

适用 STM32 和 51 单片机各类 GUI

作者:向仔州

ZLGGUI 移植.....	8
ZLGGUI 图标菜单使用.....	13
解决 ZLGGUI 字体大小和字体动态显示问题.....	17
在 ZLGGUI 里面加中文字模.....	18
STM32F103 移植 LittlevGL 图形库.....	20
lv_conf.h 配置文件详解.....	32
littleVGL 基础 API 接口.....	37
lv_obj_t* lv_obj_create(....) //创建一个对象.....	37
lv_res_t lv_obj_del(....) //立即删除对象.....	37
lv_obj_del_async(....) //异步删除对象, 就是必须要等到执行 lv_task_handler()才会删除掉对象。.....	38
lv_obj_clean(....) //删除当前对象下的所有子对象.....	39
lv_obj_move_background(....) //将对象的层级后置.....	40
lv_obj_move_foreground(....) //将对象层级前置也就是按钮 2 在前面这就不演示了.....	40
lv_obj_set_pos(....) //对象移动坐标设置.....	41
lv_obj_set_size(....) //设置对象图形大小.....	41
lv_obj_align(....) //对齐函数.....	42
lv_label 标签控件.....	44
lv_label_create(....) //创建标签对象, 默认左上角.....	44
lv_label_set_body_draw(....) //打开 label 对象背景颜色开关.....	44
static inline void lv_label_set_style(....)//设置标签背景颜色.....	44
lv_label_set_text(....) //给标签写入自定义字符串.....	45
lv_label_set_array_text(....) //传入数组变量来显示 label.....	45
lv_label_set_static_text(....)//静态标签字符显示, 如果你的字符只需要固定显示, 不需要更换字符内容就用静态字符显示。因为系统不会为你字符开辟内存。.....	45
lv_label_set_long_mode(....) //label 文本模式设置.....	46
lv_label_set_anim_speed(....) //标签滚动速度设置.....	47
lv_label_set_align(....) //控件对齐方式.....	47
lv_label_set_recolor(....) //文本重新设置颜色功能开关.....	47
lv_label_ins_text(....) //插入文本.....	48
lv_style 样式系统, 界面美观设计.....	48
static inline void lv_label_set_style(....)//样式设置.....	48
body..... 对象背景.....	49
lv_color_t body.main_color //我们设置主背景颜色.....	49
lv_color_t body.grad_color //如果 grad_color != main_color , 那么背景色默认是渐变颜色.....	49
body.radius //设置圆角半径.....	49
body.radius //根据对象高度自动设置全圆角.....	49
body.opa //背景透明度.....	50
body.border.color //边框设置.....	50
body.border.width //边框宽度.....	50
body.border.part //边框类型.....	50
body.border.opa //边框透明度.....	50
body.shadow.color //对象背景阴影.....	51
body.shadow.width //对象阴影宽度.....	51
body.shadow.type //设置四边都有阴影.....	51

body.padding.top //文本与背景边框上部内边距.....	51	
text.color //文本颜色设置.....	52	
text.font //字体大小设置.....	52	
text.letter_space //字符之间列间距.....	52	
text.line_space //字符之间行间距.....	52	
text.opa //字体透明度.....	52	
LV_COLOR_MAKE(r , g , b) //自定义颜色.....	52	
QTCreator 做 littleVGL 模拟器(为后面按钮事件驱动做铺垫).....	53	
lv_cont 容器(可以当窗体使用).....	59	
lv_obj_t * lv_cont_create(...) //创建容器.....	59	
void lv_cont_set_layout(...)	//这其实就是容器的布局管理器.....	59
static inline void lv_cont_set_fit(...)	//容器大小变化自动设置.....	59
lv_cont 容器样式设置.....	55	
body.padding.inner //容器子对象之间间距设置.....	55	
body.padding.top //子对象与容器顶部边的距离.....	55	
body.padding.left //子对象与容器左边间距.....	55	
lv_btn 按钮控件.....	62	
lv_obj_t * lv_btn_create(...) //创建按钮.....	62	
void lv_btn_set_toggle(...)	//将按钮设置为切换按钮.....	62
void lv_btn_set_state(...)	//设置按钮状态, 如按下状态, 松开状态这些.....	62
void lv_btn_set_ink_in_time(...)	//设置按钮点击后波纹开始时长.....	65
void lv_btn_set_ink_wait_time(...)	//设置波纹等待维持时长.....	65
void lv_btn_set_ink_out_time(...)	//设置波纹出场动画时长.....	65
如何设置 lv_btn 按钮字体显示.....	66	
void lv_obj_set_drag(...)	//对象具备拖拽移动功能.....	66
设置按钮颜色样式(可以分为按下和松开两种样式).....	67	
Events 事件(用按钮被按下处理事件做演示).....	68	
void lv_obj_set_event_cb(...)	//注册对象具备事件发生功能.....	69
typedef void (*lv_event_cb_t)(...)	//回调函数定义要求.....	69
littleVGL 移植触摸屏驱动.....	73	
确认触摸屏驱动在 LCD 屏幕上能画画.....	73	
LittleVGL 支持中文汉字显示.....	78	
实现预先指定中文内容显示, 在 PC 模拟器上面实验.....	78	
实现预先指定中文内容显示, 在 STM32 上面实验.....	80	
用中文显示方式在控件上显示字体, 按钮为例(前提你已经按照要求解决了 UTF-8 编码问题).....	83	
让程序自动修改字体.....	83	
手动发送 Event 事件触发方法.....	84	
给固定控件增加新的事件发送功能.....	84	
lv_res_t lv_event_send(...) //固定控件自定义发送事件函数.....	84	
const void * lv_event_get_data(void)	//在事件回调函数中执行接收发送事件函数自定义的数据.....	84
没有控件, 自己定义事件发送功能.....	85	
lv_res_t lv_event_send_func(...) // 自定义事件发送回调函数.....	85	
LED 灯软件界面显示.....	86	
lv_obj_t * lv_led_create(...) //创建 LED 对象.....	86	
void lv_led_set_bright(...)	//设置 LED 亮度.....	86
void lv_led_on(...)	//点亮 LED.....	86
画圆弧.....	87	
lv_obj_t * lv_arc_create(...) //创建弧形.....	87	
设置弧线半径在 littleVGL 是设置弧形长宽大小来解决.....	87	

<code>void lv_arc_set_angles(...)</code> //设置弧形起点和终点.....	87
进度条使用	88
<code>lv_obj_t * lv_bar_create(...)</code> //创建进度条.....	88
<code>void lv_bar_set_value(...)</code> //设置进度条值.....	88
<code>void lv_bar_set_anim_time(...)</code> //设置动画时长.....	89
<code>void lv_bar_set_range(...)</code> //设置进度条范围.....	89
<code>void lv_bar_set_style(...)</code> //进度条有两个样式.....	89
重点讲下进度条边框值设置	90
水平改为垂直进度条实现	90
复选框使用	91
<code>lv_obj_t * 返回句柄 = lv_cb_create(...)</code> //创建复选框.....	91
<code>lv_cb_set_text(...)</code> //设置复选框 动态文本.....	91
<code>lv_cb_set_static_text(...)</code> //静态文本.....	92
<code>lv_cb_set_checked(...)</code> //复选框默认为选中状态.....	92
复选框选中后，触发事件回调函数实现	92
<code>lv_obj_set_event_cb(...)</code> //创建复选框事件回调函数.....	92
返回值 int = lv_cb_is_checked(...) //获取复选框选中/不选中状态，选中返回 1 不选中返回 0.....	92
画线使用	93
<code>lv_obj_t *返回对象 = lv_line_create(...)</code> //创建线条.....	93
<code>lv_line_set_auto_size(...)</code> //如果要手动设置线条大小，先关闭掉默认的线条自动大小.....	93
<code>lv_obj_set_size(...)</code> //手动设置线条大小(一般都是自动设置),这里只是尝试.....	93
<code>lv_line_set_points(...)</code> //设置线条坐标点,有了坐标点才有线条.....	93
滑块控件	95
<code>lv_obj_t *滑块对象 = lv_slider_create(...)</code> //创建滑块(默认范围 0 ~ 100).....	95
<code>lv_slider_set_value(...)</code> //设置滑块当前值，开启动画效果.....	95
<code>lv_slider_set_anim_time(...)</code> //设置动画时长.....	95
<code>lv_slider_set_range(...)</code> //设置滑动范围.....	96
滑块事件触发回调函数使用	96
返回值 int16_t = lv_slider_get_value(滑块对象) //获取滑块当前值.....	96
滑块样式设置	97
<code>lv_obj_align(...)</code> //将当前对象与另外一个对象对齐.....	98
开关控件	99
<code>lv_obj_t *对象 = lv_sw_create(...)</code> //创建开关.....	99
<code>lv_sw_on(...)</code> //开机，开关默认打开，使用动画效果.....	99
<code>lv_sw_off(...)</code> //开机，开关默认关闭.....	99
开关，打开/关闭事件回调函数	99
返回值 bool = lv_sw_get_state(开关对象) //获取开关打开/关闭状态，打开返回 1,关闭返回 0.....	99
开关样式设置	100
<code>lv_sw_set_style(开关对象,样式设置,样式内容)</code> //开关样式设置.....	100
矩阵按钮	101
<code>lv_obj_t *矩阵对象 = lv_btnm_create(...)</code> //创建矩阵按钮.....	101
<code>lv_btnm_set_map(...)</code> //将按钮字符加载进矩阵.....	101
矩阵按钮大小设置	101
“\n” 使用换行符显示两排按钮.....	102
<code>lv_btnm_set_btn_ctrl(...)</code> //设置某单个按钮的控制属性.....	102
矩阵按钮事件回调函数实现	102
取消矩阵按钮一直被压着触发事件回调.....	102
指定矩阵按钮处于禁用状态.....	102

<code>lv_btm_clear_btn_ctrl(...)//清除指定单个按钮控制属性</code>	103
<code>lv_btm_set_ctrl_map(...)//设置控制映射表，批量修改按钮属性</code>	103
<code>lv_btm_set_one_toggle(...)//根据映射表，只能有一个按钮被切换</code>	105
返回值 <code>uint16_t = lv_btm_get_active_btn(矩阵对象) //获取矩阵按钮中哪一个按钮被按下，配合事件回调使用</code>	105
刻度指示器	106
<code>lv_obj_t *刻度器对象 = lv_lmeter_create(...) //创建刻度控件，默认刻度范围0~100</code>	106
<code>lv_lmeter_set_range(...)//刻度范围</code>	106
<code>lv_lmeter_set_value(...)//当前刻度值</code>	106
<code>lv_lmeter_set_scale(...)//设置刻度角度和刻度线数量</code>	106
仪表盘控件	107
<code>lv_obj_t *仪表盘对象 = lv_gauge_create(...) //创建指针仪表盘控件</code>	107
<code>lv_gauge_set_range(...)//刻度范围</code>	107
<code>lv_gauge_set_critical_value(...)//关键数据点</code>	107
<code>lv_gauge_set_scale(...)//设置刻度线</code>	107
设置仪表盘样式，仪表盘样式和刻度线很像	108
<code>lv_gauge_set_needle_count(...)//创建多根指针</code>	108
<code>lv_gauge_set_value(...)//设置指针旋转到某个数值</code>	108
消息对话框使用	109
<code>lv_obj_t *消息对话框对象 = lv_mbox_create(..)//创建消息对话框，默认处于打开状态</code>	109
<code>lv_mbox_set_text(...)//消息对话框显示字符</code>	109
<code>lv_mbox_add_btns(...)//向消息对话框添加按钮控件</code>	109
<code>lv_mbox_set_anim_time(...)//消息对话框关闭时间</code>	110
消息对话框事件回调函数实现	110
<code>uint16_t 返回值 = lv_mbox_get_active_btn(消息对话框对象) //获取消息对话框哪一个按钮被按下</code>	110
<code>lv_mbox_start_auto_close(...)//延时关闭消息对话框</code>	111
消息对话框样式设置	111
<code>lv_obj_t * 消息内部按钮对象 = lv_mbox_get_btm(...)</code>	111
内部矩阵按钮对象	111
页面控件	113
<code>lv_obj_t *页面对象 = lv_page_create(...)//创建页面</code>	113
<code>int16_t 返回值 = lv_page_get_fit_width(页面对象) //获取页面的可填区域有多宽</code>	113
<code>lv_page_set_sb_mode(...)//设置页面模式</code>	113
向页面放入按钮	113
<code>lv_page_set_edge_flash(...)//页面滑动到顶部/底部出现下方阴影</code>	113
图表控件使用，设置曲线，柱状图	114
<code>lv_obj_t *图表对象 = lv_chart_create(...)//创建图表</code>	114
<code>lv_chart_set_div_line_count(...)//设置水平分割线和垂直分割线条数</code>	114
<code>lv_chart_series_t *数据线对象 = lv_chart_add_series(...)//添加数据线</code>	114
<code>lv_chart_init_points(...)//设置数据线每个点的值</code>	114
<code>lv_chart_set_range(...)//设置 y 轴高度</code>	115
<code>lv_chart_set_points(...)//设置多个数据点，图表默认只支持 10 个数据点</code>	115
<code>lv_chart_set_type(...)//设置图表类型，设置散点图</code>	115
<code>lv_chart_clear_serie(...)//清除图表里面的数据</code>	116
<code>lv_chart_refresh(图表对象)//刷新图表，为了保险起见，更新曲线时,使用刷新图表函数</code>	116
<code>lv_chart_set_point_count(...)//设置图表数据点个数(重要)</code>	116

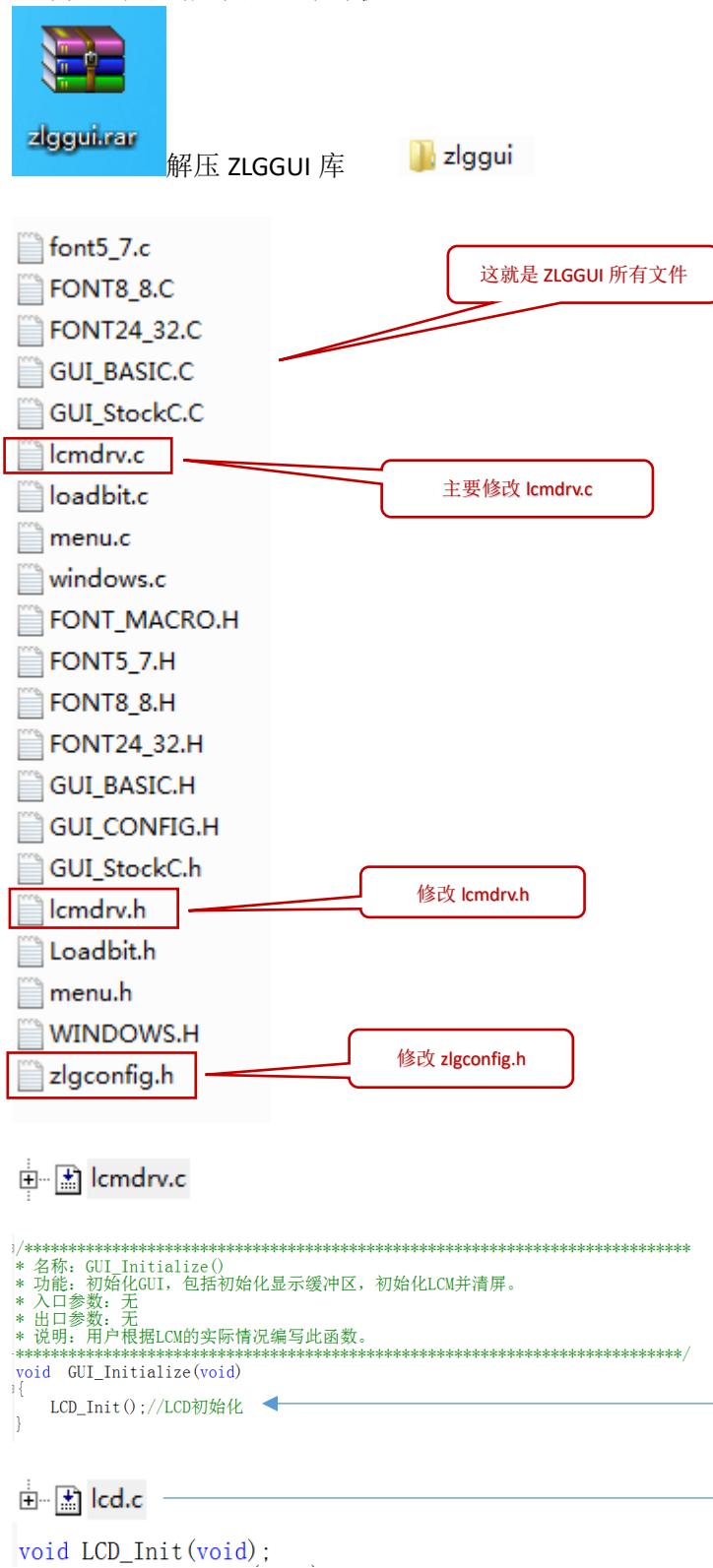
lv_chart_set_series_opa(...)	//设置面积图透明度	117
lv_chart_set_series_width(...)	//设置曲线宽度	117
lv_chart_set_series_darking(...)	//设置曲线黑阴影效果	117
数据线更新模式		117
lv_chart_set_update_mode(...)	//数据更新模式方式	117
lv_chart_set_next(...)	//向现有曲线最后一个点新增单个数据	117
刻度线设置		118
lv_chart_set_x_tick_length(...)	//设置刻度线 x 方向长度	119
lv_chart_set_x_tick_texts(...)	//刻度线加标题	119
lv_chart_set_margin(...)	//显示刻度线高度,一定要执行,不然刻度线不显示	119
表格控件		119
lv_obj_t *表格对象 = lv_table_create(...)	//创建表格	119
lv_table_set_row_cnt(...)	//行数设置	119
lv_table_set_col_cnt(...)	//列数设置	119
lv_table_set_cell_value(...)	//向某行某列写入数据(字符串写入)	120
lv_table_set_col_width(...)	//设置某列宽度	120
表格样式设置		120
lv_table_set_cell_align(...)	//单元格字符对齐方式	121
lv_table_set_cell_type(...)	//将某个单元格单独取编号,编译样式单独设置	121
预加载控件使用(类似 windows 鼠标旋转等待)		122
lv_obj_t *预加载对象 = lv_preload_create(...)	//创建预加载控件	122
预加载样式设置		122
lv_reload_set_spin_time(...)	//设置小圆弧旋转时间	122
lv_reload_set_dir(...)	//选择动画逆时针旋转	122
TAB 选项卡控件使用		123
lv_obj_t *选项卡对象 = lv_tabview_create(...)	//创建选项卡	123
lv_obj_t *Tab 页对象 = lv_tabview_add_tab(...)	//增加 tab 页	123
向 Tab 页放入控件和数据		123
lv_tabview_set_anim_time(...)	//延长切换时间	124
lv_tabview_set_btns_pos(...)	//设置选项卡按钮位置	124
文本域控件创建		124
lv_obj_t * 文本域对象 = lv_ta_create(...)	//创建文本域	124
lv_ta_add_char(...)	//向文本域添加字符	125
lv_ta_add_text(...)	//向文本域添加字符串	125
lv_ta_del_char(文本域对象)	//删除光标左侧字符	126
lv_ta_del_char_forward(文本域对象)	//删除光标右侧字符	126
lv_ta_set_text(...)	//直接设置文本内容,不管文本以前什么内容,一次性设置新字符串到文本	126
lv_ta_set_cursor_type(...)	//设置光标形状	126
lv_ta_set_pwd_mode(...)	//密码保护模式	127
lv_ta_set_pwd_show_time(...)	//密码保护模式下,输入字符显示指定秒然后变成	127
lv_ta_set_one_line(...)	//设置文本为单行模式	127
键盘控件		127
lv_obj_t *键盘对象 = lv_kb_create(...)	//创建键盘控件	127
lv_kb_set_ta(...)	//将键盘和文本域绑定起来(键盘按下的值自动传入文本)	127
lv_kb_set_mode(...)	//设置键盘模式	127
键盘事件回调函数实现		128
递增递减控件		129
lv_obj_t *增减对象 = lv_spinbox_create(...)	//创建递增递减控件	129
lv_spinbox_set_range(...)	//设置增减控件数值范围	129

<code>lv_spinbox_set_step(...)</code> // 每次递增，递减大小.....	129
<code>lv_spinbox_increment(增减对象)</code> //让增减控件递增一次.....	129
<code>lv_spinbox_decrement(增减对象)</code> //让增减控件递减一次.....	130
<code>lv_spinbox_set_padding_left(...)</code> //增加空格间隙.....	131
<code>lv_spinbox_set_value(...)</code> //增减控件创建默认值.....	131
<code>lv_spinbox_set_digit_format(...)</code> //设置数值长度，小数点位置.....	131
增减控件事件回调，获取增减控件值.....	131
<code>lv_spinbox_get_value(增减对象)</code>	131
图片控件使用	132
显示系统内置图标	132
<code>lv_obj_t *图片对象 = lv_img_create(...)</code> //创建图片控件.....	132
<code>lv_img_set_src(...)</code> //显示图标.....	132
加载图片方式	132
<code>LV_IMG_DECLARE(传入图片定义的结构体变量)</code> //导入其它文件的图片数组到本文件.....	133
图片按钮控件	135
<code>lv_obj_t *图片按钮对象 = lv_imgbtn_create(...)</code> //创建图片按钮.....	135
<code>lv_imgbtn_set_src(...)</code> //选择按钮状态与某个图片关联.....	135
窗体控件	136
<code>lv_obj_t *窗体对象 = lv_win_create(...)</code> //创建窗体控件.....	136
<code>lv_win_set_title(...)</code> //给窗体 title 加入名字.....	136
<code>lv_obj_t *窗体按钮对象 = lv_win_add_btn(...)</code> //窗体创建控制按钮.....	136
窗体事件回调函数使用	137
<code>lv_obj_t *窗体对象 = lv_win_get_from_btn(窗体按钮对象)</code> //获取窗体按钮对象里的窗体.....	137
列表控件	137
<code>lv_obj_t *列表对象 = lv_list_create(...)</code> //创建列表对象.....	138
<code>lv_obj_t *列表按钮对象 = lv_list_add_btn(...)</code> //添加列表按钮.....	138
给每个列表按钮设置事件回调	138
<code>lv_list_set_btn_selected(...)</code> //指定列表中某个按钮处于开机选中状态.....	139
<code>lv_list_set_single_mode(...)</code> //设置列表按钮字符滚动.....	139
点击外部按钮，滑动列表里面的按钮，在电阻屏或者按键应用中使用	139
<code>lv_list_up(列表对象)</code> //列表里面的按钮向上移动.....	139
<code>lv_list_down(列表对象)</code> //列表里面的按钮向下移动.....	140
<code>lv_list_focus(...)</code> //指定列表里面某个按钮直接滑动到显示区.....	140
下拉列表框控件	141
<code>lv_obj_t *下拉列表对象 = lv_ddlist_create(...)</code> //创建下拉列表.....	141
<code>lv_ddlist_set_fix_width(...)</code> //下拉列表专用宽度设置.....	141
<code>lv_ddlist_set_fix_height(...)</code> //下拉列表专用高度设置.....	141
<code>lv_ddlist_set_options(...)</code> //下拉列表加入一行选择内容.....	141
获取下拉框选中第几行字符	141
<code>lv_ddlist_set_selected(...)</code> //选择下拉第几行字符.....	141
<code>lv_ddlist_set_draw_arrow(...)</code> //下拉菜单右侧加入箭头.....	142
<code>lv_ddlist_set_align(...)</code> //下拉菜单字符对齐方式.....	142
根据下拉菜单的选择，执行内容	142
<code>uint16_t 返回值 = lv_ddlist_get_selected(下拉列表对象)</code> //获取选择下拉菜单的某行索引.....	142
下拉菜单样式设置	143
滚轮控件(下拉列表的另一种形式)	144
<code>lv_obj_t *滚轮对象 = lv_roller_create(...)</code> //创建滚轮下拉表.....	144
<code>lv_roller_set_fix_width(...)</code> //设置滚动条宽度.....	144

lv_roller_set_options(...) <i>//设置滚轮行数和模式</i>	144
lv_roller_set_visible_row_count(...) <i>//滚轮可见行数</i>	144
lv_roller_set_selected(...) <i>//滚轮开机默认选中第几行</i>	144
滚轮事件回调函数.....	144
uint16_t 返回值 = lv_roller_get_selected(滚轮对象) <i>//获取滚轮选中的索引值</i>	144
画布使用, 画布可以绘制原始图形, 正方形, 长方形, 或者任意图形.....	145
lv_canvas_set_buffer(...) <i>//设置画布缓冲区</i>	145
返回值=LV_CANVAS_BUF_SIZE_TRUE_COLOR(...) <i>//设置画布 TRUE_COLOR</i>	145
lv_obj_t *画布对象 = lv_canvas_create(...) <i>//创建画布</i>	145
lv_canvas_fill_bg(...) <i>//修改画布背景颜色</i>	145
lv_canvas_draw_rect(...) <i>//在画布中绘制矩形</i>	146
lv_canvas_draw_text(...) <i>//在画布中绘制文本</i>	146
theme 主题.....	147
lv_theme_t *主题对象 = lv_theme_night_init(...) <i>//创建 8 号主题</i>	147
lv_theme_set_current(主题对象) <i>//使用主题</i>	147
适合 OLED 小型界面库.....	149

ZLGGUI 移植

该 GUI 适用在 51 单片机这种小内存上，也适用在 STM32 这种大内存上。当然 STM32 内存也不大，只是相对于 51 单片机。



lcdrv.c

```
/****************************************************************************
 * 名称: GUI_FillSCR()
 * 功能: 全屏填充。直接使用数据填充显示缓冲区。
 * 入口参数: dat 填充的数据
 * 出口参数: 无
 * 说明: 用户根据LCM的实际情况编写此函数。
 *****/
void GUI_FillSCR(TCOLOR dat)
{
    LCD_Color_Fill(0, 0, 240, 320, &dat); //填充
}
```

STM32 要提供 LCD 图形颜色填充函数放在 ZLGUI 的 lcdrv.c
GUI_FillSCR 填充函数里面

lcd.c

```
void LCD_Color_Fill(u16 sx, u16 sy, u16 ex, u16 ey, u16 *color); //填充指定颜色
```

lcdrv.c

```
/****************************************************************************
 * 名称: GUI_ClearSCR()
 * 功能: 清屏。
 * 入口参数: 无
 * 出口参数: 无
 * 说明: 用户根据LCM的实际情况编写此函数。
 *****/
void GUI_ClearSCR()
{
    LCD_Clear(BLUE); //清屏
}
```

STM32 要提供 LCD 清屏函数放在 ZLGUI 的 lcdrv.c
GUI_ClearSCR 函数里面

lcd.c

```
void LCD_Clear(u16 Color); //清屏
```

lcdrv.c

```
/****************************************************************************
 * 名称: GUI_Point()
 * 功能: 在指定位置上画点。
 * 入口参数: x 指定点所在列的位置
 *           y 指定点所在行的位置
 *           color 显示颜色(对于黑白LCM, 为0时灭, 为1时显示)
 * 出口参数: 返回值为1时表示操作成功, 为0时表示操作失败。(操作失败原因是指定地址超出有效范围)
 * 说明: 用户根据LCM的实际情况编写此函数。
 *****/
uint8 GUI_Point(uint16 x, uint16 y, TCOLOR color)
{
    LCD_DrawPoint_GUI(x, y, color);
    return 1;
}
```

STM32 要提供 LCD 画点函数放在 ZLGUI 的 lcdrv.c GUI_Point
函数里面

lcd.c

```
void LCD_DrawPoint(u16 x, u16 y); //画点
```

lcdrv.c

```

/*
 * 名称: GUI_ReadPoint()
 * 功能: 读取指定点的颜色。
 * 入口参数: x 指定点所在列的位置
 *           y 指定点所在行的位置
 *           ret 保存颜色值的指针
 * 出口参数: 返回0表示指定地址超出缓冲区范围
 * 说明: 对于单色, 设置ret的d0位为1或0, 4级灰度则为d0、d1有效, 8位RGB则d0--d7有效,
 *       RGB结构则R、G、B变量有效。
 */
uint8 GUI_ReadPoint(uint16 x, uint16 y, TCOLOR *ret)
{
    TCOLOR temp;
    temp = LCD_ReadPoint(x, y);
    *ret = temp;
    return 0;

    return 1;
}

```

lcd.c

```

u16 LCD_ReadPoint(u16 x, u16 y); //读点

```

lcdrv.c

```

/*
 * 名称: GUI_HLine()
 * 功能: 画水平线。
 * 入口参数: x0 水平线起点所在列的位置
 *           y0 水平线起点所在行的位置
 *           x1 水平线终点所在列的位置
 *           color 显示颜色(对于黑白LCM, 为0时灭, 为1时显示)
 * 出口参数: 无
 * 说明: 对于单色、4级灰度的液晶, 可通过修改此函数作图提高速度, 如单色LCM, 可以一次更新8个点, 而不需要一个点一个点的写到LCM中。
 */
void GUI_HLine(uint16 x0, uint16 y0, uint16 x1, TCOLOR color)
{
    uint8 temp;
    if(x0>x1) // 对x0、x1大小进行排列, 以便画图
    {
        temp = x1;
        x1 = x0;
        x0 = temp;
    }
    do
    {
        GUI_Point(x0, y0, color); // 逐点显示, 描出垂直线
        x0++;
    }
    while(x1>=x0);
}

```

lcdrv.c

```

/*
 * 名称: GUI_RLine()
 * 功能: 画垂直线。
 * 入口参数: x0 垂直线起点所在列的位置
 *           y0 垂直线起点所在行的位置
 *           y1 垂直线终点所在行的位置
 *           color 显示颜色
 * 出口参数: 无
 * 说明: 对于单色、4级灰度的液晶, 可通过修改此函数作图提高速度, 如单色LCM, 可以一次更新8个点, 而不需要一个点一个点的写到LCM中。
 */
void GUI_RLine(uint16 x0, uint16 y0, uint16 y1, TCOLOR color)
{
    uint8 temp;
    if(y0>y1) // 对y0、y1大小进行排列, 以便画图
    {
        temp = y1;
        y1 = y0;
        y0 = temp;
    }
    do
    {
        GUI_Point(x0, y0, color); // 逐点显示, 描出垂直线
        y0++;
    }
    while(y1>=y0);
}

```

STM32 要提供 LCD 读点函数放在 ZLGUI 的 lcdrv.c
GUI_ReadPoint 函数里面

//读点

这些 GUI 代码就是将画点函数在 GUI 库里面进行水平线和垂直线的实现

下面移植配置文件

lcdrv.h 文件修改

```
*****  
#ifndef LCMDRV_H  
#define LCMDRV_H  
  
/* 定义颜色数据类型(可以是数据结构) */  
//#define TCOLOR uint16 //自行修改,原来是 uint8  
#define TCOLOR uint16_t //STM32 16位色LCD uint16_t  
  
/* 定义LCM像素数宏 */  
#define GUI_LCM_XMAX 240 //自行修改  
#define GUI_LCM_YMAX 320 //自行修改  
/* 定义液晶y轴的像素数 */  
/* 定义液晶x轴的像素数 */  
/* 修改 LCD 屏幕的 x  
长和 y 宽度 */
```

zlgconfig.h 修改 config 文件

```
#ifndef __ZLGCONFIG_H__  
#define __ZLGCONFIG_H__  
#include "stm32f10x.h"  
#ifndef TRUE  
#define TRUE 1  
#endif  
  
#ifndef FALSE  
#define FALSE 0  
#endif
```

增加 GUI 使用的 mcu
平台的头文件,因为
GUI 库里面用到了变量
类型的定义

这样就能全部编译通过了

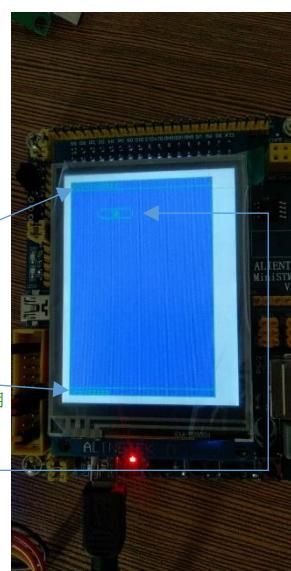
使用 ZLGGUI 范例

```
#include "zlgconfig.h"
```

主函数文件包含 ZLGGUI 配置头文件

```
int main(void)  
{  
  
    uint32_t i=0;  
    uint8_t str[]="xxxxxxxx";  
  
    WINDOWS demow;//定义一个窗口  
  
    RCC_configuration();//初始化时钟  
    USART_config(115200);//串口初始化  
    printf("xxxxxxxx\r\n");  
    //LCD_Init();  
    GUI_Initialize();//GUI初始化函数替代LCD初始化函数  
  
    delay_ms(1000);  
  
    demow.x = 10; //窗口开始x位置  
    demow.y = 10; //窗口开始y位置  
    demow.with = 200; //窗口宽度  
    demow.hight = 300; //窗口高度  
    demow.title = str; //窗口标题字符串显示  
    demow.state = str; //窗口状态字符串显示  
  
    GUI_SetColor(GREEN, BLUE); //设置窗口前景色和背景色 ZLGGUI手册有API函数说明  
    GUI_WindowsDraw(&demow); //显示窗口  
  
    GUI_SetColor(GREEN, BLUE); //设置按钮前景色和背景色  
    GUI_Button_OK(50, 50); //显示按钮
```

如果要给控件设置颜
色,一定在控件显示之
前设置前景色,背景色



主菜单，下拉菜单操作

MENU.H

```
/* 定义主菜单宽度, 及最大菜单个数 */
#define MMENU_WIDTH 34
#define MMENU_NO 6
```

```
/* 定义菜单的宽度(下拉菜单), 及最大子菜单个数 */
#define SMENU_WIDTH 66
#define SMENU_NO 8
```

/* 主菜单数据结构 */

```
typedef struct
{
    WINDOWS *win; // 所属窗口
    uint8 no; // 主菜单个数
    char *str[MMENU_NO]; // 主菜单字符串
} MMENU;
```

```
uint8_t str[] = "xxxxzzzz";
char str1[] = "menu"; //主菜单名字
```

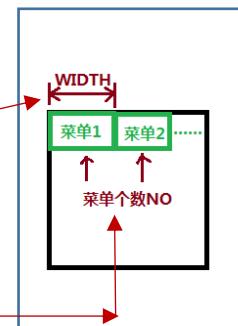
```
WINDOWS demow; //定义一个窗口
MMENU menu; //定义一个主菜单
RCC_configuration(); //初始化时钟
USART_config(115200); //串口初始化
printf("xxxxzzzz\r\n");
//LCD_Init();
GUI_Initialize(); //GUI初始化函数替代LCD初始化函数
delay_ms(1000);
```

```
demow.x = 10; //窗口开始x位置
demow.y = 10; //窗口开始y位置
demow.with = 200; //窗口宽度
demow.hight = 300; //窗口高度
demow.title = str; //窗口标题字符串显示
demow.state = str; //窗口状态字符串显示
```

```
GUI_SetColor(GREEN, BLUE); //设置窗口前景色和背景色
GUI_WindowsDraw(&demow); //显示窗口
```

```
menu.win = &demow; //主菜单依附于哪个窗口, 这里依附于demow窗口
menu.str[0] = str1; //根据menu.h主菜单有6个, 第0个菜单显示的菜单字符串
menu.no = 2; //设置几个主菜单, 我这里设置2个
GUI_MMenuDraw(&menu); //显示菜单
```

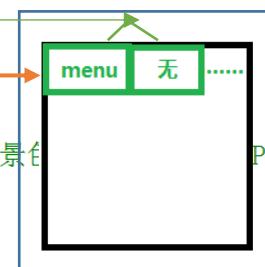
设置主菜单宽度



34
6

66
8

这里是 char *str[6] 结构体里面, 那么调用这个数组指针只能用下标去赋值, 结构体. 数组[0] 表示字符串存入第 0 个下标数组



PI 函数说明

```

GUI_SetColor(RED, YELLOW); //可以在菜单设置之前，设置菜单背景色，前景色
menu.win = &demow; //主菜单依附于哪个窗口，这里依附于demow窗口
menu.str[0] = str1; //根据menu.h主菜单有6个，第0个菜单显示的菜单字符
menu.no = 2; //设置几个主菜单，我这里设置2个
GUI_MMenuDraw(&menu); //显示菜单

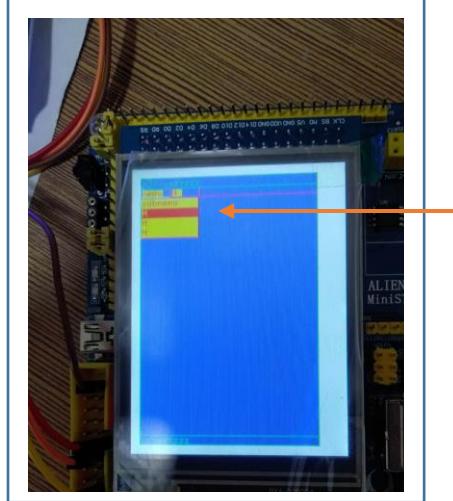
```

如何加入子菜单

```

uint32_t i=0;
uint8_t str[] = "XXXXZZZZ";
char str1[] = "menu"; //主菜单名字
char str2[] = "submenu"; //子菜单名
demow.x = 10; //窗口开始x位置
demow.y = 10; //窗口开始y位置
demow.with = 200; //窗口宽度
demow.hight = 300; //窗口高度
demow.title = str; //窗口标题字符显示
demow.state = str; //窗口状态字符显示

```



```

GUI_SetColor(GREEN, BLUE); //设置窗口前景色和背景色 ZLGGUI手册有API函数说明
GUI_WindowsDraw(&demow); //显示窗口

```

```

GUI_SetColor(RED, YELLOW); //可以在菜单设置之前，设置菜单背景色，前景色
menu.win = &demow; //主菜单依附于哪个窗口，这里依附于demow窗口
menu.str[0] = str1; //根据menu.h主菜单有6个，第0个菜单显示的菜单字符
menu.no = 2; //设置几个主菜单，我这里设置2个
GUI_MMenuDraw(&menu); //显示菜单

```

```

smeu.win = &demow; //子菜单依附的窗口
smeu.mmenu_no = 0; //子菜单在第几个主菜单下拉
smeu.no = 4; //下拉子菜单个数
smeu.str[0] = str2; //下拉子菜单第1个子菜单名字
smeu.state = 1; //选中第几个子菜单
GUI_SMenuDraw(&smeu); //显示子菜单

```

ZLGGUI 图标菜单使用

因为我是 16 位彩色屏幕，所以为了安全起见，我们要修改下底层驱动的颜色宏

[LCMDRV.H](#)

```

//*****
* 名称: GUI_CmpColor()
* 功能: 判断颜色值是否一致。
* 入口参数: color1 颜色值1
*          color2 颜色值2
* 出口参数: 返回1表示相同，返回0表示不相同。
* 说明: 由于颜色类型TCOLOR可以是结构类型，所以需要用户编写比较函数。
*****//#define GUI_CmpColor(color1, color2) ( (color1&0x01) == (color2&0x01) )

//*****
* 名称: GUI_CopyColor()
* 功能: 颜色值复制。
* 入口参数: color1 目标颜色变量
*          color2 源颜色变量
* 出口参数: 无
* 说明: 由于颜色类型TCOLOR可以是结构类型，所以需要用户编写复制函数。
*****//#define GUI_CopyColor(color1, color2) *color1 = color2

/*彩色屏幕定义*/
#define GUI_CmpColor(color1, color2) ((color1) == (color2))
#define GUI_CopyColor(color1, color2) ((*color1) = (color2))

#endif

```

屏蔽的是单色屏
定义的色彩宏

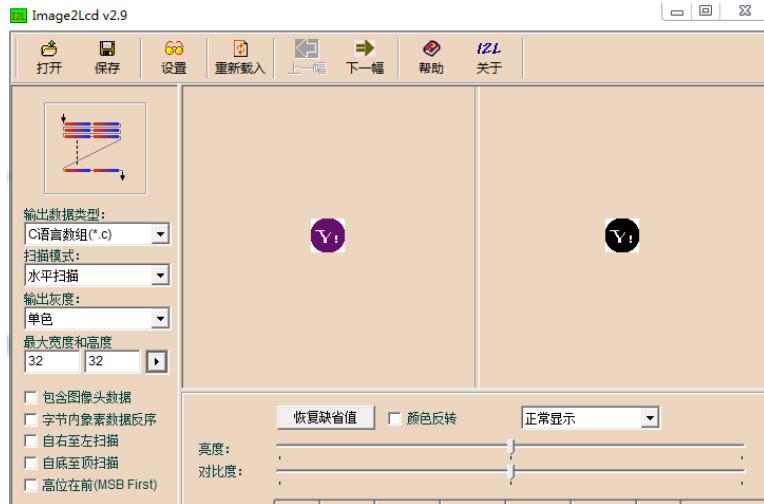
16 位彩色屏，定
义这两个宏

图标菜单需要两个图片，一个 ico 格式图片和一个 bmp 图片



找到 1 个 png 图片在网页找个工具转换成.ico 格式是 32x32

用图片转数组工具，将 ico 图片数组数据输出出来



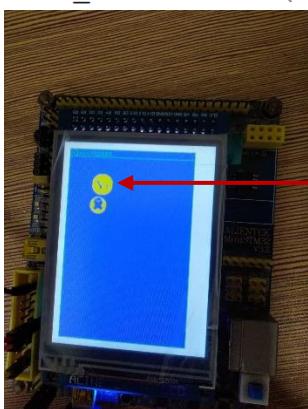
选择单色输出，大小设置成 32x32(因为 zlggui 代码要求的)

```
unsigned char gImage_xzzico[128] = { /* 0X00, 0X01, 0X20, 0X00, 0X20, 0X00, */  
    0X00, 0X3F, 0XFC, 0X00, 0X00, 0xFF, 0xFF, 0X00, 0X03, 0xFF, 0xFF, 0XCO, 0X07, 0xFF, 0xFF, 0XE0,  
    0X0F, 0xFF, 0xFF, 0XF0, 0X1F, 0xFF, 0xFF, 0XF8, 0X3F, 0xFF, 0XFC, 0X3F, 0xFF, 0XFF, 0XFC,  
    0X7F, 0xFF, 0XFF, 0XFE, 0X7F, 0xFF, 0XFF, 0XFE, 0XFF, 0XFF, 0XFF, 0XFF, 0XFO, 0X07, 0xFF, 0XFF,  
    0XFF, 0X3F, 0xFF, 0XFF, 0XFF, 0X9F, 0XDF, 0XFF, 0XFF, 0X9F, 0XBF, 0XFF, 0XFF, 0XCF, 0XBF, 0XFF,  
    0XFF, 0X7C, 0X7E, 0X7F, 0XFF, 0XE6, 0XFE, 0X7F, 0XFF, 0XF1, 0XFE, 0X7F, 0XFF, 0XF1, 0XFE, 0X7F,  
    0XFF, 0XF1, 0XFF, 0XFF, 0XFF, 0XF1, 0XFE, 0X7F, 0X7F, 0XF1, 0XFE, 0X7E, 0X7F, 0XFF, 0XFF, 0XFE,  
    0X3F, 0XFF, 0XFF, 0XFC, 0X3F, 0XFF, 0XFF, 0XFC, 0X1F, 0XFF, 0XFF, 0XF8, 0X0F, 0XFF, 0XFF, 0XFO,  
    0X07, 0XFF, 0XFF, 0XE0, 0X03, 0XFF, 0XFF, 0XCO, 0X00, 0XFF, 0X00, 0X00, 0X3F, 0XFC, 0X00,  
};
```

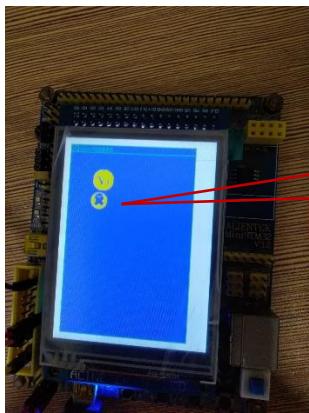
```
MENUICO xzzmenuico; // 定义图标菜单数据结构
```

```
GUI_SetColor(YELLOW, BLUE); // 设置图标前景色和背景色  
xzzmenuico.x = 50;  
xzzmenuico.y = 50;  
xzzmenuico.title = str; // 这个title必须要有随便取什么名字  
xzzmenuico.icodat = gImage_xzzico;  
xzzmenuico.state = 1;
```

将 ico 图标数组放进来



图标就显示出来了



找一个类似按钮的. ico 图片，格式要 32x32

menu.c 我们要修改 menu.c 文件

```

/*******************************
 * 名称: GUI_MenuIcoDraw()
 * 功能: 显示图标菜单。
 * 入口参数: ico 图标菜单句柄
 * 出口参数: 返回0表示操作失败, 返回1表示操作成功
****************************/
uint8 GUI_MenuIcoDraw(MENUICO *ico)
{
    /* 参数过滤 */
    if( ((ico->x)<5) || ((ico->x)>(GUI_LCM_XMAX-37)) ) return(0); // 显示起始地址判断
    if( ((ico->icodat)==NULL) || ((ico->title)==NULL) ) return(0); // 显示数据内容判断

    GUI_LoadPic(ico->x, ico->y, (uint8 *) ico->icodat, 32, 32); // 显示ICO图
    GUI_HLine(ico->x-5, ico->y+32, ico->x+37, back_color); // 显示一空行
    if((ico->state)==0)
    {
        //GUI_LoadPic(ico->x-5, ico->y+33, (uint8 *) ico->title, 42, 13);
        GUI_LoadPic(ico->x-5, ico->y+33, (uint8 *) ico->title, 32, 32);
    }
    else
    {
        //GUI_LoadPic1(ico->x-5, ico->y+33, (uint8 *) ico->title, 42, 13);
        GUI_LoadPic1(ico->x-5, ico->y+33, (uint8 *) ico->title, 32, 32);
    }
}

```

原版的 title 图标
显示要求用
42*13 的图标

但是网上找不到 42*13 图标，
需要美工单独做，所以我将底
层函数改成支持 32*32 的图标

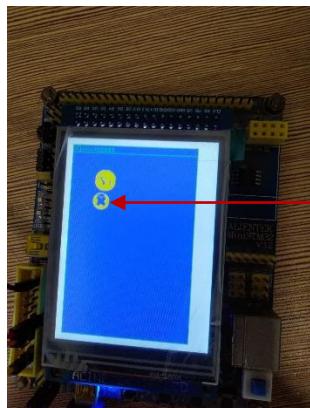
这是 icodat 要的图标数组，我前面已经用过了，我这里不管

```
unsigned char gImage_xzzico[128] = { /* 0X00, 0X01, 0X20, 0X00, 0X20, 0X00, */  
    0X00, 0X3F, 0XFC, 0X00, 0X00, 0XFF, 0XFF, 0X00, 0X03, 0XFF, 0XFF, 0XC0, 0X07, 0XFF, 0XFF, 0XE0,  
    0X0F, 0XFF, 0XFF, 0XF0, 0X1F, 0XFF, 0XFF, 0XF8, 0X3F, 0XFF, 0XFF, 0XFC, 0X3F, 0XFF, 0XFF, 0XFC,  
    0X7F, 0XFF, 0XFF, 0XFE, 0X7F, 0XFF, 0XFF, 0XFE, 0XFF, 0XFF, 0XFF, 0XFF, 0XFF, 0X0F, 0X07, 0XFF,  
    0XFF, 0X3F, 0XFF, 0XFF, 0XFF, 0X9F, 0XDF, 0XFF, 0XFF, 0X9F, 0XBF, 0XFF, 0XFF, 0XCF, 0XBF, 0XFF,  
    0XFF, 0X7C, 0X7E, 0X7F, 0XFF, 0XE6, 0XFE, 0X7F, 0XFF, 0XF1, 0XFE, 0X7F, 0XFF, 0XF1, 0XFE, 0X7F,  
    0XFF, 0XF1, 0XFF, 0XFF, 0XF1, 0XFE, 0X7F, 0XF1, 0XFE, 0X7E, 0X7F, 0XFF, 0XFF, 0XFE, 0X3F, 0XFF,  
    0XFC, 0X3F, 0XFF, 0XFC, 0X01, 0XFF, 0X0F, 0X01, 0XFF, 0X03, 0XFF, 0X01, 0XFF, 0X01, 0XFF, 0X0F,  
    0X07, 0XFF, 0XFF, 0X0E, 0X03, 0XFF, 0X01, 0XFF, 0X03, 0XFF, 0X01, 0XFF, 0X01, 0XFF, 0X0F, 0X00,  
};  
  
unsigned char gImage_xzzbmp[128] = { /* 0X00, 0X01, 0X20, 0X00, 0X20, 0X00, */  
    0X00,  
    0X00, 0X3C, 0X3C, 0X00, 0X00, 0XE0, 0X07, 0X00, 0X01, 0XC0, 0X03, 0X80, 0X03, 0X90, 0X0D, 0X0C,  
    0X07, 0XC4, 0X23, 0XE0, 0X07, 0XC2, 0X43, 0XE0, 0X0F, 0X81, 0X81, 0X0F, 0X0F, 0X00, 0X00, 0X0F,  
    0X1F, 0X80, 0X01, 0XFF, 0X1F, 0XCO, 0X01, 0XFF, 0X1F, 0XCO, 0X03, 0XFF, 0X1F, 0XE0, 0X07, 0XFF,  
    0X1F, 0XE0, 0X07, 0XFF, 0X1F, 0XCO, 0X03, 0XFF, 0X1F, 0XCO, 0X01, 0XFF, 0X1F, 0X80, 0X01, 0XFF,  
    0X0F, 0X00, 0X00, 0XFF, 0X81, 0X81, 0XFF, 0X07, 0XC3, 0XE3, 0XE0, 0X07, 0XEF, 0XFF, 0XE0,  
    0X03, 0XFF, 0XFF, 0XCO, 0X01, 0XFF, 0XFF, 0X80, 0X00, 0X7F, 0XFF, 0X00, 0X00, 0X3F, 0XFC, 0X00,  
    0X00, 0X07, 0XFF, 0X00,  
};
```

这就是我要的 title 图标数组

```
GUI_SetColor(YELLOW, BLUE); //设置图标前景色和背景色  
xzzmenuico.x = 50;  
xzzmenuico.y = 50;  
xzzmenuico.title = gImage_xzzbmp; //这个title我改成了图标数组  
xzzmenuico.icodat = gImage_xzzico;  
xzzmenuico.state = 0;  
GUI_MenuIcoDraw(&xzzmenuico);
```

将 title 需要的图标数组赋值给 title



这就是和 icodat 对应的可以控制的图标

但是我感觉这个图标怎么没有和上面的图标完全对上？这是你 GUI_MenuIcoDraw 底层函数的座标偏移没有修改。

```
/* 名称: GUI_MenuIcoDraw()  
* 功能: 显示图标菜单  
* 入口参数: ico 图标菜单句柄  
* 出口参数: 返回0表示操作失败, 返回1表示操作成功  
*****  
uint8 GUI_MenuIcoDraw(MENUICO *ico)  
{  
    /* 参数过滤 */  
    if( ((ico->x)<5) || ((ico->x)>(GUI_LCM_XMAX-37)) ) return(0); // 显示起始地址判断  
    if( ((ico->icodat)==NULL) || ((ico->title)==NULL) ) return(0); // 显示数据内容判断  
    GUI_LoadPic(ico->x, ico->y, (uint8 *) ico->icodat, 32, 32); // 显示ICO图  
    GUI_HLine(ico->x-5, ico->y+32, ico->x+37, back_color); // 显示一空行  
    if( (ico->state)==0 )  
    { //GUI_LoadPic(ico->x-5, ico->y+33, (uint8 *) ico->title, 42, 13);  
        GUI_LoadPic(ico->x-5, ico->y+33, (uint8 *) ico->title, 32, 32);  
    }  
    else  
    { //GUI_LoadPic(ico->x-5, ico->y+33, (uint8 *) ico->title, 42, 13);  
        GUI_LoadPic(ico->x-5, ico->y+33, (uint8 *) ico->title, 32, 32);  
    }  
    return(1);  
}
```

