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest_buf, const lv_
↪ area_t * dest_area, const lv_area_t * fill_area, lv_color_t color);
18 {
19     /*It's an example code which should be done by your GPU*/
20     uint32_t x, y;
21     dest_buf += dest_width * fill_area->y1; /*Go to the first line*/
22
23     for(y = fill_area->y1; y < fill_area->y2; y++) {
24         for(x = fill_area->x1; x < fill_area->x2; x++) {
25             dest_buf[x] = color;
26         }
27         dest_buf += dest_width; /*Go to the next line*/
28     }
29 }
30
31 void my_gpu_blend_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest, const lv_color_
↪ t * src, uint32_t length, lv_opa_t opa)
32 {
33     /*It's an example code which should be done by your GPU*/
34     uint32_t i;
35     for(i = 0; i < length; i++) {
36         dest[i] = lv_color_mix(dest[i], src[i], opa);
37     }
38 }
39
40 void my_rounder_cb(lv_disp_drv_t * disp_drv, lv_area_t * area)
41 {
42     /* Update the areas as needed. Can be only larger.
43      * For example to always have lines 8 px height:*/
44     area->y1 = area->y1 & 0x07;
45     area->y2 = (area->y2 & 0x07) + 8;
46 }
47
48 void my_set_px_cb(lv_disp_drv_t * disp_drv, uint8_t * buf, lv_coord_t buf_w, lv_
↪ coord_t x, lv_coord_t y, lv_color_t color, lv_opa_t opa)
49 {
50     /* Write to the buffer as required for the display.
51      * Write only 1-bit for monochrome displays mapped vertically:*/
52     buf += buf_w * (y >> 3) + x;
53     if(lv_color_brightness(color) > 128) (*buf) |= (1 << (y % 8));
54     else (*buf) &= ~(1 << (y % 8));
55 }

```

(下页继续)

(续上页)

```
56
57 void my_monitor_cb(lv_disp_drv_t * disp_drv, uint32_t time, uint32_t px)
58 {
59     printf("%d px refreshed in %d ms\n", time, ms);
60 }
61
62 void my_clean_dcache_cb(lv_disp_drv_t * disp_drv, uint32_t)
63 {
64     /* Example for Cortex-M (CMSIS) */
65     SCB_CleanInvalidateDCache();
66 }
```

4.3 相关 APIs

TODO

5.1 输入设备的类型

要设置输入设备，必须初始化 `lv_indev_drv_t` 变量：

```
1  lv_indev_drv_t indev_drv;  
2  lv_indev_drv_init(&indev_drv);      /*Basic initialization*/  
3  indev_drv.type = ...                /*See below.*/  
4  indev_drv.read_cb = ...             /*See below.*/  
5  
6  /*Register the driver in LVGL and save the created input device object*/  
7  lv_indev_t * my_indev = lv_indev_drv_register(&indev_drv);
```

类型 (`indev_drv.type`) 可以是：

- **LV_INDEV_TYPE_POINTER** 触摸板或鼠标
- **LV_INDEV_TYPE_KEYPAD** 键盘或小键盘
- **LV_INDEV_TYPE_ENCODER** 带有左，右，推动选项的编码器
- **LV_INDEV_TYPE_BUTTON** 外部按钮按下屏幕

read_cb (`indev_drv.read_cb`) 是一个函数指针，将定期调用该函数指针以报告输入设备的当前状态。它还可以缓冲数据并在没有更多数据要读取时返回 `false`，或者在缓冲区不为空时返回 `true`。

进一步了解有关 输入设备 的更多信息。

5.2 触摸板，鼠标或任何指针

可以单击屏幕点的输入设备属于此类别。

```

1   indev_drv.type = LV_INDEV_TYPE_POINTER;
2   indev_drv.read_cb = my_input_read;
3
4   ...
5
6   bool my_input_read(lv_indev_drv_t * drv, lv_indev_data_t*data)
7   {
8       data->point.x = touchpad_x;
9       data->point.y = touchpad_y;
10      data->state = LV_INDEV_STATE_PR or LV_INDEV_STATE_REL;
11      return false; /*No buffering now so no more data read*/
12  }
```

即使状态为 LV_INDEV_STATE_REL，触摸板驱动程序也必须返回最后的 X/Y 坐标。

要设置鼠标光标，请使用 `lv_indev_set_cursor(my_indev, &img_cursor)`。(my_indev 是 `lv_indev_drv_register` 的返回值) 键盘或键盘

5.3 触摸板或键盘

带有所有字母的完整键盘或带有一些导航按钮的简单键盘均属于此处。

要使用键盘/触摸板：

- 注册具有 LV_INDEV_TYPE_KEYPAD 类型的 read_cb 函数。
- 在 lv_conf.h 中启用 LV_USE_GROUP
- 必须创建一个对象组：lv_group_t * g = lv_group_create()，并且必须使用 lv_group_add_obj(g, obj) 向其中添加对象
- 必须将创建的组分配给输入设备：lv_indev_set_group(my_indev, g)(my_indev 是 lv_indev_drv_register 的返回值)
- 使用 LV_KEY _... 在组中的对象之间导航。有关可用的密钥，请参见 lv_core/lv_group.h。

```

1   indev_drv.type = LV_INDEV_TYPE_KEYPAD;
2   indev_drv.read_cb = keyboard_read;
3
4   ...
5
```

(下页继续)

(续上页)

```

6  bool keyboard_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
7      data->key = last_key();           /*Get the last pressed or released key*/
8
9      if(key_pressed()) data->state = LV_INDEV_STATE_PR;
10     else data->state = LV_INDEV_STATE_REL;
11
12     return false; /*No buffering now so no more data read*/
13 }

```

5.4 编码器

可以通过下面四种方式使用编码器：

1. 按下按钮
2. 长按其按钮
3. 转左
4. 右转

简而言之，编码器输入设备的工作方式如下：

- 通过旋转编码器，可以专注于下一个/上一个对象。
- 在简单对象 (如按钮) 上按下编码器时，将单击它。
- 如果将编码器按在复杂的对象 (如列表，消息框等) 上，则该对象将进入编辑模式，从而转动编码器即可在对象内部导航。
- 长按按钮，退出编辑模式。

要使用编码器 (类似于键盘)，应将对象添加到组中。

```

1  indev_drv.type = LV_INDEV_TYPE_ENCODER;
2  indev_drv.read_cb = encoder_read;
3
4  ...
5
6  bool encoder_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
7      data->enc_diff = enc_get_new_moves();
8
9      if(enc_pressed()) data->state = LV_INDEV_STATE_PR;
10     else data->state = LV_INDEV_STATE_REL;
11
12     return false; /*No buffering now so no more data read*/
13 }

```

5.4.1 使用带有编码器逻辑的按钮

除了标准的编码器行为外，您还可以利用其逻辑来使用按钮导航（聚焦）和编辑小部件。如果只有几个按钮可用，或者除编码器滚轮外还想使用其他按钮，这将特别方便。

需要有 3 个可用的按钮：

- **LV_KEY_ENTER** 将模拟按下或推动编码器按钮
- **LV_KEY_LEFT** 将向左模拟转向编码器
- **LV_KEY_RIGHT** 将正确模拟转向编码器
- 其他键将传递给焦点小部件

如果按住这些键，它将模拟 `indev_drv.long_press_rep_time` 中指定的时间段内的编码器单击。

```

1  indev_drv.type = LV_INDEV_TYPE_ENCODER;
2  indev_drv.read_cb = encoder_with_keys_read;
3
4  ...
5
6  bool encoder_with_keys_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
7      data->key = last_key();           /*Get the last pressed or released key*/
8                                          /* use_
↪LV_KEY_ENTER for encoder press */
9      if(key_pressed()) data->state = LV_INDEV_STATE_PR;
10     else {
11         data->state = LV_INDEV_STATE_REL;
12         /* Optionally you can also use enc_diff, if you have encoder*/
13         data->enc_diff = enc_get_new_moves();
14     }
15
16     return false; /*No buffering now so no more data read*/
17 }

```

5.5 按键

按钮是指屏幕旁边的外部“硬件”按钮，它们被分配给屏幕的特定坐标。如果按下按钮，它将模拟在指定坐标上的按下。（类似于触摸板）

使用 `lv_indev_set_button_points(my_indev, points_array)` 将按钮分配给坐标。`points_array` 应该看起来像 `const lv_point_t points_array [] = {{12,30}, {60,90}, ...}`

`points_array` 不能超出范围。将其声明为全局变量或函数内部的静态变量。

```

1  indev_drv.type = LV_INDEV_TYPE_BUTTON;
2  indev_drv.read_cb = button_read;
3
4  ...
5
6  bool button_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
7      static uint32_t last_btn = 0;    /*Store the last pressed button*/
8      int btn_pr = my_btn_read();      /*Get the ID (0,1,2...) of the pressed_
↪button*/
9      if(btn_pr >= 0) {                 /*Is there a button press? (E.g. -1_
↪indicated no button was pressed)*/
10         last_btn = btn_pr;            /*Save the ID of the pressed button*/
11         data->state = LV_INDEV_STATE_PR; /*Set the pressed state*/
12     } else {
13         data->state = LV_INDEV_STATE_REL; /*Set the released state*/
14     }
15
16     data->btn = last_btn;              /*Save the last button*/
17
18     return false;                    /*No buffering now so no more data read*/
19 }

```

5.6 其它功能

除了 read_cb 之外，还可以在 lv_indev_drv_t 中指定 feedback_cb 回调。输入设备发送任何类型的事件时，都会调用 feedback_cb。(独立于其类型)。它允许为用户提供反馈，例如在 LV_EVENT_CLICK 上播放声音。

可以在 lv_conf.h 中设置以下参数的默认值，但可以在 lv_indev_drv_t 中覆盖默认值：

- **拖拽限制 (drag_limit)** 实际拖动对象之前要滑动的像素数 drag_throw 拖曳速度降低 [%]。更高的价值意味着更快的减速
- **(drag_throw)** 拖曳速度降低 [%]。更高的价值意味着更快的减速
- **(long_press_time)** 按下时间发送 LV_EVENT_LONG_PRESSED (以毫秒为单位)
- **(long_press_rep_time)** 发送 LV_EVENT_LONG_PRESSED_REPEAT 的时间间隔 (以毫秒为单位)
- **(read_task)** 指向读取输入设备的 lv_task 的指针。可以通过 lv_task_...() 函数更改其参数

每个输入设备都与一个显示器关联。默认情况下，新的输入设备将添加到最后创建的或显式选择的显示设备 (使用 lv_disp_set_default())。相关的显示已存储，并且可以在驱动程序的显示字段中更改。

5.7 相关 API

TODO

LVGL 需要系统滴答声才能知道动画和其他任务的经过时间。

为此我们需要定期调用 `lv_tick_inc(tick_period)` 函数，并以毫秒为单位告知调用周期。例如，`lv_tick_inc(1)` 用于每毫秒调用一次。

为了精确地知道经过的毫秒数，`lv_tick_inc` 应该在比 `lv_task_handler()` 更高优先级的例程中被调用（例如在中断中），即使 `lv_task_handler` 的执行花费较长时间。

使用 FreeRTOS 时，可以在 `vApplicationTickHook` 中调用 `lv_tick_inc`。

在基于 Linux 的设备上（例如在 Raspberry Pi 上），`lv_tick_inc` 可以在如下所示的线程中调用，比如：

```
1  void * tick_thread (void *args)
2  {
3      while(1) {
4          usleep(5*1000);    /*Sleep for 5 millisecond*/
5          lv_tick_inc(5);     /*Tell LVGL that 5 milliseconds were
↳elapsed*/
6      }
7  }
```

任务处理器 (Task Handler)

要处理 LVGL 的任务，我们需要定期通过以下方式之一调用 `lv_task_handler()`：

- `main` 函数中设置 `while(1)` 调用
- 定期定时中断 (低优先级然后是 `lv_tick_inc()`) 中调用
- 定期执行的 OS 任务中调用

计时并不严格，但应保持大约 5 毫秒以保持系统响应。

范例：

```
1  while(1) {  
2      lv_task_handler();  
3      my_delay_ms(5);  
4  }
```

要了解有关任务的更多信息，请参阅 [任务](#) 部分。

CHAPTER 8

睡眠管理

没有用户输入时，MCU 可以进入睡眠状态。在这种情况下，mian 函数中的 while(1) 应该看起来像这样：

```
1  while(1) {
2      /*Normal operation (no sleep) in < 1 sec inactivity*/
3      if(lv_disp_get_inactive_time(NULL) < 1000) {
4          lv_task_handler();
5      }
6      /*Sleep after 1 sec inactivity*/
7      else {
8          timer_stop(); /*Stop the timer where lv_tick_inc() is called*/
9          sleep();      /*Sleep the MCU*/
10     }
11     my_delay_ms(5);
12 }
```

如果发生唤醒 (按，触摸或单击等)，还应该在输入设备读取功能中添加以下几行：

```
1  lv_tick_inc(LV_DISP_DEF_REFR_PERIOD); /*Force task execution on wake-up*/
2  timer_start(); /*Restart the timer where lv_tick_inc() is
↪called*/
3  lv_task_handler(); /*Call `lv_task_handler()` manually to
↪process the wake-up event*/
```

除了 lv_disp_get_inactive_time() 外，还可以调用 lv_anim_count_running() 来查看每个动画是否完成。

LVGL 默认情况下 **不是线程安全的**。

但是，在以下情况中，调用 LVGL 相关函数是有效的：

- 在事件 (Events) 中。在”事件”中了解更多信息。
- 在 (lv_tasks) 中。在”任务”中了解更多信息。

9.1 任务和线程

如果需要使用实际的任务或线程，则需要一个互斥锁，该互斥锁应在调用 `lv_task_handler` 之前被调用，并在其之后释放。同样，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，就可以在真正的多任务环境中使用 LVGL。只需使用互斥锁 (mutex) 即可避免同时调用 LVGL 函数。

9.2 中断

避免从中断中调用 LVGL 函数 (`lv_tick_inc()` 和 `lv_disp_flush_ready()` 除外)。但是，如果需要执行此操作，则必须在 `lv_task_handler` 运行时禁用 LVGL 函数的中断。设置标志或某个值并在 `lv_task` 中定期检查是一种不错的方法。

LVGL 内置有日志模块，用于记录用户库中正在发生的事情。

10.1 日志级别

要启用日志记录，需要在 `lv_conf.h` 中将 **LV_USE_LOG** 设置为 1，并将 **LV_LOG_LEVEL** 设置为以下值之一：

- **LV_LOG_LEVEL_TRACE** 记录所有信息
- **LV_LOG_LEVEL_INFO** 记录重要事件
- **LV_LOG_LEVEL_WARN** 记录是否发生了警告事件
- **LV_LOG_LEVEL_ERROR** 记录错误信息，当系统可能发生故障时或致命错误
- **LV_LOG_LEVEL_NONE** 不要记录任何东西

级别高于设置的日志级别的事件也将被记录。例如。如果使用 **LV_LOG_LEVEL_WARN**，也会记录错误。

10.2 使用 printf 记录

如果您的系统支持 printf，则只需在 lv_conf.h 中启用 ****LV_LOG_PRINTF**** 即可发送带有 printf 的日志。

10.3 自定义日志功能

如果不能使用 printf 或想要使用自定义函数进行日志记录，可以使用 lv_log_register_print_cb() 注册“logger”回调。

例如：

```
1  void my_log_cb(lv_log_level_t level, const char * file, int line, const char * fn_
   ↳name, const char * dsc)
2  {
3      /*Send the logs via serial port*/
4      if(level == LV_LOG_LEVEL_ERROR) serial_send("ERROR: ");
5      if(level == LV_LOG_LEVEL_WARN)  serial_send("WARNING: ");
6      if(level == LV_LOG_LEVEL_INFO)  serial_send("INFO: ");
7      if(level == LV_LOG_LEVEL_TRACE) serial_send("TRACE: ");
8
9      serial_send("File: ");
10     serial_send(file);
11
12     char line_str[8];
13     sprintf(line_str, "%d", line);
14     serial_send("#");
15     serial_send(line_str);
16
17     serial_send(": ");
18     serial_send(fn_name);
19     serial_send(": ");
20     serial_send(dsc);
21     serial_send("\n");
22 }
23
24 ...
25
26 lv_log_register_print_cb(my_log_cb);
```

10.4 添加日志

还可以通过 `LV_LOG_TRACE/INFO/WARN/ERROR(description)` 函数使用日志模块。

在 LVGL 中，用户界面的基本构建块是对象，也称为小部件 (widget)。例如，按钮，标签，图像，列表，图表或文本区域。

查看 [LVGL 所有的对象类型 \(widget\)](#)。

11.1 对象的属性 (Attributes)

11.1.1 对象的基本属性

所有对象类型都共享一些基本属性：

- Position (位置)
- Size (尺寸)
- Parent (父母)
- Drag enable (拖动启用)
- Click enable (单击启用)
- position (位置)
- 等等

我们可以使用 `lv_obj_set _...` 和 `lv_obj_get _...` 等前缀的函数设置或者获取这些属性。例如：

```

1  /* 设置基础对象的属性 */
2  lv_obj_set_size(btn1, 100, 50);    /* 设置按键的大小 */
3  lv_obj_set_pos(btn1, 20, 30);     /* 设置按键的位置 */

```

11.1.2 对象的特殊属性

有些对象类型也具有特殊的属性。例如，滑块具有

- Min. max. values (最小最大值)
- Current value (当前值)
- Custom styles (自定义样式)

对于这些属性，每种对象类型都有唯一的 API 函数。例如一个滑块的 API 调用过程：

```

1  /* 设置滑块的特殊属性 */
2  lv_slider_set_range(slider1, 0, 100);          /* 设置滑块的最小值和最大值 */
3  lv_slider_set_value(slider1, 40, LV_ANIM_ON);  /* 设置当前值 (屏幕坐标系位置) */
4  lv_slider_set_action(slider1, my_action);      /* 设置回调函数 */

```

要查看 API 的实现代码，可以检查相应的头文件（例如滑块对象的头文件 `lv_objx/lv_slider.h`）

11.2 对象的工作机制

11.2.1 亲子结构

父对象可以作为其子对象的容器。每个对象只能一个父对象（**屏幕除外**），但是一个父对象可以有无限多个子对象。父对象的类型没有限制，但是有特殊的父对象（例如，按钮）和特殊的子对象（例如，标签）。

11.3 追随原则

如果更改了父对象的位置，则子对象将与父对象一起移动，并且子对象的位置都保持相对于父对象位置不变。例如，坐标 (0,0) 表示子对象将独立于父对象的位置保留在父对象的左上角，代码：

```

1  lv_obj_t * par = lv_obj_create(lv_scr_act(), NULL); /* 在当前屏幕中创建一个对象 */
2  lv_obj_set_size(par, 100, 80);                     /* 设置对象的大小 */
3
4  lv_obj_t * obj1 = lv_obj_create(par, NULL);         /* 基于前面创建的对象 (par) 创建一个子对象 (obj1)，之前的对象成为父对象 */
5  lv_obj_set_pos(obj1, 10, 10);                      /* 设置子对象的位置 */

```

当我们修改父对象的位置，子对象也会一起移动，以保持和父对象的相对位置不变：



图 1: 一个父子对象

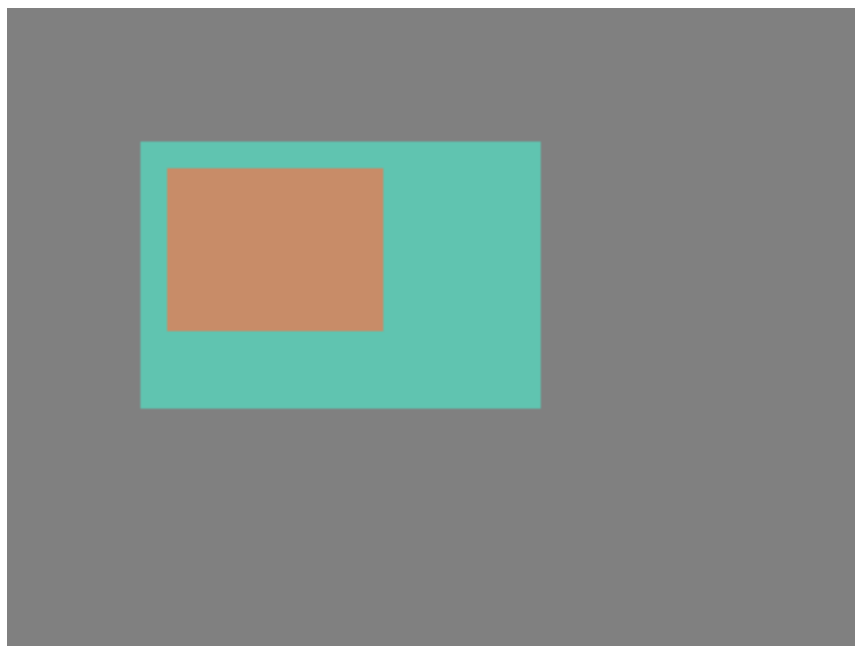


图 2: 子对象跟随父对象

```
lv_obj_set_pos(par, 50, 50);    /* 移动父对象，子对象也会跟着移动，以保持相对位置不变 */
```

11.3.1 子对象仅在父对象的范围内可见

如果子对象的部分或全部不在其父级之内，则超出父对象的部分将不可见。



图 3: 子对象超出父对象的部分不可见

```
lv_obj_set_x(obj1, -30);    /* 将子对象移出一部分到从父对象的范围内之外 */
```

11.3.2 创建-删除对象

在 LVGL 中，可以在运行时动态地创建和删除对象。这意味着仅当前创建的对象需要消耗 RAM。例如，如果需要图表，我们可以在需要时创建它，并在不可见或不需要时将其删除。

每个对象类型都有各自的创建函数。它需要两个参数：

- 指向父对象的指针。创建屏幕时以 NULL 作为父级。
- 用于复制具有相同类型的对象的指针 (可选)。如果不进行复制操作作为 NULL。

使用 lv_obj_t 指针作为句柄在 C 代码中引用所有对象。以后可以使用该指针设置或获取对象的属性。

创建函数如下所示：

```
lv_obj_t * lv_ <type>_create(lv_obj_t * parent, lv_obj_t * copy);
```


所有对象类型都有一个通用的删除功能。它删除对象及其所有子对象。

```
void lv_obj_del(lv_obj_t * obj);
```

`lv_obj_del` 将立即删除该对象。如果出于某种原因不能立即删除该对象，则可以使用 `lv_obj_del_async(obj)`，例如，如果要删除子对象的 `LV_EVENT_DELETE` 信号中对象的父对象，这很有用。

我们可以使用 `lv_obj_clean` 删除对象的所有子对象（但不会删除对象本身）：

```
void lv_obj_clean(lv_obj_t * obj);
```

11.4 屏幕对象

11.4.1 创建屏幕对象

屏幕是没有父对象的特殊对象。应该像这样创建它们：

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

可以使用任何对象类型创建屏幕。例如：创建墙纸的基础对象或图像。

11.4.2 获取活动屏幕

这始终是每个显示屏上的活动屏幕。默认情况下，该库为每个显示创建并加载“基础对象”作为屏幕。

要获取当前活动的屏幕使用函数 `lv_scr_act()`

11.4.3 载入屏幕

调用函数 `lv_scr_load(scr1)` 加载屏幕。

11.4.4 加载屏幕动画

我们可以调用函数：`lv_scr_load_anim(scr, transition_type, time, delay, auto_del)` 加载屏幕动画。参数 `transition_type` 是动画过渡类型，该参数可设为：

- `LV_SCR_LOAD_ANIM_NONE` 延迟 x 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` 将旧屏幕和新屏幕都移至给定方向
- “`LV_SCR_LOAD_ANIM_FADE_ON`” 使新屏幕淡出旧屏幕

将 `auto_del` 设置为 `true` 会在动画结束时自动删除旧屏幕。

在延迟时间之后开始动画播放时，新屏幕将变为活动状态（由 `lv_scr_act()` 返回）。

11.4.5 处理多个显示

屏幕在当前选择的默认屏幕上创建。默认显示设备使用 `lv_disp_drv_register` 注册的最后一个屏幕作为显示，或者可以使用 `lv_disp_set_default(disp)` 显式选择新的默认显示屏幕。

`lv_scr_act()` , `lv_scr_load()` 和 `lv_scr_load_anim()` 将会在默认的屏幕上操作。

访问多显示器支持以了解更多信息。

11.5 零件 (Parts)

widget 可以包含多个 Parts 。例如，按钮仅具有主要部分，而滑块则由背景，指示器和旋钮组成。

Parts 名称的构造类似于 `LV_ + <TYPE> _PART_ <NAME>` 。比如 `LV_BTN_PART_MAIN` 、`LV_SLIDER_PART_KNOB` 。通常在将样式添加到对象时使用 Parts。使用 Parts 可以将不同的样式分配给对象的不同 Parts 。

11.6 状态-States

对象可以处于以下状态的组合：

- **LV_STATE_DEFAULT** 默认或正常状态
- **LV_STATE_CHECKED** 选中或点击
- **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_EDITED** 由编码器编辑
- **LV_STATE_HOVERED** 鼠标悬停（现在还不支持）
- **LV_STATE_PRESSED** 按下
- **LV_STATE_DISABLED** 禁用或无效

当用户按下，释放，聚焦等对象时，状态通常由库自动检测更改。当然状态也可以手动检测更改。要完全覆盖当前状态，调用 `lv_obj_set_state(obj, part, LV_STATE...)` 要设置或清除某个状态 (但不更改其他状态)，调用 `lv_obj_add/clear_state(obj, part, LV_STATE...)` 可以组合使用状态值。例如：`lv_obj_set_state(obj, part, LV_STATE_PRESSED | LV_PRESSED_CHECKED)` 。

要了解有关状态的更多信息，请阅读 [样式 \(Styles\)](#) 概述的相关部分。

12.1 创建对象层级顺序

默认情况下，LVGL 在背景上绘制旧对象，在前景上绘制新对象。

例如，假设我们向父对象添加了一个名为 `button1` 的按钮，然后又添加了另一个名为 `button2` 的按钮。由于先创建了 `button1`，所以 `button1` 会被 `button2` 及其子对象覆盖。



```

1      /*Create a screen*/
2      lv_obj_t * scr = lv_obj_create(NULL, NULL);
3      lv_scr_load(scr);
↪ /*Load the screen*/
4
5      /*Create 2 buttons*/
6      lv_obj_t * btn1 = lv_btn_create(scr, NULL);          /*Create a button on the_
↪ screen*/
7      lv_btn_set_fit(btn1, true, true);                    /*Enable to automatically set_
↪ the size according to the content*/
8      lv_obj_set_pos(btn1, 60, 40);                        /*Set the_
↪ position of the button*/
9
10     lv_obj_t * btn2 = lv_btn_create(scr, btn1);           /*Copy the first button*/
11     lv_obj_set_pos(btn2, 180, 80);                        /*Set the position of the_
↪ button*/
12
13     /*Add labels to the buttons*/
14     lv_obj_t * label1 = lv_label_create(btn1, NULL);       /*Create a label on the_
↪ first button*/
15     lv_label_set_text(label1, "Button 1");                /*Set the text of the_
↪ label*/
16
17     lv_obj_t * label2 = lv_label_create(btn2, NULL);       /*Create a label on the_
↪ second button*/
18     lv_label_set_text(label2, "Button 2");                /*Set the text of the_
↪ label*/
19
20     /*Delete the second label*/
21     lv_obj_del(label2);

```

12.2 将图层设到前台 (foreground) 展示

有几种方法可以将对象置于前台：

- 使用 `lv_obj_set_top(obj, true)`。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。它的工作原理类似于 PC 机上典型的 GUI，当点击背景中的窗口时，它会在前台展示。
- 使用 `lv_obj_move_foreground(obj)` 显式地告诉库将对象带到前台。类似地，使用 `lv_obj_move_background(obj)` 将对象 `obj` 移动到后台。
- 当使用 `lv_obj_set_parent(obj, new_parent)` 时，`obj` 将在 `new_parent` 的前面。

12.3 顶层和系统层

LVGL 有两个特殊的图层; `layer_top` 和 `layer_sys`。两者在显示器的所有屏幕上都是可见且通用的。但是, 它们不会在多个物理显示器之间共享。 `layer_top` 始终位于默认屏幕 (`lv_scr_act()`) 的顶部, `layer_sys` 则位于 `layer_top` 的顶部。用户可以使用 `layer_top` 来创建一些随处可见的内容。例如, 菜单栏, 弹出窗口等。如果启用了 `click` 属性, 那么 `layer_top` 将吸收所有用户单击并充当模态。

```
lv_obj_set_click(lv_layer_top(), true);
```

`layer_sys` 也用于 LVGL。例如, 它将鼠标光标放在那里以确保它始终可见。

CHAPTER 13

事件 (Events)

LVGL 中可触发事件，用于与用户进行交互。例如一个对应对象的事件可以有：

- 被点击
- 被拖拽
- 被更改了数值
- 等等

我们可以将回调函数分配给对象以处理这些事件。例如：

```
1  lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);
2  lv_obj_set_event_cb(btn, my_event_cb);      /* 指定一个事件回调函数 */
3
4  ...
5
6  static void my_event_cb(lv_obj_t * obj, lv_event_t event)
7  {
8      switch(event) {
9          case LV_EVENT_PRESSED:                /* 对象被按下 */
10             printf("Pressed\n");
11             break;
12
13          case LV_EVENT_SHORT_CLICKED:          /* 对象被点击 */
14             printf("Short clicked\n");
15             break;
```

(下页继续)

(续上页)

```
16
17     case LV_EVENT_CLICKED:                /* 对象被短点击 */
18         printf("Clicked\n");
19         break;
20
21     case LV_EVENT_LONG_PRESSED:            /* 对象被长按 */
22         printf("Long press\n");
23         break;
24
25     case LV_EVENT_LONG_PRESSED_REPEAT:     /* 对象被重复长按 */
26         printf("Long press repeat\n");
27         break;
28
29     case LV_EVENT_RELEASED:                /* 对象被释放 */
30         printf("Released\n");
31         break;
32     }
33
34     /*Etc.*/
35 }
```

注意：多个对象可以使用同一回调函数。

13.1 事件类型

事件类型有如下几种：

13.1.1 通用事件

所有对象（例如 Buttons/Labels/Sliders 等）都可以接收这些通用事件。

13.1.2 与输入设备有关的事件

当用户按下/释放对象时，将发送这些消息。它们不仅用于指针，还可以用于键盘，编码器和按钮输入设备。访问输入设备概述部分以了解有关它们的更多信息。

- **LV_EVENT_PRESSED** 该对象被按下
- **LV_EVENT_PRESSING** 按下对象（按下时连续发送）
- **LV_EVENT_PRESS_LOST** 输入设备仍在按，但不再在对象上

- **LV_EVENT_SHORT_CLICKED** 在 LV_INDEV_LONG_PRESS_TIME 时间之前发布。如果拖动则不调用。
- **LV_EVENT_LONG_PRESSED** 按下 LV_INDEV_LONG_PRESS_TIME 时间。如果拖动则不调用。
- **LV_EVENT_LONG_PRESSED_REPEAT** 在每 LV_INDEV_LONG_PRESS_REPEAT_TIME 毫秒的 LV_INDEV_LONG_PRESS_TIME 之后调用。如果拖动则不调用。
- **LV_EVENT_CLICKED** 如果未拖动则调用释放（无论长按）
- **LV_EVENT_RELEASED** 在上面每种情况下都被调用，即使对象已被拖动也被释放。如果在按下并从对象外部释放时从对象上滑出，则不会调用。在这种情况下，将发送 LV_EVENT_PRESS_LOST。

13.1.3 指针相关的事件

这些事件仅由类似指针的输入设备（例如鼠标或触摸板）触发

- **LV_EVENT_DRAG_BEGIN** 开始拖动对象
- **LV_EVENT_DRAG_END** 拖动完成（包括拖动）
- **LV_EVENT_DRAG_THROW_BEGIN** 拖动开始（用“动量”拖动后释放）

13.1.4 与键盘和编码器相关的事件

这些事件由键盘和编码器输入设备发送。在 [overview/indev](输入设备) 部分中了解有关组的更多信息。

- **LV_EVENT_KEY** 键值发送到对象。通常在按下它或在长按之后重复时。可以通过以下方式检索键值
`uint32_t * key = lv_event_get_data()`
- **LV_EVENT_FOCUSED** 该对象集中在其组中
- **LV_EVENT_DEFOCUSED** 该对象在其组中散焦

13.1.5 一般事件

LVGL 库发送的其他一般事件。

- **LV_EVENT_DELETE** 该对象正在被删除。释放相关的用户分配数据。

13.1.6 特殊事件

这些事件特定于特定的对象类型。

- **LV_EVENT_VALUE_CHANGED** 对象值已更改（例如，对于滑块）
- **LV_EVENT_INSERT** 有内容插入到对象中。（通常到文本区域）
- **LV_EVENT_APPLY** 单击“确定”，“应用”或类似的特定按钮。（通常来自键盘对象）
- **LV_EVENT_CANCEL** 单击“关闭”，“取消”或类似的特定按钮。（通常来自键盘对象）
- **LV_EVENT_REFRESH** 查询以刷新对象。永远不会由库发送，但可以由用户发送。

请访问特定对象类型的文档，以了解对象类型使用了哪些事件。

13.2 自定义事件包含的数据

一些事件可能包含自定义数据。例如，在某些情况下，**LV_EVENT_VALUE_CHANGED** 会告知新值。有关更多信息，请参见特定对象类型的文档。要在事件回调中获取自定义数据，请使用 `lv_event_get_data()`。

自定义数据的类型取决于发送对象，但如果是下面两种情况需要特殊对待：

- 数值，则为 `uint32_t *` 或 `int32_t *` 类型
- 字符，则为 `char *` 或 `const char *` 类型

13.3 手动发送事件

13.3.1 任意事件

要将事件手动发送到对象，请使用 `lv_event_send(obj, LV_EVENT_..., &custom_data)`。

例如，它可以通过模拟按钮按下来手动关闭消息框（尽管有更简单的方法）：

```
1  /*Simulate the press of the first button (indexes start from zero)*/
2  uint32_t btn_id = 0;
3  lv_event_send(mbox, LV_EVENT_VALUE_CHANGED, &btn_id);
```

13.3.2 刷新事件

LV_EVENT_REFRESH 是特殊事件，因为它旨在供用户用来通知对象刷新自身。一些例子：

- 通知标签根据一个或多个变量（例如当前时间）刷新其文本
- 语言更改时刷新标签
- 如果满足某些条件，请启用按钮（例如，输入正确的 PIN）
- 如果超出限制，则向对象添加样式/从对象删除样式等

处理类似情况的最简单方法是利用以下函数：

`lv_event_send_refresh(obj)` 只是 `lv_event_send(obj, LV_EVENT_REFRESH, NULL)` 的包装。因此，它仅向对象发送 LV_EVENT_REFRESH。

`lv_event_send_refresh_recursive(obj)` 将 LV_EVENT_REFRESH 事件发送给对象及其所有子对象。如果将 NULL 作为参数传递，则将刷新所有显示的所有对象。

CHAPTER 14

输入设备 (Input devices)

在 LVGL 中输入设备，有下面几种类型：

- 指针式输入设备，如触摸板或鼠标
- 键盘，如普通键盘或简单的数字键盘
- 带有左/右转向和推入选项的编码器
- 外部硬件按钮，分配给屏幕上的特定点

在进一步阅读本文之前，请先阅读 [输入设备接口](#) 的部分内容

14.1 指针

指针输入设备可以具有光标。(通常是鼠标)

```
1      ...
2      lv_indev_t * mouse_indev = lv_indev_drv_register(&indev_drv);
3
4      LV_IMG_DECLARE(mouse_cursor_icon);           /*Declare the image_
↪file.*/
5      lv_obj_t * cursor_obj = lv_img_create(lv_scr_act(), NULL); /*Create an image_
↪object for the cursor */
6      lv_img_set_src(cursor_obj, &mouse_cursor_icon);      /*Set the image_
↪source*/
7      lv_indev_set_cursor(mouse_indev, cursor_obj);        /*Connect the image_
↪object to the driver*/
```

(下页继续)

请注意，光标对象应设置为 `lv_obj_set_click(cursor_obj, false)`。对于图像，默认情况下禁用单击。

14.2 键盘和编码器

我们可以使用键盘或编码器完全控制用户界面，而无需触摸板或鼠标。它的作用类似于 PC 上的 TAB 键，可以在应用程序或网页中选择元素。

14.2.1 组

想要通过键盘或编码器控制的对象需要添加到 **Group** 中。在每个组中，只有一个集中的对象可以接收按下的键或编码器的动作。例如，如果将文本区域作为焦点，并且在键盘上按了某个字母，则将发送键并将其插入到文本区域中。同样，如果将滑块聚焦，然后按向左或向右箭头，则滑块的值将被更改。

需要将输入设备与组关联。一台输入设备只能将键发送到一组，但一组也可以从多个输入设备接收数据。

要创建组，请使用 `lv_group_t * g = lv_group_create()`，然后将对象添加到组中，请使用 `lv_group_add_obj(g, obj)`。

要将组与输入设备关联，请使用 `lv_indev_set_group(indev, g)`，其中 `indev` 是 `lv_indev_drv_register()` 的返回值

14.2.2 按键

有一些预定义的键具有特殊含义：

- **LV_KEY_NEXT** 专注于下一个对象
- **LV_KEY_PREV** 专注于上一个对象
- **LV_KEY_ENTER** 触发器 `LV_EVENT_PRESSED/CLICKED/LONG_PRESSED` 等事件
- **LV_KEY_UP** 增加值或向上移动
- **LV_KEY_DOWN** 减小值或向下移动
- **LV_KEY_RIGHT** 增加值或向右移动
- **LV_KEY_LEFT** 减小值或向左移动
- **LV_KEY_ESC** 关闭或退出 (例如，关闭下拉列表)
- **LV_KEY_DEL** 删除 (例如，“文本”区域中右侧的字符)
- **LV_KEY_BACKSPACE** 删除左侧的字符 (例如，在文本区域中)

- **LV_KEY_HOME** 转到开头/顶部 (例如, 在“文本”区域中)
- **LV_KEY_END** 转到末尾 (例如, 在“文本”区域中)

最重要的特殊键是 **LV_KEY_NEXT/PREV** , **LV_KEY_ENTER** 和 **LV_KEY_UP/DOWN/LEFT/RIGHT** 。在 `read_cb` 函数中, 应将某些键转换为这些特殊键, 以便在组中导航并与所选对象进行交互。

通常, 仅使用 **LV_KEY_LEFT/RIGHT** 就足够了, 因为大多数对象都可以用它们完全控制。

对于编码器, 应仅使用 **LV_KEY_LEFT** , **LV_KEY_RIGHT** 和 **LV_KEY_ENTER** 。

14.2.3 编辑和浏览模式

由于键盘有很多键, 因此很容易在对象之间导航并使用键盘进行编辑。但是, 编码器的“键”数量有限, 因此很难使用默认选项进行导航。创建导航和编辑是为了避免编码器出现此问题。

在导航模式下, 编码器 **LV_KEY_LEFT/RIGHT** 转换为 **LV_KEY_NEXT/PREV** 。因此, 将通过旋转编码器选择下一个或上一个对象。按 **LV_KEY_ENTER** 将更改为编辑模式。

在“编辑”模式下, **LV_KEY_NEXT/PREV** 通常用于编辑对象。根据对象的类型, 短按或长按可将其 **LV_KEY_ENTER** 更改回导航模式。通常, 无法按下的对象 (如 **Slider**) 会在短按时离开“编辑”模式。但是, 对于具有短单击含义的对象 (例如 **Button**), 需要长按。

14.2.4 样式

如果通过触摸板单击对象或通过编码器或键盘将其聚焦, 则转到 **LV_STATE_FOCUSED** 。因此, 将重点应用样式。

如果对象进入编辑模式, 它将进入 **LV_STATE_FOCUSED|LV_STATE_EDITED** 状态, 因此将显示这些样式属性。

14.3 相关 API

TODO

LVGL 中显示的基本概念在 [显示接口](#) 部分中进行了说明。因此，在进一步阅读之前，请先阅读 [显示接口](#) 部分。

15.1 多种显示支持

在 LVGL 中，可以有多个显示，每个显示都有自己的驱动程序和对象。唯一的限制是，每个显示器都必须具有相同的色深（如中所述 `LV_COLOR_DEPTH`）。如果在这方面显示有所不同，则可以在驱动程序中将渲染的图像转换为正确的格式 `flush_cb`。

创建更多的显示很容易：只需初始化更多的显示缓冲区并为每个显示注册另一个驱动程序。创建 UI 时，用于 `lv_disp_set_default (disp)` 告诉库在其上创建对象的显示。

为什么要多显示器支持？这里有些例子：

- 具有带有本地 UI 的“常规”TFT 显示屏，并根据需要在 VNC 上创建“虚拟”屏幕。（需要添加 VNC 驱动程序）。
- 具有大型 TFT 显示屏和小型单色显示屏。
- 在大型仪器或技术中具有一些较小且简单的显示器。
- 有两个大型 TFT 显示屏：一个用于客户，一个用于店员。

15.1.1 仅使用一个显示器

使用更多显示器可能很有用，但在大多数情况下，并非必需。因此，如果仅注册一个显示器，则整个多显示器概念将被完全隐藏。默认情况下，最后创建的（唯一的）显示用作默认设置。

`lv_scr_act()`，`lv_scr_load(scr)`，`lv_layer_top()`，`lv_layer_sys()`，`LV_HOR_RES` 以及 `LV_VER_RES` 始终应用在最后创建的（默认）屏幕上。如果将 `NULL` 作为 `disp` 参数传递给显示相关功能，通常将使用默认显示。例如。`lv_disp_trig_activity(NULL)` 将在默认屏幕上触发用户活动。（请参见下面的非活动状态）。

15.1.2 镜面展示

要将显示器的图像镜像到另一台显示器，则无需使用多显示器支持。只需将在 `drv.flush_cb` 中接收的缓冲区也转移到另一个显示器。

15.1.3 分割影像

可以从较小的显示创建较大的显示。可以如下创建它：

1. 将显示器的分辨率设置为大显示器的分辨率。
2. 在中 `drv.flush_cb`，截断并修改 `area` 每个显示的参数。
3. 将缓冲区的内容发送到带有截断区域的每个显示。

15.1.4 屏幕

每个显示器都有每组屏幕和屏幕上的对象。

确保不要混淆显示和屏幕：

- **显示**是绘制像素的物理硬件。
- **屏幕**是与特定显示器关联的高级根对象。一个显示器可以具有与其关联的多个屏幕，但反之则不然。

屏幕可以视为没有父级的最高级别的容器。屏幕的大小始终等于其显示，屏幕的大小始终为 (0;0)。因此，无法更改屏幕坐标，即，不能在屏幕上使用 `lv_obj_set_pos()`，`lv_obj_set_size()` 或类似功能。

可以从任何对象类型创建屏幕，但是，两种最典型的类型是‘**基础对象**’和‘**图像**’（用于创建墙纸）。

要创建屏幕，请使用 `lv_obj_t * scr = lv_<type>_create(NULL, copy)`。复制可以是另一个屏幕来复制它。

要加载屏幕，请使用 `lv_scr_load(scr)`。要获取活动屏幕，请使用 `lv_scr_act()`。这些功能在默认显示屏上起作用。如果要指定要处理的显示器，请使用 `lv_disp_get_scr_act(disp)` 和 `lv_disp_load_scr(disp, scr)`。屏幕也可以加载动画。在这里阅读更多。

可以使用“`lv_obj_del(scr)`”删除屏幕，但请确保不删除当前加载的屏幕。

15.1.5 透明屏幕

通常，屏幕的不透明度是 LV_OPA_COVER 为其子级提供坚实的背景。如果不是这种情况（不透明度 <100%），则显示器的背景颜色或图像将可见。有关更多详细信息，请参见显示背景部分。如果显示器的背景不透明性也不是，则 LV_OPA_COVER LVGL 不会绘制纯色背景。

此配置（透明屏幕和显示器）可用于创建 OSD 菜单，例如在其中将视频播放到下层，并在上层创建菜单。

为了处理透明显示器，LVGL 需要使用特殊（较慢）的颜色混合算法，因此需要使用 LV_COLOR_SCREEN_TRANSP 启用此功能 lv_conf.h。由于此模式在像素的 Alpha 通道上运行，因此也是必需的。32 位颜色的 Alpha 通道在没有对象的情况下将为 0，在有实体对象的情况下将为 255。LV_COLOR_DEPTH = 32

总而言之，要启用透明屏幕和显示以创建类似于 OSD 菜单的 UI：

- 在 lv_conf.h 中启用 LV_COLOR_SCREEN_TRANSP
- 请务必使用 LV_COLOR_DEPTH 32
- 将屏幕的不透明度设置为 LV_OPA_TRANSP 例如 lv_obj_set_style_local_bg_opa(lv_scr_act(), LV_OBJMASK_PART_MAIN, LV_STATE_DEFAULT, LV_OPA_TRANSP)
- 使用 lv_disp_set_bg_opa(NULL, LV_OPA_TRANSP); 将显示不透明度设置为 LV_OPA_TRANSP

15.1.6 显示器功能

不活跃

在每个显示器上测量用户的不活动状态。每次使用输入设备（如果与显示器关联）都被视为一项活动。要获取自上一次活动以来经过的时间，请使用 lv_disp_get_inactive_time(disp)。如果传递了 NULL，则所有显示（不是默认显示）将返回整体最小的不活动时间。

可以使用手动触发活动 lv_disp_trig_activity(disp)。如果 disp 为 NULL，则将使用默认屏幕（并非所有显示）。

背景

每个显示都有背景色，背景图像和背景不透明度属性。当当前屏幕是透明的或未定位为覆盖整个显示器时，它们将变为可见。

背景色是填充显示的一种简单颜色。可以使用“lv_disp_set_bg_color(disp, color)”进行调整；

背景图像是用作墙纸的文件路径或指向 lv_img_dsc_t 变量（转换后的图像）的指针。可以使用 lv_disp_set_bg_color(disp, & my_img); 进行设置。如果设置了背景图片（非 NULL），则背景将不会填充 bg_color。

可以使用 lv_disp_set_bg_opa(disp, opa) 调整背景颜色或图像的不透明度。

这些函数的“disp”参数可以为 NULL，以将其引用到默认显示。

15.1.7 色彩

颜色模块处理所有与颜色相关的功能，例如更改颜色深度，从十六进制代码创建颜色，在颜色深度之间转换，混合颜色等。

颜色模块定义了以下变量类型：

- **lv_color1_t** 存储单色。为了兼容性，它也具有 R，G，B 字段，但它们始终是相同的值（1 个字节）
- **lv_color8_t** 用于存储 8 位颜色（1 字节）的 R（3 位），G（3 位），B（2 位）的结构体。
- **lv_color16_t** 用于存储 16 位颜色（2 字节）的 R（5 位），G（6 位），B（5 位）的结构体。
- **lv_color32_t** 用于存储 24 位颜色（4 字节）的 R（8 位），G（8 位），B（8 位）的结构体。
- **lv_color_t** 等于 lv_color1/8/16/24_t 根据颜色深度设置
- **lv_color_int_t** uint8_t，uint16_t 或 uint32_t 根据颜色深度设置。用于从纯数字构建颜色阵列。
- **lv_opa_t** 一种简单的 uint8_t 类型，用于描述不透明度。

lv_color_t，lv_color1_t，lv_color8_t，lv_color16_t 和 lv_color32_t 类型具有四个字段：

- **ch.red** 红色通道
- **ch.green** 绿色通道
- **ch.blue** 蓝色通道
- **full** 红 + 绿 + 蓝为一个数字

通过将 LV_COLOR_DEPTH 定义设置为 1（单色），8、16 或 32，可以在 lv_conf.h 中设置当前颜色深度。

转换颜色

可以将一种颜色从当前颜色深度转换为另一种颜色。转换器函数以数字返回，因此必须使用完整字段：

```

1  lv_color_t c;
2  c.red   = 0x38;
3  c.green = 0x70;
4  c.blue  = 0xCC;
5
6  lv_color1_t c1;
7  c1.full = lv_color_to1(c);      /*Return 1 for light colors, 0 for dark colors*/
8
9  lv_color8_t c8;
10 c8.full = lv_color_to8(c);      /*Give a 8 bit number with the converted color*/
11
12 lv_color16_t c16;
13 c16.full = lv_color_to16(c); /*Give a 16 bit number with the converted color*/
14

```

(下页继续)

(续上页)

```

15     lv_color32_t c24;
16     c32.full = lv_color_to32(c);    /*Give a 32 bit number with the converted color*/

```

交换 16 种颜色

可以在 `lv_conf.h` 中设置 `LV_COLOR_16_SWAP` 以交换 RGB565 颜色的字节。如果通过 SPI 等面向字节的接口发送 16 位颜色，则很有用。

由于 16 位数字以 Little Endian 格式存储（低位地址的低位字节），因此接口将首先发送低位字节。但是，显示通常首先需要较高的字节。字节顺序不匹配会导致色彩严重失真。

创建和混合颜色

可以使用 `LV_COLOR_MAKE` 宏以当前颜色深度创建颜色。它使用 3 个参数（红色，绿色，蓝色）作为 8 位数字。例如，创建浅红色：`my_color = COLOR_MAKE(0xFF, 0x80, 0x80)`。

颜色也可以从十六进制代码创建：`my_color = lv_color_hex(0x288ACF)` 或 `my_color = lv_folor_hex3(0x28C)`。

可以混合两种颜色：`mixed_color = lv_color_mix(color1, color2, ratio)`。比例可以是 0..255。0 表示全彩色 2，255 表示全彩色 1。

也可以使用 `lv_color_hsv_to_rgb(hue, saturation, value)` 从 HSV 空间创建颜色。色相应应在 0..360 范围内，饱和度和值应在 0..100 范围内。

不透明度

为了描述不透明度，创建了 `lv_opa_t` 类型作为 `uint8_t` 的包装。还介绍了一些定义：

- **LV_OPA_TRANSP** 值：0，表示不透明度使颜色完全透明
- **LV_OPA_10** 值：25，表示颜色仅覆盖一点
- **LV_OPA_20 ... OPA_80** 比较常用
- **LV_OPA_90** 值：229，表示几乎完全覆盖的颜色
- **LV_OPA_COVER** 值：255，表示颜色完全覆盖

还可以将 `LV_OPA_*` 定义 `lv_color_mix()` 作为比率使用。

内置颜色

颜色模块定义了最基本的颜色，例如：

- #FFFFFF LV_COLOR_WHITE
- #000000 LV_COLOR_BLACK
- #808080 LV_COLOR_GRAY
- #c0c0c0 LV_COLOR_SILVER
- #ff0000 LV_COLOR_RED
- #800000 LV_COLOR_MAROON
- #00ff00 LV_COLOR_LIME
- #008000 LV_COLOR_GREEN
- #808000 LV_COLOR_OLIVE
- #0000ff LV_COLOR_BLUE
- #000080 LV_COLOR_NAVY
- #008080 LV_COLOR_TEAL
- #00ffff LV_COLOR_CYAN
- #00ffff LV_COLOR_AQUA
- #800080 LV_COLOR_PURPLE
- #ff00ff LV_COLOR_MAGENTA
- #ffa500 LV_COLOR_ORANGE
- #ffff00 LV_COLOR_YELLOW

以及 LV_COLOR_WHITE （全白）。

15.2 相关 API

TODO

CHAPTER 16

字体 (Fonts)

在 LVGL 中，字体是位图和呈现字母（字形）图像所需的其他信息的集合。字体存储在 `lv_font_t` 变量中，可以在样式的 `text_font` 字段中进行设置。例如：

```
1 lv_style_set_text_font(&my_style, LV_STATE_DEFAULT, &lv_font_montserrat_28);  /  
↪ *Set a larger font*/
```

字体具有 **bpp（每像素位数）** 属性。它显示了多少位用于描述字体中的像素。为像素存储的值确定像素的不透明度。这样，如果 **bpp** 较高，则字母的边缘可以更平滑。可能的 **bpp** 值是 1、2、4 和 8（值越高表示质量越好）。

bpp 还会影响存储字体所需的内存大小。例如，**bpp = 4** 使字体比 **bpp = 1** 大近 4 倍。

16.1 Unicode 支持

LVGL 支持 **UTF-8** 编码的 Unicode 字符。需要配置的编辑器，以将的代码/文本保存为 UTF-8（通常是默认值），并确保在 `lv_conf.h` 中将 `LV_TXT_ENC` 其设置为：`LV_TXT_ENC_UTF8`。（这是默认值）测试一下试试：

```
1 lv_obj_t * label1 = lv_label_create(lv_scr_act(), NULL);  
2 lv_label_set_text(label1, LV_SYMBOL_OK);
```

如果一切正常，则应显示一个 ✓ 字符。

16.2 内置字体

有几种不同大小的内置字体，可以通过 `LV_FONT_...` 定义在 `lv_conf.h` 中启用。

16.2.1 普通字体

包含所有 ASCII 字符，度数符号 (U + 00B0)，项目符号 (U + 2022) 和内置符号 (请参见下文)。

- `LV_FONT_MONTERRAT_12` 12 px font
- `LV_FONT_MONTERRAT_14` 14 px font
- `LV_FONT_MONTERRAT_16` 16 px font
- `LV_FONT_MONTERRAT_18` 18 px font
- `LV_FONT_MONTERRAT_20` 20 px font
- `LV_FONT_MONTERRAT_22` 22 px font
- `LV_FONT_MONTERRAT_24` 24 px font
- `LV_FONT_MONTERRAT_26` 26 px font
- `LV_FONT_MONTERRAT_28` 28 px font
- `LV_FONT_MONTERRAT_30` 30 px font
- `LV_FONT_MONTERRAT_32` 32 px font
- `LV_FONT_MONTERRAT_34` 34 px font
- `LV_FONT_MONTERRAT_36` 36 px font
- `LV_FONT_MONTERRAT_38` 38 px font
- `LV_FONT_MONTERRAT_40` 40 px font
- `LV_FONT_MONTERRAT_42` 42 px font
- `LV_FONT_MONTERRAT_44` 44 px font
- `LV_FONT_MONTERRAT_46` 46 px font
- `LV_FONT_MONTERRAT_48` 48 px font

16.2.2 特殊字体

- LV_FONT_MONTERRAT_12_SUBPX 与常规 12 像素字体相同，但具有亚像素渲染
- LV_FONT_MONTERRAT_28_COMPRESSED 与普通的 28 px 字体相同，但压缩字体为 3 bpp
- LV_FONT_DEJAVU_16_PERSIAN_HEBREW 正常范围内的 16 像素字体 + 希伯来语，阿拉伯语，Perisan 字母及其所有形式
- LV_FONT_SIMSUN_16_CJK 16 px 字体，具有正常范围 + 1000 个最常见的 CJK 部首
- LV_FONT_UNSCII_8 仅包含 ASCII 字符的 8 px 像素完美字体

内置字体是全局变量，其名称像 lv_font_montserrat_16 一样，代表 16 px 高的字体。要以某种样式使用它们，只需添加一个指向如上所示的字体变量的指针。

内置字体的 bpp = 4，包含 ASCII 字符并使用 Montserrat 字体。

除了 ASCII 范围，以下符号也从 FontAwesome 字体添加到内置字体中。

这些符号可以这样使用：

```
lv_label_set_text(my_label, LV_SYMBOL_OK);
```

或与字符串一起使用：

```
lv_label_set_text(my_label, LV_SYMBOL_OK "Apply");
```

或更多个符号一起使用：

```
lv_label_set_text(my_label, LV_SYMBOL_OK LV_SYMBOL_WIFI LV_SYMBOL_PLAY);
```

16.3 特殊功能

16.3.1 双向支持

大多数语言使用从左到右（简称 LTR）的书写方向，但是某些语言（例如希伯来语，波斯语或阿拉伯语）使用从右到左（简称 RTL）的书写方向。

LVGL 不仅支持 RTL 文本，而且还支持混合（又称双向，BiDi）文本渲染。一些例子：

可以通过 lv_conf.h 中的 LV_USE_BIDI 启用 BiDi 支持

所有文本都有一个基本方向（LTR 或 RTL），该方向确定一些渲染规则和文本的默认对齐方式（左或右）。但是，在 LVGL 中，基本方向不仅适用于标签。这是可以为每个对象设置的常规属性。如果未设置，则它将从父级继承。因此，设置屏幕的基本方向就足够了，每个对象都会继承它。

屏幕的默认基本方向可以通过 lv_conf.h 中的 LV_BIDI_BASE_DIR_DEF 设置，其他对象从其父对象继承基本方向。


























































	LV_SYMBOL_AUDIO		LV_SYMBOL_WARNING
	LV_SYMBOL_VIDEO		LV_SYMBOL_SHUFFLE
	LV_SYMBOL_LIST		LV_SYMBOL_UP
	LV_SYMBOL_OK		LV_SYMBOL_DOWN
	LV_SYMBOL_CLOSE		LV_SYMBOL_LOOP
	LV_SYMBOL_POWER		LV_SYMBOL_DIRECTORY
	LV_SYMBOL_SETTINGS		LV_SYMBOL_UPLOAD
	LV_SYMBOL_TRASH		LV_SYMBOL_CALL
	LV_SYMBOL_HOME		LV_SYMBOL_CUT
	LV_SYMBOL_DOWNLOAD		LV_SYMBOL_COPY
	LV_SYMBOL_DRIVE		LV_SYMBOL_SAVE
	LV_SYMBOL_REFRESH		LV_SYMBOL_CHARGE
	LV_SYMBOL_MUTE		LV_SYMBOL_PASTE
	LV_SYMBOL_VOLUME_MID		LV_SYMBOL_BELL
	LV_SYMBOL_VOLUME_MAX		LV_SYMBOL_KEYBOARD
	LV_SYMBOL_IMAGE		LV_SYMBOL_GPS
	LV_SYMBOL_EDIT		LV_SYMBOL_FILE
	LV_SYMBOL_PREV		LV_SYMBOL_WIFI
	LV_SYMBOL_PLAY		LV_SYMBOL_BATTERY_FULL
	LV_SYMBOL_PAUSE		LV_SYMBOL_BATTERY_3
	LV_SYMBOL_STOP		LV_SYMBOL_BATTERY_2
	LV_SYMBOL_NEXT		LV_SYMBOL_BATTERY_1
	LV_SYMBOL_EJECT		LV_SYMBOL_BATTERY_EMPTY
	LV_SYMBOL_LEFT		LV_SYMBOL_USB
	LV_SYMBOL_RIGHT		LV_SYMBOL_BLUETOOTH
	LV_SYMBOL_PLUS		LV_SYMBOL_BACKSPACE
	LV_SYMBOL_MINUS		LV_SYMBOL_SD_CARD
	LV_SYMBOL_EYE_OPEN		LV_SYMBOL_NEW_LINE
	LV_SYMBOL_EYE_CLOSE		

图 1: LVGL 的内置符号

The names of these states in Arabic
are مصر, البحرين and الكويت respectively.

The title is مفتاح معايير الويب!

图 2: LVGL 的双向文字范例

要设置对象的基本方向，请使用 `lv_obj_set_base_dir(obj, base_dir)`。可能的的基本方向是：

- **LV_BIDI_DIR_LTR**: 从左到右基本方向
- **LV_BIDI_DIR_RTL**: 从右到左基本方向
- **LV_BIDI_DIR_AUTO**: A 自动检测基准方向
- **LV_BIDI_DIR_INHERIT**: 从父级继承基本方向（非屏幕对象的默认方向）

此列表总结了 RTL 基本方向对对象的影响：

默认情况下在右侧创建对象

- `lv_tabview`: displays tabs from right to left
- `lv_checkbox`: Show the box on the right
- `lv_btnmatrix`: Show buttons from right to left
- `lv_list`: Show the icon on the right
- `lv_dropdown`: Align the options to the right
- `lv_table`, `lv_btnmatrix`, `lv_keyboard`, `lv_tabview`, `lv_dropdown`, `lv_roller` 中的文本为“BiDi 处理”，可以正确显示

16.3.2 阿拉伯语和波斯语支持

显示阿拉伯字符和波斯字符有一些特殊规则：字符的形式取决于它们在文本中的位置。如果隔离，开始，中间或结束位置，则需要使用同一字母的不同形式。除了这些合取规则外，还应考虑其他规则。

如果启用了 `LV_USE_ARABIC_PERSIAN_CHARS`，则 LVGL 支持应用这些规则。

但是，存在一些限制：

- 仅支持显示文本（例如在标签上），文本输入（例如文本区域）不支持此功能
- 静态文本（即 `const`）不会被处理。例如，由 `lv_label_set_text()` 设置的文本将被处理为“阿拉伯语”，但 `lv_label_set_text_static()` 不会。

- 文本获取函数（例如 `lv_label_get_text()`）将返回处理后的文本。

16.3.3 亚像素渲染

亚像素渲染意味着通过在红色，绿色和蓝色通道而不是像素级别上渲染将水平分辨率提高三倍。它利用了每个像素的物理颜色通道的位置。这样可以产生更高质量的字母抗锯齿。在这里学习更多。亚像素渲染需要生成具有特殊设置的字体：

- 在网络转换器勾选 Subpixel 框
- 在命令行工具中使用 `--lcd` 标志。请注意，生成的字体大约需要 3 倍的内存。

仅当像素的颜色通道具有水平布局时，子像素渲染才有效。也就是说，R，G，B 通道彼此相邻而不位于彼此之上。颜色通道的顺序也需要与库设置匹配。默认情况下，LVGL 采用 RGB 顺序，但是可以通过在 `lv_conf.h` 中将 `LV_SUBPX_BGR` 设置为 **1** 来交换它。

16.3.4 压缩字体

- 字体的位图可以通过以下方式压缩
- 勾选 Compressed 在线转换器中的复选框
- 不将 `--no-compress` 标志传递给脱机转换器（默认情况下应用压缩）

较大的字体和较高的 **bpp** 压缩更有效。但是，渲染压缩字体的速度要慢 30%。因此，建议仅压缩用户界面的最大字体，因为 **大** - 他们需要最多的内存 - 他们可以更好地压缩 - 并且它们的使用频率可能不如中等大小的字体。（因此性能成本较小）

16.4 添加新字体

有几种方法可以向项目中添加新字体：

1. 最简单的方法是使用在线字体转换器 (<https://lvgl.io/tools/fontconverter>)。只需设置参数，单击“转换”按钮，将字体复制到的项目中并使用它。**请务必仔细阅读该网站上提供的步骤，否则转换时会出错。**
2. 使用脱机字体转换器 (https://github.com/lvgl/lv_font_conv)。（需要安装 Node.js）
3. 如果要创建类似内置字体（Roboto 字体和符号）但大小和/或范围不同的东西，可以在 `lvgl/scripts/built_in_font` 文件夹中使用 `built_in_font_gen.py` 脚本。（它需要安装 Python 和 `lv_font_conv`）

要在文件中声明字体，请使用 `LV_FONT_DECLARE(my_font_name)`。

要使字体在全局范围内可用（如内置字体），请将它们添加到 `lv_conf.h` 中的 `LV_FONT_CUSTOM_DECLARE` 中。

16.5 添加新符号

内置符号是从 FontAwesome 字体创建的。

1. 在 <https://fontawesome.com> 上搜索符号。例如 USB 符号。复制它的 Unicode ID, 在这种情况下为 0xf287。
2. 打开在线字体转换器。添加添加 FontAwesome.woff。。
3. 设置名称, 大小, BPP 等参数。将使用此名称在代码中声明和使用字体。
4. 将符号的 Unicode ID 添加到范围字段。例如。USB 符号为 0xf287。可以用, 枚举更多符号。
5. 转换字体并将其复制到项目。确保编译字体的.c 文件。
6. 使用 `extern lv_font_t my_font_name11` 声明字体; 或者只是 `LV_FONT_DECLARE(my_font_name);`

16.5.1 使用符号

1. 将 Unicode 值转换为 UTF8。可以在此网站上进行操作。对于 0xf287, 十六进制 UTF-8 字节为 EF 8A 87。
2. 根据 UTF8 值创建定义: `#define MY_USB_SYMBOL "\xEF\x8A\x87"`
3. 创建一个标签并设置文本。例如: `lv_label_set_text(label, MY_USB_SYMBOL)`

注- `lv_label_set_text(label, MY_USB_SYMBOL)` 使用 `style.text.font` 属性中定义的字体搜索此符号。要使用该符号, 可能需要对其进行更改。例如 `style.text.font = my_font_name`

16.6 在运行时加载字体

`lv_font_load` 可用于从文件中加载字体。要加载的字体需要具有特殊的二进制格式。(不是 TTF 或 WOFF)。使用带有 `--format bin` 选项的 `lv_font_conv` 生成与 LVGL 兼容的字体文件。

请注意, 要加载字体, 需要启用 LVGL 的文件系统, 并需要添加驱动程序。

范例:

