LVGL Documentation v7.10.1

Contributors of LVGL

Aug 06, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

LVGL (Light and Versatile Graphics Library) is a free and open-source graphics library providing everything you need to create embedded GUI with easy-to-use graphical elements, beautiful visual effects and low memory footprint.

1.1 Key features

- Powerful building blocks such as buttons, charts, lists, sliders, images etc.
- Advanced graphics with animations, anti-aliasing, opacity, smooth scrolling
- Various input devices such as touchpad, mouse, keyboard, encoder etc.
- Multi-language support with UTF-8 encoding
- Multi-display support, i.e. use more TFT, monochrome displays simultaneously
- Fully customizable graphic elements
- Hardware independent to use with any microcontroller or display
- Scalable to operate with little memory (64 kB Flash, 16 kB RAM)
- OS, External memory and GPU supported but not required
- Single frame buffer operation even with advanced graphical effects
- Written in C for maximal compatibility (C++ compatible)
- Simulator to start embedded GUI design on a PC without embedded hardware
- Binding to MicroPython
- Tutorials, examples, themes for rapid GUI design
- Documentation is available as online and offline
- Free and open-source under MIT license

1.2 Requirements

Basically, every modern controller (which is able to drive a display) is suitable to run LVGL. The minimal requirements are:

1.3 License

The LVGL project (including all repositories) is licensed under MIT license. It means you can use it even in commercial projects.

It's not mandatory but we highly appreciate it if you write a few words about your project in the My projects category of the Forum or a private message from lvgl.io.

Although you can get LVGL for free there is a huge work behind it. It's created by a group of volunteers who made it available for you in their free time.

To make the LVGL project sustainable, please consider *Contributing* to the project. You can choose from *many ways of contributions* such as simply writing a tweet about you are using LVGL, fixing bugs, translating the documentation, or even becoming a maintainer.

1.4 Repository layout

All repositories of the LVGL project are hosted n GitHub: https://github.com/lvgl

You fill these repositories there:

- lvgl The library itself
- lv_examples Examples and demos
- lv_drivers Display and input device drivers
- docs Source of the documentation's site (https://docs.lvgl.io)
- blog Source of the blog's site (https://blog.lvgl.io)
- sim Source of the online simulator's site (https://sim.lvgl.io)
- lv_sim_... Simulator projects for various IDEs and platforms
- lv_port_... LVGL ports to development boards
- lv_binding_.. Bindings to other languages
- lv_... Ports to other platforms

The lvgl, lv_examples and lv_drivers are the core repositories which gets the most attentions regarding maintenance.

1.5 Release policy

The core repositories follow the rules of Semantic versioning:

- Major versions for incompatible API changes. E.g. v5.0.0, v6.0.0
- Minor version for new but backward-compatible functionalities. E.g. v6.1.0, v6.2.0
- Patch version for backward-compatible bug fixes. E.g. v6.1.1, v6.1.2

1.5.1 Branches

The core repositories have at least the following branches:

- `master` latest version, patches are merged directly here.
- `dev` merge new features here until they are merged into `master`.
- `release/vX` stable versions of the major releases

1.5.2 Release cycle

LVGL has 2 weeks release cycle. On every first and third Tuesday of a month:

1. A major, minor or bug fix release is created (based on the new features) from the `master` branch
2. `master` is merged into `release/vX`
3. Immediately after the release `dev` is merged into `master`
4. In the upcoming 2 weeks the new features in `master` can be tested
5. Bug fixes are merged directly into `master`
6. After 2 weeks start again from the first point

1.5.3 Tags

Tags like `vX.Y.Z` are created for every release.

1.5.4 Changelog

The changes are recorded in CHANGELOG.md.

1.5.5 Side projects

The docs is rebuilt on every release. By default, the `latest` documentation is displayed which is for the current `master` branch of lvgl. The documentation of earlier versions is available from the menu on the left.

The simulator, porting, and other projects are updated with best effort. Pull requests are welcome if you updated one of them.

1.5.6 Version support

In the core repositories each major version has a branch (e.g. `release/v6`). All the minor and patch releases of that major version are merged there.

It makes possible to add fixed older versions without bothering the newer ones.

All major versions are officially supported for 1 year.