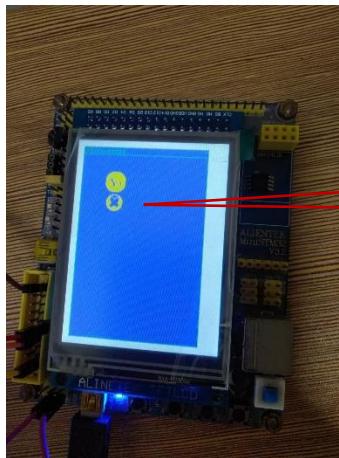
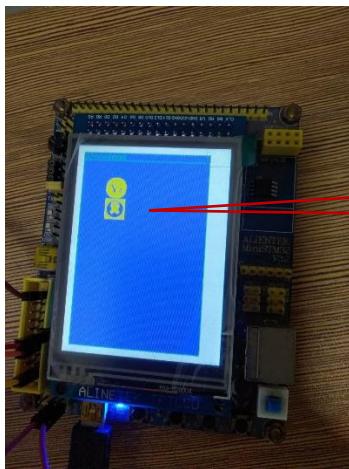
我把这个 x-5 改成 0 就可以了，其实自己也能猜到问题在哪里

测试 title 图标动态修改效果

```
GUI_SetColor(YELLOW, BLUE); //设置图标前景色和背景色  
xzzmenuico.x = 50;  
xzzmenuico.y = 50;  
xzzmenuico.title = gImage_xzzbmp; //这个title我改成了图标数组  
xzzmenuico.icodat = gImage_xzzico;  
xzzmenuico.state = 0;  
GUI_MenuIcoDraw(&xzzmenuico);
```

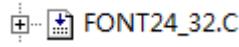
```
while(1)  
{  
    xzzmenuico.state = 0;  
    GUI_MenuIcoDraw(&xzzmenuico),  
    delay_ms(1000);  
    xzzmenuico.state = 1;  
    GUI_MenuIcoDraw(&xzzmenuico);  
    delay_ms(1000);  
}  
  
return 0;
```

设置图标菜单的 state 值就可以了，1 是正色显示，0 反色显示



所以这个图标还得靠美工按照需求来做

解决 ZLGGUI 字体大小和字体动态显示问题



FONT24_32.C 我们调用 FONT24_32.c 里面的函数来显示字体大点的数字

```
/* 名称: GUI_PutChar24_32()  
* 功能: 显示ASCII码(24*32字体), 显示值为'0'-'9'、'.'、'+'及'-'(若为其它值,  
*       入口参数: x 指定显示位置, y 坐标  
*               y 指定显示位置, z 坐标  
*               z 透明度, 范围0~255  
*       出口参数: 返回值为1时表示操作成功, 为0时表示操作失败。  
* 说明: 操作失败原因是指定地址超出有效范围。  
***** */  
uint8 GUI_PutChar24_32(uint32 x, uint32 y, uint8 ch)  
{  
    uint8 dat;  
    uint8 i, j;  
    TCOLOR bckc;  
  
    /* 参数过滤 */  
    if(x>=GUI_LCM_XMAX-32) return(0);  
    if(y>=GUI_LCM_YMAX-32) return(0);  
    for(i=0; i<14; i++)  
    {  
        if(FONT24x32_TAB[i]==ch) break;  
    }  
    ch = i;  
  
    for(i=0; i<32; i++)  
    {  
        for(j=0; j<24; j++)  
        {  
            /* 从 font.dat 中读取 0~8 16 点, 读取占位数据  
             * font.dat[0..0] = 0, font.dat[0..3] = FONT24x32_TAB[i*3+j]>3;  
             * 设置相应的点为color或back_color */  
            if(font_dat[0..0]>3) font_dat[0..3+j] = FONT24x32_TAB[i*3+j];  
            else GUI_CopyColor(&bckc, back_color);  
            if(font_dat[0..0]<=3) GUI_CopyColor(&bckc, disp_color);  
            GUI_Point(x, y, bckc);  
        }  
    }  
}
```

原版的数字显示代码有 BUG
问题, 显示乱码

下面是我重新修改的代码

```
uint8 xzzGUI_PutChar24_32(uint32 x, uint32 y, uint8 ch)
{
    uint8 font_dat;
    uint8 i, j, k;
    TCOLOR bakc;

    /* 参数过滤 */
    if( x>(GUI_LCM_XMAX-32) ) return(0);
    if( y>(GUI_LCM_YMAX-32) ) return(0);
    for(i=0; i<14; i++)
    {
        if(FONT24x32_TAB[i]==ch) break;
    }
    ch = i;

    for(i=0; i<32; i++) // 显示共 32 行
    {
        for(j=0; j<3; j++) // 每行共 3 字节
        {
            font_dat = FONT24x32[ch][i*3+j];
            /* 设置相应的点为 color 或为 back_color */
            for(k=0; k<8; k++)
            {
                if( (font_dat&DCB2HEX_TAB[k])==0 )
                    GUI_CopyColor(&bakc, back_color);
                else
                    GUI_CopyColor(&bakc, disp_color);
            }
            GUI_Point(x, y, bakc);
            x++;
        }
        y++;
    }

    // 指向下一行
    x -= 24;

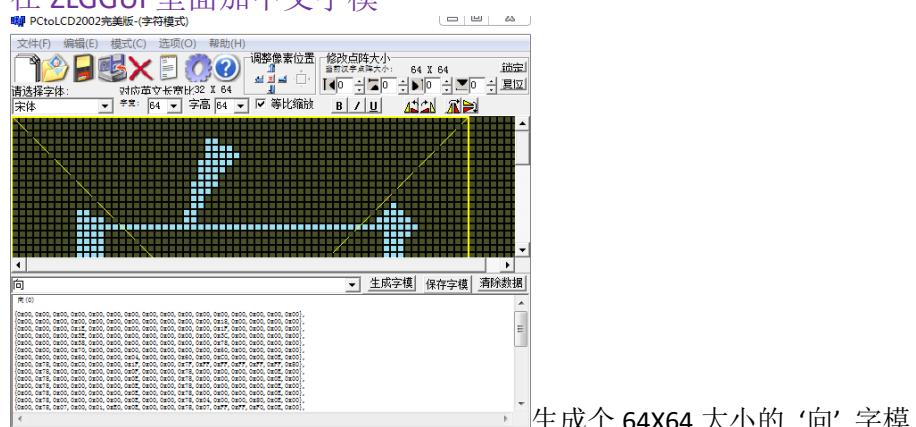
    // 恢复 x 值
}

return(1);
}
```

变大的数字显示出来了



在 ZLGGUI 里面加中文字模



生成个 64X64 大小的 '向' 字模



‘向’这个中文就显示出来了。窗口标题字体大小也可以用这个思路去修改。

STM32F103 移植 LittlevGL 图形库

LittlevGL 只支持 16 位, 32 位, 64 位处理器

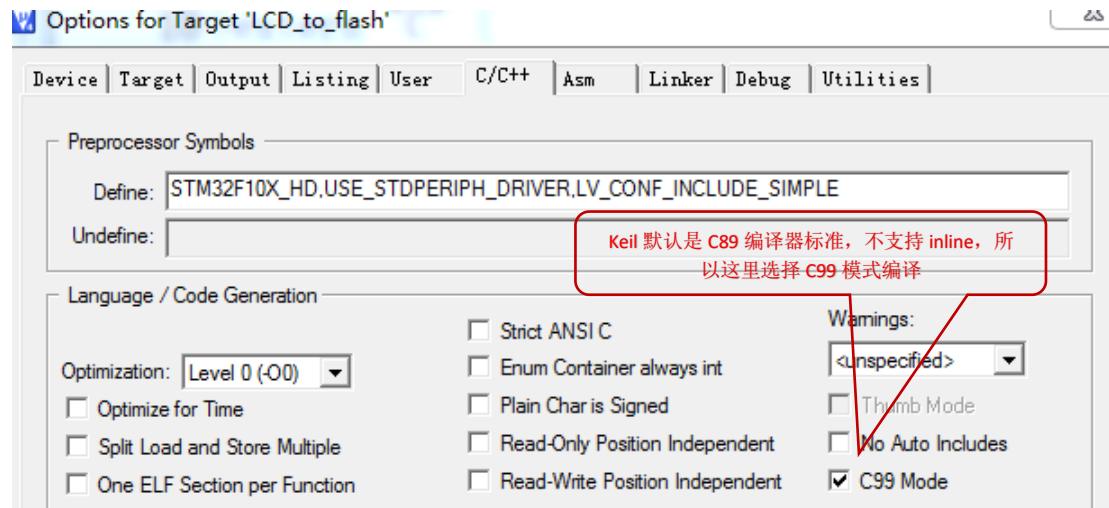
LittlevGL 要求 ROM/Flash 最小 64KB, 如果使用比较重要的组件 ROM/Flash 大于 180KB

LittlevGL 要求 RAM 8 到 16KB, 栈区:最小 2KB 最好大于 4KB,

堆区:大于 2KB,如果使用多个控件对象, 堆区要大于 4KB

显示缓冲区: 大于“水平分辨率”像素 (建议大于 10×“水平分辨率”)

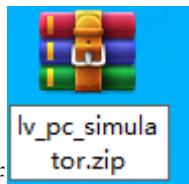
使用 C99 或更高版本的编译器



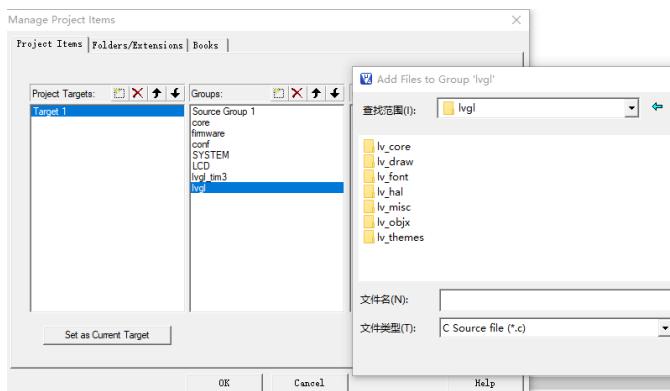
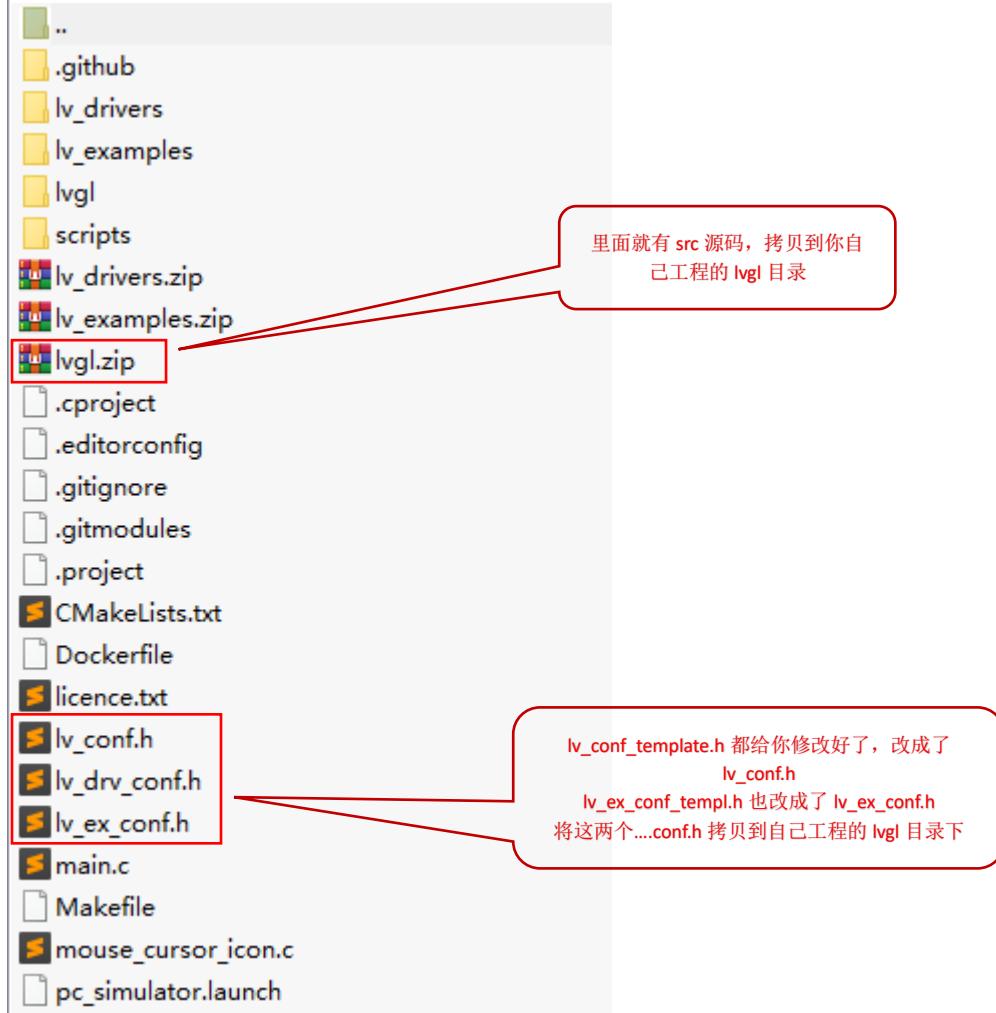
1. 移植 LittlevGL 内核文件

lvgl-master.zip

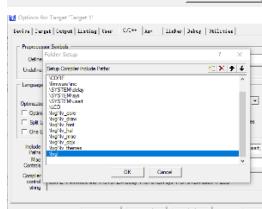




如果我们是在 `lv_pc_simulator` 压缩包解压的，那么拷贝方式就简单多了



将 `lvgl` 目录的 c 文件全部添加进工程



加入 `lvgl` 的 h 文件路径，开始编译。

```

lvgl\lv_themes\lv_theme_night.c: 0 warnings, 1 error
compiling lv_theme_template.c...
lvgl\lv_themes\lv_theme.h(19): error: #5: cannot open source input file "../../../../lv_conf.h": No such file or directory
#include "../../../../lv_conf.h"
lvgl\lv_themes\lv_theme_template.c: 0 warnings, 1 error
compiling lv_theme_zen.c...
lvgl\lv_themes\lv_theme.h(19): error: #5: cannot open source input file "../../../../lv_conf.h": No such file or directory
#include "../../../../lv_conf.h"
lvgl\lv_themes\lv_theme_zen.c: 0 warnings, 1 error
".\out\STM32F103C8T6.axf" - 79 Error(s), 0 Warning(s).
Target not created

```

将绝对路径取消掉，这么多 lv_conf.h 的绝对路径怎么取消呢？

```

#ifndef LV_CONF_INCLUDE_SIMPLE
#include "lv_conf.h"
#else
#include "../../../../../lv_conf.h"
#endif

```

我们打开这个宏

Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities

Preprocessor Symbols

Define: STM32F10X_HD,USE_STDPERIPH_DRIVER,LV_CONF_INCLUDE_SIMPLE

Keil 编译器加上
LV_CONF_INCLUDE_SIMPLE 宏

```

compiling lv_ta.c...
.\lvgl\lv_conf.h(491): error: #5: cannot open source input file "lvgl/src/lv_conf_checker.h": No such file or directory
#include "lvgl/src/lv_conf_checker.h"
lvgl\lv_objx\lv_ta.c: 0 warnings, 1 error
compiling lv_table.c...
.\lvgl\lv_conf.h(491): error: #5: cannot open source input file "lvgl/src/lv_conf_checker.h": No such file or directory
#include "lvgl/src/lv_conf_checker.h"
lvgl\lv_objx\lv_table.c: 0 warnings, 1 error
compiling lv_tabview.c...
.\lvgl\lv_conf.h(491): error: #5: cannot open source input file "lvgl/src/lv_conf_checker.h": No such file or directory

```

```

490 /*Be sure every define has a default value*/
491 #include "lvgl/src/lv_conf_checker.h"
492
493 #endif /*LV_CONF_H*/
494
495 #endif /*End of "Content enable"*/

```

```

490 /*Be sure every define has a default value*/
491 #include "lv_conf_checker.h"
492
493 #endif /*LV_CONF_H*/
494
495 #endif /*End of "Content enable"*/
10c

```

```

lvgl\lv_font\lv_font_roboto_12.c(1): error: #5: cannot open source input file "../../../../lvgl.h": No such file or directory
#include "../../../../lvgl.h"
lvgl\lv_font\lv_font_roboto_12.c: 0 warnings, 1 error
compiling lv_font_roboto_16.c...
lvgl\lv_font\lv_font_roboto_16.c(1): error: #5: cannot open source input file "../../../../lvgl.h": No such file or directory
#include "../../../../lvgl.h"
lvgl\lv_font\lv_font_roboto_16.c: 0 warnings, 1 error
compiling lv_font_roboto_22.c...
lvgl\lv_font\lv_font_roboto_22.c(1): error: #5: cannot open source input file "../../../../lvgl.h": No such file or directory
#include "../../../../lvgl.h"
1 #include "../../../../../lvgl.h"
2
3 /* **** */
4 * Size: 12 px

```

老方法把绝对路径取消掉

```

lvgl\lv_font\lv_font_roboto_22.c: 0 warnings, 1 error
compiling lv_font_roboto_28.c...
lvgl\lv_font\lv_font_roboto_28.c(1): error: #5: cannot open source input file "lvgl.h": No such file or directory
#include "lvgl.h"
lvgl\lv_font\lv_font_roboto_28.c: 0 warnings, 1 error
compiling lv_font_unscii_8.c...
lvgl\lv_font\lv_font_unscii_8.c(1): error: #5: cannot open source input file "lvgl.h": No such file or directory
#include "lvgl.h"
lvgl\lv_font\lv_font_unscii_8.c: 0 warnings, 1 error

```

	文件夹	2019/7/8 19:10			
src					
.clang-format	2,826	1,141	CLANG-FORMAT...	2019/7/8 19:10	C185BD
.editorconfig	125	96	EDITORCONFIG ...	2019/7/8 19:10	1810B7.
.gitignore	43	35	GITIGNORE 文件	2019/7/8 19:10	41C372
.gitmodules	0	0	GITMODULES 文件	2019/7/8 19:10	0000000
LICENCE.txt	1,083	642	TXT 文件	2019/7/8 19:10	F19EE89
lv_conf_template.h	15,488	5,469	H 文件	2019/7/8 19:10	0F01D0
lvgl.h	2,070	465	H 文件	2019/7/8 19:10	19C174
lvgl.mk	339	94	MK 文件	2019/7/8 19:10	DFD01E
README.md	11,174	4,303	MD 文件	2019/7/8 19:10	41108C

将 lv_pc_simulator 压缩包里面的 lvgl.zip 压缩包内部的 lvgl.h 拷贝到自己工程的 lvgl 目录下

```

.\lvgl\lvgl.h(17): error: #5: cannot open source input file "src/lv_version.h": No such file or directory
#include "src/lv_version.h"
lvgl\lv_font\lv_font_roboto_28.c: 0 warnings, 1 error
compiling lv_font_unscii_8.c...
.\lvgl\lvgl.h(17): error: #5: cannot open source input file "src/lv_version.h": No such file or directory
#include "src/lv_version.h"
lvgl\lv_font\lv_font_unscii_8.c: 0 warnings, 1 error

```

```

12 /* **** INCLUDES **** */
13 *      INCLUDES
14 *      **** */
15
16
17 #include "src/lv_version.h"
18
19 #include "src/lv_misc/lv_log.h"
20 #include "src/lv_misc/lv_task.h"
21 #include "src/lv_misc/lv_math.h"
22 #include "src/lv_misc/lv_async.h"
23
24 #include "src/lv_hal/lv_hal.h"
25
26 #include "src/lv_core/lv_obj.h"
27 #include "src/lv_core/lv_group.h"
28
29 #include "src/lv_core/lv_refr.h"
30 #include "src/lv_core/lv_disp.h"
31
32 #include "src/lv_themes/lv_theme.h"
33
34 #include "src/lv_font/lv_font.h"
35 #include "src/lv_font/lv_font_fmt_txt.h"
36
37 #include "src/lv_objx/lv_btn.h"
38 #include "src/lv_objx/lv_imgbtn.h"
39 #include "src/lv_objx/lv_img.h"
40 #include "src/lv_objx/lv_label.h"
41 #include "src/lv_objx/lv_line.h"
42 #include "src/lv_objx/lv_page.h"
43 #include "src/lv_objx/lv_cont.h"

```

```

#include "lv_version.h"
#include "lv_misc/lv_log.h"
#include "lv_misc/lv_task.h"
#include "lv_misc/lv_math.h"
#include "lv_misc/lv_async.h"
#include "lv_hal/lv_hal.h"
#include "lv_core/lv_obj.h"
#include "lv_core/lv_group.h"
#include "lv_core/lv_refr.h"
#include "lv_core/lv_disp.h"

```

编译还有一个错误 lv_font_roboto_16.c(2247): error: #69: integer conversion resulted in truncation，这个问题下面会解决。

修改栈区大小，littevg1 要求最小栈区大小为 2K。

```

Stack_Size EQU 0x00000800
Stack_Mem AREA STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem SPACE Stack_Size

```

0x800 就是 2048 0x400 就是 1024

修改 lv_conf.h 配置文件

```
lv_conf.h
#ifndef LV_CONF_H
#define LV_CONF_H
/* clean-format off */
```

lv_conf.h 第一个
if 改成 1

修改 lv_font.c , 如果用的 C99 标准编译器, 那么不需要将 inline 改成双下划线

```
lv_font.c
static __inline uint8_t lv_font_get_line_height(const lv_font_t * font_p)
{
    return font_p->line_height;
}
```

```
lv_fontRoboto_16.c
#include "lvgl/src/lv_conf_checker.h"
```

取消掉 lvgl/src, 用不着绝对路径

```
lvgl.h
#include "src/lv_version.h"
```

```
#include "src/lv_misc/lv_log.h"
#include "src/lv_misc/lv_task.h"
#include "src/lv_misc/lv_math.h"
```

取消掉 src, 用不着绝对路径

```
#include "lv_version.h"
#include "lv_misc/lv_log.h"
#include "lv_misc/lv_task.h"
#include "lv_misc/lv_math.h"
#include "lv_misc/lv_async.h"

#include "lv_hal/lv_hal.h"

#include "lv_core/lv_obj.h"
#include "lv_core/lv_group.h"
```

```
lvgl\lv_font\lv_fontRoboto_16.c(2247): error: #69: integer conversion resulted in truncation
```

```
lv_font_fmt.txt
```

```

} lv_font_fmt_txt_dsc_t;      修改 lv_font_fmt_txt_dsc_t 结构体
*/
const void * kern_dsc;
// /*Scale kern values in 12.4 format*/
//     uint16_t kern_scale;          将结构体中间的 kern_scale 取消

    uint32_t last_letter;
    uint32_t last_glyph_id;
    uint16_t kern_scale;

} lv_font_fmt_txt_dsc_t;      将 kern_scale 放在 lv_font_fmt_txt_dsc_t 结构体最后
这样就可以编译通过了，不知道是不是 keil 的 BUG

```

lv_font_roboto_16.c(2247): error: #69: integer conversion resulted in truncation 报错问题

其实和上面 lv_font_fmt_txt_dsc_t 问题一样，也是结构体定义顺序问题，有可能和 keil4 编译器有关，keil5 就没这个问题。

进入 [lv_font_fmt_txt.h](#)

```

const void * kern_dsc;

/*Scale kern values in 12.4 format/
uint16_t kern_scale;          将 uint16_t kern_scale 放在
/*Number of cmap tables*/
uint16_t cmap_num :10;        结构体最后就可以了

/*Bit per pixel: 1, 2, 4 or 8*/
uint16_t bpp :3;

/*Type of `kern_dsc`*/
uint16_t kern_classes :1;

/*
 * storage format of the bitmap
 * from `lv_font_fmt_txt_bitmap_format_t`
 */
uint16_t bitmap_format :2;

/*Cache the last letter and is glyph id*/
uint32_t last_letter;
uint32_t last_glyph_id;

} lv_font_fmt_txt_dsc_t;

    uint16_t bitmap_format :2;
    /*Cache the last letter and is glyph id*/
    uint32_t last_letter;
    uint32_t last_glyph_id;
    /*Scale kern values in 12.4 format/
    uint16_t kern_scale;          放到最后之后，问题得到解决
} lv_font_fmt_txt_dsc_t;

```

编译通过，没有报错

```
56 #define LV_DPI           100 /*[px]*/
LV_DPI 100 我们改成 60，DPI 是配置缩放比例的，如果 DPI 值越大，那么界面看起来就像放大了的一样。
```

```
71 /* Size of the memory used by lv_mem_alloc
72 #define LV_MEM_SIZE      (128U * 1024U)
```

Littlevgl 内存管理，这里默认内存是 128K，但是我们用不到这么大，我们 STM32 也不支持这么大。我们改成 16U*1024U，也就是 16K。

```
144 /* 1: Enable GPU interface*/
145 #define LV_USE_GPU        1
```

我们 STM32 没有 GPU，所以我们设置 LV_USE_GPU 为 0。
如果你的 CPU 有 GPU，那么 LV_USE_GPU 就设置为 1，然后去回调函数去实现 GPU 代码。

```
148 #define LV_USE_FILESYSTEM 1
149 #if LV_USE_FILESYSTEM
```

文件系统我们不需要，设置为 0 就是

```
232 /*=====
233  * THEME USAGE
234  =====*/
235 #define LV_THEME_LIVE_UPDATE    1 /*1: Allow theme switch*/
236
237 #define LV_USE_THEME_TEMPL      1 /*Just for test*/
238 #define LV_USE_THEME_DEFAULT    1 /*Built mainly from template*/
239 #define LV_USE_THEME_ALIEN     1 /*Dark futuristic theme*/
240 #define LV_USE_THEME_NIGHT      1 /*Dark elegant theme*/
241 #define LV_USE_THEME_MONO       1 /*Mono color theme for monospace*/
242 #define LV_USE_THEME_MATERIAL   1 /*Flat theme with bold font*/
243 #define LV_USE_THEME_ZEN        1 /*Peaceful, mainly light colors*/
244 #define LV_USE_THEME_NEMO       1 /*Water-like theme based on Nemo*/

```

如果要跑官方例程，这些主题全部置 1，如果自己写项目，这些主题都不需要的，全部置 0 就可以了，节省 flash 空间。

```
256 /* Robot fonts with bpp = 4
257  * https://fonts.google.com/specimen/Roboto */
258 #define LV_FONT_ROBOTO_12      0
259 #define LV_FONT_ROBOTO_16      1
260 #define LV_FONT_ROBOTO_22      0
261 #define LV_FONT_ROBOTO_28      0
262
263 /*Pixel perfect monospace font
264  * http://pelulamu.net/unscii/ */
265 #define LV_FONT_UNSCII_8       0
```

字体我们暂时使用 16 号字体，其余字体都为 0。

下面移植硬件驱动

添加一个定时器，为 littleVGL 提供心跳节拍。

```
lvgl_tim3.c
1 #include "lvgl_tim3.h"
2 #include "lvgl.h"
3
4 //通用定时器中断初始化
5 //这里时钟选择为APB1的2倍，而APB1为36M
6 //arr: 自动重装值。
7 //psc: 时钟预分频数
8 //这里使用的是定时器3!
9 void Lvgl_TIM3_Int_Init(u16 arr, u16 psc)
10 {
11     TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
12     NVIC_InitTypeDef NVIC_InitStructure;
13
14     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //时钟使能

void TIM3_IRQHandler(void) //TIM3中断
{
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) //检查指定的TIM中断发生与否:TIM 中断源
    {
        lv_tick_inc(1); //lvgl 1ms心跳
    }
    TIM_ClearITPendingBit(TIM3, TIM_IT_Update); //清除TIMx的中断待处理位:TIM 中断源
}

int main(void)
{
    RCC_Configuration(); //初始化时钟
    delay_init(); //延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置NVIC中断优先级
    uart_init(115200); //串口1初始化
    Lvgl_TIM3_Int_Init(999, 71); //Lvgl TIM3 初始化 1ms中断
}
```

把 lvgl.h 包含进定时器驱动程序

在定时器中断函数中执行 lvgl 的心跳

在 main 函数初始化的时候一定要打开定时器

现在进行总体编译。发现 306 个警告。

```
Program Size: Code=60632 RO-data=10220 RW-data=4036 ZI-data=28500
FromELF: creating hex file...
".\out\STM32F1LCD.axf" - 0 Error(s), 306 Warning(s).
```

其实这 300 多个警告就 3 类

```
1141     }
1142     }
1143 }
1144
Build Output
lvgl\lv_obj\lv_chart.c(l141): warning: #68-D: integer conversion resulted in a change of sign
lv_draw_label(a, mask, style, opa_scale, buf, LV_TXT_FLAG_CENTER, NULL, -1, -1, NULL);
lvgl\lv_obj\lv_chart.c(l141): warning: #68-D: integer conversion resulted in a change of sign
lv_draw_label(a, mask, style, opa_scale, buf, LV_TXT_FLAG_CENTER, NULL, -1, -1, NULL);
```

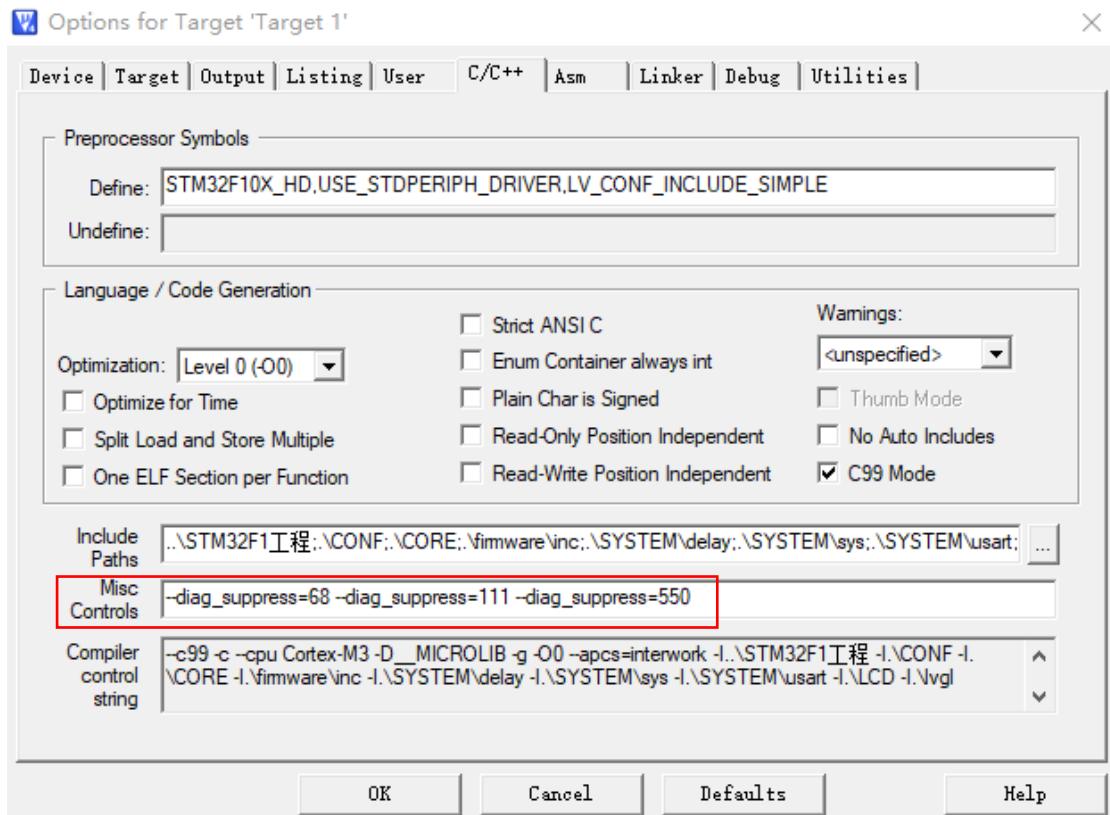
lv_draw_label 函数，其实就是这两个-1 的问题，但是这两个-1 问题不大。

```
412     /*To avoid warning*/
413     return NULL;
414 }
415
Build Output
compiling lv_cont.c...
compiling lv_ddlist.c...
lvgl\lv_obj\lv_ddlist.c(413): warning: #111-D: statement is unreachable
```

111-d 警告其实就是 return NULL 的问题。但是问题不大。

现在我只有在 keil 配置里面将这两个警告处理掉。

--diag_suppress=68 警告处理符号



这就是将 68,111,550 号警告处理掉，到底要不要这样做，其实还是得谨慎考虑。

```
*****  
Program Size: Code=60632 RO-data=10220 RW-data=4036 ZI-data=28500  
FromELF: creating hex file...  
.\\out\\STM32F1LCD.axf" - 0 Error(s), 0 Warning(s).
```

编译成功，无错误，无警告。

移植显示驱动程序

scripts	..	
lv_drivers.zip	lv_port_disp_template.c	6,714
lv_examples.zip	lv_port_disp_template.h	701
lvgl.zip	lv_port_fs_template.c	9,579
.cproject	lv_port_fs_template.h	691
.editorconfig	lv_port_indev_template.c	10,676
.gitignore	lv_port_indev_template.h	707

在 lvgl.zip 下 lvgl 目录下 porting 目录下的 lv_port_disp_template.c/.h 文件解压到自己工程的 lvgl_drv 目录下。

lv_port_disp_template.c 是显示驱动

lv_port_fs_template.c 是文件系统驱动

lv_port_indev_template.h 是输入触摸屏驱动

我现在主要移植显示驱动。

lv_port_disp_template.c
lv_port_disp_template.h

将这两个文件名字改成我想要的文件名。

```
#if 1

#ifndef LV_PORT_DISP_H
#define LV_PORT_DISP_H
```

h 文件 if 置 1

```
19 | #include "lvgl/lvgl.h"
20 |
21 | /*****
```

Build Output

```
Build target 'Target 1'
compiling lv_port_disp.c...
lvgl drv\lv_port_disp.h(19): error: #5: cannot open source input file "lvgl/lvgl.h": No such file or directory
```

```
void lv_port_disp_init(void)
{
    /*
     * Initialize your display
     */
    disp_init();

    /* Example for 1 */
    static lv_disp_buf_t disp_buf_1;
    static lv_color_t buf1_1[LV_HOR_RES_MAX * 10];           /*A buffer for 10 rows*/
    lv_disp_buf_init(&disp_buf_1, buf1_1, NULL, LV_HOR_RES_MAX * 10); /*Initialize the display buffer*/

    /* Example for 2 */
    static lv_disp_buf_t disp_buf_2;
    static lv_color_t buf2_1[LV_HOR_RES_MAX * 10];           /*A buffer for 10 rows*/
    static lv_color_t buf2_2[LV_HOR_RES_MAX * 10];           /*An other buffer for 10 row*/
    lv_disp_buf_init(&disp_buf_2, buf2_1, buf2_2, LV_HOR_RES_MAX * 10); /*Initialize the display buffer*/

    /* Example for 3 */
    static lv_disp_buf_t disp_buf_3;
    static lv_color_t buf3_1[LV_HOR_RES_MAX * LV_VER_RES_MAX]; /*A screen sized buffer*/
    static lv_color_t buf3_2[LV_HOR_RES_MAX * LV_VER_RES_MAX]; /*An other screen sized buffer*/
    lv_disp_buf_init(&disp_buf_3, buf3_1, buf3_2, LV_HOR_RES_MAX * LV_VER_RES_MAX); /*Initialize the display buffer*/
```

第 1 种，单缓存

第 2 种，双缓存，硬件内部缓存够大，就可以用这个

第 3 种，最流畅的方法

三种方式设置 littlvgi 缓冲区。我就用第 1 种单缓存。可能显示滑动有点卡，将就用。

```
void lv_port_disp_init(void)
{
    /*
     * Initialize your display
     */
    disp_init();

    /* Example for 1 */
    static lv_disp_buf_t disp_buf_1;
    static lv_color_t buf1_1[LV_HOR_RES_MAX * 10];           /*A buffer for 10 rows*/
    lv_disp_buf_init(&disp_buf_1, buf1_1, NULL, LV_HOR_RES_MAX * 10); /*Initialize the display buffer*/

    /* Example for 2 */
    static lv_disp_buf_t disp_buf_2;
    static lv_color_t buf2_1[LV_HOR_RES_MAX * 10];           /*A buffer for 10 rows*/
    static lv_color_t buf2_2[LV_HOR_RES_MAX * 10];           /*An other buffer for 10 row*/
    lv_disp_buf_init(&disp_buf_2, buf2_1, buf2_2, LV_HOR_RES_MAX * 10); /*Initialize the display buffer*/

    /* Example for 3 */
    static lv_disp_buf_t disp_buf_3;
    static lv_color_t buf3_1[LV_HOR_RES_MAX * LV_VER_RES_MAX]; /*A screen sized buffer*/
    static lv_color_t buf3_2[LV_HOR_RES_MAX * LV_VER_RES_MAX]; /*An other screen sized buffer*/
    lv_disp_buf_init(&disp_buf_3, buf3_1, buf3_2, LV_HOR_RES_MAX * LV_VER_RES_MAX); /*Initialize the display buffer*/
```

使用单缓存

```

lv_disp_drv_t disp_drv;           /*Descriptor of a display driver*/
lv_disp_drv_init(&disp_drv);      /*Basic initialization*/

/*Set up the functions to access to your display*/

/*Set the resolution of the display*/
disp_drv.hor_res = lcddev.width;
disp_drv.ver_res = lcddev.height;

```

```

/*Used to copy the buffer's content to the display*/
disp_drv.flush_cb = disp_flush;

```

```

/*Set a display buffer*/
disp_drv.buffer = &disp_buf_1;

```

这里改成显示缓冲区 1，单缓存 buf

这是底层绘图函数，我们将里面的源代码删除掉，用上我们的 LCD 代码

```

static void disp_flush(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t * color_p)
{
    /*The most simple case (but also the slowest) to put all pixels to the screen one-by-one*/

    int32_t x;
    int32_t y;
    for(y = area->y1; y <= area->y2; y++) {
        for(x = area->x1; x <= area->x2; x++) {
            /* Put a pixel to the display. For example: */
            /* put_px(x, y, *color_p)*/
            color_p++;
        }
    }

    /* IMPORTANT!!!
     * Inform the graphics library that you are ready with the flushing*/
    lv_disp_flush_ready(disp);
}

```

修改后如下

```

static void disp_flush(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t * color_p)
{
    LCD_Color_Fill(area->x1, area->y1, area->x2, area->y2, (u16*)color_p); //这是我自己的LCD填充矩形代码
    lv_disp_flush_ready(disp_drv); //这部一定要执行，屏幕才会有变化
}

```

以上操作移植 lvgl 完成

下面我们实验试试看

```

#include "stm32f10x.h"
#include "usart.h"
#include "delay.h"
#include "stm32f10x_rcc.h"
#include "lcd.h"

#include "lvgl_tim3.h"
#include "lvgl.h"
#include "lv_port_disp.h"

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    delay_init(); // 延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 设置NVIC中断分组2:2位抢占优先级,
    uart_init(115200); // 串口1初始化
    Lvgl_TIM3_Init(999, 71); // Lvgl TIM3 初始化 1ms中断, 一定要提前执行
    LED_INIT(); // LED 测试板子是否启动, 可以不要

    // LCD_Init(); // 交由 lv_port_disp_init 里面的 disp_init() 函数去初始化

    lv_init(); // lv 初始化
    lv_port_disp_init(); // lv 接口初始化函数一定要放在 lv_init() 函数之后

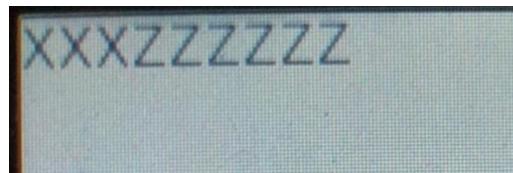
    lv_obj_t *scr = lv_disp_get_scr_act(NULL); // 第1步必须创建显示句柄
    lv_obj_t *label = lv_label_create(scr, NULL); // 创建 label 标签
    lv_label_set_text(label, "XXXXXXXXXX"); // label 标签显示你需要的字符

    while(1)
    {
        lv_task_handler(); // 这句执行后有界面效果
    }
}

```

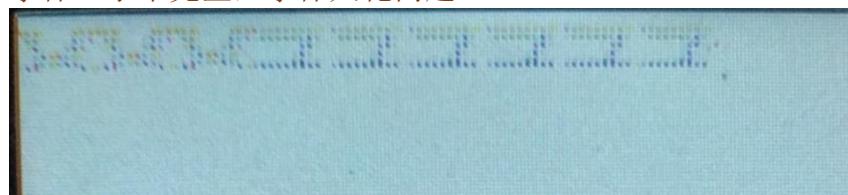
记住，这句代码一定要不停循环执行，这是 lvgl 事件循环，如果不执行就无法显示图形，感觉像移植失败一样。

理论上应该在左上角显示完整的 XXXXXXXX 字符，但是我发现字符是显示了但是字体比较花。



正常应该是这样。

字体显示不完整，字体太花问题



你看字体显示好花？但是感觉是你自己设置的字符，这是为什么呢？

检查后我发现原来是 lv_conf.h 里面颜色深度设置不符合 LCD 显示要求

```

/* Color depth:
 * - 1: 1 byte per pixel
 * - 8: RGB233
 * - 16: RGB565
 * - 32: ARGB8888
 */
#define LV_COLOR_DEPTH 32

```

你看，我 LCD 是 16 位颜色位宽(深度)的，但是我设置的是 32 位颜色位宽(深度)

```
#define LV_COLOR_DEPTH 16 // 将颜色深度改成 16
```



运行后，显示正常了。这就是 lvgl 完整移植过程，后面进行功能测试了。

代码示例

```
int main(void)
{
    RCC_Configuration(); // 初始化时钟
    delay_init(); // 延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(115200); // 串口 1 初始化
    Lvgl_TIM3_Int_Init(999, 71); // Lvgl TIM3 初始化 1ms 中断，一定要提前执行
    LED_INIT(); // LED 测试板子是否启动, 可以不要

    // LCD_Init(); // 交由 lv_port_disp_init 里面的 disp_init() 函数去初始化

    lv_init(); // lv 初始化
    lv_port_disp_init(); // lv 接口初始化函数一定要放在 lv_init() 函数之后

    lv_obj_t *scr = lv_disp_get_scr_act(NULL); // 第 1 步必须创建显示句柄

    lv_obj_t *label = lv_label_create(scr, NULL); // 创建 label 标签
    lv_label_set_text(label, "XXXZZZZZ"); // label 标签显示你需要的字符

    while(1)
    {
        lv_task_handler(); // 这句执行后有界面效果
    }
}
```

lv_conf.h 配置文件详解

```
22 /* Maximal horizontal and vertical resolution to
23 #define LV_HOR_RES_MAX (320) // 320
24 #define LV_VER_RES_MAX (480) // 480
```

这是配置 HOR 水平分辨率和垂直 VER 分辨率的配置项

这两个配置项还可以结合 LCD 驱动里面的 LCD_Display_Dir(dir) 函数实现横屏和竖屏。
LCD_Display_Dir(dir) 函数是你 LCD 的驱动函数, 我用的原子的函数库。

如果 LCD_Display_Dir(1) // 设置为横屏, 下面 Lvgl 就需要修改。

```
#define LV_HOR_RES_MAX (800) // 如果设置成横屏, 这里是 7 寸屏幕, 水平分辨率改为 800
#define LV_VER_RES_MAX (480) // 垂直分辨率设置为 480
```

```
/* Swap the 2 bytes of RGB565 color.
 * Useful if the display has a 8 bit interface (e.g. SPI) */
#define LV_COLOR_16_SWAP 0
```

这里保持 0 不要动

```
/* 1: Enable screen transparency.
 * Useful for OSD or other overlapping
 * Requires `LV_COLOR_DEPTH = 32` color
#define LV_COLOR_SCREEN_TRANSP 0
```

这里设置为 1, 层叠 GUI 可以做到窗口透明, 而且必须是 32 位 ARGB 才行。我这里是 16 位 LCD, 所以这里保持为 0.

```
/*Images pixels with this color will not be drawn (with chroma keying)*/
#define LV_COLOR_TRANSP    LV_COLOR_LIME           /*LV_COLOR_LIME: pure green*/
这个主要是用在图片控件上, LV_COLOR_LIME 就是指定要透明的颜色, LV_COLOR_LIME 这里表示绿色, 如果我显示一幅图片, 这个图片绿色部分就会变成透明的。
```

```
/* Enable anti-aliasing (lines, and rads)
#define LV_ANTIALIAS          1
```

使能抗锯齿, 如果不设置为 1, 你的文字边沿就有锯齿感, 不光滑。我这里设置为 1。
设置为 1 使能抗锯齿要多占用 5KB 的 flash 空间, 如果你显示得字体小其实可以不用抗锯齿, 看都看不出来锯齿感。

```
/* Default display refresh period.
 * Can be changed in the display driver (`lv_disp_drv_t`).
#define LV_DISP_DEF_REFR_PERIOD      30             /*[ms]*/
```

屏幕刷新周期, 这里设置的 30ms 刷新一次。

```
/* Dot Per Inch: used to initialize default sizes.
 * E.g. a button with width = LV_DPI / 2 -> half inch wide
 * (Not so important, you can adjust it to modify default sizes and spaces)
#define LV_DPI                  60               /*[px]*/
```

调节界面控件整体显示放大比例

```
/* Type of coordinates. Should be `int16_t` (or `int32_t`)
typedef int16_t lv_coord_t;
```

设置坐标像素最大能到多少, 这里是 320 x 240, 所以 int16_t 完全够用。

```
/* 1: use custom malloc/free, 0: use the built-in `lv_mem_alloc` */
#define LV_MEM_CUSTOM          0
#if LV_MEM_CUSTOM == 0
/* Size of the memory used by `lv_mem_alloc` in bytes (>= 2kB) */
#define LV_MEM_SIZE            (16U * 1024U)
```

LV_MEM_CUSTOM 设置为 0, 就是使用 lvgl 自己默认的内存管理, 如果设置为 1, 你就必须给 lvgl 指定内存管理函数 malloc 和 free 了, 我们一般使用 lvgl 默认的。

这里 $16U * 1024U = 16k$ 能满足大部分 lvgl 应用, 如果你界面很复杂, 层级很多, 你就自己修改内存大小。

```
/* Set an address for the memory pool instead
 * Can be in external SRAM too. */
#define LV_MEM_ADR              0
```

LV_MEM_ADR 是你如果使用外部 SRAM 来加速图形显示, 让图形更流畅, 你就可以给 ADR 设置地址来重定向堆空间地址。

对于正点原子战舰开发板, 外部 SRAM 地址为 0x6800 0000

```
# define LV_MEM_ADR  (0x6800 0000 + LV_HOR_RES_MAX* LV_VER_RES_MAX*2) //这就是
外部 SRAM 起始堆空间地址
```

LV_HOR_RES_MAX* LV_VER_RES_MAX*2 为什么要加上显示缓冲区地址, 让堆空间地址在显示缓冲区之后呢? 这是因为你重定向之后, lvgl 默认的显示缓冲区地址也指定到 SRAM 了。所以并不是 STM32 内部 SRAM 存显示缓冲区, 外部 SRAM 地址存堆空间, 不是这样的。

```
/* Automatically defrag. on free. Defrag. if
# define LV_MEM_AUTO_DEFRAG 1
#else /*LV_MEM_CUSTOM*/
# define LV_MEM_CUSTOM_INCLUDE <stdlib.h>
# define LV_MEM_CUSTOM_ALLOC malloc
# define LV_MEM_CUSTOM_FREE free
#endif /*LV_MEM_CUSTOM*/
```

LV_MEM_AUTO_DEFRAG 这里置 1，是 lvgl 在空闲的时候对内存碎片进行自动整理。

```
/* Garbage Collector settings
 * Used if lvgl is binded to higher level language and
#define LV_ENABLE_GC 0
#if LV_ENABLE_GC != 0
# define LV_GC_INCLUDE "gc.h"
# define LV_MEM_CUSTOM_REALLOC your_realloc
# define LV_MEM_CUSTOM_GET_SIZE your_mem_get_size
#endif /* LV_ENABLE_GC */
```

内存垃圾回收机制，使用 java 或者 C++ 可能会用到，我们这里用不到，设置为 0。

```
/* Input device read period in milliseconds */
#define LV_INDEV_DEF_READ_PERIOD 30
```

每间隔 30ms 会读取触摸屏状态，如果你时间设置太长，可能触摸效果就会不好。

```
/* Drag threshold in pixels */
#define LV_INDEV_DEF_DRAG_LIMIT 10
```

当你滑动某个对象时候，你至少移动这个对象 10 个像素，系统才认为你是拖拽操作。

```
/* Drag throw slow-down in [%]. Greater value ->
#define LV_INDEV_DEF_DRAG_THROW 20
```

当你拖动某个对象的时候，在屏幕上一滑，滑动之后这个对象不会马上停下来，而是对象向前滑动一段时间才会停下来，如果 LV_INDEV_DEF_DRAG_THROW 这个值设置很大，对象就会很快停下来。

```
/* Long press time in milliseconds.
 * Time to send `LV_EVENT_LONG_PRESSED` */
#define LV_INDEV_DEF_LONG_PRESS_TIME 400
```

就是长按某个对象，长按多少毫秒(ms)，该对象才会执行操作，我这里设置的 400ms。

```
/* Repeated trigger period in long press [ms]
 * Time between `LV_EVENT_LONG_PRESSED_REPEAT` */
#define LV_INDEV_DEF_LONG_PRESS_REPEAT_TIME 100
```

在每间隔 100ms 系统认为是重复长按操作

```
/*1: Enable the Animations */
#define LV_USE_ANIMATION           1
#if LV_USE_ANIMATION
```

设置为 1 使能动画，为 0 不使能动画。

```
/* 1: Enable shadow drawing*/
#define LV_USE_SHADOW             1
```

阴影效果是否使能，为 1 使能阴影效果。

```
/* 1: Enable object groups (for keyboard/encoder navigation) */
#define LV_USE_GROUP              1
#if LV_USE_GROUP
typedef void * lv_group_user_data_t;
#endif /*LV USE GROUP*/
```

分组，就是通过按钮进行上下左右导航，默认情况下我们使能，置 1。

```
/* 1: Enable GPU interface*/
#define LV_USE_GPU                 0
```

不使用 GPU

```
/* 1: Enable file system (might be reenabled later)
#define LV_USE_FILESYSTEM          0
#if LV_USE_FILESYSTEM
```

lvgl 自带的文件系统，我们不使用

```
/*1: Add a `user_data` to drivers and objects*/
#define LV_USE_USER_DATA           1
```

是否使用用户数据，其实就是句柄(对象)之间传参，或者句柄(对象)自带数据的功能，我们下面演示下。

```
#if LV_USE_OBJ_REALIGN
    lv_realign_t realign;      /**< Information about the last call to
#endif

#if LV_USE_USER_DATA
    lv_obj_user_data_t user_data; /***< Custom user data for object. */
#endif
```

就是 `lv_obj_t` 创建的句柄(对象)，拥有传输数据的能力。
可以传任意类型的数据，然后接受函数用指定类型去解析。

`typedef void * lv_obj_user_data_t;`
你看数据类型是 `void *` 就是可以

```

typedef struct{
    u16 newdata; //自己定义的数据，像传给一个对象
}MY_DATA;

MY_DATA my_data = { 100,200 };

void change_obj_pos( lv_obj_t* obj ) //如果需要修改obj对象什么内容，就传参进来在里面修改
{
    MY_DATA* data; //这里你要创建你知道obj携带的哪种数据类型的数据变量，用来接收obj携带的数据
    data = ( MY_DATA *)lv_obj_get_user_data( obj ); //将obj携带数据解析出来，这里一定要强转
    data = { 500 , 200 }; //修改data数据就相当于修改obj携带的MY_data数据
}

void main()
{
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL); //创建一个对象给obj，lv_scr_act()表示这个屏幕的基础对象，也就是后面创建对象的最父的一个对象
    lv_obj_set_user_data( obj , ( void *)&my_data); //将my_data数据挂载进obj
    change_obj_pos(obj); //现在obj携带着my_data数据传递给change....函数
}