```

1  lv_font_t * my_font;
2  my_font = lv_font_load(X/path/to/my_font.bin);
3
4  /*Use the font*/
5
6  /*Free the font if not required anymore*/
7  lv_font_free(my_font);

```

16.7 添加新的字体引擎

LVGL 的字体界面设计得非常灵活。不需要使用 LVGL 的内部字体引擎，但可以添加自己的字体。例如，使用 FreeType 从 TTF 字体实时渲染字形，或使用外部闪存存储字体的位图，并在库需要它们时读取它们。

可以在 `lv_freetype` 库中找到使用 FreeType 的传统。

为此，需要创建一个自定义 `lv_font_t` 变量：

```

1      /*Describe the properties of a font*/
2      lv_font_t my_font;
3      my_font.get_glyph_dsc = my_get_glyph_dsc_cb;          /*Set a callback to get info
↳about glyphs*/
4      my_font.get_glyph_bitmap = my_get_glyph_bitmap_cb;    /*Set a callback to get
↳bitmap of a glyph*/
5      my_font.line_height = height;                         /*The real line height where
↳any text fits*/
6      my_font.base_line = base_line;                       /*Base line measured from the
↳top of line_height*/
7      my_font.dsc = something_required;                     /*Store any implementation
↳specific data here*/
8      my_font.user_data = user_data;                       /*Optionally some extra user
↳data*/
9
10     ...
11
12     /* Get info about glyph of `unicode_letter` in `font` font.
13      * Store the result in `dsc_out`.
14      * The next letter (`unicode_letter_next`) might be used to calculate the width
↳required by this glyph (kerning)
15     */
16     bool my_get_glyph_dsc_cb(const lv_font_t * font, lv_font_glyph_dsc_t * dsc_out,
↳uint32_t unicode_letter, uint32_t unicode_letter_next)
17     {
18         /*Your code here*/
19
20         /* Store the result.
21          * For example ...
22          */
23         dsc_out->adv_w = 12;                                /*Horizontal space required by the glyph in
↳[px]*/
24         dsc_out->box_h = 8;                                /*Height of the bitmap in [px]*/
25         dsc_out->box_w = 6;                                /*Width of the bitmap in [px]*/
26         dsc_out->ofs_x = 0;                                /*X offset of the bitmap in [pf]*/
27         dsc_out->ofs_y = 3;                                /*Y offset of the bitmap measured from the as
↳line*/

```

(下页继续)

(续上页)

```
28         dsc_out->bpp    = 2;           /*Bits per pixel: 1/2/4/8*/
29
30         return true;                     /*true: glyph found; false: glyph was not_
↪found*/
31     }
32
33     /* Get the bitmap of `unicode_letter` from `font`. */
34     const uint8_t * my_get_glyph_bitmap_cb(const lv_font_t * font, uint32_t unicode_
↪letter)
35     {
36         /* Your code here */
37
38         /* The bitmap should be a continuous bitstream where
39          * each pixel is represented by `bpp` bits */
40
41         return bitmap;                  /*Or NULL if not found*/
42     }
```

16.8 相关 API

TODO

图像可以是存储位图本身和一些元数据的文件或变量。

17.1 储存图片

我们可以将图像存储在两个位置

- 作为内部存储器（RAM 或 ROM）中的变量
- 作为文件

17.2 变量

内部存储在变量中的图像主要由具有以下字段的 `lv_img_dsc_t` 结构组成：

- 标头
 - `cf` 颜色格式。见下文
 - `w` 宽度（以像素为单位）（ ≤ 2048 ）
 - `h` 高度（以像素为单位）（ ≤ 2048 ）
 - 始终为零 3 位需要始终为零
 - `reserved` 保留以备将来使用
- 指向存储图像本身的数组的数据指针

- `data_size` 数据的长度（以字节为单位）

这些通常作为 C 文件存储在项目中。它们像其他任何常量数据一样链接到生成的可执行文件中。

17.3 文件

要处理文件，您需要将驱动器添加到 LVGL。简而言之，驱动器是在 LVGL 中注册的用于文件操作的功能的集合（打开，读取，关闭等）。您可以向标准文件系统（SD 卡上的 FAT32）添加接口，也可以创建简单的文件系统以从 SPI 闪存读取数据。在每种情况下，驱动器都只是一种将数据读取和/或写入内存的抽象。请参阅文件系统部分以了解更多信息。

存储为文件的图像未链接到生成的可执行文件中，并且在绘制之前必须将其读取到 RAM 中。结果，它们不像可变图像那样对资源友好。但是，它们更容易替换，而无需重新编译主程序。

17.4 色彩格式

lvgl 支持多种内置颜色格式：

- **LV_IMG_CF_TRUE_COLOR** 只需存储 RGB 颜色（使用 LVGL 配置的任何颜色深度）。
- **LV_IMG_CF_TRUE_COLOR_ALPHA** 与 **LV_IMG_CF_TRUE_COLOR** 类似，但它还会为每个像素添加一个 alpha（透明度）字节。
- **LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED** 与 **LV_IMG_CF_TRUE_COLOR** 类似，但是如果像素具有 **LV_COLOR_TRANSP**（在 `lv_conf.h` 中设置）颜色，则该像素将是透明的。
- **LV_IMG_CF_INDEXED_1/2/4/8BIT** 使用 2、4、16 或 256 色调色板，并以 1、2、4 或 8 位存储每个像素。
- **LV_IMG_CF_ALPHA_1/2/4/8BIT** 仅将 Alpha 值存储在 1、2、4 或 8 位上。像素采用 `style.image.color` 和设置的不透明度的颜色。源图像必须是 Alpha 通道。这对于类似于字体的位图是理想的（整个图像是一种颜色，但是您希望能够更改它）。

LV_IMG_CF_TRUE_COLOR 图像的字节按以下顺序存储。

对于 32 位色深：

- Byte 0: 蓝色 (Blue)
- Byte 1: 绿色 (Green)
- Byte 2: 红色 (Red)
- Byte 3: 透明度 (Alpha)

对于 16 位色深：

- Byte 0: 绿色 (Green) 3 位低位，蓝色 (Blue) 5 位
- Byte 1: 红色 (Red) 5 位，绿色 (Green) 3 位

- Byte 2: 透明度 (Alpha) 字节 (仅适用于 LV_IMG_CF_TRUE_COLOR_ALPHA)

对于 8 位色深:

- Byte 0: Red 3 bit, Green 3 bit, Blue 2 bit
- Byte 2: Alpha byte (only with LV_IMG_CF_TRUE_COLOR_ALPHA)

我们可以将图像以 Raw 格式存储, 以指示它不是内置的颜色格式, 并且需要使用外部 Image 解码器来解码图像。

- **LV_IMG_CF_RAW** 表示基本的原始图像 (例如 PNG 或 JPG 图像)。
- **LV_IMG_CF_RAW_ALPHA** 表示图像具有 Alpha, 并且为每个像素添加了 Alpha 字节。
- **LV_IMG_CF_RAW_CHROME_KEYED** 表示该图像已按 chrome 键锁定, 如上文 LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED 中所述。

17.5 添加和使用图像

可以通过下面两种方式将图像添加到 LVGL:

- 使用在线转换器
- 手动创建图像

17.5.1 在线转换器

在线图像转换器可在这里找到: <https://lvgl.io/tools/imageconverter>

通过在线转换器可以很容易地将图像转换并添加到 LVGL。

1. 准备好要转换的 BMP, PNG 或 JPG 图像。
2. 给图像起一个将在 LVGL 中使用的名称。
3. 选择颜色格式。
4. 选择所需的图像类型。选择二进制文件将生成一个 .bin 文件, 该文件必须单独存储并使用文件支持进行读取。选择一个变量将生成一个标准的 C 文件, 该文件可以链接到您的项目中。
5. 点击转换按钮。转换完成后, 浏览器将自动下载结果文件。

在转换器 C 数组 (变量) 中, 所有颜色深度 (1、8、16 或 32) 的位图都包含在 C 文件中, 但是只有与 lv_conf.h 中的 LV_COLOR_DEPTH 匹配的颜色深度才会实际链接到生成的可执行文件。

如果是二进制文件, 则需要指定所需的颜色格式:

- RGB332 用于 8 位色深
- RGB565 用于 16 位色深
- RGB565 交换为 16 位颜色深度 (交换了两个字节)

- RGB888 用于 32 位色深

17.5.2 手动创建图像

如果要在运行时生成图像，则可以制作图像变量以使用 LVGL 显示它。例如：

```
1  uint8_t my_img_data[] = {0x00, 0x01, 0x02, ...};
2
3  static lv_img_dsc_t my_img_dsc = {
4      .header.always_zero = 0,
5      .header.w = 80,
6      .header.h = 60,
7      .data_size = 80 * 60 * LV_COLOR_DEPTH / 8,
8      .header.cf = LV_IMG_CF_TRUE_COLOR,           /*Set the color format*/
9      .data = my_img_data,
10 };
```

如果颜色格式为 LV_IMG_CF_TRUE_COLOR_ALPHA，则可以将 data_size 设置为 80 * 60 * LV_IMG_PX_SIZE_ALPHA_BYTE。

在运行时创建和显示图像的另一个（可能更简单）的选项是使用 Canvas 对象。

17.5.3 使用图片

在 LVGL 中使用图像的最简单方法是使用 lv_img 对象显示图像：

```
1  lv_obj_t * icon = lv_img_create(lv_scr_act(), NULL);
2
3  /*From variable*/
4  lv_img_set_src(icon, &my_icon_dsc);
5
6  /*From file*/
7  lv_img_set_src(icon, "S:my_icon.bin");
```

如果图像是使用在线转换器转换的，则应使用 LV_IMG_DECLARE (my_icon_dsc) 在文件中声明要使用它的位置。

17.5.4 图像解码器

如您在“颜色格式”部分中所见，LVGL 支持多种内置图像格式。在许多情况下，这些都是您所需要的。LVGL 不直接支持通用图像格式，例如 PNG 或 JPG。

要处理非内置图像格式，您需要使用外部库，并通过图像解码器接口将它们附加到 LVGL。

图像解码器包含 4 个回调：

- **info** 获取有关图像的一些基本信息（宽度，高度和颜色格式）。
- **open** 打开图像：存储解码后的图像或将其设置为 NULL 以指示可以逐行读取图像。
- **read** 如果打开未完全打开图像，则此功能应从给定位置提供一些解码数据（最多 1 行）。
- **close** 关闭打开的图像，释放分配的资源。

17.5.5 自定义图像格式

创建自定义图像的最简单方法是使用在线图像转换器，并将 Raw，Raw 设置为 alpha 或 chrome 键设置为 Raw。它只会占用您上传的二进制文件的每个字节，并将其写为图像“位图”。然后，您需要连接一个图像解码器，它将解析该位图并生成真实的可渲染位图。

header.cf 将分别为 LV_IMG_CF_RAW, LV_IMG_CF_RAW_ALPHA 或 LV_IMG_CF_RAW_CHROME_KEYED。您应该根据需求选择正确的格式：完全不透明的图像，使用 Alpha 通道或使用色度键。

解码后，原始格式被库视为真彩色。换句话说，图像解码器必须根据 [# color-formats]（颜色格式）部分中所述的格式将 Raw 图像解码为 True color。

如果要创建自定义图像，则应使用 LV_IMG_CF_USER_ENCODED_0..7 颜色格式。但是，该库只能以 True color 格式（或 Raw，但最终应该以 True color 格式）绘制图像。因此，库不知道 LV_IMG_CF_USER_ENCODED_... 格式，因此应从 [# color-formats]（颜色格式）部分将其解码为已知格式之一。可以先将图像解码为非真彩色格式，例如 LV_IMG_INDEXED_4BITS，然后调用内置的解码器函数将其转换为真彩色。

对于用户编码格式，应根据新格式更改打开功能（dsc-> header.cf）中的颜色格式。

17.5.6 注册图像解码器

这是使 LVGL 与 PNG 图像配合使用的示例。

首先，需要先创建一个新的图像解码器并设置一些功能来打开/关闭 PNG 文件。它看起来应该像这样：

```
1  /*Create a new decoder and register functions */
2  lv_img_decoder_t * dec = lv_img_decoder_create();
3  lv_img_decoder_set_info_cb(dec, decoder_info);
4  lv_img_decoder_set_open_cb(dec, decoder_open);
5  lv_img_decoder_set_close_cb(dec, decoder_close);
6
```

(下页继续)

(续上页)

```

7  /**
8   * Get info about a PNG image
9   * @param decoder pointer to the decoder where this function belongs
10  * @param src can be file name or pointer to a C array
11  * @param header store the info here
12  * @return LV_RES_OK: no error; LV_RES_INV: can't get the info
13  */
14  static lv_res_t decoder_info(lv_img_decoder_t * decoder, const void * src, lv_img_
↳header_t * header)
15  {
16      /*Check whether the type `src` is known by the decoder*/
17      if(is_png(src) == false) return LV_RES_INV;
18
19      /* Read the PNG header and find `width` and `height` */
20      ...
21
22      header->cf = LV_IMG_CF_RAW_ALPHA;
23      header->w = width;
24      header->h = height;
25  }
26
27  /**
28   * Open a PNG image and return the decoded image
29   * @param decoder pointer to the decoder where this function belongs
30   * @param dsc pointer to a descriptor which describes this decoding session
31   * @return LV_RES_OK: no error; LV_RES_INV: can't get the info
32   */
33  static lv_res_t decoder_open(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t *
↳dsc)
34  {
35
36      /*Check whether the type `src` is known by the decoder*/
37      if(is_png(src) == false) return LV_RES_INV;
38
39      /*Decode and store the image. If `dsc->img_data` is `NULL`, the `read_line`
↳function will be called to get the image data line-by-line*/
40      dsc->img_data = my_png_decoder(src);
41
42      /*Change the color format if required. For PNG usually 'Raw' is fine*/
43      dsc->header.cf = LV_IMG_CF_...
44
45      /*Call a built in decoder function if required. It's not required if `my_png_
↳decoder` opened the image in true color format.*/

```

(下页继续)

(续上页)

```

46     lv_res_t res = lv_img_decoder_built_in_open(decoder, dsc);
47
48     return res;
49 }
50
51 /**
52  * Decode `len` pixels starting from the given `x`, `y` coordinates and store
↳ them in `buf`.
53  * Required only if the "open" function can't open the whole decoded pixel array.
↳ (dsc->img_data == NULL)
54  * @param decoder pointer to the decoder the function associated with
55  * @param dsc pointer to decoder descriptor
56  * @param x start x coordinate
57  * @param y start y coordinate
58  * @param len number of pixels to decode
59  * @param buf a buffer to store the decoded pixels
60  * @return LV_RES_OK: ok; LV_RES_INV: failed
61  */
62     lv_res_t decoder_built_in_read_line(lv_img_decoder_t * decoder, lv_img_decoder_
↳ dsc_t * dsc, lv_coord_t x,
63
64     ↳ lv_coord_t y, lv_coord_t len, uint8_t * buf)
65     {
66         /*With PNG it's usually not required*/
67
68         /*Copy `len` pixels from `x` and `y` coordinates in True color format to `buf`
↳ */
69     ↳ */
70
71     }
72
73     /**
74     * Free the allocated resources
75     * @param decoder pointer to the decoder where this function belongs
76     * @param dsc pointer to a descriptor which describes this decoding session
77     */
78     static void decoder_close(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
79     {
80         /*Free all allocated data*/
81
82         /*Call the built-in close function if the built-in open/read_line was used*/
83         lv_img_decoder_built_in_close(decoder, dsc);
84     }

```

因此，总而言之：

在解码器信息 (decoder_info) 中，您应该收集有关图像的一些基本信息，并将其存储在标头中。

在 decoder_open 中，应尝试打开 dsc->src 指向的图像源。它的类型已经在 dsc->src_type == LV_IMG_SRC_FILE / VARIABLE 中。如果解码器不支持此格式/类型，请返回 LV_RES_INV。但是，如果可以打开图像，则应在 dsc->img_data 中设置指向已解码的真彩色图像的指针。如果格式已知，但您不想在图像解码时（例如没有内存）将 dsc->img_data = NULL 设置为调用 read_line 以获取像素。

在 decoder_close 中，您应该释放所有分配的资源。

coder_read 是可选的。解码整个图像需要额外的内存和一些计算开销。但是，如果可以解码图像的一行而不解码整个图像，则可以节省内存和时间。为了表明这一点，应该使用行读取功能，在打开功能中设置 dsc->img_data = NULL。

17.5.7 手动使用图像解码器

如果您尝试绘制原始图像（即使用 lv_img 对象），LVGL 将自动使用注册的图像解码器，但是您也可以手动使用它们。创建一个 lv_img_decoder_dsc_t 变量来描述解码会话，然后调用 lv_img_decoder_open ()。

```
1  lv_res_t res;
2  lv_img_decoder_dsc_t dsc;
3  res = lv_img_decoder_open(&dsc, &my_img_dsc, LV_COLOR_WHITE);
4
5  if(res == LV_RES_OK) {
6      /*Do something with `dsc->img_data`*/
7      lv_img_decoder_close(&dsc);
8  }
```

17.6 图像缓存

有时打开图像需要很多时间。连续解码 PNG 图像或从速度较慢的外部存储器加载图像会效率低下，并且不利于用户体验。

因此，LVGL 缓存给定数量的图像。缓存意味着一些图像将保持打开状态，因此 LVGL 可以从 dsc->img_data 快速访问它们，而无需再次对其进行解码。

当然，缓存图像会占用大量资源，因为它使用更多的 RAM（用于存储解码的图像）。LVGL 尝试尽可能地优化流程（请参见下文），但是您仍然需要评估这是否对您的平台有利。如果您有一个深度嵌入的目标，可以从相对较快的存储介质中解码小图像，则不建议使用图像缓存。

17.6.1 缓存大小

可以在 `lv_conf.h` 的 `LV_IMG_CACHE_DEF_SIZE` 中定义高速缓存条目的数量。默认值为 1，因此只有最近使用的图像将保持打开状态。

可以在运行时使用 `lv_img_cache_set_size (entry_num)` 更改缓存的大小。

17.6.2 图片价值

当使用的图像多于缓存条目时，LVGL 无法缓存所有图像。而是，库将关闭其中一个缓存的图像（以释放空间）。

为了决定关闭哪个图像，LVGL 使用它先前对打开图像所花费的时间进行的测量。保存打开速度较慢的图像的高速缓存条目被认为更有价值，并尽可能长时间地保留在高速缓存中。

如果要或需要覆盖 LVGL 的测量值，则可以在 `dsc->time_to_open = time_ms` 的解码器打开功能中手动设置打开时间的值，以提供更高或更低的值。（将其保留以让 LVGL 对其进行设置。）

每个缓存条目都有一个“生命”值。每次通过缓存打开图像时，所有条目的寿命都会减少，从而使它们更旧。使用缓存的图像时，其使用寿命会随着打开值的时间增加而增加，从而使其更加生动。

如果缓存中没有更多空间，则始终关闭寿命最小的条目。

17.6.3 内存使用情况

请注意，缓存的图像可能会不断消耗内存。例如，如果缓存了 3 个 PNG 图像，则它们在打开时将消耗内存。

因此，用户有责任确保有足够的 RAM 来缓存，即使同时是最大的图像。

17.6.4 清理缓存

假设您已将 PNG 图片加载到 `lv_img_dsc_t my_png` 变量中，并在 `lv_img` 对象中使用了它。如果图像已被缓存，然后您更改了基础 PNG 文件，则需要通知 LVGL 重新缓存图像。否则，没有简单的方法可以检测到基础文件已更改并且 LVGL 仍会绘制旧图像。

为此，请使用 `lv_img_cache_invalidate_src (& my_png)`。如果将 NULL 作为参数传递，则将清除整个缓存。

17.7 相关 API

TODO

文件系统 (File system)

LVGL 具有“文件系统” (File system) 抽象模块，使可以附加任何类型的文件系统。文件系统由驱动器号标识。例如，如果 SD 卡与字母 `S` 相关联，则可以访问 "S:path/to/file.txt" 之类的文件。

18.1 添加驱动程序

要添加驱动程序，lv_fs_drv_t 需要这样初始化：

```
1  lv_fs_drv_t drv;
2  lv_fs_drv_init(&drv);                                /*Basic initialization*/
3
4  drv.letter = 'S';                                     /*An uppercase letter to identify the_
↪drive */
5  drv.file_size = sizeof(my_file_object);              /*Size required to store a file object*/
6  drv.rddir_size = sizeof(my_dir_object);              /*Size required to store a directory_
↪object (used by dir_open/close/read)*/
7  drv.ready_cb = my_ready_cb;                           /*Callback to tell if the drive is_
↪ready to use */
8  drv.open_cb = my_open_cb;                             /*Callback to open a file */
9  drv.close_cb = my_close_cb;                           /*Callback to close a file */
10 drv.read_cb = my_read_cb;                             /*Callback to read a file */
11 drv.write_cb = my_write_cb;                           /*Callback to write a file */
12 drv.seek_cb = my_seek_cb;                             /*Callback to seek in a file (Move_
↪cursor) */
```

(下页继续)

(续上页)

```

13     drv.tell_cb = my_tell_cb;                                /*Callback to tell the cursor position */
14     ↪ */
15     drv.trunc_cb = my_trunc_cb;                               /*Callback to delete a file */
16     drv.size_cb = my_size_cb;                                /*Callback to tell a file's size */
17     drv.rename_cb = my_rename_cb;                            /*Callback to rename a file */
18     drv.dir_open_cb = my_dir_open_cb;                        /*Callback to open directory to read
19     ↪its content */
20     drv.dir_read_cb = my_dir_read_cb;                        /*Callback to read a directory's
21     ↪content */
22     drv.dir_close_cb = my_dir_close_cb;                      /*Callback to close a directory */
23     drv.free_space_cb = my_free_space_cb;                    /*Callback to tell free space on the
24     ↪drive */
25     drv.user_data = my_user_data;                            /*Any custom data if required*/
26     lv_fs_drv_register(&drv);                                /*Finally register the drive*/

```

任何回调都可以为 NULL，以指示不支持该操作。

作为使用回调的示例，如果使用 `lv_fs_open(&file, "S:/folder/file.txt", LV_FS_MODE_WR)`，LVGL 会：

1. 验证是否存在带字母“S”的已注册驱动器。
2. 检查是否实现了 `open_cb`（非 NULL）。
3. 调用 `open_cb` 设置 "folder/file.txt" 路径。

18.2 使用范例

下面的示例显示如何从文件读取：

```

1     lv_fs_file_t f;
2     lv_fs_res_t res;
3     res = lv_fs_open(&f, "S:folder/file.txt", LV_FS_MODE_RD);
4     if(res != LV_FS_RES_OK) my_error_handling();
5
6     uint32_t read_num;
7     uint8_t buf[8];
8     res = lv_fs_read(&f, buf, 8, &read_num);
9     if(res != LV_FS_RES_OK || read_num != 8) my_error_handling();

```

(下页继续)

(续上页)

```
10
11 lv_fs_close(&f);
```

lv_fs_open 中的模式可以是 LV_FS_MODE_WR 以打开以进行写入，也可以是 LV_FS_MODE_RD。两者的 LV_FS_MODE_WR

本示例说明如何读取目录的内容。由驱动程序决定如何标记目录，但是在目录名称前面插入 '/' 可能是一个好习惯。

```
1  lv_fs_dir_t dir;
2  lv_fs_res_t res;
3  res = lv_fs_dir_open(&dir, "S:/folder");
4  if(res != LV_FS_RES_OK) my_error_handling();
5
6  char fn[256];
7  while(1) {
8      res = lv_fs_dir_read(&dir, fn);
9      if(res != LV_FS_RES_OK) {
10         my_error_handling();
11         break;
12     }
13
14     /*fn is empty, if not more files to read*/
15     if(strlen(fn) == 0) {
16         break;
17     }
18
19     printf("%s\n", fn);
20 }
21
22 lv_fs_dir_close(&dir);
```

18.3 使用图像驱动程序

图像对象也可以从文件中打开（除了存储在闪存中的变量）。

要初始化 图像，需要以下回调：

- open
- close
- read
- seek

- tell

18.4 相关 API

TODO

动画效果 (Animations)

我们可以使用动画在开始值和结束值之间自动更改变量的值。动画将通过定期调用带有相应 `value` 参数的“animator”函数来发生。

动画功能函数具有以下原型：

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

该原型与 LVGL 的大多数设置功能兼容。例如 `lv_obj_set_x(obj,value)` 或 `lv_obj_set_width(obj,value)`

19.1 创建动画

要创建动画，必须初始化 `lv_anim_t` 变量，并使用 `lv_anim_set _...()` 函数进行配置。

```
1  /* 初始化动画
2     *-----*/
3
4  lv_anim_t a;
5  lv_anim_init(&a);
6
7  /* 必选设置
8     *-----*/
9
10 /* 设置“动画制作”功能 */
```

(下页继续)

(续上页)

```

11     lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t) lv_obj_set_x);
12
13     /* 设置“动画制作”功能 */
14     lv_anim_set_var(&a, obj);
15
16     /* 动画时长 [ms] */
17     lv_anim_set_time(&a, duration);
18
19     /* 设置开始和结束值。例如。 0、150 */
20     lv_anim_set_values(&a, start, end);
21
22     /* 可选设置
23      *-----*/
24
25     /* 开始动画之前的等待时间 [ms] */
26     lv_anim_set_delay(&a, delay);
27
28     /* 设置路径（曲线）。默认为线性 */
29     lv_anim_set_path(&a, &path);
30
31     /* 设置一个回调以在动画准备好时调用。 */
32     lv_anim_set_ready_cb(&a, ready_cb);
33
34     /* 设置在动画开始时（延迟后）调用的回调。 */
35     lv_anim_set_start_cb(&a, start_cb);
36
37     /* 在此持续时间内，也向后播放动画。默认值为 0（禁用） [ms] */
38     lv_anim_set_playback_time(&a, wait_time);
39
40     /* 播放前延迟。默认值为 0（禁用） [ms] */
41     lv_anim_set_playback_delay(&a, wait_time);
42
43     /* 重复次数。默认值为 1。LV_ANIM_REPEAT_INFINIT 用于无限重复 */
44     lv_anim_set_repeat_count(&a, wait_time);
45
46     /* 重复之前要延迟。默认值为 0（禁用） [ms] */
47     lv_anim_set_repeat_delay(&a, wait_time);
48
49     /* true（默认）：立即应用开始值，false：延迟设置动画后再应用开始值。真正开始。 */
50     lv_anim_set_early_apply(&a, true/false);
51
52     /* 应用动画效果
53      *-----*/
54     lv_anim_start(&a);                                /* 应用动画效果 */

```


可以同时在同一变量上应用 **多个不同的动画**。例如，用 `lv_obj_set_x` 和 `lv_obj_set_y` 设置 x 和 y 坐标的动画。但是，只有一个动画可以存在给定的变量和函数对。因此，`lv_anim_start()` 将删除已经存在的可变函数动画。

19.2 动画路径 (path)

可以设置 **动画的路径**。在最简单的情况下，它是线性的，这意味着开始和结束之间的当前值线性变化。路径主要是根据动画的当前状态计算要设置的下一个值的功能。当前，有以下内置路径功能：

- `lv_anim_path_linear` 线性动画
- `lv_anim_path_step` 一步到位
- `lv_anim_path_ease_in` 渐进效果
- `lv_anim_path_ease_out` 渐退效果
- `lv_anim_path_ease_in_out` 渐进和渐退效果
- `lv_anim_path_overshoot` 超出最终值
- `lv_anim_path_bounce` 从最终值反弹一点 (就像撞墙一样)

可以这样初始化路径：

```
1  lv_anim_path_t path;
2  lv_anim_path_init(&path);
3  lv_anim_path_set_cb(&path, lv_anim_path_overshoot);
4  lv_anim_path_set_user_data(&path, &foo); /* 自定义数据 (可选) */
5
6  /* 在动画中设置路径 */
7  lv_anim_set_path(&a, &path);
```

19.3 速度与时间

默认情况下，可以设置动画时间。但是，在某些情况下，动画速度更加实用。

`lv_anim_speed_to_time(speed, start, end)` 函数以毫秒为单位计算从给定速度的起始值到结束值所需的时间。速度以单位/秒为单位进行解释。例如，`lv_anim_speed_to_time(20, 0, 100)` 将给出 5000 毫秒。例如，在 `lv_obj_set_x` 单位为像素的情况下，因此 20 表示 20 px / sec 的速度。

19.4 删除动画

可以通过提供 `animation` 变量及其动画功能来删除 `lv_anim_del(var, func)` 的动画。

19.5 相关 API

TODO

任务 (Task)

LVGL 具有内置的任务系统。您可以注册一个函数以使其定期被调用。在 `lv_task_handler ()` 中处理和调用任务，该任务需要每几毫秒定期调用一次。有关更多信息，请参见移植。

任务是非抢占式的，这意味着一个任务无法中断另一个任务。因此，您可以在任务中调用任何与 LVGL 相关的功能。

20.1 创建一个任务

要创建新任务，请使用 `lv_task_create(task_cb, period_ms, LV_TASK_PRIO_OFF/LOWEST/LOW/MID/HIGH/HIGHEST, user_data)`。它将创建一个 `lv_task_t *` 变量，以后可用于修改任务的参数。`lv_task_create_basic ()` 也可以使用。它允许您创建新任务而无需指定任何参数。

任务回调应具有 `void (* lv_task_cb_t) (lv_task_t *)`；原型。

范例：

```
1  void my_task(lv_task_t * task)
2  {
3      /*Use the user_data*/
4      uint32_t * user_data = task->user_data;
5      printf("my_task called with user data: %d\n", *user_data);
6
7      /*Do something with LVGL*/
8      if(something_happened) {
```

(下页继续)

(续上页)

```
9         something_happened = false;
10         lv_btn_create(lv_scr_act(), NULL);
11     }
12 }
13
14 ...
15
16 static uint32_t user_data = 10;
17 lv_task_t * task = lv_task_create(my_task, 500, LV_TASK_PRIO_MID, &user_data);
```

20.2 准备并重置

`lv_task_ready (task)` 使任务在 `lv_task_handler ()` 的下次调用上运行。

`lv_task_reset (task)` 重置任务的周期。在定义的毫秒周期过去后，它将再次调用。

20.3 设定参数

您可以稍后修改任务的一些参数：

- `lv_task_set_cb(task, new_cb)`
- `lv_task_set_period(task, new_period)`
- `lv_task_set_prio(task, new_priority)`

20.4 一次行的任务

通过调用 `lv_task_once (task)`，可以使任务仅运行一次。首次调用该任务后，该任务将自动删除。

20.5 测量空闲时间

您可以使用 `lv_task_get_idle ()` 获得空闲百分比时间 `lv_task_handler`。请注意，它不会衡量整个系统的空闲时间，只会衡量 `lv_task_handler`。如果您使用操作系统并在任务中调用 `lv_task_handler`，可能会产生误导，因为它实际上无法衡量 OS 在空闲线程中花费的时间。

20.6 异步调用

在某些情况下，无法立即执行某些操作。例如，您不能立即删除对象，因为其他对象仍在使用它，或者您不想立即阻止执行。对于这些情况，可以使用 `lv_async_call(my_function, data_p)` 在下次调用 `lv_task_handler` 时调用 `my_function`。调用 `data_p` 时会将其传递给函数。请注意，仅保存了数据的指针，因此您需要确保在调用函数时变量将为“有效”。您可以使用静态，全局或动态分配的数据。

例如：

```
1  void my_screen_clean_up(void * scr)
2  {
3      /*Free some resources related to `scr`*/
4
5      /*Finally delete the screen*/
6      lv_obj_del(scr);
7  }
8
9  ...
10
11     /*Do somethings with the object on the current screen*/
12
13     /*Delete screen on next call of `lv_task_handler`. So not now.*/
14     lv_async_call(my_screen_clean_up, lv_scr_act());
15
16     /*The screen is still valid so you can do other things with it*/
```

如果只想删除一个对象，而无需清除 `my_screen_cleanup` 中的任何内容，则可以使用 `lv_obj_del_async`，它将在下次调用 `lv_task_handler` 时删除该对象。

20.7 相关 API

TODO

使用 LVGL，无需手动绘制任何内容。只需创建对象 (如按钮和标签)，移动并更改它们，LVGL 就会刷新并重新绘制所需的内容。

但是，对 LVGL 中的绘图方式有基本了解可能会很有用。

基本概念是不要直接绘制到屏幕上，而是先绘制到内部缓冲区，然后在渲染准备好后将其复制到屏幕上。它具有两个主要优点：

- **避免闪烁**绘制用户界面的各层。例如，当绘制背景 + 按钮 + 文本时，每个“阶段”将在短时间内可见。
- **让动画更快一些**修改 RAM 中的缓冲区并最终一次写入一个像素，而不是直接在每次像素访问时直接读取/写入显示器。(例如，通过具有 SPI 接口的显示控制器) 因此，它适用于多次重绘的像素 (例如背景 + 按钮 + 文字)。

21.1 缓冲类型

共有 3 种类型的缓冲区：

1. **一个缓冲区** LVGL 将屏幕的内容绘制到缓冲区中并将其发送到显示器。缓冲区可以小于屏幕。在这种情况下，较大的区域将被重画成多个部分。如果只有很小的区域发生变化 (例如按下按钮)，则只会刷新那些区域。
2. **两个非屏幕大小的缓冲区**具有两个缓冲区的 LVGL 可以吸引到一个缓冲区中，而另一缓冲区的内容发送到后台显示。应该使用 DMA 或其他硬件将数据传输到显示器，以让 CPU 同时绘图。这样，显示的渲染和刷新并行处理。与 **一个缓冲区**的情况类似，如果缓冲区小于要刷新的区域，LVGL 将按块绘制显示内容

3. **两个屏幕大小的缓冲区**与两个非屏幕大小的缓冲区相反，LVGL 将始终提供整个屏幕的内容，而不仅仅是块。这样，驱动程序可以简单地将帧缓冲区的地址更改为从 LVGL 接收的缓冲区。因此，当 MCU 具有 LCD/TFT 接口且帧缓冲区只是 RAM 中的一个位置时，此方法效果最佳。

21.2 屏幕刷新机制

1. GUI 上发生某些事情，需要重绘。例如，已按下按钮，更改了图表或发生了动画等。
2. LVGL 将更改后的对象的旧区域和新区域保存到称为无效区域缓冲区的缓冲区中。为了优化，在某些情况下，对象不会添加到缓冲区中：
 - 隐藏的对象未添加。
 - 不添加完全超出其父对象的对象。
 - 父区之外的区域将裁剪到父区。
 - 未添加其他屏幕上的对象。
1. 在每个 LV_DISP_DEF_REFR_PERIOD 中 (在 lv_conf.h 中设置):
 - LVGL 检查无效区域并加入相邻或相交的区域。
 - 获取第一个连接区域，如果它小于显示缓冲区，则只需将区域的内容绘制到显示缓冲区即可。如果该区域不适合缓冲区，请在显示缓冲区上绘制尽可能多的线。
 - 绘制区域后，从显示驱动程序调用 flush_cb 刷新显示。
 - 如果该区域大于缓冲区，则也重新绘制其余部分。
 - 对所有连接的区域执行相同的操作。

重绘区域时，库会搜索覆盖要重绘区域的最顶层对象，然后从该对象开始绘制。例如，如果按钮的标签已更改，则库将看到足以在文本下方绘制按钮，并且也不需要绘制背景。

关于绘制机制，缓冲区类型之间的差异如下：

1. **一个缓冲区** - LVGL 需要等待 lv_disp_flush_ready() (在 flush_cb 末尾调用)，然后才能开始重画下一部分。
2. **两个非屏幕大小的缓冲区** - 当第一个缓冲区发送到 flush_cb 时，LVGL 可以立即绘制到第二个缓冲区，因为刷新应由 DMA(或类似硬件) 在后台完成。
3. **两个屏幕大小的缓冲区** - 调用 flush_cb 之后，如果显示为帧缓冲区，则第一个缓冲区。它的内容被复制到第二个缓冲区，所有更改都绘制在其顶部。

21.3 蒙版 (遮罩)

蒙版是 LVGL 绘图引擎的基本概念。要使用 LVGL，不需要了解这里描述的机制，但是可能会对了解图纸在引擎盖下的工作方式感兴趣。

要学习遮罩，让我们先学习绘画的步骤：

1. 根据对象的样式创建绘制描述符 (例如 `lv_draw_rect_dsc_t`)。它告诉绘图的参数，例如颜色，宽度，不透明度，字体，半径等。
2. 使用初始化的描述符和其他一些参数调用 `draw` 函数。它将原始形状渲染到当前绘制缓冲区。
3. 如果形状非常简单并且不需要遮罩，请转到 # 5。其他创建所需的蒙版 (例如，圆角的矩形蒙版)
4. 将所有创建的蒙版应用一行或几行。它使用创建的遮罩的“形状”在遮罩缓冲区中创建 0..255 值。例如。如果根据掩码的参数设置为“线掩码”，则将缓冲区的一侧保持不变 (默认为 255)，并将其余部分设置为 0 以指示应将另一侧除去。
5. 将图像或矩形混合到屏幕上。在混合蒙版 (使某些像素透明或不透明)，混合模式 (加性，减性等) 期间，将处理不透明性。
6. 从 # 4 开始重复。

使用蒙版可创建几乎所有基本图元：

- **字母**从字母创建遮罩，并使用该遮罩绘制一个“字母色”矩形。
- **线**由 4 个“线遮罩”创建的线，以遮盖该线的左，右，顶部和底部，以获得完美垂直的线尾
- **圆角矩形**实时为圆角矩形的每一行创建一个遮罩，并根据该遮罩绘制一个普通的填充矩形。
- **剪辑角**以剪辑圆角上的溢出内容，并应用圆角矩形蒙版。
- **矩形边框**与圆角矩形相同，但内部也被遮罩
- **圆弧绘制**绘制了圆形边框，但应用了弧形遮罩。
- **ARGB 图像**将 alpha 通道分离为一个蒙版，并且该图像被绘制为普通 RGB 图像。

如以上 # 3 所述，在某些情况下，不需要面具：

- 一个单色的，不是圆角的矩形
- RGB 图像

LVGL 具有以下内置掩码类型，可以实时计算和应用：

- **LV_DRAW_MASK_TYPE_LINE** 删除线的一侧 (顶部，底部，左侧或右侧)。 `lv_draw_line` 使用其中的 4 个。本质上，每条 (倾斜) 线都通过形成一个矩形以 4 个线罩为边界。
- **LV_DRAW_MASK_TYPE_RADIUS** 删除也可以具有半径的矩形的内部或外部。通过将半径设置为较大的值 (`LV_RADIUS_CIRCLE`)，它还可用于创建圆
- **LV_DRAW_MASK_TYPE_ANGLE** 删除圆扇形。 `lv_draw_arc` 使用它删除“空”扇区。

- **LV_DRAW_MASK_TYPE_FADE** 创建垂直淡入淡出 (更改不透明度)
- **LV_DRAW_MASK_TYPE_MAP** 遮罩存储在数组中, 并应用了必要的部分

在绘制过程中会自动创建和删除蒙版, 但是 `lv_objmask` 允许用户添加蒙版。这是一个例子:

21.3.1 范例

几个对象蒙版

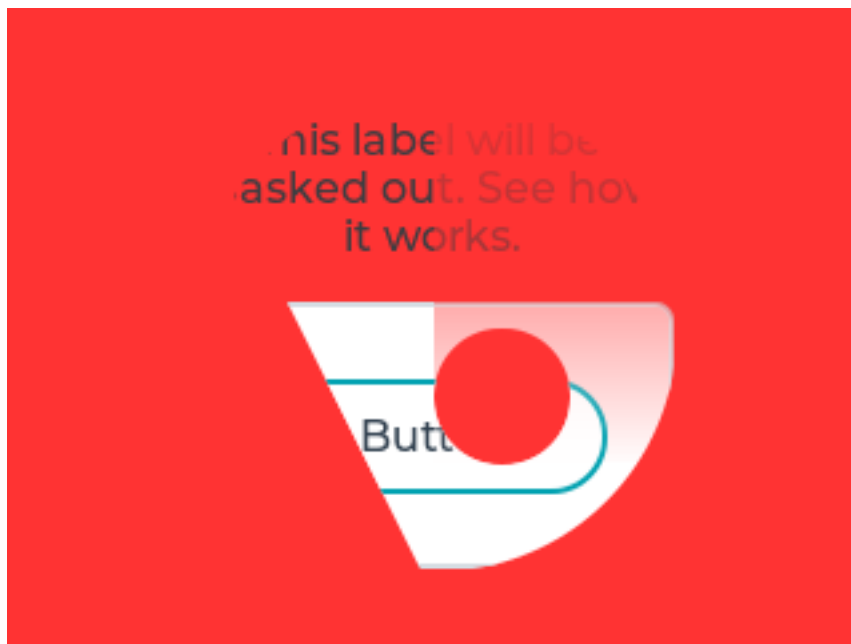


图 1: 几个对象蒙版

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #if LV_USE_OBJMASK
3
4  void lv_ex_objmask_1(void)
5  {
6
7      /*Set a very visible color for the screen to clearly see what happens*/
8      lv_obj_set_style_local_bg_color(lv_scr_act(), LV_OBJ_PART_MAIN, LV_STATE_
↪DEFAULT, lv_color_hex3(0xf33));
9
10     lv_obj_t * om = lv_objmask_create(lv_scr_act(), NULL);
11     lv_obj_set_size(om, 200, 200);
12     lv_obj_align(om, NULL, LV_ALIGN_CENTER, 0, 0);
```

(下页继续)

(续上页)

```

13     lv_obj_t * label = lv_label_create(om, NULL);
14     lv_label_set_long_mode(label, LV_LABEL_LONG_BREAK);
15     lv_label_set_align(label, LV_LABEL_ALIGN_CENTER);
16     lv_obj_set_width(label, 180);
17     lv_label_set_text(label, "This label will be masked out. See how it works.
↪");
18     lv_obj_align(label, NULL, LV_ALIGN_IN_TOP_MID, 0, 20);
19
20     lv_obj_t * cont = lv_cont_create(om, NULL);
21     lv_obj_set_size(cont, 180, 100);
22     lv_obj_set_drag(cont, true);
23     lv_obj_align(cont, NULL, LV_ALIGN_IN_BOTTOM_MID, 0, -10);
24
25     lv_obj_t * btn = lv_btn_create(cont, NULL);
26     lv_obj_align(btn, NULL, LV_ALIGN_CENTER, 0, 0);
27     lv_obj_set_style_local_value_str(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
↪"Button");
28     //uint32_t t;
29
30     //lv_refr_now(NULL);
31     //t = lv_tick_get();
32     //while(lv_tick_elaps(t) < 1000);
33
34     lv_area_t a;
35     lv_draw_mask_radius_param_t r1;
36
37     a.x1 = 10;
38     a.y1 = 10;
39     a.x2 = 190;
40     a.y2 = 190;
41     lv_draw_mask_radius_init(&r1, &a, LV_RADIUS_CIRCLE, false);
42     lv_objmask_add_mask(om, &r1);
43
44     //lv_refr_now(NULL);
45     //t = lv_tick_get();
46     //while(lv_tick_elaps(t) < 1000);
47
48     a.x1 = 100;
49     a.y1 = 100;
50     a.x2 = 150;
51     a.y2 = 150;
52     lv_draw_mask_radius_init(&r1, &a, LV_RADIUS_CIRCLE, true);
53     lv_objmask_add_mask(om, &r1);

```

(下页继续)

(续上页)

```

54
55     //lv_refr_now(NULL);
56     //t = lv_tick_get();
57     //while(lv_tick_elaps(t) < 1000);
58
59     lv_draw_mask_line_param_t l1;
60     lv_draw_mask_line_points_init(&l1, 0, 0, 100, 200, LV_DRAW_MASK_LINE_SIDE_
↪TOP);
61     lv_objmask_add_mask(om, &l1);
62
63     //lv_refr_now(NULL);
64     //t = lv_tick_get();
65     //while(lv_tick_elaps(t) < 1000);
66
67     lv_draw_mask_fade_param_t f1;
68     a.x1 = 100;
69     a.y1 = 0;
70     a.x2 = 200;
71     a.y2 = 200;
72     lv_draw_mask_fade_init(&f1, &a, LV_OPA_TRANSP, 0, LV_OPA_COVER, 150);
73     lv_objmask_add_mask(om, &f1);
74 }
75
76 #endif

```

文字遮罩

上述效果的示例代码:

```

1     #include "../lv_examples.h"
2     #if LV_USE_OBJMASK
3
4     #define MASK_WIDTH 100
5     #define MASK_HEIGHT 50
6
7     void lv_ex_objmask_2(void)
8     {
9
10        /* Create the mask of a text by drawing it to a canvas*/
11        static lv_opa_t mask_map[MASK_WIDTH * MASK_HEIGHT];
12
13        /*Create a "8 bit alpha" canvas and clear it*/
14        lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);

```

(下页继续)

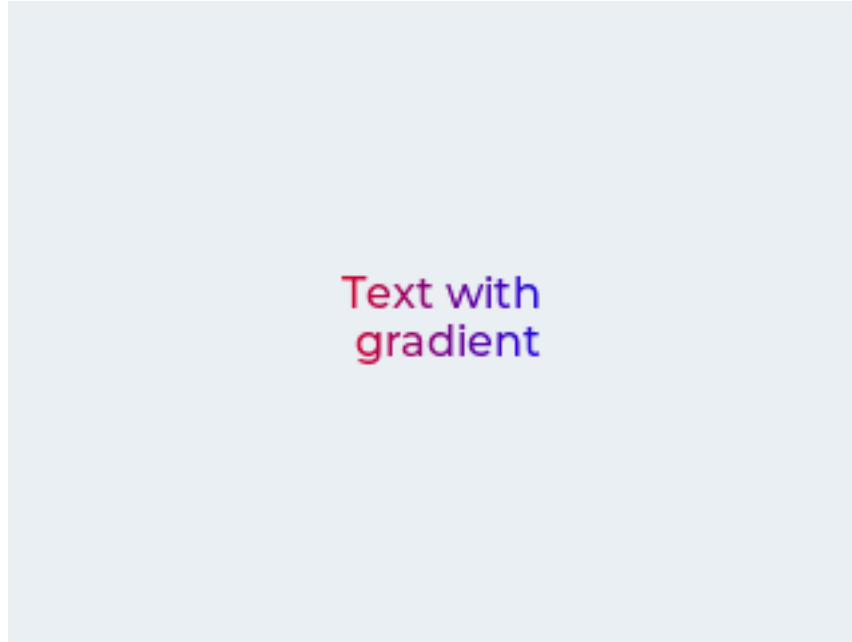


图 2: 文字遮罩

(续上页)

```

15     lv_canvas_set_buffer(canvas, mask_map, MASK_WIDTH, MASK_HEIGHT, LV_IMG_CF_
↪ALPHA_8BIT);
16     lv_canvas_fill_bg(canvas, LV_COLOR_BLACK, LV_OPA_TRANSP);
17
18     /*Draw a label to the canvas. The result "image" will be used as mask*/
19     lv_draw_label_dsc_t label_dsc;
20     lv_draw_label_dsc_init(&label_dsc);
21     label_dsc.color = LV_COLOR_WHITE;
22     lv_canvas_draw_text(canvas, 5, 5, MASK_WIDTH, &label_dsc, "Text with_
↪gradient", LV_LABEL_ALIGN_CENTER);
23
24     /*The mask is reads the canvas is not required anymore*/
25     lv_obj_del(canvas);
26
27     /*Create an object mask which will use the created mask*/
28     lv_obj_t * om = lv_objmask_create(lv_scr_act(), NULL);
29     lv_obj_set_size(om, MASK_WIDTH, MASK_HEIGHT);
30     lv_obj_align(om, NULL, LV_ALIGN_CENTER, 0, 0);
31
32     /*Add the created mask map to the object mask*/
33     lv_draw_mask_map_param_t m;
34     lv_area_t a;
35     a.x1 = 0;

```

(下页继续)

(续上页)

```
36     a.y1 = 0;
37     a.x2 = MASK_WIDTH - 1;
38     a.y2 = MASK_HEIGHT - 1;
39     lv_draw_mask_map_init(&m, &a, mask_map);
40     lv_objmask_add_mask(om, &m);
41
42     /*Create a style with gradient*/
43     static lv_style_t style_bg;
44     lv_style_init(&style_bg);
45     lv_style_set_bg_opa(&style_bg, LV_STATE_DEFAULT, LV_OPA_COVER);
46     lv_style_set_bg_color(&style_bg, LV_STATE_DEFAULT, LV_COLOR_RED);
47     lv_style_set_bg_grad_color(&style_bg, LV_STATE_DEFAULT, LV_COLOR_BLUE);
48     lv_style_set_bg_grad_dir(&style_bg, LV_STATE_DEFAULT, LV_GRAD_DIR_HOR);
49
50     /* Create and object with the gradient style on the object mask.
51      * The text will be masked from the gradient*/
52     lv_obj_t * bg = lv_obj_create(om, NULL);
53     lv_obj_reset_style_list(bg, LV_OBJ_PART_MAIN);
54     lv_obj_add_style(bg, LV_OBJ_PART_MAIN, &style_bg);
55     lv_obj_set_size(bg, MASK_WIDTH, MASK_HEIGHT);
56
57 }
58
59 #endif
```

22.1 概述

基础对象实现了屏幕上小部件的基本属性，例如：

- 坐标
- 父对象
- 子对象
- 主要风格
- 属性，例如点击启用、拖动启用等。

在面向对象的思想中，它是继承 LVGL 中所有其他对象的基类。这尤其有助于减少代码重复。

Base 对象的功能也可以与其他小部件一起使用。例如 `lv_obj_set_width(slider, 100)`

Base 对象可以直接用作简单的小部件。然后就是矩形。

22.2 坐标

22.2.1 尺寸

可以使用 `lv_obj_set_width(obj, new_width)` 和 `lv_obj_set_height(obj, new_height)` 在单个坐标轴方向 (横向、纵向) 上修改对象的大小, 或者可以使用 `lv_obj_set_size(obj, new_width, new_height)` 同时修改两个坐标轴方向 (横向及纵向) 的大小。

样式可以向对象添加边距。**Margin**(边距) 说“我想要我周围的空间”。那么我们可以设置对象的宽度: `lv_obj_set_width_margin(obj, new_width)` 或高度: `lv_obj_set_height_margin(obj, new_height)`。更确切地讲是这样: `new_width = left_margin + object_width + right_margin`。

要获取包含边距的宽度或高度, 请使用 `lv_obj_get_width/height_margin(obj)`。

样式也可以向对象添加填充。填充的意思是 **我不要我的孩子们离我的身体太近, 所以要保留这个空间**。通过 `lv_obj_set_width_fit(obj, new_width)` 或 `lv_obj_set_height_fit(obj, new_height)` 设置需要填充减小的宽度或高度。

可以以更精确的方式: `new_width = left_pad + object_width + right_pad`。要获得通过填充减少的宽度或高度, 请使用 `lv_obj_get_width/height_fit(obj)`。可以将其视为 **对象的有用大小**。

当其他窗口小部件使用布局或自动调整时, 边距和填充变得很重要。

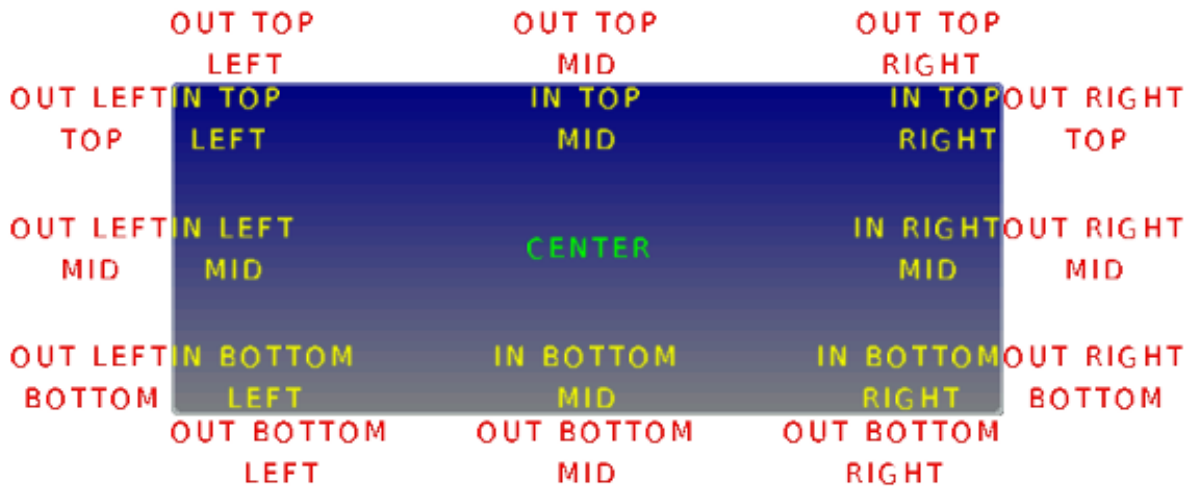
22.2.2 位置

可以使用 `lv_obj_set_x(obj, new_x)` 和 `lv_obj_set_y(obj, new_y)` 设置对象相对于父级的 x 和 y 坐标, 或者同时使用 `lv_obj_set_pos(obj, new_x, new_y)` 设置相对于父级的 x 和 y 坐标。

22.2.3 对齐

可以使用 `lv_obj_align(obj, obj_ref, LV_ALIGN_..., x_ofs, y_ofs)` 将对象与另一个对象对齐。

- `obj` 是要对齐的对象。
- `obj_ref` 是参考对象。`obj` 将与其对齐。如果 `obj_ref = NULL`, 则将使用 `obj` 的父级。
- 第三个参数是对齐方式的类型。这些是可能的选项:



对齐类型的构建类似于 LV_ALIGN_OUT_TOP_MID。- 最后两个参数允许在对齐对象 后将其移动指定数量的像素。

例如，要在图像下方对齐文本：lv_obj_align(text, image, LV_ALIGN_OUT_BOTTOM_MID, 0, 10)。或在父级中间对齐文本：lv_obj_align(text, NULL, LV_ALIGN_CENTER, 0, 0)

lv_obj_align_origo 的工作方式与 lv_obj_align 类似，但它使对象的中心对齐。

例如，lv_obj_align_origo(btn, image, LV_ALIGN_OUT_BOTTOM_MID, 0, 0) 将使按钮的中心与图像底部对齐。

如果在 lv_conf.h 中启用了 LV_USE_OBJ_REALIGN，则路线的参数将保存在对象中。然后，只需调用 lv_obj_realign(obj) 即可重新对齐对象。等效于使用相同的参数再次调用 lv_obj_align。

如果对齐是通过 lv_obj_align_origo 进行的，则在重新对齐对象时将使用它。

函数 lv_obj_align_x/y 和 lv_obj_align_origo_x/y 只能在一个轴上对齐。

如果使用 lv_obj_set_auto_realign(obj, true)，并且对象的大小在 lv_obj_set_width/height/size() 函数中更改，则对象将自动重新对齐。当将尺寸动画应用于对象并且需要保留原始位置时，这非常有用。

(请注意，屏幕的坐标无法更改。尝试在屏幕上使用这些功能将导致不确定的行为。

22.3 父子关系

可以使用 `lv_obj_set_parent(obj, new_parent)` 为对象设置新的父对象。要获取当前的父对象，请使用 `lv_obj_get_parent(obj)`。

要获取对象的子代，请使用 `lv_obj_get_child(obj, child_prev)`（从最后到第一）或 `lv_obj_get_child_back(obj, child_prev)`（从第一到最后）。要获得第一个子代，请将 `NULL` 作为第二个参数传递，并使用返回值遍历子代。如果没有更多的子级，该函数将返回 `NULL`。例如：

```
1  lv_obj_t * child = lv_obj_get_child(parent, NULL);
2  while(child) {
3      /*Do something with "child" */
4      child = lv_obj_get_child(parent, child);
5  }
```

`lv_obj_count_children(obj)` 告知对象上的子代数。

`lv_obj_count_children_recursive(obj)` 也会告知子代数，使用递归计算子代数。

22.4 屏幕

创建了 `lv_obj_t * screen = lv_obj_create(NULL, NULL)` 之类的屏幕后，可以使用 `lv_scr_load(screen)` 加载它。`lv_scr_act()` 函数提供了指向当前屏幕的指针。

如果有更多的显示，那么我们就要知道这些功能，可以在最后创建的或明确选择的显示上运行（使用 `lv_disp_set_default()`）。

要获取对象的屏幕，请使用 `lv_obj_get_screen(obj)` 函数。

22.5 层次

有两个自动生成的层：

- 顶层 `lv_layer_top()`
- 系统层 `lv_layer_sys()`

它们独立于屏幕 (`lv_scr_act()`)，并且将显示在每个屏幕上。顶层位于屏幕上每个对象的上方，而系统层也位于顶层上方。可以将任何弹出窗口自由添加到顶层。但是，系统层仅限于 **系统级** 的内容（例如，鼠标光标将放在 `lv_indev_set_cursor()` 中）。

层级关系：`lv_scr_act() < lv_layer_top() < lv_layer_sys()`

`lv_layer_top()` 和 `lv_layer_sys()` 函数提供了指向顶层或系统层的指针。

可以使用 `lv_obj_move_foreground(obj)` 和 `lv_obj_move_background(obj)` 将对象置于前景或发送至背景。

阅读 [对象层级 \(Layers\)](#) 部分，以了解有关图层的更多信息。

22.6 事件处理

要为对象设置事件回调，请使用 `lv_obj_set_event_cb(obj, event_cb)`，

要将事件手动发送到对象，请使用 `lv_event_send(obj, LV_EVENT_..., data)`

阅读 [事件概述](#) 以了解有关事件的更多信息。

22.7 部件

小部件可以包含多个部分。例如，按钮仅具有主要部分，而滑块则由背景，指示器和旋钮组成。

零件名称的构造类似于 `LV_ + <TYPE> _PART_ <NAME>`。例如 `LV_BTN_PART_MAIN` 或 `LV_SLIDER_PART_KNOB` 通常在将样式添加到对象时使用零件。使用零件可以将不同的样式分配给对象的不同零件。

要了解有关零件的更多信息，请阅读 [样式概述](#) 的相关部分。

22.8 状态

对象可以处于以下状态的组合：

- **LV_STATE_DEFAULT** 默认或正常状态
- **LV_STATE_CHECKED** 选中或点击
- **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_EDITED** 由编码器编辑
- **LV_STATE_HOVERED** 鼠标悬停 (现在还不支持)
- **LV_STATE_PRESSED** 按下
- **LV_STATE_DISABLED** 禁用或无效

当用户按下、释放、聚焦等对象时，状态通常由库自动检测更改。当然状态也可以手动检测更改。

要完全覆盖当前状态，调用 `lv_obj_set_state(obj, part, LV_STATE...)`

要设置或清除某个状态 (但不更改其他状态)，调用 `lv_obj_add/clear_state(obj, part, LV_STATE...)`

可以组合使用状态值。例如: `lv_obj_set_state(obj, part, LV_STATE_PRESSED | LV_PRESSED_CHECKED)`。

要了解有关状态的更多信息, 请阅读 [样式概述](#) 的相关部分。

22.9 风格

确保首先阅读了解了 [样式概述](#) 部分内容。

要将样式添加到对象, 请使用 `lv_obj_add_style(obj, part, &new_style)` 函数。基础对象使用所有类似矩形的样式属性。

要从对象中删除所有样式, 请使用 `lv_obj_reset_style_list(obj, part)`

如果修改对象已使用的样式, 为了刷新受影响的对象, 可以使用每个对象在对象上使用 `lv_obj_refresh_style(obj)` 或使用 `lv_obj_report_style_mod(& style)` 通知具有给定样式的所有对象。如果 `lv_obj_report_style_mod` 的参数为 `NULL`, 则将通知所有对象。

22.10 属性

调用 `lv_obj_set_...(obj, true/false)` 可以启用/禁用一些属性:

- **hidden** - 隐藏对象。它不会被绘制, 输入设备会将其视为不存在。它的子项也将被隐藏。
- **click** - 允许通过输入设备单击对象。如果禁用, 则单击事件将传递到此事件后面的对象。(默认情况下无法点击标签)
- **top** - 如果启用, 则单击此对象或其任何子级时, 该对象将进入前台。
- **drag** - 启用拖动 (通过输入设备移动)
- **drag_dir** - 启用仅在特定方向上拖动。可以是 `LV_DRAG_DIR_HOR / VER / ALL`。
- **drag_throw** - 通过拖动启用“投掷”, 就像对象将具有动量一样
- **drag_parent** - 如果启用, 则对象的父对象将在拖动过程中移动。看起来就像拖动父级。递归检查, 因此也可以传播给祖父母。
- **parent_event** - 也将事件传播给父母。递归检查, 因此也可以传播给祖父母。
- **opa_scale_enable** - 启用不透明度缩放。请参见 `[# opa-scale](Opa 比例尺)` 部分。