1.6 FAQ

1.6.1 Where can I ask questions?

You can ask questions in the Forum: https://forum.lvgl.io/.

We use GitHub issues for development related discussion. So you should use them only if your question or issue is tightly related to the development of the library.

1.6.2 Is my MCU/hardware supported?

Every MCU which is capable of driving a display via Parallel port, SPI, RGB interface or anything else and fulfills the *Requirements* is supported by LLVGL.

It includes:

- "Common" MCUs like STM32F, STM32H, NXP Kinetis, LPC, iMX, dsPIC33, PIC32 etc.
- Bluetooth, GSM, WiFi modules like Nordic NRF and Espressif ESP32
- Linux frame buffer like /dev/fb0 which includes Single-board computers too like Raspberry Pi
- And anything else with a strong enough MCU and a periphery to drive a display

1.6.3 Is my display supported?

LVGL needs just one simple driver function to copy an array of pixels into a given area of the display. If you can do this with your display then you can use that display with LVGL.

Some examples of the supported display types:

- TFTs with 16 or 24 bit color depth
- Monitors with HDMI port
- Small monochrome displays
- Gray-scale displays
- even LED matrices
- or any other display where you can control the color/state of the pixels

See the *Porting* section to learn more.

1.6.4 Nothing happens, my display driver is not called. What have I missed?

Be sure you are calling `lv_tick_inc(x)` in an interrupt and `lv_task_handler()` in your main `while(1)`.

Learn more in the *Tick* and *Task handler* section.

1.6.5 Why the display driver is called only once? Only the upper part of the display is refreshed.

Be sure you are calling `lv_disp_flush_ready(drv)` at the end of your "*display flush callback*".

1.6.6 Why I see only garbage on the screen?

Probably there a bug in your display driver. Try the following code without using LVGL. You should see a square with red-blue gradient

```
#define BUF_W 20
#define BUF_H 10

lv_color_t buf[BUF_W * BUF_H];
lv_color_t * buf_p = buf;
uint16_t x, y;
for(y = 0; y &lt; BUF_H; y++) {
    lv_color_t c = lv_color_mix(LV_COLOR_BLUE, LV_COLOR_RED, (y * 255) / BUF_H);
    for(x = 0; x &lt; BUF_W; x++){
        (*buf_p) =  c;
        buf_p++;
    }
}

lv_area_t a;
a.x1 = 10;
a.y1 = 40;
a.x2 = a.x1 + BUF_W - 1;
a.y2 = a.y1 + BUF_H - 1;
my_flush_cb(NULL, &a, buf);
```

1.6.7 Why I see non-sense colors on the screen?

Probably LVGL's color format is not compatible with your displays color format. Check `LV_COLOR_DEPTH` in *lv_conf.h*.

If you are using 16 bit colors with SPI (or other byte-oriented interface) probably you need to set `LV_COLOR_16_SWAP  1` in *lv_conf.h*. It swaps the upper and lower bytes of the pixels.

1.6.8 How to speed up my UI?

- Turn on compiler optimization and enable cache if your MCU has
- Increase the size of the display buffer
- Use 2 display buffers and flush the buffer with DMA (or similar periphery) in the background
- Increase the clock speed of the SPI or Parallel port if you use them to drive the display
- If your display has SPI port consider changing to a model with parallel because it has much higher throughput
- Keep the display buffer in the internal RAM (not in external SRAM) because LVGL uses it a lot and it should have a small access time

1.6.9 How to reduce flash/ROM usage?

You can disable all the unused features (such as animations, file system, GPU etc.) and object types in *lv_conf.h*.

If you are using GCC you can add

- `-fdata-sections -ffunction-sections` compiler flags
- `--gc-sections` linker flag

to remove unused functions and variables from the final binary

1.6.10 How to reduce the RAM usage

- Lower the size of the *Display buffer*
- Reduce `LV_MEM_SIZE` in *lv_conf.h*. This memory used when you create objects like buttons, labels, etc.
- To work with lower `LV_MEM_SIZE` you can create the objects only when required and deleted them when they are not required anymore

1.6.11 How to work with an operating system?

To work with an operating system where tasks can interrupt each other (preemptive) you should protect LVGL related function calls with a mutex. See the *Operating system and interrupts* section to learn more.