```

这就是#define LV_USE_USER_DATA 1 的作用，让创建的对象可以携带数据。

```

/*1: Enable the log module*/
#define LV_USE_LOG      1
#ifndef LV_USE_LOG
/* How important log should be added:
 * LV_LOG_LEVEL_TRACE      A lot of logs to give detailed information
 * LV_LOG_LEVEL_INFO        Log important events
 * LV_LOG_LEVEL_WARN        Log if something unwanted happened but didn't cause a
 * LV_LOG_LEVEL_ERROR       Only critical issue, when the system may fail
 * LV_LOG_LEVEL_NONE        Do not log anything
 */

```

新版本 6.0 加入了日志模块，我们用不到，直接置 0。可以节省 3K flash。

```

/*=====
 *  THEME USAGE
 =====*/
#define LV_THEME_LIVE_UPDATE      1 /*1: Allow them
#define LV_USE_THEME_TEMPL      1 /*Just for test
#define LV_USE_THEME_DEFAULT     1 /*Built mainly
#define LV_USE_THEME_ALIEN       1 /*Dark futurist
#define LV_USE_THEME_NIGHT       1 /*Dark elegant
#define LV_USE_THEME_MONO        1 /*Mono color th
#define LV_USE_THEME_MATERIAL    1 /*Flat theme wi
#define LV_USE_THEME_ZEN         1 /*Peaceful, maj
#define LV_USE_THEME_NEMO        1 /*Water-like th

```

这就是官方自带的 8 种主题。

```

/* Enable it if you have fonts with
 * The limit depends on the font size
 * but with > 10,000 characters if you
#define LV_FONT_FMT_TXT_LARGE   1

```

如果你有一个字符，超过 10000 个字体，那么这里就需要置 1，针对大字体操作。

```

/* Select a character encoding for strings.
 * Your IDE or editor should have the same character encoding
 * - LV_TXT_ENC_UTF8
 * - LV_TXT_ENC_ASCII
 */
#define LV_TXT_ENC LV_TXT_ENC_UTF8

```

设置 utf8 编码方式，这样可以显示全球的字符编码。

lv_conf.h 后面的内容也都是些控件的使用或者不使用的定义，如果你想节省内存 sram 和 flash，那么不用的控件你就写 0。

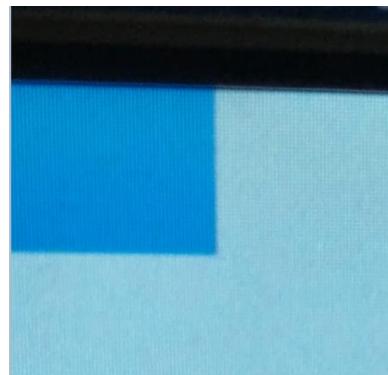
littleVGL 基础 API 接口

```
lv_obj_t* lv_obj_create( lv_obj_t* parent , const lv_obj_t* copy) //创建一个对象
```

parent: 传入父对象的 obj, 如果为 NULL, 那么自动创建一个 screen 屏幕, 这个屏幕就是基础对象。

copy: 表示创建这个对象的时候, 可以把其它对象的参数复制给我现在创建这个对象。Copy 就是用来接收其它对象的。

```
lv_init(); //lv初始化  
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象  
  
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



在屏幕左上角有个蓝色的框，这个蓝色的框就是 obj 对象。

```
lv_res_t lv_obj_del( lv_obj_t* obj ) //立即删除对象
```

obj: 填入要删除的对象

```
lv_init(); //lv初始化  
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象  
  
lv_obj_t* btn1 = lv_btn_create(scr, NULL); //创建按钮对象  
lv_obj_set_pos(btn1, 100, 50); //设置按钮坐标位置x=200, y=50  
  
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果
```



按钮显示

这是我创建的按钮对象，
没有删除，下面有显示

```
lv_init(); //lv初始化  
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
```

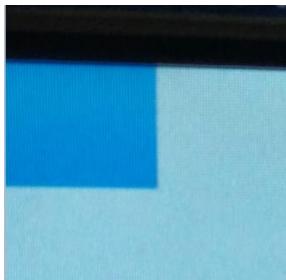
```
lv_obj_t* btn1 = lv_btn_create(scr, NULL); //创建按钮对象
```

```
lv_obj_set_pos(btn1, 100, 50); //设置按钮坐标位置x=200, y=50
```

```
lv_obj_del(btn1); //立即删除按钮对象
```

```
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```

现在我删除按钮对象



你看，按钮不见了，被删除了。

```
lv_obj_del_async(lv_obj_t* obj) //异步删除对象，就是要等到执行 lv_task_handler(); 才会  
删除掉对象。
```

obj: 填入要删除的对象

```
lv_init(); //lv初始化
```

```
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

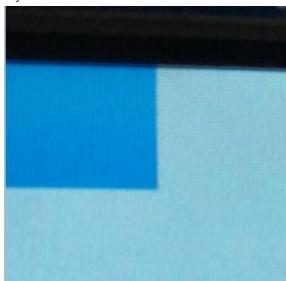
```
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
```

```
lv_obj_t* btn1 = lv_btn_create(scr, NULL); //创建按钮对象
```

```
lv_obj_set_pos(btn1, 100, 50); //设置按钮坐标位置x=200, y=50
```

```
lv_obj_del_async(btn1); //不会立刻删除对象，等着while里面的lv_task_handler();执行了才删除对象
```

```
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



按钮在执行 lv_task_handler(); 之后被删除了。

```

lv_obj_clean(lv_obj_t* obj) //删除当前对象下的所有子对象
obj: 填入父对象，系统会自动删除父对象下面的子对象
    lv_init(); //lv初始化
    lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后

    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

    lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
    lv_obj_set_size(obj, 200, 200); //obj大小

    lv_obj_t* child1 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child1
    lv_obj_set_size(child1, 50, 50); //设置child1按钮大小
    lv_obj_set_pos(child1, 50, 50); //设置child1按钮在obj对象里面的位置

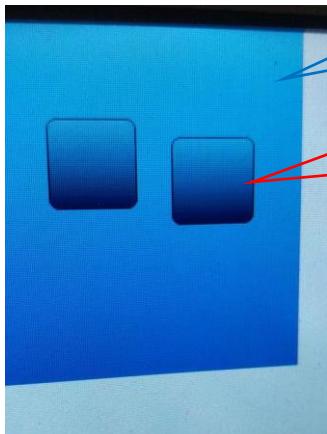
    lv_obj_t* child2 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child2
    lv_obj_set_size(child2, 50, 50); //设置child2按钮大小
    lv_obj_set_pos(child2, 120, 60); //设置child2按钮在obj对象里面的位置

```

```

while(1)
{
    lv_task_handler(); //这句执行后有界面效果
}

```



这就是 obj 下面的子对象，两个按钮，我现在要一次性删除这两个按钮

```

lv_init(); //lv初始化
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
lv_obj_set_size(obj, 200, 200); //obj大小

lv_obj_t* child1 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child1
lv_obj_set_size(child1, 50, 50); //设置child1按钮大小
lv_obj_set_pos(child1, 50, 50); //设置child1按钮在obj对象里面的位置

lv_obj_t* child2 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child2
lv_obj_set_size(child2, 50, 50); //设置child2按钮大小
lv_obj_set_pos(child2, 120, 60); //设置child2按钮在obj对象里面的位置

```

lv_obj_clean(obj); //删除obj下面的所有子对象，保留obj对象

```

while(1)
{
    lv_task_handler(); //这句执行后有界面效果
}

```



两个按钮子对象一次性删除成功。

删除所有子对象

```

lv_obj_move_background(lv_obj_t* obj); //将对象的层级后置

lv_init(); //lv初始化
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
lv_obj_set_size(obj, 200, 200); //obj大小

lv_obj_t* child1 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child1
lv_obj_set_size(child1, 50, 50); //设置child1按钮大小
lv_obj_set_pos(child1, 100, 50); //设置child1按钮在obj对象里面的位置

lv_obj_t* child2 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child2
lv_obj_set_size(child2, 50, 50); //设置child2按钮大小
lv_obj_set_pos(child2, 120, 60); //设置child2按钮在obj对象里面的位置

while(1)
{
    lv_task_handler(); //这句执行后有界面效果
}

```



你看按钮 2 对象是最后创建的，覆盖在按钮 1 之上

```

lv_init(); //lv初始化
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
lv_obj_set_size(obj, 200, 200); //obj大小

lv_obj_t* child1 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child1
lv_obj_set_size(child1, 50, 50); //设置child1按钮大小
lv_obj_set_pos(child1, 100, 50); //设置child1按钮在obj对象里面的位置

lv_obj_t* child2 = lv_btn_create(obj, NULL); //在obj对象之下创建子对象child2
lv_obj_set_size(child2, 50, 50); //设置child2按钮大小
lv_obj_set_pos(child2, 120, 60); //设置child2按钮在obj对象里面的位置

```

```

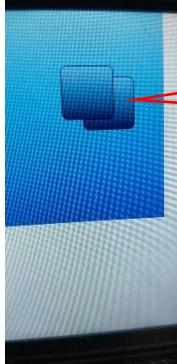
lv_obj_move_background(child2); //将按钮2对象层级后置

```

```

while(1)
{
    lv_task_handler(); //这句执行后有界面效果
}

```



将按钮 2 对象置后

```

lv_obj_move_foreground(lv_obj_t* obj); //将对象层级前置也就是按钮 2 在前面这就不演示了

```

```
lv_obj_set_pos(lv_obj_t* obj, lv_coord_t x, lv_coord_t y) //对象移动坐标设置  
obj: 填入对象  
x: 移动的 x 位置  
y: 移动的 y 位置  
  
lv_init() //lv初始化  
lv_port_disp_init() //lv接口初始化函数一定要放在lv_init()函数之后  
  
lv_obj_t *scr = lv_scr_act() //获取当前活跃的屏幕对象  
  
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象  
lv_obj_set_size(obj, 50, 50); //obj大小  
lv_obj_set_pos(obj, 100, 100); //设置对象坐标在x=100, y=100
```

```
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



移动到 100,100 的位置

```
lv_obj_set_size(lv_obj_t * obj, lv_coord_t w, lv_coord_t h) //设置对象图形大小  
  
lv_init() //lv初始化  
lv_port_disp_init() //lv接口初始化函数一定要放在lv_init()函数之后  
  
lv_obj_t *scr = lv_scr_act() //获取当前活跃的屏幕对象  
  
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象  
lv_obj_set_size(obj, 50, 50); //obj大小50, 50个像素  
lv_obj_set_pos(obj, 100, 100); //设置对象坐标在x=100, y=100  
  
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



其它按钮，控件的大小也是这样设置。

```
lv_obj_align(lv_obj_t * obj, const lv_obj_t * base, lv_align_t align, lv_coord_t x_mod, lv_coord_t y_mod) //对齐函数
```

obj: 要对齐的子对象

base: 子对象相对于什么对象对齐? 这里 base 就是子对象相对于的对象。

align: 对齐方式

LV_ALIGN_OUT_BOTTOM_LEFT base 对象底部, 左边沿开始对齐

LV_ALIGN_OUT_BOTTOM_RIGHT base 对象底部, 右边沿开始对齐

.....参数太多自行查枚举

x_mod 和 y_mod: 对齐后需要移动的间隔坐标。

```
lv_init() //lv初始化
```

```
lv_port_disp_init() //lv接口初始化函数一定要放在lv_init()函数之后
```

```
lv_obj_t *scr = lv_scr_act() //获取当前活跃的屏幕对象
```

```
lv_obj_t* obj = lv_obj_create(scr, NULL); //scr就是父对象
```

```
lv_obj_set_size(obj, 100, 100); //obj大小100, 100个像素
```

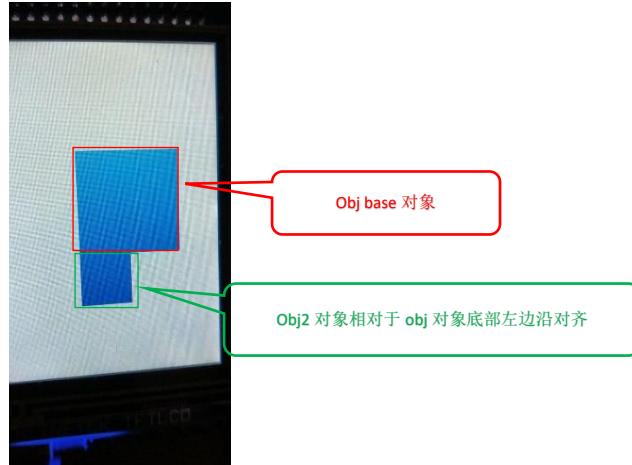
```
lv_obj_set_pos(obj, 100, 100); //设置对象坐标在x=100, y=100
```

```
lv_obj_t* obj2 = lv_obj_create(scr, NULL); //scr就是父对象
```

```
lv_obj_set_size(obj2, 50, 50); //obj大小50, 50个像素
```

```
lv_obj_align(obj2, obj, LV_ALIGN_OUT_BOTTOM_LEFT, 0, 0); //obj2在obj底部左边沿对齐
```

```
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



```
lv_obj_align(obj2, obj, LV_ALIGN_OUT_BOTTOM_RIGHT, 0, 0); //obj2在obj底部右边沿对齐
```



```
lv_obj_align(obj2, obj, LV_ALIGN_OUT_BOTTOM_RIGHT, 0, 20); //obj2在obj底部右边沿对齐, y_mod=20, y方向增加20间距
```



你看在 y 方向增加了 20 像素间距, x_mod 同理。

```
lv_obj_align(obj2, obj, LV_ALIGN_OUT_BOTTOM_RIGHT, -20, 20); //obj2在obj底部右边沿对齐, x为负就向左边移动, x为正就向右边移动
```



lv_label 标签控件

```
lv_label_create(lv_obj_t * par, const lv_obj_t * copy) //创建标签对象， 默认左上角
```

par： 标签对象需要在哪个父对象上面显示

copy:

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t * label = lv_label_create(scr, NULL); //创建标签对象  
lv_obj_set_pos(label, 50, 50); //设置标签位置
```

```
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



Test 是默认字符

```
lv_label_set_body_draw(lv_obj_t * label, bool en) //打开 label 对象背景颜色开关
```

label: 标签对象传入

en: true(背景颜色打开) false(背景颜色关闭)

```
static inline void lv_label_set_style(lv_obj_t * label, lv_label_style_t type, const lv_style_t * style)  
//设置标签背景颜色
```

label: 标签对象传入

type: 我们要设置哪一个部件的样式，这里我是标签部件 LV_LABEL_STYLE_MAIN

style: 传入样式颜色，我这里先用标签系统自带的样式&lv_style_plain_color(蓝色背景)

```
lv_obj_t * label = lv_label_create(scr, NULL); //创建标签对象  
lv_obj_set_pos(label, 50, 50); //设置标签位置  
lv_label_set_body_draw(label, true); //打开对象的背景颜色，这里是打开label标签的背景颜色  
lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &lv_style_plain_color); //打开label标签背景颜色开关后，需要用该函数设置背景颜色
```

```
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



标签背景颜色设置成功

```
lv_label_set_text(lv_obj_t * label, const char * text) //给标签写入自定义字符串
```

label: 标签对象传入

text: 输入你想写的字符串

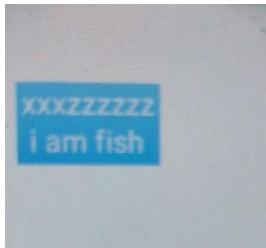
```
lv_obj_t * label = lv_label_create(scr, NULL); //创建标签对象  
lv_obj_set_pos(label, 50, 50); //设置标签位置  
lv_label_set_body_draw(label, true); //打开对象的背景颜色, 这里是打开label标签的背景颜色  
lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &lv_style_plain_color); //打开label标签背景颜色开关后, 需要用该函数设置背景颜色  
lv_label_set_text(label, "xxxxxxxx"); //设置标签文本, 也就是标签内容  
  
while(1)  
{  
    lv_task_handler(); //这句执行后有界面效果  
}
```



标签内容设置成功

'\n' 可以给标签字符换行显示

```
lv_label_set_text(label, "xxxxxxxx\n i am fish"); // '\n' 是可以换行显示得
```



```
lv_label_set_array_text(lv_obj_t * label, const char * array, uint16_t size)
```

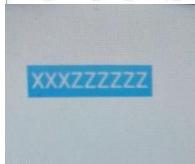
//传入数组变量来显示 label

label: 标签对象传入

array: 数组地址传入

size: 数组长度, 记得把数组末尾的结束符\0减去

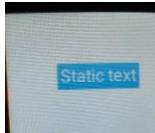
```
int main(void)  
{  
    char array[] = {"xxxxxxxx"};  
  
    lv_obj_t * label = lv_label_create(scr, NULL); //创建标签对象  
    lv_obj_set_pos(label, 50, 50); //设置标签位置  
    lv_label_set_body_draw(label, true); //打开对象的背景颜色, 这里是打开label标签的背景颜色  
    lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &lv_style_plain_color); //打开label标签背景颜色开关后, 需要用该函数设置背景颜色  
    lv_label_set_array_text(label, array, sizeof(array)-1); //用数组变量的方式给label标签显示字符串, sizeof(array)-1 将数组字符串最后的\0取消掉
```



lv_label_set_static_text(lv_obj_t * label, const char * text) //静态标签字符显示, 如果你的字符只需要固定显示, 不需要更换字符内容就用静态字符显示。因为系统不会为你字符开辟内存。

label: 标签对象传入

text: 输入显示得字符串



```

lv_label_set_long_mode(lv_obj_t * label, lv_label_long_mode_t long_mode) //label 文本模式设置
label: 标签对象传入
long_mode: LV_LABEL_LONG_EXPAND 长文本模式(长文本模式就是对象或者背景色跟着字符大小自动变化背景大小)
          LV_LABEL_LONG_BREAK 文本自动换行模式(需要先设置文本宽度)
          LV_LABEL_LONG_SCROLL 动态文本模式
          LV_LABEL_LONG_SCROLL_CIRC 循环滚动模式

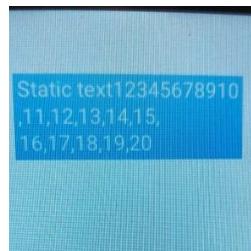
```

```

lv_obj_t * label = lv_label_create(scr, NULL); //创建标签对象
lv_obj_set_pos(label, 50, 50); //设置标签位置
lv_label_set_body_draw(label, true); //打开对象的背景颜色, 这里是打开label标签的背景颜色
lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &lv_style_plain_color); //打开label标签背景颜色开关

lv_label_set_long_mode(label, LV_LABEL_LONG_EXPAND); //长文本模式
lv_label_set_static_text(label, "Static text12345678910\n, 11, 12, 13, 14, 15, \n16, 17, 18, 19, 20");
//其实label标签默认就是长文本模式

```

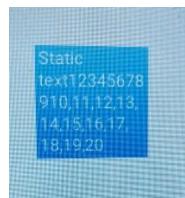


LV_LABEL_LONG_BREAK 使用

```

lv_label_set_long_mode(label, LV_LABEL_LONG_BREAK); //长文本自动换行模式
lv_obj_set_size(label, 100, 0); //设置标签宽度100, 高度是无效的
lv_label_set_static_text(label, "Static text12345678910, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20"); //我取消\n换行

```



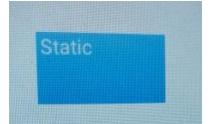
你看你的字符串没有\n, 当字符串长度达到标签宽度时, 自动换行。

LV_LABEL_LONG_SCROLL 使用

```

lv_label_set_long_mode(label, LV_LABEL_LONG_SCROLL); //文本长度超过标签宽度自动滚动显示
lv_obj_set_size(label, 100, 50); //设置标签宽度100, 文本滚动, 高度是有效的, 高度一定要大于字体高度
lv_label_set_static_text(label, "Static "); //文本会自动滚动

```

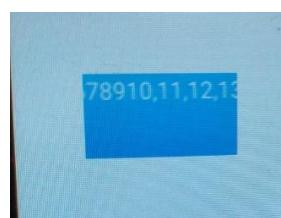
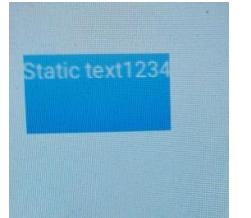


怎么文本没有滚动? 这是因为你字符串大小小于标签宽度。

```

lv_label_set_long_mode(label, LV_LABEL_LONG_SCROLL); //文本长度超过标签宽度自动滚动显示
lv_obj_set_size(label, 100, 50); //设置标签宽度100, 文本滚动, 高度是有效的, 高度一定要大于字体高度
lv_label_set_static_text(label, "Static text12345678910, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20"); //文本会自动滚动

```



你看 大于文本长度自动滚动。

LV_LABEL_LONG_SCROLL_CIRC 使用

```
lv_label_set_long_mode(label, LV_LABEL_LONG_SCROLL_CIRC); //循环滚动  
lv_obj_set_size(label, 100, 50); //设置标签宽度100, 文本滚动, 高度是有效的, 高度一定要大于字体高度  
lv_label_set_static_text(label, "Static text12345678910, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20");
```



滚动到字符末尾, 接着从第 1 个字符开始继续滚动。

lv_label_set_anim_speed(lv_obj_t * label, uint16_t anim_speed) //标签滚动速度设置

```
lv_label_set_long_mode(label, LV_LABEL_LONG_SCROLL_CIRC); //循环滚动  
lv_label_set_anim_speed(label, LV_LABEL_DEF_SCROLL_SPEED*3); //LV_LABEL_DEF_SCROLL_SPEED默认滚动速度 x 2  
lv_obj_set_size(label, 100, 50); //设置标签宽度100, 文本滚动, 高度是有效的, 高度一定要大于字体高度  
lv_label_set_static_text(label, "Static text12345678910, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20"); //文本会自动滚动
```

lv_label_set_align(lv_obj_t * label, lv_label_align_t align) //控件对齐方式

label: 标签对象传入

align: LV_LABEL_ALIGN_LEFT 左对齐

LV_LABEL_ALIGN_CENTER 中间对齐

LV_LABEL_ALIGN_RIGHT 右对齐

```
lv_label_set_long_mode(label, LV_LABEL_LONG_CROP); //一定设置成长文本模式
```

```
lv_obj_set_size(label, 100, 50); //设置标签宽度100
```

```
lv_label_set_static_text(label, "Static");
```

```
lv_label_set_align(label, LV_LABEL_ALIGN_CENTER); //文本居中
```



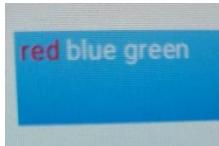
这就是居中, 一定是长文本模式下。

lv_label_set_recolor(lv_obj_t * label, bool en) //文本重新设置颜色功能开关

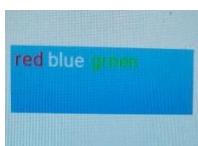
```
lv_label_set_long_mode(label, LV_LABEL_LONG_CROP); //一定设置成长文本模式  
lv_label_set_recolor(label, true); //使能文本重新设置字体颜色  
lv_obj_set_size(label, 150, 50); //设置标签宽度150  
lv_label_set_static_text(label, "#ff0000 red# blue green"); //直接用#号在字符串里添加颜色
```

字符串"##..." #号包裹的区域就是你要绘制的颜色号, 和你要绘制颜色的字符。

如: "#ff0000 red# blue green"



```
lv_label_set_static_text(label, "#ff0000 red# blue #00ff00 green#");
```



你看 两个#号 各包含一边, 就是两个颜色。

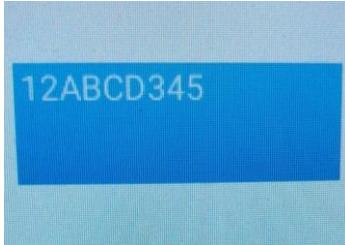
```

lv_label_ins_text(lv_obj_t * label, uint32_t pos, const char * txt) //插入文本
label: 标签对象传入
pos: 从第几个字符插入
txt: 插入字符串内容

lv_label_set_long_mode(label, LV_LABEL_LONG_CROP); //一定设置成长文本模式
lv_obj_set_size(label, 150, 50); //设置标签宽度150
lv_label_set_text(label, "12345"); //这是默认设置的文本，一定是动态文本才能做插入文本
lv_label_ins_text(label, 2, "ABCD"); //插入文本后 "12ABCD345"

```

lv_label_ins_text(标签, 我在第 2 个字符后开始插入, 插入内容)



你看在中间字符插入

lv_label_ins_text(标签, LV_LABEL_POS_LAST, 插入内容) //在字符串末尾插入内容

lv_style 样式系统, 界面美观设计

```

static inline void lv_label_set_style(lv_obj_t * label, lv_label_style_t type, const lv_style_t * style)
label: 标签对象传入
type: 设置哪一个控件颜色
style: 设置样式的结构体传入

```

```

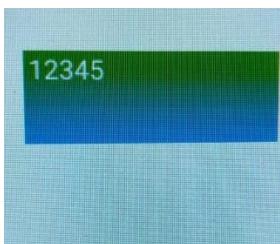
lv_style_t xzz_style; 定义一个样式的结构体对象
lv_style_copy(&xzz_style, &lv_style_plain_color); //将系统样式赋值给自己创建的样式, 这样后面设置样式才能成功
xzz_style.body.main_color = LV_COLOR_GREEN; //修改对象的背景颜色
lv_label_set_body_draw(label, true); //打开对象的背景颜色, 这里是打开 label 标签的背景颜色
lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &xzz_style); //修改标签背景颜色

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style, &lv_style_plain_color); //将系统样式赋值给自己创建的样式, 这样后面设置样式才能成功
xzz_style.body.main_color = LV_COLOR_GREEN; //修改对象的背景颜色, 下面传入的是标签对象, 那么就是修改标签背景颜色

lv_obj_t * label = lv_label_create(scr, NULL); //创建标签对象
lv_obj_set_pos(label, 50, 50); //设置标签位置
lv_label_set_body_draw(label, true); //打开对象的背景颜色, 这里是打开label标签的背景颜色
lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &xzz_style); //修改标签背景颜色

lv_label_set_long_mode(label, LV_LABEL_LONG_CROP); //一定设置成长文本模式
lv_obj_set_size(label, 150, 50); //设置标签宽度150
lv_label_set_text(label, "12345");

```



确实上一行设置成绿色背景了, 但是为什么下面半行不是绿色的?

body..... 对象背景

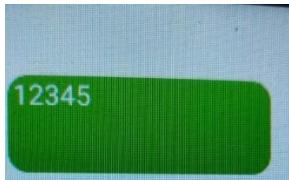
```
lv_color_t body.main_color //我们设置主背景颜色  
lv_color_t body.grad_color //如果 grad_color != main_color , 那么背景色默认是渐变颜色  
lv_style_t xzz_style; //创建样式对象  
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式，这样后面设置样式才能成功  
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色  
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和main_color一样的色  
  
lv_obj_t * label = lv_label_create(scr,NULL); //创建标签对象  
lv_obj_set_pos(label,50,50); //设置标签位置  
lv_label_set_body_draw(label,true); //打开对象的背景颜色，这里是打开label标签的背景颜色  
lv_label_set_style(label,LV_LABEL_STYLE_MAIN,&xzz_style); //修改标签背景颜色  
  
lv_label_set_long_mode(label,LV_LABEL_LONG_CROP); //一定设置成长文本模式  
lv_obj_set_size(label,150,50); //设置标签宽度150  
lv_label_set_text(label,"12345");
```



你看背景色渐变问题解决了。

body.radius //设置圆角半径

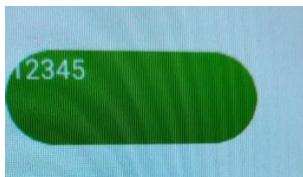
```
lv_style_t xzz_style; //创建样式对象  
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式，这样后面设置样式才能成功  
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色  
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和main_color一样的色  
xzz_style.body.radius = 10; //设置圆角半径为10  
  
lv_obj_t * label = lv_label_create(scr,NULL); //创建标签对象  
lv_obj_set_pos(label,50,50); //设置标签位置  
lv_label_set_body_draw(label,true); //打开对象的背景颜色，这里是打开label标签的背景颜色  
lv_label_set_style(label,LV_LABEL_STYLE_MAIN,&xzz_style); //修改标签背景颜色  
  
lv_label_set_long_mode(label,LV_LABEL_LONG_CROP); //一定设置成长文本模式  
lv_obj_set_size(label,150,50); //设置标签宽度150  
lv_label_set_text(label,"12345");
```



你看，四边倒角 10px 半径。

body.radius //根据对象高度自动设置全圆角

```
lv_style_t xzz_style; //创建样式对象  
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式，这样后面设置样式才能成功  
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色  
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和main_color一样的色  
xzz_style.body.radius = LV_RADIUS_CIRCLE; //根据对象高度自动设置全圆角LV_RADIUS_CIRCLE  
  
lv_obj_t * label = lv_label_create(scr,NULL); //创建标签对象  
lv_obj_set_pos(label,50,50); //设置标签位置  
lv_label_set_body_draw(label,true); //打开对象的背景颜色，这里是打开label标签的背景颜色  
lv_label_set_style(label,LV_LABEL_STYLE_MAIN,&xzz_style); //修改标签背景颜色
```



你看 圆角更全更规范。

body.opa //背景透明度

```
lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式，这样xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和
xzz_style.body.radius = LV_RADIUS_CIRCLE; //根据对象高度自动设置全圆角LV_RADIUS_CIRCLE
xzz_style.body.opa = 100; //背景色透明度100，就是不色很透明
```



body.opa = 50 背景半透明

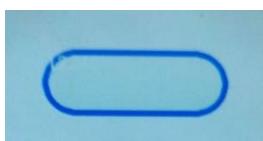
```
lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和
xzz_style.body.radius = LV_RADIUS_CIRCLE; //根据对象高度自动设置全圆角LV_RADIUS_CIRCLE
xzz_style.body.opa = 50; //背景色透明度50，就是半透明了
```



body.opa = 10 就是完全透明

body.border.color //边框设置
body.border.width //边框宽度

```
lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和
xzz_style.body.radius = LV_RADIUS_CIRCLE; //根据对象高度自动设置全圆角LV_RADIUS_CIRCLE
xzz_style.body.opa = 50; //背景色透明度50，就是半透明了
xzz_style.body.border.color = LV_COLOR_BLUE; //边框颜色为蓝色
xzz_style.body.border.width = 5; //边框宽度
```



body.border.part //边框类型

body.border.opa //边框透明度

```
lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和
xzz_style.body.radius = LV_RADIUS_CIRCLE; //根据对象高度自动设置全圆角LV_RADIUS_CIRCLE
xzz_style.body.opa = 50; //背景色透明度50，就是半透明了
xzz_style.body.border.color = LV_COLOR_BLUE; //边框颜色为蓝色
xzz_style.body.border.width = 5; //边框宽度
xzz_style.body.border.part = LV_BORDER_TOP; //LV_BORDER_TOP边框显示在上边位置
xzz_style.body.border.opa = 200; //边框透明度
```



```

body.shadow.color //对象背景阴影
body.shadow.width //对象阴影宽度
body.shadow.type //设置四边都有阴影

lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style, &lv_style_plain_color); //将系统样式赋值给自己
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，这里设置的20px
xzz_style.body.radius = LV_RADIUS_CIRCLE; //根据对象高度自动设置全圆角

xzz_style.body.border.color = LV_COLOR_BLUE; //边框颜色为蓝色
xzz_style.body.border.width = 5; //边框宽度
xzz_style.body.border.opa = 200; //边框透明度

xzz_style.body.shadow.color = LV_COLOR_GREEN; //对象背景阴影为绿色
xzz_style.body.shadow.width = 50; //对象阴影宽度为50
xzz_style.body.shadow.type = LV_SHADOW_FULL; //设置四边都有阴影

```



body.padding.top //文本与背景边框上部内边距

这就是文本内容与边框的内部距离，我

这里设置的20px

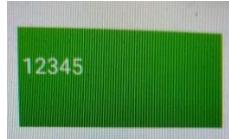


```

lv_style_t xzz_style; //创建样式对象
lv_style_copy(&xzz_style, &lv_style_plain_color); //将系统样式赋值给自己
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色

```

xzz_style.body.padding.top = 20; //文本与背景边框的内边距

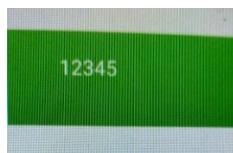


你看 12345 离顶部边框间距。

```

xzz_style.body.padding.top = 20; //文本与背景边框的内边距
xzz_style.body.padding.left = 40; //文本与左边框距离

```



```
text.color //文本颜色设置  
text.font //字体大小设置
```

```
lv_style_t xzz_style; //创建样式对象  
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式，这样后面设置样式才  
xzz_style.body.main_color = LV_COLOR_GREEN; //主背景色  
xzz_style.body.grad_color = LV_COLOR_GREEN; //为了背景色不色渐变颜色，给grad_color加入和main_color一  
  
xzz_style.body.padding.top = 20; //文本与背景边框的内边距  
xzz_style.body.padding.left = 40; //文本与左边框距离  
xzz_style.text.color = LV_COLOR_RED; //设置文本的颜色为红色  
xzz_style.text.font = &lv_font_roboto_28; //记住打开lv_conf.h的字体配置#define LV_FONT_ROBOTO_28
```



你看字体变成红色了，字体大小为 28 号

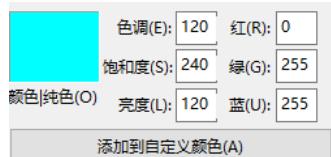
```
text.letter_space //字符之间列间距  
text.line_space //字符之间行间距  
xzz_style.text.color = LV_COLOR_RED; //设置文本的颜色为红色  
xzz_style.text.font = &lv_font_roboto_16;  
xzz_style.text.letter_space = 10; //字符之间列间距  
xzz_style.text.line_space = 3; //两排字符之间行间距
```



```
text.opa //字体透明度  
xzz_style.text.opa = 50; //字体半透明50
```

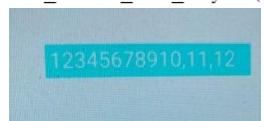


```
LV_COLOR_MAKE(r,g,b) //自定义颜色
```



```
lv_style_t xzz_style; //创建样式对象  
lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式，这样后  
xzz_style.body.main_color = LV_COLOR_MAKE(0, 255, 255); //自定义背景颜色  
xzz_style.body.grad_color = LV_COLOR_MAKE(0, 255, 255); //自定义背景颜色
```

```
lv_obj_t * label = lv_label_create(scr,NULL); //创建标签对象  
lv_obj_set_pos(label, 50, 50); //设置标签位置  
lv_label_set_body_draw(label, true); //打开对象的背景颜色，这里是打开label标签的背景颜色  
lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &xzz_style); //修改标签背景颜色
```



QTCreator 做 littleVGL 模拟器(为后面按钮事件驱动做铺垫)

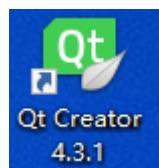
因为没有在开发板上面移植触摸屏驱动，所以我们先用 PC 模拟器来做按钮之类的驱动案例



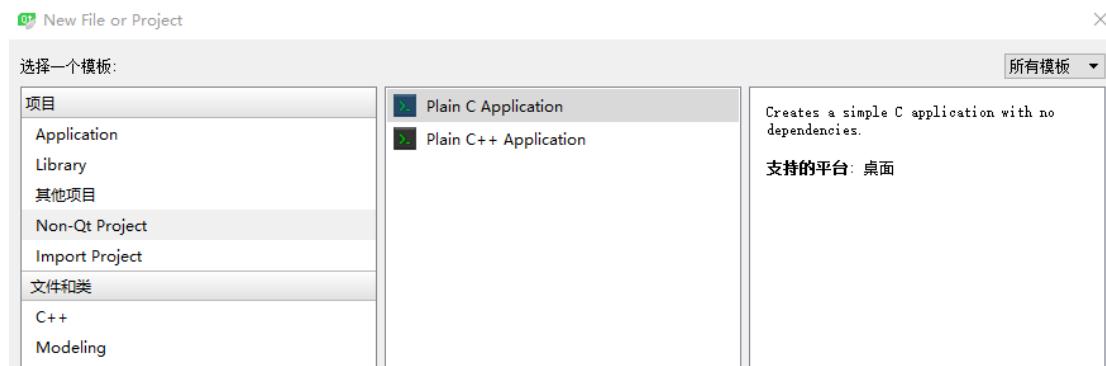
准备 lv_pc_simulator.zip 和 SDL2-devel-2.0.12-mingw.tar.gz 两个压缩包



在全英文路径下创建一个目录，我这里路径是 D:\PC_LittleVgl



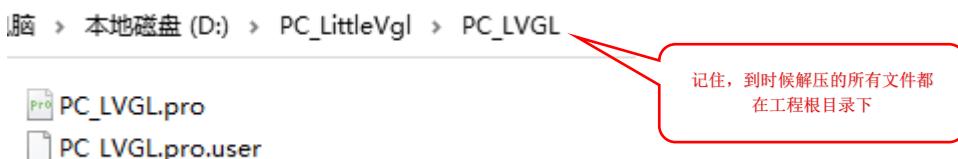
使用 Qtcreator 创建工程



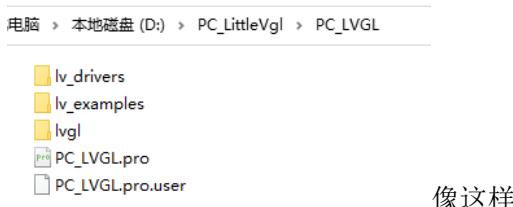
```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World!\n");
6     return 0;
7 }

```



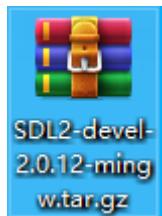
lv_drivers.zip
 lv_examples.zip
 lvgl.zip 将这三个文件解压到工程根目录下



	文件夹	2019/7/8 23:38
lv_drivers	39,887	39,887 WinRAR ZIP 压缩... 2019/7/8 23:57 B87F
lv_examples	2,073,134	2,073,134 WinRAR ZIP 压缩... 2019/7/8 23:57 8DF7
lvgl	732,584	732,584 WinRAR ZIP 压缩... 2019/7/8 23:57 1C56
PC_LVGL.pro	14,217	14,217 CPROJECT 文件 2019/7/8 23:38 2B86
PC_LVGL.pro.user	152	152 EDITORCONFIG ... 2019/7/8 23:38 1CF2
.cproject	70	70 GITIGNORE 文件 2019/7/8 23:38 A0D4
.editorconfig	270	270 GITMODULES 文件 2019/7/8 23:38 C887
.gitignore	925	925 PROJECT 文件 2019/7/8 23:38 5614
.gitmodules	661	661 TXT 文件 2019/7/8 23:38 B21E
.project	874	874 文件 2019/7/8 23:38 42B7
CMakeLists.txt	925	925 PROJECT 文件 2019/7/8 23:38 5614
Dockerfile	661	661 TXT 文件 2019/7/8 23:38 B21E
licence.txt	874	874 文件 2019/7/8 23:38 42B7
lv_conf.h	1,083	1,083 TXT 文件 2019/7/8 23:38 F19E
lv_drv_conf.h	15,490	15,490 H 文件 2019/7/8 23:38 895B
lv_ex_conf.h	9,593	9,593 H 文件 2019/7/8 23:38 7A2E
main.c	1,310	1,310 H 文件 2019/7/8 23:38 098A
Makefile	5,852	5,852 C 文件 2019/7/8 23:38 C109
mouse_cursor_icon.c	798	798 文件 2019/7/8 23:38 E8DC
	21,242	21,242 C 文件 2019/7/8 23:38 B591



工程像这样



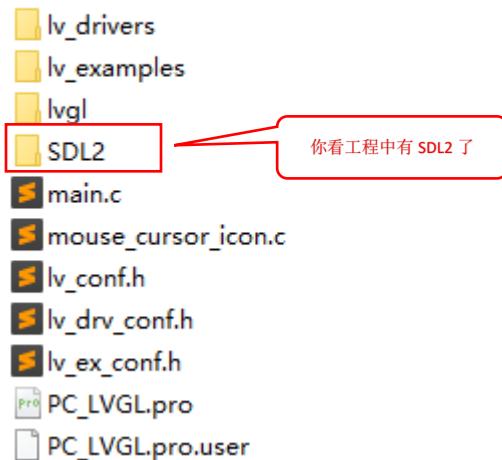
现在来解压 SDL2 里面的文件

SDL2-2.0.12\i686-w64-mingw32\include 文件夹下 SDL2 目录复制到工程根目录路径中

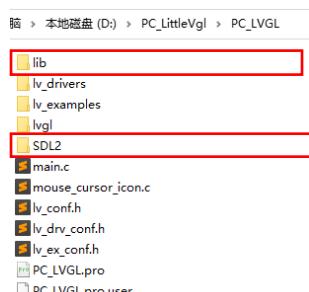


如果 Qt 用的是 mingw64, 则需要对应 SDL2-2.0.12\x86_64-w64-mingw32\include 文件夹下 SDL2 目录复制到工程根目录路径中, 我这里应该用不到。

脑 > 本地磁盘 (D:) > PC_LittleVgl > PC_LVGL



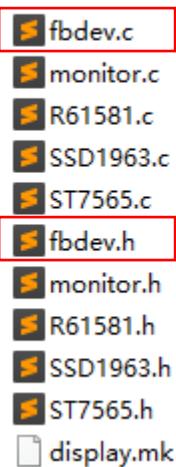
SDL2-2.0.12\i686-w64-mingw32 文件夹下 lib 目录复制到工程根目录路径下。



现在工程中目录齐全了。

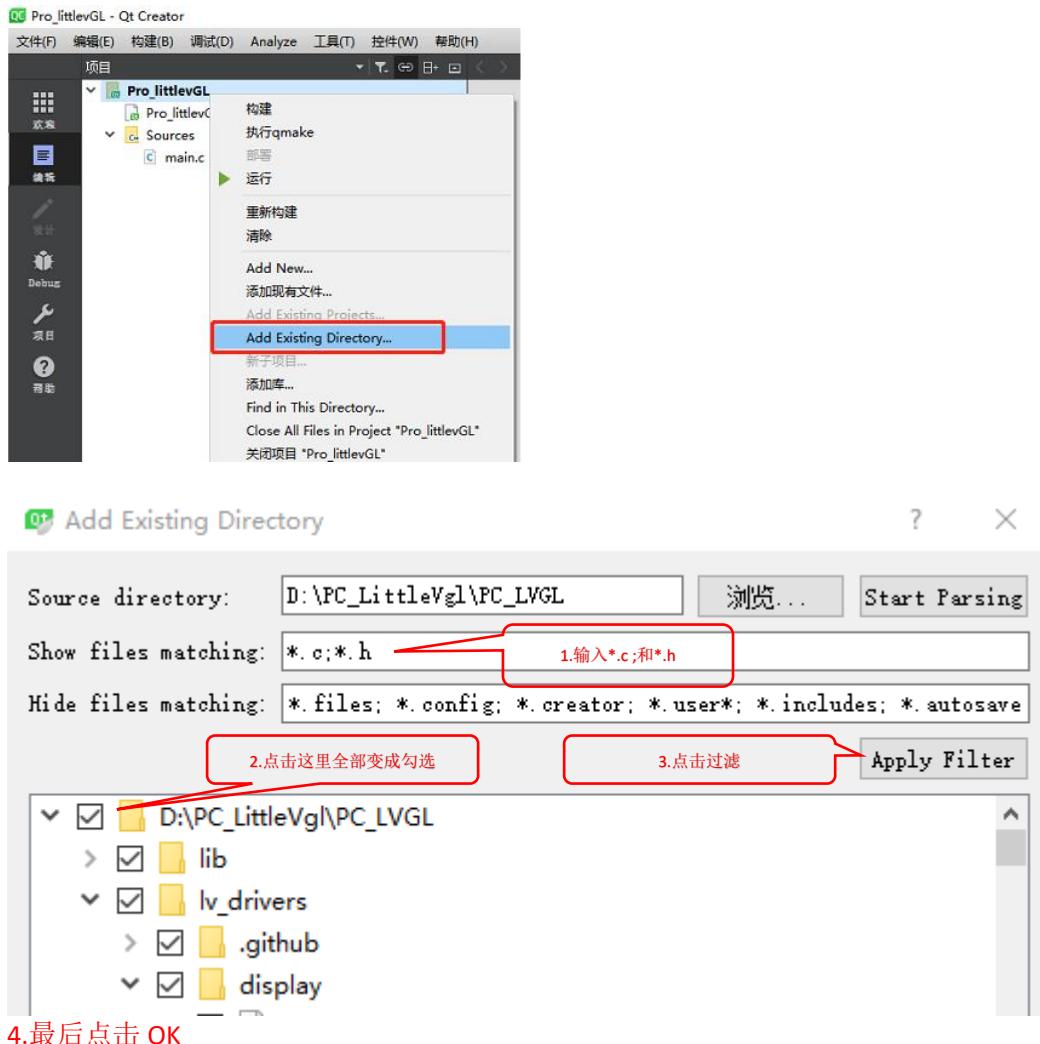
这里有个重点要注意的地方

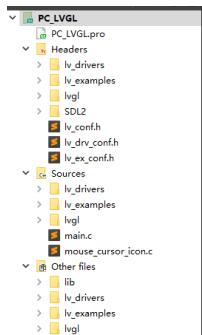
进入 D:\PC_LittleVgl\PC_LVGL\lv_drivers\display 目录



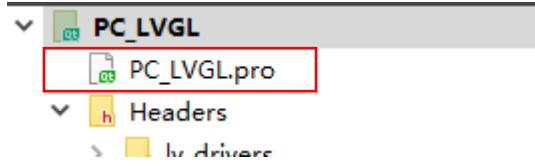
将 fbdev.c 和 fbdev.h 删除掉，这两个文件只有 linux 系统才使用，windows 使用会报错。

下面打开工程文件





加载进来之后目录结构就是这样



在.pro文件中加入库文件和头文件路径

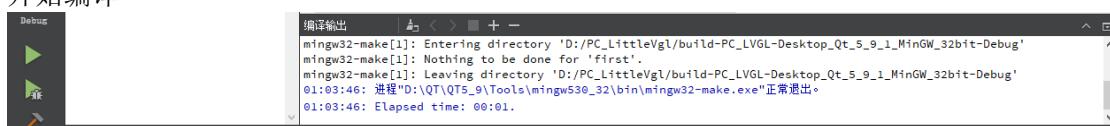
```
LIBS += -L$$PWD/lib/ -lmingw32 -lSDL2main -lSDL2
```

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

LIBS += -L$$PWD/lib/ -lmingw32 -lSDL2main -lSDL2

SOURCES += \
    lv_drivers/display/monitor.c \
    lv_drivers/display/R61581.c \
```

开始编译



编译成功



这是你没有加入 SDL2.dll 库，但是必须运行编译之后才能加入，因为生成了 debug 目录。

将 `SDL2-2.0.12\i686-w64-mingw32\bin` 里 `SDL2.dll` 文件复制到 `D:\PC_LittleVgl\build-PC_LVGL/Desktop_Qt_5_9_1_MinGW_32bit-Debug\debug` 目录中



如上所述：将这两个文件复制到你工程编译出来的 `debug` 目录中

再次编译运行



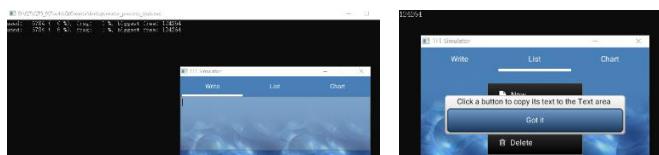
有界面了。

下面进入 main.c 文件，找个 demo 来跑跑

```
lv_ex_conf.h
Sources
lv_drivers
lv_examples
lvgl
main.c
mouse_cursor_icon.c
Other files
lib
lv_drivers
lv_examples
lvgl

59 int main(int argc, char ** argv)
60 {
61     (void) argc;      /*Unused*/
62     (void) argv;      /*Unused*/
63
64     /*Initialize LittlevGL*/
65     lv_init();
66
67     /*Initialize the HAL (display, input devices, tick) for
68      hal_init();
69
70
71     /*Select display 1*/
72     demo_create();
```

取消掉 demo_create 注释，运行看看。



运行成功，后面可以开始在 main 中加入你的调试代码了。

如何在框架中加入自己的代码？

```
int main(int argc, char ** argv)
{
    (void) argc;      /*Unused*/
    (void) argv;      /*Unused*/
    /*Initialize LittlevGL*/
    lv_init();
    /******加入自己代码******/
    lv_style_t xzz_style; //因为是子函数使用样式结构，所以只有定义为全局
    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
    lv_style_copy(&xzz_style,&lv_style_plain_color); //将系统样式赋值给自己创建的样式,
    xzz_style.body.main_color = LV_COLOR_GREEN; //自定义背景颜色
    xzz_style.body.grad_color = LV_COLOR_GREEN; //自定义背景颜色
    lv_obj_t * label = lv_label_create(scr,NULL); //创建标签对象
    lv_obj_set_pos(label,50,50); //设置标签位置
    lv_label_set_body_draw(label,true); //打开对象的背景颜色，这里是打开label标签的背景
    lv_label_set_style(label,LV_LABEL_STYLE_MAIN,&xzz_style); //修改标签背景颜色
    lv_label_set_long_mode(label,LV_LABEL_LONG_BREAK); //一定设置成长文本模式
    lv_obj_set_size(label,150,50); //设置标签宽度150
    lv_label_set_text(label,"12345678910,11,12");
    /*******/
    while(1) {
        /* Periodically call the lv_task handler.
         * It could be done in a timer interrupt or an OS task too.*/
        lv_task_handler();
        usleep(5 * 1000);
    }
#endif
```

执行成功

在主函数中删除 DEMO 成，然后加入们自己的代码。但是 while(1) 后面的事件循环函数要保留



lv_cont 容器(可以当窗体使用)

lv_cont 容器就是用来存放各种类型控件或者各种子对象。

```
lv_obj_t * lv_cont_create(lv_obj_t * par, const lv_obj_t * copy) //创建容器  
par: 传入容器跟随的父对象  
copy: 可以填 NULL
```

```
lv_init();  
  
/*Initialize the HAL (display, input devices, tick) for LittlevGL*/  
hal_init();  
*****加入自己代码*****  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *cont = lv_cont_create(scr, NULL); //创建一个容器  
lv_obj_align(cont, NULL, LV_ALIGN_CENTER, 0, 0); //容器居中对齐
```



我们向容器窗口添加两个按钮

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *cont = lv_cont_create(scr, NULL); //创建一个容器  
lv_obj_align(cont, NULL, LV_ALIGN_CENTER, 0, 0); //容器居中对齐  
  
lv_obj_t * btn1 = lv_btn_create(cont, NULL); //向容器添加1个按钮  
lv_obj_t * btn2 = lv_btn_create(cont, NULL); //向容器添加2个按钮
```



下面用容器的布局管理器来设置两个按钮对称分开

```
void lv_cont_set_layout(lv_obj_t * cont, lv_layout_t layout) //这其实就是容器的布局管理器  
cont: 容器对象  
layout: 布局对齐方式 LV_LAYOUT_CENTER, LV_LAYOUT_COL_L .....还要很多下一页会讲到。
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *cont = lv_cont_create(scr, NULL); //创建一个容器  
lv_obj_align(cont, NULL, LV_ALIGN_CENTER, 0, 0); //容器居中对齐  
  
lv_obj_t * btn1 = lv_btn_create(cont, NULL); //向容器添加1个按钮  
lv_obj_t * btn2 = lv_btn_create(cont, NULL); //向容器添加2个按钮  
  
lv_cont_set_layout(cont, LV_LAYOUT_CENTER); //布局管理器LV_LAYOUT_CENTER以中心的方式将对象对称排列
```