22.11 保护

库中有一些自动发生的特定操作。为防止一种或多种此类行为，可以保护对象免受它们侵害。存在以下保护：

- **LV_PROTECT_NONE** 没有保护
- **LV_PROTECT_POS** 防止自动定位 (例如容器中的布局)
- **LV_PROTECT_FOLLOW** 防止在自动排序 (例如容器布局) 中遵循该对象 (“换行”)
- **LV_PROTECT_PARENT** 防止自动更改父母。(例如，页面将在背景上创建的子代移动到可滚动页面)
- **LV_PROTECT_PRESS_LOST** 防止手指滑过对象时丢失。(例如，如果按下某个按钮，则可以将其释放)
- **LV_PROTECT_CLICK_FOCUS** 如果对象在“组”中并且启用了单击焦点，则阻止其自动聚焦。
- **LV_PROTECT_CHILD_CHG** 禁用子对象更换信号。库内部使用

`lv_obj_add/clear_protect(obj, LV_PROTECT_...)` 设置/清除保护。也可以使用保护类型的“或”值。

22.12 组

一旦将对象添加到具有 `lv_group_add_obj(group, obj)` 的组中，则可以通过 `lv_obj_get_group(obj)` 获得该对象的当前组。

`lv_obj_is_focused(obj)` 告知对象当前是否集中在其组上。如果未将对象添加到组中，则将返回 `false`。

阅读 [输入设备概述](#) 以了解有关组的更多信息。

22.13 扩展点击区域

默认情况下，只能在对象的坐标上单击对象，但是可以使用 `lv_obj_set_ext_click_area(obj, left, right, top, bottom)` 扩展该区域。左侧 (`left`)/右侧 (`right`)/顶部 (`top`)/底部 (`bottom`) 描述了可点击区域应在每个方向上超出默认范围的程度。

前提是要在 `lv_conf.h` 使能 `LV_USE_EXT_CLICK_AREA` 启用此功能。可能的值为：

- **LV_EXT_CLICK_AREA_FULL** 将所有 4 个坐标存储为 `lv_coord_t`
- **LV_EXT_CLICK_AREA_TINY** 仅将水平和垂直坐标 (使用左/右和上/下的较大值) 存储为 `uint8_t`
- **LV_EXT_CLICK_AREA_OFF** 禁用此功能

22.14 事件

仅 [通用事件](#) 是按对象类型发送的。

了解有关 [事件](#) 的更多信息。

22.15 按键

对象类型不处理任何输入按键。

进一步了解 [按键](#) 。

22.16 范例

22.16.1 具有自定义样式的基础对象

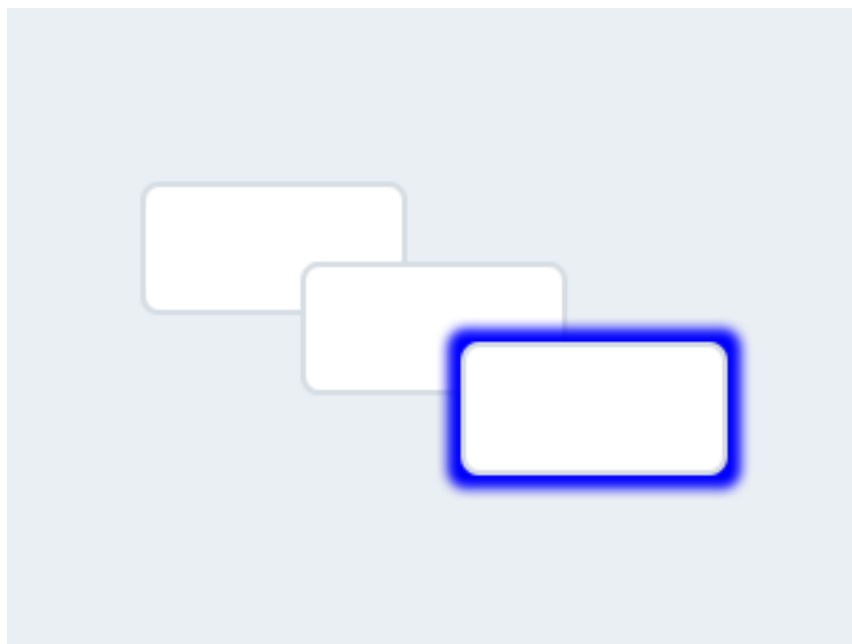


图 1: lvgl 中基础对象演示

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2
3  void lv_ex_obj_1(void)
4  {
```

(下页继续)

(续上页)

```

5      lv_obj_t * obj1;
6      obj1 = lv_obj_create(lv_scr_act(), NULL);
7      lv_obj_set_size(obj1, 100, 50);
8      lv_obj_align(obj1, NULL, LV_ALIGN_CENTER, -60, -30);
9
10     /*Copy the previous object and enable drag*/
11     lv_obj_t * obj2;
12     obj2 = lv_obj_create(lv_scr_act(), obj1);
13     lv_obj_align(obj2, NULL, LV_ALIGN_CENTER, 0, 0);
14     lv_obj_set_drag(obj2, true);
15
16     static lv_style_t style_shadow;
17     lv_style_init(&style_shadow);
18     lv_style_set_shadow_width(&style_shadow, LV_STATE_DEFAULT, 10);
19     lv_style_set_shadow_spread(&style_shadow, LV_STATE_DEFAULT, 5);
20     lv_style_set_shadow_color(&style_shadow, LV_STATE_DEFAULT, LV_COLOR_BLUE);
21
22     /*Copy the previous object (drag is already enabled)*/
23     lv_obj_t * obj3;
24     obj3 = lv_obj_create(lv_scr_act(), obj2);
25     lv_obj_add_style(obj3, LV_OBJ_PART_MAIN, &style_shadow);
26     lv_obj_align(obj3, NULL, LV_ALIGN_CENTER, 60, 30);
27 }

```

22.17 相关 API

22.17.1 Typedefs

```

1      Typedefs uint8_t lv_design_mode_t
2      Typedefs uint8_t lv_design_res_t
3      /* 设计回调用于在屏幕上绘制对象。它接受对象，遮罩区域以及绘制对象的方式。*/
4      Typedefs lv_design_res_t(* lv_design_cb_t)(struct _lv_obj_t * obj, const lv_area_t *
↪ clip_area, lv_design_mode_t mode)
5
6      /* 发送到对象的事件类型。*/
7      Typedefs uint8_t lv_event_t
8
9      /* 事件回调。事件用于通知用户对该对象正在执行的某些操作。有关详细信息，请参见 lv_event_t。*/
10     Typedefs void (* lv_event_cb_t)(struct _lv_obj_t * obj, lv_event_t event )
11
12

```

(下页继续)

(续上页)

```

13     Typedefs uint8_t lv_signal_t
14     Typedefs lv_res_t (* lv_signal_cb_t) (struct _lv_obj_t * obj, lv_signal_t sign, void
    ↳ * param )
15     Typedefs uint8_t lv_protect_t
16     Typedefs uint8_t lv_state_t
17     Typedefs struct _lv_obj_t lv_obj_t
18     Typedefs uint8_t lv_obj_part_t

```

22.17.2 enums

```

1     enum [anonymous]
2     设计模式
3
4     值:
5     enumerator LV_DESIGN_DRAW_MAIN /* 绘制对象的主要部分 */
6     enumerator LV_DESIGN_DRAW_POST /* 在对象上绘画 */
7     enumerator LV_DESIGN_COVER_CHK /* 检查对象是否完全覆盖 “ mask_p” 区域 */
8
9
10    enum [anonymous]
11    设计结果
12    值:
13    enumerator LV_DESIGN_RES_OK /* 准备好 */
14    enumerator LV_DESIGN_RES_COVER /* LV_DESIGN_COVER_CHK 如果区域完全覆盖, 则返回 */
15    enumerator LV_DESIGN_RES_NOT_COVER /* LV_DESIGN_COVER_CHK 如果未覆盖区域, 则返回 */
16    enumerator LV_DESIGN_RES_MASKED /* LV_DESIGN_COVER_CHK 如果区域被遮盖, 则返回 (子类
    也不会遮盖) */
17
18
19    enum [anonymous]
20    值:
21    enumerator LV_EVENT_PRESSED /* 该对象已被按下 */
22    enumerator LV_EVENT_PRESSING /* 按下对象 (按下时连续调用) */
23    enumerator LV_EVENT_PRESS_LOST /* 用户仍在按, 但将光标/手指滑离对象 */
24    enumerator LV_EVENT_SHORT_CLICKED /* 用户在短时间内按下对象, 然后释放它。如果拖动则
    不调用。 */
25    enumerator LV_EVENT_LONG_PRESSED /* 对象已被按下至少 LV_INDEV_LONG_PRESS_TIME。
    如果拖动则不调用。 */
26    enumerator LV_EVENT_LONG_PRESSED_REPEAT /* LV_INDEV_LONG_PRESS_TIME 每 LV_INDEV_
    ↳ LONG_PRESS_REP_TIME 毫秒调用一次。如果拖动则不调用。 */
27    enumerator LV_EVENT_CLICKED /* 如果未拖动则调用释放 (无论长按) */
28    enumerator LV_EVENT_RELEASED /* 在对象释放后的每种情况下调用 */

```

(下页继续)

(续上页)

```

29     enumerator LV_EVENT_DRAG_BEGIN
30     enumerator LV_EVENT_DRAG_END
31     enumerator LV_EVENT_DRAG_THROW_BEGIN
32     enumerator LV_EVENT_GESTURE /* 对象已被手势 */
33     enumerator LV_EVENT_KEY
34     enumerator LV_EVENT_FOCUSED
35     enumerator LV_EVENT_DEFOCUSED
36     enumerator LV_EVENT_LEAVE
37     enumerator LV_EVENT_VALUE_CHANGED /* 对象的值已更改 (即, 滑块已移动) */
38     enumerator LV_EVENT_INSERT
39     enumerator LV_EVENT_REFRESH
40     enumerator LV_EVENT_APPLY /* 单击“确定”, “应用”或类似的特定按钮 */
41     enumerator LV_EVENT_CANCEL /* 单击了“关闭”, “取消”或类似的特定按钮 */
42     enumerator LV_EVENT_DELETE /* 对象被删除 */
43     enumerator _LV_EVENT_LAST /* 活动数量 */
44
45
46     enum [anonymous]
47     信号供对象本身使用或扩展对象的功能。应用程序应使用 lv_obj_set_event_cb 来通知对象上发生的事件。
48     值:
49     enumerator LV_SIGNAL_CLEANUP /* 对象被删除 */
50     enumerator LV_SIGNAL_CHILD_CHG /* 子项已删除/添加 */
51     enumerator LV_SIGNAL_COORD_CHG /* 对象坐标/大小已更改 */
52     enumerator LV_SIGNAL_PARENT_SIZE_CHG /* 父类的大小已更改 */
53     enumerator LV_SIGNAL_STYLE_CHG /* 对象的样式已更改 */
54     enumerator LV_SIGNAL_BASE_DIR_CHG /* 基本目录已更改 */
55     enumerator LV_SIGNAL_REFR_EXT_DRAW_PAD /* 对象的额外填充已更改 */
56     enumerator LV_SIGNAL_GET_TYPELVGL /* 需要检索对象的类型 */
57     enumerator LV_SIGNAL_GET_STYLE /* 获取对象的样式 */
58     enumerator LV_SIGNAL_GET_STATE_DSC /* 获取对象的状态 */
59     enumerator LV_SIGNAL_HIT_TEST /* 命中测试 */
60     enumerator LV_SIGNAL_PRESSED /* 该对象已被按下 */
61     enumerator LV_SIGNAL_PRESSING /* 按下对象 (按下时连续调用) */
62     enumerator LV_SIGNAL_PRESS_LOST /* 用户仍在按, 但将光标/手指滑离对象 */
63     enumerator LV_SIGNAL_RELEASED /* 用户在短时间内按下对象, 然后释放它。如果拖动则不
调用。 */
64     enumerator LV_SIGNAL_LONG_PRESS /* 对象已被按下至少 LV_INDEV_LONG_PRESS_TIME。
如果拖动则不调用。 */
65     enumerator LV_SIGNAL_LONG_PRESS_REP /* LV_INDEV_LONG_PRESS_TIME 每 LV_INDEV_
→LONG_PRESS_REP_TIME 毫秒调用一次。如果拖动则不调用。 */
66     enumerator LV_SIGNAL_DRAG_BEGIN
67     enumerator LV_SIGNAL_DRAG_THROW_BEGIN
68     enumerator LV_SIGNAL_DRAG_END

```

(下页继续)

(续上页)

```

69     enumerator LV_SIGNAL_GESTURE /* 对象已被手势 */
70     enumerator LV_SIGNAL_LEAVE /* 通过输入设备单击或选择另一个对象 */
71     enumerator LV_SIGNAL_FOCUS
72     enumerator LV_SIGNAL_DEFOCUS
73     enumerator LV_SIGNAL_CONTROL
74     enumerator LV_SIGNAL_GET_EDITABLE
75
76     enum [anonymous]
77     值:
78     enumerator LV_PROTECT_NONE
79     enumerator LV_PROTECT_CHILD_CHG /* 禁用子类更换信号。被库使用 */
80     enumerator LV_PROTECT_PARENT /* 防止自动更改父级 (例如, 在 lv_page 中) */
81     enumerator LV_PROTECT_POS /* 防止自动定位 (例如, 在 lv_cont 布局中) */
82     enumerator LV_PROTECT_FOLLOW /* 防止在自动排序中遵循对象 (例如, 在 lv_cont PRETTY 布
局中) */
83     enumerator LV_PROTECT_PRESS_LOST /* 如果 inde v 按的是该对象, 但在按时却扫了出去, 请勿搜
索其他对象。 */
84     enumerator LV_PROTECT_CLICK_FOCUS /* 单击以防止聚焦对象 */
85     enumerator LV_PROTECT_EVENT_TO_DISABLED /* 甚至将事件传递给禁用的对象 */
86
87
88     enum [anonymous]
89     值:
90     enumerator LV_STATE_DEFAULT /* 默认 */
91     enumerator LV_STATE_CHECKED /* 选中 */
92     enumerator LV_STATE_FOCUSED /* 聚焦 */
93     enumerator LV_STATE_EDITED /* 编辑 */
94     enumerator LV_STATE_HOVERED /* 盘旋 */
95     enumerator LV_STATE_PRESSED /* 按下 */
96     enumerator LV_STATE_DISABLED /* 禁用 */
97
98     enum [anonymous]
99     值:
100     enumerator LV_OBJ_PART_MAIN
101     enumerator _LV_OBJ_PART_VIRTUAL_LAST
102     enumerator _LV_OBJ_PART_REAL_LAST
103     enumerator LV_OBJ_PART_ALL

```

22.17.3 函数

```

1  void lv_init( void )    /* LVGL 初始化 */
2  void lv_deinit( void ) /* 取消初始化 “lv” 库当前仅在不使用自定义分配器或启用 GC 时实现。*/
3  lv_obj_t * lv_obj_create(lv_obj_t *parent,constlv_obj_t * copy)
4  功能：创建一个基本对象
5  返回：指向新对象的指针
6  形参：
7  parent：指向父对象的指针。如果为 NULL，则将创建一个屏幕
8  copy： 指向基础对象的指针，如果不为 NULL，则将从其复制新对象
9
10
11  lv_res_t lv_obj_del(lv_obj_t * obj )
12  功能：删除 “obj” 及其所有子项
13  返回：LV_RES_INV，因为对象已删除
14  形参：
15  obj：指向要删除的对象的指针
16
17
18  void lv_obj_del_anim_ready_cb(lv_anim_t * a )
19  功能：动画就绪回调中易于使用的函数，可在动画就绪时删除对象
20  形参：
21  a：指向动画的指针
22
23
24  void lv_obj_del_async(struct _lv_obj_t * obj )
25  功能：辅助函数，用于异步删除对象。在无法直接在 LV_EVENT_DELETE 处理程序（即父级）中删除对象的情
    况下很有用。
26  形参：
27  obj：要删除的对象
28
29
30  void lv_obj_clean(lv_obj_t * obj )
31  功能：删除对象的所有子对象
32  形参：
33  obj：指向对象的指针
34
35
36  void lv_obj_invalidate_area(constlv_obj_t * obj,constlv_area_t *area)
37  功能：将对象的区域标记为 void 。此区域将由 “lv_refr_task” 重绘
38  形参：
39  obj：指向对象的指针
40  area：要重绘的区域
41

```

(下页继续)

(续上页)

```
void lv_obj_invalidate(const lv_obj_t * obj )
```

功能：将对象标记为 void ， 因此将通过 “ lv_refr_task” 重绘其当前位置

形参：

obj：指向对象的指针

```
bool lv_obj_area_is_visible(const lv_obj_t * obj, lv_area_t * area )
```

判断对象的某个区域现在是否可见 （甚至部分可见）

返回:true：可见；假：不可见 （隐藏， 在父母之外， 在其他屏幕上等）

形参：

obj：指向对象的指针

area：检查。该区域的可见部分将写回到这里。

```
bool lv_obj_is_visible(const lv_obj_t * obj )
```

功能：判断一个对象现在是否可见 （甚至部分可见）

返回:true：可见；假：不可见 （隐藏， 在父母之外， 在其他屏幕上等）

形参：

obj：指向对象的指针

```
void lv_obj_set_parent(lv_obj_t * obj, lv_obj_t * parent)
```

功能：为对象设置一个新的父对象。其相对位置将相同。

形参：

obj：指向对象的指针。不能是屏幕。

parent：指向新父对象的指针。（不能为 NULL）

```
void lv_obj_move_foreground(lv_obj_t * obj )
```

功能：移动并反对前景

形参：

obj：指向对象的指针

```
void lv_obj_move_background(lv_obj_t * obj )
```

功能：移动并反对背景

形参：

obj：指向对象的指针

```
void lv_obj_set_pos(lv_obj_t * obj, lv_coord_t x, lv_coord_t y )
```

功能：设置对象的相对位置 （相对于父对象）

(下页继续)

(续上页)

形参:

obj: 指向对象的指针

x: 距父级左侧的新距离

y: 距父级顶部的新距离

```
void lv_obj_set_x(lv_obj_t * obj,lv_coord_t x )
```

功能: 设置对象的 x 坐标

形参:

obj: 指向对象的指针

x: 从父级到左侧的新距离

```
void lv_obj_set_y(lv_obj_t * obj,lv_coord_t y )
```

功能: 设置对象的 y 坐标

形参:

obj: 指向对象的指针

y: 距父级顶部的新距离

```
void lv_obj_set_size(lv_obj_t * obj,lv_coord_t w,lv_coord_t h )
```

功能: 设置对象的大小

形参:

obj: 指向对象的指针

w: 新宽度

h: 新高度

```
void lv_obj_set_width(lv_obj_t * obj,lv_coord_t w )
```

功能: 设置对象的宽度

形参:

obj: 指向对象的指针

w: 新宽度

```
void lv_obj_set_height(lv_obj_t * obj,lv_coord_t h )
```

功能: 设定物件的高度

形参:

obj: 指向对象的指针

h: 新高度

```
void lv_obj_set_width_fit(lv_obj_t * obj,lv_coord_t w )
```

(下页继续)

(续上页)

功能：设置通过左右填充减少的宽度。

形参：

obj：指向对象的指针

w：没有填充的宽度

```
void lv_obj_set_height_fit(lv_obj_t * obj,lv_coord_t h )
```

功能：设置通过顶部和底部填充减小的高度。

形参：

obj：指向对象的指针

h：不带衬垫的高度

```
void lv_obj_set_width_margin(lv_obj_t * obj,lv_coord_t w )
```

功能：通过考虑左边距和右边距来设置对象的宽度。物体的宽度为 $obj_w = w - margin_left - margin_right$

形参：

obj：指向对象的指针

w：包括边距在内的新高度

```
void lv_obj_set_height_margin(lv_obj_t * obj,lv_coord_t h )
```

功能：通过考虑顶部和底部边距来设置对象的高度。物体高度将为 $obj_h = h - margin_top - margin_bottom$

形参：

obj：指向对象的指针

h：包括边距在内的新高度

```
void lv_obj_align(lv_obj_t * obj,constlv_obj_t * base,lv_align_t align,lv_coord_t x_ofs,lv_coord_t y_ofs )
```

功能：将一个对象与另一个对象对齐。

形参：

obj：指向要对齐的对象的指针

base：指向对象的指针（如果为 NULL，则使用父对象）。'obj' 将与其对齐。

align：对齐类型（请参见“lv_align_t”enum）

x_ofs：对齐后的 x 坐标偏移

y_ofs：对齐后的 y 坐标偏移

```
void lv_obj_align_x(lv_obj_t * obj,constlv_obj_t * base,lv_align_t align,lv_coord_t x_ofs )
```

功能：将一个对象与另一个对象水平对齐。

(下页继续)

(续上页)

形参:

obj: 指向要对齐的对象的指针

base: 指向对象的指针 (如果为 NULL, 则使用父对象)。'obj' 将与其对齐。

align: 对齐类型 (请参见 “ lv_align_t” enum)

x_ofs: 对齐后的 x 坐标偏移

```
void lv_obj_align_y(lv_obj_t * obj, const lv_obj_t * base, lv_align_t align, lv_coord_t y_ofs )
```

功能: 将一个对象与另一个对象垂直对齐。

形参:

obj: 指向要对齐的对象的指针

base: 指向对象的指针 (如果为 NULL, 则使用父对象)。'obj' 将与其对齐。

align: 对齐类型 (请参见 “ lv_align_t” enum)

y_ofs: 对齐后的 y 坐标偏移

```
void lv_obj_align_mid(lv_obj_t * obj, const lv_obj_t * base, lv_align_t align, lv_coord_t x_ofs, lv_coord_t y_ofs )
```

功能: 将一个对象与另一个对象对齐。

形参:

obj: 指向要对齐的对象的指针

base: 指向对象的指针 (如果为 NULL, 则使用父对象)。'obj' 将与其对齐。

align: 对齐类型 (请参见 “ lv_align_t” enum)

x_ofs: 对齐后的 x 坐标偏移

y_ofs: 对齐后的 y 坐标偏移

```
void lv_obj_align_mid_x(lv_obj_t * obj, const lv_obj_t * base, lv_align_t align, lv_coord_t x_ofs )
```

功能: 将对象的中点与另一个对象水平对齐。

形参:

obj: 指向要对齐的对象的指针

base: 指向对象的指针 (如果为 NULL, 则使用父对象)。'obj' 将与其对齐。

align: 对齐类型 (请参见 “ lv_align_t” enum)

x_ofs: 对齐后的 x 坐标偏移

```
void lv_obj_align_mid_y(lv_obj_t * obj, const lv_obj_t * base, lv_align_t align, lv_coord_t y_ofs )
```

功能: 将对象的中点与另一个对象垂直对齐。

形参:

obj: 指向要对齐的对象的指针

(下页继续)

(续上页)

base: 指向对象的指针 (如果为 NULL, 则使用父对象)。'obj' 将与其对齐。

align: 对齐类型 (请参见 “ lv_align_t” enum)

y_ofs: 对齐后的 y 坐标偏移

```
void lv_obj_realign(lv_obj_t * obj )
```

功能: 根据最后一个 lv_obj_align 参数重新对齐对象。

形参:

obj: 指向对象的指针

```
void lv_obj_set_auto_realign(lv_obj_t * obj, bool en )
```

功能: 当对象的大小根据最后一个 lv_obj_align 参数更改时, 启用对象的自动重新对齐。

形参:

obj: 指向对象的指针

en: true: 启用自动重新对齐; false: 禁用自动重新对齐

```
void lv_obj_set_ext_click_area(lv_obj_t * OBJ, lv_coord_t left, lv_coord_t right, lv_
↳ coord_t top, lv_coord_t bottom)
```

功能: 设置扩展的可点击区域的大小

形参:

obj: 指向对象的指针

left: 可扩展点击位于左侧 [px]

right: 可扩展点击位于右侧 [px]

top: 扩展的可点击位于顶部 [px]

bottom: 可扩展点击位于底部 [px]

```
void lv_obj_add_style(lv_obj_t * obj, uint8_t part, lv_style_t * style )
```

功能: 将新样式添加到对象的样式列表。

形参:

obj: 指向对象的指针

part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_

↳ KNOB

style: 指向要添加样式的指针 (仅会保存其指针)

```
void lv_obj_remove_style(lv_obj_t * obj, uint8_t part, lv_style_t * style )
```

功能: 从对象的样式列表中删除样式。

形参:

obj: 指向对象的指针

part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_

↳ KNOB

(下页继续)

(续上页)

style: 指向要删除的样式的指针

```
void lv_obj_clean_style_list(lv_obj_t * obj,uint8_t part )
```

功能: 将样式重置为默认 (空) 状态。释放所有已使用的内存, 并取消待处理的相关转换。通常用于 `LV_`
↪ SIGN_CLEAN_UP。

形参:

obj: 指向对象的指针

part: 应重置样式列表的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_`
↪ KNOB

```
void lv_obj_reset_style_list(lv_obj_t * obj,uint8_t part )
```

功能: 将样式重置为默认 (空) 状态。释放所有已使用的内存, 并取消待处理的相关转换。还通知对象有关样式更改的信息。

形参:

obj: 指向对象的指针

part: 应重置样式列表的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_`
↪ KNOB

```
void lv_obj_refresh_style(lv_obj_t * obj,uint8_t part,lv_style_property_t prop )
```

功能: 通知对象 (及其子对象) 其样式已修改

形参:

obj: 指向对象的指针

prop: LV_STYLE_PROP_ALL 或 LV_STYLE_... 财产。它用于优化需要刷新内容。

```
void lv_obj_report_style_mod( lv_style_t *style)
```

功能: 修改样式时通知所有对象

形参:

style: 指向样式的指针。仅具有这种样式的对象将被通知 (NULL 通知所有对象)

```
void _lv_obj_set_style_local_color(lv_obj_t * OBJ,uint8_t part,lv_style_property_`  
↪ t prop,lv_color_t the)
```

功能: 在给定状态下设置对象的一部分的局部样式属性。

注意:

1. 不应直接使用。请改用特定的属性 get 函数。例如: lv_obj_style_get_border_opa()
2. 由于性能原因, 不会检查属性是否确实具有颜色类型

形参:

obj: 指向对象的指针

part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_`
↪ KNOB

(下页继续)

(续上页)

```

284     prop: 样式属性与状态或。例如 LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_
↪STATE_POS)
285     the: 要设置的值
286
287
288     void _lv_obj_set_style_local_int(lv_obj_t * OBJ,uint8_t part,lv_style_property_t_
↪prop,lv_style_int_t the)
289     功能: 在给定状态下设置对象的一部分的局部样式属性。
290     注意:
291     1. 不应直接使用。请改用特定的属性 get 函数。例如: lv_obj_style_get_border_opa()
292     2. 出于性能原因, 不检查该属性是否确实具有整数类型
293     形参:
294     obj: 指向对象的指针
295     part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN,LV_BTN_PART_MAIN,LV_SLIDER_PART_
↪KNOB
296     prop: 样式属性与状态或。例如 LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_
↪STATE_POS)
297     the: 要设置的值
298
299
300     void _lv_obj_set_style_local_opa(lv_obj_t * obj,uint8_t part,lv_style_property_t_
↪prop,lv_opa_t opa )
301     功能: 在给定状态下设置对象的一部分的局部样式属性。
302     注意:
303     1. 不应直接使用。请改用特定的属性 get 函数。例如: lv_obj_style_get_border_opa()
304     2. 由于性能原因, 不检查属性是否确实具有不透明度类型
305     形参:
306     obj: 指向对象的指针
307     part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN,LV_BTN_PART_MAIN,LV_SLIDER_PART_
↪KNOB
308     prop: 样式属性与状态或。例如 LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_
↪POS)
309     the: 要设置的值
310
311
312     void _lv_obj_set_style_local_ptr(lv_obj_t * obj,uint8_t type,lv_style_property_t_
↪prop,constvoid *the)
313     功能: 在给定状态下设置对象的一部分的局部样式属性。
314     注意:
315     1. 不应直接使用。请改用特定的属性 get 函数。例如: lv_obj_style_get_border_opa()
316     2. 出于性能原因, 不检查该属性是否确实具有指针类型
317     形参:
318     obj: 指向对象的指针

```

(下页继续)

(续上页)

```

319     part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_
↪ KNOB
320     prop: 样式属性与状态或。例如 LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_
↪ POS)
321     the: 要设置的值
322
323
324     bool lv_obj_remove_style_local_prop(lv_obj_t * obj, uint8_t part, lv_style_property_
↪ t prop )
325     功能: 从具有给定状态的对象的一部分中删除局部样式属性。
326     注意:
327     不应直接使用。请改用特定的属性删除功能。例如: lv_obj_style_remove_border_opa()
328     返回: true: 已找到并删除了该属性; false: 找不到该属性
329     形参:
330     obj: 指向对象的指针
331     part: 对象的应删除样式属性的部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_
↪ KNOB
332     prop: 样式属性与状态或。例如 LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_
↪ POS)
333
334
335     void _lv_obj_disable_style_caching(lv_obj_t * obj, bool dis )
336     功能: 启用/禁用对象的样式缓存
337     形参:
338     obj: 指向对象的指针
339     dis: true: 禁用; false: 启用 (重新启用)
340
341
342     void lv_obj_set_hidden(lv_obj_t * obj, bool en )
343     功能: 隐藏对象。它不会显示和单击。
344     形参:
345     obj: 指向对象的指针
346     en: true: 隐藏对象
347
348     void lv_obj_set_adv_hittest(lv_obj_t * obj, bool en )
349     功能: 设置是否在对象上启用高级命中测试
350     形参:
351     obj: 指向对象的指针
352     en: true: 启用高级命中测试
353
354
355     void lv_obj_set_click(lv_obj_t * obj, bool en )
356     功能: 启用或禁用对象单击

```

(下页继续)

(续上页)

形参:

obj: 指向对象的指针

en: true: 使对象可点击

```
void lv_obj_set_top(lv_obj_t * obj, bool en )
```

功能: 如果单击该对象或其任何子对象, 则启用该对象

形参:

obj: 指向对象的指针

en: true: 启用自动顶部功能

```
void lv_obj_set_drag(lv_obj_t * obj, bool en )
```

功能: 启用对象拖动

形参:

obj: 指向对象的指针

en: true: 使对象可拖动

```
void lv_obj_set_drag_dir(lv_obj_t * obj, lv_drag_dir_t drag_dir )
```

功能: 设置可以拖动对象的方向

形参:

obj: 指向对象的指针

drag_dir: 允许的拖动方向的按位或

```
void lv_obj_set_drag_throw(lv_obj_t * obj, bool en )
```

功能: 拖动后启用对象投掷

形参:

obj: 指向对象的指针

en: true: 启用拖动

```
void lv_obj_set_drag_parent(lv_obj_t * obj, bool en )
```

功能: 启用将父项用于拖动相关的操作。如果尝试拖动对象, 则将父对象移动

形参:

obj: 指向对象的指针

en: true: 为对象启用“拖动父对象”

```
void lv_obj_set_focus_parent(lv_obj_t * obj, bool en )
```

功能: 启用以将父级用于焦点状态。当对象聚焦时, 父级将获得状态 (仅可见)

形参:

(下页继续)

(续上页)

```

400 obj: 指向对象的指针
401 en: true: 为对象启用 “焦点父对象”
402
403
404 void lv_obj_set_gesture_parent(lv_obj_t * obj, bool en )
405 功能: 启用将父项用于手势相关操作。如果尝试手势对象, 则将父对象移动
406 形参:
407 obj: 指向对象的指针
408 en: true: 为对象启用 “手势父级”
409
410
411 void lv_obj_set_parent_event(lv_obj_t * obj, bool en )
412 功能: 也将事件传播给父级
413 形参:
414 obj: 指向对象的指针
415 en: true: 启用事件传播
416
417
418 void lv_obj_set_base_dir(lv_obj_t * obj, lv_bidi_dir_t dir )
419 功能: 设置对象的基本方向
420 注意:
421 仅在启用 LV_USE_BIDI 的情况下才有效。
422 形参:
423 obj: 指向对象的指针
424 dir: 新的基本方向。 LV_BIDI_DIR_LTR/RTL/AUTO/INHERIT
425
426
427 void lv_obj_add_protect(lv_obj_t * obj, uint8_t prot )
428 功能: 在保护字段中设置一个或多个位
429 形参:
430 obj: 指向对象的指针
431 prot: 来自 “或” 的值 lv_protect_t
432
433
434 void lv_obj_clear_protect(lv_obj_t * obj, uint8_t prot )
435 功能: 清除保护字段中的一个或多个位
436 形参:
437 obj: 指向对象的指针
438 prot: 来自 “或” 的值 lv_protect_t
439
440
441 void lv_obj_set_state(lv_obj_t * obj, lv_state_t state)
442 功能: 设置对象的状态 (完全覆盖)。如果在样式中指定, 则过渡动画将从前一状态开始到当前状态

```

(下页继续)

(续上页)

```

443 形参:
444  obj: 指向对象的指针
445  state: 新状态
446
447
448  void lv_obj_add_state(lv_obj_t * obj, lv_state_t state)
449  功能: 将一个或多个给定状态添加到对象。其他状态位将保持不变。如果在样式中指定, 则过渡动画将从前一状态开始到当前状态
450  形参:
451  obj: 指向对象的指针
452  state: 要添加的状态位。例如 LV_STATE_PRESSED | LV_STATE_FOCUSED
453
454
455  void lv_obj_clear_state(lv_obj_t * obj, lv_state_t state)
456  功能: 删除对象的给定状态。其他状态位将保持不变。如果在样式中指定, 则过渡动画将从前一状态开始到当前状态
457  形参:
458  obj: 指向对象的指针
459  state: 要删除的状态位。例如 LV_STATE_PRESSED | LV_STATE_FOCUSED
460
461
462  void lv_obj_finish_transitions(lv_obj_t * obj, uint8_t part )
463  功能: 在对象的一部分上完成所有挂起的过渡
464  形参:
465  obj: 指向对象的指针
466  part: 对象的一部分, 例如 LV_BRN_PART_MAIN 或 LV_OBJ_PART_ALL 所有部分
467
468
469  void lv_obj_set_event_cb(lv_obj_t * obj, lv_event_cb_t event_cb )
470  功能: 为对象设置事件处理函数。用户用于对对象发生的事件做出反应。
471  形参:
472  obj: 指向对象的指针
473  event_cb: 新事件功能
474
475
476  lv_res_t lv_event_send(lv_obj_t * obj, lv_event_t event, const void * data)
477  功能: 向对象发送事件
478  返回: LV_RES_OK: obj 在事件中没有被删除; LV_RES_INV: obj 在事件中被删除
479  形参:
480  obj: 指向对象的指针
481  event: 来自的事件类型 lv_event_t。
482  data: 取决于对象类型和事件的任意数据。(通常 NULL)
483

```

(下页继续)

(续上页)

```
lv_res_t lv_event_send_refresh(lv_obj_t * obj )
```

功能：将 LV_EVENT_REFRESH 事件发送到对象

返回：LV_RES_OK：成功，LV_RES_INV：对象由于该事件而变得 void（例如，删除）。

形参：

obj：指向一个对象。（不能为 NULL）

```
void lv_event_send_refresh_recursive(lv_obj_t * obj )
```

功能：将 LV_EVENT_REFRESH 事件发送给对象及其所有子对象

形参：

obj：指向一个对象的指针，或者为 NULL 以刷新所有显示的所有对象

```
lv_res_t lv_event_send_func(lv_event_cb_t event_xcb,lv_obj_t * obj,lv_event_t_
↪event,constvoid *data)
```

功能：使用对象，事件和数据调用事件函数。

返回：LV_RES_OK：obj 在事件中没有被删除；LV_RES_INV：obj 在事件中被删除

形参：

event_xcb：事件回调函数。如果 NULL LV_RES_OK 返回，则不执行任何操作。（参数名称中的“x”表示它不是完全通用的函数，因为它不遵循约定）func_name(object, callback, ...)

obj：指向与事件关联的对象的指针（可以 NULL 简单地调用 event_cb）

event：一个事件

data：指向自定义数据的指针

```
constvoid * lv_event_get_data(void )
```

功能：获取 data 当前事件的参数

返回

该 data 参数

```
void lv_obj_set_signal_cb(lv_obj_t * obj,lv_signal_cb_t signal_cb )
```

功能：设置对象的信号功能。由库内部使用。始终在新信号中调用前一个信号功能。

形参：

obj：指向对象的指针

signal_cb：新信号功能

```
lv_res_t lv_signal_send(lv_obj_t * obj,lv_signal_t signal,void * param )
```

功能：向对象发送事件

返回：LV_RES_OK 或 LV_RES_INV

形参：

obj：指向对象的指针

(下页继续)

(续上页)

event: 来自的事件类型 lv_event_t。

void lv_obj_set_design_cb(lv_obj_t * obj, lv_design_cb_t design_cb)

功能: 为对象设置新的设计功能

形参:

obj: 指向对象的指针

design_cb: 新的设计功能

void * lv_obj_allocate_ext_attr(lv_obj_t * obj, uint16_t ext_size)

功能: 分配一个新的分机。对象的数据

返回: 指向分配的 ext 的指针

形参:

obj: 指向对象的指针

ext_size: 新分机的大小。数据

void lv_obj_refresh_ext_draw_pad(lv_obj_t * obj)

功能: 向对象发送 “ LV_SIGNAL_REFR_EXT_SIZE” 信号以刷新扩展的绘制区域。lv_obj_↪invalidate(obj) 此功能后, 需要手动使对象 void 。

形参:

obj: 指向对象的指针

lv_obj_t * lv_obj_get_screen(const lv_obj_t * obj)

功能: 返回对象的屏幕

返回: 指向屏幕的指针

形参:

obj: 指向对象的指针

lv_disp_t * lv_obj_get_disp(const lv_obj_t * obj)

功能: 获取对象的显示

返回: 指针对象的显示

lv_obj_t * lv_obj_get_parent(const lv_obj_t * obj)

功能: 返回对象的父对象

返回: 指向 “ obj” 的父对象的指针

形参:

obj: 指向对象的指针

(下页继续)

(续上页)

```
lv_obj_t * lv_obj_get_child(constlv_obj_t * obj,constlv_obj_t * child )
```

功能：遍历对象的子项（从“最新的，最后创建的”开始）

返回：'act_child' 之后的子级；如果没有更多子级，则为 NULL

形参：

obj：指向对象的指针

child：第一次调用时为 NULL，以获取下一个子类，以后再返回上一个返回值

```
lv_obj_t * lv_obj_get_child_back(constlv_obj_t * obj,constlv_obj_t * child )
```

功能：遍历对象的子项（从“最早的”开始，首先创建）

返回：'act_child' 之后的子级；如果没有更多子级，则为 NULL

形参：

obj：指向对象的指针

child：第一次调用时为 NULL，以获取下一个子类，以后再返回上一个返回值

```
uint16_t lv_obj_count_children(constlv_obj_t * obj )
```

功能：计算对象的子代（仅直接在“obj”上的子代）

返回：'obj' 的子代号

形参：

obj：指向对象的指针

```
uint16_t lv_obj_count_children_recursive(constlv_obj_t * obj )
```

功能：递归计算对象的子代

返回：'obj' 的子代号

形参：

obj：指向对象的指针

```
void lv_obj_get_coords(constlv_obj_t * obj,lv_area_t * cords_p )
```

功能：将对象的坐标复制到区域

形参：

obj：指向对象的指针

cords_p：指向存储坐标的区域的指针

```
void lv_obj_get_inner_coords(constlv_obj_t * obj,lv_area_t * coords_p )
```

功能：减少由 lv_obj_get_coords() 对象的图形可用区域重试的区域。（没有边框的大小或其他额外的图形元素）

形参：

coords_p：将结果区域存储在此处

(下页继续)

(续上页)

```
609
610 lv_coord_t lv_obj_get_x(constlv_obj_t * obj )
611 功能：获取对象的 x 坐标
612 返回：“ obj” 到其父对象左侧的距离
613 形参：
614 obj：指向对象的指针
615
616
617 lv_coord_t lv_obj_get_y(constlv_obj_t * obj )
618 功能：获取对象的 y 坐标
619 返回：“ obj” 到其父对象顶部的距离
620 形参：
621 obj：指向对象的指针
622
623
624 lv_coord_t lv_obj_get_width(constlv_obj_t * obj )
625 功能：获取对象的宽度
626 返回：宽度
627 形参：
628 obj：指向对象的指针
629
630
631 lv_coord_t lv_obj_get_height(constlv_obj_t * obj )
632 功能：获取对象的高度
633 返回：高度
634 形参：
635 obj：指向对象的指针
636
637
638 lv_coord_t lv_obj_get_width_fit(constlv_obj_t * obj )
639 功能：通过左右填充减少宽度。
640 返回：仍然适合容器的宽度
641 形参：
642 obj：指向对象的指针
643
644
645 lv_coord_t lv_obj_get_height_fit(constlv_obj_t * obj )
646 功能：通过顶部底部填充减少高度。
647 返回：仍然适合容器的高度
648 形参：
649 obj：指向对象的指针
650
651
```

(下页继续)

(续上页)

```

652 lv_coord_t lv_obj_get_height_margin(lv_obj_t * obj )
653 功能：通过考虑顶部和底部边距来获取对象的高度。返回的高度将是 obj_h + margin_top + margin_
↪bottom
654 返回：包括您的边距在内的高度
655 形参：
656 obj：指向对象的指针
657
658
659 lv_coord_t lv_obj_get_width_margin(lv_obj_t * obj )
660 功能：通过考虑左边距和右边距来获得对象的宽度。返回的宽度将是 obj_w + margin_left + margin_
↪right
661 返回：包括您的边距在内的高度
662 形参：
663 obj：指向对象的指针
664
665
666 lv_coord_t lv_obj_get_width_grid(lv_obj_t * obj,uint8_t div,uint8_t span )
667 功能：划分对象的宽度并获得给定列数的宽度。考虑填充。
668 返回：根据给定参数的宽度
669 形参：
670 obj：指向对象的指针
671 div：表示假设有多少列。如果为 1，则宽度将设置为父级的宽度；如果为 2，则只有父级宽度的一半-父级的
内部填充；如果为 3，则只有第三个父级宽度-2 * 父级的内部填充
672 span：合并了多少列
673
674
675 lv_coord_t lv_obj_get_height_grid(lv_obj_t * obj,uint8_t div,uint8_t span )
676 功能：划分对象的高度并获得给定列数的宽度。考虑填充。
677 返回：根据给定参数的高度
678 形参：
679 obj：指向对象的指针
680 div：表示假设有多少行。如果为 1，则将设置父级的高度；如果为 2，则只有父级的一半高度-父级的内部填
充；如果只有 3，则只有第三级父级的高度-2 * 父级的内部填充
681 span：合并了多少行
682
683
684 bool lv_obj_get_auto_realign(const lv_obj_t * obj )
685 功能：获取对象的自动重新对齐属性。
686 返回:true：启用自动重新对齐；false：禁用自动重新对齐
687 形参：
688 obj：指向对象的指针
689
690

```

(下页继续)

(续上页)

```
lv_coord_t lv_obj_get_ext_click_pad_left(constlv_obj_t * obj )
```

功能：获取扩展的可点击区域的左侧填充

返回：扩展的左填充

形参：

obj：指向对象的指针

```
lv_coord_t lv_obj_get_ext_click_pad_right(constlv_obj_t * obj )
```

功能：获取扩展的可点击区域的正确填充

返回：扩展的右填充

形参：

obj：指向对象的指针

```
lv_coord_t lv_obj_get_ext_click_pad_top(constlv_obj_t * obj )
```

功能：获取扩展的可点击区域的顶部填充

返回：扩展的顶部填充

形参：

obj：指向对象的指针

```
lv_coord_t lv_obj_get_ext_click_pad_bottom(constlv_obj_t * obj )
```

功能：获取扩展的可点击区域的底部填充

返回：扩展的底部填充

形参：

obj：指向对象的指针

```
lv_coord_t lv_obj_get_ext_draw_pad(constlv_obj_t * obj )
```

功能：获取对象的扩展尺寸属性

返回：扩展尺寸属性

形参：

obj：指向对象的指针

```
lv_style_list_t * lv_obj_get_style_list(constlv_obj_t * obj,uint8_t 部分)
```

功能：获取对象零件的样式列表。

返回：指向样式列表的指针。(可以 NULL)

形参：

obj：指向对象的指针。

part: part 应该获取样式列表的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_↪PART_KNOB

(下页继续)

(续上页)

```
lv_style_int_t _lv_obj_get_style_int(const lv_obj_t * obj, uint8_t part, lv_style_
↪property_t prop )
```

功能：获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内
返回：

给定零件在当前状态下的属性值。如果找不到该属性，则将返回默认值。

注意：

1. 不应直接使用。请改用特定的属性 get 函数。例如：lv_obj_style_get_border_width()

2. 出于性能原因，不检查该属性是否确实具有整数类型

形参：

obj：指向对象的指针

part：应获取样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_

↪KNOB

prop：要获取的属性。例如 LV_STYLE_BORDER_WIDTH。对象的状态将在内部添加

```
lv_color_t _lv_obj_get_style_color(const lv_obj_t * obj, uint8_t part, lv_style_
↪property_t prop )
```

功能：获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内
返回：

给定零件在当前状态下的属性值。如果找不到该属性，则将返回默认值。

注意：

1. 不应直接使用。请改用特定的属性 get 函数。例如：lv_obj_style_get_border_color()

2. 由于性能原因，不会检查属性是否确实具有颜色类型

形参：

obj：指向对象的指针

part：应获取样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_

↪KNOB

prop：要获取的属性。例如 LV_STYLE_BORDER_COLOR。对象的状态将在内部添加

```
lv_opa_t _lv_obj_get_style_opa(const lv_obj_t * obj, uint8_t part, lv_style_property_
↪t prop )
```

功能：获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡，则将其考虑在内
返回：

给定零件在当前状态下的属性值。如果找不到该属性，则将返回默认值。

注意：

1. 不应直接使用。请改用特定的属性 get 函数。例如：lv_obj_style_get_border_opa()

2. 由于性能原因，不检查属性是否确实具有不透明度类型

形参：

obj：指向对象的指针

part：应获取样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_

↪KNOB

(下页继续)

(续上页)

prop: 要获取的属性。例如 LV_STYLE_BORDER_OPA。对象的状态将在内部添加

```
const void * _lv_obj_get_style_ptr(const lv_obj_t * obj, uint8_t part, lv_style_
↪property_t prop )
```

功能: 获取处于对象当前状态的对象的一部分的样式属性。如果正在运行过渡, 则将其考虑在内

返回:

给定零件在当前状态下的属性值。如果找不到该属性, 则将返回默认值。

注意:

1. 不应直接使用。请改用特定的属性 get 函数。例如: lv_obj_style_get_border_opa()

2. 出于性能原因, 不检查该属性是否确实具有指针类型

形参:

obj: 指向对象的指针

```
part: 应获取样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_
↪KNOB
```

prop: 要获取的属性。例如 LV_STYLE_TEXT_FONT。对象的状态将在内部添加

```
lv_style_t * lv_obj_get_local_style(lv_obj_t * obj, uint8_t part)
```

功能: 获取对象部分的局部样式。

返回:

指向本地样式的指针 (如果存在) NULL。

形参:

obj: 指向对象的指针

```
part: 应设置样式属性的对象部分。例如 LV_OBJ_PART_MAIN, LV_BTN_PART_MAIN, LV_SLIDER_PART_
↪KNOB
```

```
bool lv_obj_get_hidden(const lv_obj_t * obj )
```

功能: 获取对象的隐藏属性

返回:

true: 对象被隐藏

形参:

obj: 指向对象的指针

```
bool lv_obj_get_adv_hittest(const lv_obj_t * obj )
```

功能: 获取是否在对象上启用了高级命中测试

返回:

true: 启用高级匹配测试

形参:

obj: 指向对象的指针

(下页继续)

(续上页)

```
810
811 bool lv_obj_get_click(constlv_obj_t * obj )
812 功能：获取对象的点击启用属性
813 返回：
814 true：可点击对象
815 形参：
816 obj：指向对象的指针
817
818
819 bool lv_obj_get_top(constlv_obj_t * obj )
820 功能：获取对象的顶部启用属性
821 返回：
822 true：启用自动顶部功能
823 形参：
824 obj：指向对象的指针
825
826
827 bool lv_obj_get_drag(constlv_obj_t * obj )
828 功能：获取对象的拖动启用属性
829 返回：
830 true：对象可拖动
831 形参：
832 obj：指向对象的指针
833
834
835 lv_drag_dir_t lv_obj_get_drag_dir(constlv_obj_t * obj )
836 功能：获取可以拖动对象的方向
837 返回：
838 可以将对象拖入允许方向的按位或
839 形参：
840 obj：指向对象的指针
841
842
843 bool lv_obj_get_drag_throw(constlv_obj_t * obj )
844 功能：获取对象的拖动启动属性
845 返回：
846 true：启用拖动
847 形参：
848 obj：指向对象的指针
849
850
851 bool lv_obj_get_drag_parent(constlv_obj_t * obj )
852 功能：获取对象的拖动父级属性
```

(下页继续)

(续上页)

```
853  返回:
854  true: 启用拖动父级
855  形参:
856  obj: 指向对象的指针
857
858
859  bool lv_obj_get_focus_parent(constlv_obj_t * obj )
860  功能: 获取对象的焦点父级属性
861  返回:
862  true: 启用焦点父级
863  形参:
864  obj: 指向对象的指针
865
866
867  bool lv_obj_get_parent_event(constlv_obj_t * obj )
868  功能: 获取对象的拖动父级属性
869  返回:
870  true: 启用拖动父级
871  形参:
872  obj: 指向对象的指针
873
874
875  bool lv_obj_get_gesture_parent(constlv_obj_t * obj )
876  功能: 获取对象的手势父级属性
877  返回:
878  true: 启用手势父级
879  形参:
880  obj: 指向对象的指针
881
882
883  lv_bidi_dir_t lv_obj_get_base_dir(constlv_obj_t * obj )
884  uint8_t lv_obj_get_protect(constlv_obj_t * obj )
885  功能: 获取对象的保护字段
886  返回:
887  保护字段 ( “的” 或值 lv_protect_t)
888  形参:
889  obj: 指向对象的指针
890
891
892  bool lv_obj_is_protected(constlv_obj_t * obj,uint8_t prot )
893  功能: 检查是否已设置给定保护位字段的至少一位
894  返回:
895  false: 未设置任何给定位,true: 至少设置了一位
```

(下页继续)

(续上页)

形参:

obj: 指向对象的指针

prot: 保护要测试的位 (的“或”值 lv_protect_t)

lv_state_t lv_obj_get_state(const lv_obj_t * obj, uint8_t part)

lv_signal_cb_t lv_obj_get_signal_cb(const lv_obj_t * obj)

功能: 获取对象的信号功能

返回:

信号功能

形参:

obj: 指向对象的指针

lv_design_cb_t lv_obj_get_design_cb(const lv_obj_t * obj)

功能: 获取对象的设计功能

返回:

设计功能

形参:

obj: 指向对象的指针

lv_event_cb_t lv_obj_get_event_cb(const lv_obj_t * obj)

功能: 获取对象的事件函数

返回:

事件功能

形参:

obj: 指向对象的指针

bool lv_obj_is_point_on_coords(lv_obj_t * obj, const lv_point_t * 点)

功能:

检查给定的屏幕空间点是否在对象的坐标上。该方法主要用于高级命中测试算法, 以检查该点是否在对象内 (作为优化)。

形参:

obj: 要检查的对象

point: 屏幕空间点

bool lv_obj_hittest(lv_obj_t * obj, lv_point_t * point)

功能: 对在屏幕空间中特定位置的对象进行命中测试。

返回:

如果对象被认为在该点之下, 则返回 true

(下页继续)

(续上页)

形参:

obj: 对象进行命中测试

point: 屏幕空间点

```
void * lv_obj_get_ext_attr(constlv_obj_t * obj )
```

获取 ext 指针

返回:

ext 指针, 而不是动态版本用作 ext-> data1, 而不是 da(ext)-> data1

形参:

obj: 指向对象的指针

```
void lv_obj_get_type(constlv_obj_t * obj,lv_obj_type_t * buf )
```

功能: 获取对象及其祖先类型。将其名称以 type_buf 当前类型开头。例如 buf.type [0] = " lv_btn" ,
 ↪buf.type [1] = " lv_cont" ,buf.type [2] = " lv_obj"

形参:

obj: 指向应获取类型的对象的指针

buf: 指向用于 lv_obj_type_t 存储类型的缓冲区的指针

```
lv_obj_user_data_t lv_obj_get_user_data(constlv_obj_t * obj )
```

功能: 获取对象的用户数据

返回:

用户数据

形参:

obj: 指向对象的指针

```
lv_obj_user_data_t * lv_obj_get_user_data_ptr(constlv_obj_t * obj )
```

功能: 获取指向对象的用户数据的指针

返回:

指向用户数据的指针

形参:

obj: 指向对象的指针

```
void lv_obj_set_user_data(lv_obj_t * obj,lv_obj_user_data_t 数据)
```

功能: 设置对象的用户数据。数据将被复制。

形参:

obj: 指向对象的指针

data: 用户数据

(下页继续)

(续上页)

```
void * lv_obj_get_group(const lv_obj_t * obj )
```

功能：获取对象组

返回：

指向对象组的指针

形参：

obj: 指向对象的指针

```
bool lv_obj_is_focused(const lv_obj_t * obj )
```

功能：判断对象是否是组的聚焦对象。

返回：

true: 对象已聚焦, false: 对象未聚焦或不在组中

形参：

obj: 指向对象的指针

```
lv_obj_t * lv_obj_get_focused_obj(const lv_obj_t * obj )
```

功能：通过考虑获得真正专注的对象 focus_parent。

返回：

真正聚焦的对象

形参：

obj: 起始对象

```
lv_res_t lv_obj_handle_get_type_signal(lv_obj_type_t * buf, const char * name)
```

功能：用于信号回调中以处理 LV_SIGNAL_GET_TYPE 信号

返回：

LV_RES_OK

形参：

buf: 的指针 lv_obj_type_t。(param 在信号回调中)

name: 对象的名称。例如 “ lv_btn”。(仅保存指针)

```
void lv_obj_init_draw_rect_dsc(lv_obj_t * obj, uint8_t type, lv_draw_rect_dsc_t *  
↪ draw_dsc )
```

功能：根据对象的样式初始化矩形描述符

注意：

仅设置相关字段。例如，是否将不评估其他边框属性。border width == 0

形参：

obj: 指向对象的指针

type: 样式类型。例如 LV_OBJ_PART_MAIN, LV_BTN_SLIDER_KOB

draw_dsc: 描述符的初始化

(下页继续)

(续上页)

```

1022
1023
1024     void lv_obj_init_draw_label_dsc(lv_obj_t * obj,uint8_t 类型,lv_draw_label_dsc_t * _
↪draw_dsc )
1025     void lv_obj_init_draw_img_dsc(lv_obj_t * obj,uint8_t part,lv_draw_img_dsc_t * _
↪draw_dsc )
1026     void lv_obj_init_draw_line_dsc(lv_obj_t * obj,uint8_t part,lv_draw_line_dsc_t * _
↪draw_dsc )
1027     lv_coord_t lv_obj_get_draw_rect_ext_pad_size(lv_obj_t * obj,uint8_t part )
1028     功能：获取所需的额外大小（围绕对象部分）以绘制阴影，轮廓，值等。
1029     形参：
1030     obj：指向对象的指针
1031     part：对象的一部分
1032
1033
1034     void lv_obj_fade_in(lv_obj_t * obj,uint32_t 时间,uint32_t 延迟)
1035     功能：使用 opa_scale 动画淡入（从透明到完全覆盖）对象及其所有子对象。
1036     形参：
1037     obj：淡入的对象
1038     time：动画的持续时间 [ms]
1039     delay：等待动画开始播放 [ms]
1040
1041
1042     void lv_obj_fade_out(lv_obj_t * obj,uint32_t 时间,uint32_t 延迟)
1043     功能：使用 opa_scale 动画淡出（从完全覆盖到透明）对象及其所有子对象。
1044     形参：
1045     obj：淡入的对象
1046     time：动画的持续时间 [ms]
1047     delay：等待动画开始播放 [ms]
1048
1049
1050     bool lv_debug_check_obj_type(constlv_obj_t * obj,constchar * obj_type )
1051     功能：检查是否有给定类型的对象
1052     返回：
1053     true：有效
1054     形参：
1055     obj：指向对象的指针
1056     obj_type：对象的类型。（例如 “ lv_btn” ）
1057
1058
1059     bool lv_debug_check_obj_valid(constlv_obj_t * obj )
1060     功能：检查是否还有任何对象“处于活动状态”以及层次 struct 的一部分
1061     返回：

```

(下页继续)

(续上页)

```
1062 true: 有效
1063 形参:
1064 obj: 指向对象的指针
1065 obj_type: 对象的类型。(例如 “ lv_btn” )
```


23.1 概述

弧由背景弧和前景弧组成。两者都可以具有起始角度和终止角度以及厚度。

23.2 零件和样式

弧的主要部分称为 `LV_ARC_PART_MAIN`。它使用典型的背景样式属性绘制背景，并使用线型属性绘制圆弧。圆弧的大小和位置将遵守填充样式的属性。

`LV_ARC_PART_INDIC` 是虚拟零件，它使用线型属性绘制另一个弧。它的填充值是相对于背景弧线解释的。指示器圆弧的半径将根据最大填充值进行修改。

`LV_ARC_PART_KNOB` 是虚拟零件，它绘制在弧形指示器的末端。它使用所有背景属性和填充值。使用零填充时，旋钮的大小与指示器的宽度相同。较大的填充使其较大，较小的填充使其较小。

23.3 用法

23.3.1 角度

要设置背景角度, 请使用 `lv_arc_set_bg_angles(arc, start_angle, end_angle)` 函数或 `lv_arc_set_bg_start/end_angle(arc, start_angle)`。零度位于对象的右中间 (3 点钟), 并且度沿顺时针方向增加。角度应在 `[0; 360]` 范围内。

同样, `lv_arc_set_angles(arc, start_angle, end_angle)` 函数或 `lv_arc_set_start/end_angle(arc, start_angle)` 函数设置指示器弧的角度。

23.3.2 回转

可以使用 `lv_arc_set_rotation(arc, deg)` 添加到 0 度位置的偏移量。

23.3.3 范围和值

除了手动设置角度外, 弧还可以具有范围和值。要设置范围, 请使用 `lv_arc_set_range(arc, min, max)`, 并设置一个值, 请使用 `lv_arc_set_value(arc, value)`。使用范围和值, 指示器的角度将在背景角度之间映射。

注意, 设置角度和值是独立的。应该使用值和角度设置。两者混合可能会导致意外的效果。

23.3.4 类型

弧可以具有不同的“类型”。它们用 `lv_arc_set_type` 设置。存在以下类型:

- `LV_ARC_TYPE_NORMAL` 指示器弧顺时针绘制 (最小电流)
- `LV_ARC_TYPE_REVERSE` 指示器弧沿逆时针方向绘制 (最大电流)
- `LV_ARC_TYPE_SYMMETRIC` 从中间点绘制到当前值的指示弧。

23.4 事件

除 通用事件 外, 弧还发送以下特殊事件:

- `LV_EVENT_VALUE_CHANGED` 在按下/拖动弧以设置新值时发送。

了解有关 事件 的更多信息。

23.5 按键

对象类型不处理任何输入按键。

进一步了解 按键。

23.6 范例

23.6.1 简单弧

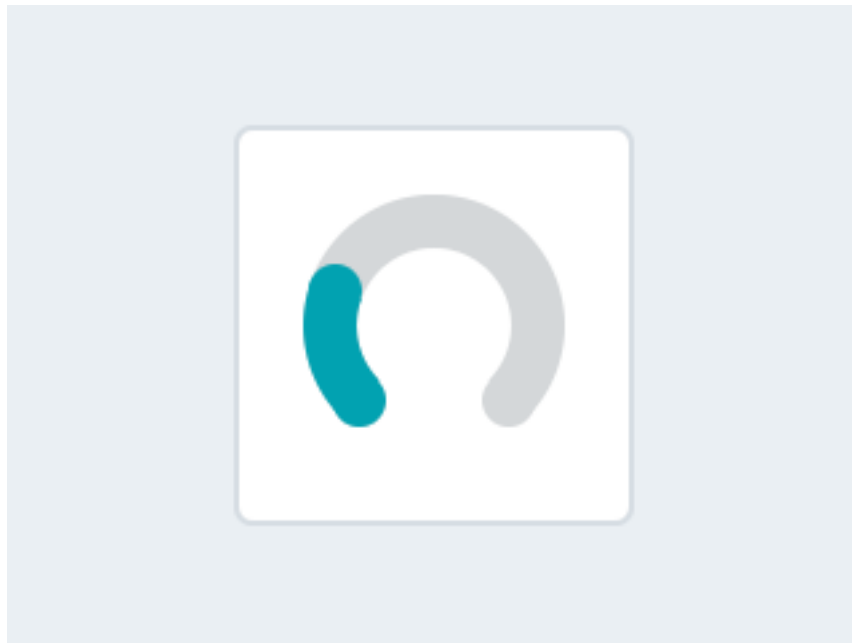


图 1: 简单弧

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2
3  #if LV_USE_ARC
4
5  void lv_ex_arc_1(void)
6  {
7      /*Create an Arc*/
8      lv_obj_t * arc = lv_arc_create(lv_scr_act(), NULL);
9      lv_arc_set_end_angle(arc, 200);
10     lv_obj_set_size(arc, 150, 150);
11     lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0);
```

(下页继续)