CHAPTER

TWO

GET STARTED

There are several ways to get your feet wet with LVGL. This list shows the recommended way of learning the library:

1. Check the Online demos to see LVGL in action (3 minutes)
2. Read the Introduction page of the documentation (5 minutes)
3. Read the Quick overview page of the documentation (15 minutes)
4. Set up a Simulator (10 minutes)
5. Try out some Examples
6. Port LVGL to a board. See the Porting guide or check the ready to use Projects
7. Read the Overview page to get a better understanding of the library. (2-3 hours)
8. Check the documentation of the Widgets to see their features and usage
9. If you have questions got to the Forum
10. Read the Contributing guide to see how you can help to improve LVGL (15 minutes)

2.1 Quick overview

Here you can learn the most important things about LVGL. You should read it first to get a general impression and read the detailed *Porting* and *Overview* sections after that.

2.1.1 Get started in a simulator

Instead of porting LVGL to an embedded hardware, it's highly recommended to get started in a simulator first.

LVGL is ported to many IDEs to be sure you will find your faviourite one. Go to *Simulators* to get ready-to-use projects which can be run on your PC. This way you can save the porting for now and make some experience with LVGL immediately.

2.1.2 Add LVGL into your project

The following steps show how to setup LVGL on an embedded system with a display and a touchpad.

- Download or Clone the library from GitHub with `git clone https://github.com/lvgl/lvgl.git`
- Copy the `lvgl` folder into your project
- Copy `lvgl/lv_conf_template.h` as `lv_conf.h` next to the `lvgl` folder, change the first `#if 0` to `1` to enable the file's content and set at least `LV_HOR_RES_MAX`, `LV_VER_RES_MAX` and `LV_COLOR_DEPTH` defines.
- Include `lvgl/lvgl.h` where you need to use LVGL related functions.
- Call `lv_tick_inc(x)` every `x` milliseconds **in a Timer or Task** (`x` should be between 1 and 10). It is required for the internal timing of LVGL. Alternatively, configure `LV_TICK_CUSTOM` (see `lv_conf.h`) so that LVGL can retrieve the current time directly.
- Call `lv_init()`
- Create a display buffer for LVGL. LVGL will render the graphics here first, and seed the rendered image to the display. The buffer size can be set freely but 1/10 screen size is a good starting point.

```
static lv_disp_buf_t disp_buf;
static lv_color_t buf[LV_HOR_RES_MAX * LV_VER_RES_MAX / 10];                    /
↪*Declare a buffer for 1/10 screen size*/
lv_disp_buf_init(&disp_buf, buf, NULL, LV_HOR_RES_MAX * LV_VER_RES_MAX / 10);    /
↪*Initialize the display buffer*/
```

- Implement and register a function which can **copy the rendered image** to an area of your display:

```
lv_disp_drv_t disp_drv;               /*Descriptor of a display driver*/
lv_disp_drv_init(&disp_drv);          /*Basic initialization*/
disp_drv.flush_cb = my_disp_flush;    /*Set your driver function*/
disp_drv.buffer = &disp_buf;          /*Assign the buffer to the display*/
lv_disp_drv_register(&disp_drv);      /*Finally register the driver*/

void my_disp_flush(lv_disp_drv_t * disp, const lv_area_t * area, lv_color_t * color_p)
{
    int32_t x, y;
    for(y = area->y1; y <= area->y2; y++) {
        for(x = area->x1; x <= area->x2; x++) {
            set_pixel(x, y, *color_p);  /* Put a pixel to the display.*/
            color_p++;
        }
    }

    lv_disp_flush_ready(disp);         /* Indicate you are ready with the flushing*/
}
```

- Implement and register a function which can **read an input device**. E.g. for a touch pad:

```
lv_indev_drv_t indev_drv;                  /*Descriptor of a input device driver*/
lv_indev_drv_init(&indev_drv);             /*Basic initialization*/
indev_drv.type = LV_INDEV_TYPE_POINTER;    /*Touch pad is a pointer-like device*/
indev_drv.read_cb = my_touchpad_read;      /*Set your driver function*/
lv_indev_drv_register(&indev_drv);         /*Finally register the driver*/

bool my_touchpad_read(lv_indev_t * indev, lv_indev_data_t * data)
```