按钮以中心布局分离开了，但是感觉这两个按钮不完整？

这是因为按钮大小超过了容器大小

```
static inline void lv_cont_set_fit(lv_obj_t * cont, lv_fit_t fit) //容器大小变化自动设置  
cont: 容器对象  
fit: LV_FIT_TIGHT 大小自适应  
     LV_FIT_NONE 取消大小自适应
```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t *cont = lv_cont_create(scr, NULL); //创建一个容器
lv_obj_align(cont, NULL, LV_ALIGN_CENTER, 0, 0); //容器居中对齐

lv_obj_t *btn1 = lv_btn_create(cont, NULL); //向容器添加1个按钮
lv_obj_t *btn2 = lv_btn_create(cont, NULL); //向容器添加2个按钮

lv_cont_set_layout(cont, LV_LAYOUT_CENTER); //布局管理器LV_LAYOUT_CENTER以中心的方式将对象对称排列
lv_cont_set_fit(cont, LV_FIT_TIGHT); //容器大小随里面对象的大小自适应变化

```



你看容器大小随两个按钮随意变化

我们把容器变很大，来看看 `lv_cont_set_layout` 函数能进行哪些对齐方式

LV_LAYOUT_CENTER 中心列对齐

```

lv_cont_set_layout(cont, LV_LAYOUT_CENTER); //布局管理器LV_LAYOUT_CENTER以中心的方式将对象对称排列
lv_cont_set_fit(cont, LV_FIT_NONE); //取消容器大小自适应变化
lv_obj_set_size(cont, 300, 300);
lv_obj_set_pos(cont, 0, 0);

```



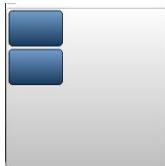
这是中心列对齐方式

LV_LAYOUT_COL_L 左上角列排列对齐

```

lv_cont_set_layout(cont, LV_LAYOUT_COL_L); //LV_LAYOUT_COL_L左上角对齐排列
lv_cont_set_fit(cont, LV_FIT_NONE); //取消容器大小自适应变化
lv_obj_set_size(cont, 300, 300);
lv_obj_set_pos(cont, 0, 0);

```



LV_LAYOUT_COL_M 列 顶部中间对齐

```

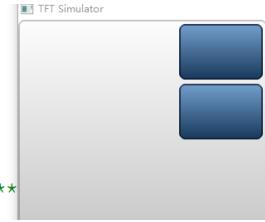
lv_cont_set_layout(cont, LV_LAYOUT_COL_M); //LV_LAYOUT_COL_M 列 顶部中间对齐
lv_cont_set_fit(cont, LV_FIT_NONE); //取消容器大小自适应变化
lv_obj_set_size(cont, 300, 300);
lv_obj_set_pos(cont, 0, 0);

```



LV_LAYOUT_COL_R 列 右上角对齐

```
lv_cont_set_layout(cont,LV_LAYOUT_COL_R); //LV_LAYOUT_COL_R 列 右上角对齐  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);  
*****
```



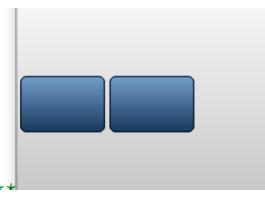
LV_LAYOUT_ROW_T 行 左上角对齐

```
lv_cont_set_layout(cont,LV_LAYOUT_ROW_T); //LV_LAYOUT_ROW_T 行 左上角对齐  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);  
*****
```



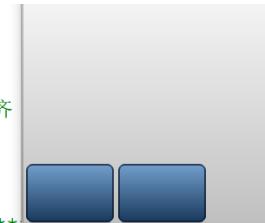
LV_LAYOUT_ROW_M 行 中间对齐

```
lv_cont_set_layout(cont,LV_LAYOUT_ROW_M); //LV_LAYOUT_ROW_M 行 中间对齐  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);  
*****
```



LV_LAYOUT_ROW_B 行 底部对齐

```
lv_cont_set_layout(cont,LV_LAYOUT_ROW_B); //LV_LAYOUT_ROW_B 行 底部对齐  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);  
*****
```



LV_LAYOUT_PRETTY 子对象之间间隔距离

```
lv_cont_set_layout(cont,LV_LAYOUT_PRETTY); //LV_LAYOUT_PRETTY 子对象之间间隔距离  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);  
*****
```



LV_LAYOUT_GRID 格子布局管理器，一定是 4 个子对象以上才有意义(类似 QT 格子布局管理器)

```
lv_obj_t * btn1 = lv_btn_create(cont,NULL); //向容器添加1个按钮  
lv_obj_t * btn2 = lv_btn_create(cont,NULL); //向容器添加2个按钮  
lv_obj_t * btn3 = lv_btn_create(cont,NULL); //向容器添加3个按钮  
lv_obj_t * btn4 = lv_btn_create(cont,NULL); //向容器添加4个按钮
```

```
lv_cont_set_layout(cont,LV_LAYOUT_GRID); //LV_LAYOUT_GRID 格子布局管理器，一定是4个子对象以上才有意义  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);  
*****
```



你看是不是和 QT 格子布局管理器很像，但是感觉间距太大。下面样式设置会解决这个问题

lv_cont 容器样式设置

body.padding.inner //容器子对象之间间距设置



你看本来容器中子对象之间间距这么近

```
lv_obj_t * btn1 = lv_btn_create(cont,NULL); //向容器添加1个按钮  
lv_obj_t * btn2 = lv_btn_create(cont,NULL); //向容器添加2个按钮
```

```
lv_cont_set_layout(cont,LV_LAYOUT_COL_L);  
lv_cont_set_fit(cont,LV_FIT_NONE); //取消容器大小自适应变化  
lv_obj_set_size(cont,300,300);  
lv_obj_set_pos(cont,0,0);
```

```
lv_style_copy(&cont_style,&lv_style_plain); //将系统自带的lv_style_plain样式拷贝过来  
cont_style.body.padding.inner = 50; //子对象之间间距50个像素
```

```
lv_cont_set_style(cont,LV_CONT_STYLE_MAIN,&cont_style); //将自己设置的样式传入生效
```

TFT Simulator



你看 对象之间间距增加了 50px。

body.padding.top //子对象与容器顶部边的间距

```
lv_style_copy(&cont_style,&lv_style_plain); //将系统自带的lv_style_plain样式拷贝过来  
cont_style.body.padding.inner = 50; //子对象之间间距50个像素  
cont_style.body.padding.top = 30; //子对象与容器顶部边的间距
```

```
lv_cont_set_style(cont,LV_CONT_STYLE_MAIN,&cont_style); //将自己设置的样式传入生效
```

TFT Simulator



body.padding.left //子对象与容器左边间距

```
lv_style_copy(&cont_style,&lv_style_plain); //将系统自带的lv_style_plain样式拷贝过来  
cont_style.body.padding.inner = 50; //子对象之间间距50个像素  
cont_style.body.padding.top = 30; //子对象与容器顶部边的间距  
cont_style.body.padding.left = 50; //子对象与容器左边的间距
```

```
lv_cont_set_style(cont,LV_CONT_STYLE_MAIN,&cont_style); //将自己设置的样式传入生效
```

TFT Simulator



以上代码在开发板上测试通过。

lv_btn 按钮控件



按钮状态分为

正常态

LV_BTN_STATE_REL : 按钮的正常释放状态

LV_BTN_STATE_PR : 按钮正常按下状态

切换状态 (toggle)

LV_BTN_STATE_TGL_REL: 按钮的切换释放状态

LV_BTN_STATE_TGL_PR: 按钮的切换按下状态

禁止态

LV_BTN_STATE_INA: 按钮的禁用无效状态



按钮从类型来分: 普通按钮和切换按钮(toggle)

```
lv_obj_t * lv_btn_create(lv_obj_t * par, const lv_obj_t * copy) //创建按钮
```

par: 按钮嵌入哪个父对象

copy: 创建新对象时, 可以把其它对象属性复制过来, 不用可以填 NULL

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮
```



```
void lv_btn_set_toggle(lv_obj_t * btn, bool tgl) //将按钮设置为切换按钮
```

btn: 哪一个普通按钮要变成切换按钮, 那么就传入这个按钮的对象

tgl: false 不打开切换按钮功能, true 打开切换按钮功能

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮
```

```
lv_btn_set_toggle(btn1, true); //启动btn1为切换按钮
```



切换按钮就是你点击一下, 按钮是一种颜色, 再点击一下, 按钮又恢复到默认颜色。

TFT Simulator



默认颜色

TFT Simulator



第 1 次点击颜色

TFT Simulator



第 2 次点击恢复默认颜色

这种切换按钮可以用 get 函数来获取按钮状态, 后面会讲。

```
void lv_btn_set_state(lv_obj_t * btn, lv_btn_state_t state)
```

//设置按钮状态, 如按下状态, 松开状态这些

btn: 哪一个普通按钮需要设置状态, 那么就传入这个按钮的对象

state: 设置状态的参数 LV_BTN_STATE_REL

设置按钮被释放(正常状态生效)

LV_BTN_STATE_PR

设置按钮被按下(正常状态生效)

LV_BTN_STATE_TGL_REL

设置按钮被释放(切换状态下生效)

LV_BTN_STATE_TGL_PR

设置按钮被按下(切换状态下生效)

LV_BTN_STATE_INA

按钮设置为不激活, 无法点击

LV_BTN_STATE_INA 按钮无法被点击使用

```
lv_style_t cont_style;  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮  
lv_btn_set_state(btn1, LV_BTN_STATE_INA); //设置按钮状态 LV_BTN_STATE_INA 无法点击状态
```



你看，按钮无法点击。

LV_BTN_STATE_REL 普通按钮释放状态使用

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮  
lv_btn_set_state(btn1, LV_BTN_STATE_REL);  
//设置按钮状态 LV_BTN_STATE_REL 普通按钮释放状态
```



LV_BTN_STATE_PR 普通按钮被按下状态使用

```
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮  
lv_btn_set_state(btn1, LV_BTN_STATE_PR);  
//设置按钮状态 LV_BTN_STATE_PR 普通按钮按下状态
```



你看启动后按钮默认被设置成按下状态，但是你去点击按钮就会恢复释放状态。

LV_BTN_STATE_TGL_REL 切换按钮默认释放

```
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮  
lv_btn_set_toggle(btn1, true); //启动按钮为切换按钮  
lv_btn_set_state(btn1, LV_BTN_STATE_TGL_REL); //一定要使能切换按钮，设置切换状态按钮才生效  
//设置按钮状态 LV_BTN_STATE_TGL_REL 切换按钮默认为释放
```



你看切换按钮释放状态，底部有点翻白。

```
lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮  
lv_btn_set_toggle(btn1, true); //启动按钮为切换按钮  
lv_btn_set_state(btn1, LV_BTN_STATE_TGL_PR); //一定要使能切换按钮  
//设置按钮状态 LV_BTN_STATE_TGL_PR 切换按钮默认为按下
```



切换按钮按下状态底部完全是暗色的。

```

void lv_btn_set_ink_in_time(lv_obj_t * btn, uint16_t time) //设置按钮点击后波纹开始时长
btn: 哪一个普通按钮需要设置波纹, 那么就传入这个按钮的对象
time: 波纹开始到结束时间, 单位为 ms
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮
lv_btn_set_ink_in_time(btn1, 2000); //按钮点击波纹开始到结束为2秒

```



```

void lv_btn_set_ink_wait_time(lv_obj_t * btn, uint16_t time) //设置波纹等待维持时长
btn: 哪一个普通按钮需要设置波纹, 那么就传入这个按钮的对象
time: 波纹开始到结束时间, 单位为 ms

```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮
lv_btn_set_ink_in_time(btn1, 2000); //按钮点击波纹开始到结束为2秒
lv_btn_set_ink_wait_time(btn1, 5000); //波纹等待维持时长为5秒

```



在波纹要结束的时候, 波纹等待维持时长就发挥作用了, 感觉波纹消失得慢了许多。

```

void lv_btn_set_ink_out_time(lv_obj_t * btn, uint16_t time) //设置波纹出场动画时长
btn: 哪一个普通按钮需要设置波纹, 那么就传入这个按钮的对象
time: 波纹开始到结束时间, 单位为 ms

```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮
lv_btn_set_ink_in_time(btn1, 2000); //按钮点击波纹开始到结束为2秒
lv_btn_set_ink_wait_time(btn1, 5000); //波纹等待维持时长为5秒
lv_btn_set_ink_out_time(btn1, 2000); //波纹出场时间动画

```

也是调整波纹时间的, 自己测试。

```

/*Button (dependencies: lv_cont*)
#define LV_USE_BTN      1
#if LV_USE_BTN != 0
/*Enable button-state animations - d
#define LV_BTN_INK_EFFECT  1
#endif

```

必须在 lv_conf.h 将 # define LV_BTN_INK_EFFECT 1 使能, 不然按钮没有波纹效果。

如何设置 lv_btn 按钮字体显示

lv_btn 和 QT 或者 UCGUI 字体设置不一样，不是 btn 对象自带字体设置函数。lv 要求 btn 对象必须借助 label 控件才能设置字体。

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *btn1 = lv_btn_create(scr, NULL); //创建普通按钮
```

```
lv_obj_t *label_btn = lv_label_create(btn1, NULL); //把按钮对象放入标签  
lv_label_set_text(label_btn, "PUSHButton"); //用标签设置按钮字符
```

就是这种用 label 标签设置字符，按钮显示，这种偷梁换柱的感觉。



```
void lv_obj_set_drag(lv_obj_t * obj, bool en) //对象具备拖拽移动功能
```

obj: 传入被拖拽对象

en: true 拖拽功能使能, false 取消拖拽功能

我这里用按钮为例

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
btn1 = lv_btn_create(scr, NULL); //创建普通按钮，对象改成全局的了  
lv_obj_t *label_btn = lv_label_create(btn1, NULL); //把按钮对象放入标签  
lv_label_set_text(label_btn, "PUSHButton"); //用标签设置按钮字符  
lv_obj_set_drag(btn1, true); //按钮具备拖拽功能
```



鼠标按住按钮不放，拖拽按钮到右下角



设置按钮颜色样式(可以分为按下和松开两种样式)

```
hal_init();  
/*********************加入自己代码*****  
  
lv_style_t btn_style_release; // 按钮释放样式设置  
lv_style_t btn_style_press; // 按钮按下样式设置  
  
lv_obj_t *scr = lv_scr_act(); // 获取当前活跃的屏幕对象  
lv_obj_t * btn1 = lv_btn_create(scr, NULL); // 创建普通按钮  
  
/*这是释放的样式*/  
lv_style_copy(&btn_style_release, &lv_style_plain_color);  
//将系统默认样式复制给btn_style进行样式修改  
  
btn_style_release.body.main_color = LV_COLOR_MAKE(0x1e, 0x9f, 0xff); // 设置背景为纯色  
btn_style_release.body.grad_color = LV_COLOR_MAKE(0x1e, 0x9f, 0xfe);  
btn_style_release.body.opa = 80; // 透明度为80  
btn_style_release.body.radius = LV_RADIUS_CIRCLE; // 绘制圆角按钮  
btn_style_release.body.shadow.color = LV_COLOR_MAKE(0x1e, 0x9f, 0xff); // 设置背景为纯色  
btn_style_release.body.shadow.type = LV_SHADOW_FULL;  
btn_style_release.body.shadow.width = 5; // 阴影宽度  
btn_style_release.body.padding.left = 10; // 设置左内边距  
btn_style_release.body.padding.right = 10; // 设置右内边距  
  
/*这是按下的样式*/  
lv_style_copy(&btn_style_press, &lv_style_plain_color); // 将系统默认样式复制给btn_style进行样式修改  
  
btn_style_press.body.main_color = LV_COLOR_ORANGE; // 设置背景为橙色  
btn_style_press.body.grad_color = LV_COLOR_ORANGE; // 设置背景为橙色;  
btn_style_press.body.opa = 80; // 透明度为80  
btn_style_press.body.radius = LV_RADIUS_CIRCLE; // 绘制圆角按钮  
btn_style_press.body.shadow.color = LV_COLOR_ORANGE; // 设置背景为橙色  
btn_style_press.body.shadow.type = LV_SHADOW_FULL;  
btn_style_press.body.shadow.width = 5; // 阴影宽度  
btn_style_press.body.padding.left = 10; // 设置左内边距  
btn_style_press.body.padding.right = 10; // 设置右内边距  
  
/*将样式设置给按钮*/  
lv_btn_set_style(btn1, LV_BTN_STYLE_REL, &btn_style_release);  
lv_btn_set_style(btn1, LV_BTN_STYLE_PR, &btn_style_press);
```



松开的按钮颜色



按下之后的按钮颜色

代码示例

```
int main(int argc, char ** argv)
{
    (void) argc; /*Unused*/
    (void) argv; /*Unused*/
    /*Initialize LittlevGL*/
    lv_init();
    /*Initialize the HAL (display, input devices, tick) for LittlevGL*/
    hal_init();

    /*****加入自己代码*****/
    lv_style_t btn_style_release; // 按钮释放样式设置
    lv_style_t btn_style_press; // 按钮按下样式设置

    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
    lv_obj_t * btn1 = lv_btn_create(scr, NULL); //创建普通按钮
    /*这是释放的样式*/
    lv_style_copy(&btn_style_release, &lv_style_plain_color);
    //将系统默认样式复制给 btn_style 进行样式修改
    btn_style_release.body.main_color = LV_COLOR_MAKE(0x1e, 0x9f, 0xff); //设置背景为纯色
    btn_style_release.body.grad_color = LV_COLOR_MAKE(0x1e, 0x9f, 0xfe);
    btn_style_release.body.opa = 80; //透明度为 80
    btn_style_release.body.radius = LV_RADIUS_CIRCLE; //绘制圆角按钮
    btn_style_release.body.shadow.color = LV_COLOR_MAKE(0x1e, 0x9f, 0xff); //设置背景为纯色
    btn_style_release.body.shadow.type = LV_SHADOW_FULL;
    btn_style_release.body.shadow.width = 5; //阴影宽度
    btn_style_release.body.padding.left = 10; //设置左内边距
    btn_style_release.body.padding.right = 10; //设置右内边距
    /*这是按下的样式*/
    lv_style_copy(&btn_style_press, &lv_style_plain_color); //将系统默认样式复制给 btn_style 进行样式修改

    btn_style_press.body.main_color = LV_COLOR_ORANGE; //设置背景为橙色
    btn_style_press.body.grad_color = LV_COLOR_ORANGE; //设置背景为橙色;
    btn_style_press.body.opa = 80; //透明度为 80
    btn_style_press.body.radius = LV_RADIUS_CIRCLE; //绘制圆角按钮
    btn_style_press.body.shadow.color = LV_COLOR_ORANGE; //设置背景为橙色
    btn_style_press.body.shadow.type = LV_SHADOW_FULL;
    btn_style_press.body.shadow.width = 5; //阴影宽度
    btn_style_press.body.padding.left = 10; //设置左内边距
    btn_style_press.body.padding.right = 10; //设置右内边距

    /*将样式设置给按钮*/
    lv_btn_set_style(btn1, LV_BTN_STYLE_REL, &btn_style_release);
    lv_btn_set_style(btn1, LV_BTN_STYLE_PR, &btn_style_press);

    /*****后面是触摸按下硬件回调代码，这里就不用展示了*****/

    while(1) {
        /* Periodically call the lv_task handler.
         * It could be done in a timer interrupt or an OS task too.*/
        lv_task_handler();
    }
}
```

Events 事件(用按钮被按下处理事件做演示)

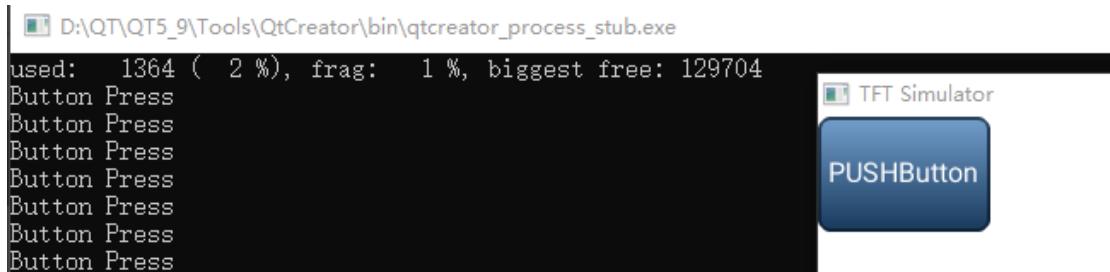
```
void lv_obj_set_event_cb(lv_obj_t * obj, lv_event_cb_t event_cb) //注册对象具备事件发生功能  
obj: 需要产生事件的对象放入 obj  
event_cb: 传入自定义的回调函数
```

```
typedef void (*lv_event_cb_t)(struct _lv_obj_t * obj, lv_event_t event) //回调函数定义要求  
obj: 什么对象被触发  
event: 触发的什么事件?
```

```
void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数  
{  
    printf("Button Press\n");  
}  
  
/*********************  
 * GLOBAL FUNCTIONS  
******/  
  
int main(int argc, char ** argv)  
{  
    (void) argc; /*Unused*/  
    (void) argv; /*Unused*/  
  
    /*Initialize LittlevGL*/  
    lv_init();  
  
    /*Initialize the HAL (display, input devices, tick) for LittlevGL*/  
    hal_init();  
    /*加入自己代码******/  
  
    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
    lv_obj_t *btn1 = lv_btn_create(scr, NULL); //创建普通按钮  
    lv_obj_t *label_btn = lv_label_create(btn1, NULL); //把按钮对象放入标签  
    lv_label_set_text(label_btn, "PUSHButton"); //用标签设置按钮字符  
  
    lv_obj_set_event_cb(btn1, Btn_Call_func); //给按钮增加事件回调函数  
    while(1) {  
        /* Periodically call the lv_task handler.  
         * It could be done in a timer interrupt or an OS task too. */  
        lv_task_handler();  
        usleep(5 * 1000);  
    }  
}
```

这是我自定义的回调函数

我将按钮注册成事件触发



你看，我按一下按钮回调函数执行多次，证明按钮按一下触发了很多其它事件，这样看似按钮判断成功了，其实有很多 BUG。

所以我必须在回调函数中再次判断是不是按钮真正按下了

LV_EVENT_PRESSED 按下事件使用

```
lv_obj_t *btn1;//按钮对象必须做成全局的  
  
void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数  
{  
    if(obj == btn1) //如果btn1 按钮发生事件触发  
    {  
        if(event == LV_EVENT_PRESSED) //判断回调事件为按下事件  
        {  
            printf("Button Press\n");  
        }  
    }  
}  
  
int main(int argc, char ** argv)  
{  
    (void) argc; /*Unused*/  
    (void) argv; /*Unused*/  
  
    /*Initialize LittlevGL*/  
    lv_init();  
  
    /*Initialize the HAL (display, input devices, tick) for LittlevGL*/  
    hal_init();  
    //*****加入自己代码*****  
  
    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
    btn1 = lv_btn_create(scr, NULL); //创建普通按钮，对象改成全局的了  
    lv_obj_t *label_btn = lv_label_create(btn1, NULL); //把按钮对象放入标签  
    lv_label_set_text(label_btn, "PushButton"); //用标签设置按钮字符  
  
    lv_obj_set_event_cb(btn1, Btn_Call_func); //给按钮增加事件回调函数  
  
    //*****  
  
    while(1) {
```

```
used: 1364 ( 2 %), frag: 1 %, biggest free: 129704  
Button Press  
used: 1364 ( 2 %), frag: 1 %, biggest free: 129704  
Button Press
```

你看，按钮按下 1 次，回调函数就打印一次。

我必须将按钮对象改成全局的，这样才能在回调函数中判断按钮是否按下了

判断是 btn1 按下后，再次判断 btn1 为什么事件？万一不是按下事件而是滑动事件呢？所以这里就是判断是不是按下事件

第 1 次判断是不是 btn1 按钮按下了，这是不是多此一举呢？其实不然，如果你有两个 btn1, btn2 对象，然后这两个按钮都是注册的同一个 Btn_Call_func 回调函数，那么你就搞不清楚是哪个按钮按下

按钮对象改成全局

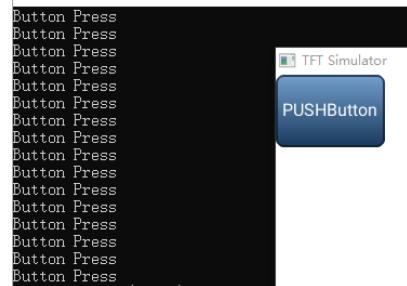
```

LV_EVENT_PRESSING 判断按钮按下事件为按下不放
lv_obj_t *btn1; //按钮对象必须做成全局的

void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_PRESSING) //判断回调事件为按下不放事件
        {
            printf("Button Press\n");
        }
    }
}

```

后面代码省略.....



只要按钮按着不放就会一直执行回调，然后执行 if(event==LV_EVENT_PRESSING)里面的函数

LV_EVENT_PRESS_LOST 按住按钮不放鼠标拖动到按钮之外被触发

```

void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_PRESS_LOST) //判断回调事件为按下不放，然后按住按钮鼠标拖动到按钮之外被触发
        {
            printf("Button Press\n");
        }
    }
}

```



LV_EVENT_SHORT_CLICKED 短按，按下按钮后必须在规定短时间内松开，事件被触发。如果过了很久再松开事件不会被触发

```

void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_SHORT_CLICKED) //判断回调事件为短按
        {
            printf("Button Press\n");
        }
    }
}

```

`LV_EVENT_LONG_PRESSED` 长按 按下去不会马上触发，要按下超过一段时间才会被触发，而且只触发一次，如果你想再次触发，就必须松开重新再按

```
void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_LONG_PRESSED) //判断回调事件为 长按
        {
            printf("Button Press\n");
        }
    }
}
```

`LV_EVENT_LONG_PRESSED_REPEAT` 长按重复触发 按着鼠标不放，会不停的反复触发

```
void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_LONG_PRESSED_REPEAT) //判断回调事件为 长按重复触发
        {
            printf("Button Press\n");
        }
    }
}
```

`LV_EVENT_DRAG_BEGIN` 开始拖拽对象时触发

`LV_EVENT_DRAG_END` 拖拽对象结束时触发

`lv_obj_set_drag(btn1,true);`//一定要打开对象拖拽功能

```
void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_DRAG_BEGIN) //判断回调事件为 开始拖拽按钮时触发
        {
            printf("Button DRAG_BEGIN\n");
        }
        else if(event == LV_EVENT_DRAG_END)//判断回调事件为 拖拽按钮结束时触发
        {
            printf("Button DRAG_END\n");
        }
        else
        {
        }
    }
}
```



你看拖拽开始和结束都有效果。

littleVGL 移植触摸屏驱动

触摸屏和 LCD 实现触摸屏画板基础请查阅我的《STMPeripheral_oeration...》文档
在文档 STM32 触摸屏操作....28 章节

确认触摸屏驱动在 LCD 屏幕上能画画

```
/*
* 读取 XPT2046 的 X 通道和 Y 通道的 AD 值 ( 12 bit, 最大是 4096)
* X_Ad : 存放 X 通道 AD 值的地址
* Y_Ad : 存放 Y 通道 AD 值的地址
*/
void XPT2046_ReadAdc_XY ( uint16_t * X_Ad, uint16_t * Y_Ad ) ←
{
    uint16_t tempx = 0, tempy = 0;

    GPIO_ResetBits(GPIOC, GPIO_Pin_13); //CS=0 拉低片选
    delay_us(1);
    tempx = XPT2046_ReadAdc ( 0xd0 );//ReadAdc函数写了 0xd0 是x通道
    delay_us(1);
    tempy = XPT2046_ReadAdc ( 0x90 );//ReadAdc函数写了 0x90 是y通道
    GPIO_SetBits(GPIOC, GPIO_Pin_13); //CS=1 拉高片选, 如果你一直读触摸屏数据, 可以不用拉高

    *X_Ad = tempx;
    *Y_Ad = tempy;
}
```

读取触摸屏 X 和 Y 的坐标值

```
/*对获取的 X_Ad 和 Y_Ad 进行算法滤波得到真正的 XY 值
*x:真正滤波后的 x 值返回
*y:真正滤波后的 y 值返回
*/
void XPT2046_ReadAdc_Smooth_XY(uint16_t *X, uint16_t *Y)
{
    uint16_t sAD_X = 0, sAD_Y = 0;
    uint16_t xbuffer[10] = {0};
    uint16_t ybuffer[10] = {0};
    uint8_t count = 0;
    int i = 0;
    uint16_t Xmax = 0, Xmin = 0, Ymax = 0, Ymin = 0;
    while((GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_1) == 0) && count < 10) //如果触摸屏被按着, PEN 引脚输出低电平
    {
        XPT2046_ReadAdc_XY(&sAD_X, &sAD_Y); //采集 ADC X 和 Y 值 ←
        xbuffer[count] = sAD_X;
        ybuffer[count] = sAD_Y;
        count++;
    }
    if(count > 9)
    {
        Xmax = Xmin = xbuffer[0];
        Ymax = Ymin = ybuffer[0];
        for(i = 1; i < 10; i++) //去除 x 的最大值和最小值然后给 xbuffer
        {
            if(xbuffer[i] < Xmin)
                Xmin = xbuffer[i];
            else
                Xmax = xbuffer[i];
        }
        for(i = 1; i < 10; i++) //去除 y 的最大值和最小值然后给 ybuffer
        {
            if(ybuffer[i] < Ymin)
                Ymin = ybuffer[i];
            else
                Ymax = ybuffer[i];
        }
        /*计算平均值滤波*/
        *X = xbuffer[0];
        *Y = ybuffer[0];
        for(i = 0; i < 10; i++)
        {
            *X = *X + xbuffer[i];
            *Y = *Y + ybuffer[i];
        }
        *X = (int)(*X/10);
        *Y = (int)(*Y/10);
    }
}
```

```

/*全局变量存放触摸屏坐标校准值*/
static float Xfac = 0,Yfac = 0,Xoffset = 0, Yoffset = 0,Xfacbase = 0, Yfacbase = 0;

/*触摸屏初始化，触摸屏校准*/
void touch_init()
{
    uint16_t Xvalue = 0, Yvalue = 0;
    int tx1 = 0 , ty1 = 0;
    int tx2 = 0 , ty2 = 0;
    int tx3 = 0 , ty3 = 0;
    int tx4 = 0 , ty4 = 0;
    int centreX = 0, centreY = 0;
    float Xfacbase = 0 , Yfacbase = 0;
    int X = 0, Y = 0;
    int time = 0;
    AnalogSpi_init(); //模拟 SPI IO 初始化
    LCD_Clear(WHITE);
    LCD_Fill(20,20,30,RED); //LCD 在屏幕左上角画点
    delay_ms(2000);
    for(time = 0;time<5;time++) //获取左上角触摸值
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue ); //读取触摸屏值
        delay_ms(1000);
        printf("x1 = %d y1 = %d\r\n",Xvalue,Yvalue);
        if(Xvalue > 0)
        {
            tx1 = Xvalue;
            ty1 = Yvalue;
        }
    }
    Xvalue = 0;
    Yvalue = 0;
    LCD_Fill(200,20,210,30,RED); //LCD 在屏幕右上角画点
    delay_ms(2000);
    for(time = 0;time<5;time++) //获取右上角触摸值
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue ); //读取触摸屏值
        delay_ms(1000);
        printf("x2 = %d y2 = %d\r\n",Xvalue,Yvalue);
        if(Xvalue > 0)
        {
            tx2 = Xvalue;
            ty2 = Yvalue;
        }
    }
    Xvalue = 0;
    Yvalue = 0;
    LCD_Fill(20,300,30,310,RED); //LCD 在屏幕左下角画点
    delay_ms(2000);
    for(time = 0;time<5;time++) //获取左下角触摸值
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue ); //读取触摸屏值
        delay_ms(1000);
        printf("x3 = %d y3 = %d\r\n",Xvalue,Yvalue);

        if(Xvalue > 0)
        {
            tx3 = Xvalue;
            ty3 = Yvalue;
        }
    }
    Xvalue = 0;
    Yvalue = 0;
    LCD_Fill(200,300,210,310,RED); //LCD 在屏幕右下角画点
    delay_ms(2000);
    for(time = 0;time<5;time++) //获取右下角触摸值
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue ); //读取触摸屏值
        delay_ms(1000);
        printf("x4 = %d y4 = %d\r\n",Xvalue,Yvalue);
        if(Xvalue > 0)
        {
            tx4 = Xvalue;
            ty4 = Yvalue;
        }
    }
    Xvalue = 0;
    Yvalue = 0;
    Xfacbase = (float)((200-20)+(200-20))/2.0; //校准系数算法结果存放全局变量
    Xfac = Xfacbase/(float)((tx2-tx1) + (tx4-tx3))/2.0; //伸缩系数算法结果存放全局变量
    Yfacbase = (float)((300-20)+(300-20))/2.0; //校准系数算法结果存放全局变量
    Yfac = Yfacbase/(float)((ty3-ty1)+(ty4-ty2))/2.0; //伸缩系数算法结果存放全局变量
    printf("Xfac = %f Yfac = %f \r\n",Xfac,Yfac);
    Xoffset = (float)((240-Xfac*(tx2+tx1))/2); //偏移量存放全局变量
    Yoffset = (float)((320-Yfac*(ty3+ty1))/2); //偏移量存放全局变量
}

```

```

/*这就是轮询一直读取触摸屏值，然后加入算法进行实时定位*/
void read_time_touch(uint16_t *X, uint16_t *Y)
{
    uint16_t Xvalue = 0, Yvalue = 0;

    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );//读取触摸屏值
    if(Xvalue > 0)
    {
        *X = 0;
        *Y = 0;
    }
    *X = Xfac*Xvalue+Xoffset; //这就是已经校准出的算法系数，用来计算实际 x 位置
    *Y = Yfac*Yvalue+Yoffset; //这就是已经校准出的算法系数，用来计算实际 y 位置
}

int main(void)
{
    uint16_t X = 0, Y = 0;

    RCC_Configuration(); //初始化时钟
    delay_init(); //延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置NVIC中
    uart_init(115200); //串口1初始化

    LED_Init(); //LED 测试板子是否启动，可以不要

    LCD_Init(); //初始化LCD
    LCD_DisplayOn(); //开启LCD显示

    touch_init(); //初始化SPI 触摸芯片，校准触摸屏，得到校准系数

    while(1)
    {
        read_time_touch(&X, &Y); //用校准系数实时定位当前触摸位置
        LCD_Fill(X, Y, X+10, Y+10, RED); //画出当前触摸点
    }
}

```

以上触摸画板验证成功后，进行 lvgl 触摸屏移植。



lv_port_disp_template.c/h 改为 lv_port_indev.c/h

```
#include "lv_port_indev.h"
```

```
#if 1
/*
 * INCLUDES
 */
#include "lv_port_indev.h"

/*
 * DEFINES
 */
#define LV_INDEV_TYPEDEFS
#define LV_INDEV_STATIC_PROTOTYPES

static void touchpad_init(void);
static bool touchpad_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
static bool touchpad_is_pressed(void);
static void touchpad_get_xy(lv_coord_t * x, lv_coord_t * y);
```

记住头文件这里要置 1

将头文件也改成这样

```
/copy this file as lv_port_indev.h and
#ifndef LV_PORT_INDEV_H
#define LV_PORT_INDEV_H

#ifdef __cplusplus
extern "C" {
#endif

/*
 * INCLUDES
 */
#include "lvgl.h"
void lv_port_indev_init(void);

#ifdef __cplusplus
} /* extern "C" */
#endif
```

C 文件把“lvgl/lvgl”路径取消掉就剩 lvgl

core
core_cm3.c
firmware
conf
stm32f10x_itc
system_stm32f10x.c
SYSTEM
LCD
lcd.c
vgl
vgl_tim3
vgl_drv
lv_port_disp.c
zxx_touch
xpt2046.c
zxx_touch.c

```
26 static void touchpad_init(void);
27 static bool touchpad_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data)
28 static bool touchpad_is_pressed(void);
29 static void touchpad_get_xy(lv_coord_t * x, lv_coord_t * y);
30
31 static void mouse_init(void);
32 static bool mouse_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
33 static bool mouse_is_pressed(void);
34 static void mouse_get_xy(lv_coord_t * x, lv_coord_t * y);
35
36 static void keypad_init(void);
37 static bool keypad_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
38 static uint32_t keypad_get_key(void);
39
40 static void encoder_init(void);
41 static bool encoder_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
42 static void encoder_handler(void);
43
44 static void button_init(void);
45 static bool button_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
46 static int8_t button_get_pressed_id(void);
47 static bool button_is_pressed(uint8_t id);
```

看起代码这么多，主要修改这四个函数

```
/*Initialize your touchpad*/
static void touchpad_init(void)
{
    /*Your code comes here*/
    touch_init(); // 初始化SPI 触摸芯片，校准触摸屏，得到校准系数
}
```

触摸屏初始化和校准代码都放入 touchpad_init 函数进行执行

```
/* Will be called by the library to read the touchpad */
static bool touchpad_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data)
{
    static lv_coord_t last_x = 0;
    static lv_coord_t last_y = 0;

    /*Save the pressed coordinates and the state*/
    if(touchpad_is_pressed()) {
        touchpad_get_xy(&last_x, &last_y);
        data->state = LV_INDEV_STATE_PR;
    } else {
        data->state = LV_INDEV_STATE_REL;
    }

    /*Set the last pressed coordinates*/
    data->point.x = last_x;
    data->point.y = last_y;

    /*Return `false` because we are not buffering and no more data to read*/
    return false;
}
```

touchpad_read 代码不用动，只需要理解就是。

发现触摸按下执行
get_xy 获取 xy 坐标

触摸松开不获取坐标

```

/*Return true is the touchpad is pressed*/
static bool touchpad_is_pressed(void)
{
    uint16_t Xvalue = 0, Yvalue = 0;

    /*Your code comes here*/
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    if(Xvalue > 0)
    {
        return true;
    }
    return false;
}

```

touchpad_is_pressed 函数需要添加以上内容， touchpad_is_pressed 是被 touchpad_read 调用

```

/*Get the x and y coordinates if the touchpad is pressed*/
static void touchpad_get_xy(lv_coord_t * x, lv_coord_t * y)
{
    /*Your code comes here*/

    (*x) = 0;
    (*y) = 0;
    read_time_touch(x, y); //用校准系数实时定位当前触摸位置, 主要形参数据类型有点出入自己修改
}

```

一旦触摸被按下，就会执行 touchpad_get_xy 函数， touchpad_get_xy 也是被 touchpad_read 调用

以上就完成触摸屏在 lvgl 下的移植了。

我们还是来实现按钮拖拽功能

```

lv_obj_t *btn1;//按钮对象必须做成全局的

void Btn_Call_func(struct _lv_obj_t * obj, lv_event_t event) //这就是事件回调函数
{
    if(obj == btn1) //如果btn1 按钮发生事件触发
    {
        if(event == LV_EVENT_DRAG_BEGIN) //判断回调事件为 开始拖拽按钮时触发
        {
            printf("Button DRAG_BEGIN\r\n");
        }
        else if(event == LV_EVENT_DRAG_END)//判断回调事件为 拖拽按钮结束时触发
        {
            printf("Button DRAG_END\r\n");
        }
        else
        {
        }
    }
}

int main(void)
{
    uint16_t X = 0, Y = 0;

    RCC_Configuration(); //初始化时钟
    delay_init(); //延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置NVIC中断分组2:2位抢
    uart_init(115200); //串口1初始化
    Lvgl_TIM3_Init(999, 71); //Lvgl TIM3 初始化 1ms中断，一定要提前执行
    LED_INIT(); //LED 测试板子是否启动，可以不要

    lv_init(); //lv初始化
    lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后
    lv_port_indev_init(); //lv触摸接口初始化一定要放在lv_init()函数之后

    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
    btn1 = lv_btn_create(scr, NULL); //创建普通按钮，对象改成全局的了
    lv_obj_set_drag(btn1, true); //按钮具备拖拽功能
    lv_obj_set_event_cb(btn1, Btn_Call_func); //给按钮增加事件回调函数

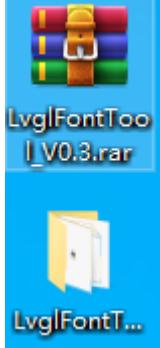
    while(1)
    {
        lv_task_handler(); //这句执行后有界面效果
        //LCD_Fill(X, Y, X+10, Y+10, RED); //画出当前触摸点
    }
}

```

4 点校准后，执行按钮可以正常拖拽

LittleVGL 支持中文汉字显示

实现预先指定中文内容显示，在 PC 模拟器上面实验

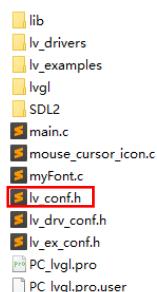


awesome
platforms
config.ini
LvgiFontTool.exe
libgcc_s_dw2-1.dll
libstdc++-6.dll
libwinpthread-1.dll
Qt5Core.dll
Qt5Gui.dll
Qt5Widgets.dll

下载开源玩家做的，在 Lvgi 下各种文字生成字模的软件 使用

myFont.c 我写的 myFont.c 不知道大小写有没有影响。C 文件生成。将 C 文件拷贝进工程根目录下

我的电脑 > 本地磁盘 (D:) > PC_simulator_lvgl >



在 QT Creator 添加现有文件中加入进来

```

LV_FONT_DECLARE(myfont); //导入你生成字库的C文件
/*****************
 * GLOBAL FUNCTIONS
 *****************/
int main(int argc, char ** argv)
{
    (void) argc; /*Unused*/
    (void) argv; /*Unused*/
}

```

`#include "../lvgl.h"` 生成的 myfont.c 包含错误的路径，直接改成 PC 模拟器版本 "lvgl/lvgl.h" 就可以了

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

static lv_style_t mystyle;
lv_style_copy(&mystyle, &lv_style_plain_color); //将系统颜色拷贝过来修改
mystyle.text.color = LV_COLOR_RED; //中文显示为红色
mystyle.text.font = &myfont; /* 添加自定义字体 */

lv_obj_t *label1 = lv_label_create(scr, NULL); //把窗口对象放入标签
lv_label_set_style(label1, LV_LABEL_STYLE_MAIN, &mystyle); /* 设置样式 */
lv_label_set_text(label1, "我是向仔州"); //直接写要显示得中文，lvgl自动去字库搜索

```

直接写入中文，
模拟器就能显示

```

lv_obj_t *label2 = lv_label_create(scr, NULL); //把窗口对象放入标签
lv_obj_set_pos(label2, 100, 100); //你会发现新建立的label对象主要没加入你字库的style，就使用系统默认的style
lv_label_set_text(label2, "abcdef"); //用标签设置字符

```



但是这种方式，抛开 X86 架构的 PC 模拟器，在 STM32 上运行就出现新问题了。

实现预先指定中文内容显示，在 STM32 上面实验



LV_FONT_DECLARE(myFont); //导入C文件字库

```
lv_init(); //lv初始化
lv_port_disp_init(); //lv接口初始化函数一定要放在lv_init()函数之后
// lv_port_indev_init(); //lv触摸接口初始化一定要放在lv_init()函数之后

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

static lv_style_t mystyle;
lv_style_copy(&mystyle, &lv_style_plain_color); //将系统颜色拷贝过来修改
mystyle.text.color = LV_COLOR_RED; //中文显示为红色
mystyle.text.font = &myFont; /*添加自定义字体*/

lv_obj_t *label1 = lv_label_create(scr, NULL); //把窗口对象放入标签
lv_label_set_style(label1, LV_LABEL_STYLE_MAIN, &mystyle); /*设置样式*/
lv_label_set_text(label1, "我是向仔州"); //直接写要显示得中文，lvgl自动去字库搜索

lv_obj_t *label2 = lv_label_create(scr, NULL); //把窗口对象放入标签
lv_obj_set_pos(label2, 100, 100); //你会发现新建立的label对象主要没加入你字库的style，就使用系统默认的style
lv_label_set_text(label2, "abcdef"); //用标签设置字符

while(1)
```

```
myFont.c(152): error: #144: a value of type "const uint16_t *" cannot be used to initialize an entity of type "uint16_t *"
    .unicode_list = unicode_list_1,
myFont.c: 0 warnings, 1 error
".\out\STM32F1LCD.axf" - 1 Error(s), 2 Warning(s).
```

直接编译报错

在 myFont.c 文件里面有这一段

```
static const lv_font_fmt_txt_cmap_t cmaps[] = {
{
    .range_start = 20180,
    .range_length = 5,
    .type = LV_FONT_FMT_TXT_CMAP_SPARSE_TINY,
    .glyph_id_start = 0,
    .unicode_list = unicode_list_1,
    .glyph_id_ofs_list = NULL,
    .list_length = 5,
}
};
```

unicode_list lvgl 库里面定义是不带 const

```
static const uint16_t unicode_list_1[] = {
0xb211, /*(我)*/
0x662f, /*(是)*/
0x5411, /*(向)*/
0x4ed4, /*(仔)*/
0x5dde, /*(州)*/
0x0000, /*End indicator*/
};
```

但是 myFont.c 生成的时候 GBK 区位码是带 const 的

```

    uint16_t * unicode_list;

    /** if(type == LV_FONT_FMT_TXT_CMAP_FORMATO_...
     * if(type == LV_FONT_FMT_TXT_CMAP_SPARSE_...
     */
    const void * glyph_id_ofs_list;

    /** Length of `unicode_list` and/or `glyph_id
    uint16_t list_length;

    /** Type of this character map*/
    lv_font_fmt_txt_cmap_type_t type :2;
}lv_font_fmt_txt_cmap_t;

```

X86 PC 模拟器

```

    uint16_t * unicode_list;
    /** if(type == LV_FONT_FMT_TXT_CMAP_FORMATO_...) it's `uint8_t *`
     * if(type == LV_FONT_FMT_TXT_CMAP_SPARSE_...) it's `uint16_t *`
     */
    const void * glyph_id_ofs_list;
    /** Length of `unicode_list` and/or `glyph_id_ofs_list`*/
    uint16_t list_length;
    /** Type of this character map*/
    lv_font_fmt_txt_cmap_type_t type :2;
}lv_font_fmt_txt_cmap_t;

```

STM32 Keil4 IDE

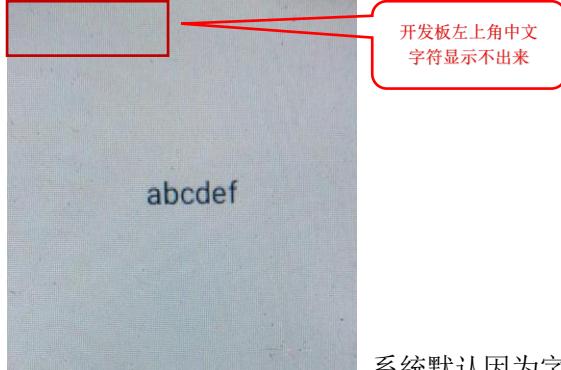
你看两边 lvgl 库代码一模一样，居然出现了 Keil4 编译不过的情况，那么只能说 X86 PC 模拟器做了优化，或者是 QT Creator 做了优化。

```

static uint16_t unicode_list_1[] = {
    0x6211, /*(我)*/
    0x662f, /*(是)*/
    0x5411, /*(向)*/
    0x4ed4, /*(仔)*/
    0x5dde, /*(州)*/
    0x0000, /*End indicator*/
};

```

我直接取消 const 编译通过下载程序进开发板看看。



系统默认因为字符显示出来了，中文字符居然显示不出来。

我是按照 PC 模拟器一模一样复制过来的，这个问题出在哪里呢？Keil 编码方式不对？还是什么的？

经过我一天的测试，发现确实是 IDE 编码的问题 尤其是 Keil

1.你再 GB2312 或者 ANSI 模式下写中文是绝对不行的

```

lv_obj_t *label1 = lv_label_create(scr, NULL);
// lv_obj_align(label1, NULL, LV_ALIGN_CENTER,
    lv_label_set_style(label1, LV_LABEL_STYLE_MAIN);
    lv_label_set_text(label1, "我是向仔州"); //直接

```



直接无法显示

2.你可能会想，什么都不改，我就把 IDE 编码方式改成 UTF-8(unicode) 在编译下裁

```

lv_label_set_text(label1, "xECExD2xCAlxC7xCFlxF2xD7xD0xD6xD");

```

但是还是无法显示，因为你看到没有，你虽然编码方式改成了 UTF-8，但是你以前写的中文 xCExD2xCExC7xCf.....还是用的 GBK(GB2312)的编码方式。

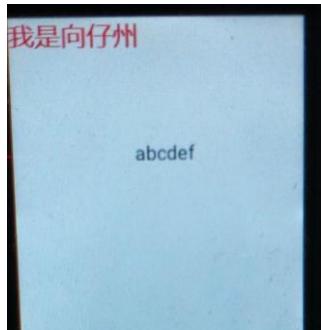
3.所以要在 UTF-8 模式下再次写回中文字符才行

```

lv_label_set_text(label1, "我是向仔州"); //xB0xD3uCxCFlxD4'xC5xC3xD6xD0xC8gxCAC1vg1xD79xAEzxD7xD

```

你看我在 UFT-8 模式下再次写回中文就可以了



你看，显示成功。

4.这里有个弊端，如果我的 IDE 又从 UTF-8 模式改回 GB2312 会是什么结果呢？GB2312 中文注释好看些。

lv_label_set_text(label1, "我是向仔州"); //直接写要显示得中文，1vg1自动去字库搜索
我修改回 GB2312 了，但是“我是向仔州”这几个字还是在 UTF-8 下面写的，没有动，应该问题不大。显示正常。

5.那么我重新启动 IDE 呢？

lv_label_set_text(label1, "鎚戛櫟錫或粃宸?"); //直接写要显示得中文，1vg1自动去字库搜索
看到没，我重启 IDE 就出现乱码，而且一不小心就会出现变异不过。

在这种情况下，最好将 UTF-8 编写的字符放在一个变量里面。

```
int main(void)
{
    uint16_t X = 0, Y = 0;
    char *str = "我是向仔州"; //将中文字符写在变量里面

    RCC_configuration(); //直接写要显示得中文，1vg1自动去字库搜索
    delay_init(); //用变量去显示中文

    lv_label_set_text(label1, str); //直接写要显示得中文，1vg1自动去字库搜索
} //直接写要显示得中文，1vg1自动去字库搜索
```

显示中文正常

我再次将 IDE 修改回 GB2312，重启 IDE

```
int main(void)
{
    uint16_t X = 0, Y = 0;
    char *str = "鎚戛櫟錫或粃宸?"; //我重启 IDE 后虽然会显示 UTF-8 中文的乱码
```

lv_label_set_text(label1, str); //直接写要显示得中文，1vg1自动去字库搜索
但是我后面代码结构看起来就不那么花眼睛。
所以你 char *str 后面加上注释中文也可以解决这个问题。

像这种中文显示，最好的办法是在 keil 工程开发之前就全部用 UTF-8 的方式去写代码。
还有一种就是在 GB2312 模式下先用英文写 label 字符表示。等你的代码调试没有问题了，
把英文字符全部在 UTF-8 模式下改成中文，也是可以的。

用中文显示方式在控件上显示字体，按钮为例(前提你已经按照要求解决了 UTF-8 编码问题)

```
LV_FONT_DECLARE(myFont); //导入C文件字库

int main(void)
{
    uint16_t X = 0, Y = 0;
    char *str = "錦戛櫟鍚或粃宸?; //这里面就是(我是向仔州字体) GB2312 模式要注释 UTF8 的字体
    lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

    lv_obj_t *btn = lv_btn_create(scr, NULL);
    lv_obj_set_size(btn, 110, 100);
    lv_obj_set_pos(btn, 50, 50);