(续上页)

```
12     }  
13  
14     #endif
```

23.6.2 圆弧加载进度条

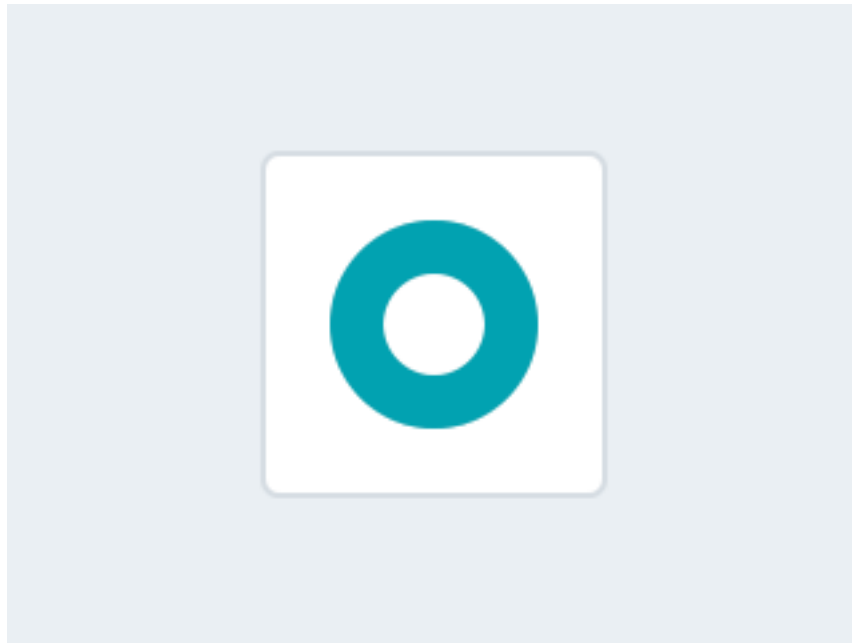


图 2: 圆弧加载进度条

上述效果的示例代码:

```
1  #include "../../lv_examples.h"  
2  #if LV_USE_ARC  
3  
4  /**  
5   * An `lv_task` to call periodically to set the angles of the arc  
6   * @param t  
7   */  
8  static void arc_loader(lv_task_t * t)  
9  {  
10     static int16_t a = 270;  
11  
12     a+=5;  
13  
14     lv_arc_set_end_angle(t->user_data, a);  
15 }
```

(下页继续)

(续上页)

```

16         if(a >= 270 + 360) {
17             lv_task_del(t);
18             return;
19         }
20     }
21
22     /**
23      * Create an arc which acts as a loader.
24      */
25     void lv_ex_arc_2(void)
26     {
27         /*Create an Arc*/
28         lv_obj_t * arc = lv_arc_create(lv_scr_act(), NULL);
29         lv_arc_set_bg_angles(arc, 0, 360);
30         lv_arc_set_angles(arc, 270, 270);
31         lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0);
32
33         /* Create an `lv_task` to update the arc.
34          * Store the `arc` in the user data*/
35         lv_task_create(arc_loader, 20, LV_TASK_PRIO_LOWEST, arc);
36     }
37
38     #endif

```

23.7 相关 API

23.7.1 Typedefs

```

1     typedef uint8_t lv_arc_type_t;
2     typedef uint8_t lv_arc_part_t;
3
4     /* 弧的数据 */
5     typedef struct {
6         /* 此类型的新数据 */
7         uint16_t rotation_angle; /* 旋转角度 */
8         uint16_t arc_angle_start; /* 开始角度 */
9         uint16_t arc_angle_end; /* 结束角度 */
10        uint16_t bg_angle_start; /* 背景开始角度 */
11        uint16_t bg_angle_end; /* 背景结束角度 */
12        lv_style_list_t style_arc; /* 样式 */
13        lv_style_list_t style_knob; /* 旋钮样式 */

```

(下页继续)

(续上页)

```

14
15     int16_t cur_value; /* 弧当前值 */
16     int16_t min_value; /* 弧的最小值 */
17     int16_t max_value; /* 弧的最大值 */
18     uint16_t dragging : 1;
19     uint16_t type : 2;
20     uint16_t adjustable : 1;
21     uint16_t min_close : 1; /* 1: 最后一个压角更接近最小端 */
22     uint16_t chg_rate; /* 阻力角圆弧变化率 (度/秒) */
23     uint32_t last_tick; /* 弧的最后拖拽事件时间戳 */
24     int16_t last_angle; /* 弧的最后拖拽角 */
25 } lv_arc_ext_t;

```

23.7.2 enums

```

1     enum {
2         LV_ARC_TYPE_NORMAL, /* 正常类型 */
3         LV_ARC_TYPE_SYMMETRIC, /* 对称类型 */
4         LV_ARC_TYPE_REVERSE /* 逆向类型 */
5     };
6
7
8     /* 弧的部分 */
9     enum {
10         LV_ARC_PART_BG = LV_OBJ_PART_MAIN,
11         LV_ARC_PART_INDIC,
12         LV_ARC_PART_KNOB,
13         _LV_ARC_PART_VIRTUAL_LAST,
14         _LV_ARC_PART_REAL_LAST = _LV_OBJ_PART_REAL_LAST,
15     };

```

23.7.3 函数

```

1     lv_obj_t * lv_arc_create(lv_obj_t * par, const lv_obj_t * copy)
2     功能：创建弧对象
3     返回：指向创建的弧的指针
4     形参：
5     par：指向对象的指针，它将是新弧的父对象
6     copy：指向弧对象的指针，如果不为 NULL，则将从其复制新对象
7

```

(下页继续)

(续上页)

```
void lv_arc_set_start_angle(lv_obj_t * arc, uint16_t start)
```

功能：设置圆弧的起始角度。0 度：右，90 底等。

形参：

arc：指向弧对象的指针

start：起始角度

```
void lv_arc_set_end_angle(lv_obj_t * arc, uint16_t end )
```

功能：设置圆弧的起始角度。0 度：右，90 底等。

形参：

arc：指向弧对象的指针

end：结束角度

```
void lv_arc_set_angles(lv_obj_t * arc, uint16_t start, uint16_t end)
```

功能：设置开始和结束角度

形参：

arc：指向弧对象的指针

start：起始角度

end：结束角度

```
void lv_arc_set_bg_start_angle(lv_obj_t * arc, uint16_t start)
```

功能：设置弧形背景的起始角度。0 度：右，90 底等。

形参：

arc：指向弧对象的指针

start：起始角度

```
void lv_arc_set_bg_end_angle(lv_obj_t * arc, uint16_t end)
```

功能：设置弧形背景的起始角度。0 度：右，90 底等。

形参：

arc：指向弧对象的指针

end：结束角度

```
void lv_arc_set_bg_angles(lv_obj_t * arc, uint16_t start, uint16_t end)
```

功能：设置弧形背景的开始和结束角度

形参：

arc：指向弧对象的指针

start：起始角度

end：结束角度

```
void lv_arc_set_rotation(lv_obj_t * arc, uint16_t rotation_angle)
```

功能：设置整个圆弧的旋转

形参：

arc：指向弧对象的指针

rotation_angle：旋转角度

(下页继续)

(续上页)

```
void lv_arc_set_type(lv_obj_t * arc, lv_arc_type_t type)
```

功能：设置圆弧的类型。

形参：

arc: 指向弧对象的指针

type: 圆弧型

```
void lv_arc_set_value(lv_obj_t * arc, int16_t 值)
```

功能：在圆弧上设置一个新值

形参：

arc: 指向弧对象的指针

value: 新价值

```
void lv_arc_set_range(lv_obj_t * arc, int16_t min, int16_t max)
```

功能：设置圆弧的最小值和最大值

形参：

arc: 指向弧对象的指针

min: 最小值

max: 最大值

```
void lv_arc_set_chg_rate(lv_obj_t * arc, uint16_t threshold)
```

功能：设置圆弧旋钮增量位置的阈值。

形参

arc: 指向弧对象的指针

threshold: 增量阈值

```
void lv_arc_set_adjustable(lv_obj_t * arc, bool adjustable)
```

功能：设置圆弧是否可调。

形参：

arc: 指向弧对象的指针

adjustable: 圆弧是否具有可以拖动的旋钮

```
uint16_t lv_arc_get_angle_start(lv_obj_t * arc)
```

功能：获取圆弧的起始角度。

返回：起始角度 [0..360]

形参：

arc: 指向弧对象的指针

```
uint16_t lv_arc_get_angle_end(lv_obj_t * arc)
```

功能：获取圆弧的末端角度。

返回：端角 [0..360]

形参：

arc: 指向弧对象的指针

(下页继续)

(续上页)

```
94
95  uint16_t lv_arc_get_bg_angle_start(lv_obj_t * arc)
96  功能：获取弧形背景的起始角度。
97  返回：起始角度 [0..360]
98  形参：
99  arc：指向弧对象的指针
100
101  uint16_t lv_arc_get_bg_angle_end(lv_obj_t * arc)
102  功能：获取弧形背景的终止角度。
103  返回：端角 [0..360]
104  形参：
105  arc：指向弧对象的指针
106
107  lv_arc_type_t lv_arc_get_type(constlv_obj_t *arc)
108  功能：获取圆弧是否为类型。
109  返回：弧形类型
110  形参：
111  arc：指向弧对象的指针
112
113  int16_t lv_arc_get_value(constlv_obj_t *arc)
114  功能：获取圆弧的值
115  返回：弧的值
116  形参
117  arc：指向弧对象的指针
118
119  int16_t lv_arc_get_min_value(constlv_obj_t *arc)
120  功能：获得圆弧的最小值
121  返回：圆弧的最小值
122  形参：
123  arc：指向弧对象的指针
124
125  int16_t lv_arc_get_max_value(constlv_obj_t *arc)
126  功能：获取圆弧的最大值
127  返回：弧的最大值
128  形参：
129  arc：指向弧对象的指针
130
131  bool lv_arc_is_dragged(constlv_obj_t *arc)
132  功能：给出弧线是否被拖动
133  返回:true：拖动进行中,false：未拖动
134  形参：
135  arc：指向弧对象的指针
136
```

(下页继续)

(续上页)

```
137     bool lv_arc_get_adjustable(lv_obj_t * arc)
```

```
138     功能：获取圆弧是否可调。
```

```
139     返回：圆弧是否具有可以拖动的旋钮
```

```
140     形参：
```

```
141     arc：指向弧对象的指针
```


24.1 概述

条对象上有一个背景和一个指示器。指示器的宽度根据条的当前值进行设置。

如果对象的宽度小于其高度，则可以创建垂直条。

不仅可以结束，还可以设置条的起始值，从而改变指示器的起始位置。

24.2 零件和样式

进度条的主要部分称为 LV_BAR_PART_BG，它使用典型的背景样式属性。

LV_BAR_PART_INDIC 是一个虚拟部件，还使用了所有典型的背景属性。默认情况下，指示器的最大尺寸与背景的尺寸相同，但是在其中设置正的填充值 LV_BAR_PART_BG 将使指示器变小。（负值会使它变大）如果在指标上使用了值样式属性，则将根据指标的当前大小来计算对齐方式。例如，中心对齐的值始终显示在指示器的中间，而不管其当前大小如何。

24.3 用法

24.3.1 值和范围

可以通过 `lv_bar_set_value(bar, new_value, LV_ANIM_ON/OFF)` 设置新值。该值以一个范围（最小值和最大值）解释，可以使用 `lv_bar_set_range(bar, min, max)` 进行修改。默认范围是 1..100。

`lv_bar_set_value` 中的新值可以根据最后一个参数（`LV_ANIM_ON/OFF`）设置是否带有动画。动画的时间可以通过 `lv_bar_set_anim_time(bar, 100)` 进行调整。时间以毫秒为单位。

也可以使用 `lv_bar_set_start_value(bar, new_value, LV_ANIM_ON/OFF)` 设置进度条的起始值

24.3.2 模式

如果已通过 `lv_bar_set_type(bar, LV_BAR_TYPE_SYMMETRICAL)` 启用，则条形可以对称地绘制为零（从零开始，从左至右绘制）。

24.4 事件

仅 [通用事件](#) 是按对象类型发送的。

了解有关‘事件’的更多信息。

24.5 按键

对象类型不处理任何输入按键。

进一步了解 [按键](#)。

24.6 范例

24.6.1 简单的进度条

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2  #if LV_USE_BAR
3
4  void lv_ex_bar_1(void)
5  {
```

(下页继续)



图 1: 进度条简单演示

(续上页)

```
6      lv_obj_t * bar1 = lv_bar_create(lv_scr_act(), NULL);
7      lv_obj_set_size(bar1, 200, 20);
8      lv_obj_align(bar1, NULL, LV_ALIGN_CENTER, 0, 0);
9      lv_bar_set_anim_time(bar1, 2000);
10     lv_bar_set_value(bar1, 100, LV_ANIM_ON);
11 }
12
13 #endif
```

24.7 相关 API

TODO

25.1 概述

按钮是简单的矩形对象。它们源自容器，因此也可以提供布局 and 配合。此外，可以启用它以在单击时自动进入检查状态。

25.2 零件和样式

这些按钮仅具有一种主要样式，称为 `LV_BTN_PART_MAIN`，并且可以使用以下组中的所有属性：

- 背景 (background)
- 边界 (border)
- 边框 (outline)
- 阴影 (shadow)
- 数值 (value)
- 模式 (pattern)
- 过渡 (transitions)

启用布局或适合时，它还将使用 `padding` 属性。

25.3 用法

为了简化按钮的使用，可以使用 `lv_btn_get_state(btn)` 来获取按钮的状态。它返回以下值之一：

- **LV_BTN_STATE_RELEASED** 松开
- **LV_BTN_STATE_PRESSED** 被点击
- **LV_BTN_STATE_CHECKED_RELEASED** 点击后松开
- **LV_BTN_STATE_CHECKED_PRESSED** 重复点击
- **LV_BTN_STATE_DISABLED** 禁用
- **LV_BTN_STATE_CHECKED_DISABLED**

使用 `lv_btn_set_state(btn, LV_BTN_STATE_...)` 可以手动更改按钮状态。

如果需要状态的更精确描述（例如，重点突出），则可以使用常规 `lv_obj_get_state(btn)`。

25.4 可检查

可以使用 `lv_btn_set_checkable(btn, true)` 将按钮配置为切换按钮。在这种情况下，单击时，按钮将自动进入 `LV_STATE_CHECKED` 状态，或再次单击时返回到 `LV_STATE_CHECKED` 状态。

25.5 布局和适配

与容器类似，按钮也具有布局和适合属性。

- `lv_btn_set_layout(btn, LV_LAYOUT_...)` 设置布局。默认值为 `LV_LAYOUT_CENTER`。因此，如果添加标签，则标签将自动与中间对齐，并且无法通过 `lv_obj_set_pos()` 移动。您可以使用 `lv_btn_set_layout(btn, LV_LAYOUT_OFF)` 禁用布局。
- `lv_btn_set_fit/fit2/fit4(btn, LV_FIT_...)` 允许根据子代，父代和适合类型自动设置按钮的宽度和/或高度。

25.6 事件

除了 通用事件 外，按钮还发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED**-切换按钮时发送。

了解有关 事件 的更多信息。

25.7 按键

以下按键由按钮处理：

- **LV_KEY_RIGHT/UP**-如果启用了切换，则进入切换状态。
- **LV_KEY_LEFT/DOWN**-如果启用了切换，则进入非切换状态。

请注意，LV_KEY_ENTER 的状态已转换为 LV_EVENT_PRESSED/PRESSING/RELEASED 等。

进一步了解 按键。

25.8 范例

25.8.1 简单的按钮

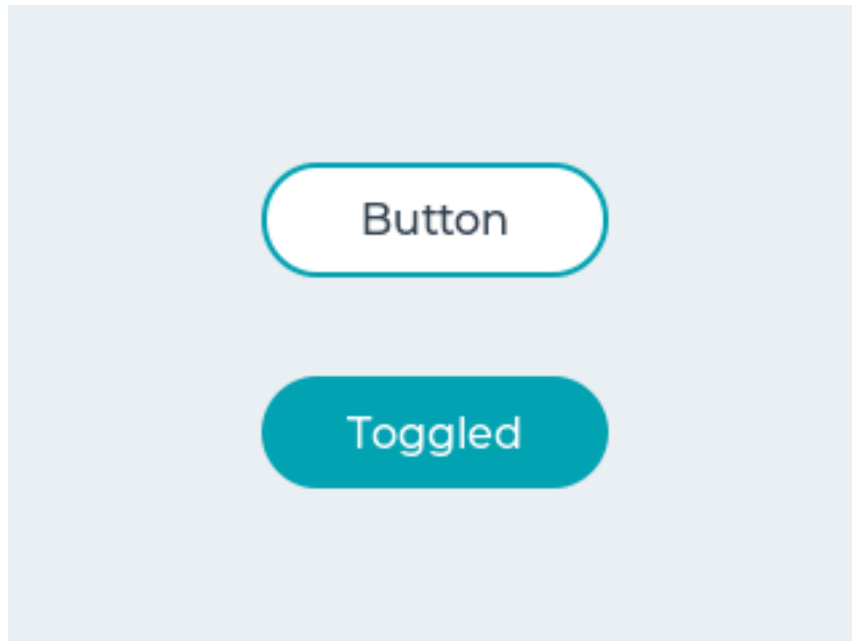


图 1: 创建两个简单的按钮

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_BTN
4
5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
```

(下页继续)

(续上页)

```
7     if(event == LV_EVENT_CLICKED) {
8         printf("Clicked\n");
9     }
10    else if(event == LV_EVENT_VALUE_CHANGED) {
11        printf("Toggled\n");
12    }
13}
14
15void lv_ex_btn_1(void)
16{
17    lv_obj_t * label;
18
19    lv_obj_t * btn1 = lv_btn_create(lv_scr_act(), NULL);
20    lv_obj_set_event_cb(btn1, event_handler);
21    lv_obj_align(btn1, NULL, LV_ALIGN_CENTER, 0, -40);
22
23    label = lv_label_create(btn1, NULL);
24    lv_label_set_text(label, "Button");
25
26    lv_obj_t * btn2 = lv_btn_create(lv_scr_act(), NULL);
27    lv_obj_set_event_cb(btn2, event_handler);
28    lv_obj_align(btn2, NULL, LV_ALIGN_CENTER, 0, 40);
29    lv_btn_set_checkable(btn2, true);
30    lv_btn_toggle(btn2);
31    lv_btn_set_fit2(btn2, LV_FIT_NONE, LV_FIT_TIGHT);
32
33    label = lv_label_create(btn2, NULL);
34    lv_label_set_text(label, "Toggled");
35}
36#endif
```

25.8.2 按钮样式

上述效果的示例代码：

```
1  #include ".././../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_BTN
4
5  /**
6   * Advanced button transition examples
7   */
```

(下页继续)

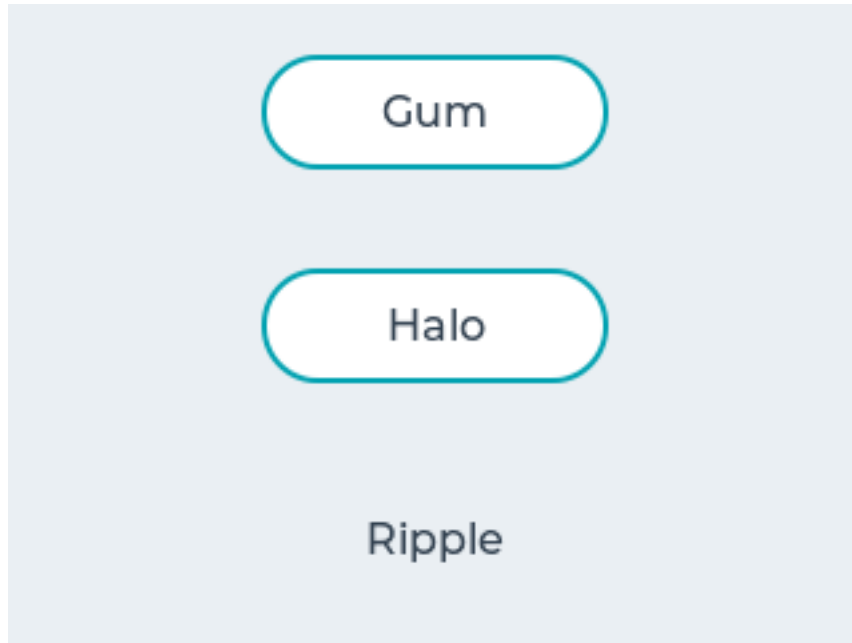


图 2: 按钮样式

(续上页)

```

8  void lv_ex_btn_2(void)
9  {
10     static lv_anim_path_t path_overshoot;
11     lv_anim_path_init(&path_overshoot);
12     lv_anim_path_set_cb(&path_overshoot, lv_anim_path_overshoot);
13
14     static lv_anim_path_t path_ease_out;
15     lv_anim_path_init(&path_ease_out);
16     lv_anim_path_set_cb(&path_ease_out, lv_anim_path_ease_out);
17
18     static lv_anim_path_t path_ease_in_out;
19     lv_anim_path_init(&path_ease_in_out);
20     lv_anim_path_set_cb(&path_ease_in_out, lv_anim_path_ease_in_out);
21
22     /*Gum-like button*/
23     static lv_style_t style_gum;
24     lv_style_init(&style_gum);
25     lv_style_set_transform_width(&style_gum, LV_STATE_PRESSED, 10);
26     lv_style_set_transform_height(&style_gum, LV_STATE_PRESSED, -10);
27     lv_style_set_value_letter_space(&style_gum, LV_STATE_PRESSED, 5);
28     lv_style_set_transition_path(&style_gum, LV_STATE_DEFAULT, &path_
↪overshoot);
29     lv_style_set_transition_path(&style_gum, LV_STATE_PRESSED, &path_ease_in_
↪out);

```

(下页继续)

(续上页)

```

30         lv_style_set_transition_time(&style_gum, LV_STATE_DEFAULT, 250);
31         lv_style_set_transition_delay(&style_gum, LV_STATE_DEFAULT, 100);
32         lv_style_set_transition_prop_1(&style_gum, LV_STATE_DEFAULT, LV_STYLE_
↪ TRANSFORM_WIDTH);
33         lv_style_set_transition_prop_2(&style_gum, LV_STATE_DEFAULT, LV_STYLE_
↪ TRANSFORM_HEIGHT);
34         lv_style_set_transition_prop_3(&style_gum, LV_STATE_DEFAULT, LV_STYLE_
↪ VALUE_LETTER_SPACE);
35
36         lv_obj_t * btn1 = lv_btn_create(lv_scr_act(), NULL);
37         lv_obj_align(btn1, NULL, LV_ALIGN_CENTER, 0, -80);
38         lv_obj_add_style(btn1, LV_BTN_PART_MAIN, &style_gum);
39
40         /*Instead of creating a label add a values string*/
41         lv_obj_set_style_local_value_str(btn1, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
↪ "Gum");
42
43         /*Halo on press*/
44         static lv_style_t style_halo;
45         lv_style_init(&style_halo);
46         lv_style_set_transition_time(&style_halo, LV_STATE_PRESSED, 400);
47         lv_style_set_transition_time(&style_halo, LV_STATE_DEFAULT, 0);
48         lv_style_set_transition_delay(&style_halo, LV_STATE_DEFAULT, 200);
49         lv_style_set_outline_width(&style_halo, LV_STATE_DEFAULT, 0);
50         lv_style_set_outline_width(&style_halo, LV_STATE_PRESSED, 20);
51         lv_style_set_outline_opa(&style_halo, LV_STATE_DEFAULT, LV_OPA_COVER);
52         lv_style_set_outline_opa(&style_halo, LV_STATE_FOCUSED, LV_OPA_COVER); //
↪ *Just to be sure, the theme might use it*/
53         lv_style_set_outline_opa(&style_halo, LV_STATE_PRESSED, LV_OPA_TRANSP);
54         lv_style_set_transition_prop_1(&style_halo, LV_STATE_DEFAULT, LV_STYLE_
↪ OUTLINE_OPA);
55         lv_style_set_transition_prop_2(&style_halo, LV_STATE_DEFAULT, LV_STYLE_
↪ OUTLINE_WIDTH);
56
57         lv_obj_t * btn2 = lv_btn_create(lv_scr_act(), NULL);
58         lv_obj_align(btn2, NULL, LV_ALIGN_CENTER, 0, 0);
59         lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_halo);
60         lv_obj_set_style_local_value_str(btn2, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
↪ "Halo");
61
62         /*Ripple on press*/
63         static lv_style_t style_ripple;
64         lv_style_init(&style_ripple);

```

(下页继续)

(续上页)

```

65     lv_style_set_transition_time(&style_ripple, LV_STATE_PRESSED, 300);
66     lv_style_set_transition_time(&style_ripple, LV_STATE_DEFAULT, 0);
67     lv_style_set_transition_delay(&style_ripple, LV_STATE_DEFAULT, 300);
68     lv_style_set_bg_opa(&style_ripple, LV_STATE_DEFAULT, 0);
69     lv_style_set_bg_opa(&style_ripple, LV_STATE_PRESSED, LV_OPA_80);
70     lv_style_set_border_width(&style_ripple, LV_STATE_DEFAULT, 0);
71     lv_style_set_outline_width(&style_ripple, LV_STATE_DEFAULT, 0);
72     lv_style_set_transform_width(&style_ripple, LV_STATE_DEFAULT, -20);
73     lv_style_set_transform_height(&style_ripple, LV_STATE_DEFAULT, -20);
74     lv_style_set_transform_width(&style_ripple, LV_STATE_PRESSED, 0);
75     lv_style_set_transform_height(&style_ripple, LV_STATE_PRESSED, 0);
76
77     lv_style_set_transition_path(&style_ripple, LV_STATE_DEFAULT, &path_ease_
↪ out);
78     lv_style_set_transition_prop_1(&style_ripple, LV_STATE_DEFAULT, LV_STYLE_
↪ BG_OPA);
79     lv_style_set_transition_prop_2(&style_ripple, LV_STATE_DEFAULT, LV_STYLE_
↪ TRANSFORM_WIDTH);
80     lv_style_set_transition_prop_3(&style_ripple, LV_STATE_DEFAULT, LV_STYLE_
↪ TRANSFORM_HEIGHT);
81
82     lv_obj_t * btn3 = lv_btn_create(lv_scr_act(), NULL);
83     lv_obj_align(btn3, NULL, LV_ALIGN_CENTER, 0, 80);
84     lv_obj_add_style(btn3, LV_BTN_PART_MAIN, &style_ripple);
85     lv_obj_set_style_local_value_str(btn3, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
↪ "Ripple");
86 }
87 #endif

```

25.9 相关 API

25.9.1 Typedefs

```

1     typedef uint8_t lv_btn_state_t
2     typedef uint8_t lv_btn_part_t

```

25.9.2 enums

```

1  /** Possible states of a button.
2   * It can be used not only by buttons but other button-like objects too*/
3  enum {
4      LV_BTN_STATE_RELEASED,
5      LV_BTN_STATE_PRESSED,
6      LV_BTN_STATE_DISABLED,
7      LV_BTN_STATE_CHECKED_RELEASED,
8      LV_BTN_STATE_CHECKED_PRESSED,
9      LV_BTN_STATE_CHECKED_DISABLED,
10     _LV_BTN_STATE_LAST, /* Number of states*/
11 };
12
13 /**Styles*/
14 enum {
15     LV_BTN_PART_MAIN = LV_OBJ_PART_MAIN,
16     _LV_BTN_PART_VIRTUAL_LAST,
17     _LV_BTN_PART_REAL_LAST = _LV_OBJ_PART_REAL_LAST,
18 };

```

25.9.3 函数

```

1  lv_obj_t * lv_btn_create(lv_obj_t * par, const lv_obj_t * copy)
2  功能：创建一个按钮对象
3  返回：
4  指向创建的按钮的指针
5  形参：
6  par：指向对象的指针，它将是新按钮的父对象
7  copy：指向按钮对象的指针，如果不为 NULL，则将从其复制新对象
8
9
10 void lv_btn_set_checkable(lv_obj_t * btn, bool tgl )
11 启用切换状态。释放时，按钮将从/切换到切换状态。
12 形参：
13 btn：指向按钮对象的指针
14 tgl：true：启用切换状态，false：禁用
15
16
17 void lv_btn_set_state(lv_obj_t * btn, lv_btn_state_t state)
18 功能：设置按钮的状态
19 形参：
20 btn：指向按钮对象的指针

```

(下页继续)

(续上页)

```

21     state: 按钮的新状态 (来自 lv_btn_state_t enum)
22
23
24     void lv_btn_toggle(lv_obj_t * btn )
25     功能: 切换按钮的状态 (ON-> OFF, OFF-> ON)
26     形参:
27     btn: 指向按钮对象的指针
28
29
30     void lv_btn_set_layout(lv_obj_t * btn, lv_layout_t layout)
31     功能: 在按钮上设置布局
32     形参:
33     btn: 指向按钮对象的指针
34     layout: 来自 “ lv_cont_layout_t” 的布局
35
36
37     void lv_btn_set_fit4(lv_obj_t * btn, lv_fit_t left, lv_fit_t right, lv_fit_t top,
lv_fit_t bottom)
38     功能: 分别在所有四个方向上设置适合策略。它告诉您如何自动更改按钮大小。
39     形参:
40     btn: 指向按钮对象的指针
41     left: 从左适合策略 lv_fit_t
42     right: 合适的策略来自 lv_fit_t
43     top: 最适合的策略, 来自 lv_fit_t
44     bottom: 自下而上的策略 lv_fit_t
45
46
47     void lv_btn_set_fit2(lv_obj_t * btn, lv_fit_t hor, lv_fit_t ver )
48     功能: 分别水平和垂直设置适合策略。它告诉您如何自动更改按钮大小。
49     形参:
50     btn: 指向按钮对象的指针
51     hor: 来自的水平拟合策略 lv_fit_t
52     ver: 垂直适合策略, 来自 lv_fit_t
53
54
55     void lv_btn_set_fit(lv_obj_t * btn, lv_fit_t fit )
56     功能: 一次在所有 4 个方向上设置拟合策略。它告诉您如何自动更改按钮的大小。
57     形参:
58     btn: 指向按钮对象的指针
59     fit: 符合策略 lv_fit_t
60
61
62     lv_btn_state_t lv_btn_get_state(const lv_obj_t * btn )

```

(下页继续)

(续上页)

63 功能：获取按钮的当前状态
64 返回：
65 按钮的状态（来自 `lv_btn_state_t` 枚举）如果按钮处于禁用状态，`LV_BTN_STATE_DISABLED` 则将其与其他按钮状态进行“或”运算。

66 形参：
67 `btn`：指向按钮对象的指针
68
69

70 `bool lv_btn_get_checkable(const lv_obj_t * btn)`

71 功能：获取按钮的切换启用属性
72 返回：
73 `true`：启用检查，`false`：停用
74 形参：

75 `btn`：指向按钮对象的指针
76
77

78 `lv_layout_t lv_btn_get_layout(const lv_obj_t * btn)`

79 功能：获取按钮的布局
80 返回：
81 来自“`lv_cont_layout_t`”的布局
82 形参：

83 `btn`：指向按钮对象的指针
84
85

86 `lv_fit_t lv_btn_get_fit_left(const lv_obj_t * btn)`

87 功能：获取合适的左模式
88 返回：
89 左部元素 `lv_fit_t`
90 形参：

91 `btn`：指向按钮对象的指针
92
93

94 `lv_fit_t lv_btn_get_fit_right(const lv_obj_t * btn)`

95 功能：获得合适的右模式
96 返回：
97 右部元素 `lv_fit_t`
98 形参：

99 `btn`：指向按钮对象的指针
100
101

102 `lv_fit_t lv_btn_get_fit_top(const lv_obj_t * btn)`

103 功能：获取最适合顶部模式
104 返回：

(下页继续)

(续上页)

```
105  顶部元素 lv_fit_t
106  形参:
107  btn: 指向按钮对象的指针
108
109
110  lv_fit_t lv_btn_get_fit_bottom(const lv_obj_t * btn )
111  功能: 获取最合适底部模式
112  返回:
113  底部元素 lv_fit_t
114  形参:
115  btn: 指向按钮对象的指针
```

按钮矩阵 (lv_btnmatrix)

26.1 概述

Button Matrix 对象可以在行和列中显示多个按钮。

想要使用按钮矩阵而不是容器和单个按钮对象的主要原因是：

- 对于基于网格的按钮布局，按钮矩阵更易于使用。
- 每个按钮矩阵消耗的内存少得多。

26.2 零件和样式

Button 矩阵的主要部分称为 LV_BTNMATRIX_PART_BG。它使用典型的背景样式属性绘制背景。

LV_BTNMATRIX_PART_BTN 是虚拟部件，它引用按钮矩阵上的按钮。它还使用所有典型的背景属性。

背景中的顶部/底部/左侧/右侧填充值用于在侧面保留一些空间。在按钮之间应用内部填充。

26.3 用法

26.3.1 按钮的文字

每个按钮上都有一个文本。要指定它们，需要使用称为 `map` 的描述符字符串数组。可以使用 `lv_btnmatrix_set_map(btnm, my_map)` 设置地图。映射的声明应类似于 `const char * map[] = {"btn1", "btn2", "btn3", ""}`。请注意，最后一个元素必须为空字符串！

在按钮矩阵中使用 `"\n"` 进行换行。例如。{"btn1", "btn2", "\n", "btn3", ""}。每行按钮的宽度自动计算。

26.3.2 控制按钮

可以使用 `lv_btnmatrix_set_btn_width(btnm, btn_id, width)` 相对于同一行中的另一个按钮设置按钮宽度。例如。在带有两个按钮的行中：btnA, width = 1 和 btnB, width = 2, btnA 将具有 33% 的宽度, btnB 将具有 66% 的宽度。这类似于 flex-grow 属性在 CSS 中的工作方式。

除了宽度，每个按钮都可以使用以下参数进行自定义：

- **LV_BTNMATRIX_CTRL_HIDDEN** - 将按钮设为隐藏（隐藏的按钮仍会占用布局中的空间，它们只是不可见或不可单击）
- **LV_BTNMATRIX_CTRL_NO_REPEAT** - 长按按钮时禁用重复
- **LV_BTNMATRIX_CTRL_DISABLED** - 禁用按钮
- **LV_BTNMATRIX_CTRL_CHECKABLE** - 启用按钮切换
- **LV_BTNMATRIX_CTRL_CHECK_STATE** - 设置切换状态
- **LV_BTNMATRIX_CTRL_CLICK_TRIG** - 如果为 0，则按钮将在按下时作出反应；如果为 1，则将在释放时作出反应

设置或清除按钮的控件属性，分别使用 `lv_btnmatrix_set_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)` 和 `lv_btnmatrix_clear_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)`。可以对更多 `LV_BTNM_CTRL_...` 值进行排序

为按钮矩阵的所有按钮设置/清除相同的控件属性，使用 `lv_btnmatrix_set_btn_ctrl_all(btnm, btn_id, LV_BTNM_CTRL_...)` 和 `lv_btnmatrix_clear_btn_ctrl_all(btnm, btn_id, LV_BTNM_CTRL_...)`。

设置按钮矩阵的控制映射（类似于文本映射），请使用 `lv_btnmatrix_set_ctrl_map(btnm, ctrl_map)`。ctrl_map 的元素应类似于 `ctrl_map[0] = width | LV_BTNM_CTRL_NO_REPEAT | LV_BTNM_CTRL_TGL_ENABLE`。元素的数量应等于按钮的数量（不包括换行符）。

26.3.3 一次检查

可以通过 `lv_btnmatrix_set_one_check(btnm, true)` 启用“一次检查”功能，以仅一次检查（切换）一个按钮。

26.3.4 重新着色

可以对标签上的文本重新着色，类似于 `Label` 对象的重新着色功能。要启用它，请使用 `lv_btnmatrix_set_recolor(btnm, true)`。之后，带有 `#FF0000` 红色 `#` 文本的按钮将变为红色。

26.3.5 对齐按钮的文字

要对齐按钮上的文本，请使用 `lv_btnmatrix_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`。按钮矩阵中的所有文本项都将符合设置后的对齐属性。

26.3.6 注意事项

`Button` 矩阵对象的权重非常轻，因为按钮不是在虚拟飞行中绘制的。这样，一个按钮仅使用 8 个额外的字节，而不是普通 `Button` 对象的 100-150 字节大小（加上其容器的大小和每个按钮的标签）。

此设置的缺点是，将各个按钮的样式设置为与其他按钮不同的功能受到限制（除了切换功能之外）。如果您需要该功能，则使用单个按钮很有可能是一种更好的方法。

26.4 事件

除了通用事件之外，按钮矩阵还发送以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** - 在按下/释放按钮时或在长按之后重复时发送。事件数据设置为按下/释放按钮的 ID。

了解有关事件的更多信息。

26.5 按钮

以下按键由按钮处理：

- **LV_KEY_RIGHT/UP/LEFT/RIGHT** - 在按钮之间导航以选择一个
- **LV_KEY_ENTER** - 按下/释放所选按钮

进一步了解 [按键](#)。

26.6 范例

26.6.1 简单按钮矩阵



图 1: 简单按钮矩阵示例

上述效果的示例代码:

```

1  #include "../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_BTNMATRIX
4
5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
7      if(event == LV_EVENT_VALUE_CHANGED) {
8          const char * txt = lv_btnmatrix_get_active_btn_text(obj);
9
10         printf("%s was pressed\n", txt);
11     }
12 }
13
14
15 static const char * btm_map[] = {"1", "2", "3", "4", "5", "\n",
16                                   "6", "7", "8",
17                                   "9", "0", "\n",
                                   "Action1",
                                   "Action2", ""};

```

(下页继续)

(续上页)

```

18
19 void lv_ex_btnmatrix_1(void)
20 {
21     lv_obj_t * btnm1 = lv_btnmatrix_create(lv_scr_act(), NULL);
22     lv_btnmatrix_set_map(btnm1, btnm_map);
23     lv_btnmatrix_set_btn_width(btnm1, 10, 2);           /*Make "Action1" twice_
↪as wide as "Action2"*/
24     lv_btnmatrix_set_btn_ctrl(btnm1, 10, LV_BTNMATRIX_CTRL_CHECKABLE);
25     lv_btnmatrix_set_btn_ctrl(btnm1, 11, LV_BTNMATRIX_CTRL_CHECK_STATE);
26     lv_obj_align(btnm1, NULL, LV_ALIGN_CENTER, 0, 0);
27     lv_obj_set_event_cb(btnm1, event_handler);
28 }
29
30 #endif

```

26.7 相关 API

26.7.1 Typedefs

```

1 typedef uint16_t lv_btnmatrix_ctrl_t
2 typedef uint8_t lv_btnmatrix_part_t

```

26.7.2 enums

```

1 enum [anonymous ]
2 键入以存储按钮控制位（禁用，隐藏等）。前 3 位用于存储宽度
3 值：
4
5     enumerator LV_BTNMATRIX_CTRL_HIDDEN      /* 隐藏按钮 */
6
7
8     enumerator LV_BTNMATRIX_CTRL_NO_REPEAT  /* 不要重复按此按钮。*/
9
10
11     enumerator LV_BTNMATRIX_CTRL_DISABLED  /* 禁用此按钮。*/
12
13
14     enumerator LV_BTNMATRIX_CTRL_CHECKABLE /* 可以切换按钮。*/
15

```

(下页继续)

(续上页)

```

16
17     enumerator LV_BTNMATRIX_CTRL_CHECK_STATE/* 当前已切换按钮（例如，选中）。*/
18
19     /* 1: 在 CLICK 上发送 LV_EVENT_SELECTED, 0: 在 PRESS 上发送 LV_EVENT_SELECTED */
20     enumerator LV_BTNMATRIX_CTRL_CLICK_TRIG
21
22
23     enum [anonymous ]
24     值:
25     enumerator LV_BTNMATRIX_PART_BG
26     enumerator LV_BTNMATRIX_PART_BTN

```

26.7.3 函数

```

1     LV_EXPORT_CONST_INT( LV_BTNMATRIX_BTN_NONE )
2     lv_obj_t * lv_btnmatrix_create(lv_obj_t * par, constlv_obj_t *copy)
3     功能: 创建一个按钮矩阵对象
4     返回:
5     指向创建的按钮矩阵的指针
6     形参:
7     par: 指向对象的指针, 它将是新按钮矩阵的父对象
8     copy: 指向按钮矩阵对象的指针, 如果不为 NULL, 则将从其复制新对象
9
10
11     void lv_btnmatrix_set_map(lv_obj_t * btnm, constchar *map[] )
12     功能: 设置新地图。将根据地图创建/删除按钮。按钮矩阵保留对地图的引用, 因此在矩阵有效期内不得释放
13     ↪char 串数组。
14     形参:
15     btnm: 指向按钮矩阵对象的指针
16     map: 指针的 char 串数组。最后一个 char 串必须是: ""。使用 " \ n" 进行换行。
17
18     void lv_btnmatrix_set_ctrl_map(lv_obj_t * btnm, constlv_btnmatrix_ctrl_t ctrl_map_
19     ↪[] )
20     功能: 设置按钮矩阵的按钮控制图（隐藏, 禁用等）。控制图数组将被复制, 因此在此函数返回: 后可以将其释放。
21     形参:
22     btnm: 指向按钮矩阵对象的指针
23     ctrl_map: 指向 lv_btn_ctrl_t 控制字节数组的指针。数组的长度和元素的位置必须与单个按钮的数量和顺
24     序匹配（即, 不包括换行符）。地图的元素应类似于: ctrl_map[0] = width | LV_BTNMATRIX_CTRL_NO_
25     ↪REPEAT | LV_BTNMATRIX_CTRL_TGL_ENABLE

```

(下页继续)

(续上页)

```
void lv_btnmatrix_set_focused_btn(lv_obj_t * btnm, uint16_t id )
```

功能：设置焦点按钮，即在视觉上突出显示它。

形参：

btnm: 指向按钮矩阵对象的指针

id: 要聚焦的按钮的索引 (LV_BTNMATRIX_BTN_NONE 以除去焦点)

```
void lv_btnmatrix_set_recolor(constlv_obj_t * btnm, bool en )
```

功能：启用按钮文本的重新着色

形参：

btnm: 指向按钮矩阵对象的指针

en: true: 启用重新着色; false: 禁用

```
void lv_btnmatrix_set_btn_ctrl(lv_obj_t * btnm, uint16_t btn_id, lv_btnmatrix_ctrl_t ctrl )
```

功能：设置按钮矩阵的按钮属性

形参：

btnm: 指向按钮矩阵对象的指针

btn_id: 基于 0 的按钮索引进行修改。(不计算新行)

```
void lv_btnmatrix_clear_btn_ctrl(constlv_obj_t * btnm, uint16_t btn_id, lv_btnmatrix_ctrl_t ctrl )
```

功能：清除按钮矩阵的按钮属性

形参：

btnm: 指向按钮矩阵对象的指针

btn_id: 基于 0 的按钮索引进行修改。(不计算新行)

```
void lv_btnmatrix_set_btn_ctrl_all(lv_obj_t * btnm, lv_btnmatrix_ctrl_t ctrl )
```

功能：设置按钮矩阵中所有按钮的属性

形参：

btnm: 指向按钮矩阵对象的指针

ctrl: 要从中设置的属性 lv_btnmatrix_ctrl_t。值可以进行或运算。

```
void lv_btnmatrix_clear_btn_ctrl_all(lv_obj_t * btnm, lv_btnmatrix_ctrl_t ctrl )
```

功能：清除按钮矩阵中所有按钮的属性

形参：

btnm: 指向按钮矩阵对象的指针

ctrl: 要从中设置的属性 lv_btnmatrix_ctrl_t。值可以进行或运算。

en: true: 设置属性; false: 清除属性

(下页继续)

(续上页)

```
void lv_btnmatrix_set_btn_width(lv_obj_t * btnm, uint16_t btn_id, uint8_t width )
```

功能：设置单个按钮的相对宽度。该方法将导致矩阵再生并且是相对昂贵的操作。建议使用来指定初始宽度，`lv_btnmatrix_set_ctrl_map` 并且此方法仅用于动态更改。

形参：

`btnm`：指向按钮矩阵对象的指针

`btn_id`：基于 0 的按钮索引进行修改。

`width`：相对宽度（与同一行中的按钮相比）。[1..7]

```
void lv_btnmatrix_set_one_check(lv_obj_t * btnm, bool one_chk )
```

功能：使按钮矩阵像选择器小部件一样（一次只能切换一个按钮）。Checkable 必须在使用 `lv_btnmatrix_set_ctrl` 或选择的按钮上启用 `lv_btnmatrix_set_btn_ctrl_all`。

形参：

`btnm`：按钮矩阵对象

`one_chk`：是否启用“一次检查”模式

```
void lv_btnmatrix_set_align(lv_obj_t * btnm, lv_label_align_t align )
```

功能：设置地图文字的对齐方式（向左，向右或居中）

形参：

`btnm`：指向 `btnmatrix` 对象的指针

`align`：LV_LABEL_ALIGN_LEFT, LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER

```
const char ** lv_btnmatrix_get_map_array(const lv_obj_t * btnm )
```

功能：获取按钮矩阵的当前图

返回：

当前地图

形参：

`btnm`：指向按钮矩阵对象的指针

```
bool lv_btnmatrix_get_recolor(const lv_obj_t * btnm )
```

功能：检查按钮的文本是否可以使用重新着色

返回：

`true`：启用文本重新着色；`false`：禁用

形参：

`btnm`：指向按钮矩阵对象的指针

```
uint16_t lv_btnmatrix_get_active_btn(const lv_obj_t * btnm )
```

(下页继续)

(续上页)

功能：获取用户最后按下的按钮的索引（按下，释放等）。在 event_cb 获取按钮的文本，检查是否隐藏等方面很有用。

返回：

最后释放按钮的索引（LV_BTNMATRIX_BTN_NONE：如果未设置）

形参：

btnm：指向按钮矩阵对象的指针

```
const char * lv_btnmatrix_get_active_btn_text(constlv_obj_t * btnm )
```

功能：获取用户最后一次“激活”按钮的文本（按下，释放等） event_cb

返回：

最后释放的按钮的文本（NULL：如果未设置）

形参：

btnm：指向按钮矩阵对象的指针

```
uint16_t lv_btnmatrix_get_focused_btn(constlv_obj_t * btnm )
```

功能：获取焦点按钮的索引。

返回：

焦点按钮的索引（LV_BTNMATRIX_BTN_NONE：如果未设置）

形参：

btnm：指向按钮矩阵对象的指针

```
const char * lv_btnmatrix_get_btn_text(constlv_obj_t * btnm, uint16_t btn_id )
```

功能：获取按钮的文字

返回：

btn_index`按钮的文本

形参：

btnm：指向按钮矩阵对象的指针

btn_id：索引一个不计算换行符的按钮。（lv_btnmatrix_get_pressed / released 的返回：值）

```
bool lv_btnmatrix_get_btn_ctrl(lv_obj_t * btnm, uint16_t btn_id, lv_btnmatrix_ctrl_
↪t ctrl )
```

功能：获取按钮矩阵的按钮的控制值是启用还是禁用

返回：

true：禁用长按重复；false：长按重复启用

形参：

btnm：指向按钮矩阵对象的指针

btn_id：索引一个不计算换行符的按钮。（例如 lv_btnmatrix_get_pressed / released 的返回：值）

ctrl：要检查的控制值（可以使用 ORed 值）

(下页继续)

(续上页)

```
148
149     bool lv_btnmatrix_get_one_check(const lv_obj_t * btnm )
150     功能：查找是否启用了“一次切换”模式。
151     返回：
152     是否启用“一次切换”模式
153     形参：
154     btnm：按钮矩阵对象
155
156
157     lv_label_align_t lv_btnmatrix_get_align(const lv_obj_t * btnm )
158     功能：获取 align 属性
159     返回：
160     LV_LABEL_ALIGN_LEFT, LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER
161     形参：
162     btnm：指向 btnmatrix 对象的指针
```

27.1 概述

Calendar 对象是经典的日历，可以：

- 突出显示当天
- 突出显示任何用户定义的日期
- 显示日期名称
- 单击按钮进入下一个/上一个月
- 突出显示点击的日子

27.2 零件和样式

日历的主要部分称为 LV_CALENDAR_PART_BG 。它使用典型的背景样式属性绘制背景。

除以下虚拟部分外：

- LV_CALENDAR_PART_HEADER 显示当前年和月名称的上部区域。它还具有用于移动下一个/上个月的按钮。它使用典型的背景属性以及填充来调整其大小和边距，以设置距日历顶部的距离和日历下方的日期。
- LV_CALENDAR_PART_DAY_NAMES 在标题下方显示日期名称。它使用文本样式属性填充来与背景（左，右），标题（上）和日期（下）保持一定距离。

- LV_CALENDAR_PART_DATES 显示从 1..28 / 29/30/31 开始的日期数字（取决于当月）。根据本部分中定义的状态来绘制状态的不同“状态”：
 - 正常日期：以 LV_STATE_DEFAULT 样式绘制
 - 按日期范围：以 LV_STATE_PRESSED 样式绘制
 - 今天：以 LV_STATE_FOCUSED 样式绘制
 - 高亮显示的日期：以 LV_STATE_CHECKED 样式绘制

27.3 用法

27.3.1 概述

要在日历中设置和获取日期，使用 `lv_calendar_date_t` 类型，该类型是具有 年，月和 日字段的结构。

27.3.2 当前日期

要设置当前日期（今天），请使用 `lv_calendar_set_today_date(calendar, &today_date)` 函数。

27.3.3 显示日期

要设置显示日期，请使用 `lv_calendar_set_shown_date(calendar, &shown_date)`；

27.3.4 高亮日期

高亮显示的日期列表应存储在由 `lv_calendar_set_highlighted_dates(calendar, &highlighted_dates)` 加载的 `lv_calendar_date_t` 数组中。

仅将保存数组指针，因此数组应为静态或全局变量。

27.3.5 日期名称

可以使用 `lv_calendar_set_day_names(calendar, day_names)` 来调整日期的名称，其中 `day_names` 类似于 `const char * day_names[7] = {"Su", "Mo", ...};`

27.3.6 月份名称

与 `day_names` 相似，可以使用 `lv_calendar_set_month_names(calendar, month_names_array)` 设置月份名称。

27.4 事件

除了 通用事件 外，日历还会发送以下特殊事件：当当前月份更改时，还会发送 **LV_EVENT_VALUE_CHANGED**。

在与输入设备相关的事件中，`lv_calendar_get_pressed_date(calendar)` 指示当前正在按下的日期，如果没有按下任何日期，则返回 `NULL`。

了解有关 事件 的更多信息。

按键 #####s 对象类型不处理任何键。

进一步了解 按键。

27.5 范例

27.5.1 简单日历示例

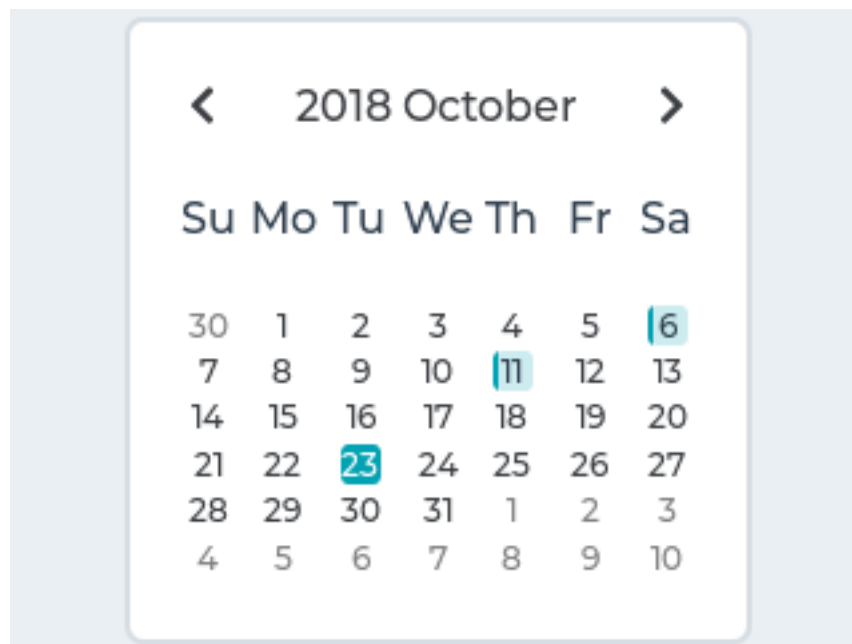


图 1: 可以选择日期的日历

上述效果的示例代码：

```

1  #include "../../lv_examples.h"
2  #include <stdio.h>
3
4  #if LV_USE_CALEDNDAR
5
6  static void event_handler(lv_obj_t * obj, lv_event_t event)
7  {
8      if(event == LV_EVENT_VALUE_CHANGED) {
9          lv_calendar_date_t * date = lv_calendar_get_pressed_date(obj);
10         if(date) {
11             printf("Clicked date: %02d.%02d.%d\n", date->day, date->
↪month, date->year);
12         }
13     }
14 }
15
16 void lv_ex_calendar_1(void)
17 {
18     lv_obj_t * calendar = lv_calendar_create(lv_scr_act(), NULL);
19     lv_obj_set_size(calendar, 235, 235);
20     lv_obj_align(calendar, NULL, LV_ALIGN_CENTER, 0, 0);
21     lv_obj_set_event_cb(calendar, event_handler);
22
23     /*Make the date number smaller to be sure they fit into their area*/
24     lv_obj_set_style_local_text_font(calendar, LV_CALEDNDAR_PART_DATE, LV_
↪STATE_DEFAULT, lv_theme_get_font_small());
25
26     /*Set today's date*/
27     lv_calendar_date_t today;
28     today.year = 2018;
29     today.month = 10;
30     today.day = 23;
31
32     lv_calendar_set_today_date(calendar, &today);
33     lv_calendar_set_showed_date(calendar, &today);
34
35     /*Highlight a few days*/
36     static lv_calendar_date_t highlighted_days[3];          /*Only its pointer
↪will be saved so should be static*/
37     highlighted_days[0].year = 2018;
38     highlighted_days[0].month = 10;
39     highlighted_days[0].day = 6;
40
41     highlighted_days[1].year = 2018;

```

(下页继续)

(续上页)

```
42     highlighted_days[1].month = 10;
43     highlighted_days[1].day = 11;
44
45     highlighted_days[2].year = 2018;
46     highlighted_days[2].month = 11;
47     highlighted_days[2].day = 22;
48
49     lv_calendar_set_highlighted_dates(calendar, highlighted_days, 3);
50 }
51
52 #endif
```

27.6 相关 API

TODO

28.1 概述

画布继承自 [图像](#)，用户可以在其中绘制任何内容。可以使用 `lvgl` 的绘图引擎在此处绘制矩形，文本，图像，线弧。除了一些“效果”，还可以应用，例如旋转，缩放和模糊。

28.2 零件和样式

画布的一个主要部分称为 `LV_CANVAS_PART_MAIN`，只有 `image_recolor` 属性用于为 `LV_IMG_CF_ALPHA_1/2/4/8BIT` 图像赋予颜色。

28.3 用法

28.3.1 缓冲区

画布需要一个缓冲区来存储绘制的图像。要将缓冲区分配给画布，请使用 `lv_canvas_set_buffer(canvas, buffer, width, height, LV_IMG_CF_...)`。其中 `buffer` 是用于保存画布图像的静态缓冲区（而不仅仅是局部变量）。例如，`static lv_color_t buffer[LV_CANVAS_BUF_SIZE_TRUE_COLOR(width, height)]`。`LV_CANVAS_BUF_SIZE_...` 宏有助于确定不同颜色格式的缓冲区大小。

画布支持所有内置的颜色格式，例如 LV_IMG_CF_TRUE_COLOR 或 LV_IMG_CF_INDEXED_2BIT。请参阅 [颜色格式](#) 部分中的完整列表。

28.3.2 调色板

对于 LV_IMG_CF_INDEXED... 颜色格式，需要使用 `lv_canvas_set_palette(canvas, 3, LV_COLOR_RED)` 初始化调色板。它将 `index=3` 的像素设置为红色。

28.3.3 画画

要在画布上设置像素，请使用 `lv_canvas_set_px(canvas, x, y, LV_COLOR_RED)`。对于 LV_IMG_CF_INDEXED... 或 LV_IMG_CF_ALPHA...，需要将颜色的索引或 alpha 值作为颜色传递。例如：`lv_color_t c; c.full = 3;`

`lv_canvas_fill_bg(canvas, LV_COLOR_BLUE, LV_OPA_50)` 将整个画布填充为蓝色，不透明度为 50%。请注意，如果当前的颜色格式不支持颜色（例如 LV_IMG_CF_ALPHA_2BIT），则会忽略颜色。同样，如果不支持不透明度（例如 LV_IMG_CF_TRUE_COLOR），则会将其忽略。

可以使用 `lv_canvas_copy_buf(canvas, buffer_to_copy, x, y, width, height)` 将像素数组复制到画布。缓冲区和画布的颜色格式需要匹配。

画一些东西到画布上使用

- `lv_canvas_draw_rect(canvas, x, y, width, height, &draw_dsc)`
- `lv_canvas_draw_text(canvas, x, y, max_width, &draw_dsc, txt, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`
- `lv_canvas_draw_img(canvas, x, y, &img_src, &draw_dsc)`
- `lv_canvas_draw_line(canvas, point_array, point_cnt, &draw_dsc)`
- `lv_canvas_draw_polygon(canvas, points_array, point_cnt, &draw_dsc)`
- `lv_canvas_draw_arc(canvas, x, y, radius, start_angle, end_angle, &draw_dsc)`

`draw_dsc` 是 `lv_draw_rect/label/img/line_dsc_t` 变量，应首先使用 `lv_draw_rect/label/img/line_dsc_init()` 函数初始化，然后使用所需的颜色和其他值对其进行修改。

绘制功能可以绘制为任何颜色格式。例如，可以在 LV_IMG_VF_ALPHA_8BIT 画布上绘制文本，然后将结果图像用作 `lv_objmask` 中的蒙版。

28.3.4 旋转和缩放

`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。该函数需要以下参数：

- `canvas` 指向画布对象以存储转换结果的指针。
- `img pointer` 转换为图像描述符。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` 缩放系数 (256 不缩放，512 倍大小，128 个一半大小)；
- `offset_x` 偏移量 X，以指示将结果数据放在目标画布上的位置
- `offset_y` 偏移量 Y，以指示将结果数据放在目标画布上的位置
- `pivot_x` 旋转的 X 轴。相对于源画布。设置为 源宽度 / 2 以围绕中心旋转
- `pivot_y` 旋转的枢轴 Y。相对于源画布。设置为 源高度 / 2 以围绕中心旋转
- `antialias true`：在转换过程中应用抗锯齿。看起来更好，但速度较慢。

请注意，画布无法自身旋转。您需要源和目标画布或图像。

28.4 模糊效果

画布的给定区域可以使用 `lv_canvas_blur_hor(canvas, &area, r)` 水平模糊，垂直使用 `lv_canvas_blur_ver(canvas, &area, r)` 模糊。`r` 是模糊的半径（值越大表示毛刺强度越大）。`area` 是应该应用模糊的区域（相对于画布进行解释）

28.5 事件

默认情况下，画布的单击被禁用（由 `Image` 继承），因此不生成任何事件。

如果启用了单击 (`lv_obj_set_click (canvas, true)`)，则仅 [通用事件](#) 由对象类型发送。

了解有关 [事件](#) 的更多信息。

28.6 按键

画布对象类型不处理任何键。

进一步了解 [按键](#)。

28.7 范例

28.7.1 在画布上绘图并旋转



图 1: 在画布上绘图并旋转

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #if LV_USE_CANVAS
3
4
5  #define CANVAS_WIDTH 200
6  #define CANVAS_HEIGHT 150
7
8  void lv_ex_canvas_1(void)
9  {
10     lv_draw_rect_dsc_t rect_dsc;
11     lv_draw_rect_dsc_init(&rect_dsc);
12     rect_dsc.radius = 10;
13     rect_dsc.bg_opa = LV_OPA_COVER;
14     rect_dsc.bg_grad_dir = LV_GRAD_DIR_HOR;
15     rect_dsc.bg_color = LV_COLOR_RED;
16     rect_dsc.bg_grad_color = LV_COLOR_BLUE;
17     rect_dsc.border_width = 2;
18     rect_dsc.border_opa = LV_OPA_90;
```

(下页继续)

(续上页)

```

19     rect_dsc.border_color = LV_COLOR_WHITE;
20     rect_dsc.shadow_width = 5;
21     rect_dsc.shadow_ofs_x = 5;
22     rect_dsc.shadow_ofs_y = 5;
23
24     lv_draw_label_dsc_t label_dsc;
25     lv_draw_label_dsc_init(&label_dsc);
26     label_dsc.color = LV_COLOR_YELLOW;
27
28     static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(CANVAS_WIDTH, CANVAS_
↵HEIGHT)];
29
30     lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);
31     lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH, CANVAS_HEIGHT, LV_IMG_CF_
↵TRUE_COLOR);
32     lv_obj_align(canvas, NULL, LV_ALIGN_CENTER, 0, 0);
33     lv_canvas_fill_bg(canvas, LV_COLOR_SILVER, LV_OPA_COVER);
34
35     lv_canvas_draw_rect(canvas, 70, 60, 100, 70, &rect_dsc);
36
37     lv_canvas_draw_text(canvas, 40, 20, 100, &label_dsc, "Some text on text_
↵canvas", LV_LABEL_ALIGN_LEFT);
38
39     /* Test the rotation. It requires an other buffer where the original image_
↵is stored.
40
41     * So copy the current image to buffer and rotate it to the canvas */
42     static lv_color_t cbuf_tmp[CANVAS_WIDTH * CANVAS_HEIGHT];
43     memcpy(cbuf_tmp, cbuf, sizeof(cbuf_tmp));
44     lv_img_dsc_t img;
45     img.data = (void *)cbuf_tmp;
46     img.header.cf = LV_IMG_CF_TRUE_COLOR;
47     img.header.w = CANVAS_WIDTH;
48     img.header.h = CANVAS_HEIGHT;
49
50     lv_canvas_fill_bg(canvas, LV_COLOR_SILVER, LV_OPA_COVER);
51     lv_canvas_transform(canvas, &img, 30, LV_IMG_ZOOM_NONE, 0, 0, CANVAS_
↵WIDTH / 2, CANVAS_HEIGHT / 2, true);
52
53     }
54
55     #endif

```

28.7.2 色度键控的透明画布

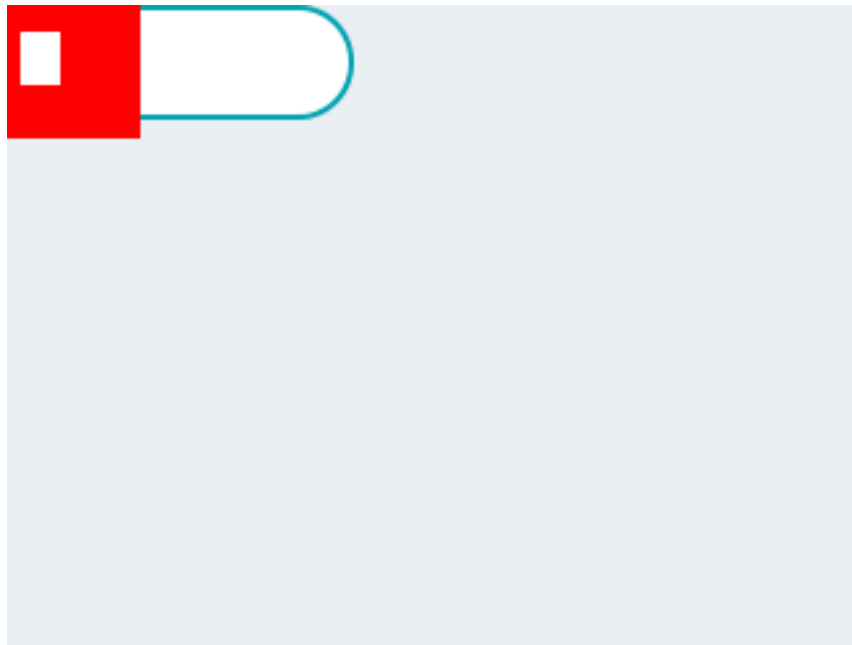


图 2: 色度键控的透明画布

上述效果的示例代码:

```

1  #include "../lv_examples.h"
2  #if LV_USE_CANVAS
3
4  #define CANVAS_WIDTH  50
5  #define CANVAS_HEIGHT 50
6
7  /**
8   * Create a transparent canvas with Chroma keying and indexed color format.
9   *↪ (palette).
10  */
11  void lv_ex_canvas_2(void)
12  {
13      /*Create a button to better see the transparency*/
14      lv_btn_create(lv_scr_act(), NULL);
15
16      /*Create a buffer for the canvas*/
17      static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_INDEXED_1BIT(CANVAS_WIDTH, ↪
18      ↪CANVAS_HEIGHT)];
19
20      /*Create a canvas and initialize its the palette*/
21      lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);

```

(下页继续)

(续上页)

```
20     lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH, CANVAS_HEIGHT, LV_IMG_CF_
↪INDEXED_1BIT);
21     lv_canvas_set_palette(canvas, 0, LV_COLOR_TRANSP);
22     lv_canvas_set_palette(canvas, 1, LV_COLOR_RED);
23
24     /*Create colors with the indices of the palette*/
25     lv_color_t c0;
26     lv_color_t c1;
27
28     c0.full = 0;
29     c1.full = 1;
30
31     /*Transparent background*/
32     lv_canvas_fill_bg(canvas, c1, LV_OPA_TRANSP);
33
34     /*Create hole on the canvas*/
35     uint32_t x;
36     uint32_t y;
37     for( y = 10; y < 30; y++) {
38         for( x = 5; x < 20; x++) {
39             lv_canvas_set_px(canvas, x, y, c0);
40         }
41     }
42
43 }
44 #endif
```

28.8 相关 API

28.8.1 Typedefs

```
1     typedef uint8_t lv_canvas_part_t
```

28.8.2 enums

```
enum [anonymous]
值:
enumerator LV_CANVAS_PART_MAIN
```

28.8.3 函数

```
lv_obj_t * lv_canvas_create(lv_obj_t * par, const lv_obj_t * copy)
功能: 创建一个画布对象
返回:
指向创建的画布的指针
形参:
par: 指向对象的指针, 它将是新画布的父对象
copy: 指向画布对象的指针, 如果不为 NULL, 则将从其复制新对象

void lv_canvas_set_buffer(lv_obj_t * canvas, void * buf, lv_coord_t w, lv_coord_t h,
lv_img_cf_t cf )
功能: 为画布设置缓冲区。
形参:
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
canvas: 指向画布对象的指针
w: 画布的宽度
h: 画布的高度
cf: 颜色格式。LV_IMG_CF_...

void lv_canvas_set_px(lv_obj_t * canvas, lv_coord_t x, lv_coord_t y, lv_color_t c )
功能: 设置画布上像素的颜色
形参:
canvas:
x: 要设置点的 x 坐标
y: 要设置点的 x 坐标
c: 点的颜色

void lv_canvas_set_palette(lv_obj_t * canvas, uint8_t id, lv_color_t c )
功能: 使用索引格式设置画布的调色板颜色。仅对 LV_IMG_CF_INDEXED1/2/4/8
形参:
```

(下页继续)

(续上页)

canvas: 指向画布对象的指针

id: 要设置的调色板颜色:

为 LV_IMG_CF_INDEXED1: 0..1

为 LV_IMG_CF_INDEXED2: 0..3

为 LV_IMG_CF_INDEXED4: 0..15

为 LV_IMG_CF_INDEXED8: 0..255

c: 要设置的颜色

```
lv_color_t lv_canvas_get_px(lv_obj_t * canvas, lv_coord_t x, lv_coord_t y )
```

功能: 获取画布上像素的颜色

返回:

点的颜色

形参:

canvas:

x: 要设置点的 x 坐标

y: 要设置点的 y 坐标

```
lv_img_dsc_t * lv_canvas_get_img(lv_obj_t * canvas )
```

功能: 获取画布的图像作为指向 lv_img_dsc_t 变量的指针。

返回:

指向图像描述符的指针。

形参:

canvas: 指向画布对象的指针

```
void lv_canvas_copy_buf(lv_obj_t * canvas, const void * to_copy, lv_coord_t x, lv_coord_t y, lv_coord_t w, lv_coord_t h )
```

功能: 将缓冲区复制到画布

形参:

canvas: 指向画布对象的指针

to_copy: 要复制的缓冲区。颜色格式必须与画布的缓冲区颜色格式匹配

x: 目标位置的左侧

y: 目标位置的顶部

w: 要复制的缓冲区的宽度

h: 要复制的缓冲区的高度

```
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 pivot_x, int32_t pivot_y, bool antialias )
```

功能: 转换并成像并将结果存储在画布上。

(下页继续)

(续上页)

形参:

canvas: 指向画布对象的指针, 以存储转换结果。

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

pivot_x: 旋转 X 轴。相对于源画布设置为围绕中心旋转 source width / 2

pivot_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。

```
void lv_canvas_blur_hor(lv_obj_t *canvas, constlv_area_t * area, uint16_t r )
```

功能: 在画布上应用水平模糊

形参:

canvas: 指向画布对象的指针

area: 要模糊的区域。如果 NULL 整个画布会模糊。

r: 模糊半径

```
void lv_canvas_blur_ver(lv_obj_t *canvas, constlv_area_t * area, uint16_t r )
```

功能: 在画布上应用垂直模糊

形参:

canvas: 指向画布对象的指针

area: 要模糊的区域。如果 NULL 整个画布会模糊。

r: 模糊半径

```
void lv_canvas_fill_bg(lv_obj_t *canvas, lv_color_t color, lv_opa_t opa )
```

功能: 用颜色填充画布

形参:

canvas: 指向画布的指针

color: 背景色

opa: 所需的不透明度

```
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, constlv_draw_rect_dsc_t * rect_dsc )
```

功能: 在画布上画一个矩形

形参:

canvas: 指向画布对象的指针

x: 矩形的左坐标

y: 矩形的顶部坐标

(下页继续)

(续上页)

116 w: 矩形的宽度

117 h: 矩形的高度

118 rect_dsc: 矩形的描述符

120
121 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, constchar * txt, lv_label_align_t align)

122 功能: 在画布上绘制文本。

123 形参:

124 canvas: 指向画布对象的指针

125 x: 文本的左坐标

126 y: 文本的顶部坐标

127 max_w: 文字的最大宽度。文本将被包装以适合此大小

128 label_draw_dsc: 指向有效标签描述符的指针 lv_draw_label_dsc_t

129 txt: 要显示的文字

130 align: 对齐文字 (LV_LABEL_ALIGN_LEFT/RIGHT/CENTER)

131
132
133 void lv_canvas_draw_img(lv_obj_t * canvas, lv_coord_t x, lv_coord_t y, constvoid *_
↪src, constlv_draw_img_dsc_t * img_draw_dsc)

134 功能: 在画布上绘制图像

135 形参:

136 canvas: 指向画布对象的指针

137 x: 图像的左坐标

138 y: 图像的最高坐标

139 src: 图像源。可以是 lv_img_dsc_t 变量的指针或图像的路径。

140 img_draw_dsc: 指向有效标签描述符的指针 lv_draw_img_dsc_t

141
142
143 void lv_canvas_draw_line(lv_obj_t *canvas, constlv_point_t points[], uint32_t_
↪point_cnt, constlv_draw_line_dsc_t * line_draw_dsc)

144 功能: 在画布上画一条线

145 形参:

146 canvas: 指向画布对象的指针

147 points: 线的点

148 point_cnt: 点数

149 line_draw_dsc: 指向初始化 lv_draw_line_dsc_t 变量的指针

150
151
152 void lv_canvas_draw_polygon(lv_obj_t *canvas, constlv_point_t 点 [], uint32_t point_
↪cnt, constlv_draw_rect_dsc_t * poly_draw_dsc)

153 功能: 在画布上绘制多边形

154 形参:

(下页继续)

(续上页)

```
155 canvas: 指向画布对象的指针
156 points: 多边形的点
157 point_cnt: 点数
158 poly_draw_dsc: 指向初始化 lv_draw_rect_dsc_t 变量的指针
159
160
161 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 )
162 功能: 在画布上画圆弧
163 形参:
164 canvas: 指向画布对象的指针
165 x: 弧的 origo x
166 y: 弧线的起源
167 r: 圆弧半径
168 start_angle: 起始角度 (度)
169 end_angle: 结束角度 (度)
170 arc_draw_dsc: 指向初始化 lv_draw_line_dsc_t 变量的指针
```

29.1 概述

复选框 (Checkbox) 对象是从 Button 背景构建的，Button 背景还包含 Button 项目符号和 Label，以实现经典的复选框。

29.2 零件和样式

该复选框的主要部分称为 LV_CHECKBOX_PART_BG。它是“项目符号”及其旁边的文本的容器。背景使用所有典型的背景样式属性。

项目符号是真正的基础对象 (lv_obj)，可以用 LV_CHECKBOX_PART_BULLET 引用。项目符号会自动继承背景状态。因此，背景被按下时，项目符号也会进入按下状态。项目符号还使用所有典型的背景样式属性。

标签没有专用部分。因为文本样式属性始终是继承的，所以可以在背景样式中设置其样式。

29.3 用法

29.3.1 文本

可以通过 `lv_checkbox_set_text(cb, "New text")` 函数修改文本。它将动态分配文本。

要设置静态文本, 请使用 `lv_checkbox_set_static_text(cb, txt)`。这样, 将仅存储 `txt` 指针, 并且在存在复选框时不应释放该指针。

29.3.2 选中/取消选中

可以通过 `lv_checkbox_set_checked(cb, true/false)` 手动选中/取消选中复选框。设置为 `true` 将选中该复选框, 而设置为 `false` 将取消选中该复选框。

29.3.3 禁用复选框

要禁用复选框, 调用 `lv_checkbox_set_disabled(cb, true)`。

29.3.4 获取/设置复选框状态

可以使用 `lv_checkbox_get_state(cb)` 函数获取 `Checkbox` 的当前状态, 该函数返回当前状态。可以使用 `lv_checkbox_set_state(cb, state)` 设置复选框的当前状态。枚举 `lv_btn_state_t` 定义的可用状态为:

- **LV_BTN_STATE_RELEASED**
- **LV_BTN_STATE_PRESSED**
- **LV_BTN_STATE_DISABLED**
- **LV_BTN_STATE_CHECKED_RELEASED**
- **LV_BTN_STATE_CHECKED_PRESSED**
- **LV_BTN_STATE_CHECKED_DISABLED**

29.4 事件

除了 通用事件, 复选框还支持以下 特殊事件:

- **LV_EVENT_VALUE_CHANGED** - 切换复选框时发送。

请注意, 与通用输入设备相关的事件 (如 `LV_EVENT_PRESSED`) 也以非活动状态发送。需要使用 `lv_cb_is_inactive(cb)` 检查状态, 以忽略非活动复选框中的事件。

了解有关 [事件](#) 的更多内容。

29.5 按键

复选框可处理以下按键：

- **LV_KEY_RIGHT/UP** - 如果启用了切换，则进入切换状态
- **LV_KEY_LEFT/DOWN** - 如果启用了切换，则进入非切换状态

请注意，与往常一样，LV_KEY_ENTER 的状态会转换为 LV_EVENT_PRESSED / PRESSING / RELEASED 等。

了解有关 [按键](#) 的更多内容。

29.6 范例

29.6.1 简单复选框



图 1: 一个简单的复选框

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_CHECKBOX
4
```

(下页继续)

(续上页)

```
5     static void event_handler(lv_obj_t * obj, lv_event_t event)
6     {
7         if(event == LV_EVENT_VALUE_CHANGED) {
8             printf("State: %s\n", lv_checkbox_is_checked(obj) ? "Checked" :
↪ "Unchecked");
9         }
10    }
11
12    void lv_ex_checkbox_1(void)
13    {
14        lv_obj_t * cb = lv_checkbox_create(lv_scr_act(), NULL);
15        lv_checkbox_set_text(cb, "I agree to terms and conditions.");
16        lv_obj_align(cb, NULL, LV_ALIGN_CENTER, 0, 0);
17        lv_obj_set_event_cb(cb, event_handler);
18    }
19
20    #endif
```

29.7 相关 API

TODO

30.1 概述

图表是可视化数据点的基本对象。它们支持折线图（将点与线连接和/或在其上绘制点）和柱形图。

图表还支持分隔线，2 y 轴，刻度线和刻度线文本。

30.2 零件和样式

图表的主要部分称为 `LV_CHART_PART_BG`，它使用所有典型的背景属性。文本样式属性确定轴文本的样式，而线属性确定刻度线的样式。填充值在侧面增加了一些空间，因此使序列区域更小。填充也可用于为轴文本和刻度线留出空间。

该系列的背景称为 `LV_CHART_PART_SERIES_BG`，它位于主要背景上。在此部分上绘制了分隔线和系列数据。除典型的背景样式属性外，分割线还使用线型属性。填充值指示此零件与轴文本之间的间隔。

`LV_CHART_PART_SERIES` 可以引用该系列的样式。对于列类型，使用以下属性：

- 半径：数据点的半径
- `padding_inner`：相同 x 坐标的列之间的间隔

如果是线型图，则使用以下属性：

- 线属性来描述线
- 点的大小半径

- `bg_opa`: 线条下方区域的整体不透明度
- `bg_main_stop`: 的`%bg_opa`在顶部以创建一个 alpha 褪色 (0: 在顶部透明, 255: `bg_opa` 在顶部)
- `bg_grad_stop`: 底部 `bg_opa` 的百分比以创建 alpha 渐变 (0: 底部透明, 255: `bg_opa` 顶部)
- `bg_drag_dir`: 应该 `LV_GRAD_DIR_VER` 允许通过 `bg_main_stop` 和 `bg_grad_stop` 进行 Alpha 淡入

`LV_CHART_PART_CURSOR` 引用游标。可以添加任意数量的光标, 并且可以通过与行相关的样式属性来设置其外观。创建游标时设置游标的颜色, 并用该值覆盖 `line_color` 样式。

30.3 用法

30.3.1 数据系列

您可以通过 `lv_chart_add_series(chart, color)` 向图表添加任意数量的系列。它为包含所选颜色的 `lv_chart_u series_t` 结构分配数据, 如果不使用外部数组, 如果分配了外部数组, 则与该系列关联的任何内部点都将被释放, 而序列指向外部数组。

30.3.2 系列类型

存在以下数据显示类型:

- `LV_CHART_TYPE_NONE` - 不显示任何数据。它可以用来隐藏系列。
- `LV_CHART_TYPE_LINE` - 在两点之间画线。
- `LV_CHART_TYPE_COLUMN` - 绘制列。

可以使用 `lv_chart_set_type(chart, LV_CHART_TYPE_...)` 指定显示类型。可以对类型进行“或”运算 (例如 `LV_CHART_TYPE_LINE`)。

30.3.3 修改数据

有几个选项可以设置系列数据:

1. 在数组中手动 设置值, 例如 `ser1->points[3] = 7`, 然后使用 `lv_chart_refresh(chart)` 刷新图表。
2. 使用 `lv_chart_set_point_id(chart, ser, value, id)`, 其中 `id` 是您要更新的点的索引。
3. 使用 `lv_chart_set_next(chart, ser, value)`。
4. 使用 `lv_chart_init_points(chart, ser, value)` 将所有点初始化为给定值。
5. 使用 `lv_chart_set_points(chart, ser, value_array)` 设置数组中的所有点。

使用 `LV_CHART_POINT_DEF` 作为值可使库跳过该点, 列或线段的绘制。

30.3.4 覆盖系列的默认起点

如果希望绘图从默认点（序列的点 [0]）之外的其他点开始，则可以使用 `lv_chart_set_x_start_point(chart, ser, id)` 函数设置替代索引，其中 `id` 是要开始的新索引位置从。

30.3.5 设置外部数据源

可以使用以下函数从外部数据源更新图表系列：`lv_chart_set_ext_array(chart, ser, array, point_cnt)`，其中 `array` 是 `lv_coord_t` 与 `point_cnt` 元素的外部数组。注意：更新外部数据源后，应调用 `lv_chart_refresh(chart)` 来更新图表。

30.3.6 获取当前图表信息

有四个功能可获取有关图表的信息：

- `lv_chart_get_type(chart)` 返回当前图表类型。
- `lv_chart_get_point_count(chart)` 返回当前图表点数。
- `lv_chart_get_x_start_point(ser)` 返回指定系列的当前绘图索引。
- `lv_chart_get_point_id(chart, ser, id)` 返回指定系列的特定索引处的数据值。

30.3.7 更新模式

`lv_chart_set_next` 可以以两种方式运行，具体取决于更新模式：

- **LV_CHART_UPDATE_MODE_SHIFT** - 将旧数据向左移动，然后向右添加新数据。
- **LV_CHART_UPDATE_MODE_CIRCULAR** - 循环添加新数据（如 ECG 图）。

可以使用 `lv_chart_set_update_mode(chart, LV_CHART_UPDATE_MODE_...)` 更改更新模式。

30.3.8 浮标个数

可以通过 `lv_chart_set_point_count(chart, point_num)` 修改系列中的点数。默认值为 10。注意：当将外部缓冲区分配给序列时，这也会影响处理的点数。

30.3.9 垂直范围

可以使用 `lv_chart_set_range(chart, y_min, y_max)` 在 y 方向上指定最小值和最大值。点的值将按比例缩放。默认范围是：0..100。

30.3.10 分割线

水平和垂直分隔线的数量可以通过 `lv_chart_set_div_line_count(chart, hdiv_num, vdiv_` 进行修改。默认设置为 3 条水平分割线和 5 条垂直分割线。

30.3.11 刻度线和标签

刻度和标签可以添加到轴上。

`lv_chart_set_x_tick_text(chart, list_of_values, num_tick_marks, LV_CHART_AXIS_...)` 设置 x 轴上的刻度和文本。`list_of_values` 是一个字符串，带有 '\n' 终止文本（期望最后一个），其中包含用于刻度的文本。`list_of_values` 是一个字符串，带有 '\n' 终止文本（期望最后一个），其中包含用于刻度的文本。例如。`const char * list_of_values = "first\nsec\nthird"`。`list_of_values` 可以为 NULL。如果设置了 `list_of_values`，则 `num_tick_marks` 告诉两个标签之间的刻度数。如果 `list_of_values` 为 NULL，则它指定滴答声的总数。

主刻度线绘制在放置文本的位置，次刻度线绘制在其他位置。“`lv_chart_set_x_tick_length(chart, major_tick_len, minor_tick_len)`”设置 x 轴上刻度线的长度。

y 轴也存在相同的功能：`lv_chart_set_y_tick_text` 和 `lv_chart_set_y_tick_length`。

30.3.12 光标

可以使用 `lv_chart_cursor_t * c1 = lv_chart_add_cursor(chart, color, dir);` 添加光标。`dir` LV_CHART_CURSOR_NONE/RIGHT/UP/LEFT/DOWN 的可能值或它们的 OR-ed 值，用于指示应在哪个方向上绘制光标。

`lv_chart_set_cursor_point(chart, cursor, &point)` 设置光标的位置。`point` 是指向 `lv_point_t` 变量的指针。例如。`lv_point_t point = {10, 20};`。该点相对于图表的序列区域。

`lv_coord_t p_index = lv_chart_get_nearest_index_from_coord(chart, x)` 告诉哪个点索引最接近 X 坐标（相对于序列区域）。例如，当单击图表时，它可用于将光标捕捉到一个点。

`lv_chart_get_x_from_index(chart, series, id)` 和 `lv_chart_get_y_from_index(chart, series, id)` 告诉给定点的 X 和 Y 坐标。将光标放置到给定点很有用。

可以使用 `lv_chart_get_series_area(chart, &area)` 检索当前系列区域，其中 `area` 是指向 `lv_area_t` 变量的指针，用于存储结果。该区域具有绝对坐标。

30.4 事件

仅通用事件是按对象类型发送的。

了解有关 [事件](#) 的更多信息。

30.5 按键

对象类型不处理任何键。

进一步了解 [按键](#) 。

30.6 范例

30.6.1 折线图

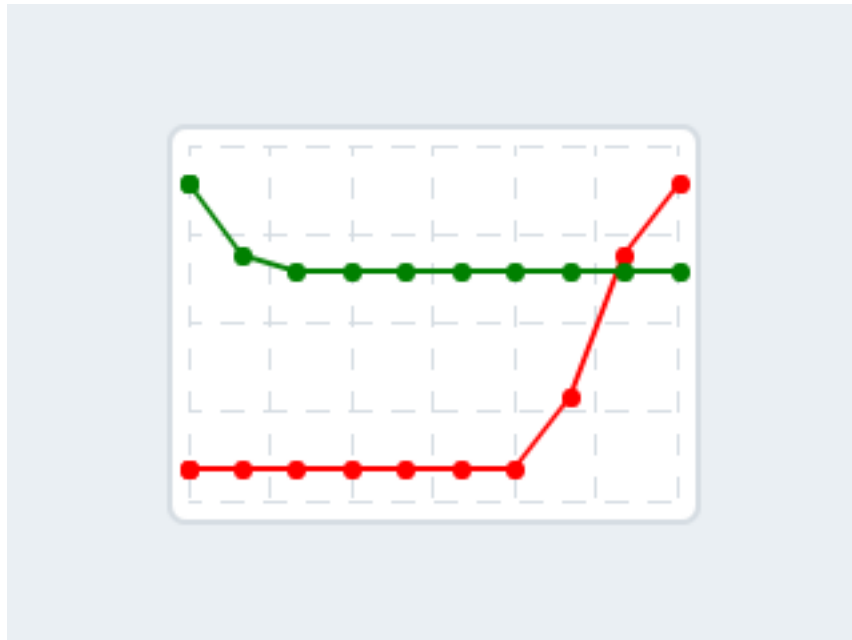


图 1: 折线图

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2  #if LV_USE_CHART
3
4  void lv_ex_chart_1(void)
```

(下页继续)

(续上页)

```
5  {
6      /*Create a chart*/
7      lv_obj_t * chart;
8      chart = lv_chart_create(lv_scr_act(), NULL);
9      lv_obj_set_size(chart, 200, 150);
10     lv_obj_align(chart, NULL, LV_ALIGN_CENTER, 0, 0);
11     lv_chart_set_type(chart, LV_CHART_TYPE_LINE);    /*Show lines and points_
↪too*/
12
13     /*Add two data series*/
14     lv_chart_series_t * ser1 = lv_chart_add_series(chart, LV_COLOR_RED);
15     lv_chart_series_t * ser2 = lv_chart_add_series(chart, LV_COLOR_GREEN);
16
17     /*Set the next points on 'ser1'*/
18     lv_chart_set_next(chart, ser1, 10);
19     lv_chart_set_next(chart, ser1, 10);
20     lv_chart_set_next(chart, ser1, 10);
21     lv_chart_set_next(chart, ser1, 10);
22     lv_chart_set_next(chart, ser1, 10);
23     lv_chart_set_next(chart, ser1, 10);
24     lv_chart_set_next(chart, ser1, 10);
25     lv_chart_set_next(chart, ser1, 30);
26     lv_chart_set_next(chart, ser1, 70);
27     lv_chart_set_next(chart, ser1, 90);
28
29     /*Directly set points on 'ser2'*/
30     ser2->points[0] = 90;
31     ser2->points[1] = 70;
32     ser2->points[2] = 65;
33     ser2->points[3] = 65;
34     ser2->points[4] = 65;
35     ser2->points[5] = 65;
36     ser2->points[6] = 65;
37     ser2->points[7] = 65;
38     ser2->points[8] = 65;
39     ser2->points[9] = 65;
40
41     lv_chart_refresh(chart); /*Required after direct set*/
42 }
43
44 #endif
```

30.6.2 折线图添加褪色区域效果

图 2: 折线图添加褪色区域效果

上述效果的示例代码:

```

1  #include "../lv_examples.h"
2  #if LV_USE_CHART
3
4  /**
5   * Add a faded area effect to the line chart
6   */
7  void lv_ex_chart_2(void)
8  {
9      /*Create a chart*/
10     lv_obj_t * chart;
11     chart = lv_chart_create(lv_scr_act(), NULL);
12     lv_obj_set_size(chart, 200, 150);
13     lv_obj_align(chart, NULL, LV_ALIGN_CENTER, 0, 0);
14     lv_chart_set_type(chart, LV_CHART_TYPE_LINE);    /*Show lines and points_
↪too*/
15
16     /*Add a faded are effect*/
17     lv_obj_set_style_local_bg_opa(chart, LV_CHART_PART_SERIES, LV_STATE_
↪DEFAULT, LV_OPA_50); /*Max. opa.*/
18     lv_obj_set_style_local_bg_grad_dir(chart, LV_CHART_PART_SERIES, LV_STATE_
↪DEFAULT, LV_GRAD_DIR_VER);
19     lv_obj_set_style_local_bg_main_stop(chart, LV_CHART_PART_SERIES, LV_STATE_
↪DEFAULT, 255);    /*Max opa on the top*/
20     lv_obj_set_style_local_bg_grad_stop(chart, LV_CHART_PART_SERIES, LV_STATE_
↪DEFAULT, 0);      /*Transparent on the bottom*/
21
22
23     /*Add two data series*/
24     lv_chart_series_t * ser1 = lv_chart_add_series(chart, LV_COLOR_RED);
25     lv_chart_series_t * ser2 = lv_chart_add_series(chart, LV_COLOR_GREEN);
26
27     /*Set the next points on 'ser1'*/
28     lv_chart_set_next(chart, ser1, 31);
29     lv_chart_set_next(chart, ser1, 66);
30     lv_chart_set_next(chart, ser1, 10);
31     lv_chart_set_next(chart, ser1, 89);
32     lv_chart_set_next(chart, ser1, 63);
33     lv_chart_set_next(chart, ser1, 56);

```

(下页继续)

(续上页)

```
34     lv_chart_set_next(chart, ser1, 32);
35     lv_chart_set_next(chart, ser1, 35);
36     lv_chart_set_next(chart, ser1, 57);
37     lv_chart_set_next(chart, ser1, 85);
38
39     /*Directly set points on 'ser2'*/
40     ser2->points[0] = 92;
41     ser2->points[1] = 71;
42     ser2->points[2] = 61;
43     ser2->points[3] = 15;
44     ser2->points[4] = 21;
45     ser2->points[5] = 35;
46     ser2->points[6] = 35;
47     ser2->points[7] = 58;
48     ser2->points[8] = 31;
49     ser2->points[9] = 53;
50
51     lv_chart_refresh(chart); /*Required after direct set*/
52 }
53
54 #endif
```

30.7 相关 API

30.7.1 Typedefs

```
1  typedef uint8_t lv_chart_type_t
2  typedef uint8_t lv_chart_update_mode_t
3  typedef uint8_t lv_chart_axis_t
4  typedef uint8_t lv_cursor_direction_t
5  typedef uint8_t lv_chart_axis_options_t
```

30.7.2 enums

```
1  enum [anonymous]
2  图表类型
3  值:
4  enumerator LV_CHART_TYPE_NONE
5  不要画系列
```

(下页继续)

(续上页)

```

6     enumerator LV_CHART_TYPE_LINE
7     用线连接点
8     enumerator LV_CHART_TYPE_COLUMN
9     绘制列
10
11
12     enum [anonymous]
13     图表更新模式 lv_chart_set_next
14     值:
15     enumerator LV_CHART_UPDATE_MODE_SHIFT
16     向左移动旧数据, 向右添加新数据
17     enumerator LV_CHART_UPDATE_MODE_CIRCULAR
18     循环添加新数据
19
20
21     enum [anonymous]
22     值:
23     enumerator LV_CHART_AXIS_PRIMARY_Y
24     enumerator LV_CHART_AXIS_SECONDARY_Y
25     enumerator _LV_CHART_AXIS_LAST
26
27
28     enum [anonymous]
29     值:
30     enumerator LV_CHART_CURSOR_NONE
31     enumerator LV_CHART_CURSOR_RIGHT
32     enumerator LV_CHART_CURSOR_UP
33     enumerator LV_CHART_CURSOR_LEFT
34     enumerator LV_CHART_CURSOR_DOWN
35
36
37     enum [anonymous]
38     轴数据
39     值:
40     enumerator LV_CHART_AXIS_SKIP_LAST_TICK
41     不要画最后的刻度
42     enumerator LV_CHART_AXIS_DRAW_LAST_TICK
43     画最后一个刻度
44     enumerator LV_CHART_AXIS_INVERSE_LABELS_ORDER
45     以倒序绘制刻度标签
46
47
48     enum [anonymous]

```

(下页继续)

(续上页)

值:

```

49  enumerator LV_CHART_PART_BG
50
51  enumerator LV_CHART_PART_SERIES_BG
52  enumerator LV_CHART_PART_SERIES
53  enumerator LV_CHART_PART_CURSOR

```

30.7.3 函数

```

1  LV_EXPORT_CONST_INT( LV_CHART_POINT_DEF )
2  LV_EXPORT_CONST_INT( LV_CHART_TICK_LENGTH_AUTO )
3  lv_obj_t * lv_chart_create(lv_obj_t * par, constlv_obj_t *copy)
4  功能: 创建图表背景对象
5  返回:
6  指向创建的图表背景的指针
7  形参:
8  par: 指向对象的指针, 它将是新图表背景的父对象
9  copy: 指向图表背景对象的指针, 如果不为 NULL, 则将从其复制新对象
10
11
12  lv_chart_series_t * lv_chart_add_series(lv_obj_t *chart, lv_color_t color)
13  功能: 分配数据系列并将其添加到图表
14  返回:
15  指向已分配数据系列的指针
16  形参:
17  chart: 指向图表对象的指针
18  color: 数据系列的颜色
19
20
21  void lv_chart_clear_series(lv_obj_t *chart, lv_chart_series_t *series)
22  功能: 明确系列的要点
23  形参:
24  chart: 指向图表对象的指针
25  series: 指向要清除的图表系列的指针
26
27
28  void lv_chart_set_div_line_count(lv_obj_t *chart, uint8_t hdiv, uint8_t vdiv )
29  功能: 设置水平和垂直分割线的数量
30  形参:
31  chart: 指向图形背景对象的指针
32  hdiv: 水平分割线数
33  vdiv: 垂直分割线数
34

```

(下页继续)

(续上页)

```

35
36 void lv_chart_set_y_range(lv_obj_t *chart, lv_chart_axis_t axis, lv_coord_t ymin,
lv_coord_t ymax )

```

功能：在轴上设置最小和最大 y 值

形参：

chart：指向图形背景对象的指针

axis：LV_CHART_AXIS_PRIMARY_Y 或 LV_CHART_AXIS_SECONDARY_Y

ymin：y 最小值

ymax：y 最大值

```

44
45 void lv_chart_set_type(lv_obj_t *chart, lv_chart_type_t type)

```

功能：设置图表的新类型

形参：

chart：指向图表对象的指针

type：图表的新类型（来自“lv_chart_type_t”enum）

```

50
51
52 void lv_chart_set_point_count(lv_obj_t *chart, uint16_t point_cnt )

```

功能：设置图表上数据线上的点数

形参：

chart：指向图表对象的指针 r

point_cnt：数据线上的新点数

```

57
58
59 void lv_chart_init_points(lv_obj_t *chart, lv_chart_series_t * ser, lv_coord_t y )

```

功能：用一个值初始化所有数据点

形参：

chart：指向图表对象的指针

ser：指向“图表”上的数据系列的指针

y：所有点的新值

```

60
61
62
63
64
65
66
67 void lv_chart_set_points(lv_obj_t *chart, lv_chart_series_t * ser, lv_coord_t y_
↪array []) )

```

功能：设置数组中的点的值

形参：

chart：指向图表对象的指针

ser：指向“图表”上的数据系列的指针

y_array：'lv_coord_t' 点的数组（带有'points count' 元素）

```

70
71
72
73
74
75 void lv_chart_set_next(lv_obj_t *chart, lv_chart_series_t * ser, lv_coord_t y )

```

(下页继续)

(续上页)

```

76  功能：向右移动所有数据并在数据线上设置最右边的数据
77  形参：
78  chart：指向图表对象的指针
79  ser：指向“图表”上的数据系列的指针
80  y：最正确数据的新价值
81
82
83  void lv_chart_set_update_mode(lv_obj_t *chart, lv_chart_update_mode_t update_mode )
84  功能：设置图表对象的更新模式。
85  形参：
86  chart：指向图表对象的指针
87  update：模式
88
89
90  void lv_chart_set_x_tick_length(lv_obj_t *chart, uint8_t major_tick_len, uint8_t_
↪minor_tick_len )
91  功能：设置 x 轴上刻度线的长度
92  形参：
93  chart：指向图表的指针
94  major_tick_len：主要刻度的长度或 LV_CHART_TICK_LENGTH_AUTO 自动设置（添加标签的位置）
95  minor_tick_len：次刻度的长度，LV_CHART_TICK_LENGTH_AUTO 可自动设置（不添加标签）
96
97
98  void lv_chart_set_y_tick_length(lv_obj_t *chart, uint8_t major_tick_len, uint8_t_
↪minor_tick_len )
99  功能：设置 y 轴上刻度线的长度
100  形参：
101  chart：指向图表的指针
102  major_tick_len：主要刻度的长度或 LV_CHART_TICK_LENGTH_AUTO 自动设置（添加标签的位置）
103  minor_tick_len：次刻度的长度，LV_CHART_TICK_LENGTH_AUTO 可自动设置（不添加标签）
104
105
106  void lv_chart_set_secondary_y_tick_length(lv_obj_t *chart, uint8_t major_tick_len,
uint8_t minor_tick_len )
107  功能：设置次要 y 轴上刻度线的长度
108  形参：
109  chart：指向图表的指针
110  major_tick_len：主要刻度的长度或 LV_CHART_TICK_LENGTH_AUTO 自动设置（添加标签的位置）
111  minor_tick_len：次刻度的长度，LV_CHART_TICK_LENGTH_AUTO 可自动设置（不添加标签）
112
113
114  void lv_chart_set_x_tick_texts(lv_obj_t *chart, constchar * list_of_values, uint8_
↪t num_tick_marks, lv_chart_axis_options_t options )

```

(下页继续)

(续上页)

功能：设置图表的 X 轴刻度计数和标签

形参：

chart：指向图表对象的指针

list_of_values：字符串值列表，以终止，除了最后一个

num_tick_marks：如果 list_of_values 为 NULL：每个轴的总刻度数，否则两个值标签之间的刻度数

options：额外的选择

```
void lv_chart_set_secondary_y_tick_texts(lv_obj_t *chart, constchar * list_of_
↪values, uint8_t num_tick_marks, lv_chart_axis_options_t options )
```

功能：设置次要 y 轴刻度计数和图表标签

形参：

chart：指向图表对象的指针

list_of_values：字符串值列表，以终止，除了最后一个

num_tick_marks：如果 list_of_values 为 NULL：每个轴的总刻度数，否则两个值标签之间的刻度数

options：额外的选择

```
void lv_chart_set_y_tick_texts(lv_obj_t *chart, constchar * list_of_values, uint8_
↪t num_tick_marks, lv_chart_axis_options_t options )
```

功能：设置图表的 Y 轴刻度计数和标签

形参：

chart：指向图表对象的指针

list_of_values：字符串值列表，以终止，除了最后一个

num_tick_marks：如果 list_of_values 为 NULL：每个轴的总刻度数，否则两个值标签之间的刻度数

options：额外的选择

```
void lv_chart_set_x_start_point(lv_obj_t *chart, lv_chart_series_t * ser, uint16_t_
↪id )
```

功能：设置数据数组中 x 轴起点的索引

形参：

chart：指向图表对象的指针

ser：指向“图表”上的数据系列的指针

id：数据数组中 x 点的索引

```
void lv_chart_set_ext_array(lv_obj_t *chart, lv_chart_series_t * ser, lv_coord_t_
↪array[], uint16_t point_cnt )
```

功能：设置要用于图表的外部数据点阵列注意：用户有责任确保 point_cnt 与外部阵列大小匹配。

形参：

chart：指向图表对象的指针

(下页继续)

(续上页)

```

154     ser: 指向“图表”上的数据系列的指针
155     array: 图表的外部点数组
156
157
158     void lv_chart_set_point_id(lv_obj_t *chart, lv_chart_series_t * ser, lv_coord_t_
↪value, uint16_t id )
159     功能: 直接基于索引在图表系列中设置单个点值
160     形参:
161     chart: 指向图表对象的指针
162     ser: 指向“图表”上的数据系列的指针
163     value: 分配给数组点的值
164     id: 数组中 x 点的索引
165
166
167     void lv_chart_set_series_axis(lv_obj_t *chart, lv_chart_series_t * ser, lv_chart_
↪axis_t axis)
168     功能: 设置系列的 Y 轴
169     形参:
170     chart: 指向图表对象的指针
171     ser: 指向系列的指针
172     axis: LV_CHART_AXIS_PRIMARY_Y 或 LV_CHART_AXIS_SECONDARY_Y
173
174
175     lv_chart_type_t lv_chart_get_type(const lv_obj_t *chart)
176     获取图表类型
177     返回:
178     图表类型 (来自 “ lv_chart_t” enum )
179     形参:
180     chart: 指向图表对象的指针
181
182
183     uint16_t lv_chart_get_point_count(const lv_obj_t *chart)
184     功能: 获取图表上每条数据线的的数据点编号
185     返回:
186     每条数据线上的点号
187     形参:
188     chart: 指向图表对象的指针
189
190
191     uint16_t lv_chart_get_x_start_point(lv_chart_series_t * ser )
192     功能: 获取数据数组中 x 轴起点的当前索引
193     返回:
194     数据数组中当前 x 起点的索引

```

(下页继续)

(续上页)

形参:

ser: 指向“图表”上的数据系列的指针

```
lv_coord_t lv_chart_get_point_id(lv_obj_t *chart, lv_chart_series_t * ser, uint16_t id )
```

功能: 直接基于索引获取图表系列中的单个点值

返回:

索引 ID 处数组点的值

形参:

chart: 指向图表对象的指针

ser: 指向“图表”上的数据系列的指针

id: 数组中 x 点的索引

```
lv_chart_axis_t lv_chart_get_series_axis(lv_obj_t *chart, lv_chart_series_t * ser )
```

功能: 获取系列的 Y 轴

返回:

LV_CHART_AXIS_PRIMARY_Y 要么 LV_CHART_AXIS_SECONDARY_Y

形参:

chart: 指向图表对象的指针

ser: 指向系列的指针

```
void lv_chart_refresh(lv_obj_t *chart)
```

功能: 如果其数据线已更改, 则刷新图表

形参:

chart: 指向图表对象的指针

31.1 概述

容器本质上是具有布局和自动调整大小功能的 **基本对象**。

31.2 零件和样式

容器只有一个主要样式称为 `LV_CONT_PART_MAIN`，它可以使用所有通常的 `background` 属性和填充来自动调整布局大小。

31.3 用法

31.3.1 布局

可以在容器上应用布局以自动订购其子代。布局间距来自样式的 **填充** (`pad`) 属性。可能的布局选项：

- **LV_LAYOUT_OFF** - 不要对齐子代。
- **LV_LAYOUT_CENTER** - 将子项与列中的中心对齐，并 `pad_inner` 在它们之间保持间距。
- **LV_LAYOUT_COLUMN_LEFT** - 在左对齐的列中对齐子级。请 `pad_left` 在左边，空间 `pad_top` 空间的顶部和 `pad_inner` 孩子之间的空间。

- **LV_LAYOUT_COLUMN_MID** - 在中心列中对齐子代。padd_top 在顶部和 pad_inner 孩子之间保持空间。
- **LV_LAYOUT_COLUMN_RIGHT** - 在右对齐的列中对齐子代。保持 padd_right 右边的 pad_top 空间，顶部的“pad_inner 空间和孩子之间的空间。
- **LV_LAYOUT_ROW_TOP** - 在顶部对齐的行中对齐子级。请 pad_left 在左边，空间 pad_top 空间的顶部和 pad_inner“ 孩子之间的空间。
- **LV_LAYOUT_ROW_MID** - 在居中的行中对齐子级。pad_left 在左边和 pad_inner 孩子之间保持空间。
- **LV_LAYOUT_ROW_BOTTOM** - 在底部对齐的行中对齐子级。请 pad_left 在左边，空间 pad_bottom 空间的底部和 pad_inner 孩子之间的空间。
- **LV_LAYOUT_PRETTY_TOP** - 将作为连续多的对象可能（至少 pad_inner 空间和 pad_left/right 空间两侧）。在孩子之间的每一行中平均分配空间。如果这是连续不同身高的孩子，请对齐其上边缘。
- **LV_LAYOUT_PRETTY_MID** - 与 LV_LAYOUT_PRETTY_MID 相同，但是如果此处的孩子连续排成不同的高度，则对齐他们的中线。
- **LV_LAYOUT_PRETTY_BOTTOM** - 与 ·· LV_LAYOUT_PRETTY_BOTTOM ·· 相同，·· 但是如果这是连续高度不同的子项，请对齐其底线。
- **LV_LAYOUT_GRID** - 类似于 LV_LAYOUT_PRETTY 但不能平均划分水平空间，只是让它们之间的 pad_left/right 边缘和 pad_inner 空间分开。

31.3.2 自动调整

容器具有自动适应功能，可以根据其子代和/或父代自动更改容器的大小。存在以下选项：

- **LV_FIT_NONE** - 不要自动更改大小。
- **LV_FIT_TIGHT** - 将容器收缩包装在其所有子容器周围，同时 pad_top/bottom/left/right 在边缘保留空间。
- **LV_FIT_PARENT** - 将大小设置为父项的大小减去 pad_top/bottom/left/right（来自父项的样式）空间。
- **LV_FIT_MAX** - 使用 LV_FIT_PARENT 而不是父小，LV_FIT_TIGHT 时大。它将确保该容器至少是其父容器的大小。

要为所有方向设置自动适合模式，请使用。要在水平和垂直方向上使用不同的自动拟合，请使用。要在所有四个方向上使用不同的自动拟合，请使用。lv_cont_set_fit(cont, **LV_FIT**...)lv_cont_set_fit2(cont, hor_fit_type, ver_fit_type)lv_cont_set_fit4(cont, left_fit_type, right_fit_type, top_fit_type, bottom_fit_type)

31.4 事件

仅 通用事件 是按对象类型发送的。

了解有关 事件 的更多信息。

31.5 按键

对象类型不处理任何键。

进一步了解 按键 。

31.6 范例

31.6.1 容器示例

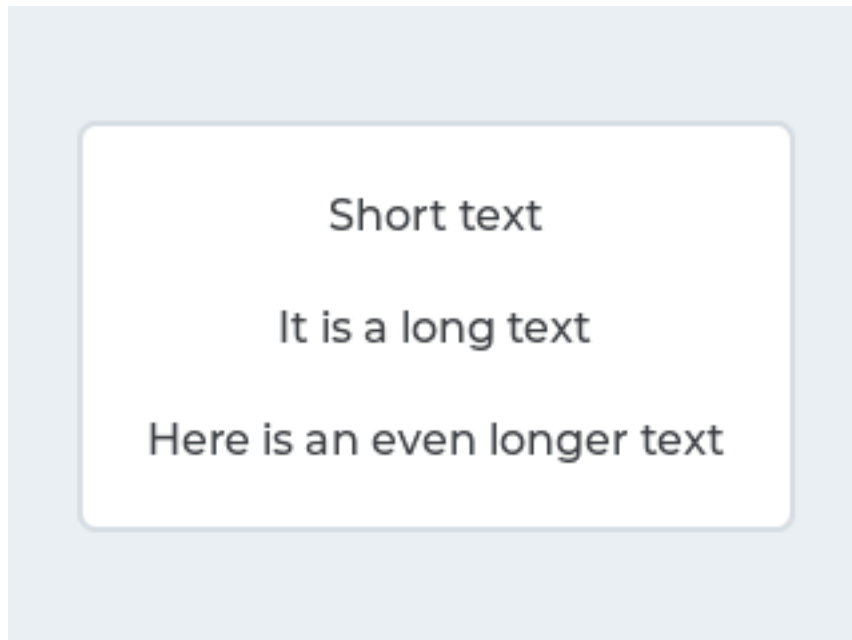


图 1: 简单的容器示例

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #if LV_USE_CONT
3
4  void lv_ex_cont_1(void)
```

(下页继续)

(续上页)

```
5  {
6      lv_obj_t * cont;
7
8      cont = lv_cont_create(lv_scr_act(), NULL);
9      lv_obj_set_auto_realign(cont, true);          /*Auto realign_
↳when the size changes*/
10     lv_obj_align_origo(cont, NULL, LV_ALIGN_CENTER, 0, 0); /*This parametrs_
↳will be sued when realigned*/
11     lv_cont_set_fit(cont, LV_FIT_TIGHT);
12     lv_cont_set_layout(cont, LV_LAYOUT_COLUMN_MID);
13
14     lv_obj_t * label;
15     label = lv_label_create(cont, NULL);
16     lv_label_set_text(label, "Short text");
17
18     /*Refresh and pause here for a while to see how `fit` works*/
19     uint32_t t;
20     lv_refr_now(NULL);
21     t = lv_tick_get();
22     while(lv_tick_elaps(t) < 500);
23
24     label = lv_label_create(cont, NULL);
25     lv_label_set_text(label, "It is a long text");
26
27     /*Wait here too*/
28     lv_refr_now(NULL);
29     t = lv_tick_get();
30     while(lv_tick_elaps(t) < 500);
31
32     label = lv_label_create(cont, NULL);
33     lv_label_set_text(label, "Here is an even longer text");
34 }
35
36 #endif
```

31.7 相关 API

TODO

颜色选择器 (lv_cpicker)

32.1 概述

顾名思义，拾色器允许选择颜色。可以依次选择颜色的色相，饱和度和值。

小部件有两种形式：圆形（圆盘）和矩形。

在这两种形式中，长按对象，颜色选择器将更改为颜色的下一个参数（色相，饱和度或值）。此外，双击将重置当前参数。

32.2 零件和样式

拾色器的主要部分称为 `LV_CPICKER_PART_BG`。以圆形形式，它使用 `scale_width` 设置圆的宽度，并使用 `pad_inner` 在圆和内部预览圆之间填充。在矩形模式下，半径可以用于在矩形上应用半径。

该对象具有称为的虚拟部分 `LV_CPICKER_PART_KNOB`，它是在当前值上绘制的矩形（或圆形）。它使用所有矩形（如样式属性和填充）使其大于圆形或矩形背景的宽度。

32.3 用法

32.3.1 类型

可以使用 `lv_cpicker_set_type(cpicker, LV_CPICKER_TYPE_RECT/DISC)` 更改颜色选择器的类型

32.3.2 设定颜色

可以使用 `lv_cpicker_set_hue/saturation/value(cpicker, x)` 手动设置 `colro`, 或者使用 `lv_cpicker_set_hsv(cpicker, hsv)` 或 `lv_cpicker_set_color(cpicker, rgb)` 一次全部设置

32.3.3 色彩模式

可以使用 `lv_cpicker_set_color_mode(cpicker, LV_CPICKER_COLOR_MODE_HUE/SATURATION/VALUE)` 手动选择当前颜色。

使用 `lv_cpicker_set_color_mode_fixed(cpicker, true)` 固定颜色（不要长按更改）

32.3.4 旋钮颜色

`lv_cpicker_set_knob_colored(cpicker, true)` 使旋钮自动将所选颜色显示为背景色。

32.3.5 事件

仅 [通用事件](#) 是按对象类型发送的。

了解有关 [事件](#) 的更多信息。

32.4 按键

- **LV_KEY_UP, LV_KEY_RIGHT** 将当前参数的值增加 1
- **LV_KEY_DOWN, LV_KEY_LEFT** 将当前参数减 1
- **LV_KEY_ENTER** 长按将显示下一个模式。通过双击将重置当前参数。

进一步了解 [按键](#)。

32.5 范例

32.5.1 光盘颜色选择器



图 1: 光盘颜色选择器

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #if LV_USE_CPICKER
3
4  void lv_ex_cpicker_1(void)
5  {
6      lv_obj_t * cpicker;
7
8      cpicker = lv_cpicker_create(lv_scr_act(), NULL);
9      lv_obj_set_size(cpicker, 200, 200);
10     lv_obj_align(cpicker, NULL, LV_ALIGN_CENTER, 0, 0);
11 }
12
13 #endif
```

32.6 相关 API

32.6.1 函数

```
lv_obj_t * lv_cpicker_create(lv_obj_t * par, constlv_obj_t *copy)
```

功能：创建一个颜色选择器对象

返回：

指向创建的颜色选择器的指针

形参：

par: 指向对象的指针，它将是新 colorpicker 的父对象

copy: 指向颜色选择器对象的指针，如果不为 NULL，则将从其复制新对象

```
void lv_cpicker_set_type(lv_obj_t * cpicker, lv_cpicker_type_t type)
```

功能：为选色器设置新类型

形参：

cpicker: 指向颜色选择器对象的指针

type: 新型的颜色选择器（来自“lv_cpicker_type_t”枚举）

```
bool lv_cpicker_set_hue(lv_obj_t * cpicker, uint16_t hue)
```

功能：设置颜色选择器的当前色相。

返回：

如果更改，则为 true，否则为 false

形参：

cpicker: 指向 colorpicker 对象的指针

hue: 当前选择的色相 [0..360]

```
bool lv_cpicker_set_saturation(lv_obj_t * cpicker, uint8_t saturation)
```

功能：设置颜色选择器的当前饱和度。

返回：

如果更改，则为 true，否则为 false

形参：

cpicker: 指向 colorpicker 对象的指针

saturation: 当前选择的饱和度 [0..100]

```
bool lv_cpicker_set_value(lv_obj_t * cpicker, uint8_t val )
```

功能：设置颜色选择器的当前值。

返回：

如果更改，则为 true，否则为 false

形参：

(下页继续)

(续上页)

```
40  cpicker: 指向 colorpicker 对象的指针
41  val: 当前选择的值 [0..100]
42
43
44  bool lv_cpicker_set_hsv(lv_obj_t * cpicker, lv_color_hsv_t hsv )
45  功能: 设置颜色选择器的当前 hsv。
46  返回:
47  如果更改, 则为 true, 否则为 false
48  形参:
49  cpicker: 指向 colorpicker 对象的指针
50  hsv: 当前选择的 hsv
51
52
53  bool lv_cpicker_set_color(lv_obj_t * cpicker, lv_color_t color)
54  功能: 设置颜色选择器的当前颜色。
55  返回:
56  如果更改, 则为 true, 否则为 false
57  形参:
58  cpicker: 指向 colorpicker 对象的指针
59  color: 当前选择的颜色
60
61
62  void lv_cpicker_set_color_mode(lv_obj_t * cpicker, lv_cpicker_color_mode_t mode)
63  功能: 设置当前的色彩模式。
64  形参:
65  cpicker: 指向 colorpicker 对象的指针
66  mode: 色彩模式 (色相/饱和度/色度)
67
68
69  void lv_cpicker_set_color_mode_fixed(lv_obj_t * cpicker, bool fixed)
70  功能: 长时间按中心更改颜色模式时设置
71  形参:
72  cpicker: 指向 colorpicker 对象的指针
73  fixed: 长按时无法更改色彩模式
74
75
76  void lv_cpicker_set_knob_colored(lv_obj_t * cpicker, bool en )
77  功能: 使旋钮上色为当前颜色
78  形参:
79  cpicker: 指向 colorpicker 对象的指针
80  en: true: 为旋钮着色; false: 不给旋钮上色
81
82
```

(下页继续)

(续上页)

```
83 lv_cpicker_color_mode_t lv_cpicker_get_color_mode(lv_obj_t * cpicker )
```

```
84 功能：获取当前的颜色模式。
```

```
85 返回：
```

```
86 色彩模式 （色相/饱和度/色度）
```

```
87 形参：
```

```
88 cpicker: 指向 colorpicker 对象的指针
```

```
89
```

```
90
```

```
91 bool lv_cpicker_get_color_mode_fixed(lv_obj_t * cpicker )
```

```
92 功能：长按中心时获取颜色模式是否更改
```

```
93 返回：
```

```
94 长按无法更改模式
```

```
95 形参：
```

```
96 cpicker: 指向 colorpicker 对象的指针
```

```
97
```

```
98
```

```
99 uint16_t lv_cpicker_get_hue(lv_obj_t * cpicker )
```

```
100 功能：获取颜色选择器的当前色相。
```

```
101 返回：
```

```
102 当前选择的色相
```

```
103 形参：
```

```
104 cpicker: 指向 colorpicker 对象的指针
```

```
105
```

```
106
```

```
107 uint8_t lv_cpicker_get_saturation(lv_obj_t * cpicker )
```

```
108 功能：获取选色器的当前饱和度。
```

```
109 返回：
```

```
110 当前选择的饱和度
```

```
111 形参：
```

```
112 cpicker: 指向 colorpicker 对象的指针
```

```
113
```

```
114
```

```
115 uint8_t lv_cpicker_get_value(lv_obj_t * cpicker )
```

```
116 功能：获取颜色选择器的当前色相。
```

```
117 返回：
```

```
118 当前选择值
```

```
119 形参：
```

```
120 cpicker: 指向 colorpicker 对象的指针
```

```
121
```

```
122
```

```
123 lv_color_hsv_t lv_cpicker_get_hsv(lv_obj_t * cpicker )
```

```
124 功能：获取选色器的当前选定的 hsv。
```

```
125 返回：
```

(下页继续)

(续上页)

```
126  当前选择的 hsv
127  形参:
128  cpicker: 指向 colorpicker 对象的指针
129
130
131  lv_color_t lv_cpicker_get_color(lv_obj_t * cpicker )
132  功能: 获取颜色选择器的当前选定颜色。
133  返回:
134  当前选择的颜色
135  形参:
136  cpicker: 指向 colorpicker 对象的指针
137
138
139  bool lv_cpicker_get_knob_colored(lv_obj_t * cpicker )
140  功能: 旋钮是否着色为当前颜色
141  返回:
142  true: 为旋钮着色; false: 不给旋钮上色
143  形参:
144  cpicker: 指向颜色选择器对象的指针
```

下拉列表 (lv_dropdown)

33.1 概述

下拉列表允许用户从列表中选择一个值。

下拉列表默认情况下处于关闭状态，并显示单个值或预定义的文本。激活后（通过单击下拉列表），将创建一个列表，用户可以从中选择一个选项。当用户选择新值时，该列表将被删除。

33.2 小部件和样式

调用下拉列表的主要部分，LV_DROPDOWN_PART_MAIN 它是一个简单的 `lv_obj` 对象。它使用所有典型的背景属性。按下，聚焦，编辑等阶梯也照常应用。

单击主对象时创建的列表是 `Page`。它的背景部分可以被引用，LV_DROPDOWN_PART_LIST 并为矩形本身使用所有典型的背景属性，并为选项使用文本属性。要调整选项之间的间距，请使用 `text_line_space` 样式属性。填充值可用于在边缘上留出一些空间。

页面的可滚动部分被隐藏，其样式始终为空（透明，无填充）。

滚动条可以被引用 LV_DROPDOWN_PART_SCROLLBAR 并使用所有典型的背景属性。

可以 LV_DROPDOWN_PART_SELECTED 使用所有典型的背景属性引用并使用所选的选项。它将以其默认状态在所选项上绘制一个矩形，并在按下状态下在被按下的选项上绘制一个矩形。

33.3 用法

33.3.1 设定选项

选项作为带有 `lv_dropdown_set_options(dropdown, options)` 的字符串传递到下拉列表。选项应用 `\n` 分隔。例如: `"First\nSecond\nThird"`。该字符串将保存在下拉列表中, 因此也可以保存在本地变量中。

`lv_dropdown_add_option(dropdown, "New option", pos)` 函数向 `pos` 索引插入一个新选项。

为了节省内存, 还可以使用 `lv_dropdown_set_static_options(dropdown, options)` 从静态 (常量) 字符串设置选项。在这种情况下, 当存在下拉列表且不能使用 `lv_dropdown_add_option` 时, `options` 字符串应处于活动状态

可以使用 `lv_dropdown_set_selected(dropdown, id)` 手动选择一个选项, 其中 `id` 是选项的索引。

33.3.2 获取选择的选项

使用获取当前选择的选项 `lv_dropdown_get_selected(dropdown)`。它将返回所选选项的索引。

`lv_dropdown_get_selected_str(dropdown, buf, buf_size)` 将所选选项的名称复制到 `buf`。

33.3.3 方向

该列表可以在任何一侧创建。默认值 `LV_DROPDOWN_DOWN` 可以通过功能进行修改。

`lv_dropdown_set_dir(dropdown, LV_DROPDOWN_DIR_LEFT/RIGHT/UP/DOWN)`

如果列表垂直于屏幕之外, 它将与边缘对齐。

33.3.4 符号

可以使用 `lv_dropdown_set_symbol(dropdown, LV_SYMBOL_...)` 将符号 (通常是箭头) 添加到下拉列表中

如果下拉列表的方向为 `LV_DROPDOWN_DIR_LEFT`, 则该符号将显示在左侧, 否则显示在右侧。

33.3.5 最大高度

可以通过 `lv_dropdown_set_max_height(dropdown, height)` 设置下拉列表的最大高度。默认情况下, 它设置为 3/4 垂直分辨率。

33.3.6 显示所选

主要部分可以显示所选选项或静态文本。可以使用 `lv_dropdown_set_show_selected(sropdown, true/false)` 进行控制。

可以使用 `lv_dropdown_set_text(dropdown, "Text")` 设置静态文本。仅保存文本指针。

如果也不想突出显示所选选项，则可以将自定义透明样式用于 `LV_DROPDOWN_PART_SELECTED`。

33.3.7 动画时间

下拉列表的打开/关闭动画时间由 `lv_dropdown_set_anim_time(ddlist, anim_time)` 调整。动画时间为零表示没有动画。

33.3.8 手动打开/关闭

要手动打开或关闭下拉列表，可以使用 `lv_dropdown_open/close(dropdown, LV_ANIM_ON/OFF)` 功能。

33.4 事件

除了 通用事件 外，下拉列表还发送以下 特殊事件：

- **LV_EVENT_VALUE_CHANGED** - 选择新选项时发送。

了解有关 事件 的更多信息。

33.5 按键

以下按键由按钮处理：

- **LV_KEY_RIGHT/DOWN** - 选择下一个选项。
- **LV_KEY_LEFT/UP** - 选择上一个选项。
- **LV_KEY_ENTER** - 应用选定的选项（发送 `LV_EVENT_VALUE_CHANGED` 事件并关闭下拉列表）。

进一步了解 按键。

33.6 范例

33.6.1 简单的下拉列表

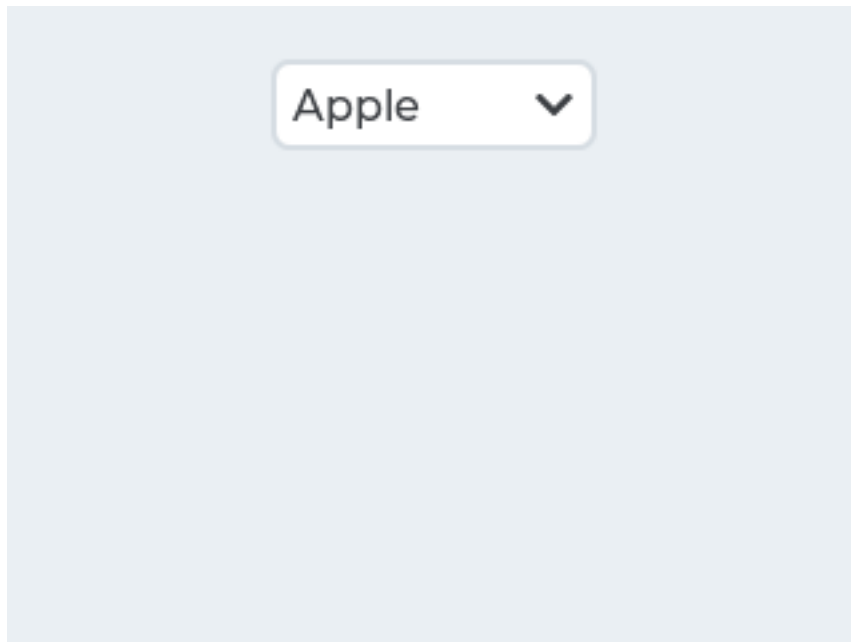


图 1: 简单的下拉列表

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_DROPDOWN
4
5
6  static void event_handler(lv_obj_t * obj, lv_event_t event)
7  {
8      if(event == LV_EVENT_VALUE_CHANGED) {
9          char buf[32];
10         lv_dropdown_get_selected_str(obj, buf, sizeof(buf));
11         printf("Option: %s\n", buf);
12     }
13 }
14
15 void lv_ex_dropdown_1(void)
16 {
17
18     /*Create a normal drop down list*/
```

(下页继续)

(续上页)

```
19     lv_obj_t * ddlist = lv_dropdown_create(lv_scr_act(), NULL);
20     lv_dropdown_set_options(ddlist, "Apple\n"
21                                "Banana\n"
22                                "Orange\n"
23                                "Melon\n"
24                                "Grape\n"
25                                "Raspberry");
26
27     lv_obj_align(ddlist, NULL, LV_ALIGN_IN_TOP_MID, 0, 20);
28     lv_obj_set_event_cb(ddlist, event_handler);
29 }
30
31 #endif
```

33.6.2 删除“上”列表

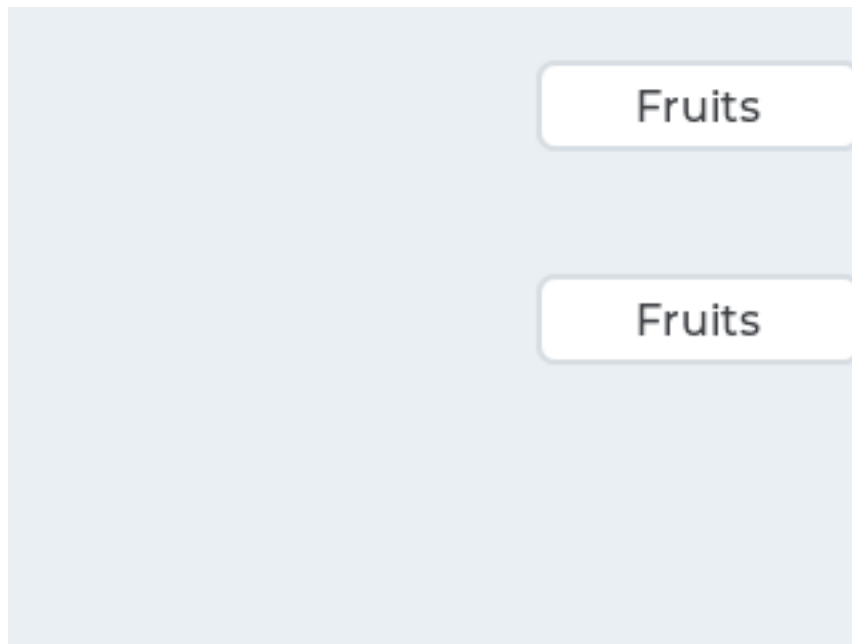


图 2: 删除“上”列表

上述效果的示例代码:

```
1     #include "../lv_examples.h"
2     #include <stdio.h>
3     #if LV_USE_DROPDOWN
4
5
```