(continues on next page)

(continued from previous page)

```
{
    data->state = touchpad_is_pressed() ? LV_INDEV_STATE_PR : LV_INDEV_STATE_REL;
    if(data->state == LV_INDEV_STATE_PR) touchpad_get_xy(&data->point.x, &data->point.
↪y);

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

- Call `lv_task_handler()` periodically every few milliseconds in the main `while(1)` loop, in Timer interrupt or in an Operation system task. It will redraw the screen if required, handle input devices etc.

For a more detailed guide go to the Porting section.

2.1.3 Learn the basics

Widgets

The graphical elements like Buttons, Labels, Sliders, Charts etc are called objects or widgets in LVGL. Go to *Widgets* to see the full list of available widgets.

Every object has a parent object where it is create. For example if a label is created on a button, the button is the parent of label. The child object moves with the parent and if the parent is deleted the children will be deleted too.

Children can be visible only on their parent. It other words, the parts of the children out of the parent are clipped.

A *screen* is the "root" parent. You can have any number of screens. To get the current screen call `lv_scr_act()`, and to load a screen use `lv_scr_load(scr1)`.

You can create a new object with `lv_<type>_create(parent, obj_to_copy)`. It will return an `lv_obj_t *` variable which should be used as a reference to the object to set its parameters. The first parameter is the desired *parent*, the second parameters can be an object to copy (`NULL` if unused). For example:

```
lv_obj_t * slider1 = lv_slider_create(lv_scr_act(), NULL);
```

To set some basic attribute `lv_obj_set_<paramters_name>(obj, <value>)` function can be used. For example:

```
lv_obj_set_x(btn1, 30);
lv_obj_set_y(btn1, 10);
lv_obj_set_size(btn1, 200, 50);
```

The objects has type specific parameters too which can be set by `lv_<type>_set_<paramters_name>(obj, <value>)` functions. For example:

```
lv_slider_set_value(slider1, 70, LV_ANIM_ON);
```

To see the full API visit the documentation of the widgets or the related header file (e.g. lvgl/src/lv_widgets/lv_slider.h).

Events

Events are used to inform the user if something has happened with an object. You can assign a callback to an object which will be called if the object is clicked, released, dragged, being deleted etc. It should look like this:

```
lv_obj_set_event_cb(btn, btn_event_cb);                 /*Assign a callback to the␣
↪button*/

...