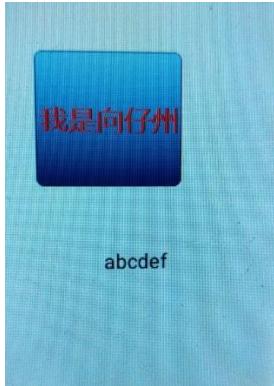
    static lv_style_t mystyle;
    lv_style_copy(&mystyle, &lv_style_plain_color); //将系统颜色拷贝过来修改
    mystyle.text.color = LV_COLOR_RED; //中文显示为红色
    mystyle.text.font = &myFont; /* 添加自定义字体 */

    lv_obj_t *label1 = lv_label_create(btn, NULL); //把按钮对象放入标签
    lv_label_set_style(label1, LV_LABEL_STYLE_MAIN, &mystyle); /* 设置样式 */
    lv_label_set_text(label1, str); //直接写要显示得中文，lvgl自动去字库搜索

    lv_obj_t *label2 = lv_label_create(scr, NULL); //把窗口对象放入标签
    lv_obj_set_pos(label2, 100, 200); //你会发现新建立的label对象主要没加入你字库的style，就使用
    lv_label_set_text(label2, "abcdef"); //用标签设置字符

    while(1)
    {
        lv_task_handler(); //这句执行后有界面效果
    }
}
```

创建按钮，在按钮内部显示中文字符。



你看 按钮显示字符成功。

让程序自动修改字体

```
int main(void)
{
    uint16_t X = 0, Y = 0;
    char *str1 = "錦戛櫟鍚或粃宸?; //【向仔】
    char *str2 = "錦戛櫟"; //我是

    static lv_style_t mystyle;
    lv_style_copy(&mystyle, &lv_style_plain_color); //将系统颜色拷贝过来修改
    mystyle.text.color = LV_COLOR_RED; //中文显示为红色
    mystyle.text.font = &myFont; /* 添加自定义字体 */

    lv_obj_t *label1 = lv_label_create(btn, NULL); //把按钮对象放入标签
    lv_label_set_style(label1, LV_LABEL_STYLE_MAIN, &mystyle); /* 设置样式 */
    lv_label_set_text(label1, str1); //直接写要显示得中文，lvgl自动去字库搜索

    lv_obj_t *label2 = lv_label_create(scr, NULL); //把窗口对象放入标签
    lv_obj_set_pos(label2, 100, 200); //你会发现新建立的label对象主要没加入你字库的
    lv_label_set_text(label2, "abcdef"); //用标签设置字符

    while(1)
    {
        lv_label_set_text(label1, str2); //直接写要显示得中文，lvgl自动去字库搜索，向仔
        delay_ms(1000);
        lv_task_handler(); //这句执行后界面才有中文更新效果

        lv_label_set_text(label1, str1); //直接写要显示得中文，lvgl自动去字库搜索，我是
        delay_ms(1000);
        lv_task_handler(); //这句执行后界面才有修改中文更新效果

        //LCD_Fill(X, Y, X+10, Y+10, RED); //画出当前触摸点
    }
}
```

我们直接在死循环中修改按钮字体内容



这就是修改按钮字体内容的方法，你可以把这种方法用在按钮按下后触发字体修改应该中。

手动发送 Event 事件触发方法

给固定控件增加新的事件发送功能

```
lv_res_t lv_event_send(lv_obj_t * obj, lv_event_t event, const void * data) //固定控件自定义发送事件函数
```

obj: 需要事件发送的对象

event: 事件编号, 0~19 都是控件的事件编号, 19~255 事件编号可以自己写, 我建议从 30 开始编写自定义事件编号

const void * data: 可以向事件回调函数传你自己定义的数据

lv_res_t: 返回 LV_RES_OK 表示你事件发送对象还在

返回 LV_RES_INV 表示你事件发送对象不在了, 被删除了

```
const void * lv_event_get_data(void) //在事件回调函数中执行接收发送事件函数自定义的数据
```

const void *: 在事件函数中, 自定义指针, 接收事件自定义数据。

```
typedef struct{
    char name[20];
    int age;
}USER_DATA; //自定义数据结构
```

```
lv_obj_t * btn1;
```

```
static void btn_event_cb(lv_obj_t *obj, lv_event_t event)
```

```
{
```

4.event == 29,28 都不行, 必须等于 30 才能为真, 因为这是你发送事件函数自定义的事件编号

```
if(event == 30) //30 编号 就是 我lv_event_send 自定义的event事件号
```

```
{
```

```
    USER_DATA *getdata = (USER_DATA *)lv_event_get_data();
    printf("getdata name = %s\n", getdata->name);
    printf("getdata age = %d\n", getdata->age);
```

```
}
```

```
else
```

```
{
```

```
}
```

```
}
```

```
USER_DATA user_data = {"xxxxxxxx", 100}; //初始化 自定义结构体数据
```

```
*****加入自己代码*****
```

```
lv_res_t ret;
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
btn1 = lv_btn_create(scr, NULL); //创建按钮
```

```
lv_obj_set_event_cb(btn1, btn_event_cb); //给按钮设置事件接收函数
```

```
ret = lv_event_send(btn1, 30, &user_data); //设置事件发送, 事件接收函数必须是按钮定义的事件函数
```

```
if(ret == LV_RES_OK)
```

```
{
```

```
    printf("Object exist\n");
```

2.系统启动后, 会自动执行一次按钮的回调函数, 因为我这里触发的, 而且自定义数据 user_data 也发过去了

```
getdata name = xxxxxxxx  
getdata age = 100  
Object exist
```

TFT Simulator

数据在事件回调函数接收成功。

没有控件，自己定义事件发送功能

```
lv_res_t lv_event_send_func(lv_event_cb_t event_xcb, lv_obj_t * obj, lv_event_t event, const void  
* data) // 自定义事件发送回调函数  
event_xcb:    自定义回调函数 void (*lv_event_cb_t)( lv_obj_t * obj, lv_event_t event);  
obj:          需要事件发送的对象，不一定是控件，当前屏幕也可以  
event:         事件编号，0~19 都是控件的事件编号，19~255 事件编号可以自己写，我建议从 30 开始  
编写自定义事件编号  
  
const void * data: 可以向事件回调函数传你自己定义的数据  
lv_res_t:       返回 LV_RES_OK 表示你事件发送对象还在  
                返回 LV_RES_INV 表示你事件发送对象不在了，被删除了
```

```
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(event == 28) //这次事件号自定义为28  
    {  
        printf("xzz_event_cb \n");  
    }  
    else  
    {  
  
    }  
}  
*****加入自己代码*****  
lv_res_t ret;  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
ret = lv_event_send_func(xzz_event_cb, scr, 28, NULL); //自定义回调函数事件发送  
if(ret == LV_RES_OK)  
{  
    printf("Object exist\n");  
}  
else  
{  
    printf("Object delete\n");  
}
```

```
D:\QT\QT5_9\Tools\QtCreator\bin\qtcreator_process_stub.exe  
xzz_event_cb  
Object exist  
used: 1084 ( 1 %), frag: 0 %, biggest free: 129988
```

TFT Simulator

LED 灯软件界面显示

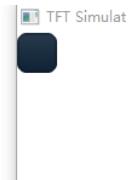
```
lv_obj_t * lv_led_create(lv_obj_t * par, const lv_obj_t * copy) //创建 LED 对象  
par: 传入 LED 嵌入的父对象  
copy: 平时填 NULL  
void lv_led_set_bright(lv_obj_t * led, uint8_t bright) //设置 LED 亮度  
led: 传入 LED 对象  
bright: 亮度设置, 最小 100, 最大 255  
void lv_led_on(lv_obj_t * led) //点亮 LED
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *LED = lv_led_create(scr, NULL);  
lv_led_set_bright(LED, 255); //设置LED亮度, 亮度范围100~255  
lv_led_on(LED); //点亮LED
```



你看 LED 点亮了就是这副颜色，不是很好看

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *LED = lv_led_create(scr, NULL);  
lv_led_set_bright(LED, 255); //设置LED亮度, 亮度范围100~255  
lv_led_off(LED); //关闭LED
```



关闭 LED 也不过是黑色显示，不是很好看嘛。

LED 样式设置，让 LED 看起好看

```
static lv_style_t led_style;  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *LED = lv_led_create(scr, NULL);  
lv_style_copy(&led_style, &lv_style_pretty_color); //样式拷贝  
  
led_style.body.radius = LV_RADIUS_CIRCLE; //绘制圆角  
led_style.body.main_color = LV_COLOR_RED; //主背景上半部分 红色  
led_style.body.grad_color = LV_COLOR_RED; //主背景下半部分 红色  
led_style.body.border.color = LV_COLOR_GREEN; //边框颜色为绿色  
led_style.body.border.width = 5; //边框宽度  
led_style.body.shadow.color = LV_COLOR_BLUE; //阴影为蓝色  
led_style.body.shadow.width = 5; //阴影大小  
  
lv_obj_set_style(LED, &led_style); //设置LED样式  
lv_led_set_bright(LED, 255); //设置LED亮度, 亮度范围100~255  
lv_led_on(LED); //打开LED
```

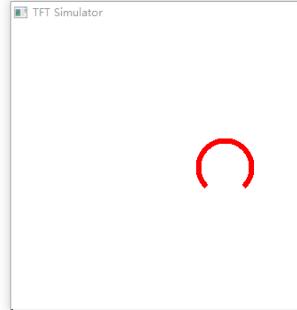


```
static lv_style_t led_style;  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *LED = lv_led_create(scr, NULL);  
lv_style_copy(&led_style, &lv_style_pretty_color); //样式拷贝  
  
led_style.body.radius = LV_RADIUS_CIRCLE; //绘制圆角  
led_style.body.main_color = LV_COLOR_RED; //主背景上半部分 红色  
led_style.body.grad_color = LV_COLOR_RED; //主背景下半部分 红色  
led_style.body.border.color = LV_COLOR_GREEN; //边框颜色为绿色  
led_style.body.border.width = 5; //边框宽度  
led_style.body.shadow.color = LV_COLOR_BLUE; //阴影为蓝色  
led_style.body.shadow.width = 5; //阴影大小  
  
lv_obj_set_style(LED, &led_style); //设置LED样式  
lv_led_set_bright(LED, 255); //设置LED亮度, 亮度范围100~255  
lv_led_off(LED); //关闭LED
```



画圆弧

```
lv_obj_t * lv_arc_create(lv_obj_t * par, const lv_obj_t * copy) //创建弧形  
par: 圆弧嵌入的父对象  
copy: 平时填 NULL  
static lv_style_t arc_style;  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *arc = lv_arc_create(scr, NULL); //创建圆弧对象  
lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0); // 弧形在屏幕居中对齐  
  
lv_style_copy(&arc_style, &lv_style_plain_color);  
arc_style.line.width = 6; //弧线宽度  
arc_style.line.color = LV_COLOR_RED; //弧线红色  
  
lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &arc_style); //设置弧线样式
```



设置弧线半径在 littleVGL 是设置弧形长宽大小来解决

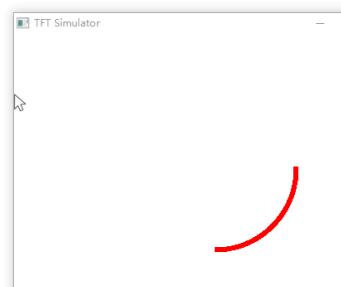
```
lv_style_copy(&arc_style, &lv_style_plain_color);  
arc_style.line.width = 6; //弧线宽度  
arc_style.line.color = LV_COLOR_RED; //弧线红色  
  
lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &arc_style); //设置弧线样式  
  
lv_obj_set_size(arc, 200, 200); //设置弧形半径大小
```



你发现弧线没有在屏幕居中对齐，其实要把 `lv_obj_set_size` 放在 `lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0)` 屏幕居中对齐前面就能居中对齐了

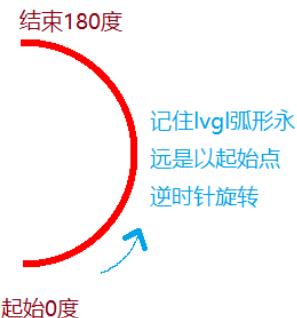
```
void lv_arc_set_angles(lv_obj_t * arc, uint16_t start, uint16_t end) //设置弧形起点和终点  
arc: 弧形对象  
start: 起点位置  
end: 终点位置
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *arc = lv_arc_create(scr, NULL); //创建圆弧对象  
lv_obj_set_size(arc, 200, 200); //设置弧形半径大小  
lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0); // 弧形在屏幕居中对齐  
  
lv_style_copy(&arc_style, &lv_style_plain_color);  
arc_style.line.width = 6; //弧线宽度  
arc_style.line.color = LV_COLOR_RED; //弧线红色  
  
lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &arc_style); //设置弧线样式  
  
lv_arc_set_angles(arc, 0, 90); //设置弧形起点和 终点
```



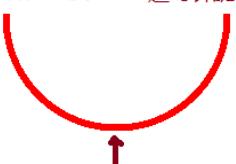
```
lv_arc_set_angles(arc, 起点, 终点);
```

```
lv_arc_set_angles(arc, 0, 180); //设置弧形起点和 终点
```



```
lv_arc_set_angles(arc, 270, 90); //设置弧形起点和 终点
```

起始270度 逆时针旋转180度结束



进度条使用

```
lv_obj_t * lv_bar_create(lv_obj_t * par, const lv_obj_t * copy) //创建进度条
```

par: 进度条父对象

copy: 平时填 NULL

```
void lv_bar_set_value(lv_obj_t * bar, int16_t value, lv_anim_enable_t anim) //设置进度条值
```

bar: 进度条对象

value: 设置进度条当前值

anim: LV_ANIM_OFF 进度条直接显示占用大小

LV_ANIM_ON 进度条有个显示占用大小的动画过程

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

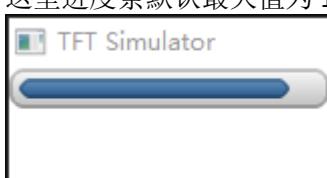
```
lv_obj_t *bar = lv_bar_create(scr, NULL);
```

```
lv_obj_set_size(bar, 160, 20); //水平进度条宽度>高度
```

```
lv_bar_set_value(bar, 90, LV_ANIM_ON);
```

```
//直接设置进度条加载到90的位置, LV_ANIM_ON表示要显示进度走到90的动画
```

这里进度条默认最大值为 100



```
void lv_bar_set_anim_time(lv_obj_t * bar, uint16_t anim_time) //设置动画时长
bar: 进度条对象
anim_time: 动画时长, 毫秒单位
lv_bar_set_anim_time 一定要放在 lv_bar_set_value 之前执行
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *bar = lv_bar_create(scr, NULL);
lv_obj_set_size(bar, 160, 20); //水平进度条宽度>高度
lv_bar_set_anim_time(bar, 3000); //设置动画时长3秒, 前提必须将LV_ANIM_ON打开, 该函数必须放在lv_bar_set_value之前
lv_bar_set_value(bar, 90, LV_ANIM_ON); //直接设置进度条加载到90的位置, LV_ANIM_ON表示要显示进度走到90的动画
```



```
void lv_bar_set_range(lv_obj_t * bar, int16_t min, int16_t max) //设置进度条范围
bar: 进度条对象
min: 进度条最小值
max: 进度条最大值
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *bar = lv_bar_create(scr, NULL);
lv_obj_set_size(bar, 160, 20); //水平进度条宽度>高度

lv_bar_set_anim_time(bar, 3000); //设置动画时长3秒, 前提必须将LV_ANIM_ON打开, 该函数必须放在lv_bar_set_value之前
lv_bar_set_range(bar, 0, 200); //设置进度条最大值范围
lv_bar_set_value(bar, 100, LV_ANIM_ON); //进度设置到100
```



进度条样式设置

```
void lv_bar_set_style(lv_obj_t * bar, lv_bar_style_t type, const lv_style_t * style) //进度条有两个样式
bar: 进度条对象
type: 决定设置进度条背景样式, 还是进度条样式
    LV_BAR_STYLE_BG 背景样式
    LV_BAR_STYLE_INDIC 进度条样式
Style: 传入样式
static lv_style_t bar_style_background; //修饰进度条背景样式
static lv_style_t bar_style_indic; //修饰进度条进度样式

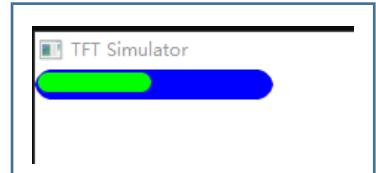
lv_bar_set_anim_time(bar, 3000); //设置动画时长3秒, 前提必须将LV_ANIM_ON打开, 该函数必须放在lv_bar_set_value之前

lv_bar_set_range(bar, 0, 200); //设置进度条最大值范围
lv_bar_set_value(bar, 100, LV_ANIM_ON); //进度设置到100

lv_style_copy(&bar_style_background, &lv_style_plain_color); //背景样式
bar_style_background.body.main_color = LV_COLOR_BLUE; //背景上半边为蓝色
bar_style_background.body.grad_color = LV_COLOR_BLUE; //背景下半边为蓝色
bar_style_background.body.radius = LV_RADIUS_CIRCLE; //背景绘制圆角

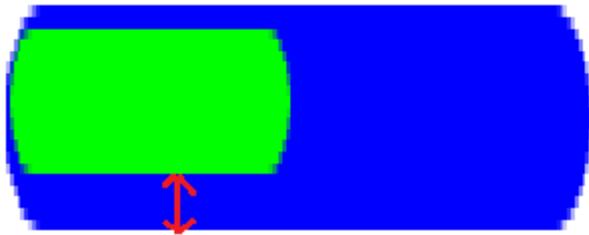
lv_style_copy(&bar_style_indic, &lv_style_plain_color); //进度条样式
bar_style_indic.body.main_color = LV_COLOR_LIME; //进度条为灰色
bar_style_indic.body.grad_color = LV_COLOR_LIME; //进度条为灰色
bar_style_indic.body.radius = LV_RADIUS_CIRCLE; //进度条绘制圆角
bar_style_indic.body.padding.left = 2; //进度条离背景左边框2, 这个间距不能太大, 不然进度条就消失了
bar_style_indic.body.padding.top = 2; //进度条离背景上边框2
bar_style_indic.body.padding.right = 5; //进度条离背景右边框5
bar_style_indic.body.padding.bottom = 5; //进度条离背景下边框5

lv_bar_set_style(bar, LV_BAR_STYLE_BG, &bar_style_background); //进度条背景样式
lv_bar_set_style(bar, LV_BAR_STYLE_INDIC, &bar_style_indic); //进度条样式
```



重点讲下进度条边框值设置

```
bar_style_indic.body.padding.left = 2; //进度条离背景左边框2,这  
bar_style_indic.body.padding.top = 2; //进度条离背景上边框2  
bar_style_indic.body.padding.right = 5; //进度条离背景右边框5  
bar_style_indic.body.padding.bottom = 5; //进度条离背景下边框5
```



进度条离下边框为5，如果我改成10，因为
进度条背景范围太小很可能导致进度条就不
见了。其余top, left, right都是一个原理

水平改为垂直进度条实现

```
static lv_style_t bar_style_background; //修饰进度条背景样式  
static lv_style_t bar_style_indic; //修饰进度条进度样式  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *bar = lv_bar_create(scr, NULL);  
lv_obj_set_size(bar, 160, 20); //水平进度条宽度 > 高度  
lv_obj_set_size(bar, 20, 160); //垂直进度条宽度 < 高度  
  
lv_bar_set_anim_time(bar, 3000);  
//设置动画时长3秒，前提必须将LV_ANIM_ON打开，该函数必须放在lv_bar_set_value之前  
  
lv_bar_set_range(bar, 0, 200); //设置进度条最大值范围  
lv_bar_set_value(bar, 100, LV_ANIM_ON); //进度设置到100  
  
lv_style_copy(bar_style_background, lv_style_plain_color); //背景样式
```



代码示例

```
static lv_style_t bar_style_background; //修饰进度条背景样式  
static lv_style_t bar_style_indic; //修饰进度条进度样式  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *bar = lv_bar_create(scr, NULL);  
lv_obj_set_size(bar, 160, 20); //水平进度条宽度 > 高度  
lv_obj_set_size(bar, 20, 160); //垂直进度条宽度 < 高度  
lv_bar_set_anim_time(bar, 3000);  
//设置动画时长3秒，前提必须将LV_ANIM_ON打开，该函数必须放在lv_bar_set_value之前  
  
lv_bar_set_range(bar, 0, 200); //设置进度条最大值范围
```

```

lv_bar_set_value(bar, 100, LV_ANIM_ON); //进度设置到 100

lv_style_copy(&bar_style_background, &lv_style_plain_color); //背景样式
bar_style_background.body.main_color = LV_COLOR_BLUE; //背景上半边为蓝色
bar_style_background.body.grad_color = LV_COLOR_BLUE; //背景下半边为蓝色
bar_style_background.body.radius = LV_RADIUS_CIRCLE; //背景绘制圆角

lv_style_copy(&bar_style_indic, &lv_style_plain_color); //进度条样式
bar_style_indic.body.main_color = LV_COLOR_LIME; //进度条为灰色
bar_style_indic.body.grad_color = LV_COLOR_LIME; //进度条为灰色
bar_style_indic.body.radius = LV_RADIUS_CIRCLE; //进度条绘制圆角
bar_style_indic.body.padding.left = 2; //进度条离背景左边框 2, 这个间距不能太大, 不然进度条就消失了
bar_style_indic.body.padding.top = 2; //进度条离背景上边框 2
bar_style_indic.body.padding.right = 5; //进度条离背景右边框 5
bar_style_indic.body.padding.bottom = 5; //进度条离背景下边框 5

lv_bar_set_style(bar, LV_BAR_STYLE_BG, &bar_style_background); //进度条背景样式
lv_bar_set_style(bar, LV_BAR_STYLE_INDIC, &bar_style_indic); //进度条样式

```

复选框使用

`lv_obj_t * 返回句柄 = lv_cb_create(父对象, NULL) //创建复选框`

案例 1:

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t * cb1 = lv_cb_create(scr, NULL); //创建复选框
lv_obj_align(cb1, NULL, LV_ALIGN_CENTER, 0, 0); //屏幕居中对齐

```



`lv_cb_set_text(传入复选框对象, "填入字符串") //设置复选框 动态文本`
案例 2:

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t * cb1 = lv_cb_create(scr, NULL); //创建复选框
lv_obj_align(cb1, NULL, LV_ALIGN_CENTER, 0, 0); //屏幕居中对齐
lv_cb_set_text(cb1, "xxxxzz"); //设置复选框 动态文本

```



```

lv_cb_set_static_text(传入复选框对象,"填入字符串") //静态文本
void text()
{
    lv_cb_set_static_text(cb1,"static text"); //静态文本
}
void main()
{
    text();
    while(1);
}

```

如果在子函数(栈空间)使用静态文本，就一定会崩溃，因为子函数执行完就释放空间了。这样复选框显示得文本就会出现 BUG

所以静态文本比较适合那种系统初始化的时候就是设置好，或者子函数永远不会执行完这种系统。

lv_cb_set_checked(传入复选框对象,true) //复选框默认为选中状态
false 为默认不选中状态

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t * cb1 = lv_cb_create(scr,NULL); //创建复选框
lv_obj_align(cb1,NULL,LV_ALIGN_CENTER,0,0); //屏幕居中对齐
lv_cb_set_text(cb1,"xxxxxx"); //设置复选框 动态文本
lv_cb_set_checked(cb1,true); //复选框默认为选中状态

```

● xxxxxx

复选框选中后，触发事件回调函数实现

lv_obj_set_event_cb(传入复选框对象,传入事件回调函数) //创建复选框事件回调函数

返回值 int=lv_cb_is_checked(传入复选框对象)//获取复选框选中/不选中状态，选中返回 1 不选中返回 0

```

LV_EVENT_VALUE_CHANGED 事件 //对象的数值发生改变时被触发，如滑块控件，复选框
lv_obj_t * cb1; //因为复选框触发事件函数会用到句柄，所以定义成全局
const char cb_txt[] = {"static text"}; //如果使用静态文本，那么字符定义成全局，不会被栈释放

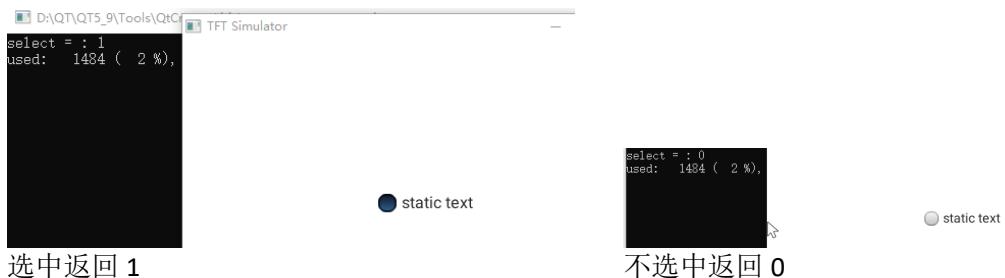
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == cb1) //监听复选框事件触发
    {
        if(event == LV_EVENT_VALUE_CHANGED)
        {
            printf("select = : %d\n",lv_cb_is_checked(cb1));
        }
    }
    else
    {
    }
}

```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
cb1 = lv_cb_create(scr,NULL); //创建复选框
lv_obj_align(cb1,NULL,LV_ALIGN_CENTER,0,0); //屏幕居中对齐
lv_cb_set_static_text(cb1,cb_txt); //静态文本
lv_obj_set_event_cb(cb1,xzz_event_cb); //创建事件回调函数

```



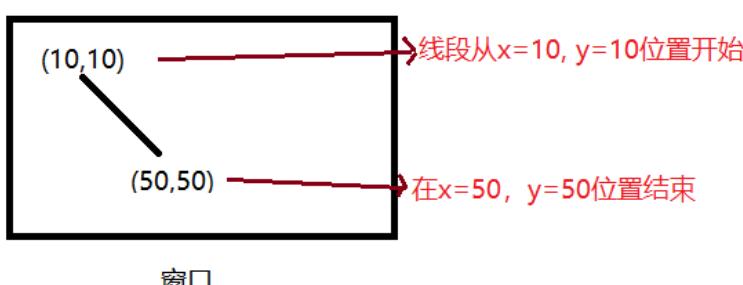
画线使用

```
lv_obj_t *返回对象 = lv_line_create(窗口对象,NULL) //创建线条
lv_line_set_auto_size(线条对象,false) //如果要手动设置线条大小，先关闭掉默认的线条自动大小
lv_obj_set_size(线条对象,宽度,高度) //手动设置线条大小(一般都是自动设置),这里只是尝试
```

lv_line_set_points(线条对象,坐标点集合,坐标点个数) //设置线条坐标点,有了坐标点才有线条
线条对象：就是线条本身

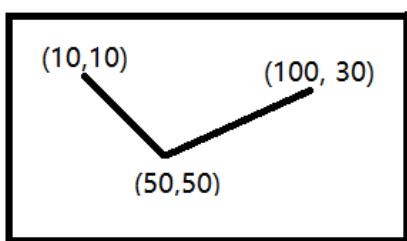
坐标点集合(关键参数)：是一个二维数组，数组中的每一个元素就是一个坐标点

```
array[] = { {10, 10}, {50,50} }
```



窗口

array[] = { {10, 10}, {50,50}, {100 , 30} } 三个坐标点就多加一个元素



窗口

坐标点个数：就是计算二维数组有多少个元素，三个坐标点就填 3,四个坐标点就填 4，最好用 sizeof(...)/sizeof(...[])来计算

```
const lv_point_t PointArray[] = {{10,10},{50,70},{80,50}}; //设置画线的坐标点集合
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *line1 = lv_line_create(scr,NULL); //创建线条
lv_line_set_auto_size(line1,false); //关闭掉默认的线条自动大小
lv_obj_set_size(line1,200,200); //手动设置线条大小
lv_obj_align(line1,NULL,LV_ALIGN_CENTER,0,0); //与屏幕自动居中
lv_line_set_points(line1,PointArray,3); //设置3个坐标点，线段按照这3个坐标点生成
```

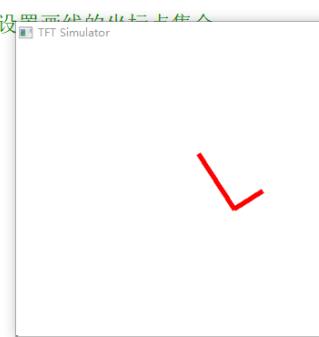
```
const lv_point_t PointArray[] = {{10,10},{50,70},{80,50}}; //设置画线的坐标点集合  
lv_style_t line_style;//创建样式  
lv_style_copy(&line_style,&lv_style_plain_color); //拷贝样式  
line_style.line.color = LV_COLOR_RED; //设置线条为红色
```



```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *line1 = lv_line_create(scr,NULL); //创建线条  
lv_line_set_auto_size(line1,false); //关闭掉默认的线条自动大小  
lv_obj_set_size(line1,200,200); //手动设置线条大小  
lv_obj_align(line1,NULL,LV_ALIGN_CENTER,0,0); //与屏幕自动居中  
lv_line_set_points(line1,PointArray,3); //设置3个坐标点，线段按照这3个坐标点生成
```

用样式修改线条颜色

```
const lv_point_t PointArray[] = {{10,10},{50,70},{80,50}}; //设置画线的坐标点集合  
lv_style_t line_style;//创建样式  
lv_style_copy(&line_style,&lv_style_plain_color); //拷贝样式  
line_style.line.color = LV_COLOR_RED; //设置线条为红色  
line_style.line.width = 4; //线条变宽4，最大只能到4
```



```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *line1 = lv_line_create(scr,NULL); //创建线条  
//lv_line_set_auto_size(line1,false); //关闭掉默认的线条自动大小  
//lv_obj_set_size(line1,200,200); //手动设置线条大小  
lv_obj_align(line1,NULL,LV_ALIGN_CENTER,0,0); //与屏幕自动居中  
lv_line_set_points(line1,PointArray,3); //设置3个坐标点，线段按照这3个坐标点生成
```

lv_line_set_style(line1,LV_LINE_STYLE_MAIN,&line_style); //设置样式进系统

线条宽度不能超过 4 是因为模拟器有 BUG，在开发板上线条宽度为 6,8,7,10 都没有问题

```
const lv_point_t PointArray[] = {{10,10},{50,70},{80,50}}; //设置画线的坐标点集合  
lv_style_t line_style;//创建样式  
lv_style_copy(&line_style,&lv_style_plain_color); //拷贝样式  
line_style.line.color = LV_COLOR_RED; //设置线条为红色  
line_style.line.width = 4; //线条变宽4，最大只能到4  
line_style.line.rounded = 1; //设置线条端头 为圆弧
```



```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *line1 = lv_line_create(scr,NULL); //创建线条  
//lv_line_set_auto_size(line1,false); //关闭掉默认的线条自动大小  
//lv_obj_set_size(line1,200,200); //手动设置线条大小  
lv_obj_align(line1,NULL,LV_ALIGN_CENTER,0,0); //与屏幕自动居中  
lv_line_set_points(line1,PointArray,3); //设置3个坐标点，线段按照这3个坐标点生成
```

lv_line_set_style(line1,LV_LINE_STYLE_MAIN,&line_style); //设置样式进系统

线条两端有圆弧，只是看不清楚而已。

滑块控件

```
lv_obj_t *滑块对象 = lv_slider_create(父窗口,NULL)//创建滑块(默认范围 0 ~ 100)  
lv_slider_set_value(滑块对象,滑块当前值,LV_ANIM_ON) //设置滑块当前值, 开启动画效果
```

案例 1:

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *slider = lv_slider_create(scr, NULL); //创建滑块  
lv_obj_set_size(slider, 120, 20); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值, 开启动画效果
```



```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *slider = lv_slider_create(scr, NULL); //创建滑块  
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值, 开启动画效果
```



我修改滑块的宽度和高度，滑块变粗了。

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *slider = lv_slider_create(scr, NULL); //创建滑块  
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_value(slider, 80, LV_ANIM_OFF); //设置滑块当前值, 关闭动画效果
```



关闭动画效果后，开机没有滑块滑动的效果。

```
lv_slider_set_anim_time(滑块对象, 毫秒) //设置动画时长
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *slider = lv_slider_create(scr, NULL); //创建滑块  
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_anim_time(slider, 3000); //设置滑块动画时长, 3 秒, (记住: 动画时长执行在  
lv_slider_set_value 设置滑块值之前)  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值, 打开动画效果
```



用了 3 秒时间，滑块在滑动到当前值

```
lv_slider_set_range(滑块对象,滑动最小值,滑动最大值) //设置滑动范围
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *slider = lv_slider_create(scr, NULL); //创建滑块  
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_range(slider, 0, 200); //设置滑动范围0~200  
lv_slider_set_anim_time(slider, 3000); //设置滑块动画时长, 3秒, (记住: 动画时长执行在lv_slider)  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值80, 打开动画效果
```



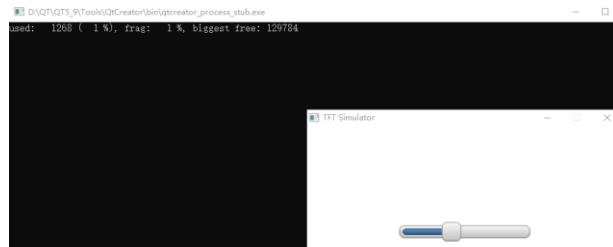
因为滑动范围是 0 ~ 200, 当前滑动值设置的 80, 所以滑块显示滑动一半不到。

滑块事件触发回调函数使用

返回值 int16_t = lv_slider_get_value(滑块对象) //获取滑块当前值

LV_EVENT_VALUE_CHANGED 事件在滑块应用中的缺陷

```
lv_obj_t *slider; //因为滑块触发事件函数会用到句柄, 所以定义成全局  
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(obj == slider) //监听滑块事件触发  
    {  
        if(event == LV_EVENT_VALUE_CHANGED) //滑块数字改变就会被触发  
        {  
            printf("value = : %d\n", lv_slider_get_value(slider)); //输出滑块 进度值  
        }  
    }  
    else  
    {  
  
    }  
}  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
slider = lv_slider_create(scr, NULL); //创建滑块  
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_range(slider, 0, 200); //设置滑动范围0~200  
lv_slider_set_anim_time(slider, 3000); //设置滑块动画时长, 3秒, (记住: 动画时长执行在lv_slider)  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值80, 打开动画效果  
lv_obj_set_event_cb(slider, xzz_event_cb); //注册事件回调函数
```



滑块开机动画是不会发生事件回调的



手动触发

这里就有一个问题，如果手动一直拖动滑块，那么事件就会不停的被触发，因为 `LV_EVENT_VALUE_CHANGED` 事件是只要数字改变就会被触发

```
lv_obj_t *slider; //因为滑块触发事件函数会用到句柄，所以定义成全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == slider) //监听滑块事件触发
    {
        if(event == LV_EVENT_VALUE_CHANGED) //滑块数字改变就会被触发
        {
            printf("value = : %d\n",lv_slider_get_value(slider)); //输出滑块 进度值
        }
    }
    else
    {

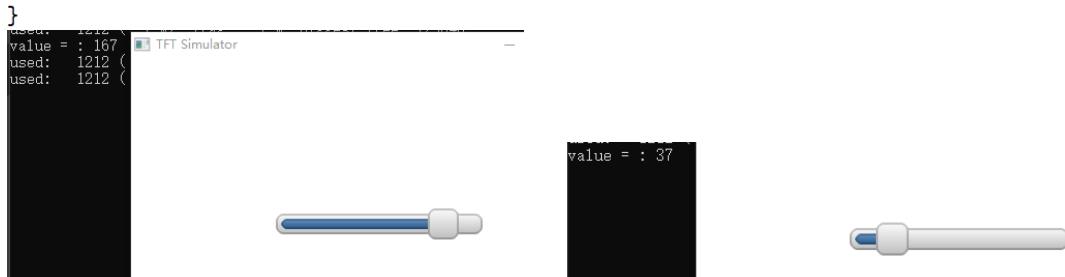
    }
}
```

不停的移动滑块，那么事件就会不停的被触发，在实时操作系统里面，会占用大量的 CPU 时间，导致系统效率低下

`LV_EVENT_RELEASED` 使用鼠标松开事件来获取滑块值

```
lv_obj_t *slider; //因为滑块触发事件函数会用到句柄，所以定义成全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == slider) //监听滑块事件触发
    {
        if(event == LV_EVENT_RELEASED) //鼠标松开滑块才会被触发
        {
            printf("value = : %d\n",lv_slider_get_value(slider)); //输出滑块 进度值
        }
    }
    else
    {

    }
}
```

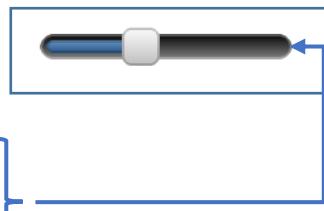


鼠标松开才会触发事件，节省了 CPU 资源。

滑块样式设置

`LV_SLIDER_STYLE_BG` 设置滑块背景样式

```
lv_obj_t *slider;
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
slider = lv_slider_create(scr, NULL); //创建滑块
lv_style_t slider_bg_style; //背景样式
lv_style_copy(&slider_bg_style,&lv_style_pretty); //拷贝样式，设置背景样式
slider_bg_style.body.main_color = LV_COLOR_BLACK; //背景上半部分是黑色(渐变)
slider_bg_style.body.grad_color = LV_COLOR_GRAY; //背景下半部分是灰色(渐变)
slider_bg_style.body.radius = LV_RADIUS_CIRCLE; //背景四角有弧度
```



```
slider_bg_style.body.border.color = LV_COLOR_WHITE;//边框为白色  
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_range(slider, 0, 200); //设置滑动范围 0~200  
lv_slider_set_anim_time(slider, 3000); //设置滑块动画时长, 3 秒,  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值 80, 打开动画效果  
lv_slider_set_style(slider, LV_SLIDER_STYLE_BG, &slider_bg_style); //设置背景样式
```

设置指示器样式

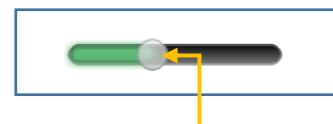
```
lv_style_t slider_bg_style; //背景样式
    lv_style_copy(&slider_bg_style,&lv_style_pretty); //拷贝样式, 设置背景样式
    slider_bg_style.body.main_color = LV_COLOR_BLACK; //背景上半部分是黑色(渐变)
    slider_bg_style.body.grad_color = LV_COLOR_GRAY; //背景下半部分是灰色(渐变)
    slider_bg_style.body.radius = LV_RADIUS_CIRCLE; //背景四角有弧度
    slider_bg_style.body.border.color = LV_COLOR_WHITE; //边框为白色
```

```
lv_style_t slider_indic_style; //指示器样式
lv_style_copy(&slider_indic_style, &lv_style_pretty); //拷贝样式, 设置指示器样式
slider_indic_style.body.main_color = LV_COLOR_MAKE(0xf, 0xBB, 0x78); //指示器上半部分(渐变)
slider_indic_style.body.grad_color = LV_COLOR_MAKE(0xf, 0xB8, 0x78); //指示器下半部分(渐变)
slider_indic_style.body.radius = LV_RADIUS_CIRCLE; //指示器四角带圆弧
slider_indic_style.body.shadow.width = 8; //指示器阴影宽度
slider_indic_style.body.shadow.color = slider_indic_style.body.main_color; //指示器阴影颜色
slider_indic_style.body.padding.left = 3; //设置指示器背景和左边框之间距离
slider_indic_style.body.padding.right = 3; //设置指示器背景和右边框之间距离
slider_indic_style.body.padding.top = 3; //设置指示器背景和上边框之间距离
slider_indic_style.body.padding.bottom = 3; //设置指示器背景和下边框之间距离
```

```
lv_obj_set_size(slider, 200, 30); //设置滑块宽度和高度  
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0); //滑块与屏幕居中  
lv_slider_set_range(slider, 0, 200); //设置滑动范围 0~200  
lv_slider_set_anim_time(slider, 3000); //设置滑块动画时长, 3 秒,  
lv_slider_set_value(slider, 80, LV_ANIM_ON); //设置滑块当前值 80, 打开动画效果  
lv_slider_set_style(slider, LV_SLIDER_STYLE_BG, &slider_bg_style); //设置背景样式  
lv_slider_set_style(slider, LV_SLIDER_STYLE_INDIC, &slider_indic_style); //设置指示器样式
```

设置旋钮样式

```
lv_style_t slider_nob_style; //旋钮样式  
lv_style_copy(&slider_nob_style,&lv_style_pretty); //拷贝样式, 设置旋钮样式  
slider_nob_style.body.radius = LV_RADIUS_CIRCLE; //旋钮为圆角  
slider_nob_style.body.opa = LV_OPA_70; //旋钮背景色透明度  
.....  
lv_slider_set_style(slider,LV_SLIDER_STYLE_BG,&slider_bg_style); //设置背景样式  
lv_slider_set_style(slider,LV_SLIDER_STYLE_INDIC,&slider_indic_style); //设置指示器样式  
lv_slider_set_style(slider,LV_SLIDER_STYLE_KNOB,&slider_nob_style); //设置旋钮样式
```



lv obj align(当前对象,参考对象,对齐方式,x 偏移,y 偏移) //将当前对象与另外一个对象对齐

当前对象：才创建的对象，我们定义为对象 1

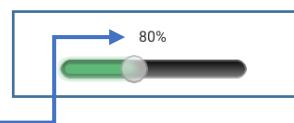
参考对象：以前就创建好的对象，我们定义为对象 2

对齐方式: LV ALIGN OUT TOP MID 我们要求对象 1 在对象 2 顶部对齐

OUT TOP LEFT	OUT TOP MID	OUT TOP RIGHT
OUT LEFTIN TOP TOP LEFT	IN TOP MID	IN OUTTOP RIGHT RIGHT TOP
OUT LEFTIN LEFT MID MID	CENTER	IN RIGHTOUT RIGHT MID MID
OUT LEFTIN BOTTOM BOTTOM LEFT	IN BOTTOM MID	IN BOTTOMOUT RIGHT RIGHT BOTTOM
OUT BOTTOM	OUT BOTTOM	OUT BOTTOM

x,y 偏移：就是对象 2 离对象 1 偏离多少

```
lv_obj_set_event_cb(slider,xzz_event_cb); //注册事件回调函数  
label = lv_label_create(scr,NULL); //创建文本框显示滑块数据  
lv_obj_align(label,slider,LV_ALIGN_OUT_TOP_MID,0,-10); //位置布局到滑块上端  
lv_label_set_text(label,"80%"); //先默认显示滑块值
```



```

lv_obj_t *slider; //因为滑块触发事件函数会用到句柄, 所以定义成全局
lv_obj_t *label; //标签对象
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    char buff[16]; //字符串显示滑块值
    if(obj == slider) //监听滑块事件触发
    {
        if(event == LV_EVENT_RELEASED || event == LV_EVENT_VALUE_CHANGED) //鼠标松开滑块和移动滑块会被触发
        {
            sprintf(buff, "%d%%", lv_slider_get_value(slider)); //获取滑块当前值
            lv_label_set_text(label, buff); //设置滑块当前值是多少, 显示
            lv_obj_realign(label);
        }
    }
    else
    {
    }
}

```

开关控件

lv_obj_t *对象 = lv_sw_create(父窗口,NULL) //创建开关

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *sw = lv_sw_create(scr, NULL); //创建开关
lv_obj_set_size(sw, 100, 40); //设置开关大小
lv_obj_align(sw, NULL, LV_ALIGN_CENTER, 0, 0); //开关居中

```



开关未打开



开关打开

lv_sw_on(开关对象, LV_ANIM_ON) //开机, 开关默认打开, 使用动画效果
lv_sw_off(开关对象, LV_ANIM_ON) //开机, 开关默认关闭

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *sw = lv_sw_create(scr, NULL); //创建开关
lv_obj_set_size(sw, 100, 40); //设置开关大小
lv_obj_align(sw, NULL, LV_ALIGN_CENTER, 0, 0); //开关居中
lv_sw_on(sw, LV_ANIM_ON); //开机按钮默认打开, 使用动画效果

```



开关, 打开/关闭事件回调函数

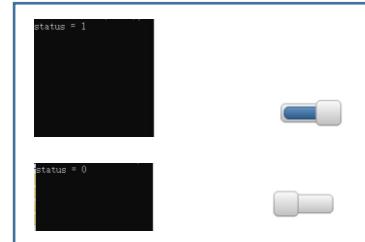
返回值 bool = lv_sw_get_state(开关对象) //获取开关打开/关闭状态, 打开返回 1, 关闭返回 0

```

lv_obj_t *sw;
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == sw)
    {
        if(event == LV_EVENT_VALUE_CHANGED)
        {
            printf("status = %d\n", lv_sw_get_state(sw)); //获取开关状态
        }
    }
}

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
sw = lv_sw_create(scr, NULL); //创建开关
lv_obj_set_size(sw, 100, 40); //设置开关大小
lv_obj_align(sw, NULL, LV_ALIGN_CENTER, 0, 0); //开关居中
lv_sw_on(sw, LV_ANIM_ON); //开机按钮默认打开, 使用动画效果
lv_obj_set_event_cb(sw, xzz_event_cb); //注册开关打开/关闭事件回调函数

```



开关样式设置

LV_SW_STYLE_KNOB_OFF 和 LV_SW_STYLE_KNOB_ON 开关闭和打开样式设置
lv_sw_set_style(开关对象,样式设置,样式内容) //开关样式设置

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
sw = lv_sw_create(scr, NULL); //创建开关

lv_style_t sw_bg_style; //背景样式
lv_style_copy(&sw_bg_style, &lv_style_plain_color); //拷贝样式
sw_bg_style.body.main_color = LV_COLOR_MAKE(0xcc, 0xcc, 0xcc);
sw_bg_style.body.grad_color = LV_COLOR_MAKE(0xcc, 0xcc, 0xcc);
sw_bg_style.body.radius = LV_RADIUS_CIRCLE; //圆角
sw_bg_style.body.padding.left = -3;
sw_bg_style.body.padding.right = -3;
sw_bg_style.body.padding.top = -3;
sw_bg_style.body.padding.bottom = -3;

lv_style_t sw_indic_style; //开关指示器样式
lv_style_copy(&sw_indic_style, &lv_style_plain_color);
sw_indic_style.body.main_color = LV_COLOR_MAKE(0x65, 0xc4, 0x66);
sw_indic_style.body.grad_color = LV_COLOR_MAKE(0x65, 0xc4, 0x66);
sw_indic_style.body.radius = LV_RADIUS_CIRCLE; //圆角
sw_indic_style.body.padding.left = 0; //让指示器与背景边框无距离
sw_indic_style.body.padding.right = 0;
sw_indic_style.body.padding.bottom = 0;
sw_indic_style.body.padding.top = 0;

lv_style_t sw_knob_off_style; //开关闭关样式
lv_style_copy(&sw_knob_off_style, &lv_style_plain_color);
sw_knob_off_style.body.main_color = LV_COLOR_WHITE;
sw_knob_off_style.body.grad_color = LV_COLOR_WHITE;
sw_knob_off_style.body.radius = LV_RADIUS_CIRCLE; //圆角;
sw_knob_off_style.body.shadow.color = LV_COLOR_MAKE(0xA0, 0xA0, 0xA0); //阴影颜色
sw_knob_off_style.body.shadow.width = 6; //阴影宽度

lv_style_t sw_knob_on_style; //开关打开样式
lv_style_copy(&sw_knob_on_style, &sw_knob_off_style); //开关打开样式和关闭样式一样

lv_obj_set_size(sw, 100, 40); //设置开关大小
lv_obj_align(sw, NULL, LV_ALIGN_CENTER, 0, 0); //开关居中
lv_sw_on(sw, LV_ANIM_ON); //开机按钮默认打开，使用动画效果
lv_obj_set_event_cb(sw, xzz_event_cb); //注册开关打开/关闭事件回调函数
lv_sw_set_style(sw, LV_SW_STYLE_BG, &sw_bg_style); //开关背景样式
lv_sw_set_style(sw, LV_SW_STYLE_INDIC, &sw_indic_style); //开关指示器样式
lv_sw_set_style(sw, LV_SW_STYLE_KNOB_OFF, &sw_knob_off_style); //开关闭关样式
lv_sw_set_style(sw, LV_SW_STYLE_KNOB_ON, &sw_knob_on_style); //开关打开样式
```



矩阵按钮

将按钮建立在矩形框当中



矩阵框

```
lv_obj_t *矩阵对象 = lv_btm_create(父窗口对象,NULL) //创建矩阵按钮
```

```
lv_btm_set_map(传入矩阵对象,传入字符串) //将按钮字符加载进矩阵
```

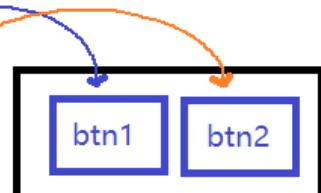
传入矩阵对象: 就是传入 create 创建的对象

传入字符串: 注意, 该字符串就是矩阵里面创建多少个按钮的关键, 所以这里的字符串很特殊。

如: const char *btm1_map[] = {"btn1","btn2",""};

添加两个btn1, btn2字符, 就是创建两个按钮。

记住最后必须以空字符""结尾, 因为系统是靠判断空字符串来结尾的



矩阵对象

创建按钮的字符串一定要在堆区或者全局区, 不能因为子函数执行完就自动释放栈。这样会造成 BUG

```
const char *btm1_map[] = {"btn1","btn2",""};//创建2个按钮, ""结尾  
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
lv_obj_t *btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵
```



所以矩阵按钮就是用字符串创建的按钮

矩阵按钮大小设置

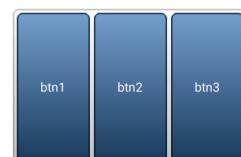
```
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
  
lv_obj_t *btm1 = lv_btm_create(scr,NULL); //创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵  
lv_obj_set_size(btm1,100,100); //设置矩阵按钮大小
```



矩阵框内部按钮, 是根据矩阵框大小统一设置的。

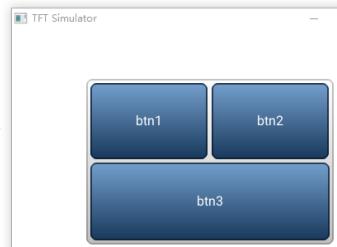
创建 3 个按钮如下:

```
const char *btm1_map[] = {"btn1","btn2","btn3","",""};//创建3个按钮  
  
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
  
lv_obj_t *btm1 = lv_btm_create(scr,NULL); //创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵
```



"\n"使用换行符显示两排按钮

```
const char *btm1_map[] = {"btn1","btn2","\n","btn3","",""};//创建3个按钮, "\n"换行  
  
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
  
lv_obj_t *btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵
```



使用换行符可以将按钮控件进行换行显示

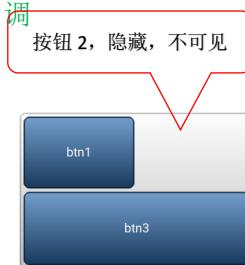
lv_btm_set_btn_ctrl(矩阵对象,字符数组某个元素,属性) //设置某单个按钮的控制属性

矩阵对象: 就是传入 create 创建的对象

字符数组某个元素:根据字符串某个元素决定操作某个按钮

属性:

```
LV_BTNM_CTRL_HIDDEN //不可见  
LV_BTNM_CTRL_NO_REPEAT //取消鼠标压着按钮不停的触发事件回调  
LV_BTNM_CTRL_INACTIVE //按钮为禁用状态  
  
const char *btm1_map[] = {"btn1","btn2","\n","btn3","",""};//创建3个按钮  
  
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
  
lv_obj_t *btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵  
lv_btm_set_btn_ctrl(btm1,1,LV_BTNM_CTRL_HIDDEN); //数组1按钮 不可见
```



选择数组(1),也就是按钮 2

矩阵按钮事件回调函数实现