(下页继续)

(续上页)

```
6  /**
7   * Create a drop LEFT menu
8   */
9  void lv_ex_dropdown_2(void)
10 {
11     /*Create a drop down list*/
12     lv_obj_t * ddlist = lv_dropdown_create(lv_scr_act(), NULL);
13     lv_dropdown_set_options(ddlist, "Apple\n"
14                                 "Banana\n"
15                                 "Orange\n"
16                                 "Melon\n"
17                                 "Grape\n"
18                                 "Raspberry");
19
20     lv_dropdown_set_dir(ddlist, LV_DROPDOWN_DIR_LEFT);
21     lv_dropdown_set_symbol(ddlist, NULL);
22     lv_dropdown_set_show_selected(ddlist, false);
23     lv_dropdown_set_text(ddlist, "Fruits");
24     /*It will be called automatically when the size changes*/
25     lv_obj_align(ddlist, NULL, LV_ALIGN_IN_TOP_RIGHT, 0, 20);
26
27     /*Copy the drop LEFT list*/
28     ddlist = lv_dropdown_create(lv_scr_act(), ddlist);
29     lv_obj_align(ddlist, NULL, LV_ALIGN_IN_TOP_RIGHT, 0, 100);
30 }
31
32 #endif
```

33.7 相关 API

TODO

34.1 概述

量规是一种带有刻度标签和一根或多根针的仪表。

34.2 小部件和样式

量规的主要部分称为 `LV_GAUGE_PART_MAIN`。它使用典型的背景样式属性绘制背景，并使用线和比例样式属性绘制“较小”比例线。它还使用 `text` 属性设置比例标签的样式。`pad_inner` 用于设置刻度线和刻度标签之间的空间。

`LV_GAUGE_PART_MAJOR` 是一个虚拟小部件，它使用 `line` 和 `scale` 样式属性描述了主要的比例尺线（添加了标签）。

`LV_GAUGE_PART_NEEDLE` 也是虚拟小部件，它通过线型属性来描述针。的大小和典型的背景属性用于描述在所述针（多个）的枢转点的矩形（或圆形）。`pad_inner` 用于使针比刻度线的外半径小。

34.3 用法

34.3.1 设定值和针

量规可以显示多于一根针。使用 `lv_gauge_set_needle_count(gauge, needle_num, color_array)` 函数设置针数和每根针具有颜色的数组。数组必须是静态或全局变量，因为仅存储其指针。

可以使用 `lv_gauge_set_value(gauge, needle_id, value)` 来设置针的值。

34.3.2 规模

可以使用 `lv_gauge_set_scale(gauge, angle, line_num, label_cnt)` 函数来调整刻度角度以及刻度线和标签的数量。默认设置为 220 度，6 个比例标签和 21 条线。

量表的刻度可以偏移。可以通过 `lv_gauge_set_angle_offset(gauge, angle)` 进行调整。

34.3.3 范围

量规的范围可以通过 `lv_gauge_set_range(gauge, min, max)` 指定。默认范围是 0..100。

34.3.4 针图

图像也可用作针。图像应指向右侧（如 ==>）。要设置图像，请使用 `lv_gauge_set_needle_img(gauge1, &img, pivot_x, pivot_y)`。ivot_x 和 pivot_y 是旋转中心距左上角的偏移量。图像将使用来自 LV_GAUGE_PART_NEEDLE 中的样式的 `image_recolor_opa` 强度重新着色为针的颜色。

34.3.5 临界值

要设置临界值，请使用 `lv_gauge_set_critical_value(gauge, value)`。此值之后，比例尺颜色将更改为 `scale_end_color`。默认临界值为 80。

34.4 事件

仅 通用事件 是按对象类型发送的。

了解有关 事件 的更多内容。

34.5 按键

对象类型不处理任何键。

了解有关 [按键](#) 的更多内容。

34.6 范例

34.6.1 简易量规

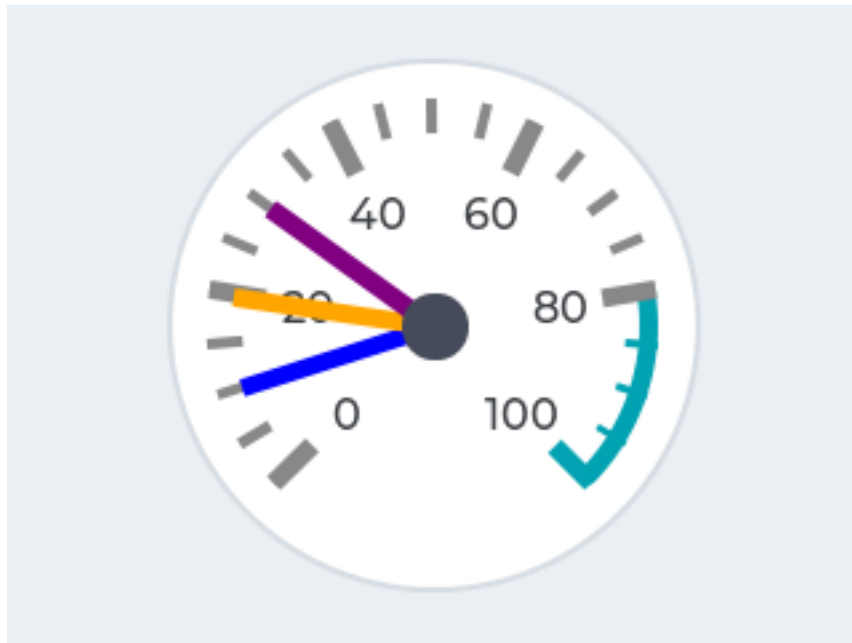


图 1: 简单按钮矩阵示例

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2  #if LV_USE_GAUGE
3
4
5  void lv_ex_gauge_1(void)
6  {
7      /*Describe the color for the needles*/
8      static lv_color_t needle_colors[3];
9      needle_colors[0] = LV_COLOR_BLUE;
10     needle_colors[1] = LV_COLOR_ORANGE;
11     needle_colors[2] = LV_COLOR_PURPLE;
```

(下页继续)

(续上页)

```
12
13     /*Create a gauge*/
14     lv_obj_t * gauge1 = lv_gauge_create(lv_scr_act(), NULL);
15     lv_gauge_set_needle_count(gauge1, 3, needle_colors);
16     lv_obj_set_size(gauge1, 200, 200);
17     lv_obj_align(gauge1, NULL, LV_ALIGN_CENTER, 0, 0);
18
19     /*Set the values*/
20     lv_gauge_set_value(gauge1, 0, 10);
21     lv_gauge_set_value(gauge1, 1, 20);
22     lv_gauge_set_value(gauge1, 2, 30);
23 }
24
25 #endif
```

34.6.2 自定义量规指针样式

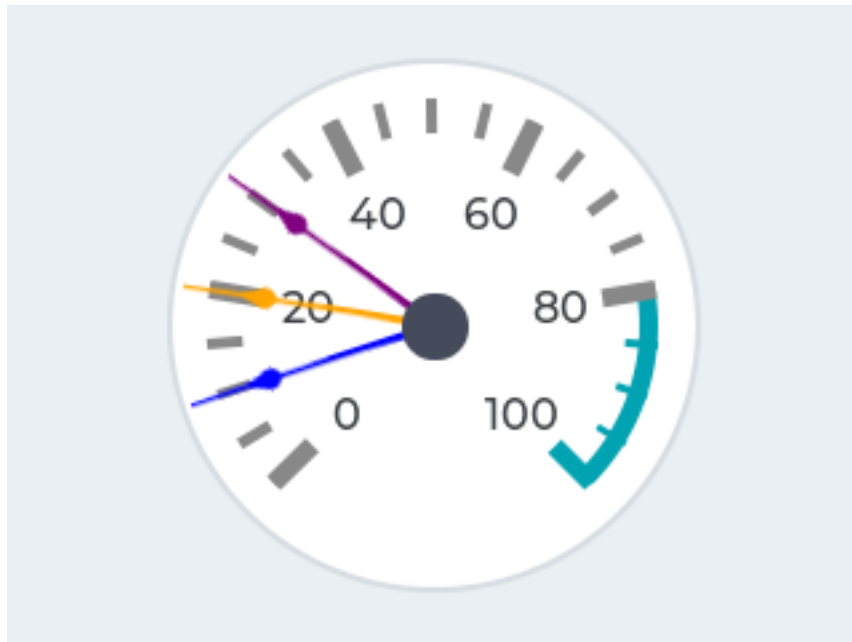


图 2: 自定义量规指针样式

上述效果的示例代码:

```
1     #include "../../lv_examples.h"
2     #if LV_USE_GAUGE
3
4
```

(下页继续)

(续上页)

```
5  void lv_ex_gauge_2(void)
6  {
7      /*Describe the color for the needles*/
8      static lv_color_t needle_colors[3];
9      needle_colors[0] = LV_COLOR_BLUE;
10     needle_colors[1] = LV_COLOR_ORANGE;
11     needle_colors[2] = LV_COLOR_PURPLE;
12
13     LV_IMG_DECLARE(img_hand);
14
15     /*Create a gauge*/
16     lv_obj_t * gauge1 = lv_gauge_create(lv_scr_act(), NULL);
17     lv_gauge_set_needle_count(gauge1, 3, needle_colors);
18     lv_obj_set_size(gauge1, 200, 200);
19     lv_obj_align(gauge1, NULL, LV_ALIGN_CENTER, 0, 0);
20     lv_gauge_set_needle_img(gauge1, &img_hand, 4, 4);
21     /*Allow recoloring of the images according to the needles' color*/
22     lv_obj_set_style_local_image_recolor_opa(gauge1, LV_GAUGE_PART_NEEDLE, LV_
↪STATE_DEFAULT, LV_OPA_COVER);
23
24     /*Set the values*/
25     lv_gauge_set_value(gauge1, 0, 10);
26     lv_gauge_set_value(gauge1, 1, 20);
27     lv_gauge_set_value(gauge1, 2, 30);
28 }
29
30 #endif
```

34.7 相关 API

TODO

35.1 概述

图像是从 **Flash**（作为数组）或从外部作为文件显示的基本对象。图像也可以显示符号（`LV_SYMBOL_...`）。使用 **图像解码器** 接口，也可以支持自定义图像格式。

35.2 零件和样式

图像只有一个称为 `LV_IMG_PART_MAIN` 的主要部分，该部分使用典型的背景样式属性绘制背景矩形和图像属性。填充值用于使背景实际变大。（它不会更改图像的实际大小，但仅在绘图期间应用大小修改）

35.3 用法

35.3.1 图片来源

为了提供最大的灵活性，图像的来源可以是：

- 代码中的变量（带有像素的 **C** 数组）。
- 外部存储的文件（例如 **SD** 卡上的文件）。
- **符号** 文字。

要设置图像的来源，调用 `lv_img_set_src(img, src)`。

要从 PNG, JPG 或 BMP 图像生成像素阵列，请使用 [在线图像转换工具](#)，并使用其指针设置转换后的图像：`lv_img_set_src(img1, &converted_img_var)`；要使变量在 C 文件中可见，需要使用 `LV_IMG_DECLARE(converted_img_var)` 进行声明。

要使用 **外部文件**，还需要使用在线转换器工具转换图像文件，但是现在应该选择二进制输出格式。还需要使用 LVGL 的文件系统模块，并为基本文件操作注册具有某些功能的驱动程序。进入文件系统以了解更多信息。要设置来自文件的图像，请使用 `lv_img_set_src(img, "S:folder1/my_img.bin")`。

可以类似于“**标签**”设置符号。在这种情况下，图像将根据样式中指定的字体呈现为文本。它可以使用轻量级的单色“字母”代替实际图像。可以设置符号，例如 `lv_img_set_src(img1, LV_SYMBOL_OK)`。

35.3.2 标签作为图片

图像和标签有时用于传达相同的内容。例如，描述按钮的作用。因此，图像和标签可以互换。为了处理这些图像，甚至可以使用 `LV_SYMBOL_DUMMY` 作为文本的前缀来显示文本。例如，`lv_img_set_src(img, LV_SYMBOL_DUMMY "Some text")`。

35.3.3 透明度

内部（可变）和外部图像支持 2 种透明度处理方法：

- **Chrome keying** - 具有 `LV_COLOR_TRANSP` (`lv_conf.h`) 颜色的像素将是透明的。
- **Alpha byte** - 一个 alpha 字节被添加到每个像素。

35.3.4 调色板和 Alpha 指数

除了本色（RGB）颜色格式外，还支持以下格式：

- **Indexed** - 索引，图像具有调色板。
- **Alpha indexed** - Alpha 索引，仅存储 Alpha 值。

可以在字体转换器中选择这些选项。要了解有关颜色格式的更多信息，请阅读 [图像 \(images\)](#) 部分。

35.3.5 重新着色

根据像素的亮度，可以在运行时将图像重新着色为任何颜色。在不存储同一图像的更多版本的情况下，显示图像的不同状态（选中，未激活，按下等）非常有用。可以通过在 `LV_OPA_TRANSP`（不重新着色，值：0）和 `LV_OPA_COVER`（完全重新着色，值：255）之间设置 `img.intense` 来启用该样式。默认值为 `LV_OPA_TRANSP`，因此此功能被禁用。

35.3.6 自动调整尺寸

调用 `lv_img_set_auto_size(image, true)` 函数，将设置图像对象的大小自动设置为图像源的宽度和高度。如果启用了自动调整大小，则在设置新文件时，对象大小将自动更改。以后，可以手动修改大小。如果图像不是屏幕，默认情况下将启用自动调整大小。

35.3.7 镶嵌

使用 `lv_img_set_offset_x(img, x_ofs)` 和 `lv_img_set_offset_y(img, y_ofs)`，可以向显示的图像添加一些偏移。如果对象尺寸小于图像源尺寸，则很有用。使用 `offset` 参数，可以通过对 `x` 或 `y` 偏移量进行动画处理来创建纹理图集或“运行中的图像”效果。

35.4 转换

使用 `lv_img_set_zoom(img, factor)` 图像将被缩放。将 `factor` 设置为 256 或 `LV_IMG_ZOOM_NONE` 以禁用缩放。较大的值将放大图像（例如 512 倍），较小的值将缩小图像（例如 128 倍）。分数刻度也适用。例如：281 为 10% 放大。

要旋转图像，请使用 `lv_img_set_angle(img, angle)`。角度精度为 0.1 度，因此对于 45.8° 设置 458。

默认情况下，旋转的枢轴点是图像的中心。可以使用 `lv_img_set_pivot(img, pivot_x, pivot_y)` 进行更改。0;0 是左上角。

可以使用 `lv_img_set_antialias(img, true/false)` 调整转换的质量。启用抗锯齿功能后，转换的质量更高，但速度较慢。

转换需要整个图像可用。因此，可以转换索引图像（`LV_IMG_CF_INDEXED...`），仅 `alpha` 图像（`LV_IMG_CF_ALPHA...`）或文件中的图像。换句话说，转换仅适用于存储为 C 数组的真彩色图像，或者自定义图像解码器返回整个图像。

注意，图像对象的真实坐标在变换期间不会改变。即 `lv_obj_get_width/height/x/y()` 将返回原始的非缩放坐标。

35.5 旋转

图像可以旋转

35.6 事件

默认情况下，禁用图像对象的单击，仅发送与非输入设备相关的常规事件。如果要捕获图像对象的所有一般事件，则应使用以下命令启用其单击：`lv_obj_set_click(img, true)`

了解有关 [事件](#) 的更多内容。

35.7 按键

对象类型不处理任何键。

了解有关 [按键](#) 的更多内容。

35.8 范例

35.8.1 图片来自于数组



图 1: 展示的图片来自于变量

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2  #if LV_USE_IMG
3
```

(下页继续)

(续上页)

```
4  /* Find the image here: https://github.com/lvgl/lv\_examples/tree/master/assets */
5  LV_IMG_DECLARE(img_cogwheel_argb);
6
7  void lv_ex_img_1(void)
8  {
9      lv_obj_t * img1 = lv_img_create(lv_scr_act(), NULL);
10     lv_img_set_src(img1, &img_cogwheel_argb);
11     lv_obj_align(img1, NULL, LV_ALIGN_CENTER, 0, -20);
12
13     lv_obj_t * img2 = lv_img_create(lv_scr_act(), NULL);
14     lv_img_set_src(img2, LV_SYMBOL_OK "Accept");
15     lv_obj_align(img2, img1, LV_ALIGN_OUT_BOTTOM_MID, 0, 20);
16 }
17
18 #endif
```

35.8.2 给图像重新着色

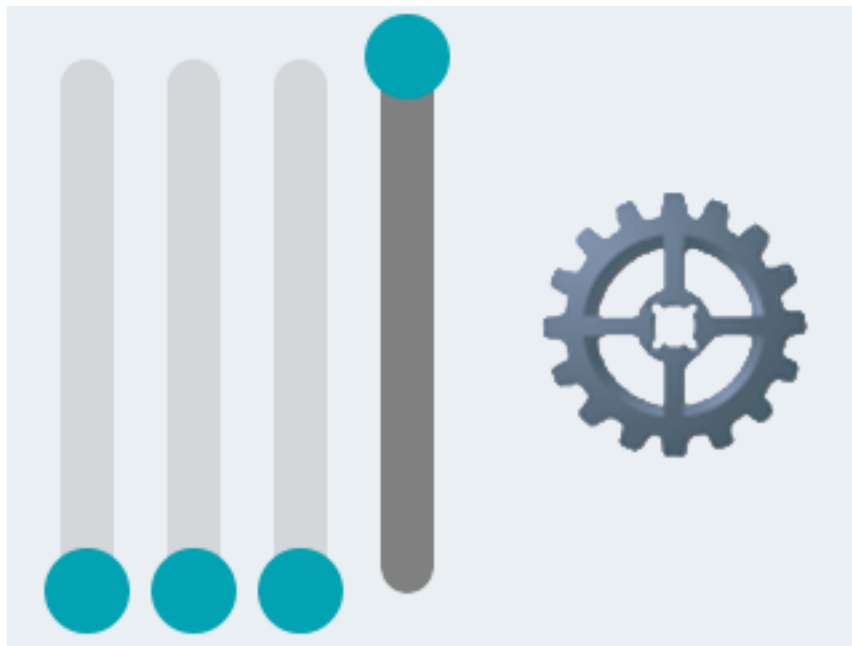


图 2: 给图像重新着色

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #if LV_USE_IMG
3
```

(下页继续)

(续上页)

```

4  #define SLIDER_WIDTH 20
5
6  static void create_sliders(void);
7  static void slider_event_cb(lv_obj_t * slider, lv_event_t event);
8
9  static lv_obj_t * red_slider, * green_slider, * blue_slider, * intense_slider;
10 static lv_obj_t * img1;
11 LV_IMG_DECLARE(img_cogwheel_argb);
12
13 void lv_ex_img_2(void)
14 {
15     /*Create 4 sliders to adjust RGB color and re-color intensity*/
16     create_sliders();
17
18     /* Now create the actual image */
19     img1 = lv_img_create(lv_scr_act(), NULL);
20     lv_img_set_src(img1, &img_cogwheel_argb);
21     lv_obj_align(img1, NULL, LV_ALIGN_IN_RIGHT_MID, -20, 0);
22 }
23
24 static void slider_event_cb(lv_obj_t * slider, lv_event_t event)
25 {
26     if(event == LV_EVENT_VALUE_CHANGED) {
27         /* Recolor the image based on the sliders' values */
28         lv_color_t color = lv_color_make(lv_slider_get_value(red_slider),
↪ lv_slider_get_value(green_slider), lv_slider_get_value(blue_slider));
29         lv_opa_t intense = lv_slider_get_value(intense_slider);
30         lv_obj_set_style_local_image_recolor_opa(img1, LV_IMG_PART_MAIN,
↪ LV_STATE_DEFAULT, intense);
31         lv_obj_set_style_local_image_recolor(img1, LV_IMG_PART_MAIN, LV_
↪ STATE_DEFAULT, color);
32     }
33 }
34
35 static void create_sliders(void)
36 {
37     /* Create a set of RGB sliders */
38     /* Use the red one as a base for all the settings */
39     red_slider = lv_slider_create(lv_scr_act(), NULL);
40     lv_slider_set_range(red_slider, 0, 255);
41     lv_obj_set_size(red_slider, SLIDER_WIDTH, 200); /* Be sure it's a
↪ vertical slider */
42     lv_obj_set_style_local_bg_color(red_slider, LV_SLIDER_PART_INDIC, LV_
↪ STATE_DEFAULT, LV_COLOR_RED);

```

(下页继续)

(续上页)

```
43     lv_obj_set_event_cb(red_slider, slider_event_cb);
44
45     /* Copy it for the other three sliders */
46     green_slider = lv_slider_create(lv_scr_act(), red_slider);
47     lv_obj_set_style_local_bg_color(green_slider, LV_SLIDER_PART_INDIC, LV_
↪STATE_DEFAULT, LV_COLOR_LIME);
48
49     blue_slider = lv_slider_create(lv_scr_act(), red_slider);
50     lv_obj_set_style_local_bg_color(blue_slider, LV_SLIDER_PART_INDIC, LV_
↪STATE_DEFAULT, LV_COLOR_BLUE);
51
52     intense_slider = lv_slider_create(lv_scr_act(), red_slider);
53     lv_obj_set_style_local_bg_color(intense_slider, LV_SLIDER_PART_INDIC, LV_
↪STATE_DEFAULT, LV_COLOR_GRAY);
54     lv_slider_set_value(intense_slider, 255, LV_ANIM_OFF);
55
56     lv_obj_align(red_slider, NULL, LV_ALIGN_IN_LEFT_MID, 20, 0);
57     lv_obj_align(green_slider, red_slider, LV_ALIGN_OUT_RIGHT_MID, 20, 0);
58     lv_obj_align(blue_slider, green_slider, LV_ALIGN_OUT_RIGHT_MID, 20, 0);
59     lv_obj_align(intense_slider, blue_slider, LV_ALIGN_OUT_RIGHT_MID, 20, 0);
60 }
61
62 #endif
```

35.9 相关 API

TODO

图片按钮 (lv_imgbtn)

36.1 概述

图像按钮与简单的“按钮”对象非常相似。唯一的区别是，它在每种状态下显示用户定义的图像，而不是绘制矩形。在阅读本节之前，请先阅读 [按钮](#) 一节以更好地理解本节内容。

36.2 零件和样式

图像按钮对象只有一个主要部分，称为 `LV_IMG_BTN_PART_MAIN`，在其中使用了所有图像样式属性。可以使用 `image_recolor` 和 `image_recolor_opa` 属性在每种状态下为图像重新着色。例如，如果按下该按钮可使图像变暗。

36.3 用法

36.3.1 图片来源

调用 `lv_imgbtn_set_src(imgbtn, LV_BTN_STATE_..., &img_src)` 将图像设置为一种状态。除了“图像”按钮不支持“符号”之外，图像源的工作原理与 [图像对象](#) 中所述的相同。

如果在 `lv_conf.h` 中启用了 `LV_IMGBTN_TILED`，则 `lv_imgbtn_set_src_tiled(imgbtn, LV_BTN_STATE_..., &img_src_left, &img_src_mid, &img_src_right)` 可用。使用平铺功能，

将重复中间图像以填充对象的宽度。因此，对于 LV_IMGBTN_TILED，可以使用 lv_obj_set_width() 设置图像按钮的宽度。但是，如果没有此选项，则宽度将始终与图像源的宽度相同。

36.3.2 按钮功能

类似于普通按钮 lv_imgbtn_set_checkable(imgbtn, true/false)，lv_imgbtn_toggle(imgbtn) 和 lv_imgbtn_set_state(imgbtn, LV_BTN_STATE_...) 也可以使用。

36.4 事件

除了通用事件，以下特殊事件也通过按钮发送：

- **LV_EVENT_VALUE_CHANGED** - 切换按钮时发送。

请注意，与通用输入设备相关的事件（如 LV_EVENT_PRESSED）也以非活动状态发送。您需要使用 lv_btn_get_state(btn) 检查状态，以忽略非活动按钮中的事件。

了解有关事件的更多内容。

36.5 按键

以下按键类型由按钮处理：

- **LV_KEY_RIGHT/UP** - 如果启用了切换，则进入切换状态。
- **LV_KEY_LEFT/DOWN** - 如果启用了切换，则进入非切换状态。

请注意，与往常一样，LV_KEY_ENTER 的状态会转换为 LV_EVENT_PRESSED / PRESSING / RELEASED 等。

了解有关按键的更多内容。

36.6 范例

36.6.1 简单的图像按钮

上述效果的示例代码：

```
1  #include "../examples.h"
2  #if LV_USE_IMGBTN
3
4  void lv_ex_imgbtn_1(void)
5  {
```

(下页继续)

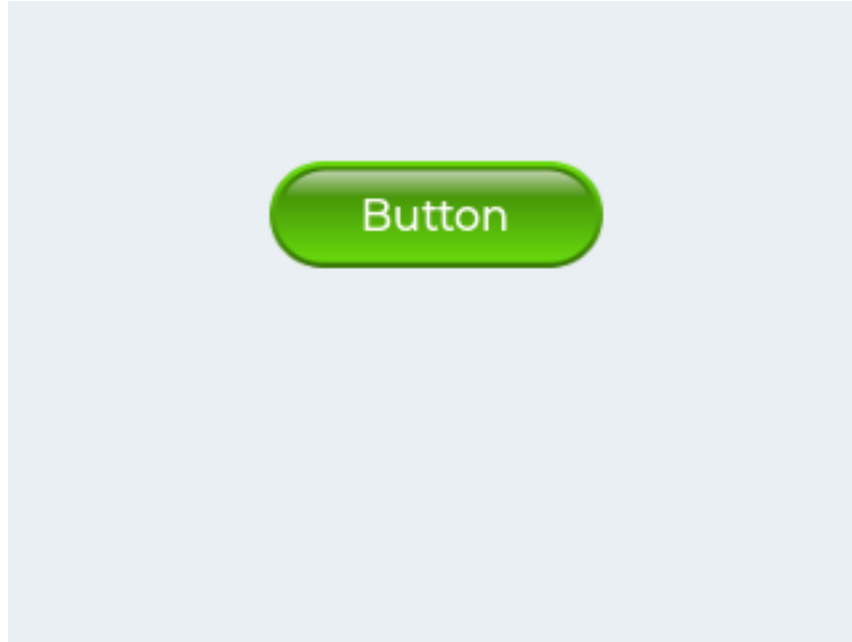


图 1: 简单的图像按钮

(续上页)

```

6      LV_IMG_DECLARE(imgbtn_green);
7      LV_IMG_DECLARE(imgbtn_blue);
8
9      /*Darken the button when pressed*/
10     static lv_style_t style;
11     lv_style_init(&style);
12     lv_style_set_image_recolor_opa(&style, LV_STATE_PRESSED, LV_OPA_30);
13     lv_style_set_image_recolor(&style, LV_STATE_PRESSED, LV_COLOR_BLACK);
14     lv_style_set_text_color(&style, LV_STATE_DEFAULT, LV_COLOR_WHITE);
15
16     /*Create an Image button*/
17     lv_obj_t * imgbtn1 = lv_imgbtn_create(lv_scr_act(), NULL);
18     lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_RELEASED, &imgbtn_green);
19     lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_PRESSED, &imgbtn_green);
20     lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_CHECKED_RELEASED, &imgbtn_blue);
21     lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_CHECKED_PRESSED, &imgbtn_blue);
22     lv_imgbtn_set_checkable(imgbtn1, true);
23     lv_obj_add_style(imgbtn1, LV_IMGBTN_PART_MAIN, &style);
24     lv_obj_align(imgbtn1, NULL, LV_ALIGN_CENTER, 0, -40);
25
26     /*Create a label on the Image button*/
27     lv_obj_t * label = lv_label_create(imgbtn1, NULL);
28     lv_label_set_text(label, "Button");

```

(下页继续)

(续上页)

```
29     }  
30  
31     #endif
```

36.7 相关 API

36.7.1 Typedefs

```
1     typedef uint16_t lv_btnmatrix_ctrl_t  
2     typedef uint8_t  lv_btnmatrix_part_t
```

36.7.2 enums

```
1     enum [anonymous ]  
2     键入以存储按钮控制位（禁用，隐藏等）。前 3 位用于存储宽度  
3     值：  
4  
5     enumerator LV_BTNMATRIX_CTRL_HIDDEN      /* 隐藏按钮 */  
6  
7  
8     enumerator LV_BTNMATRIX_CTRL_NO_REPEAT  /* 不要重复按此按钮。*/  
9  
10  
11     enumerator LV_BTNMATRIX_CTRL_DISABLED   /* 禁用此按钮。*/  
12  
13  
14     enumerator LV_BTNMATRIX_CTRL_CHECKABLE  /* 可以切换按钮。*/  
15  
16  
17     enumerator LV_BTNMATRIX_CTRL_CHECK_STATE /* 当前已切换按钮（例如，选中）。*/  
18  
19     /* 1: 在 CLICK 上发送 LV_EVENT_SELECTED, 0: 在 PRESS 上发送 LV_EVENT_SELECTED */  
20     enumerator LV_BTNMATRIX_CTRL_CLICK_TRIG  
21  
22  
23     enum [anonymous ]  
24     值：  
25     enumerator LV_BTNMATRIX_PART_BG  
26     enumerator LV_BTNMATRIX_PART_BTN
```

36.7.3 函数

```

1  LV_EXPORT_CONST_INT( LV_BTNMATRIX_BTN_NONE )
2  lv_obj_t * lv_btnmatrix_create(lv_obj_t * par, constlv_obj_t *copy)
3  功能：创建一个按钮矩阵对象
4  返回：
5  指向创建的按钮矩阵的指针
6  形参：
7  par: 指向对象的指针，它将是新按钮矩阵的父对象
8  copy: 指向按钮矩阵对象的指针，如果不为 NULL，则将从其复制新对象
9
10
11 void lv_btnmatrix_set_map(lv_obj_t * btnm, constchar *map[] )
12 功能：设置新地图。将根据地图创建/删除按钮。按钮矩阵保留对地图的引用，因此在矩阵有效期内不得释放。
↪char 串数组。
13 形参：
14 btnm: 指向按钮矩阵对象的指针
15 map: 指针的 char 串数组。最后一个 char 串必须是：“”。使用 “ \ n” 进行换行。
16
17
18 void lv_btnmatrix_set_ctrl_map(lv_obj_t * btnm, constlv_btnmatrix_ctrl_t ctrl_map↪
↪[] )
19 功能：设置按钮矩阵的按钮控制图（隐藏，禁用等）。控制图数组将被复制，因此在此函数返回：后可以将其释放。
20 形参：
21 btnm: 指向按钮矩阵对象的指针
22 ctrl_map: 指向 lv_btn_ctrl_t 控制字节数组的指针。数组的长度和元素的位置必须与单个按钮的数量和顺序匹配（即，不包括换行符）。地图的元素应类似于：ctrl_map[0] = width | LV_BTNMATRIX_CTRL_NO_
↪REPEAT | LV_BTNMATRIX_CTRL_TGL_ENABLE
23
24
25 void lv_btnmatrix_set_focused_btn(lv_obj_t * btnm, uint16_t id )
26 功能：设置焦点按钮，即在视觉上突出显示它。
27 形参：
28 btnm: 指向按钮矩阵对象的指针
29 id: 要聚焦的按钮的索引（LV_BTNMATRIX_BTN_NONE 以除去焦点）
30
31
32 void lv_btnmatrix_set_recolor(constlv_obj_t * btnm, bool en )
33 功能：启用按钮文本的重新着色
34 形参：
35 btnm: 指向按钮矩阵对象的指针
36 en: true: 启用重新着色；false: 禁用
37
38

```

(下页继续)

(续上页)

```

39 void lv_btnmatrix_set_btn_ctrl(lv_obj_t * btnm, uint16_t btn_id, lv_btnmatrix_ctrl_t
↪t ctrl )
40 功能：设置按钮矩阵的按钮属性
41 形参：
42 btnm: 指向按钮矩阵对象的指针
43 btn_id: 基于 0 的按钮索引进行修改。(不计算新行)
44
45
46 void lv_btnmatrix_clear_btn_ctrl(const lv_obj_t * btnm, uint16_t btn_id, lv_
↪btnmatrix_ctrl_t ctrl )
47 功能：清除按钮矩阵的按钮属性
48 形参：
49 btnm: 指向按钮矩阵对象的指针
50 btn_id: 基于 0 的按钮索引进行修改。(不计算新行)
51
52
53 void lv_btnmatrix_set_btn_ctrl_all(lv_obj_t * btnm, lv_btnmatrix_ctrl_t ctrl )
54 功能：设置按钮矩阵中所有按钮的属性
55 形参：
56 btnm: 指向按钮矩阵对象的指针
57 ctrl: 要从中设置的属性 lv_btnmatrix_ctrl_t。值可以进行或运算。
58
59
60 void lv_btnmatrix_clear_btn_ctrl_all(lv_obj_t * btnm, lv_btnmatrix_ctrl_t ctrl )
61 功能：清除按钮矩阵中所有按钮的属性
62 形参：
63 btnm: 指向按钮矩阵对象的指针
64 ctrl: 要从中设置的属性 lv_btnmatrix_ctrl_t。值可以进行或运算。
65 en: true: 设置属性; false: 清除属性
66
67
68 void lv_btnmatrix_set_btn_width(lv_obj_t * btnm, uint16_t btn_id, uint8_t width )
69 功能：设置单个按钮的相对宽度。该方法将导致矩阵再生并且是相对昂贵的操作。建议使用来指定初始宽度, lv_
↪btnmatrix_set_ctrl_map 并且此方法仅用于动态更改。
70 形参：
71 btnm: 指向按钮矩阵对象的指针
72 btn_id: 基于 0 的按钮索引进行修改。
73 width: 相对宽度 (与同一行中的按钮相比)。[1..7]
74
75
76 void lv_btnmatrix_set_one_check(lv_obj_t * btnm, bool one_chk )
77 功能：使按钮矩阵像选择器小部件一样 (一次只能切换一个按钮)。Checkable 必须在要使用 lv_
↪btnmatrix_set_ctrl 或选择的按钮上启用 lv_btnmatrix_set_btn_ctrl_all。

```

(下页继续)

(续上页)

形参:

btnm: 按钮矩阵对象

one_chk: 是否启用“一次检查”模式

```
void lv_btnmatrix_set_align(lv_obj_t * btnm, lv_label_align_t align )
```

功能: 设置地图文字的对齐方式 (向左, 向右或居中)

形参:

btnm: 指向 btnmatrix 对象的指针

align: LV_LABEL_ALIGN_LEFT, LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER

```
const char ** lv_btnmatrix_get_map_array(constlv_obj_t * btnm )
```

功能: 获取按钮矩阵的当前图

返回:

当前地图

形参:

btnm: 指向按钮矩阵对象的指针

```
bool lv_btnmatrix_get_recolor(constlv_obj_t * btnm )
```

功能: 检查按钮的文本是否可以使用重新着色

返回:

true: 启用文本重新着色; false: 禁用

形参:

btnm: 指向按钮矩阵对象的指针

```
uint16_t lv_btnmatrix_get_active_btn(constlv_obj_t * btnm )
```

功能: 获取用户最后按下的按钮的索引 (按下, 释放等)。在 event_cb 获取按钮的文本, 检查是否隐藏等方面很有用。

返回:

最后释放按钮的索引 (LV_BTNMATRIX_BTN_NONE: 如果未设置)

形参:

btnm: 指向按钮矩阵对象的指针

```
const char * lv_btnmatrix_get_active_btn_text(constlv_obj_t * btnm )
```

功能: 获取用户最后一次“激活”按钮的文本 (按下, 释放等) event_cb

返回:

最后释放的按钮的文本 (NULL: 如果未设置)

形参:

btnm: 指向按钮矩阵对象的指针

(下页继续)

(续上页)

```

120
121
122  uint16_t lv_btnmatrix_get_focused_btn(constlv_obj_t * btnm )
123  功能：获取焦点按钮的索引。
124  返回：
125  焦点按钮的索引 (LV_BTNMATRIX_BTN_NONE：如果未设置)
126  形参：
127  btnm：指向按钮矩阵对象的指针
128
129
130  const char * lv_btnmatrix_get_btn_text(constlv_obj_t * btnm, uint16_t btn_id )
131  功能：获取按钮的文字
132  返回：
133  btn_index` 按钮的文本
134  形参：
135  btnm：指向按钮矩阵对象的指针
136  btn_id：索引一个不计算换行符的按钮。(lv_btnmatrix_get_pressed / released 的返回：值)
137
138
139  bool lv_btnmatrix_get_btn_ctrl(lv_obj_t * btnm, uint16_t btn_id, lv_btnmatrix_ctrl_
140  ↪ t ctrl )
141  功能：获取按钮矩阵的按钮的控制值是启用还是禁用
142  返回：
143  true：禁用长按重复；false：长按重复启用
144  形参：
145  btnm：指向按钮矩阵对象的指针
146  btn_id：索引一个不计算换行符的按钮。(例如 lv_btnmatrix_get_pressed / released 的返回：值)
147  ctrl：要检查的控制值 (可以使用 ORed 值)
148
149
150  bool lv_btnmatrix_get_one_check(constlv_obj_t * btnm )
151  功能：查找是否启用了“一次切换”模式。
152  返回：
153  是否启用“一次切换”模式
154  形参：
155  btnm：按钮矩阵对象
156
157
158  lv_label_align_t lv_btnmatrix_get_align(constlv_obj_t * btnm )
159  功能：获取 align 属性
160  返回：
161  LV_LABEL_ALIGN_LEFT, LV_LABEL_ALIGN_RIGHT 或 LV_LABEL_ALIGN_CENTER
162  形参：
163  btnm：指向 btnmatrix 对象的指针

```

37.1 概述

Keyboard 对象是特殊的 按钮矩阵 (lv_imgbtn)，具有预定义的按键映射和其他功能，以实现虚拟键盘来编写文本。

37.2 零件和样式

类似于按钮 matrices，键盘包括 2 部分：

- LV_KEYBOARD_PART_BG 这是主要部分，并使用了所有典型的背景属性
- LV_KEYBOARD_PART_BTN 这是按钮的虚拟部分。它还使用所有典型的背景属性和文本属性。

37.3 用法

37.3.1 模式

键盘具有以下模式：

- LV_KEYBOARD_MODE_TEXT_LOWER - 显示小写字母
- LV_KEYBOARD_MODE_TEXT_UPPER - 显示大写字母
- LV_KEYBOARD_MODE_TEXT_SPECIAL - 显示特殊字符

- **LV_KEYBOARD_MODE_NUM** - 显示数字, +/-号和小数点。

文本模式 (TEXT) 的布局包含更改模式的按钮。

要手动设置模式, 请使用 `lv_keyboard_set_mode(kb, mode)`。默认更多是 `LV_KEYBOARD_MODE_TEXT_UPPER`。

37.3.2 分配文本区域

可以为键盘分配一个 **文本区域 (Text area)**, 以将单击的字符自动放在此处。要分配文本区域, 请使用 `lv_keyboard_set_textarea(kb, ta)`。

分配的文本区域的光标可以通过键盘进行管理: 分配了键盘后, 上一个文本区域的光标将被隐藏, 并且将显示新的文本区域。当通过“确定”或“关闭”按钮关闭键盘时, 光标也将被隐藏。游标管理器功能由“`lv_keyboard_set_cursor_manage(kb, true)`”启用。默认为不管理。

37.3.3 新的键盘布局

可以使用 `lv_keyboard_set_map(kb, map)` 和 `lv_keyboard_set_ctrl_map(kb, ctrl_map)` 为键盘指定新的地图 (布局)。了解有关 **按钮矩阵 (lv_imgbtn)** 的更多信息。记住, 使用以下关键字将具有与原始地图相同的效果:

- **LV_SYMBOL_OK** - 应用。
- ****LV_SYMBOL_CLOSE** - 关闭。
- ****LV_SYMBOL_BACKSPACE** - 从左侧删除。
- ****LV_SYMBOL_LEFT** - 向左移动光标。
- ****LV_SYMBOL_RIGHT** - 向右移动光标。
- **"ABC"** - 加载大写地图。
- **"abc"** - 加载小写地图。
- **"Enter"** - 换行。

37.4 事件

除了通用事件, 键盘还支持以下特殊事件:

- **LV_EVENT_VALUE_CHANGED** - 按下/释放按钮时发送, 或长按后重复发送。事件数据设置为按下/释放按钮的 ID。
- **LV_EVENT_APPLY** - OK 按钮被点击
- **LV_EVENT_CANCEL** - 关闭按钮被点击

键盘具有一个默认的事件处理程序回调，称为 `lv_keyboard_def_event_cb`。它处理按钮按下，地图更改，分配的文本区域等。可以将其完全替换为自定义事件处理程序，但是，可以在事件处理程序的开头调用 `lv_keyboard_def_event_cb` 来处理与以前相同的操作。

了解有关 [事件](#) 的更多内容。

37.5 按键

键盘可处理一下以下按键：

- **LV_KEY_RIGHT/UP/LEFT/RIGHT** - 要在按钮之间导航并选择一个。
- **LV_KEY_ENTER** - 按下/释放所选按钮。

了解有关 [按键](#) 的更多内容。

37.6 范例

37.6.1 带文字区域的键盘

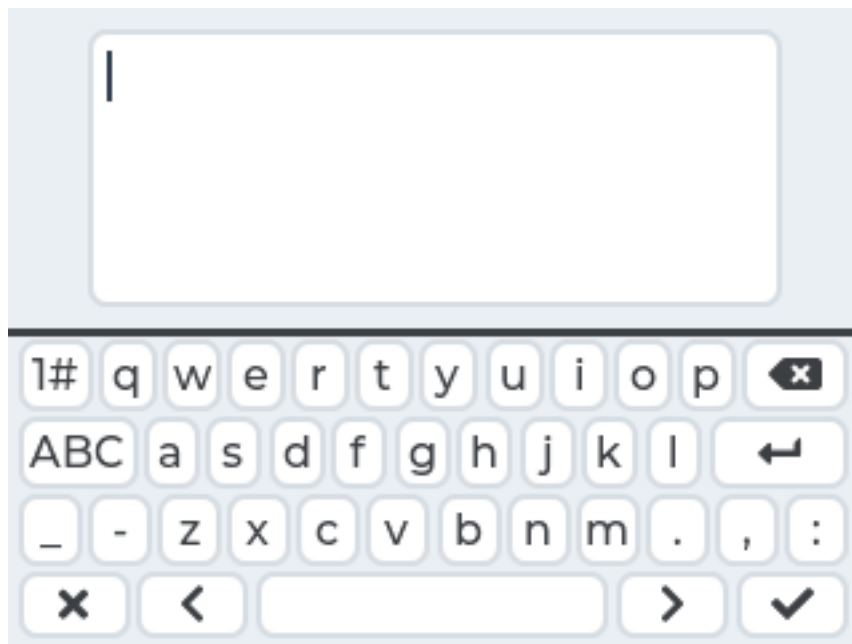


图 1: 带文字区域的键盘

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #if LV_USE_KEYBOARD
3
4  static lv_obj_t * kb;
5  static lv_obj_t * ta;
6
7
8  static void kb_event_cb(lv_obj_t * keyboard, lv_event_t e)
9  {
10     lv_keyboard_def_event_cb(kb, e);
11     if(e == LV_EVENT_CANCEL) {
12         lv_keyboard_set_textarea(kb, NULL);
13         lv_obj_del(kb);
14         kb = NULL;
15     }
16 }
17
18 static void kb_create(void)
19 {
20     kb = lv_keyboard_create(lv_scr_act(), NULL);
21     lv_keyboard_set_cursor_manage(kb, true);
22     lv_obj_set_event_cb(kb, kb_event_cb);
23     lv_keyboard_set_textarea(kb, ta);
24 }
25
26
27 static void ta_event_cb(lv_obj_t * ta_local, lv_event_t e)
28 {
29     if(e == LV_EVENT_CLICKED && kb == NULL) {
30         kb_create();
31     }
32 }
33
34 void lv_ex_keyboard_1(void)
35 {
36
37     /*Create a text area. The keyboard will write here*/
38     ta = lv_textarea_create(lv_scr_act(), NULL);
39     lv_obj_align(ta, NULL, LV_ALIGN_IN_TOP_MID, 0, LV_DPI / 16);
40     lv_obj_set_event_cb(ta, ta_event_cb);
41     lv_textarea_set_text(ta, "");
42     lv_coord_t max_h = LV_VER_RES / 2 - LV_DPI / 8;
43     if(lv_obj_get_height(ta) > max_h) lv_obj_set_height(ta, max_h);
44 }
```

(下页继续)

(续上页)

```
45         kb_create();  
46     }  
47     #endif
```

37.7 相关 API

TODO

38.1 概述

标签是用于显示文本的基本对象类型。

38.2 零件和样式

标签只有一个主要部分，称为 `LV_LABEL_PART_MAIN`。它使用所有典型的背景属性和文本属性。填充值可用于使文本的区域在相关方向上变小。

38.3 用法

38.3.1 设定文字

可以在运行时使用 `lv_label_set_text(label, "New text")` 在标签上设置文本。它将动态分配一个缓冲区，并将提供的字符串复制到该缓冲区中。因此，在该函数返回后，无需将传递给 `lv_label_set_text` 的文本保留在范围内。

使用 `lv_label_set_text_fmt(label, "Value: %d", 15)`，可以使用 `printf` 格式设置文本。

标签能够显示来自 0 终止的静态字符缓冲区的文本。为此，请使用 `lv_label_set_static_text(label, "Text")`。在这种情况下，文本不会存储在动态内存中，而是直接使用给定的缓冲区。这意味着数组不能是在函数退出时超出范围的局部变量。常数字符串可以安全地与 `lv_label_set_static_text` 一起使用

(除非与 `LV_LABEL_LONG_DOT` 一起使用, 因为它可以就地修改缓冲区), 因为它们存储在 **ROM** 存储器中, 该存储器始终可以访问。

也可以使用原始数组作为标签文本。数组不必以 `\0` 终止。在这种情况下, 文本将与 `lv_label_set_text` 一样保存到动态存储器中。要设置原始字符数组, 请使用 `lv_label_set_array_text(label, char_array, size)` 函数。

38.3.2 越线

换行符由标签对象自动处理。可以使用 `\n` 换行。例如: `"line1\nline2\n\nline4"`

38.3.3 长模式

默认情况下, 标签对象的宽度会自动扩展为文本大小。否则, 可以根据几种长模式策略来操纵文本:

- **LV_LABEL_LONG_EXPAND** - 将对象大小扩展为文本大小 (默认)
- **LV_LABEL_LONG_BREAK** - 保持对象宽度, 断开 (换行) 过长的线条并扩大对象高度
- **LV_LABEL_LONG_DOT** - 保持对象大小, 打断文本并在最后一行写点 (使用 `lv_label_set_static_text` 时不支持)
- **LV_LABEL_LONG_SCROLL** - 保持大小并来回滚动标签
- **LV_LABEL_LONG_SCROLL_CIRC** - 保持大小并循环滚动标签
- **LV_LABEL_LONG_CROP** - 保持大小并裁剪文本

可以使用 `lv_label_set_long_mode(label, LV_LABEL_LONG_...)` 指定长模式

重要的是要注意, 当创建标签并设置其文本时, 标签的大小已扩展为文本大小。除了默认的 `LV_LABEL_LONG_EXPAND`, 长模式 `lv_obj_set_width/height/size()` 无效。

因此, 需要更改长模式, 首先设置新的长模式, 然后使用 `lv_obj_set_width/height/size()` 设置大小。

另一个重要的注意事项是 `LV_LABEL_LONG_DOT` 在原地操纵文本缓冲区, 以便添加/删除点。当使用 `lv_label_set_text` 或 “`lv_label_set_array_text`” 时, 将分配一个单独的缓冲区, 并且该实现细节不会被注意。`lv_label_set_static_text` 并非如此! 如果打算使用 `LV_LABEL_LONG_DOT`, 则传递给 `lv_label_set_static_text` 的缓冲区必须可写。

38.3.4 文字对齐

文本的行可以使用 `lv_label_set_align(label, LV_LABEL_ALIGN_LEFT/RIGHT/CENTER)` 左右对齐。请注意，它将仅对齐线，而不对齐标签对象本身。

标签本身不支持垂直对齐；应该将标签放在更大的容器中，然后将整个标签对象对齐。

38.3.5 文字重新着色

在文本中，可以使用命令来重新着色部分文本。例如："`Write a #ff0000 red# word`"。可以通过 `lv_label_set_recolor()` 函数分别为每个标签启用此功能。

请注意，重新着色只能在一行中进行。因此，`\n` 不应在重新着色的文本中使用，或者用 `LV_LABEL_LONG_BREAK` 换行，否则，新行中的文本将不会重新着色。

38.3.6 很长的文字

Lvgl 通过保存一些额外的数据（~12 个字节）来加快绘图速度，可以有效地处理很长的字符（> 40k 个字符）。要启用此功能，请在 `lv_conf.h` 中设置 `LV_LABEL_LONG_TXT_HINT 1`

38.3.7 符号

标签可以在字母旁边显示符号（或单独显示）。阅读 [字体 \(font\)](#) 部分以了解有关符号的更多信息。

38.4 事件

仅 [通用事件](#) 是按对象类型发送的。

了解有关 [事件](#) 的更多内容。

38.5 按键

对象类型不处理任何键。

了解有关 [按键](#) 的更多内容。

38.6 范例

38.6.1 给标签重新着色和滚动

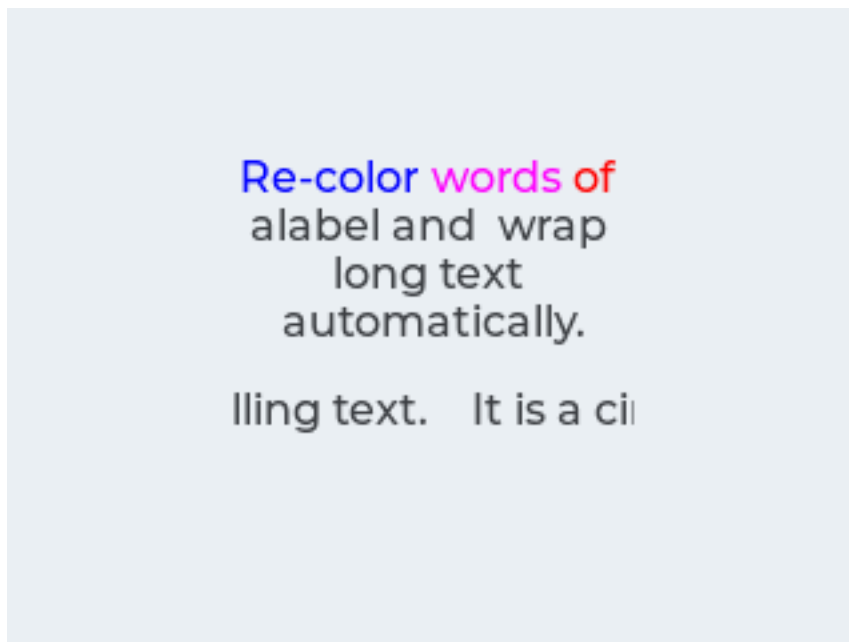


图 1: 给标签重新着色和滚动

上述效果的示例代码:

```

1  #include "../../lv_examples.h"
2  #if LV_USE_LABEL
3
4  void lv_ex_label_1(void)
5  {
6      lv_obj_t * label1 = lv_label_create(lv_scr_act(), NULL);
7      lv_label_set_long_mode(label1, LV_LABEL_LONG_BREAK);      /*Break the long
↪lines*/
8      lv_label_set_recolor(label1, true);                        /*Enable re-
↪coloring by commands in the text*/
9      lv_label_set_align(label1, LV_LABEL_ALIGN_CENTER);        /*Center aligned
↪lines*/
10     lv_label_set_text(label1, "#0000ff Re-color# #ff00ff words# #ff0000 of a#
↪label "
11                                     "and wrap long text
↪automatically.");
12     lv_obj_set_width(label1, 150);
13     lv_obj_align(label1, NULL, LV_ALIGN_CENTER, 0, -30);

```

(下页继续)

(续上页)

```

14
15     lv_obj_t * label2 = lv_label_create(lv_scr_act(), NULL);
16     lv_label_set_long_mode(label2, LV_LABEL_LONG_SROLL_CIRC);    /*Circular_
↪scroll*/
17     lv_obj_set_width(label2, 150);
18     lv_label_set_text(label2, "It is a circularly scrolling text. ");
19     lv_obj_align(label2, NULL, LV_ALIGN_CENTER, 0, 30);
20 }
21
22 #endif

```

38.6.2 文字阴影



图 2: 文字阴影

上述效果的示例代码:

```

1     #include "../../lv_examples.h"
2     #if LV_USE_LABEL
3
4     void lv_ex_label_2(void)
5     {
6         /* Create a style for the shadow*/
7         static lv_style_t label_shadow_style;
8         lv_style_init(&label_shadow_style);

```

(下页继续)

(续上页)

```

9      lv_style_set_text_opa(&label_shadow_style, LV_STATE_DEFAULT, LV_OPA_50);
10     lv_style_set_text_color(&label_shadow_style, LV_STATE_DEFAULT, LV_COLOR_
↪RED);
11
12     /*Create a label for the shadow first (it's in the background) */
13     lv_obj_t * shadow_label = lv_label_create(lv_scr_act(), NULL);
14     lv_obj_add_style(shadow_label, LV_LABEL_PART_MAIN, &label_shadow_style);
15
16     /* Create the main label */
17     lv_obj_t * main_label = lv_label_create(lv_scr_act(), NULL);
18     lv_label_set_text(main_label, "A simple method to create\n"
19                                     "shadows on
↪text\n"
20                                     "It even works
↪with\n\n"
21                                     "newlines
↪and spaces.");
22
23     /*Set the same text for the shadow label*/
24     lv_label_set_text(shadow_label, lv_label_get_text(main_label));
25
26     /* Position the main label */
27     lv_obj_align(main_label, NULL, LV_ALIGN_CENTER, 0, 0);
28
29     /* Shift the second label down and to the right by 2 pixel */
30     lv_obj_align(shadow_label, main_label, LV_ALIGN_IN_TOP_LEFT, 1, 1);
31 }
32
33 #endif

```

38.6.3 标签对齐

上述效果的示例代码：

```

1     #include "../lv_examples.h"
2     #if LV_USE_LABEL
3
4     static void text_changer(lv_task_t * t);
5
6     lv_obj_t * labels[3];
7
8     /**

```

(下页继续)

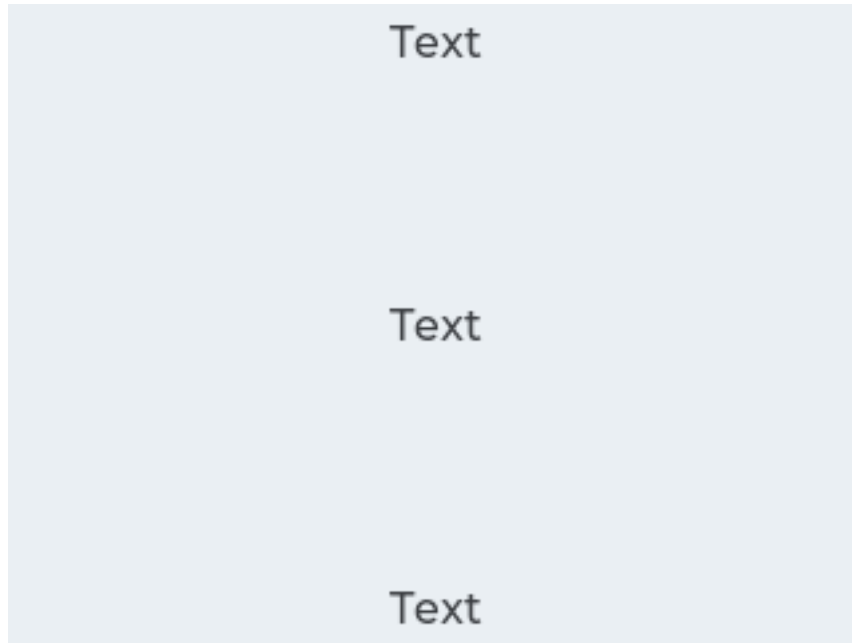


图 3: 标签对齐

(续上页)

```
9      * Create three labels to demonstrate the alignments.
10     */
11     void lv_ex_label_3(void)
12     {
13         /*`lv_label_set_align` is not required to align the object itself.
14         * It's used only when the text has multiple lines*/
15
16         /* Create a label on the top.
17         * No additional alignment so it will be the reference*/
18         labels[0] = lv_label_create(lv_scr_act(), NULL);
19         lv_obj_align(labels[0], NULL, LV_ALIGN_IN_TOP_MID, 0, 5);
20         lv_label_set_align(labels[0], LV_LABEL_ALIGN_CENTER);
21
22         /* Create a label in the middle.
23         * `lv_obj_align` will be called every time the text changes
24         * to keep the middle position */
25         labels[1] = lv_label_create(lv_scr_act(), NULL);
26         lv_obj_align(labels[1], NULL, LV_ALIGN_CENTER, 0, 0);
27         lv_label_set_align(labels[1], LV_LABEL_ALIGN_CENTER);
28
29         /* Create a label in the bottom.
30         * Enable auto realign. */
31         labels[2] = lv_label_create(lv_scr_act(), NULL);
```

(下页继续)

(续上页)

```
32     lv_obj_set_auto_realign(labels[2], true);
33     lv_obj_align(labels[2], NULL, LV_ALIGN_IN_BOTTOM_MID, 0, -5);
34     lv_label_set_align(labels[2], LV_LABEL_ALIGN_CENTER);
35
36     lv_task_t * t = lv_task_create(text_changer, 1000, LV_TASK_PRIO_MID,
↪NULL);
37     lv_task_ready(t);
38 }
39
40 static void text_changer(lv_task_t * t)
41 {
42     const char * texts[] = {"Text", "A very long text", "A text with\
↪multiple\nlines", NULL};
43     static uint8_t i = 0;
44
45     lv_label_set_text(labels[0], texts[i]);
46     lv_label_set_text(labels[1], texts[i]);
47     lv_label_set_text(labels[2], texts[i]);
48
49     /*Manually realign `labels[1]`*/
50     lv_obj_align(labels[1], NULL, LV_ALIGN_CENTER, 0, 0);
51
52     i++;
53     if(texts[i] == NULL) i = 0;
54 }
55
56 #endif
```

38.7 相关 API

TODO

39.1 概述

LED 是矩形（或圆形）对象。它的亮度可以调节。亮度降低时，LED 的颜色会变暗。

39.2 零件和样式

LED 只有一个主要部分，称为 `LV_LED_PART_MAIN`，它使用所有典型的背景样式属性。

39.3 用法

39.3.1 亮度

可以使用 `lv_led_set_bright(led, bright)` 设置它们的亮度。亮度应介于 0（最暗）和 255（最亮）之间。

39.3.2 切换

使用 `lv_led_on(led)` 和 `lv_led_off(led)` 将亮度设置为预定义的 ON 或 OFF 值。
`lv_led_toggle(led)` 在 ON 和 OFF 状态之间切换。

39.4 事件

仅支持 通用事件

了解有关 事件 的更多内容。

39.5 按键处理

对象类型不处理任何键。

了解有关 按键 的更多内容。

39.6 范例

39.6.1 自定义风格的 LED



图 1: 自定义风格的 LED

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #if LV_USE_LED
3
4  void lv_ex_led_1(void)
5  {
6      /*Create a LED and switch it OFF*/
7      lv_obj_t * led1 = lv_led_create(lv_scr_act(), NULL);
8      lv_obj_align(led1, NULL, LV_ALIGN_CENTER, -80, 0);
9      lv_led_off(led1);
10
11     /*Copy the previous LED and set a brightness*/
12     lv_obj_t * led2 = lv_led_create(lv_scr_act(), led1);
13     lv_obj_align(led2, NULL, LV_ALIGN_CENTER, 0, 0);
14     lv_led_set_bright(led2, 190);
15
16     /*Copy the previous LED and switch it ON*/
17     lv_obj_t * led3 = lv_led_create(lv_scr_act(), led1);
18     lv_obj_align(led3, NULL, LV_ALIGN_CENTER, 80, 0);
19     lv_led_on(led3);
20 }
21
22 #endif
```

39.7 相关 API

TODO

40.1 概述

Line 对象能够在一组点之间绘制直线。

40.2 零件和样式

生产线只有一个主要部分，称为 LV_LABEL_PART_MAIN。它使用所有线型属性。

40.3 用法

40.3.1 设置点

这些点必须存储在 `lv_point_t` 数组中，并通过 `lv_line_set_points(lines, point_array, point_cnt)` 函数传递给对象。

40.3.2 自动大小

可以根据其点自动设置线对象的大小。可以使用 `lv_line_set_auto_size(line, true)` 函数启用它。如果启用，则在设置点后，将根据点之间的最大 x 和 y 坐标更改对象的宽度和高度。默认情况下，自动尺寸已启用。

40.3.3 倒 y

通过默认，y == 0 点位于对象的顶部。在某些情况下可能是直觉的，因此可以使用 `lv_line_set_y_invert(line, true)` 反转 y 坐标。在这种情况下，y == 0 将是对象的底部。默认情况下，禁用 y 反转。

40.4 事件

仅支持 通用事件。

了解有关 事件 的更多内容。

40.5 按键处理

对象类型不处理任何键。

了解有关 按键 的更多内容。

40.6 范例

40.6.1 简单线

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2
3
4  void lv_ex_line_1(void)
5  {
6      /*Create an array for the points of the line*/
7      static lv_point_t line_points[] = { {5, 5}, {70, 70}, {120, 10}, {180, 60}
8      ↪, {240, 10} };
9
10     /*Create style*/
```

(下页继续)