void btn_event_cb(lv_obj_t * btn, lv_event_t event)
{
    if(event == LV_EVENT_CLICKED) {
        printf("Clicked\n");
    }
}
```

Learn more about the events in the *Event overview* section.

Parts

Widgets might be built from one or more parts. For example a button has only one part called `LV_BTN_PART_MAIN`. However, a *Page* has `LV_PAGE_PART_BG`, `LV_PAGE_PART_SCROLLABLE`, `LV_PAGE_PART_SCROLLBAR` and `LV_PAGE_PART_EDGE_FLASG`.

Some parts are *virtual* (they are not real object, just drawn on the fly, such as the scrollbar of a page) but other parts are *real* (they are real object, such as the scrollable part of the page).

Parts come into play when you want to set the styles and states of a given part of an object. (See below)

States

The objects can be in a combination of the following states:

- **LV_STATE_DEFAULT** Normal, released
- **LV_STATE_CHECKED** Toggled or checked
- **LV_STATE_FOCUSED** Focused via keypad or encoder or clicked via touchpad/mouse
- **LV_STATE_EDITED** Edit by an encoder
- **LV_STATE_HOVERED** Hovered by mouse (not supported now)
- **LV_STATE_PRESSED** Pressed
- **LV_STATE_DISABLED** Disabled or inactive

For example, if you press an object it will automatically get the `LV_STATE_PRESSED` state and when you release it, the state will be removed.

To get the current state use `lv_obj_get_state(obj, part)`. It will return the `OR`ed states. For example, this is a valid state for a checkbox: `LV_STATE_CHECKED | LV_STATE_PRESSED | LV_STATE_FOCUSED`

Styles

Styles can be assigned to the parts of an object to change their appearance. A style can describe for example the background color, border width, text font and so on. See the full list here.

The styles can be cascaded (similarly to CSS). It means you can add more styles to a part of an object. For example `style_btn` can set a default button appearance, and `style_btn_red` can overwrite some properties to make the button red-

Every style property you set is specific to a state. For example, you can set a different background color for `LV_STATE_DEFAULT` and `LV_STATE_PRESSED`. The library finds the best match between the state of the given part and the available style properties. For example if the object is in pressed state and the border width is specified for pressed state, then it will be used. However, if it's not specified for pressed state, the `LV_STATE_DEFAULT`'s border width will be used. If the border width not defined for `LV_STATE_DEFAULT` either, a default value will be used.

Some properties (typically the text-related ones) can be inherited. It means if a property is not set in an object it will be searched in its parents too. For example you can set the font once in the screen's style and every text will inherit it by default.

Local style properties also can be added to the objects.

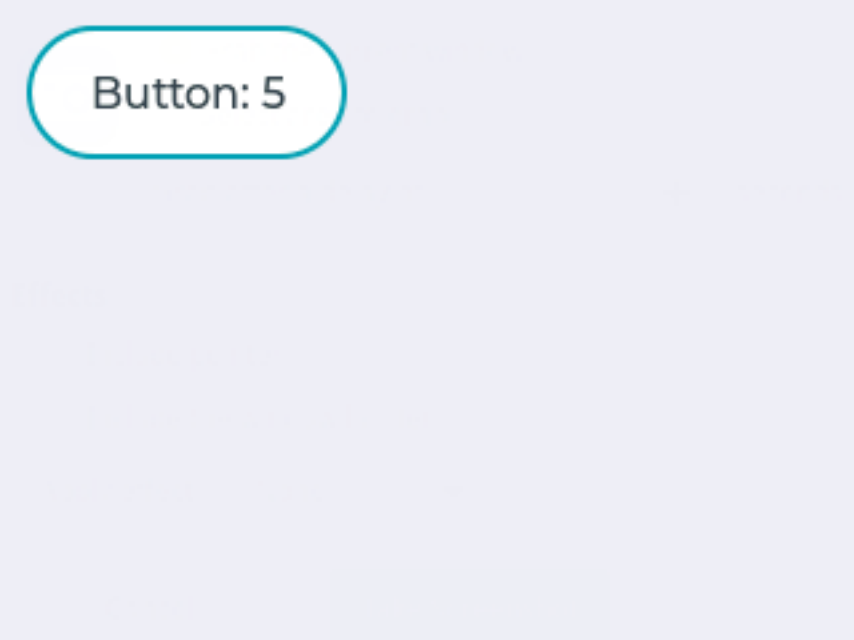
Themes

Themes are the default styles of the objects. The styles from the themes are applied automatically when the objects are created.

You can select the theme to use in `lv_conf.h`.

2.1.4 Examples

Button with label



```c
#include "../../lv_examples.h"

static void btn_event_cb(lv_obj_t * btn, lv_event_t event)
{
    if(event == LV_EVENT_CLICKED) {
        static uint8_t cnt = 0;
        cnt++;

        /*Get the first child of the button which is the label and change its text*/
        lv_obj_t * label = lv_obj_get_child(btn, NULL);
        lv_label_set_text_fmt(label, "Button: %d", cnt);
    }
}

/**
 * Create a button with a label and react on Click event.
 */
void lv_ex_get_started_1(void)
{
    lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);     /*Add a button the␣
↪current screen*/
    lv_obj_set_pos(btn, 10, 10);                            /*Set its position*/
    lv_obj_set_size(btn, 120, 50);                          /*Set its size*/
    lv_obj_set_event_cb(btn, btn_event_cb);                 /*Assign a callback to␣
↪the button*/