```
lv_obj_t *btm1;  
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(obj == btm1)  
    {  
        if(event == LV_EVENT_VALUE_CHANGED) //按钮按下触发  
        {  
            printf("key press \n");  
        }  
    }  
}  
  
const char *btm1_map[] = {"btn1","btn2","\n","btn3","",""};//创建3个按钮, "\n"换行  
  
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
  
btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1,xzz_event_cb); //创建矩阵按钮事件回调函数
```



任何一个按钮按下都是执行相同的事件回调函数

取消矩阵按钮一直被压着触发事件回调

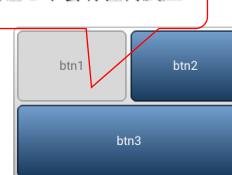
```
const char *btm1_map[] = {"btn1","btn2","\n","btn3",""};//创建3个按钮, "\n"换行  
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象  
  
btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0);//矩阵按钮居中对齐  
lv_btm_set_map(btm1, btm1_map);//将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1,xzz_event_cb);//创建矩阵按钮事件回调函数  
lv_btm_set_btn_ctrl(btm1,1,LV_BTNM_CTRL_NO_REPEAT);//取消按钮2一直按着不停的触发回调
```



指定矩阵按钮处于禁用状态

```
btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0);//矩阵按钮居中对齐  
lv_btm_set_map(btm1, btm1_map);//将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1,xzz_event_cb);//创建矩阵按钮事件回调函数  
lv_btm_set_btn_ctrl(btm1,0,LV_BTNM_CTRL_INACTIVE);//按钮1处于禁用状态
```

点击按钮 1 不会有任何反应

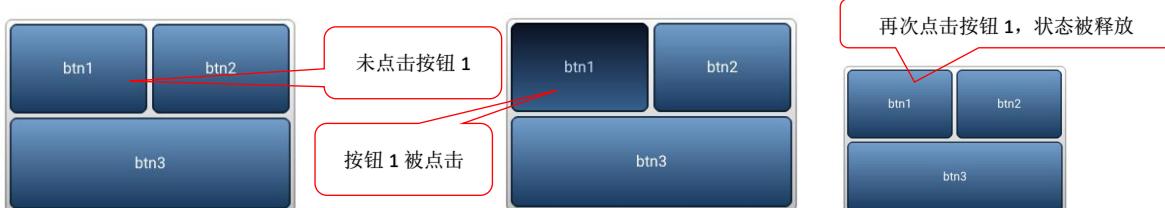


LV_BTNM_CTRL_TGL_ENABLE //按钮点击后一直处于按下状态

```
const char *btm1_map[] = {"btn1","btn2","\n","btn3",""};//创建3个按钮, "\n"换行
```

```
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象
```

```
btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0);//矩阵按钮居中对齐  
lv_btm_set_map(btm1, btm1_map);//将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1,xzz_event_cb);//创建矩阵按钮事件回调函数  
lv_btm_set_btn_ctrl(btm1,0,LV_BTNM_CTRL_TGL_ENABLE);//按钮1 点击后一直处于按下状态
```



LV_BTNM_CTRL_TGL_STATE //开机按钮默认显示为按下状态

```
const char *btm1_map[] = {"btn1","btn2","\n","btn3",""};//创建3个按钮, "\n"换行
```

```
lv_obj_t *scr = lv_scr_act();//获取当前活跃的屏幕对象
```

```
btm1 = lv_btm_create(scr,NULL);//创建矩阵按钮  
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0);//矩阵按钮居中对齐  
lv_btm_set_map(btm1, btm1_map);//将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1,xzz_event_cb);//创建矩阵按钮事件回调函数
```

```
lv_btm_set_btn_ctrl(btm1,0,LV_BTNM_CTRL_TGL_ENABLE|LV_BTNM_CTRL_TGL_STATE);  
//按钮1 开机默认按下, 点击后松开
```



`lv_btm_clear_btn_ctrl`(矩阵对象, 字符数组某个元素, 控制属性) // 清除指定单个按钮控制属性

矩阵对象: 就是传入 `create` 创建的对象

字符数组某个元素: 根据字符串某个元素决定操作某个按钮. 前面讲过

属性:

```
LV_BTN_CTRL_HIDDEN //不可见(取消)  
LV_BTN_CTRL_NO_REPEAT //取消鼠标压着按钮不停的触发事件回调(取消)  
LV_BTN_CTRL_INACTIVE //按钮为禁用状态(取消)  
.....等等
```

```
const char *btm1_map[] = {"btn1", "btn2", "\n", "btn3", ""}; //创建3个按钮, "\n"换行
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
btm1 = lv_btm_create(scr, NULL); //创建矩阵按钮  
lv_obj_align(btm1, NULL, LV_ALIGN_CENTER, 0, 0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1, btm1_map); //将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1, xzz_event_cb); //创建矩阵按钮事件回调函数
```

```
lv_btm_set_btn_ctrl(btm1, 0, LV_BTN_CTRL_TGL_ENABLE | LV_BTN_CTRL_TGL_STATE);  
//按钮1 开机默认按下, 点击后松开
```

```
lv_btm_clear_btn_ctrl(btm1, 0, LV_BTN_CTRL_TGL_STATE); //开机默认按下被我清除掉了
```



开机默认按钮 1 按下, 功能取消了

`lv_btm_set_ctrl_map`(矩阵对象, 传入一维数组) // 设置控制映射表, 批量修改按钮属性

矩阵对象: 就是传入 `create` 创建的对象

传入一维数组: 传入一个数组, 映射表数组中每个元素的属性对应按钮数组的每个元素

```
const char *btm1_map[] = {"btn1", "btn2", "\n", "btn3", ""}; //创建3个按钮, "\n"换行  
const lv_btm_ctrl_t btm1_ctrl_map[] = {  
    LV_BTN_CTRL_INACTIVE, //设置按钮1, 不可点击  
    LV_BTN_CTRL_NO_REPEAT, //设置按钮2, 不具备按钮压下连续触发回调  
    LV_BTN_CTRL_TGL_STATE //设置按钮3, 开机就按下  
};  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

映射表每一个元素, 对应一个按钮, "\n"不算, 所以映射表就是为了方便一次性修改多个按钮属性

```
btm1 = lv_btm_create(scr, NULL); //创建矩阵按钮  
lv_obj_align(btm1, NULL, LV_ALIGN_CENTER, 0, 0); //矩阵按钮居中对齐  
lv_btm_set_map(btm1, btm1_map); //将按钮字符加载进矩阵  
lv_obj_set_event_cb(btm1, xzz_event_cb); //创建矩阵按钮事件回调函数
```

```
lv_btm_set_ctrl_map(btm1, btm1_ctrl_map); //设置控制映射表
```



一次性修改每个按钮的属性

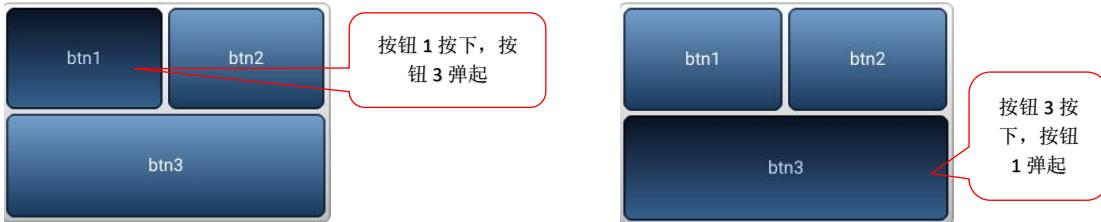
lv_btm_set_one_toggle(矩阵对象,true)//根据映射表，只能有一个按钮被切换
所有被 LV_BTNM_CTRL_TGL_ENABLE 属性设置的按钮，同一时刻就只能允许一个按钮被按下，其余按下的按钮要自动弹起

```
const char *btm1_map[] = {"btn1","btn2","\n","btn3",""};//创建3个按钮, "\n"换行
const lv_btm_ctrl_t btm1_ctrl_map[] = {
    LV_BTNM_CTRL_TGL_ENABLE, //设置按钮1, 具备按下状态
    LV_BTNM_CTRL_NO_REPEAT, //设置按钮2, 不具备按钮压下连续触发回调
    LV_BTNM_CTRL_TGL_ENABLE //设置按钮3, 具备按下状态
};

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

btm1 = lv_btm_create(scr,NULL); //创建矩阵按钮
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵
lv_obj_set_event_cb(btm1,xzz_event_cb); //创建矩阵按钮事件回调函数

lv_btm_set_ctrl_map(btm1,btm1_ctrl_map); //必须设置控制映射表
lv_btm_set_one_toggle(btm1,true); //根据映射表, 只能有一个按钮被按下
```



返回值 uint16_t = lv_btm_get_active_btn(矩阵对象) //获取矩阵按钮中哪一个按钮被按下，配合事件回调使用

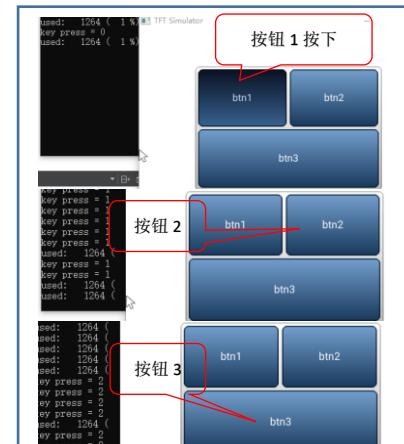
```
lv_obj_t *btm1;
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btm1)
    {
        if(event == LV_EVENT_VALUE_CHANGED) //按钮按下触发
        {
            printf("key press = %d\n", lv_btm_get_active_btn(btm1)); //获取哪个按钮被按下
        }
    }
}

const char *btm1_map[] = {"btn1","btn2","\n","btn3",""};//创建3个按钮, "\n"换行
const lv_btm_ctrl_t btm1_ctrl_map[] = {
    LV_BTNM_CTRL_TGL_ENABLE, //设置按钮1, 具备按下状态
    LV_BTNM_CTRL_NO_REPEAT, //设置按钮2, 不具备按钮压下连续触发回调
    LV_BTNM_CTRL_TGL_ENABLE //设置按钮3, 具备按下状态
};

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

btm1 = lv_btm_create(scr,NULL); //创建矩阵按钮
lv_obj_align(btm1,NULL,LV_ALIGN_CENTER,0,0); //矩阵按钮居中对齐
lv_btm_set_map(btm1,btm1_map); //将按钮字符加载进矩阵
lv_obj_set_event_cb(btm1,xzz_event_cb); //创建矩阵按钮事件回调函数

lv_btm_set_ctrl_map(btm1,btm1_ctrl_map); //必须设置控制映射表
lv_btm_set_one_toggle(btm1,true); //根据映射表, 只能有一个按钮被按下
```



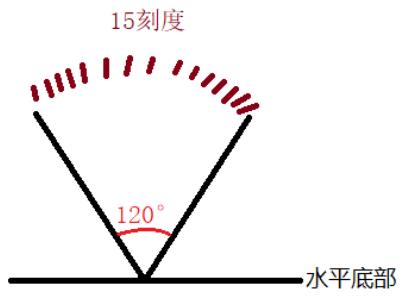
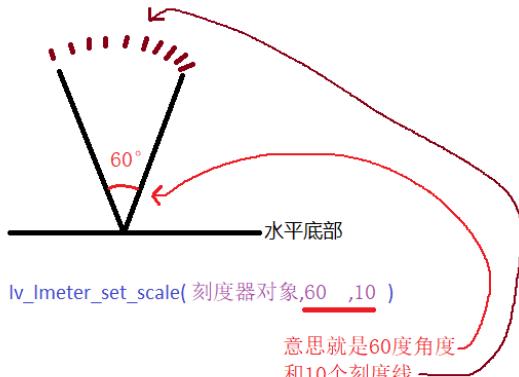
刻度指示器

```
lv_obj_t *刻度器对象 = lv_lmter_create(父窗口,NULL)//创建刻度控件， 默认刻度范围 0~100  
lv_lmter_set_range(刻度器对象,最小刻度,最大刻度) //刻度范围  
lv_lmter_set_value(刻度器对象,刻度当前值) //当前刻度值
```

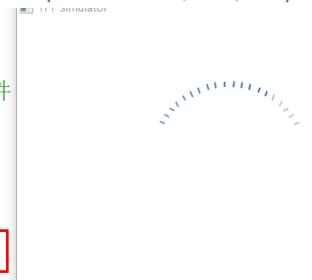
```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *lmeter = lv_lmter_create(scr,NULL); //创建刻度控件  
lv_obj_set_size(lmeter,180,180); //设置刻度控件大小  
lv_obj_align(lmeter,NULL,LV_ALIGN_CENTER,0,0); //居中  
lv_lmter_set_range(lmeter,0,200); //刻度范围  
lv_lmter_set_value(lmeter,150); //当前刻度值
```



```
lv_lmter_set_scale(刻度器对象,开始到结束角度,多少个刻度线)//设置刻度角度和刻度线数量
```



```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *lmeter = lv_lmter_create(scr,NULL); //创建刻度控件  
lv_obj_set_size(lmeter,180,180); //设置刻度控件大小  
lv_obj_align(lmeter,NULL,LV_ALIGN_CENTER,0,0); //居中  
lv_lmter_set_range(lmeter,0,200); //刻度范围  
lv_lmter_set_value(lmeter,150); //当前刻度值  
lv_lmter_set_scale(lmeter,120,20); //设置刻度角度和刻度线数量
```

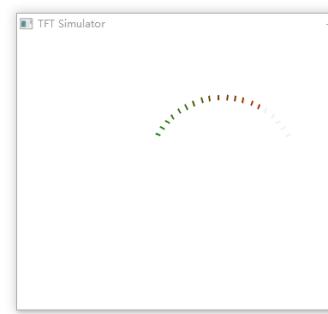


LV_LMETER_STYLE_MAIN 刻度线样式

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t *lmeter = lv_lmter_create(scr,NULL); //创建刻度控件  
lv_obj_set_size(lmeter,180,180); //设置刻度控件大小  
lv_obj_align(lmeter,NULL,LV_ALIGN_CENTER,0,0); //居中  
lv_lmter_set_range(lmeter,0,200); //刻度范围  
lv_lmter_set_value(lmeter,150); //当前刻度值  
lv_lmter_set_scale(lmeter,120,20); //设置刻度角度和刻度线数量
```

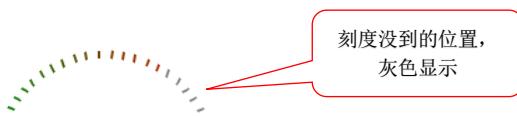
```
static lv_style_t lmeter_style;  
lv_style_copy(&lmeter_style,&lv_style_plain_color);  
lmeter_style.body.main_color = LV_COLOR_GREEN; //刻度线从绿色  
lmeter_style.body.grad_color = LV_COLOR_RED; //渐变到红色  
lv_lmter_set_style(lmeter,LV_LMETER_STYLE_MAIN,&lmeter_style); //刻度线颜色样式
```



```

static lv_style_t lmeter_style;
lv_style_copy(&lmeter_style,&lv_style_plain_color);
lmeter_style.body.main_color = LV_COLOR_GREEN;//刻度线从绿色
lmeter_style.body.grad_color = LV_COLOR_RED; //渐变到红色
lmeter_style.line.color = LV_COLOR_GRAY; //刻度线当前值没有达到的位置用灰色显示
lv_lmter_set_style(lmeter,LV_LMETER_STYLE_MAIN,&lmeter_style); //刻度线颜色样式

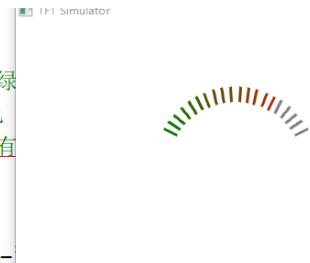
```



```

static lv_style_t lmeter_style;
lv_style_copy(&lmeter_style,&lv_style_plain_color);
lmeter_style.body.main_color = LV_COLOR_GREEN;//刻度线从绿
lmeter_style.body.grad_color = LV_COLOR_RED; //渐变到红色
lmeter_style.line.color = LV_COLOR_GRAY; //刻度线当前值没有
lmeter_style.body.padding.left = 15;//刻度线 加长
lmeter_style.line.width = 3;//刻度线增加宽度
lv_lmter_set_style(lmeter,LV_LMETER_STYLE_MAIN,&lmeter_

```



仪表盘控件

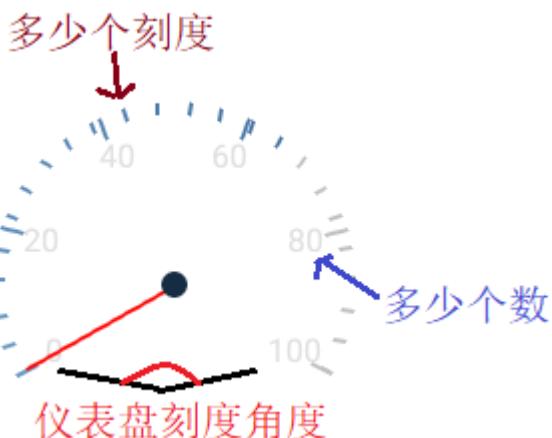
lv_obj_t *仪表盘对象 = lv_gauge_create(父窗口,NULL) //创建指针仪表盘控件
lv_gauge_set_range(仪表盘对象,最小值,最大值) //刻度范围



lv_gauge_set_critical_value(仪表盘对象,关键值 0 到多少) //关键数据点



lv_gauge_set_scale(仪表盘对象,仪表盘刻度角度,多少个刻度,多少个数字显示)//设置刻度线



```

lv_obj_t *scr = lv_scr_act(); // 获取当前活跃的屏幕对象

lv_obj_t *gaugel = lv_gauge_create(scr, NULL); // 创建指针仪表盘控件
lv_obj_set_size(gaugel, 180, 180); // 设置刻度控件大小
lv_obj_align(gaugel, NULL, LV_ALIGN_CENTER, 0, 0); // 居中
lv_gauge_set_range(gaugel, 0, 100); // 刻度范围
lv_gauge_set_critical_value(gaugel, 70); // 关键数据点为 0~70
lv_gauge_set_scale(gaugel, 240, 24, 6); // 设置刻度线

```



设置仪表盘样式，仪表盘样式和刻度线很像

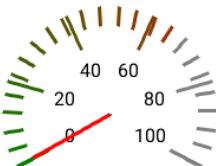
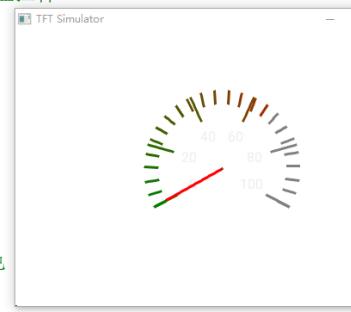
```
lv_obj_t *scr = lv_scr_act(); // 获取当前活跃的屏幕对象
```

```

lv_obj_t *gaugel = lv_gauge_create(scr, NULL); // 创建指针仪表盘控件
lv_obj_set_size(gaugel, 180, 180); // 设置刻度控件大小
lv_obj_align(gaugel, NULL, LV_ALIGN_CENTER, 0, 0); // 居中
lv_gauge_set_range(gaugel, 0, 100); // 刻度范围
lv_gauge_set_critical_value(gaugel, 70); // 关键数据点为 0~70
lv_gauge_set_scale(gaugel, 240, 24, 6); // 设置刻度线

static lv_style_t gaugel_style;
lv_style_copy(&gaugel_style, &lv_style_plain_color);
gaugel_style.body.main_color = LV_COLOR_GREEN; // 刻度线从绿色渐变到红色
gaugel_style.body.grad_color = LV_COLOR_RED;
gaugel_style.line.color = LV_COLOR_GRAY; // 刻度线当前值没有达到的位置用灰色显示
gaugel_style.body.padding.left = 15; // 刻度线加长
gaugel_style.line.width = 3; // 刻度线增加宽度
lv_gauge_set_style(gaugel, LV_GAUGE_STYLE_MAIN, &gaugel_style); // 仪表盘颜色样式
gaugel_style.text.color = LV_COLOR_BLACK; // 数字文本改成黑色

```



lv_gauge_set_needle_count(仪表盘对象, 指针数量, 每根指针颜色数组) // 创建多根指针

```
const lv_color_t colors[] = {LV_COLOR_GREEN, LV_COLOR_RED}; // 设置两根指针颜色
```

```
lv_obj_t *scr = lv_scr_act(); // 获取当前活跃的屏幕对象
```

```

lv_obj_t *gaugel = lv_gauge_create(scr, NULL); // 创建指针仪表盘控件
lv_obj_set_size(gaugel, 180, 180); // 设置刻度控件大小
lv_obj_align(gaugel, NULL, LV_ALIGN_CENTER, 0, 0); // 居中
lv_gauge_set_range(gaugel, 0, 100); // 刻度范围
lv_gauge_set_critical_value(gaugel, 70); // 关键数据点为 0~70
lv_gauge_set_scale(gaugel, 240, 24, 6); // 设置刻度线
lv_gauge_set_needle_count(gaugel, 2, colors); // 创建两根指针

```



lv_gauge_set_value(仪表盘对象, 指针 ID 号, 指针值) // 设置指针旋转到某个数值

指针 ID 号: 根据颜色数组决定指针 ID

```
const lv_color_t colors[] = {LV_COLOR_GREEN, LV_COLOR_RED}; // 设置两根指针颜色
指针值: 旋转到某个值
```

```
lv_gauge_set_value(gaugel, 1, 80); // 设置指针旋转到某个数值
```



红色这个 ID 的指针移动到了 80.

红色指针是数组下标 1,
所以指针 ID 号为 1

消息对话框使用

```
lv_obj_t *消息对话框对象 = lv_mbox_create(父窗口对象,NULL) //创建消息对话框，默认处于  
打开状态  
lv_mbox_set_text(消息对话框对象,传入字符串) //消息对话框显示字符
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *mbox = lv_mbox_create(scr,NULL); //创建消息对话框，默认处于打开状态  
lv_obj_set_width(mbox,220); //消息对话框宽度  
lv_mbox_set_text(mbox,"hello world"); //消息对话框显示字符  
lv_obj_align(mbox,NULL,LV_ALIGN_CENTER,0,0); //消息对话框居中
```



lv_mbox_add_btns(消息对话框对象, 传入字符串数组) //向消息对话框添加按钮控件
传入字符串数组：就是用定义字符串的方式，定义多个按钮。(和前面矩阵按钮类型)

```
const char *btns_map[] = {"cancel","OK","",""}; //添加cancel按钮和OK按钮，必须用""结尾  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *mbox = lv_mbox_create(scr,NULL); //创建消息对话框，默认处于打开状态  
lv_obj_set_width(mbox,220); //消息对话框宽度  
lv_mbox_set_text(mbox,"hello world"); //消息对话框显示字符  
lv_obj_align(mbox,NULL,LV_ALIGN_CENTER,0,0); //消息对话框居中  
lv_mbox_add_btns(mbox,btns_map); //向消息对话框添加按钮控件
```



消息对话框，增加标题

```
lv_mbox_set_text(mbox,"Tip\nhello world"); //消息对话框在已有字符前增加标题，用"\n"换行
```



下面给标题增加颜色

因为消息对话框本身是没有修改标题颜色功能的，所以我们必须用 label 标签间接去修改

```
/*  
* *mbox: 传入自定义的消息对话框控件  
* en: 是否使能修改对话框颜色  
*/  
void mbox_set_msg_recolor(lv_obj_t *mbox,bool en)  
{  
    lv_mbox_ext_t *mbox_ext = lv_obj_get_ext_attr(mbox); //获取消息对话框扩展属性  
    lv_label_set_recolor(mbox_ext->text,en); //mbox_ext->text就是消息对话框的标签  
    //使用lv_label_set_recolor 使其标签对象使能，这样就可以修改标签颜色了  
}
```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *mbox = lv_mbox_create(scr, NULL); //创建消息对话框，默认处于打开状态
lv_obj_set_width(mbox, 220); //消息对话框宽度
mbox_set_msg_recolor(mbox, true); //在设置对话框之前调用使能标签函数
lv_mbox_set_text(mbox, "#ff0000 Tip\nhello world"); //消息对话框在已有字符前增加标题，用"\n"换行
lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0); //消息对话框居中

```

在字符串前面用#号就是给字符加颜色，这儿用#ff0000 就是给字符 Tip 加红色。



lv_mbox_set_anim_time(消息对话框对象,动画延时时间毫秒为单位) //消息对话框关闭时间设置，这个 API 在老版本起作用，在新版本只能达到快速关闭对话框效果，设置多少 ms 延时都没有用

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *mbox = lv_mbox_create(scr, NULL); //创建消息对话框，默认处于打开状态
lv_obj_set_width(mbox, 220); //消息对话框宽度
mbox_set_msg_recolor(mbox, true); //在设置对话框之前调用使能标签函数
lv_mbox_set_text(mbox, "#ff0000 Tip\nhello world"); //消息对话框在已有字符前增
lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0); //消息对话框居中
lv_mbox_add_btns(mbox, btns_map); //向消息对话框添加按钮控件
lv_mbox_set_anim_time(mbox, 3000); //消息对话框关闭加快

```

消息对话框事件回调函数实现

uint16_t 返回值 = lv_mbox_get_active_btn(消息对话框对象) //获取消息对话框哪一个按钮被按下

返回值：根据消息对话框某一个按钮按下，返回对应的按钮 ID，

const char *btns_map[] = {"cancel", "OK", ""}; //添加cancel按钮和OK按钮，必须用""结尾
按钮 ID 来自于字符串数组下标

```

lv_obj_t *mbox; //消息对话框对象改为全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == mbox)
    {
        if(event == LV_EVENT_VALUE_CHANGED) //按钮按下触发
        {
            printf("key id = %d\n", lv_mbox_get_active_btn(mbox)); //获取哪个按钮被按下
        }
    }
}

```

const char *btns_map[] = {"cancel", "OK", ""}; //添加cancel按钮和OK按钮，必须用""结尾

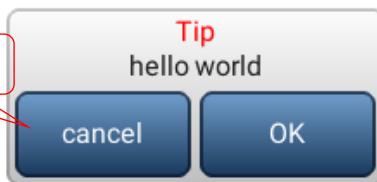
```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
mbox = lv_mbox_create(scr, NULL); //创建消息对话框，默认处于打开状态
lv_obj_set_width(mbox, 220); //消息对话框宽度
mbox_set_msg_recolor(mbox, true); //在设置对话框之前调用使能标签函数
lv_mbox_set_text(mbox, "#ff0000 Tip\nhello world"); //消息对话框在已有字符前增加标题，用"\n"换行
lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0); //消息对话框居中
lv_mbox_add_btns(mbox, btns_map); //向消息对话框添加按钮控件
lv_obj_set_event_cb(mbox, xzz_event_cb); //增加消息对话框按钮点击事件回调函数

```

```
key id = 0
```

点击 cancel 返回 0



```
key id = 1
```

Tip
hello world

cancel OK

点击 OK 返回 1

证明按钮事件获取成功，但是加入事件回调之后，消息对话框不消失了？

`lv_obj_set_event_cb(mbox, xzz_event_cb); //增加消息对话框按钮点击事件回调函数`
这是因为手动写入的事件回调函数会导致消息对话框默认的事件回调函数失效。

所以我们只能在事件回调函数里面手动关闭消息对话框

```
lv_obj_t *mbox; //消息对话框对象改为全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == mbox)
    {
        if(event == LV_EVENT_VALUE_CHANGED) //按钮按下触发
        {
            printf("key id = %d\n", lv_mbox_get_active_btn(mbox)); //获取哪个按钮被按下
            lv_obj_del(mbox); //用删除对象的方式关闭消息对话框
        }
    }
}
```

也可以使用延时关闭对话框

```
lv_mbox_start_auto_close(消息对话框对象, 延时时间(毫秒)) //延时关闭消息对话框
lv_obj_t *mbox; //消息对话框对象改为全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == mbox)
    {
        if(event == LV_EVENT_VALUE_CHANGED) //按钮按下触发
        {
            printf("key id = %d\n", lv_mbox_get_active_btn(mbox)); //获取哪个按钮被按下
            //lv_obj_del(mbox); //用删除对象的方式关闭消息对话框
            lv_mbox_start_auto_close(mbox, 2000); //延时关闭, 2秒之后关闭
        }
    }
}
```

点击按钮，关闭消息对话框成功。

消息对话框样式设置

`lv_obj_t * 消息内部按钮对象 = lv_mbox_get_btmn(消息对话框对象) //获取对话框内部矩阵按钮对象`



获取按钮对象，单独修改尺寸

```

const char *bt�s_map[] = {"cancel", "\n", "OK", ""}; //添加 cancel 按钮, 必须用""结尾

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
mbox = lv_mbox_create(scr, NULL); //创建消息对话框, 默认处于打开状态
lv_obj_set_width(mbox, 220); //消息对话框宽度
lv_obj_set_msg_recolor(mbox, true); //在设置对话框之前调用使能标签函数
lv_mbox_set_text(mbox, "#ff0000 Tip\nhello world"); //消息对话框在已有字符前增加标题, 用"\n"换行
lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0); //消息对话框居中
lv_mbox_add_btns(mbox, bt�s_map); //向消息对话框添加按钮控件
lv_obj_set_event_cb(mbox, xzz_event_cb); //增加消息对话框按钮点击事件回调函数

static lv_style_t bg_style; //消息对话框背景样式
lv_style_copy(&bg_style, &lv_style_plain_color); //注意拷贝的样式类型
bg_style.body.main_color = LV_COLOR_MAKE(250, 250, 250); //背景色上半部
bg_style.body.grad_color = LV_COLOR_MAKE(250, 250, 250); //背景色下半部
bg_style.body.radius = 10; //圆角半径
bg_style.body.border.width = 1; //边框宽度
bg_style.body.border.color = LV_COLOR_MAKE(150, 150, 150); //边框颜色
bg_style.body.shadow.color = LV_COLOR_MAKE(150, 150, 150); //阴影颜色
bg_style.body.shadow.width = 6; //阴影宽度
bg_style.body.padding.top = 10; //内部消息内容与消息对话框上边框之间距离
bg_style.body.padding.bottom = 0; //内部矩阵按钮与消息对话框底边之间距离
bg_style.body.padding.inner = 10; //消息内容与矩阵按钮之间距离
bg_style.text.color = LV_COLOR_BLACK; //消息文本颜色
lv_mbox_set_style(mbox, LV_MBOX_STYLE_BG, &bg_style); //消息对话框背景颜色样式

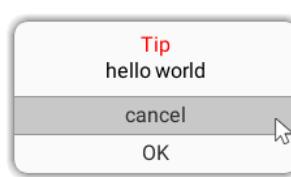
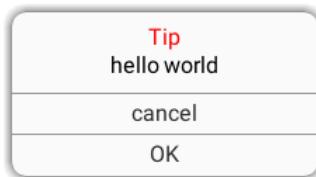
static lv_style_t btn_bg_style; //消息对话框按钮背景样式
lv_style_copy(&btn_bg_style, &lv_style_transp_tight); //注意拷贝的样式类型
btn_bg_style.body.padding.top = 0;
btn_bg_style.body.padding.left = 0;
btn_bg_style.body.padding.right = 0;
btn_bg_style.body.padding.bottom = 0;
btn_bg_style.body.padding.inner = 0;
btn_bg_style.body.radius = 10; //圆角半径
lv_mbox_set_style(mbox, LV_MBOX_STYLE_BTN_BG, &btn_bg_style); //消息对话框按钮背景样式

static lv_style_t btn_rel_style; //消息对话框按钮释放时的样式
lv_style_copy(&btn_rel_style, &lv_style_transp); //注意拷贝的样式类型
btn_rel_style.body.border.part = LV_BORDER_TOP; //只绘制上边框
btn_rel_style.body.border.width = 1; //边框宽度
btn_rel_style.body.border.color = LV_COLOR_MAKE(150, 150, 150); //边框颜色
lv_mbox_set_style(mbox, LV_MBOX_STYLE_BTN_REL, &btn_rel_style); //消息对话框按钮释放时的样式

static lv_style_t btn_press_style; //消息对话框按钮按下时的样式
lv_style_copy(&btn_press_style, &btn_rel_style); //注意拷贝的样式类型
btn_press_style.body.opa = LV_OPA_COVER; //完全不透明
btn_press_style.body.border.part = LV_BORDER_FULL; //绘制四边
btn_press_style.body.main_color = LV_COLOR_MAKE(200, 200, 200); //背景色
btn_press_style.body.grad_color = LV_COLOR_MAKE(200, 200, 200); //背景色
lv_mbox_set_style(mbox, LV_MBOX_STYLE_BTN_PR, &btn_press_style); //消息对话框按钮按下时的样式

/*这段代码一定要执行在按钮创建, 和样式设置之后*/
lv_obj_t * btn_of_mbox = lv_mbox_get_btm(mbox); //获取对话框内部矩阵按钮对象
lv_obj_set_size(btn_of_mbox, 220, 60); //设置矩阵按钮大小

```



页面控件

```
lv_obj_t *页面对象 = lv_page_create(父窗口,NULL) //创建页面
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *page = lv_page_create(scr, NULL); //创建页面  
lv_obj_set_size(page, 200, 200); //设置页面大小  
lv_obj_align(page, NULL, LV_ALIGN_CENTER, 0, 0); //页面居中
```



int16_t 返回值 = lv_page_get_fit_width(页面对象) //获取页面的可填区域有多宽

```
lv_obj_t *scr = lv_scr_act(); // 获取当前活跃的屏幕对象  
lv_obj_t *page = lv_page_create(scr, NULL); // 创建页面  
lv_obj_set_size(page, 200, 200); // 设置页面大小  
lv_obj_align(page, NULL, LV_ALIGN_CENTER, 0, 0); // 页面居中
```

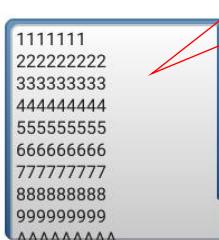
```
lv_obj_t *label1 = lv_label_create(page,NULL); //创建标签，将标签放入页面  
lv_label_set_long_mode(label1,LV_LABEL_LONG_BREAK); //设置标签为长文本模式  
lv_obj_set_width(label1,lv_page_get_fit_width(page)); //得到页面填充区域宽度  
lv_label_set_text(label1,"11111111\  
222222222\  
333333333"); //向页面写入多行文本
```



页面内容太少，无法进行拖动

字符行数增多，页面可以拖动，鼠标点击空白处拖动页面

```
lv_label_set_text(label1,"11111111\  
222222222\  
333333333\  
444444444\  
555555555\  
666666666\  
777777777\  
888888888\  
999999999\  
AAAAAAAABBBB BBBB"://向页面写入多行文本
```



```
lv page set sb mode(页面对象,参数) //设置页面模式
```

参数: LV SB MODE OFF //取消页面滚动条

LV SB MODE DRAG // 拖拽页面才会出现滚动条

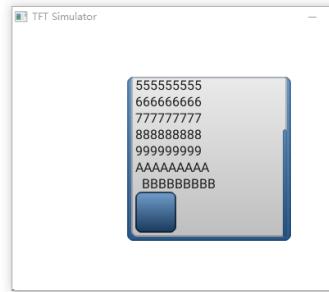
```
lv_obj_set_width(label1,lv_page_get_fit_width(page));//得到页面填充区域宽度,  
lv_page_set_sb_mode(page,LV_SB_MODE_OFF); //取消页面滚动条  
lv_label_set_text(label1,"1111111\
```



页面右边没有蓝色的杠杠

向页面放入按钮

```
lv_page_set_sb_mode(page, LV_SB_MODE_DRAG); //取消页面滚动条  
lv_label_set_text(label1, "1111111\\  
222222222\\  
333333333\\  
444444444\\  
555555555\\  
666666666\\  
777777777\\  
888888888\\  
999999999\\  
AAAAAAA\\  
BBBBBBB"); //向页面写入多行文本  
lv_obj_t *btn2 = lv_btn_create(page, NULL); //向页面放入按钮  
lv_obj_set_size(btn2, 50, 50);  
lv_obj_align(btn2, label1, LV_ALIGN_OUT_BOTTOM_LEFT, 0, 0); //将按钮放在相对于标签的左下角
```



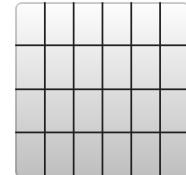
```
lv_page_set_edge_flash(页面对象, true); //页面滑动到顶部/底部出现下方阴影  
lv_page_set_sb_mode(page, LV_SB_MODE_DRAG); //取消页面滚动条  
lv_page_set_edge_flash(page, true); //页面滑动到底部出现下方阴影
```



图表控件使用，设置曲线，柱状图

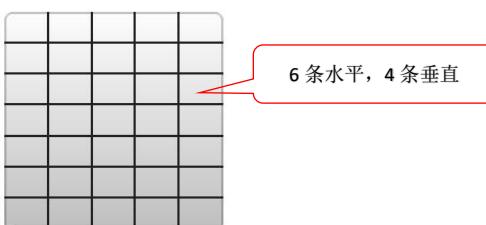
```
lv_obj_t *图表对象 = lv_chart_create(父窗口, NULL); //创建图表
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *chart1 = lv_chart_create(scr, NULL);  
lv_obj_set_size(chart1, 200, 200); //设置图标大小  
lv_obj_align(chart1, NULL, LV_ALIGN_CENTER, 0, 0); //图表居中
```



```
lv_chart_set_div_line_count(图表对象, 水平分割线多少条, 垂直分割线多少条); //设置水平分割线和垂直分割线条数
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *chart1 = lv_chart_create(scr, NULL);  
lv_obj_set_size(chart1, 200, 200); //设置图标大小  
lv_obj_align(chart1, NULL, LV_ALIGN_CENTER, 0, 0); //图表居中  
lv_chart_set_div_line_count(chart1, 6, 4); //设置水平分割线和垂直分割线条数
```



```
lv_chart_series_t *数据线对象 = lv_chart_add_series(图表对象, 指定添加的数据线颜色); //添加数据线
```

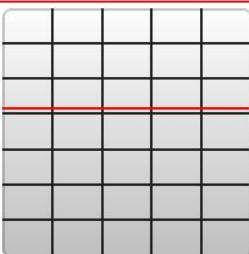
```
lv_chart_init_points(图表对象, 数据线对象, 值); //设置数据线每个点的值
```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *chart1 = lv_chart_create(scr, NULL);
lv_obj_set_size(chart1, 200, 200); //设置图标大小
lv_obj_align(chart1, NULL, LV_ALIGN_CENTER, 0, 0); //图表居中
lv_chart_set_div_line_count(chart1, 6, 4); //设置水平分割线和垂直分割线条数

lv_chart_series_t *series1 = lv_chart_add_series(chart1, LV_COLOR_RED); //添加数据线
lv_chart_init_points(chart1, series1, 60); //设置数据线每个点的值, 我这里将每个点设置成60

```



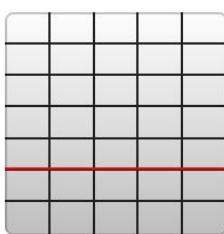
这个根线所有点都在 60 位置
为什么这个位置是 60?
因为图表控件 Y 轴默认是 0~100

```
lv_chart_set_range(图表对象,y 轴最小值,y 轴最大值) //设置 y 轴高度
```

```

lv_chart_set_range(chart1, 0, 200); //设置y轴高度
lv_chart_series_t *series1 = lv_chart_add_series(chart1, LV_COLOR_RED); //添加数据线
lv_chart_init_points(chart1, series1, 60); //设置数据线每个点的值, 我这里将每个点设置成60

```



Y 轴是 0~200 个数据宽度, 那么 60 的
位置就要低很多

lv_chart_set_points(图表对象,数据线对象,数据点数组) //设置多个数据点, 图表默认只支持
10 个数据点

```

lv_coord_t points_val[] = {10, 50, 100, 120, 160, 140, 100, 80, 60, 30}; //定义曲线10个数据点

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *chart1 = lv_chart_create(scr, NULL);
lv_obj_set_size(chart1, 200, 200); //设置图标大小
lv_obj_align(chart1, NULL, LV_ALIGN_CENTER, 0, 0); //图表居中
lv_chart_set_div_line_count(chart1, 6, 4); //设置水平分割线和垂直分割线条数

```

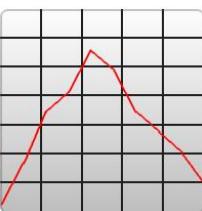
先定义 10 个数据点

```

lv_chart_set_range(chart1, 0, 200); //设置y轴高度
lv_chart_series_t *series1 = lv_chart_add_series(chart1, LV_COLOR_RED); //添加数据线
//lv_chart_init_points(chart1, series1, 60); //设置数据线每个点的值, 我这里将每个点设置成60
lv_chart_set_points(chart1, series1, points_val); //设置多个数据点

```

将 10 个数据点放入函数



曲线显示成功

lv_chart_set_type(图表对象,图表类型) //设置图表类型, 设置散点图

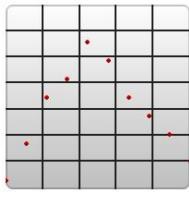
图表类型: LV_CHART_TYPE_POINT //散点图

LV_CHART_TYPE_COLUMN //柱状图

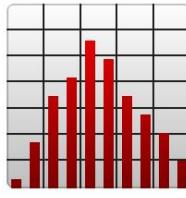
LV_CHART_TYPE_AREA //面积图

在显示数据之前可以设置图表类型

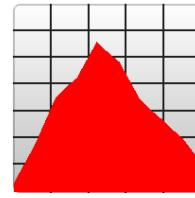
```
lv_chart_set_type(chart1, LV_CHART_TYPE_POINT); //设置图表类型, 设置散点图  
lv_chart_series_t *series1 = lv_chart_add_series(chart1, LV_COLOR_RED); //添加数据线  
lv_chart_set_points(chart1, series1, points_val); //设置多个数据点
```



散点图

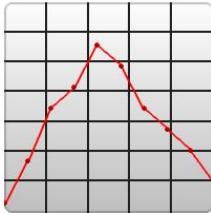


柱状图



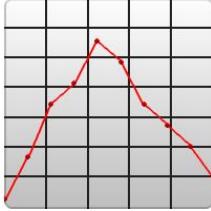
面积图

```
lv_chart_set_type(chart1, LV_CHART_TYPE_POINT | LV_CHART_TYPE_LINE); //设置图表类型, 设置散点图和折线组合
```

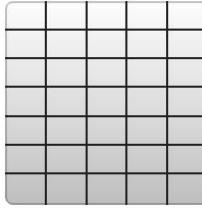


散点和折线都显示

```
lv_chart_clear_serie(图表对象, 数据线对象); //清除图表里面的数据  
lv_chart_refresh(图表对象); //刷新图表, 为了保险起见, 更新曲线时, 使用刷新图表函数  
lv_chart_set_points(chart1, series1, points_val); //设置多个数据点  
lv_chart_clear_serie(chart1, series1); //清除图表里面的曲线  
lv_chart_refresh(chart1); //刷新图表, 也可以不加, 注意BUG
```



清除前



清除后

```
lv_chart_set_point_count(图表对象, 数据点个数); //设置图表数据点个数(重要)
```

```
lv_coord_t points_val[] = {10, 50, 100, 120, 160, 140, 100, 80, 60, 30,  
15, 70, 80, 90, 50}; //定义曲线15个数据点
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *chart1 = lv_chart_create(scr, NULL);  
lv_obj_set_size(chart1, 200, 200); //设置图标大小  
lv_obj_align(chart1, NULL, LV_ALIGN_CENTER, 0, 0); //图表居中  
lv_chart_set_div_line_count(chart1, 6, 4); //设置水平分割线和垂直分割线条数
```

```
lv_chart_set_range(chart1, 0, 200); //设置y轴高度  
lv_chart_set_point_count(chart1, 15); //设置图表数据点个数, 本来默认是10个, 我现在改成15  
lv_chart_set_type(chart1, LV_CHART_TYPE_POINT | LV_CHART_TYPE_LINE); //设置图表类型,  
lv_chart_series_t *series1 = lv_chart_add_series(chart1, LV_COLOR_RED); //添加数据  
lv_chart_set_points(chart1, series1, points_val); //设置多个数据点
```

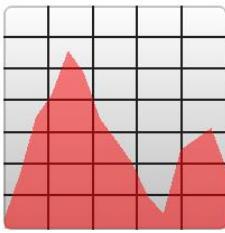


曲线显示 15 个数据点

```

lv_chart_set_series_opa(图表对象,透明度) //设置面积图透明度
透明度: LV_OPA_50 //50%透明
lv_chart_set_range(chart1,0,200); //设置y轴高度
lv_chart_set_point_count(chart1,15); //设置图表数据点个数, 本来默认是10个, 我现在改成15个
lv_chart_set_series_opa(chart1,LV_OPA_50); //透明度50%
lv_chart_set_type(chart1,LV_CHART_TYPE_AREA); //设置图表类型, 面积
lv_chart_series_t *series1 = lv_chart_add_series(chart1,LV_COLOR_RED); //添加数据线
lv_chart_set_points(chart1,series1,points_val); //设置多个数据点

```



透明图表

lv_chart_set_series_width(图表对象,宽度值) //设置曲线宽度

```

lv_chart_set_range(chart1,0,200); //设置y轴高度
lv_chart_set_point_count(chart1,15); //设置图表数据点个数, 本来默认是10个, 我现在改成15个
lv_chart_set_series_width(chart1,4); //设置曲线宽度 4
lv_chart_set_type(chart1,LV_CHART_TYPE_LINE); //设置图表类型, 线
lv_chart_series_t *series1 = lv_chart_add_series(chart1,LV_COLOR_RED); //添加数据线
lv_chart_set_points(chart1,series1,points_val); //设置多个数据点

```



曲线加宽

lv_chart_set_series_drawing(图表对象,阴影值) //设置曲线黑阴影效果

数据线更新模式

lv_chart_set_update_mode(图表对象,模式) //数据更新模式方式

模式: LV_CHART_UPDATE_MODE_SHIFT //左平移方式增加数据点
 LV_CHART_UPDATE_MODE_CIRCULAR //环形覆盖方式增加数据点

lv_chart_set_next(图表对象, 数据线对象,单个数据) //向现有曲线最后一个点新增单个数据

```

lv_coord_t points_val[] = {10,50,100,120,160,140,100,80,60,30,
                           15,70,80,90,50}; //定义曲线15个数据点

```

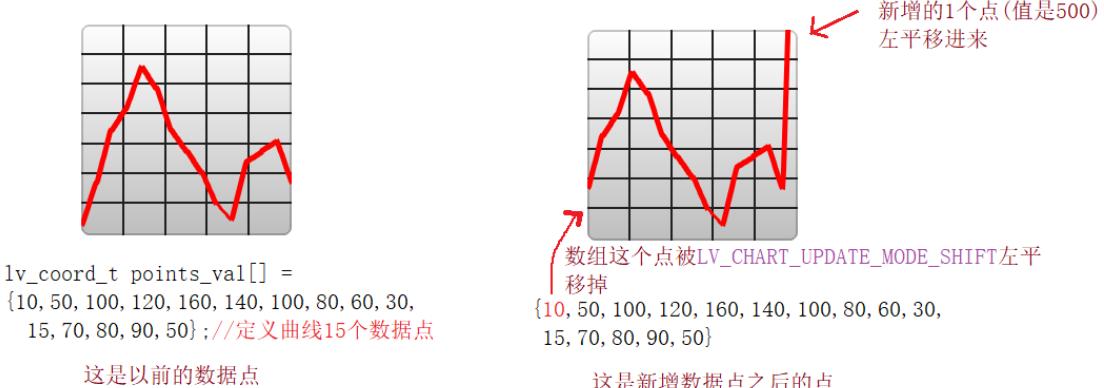


这是现有数组的数据点

```

lv_chart_set_range(chart1,0,200); //设置y轴高度
lv_chart_set_point_count(chart1,15); //设置图表数据点个数, 本来默认是10个, 我现在改成15个
lv_chart_set_series_width(chart1,4); //设置曲线宽度 4
lv_chart_set_series_darking(chart1,300); //设置曲线黑阴影效果
lv_chart_set_type(chart1,LV_CHART_TYPE_LINE); //设置图表类型, 线
lv_chart_series_t *series1 = lv_chart_add_series(chart1,LV_COLOR_RED); //添加数据线
lv_chart_set_points(chart1,series1,points_val); //设置多个数据点
lv_chart_set_update_mode(chart1,LV_CHART_UPDATE_MODE_SHIFT); //数据更新模式, 左平移模式
lv_chart_set_next(chart1,series1,500); //向现有曲线最后一个点添加数据

```

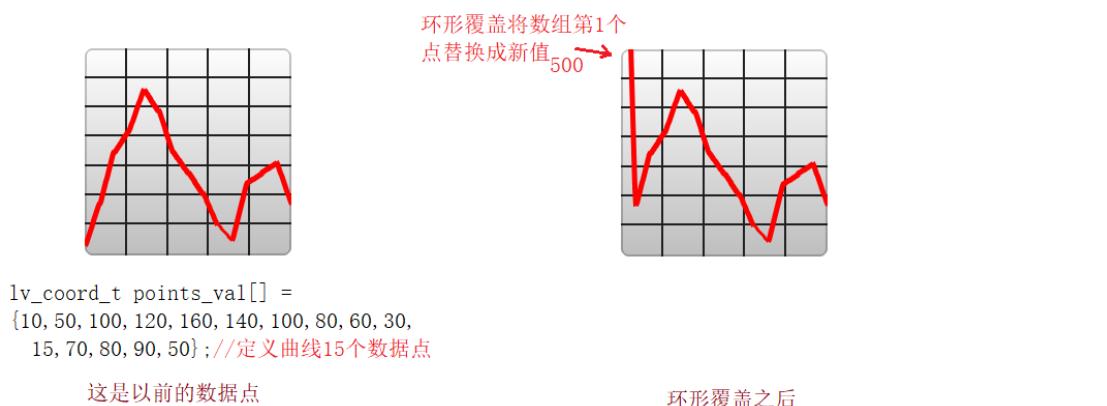


LV_CHART_UPDATE_MODE_CIRCULAR 环形覆盖

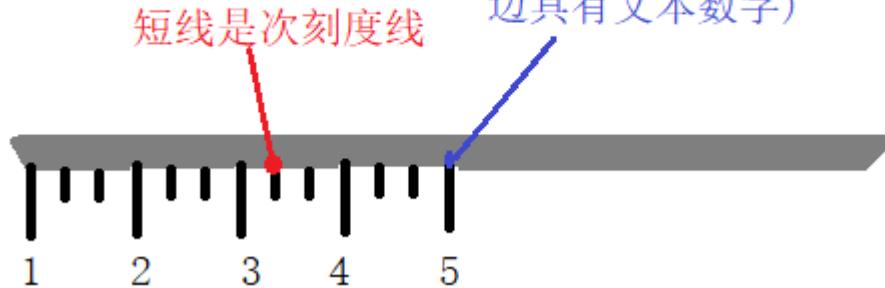
```

lv_chart_set_update_mode(chart1,LV_CHART_UPDATE_MODE_CIRCULAR); //数据更新模式, 环形覆盖模式
lv_chart_set_next(chart1,series1,500); //向现有曲线最后一个点添加数据

```



刻度线设置



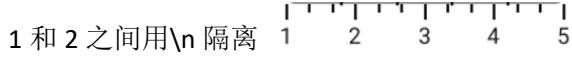
```
lv_chart_set_x_tick_length(图表对象, 主刻度线长度, 次刻度线长度) //设置刻度线 x 方向长度
```

```
lv_chart_set_x_tick_texts(图表对象, 字符串, 两个主刻度线之间多少个次刻度线, 是否绘制最后刻度线) //刻度线加标题
```