图 1: 简单线

(续上页)

```
10     static lv_style_t style_line;
11     lv_style_init(&style_line);
12     lv_style_set_line_width(&style_line, LV_STATE_DEFAULT, 8);
13     lv_style_set_line_color(&style_line, LV_STATE_DEFAULT, LV_COLOR_BLUE);
14     lv_style_set_line_rounded(&style_line, LV_STATE_DEFAULT, true);
15
16     /*Create a line and apply the new style*/
17     lv_obj_t * line1;
18     line1 = lv_line_create(lv_scr_act(), NULL);
19     lv_line_set_points(line1, line_points, 5);      /*Set the points*/
20     lv_obj_add_style(line1, LV_LINE_PART_MAIN, &style_line);      /*Set the
↪points*/
21     lv_obj_align(line1, NULL, LV_ALIGN_CENTER, 0, 0);
22 }
23
24 #endif
```

40.7 相关 API

TODO

41.1 概述

列表是从背景 [页面 \(Page\)](#) 和其上的 [按钮 \(Buttons\)](#) 构建的。按钮包含可选的类似图标的‘[图像 \(Image\)](#)’_（也可以是符号）和‘[标签 \(Label\)](#)’_。当列表足够长时，可以滚动它。

41.2 零件和样式

列表与 [页面 \(Page\)](#) 具有相同的部分

- LV_LIST_PART_BG
- LV_LIST_PART_SCRL
- LV_LIST_PART_SCRLBAR
- LV_LIST_PART_EDGE_FLASH

有关详细信息，请参见 [页面 \(Page\)](#) 部分。

列表上的按钮被视为普通按钮，它们只有一个主要部分，称为 LV_BTN_PART_MAIN。

41.3 用法

41.3.1 添加按钮

可以使用 `lv_list_add_btn(list, &icon_img, "Text")` 或符号 `lv_list_add_btn(list, SYMBOL_EDIT, "Edit text")` 添加新的列表元素（按钮）。如果不想添加图像，请使用 `NULL` 作为图像源。该函数返回指向创建的按钮的指针，以允许进行进一步的配置。

按钮的宽度根据对象的宽度设置为最大。按钮的高度会根据内容自动调整。（内容高度 + `padding_top` + `padding_bottom`）。

标签以 `LV_LABEL_LONG_SCROLL_CIRC` 长模式创建，以自动循环滚动长标签。

`lv_list_get_btn_label(list_btn)` 和 `lv_list_get_btn_img(list_btn)` 可用于获取标签和列表按钮的图像。可以直接使用 `lv_list_get_btn_text(list_btn)` 来输入文本。

41.3.2 删除按钮

要删除列表元素，请使用 `lv_list_remove(list, btn_index)`。可以通过 `lv_list_get_btn_index(list, btn)` 获得 `btn_index`，其中 `btn` 是 `lv_list_add_btn()` 的返回值。

要清除列表（删除所有按钮），请使用 `lv_list_clean(list)`

41.3.3 手动导航

可以使用 `lv_list_up(list)` 和 `lv_list_down(list)` 在列表中手动导航。

可以使用 `lv_list_focus(btn, LV_ANIM_ON/OFF)` 直接关注按钮。

上/下/焦点移动的动画时间可以通过以下命令设置：`lv_list_set_anim_time(list, anim_time)`。动画时间为零表示不是动画。

41.3.4 布局

默认情况下，列表是垂直的。要获取水平列表，请使用 `lv_list_set_layout(list, LV_LAYOUT_ROW_MID)`。

41.3.5 边缘闪烁

当列表到达最高或最低位置时，可以显示类似圆圈的效果。`lv_list_set_edge_flash(list, true)` 启用此功能。

41.3.6 滚动传播

如果列表是在其他可滚动元素（例如 [页面 \(Page\)](#)）上创建的，并且列表无法进一步滚动，则滚动可以传播到父级。这样，滚动将在父级上继续。可以通过 `lv_list_set_scroll_propagation(list, true)` 启用它

41.4 事件

仅支持 [通用事件](#)。

了解有关 [事件](#) 的更多内容。

41.5 按键处理

列表处理以下按键：

- `LV_KEY_RIGHT/DOWN` 选择下一个按钮
- `LV_KEY_LEFT/UP` 选择上一个按钮

请注意，与往常一样，`LV_KEY_ENTER` 的状态会转换为 `LV_EVENT_PRESSED/PRESSING/RELEASED` 等。

所选按钮处于 `LV_BTN_STATE_PR/TG_PR` 状态。

要手动选择按钮，请使用 `lv_list_set_btn_selected(list, btn)`。当列表散焦并再次聚焦时，它将恢复最后选择的按钮。

了解有关 [按键](#) 的更多内容。

41.6 范例

41.6.1 简单的列表

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_LIST
4
```

(下页继续)

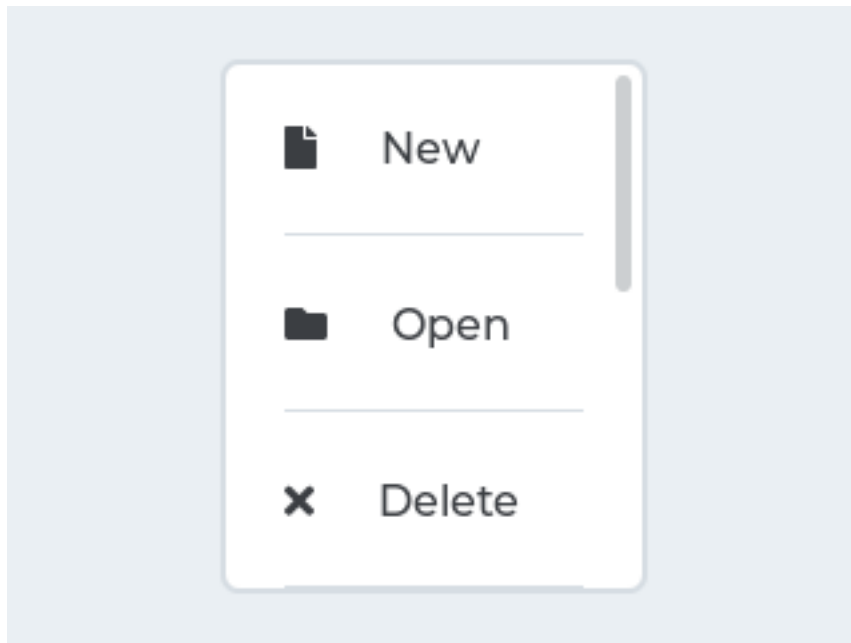


图 1: 简单的列表

(续上页)

```

5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
7      if(event == LV_EVENT_CLICKED) {
8          printf("Clicked: %s\n", lv_list_get_btn_text(obj));
9      }
10 }
11
12 void lv_ex_list_1(void)
13 {
14     /*Create a list*/
15     lv_obj_t * list1 = lv_list_create(lv_scr_act(), NULL);
16     lv_obj_set_size(list1, 160, 200);
17     lv_obj_align(list1, NULL, LV_ALIGN_CENTER, 0, 0);
18
19     /*Add buttons to the list*/
20     lv_obj_t * list_btn;
21
22     list_btn = lv_list_add_btn(list1, LV_SYMBOL_FILE, "New");
23     lv_obj_set_event_cb(list_btn, event_handler);
24
25     list_btn = lv_list_add_btn(list1, LV_SYMBOL_DIRECTORY, "Open");
26     lv_obj_set_event_cb(list_btn, event_handler);
27

```

(下页继续)

(续上页)

```
28     list_btn = lv_list_add_btn(list1, LV_SYMBOL_CLOSE, "Delete");
29     lv_obj_set_event_cb(list_btn, event_handler);
30
31     list_btn = lv_list_add_btn(list1, LV_SYMBOL_EDIT, "Edit");
32     lv_obj_set_event_cb(list_btn, event_handler);
33
34     list_btn = lv_list_add_btn(list1, LV_SYMBOL_SAVE, "Save");
35     lv_obj_set_event_cb(list_btn, event_handler);
36
37     list_btn = lv_list_add_btn(list1, LV_SYMBOL_BELL, "Notify");
38     lv_obj_set_event_cb(list_btn, event_handler);
39
40     list_btn = lv_list_add_btn(list1, LV_SYMBOL_BATTERY_FULL, "Battery");
41     lv_obj_set_event_cb(list_btn, event_handler);
42 }
43
44 #endif
```

41.7 相关 API

TODO

42.1 概述

线表对象由一些绘制比例的径向线组成。设置线表的值将按比例更改刻度线的颜色。

42.2 零件和样式

线表只有一个主要部分，称为 `LV_LINEMETER_PART_MAIN`。它使用所有典型的背景属性绘制矩形或圆形背景，并使用 `line` 和 `scale` 属性绘制比例线。活动行（与较小的值相关，即当前值）的颜色从 `line_color` 到 `scale_grad_color`。最后（当前值之后）的行设置为 `scale_end_color` color。

42.3 用法

42.3.1 设定数值

用 `lv_linemeter_set_value(linemeter, new_value)` 设置新值时，比例的比例部分将重新着色。

42.3.2 范围和角度

`lv_linemeter_set_range(linemeter, min, max)` 函数设置线表的范围。

可以通过 `lv_linemeter_set_scale(linemeter, angle, line_num)` 设置比例尺的角度和行数。默认角度为 240，默认行号为 31。

42.3.3 角度偏移

默认情况下，刻度角相对于 y 轴对称地解释。这导致“站立”线表。使用 `lv_linemeter_set_angle_offset` 可以添加缩放角度的偏移量。它可以用于例如将四分之一线表放到角落或将半线表放到右侧或左侧。

42.3.4 镜像

默认情况下，线表的线路是顺时针激活的。可以使用 `lv_linemeter_set_mirror(linemeter, true/false)` 进行更改。

42.4 事件

仅支持 通用事件。

了解有关 事件 的更多内容。

42.5 按键处理

对象类型不处理任何键。

了解有关 按键 的更多内容。

42.6 范例

42.6.1 简单的仪表

上述效果的示例代码：

```
1  #include "../examples.h"
2  #if LV_USE_LINEMETER
3
4  void lv_ex_linemeter_1(void)
5  {
```

(下页继续)

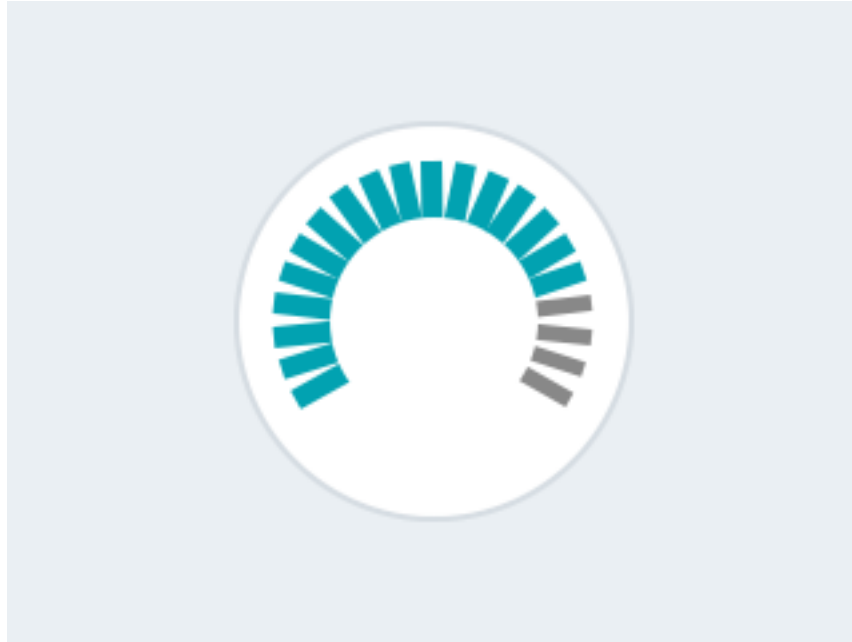


图 1: 简单的仪表

(续上页)

```
6      /*Create a line meter */
7      lv_obj_t * lmeter;
8      lmeter = lv_linemeter_create(lv_scr_act(), NULL);
9      lv_linemeter_set_range(lmeter, 0, 100);          /*Set the
↪range*/
10     lv_linemeter_set_value(lmeter, 80);              /*Set the
↪current value*/
11     lv_linemeter_set_scale(lmeter, 240, 21);        /*Set the angle
↪and number of lines*/
12     lv_obj_set_size(lmeter, 150, 150);
13     lv_obj_align(lmeter, NULL, LV_ALIGN_CENTER, 0, 0);
14 }
15
16 #endif
```

42.7 相关 API

TODO

43.1 概述

消息框充当弹出窗口。它们由背景容器，标签和按钮的 [按钮矩阵 \(lv_imgbtn\)](#) 构建而成。

文本将自动分成多行（具有 `LV_LABEL_LONG_MODE_BREAK`），高度将自动设置为包含文本和按钮（`LV_FIT_TIGHT` 垂直放置） -

43.2 零件和样式

消息框的主要部分称为 `LV_MSGBOX_PART_MAIN`，它使用所有典型的背景样式属性。使用填充会增加侧面的空间。`pad_inner` 将在文本和按钮之间添加空格。标签样式属性会影响文本样式。

按钮部分与 [按钮矩阵 \(lv_imgbtn\)](#) 的情况相同：

- `LV_MSGBOX_PART_BTN_BG` 按钮的背景
- `LV_MSGBOX_PART_BTN` 按钮

43.3 用法

43.3.1 设置文本

要设置文本，请使用 `lv_msgbox_set_text(msgbox, "My text")` 函数。不仅将保存文本指针，而且文本也可以位于局部变量中。

43.3.2 添加按钮

要添加按钮，请使用 `lv_msgbox_add_btns(msgbox, btn_str)` 函数。需要指定按钮的文本，例如 `const char * btn_str[] = {"Apply", "Close", ""}`。有关更多信息，请访问 [Button 矩阵文档](#)。仅当首次调用 `lv_msgbox_add_btns()` 时，才会创建 [按钮矩阵 \(lv_imgbtn\)](#)。

43.3.3 自动关闭

使用 `lv_msgbox_start_auto_close(mbox, delay)` 可以在动画延迟 (delay) 了几毫秒后自动关闭消息框。`lv_mbox_stop_auto_close(mbox)` 函数停止启动的自动关闭。

关闭动画的持续时间可以通过 `lv_mbox_set_anim_time(mbox, anim_time)` 设置。

43.4 事件

除了 [通用事件](#)，复选框还支持以下 [特殊事件](#)：

- **LV_EVENT_VALUE_CHANGED** 单击按钮时发送。事件数据设置为单击按钮的 ID。

消息框具有一个默认的事件回调，当单击按钮时，该事件回调将自行关闭。

了解有关 [事件](#) 的更多内容。

43.5 按键处理

消息框可处理以下按键：

- **LV_KEY_RIGHT/DOWN** 选择下一个按钮
- **LV_KEY_LEFT/TOP** 选择上一个按钮
- **LV_KEY_ENTER** 单击选定的按钮

了解有关 [按键](#) 的更多内容。

43.6 范例

43.6.1 简单消息框

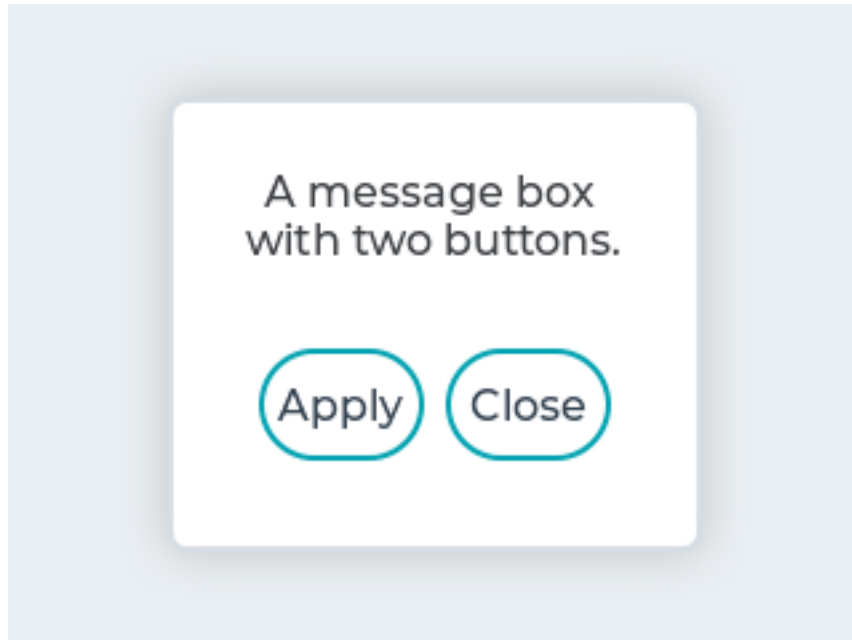


图 1: 简单消息框

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_MSGBOX
4
5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
7      if(event == LV_EVENT_VALUE_CHANGED) {
8          printf("Button: %s\n", lv_msgbox_get_active_btn_text(obj));
9      }
10 }
11
12 void lv_ex_msgbox_1(void)
13 {
14     static const char * btns[] ={"Apply", "Close", ""};
15
16     lv_obj_t * mbox1 = lv_msgbox_create(lv_scr_act(), NULL);
17     lv_msgbox_set_text(mbox1, "A message box with two buttons.");
18     lv_msgbox_add_btns(mbox1, btns);
19 }
```

(下页继续)

(续上页)

```
19     lv_obj_set_width(mbox1, 200);
20     lv_obj_set_event_cb(mbox1, event_handler);
21     lv_obj_align(mbox1, NULL, LV_ALIGN_CENTER, 0, 0); /*Align to the corner*/
22 }
23
24 #endif
```

43.6.2 模态

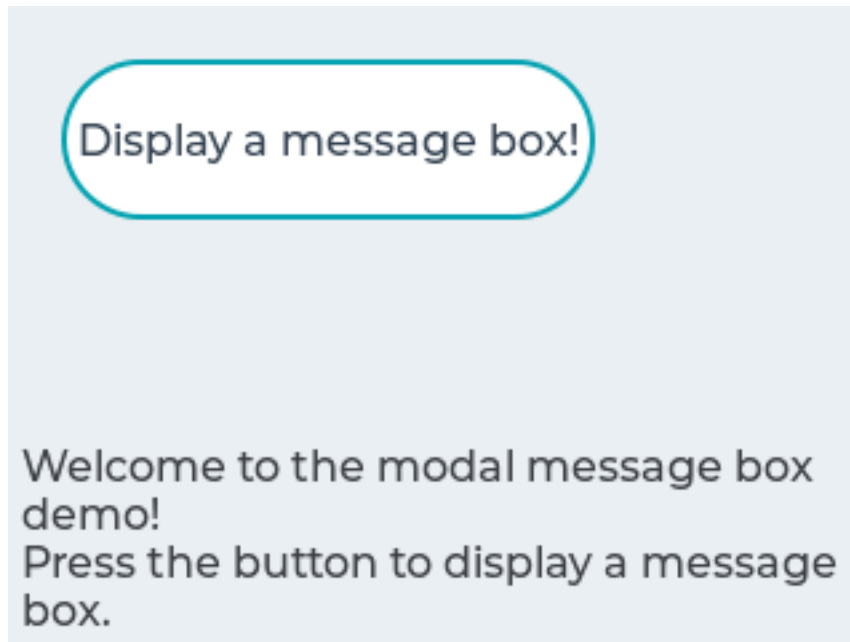


图 2: 模态

上述效果的示例代码:

```
1  #include ".././../lv_examples.h"
2  #if LV_USE_MSGBOX
3
4  static void mbox_event_cb(lv_obj_t *obj, lv_event_t evt);
5  static void btn_event_cb(lv_obj_t *btn, lv_event_t evt);
6  static void opa_anim(void *bg, lv_anim_value_t v);
7
8  static lv_obj_t *mbox, *info;
9  static lv_style_t style_modal;
10
11  static const char welcome_info[] = "Welcome to the modal message box demo!\n"
12                                     "Press the button to display a message box.";
```

(下页继续)

(续上页)

```

13
14 static const char in_msg_info[] = "Notice that you cannot touch "
15     "the button again while the message box is open.";
16
17 void lv_ex_msgbox_2(void)
18 {
19     lv_style_init(&style_modal);
20     lv_style_set_bg_color(&style_modal, LV_STATE_DEFAULT, LV_COLOR_BLACK);
21
22     /* Create a button, then set its position and event callback */
23     lv_obj_t *btn = lv_btn_create(lv_scr_act(), NULL);
24     lv_obj_set_size(btn, 200, 60);
25     lv_obj_set_event_cb(btn, btn_event_cb);
26     lv_obj_align(btn, NULL, LV_ALIGN_IN_TOP_LEFT, 20, 20);
27
28     /* Create a label on the button */
29     lv_obj_t *label = lv_label_create(btn, NULL);
30     lv_label_set_text(label, "Display a message box!");
31
32     /* Create an informative label on the screen */
33     info = lv_label_create(lv_scr_act(), NULL);
34     lv_label_set_text(info, welcome_info);
35     lv_label_set_long_mode(info, LV_LABEL_LONG_BREAK); /* Make sure text will
↪wrap */
36     lv_obj_set_width(info, LV_HOR_RES - 10);
37     lv_obj_align(info, NULL, LV_ALIGN_IN_BOTTOM_LEFT, 5, -5);
38
39 }
40
41 static void mbox_event_cb(lv_obj_t *obj, lv_event_t evt)
42 {
43     if(evt == LV_EVENT_DELETE && obj == mbox) {
44         /* Delete the parent modal background */
45         lv_obj_del_async(lv_obj_get_parent(mbox));
46         mbox = NULL; /* happens before object is actually deleted! */
47         lv_label_set_text(info, welcome_info);
48     } else if(evt == LV_EVENT_VALUE_CHANGED) {
49         /* A button was clicked */
50         lv_msgbox_start_auto_close(mbox, 0);
51     }
52 }
53
54 static void btn_event_cb(lv_obj_t *btn, lv_event_t evt)

```

(下页继续)

(续上页)

```

55 {
56     if (evt == LV_EVENT_CLICKED) {
57         /* Create a full-screen background */
58
59         /* Create a base object for the modal background */
60         lv_obj_t *obj = lv_obj_create(lv_scr_act(), NULL);
61         lv_obj_reset_style_list(obj, LV_OBJ_PART_MAIN);
62         lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style_modal);
63         lv_obj_set_pos(obj, 0, 0);
64         lv_obj_set_size(obj, LV_HOR_RES, LV_VER_RES);
65
66         static const char * btns2[] = {"Ok", "Cancel", ""};
67
68         /* Create the message box as a child of the modal background */
69         mbox = lv_msgbox_create(obj, NULL);
70         lv_msgbox_add_btns(mbox, btns2);
71         lv_msgbox_set_text(mbox, "Hello world!");
72         lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0);
73         lv_obj_set_event_cb(mbox, mbox_event_cb);
74
75         /* Fade the message box in with an animation */
76         lv_anim_t a;
77         lv_anim_init(&a);
78         lv_anim_set_var(&a, obj);
79         lv_anim_set_time(&a, 500);
80         lv_anim_set_values(&a, LV_OPA_TRANSP, LV_OPA_50);
81         lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t) opa_anim);
82         lv_anim_start(&a);
83
84         lv_label_set_text(info, in_msg_info);
85         lv_obj_align(info, NULL, LV_ALIGN_IN_BOTTOM_LEFT, 5, -5);
86     }
87 }
88
89 static void opa_anim(void * bg, lv_anim_value_t v)
90 {
91     lv_obj_set_style_local_bg_opa(bg, LV_OBJ_PART_MAIN, LV_STATE_DEFAULT, v);
92 }
93
94 #endif

```


43.7 相关 API

43.7.1 函数

```

1  lv_obj_t * lv_objmask_create(lv_obj_t * par, constlv_obj_t *copy)
2  创建对象遮罩对象
3  返回:
4  指向创建的对象掩码的指针
5  形参:
6  par: 指向对象的指针, 它将是新对象蒙版的父对象
7  copy: 指向对象掩码对象的指针, 如果不为 NULL, 则将从其复制新对象
8
9
10 lv_objmask_mask_t * lv_objmask_add_mask(lv_obj_t * objmask, void *param)
11 添加面膜
12 返回:
13 指向添加的蒙版的指针
14 形参:
15 objmask: 指向对象遮罩对象的指针
16 param: 初始化的 mask 参数
17
18
19 void lv_objmask_update_mask(lv_obj_t * objmask, lv_objmask_mask_t * mask, void *_
↪param )
20 更新已创建的蒙版
21 形参:
22 objmask: 指向对象遮罩对象的指针
23 mask: 指向创建的遮罩的指针 (由返回: lv_objmask_add_mask)
24 param: 初始化的 mask 参数 (由初始化 lv_draw_mask_line/angle/.../_init)
25
26
27 void lv_objmask_remove_mask(lv_obj_t * objmask, lv_objmask_mask_t * mask )
28 取下口罩
29 形参:
30 objmask: 指向对象遮罩对象的指针
31 mask: 指向创建的遮罩的指针 (由返回: lv_objmask_add_mask) 如果 NULL 通过, 则将删除所

```


对象蒙版 (lv_objmask)

44.1 概述

绘制其子级时，对象蒙版能够向图形添加一些蒙版。

44.2 零件和样式

对象蒙版只有一个主要部分称为 LV_OBJMASK_PART_BG，它使用典型的背景样式属性。

44.3 用法

44.3.1 添加蒙版

在向对象蒙版添加蒙版之前，应先初始化蒙版：

```
1 lv_draw_mask_<type>_param_t mask_param;  
2 lv_draw_mask_<type>_init(&mask_param, ...);  
3 lv_objmask_mask_t * mask_p = lv_objmask_add_mask(objmask, &mask_param);
```

Lvgl 支持以下蒙版类型：

- **line** 剪裁线条左上/右下的像素。可以从两个点或一个点和一个角度初始化：

- **angle** 将像素仅保持在给定的开始角度和结束角度之间
- **radius** 将像素仅保留在可以具有半径的矩形内（也可以是一个圆形）。可以反转以将像素保持在矩形之外。
- **fade** 垂直淡入（根据像素的 y 位置更改像素的不透明度）
- **map** 使用 Alpha 遮罩（字节数组）描述像素的不透明度。

遮罩中的坐标是相对于对象的。也就是说，如果对象移动，则蒙版也随之移动。

44.3.2 更新蒙版

可以使用 `lv_objmask_update_mask(objmask, mask_p, new_param)` 更新现有的掩码，其中 `mask_p` 是 `lv_objmask_add_mask` 的返回值。

44.3.3 移除蒙版

可以使用 `lv_objmask_remove_mask(objmask, mask_p)` 删除蒙版

44.4 事件

仅支持 通用事件

了解有关 事件 的更多内容。

44.5 按键处理

对象类型不处理任何键。

了解有关 按键 的更多内容。

44.6 范例

44.6.1 创建几个对象蒙版

上述效果的示例代码：

```
1  #include ".././../lv_examples.h"
2  #if LV_USE_OBJMASK
3
4  void lv_ex_objmask_1(void)
```

(下页继续)

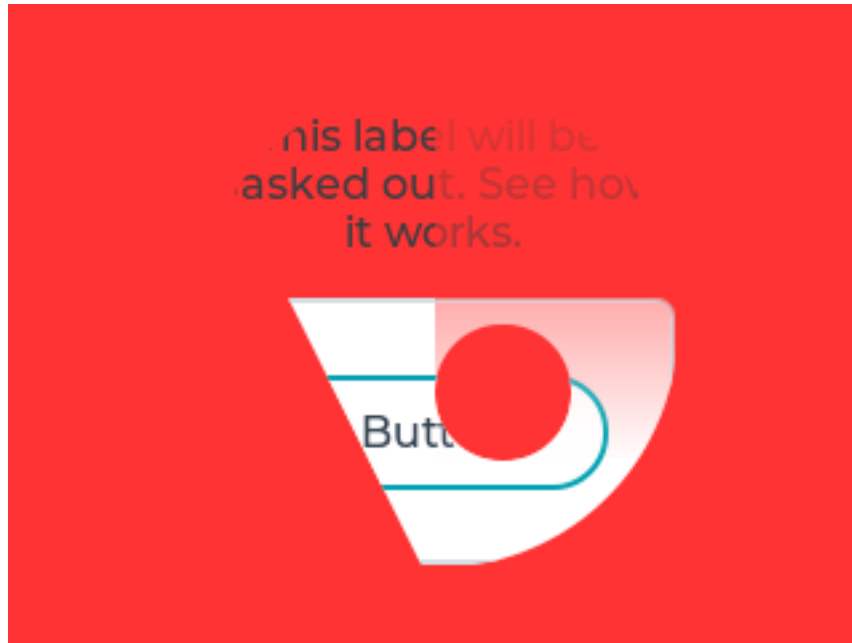


图 1: 几个对象蒙版

(续上页)

```

5      {
6
7          /*Set a very visible color for the screen to clearly see what happens*/
8          lv_obj_set_style_local_bg_color(lv_scr_act(), LV_OBJ_PART_MAIN, LV_STATE_
↪DEFAULT, lv_color_hex3(0xf33));
9
10         lv_obj_t * om = lv_objmask_create(lv_scr_act(), NULL);
11         lv_obj_set_size(om, 200, 200);
12         lv_obj_align(om, NULL, LV_ALIGN_CENTER, 0, 0);
13         lv_obj_t * label = lv_label_create(om, NULL);
14         lv_label_set_long_mode(label, LV_LABEL_LONG_BREAK);
15         lv_label_set_align(label, LV_LABEL_ALIGN_CENTER);
16         lv_obj_set_width(label, 180);
17         lv_label_set_text(label, "This label will be masked out. See how it works.
↪");
18
19         lv_obj_align(label, NULL, LV_ALIGN_IN_TOP_MID, 0, 20);
20
21         lv_obj_t * cont = lv_cont_create(om, NULL);
22         lv_obj_set_size(cont, 180, 100);
23         lv_obj_set_drag(cont, true);
24         lv_obj_align(cont, NULL, LV_ALIGN_IN_BOTTOM_MID, 0, -10);
25
26         lv_obj_t * btn = lv_btn_create(cont, NULL);

```

(下页继续)

(续上页)

```
26     lv_obj_align(btn, NULL, LV_ALIGN_CENTER, 0, 0);
27     lv_obj_set_style_local_value_str(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
↪ "Button");
28     uint32_t t;
29
30     lv_refr_now(NULL);
31     t = lv_tick_get();
32     while(lv_tick_elaps(t) < 1000);
33
34     lv_area_t a;
35     lv_draw_mask_radius_param_t r1;
36
37     a.x1 = 10;
38     a.y1 = 10;
39     a.x2 = 190;
40     a.y2 = 190;
41     lv_draw_mask_radius_init(&r1, &a, LV_RADIUS_CIRCLE, false);
42     lv_objmask_add_mask(om, &r1);
43
44     lv_refr_now(NULL);
45     t = lv_tick_get();
46     while(lv_tick_elaps(t) < 1000);
47
48     a.x1 = 100;
49     a.y1 = 100;
50     a.x2 = 150;
51     a.y2 = 150;
52     lv_draw_mask_radius_init(&r1, &a, LV_RADIUS_CIRCLE, true);
53     lv_objmask_add_mask(om, &r1);
54
55     lv_refr_now(NULL);
56     t = lv_tick_get();
57     while(lv_tick_elaps(t) < 1000);
58
59     lv_draw_mask_line_param_t l1;
60     lv_draw_mask_line_points_init(&l1, 0, 0, 100, 200, LV_DRAW_MASK_LINE_SIDE_
↪ TOP);
61     lv_objmask_add_mask(om, &l1);
62
63     lv_refr_now(NULL);
64     t = lv_tick_get();
65     while(lv_tick_elaps(t) < 1000);
66
```

(下页继续)

(续上页)

```
67     lv_draw_mask_fade_param_t f1;
68     a.x1 = 100;
69     a.y1 = 0;
70     a.x2 = 200;
71     a.y2 = 200;
72     lv_draw_mask_fade_init(&f1, &a, LV_OPA_TRANSP, 0, LV_OPA_COVER, 150);
73     lv_objmask_add_mask(om, &f1);
74 }
75
76 #endif
```

44.6.2 文字蒙版

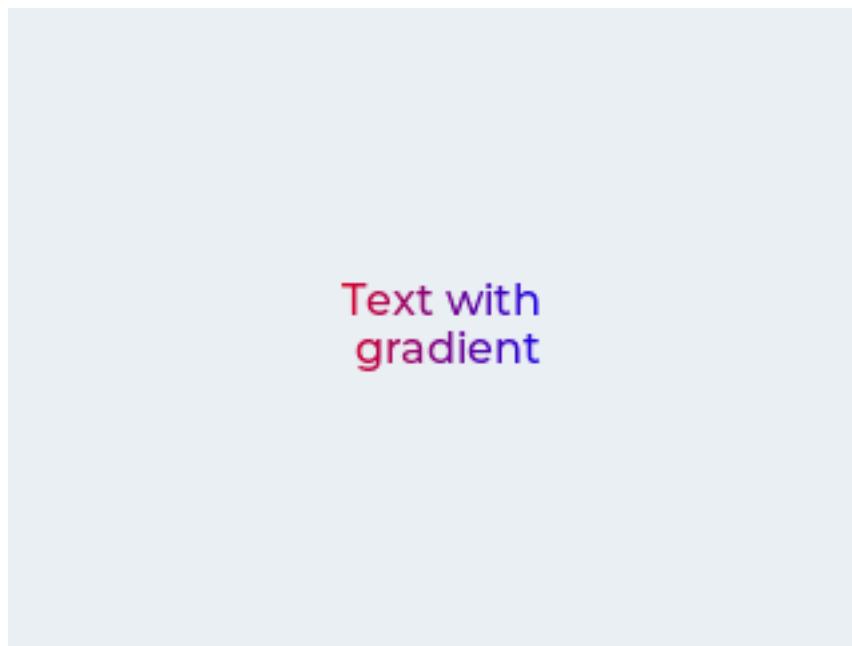


图 2: 文字蒙版

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #if LV_USE_OBJMASK
3
4  #define MASK_WIDTH 100
5  #define MASK_HEIGHT 50
6
7  void lv_ex_objmask_2(void)
8  {
```

(下页继续)

(续上页)

```

9
10      /* Create the mask of a text by drawing it to a canvas*/
11      static lv_opa_t mask_map[MASK_WIDTH * MASK_HEIGHT];
12
13      /*Create a "8 bit alpha" canvas and clear it*/
14      lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);
15      lv_canvas_set_buffer(canvas, mask_map, MASK_WIDTH, MASK_HEIGHT, LV_IMG_CF_
↪ALPHA_8BIT);
16      lv_canvas_fill_bg(canvas, LV_COLOR_BLACK, LV_OPA_TRANSP);
17
18      /*Draw a label to the canvas. The result "image" will be used as mask*/
19      lv_draw_label_dsc_t label_dsc;
20      lv_draw_label_dsc_init(&label_dsc);
21      label_dsc.color = LV_COLOR_WHITE;
22      lv_canvas_draw_text(canvas, 5, 5, MASK_WIDTH, &label_dsc, "Text with_
↪gradient", LV_LABEL_ALIGN_CENTER);
23
24      /*The mask is reads the canvas is not required anymore*/
25      lv_obj_del(canvas);
26
27      /*Create an object mask which will use the created mask*/
28      lv_obj_t * om = lv_objmask_create(lv_scr_act(), NULL);
29      lv_obj_set_size(om, MASK_WIDTH, MASK_HEIGHT);
30      lv_obj_align(om, NULL, LV_ALIGN_CENTER, 0, 0);
31
32      /*Add the created mask map to the object mask*/
33      lv_draw_mask_map_param_t m;
34      lv_area_t a;
35      a.x1 = 0;
36      a.y1 = 0;
37      a.x2 = MASK_WIDTH - 1;
38      a.y2 = MASK_HEIGHT - 1;
39      lv_draw_mask_map_init(&m, &a, mask_map);
40      lv_objmask_add_mask(om, &m);
41
42      /*Create a style with gradient*/
43      static lv_style_t style_bg;
44      lv_style_init(&style_bg);
45      lv_style_set_bg_opa(&style_bg, LV_STATE_DEFAULT, LV_OPA_COVER);
46      lv_style_set_bg_color(&style_bg, LV_STATE_DEFAULT, LV_COLOR_RED);
47      lv_style_set_bg_grad_color(&style_bg, LV_STATE_DEFAULT, LV_COLOR_BLUE);
48      lv_style_set_bg_grad_dir(&style_bg, LV_STATE_DEFAULT, LV_GRAD_DIR_HOR);
49

```

(下页继续)

(续上页)

```

50      /* Create and object with the gradient style on the object mask.
51      * The text will be masked from the gradient*/
52      lv_obj_t * bg = lv_obj_create(om, NULL);
53      lv_obj_reset_style_list(bg, LV_OBJ_PART_MAIN);
54      lv_obj_add_style(bg, LV_OBJ_PART_MAIN, &style_bg);
55      lv_obj_set_size(bg, MASK_WIDTH, MASK_HEIGHT);
56
57  }
58
59  #endif

```

44.7 相关 API

44.7.1 函数

```

1  lv_obj_t * lv_objmask_create(lv_obj_t * par, const lv_obj_t * copy)
2  创建对象遮罩对象
3  返回:
4  指向创建的对象掩码的指针
5  形参:
6  par: 指向对象的指针, 它将是新对象蒙版的父对象
7  copy: 指向对象掩码对象的指针, 如果不为 NULL, 则将从其复制新对象
8
9
10 lv_objmask_mask_t * lv_objmask_add_mask(lv_obj_t * objmask, void * param)
11 添加面膜
12 返回:
13 指向添加的蒙版的指针
14 形参:
15 objmask: 指向对象遮罩对象的指针
16 param: 初始化的 mask 参数
17
18
19 void lv_objmask_update_mask(lv_obj_t * objmask, lv_objmask_mask_t * mask, void *
↪param )
20 更新已创建的蒙版
21 形参:
22 objmask: 指向对象遮罩对象的指针
23 mask: 指向创建的遮罩的指针 (由返回: lv_objmask_add_mask)
24 param: 初始化的 mask 参数 (由初始化 lv_draw_mask_line/angle/.../_init)
25

```

(下页继续)

(续上页)

```
26
27 void lv_objmask_remove_mask(lv_obj_t * objmask, lv_objmask_mask_t * mask )
28 取下口罩
29 形参:
30 objmask: 指向对象遮罩对象的指针
31 mask: 指向创建的遮罩的指针 (由返回: lv_objmask_add_mask) 如果 NULL
```

45.1 概述

页面彼此包含两个 容器 (lv_cont) :

- 背景
- 可滚动的顶部。

45.2 零件和样式

页面的主要部分称为 LV_PAGE_PART_BG , 它是页面的背景。它使用所有典型的背景样式属性。使用填充会增加侧面的空间。

可以通过 LV_PAGE_PART_SCROLL 部分引用可滚动对象。它还使用所有典型的背景样式属性和填充来增加侧面的空间。

LV_LIST_PART_SCROLLBAR 是绘制滚动条的背景的虚拟部分。使用所有典型的背景样式属性, 使用 size 设置滚动条的宽度, 并使用 pad_right 和 pad_bottom 设置间距。

LV_LIST_PART_EDGE_FLASH 还是背景的虚拟部分, 当无法进一步沿该方向滚动列表时, 将在侧面绘制半圆。使用所有典型的背景属性。

45.3 用法

后台对象可以像页面本身一样被引用。例如。设置页面的宽度：`lv_obj_set_width(page, 100)`。

如果在页面上创建了一个子代，它将被自动移动到可滚动容器中。如果可滚动容器变大，则可以通过拖动来滚动背景（如智能手机上的列表）。

默认情况下，可滚动控件的 `LV_FIT_MAX` 适合所有方向。这意味着当子代处于背景中时，可滚动大小将与背景大小相同（减去填充）。

但是，当将对象放置在背景之外时，可滚动大小将增加以使其包含其中。

45.3.1 滚动条

可以根据以下四个策略显示滚动条：

- `LV_SCROLLBAR_MODE_OFF` 一直都不显示滚动条
- `LV_SCROLLBAR_MODE_ON` 一直都显示滚动条
- `LV_SCROLLBAR_MODE_DRAG` 拖动页面时显示滚动条
- `LV_SCROLLBAR_MODE_AUTO` 当可滚动容器的大小足以滚动时显示滚动条
- `LV_SCROLLBAR_MODE_HIDE` 暂时隐藏滚动条
- `LV_SCROLLBAR_MODE_UNHIDE` 取消隐藏以前隐藏的滚动条。也恢复原始模式

可以通过以下方式更改滚动条显示策略：`lv_page_set_scrollbar_mode(page, SB_MODE)`。默认值为 `LV_SCROLLBAR_MODE_AUTO`。

45.3.2 胶水对象

可以将孩子“粘”到页面上。在这种情况下，是否可以通过拖动该对象来滚动页面。可以通过 `lv_page_glue_obj(child, true)` 启用它。

45.3.3 焦点对象

页面上的对象可以使用 `lv_page_focus(page, child, LV_ANIM_ON/OFF)` 进行聚焦。它将移动可滚动容器以显示一个孩子。动画的时间可以通过 `lv_page_set_anim_time(page, anim_time)` 设置，以毫秒为单位。`child` 不必是页面的直接子级。如果可滚动对象也是该对象的祖父母，则此方法有效。

45.3.4 手动导航

可以使用 `lv_page_scroll_hor(page, dist)` 和 `lv_page_scroll_ver(page, dist)` 手动移动可滚动对象

45.3.5 滚动传播

如果列表是在另一个可滚动元素（如另一个页面）上创建的，并且 Page 无法进一步滚动，则滚动可以传播到父对象，以继续在父对象上滚动。可以使用 `lv_page_set_edge_flash(list, true)` 启用它

45.3.6 清除页面

页面上创建的所有对象都可以使用 `lv_page_clean(page)` 进行清理。请注意，`lv_page_clean(page)` 在这里不起作用，因为它也会删除可滚动对象。

45.3.7 可滚动的 API

有一些直接设置/获取可滚动属性的函数：

- `lv_page_get_scrl()`
- `lv_page_set_scrl_fit/fint2/fit4()`
- `lv_page_set_scrl_width()`
- `lv_page_set_scrl_height()`
- `lv_page_set_scrl_fit_width()`
- `lv_page_set_scrl_fit_height()`
- `lv_page_set_scrl_layout()`

45.4 事件

仅支持 通用事件

可滚动对象具有默认的事件回调，该事件回调将以下事件传播到后台对象：

- `LV_EVENT_PRESSED`
- `LV_EVENT_PRESSING`
- `LV_EVENT_PRESS_LOST`
- `LV_EVENT_RELEASED`
- `LV_EVENT_SHORT_CLICKED`

- LV_EVENT_CLICKED
- LV_EVENT_LONG_PRESSED
- LV_EVENT_LONG_PRESSED_REPEAT

了解有关 [事件](#) 的更多内容。

45.5 按键处理

页面处理以下键：

- LV_KEY_RIGHT/LEFT/UP/DOWN 滚动页面

了解有关 [按键](#) 的更多内容。

45.6 范例

45.6.1 带有滚动条的页面

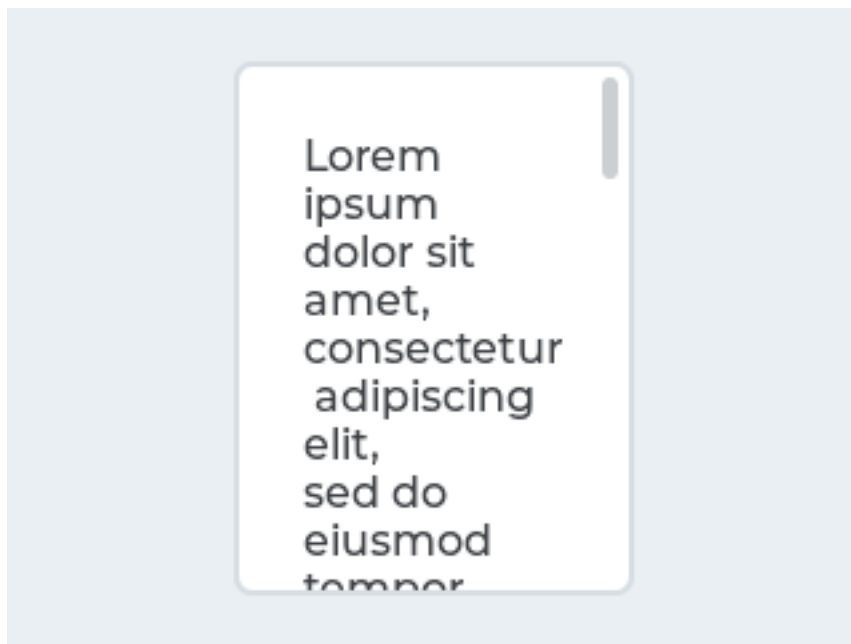


图 1: 带有滚动条的页面

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #if LV_USE_PAGE
```

(下页继续)

(续上页)

```

3
4 void lv_ex_page_1(void)
5 {
6     /*Create a page*/
7     lv_obj_t * page = lv_page_create(lv_scr_act(), NULL);
8     lv_obj_set_size(page, 150, 200);
9     lv_obj_align(page, NULL, LV_ALIGN_CENTER, 0, 0);
10
11     /*Create a label on the page*/
12     lv_obj_t * label = lv_label_create(page, NULL);
13     lv_label_set_long_mode(label, LV_LABEL_LONG_BREAK);
14     /*Automatically break long lines*/
15     lv_obj_set_width(label, lv_page_get_width_fit(page));
16     /*Set the label width to max value to not show hor. scroll bars*/
17     lv_label_set_text(label, "Lorem ipsum dolor sit amet, consectetur
18     adipiscing elit,\n"
19     "sed do eiusmod tempor
20     incididunt ut labore et dolore magna aliqua.\n"
21     "Ut enim ad minim veniam,
22     quis nostrud exercitation ullamco\n"
23     "laboris nisi ut aliquip
24     ex ea commodo consequat. Duis aute irure\n"
25     "dolor in reprehenderit
26     eu fugiat nulla
27     Excepteur sint occaecat
28     cupidatat non proident, sunt in culpa\n"
29     "qui officia deserunt
30     mollit anim id est laborum.");
31 }
32
33 #endif

```

45.7 相关 API

45.7.1 函数

```
lv_obj_t * lv_page_create(lv_obj_t * par, const lv_obj_t * copy)
```

创建页面对象

返回：

(下页继续)

(续上页)

指向创建页面的指针

形参:

par: 指向对象的指针, 它将是新页面的父对象

copy: 指向页面对象的指针, 如果不为 NULL, 则将从其复制新对象

```
void lv_page_clean(lv_obj_t *page)
```

删除 scl 对象的所有子级, 而不删除 scl 子级。

形参:

page: 指向对象的指针

```
lv_obj_t * lv_page_get_scrollable(const lv_obj_t *page)
```

获取页面的可滚动对象

返回:

指向容器的指针, 该容器是页面的可滚动部分

形参:

page: 指向页面对象的指针

```
uint16_t lv_page_get_anim_time(const lv_obj_t *page)
```

获取动画时间

返回:

动画时间 (以毫秒为单位)

形参:

page: 指向页面对象的指针

```
void lv_page_set_scrollbar_mode(lv_obj_t *page, lv_scrollbar_mode_t sb_mode )
```

在页面上设置滚动条模式

形参:

page: 指向页面对象的指针

sb_mode: 来自 “ lv_page_sb.mode_t ” 枚举的新模式

```
void lv_page_set_anim_time(lv_obj_t *page, uint16_t anim_time )
```

设置页面的动画时间

形参:

page: 指向页面对象的指针

anim_time: 动画时间 (以毫秒为单位)

```
void lv_page_set_scroll_propagation(lv_obj_t * page, bool en )
```

(下页继续)

(续上页)

启用滚动传播功能。如果启用，则如果没有更多滚动空间，页面将移动其父级。页面需要具有类似页面的父页面（例如 lv_page, lv_tabview 选项卡, lv_win 内容区域等）。如果启用，则拖动方向将 LV_DRAG_DIR_ONE 自动更改，以允许一次仅在一个方向上滚动。

形参：

page: 指向页面的指针

en: true 或 false 启用/禁用滚动传播

```
void lv_page_set_edge_flash(lv_obj_t * page, bool en )
```

启用边缘闪光效果。（到达边缘时显示弧线）

形参：

page: 指向页面的指针

en: true 或 false 启用/禁用端闪

```
void lv_page_set_scrollable_fit4(lv_obj_t *page, lv_fit_t left, lv_fit_t right, lv_fit_t top, lv_fit_t bottom)
```

分别所有四个方向上设置适合策略。它告诉如何自动更改页面大小。

形参：

page: 指向页面对象的指针

left: 从左适合政策 lv_fit_t

right: 合适的政策来自 lv_fit_t

top: 底部适合政策 lv_fit_t

bottom: 底部适合政策 lv_fit_t

```
void lv_page_set_scrollable_fit2(lv_obj_t *page, lv_fit_t hor, lv_fit_t ver )
```

分别水平和垂直设置适合策略。它告诉如何自动更改页面大小。

形参：

page: 指向页面对象的指针

hor: 来自的水平拟合政策 lv_fit_t

ver: 垂直适合政策，来自 lv_fit_t

```
void lv_page_set_scrollable_fit(lv_obj_t *page, lv_fit_t fit)
```

一次在所有 4 个方向上设置拟合策略。它告诉如何自动更改页面大小。

形参：

page: 指向按钮对象的指针

fit: 适合政策，来自 lv_fit_t

```
void lv_page_set_scrl_width(lv_obj_t *page, lv_coord_t w )
```

设置页面可滚动部分的宽度

(下页继续)

(续上页)

形参:

page: 指向页面对象的指针

w: 新的可滚动宽度 (没有启用水平对齐功能)

```
void lv_page_set_scrl_height(lv_obj_t *page, lv_coord_t h )
```

设置页面可滚动部分的高度

形参:

page: 指向页面对象的指针

h: 新的可滚动高度 (启用垂直调整没有效果)

```
void lv_page_set_scrl_layout(lv_obj_t *page, lv_layout_t layout)
```

设置页面可滚动部分的布局

形参:

page: 指向页面对象的指针

layout: 来自 “ lv_cont_layout_t ” 的布局

```
lv_scrollbar_mode_t lv_page_get_scrollbar_mode(const lv_obj_t *page)
```

在页面上设置滚动条模式

返回:

来自 “ lv_page_sb.mode_t ” 枚举的模式

形参:

page: 指向页面对象的指针

```
bool lv_page_get_scroll_propagation(lv_obj_t *page)
```

获取滚动传播属性

返回:

对或错

形参:

page: 指向页面的指针

```
bool lv_page_get_edge_flash(lv_obj_t *page)
```

获取边缘闪光效果属性。

形参:

page: 指向页面的指针返回: true 或 false

```
lv_coord_t lv_page_get_width_fit(lv_obj_t *page)
```

获得可以设置为子代仍不会导致溢出的宽度 (显示滚动条)

(下页继续)

(续上页)

返回:

仍然适合页面的宽度

形参:

page: 指向页面对象的指针

```
lv_coord_t lv_page_get_height_fit(lv_obj_t *page)
```

获得可以设置为子代仍不会导致溢出的高度 (显示滚动条)

返回:

仍然适合页面的高度

形参:

page: 指向页面对象的指针

```
lv_coord_t lv_page_get_width_grid(lv_obj_t * page, uint8_t div, uint8_t span )
```

划分对象的宽度并获得给定列数的宽度。也要考虑背景的填充和可滚动。

返回:

根据给定参数的宽度

形参:

page: 指向对象的指针

div: 表示假设有多少列。如果为 1, 则宽度将设置为父级的宽度; 如果为 2, 则只有父级宽度的一半-父级的内部填充; 如果为 3, 则只有第三个父级宽度-2 * 父级的内部填充

span: 合并了多少列

```
lv_coord_t lv_page_get_height_grid(lv_obj_t * page, uint8_t div, uint8_t span )
```

划分对象的高度并获得给定列数的宽度。也要考虑背景的填充和可滚动。

返回:

根据给定参数的高度

形参:

page: 指向对象的指针

div: 表示假设有多少行。如果为 1, 则将设置父级的高度; 如果为 2, 则只有父级的一半高度-父级的内部填充; 如果只有 3, 则只有第三级父级的高度-2 * 父级的内部填充

span: 合并了多少行

```
lv_coord_t lv_page_get_scrl_width(const lv_obj_t *page)
```

获取页面可滚动部分的宽度

返回:

滚动条的宽度

形参:

page: 指向页面对象的指针

(下页继续)

(续上页)

```
171
172 lv_coord_t lv_page_get_scrl_height(constlv_obj_t *page)
173 获取页面可滚动部分的高度
174 返回:
175 滚动条的高度
176 形参:
177 page: 指向页面对象的指针
178
179
180 lv_layout_t lv_page_get_scrl_layout(constlv_obj_t *page)
181 获取页面可滚动部分的布局
182 返回:
183 来自 “ lv_cont_layout_t” 的布局
184 形参:
185 page: 指向页面对象的指针
186
187
188 lv_fit_t lv_page_get_scrl_fit_left(constlv_obj_t *page)
189 获取左合身模式
190 返回:
191 的元素 lv_fit_t
192 形参:
193 page: 指向页面对象的指针
194
195
196 lv_fit_t lv_page_get_scrl_fit_right(constlv_obj_t *page)
197 获得合适的健身模式
198 返回:
199 的元素 lv_fit_t
200 形参:
201 page: 指向页面对象的指针
202
203
204 lv_fit_t lv_page_get_scrl_fit_top(constlv_obj_t *page)
205 获取最适合的模式
206 返回:
207 的元素 lv_fit_t
208 形参:
209 page: 指向页面对象的指针
210
211
212 lv_fit_t lv_page_get_scrl_fit_bottom(constlv_obj_t *page)
213 获取最合适的模式
```

(下页继续)

(续上页)

返回:

的元素 lv_fit_t

形参:

page: 指向页面对象的指针

```
bool lv_page_on_edge(lv_obj_t * page, lv_page_edge_t edge )
```

查找页面是否已滚动到特定边缘。

返回:

如果页面在指定的边缘, 则为 true

形参:

page: 页面对象

edge: 边缘检查

```
void lv_page_glue_obj(lv_obj_t * obj, bool glue)
```

将对象粘到页面上。之后, 页面也可以与此对象一起移动 (拖动)。

形参:

obj: 指向页面上对象的指针

glue: true: 启用胶水, false: 禁用胶水

```
void lv_page_focus(lv_obj_t *page, const lv_obj_t * obj, lv_anim_enable_t anim_en )
```

专注于一个对象。它确保对象将在页面上可见。

形参:

page: 指向页面对象的指针

obj: 指向要聚焦的对象的指针 (必须在页面上)

anim_en: LV_ANIM_ON 聚焦动画; LV_ANIM_OFF 无需动画即可对焦

```
void lv_page_scroll_hor(lv_obj_t *page, lv_coord_t dist )
```

水平滚动页面

形参:

page: 指向页面对象的指针

dist: 滚动距离 (<0: 向左滚动; > 0 向右滚动)

```
void lv_page_scroll_ver(lv_obj_t *page, lv_coord_t dist )
```

垂直滚动页面

形参:

page: 指向页面对象的指针

dist: 滚动距离 (<0: 向下滚动; > 0 向上滚动)

(下页继续)

(续上页)

```
257
258 void lv_page_start_edge_flash(lv_obj_t *page, lv_page_edge_t edge)
259     不打算由用户直接使用，而是由内部的其他对象类型直接使用。开始边缘 Flash 动画。
260     形参：
261     page：
262     edge：要闪烁的边缘。可 LV_PAGE_EDGE_LEFT/RIGHT/TOP/BOTTOM
```

46.1 概述

滚筒允许通过滚动简单地从多个选项选择一个选项。

46.2 零件和样式

滚筒的主要部件称为 `LV_ROLLER_PART_BG`。它是一个矩形，并使用所有典型的背景属性。`Roller` 标签的样式继承自背景的背景属性。要调整选项之间的间距，请使用 `text_line_space` 样式属性。填充样式属性设置了侧面的空间。

中间的选定选项可以用 `LV_ROLLER_PART_SELECTED` 虚拟零件引用。除了典型的背景属性外，它还使用文本属性来更改所选区域中文本的外观。

46.3 用法

46.3.1 设定选项

这些选项作为带有 `lv_roller_set_options(roller, options, LV_ROLLER_MODE_NORMAL/INFINITE)` 的字符串传递给 `Roller`。选项应用 `\n` 分隔。例如：`"First\nSecond\nThird"`。

`LV_ROLLER_MODE_INFINITE` 使滚子呈圆形。

可以使用 `lv_roller_set_selected(roller, id, LV_ANIM_ON/OFF)` 手动选择选项，其中 `id` 是选项的索引。

46.3.2 获取选择的选项

使用 `lv_roller_get_selected(roller)` 获取当前选定的选项，它将返回选定选项的索引。

`lv_roller_get_selected_str(roller, buf, buf_size)` 将所选选项的名称复制到 `buf`。

46.3.3 选项对齐

要水平对齐标签，请使用 `lv_roller_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)` 水平对齐标签。

46.3.4 可见行

可见行数可以通过 `lv_roller_set_visible_row_count(roller, num)` 进行调整

46.3.5 动画时间

当滚轴滚动且未完全停在某个选项上时，它将自动滚动到最近的有效选项。可以通过 `lv_roller_set_anim_time(roller, anim_time)` 更改此滚动动画的时间。动画时间为零表示没有动画。

46.4 事件

除了通用事件，滚筒还支持以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 选定新选项时发送

了解有关事件的更多内容。

46.5 按键处理

以下按键由按钮处理：

- **LV_KEY_RIGHT/DOWN** 选择下一个选项
- **LV_KEY_LEFT/UP** 选择上一个选项
- **LV_KEY_ENTER** 应用选定的选项（发送 **LV_EVENT_VALUE_CHANGED** 事件）

了解有关按键的更多内容。

46.6 范例

46.6.1 简易滚筒

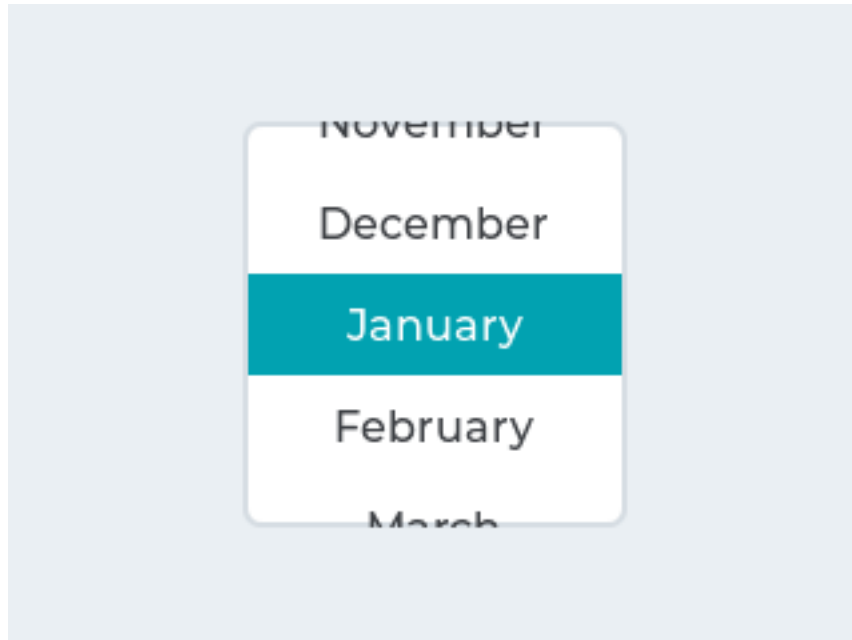


图 1: 创建一个简易的滚筒

上述效果的示例代码:

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_ROLLER
4
5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
7      if(event == LV_EVENT_VALUE_CHANGED) {
8          char buf[32];
9          lv_roller_get_selected_str(obj, buf, sizeof(buf));
10         printf("Selected month: %s\n", buf);
11     }
12 }
13
14
15 void lv_ex_roller_1(void)
16 {
17     lv_obj_t *roller1 = lv_roller_create(lv_scr_act(), NULL);
18     lv_roller_set_options(roller1,
```

(下页继续)

(续上页)