    lv_obj_t * label = lv_label_create(btn, NULL);          /*Add a label to the␣
↪button*/
    lv_label_set_text(label, "Button");                     /*Set the labels text*/
}
```

Styling buttons

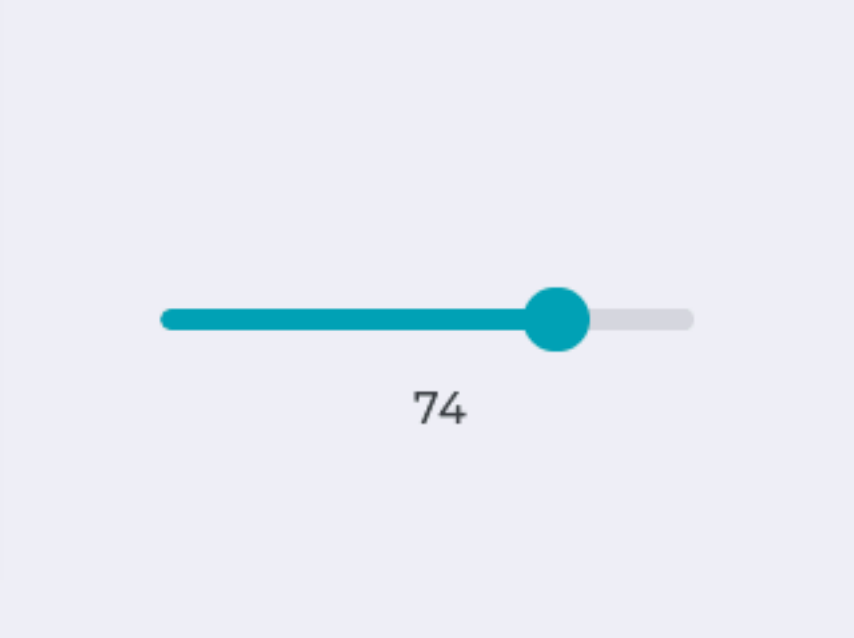
```
#include "../../lv_examples.h"


/**
 * Create styles from scratch for buttons.
 */
void lv_ex_get_started_2(void)
{
    static lv_style_t style_btn;
    static lv_style_t style_btn_red;

    /*Create a simple button style*/
    lv_style_init(&style_btn);
    lv_style_set_radius(&style_btn, LV_STATE_DEFAULT, 10);
    lv_style_set_bg_opa(&style_btn, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_bg_grad_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_GRAY);
    lv_style_set_bg_grad_dir(&style_btn, LV_STATE_DEFAULT, LV_GRAD_DIR_VER);

    /*Swap the colors in pressed state*/
    lv_style_set_bg_color(&style_btn, LV_STATE_PRESSED, LV_COLOR_GRAY);
    lv_style_set_bg_grad_color(&style_btn, LV_STATE_PRESSED, LV_COLOR_SILVER);

    /*Add a border*/
    lv_style_set_border_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_WHITE);
    lv_style_set_border_opa(&style_btn, LV_STATE_DEFAULT, LV_OPA_70);
    lv_style_set_border_width(&style_btn, LV_STATE_DEFAULT, 2);

    /*Different border color in focused state*/
    lv_style_set_border_color(&style_btn, LV_STATE_FOCUSED, LV_COLOR_BLUE);
    lv_style_set_border_color(&style_btn, LV_STATE_FOCUSED | LV_STATE_PRESSED, LV_
↪COLOR_NAVY);

    /*Set the text style*/
    lv_style_set_text_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_WHITE);

    /*Make the button smaller when pressed*/
    lv_style_set_transform_height(&style_btn, LV_STATE_PRESSED, -5);
    lv_style_set_transform_width(&style_btn, LV_STATE_PRESSED, -10);
#if LV_USE_ANIMATION
    /*Add a transition to the size change*/
    static lv_anim_path_t path;
    lv_anim_path_init(&path);
    lv_anim_path_set_cb(&path, lv_anim_path_overshoot);

    lv_style_set_transition_prop_1(&style_btn, LV_STATE_DEFAULT, LV_STYLE_TRANSFORM_
↪HEIGHT);
    lv_style_set_transition_prop_2(&style_btn, LV_STATE_DEFAULT, LV_STYLE_TRANSFORM_
↪WIDTH);
    lv_style_set_transition_time(&style_btn, LV_STATE_DEFAULT, 300);
    lv_style_set_transition_path(&style_btn, LV_STATE_DEFAULT, &path);
#endif