字符串: 字符串就是主刻度线下边标题, 标题之间用\n 来隔离

```
"1\n2\n3\n4\n5"
```

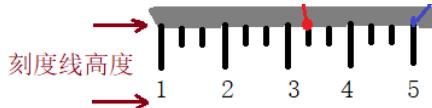
1 和 2 之间用\n 隔离



两个主刻度线之间多少个次刻度线: 填入次刻度线数量

是否绘制最后刻度线: LV_CHART_AXIS_DRAW_LAST_TICK

```
lv_chart_set_margin(图表对象, 刻度线高度) //显示刻度线高度, 一定要执行, 不然刻度线不显示
```



```
lv_chart_set_range(chart1, 0, 200); //设置y轴高度  
lv_chart_set_point_count(chart1, 15); //设置图表数据点个数, 本来默认是10个, 我现在改成15个  
lv_chart_set_series_width(chart1, 4); //设置曲线宽度 4  
lv_chart_set_series_darking(chart1, 300); //设置曲线黑阴影效果  
lv_chart_set_type(chart1, LV_CHART_TYPE_LINE); //设置图表类型, 线  
lv_chart_series_t *series1 = lv_chart_add_series(chart1, LV_COLOR_RED); //添加数据线  
lv_chart_set_points(chart1, series1, points_val); //设置多个数据点  
lv_chart_set_x_tick_length(chart1, 10, 5); //设置刻度线x方向长度  
lv_chart_set_x_tick_texts(chart1, "1\n2\n3\n4\n5", 3, LV_CHART_AXIS_DRAW_LAST_TICK); //刻度线加标题  
lv_chart_set_margin(chart1, 30); //显示刻度线高度, 一定要执行, 不然刻度线不显示
```



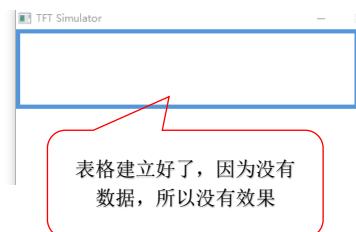
表格控件

```
lv_obj_t *表格对象 = lv_table_create(父窗口, NULL) //创建表格
```

```
lv_table_set_row_cnt(表格对象, 行数) //行数设置
```

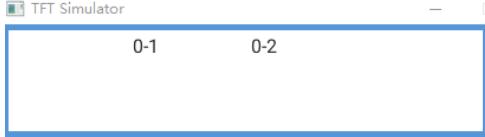
```
lv_table_set_col_cnt(表格对象, 列数) //列数设置
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *tabel = lv_table_create(scr, NULL); //创建表格  
lv_table_set_row_cnt(tabel, 3); //行数设置  
lv_table_set_col_cnt(tabel, 4); //列数设置
```



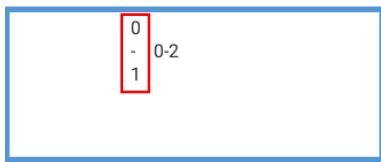
```
lv_table_set_cell_value(表格对象,行,列,字符串) //向某行某列写入数据(字符串写入)
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *tabel = lv_table_create(scr, NULL); //创建表格  
lv_table_set_row_cnt(tabel, 3); //行数设置  
lv_table_set_col_cnt(tabel, 4); //列数设置  
lv_table_set_cell_value(tabel, 0, 1, "0-1"); //向0行1列写入数据  
lv_table_set_cell_value(tabel, 0, 2, "0-2"); //向0行2列写入数据
```



lv_table_set_col_width(表格对象,列,宽度值) //设置某列宽度

```
lv_table_set_cell_value(tabel, 0, 1, "0-1"); //向0行1列写入数据  
lv_table_set_cell_value(tabel, 0, 2, "0-2"); //向0行2列写入数据  
lv_table_set_col_width(tabel, 1, 20); //设置第1列宽度20
```

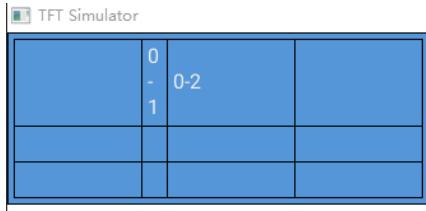


20 的宽度其实很窄

我们发现表格没有分割线，表格分割线需要样式设置

表格样式设置

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *tabel = lv_table_create(scr, NULL); //创建表格  
lv_table_set_row_cnt(tabel, 3); //行数设置  
lv_table_set_col_cnt(tabel, 4); //列数设置  
lv_table_set_cell_value(tabel, 0, 1, "0-1"); //向0行1列写入数据  
lv_table_set_cell_value(tabel, 0, 2, "0-2"); //向0行2列写入数据  
lv_table_set_col_width(tabel, 1, 20); //设置第1列宽度20  
  
static lv_style_t cell_style;  
lv_style_copy(&cell_style, &lv_style_plain_color);  
cell_style.body.border.width = 1; //背景边框宽度  
cell_style.body.border.color = LV_COLOR_BLACK; //背景边框颜色  
lv_table_set_style(tabel, LV_TABLE_STYLE_BG, &cell_style);  
  
static lv_style_t cellunit_style;  
lv_style_copy(&cellunit_style, &lv_style_plain_color);  
cellunit_style.body.border.width = 1; //单元格边框宽度  
cellunit_style.body.border.color = LV_COLOR_BLACK; //单元格背景颜色  
lv_table_set_style(tabel, LV_TABLE_STYLE_CELL1, &cellunit_style);
```



//所有单元格默认 LV_TABLE_STYLE_CELL1

`lv_table_set_cell_align(表格对象,行,列,对齐方式) //单元格字符对齐方式`
对齐方式: `LV_LABEL_ALIGN_CENTER //居中对齐`

```
lv_table_set_cell_align(table, 0, 2, LV_LABEL_ALIGN_CENTER); //第0行第2列居中
```

```
static lv_style_t cell_style;
lv_style_copy(&cell_style, &lv_style_plain_color);
cell_style.body.border.width = 1; //背景边框宽度
cell_style.body.border.color = LV_COLOR_BLACK; //背景边框颜色
lv_table_set_style(table, LV_TABLE_STYLE_BG, &cell_style);

static lv_style_t cellunit_style;
lv_style_copy(&cellunit_style, &lv_style_plain_color);
cellunit_style.body.border.width = 1; //单元格边框宽度
```

TFT Simulator

	0 - 1	0-2	

居中前

TFT Simulator

	0 - 1	0-2	

居中后

`lv_table_set_cell_type(表格对象,行,列,CELL 编号) //将某个单元格单独取编号, 编译样式单独设置`

```
lv_table_set_cell_value(table, 0, 1, "0-1"); //向0行1列写入数据
lv_table_set_cell_value(table, 0, 2, "0-2"); //向0行2列写入数据
lv_table_set_col_width(table, 1, 20); //设置第1列宽度20
```

```
lv_table_set_cell_align(table, 0, 2, LV_LABEL_ALIGN_CENTER); //第0行第2列居中
lv_table_set_cell_type(table, 0, 1, 2); //将第0行第1列表格设置为CELL2
```

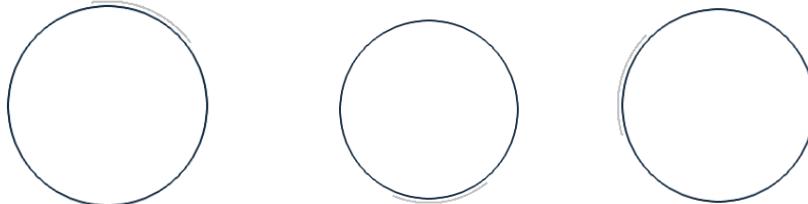
```
static lv_style_t cell2_style;
lv_style_copy(&cell2_style, &lv_style_plain);
cell2_style.body.border.width = 1; //CELL2单元格边框宽度
cell2_style.body.border.color = LV_COLOR_BLACK; //CELL2单元格边框背景色
cell2_style.body.main_color = LV_COLOR_SILVER; //CELL2单元格
cell2_style.body.grad_color = LV_COLOR_SILVER;
lv_table_set_style(table, LV_TABLE_STYLE_CELL2, &cell2_style);
```

TFT Simulator

	0 - 1	0-2	

预加载控件使用(类似 windows 鼠标选转等待)

```
lv_obj_t *preload = lv_preload_create(scr, NULL); // 创建预加载控件  
lv_obj_set_size(preload, 200, 200); // 控件大小  
lv_obj_align(preload, NULL, LV_ALIGN_CENTER, 0, 0); // 居中
```

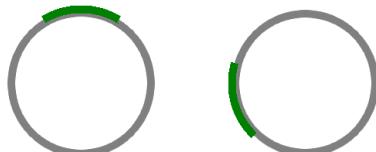


这就是旋转效果

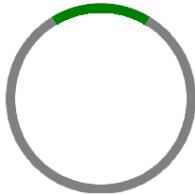
预加载样式设置

```
lv_obj_t *scr = lv_scr_act(); // 获取当前活跃的屏幕对象  
lv_obj_t *preload = lv_preload_create(scr, NULL); // 创建预加载控件  
lv_obj_set_size(preload, 200, 200); // 控件大小  
lv_obj_align(preload, NULL, LV_ALIGN_CENTER, 0, 0); // 居中
```

```
static lv_style_t bg_style; // 背景圆环样式设置  
lv_style_copy(&bg_style, &lv_style_plain_color);  
bg_style.body.border.color = LV_COLOR_GRAY; // 圆环颜色  
bg_style.body.border.width = 10; // 圆环宽度  
bg_style.line.color = LV_COLOR_GREEN; // 设置小圆弧颜色  
bg_style.line.width = 10; // 小圆弧宽度  
lv_preload_set_style(preload, LV_PRELOAD_STYLE_MAIN, &bg_style);
```



bg_style.body.padding.left = 0; // 让小圆弧和圆环重合在一起



lv_preload_set_spin_time(预加载对象, 毫秒) // 设置小圆弧旋转时间

```
lv_obj_set_size(preload, 200, 200); // 控件大小  
lv_obj_align(preload, NULL, LV_ALIGN_CENTER, 0, 0); // 居中  
lv_preload_set_spin_time(preload, 3000); // 选转动画一圈为3秒
```

旋转很慢

lv_preload_set_dir(预加载对象, LV_PRELOAD_DIR_BACKWARD) // 选择动画逆时针旋转

TAB 选项卡控件使用

```
lv_obj_t *选项卡对象 = lv_tabview_create(父窗口,NULL) //创建选项卡  
注意: 创建选项卡之后, 不会有内容显示, 必须用 lv_tabview_add_tab 给选项卡加 TAB 页才有内容显示。  
lv_obj_t *Tab 页对象 = lv_tabview_add_tab(选项卡对象,TAB 页名称字符串传入) //增加 tab 页
```

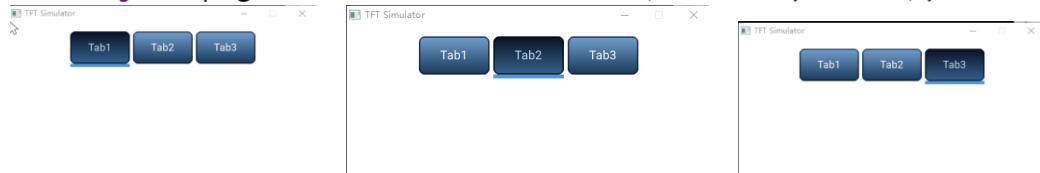
```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *tabview1 = lv_tabview_create(scr,NULL); //创建选项卡  
lv_obj_set_size(tabview1,300,300);  
lv_obj_align(tabview1,NULL,LV_ALIGN_CENTER,0,0);  
  
lv_obj_t *page1 = lv_tabview_add_tab(tabview1,"Tab1"); //增加Tab页
```



这是只有一页的 Tab。

下面增加 3 页 Tab 就能看出效果了

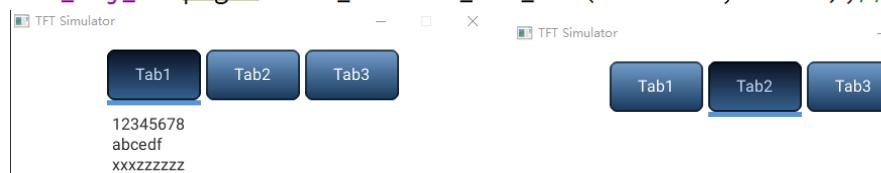
```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *tabview1 = lv_tabview_create(scr,NULL); //创建选项卡  
lv_obj_set_size(tabview1,300,300);  
lv_obj_align(tabview1,NULL,LV_ALIGN_CENTER,0,0);  
  
lv_obj_t *page1 = lv_tabview_add_tab(tabview1,"Tab1"); //增加Tab页  
lv_obj_t *page2 = lv_tabview_add_tab(tabview1,"Tab2"); //增加Tab页  
lv_obj_t *page3 = lv_tabview_add_tab(tabview1,"Tab3"); //增加Tab页
```



向 Tab 页放入控件和数据

```
lv_obj_t *page1 = lv_tabview_add_tab(tabview1,"Tab1"); //增加Tab页  
lv_obj_t *label1 = lv_label_create(page1,NULL); //向tab1页放标签  
lv_label_set_text(label1,"12345678\nabcd\fnxxxxzzzzz"); //向标签写入数据
```

```
lv_obj_t *page2 = lv_tabview_add_tab(tabview1,"Tab2"); //增加Tab页  
lv_obj_t *page3 = lv_tabview_add_tab(tabview1,"Tab3"); //增加Tab页
```



Tab1 页, 有我写入的数据, 切换到 Tab2 页, 是采用滑动的方式。

向 Tab2 页和 Tab3 页加控件和数据

```
lv_obj_t *page1 = lv_tabview_add_tab(tabview1,"Tab1");//增加Tab页
lv_obj_t *label1 = lv_label_create(page1,NULL); //向tab1页放标签
lv_label_set_text(label1,"12345678\nabcdef\nxxxxzzzzz");//向标签写入数据

lv_obj_t *page2 = lv_tabview_add_tab(tabview1,"Tab2");//增加Tab页
lv_obj_t *label2 = lv_label_create(page2,NULL); //向tab2页放标签
lv_label_set_text(label2,"22222222\nxx22222");//向标签写入数据

lv_obj_t *page3 = lv_tabview_add_tab(tabview1,"Tab3");//增加Tab页
lv_obj_t *label3 = lv_label_create(page3,NULL); //向tab3页放标签
lv_label_set_text(label3,"33333333\nxx333333");//向标签写入数据
```



lv_tabview_set_anim_time(选项卡对象,毫秒 ms) //延长切换时间

```
lv_tabview_set_anim_time(tabview1,6000); //延长切换时间6秒
```

lv_tabview_set_btns_pos(选项卡对象,位置) //设置选项卡按钮位置

位置: LV_TABVIEW_BTNS_POS_BOTTOM //按钮在页面底部

LV_TABVIEW_BTNS_POS_LEFT //左边,基本不用, 很难看

LV_TABVIEW_BTNS_POS_RIGHT //右边, 基本不用, 很难看

```
lv_tabview_set_btns_pos(tabview1,LV_TABVIEW_BTNS_POS_BOTTOM); //选项卡在底部
```

12345678
abcdef
xxxxzzzzz



文本域控件创建

```
lv_obj_t * 文本域对象 = lv_ta_create(父窗口,NULL) //创建文本域
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t * ta1 = lv_ta_create(scr,NULL); //创建文本域
lv_obj_set_size(ta1,160,160);
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,0,0);
```



文本域添加字符需要用代码添加

```
lv_ta_add_char(文本域对象,字符) //向文本域添加字符
```

```
lv_obj_t * ta1 = lv_ta_create(scr,NULL); //创建文本域  
lv_obj_set_size(ta1,160,160);  
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,0,0);  
lv_ta_add_char(ta1,'A') //向文本域添加字符
```



在光标前面添加字符

```
lv_ta_add_text(文本域对象,字符串) //向文本域添加字符串
```

```
lv_obj_t * ta1 = lv_ta_create(scr,NULL); //创建文本域  
lv_obj_set_size(ta1,160,160);  
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,0,0);  
lv_ta_add_text(ta1,"12345xxxxzzz") //向文本域添加字符串
```



因为字符是在光标处添加的，所以最好将 `lv_ta_add_char`, `lv_ta_add_text` 放入按钮回调函数

```
lv_obj_t *btn1; //按钮对象  
lv_obj_t *ta1; //文本域对象  
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(obj == btn1)  
    {  
        if(event == LV_EVENT_RELEASED) //按钮按下触发  
        {  
            lv_ta_add_text(ta1,"12345");  
        }  
    }  
}  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
btn1 = lv_btn_create(scr,NULL);  
lv_obj_set_pos(btn1,10,10);  
lv_obj_set_event_cb(btn1,xzz_event_cb);  
  
ta1 = lv_ta_create(scr,NULL); //创建文本域  
lv_obj_set_size(ta1,160,160);  
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,0,0);
```



光标放在字符前面，点击按钮，在光标字符前添加字符

注意，向文本域写入内容要用 **LVGL 内置键盘写才行**，这个键盘后面会讲

```

lv_ta_del_char(文本域对象) //删除光标左侧字符
lv_obj_t *btn1; //按钮对象
lv_obj_t *ta1; //文本域对象
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn1)
    {
        if(event == LV_EVENT_RELEASED) //按钮按下触发
        {
            lv_ta_del_char(ta1); //删除光标左侧字符
        }
    }
}

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
btn1 = lv_btn_create(scr, NULL);
lv_obj_set_pos(btn1, 10, 10);
lv_obj_set_event_cb(btn1, xzz_event_cb);

ta1 = lv_ta_create(scr, NULL); //创建文本域
lv_obj_set_size(ta1, 160, 160);
lv_obj_align(ta1, NULL, LV_ALIGN_CENTER, 0, 0);

```



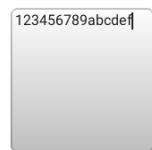
lv_ta_del_char_forward(文本域对象) //删除光标右侧字符

lv_ta_set_text(文本域对象,字符串) //直接设置文本内容，不管文本以前什么内容，一次性设置新字符串到文本

```

ta1 = lv_ta_create(scr, NULL); //创建文本域
lv_obj_set_size(ta1, 160, 160);
lv_obj_align(ta1, NULL, LV_ALIGN_CENTER, 0, 0);
lv_ta_set_text(ta1, "123456789abcdef"); //直接设置文本内容

```



lv_ta_set_cursor_type(文本域对象,参数) //设置光标形状

参数: LV_CURSOR_BLOCK //矩形块

LV_CURSOR_OUTLINE //矩形边框

LV_CURSOR_UNDERLINE //下划线

```
ta1 = lv_ta_create(scr, NULL); //创建文本域
```

```
lv_obj_set_size(ta1, 160, 160);
```

```
lv_obj_align(ta1, NULL, LV_ALIGN_CENTER, 0, 0);
```

```
lv_ta_set_cursor_type(ta1, LV_CURSOR_BLOCK); //设置光标形状, 矩形块
```



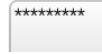
```
ta1 = lv_ta_create(scr, NULL); //创建文本域
```

```
lv_obj_set_size(ta1, 160, 160);
```

```
lv_obj_align(ta1, NULL, LV_ALIGN_CENTER, 0, 0);
```

```
lv_ta_set_cursor_type(ta1, LV_CURSOR_OUTLINE); //设置光标形状, 矩形边框
```

```
lv_ta_set_pwd_mode(文本域对象,true) //密码保护模式
lv_ta_set_pwd_mode(ta1,true); //密码保护模式
```



lv_ta_set_pwd_show_time(文本域对象,毫秒 ms) //密码保护模式下，输入字符显示指定秒然后变成*

```
ta1 = lv_ta_create(scr,NULL); //创建文本域
lv_obj_set_size(ta1,160,160);
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,0,0);
lv_ta_set_pwd_show_time(ta1,3000); //密码保护模式下，输入字符显示3秒然后变成*
lv_ta_set_pwd_mode(ta1,true); //密码保护模式
```

lv_ta_set_one_line(文本域对象,true) //设置文本为单行模式
ta1 = lv_ta_create(scr,NULL); //创建文本域

```
lv_obj_set_size(ta1,160,160);
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,0,0);
lv_ta_set_text(ta1,"12345678\n abcdefg"); //给文本换行
lv_ta_set_one_line(ta1,true); //设置文本为单行模式
```

12345678

只显示单行内容，换行的内容不显示

键盘控件

```
lv_obj_t *键盘对象 = lv_kb_create(父窗口,NULL) //创建键盘控件
lv_kb_set_ta(键盘对象,文本域对象) //将键盘和文本域绑定起来(键盘按下的值自动传入文本)
lv_kb_set_mode(键盘对象,模式) //设置键盘模式
模式:LV_KB_MODE_TEXT //文本模式
LV_KB_MODE_NUM //数字模式
```

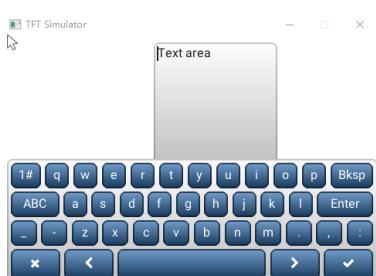
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

```
lv_obj_t * ta1 = lv_ta_create(scr,NULL); //创建文本域
lv_obj_set_size(ta1,160,160);
lv_obj_align(ta1,NULL,LV_ALIGN_CENTER,30,-70);
```

因为键盘敲入的内容需要在文本域中显示，所以要创建文本域

```
lv_obj_t *kb1 = lv_kb_create(scr,NULL); //创建键盘控件
lv_kb_set_ta(kb1,ta1); //将键盘和文本域绑定起来
lv_kb_set_mode(kb1, LV_KB_MODE_TEXT); //设置键盘模式
```

创建键盘，将键盘绑定进文本域



数字键盘设置

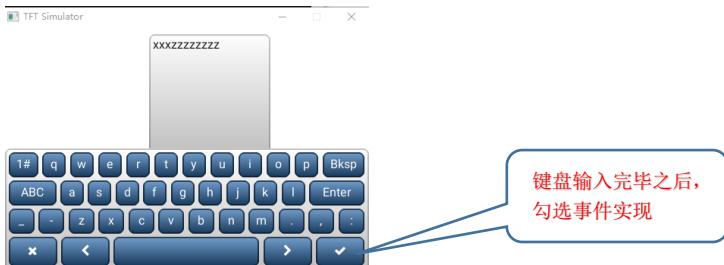
```
lv_obj_t *kb1 = lv_kb_create(scr,NULL); //创建键盘控件  
lv_kb_set_ta(kb1,ta1); //将键盘和文本域绑定起来  
lv_kb_set_mode(kb1,LV_KB_MODE_NUM); //设置键盘模式为数字键盘
```



键盘事件回调函数实现

我们前面使用的是键盘默认的事件回调函数，所以只要键盘与文本域绑定起来，文本域就可以得到键盘值，现在我要实现自定义键盘事件回调函数。

```
void lv_kb_def_event_cb(lv_obj_t * kb, lv_event_t event) //按键默认的事件回调函数  
  
lv_obj_t *kb1; //键盘对象  
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(obj == kb1)  
    {  
        if(event == LV_EVENT_VALUE_CHANGED) //键盘按键被按下  
        {  
            //键盘任何一个按键按下都会执行这里的回调函数，但是文本域没有内容  
            lv_kb_def_event_cb(obj,event);  
            //所以在处理键盘指定按键，调用系统键盘的默认时间回调函数，  
            //先将字符放入文本域  
        }  
    }  
}  
  
kb1 = lv_kb_create(scr,NULL); //创建键盘控件  
lv_kb_set_ta(kb1,ta1); //将键盘和文本域绑定起来  
lv_kb_set_mode(kb1,LV_KB_MODE_TEXT); //设置键盘模式为数字键盘  
lv_obj_set_event_cb(kb1,xzz_event_cb); //注册键盘事件回调函数
```



勾选/取消事件实现方法

LV_EVENT_APPLY //勾选事件

LV_EVENT_CANCEL //取消事件

```
lv_obj_t *kb1; //键盘对象
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == kb1)
    {
        if(event == LV_EVENT_VALUE_CHANGED) //键盘按键被按下
        {
            //键盘任何一个按键按下都会执行这里的回调函数，但是文本域没有内容
            lv_kb_def_event_cb(obj, event);
            //所以在处理键盘指定按键，调用系统键盘的默认时间回调函数，
            //先将字符放入文本域
        }
        else if(event == LV_EVENT_APPLY) //勾选事件触发
        {
            printf("apply\n");
        }
        else if(event == LV_EVENT_CANCEL) //取消事件触发
        {
            printf("cancel\n");
        }
    }
}
```



递增递减控件

lv_obj_t *增减对象 = lv_spinbox_create(父窗口,NULL) //创建递增递减控件

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

```
lv_obj_t *spinbox1 = lv_spinbox_create(scr, NULL); //创建递增递减控件
lv_obj_set_size(spinbox1, 100, 40); //设置大小
lv_obj_align(spinbox1, NULL, LV_ALIGN_CENTER, 0, 0);
```

+00000

这是默认值

```
lv_spinbox_set_range(增减对象,最小值,最大值) //设置增减控件数值范围
lv_spinbox_set_step(增减对象,增减大小) // 每次递增, 递减大小
lv_spinbox_increment(增减对象) //让增减控件递增一次
```

```

lv_obj_t *btn; //监听按钮事件
lv_obj_t *spinbox1;//增减功能要靠按钮事件来实现
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn)
    {
        if(event == LV_EVENT_RELEASED)
        {
            lv_spinbox_increment(spinbox1); //让增减控件递增一次
        }
    }
}
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

spinbox1 = lv_spinbox_create(scr, NULL); //创建递增递减控件
lv_obj_set_size(spinbox1, 100, 40); //设置大小
lv_obj_align(spinbox1, NULL, LV_ALIGN_CENTER, 0, 0);
lv_spinbox_set_range(spinbox1, -1000, 1000); //设置增减控件数值范围
lv_spinbox_set_step(spinbox1, 50); // 每次递增递减50

btn = lv_btn_create(scr, NULL); //创建按钮来实现递增递减
lv_obj_set_pos(btn, 10, 10); //设置按钮摆放坐标
lv_obj_set_event_cb(btn, xzz_event_cb); //按钮事件回调

```



lv_spinbox_decrement(增减对象) //让增减控件递减一次

```

lv_obj_t *btn; //监听按钮事件
lv_obj_t *spinbox1;//增减功能要靠按钮事件来实现
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn)
    {
        if(event == LV_EVENT_RELEASED)
        {
            lv_spinbox_decrement(spinbox1); //让增减控件递减一次
        }
    }
}

```



```

lv_spinbox_set_padding_left(增减对象,空格个数) //增加空格间隙
spinbox1 = lv_spinbox_create(scr, NULL); //创建递增递减控件
lv_obj_set_size(spinbox1, 100, 40); //设置大小
lv_obj_align(spinbox1, NULL, LV_ALIGN_CENTER, 0, 0);
lv_spinbox_set_range(spinbox1, -1000, 1000); //设置增减控件数值范围
lv_spinbox_set_step(spinbox1, 50); // 每次递增递减50
lv_spinbox_set_padding_left(spinbox1, 5); //增减5个空格间隙

```



```

lv_spinbox_set_value(增减对象,默认值) //增减控件创建默认值
lv_spinbox_set_value(spinbox1, 1000); //增减控件创建默认值

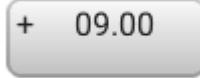
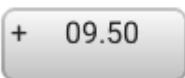
```

lv_spinbox_set_digit_format(增减对象,数值长度,小数点在第几位显示) //设置数值长度, 小数点位置

```

lv_spinbox_set_step(spinbox1, 50); // 每次递增递减50
lv_spinbox_set_padding_left(spinbox1, 5); //增减5个空格间隙
lv_spinbox_set_value(spinbox1, 1000); //增减控件创建默认值1000
lv_spinbox_set_digit_format(spinbox1, 4, 2); //设置数值长度, 小数点位置

```



虽然是小数点显示,但是递增递减还是按照 50 来计算, 只是这个 50 不会受小数点的影响。所以小数点只是一个字符显示而已, 没有计算功能。

增减控件事件回调, 获取增减控件值

lv_spinbox_get_value(增减对象)

```

static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn)
    {
        if(event == LV_EVENT_RELEASED)
        {
            lv_spinbox_decrement(spinbox1); //让增减控件递减一次
        }
    }
    else if(obj == spinbox1) //证明增减控件数值有变化
    {
        if(event == LV_EVENT_VALUE_CHANGED)
        {
            printf("value = %d\n", lv_spinbox_get_value(spinbox1));
        }
    }
}

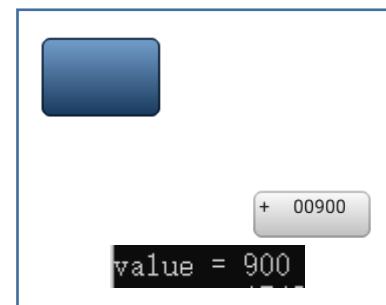
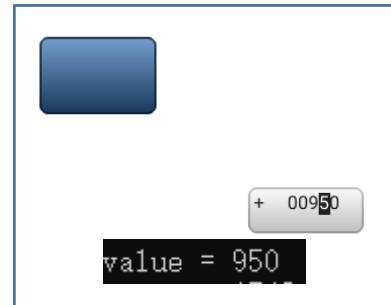
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

spinbox1 = lv_spinbox_create(scr, NULL); //创建递增递减控件
lv_obj_set_size(spinbox1, 100, 40); //设置大小
lv_obj_align(spinbox1, NULL, LV_ALIGN_CENTER, 0, 0);
lv_spinbox_set_range(spinbox1, -1000, 1000); //设置增减控件数值范围
lv_spinbox_set_step(spinbox1, 50); // 每次递增递减50
lv_spinbox_set_padding_left(spinbox1, 5); //增减5个空格间隙
lv_spinbox_set_value(spinbox1, 1000); //增减控件创建默认值1000

btn = lv_btn_create(scr, NULL); //创建按钮来实现递增递减
lv_obj_set_pos(btn, 10, 10); //设置按钮摆放坐标
lv_obj_set_event_cb(btn, xzz_event_cb); //按钮事件回调

lv_obj_set_event_cb(spinbox1, xzz_event_cb); //增减控件事件回调和按钮用同一个回调

```



图片控件使用

显示系统内置图标

```
lv_obj_t *图片对象 = lv_img_create(父窗口,NULL) //创建图片控件  
lv_img_set_src(图片对象,图标参数) //显示图标
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t *image = lv_img_create(scr,NULL); //创建图片控件  
lv_obj_align(image,NULL,LV_ALIGN_CENTER,0,0);
```

```
lv_img_set_src(image,LV_SYMBOL_AUDIO); //显示系统内置图标
```

```
lv_img_set_src(image, LV_SYMBOL_AUDIO"XXXZZZ");//可以在图标参数后面加字符
```



加载图片方式

将图片转换成 C 语言数组进行显示，需要使用在线转换工具，

将图片转成 C 语言数组。我们使用官方的工具 <https://lvgl.io/tools/imageconverter>

Image file

Browse

导入你要转换的图片

File name(s)

自己设置输出 C 文件的名字

Color format

True color

Alpha byte Add a 8 bit Alpha value to every pixel
Chroma keyed Make LV_COLOR_TRANSP (lv_conf.h) pixels to transparent

转换的颜色格式选择

True color 真彩色格式

True color with alpha 带透明度图片格式

.....

有 14 种格式，自行百度

Output format

C array

Dither images (can improve quality)
 Output in big-endian format

选择 C 数组输出文件格式也有 .bin 文件输出格式

Options

Convert

下面我来载入一个图标 PNG 图标

Image file: Xiao 1.png

File name(s): imagg

Color format: True color

Alpha byte Add a 8 bit Alpha value to every pixel
Chroma keyed Make LV_COLOR_TRANSP (lv.conf.h) pixels to transparent

Output format: C array

Options:

- Dither images (can improve quality)
- Output in big-endian format

Convert

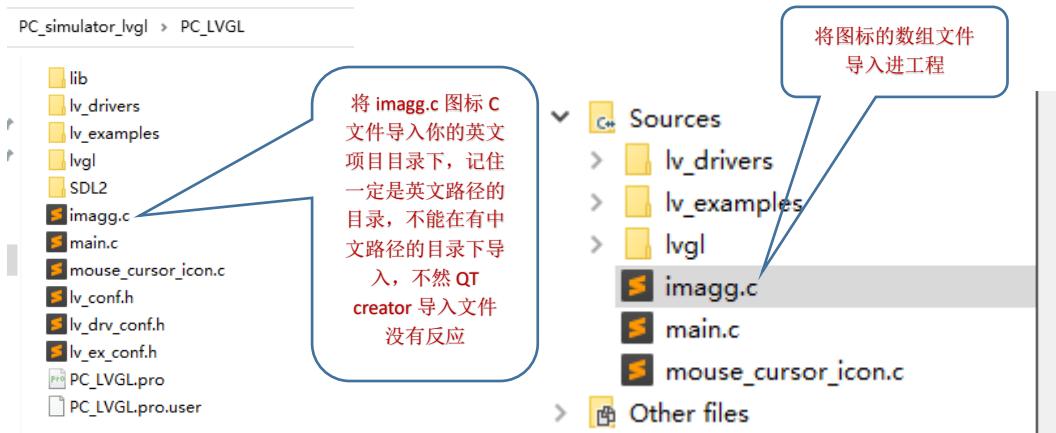
输出 C 文件格式要用英文开头, 如果数字开头, QT creator 可能编译不过

将 imagg.c 文件的 imagg 变量

转换后得到 imagg.c 文件一堆数组, 这个数组不用管, 主要是看

我采用 True color 格式转换

```
const lv_img_dsc_t imgagg = {
    .header.always_zero = 0,
    .header.w = 16,
    .header.h = 16,
    .data_size = 256 * LV_COLOR_SIZE / 8,
    .header.cf = LV_IMG_CF_TRUE_COLOR,
    .data = imgagg_map,
}:
```



`LV_IMG_DECLARE(传入图片定义的结构体变量) //导入其它文件的图片数组到本文件`

```

LV_IMG_DECLARE(imagg); //导入其它目录的图片lv_img_dsc_t 定义的变量

int main(int argc, char ** argv)
{
    (void) argc; /*Unused*/
    (void) argv; /*Unused*/

    /*Initialize LittlevGL*/
    lv_init();

    /*Initialize the HAL (display, input devices, tick) for LittlevGL*/
    hal_init();

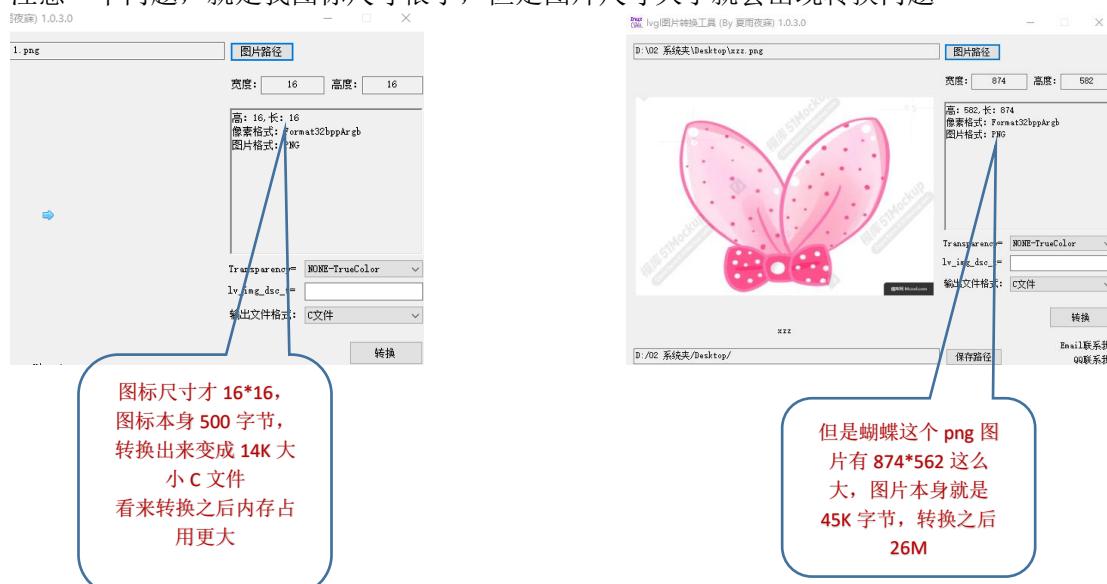
    lv_obj_t *scr = lv_scr_act();

    lv_obj_t *img1 = lv_img_create(scr,NULL); //创建图片控件
    lv_img_set_src(img1,&imagg); //显示图片, 显示图标和图片传入的参数不一样
    lv_obj_align(img1,NULL,LV_ALIGN_CENTER,0,0); //居中


```

图标运行成功。

注意一个问题，就是我图标尺寸很小，但是图片尺寸大了就会出现转换问题



你会发现数组不是全部是 0xff 就全是 0x00

其实蝴蝶 png 图片有真实数据在数组，是因为我 c 本件太大，26M，计算机软件反应不过来，所以没发现差异数据。

```
const lv_img_dsc_t imagg = {
    .header.always_zero = 0,
    .header.w = 874,
    .header.h = 582,
    .data_size = 508668 * LV_COLOR_SIZE / 8,
    .header.cf = LV_IMG_CF_TRUE_COLOR,
    .data = imagg_map,
};
```

这是蝴蝶 `imagg.c` 图片的数据量

就 `data_size` 这一项就有 500K，还要乘上颜色像素，上 M 的容量。

```
LV_IMG_DECLARE(imagg1); //导入蝴蝶图片
int main(int argc, char ** argv)
{
    (void) argc;      /*Unused*/
    (void) argv;      /*Unused*/

    /*Initialize LittlevGL*/
    lv_init();

    /*Initialize the HAL (display, input devices, tick) for LittlevGL*/
    hal_init();

    lv_obj_t *scr = lv_scr_act();

    lv_obj_t *img1 = lv_img_create(scr,NULL); //创建图片控件
    lv_img_set_src(img1,&imagg1); //显示图片,
    lv_obj_align(img1,NULL,LV_ALIGN_CENTER,0,0); //居中
```



蝴蝶图片显示成功，感觉超出了界面尺寸范围。

所以看看能不能叫美工做成 bmp 格式的图片，这样占用内存少些。

图片按钮控件

lv_obj_t *图片按钮对象 = lv_imgbtn_create(父窗口,NULL) //创建图片按钮

lv_imgbtn_set_src(图片按钮对象,状态选择,图片地址) //选择按钮状态与某个图片关联
状态选择: LV_BTN_STATE_REL //按钮松开状态
LV_BTN_STATE_PR //按钮按下状态

图片地址: 根据状态选择, 来决定传入哪中图片数组

```
LV_IMG_DECLARE(imagg); //导入向右按钮
LV_IMG_DECLARE(imagg2); //导入勾选
int main(int argc, char ** argv)
{
    (void) argc;      /*Unused*/
    (void) argv;      /*Unused*/
    /*Initialize LittlevGL*/
    lv_init();

    /*Initialize the HAL (display, input devices, tick) for LittlevGL*/
    hal_init();

    lv_obj_t *scr = lv_scr_act();
    lv_obj_t *ximbtn = lv_imgbtn_create(scr,NULL); //创建图片按钮
    lv_obj_align(ximbtn,NULL,LV_ALIGN_CENTER,0,0); //居中
    lv_imgbtn_set_src(ximbtn,LV_BTN_STATE_REL,&imagg); //设置按钮在松手状态下的一个图片显示
    lv_imgbtn_set_src(ximbtn,LV_BTN_STATE_PR,&imagg2); //设置按钮在按下状态的一个图片显示
```



按键松开状态



按键按下状态

```
lv_obj_t *scr = lv_scr_act();
lv_obj_t *ximbtn = lv_imgbtn_create(scr,NULL); //创建图片按钮
lv_obj_align(ximbtn,NULL,LV_ALIGN_CENTER,0,0); //居中
lv_imgbtn_set_src(ximbtn,LV_BTN_STATE_REL,&imagg); //设置按钮在松手状态下的一个图片显示
lv_imgbtn_set_src(ximbtn,LV_BTN_STATE_PR,&imagg2); //设置按钮在按下状态的一个图片显示

lv_obj_t *imbtn_label = lv_label_create(ximbtn,NULL); //给按钮增加一个标题, 用标签嵌入进按钮
lv_label_set_text(imbtn_label,"ximbtn");
```



按钮有了标题, 只是按钮太小, 标题太大, 看不清楚而已。

图片按钮的事件回调函数我就不写了, 网上都有资料.....

窗体控件

```
lv_obj_t *窗体对象 = lv_win_create(父窗口,NULL) //创建窗体控件  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *win1 = lv_win_create(scr,NULL); //创建窗体控件  
lv_obj_set_size(win1,200,200); //大小  
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐  
  
现在只出现了窗体的标题
```



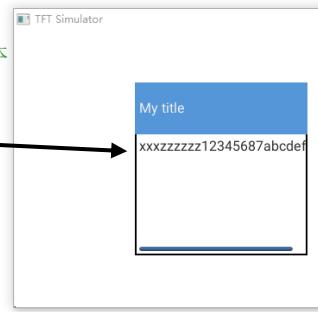
给窗体内部加入文本

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *win1 = lv_win_create(scr,NULL); //创建窗体控件  
lv_obj_set_size(win1,200,200); //大小  
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐  
  
lv_obj_t * label = lv_label_create(win1,NULL); //给窗口内部加入文本  
lv_label_set_text(label,"xxxxxxxx12345687abcdef\n");
```



给窗体加入边框样式

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *win1 = lv_win_create(scr,NULL); //创建窗体控件  
lv_obj_set_size(win1,200,200); //大小  
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐  
  
lv_obj_t * label = lv_label_create(win1,NULL); //给窗口内部加入文本  
lv_label_set_text(label,"xxxxxxxx12345687abcdef\n");  
  
static lv_style_t xbg_style; //给窗体加入边框样式  
  
lv_style_copy(&xbg_style,&lv_style_plain);  
xbg_style.body.border.width = 2;  
xbg_style.body.border.color = LV_COLOR_BLACK;  
  
lv_win_set_style(win1,LV_WIN_STYLE_BG,&xbg_style); //设置窗体样式
```



lv_win_set_title(窗体对象,字符串) //给窗体 title 加入名字

```
lv_obj_t *win1 = lv_win_create(scr,NULL); //创建窗体控件  
lv_win_set_title(win1,"xxxxzztitle"); //给窗体 title 加入名字  
lv_obj_set_size(win1,200,200); //大小  
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐
```



lv_obj_t *窗体按钮对象 = lv_win_add_btn(窗体对象,图标符号) //窗体创建控制按钮

```
lv_obj_t *win1 = lv_win_create(scr,NULL); //创建窗体控件  
lv_win_set_title(win1,"xxxxzztitle"); //给窗体 title 加入名字  
lv_obj_set_size(win1,200,200); //大小  
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐  
lv_obj_t *CloseBtn = lv_win_add_btn(win1,LV_SYMBOL_CLOSE); //窗体创建控制按钮  
  
lv_obj_t *CloseBtn = lv_win_add_btn(win1,LV_SYMBOL_CLOSE); //窗体创建控制按钮  
lv_win_set_btn_size(win1,20); //设置窗体按钮大小
```



窗体按钮的大小也会让标题框大小跟着改变

窗体事件回调函数使用

```
lv_obj_t *CloseBtn; //窗口按钮对象
lv_obj_t *win1; //窗体对象
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == CloseBtn)
    {
        if(event == LV_EVENT_RELEASED)
        {
            lv_obj_del(win1); //关闭窗体
        }
    }
}

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

win1 = lv_win_create(scr,NULL); //创建窗体控件
lv_win_set_title(win1,"xxxxzztitle"); //给窗体title加入名字
lv_obj_set_size(win1,200,200); //大小
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐
CloseBtn = lv_win_add_btn(win1,LV_SYMBOL_CLOSE); //窗体创建控制按钮
lv_win_set_btn_size(win1,20); //设置窗体按钮大小
lv_obj_set_event_cb(CloseBtn,xzz_event_cb); //窗口按钮设置事件回调
```



点击 x

关闭窗体

如果我不使用 `lv_obj_del(win1);` 来关闭窗体对象呢？我不让 `win1` 为全局变量，如何关闭窗体？

```
lv_obj_t *窗体对象 = lv_win_get_from_btn(窗体按钮对象) //获取窗体按钮对象里的窗体
```

```
lv_obj_t *CloseBtn; //窗口按钮对象

static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == CloseBtn)
    {
        if(event == LV_EVENT_RELEASED)
        {
            lv_obj_t *win = lv_win_get_from_btn(CloseBtn); //获取窗体按钮对象
            lv_obj_del(win); //借助控制按钮返回的窗体对象，删除窗体
        }
    }
}

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t *win1 = lv_win_create(scr,NULL); //创建窗体控件
lv_win_set_title(win1,"xxxxzztitle"); //给窗体title加入名字
lv_obj_set_size(win1,200,200); //大小
lv_obj_align(win1,NULL,LV_ALIGN_CENTER,0,0); //中心对齐
CloseBtn = lv_win_add_btn(win1,LV_SYMBOL_CLOSE); //窗体创建控制按钮
lv_win_set_btn_size(win1,20); //设置窗体按钮大小
lv_obj_set_event_cb(CloseBtn,xzz_event_cb); //窗口按钮设置事件回调
```

列表控件

```
lv_obj_t *列表对象 = lv_list_create(父窗口,NULL) //创建列表对象
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t *list = lv_list_create(scr,NULL); //创建列表对象
```

```
lv_obj_set_size(list,150,150); //注意高度设置
```

```
lv_obj_align(list,NULL,LV_ALIGN_CENTER,0,0); //居中
```

列表控件高度
明显没有 150

```
lv_obj_t *列表按钮对象 = lv_list_add_btn(列表对象,图标宏,传入字符串按钮标题) //添加列表按钮
```

图标宏: LV_SYMBOL_AUDIO //音频图标

LV_SYMBOL_BELL //闹铃图标

LV_SYMBOL_CALL //电话图标

LV_SYMBOL_CUT //剪贴图标

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t *list = lv_list_create(scr,NULL); //创建列表对象
```

```
lv_obj_set_size(list,150,150); //注意高度设置
```

```
lv_obj_align(list,NULL,LV_ALIGN_CENTER,0,0); //居中
```

```
lv_obj_t *btnItem1 = lv_list_add_btn(list,LV_SYMBOL_AUDIO,"xAUDIO"); //添加列表按钮
```



增加多个列表按钮

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
```

```
lv_obj_t *list = lv_list_create(scr,NULL); //创建列表对象
```

```
lv_obj_set_size(list,150,150); //注意高度设置
```

```
lv_obj_align(list,NULL,LV_ALIGN_CENTER,0,0); //居中
```

```
lv_obj_t *btnItem1 = lv_list_add_btn(list,LV_SYMBOL_AUDIO,"xAUDIO"); //添加列表按钮
```

```
lv_obj_t *btnItem2 = lv_list_add_btn(list,LV_SYMBOL_BELL,"BELL"); //添加列表按钮
```

```
lv_obj_t *btnItem3 = lv_list_add_btn(list,LV_SYMBOL_CALL,"CALL"); //添加列表按钮
```

```
lv_obj_t *btnItem4 = lv_list_add_btn(list,LV_SYMBOL_CUT,"CUT"); //添加列表按钮
```

```
lv_obj_t *btnItem5 = lv_list_add_btn(list,LV_SYMBOL_HOME,"HOME"); //添加列表按钮
```

```
lv_obj_t *btnItem6 = lv_list_add_btn(list,LV_SYMBOL_VIDEO,"VIDEO"); //添加列表按钮
```



看到没，虽然我建立了 6 个列表控件，但是 size 高度定的 150，所以只显示 150 高度的控件，后面的控件需要滑动显示

给每个列表按钮设置事件回调

```
lv_obj_t *btnItem1; //注册哪个列表按钮，就让该列表按钮定义成全局
```

```
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btnItem1)
    {
        if(event == LV_EVENT_CLICKED) //监听列表按钮1 点击松开后事件
        {
            printf("btnItem1\n");
        }
    }
}
```

```

btnItem1 = lv_list_add_btn(list,LV_SYMBOL_AUDIO,"xAUDIO");//添加列表按钮
lv_obj_t *btnItem2 = lv_list_add_btn(list,LV_SYMBOL_BELL,"BELL");//添加列表按钮
lv_obj_t *btnItem3 = lv_list_add_btn(list,LV_SYMBOL_CALL,"CALL");//添加列表按钮
lv_obj_t *btnItem4 = lv_list_add_btn(list,LV_SYMBOL_CUT,"CUT");//添加列表按钮
lv_obj_t *btnItem5 = lv_list_add_btn(list,LV_SYMBOL_HOME,"HOME");//添加列表按钮
lv_obj_t *btnItem6 = lv_list_add_btn(list,LV_SYMBOL_VIDEO,"VIDEO");//添加列表按钮

```

`lv_obj_set_event_cb(btnItem1,xzz_event_cb); //注册列表按钮1事件回调`



因为只有音频按钮注册了事件回调

其余按钮注册事件回调，只需要多加几条 `lv_obj_set_event_cb` 函数就可以了

`lv_list_set_btn_selected(列表对象,列表中指定选中按钮对象) //指定列表中某个按钮处于开机选中状态`

```

lv_obj_t *list = lv_list_create(scr,NULL);//创建列表对象
lv_obj_set_size(list,150,150); //注意高度设置
lv_obj_align(list,NULL,LV_ALIGN_CENTER,0,0); //居中

btnItem1 = lv_list_add_btn(list,LV_SYMBOL_AUDIO,"xAUDIO");//添加列表按钮
lv_obj_t *btnItem2 = lv_list_add_btn(list,LV_SYMBOL_BELL,"BELL");//添加列表按钮
lv_obj_t *btnItem3 = lv_list_add_btn(list,LV_SYMBOL_CALL,"CALL");//添加列表按钮
lv_obj_t *btnItem4 = lv_list_add_btn(list,LV_SYMBOL_CUT,"CUT");//添加列表按钮
lv_obj_t *btnItem5 = lv_list_add_btn(list,LV_SYMBOL_HOME,"HOME");//添加列表按钮
lv_obj_t *btnItem6 = lv_list_add_btn(list,LV_SYMBOL_VIDEO,"VIDEO");//添加列表按钮

lv_obj_set_event_cb(btnItem1,xzz_event_cb); //注册列表按钮1事件回调
lv_list_set_btn_selected(list,btnItem2); //让BELL按钮开机处于选中状态

```



`lv_list_set_single_mode(列表对象,true) //设置列表按钮字符滚动`
列表按钮字符超出按钮尺寸就会滚动

```

btnItem1 = lv_list_add_btn(list,LV_SYMBOL_AUDIO,"xAUDIO");//添加列表按钮
lv_obj_t *btnItem2 = lv_list_add_btn(list,LV_SYMBOL_BELL,"BELL111111111111"); //添加列表按钮
lv_obj_t *btnItem3 = lv_list_add_btn(list,LV_SYMBOL_CALL,"CALL");//添加列表按钮
lv_obj_t *btnItem4 = lv_list_add_btn(list,LV_SYMBOL_CUT,"CUT");//添加列表按钮
lv_obj_t *btnItem5 = lv_list_add_btn(list,LV_SYMBOL_HOME,"HOME");//添加列表按钮
lv_obj_t *btnItem6 = lv_list_add_btn(list,LV_SYMBOL_VIDEO,"VIDEO");//添加列表按钮

lv_obj_set_event_cb(btnItem1,xzz_event_cb); //注册列表按钮1事件回调
lv_list_set_btn_selected(list,btnItem2); //让BELL按钮开机处于选中状态
lv_list_set_single_mode(list,true); //设置列表按钮字符滚动

```



点击外部按钮，滑动列表里面的按钮，在电阻屏或者按键应用中使用

`lv_list_up(列表对象) //列表里面的按钮向上移动`