```
19         "January\n"  
20         "February\n"  
21         "March\n"  
22         "April\n"  
23         "May\n"  
24         "June\n"  
25         "July\n"  
26         "August\n"  
27         "September\n"  
28         "October\n"  
29         "November\n"  
30         "December",  
31         LV_ROLLER_MODE_INFINITE);  
32  
33     lv_roller_set_visible_row_count(roller1, 4);  
34     lv_obj_align(roller1, NULL, LV_ALIGN_CENTER, 0, 0);  
35     lv_obj_set_event_cb(roller1, event_handler);  
36 }  
37  
38 #endif
```

46.7 相关 API

TODO

47.1 概述

滑杆对象看起来像是带有旋钮的进度条 (lv_bar)。可以拖动该旋钮以设置一个值。滑块也可以是垂直或水平的。

47.2 零件和样式

滑块的主要部分称为 LV_SLIDER_PART_BG，它使用典型的背景样式属性。

LV_SLIDER_PART_INDIC 是一个虚拟部件，它也使用所有典型的背景属性。默认情况下，指标的最大大小与背景的大小相同，但在 LV_SLIDER_PART_BG 中设置正填充值将使指标变小。(负值会使它变大) 如果在指示器上使用了值样式属性，则将根据指示器的当前大小来计算对齐方式。例如，中心对齐值始终显示在指示器的中间，无论其当前大小如何。

LV_SLIDER_PART_KNOB 是一个虚拟部件，使用所有典型的背景属性来描述旋钮。与指示器类似，值文本也与旋钮的当前位置和大小对齐。默认情况下，旋钮是正方形 (具有半径)，其边长等于滑块的较小边。可以使用填充值使旋钮变大。填充值也可以是不对称的。

47.3 用法

47.3.1 值和范围

要设置初始值，请使用 `lv_slider_set_value(slider, new_value, LV_ANIM_ON/OFF)`。
`lv_slider_set_anim_time(slider, anim_time)` 设置动画时间（以毫秒为单位）。

要指定范围（最小，最大值），可以使用 `lv_slider_set_range(slider, min, max)`。

47.3.2 对称范围

除普通类型外，滑块还可以配置为两种其他类型：

- **LV_SLIDER_TYPE_NORMAL** 普通型
- **LV_SLIDER_TYPE_SYMMETRICAL** 将指标对称地绘制为零（从零开始，从左到右）
- **LV_SLIDER_TYPE_RANGE** 允许为左（起始）值使用附加旋钮。（可与 `lv_slider_set/get_left_value()` 一起使用）

可以使用 `lv_slider_set_type(slider, LV_SLIDER_TYPE_...)` 更改类型

47.3.3 仅旋钮模式

通常，可以通过拖动旋钮或单击滑块来调整滑块。在后一种情况下，旋钮移动到所单击的点，并且滑块值相应地变化。在某些情况下，希望将滑块设置为仅在拖动旋钮时做出反应。

通过调用 `lv_obj_set_adv_hittest(slider, true)`；启用此功能。

47.4 事件

除了通用事件，滑杆还支持以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 在使用键拖动或更改滑块时发送。拖动滑块时（仅当释放时）连续发送事件。使用 `lv_slider_is_dragged` 确定滑块是被拖动还是刚刚释放。

了解有关事件的更多内容。

47.5 按键处理

滑杆可处理以下按键：

- **LV_KEY_UP, LV_KEY_RIGHT** 将滑块的值增加 1
- **LV_KEY_DOWN, LV_KEY_LEFT** 将滑块的值减 1

了解有关 [按键](#) 的更多内容。

47.6 范例

47.6.1 自定义样式的滑块

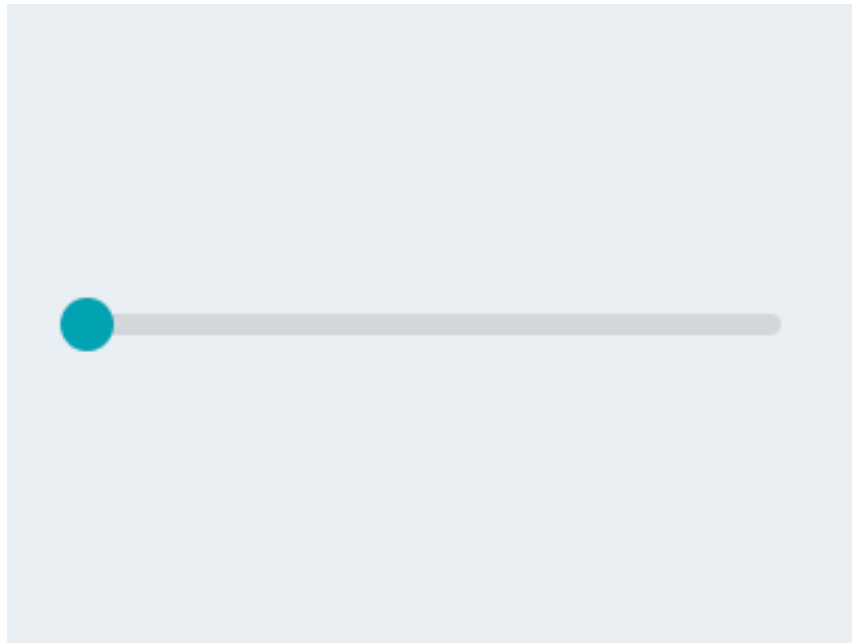


图 1: 自定义样式的滑块

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_SLIDER
4
5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
7      if(event == LV_EVENT_VALUE_CHANGED) {
8          printf("Value: %d\n", lv_slider_get_value(obj));
```

(下页继续)

(续上页)

```
9      }
10  }
11
12  void lv_ex_slider_1(void)
13  {
14      /*Create a slider*/
15      lv_obj_t * slider = lv_slider_create(lv_scr_act(), NULL);
16      lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0);
17      lv_obj_set_event_cb(slider, event_handler);
18  }
19
20  #endif
```

47.6.2 用滑块设定值

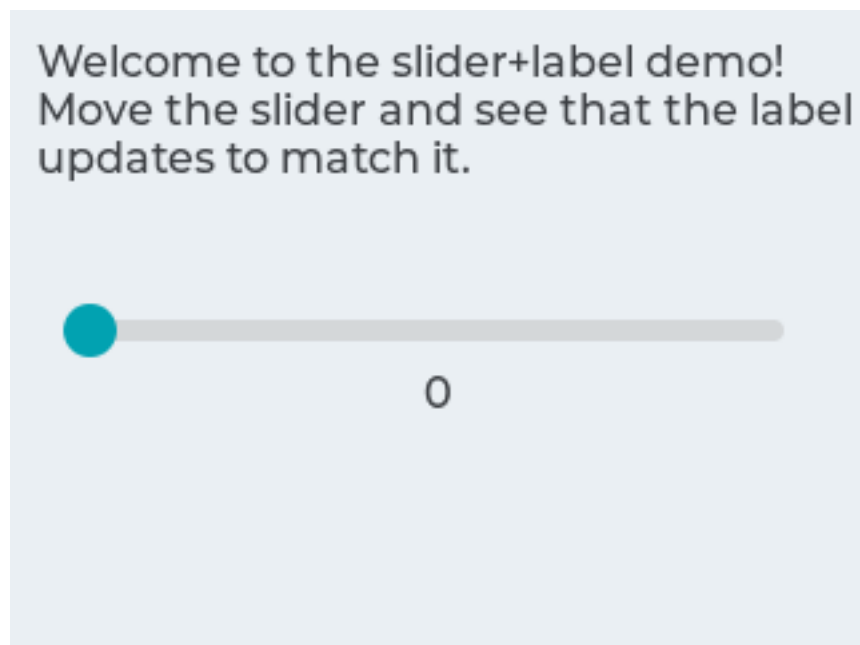


图 2: 用滑块设定值

上述效果的示例代码:

```
1  #include ".././../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_SLIDER
4
5  static void slider_event_cb(lv_obj_t * slider, lv_event_t event);
6  static lv_obj_t * slider_label;
```

(下页继续)

(续上页)

```

7
8 void lv_ex_slider_2(void)
9 {
10     /* Create a slider in the center of the display */
11     lv_obj_t * slider = lv_slider_create(lv_scr_act(), NULL);
12     lv_obj_set_width(slider, LV_DPI * 2);
13     lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0);
14     lv_obj_set_event_cb(slider, slider_event_cb);
15     lv_slider_set_range(slider, 0, 100);
16
17     /* Create a label below the slider */
18     slider_label = lv_label_create(lv_scr_act(), NULL);
19     lv_label_set_text(slider_label, "0");
20     lv_obj_set_auto_realign(slider_label, true);
21     lv_obj_align(slider_label, slider, LV_ALIGN_OUT_BOTTOM_MID, 0, 10);
22
23     /* Create an informative label */
24     lv_obj_t * info = lv_label_create(lv_scr_act(), NULL);
25     lv_label_set_text(info, "Welcome to the slider+label demo!\n"
26                                     "Move the slider and see_
↪that the label\n"
27                                     "updates to match it.");
28     lv_obj_align(info, NULL, LV_ALIGN_IN_TOP_LEFT, 10, 10);
29 }
30
31 static void slider_event_cb(lv_obj_t * slider, lv_event_t event)
32 {
33     if(event == LV_EVENT_VALUE_CHANGED) {
34         static char buf[4]; /* max 3 bytes for number plus 1 null_
↪terminating byte */
35         snprintf(buf, 4, "%u", lv_slider_get_value(slider));
36         lv_label_set_text(slider_label, buf);
37     }
38 }
39
40 #endif

```

47.7 相关 API

TODO

数字调整框 (lv_spinbox)

48.1 概述

数字调整框包含一个数字文本，可通过按键或 API 函数增加或减少数字。数字调整框的下面是修改后的文本框 (lv_textarea)。

48.2 零件和样式

数字调整框的主要部分称为 LV_SPINBOX_PART_BG，它是使用所有典型背景样式属性的矩形背景。它还使用其文本样式属性描述标签的样式。

LV_SPINBOX_PART_CURSOR 是描述光标的虚拟部分。阅读文本区域文档以获取详细说明。

48.3 用法

48.3.1 设定格式

`lv_spinbox_set_digit_format(spinbox, digit_count, separator_position)` 设置数字的格式。`digit_count` 设置位数。前导零被添加以填充左侧的空间。`separator_position` 设置小数点前的位数。0 表示没有小数点。

`lv_spinbox_set_padding_left(spinbox, cnt)` 在符号之间最左边的数字之间添加 `cnt` 个“空格”字符。

48.3.2 值和范围

`lv_spinbox_set_range(spinbox, min, max)` 设置 Spinbox 的范围。

`lv_spinbox_set_value(spinbox, num)` 手动设置 Spinbox 的值。

`lv_spinbox_increment(spinbox)` 和 `lv_spinbox_decrement(spinbox)` 递增/递减 Spinbox 的值。

`lv_spinbox_set_step(spinbox, step)` 设置增量减量。

48.4 事件

除了 通用事件，数字调整框还支持以下 特殊事件：

- **LV_EVENT_VALUE_CHANGED** 值更改时发送。（将该值设置为 `int32_t` 作为事件数据）
- **LV_EVENT_INSERT** 由父“文本”区域发送，但不应使用。

了解有关 事件 的更多内容。

48.5 按键处理

数字调整框支持一下按键：

- **LV_KEY_LEFT/RIGHT** 使用键盘向左/向右移动光标。使用编码器递减/递增所选数字。
- **LY_KEY_ENTER** 应用选定的选项（发送 `LV_EVENT_VALUE_CHANGED` 事件并关闭下拉列表）
- **LV_KEY_ENTER** 随着编码器得到的净数字。跳到最后一个之后的第一个。

了解有关 按键 的更多内容。

48.6 范例

48.6.1 简单的数字调整框

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_SPINBOX
4
5  static lv_obj_t * spinbox;
6
7
```

(下页继续)



图 1: 简单的数字调整框

(续上页)

```

8  static void lv_spinbox_increment_event_cb(lv_obj_t * btn, lv_event_t e)
9  {
10     if(e == LV_EVENT_SHORT_CLICKED || e == LV_EVENT_LONG_PRESSED_REPEAT) {
11         lv_spinbox_increment(spinbox);
12     }
13 }
14
15 static void lv_spinbox_decrement_event_cb(lv_obj_t * btn, lv_event_t e)
16 {
17     if(e == LV_EVENT_SHORT_CLICKED || e == LV_EVENT_LONG_PRESSED_REPEAT) {
18         lv_spinbox_decrement(spinbox);
19     }
20 }
21
22
23 void lv_ex_spinbox_1(void)
24 {
25     spinbox = lv_spinbox_create(lv_scr_act(), NULL);
26     lv_spinbox_set_range(spinbox, -1000, 90000);
27     lv_spinbox_set_digit_format(spinbox, 5, 2);
28     lv_spinbox_step_prev(spinbox);
29     lv_obj_set_width(spinbox, 100);
30     lv_obj_align(spinbox, NULL, LV_ALIGN_CENTER, 0, 0);

```

(下页继续)

(续上页)

```
31
32     lv_coord_t h = lv_obj_get_height(spinbox);
33     lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);
34     lv_obj_set_size(btn, h, h);
35     lv_obj_align(btn, spinbox, LV_ALIGN_OUT_RIGHT_MID, 5, 0);
36     lv_theme_apply(btn, LV_THEME_SPINBOX_BTN);
37     lv_obj_set_style_local_value_str(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, ↵
↵LV_SYMBOL_PLUS);
38     lv_obj_set_event_cb(btn, lv_spinbox_increment_event_cb);
39
40     btn = lv_btn_create(lv_scr_act(), btn);
41     lv_obj_align(btn, spinbox, LV_ALIGN_OUT_LEFT_MID, -5, 0);
42     lv_obj_set_event_cb(btn, lv_spinbox_decrement_event_cb);
43     lv_obj_set_style_local_value_str(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, ↵
↵LV_SYMBOL_MINUS);
44 }
45
46 #endif
```

48.7 相关 API

TODO

49.1 概述

旋转器对象是边界上的旋转弧，实现旋转加载效果。

49.2 零件和样式

旋转器包括一下部分：

- LV_SPINNER_PART_BG: 主要部分
- LV_SPINNER_PART_INDIC: 旋转弧（虚拟部分）

零件和样式的作用与 弧 (lv_arc) 情况相同。

49.3 用法

49.3.1 弧长

圆弧的长度可以通过 `lv_spinner_set_arc_length(spinner, deg)` 进行调整。

49.3.2 旋转速度

旋转速度可以通过 `lv_spinner_set_spin_time(preload, time_ms)` 进行调整。

49.3.3 旋转类型

支持以下旋转类型

- **LV_SPINNER_TYPE_SPINNING_ARC** 旋转弧线，在顶部减速
- **LV_SPINNER_TYPE_FILLSPIN_ARC** 旋转弧线，在顶部放慢速度，但也伸展弧线
- **LV_SPINNER_TYPE_CONSTANT_ARC** 以恒定速度旋转

使用 `lv_spinner_set_type(preload, LV_SPINNER_TYPE_...)` 进行设置

49.3.4 旋转方向

旋转方向可以通过 `lv_spinner_set_dir(preload, LV_SPINNER_DIR_FORWARD/BACKWARD)` 进行更改。

仅支持 [通用事件](#)

了解有关 [事件](#) 的更多内容。

49.4 按键处理

对象类型不处理任何键。

了解有关 [按键](#) 的更多内容。

49.5 范例

49.5.1 简单的旋转效果

上述效果的示例代码：

```
1  #include "../lv_examples.h"
2  #if LV_USE_SPINNER
3
4  void lv_ex_spinner_1(void)
5  {
6      /*Create a Preloader object*/
```

(下页继续)



图 1: 简单的旋转效果

(续上页)

```
7      lv_obj_t * preload = lv_spinner_create(lv_scr_act(), NULL);
8      lv_obj_set_size(preload, 100, 100);
9      lv_obj_align(preload, NULL, LV_ALIGN_CENTER, 0, 0);
10 }
11
12 #endif
```

49.6 相关 API

TODO

50.1 概述

开关可用于打开/关闭某物。它看起来像一个小滑块。

50.2 零件和样式

开关使用以下部分

- LV_SWITCH_PART_BG : 主要部分
- LV_SWITCH_PART_INDIC : 指标 (虚拟部分)
- LV_SWITCH_PART_KNOB : 旋钮 (虚拟部分)

零件和样式与 [滑杆 \(lv_slider\)](#) 情况相同。阅读其文档以获取详细说明。

50.3 用法

50.3.1 变更状态

可以通过单击或通过下面的函数更改开关的状态:

- `lv_switch_on(switch, LV_ANIM_ON/OFF)` 开

- `lv_switch_off(switch, LV_ANIM_ON/OFF)` 关
- `lv_switch_toggle(switch, LV_ANOM_ON/OFF)` 切换开关的位置

50.3.2 动画时间

切换开关状态时的动画时间可以使用 `lv_switch_set_anim_time(switch, anim_time)` 进行调整。

50.4 事件

除了 通用事件，开关还支持以下 特殊事件：

- **LV_EVENT_VALUE_CHANGED** 在开关更改状态时发送。

了解有关 事件 的更多内容。

50.5 按键处理

开关可处理以下按键：

- **LV_KEY_UP, LV_KEY_RIGHT** 打开滑块
- **LV_KEY_DOWN, LV_KEY_LEFT** 关闭滑块

了解有关 按键 的更多内容。

50.6 范例

50.6.1 简单开关

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_SWITCH
4
5  static void event_handler(lv_obj_t * obj, lv_event_t event)
6  {
7      if(event == LV_EVENT_VALUE_CHANGED) {
8          printf("State: %s\n", lv_switch_get_state(obj) ? "On" : "Off");
9      }
10 }
```

(下页继续)

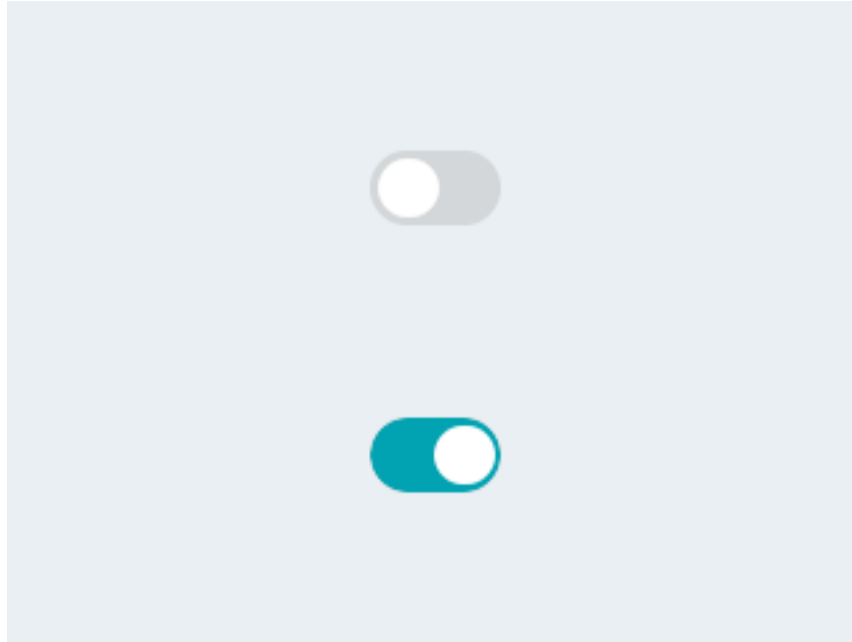


图 1: 简单开关

(续上页)

```
11
12 void lv_ex_switch_1(void)
13 {
14     /*Create a switch and apply the styles*/
15     lv_obj_t *sw1 = lv_switch_create(lv_scr_act(), NULL);
16     lv_obj_align(sw1, NULL, LV_ALIGN_CENTER, 0, -50);
17     lv_obj_set_event_cb(sw1, event_handler);
18
19     /*Copy the first switch and turn it ON*/
20     lv_obj_t *sw2 = lv_switch_create(lv_scr_act(), sw1);
21     lv_switch_on(sw2, LV_ANIM_ON);
22     lv_obj_align(sw2, NULL, LV_ALIGN_CENTER, 0, 50);
23 }
24
25 #endif
```

50.7 相关 API

TODO

51.1 概述

像往常一样，表格是从包含文本的行，列和单元格构建的。

表格对象的权重非常轻，因为仅存储了文本。没有为单元创建任何实际对象，但它们是动态绘制的。

51.2 零件和样式

表格的主要部分称为 `LV_TABLE_PART_BG`。它是一个类似于背景的矩形，并使用所有典型的背景样式属性。

对于单元，有 4 个虚拟部分。每个单元格都有类型 (1, 2, ... 16)，该类型指示要在其上应用哪个部分的样式。单元格部分可以由 `LV_TABLE_PART_CELL1 + x` 引用，其中 `x` 在 0..15 之间。

可以在 `lv_conf.h` 中通过 `LV_TABLE_CELL_STYLE_CNT` 调整单元格类型的数量。默认情况下为 4。默认的 4 种单元格类型部分也使用专用名称进行引用：

- `LV_TABLE_PART_CELL1`
- `LV_TABLE_PART_CELL2`
- `LV_TABLE_PART_CELL3`
- `LV_TABLE_PART_CELL4`

单元格还使用所有典型的背景样式属性。如果单元格内容中有换行符 (`\n`)，则在换行符后将使用线条样式属性绘制水平分隔线。

单元格中的文本样式是从单元格部分或背景部分继承的。

51.3 用法

51.3.1 行 (row) 和列 (column)

要设置行数和列数，请使用 `lv_table_set_row_cnt(table, row_cnt)` 和 `lv_table_set_col_cnt(table, col_cnt)`

51.3.2 宽度 (width) 和高度 (height)

列的宽度可以使用 `lv_table_set_col_width(table, col_id, width)` 设置。Table 对象的总宽度将设置为列宽的总和。

高度是根据单元格式样（字体，填充等）和行数自动计算的。

51.3.3 设定单元格数值类型

单元格只能存储文本，因此在将数字显示在表格中之前，需要将数字转换为文本。

`lv_table_set_cell_value(table, row, col, "Content")`。文本由表保存，因此它甚至可以是局部变量。

可以在 "Value\n60.3" 之类的文本中使用换行符。

51.3.4 对齐

可以使用 `lv_table_set_cell_align(table, row, col, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)` 调整单元格中的文本对齐方式。

51.3.5 单元格类型 (cell_type)

可以使用 4 种不同的单元格类型。每个都有自己的风格。

单元格类型可用于添加不同的样式，例如：

- 表头
- 第一栏
- 突出显示一个单元格
- 等等

可以使用 `lv_table_set_cell_type(table, row, col, type)` 选择类型，类型 (type) 可以为 1、2、3 或 4，对应上面的四种类型。

51.3.6 合并单元格

单元格可以与 `lv_table_set_cell_merge_right(table, col, row, true)` 水平合并。要合并更多相邻的单元格，请对每个单元格应用此功能。

51.3.7 裁剪文字

默认情况下，文字会自动换行以适合单元格的宽度，并且单元格的高度会自动设置。要禁用此功能并保持文本原样，请启用 `lv_table_set_cell_crop(table, row, col, true)`。

51.3.8 滚动

使表格可滚动放置在 页面 (lv_page) 上。

51.4 事件

仅支持 通用事件

了解有关 事件 的更多内容。

51.5 按键处理

对象类型不处理任何键。

了解有关 按键 的更多内容。

51.6 范例

51.6.1 简单的表格

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #if LV_USE_TABLE
3
4  void lv_ex_table_1(void)
```

(下页继续)

NAME	PRICE
Apple	\$7
Banana	\$4
Citron	\$6

图 1: 简单的表格

(续上页)

```

5  {
6      lv_obj_t * table = lv_table_create(lv_scr_act(), NULL);
7      lv_table_set_col_cnt(table, 2);
8      lv_table_set_row_cnt(table, 4);
9      lv_obj_align(table, NULL, LV_ALIGN_CENTER, 0, 0);
10
11     /*Make the cells of the first row center aligned */
12     lv_table_set_cell_align(table, 0, 0, LV_LABEL_ALIGN_CENTER);
13     lv_table_set_cell_align(table, 0, 1, LV_LABEL_ALIGN_CENTER);
14
15     /*Align the price values to the right in the 2nd column*/
16     lv_table_set_cell_align(table, 1, 1, LV_LABEL_ALIGN_RIGHT);
17     lv_table_set_cell_align(table, 2, 1, LV_LABEL_ALIGN_RIGHT);
18     lv_table_set_cell_align(table, 3, 1, LV_LABEL_ALIGN_RIGHT);
19
20     lv_table_set_cell_type(table, 0, 0, 2);
21     lv_table_set_cell_type(table, 0, 1, 2);
22
23
24     /*Fill the first column*/
25     lv_table_set_cell_value(table, 0, 0, "Name");
26     lv_table_set_cell_value(table, 1, 0, "Apple");
27     lv_table_set_cell_value(table, 2, 0, "Banana");

```

(下页继续)

(续上页)

```
28     lv_table_set_cell_value(table, 3, 0, "Citron");
29
30     /*Fill the second column*/
31     lv_table_set_cell_value(table, 0, 1, "Price");
32     lv_table_set_cell_value(table, 1, 1, "$7");
33     lv_table_set_cell_value(table, 2, 1, "$4");
34     lv_table_set_cell_value(table, 3, 1, "$6");
35
36     lv_table_ext_t * ext = lv_obj_get_ext_attr(table);
37     ext->row_h[0] = 20;
38 }
39
40 #endif
```

51.7 相关 API

TODO

52.1 概述

页签对象可用于组织选项卡中的内容。

52.2 零件和样式

Tab 视图对象包含几个部分。主要是 LV_TABVIEW_PART_BG。它是一个矩形容器，用于容纳 Tab 视图的其他部分。

在背景上创建了 2 个重要的实际部分：

- LV_TABVIEW_PART_BG_SCRL 这是 页面 (lv_page) 的可滚动部分。它使选项卡的内容彼此相邻。页面的背景始终是透明的，不能从外部访问。
- LV_TABVIEW_PART_TAB_BG 选项卡按钮是一个 按钮矩阵 (lv_btnmatrix)。单击按钮将 LV_TABVIEW_PART_BG_SCRL 滚动到相关选项卡的内容。可以通过 LV_TABVIEW_PART_TAB_BTN 访问选项卡按钮。选择选项卡时，按钮处于选中状态，可以使用 LV_STATE_CHECKED 设置样式。选项卡的按钮矩阵的高度是根据字体高度加上背景和按钮样式的填充来计算的。

列出的所有部分均支持典型的背景样式属性和填充。

LV_TABVIEW_PART_TAB_BG 还有一个实际部分，即一个指标，称为 LV_TABVIEW_PART_INDIC。它是当前选定选项卡下的一个类似矩形的细对象。当选项卡视图是动画到其它选项卡中的指示器也将被动画。它可以是使用典型背景样式属性的样式。size 样式属性将设置其厚度。

添加新选项卡后，将在 LV_TABVIEW_PART_BG_SCROLL 上为其创建一个页面，并将新按钮添加到 LV_TABVIEW_PART_TAB_BG 按钮矩阵。创建的页面可以用作普通页面，它们具有通常的页面部分。

52.3 用法

52.3.1 添加标签

可以使用 `lv_tabview_add_tab(tabview, "Tab name")` 添加新标签。它将返回指向可以创建选项卡内容的 [页面 \(lv_page\)](#) 对象的指针。

52.3.2 选中标签

通过下面的方法选中一标签：

- 在按钮矩阵部分上单击它
- 滑动
- 使用 `lv_tabview_set_tab_act(tabview, id, LV_ANIM_ON/OFF)` 函数

52.3.3 更改标签的名称

要在运行时更改选项卡 ID 的名称（底层按钮矩阵的显示文本），可以使用函数 `lv_tabview_set_tab_name(tabview, id, name)`。

52.3.4 Tab 按钮的位置

默认情况下，选项卡选择器按钮位于“选项卡”视图的顶部。可以使用 `lv_tabview_set_btns_pos(tabview, LV_TABVIEW_TAB_POS_TOP/BOTTOM/LEFT/RIGHT/NONE)` 进行更改

LV_TABVIEW_TAB_POS_NONE 将隐藏选项卡。

请注意，添加标签后，您无法将标签的位置从顶部或底部更改为左侧或右侧。

52.3.5 动画时间

动画时间由 `lv_tabview_set_anim_time(tabview, anim_time_ms)` 调整。加载新选项卡时使用。

52.3.6 滚动传播

由于选项卡的内容对象是一个 Page，因此它可以从其他类似 Page 的对象接收滚动传播。例如，如果在选项卡的内容上创建了一个文本区域，并且滚动了该文本区域，但到达末尾，则滚动可以传播到内容页面。可以使用“lv_page/textarea_set_scroll_propagation(obj, true)”启用它。

默认情况下，选项卡的内容页面已启用滚动传播，因此，当它们水平滚动时，滚动内容将传播到 LV_TABVIEW_PART_BG_SCRL，这样页面将被滚动。

可以使用 lv_page_set_scroll_propagation(tab_page, false) 禁用手动滑动。

52.4 事件

除了通用事件，页签还支持以下特殊事件：

- **LV_EVENT_VALUE_CHANGED** 通过滑动或单击选项卡按钮选择新选项卡时发送

了解有关事件的更多内容。

52.5 按键处理

复选框可处理以下按键：

- **LV_KEY_RIGHT/LEFT** 选择一个标签
- **LV_KEY_ENTER Change** 更改为所选标签

了解有关按键的更多内容。

52.6 范例

52.6.1 简单的页签

上述效果的示例代码：

```

1  #include ".././../lv_examples.h"
2  #if LV_USE_TABVIEW
3
4  void lv_ex_tabview_1(void)
5  {
6      /*Create a Tab view object*/
7      lv_obj_t *tabview;
8      tabview = lv_tabview_create(lv_scr_act(), NULL);

```

(下页继续)

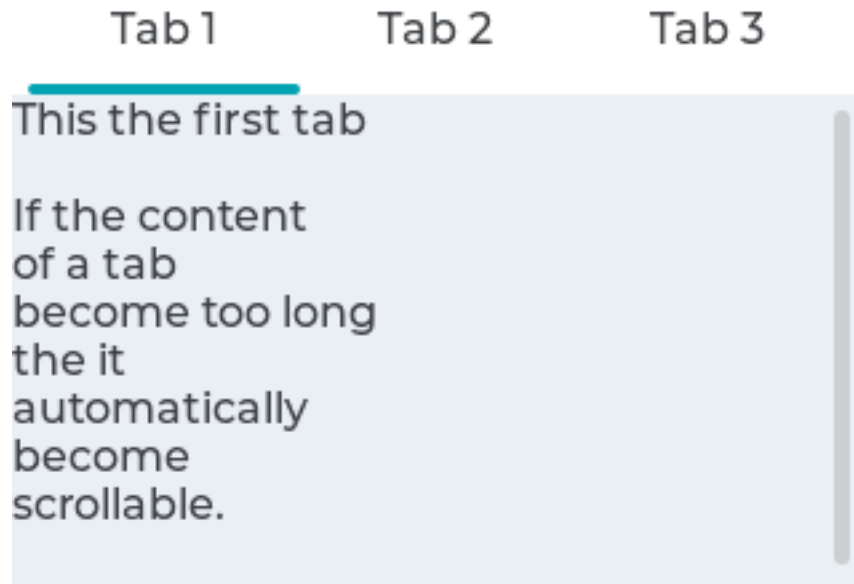


图 1: 简单的页签

(续上页)

```

9
10      /*Add 3 tabs (the tabs are page (lv_page) and can be scrolled*/
11      lv_obj_t *tab1 = lv_tabview_add_tab(tabview, "Tab 1");
12      lv_obj_t *tab2 = lv_tabview_add_tab(tabview, "Tab 2");
13      lv_obj_t *tab3 = lv_tabview_add_tab(tabview, "Tab 3");
14
15
16      /*Add content to the tabs*/
17      lv_obj_t * label = lv_label_create(tab1, NULL);
18      lv_label_set_text(label, "This the first tab\n\n"
19                                "If the content\n"
20                                "of a tab\n"
21                                "become too long\n"
22                                "the it \n"
23                                "automatically\n"
24                                "become\n"
25                                "scrollable.");
26
27      label = lv_label_create(tab2, NULL);
28      lv_label_set_text(label, "Second tab");
29
30      label = lv_label_create(tab3, NULL);
31      lv_label_set_text(label, "Third tab");

```

(下页继续)

(续上页)

```
32     }  
33     #endif
```

52.7 相关 API

TODO

文本框 (lv_textarea)

53.1 概述

文本框是一个带有标签和光标的 [页面 \(lv_page\)](#)。可以在其中添加文本或字符。长行被换行，并且当文本变得足够长时，可以滚动文本区域。

53.2 零件和样式

文本框与 [页面 \(lv_page\)](#) 具有相同的部分。期望 `LV_PAGE_PART_SCROLL`，因为它不能被引用并且始终是透明的。请参阅该页面的详细文档。

除了 `Page` 部分之外，还存在虚拟 `LV_TEXTAREA_PART_CURSOR` 部分来绘制光标。光标的区域始终是当前字符的边界框。可以通过在 `LV_TEXTAREA_PART_CURSOR` 的样式中添加背景色和背景色来创建块光标。创建行光标使光标透明并设置 `border_side` 属性。

53.3 用法

53.3.1 添加文字

可以使用以下命令将文本或字符插入当前光标的位置：

- `lv_textarea_add_char(textarea, 'c')`

- `lv_textarea_add_text(textarea, "insert this text")`

要添加宽字符, 例如 'á', 'ß' 或 CJK 字符 (中日韩统一表意文字), 请使用 `lv_textarea_add_text(ta, "á")`。

`lv_textarea_set_text(ta, "New text")` 更改整个文本。

53.3.2 占位符

可以使用 `lv_textarea_set_placeholder_text(ta, "Placeholder text")` 指定一个占位符文本-当“文本”区域为空时显示。

53.3.3 删除字符

要从当前光标位置的左侧删除字符, 请使用 `lv_textarea_del_char(textarea)`

要从右边删除, 请使用 `lv_textarea_del_char(textarea)`

53.3.4 移动光标

可以使用 `lv_textarea_set_cursor_pos(textarea, 10)` 直接修改光标位置。0 位置表示“在第一个字符之前”, `LV_TA_CURSOR_LAST` 表示“在最后一个字符之后”

可以使用

- `lv_textarea_cursor_right(textarea)`
- `lv_textarea_cursor_left(textarea)`
- `lv_textarea_cursor_up(textarea)`
- `lv_textarea_cursor_down(textarea)`

如果调用 `lv_textarea_set_cursor_click_pos(textarea, true)`, 则光标将跳至单击“文本”区域的位置。

53.3.5 隐藏光标

可以使用 `lv_textarea_set_cursor_hidden(textarea, true)` 隐藏光标。

53.3.6 光标闪烁时间

光标的闪烁时间可以通过 `lv_textarea_set_cursor_blink_time(textarea, time_ms)` 进行调整。

53.3.7 单行模式

可以将“文本”区域配置为以 `lv_textarea_set_one_line(ta, true)` 为一行。在此模式下，高度自动设置为仅显示一行，忽略换行符，并且禁用自动换行。

53.3.8 密码模式

文本区域支持可以通过 `lv_textarea_set_pwd_mode(textarea, true)` 启用的密码模式。

如果字体中存在 `•` (Bullet, U+2022) 字符，则一段时间后或输入新字符后，输入的字符将转换为该字符。如果 `•` 不存在，将使用 `*`。

在密码模式下 `lv_textarea_get_text(textarea)` 给出真实文本，而不是项目符号字符。

可见时间可以使用 `lv_textarea_set_pwd_show_time(textarea, time_ms)` 进行调整。

53.3.9 文字对齐

可以使用 `lv_textarea_set_text_align(textarea, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)` 将文本左，中或右对齐。

在单行模式下，仅当文本保持对齐时才能水平滚动文本。

53.3.10 字符过滤

可以使用 `lv_textarea_set_accepted_chars(ta, "0123456789.+")` 设置可接受字符的列表。其他字符将被忽略。

53.3.11 最大文字长度

最大字符数可以通过 `lv_textarea_set_max_length(textarea, max_char_num)` 进行限制

53.3.12 长文本

如果“文本”区域中的文本很长 (例如 >20k 个字符), 则其滚动和绘制速度可能会很慢。但是, 通过在 `lv_conf.h` 中启用 `LV_LABEL_LONG_TXT_HINT 1` 可以极大地改善它。它将保存一些有关标签的信息, 以加快其绘制速度。使用 `LV_LABEL_LONG_TXT_HINT`, 滚动和绘图将与使用“普通”短文本一样快。

53.3.13 选择文字

如果通过 `lv_textarea_set_text_sel(textarea, true)` 启用, 则可以选择一部分文本。就像用鼠标在 PC 上选择文本时一样。

53.3.14 滚动条

可以根据 `lv_textarea_set_scrollbar_mode(textarea, LV_SCROLLBAR_MODE_...)` 设置的不同策略显示滚动条。在 `Page` 对象中了解更多信息。

53.3.15 滚动传播

当“文本”区域在另一个可滚动对象 (如“页面”) 上滚动并且滚动已到达“文本”区域的边缘时, 滚动可以传播到父对象。也就是说, 当“文本”区域可以进一步滚动时, 父级将被滚动。

可以使用 `lv_ta_set_scroll_propagation(ta, true)` 启用它。

在 `页面 (lv_page)` 对象中了解更多信息。

53.3.16 边缘闪烁

当“文本”区域滚动到边缘时, 如果通过 `lv_ta_set_edge_flash(ta, true)` 启用, 则可以显示类似 Flash 动画的圆圈

53.4 事件

除了 `通用事件`, 文本框还支持以下 `特殊事件`:

- **LV_EVENT_INSERT** 在插入字符或文本之前发送。事件数据是计划插入的文本。`lv_ta_set_insert_replace(ta, “新文本”)` 替换要插入的文本。新文本不能位于局部变量中, 该局部变量会在事件回调存在时被销毁。“”表示请勿插入任何内容。
- **LV_EVENT_VALUE_CHANGED** 当文本区域的内容已更改时。
- **LV_EVENT_APPLY** 当 `LV_KEY_ENTER` 发送到处于单行模式的文本区域时。

了解有关 `事件` 的更多内容。

53.5 按键处理

文本框可处理以下按键：

- **LV_KEY_UP/DOWN/LEFT/RIGHT** 移动光标
- **Any character** 将字符添加到当前光标位置

了解有关 [按键](#) 的更多内容。

53.6 范例

53.6.1 简单的文本框

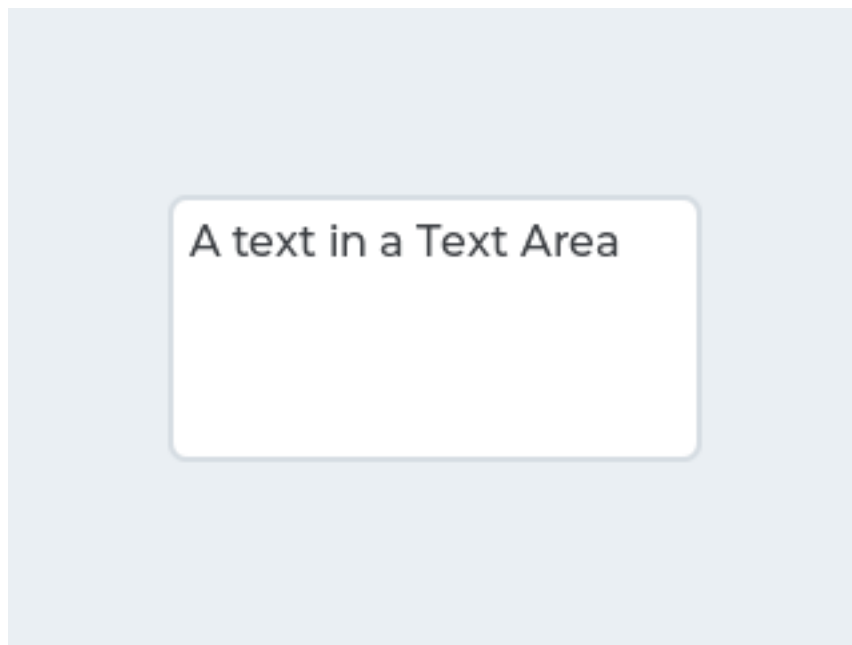


图 1: 简单的文本框

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #include <stdio.h>
3  #if LV_USE_TEXTAREA
4
5  lv_obj_t * ta1;
6
7  static void event_handler(lv_obj_t * obj, lv_event_t event)
8  {
```

(下页继续)

(续上页)

```

9         if(event == LV_EVENT_VALUE_CHANGED) {
10             printf("Value: %s\n", lv_textarea_get_text(obj));
11         }
12         else if(event == LV_EVENT_LONG_PRESSED_REPEAT) {
13             /*For simple test: Long press the Text are to add the text below*/
14             const char * txt = "\n\nYou can scroll it if the text is long_
↪enough.\n";

15             static uint16_t i = 0;
16             if(txt[i] != '\0') {
17                 lv_textarea_add_char(ta1, txt[i]);
18                 i++;
19             }
20         }
21     }
22
23     void lv_ex_textarea_1(void)
24     {
25         ta1 = lv_textarea_create(lv_scr_act(), NULL);
26         lv_obj_set_size(ta1, 200, 100);
27         lv_obj_align(ta1, NULL, LV_ALIGN_CENTER, 0, 0);
28         lv_textarea_set_text(ta1, "A text in a Text Area"); /*Set an initial_
↪text*/
29         lv_obj_set_event_cb(ta1, event_handler);
30     }
31
32     #endif

```

53.6.2 密码模式的文本区域

上述效果的示例代码：

```

1     #include "../lv_examples.h"
2     #include <stdio.h>
3     #if LV_USE_TEXTAREA && LV_USE_KEYBOARD
4
5     static void ta_event_cb(lv_obj_t * ta, lv_event_t event);
6
7     static lv_obj_t * kb;
8
9     void lv_ex_textarea_2(void)
10    {
11        /* Create the password box */

```

(下页继续)

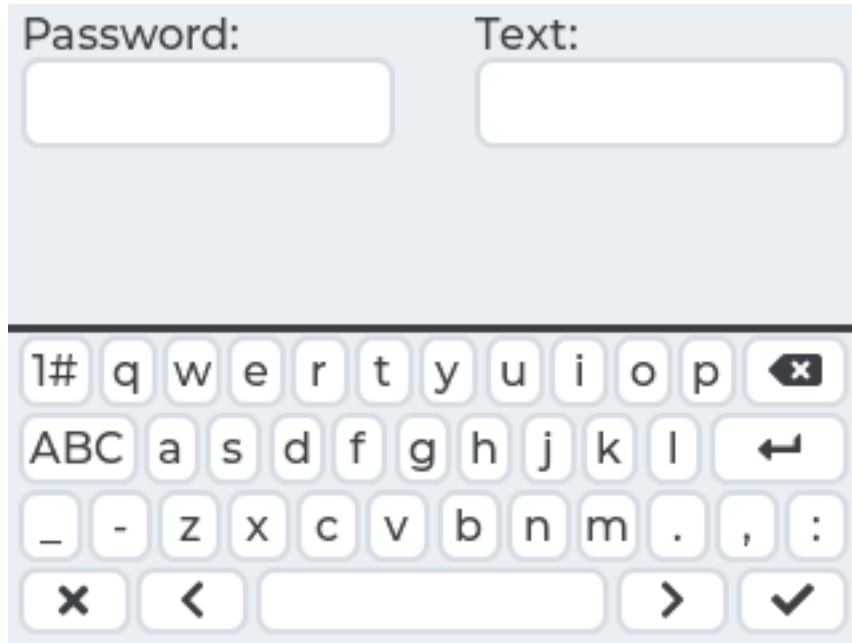


图 2: 简单的文本框

(续上页)

```

12     lv_obj_t * pwd_ta = lv_textarea_create(lv_scr_act(), NULL);
13     lv_textarea_set_text(pwd_ta, "");
14     lv_textarea_set_pwd_mode(pwd_ta, true);
15     lv_textarea_set_one_line(pwd_ta, true);
16     lv_textarea_set_cursor_hidden(pwd_ta, true);
17     lv_obj_set_width(pwd_ta, LV_HOR_RES / 2 - 20);
18     lv_obj_set_pos(pwd_ta, 5, 20);
19     lv_obj_set_event_cb(pwd_ta, ta_event_cb);
20
21     /* Create a label and position it above the text box */
22     lv_obj_t * pwd_label = lv_label_create(lv_scr_act(), NULL);
23     lv_label_set_text(pwd_label, "Password:");
24     lv_obj_align(pwd_label, pwd_ta, LV_ALIGN_OUT_TOP_LEFT, 0, 0);
25
26     /* Create the one-line mode text area */
27     lv_obj_t * oneline_ta = lv_textarea_create(lv_scr_act(), pwd_ta);
28     lv_textarea_set_pwd_mode(oneline_ta, false);
29     lv_textarea_set_cursor_hidden(oneline_ta, true);
30     lv_obj_align(oneline_ta, NULL, LV_ALIGN_IN_TOP_RIGHT, -5, 20);
31
32
33     /* Create a label and position it above the text box */
34     lv_obj_t * oneline_label = lv_label_create(lv_scr_act(), NULL);

```

(下页继续)

(续上页)

```

35     lv_label_set_text(online_label, "Text:");
36     lv_obj_align(online_label, online_ta, LV_ALIGN_OUT_TOP_LEFT, 0, 0);
37
38     /* Create a keyboard */
39     kb = lv_keyboard_create(lv_scr_act(), NULL);
40     lv_obj_set_size(kb, LV_HOR_RES, LV_VER_RES / 2);
41
42     lv_keyboard_set_textarea(kb, pwd_ta); /* Focus it on one of the text_
↪areas to start */
43     lv_keyboard_set_cursor_manage(kb, true); /* Automatically show/hide_
↪cursors on text areas */
44 }
45
46 static void ta_event_cb(lv_obj_t * ta, lv_event_t event)
47 {
48     if(event == LV_EVENT_CLICKED) {
49         /* Focus on the clicked text area */
50         if(kb != NULL)
51             lv_keyboard_set_textarea(kb, ta);
52     }
53
54     else if(event == LV_EVENT_INSERT) {
55         const char * str = lv_event_get_data();
56         if(str[0] == '\n') {
57             printf("Ready\n");
58         }
59     }
60 }
61
62 #endif

```

53.6.3 文字自动格式化

上述效果的示例代码：

```

1     #include ".././../lv_examples.h"
2     #include <stdio.h>
3     #if LV_USE_TEXTAREA && LV_USE_KEYBOARD
4
5     static void ta_event_cb(lv_obj_t * ta, lv_event_t event);
6
7     static lv_obj_t * kb;

```

(下页继续)



图 3: 文字自动格式化

(续上页)

```

8
9  /**
10   * Automatically format text like a clock. E.g. "12:34"
11   * Add the ':' automatically.
12   */
13 void lv_ex_textarea_3(void)
14 {
15     /* Create the text area */
16     lv_obj_t * ta = lv_textarea_create(lv_scr_act(), NULL);
17     lv_obj_set_event_cb(ta, ta_event_cb);
18     lv_textarea_set_accepted_chars(ta, "0123456789:");
19     lv_textarea_set_max_length(ta, 5);
20     lv_textarea_set_one_line(ta, true);
21     lv_textarea_set_text(ta, "");
22
23
24     /*Create a custom map for the keyboard*/
25
26     static const char * kb_map[] = {
27         "1", "2", "3", " ", "\n",
28         "4", "5", "6", " ", "\n",
29         "7", "8", "9", LV_SYMBOL_BACKSPACE, "\n",
30         "0", LV_SYMBOL_LEFT, LV_SYMBOL_RIGHT, " ", ""

```

(下页继续)

(续上页)

```

31     };
32
33     static const lv_btnmatrix_ctrl_t kb_ctrl[] = {
34         LV_BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_NO_REPEAT, LV_
↵BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_HIDDEN,
35         LV_BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_NO_REPEAT, LV_
↵BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_HIDDEN,
36         LV_BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_NO_REPEAT, LV_
↵BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_NO_REPEAT,
37         LV_BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_NO_REPEAT, LV_
↵BTNMATRIX_CTRL_NO_REPEAT, LV_BTNMATRIX_CTRL_HIDDEN,
38     };
39
40     /* Create a keyboard*/
41     kb = lv_keyboard_create(lv_scr_act(), NULL);
42     lv_obj_set_size(kb, LV_HOR_RES, LV_VER_RES / 2);
43     lv_keyboard_set_mode(kb, LV_KEYBOARD_MODE_NUM);
44     lv_keyboard_set_map(kb, LV_KEYBOARD_MODE_NUM, kb_map);
45     lv_keyboard_set_ctrl_map(kb, LV_KEYBOARD_MODE_NUM, kb_ctrl);
46     lv_keyboard_set_textarea(kb, ta);
47 }
48
49 static void ta_event_cb(lv_obj_t * ta, lv_event_t event)
50 {
51     if(event == LV_EVENT_VALUE_CHANGED) {
52         const char * txt = lv_textarea_get_text(ta);
53         if(txt[3] == ':') {
54             lv_textarea_del_char(ta);
55         }
56         else if(txt[0] >= '0' && txt[0] <= '9' &&
57             txt[1] >= '0' && txt[1] <= '9' &&
58             txt[2] != ':')
59         {
60             lv_textarea_set_cursor_pos(ta, 2);
61             lv_textarea_add_char(ta, ':');
62         }
63     }
64 }
65
66 #endif

```

53.7 相关 API

TODO

平铺视图 (lv_tileview)

54.1 概述

平铺视图 (Tileview) 是一个容器对象，其中的元素（称为图块）可以以网格形式排列。通过滑动，用户可以在图块之间导航。

如果 Tileview 是屏幕尺寸的，它将提供可能已经在智能手表上看到的用户界面。

54.2 零件和样式

Tileview 与 页面 (lv_page) 具有相同的部分。期望 LV_PAGE_PART_SCROLL，因为它不能被引用并且始终是透明的。请参阅该页面的详细文档。

54.3 用法

54.3.1 有效区域

磁贴不必在每个元素都存在的地方形成完整的网格。网格中可以有孔，但必须是连续的，即不能有空行或列。

使用 `lv_tileview_set_valid_positions(tileview, valid_pos_array, array_len)` 可以设置有效位置。仅可以滚动到该位置。0,0 索引表示左上方的图块。例如。`lv_point_t valid_pos_array[]`

= {{0,0}, {0,1}, {1,1}, {{LV_COORD_MIN, LV_COORD_MIN}} 给出了”L”形的图块视图。它指示 {1,1} 中没有图块，因此用户无法在此处滚动。

换句话说，`valid_pos_array` 告诉磁贴在哪里。可以即时更改它以禁用特定图块上的某些位置。例如，可能存在一个 2x2 网格，其中添加了所有图块，但第一行 ($y = 0$) 作为“主行”，第二行 ($y = 1$) 包含其上方图块的选项。假设水平滚动只能在主行中进行，而在第二行中的选项之间则不可能进行。在这种情况下，`valid_pos_array` 需要改变时，选择一个新的主瓦：

- 对于第一个主磁贴：{0,0}, {0,1}, {1,0} 以禁用 {1,1} 选项磁贴
- 对于第二个主磁贴：{0,0}, {1,0}, {1,1} 以禁用 {0,1} 选项磁贴

54.3.2 设置瓷砖

设置当前可见的图块使用：`lv_tileview_set_tile_act(tileview, x_id, y_id, LV_ANIM_ON/OFF)`

54.3.3 添加元素

要添加元素，只需在 `Tileview` 上创建一个对象并将其手动定位到所需位置即可。

`lv_tileview_add_element(tileview, element)` 应该用来使 `Tileview` 滚动(拖动)其元素一个。例如，如果图块上有一个按钮，则需要将该按钮显式添加到 `Tileview` 中，以使用户也可以使用该按钮滚动 `Tileview`。

54.3.4 滚动传播

页面状对象（如列表 (`lv_list`)）的滚动传播功能在这里可以很好地使用。例如，可以有一个完整的列表 (`lv_list`)，当列表到达最顶部或最底部时，用户将改为滚动图块视图。

54.3.5 动画时间

平铺视图的动画时间可以使用 `lv_tileview_set_anim_time(tileview, anim_time)` 进行调整。

在以下情况下应用动画

- 使用 `lv_tileview_set_tile_act` 选择一个新图块
- 当前磁贴稍微滚动然后释放（还原原始标题）
- 当前磁贴滚动超过一半大小，然后释放（移至下一个磁贴）

54.3.6 边缘闪光

当滚动到达的图块视图击中无效位置或图块视图的末尾时，可以添加“边缘闪光”效果。

使用 `lv_tileview_set_edge_flash(tileview, true)` 启用此功能。

54.4 事件

除了通用事件，平铺视图还支持以下特殊事件：

- ****LV_EVENT_VALUE_CHANGED**** 当加载了带有滚动或 `lv_tileview_set_act` 的新图块时发送。将事件数据设置为 `valid_pos_array` 中新图块的索引（其类型为 `uint32_t *`）

了解有关事件的更多内容。

54.5 按键处理

平铺视图可处理以下按键：

- **LV_KEY_UP, LV_KEY_RIGHT** 将滑块的值增加 1
- **LV_KEY_DOWN, LV_KEY_LEFT** 将滑块的值减 1

了解有关按键的更多内容。

54.6 范例

54.6.1 包含内容的平铺视图

上述效果的示例代码：

```

1  #include "../lv_examples.h"
2  #if LV_USE_TILEVIEW
3
4  void lv_ex_tileview_1(void)
5  {
6      static lv_point_t valid_pos[] = {{0,0}, {0, 1}, {1,1}};
7      lv_obj_t *tileview;
8      tileview = lv_tileview_create(lv_scr_act(), NULL);
9      lv_tileview_set_valid_positions(tileview, valid_pos, 3);
10     lv_tileview_set_edge_flash(tileview, true);
11

```

(下页继续)

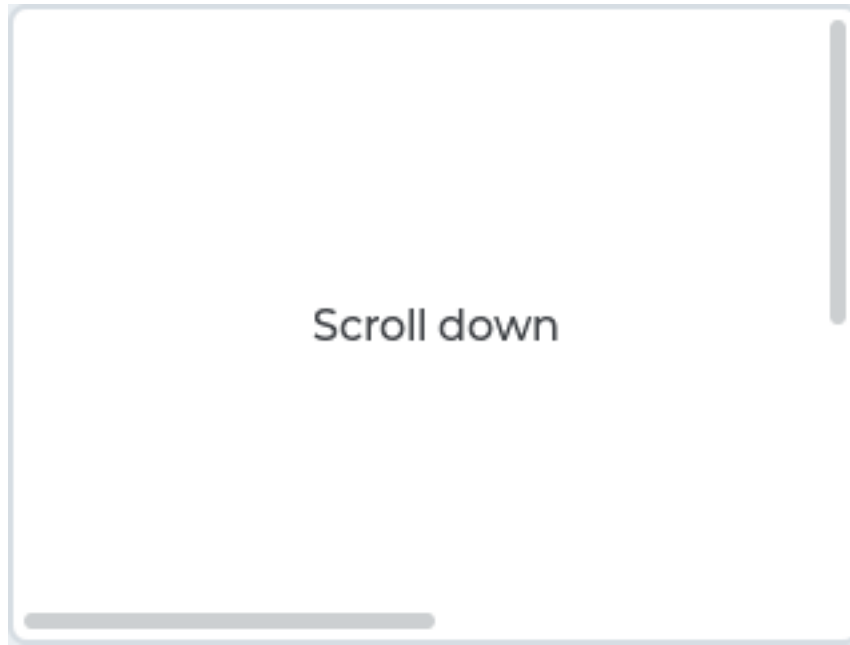


图 1: 包含内容的平铺视图

(续上页)

```
12     lv_obj_t * tile1 = lv_obj_create(tileview, NULL);
13     lv_obj_set_size(tile1, LV_HOR_RES, LV_VER_RES);
14     lv_tileview_add_element(tileview, tile1);
15
16     /*Tile1: just a label*/
17     lv_obj_t * label = lv_label_create(tile1, NULL);
18     lv_label_set_text(label, "Scroll down");
19     lv_obj_align(label, NULL, LV_ALIGN_CENTER, 0, 0);
20
21     /*Tile2: a list*/
22     lv_obj_t * list = lv_list_create(tileview, NULL);
23     lv_obj_set_size(list, LV_HOR_RES, LV_VER_RES);
24     lv_obj_set_pos(list, 0, LV_VER_RES);
25     lv_list_set_scroll_propagation(list, true);
26     lv_list_set_scrollbar_mode(list, LV_SCROLLBAR_MODE_OFF);
27
28     lv_list_add_btn(list, NULL, "One");
29     lv_list_add_btn(list, NULL, "Two");
30     lv_list_add_btn(list, NULL, "Three");
31     lv_list_add_btn(list, NULL, "Four");
32     lv_list_add_btn(list, NULL, "Five");
33     lv_list_add_btn(list, NULL, "Six");
34     lv_list_add_btn(list, NULL, "Seven");
```

(下页继续)

(续上页)

```
35     lv_list_add_btn(list, NULL, "Eight");
36
37     /*Tile3: a button*/
38     lv_obj_t * tile3 = lv_obj_create(tileview, tile1);
39     lv_obj_set_pos(tile3, LV_HOR_RES, LV_VER_RES);
40     lv_tileview_add_element(tileview, tile3);
41
42     lv_obj_t * btn = lv_btn_create(tile3, NULL);
43     lv_obj_align(btn, NULL, LV_ALIGN_CENTER, 0, 0);
44     lv_tileview_add_element(tileview, btn);
45     label = lv_label_create(btn, NULL);
46     lv_label_set_text(label, "No scroll up");
47 }
48
49 #endif
```

54.7 相关 API

TODO

55.1 概述

窗口是类似 容器 (lv_cont) 的对象，由带有标题和按钮的标题以及内容区域构建而成。

55.2 零件和样式

主要部分是 LV_WIN_PART_BG，它包含另外两个实际部分：

- LV_WIN_PART_HEADER 顶部的标题容器，带有标题和控制按钮
- LV_WIN_PART_CONTENT_SCRL 页眉下方内容的页面可滚动部分。

除此之外，LV_WIN_PART_CONTENT_SCRL 还有一个滚动条，称为 LV_WIN_PART_CONTENT_SCRL。阅读 页面 (lv_page) 的文档以获取有关滚动条的更多详细信息。

所有部分均支持典型的背景属性。标题使用标题部分的 Text 属性。

控制按钮的高度为：标头高度-标头 padding_top-标头 padding_bottom。

55.2.1 窗口标题

窗口上有一个标题，可以通过以下方式修改 `lv_win_set_title(win, "New title")`

55.2.2 控制按钮

可以使用以下命令将控制按钮添加到窗口标题的右侧: `lv_win_add_btn_right(win, LV_SYMBOL_CLOSE)` , 要在窗口标题的左侧添加按钮, 请使用 `lv_win_add_btn_left(win, LV_SYMBOL_CLOSE)` 。第二个参数是图像源, 因此它可以是符号, 指向 `lv_img_dsc_t` 变量的指针或文件的路径。

可以使用 `lv_win_add_btn_left(win, LV_SYMBOL_CLOSE)` 设置按钮的宽度。如果 `w == 0` , 则按钮将为正方形。

`lv_win_close_event_cb` 可以用作关闭窗口的事件回调。

55.2.3 滚动条

可以通过 `lv_win_set_scrollbar_mode(win, LV_SCROLLBAR_MODE_...)` 设置滚动条行为。有关详细信息, 请参见 [页面 \(lv_page\)](#) 。

55.2.4 手动滚动和聚焦

要直接滚动窗口, 可以使用 `lv_win_scroll_hor(win, dist_px)` 或 `lv_win_scroll_ver(win, dist_px)` 。

要使窗口在其上显示对象, 请使用 `lv_win_focus(win, child, LV_ANIM_ON/OFF)` 。

滚动和焦点动画的时间可以使用 `lv_win_set_anim_time(win, anim_time_ms)` 进行调整

55.2.5 布局

要设置内容的布局, 请使用 `lv_win_set_layout(win, LV_LAYOUT_...)` 。有关详细信息, 请参见 [容器 \(lv_cont\)](#) 。

55.3 事件

仅支持 通用事件

了解有关 [事件](#) 的更多内容。

55.4 按键处理

窗口可处理以下按键：

- **LV_KEY_RIGHT/LEFT/UP/DOWN** 滚动页面

了解有关 [按键](#) 的更多内容。

55.5 范例

55.5.1 简单的窗口

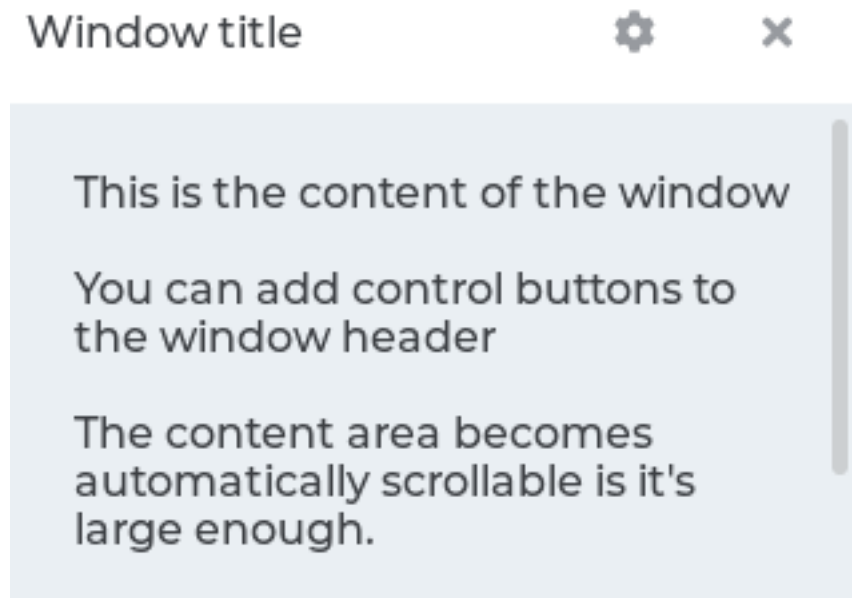


图 1: 简单的窗口

上述效果的示例代码：

```
1  #include "../../lv_examples.h"
2  #if LV_USE_WIN
3
4  void lv_ex_win_1(void)
5  {
6      /*Create a window*/
7      lv_obj_t * win = lv_win_create(lv_scr_act(), NULL);
8      lv_win_set_title(win, "Window title");          /*Set the_
↪title*/
9  }
```

(下页继续)

(续上页)

```

10
11      /*Add control button to the header*/
12      lv_obj_t * close_btn = lv_win_add_btn(win, LV_SYMBOL_CLOSE);          /
13      ↪/*Add close button and use built-in close action*/
14      lv_obj_set_event_cb(close_btn, lv_win_close_event_cb);
15      lv_win_add_btn(win, LV_SYMBOL_SETTINGS);          /*Add a setup button*/
16
17      /*Add some dummy content*/
18      lv_obj_t * txt = lv_label_create(win, NULL);
19      lv_label_set_text(txt, "This is the content of the window\n\n"
20                                "You can add control buttons_
21                                "the window header\n\n"
22                                "The content area becomes\n"
23                                "automatically scrollable is it
24                                "large enough.\n\n"
25                                " You can scroll the content\n"
26                                "See the scroll bar on the_
27                                "right!");
28      }
29
30      #endif

```

55.6 相关 API

TODO

CHAPTER 56

LVGL 项目实战 (Windows 模拟器)

下载资料

LVGL 项目实战 (基于 Linux 开发板)

[下载资料](#)

LVGL 项目实战 (基于 STM32F103)

[下载资料](#)

公司简介

深圳百问网科技有限公司 (百问网) 是中国一个专注于嵌入式 Linux 培训视频的科技公司。

致力于提供“教材、答疑、开发板、仿真器”一站式嵌入式学习解决方案：

1. 2008 年出版的《嵌入式 Linux 应用开发完全手册》销量已经过万，在 china-pub、当当等网上书店一直名列前茅，成为最畅销的 Linux 开发书籍之一
2. 2012 年发布的“韦东山嵌入式系列视频”配合所生产的“JZ2440 开发板”乃嵌入式学习利器，让数以万计的嵌入式爱好者入门，并在中国的嵌入式 Linux 领域拥有良好的声誉
3. 所生产的 OpenJTAG 是目前唯一能烧写 s3c2410/2440 6410 开发板的 NAND Flash 的仿真器
4. “论坛答疑+ wiki + qq 群 + 邮箱”负责、永久的售后模式，确保你售后无忧

联系方式

- 百问网官方 wiki: <http://wiki.100ask.org>
- 百问网官方论坛: <http://bbs.100ask.net>
- 百问网官网: <http://www.100ask.net>
- 微信公众号: 百问科技
- CSDN: 韦东山
- B 站: 韦东山
- 知乎: 韦东山嵌入式
- 微博: 百问科技
- 电子发烧友学院: 韦东山

公司名称: 深圳百问网科技有限公司 电话: 0755-86200561 淘宝: 100ask.taobao.com 技术支持
邮箱: weidongshan@qq.com 地址: 广东省深圳市龙岗区布吉南湾街道平吉大道建国大厦 B1505 邮编:
518114

微信公众号 (baiwenkeji) | 视频教程在线学习平台

《嵌入式 Linux 应用开发完全手册第 2 版 — 韦东山全系列视频文档全集》