```

lv_obj_t *btn1; //外部按钮控件全局
lv_obj_t *list; //列表按钮控件全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn1) //外部按钮事件
    {
        if(event == LV_EVENT_RELEASED) //外部按钮松开
        {
            lv_list_up(list); //列表向上移动
        }
    }
}

```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

list = lv_list_create(scr, NULL); //创建列表对象
lv_obj_set_size(list, 150, 150); //注意高度设置
lv_obj_align(list, NULL, LV_ALIGN_CENTER, 0, 0); //居中

lv_obj_t *btnItem1 = lv_list_add_btn(list, LV_SYMBOL_AUDIO, "xAUDIO"); //添加列表按钮
lv_obj_t *btnItem2 = lv_list_add_btn(list, LV_SYMBOL_BELL, "BELL1111111111"); //添加列表按钮
lv_obj_t *btnItem3 = lv_list_add_btn(list, LV_SYMBOL_CALL, "CALL"); //添加列表按钮
lv_obj_t *btnItem4 = lv_list_add_btn(list, LV_SYMBOL_CUT, "CUT"); //添加列表按钮
lv_obj_t *btnItem5 = lv_list_add_btn(list, LV_SYMBOL_HOME, "HOME"); //添加列表按钮
lv_obj_t *btnItem6 = lv_list_add_btn(list, LV_SYMBOL_VIDEO, "VIDEO"); //添加列表按钮

btn1 = lv_btn_create(scr, NULL); //创建外部按钮
lv_obj_set_pos(btn1, 10, 10);
lv_obj_set_event_cb(btn1, xzz_event_cb); //设置外部按钮事件回调

```



lv_list_down(列表对象) //列表里面的按钮向下移动

lv_list_focus(列表按钮对象,true) //指定列表里面某个按钮直接滑动到显示区

```

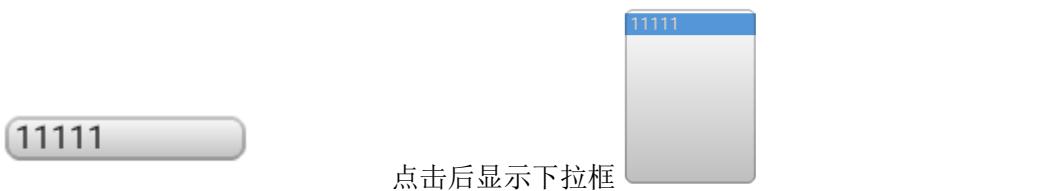
lv_obj_t *btnItem6; //列表里面的VIDEO按钮全局
lv_obj_t *btn1; //外部按钮控件全局
lv_obj_t *list; //列表按钮控件全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn1) //外部按钮事件
    {
        if(event == LV_EVENT_RELEASED) //外部按钮松开
        {
            lv_list_focus(btnItem6, true); //指定列表里面VIDEO按钮直接滑动到显示区
        }
    }
}

```



下拉列表框控件

```
lv_obj_t *下拉列表对象 = lv_ddlist_create(父窗口,NULL) //创建下拉列表  
lv_ddlist_set_fix_width(下拉列表对象,宽度值) //下拉列表专用宽度设置  
lv_ddlist_set_fix_height(下拉列表对象,高度值) //下拉列表专用高度设置  
lv_ddlist_set_options(下拉列表对象,内容为字符串) //下拉列表加入一行选择内容  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t * ddlist = lv_ddlist_create(scr,NULL); //创建下拉列表  
lv_ddlist_set_fix_width(ddlist,120); //下拉列表专用宽度设置  
lv_ddlist_set_fix_height(ddlist,160); //下拉列表专用高度设置  
lv_obj_set_pos(ddlist,100,20); //设置下拉列表位置  
lv_ddlist_set_options(ddlist,"11111"); //下拉列表加入一行选择内容
```



如果下拉列表要显示多行选择的字符怎么操作？记住，并不是我们执行多次 `lv_ddlist_set_options` 就会产生多行。

```
lv_ddlist_set_options(ddlist,"11111\n22222\n33333"); //在字符串里面加\n增加多行选择
```



这样就可以选择多个下拉选项

获取下拉框选中第几行字符

```
lv_ddlist_set_selected(下拉列表对象,填入第几行字符) //选择下拉第几行字符
```

```
lv_obj_t * ddlist; //下拉框全局  
lv_obj_t *btn1; //外部按钮控件全局  
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(obj == btn1) //外部按钮事件  
    {  
        if(event == LV_EVENT_RELEASED) //外部按钮松开  
        {  
            lv_ddlist_set_selected(ddlist,2); //选择下拉第3行字符  
        }  
    }  
}
```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

ddlist = lv_ddlist_create(scr, NULL); //创建下拉列表
lv_ddlist_set_fix_width(ddlist, 120); //下拉列表专用宽度设置
lv_ddlist_set_fix_height(ddlist, 160); //下拉列表专用高度设置
lv_obj_set_pos(ddlist, 100, 20); //设置下拉列表位置
lv_ddlist_set_options(ddlist, "11111\n22222\n33333"); //在字符里面加\n增加多行选择

```

```

btn1 = lv_btn_create(scr, NULL); //创建按钮来获取下拉框选择的某行字符
lv_obj_set_pos(btn1, 0, 0);
lv_obj_set_event_cb(btn1, xzz_event_cb);

```



```
lv_ddlist_set_draw_arrow(下拉列表对象,true) //下拉菜单右侧加入箭头
```



```
lv_ddlist_set_align(下拉列表对象,对齐参数) //下拉菜单字符对齐方式
```

对齐参数: LV_LABEL_ALIGN_CENTER //居中对齐

LV_LABEL_ALIGN_RIGHT //右对齐

```

ddlist = lv_ddlist_create(scr, NULL); //创建下拉列表
lv_ddlist_set_fix_width(ddlist, 120); //下拉列表专用宽度设置
lv_ddlist_set_fix_height(ddlist, 160); //下拉列表专用高度设置
lv_obj_set_pos(ddlist, 100, 20); //设置下拉列表位置
lv_ddlist_set_options(ddlist, "11111\n22222\n33333"); //在字符里面加\n增加多行选择
lv_ddlist_set_draw_arrow(ddlist, true); //下拉菜单右侧加入箭头
lv_ddlist_set_align(ddlist, LV_LABEL_ALIGN_CENTER); //下拉菜单字符居中对齐

```



根据下拉菜单的选择，执行内容

```
uint16_t 返回值 = lv_ddlist_get_selected(下拉列表对象) //获取选择下拉菜单的某行索引
```

```

lv_obj_t * ddlist; //下拉框全局
lv_obj_t * btn1; //外部按钮控件全局
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)
{
    if(obj == btn1) //外部按钮事件
    {
        if(event == LV_EVENT_RELEASED) //外部按钮松开
        {
            lv_ddlist_set_selected(ddlist, 2); //选择下拉第3行字符
        }
    }
    else if(obj == ddlist)
    {
        if(event == LV_EVENT_VALUE_CHANGED) //监听到下拉菜单变化事件
        {
            uint16_t id = lv_ddlist_get_selected(ddlist); //获取选择下拉菜单的某行索引
            printf("id = %d\n", id);
        }
    }
}

```

```

lv_ddlist_set_options(ddlist, "11111\n22222\n33333"); //在字符里面加\n增加多行选择
lv_ddlist_set_draw_arrow(ddlist, true); //下拉菜单右侧加入箭头
lv_ddlist_set_align(ddlist, LV_LABEL_ALIGN_CENTER); //下拉菜单字符居中对齐
lv_obj_set_event_cb(ddlist, xzz_event_cb); //设置选择下拉菜单某行字符之后执行事件回调

```



下拉菜单样式设置

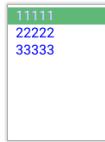
```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

ddlist = lv_ddlist_create(scr, NULL); //创建下拉列表
lv_ddlist_set_fix_width(ddlist, 120); //下拉列表专用宽度设置
lv_ddlist_set_fix_height(ddlist, 160); //下拉列表专用高度设置
lv_obj_set_pos(ddlist, 100, 20); //设置下拉列表位置
lv_ddlist_set_options(ddlist, "11111\n22222\n33333"); //在字符串里面加\n 增加多行选择
lv_ddlist_set_draw_arrow(ddlist, true); //下拉菜单右侧加入箭头
lv_ddlist_set_align(ddlist, LV_LABEL_ALIGN_CENTER); //下拉菜单字符居中对齐
lv_obj_set_event_cb(ddlist, xzz_event_cb); //设置选择下拉菜单某行字符之后执行事件回调

//下拉菜单背景样式
static lv_style_t bg_style; //下拉菜单样式设置
lv_style_copy(&bg_style, &lv_style_plain);
bg_style.body.main_color = LV_COLOR_WHITE;
bg_style.body.grad_color = bg_style.body.main_color;
bg_style.body.border.width = 1; //边框宽度
bg_style.body.border.color = LV_COLOR_MAKE(0xAA, 0xAA, 0xAA);
bg_style.body.padding.left = 10; //设置左侧内边距
bg_style.text.color = LV_COLOR_BLUE; //文本颜色
bg_style.body.shadow.color = LV_COLOR_MAKE(0xAA, 0xAA, 0xAA); //阴影颜色
bg_style.body.shadow.width = 4; //阴影宽度

lv_ddlist_set_style(ddlist, LV_DDLIST_STYLE_BG, &bg_style); //设置背景样式

//设置下拉框选项，被选中的样式
static lv_style_t sel_style; //下拉菜单样式设置
lv_style_copy(&sel_style, &lv_style_plain);
sel_style.body.main_color = LV_COLOR_MAKE(0x5F, 0xB8, 0x78); //浅绿色背景
sel_style.body.grad_color = LV_COLOR_MAKE(0x5F, 0xB8, 0x78);
sel_style.text.color = LV_COLOR_WHITE; //文本为纯白色
lv_ddlist_set_style(ddlist, LV_DDLIST_STYLE_SEL, &sel_style);
```



滚轮控件(下拉列表的另一种形式)

```
lv_obj_t *滚轮对象 = lv_roller_create(父窗口,NULL) //创建滚轮下拉表  
lv_roller_set_fix_width(滚轮对象,滚轮窗口宽度值) //设置滚动条宽度  
lv_roller_set_options(滚轮对象,传入字符串,模式) //设置滚轮行数和模式  
传入字符串: 一行就是一串字符, 使用\n来换第2行字符, 和下拉菜单一样操作  
模式: LV_ROLLER_MODE_NORMAL //标准模式  
lv_roller_set_visible_row_count(滚轮对象,可见行数) //滚轮可见行数
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *roller = lv_roller_create(scr,NULL); //创建滚轮下拉表  
lv_roller_set_fix_width(roller,150); //设置滚动条宽度  
lv_roller_set_options(roller,"1111\n2222\n3333\n4444\n5555",LV_ROLLER_MODE_NORMAL); //设置滚轮行数和模式  
lv_obj_align(roller,NULL,LV_ALIGN_CENTER,0,0); //居中  
lv_roller_set_visible_row_count(roller,3); //滚轮可见行数为3行  
lv_obj_set_event_cb(roller,xzz_event_cb); //设置滚轮事件回调
```



鼠标或者触摸屏滑动滚轮界面

```
lv_roller_set_selected(滚轮对象,默认选中行数,LV_ANIM_ON) //滚轮开机默认选中第几行  
LV_ANIM_ON: 开启动画效果
```

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *roller = lv_roller_create(scr,NULL); //创建滚轮下拉表  
lv_roller_set_fix_width(roller,150); //设置滚动条宽度  
lv_roller_set_options(roller,"1111\n2222\n3333\n4444\n5555",LV_ROLLER_MODE_NORMAL);  
lv_obj_align(roller,NULL,LV_ALIGN_CENTER,0,0); //居中  
lv_roller_set_visible_row_count(roller,3); //滚轮可见行数为3行  
lv_roller_set_selected(roller,2,LV_ANIM_ON); //滚轮开机默认选中第2行  
lv_obj_set_event_cb(roller,xzz_event_cb); //设置滚轮事件回调
```



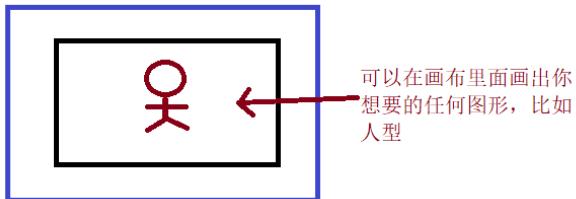
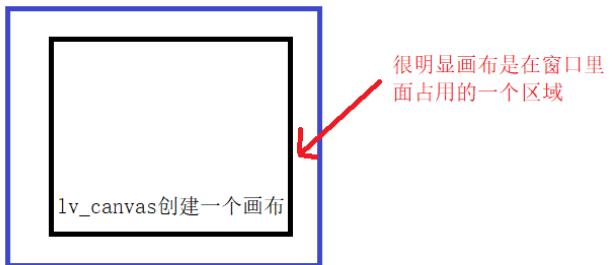
滚轮事件回调函数

```
uint16_t 返回值 = lv_roller_get_selected(滚轮对象) //获取滚轮选中的索引值  
lv_obj_t *roller; //滚轮全局  
static void xzz_event_cb(lv_obj_t * obj, lv_event_t event)  
{  
    if(obj == roller) //外部滚轮事件  
    {  
        if(event == LV_EVENT_VALUE_CHANGED) //获取滚轮改变值  
        {  
            uint16_t id = lv_roller_get_selected(roller); //获取滚轮选中的索引值  
            printf("id = %d\n", id);  
        }  
    }  
}
```



滚轮滑动到一行, 就会打印对应该行的 ID 索引值。

画布使用，画布可以绘制原始图形，正方形，长方形，或者任意图形



`lv_canvas_set_buffer(画布对象, 缓冲区数组, 宽, 高, 画布颜色) //设置画布缓冲区`

宽: 设置的是画布绘图区域的宽, 而不是画布本身的宽。

高: 设置的是画布绘图区域的高, 而不是画布本身的高。

画布颜色: `LV_IMG_CF_TRUE_COLOR //画布为真彩色(五颜六色图案)`

`LV_IMG_CF_INDEXED_1BIT //调色板格式, 绘制出来的颜色不是很多,1bit 画布就只能显示 2 中颜色, (红蓝),(绿黄), 等等...`

缓冲区数组: 缓冲区大小根据画布长宽来决定

比如画布颜色选择的 `LV_IMG_CF_TRUE_COLOR`, 那么缓冲区就要调用对应的 `LV_CANVAS_BUF_SIZE_TRUE_COLOR` 来分配大小

返回值=`LV_CANVAS_BUF_SIZE_TRUE_COLOR(画布宽度,画布高度) //设置画布 TRUE_COLOR 模式缓冲区大小`

`lv_obj_t *画布对象 = lv_canvas_create(父窗口,NULL) //创建画布`

```
#define C_WIDTH 200 //画布宽度  
#define C_HEIGHT 200 //画布高度
```

```
lv_color_t lvbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(C_WIDTH,C_HEIGHT)]; //设置画布缓冲区大小  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
lv_obj_t *canasl = lv_canvas_create(scr,NULL);  
lv_canvas_set_buffer(canasl,lvbuf,C_WIDTH,C_HEIGHT,LV_IMG_CF_TRUE_COLOR); //设置画布缓冲区  
lv_obj_align(canasl,NULL,LV_ALIGN_CENTER,0,0); //画布居中
```



画布默认是黑色

`lv_canvas_fill_bg(画布对象,颜色值) //修改画布背景颜色`

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *canvasl = lv_canvas_create(scr, NULL); //创建画布
lv_canvas_set_buffer(canvasl, lbuf, C_WIDTH, C_HEIGHT, LV_IMG_CF_TRUE_COLOR); //设置画布缓冲区
lv_obj_align(canvasl, NULL, LV_ALIGN_CENTER, 0, 0); //画布居中
lv_canvas_fill_bg(canvasl, LV_COLOR_SILVER); //修改画布背景颜色

```



画布背景色改为灰色

```

lv_canvas_draw_rect(画布对象,矩形 x 位置,矩形 y 位置,矩形宽,矩形高,传入矩形样式地址)
//在画布中绘制矩形

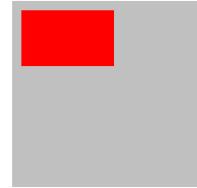
```

```

lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象
lv_obj_t *canvasl = lv_canvas_create(scr, NULL); //创建画布
lv_canvas_set_buffer(canvasl, lbuf, C_WIDTH, C_HEIGHT, LV_IMG_CF_TRUE_COLOR); //设置画布缓冲区
lv_obj_align(canvasl, NULL, LV_ALIGN_CENTER, 0, 0); //画布居中
lv_canvas_fill_bg(canvasl, LV_COLOR_SILVER); //修改画布背景颜色

static lv_style_t bg_style; //创建矩形背景样式
lv_style_copy(&bg_style, &lv_style_plain_color);
bg_style.body.main_color = LV_COLOR_RED; //矩形背景颜色为红色
bg_style.body.grad_color = LV_COLOR_RED;
lv_canvas_draw_rect(canvasl, 10, 10, 100, 60, &bg_style);

```



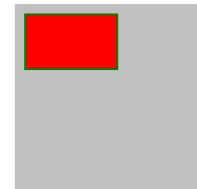
你看，矩形在以画布为原点开始计算位置，然后绘制

如果你想给矩形绘制边框，直接修改样式就是

```

static lv_style_t bg_style; //创建矩形背景样式
lv_style_copy(&bg_style, &lv_style_plain_color);
bg_style.body.main_color = LV_COLOR_RED; //矩形背景颜色为红色
bg_style.body.grad_color = LV_COLOR_RED;
bg_style.body.border.width = 2; //矩形边框宽度
bg_style.body.border.color = LV_COLOR_GREEN; //矩形边框颜色
lv_canvas_draw_rect(canvasl, 10, 10, 100, 60, &bg_style);

```



所以画布里面是画自己定义的任意图形。

```

lv_canvas_draw_text(画布对象,文本 x 位置,文本 y 位置,文本宽度,文本样式地址,"字符串文本内容",文本对齐方式) //在画布中绘制文本

```

```

static lv_style_t bg_style; //创建矩形背景样式
lv_style_copy(&bg_style, &lv_style_plain_color);
bg_style.body.main_color = LV_COLOR_RED; //矩形背景颜色为红色
bg_style.body.grad_color = LV_COLOR_RED;
bg_style.body.border.width = 2; //矩形边框宽度
bg_style.body.border.color = LV_COLOR_GREEN; //矩形边框颜色
lv_canvas_draw_rect(canvasl, 10, 10, 100, 60, &bg_style);
lv_canvas_draw_text(canvasl, 10, 100, 100, &bg_style, "xxxxxxxx", LV_LABEL_ALIGN_LEFT); //在画布中绘制文本

```



```

bg_style.text.color = LV_COLOR_RED; //文本颜色改为红色
lv_canvas_draw_rect(canvasl, 10, 10, 100, 60, &bg_style);
lv_canvas_draw_text(canvasl, 10, 100, 100, &bg_style, "xxxxxxxx", LV_LABEL_ALIGN_LEFT); //在画布中绘制文本

```



画布就是画自己想定义的图形，其余图形请查阅相关手册。

theme 主题

如果不想每个按钮，窗口都去设置样式，可以直接用 Little VGL 内置的主题

先测试系统默认情况下的主题

```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象

lv_obj_t *tabview = lv_tabview_create(scr, NULL); //创建选项卡来测试主题效果
lv_obj_t *tab1 = lv_tabview_add_tab(tabview, "Tab111");
lv_page_set_scrl_layout(tab1, LV_LAYOUT_PRETTY); //设置页面布局方式

lv_obj_t *btn1 = lv_btn_create(tab1, NULL); //创建按钮
lv_obj_t *label1 = lv_label_create(btn1, NULL); //给按钮写字，必须将按钮放入标签，不然会黑屏
lv_label_set_text(label1, "TAB1 BTN1"); //按钮加字符
```



这是 LVGL 默认样式

下面来修改主题

```
=====
 * THEME USAGE
=====
#define LV_THEME_LIVE_UPDATE 1 /*1: Allow theme update*/
#define LV_USE_THEME_TEMPL 1 /*Just for test*/
#define LV_USE_THEME_DEFAULT 1 /*Built mainly*/
#define LV_USE_THEME_ALIEN 1 /*Dark futurist*/
#define LV_USE_THEME_NIGHT 1 /*Dark elegant*/
#define LV_USE_THEME_MONO 1 /*Mono color theme*/
#define LV_USE_THEME_MATERIAL 1 /*Flat theme with shadows*/
#define LV_USE_THEME_ZEN 1 /*Peaceful, minimalist*/
#define LV_USE_THEME_NEMO 1 /*Water-like theme*/
```

在 lv_conf.h 文件下，打开所有 LV_USE_THEME... 8 个主题。注意：在单片机上跑的时候只需要打开一个主题或者两个主题来切换就可以了，这样可以节省很多内存。

现在是模拟器调试，我打开所有 8 个主题。记住，主题打开了，只是在配置文件打开了，编译占用写内存。但是主题不会全部显示在系统中。必须在主函数调用指定的主题初始化才有效。

每个主题都是用不同的函数来初始化

```
lv_theme_t *主题对象 = lv_theme_night_init(基色, NULL) //创建 8 号主题
基色：基色是什么？下面来试验
```

```
lv_theme_t * theme = lv_theme_templ_init(hue, font); //创建 templ 主题
lv_theme_t * theme = lv_theme_default_init(hue, font); //创建 default 主题
lv_theme_t * theme = lv_theme_alien_init(hue, font); //创建 alien 主题
lv_theme_t * theme = lv_theme_night_init(hue, font); //创建 night 主题
lv_theme_t * theme = lv_theme_mono_init(hue, font); //创建 mono 主题

lv_theme_t * theme = lv_theme_material_init(hue, font); //创建 material 主题
lv_theme_t * theme = lv_theme_zen_init(hue, font); //创建 zen 主题
lv_theme_t * theme = lv_theme_nemo_init(hue, font); //创建 nemo 主题
```

不同的主题有不同的 API，一共是 8 个 API 接口，看你需要使用哪一个主题，就调用哪一个 API

```
lv_theme_set_current(主题对象) //使用主题
```

main.....

```
lv_theme_t *theme = lv_theme_night_init(210,NULL); //创建8号主题  
lv_theme_set_current(theme); //使用主题  
  
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *tabview = lv_tabview_create(scr,NULL); //创建选项卡来测试主题效果  
lv_obj_t *tab1 = lv_tabview_add_tab(tabview,"Tab111");  
lv_page_set_scrl_layout(tab1, LV_LAYOUT_PRETTY); //设置页面布局方式  
  
lv_obj_t *btn1 = lv_btn_create(tab1,NULL); //创建按钮  
lv_obj_t *label1 = lv_label_create(btn1,NULL); //给按钮写字,必须将按钮放入标签.不然会黑屏  
lv_label_set_text(label1,"TAB1 BTN1"); //按钮加字符
```

我在初始化函数时只执行 8 号主题。那么界面就只显示 8 号主题风格。



你看，8号主题是黑色风格为主。

下面来修改基色

修改基色, 为 50

```
lv_theme_t *theme = lv_theme_night_init(50,NULL); //创建8号主题  
lv_theme_set_current(theme); //使用主题
```

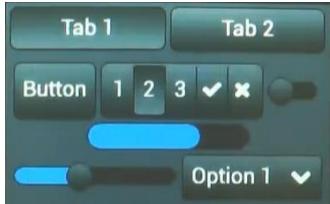
```
lv_obj_t *scr = lv_scr_act(); //获取当前活跃的屏幕对象  
  
lv_obj_t *tabview = lv_tabview_create(scr,NULL); //创建选项卡来测试主题效果  
lv_obj_t *tab1 = lv_tabview_add_tab(tabview,"Tab111");  
lv_page_set_scrl_layout(tab1, LV_LAYOUT_PRETTY); //设置页面布局方式 中间显示  
lv_obj_t *btn1 = lv_btn_create(tab1,NULL); //创建按钮  
lv_obj_t *label1 = lv_label_create(btn1,NULL); //给按钮写字,必须将按钮放入标签.不然会黑屏  
lv_label_set_text(label1,"TAB1 BTN1"); //按钮加字符
```



基色修改后, 背景明显变的泛黄了

这是没修改基色的样子

基色修改, 最明显的是控件的变化, 比如你加了进度条



基色没变的样子



基色变了的样子, 主要是滚动条变化明显

适合 OLED 小型界面库

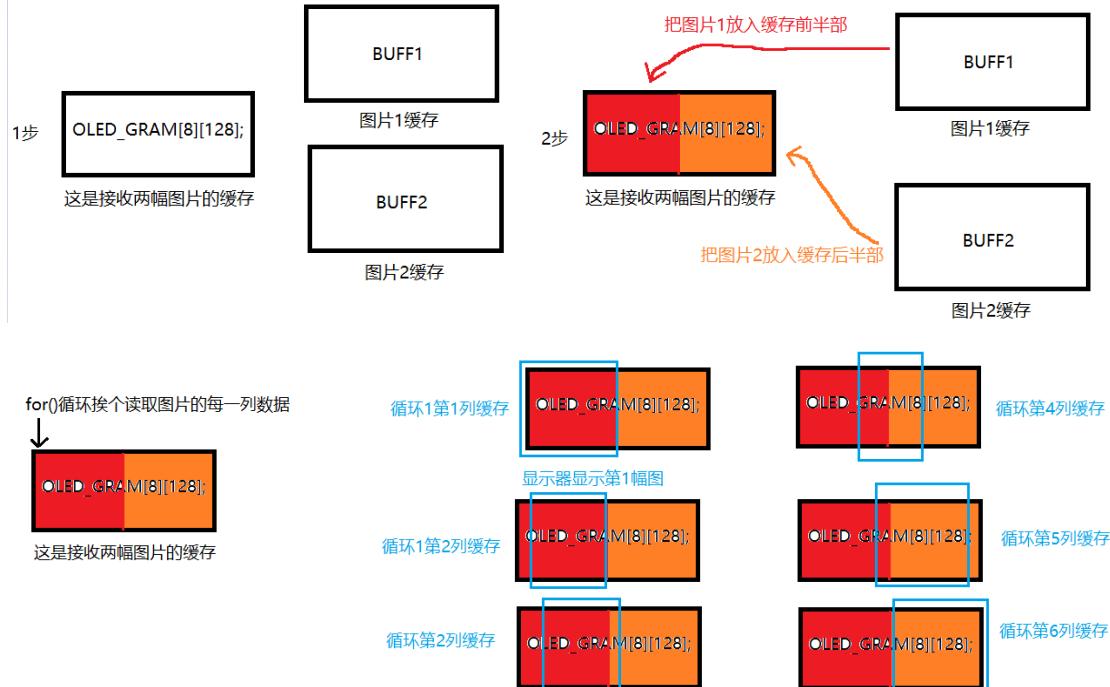
1. 实现好 OLED 底层画点函数，自行实现

2. 移植 OLED 动画算法，如下：

OLED 不管显示任何图片都需要一个缓存

unsigned char OLED_GRAM[8][128]; //界面刷新缓存，这个变量很关键

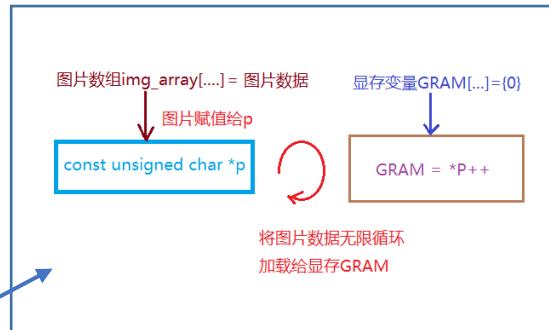
为什么需要界面刷新缓存？因为该缓存可以存储一张一维数组的图片，也可以将几张图片拼接之后放入该缓存，使其 OLED_GRAM 可以滑动图片，实现界面滑动效果



这就是整个水平画图图片的原理

用显存方式显示单张图片，这样比传统直接将图片的一维数组写入驱动要麻烦些，但是可以做到滑动窗口的效果。现在先测试显存显示单张图片的情况。

```
/*
功能：将图片加载到指定显存位置
* uint8_t x: 要加载位置的 x 坐标, 0~127
* uint8_t y: 要加载位置的 y 坐标, 0~7
* uint8_t (*ram)[128]: 显示界面缓存的二维数组指针
* const unsigned char *p: 要加载到显存中的图片指针
记住，图片只能是一维数组
* uint8_t imgWidth: 图片的宽度像素数目
* uint8_t imgHeight: 图片的高度像素数目
*/
void load_Img(unsigned char x,
              unsigned char y, unsigned char (*ram)[128], const unsigned char *p,
              unsigned char imgWidth, unsigned char imgHeight)
{
    unsigned int i,j;
    for(j=0; j<(imgHeight/8); j++)
    {
        for(i=0; i<imgWidth; i++)
        {
            *(ram+7-y-j)+x+i) = *(p+i+j*imgWidth);
        }
    }
}
```



```

/*
* 功能：将指定显存更新到当前屏幕上显示
* uint8_t (*ram)[128]: 显示界面缓存的二维数组指针
*/
void ma_OLED_Refresh_Gram(unsigned char (*ram)[128])
{
    unsigned char i,n;

    for(i=0;i<8;i++)
    {
        /* 设置坐标点 */
        OLED_WR_Byte (0xb0+i,OLED_CMD);
        OLED_WR_Byte (0x00,OLED_CMD);
        OLED_WR_Byte (0x10,OLED_CMD);

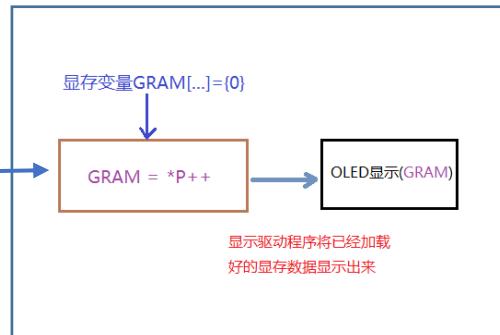
        /* 向 GRAM 中写入显示的数据 */
        OLED_DC_H(); //DC = 1 示数据;
        OLED_CS_L(); //CS = 0

        /* 以后使用 DMA 替代，OLED 的显存由二维数组，替换为 8 个一维数组 */
        for(n=0;n<128;n++)
        {
            OLED_Write_Data(*(*(ram+i)+n));

        }
        OLED_CS_H(); //CS = 1
        OLED_DC_H(); //DC = 1
    }
}

```

The diagram illustrates the process of refreshing the OLED display. It starts with setting the page address (0~7) to specify the row of memory to be read. Then, it sets the display position (low address) and high address. The next step is to read the data from memory, which is indicated by the formula $GRAM = *P++$. Finally, the OLED driver displays the data from the GRAM variable.



数字显示测试，数码管段码样式显示，字库

如果在测试过程中发现数字‘7’是颠倒的，那么就是oled_asc2_3216[12][64]编码表取模方向没对，可以重新取模。也可以修改OLED显示方向。

下面我们修改 OLED 显示方向，就是修改初始化函数。

```

//初始化 SSD1306
void OLED_Init(void)
{
    OLED_RST_H(); //RESET = 1
    delay_ms(1);
    OLED_RST_L(); //RESET = 0
    delay_ms(10);
    OLED_RST_H(); //RESET = 1 复位 OLED

    OLED_WR_Byte(0xAE,OLED_CMD); //--turn off oled panel
    OLED_WR_Byte(0x00,OLED_CMD); //--set low column address
    OLED_WR_Byte(0x10,OLED_CMD); //--set high column address
    OLED_WR_Byte(0x40,OLED_CMD); //--set start line address Set Mapping RAM Display Start Line (0x00~0x3F)
    OLED_WR_Byte(0x81,OLED_CMD); //--set contrast control register
    OLED_WR_Byte(0xCF,OLED_CMD); // Set SEG Output Current Brightness
    OLED_WR_Byte(0xA1,OLED_CMD); //--SET SEG/Column Mapping 0xa1 左右反置 0xa0 正常
    OLED_WR_Byte(0xC8,OLED_CMD); //Set COM/Row Scan Direction 0xc0 上下反置 0xc8 正常
    OLED_WR_Byte(0xA6,OLED_CMD); //--set normal display
    OLED_WR_Byte(0xA8,OLED_CMD); //--set multiplex ratio(1 to 64)
    OLED_WR_Byte(0x3f,OLED_CMD); //--1/64 duty
    OLED_WR_Byte(0xD3,OLED_CMD); //--set display offset Shift Mapping RAM Counter (0x00~0x3F)
    OLED_WR_Byte(0x00,OLED_CMD); //--not offset
    OLED_WR_Byte(0xd5,OLED_CMD); //--set display clock divide ratio/oscillator frequency
    OLED_WR_Byte(0x80,OLED_CMD); //--set divide ratio, Set Clock as 100 Frames/Sec
    OLED_WR_Byte(0xD9,OLED_CMD); //--set pre-charge period
    OLED_WR_Byte(0xf1,OLED_CMD); //Set Pre-Charge as 15 Clocks & Discharge as 1 Clock
    OLED_WR_Byte(0xDA,OLED_CMD); //--set com pins hardware configuration
    OLED_WR_Byte(0x12,OLED_CMD);
    OLED_WR_Byte(0xDB,OLED_CMD); //--set vcomh
    OLED_WR_Byte(0x40,OLED_CMD); //Set VCOM Deselect Level
    OLED_WR_Byte(0x20,OLED_CMD); //-- Set Page Addressing Mode (0x00/0x01/0x02)
    OLED_WR_Byte(0x02,OLED_CMD); //
    OLED_WR_Byte(0x8D,OLED_CMD); //--set Charge Pump enable/disable
    OLED_WR_Byte(0x14,OLED_CMD); //--set(0x10) disable
    OLED_WR_Byte(0xA4,OLED_CMD); // Disable Entire Display On (0xa4/0xa5)
    OLED_WR_Byte(0xA6,OLED_CMD); // Disable Inverse Display On (0xa6/a7)

    OLED_WR_Byte(0xA1,OLED_CMD); //段重定义设置,bit0:0,0->0;1,0->127; 0xA0<->0xA1 这里修改了
    OLED_WR_Byte(0xC0,OLED_CMD); //设置 COM 扫描方向; 0xC0<->0xC1:垂直反向 这是修改了

    OLED_WR_Byte(0xAF,OLED_CMD); //--turn on oled panel

    OLED_WR_Byte(0xAF,OLED_CMD); /*display ON*/
    OLED_Clear();
    OLED_Set_Pos(0,0); //起始坐标设置为 x=0, y=0(PAGE0)
}

这样之后 就可以正常显示数据了。

```

显示图片试试

```
const unsigned char gImage_19_IceRain[128] = { /* 0x12,0x01,0x00,0x20,0x00,0x20, */  
    0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x03,0x07,0x0f,0x1c,0x3c,0x38,0x38,0x30,0x30,  
    0x30,0x30,0x38,0x38,0x3c,0x1c,0x0e,0x07,0x03,0x01,0x00,0x00,0x00,0x00,0x00,0x00,  
    0x07,0x1f,0x3f,0x38,0x70,0x70,0xe0,0xe0,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
    0x00,0x00,0x00,0x00,0x00,0x00,0x80,0xe0,0xe0,0x70,0x70,0x38,0x3f,0x1f,0x07,  
    0xc0,0xf0,0xf8,0x3c,0x1c,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,  
    0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x0e,0x1c,0x3c,0xf8,0x0f,0x0c,  
    0x00,0x00,0x00,0x00,0x00,0x18,0x3c,0x3c,0x18,0x00,0x00,0x00,0x00,0x00,0x18,0x3c,  
    0x3c,0x18,0x00,0x00,0x00,0x00,0x00,0x18,0x3c,0x3c,0x18,0x00,0x00,0x00,0x00,0x00,  
};
```

记住图片取的是 32×32 的模

```
main()
{
    OLED_Init(); //对 OLED 函数先进行初始化

    load_Img(50, 2, OLED_GRAM, gImage_IceRain, 32, 32); //在行 50, 列(页)2 位置, 显示
    //将编码表 oled_asc2_3216[7] 元素 7, 也就是显示数字 7, 放入 OLED_GRAM 缓存。
    //段码表长宽 16x32

    ma_OLED_Refesh_Gram(OLED_GRAM); //将存放数字 7 的缓存, 放入驱动显示。
}
```

测试，显示正常

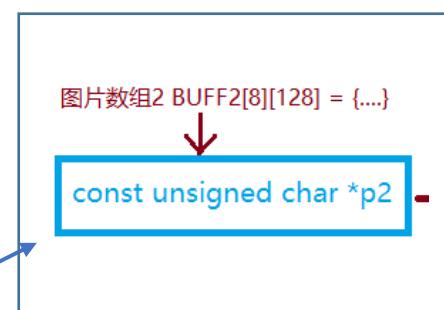
字符/图片滑动显示

图片滑动其实就是两幅图片放在同一个显存里面，显示驱动导入显存，挨着显示。

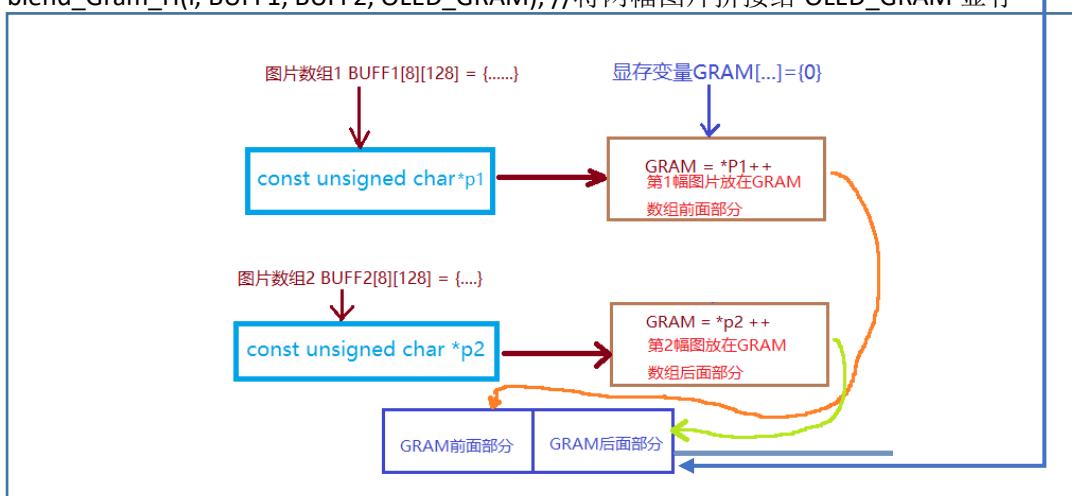
GRAM 显存变量在这里就能发挥它本身的作用了。为什么前面我说要用这么麻烦的方式显示图片，就是用在这里。

```
unsigned char BUFF1[8][128];
```

```
load Img(50, 0, BUFF1, glImage 19 IceRain, 32, 32);
```

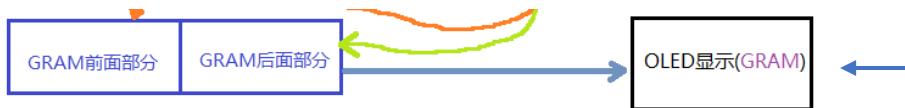


```
blend_Gram_H(i, BUFF1, BUFF2, OLED_GRAM); //将两幅图片拼接给 OLED_GRAM 显存
```



因为要滚动显示,所以 blend_Gram_H 的 x 参要循环增加,把每次增加地址后给 OLED_GRAM 的值,发送给驱动进行显示

```
for(char i = 0; i<128; i++)
{
    blend_Gram_H(i, BUFF1, BUFF2, OLED_GRAM); //循环拼接
    ma_OLED_Refresh_Gram(OLED_GRAM); //驱动显示
    delay_ms(10);
}
```



这样两幅图片就滚动显示了

```
/*
 * 功能: 将两张显存在水平方向混合
 *
 * uint8_t x: 要加载位置的 x 坐标, 0~128
 * uint8_t (*ram1)[128]: 显示界面缓存的二维数组指针
 * uint8_t (*ram2)[128]: 显示界面缓存的二维数组指针
 * uint8_t (*blend)[128]: 混合后输出的显示界面缓存的二维数组指针
 */
void blend_Gram_H(unsigned char x,
                  unsigned char (*ram1)[128],
                  unsigned char (*ram2)[128],
                  unsigned char (*blend)[128])
{
    unsigned char i=0;
    for(i=0; i<128-x; i++)
    {
        blend[0][i] = *(ram1+0)+i+x;
        blend[1][i] = *(ram1+1)+i+x;
        blend[2][i] = *(ram1+2)+i+x;
        blend[3][i] = *(ram1+3)+i+x;
        blend[4][i] = *(ram1+4)+i+x;
        blend[5][i] = *(ram1+5)+i+x;
        blend[6][i] = *(ram1+6)+i+x;
        blend[7][i] = *(ram1+7)+i+x;
    }
    for(i=128-x; i<128; i++)
    {
        blend[0][i] = *(ram2+0)+i-(128-x));
        blend[1][i] = *(ram2+1)+i-(128-x));
        blend[2][i] = *(ram2+2)+i-(128-x));
        blend[3][i] = *(ram2+3)+i-(128-x));
    }
}
```

```

blend[4][i] = *(*(ram2+4)+i-(128-x));
blend[5][i] = *(*(ram2+5)+i-(128-x));
blend[6][i] = *(*(ram2+6)+i-(128-x));
blend[7][i] = *(*(ram2+7)+i-(128-x));
}
}
```

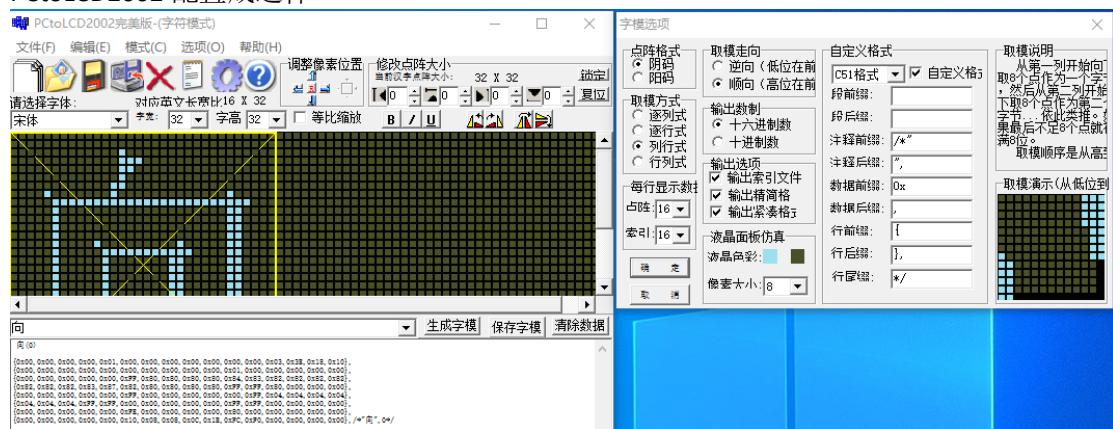
OLED 汉字显示

首先确认我们屏幕初始化时字符刷新的方式

OLED_WR_Byte(0xA1,OLED_CMD); //段重定义设置,bit0<0-->0;1>-127; 0x0A<-->0xA1 这是修改水平颠倒方向
OLED_WR_Byte(0xC0,OLED_CMD); //设置 COM 扫描方向; 0xC0<-->0xC1;垂直反向 这是修改垂直颠倒显示方向

按照这种刷新方式

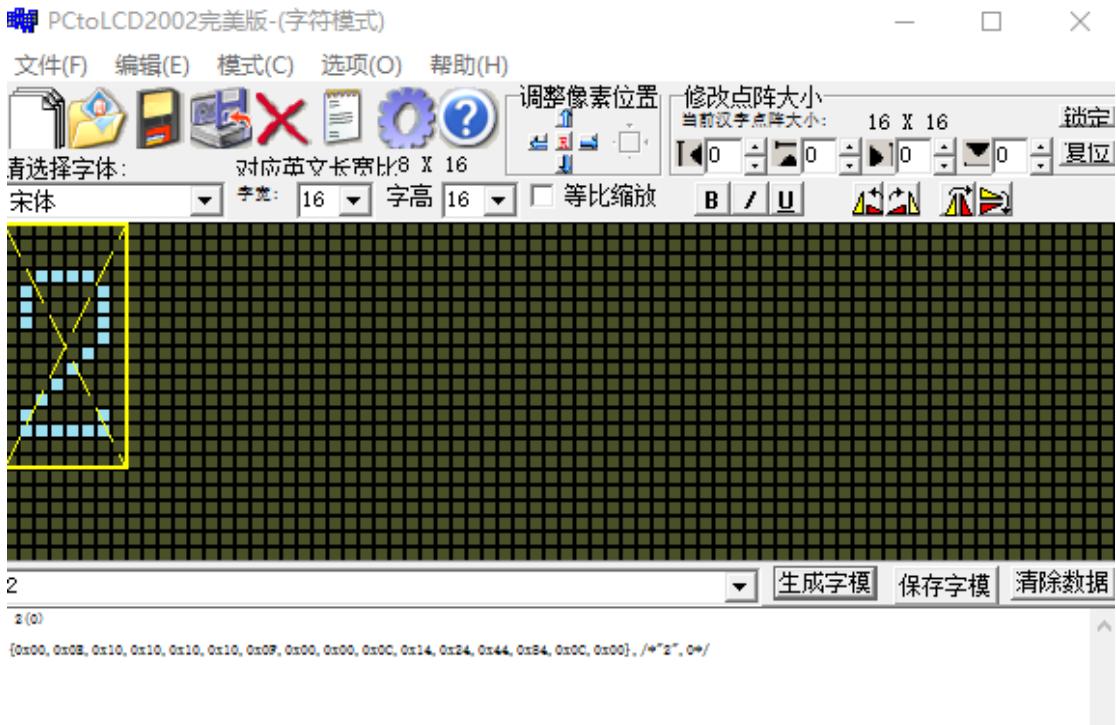
PCtoLCD2002 配置成这样



```
main()
{
    /*OLED_初始化*/
    load_Img(50, 2, OLED_GRAM, xiangarray, 32, 32);
    ma OLED Refresh Gram(OLED GRAM);
```

} 在屏幕中央显示中文没有问题

生成数字 ASCII 字模试试

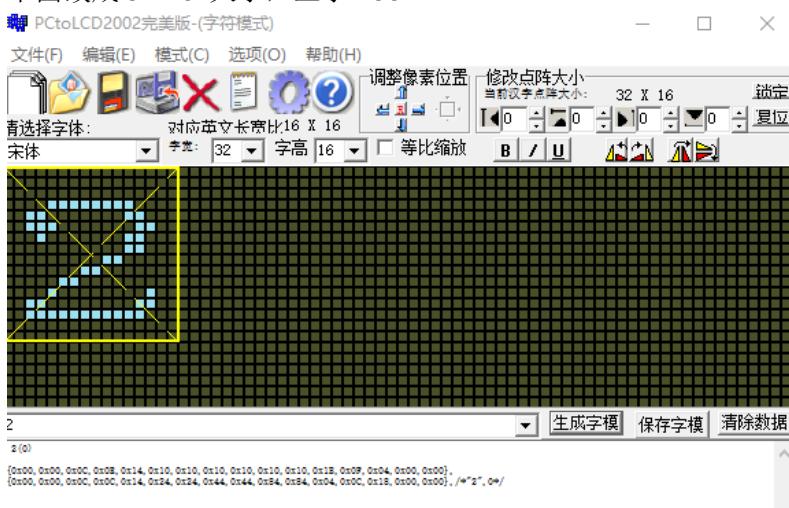


16x16 的 ASCII 就是个一维数组

```
/*数字字符 16x16*/
unsigned char const F16X16[12][32] =
{
    {0x00, 0x07, 0x08, 0x10, 0x10, 0x08, 0x07, 0x00, 0x00, 0xF0, 0x08, 0x04, 0x04, 0x08, 0xF0, 0x00}, /*"0", 0*/
    {0x00, 0x00, 0x08, 0x08, 0x1F, 0x00, 0x00, 0x00, 0x00, 0x04, 0x04, 0xFC, 0x04, 0x04, 0x00}, /*"1", 0*/
    {0x00, 0xE, 0x10, 0x10, 0x10, 0x0F, 0x00, 0x00, 0x0C, 0x14, 0x24, 0x44, 0x84, 0x0C, 0x00}, /*"2", 0*/
    {0x00, 0xC, 0x10, 0x10, 0x11, 0x0E, 0x00, 0x18, 0x04, 0x84, 0x44, 0x84, 0x38, 0x00}, /*"3", 0*/
}
```

但是实际使用发现显示的乱码，字符有错位的情况

下面改成 32x16 大小，显示 ASCII



显示的是 2 维数组

```
unsigned char xiangarray[32] = {
    0x00, 0x00, 0x0C, 0x0E, 0x14, 0x10, 0x10, 0x10, 0x1B, 0x0F, 0x04, 0x00, 0x00,
    0x00, 0x00, 0x0C, 0x0C, 0x14, 0x24, 0x44, 0x84, 0x84, 0x04, 0x0C, 0x18, 0x00, 0x00}; /*2*/
```

实际显示成功。看来 ASCII 码字符不能 16x16 这种 1:1 的显示方式，宽一定要是高的 2 倍。

实际上字符错误不是 1:1 的问题，是我使用的 `load_Img` 加载算法问题。将 16x16 的字符改成 8x16。但是取模数组还是保持 16x16。(如果是中文字符，那还是使用 16x16 显示方式)

```
unsigned char xiangarray[32] = {  
0x00,0x07,0x08,0x10,0x10,0x08,0x07,0x00,0x00,0xF0,0x08,0x04,0x04,0x08,0xF0,0x00}/*0*/
```

```
load_Img(50, 2, OLED_GRAM, xiangarray, 8, 16); //数组还是 16x16 的数据，只是 load_img 函数  
改成 8x16 显示，数字就正常了。  
ma_OLED_Refresh_Gram(OLED_GRAM);
```

显示成功

但是注意，在中文字符显示的时候，还是要 16x16，1:1 输入才正常

连续中文字符 16x16 显示方式

```
const unsigned char F16x16_TempDot[8][32] =  
{  
{0x08,0x06,0x40,0x31,0x00,0x7F,0x40,0x4F,0x40,0x7F,0x00,0x1F,0x00,0xFF,0x00,0x00,  
0x20,0x20,0x7E,0x80,0x01,0xE2,0x0C,0xF0,0x08,0xE4,0x00,0xE2,0x01,0xFE,0x00,0x00},/*"        测  
",0*/  
{0x08,0x06,0x40,0x31,0x00,0x00,0x7F,0x49,0x49,0x49,0x49,0x49,0x7F,0x00,0x00,0x00,  
0x20,0x20,0x7E,0x80,0x02,0x7E,0x42,0x42,0x7E,0x42,0x42,0x42,0x7E,0x02,0x00},/*"        温  
",1*/  
};
```

生成 2 维数组，用下标方式，一个字符一个字符显示

```
load_Img(50, 2, OLED_GRAM, F16x16_TempDot[0], 16, 16); //显示‘测’  
ma_OLED_Refresh_Gram(OLED_GRAM);
```

```
load_Img(50, 2, OLED_GRAM, F16x16_TempDot[1], 16, 16); //显示‘温’  
ma_OLED_Refresh_Gram(OLED_GRAM);
```

在切换屏幕页面的时候发现缓存遗留问题 `unsigned char OLED_GRAM[8][128];`

有可能是OLED屏特殊原因，做界面翻页的时候出现如下情况：



这是因为全局显存 `OLED_GRAM[8][128]` 前半部分保留了 `12345`，后半部分，加载的新数据 `ADCDEF`。如果要完全清除 `12345`，就必须把 `ADCDEF` 坐标和 `12345` 坐标重合。这样做不科学

所以在加载显存之前，用 `memset` 进行显存清0

我是按键切换菜单，所以在按键按下之后，执行 `memset()`

```
button_fb() //按键按下，执行按键回调函数  
{  
    memset(OLED_CharGRAM, 0, sizeof(OLED_CharGRAM)); //清空显示缓存  
    OLED_Clear(); //清屏  
    //再次加载新图片，进行显示  
    load_Img(x+32, y, OLED_CharGRAM, oled_asc2_3216[Tens], 16, 32);  
    ma_OLED_Refresh_Gram(OLED_CharGRAM);  
}
```