    /*Create a red style. Change only some colors.*/
    lv_style_init(&style_btn_red);
    lv_style_set_bg_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_RED);
    lv_style_set_bg_grad_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_MAROON);
```

(continues on next page)

(continued from previous page)

```
    lv_style_set_bg_color(&style_btn_red, LV_STATE_PRESSED, LV_COLOR_MAROON);
    lv_style_set_bg_grad_color(&style_btn_red, LV_STATE_PRESSED, LV_COLOR_RED);
    lv_style_set_text_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_WHITE);
#if LV_USE_BTN
    /*Create buttons and use the new styles*/
    lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);     /*Add a button the␣
↪current screen*/
    lv_obj_set_pos(btn, 10, 10);                            /*Set its position*/
    lv_obj_set_size(btn, 120, 50);                          /*Set its size*/
    lv_obj_reset_style_list(btn, LV_BTN_PART_MAIN);         /*Remove the styles␣
↪coming from the theme*/
    lv_obj_add_style(btn, LV_BTN_PART_MAIN, &style_btn);

    lv_obj_t * label = lv_label_create(btn, NULL);          /*Add a label to the␣
↪button*/
    lv_label_set_text(label, "Button");                     /*Set the labels text*/

    /*Create a new button*/
    lv_obj_t * btn2 = lv_btn_create(lv_scr_act(), btn);
    lv_obj_set_pos(btn2, 10, 80);
    lv_obj_set_size(btn2, 120, 50);                            /*Set its size*/
    lv_obj_reset_style_list(btn2, LV_BTN_PART_MAIN);         /*Remove the styles␣
↪coming from the theme*/
    lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_btn);
    lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_btn_red);   /*Add the red style␣
↪on top of the current */
    lv_obj_set_style_local_radius(btn2, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, LV_RADIUS_
↪CIRCLE); /*Add a local style*/

    label = lv_label_create(btn2, NULL);          /*Add a label to the button*/
    lv_label_set_text(label, "Button 2");                     /*Set the labels text*/
#endif
}
```

Slider and alignment




```
#include "../../lv_examples.h"

static lv_obj_t * label;

static void slider_event_cb(lv_obj_t * slider, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        /*Refresh the text*/
        lv_label_set_text_fmt(label, "%d", lv_slider_get_value(slider));
    }
}

/**
 * Create a slider and write its value on a label.
 */
void lv_ex_get_started_3(void)
{
    /* Create a slider in the center of the display */
    lv_obj_t * slider = lv_slider_create(lv_scr_act(), NULL);
    lv_obj_set_width(slider, 200);                          /*Set the width*/
    lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0);    /*Align to the center of␣
↪the parent (screen)*/
    lv_obj_set_event_cb(slider, slider_event_cb);         /*Assign an event function*/

    /* Create a label below the slider */
    label = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(label, "0");
    lv_obj_set_auto_realign(slider, true);                                /*To keep center␣
↪alignment when the width of the text changes*/
    lv_obj_align(label, slider, LV_ALIGN_OUT_BOTTOM_MID, 0, 20);    /*Align below the␣
↪slider*/
}
```

(continues on next page)

(continued from previous page)

2.1.5 Micropython

Learn more about *Micropython*.

```
# Create a Button and a Label
scr = lv.obj()
btn = lv.btn(scr)
btn.align(lv.scr_act(), lv.ALIGN.CENTER, 0, 0)
label = lv.label(btn)
label.set_text("Button")

# Load the screen
lv.scr_load(scr)
```

2.2 Simulator on PC

You can try out the LVGL **using only your PC** (i.e. without any development boards). The LVGL will run on a simulator environment on the PC where anyone can write and experiment the real LVGL applications.

Simulator on the PC have the following advantages:

- Hardware independent - Write a code, run it on the PC and see the result on the PC monitor.
- Cross-platform - Any Windows, Linux or OSX PC can run the PC simulator.
- Portability - the written code is portable, which means you can simply copy it when using an embedded hardware.
- Easy Validation - The simulator is also very useful to report bugs because it means common platform for every user. So it's a good idea to reproduce a bug in simulator and use the code snippet in the Forum.

2.2.1 Select an IDE

The simulator is ported to various IDEs (Integrated Development Environments). Choose your favorite IDE, read its README on GitHub, download the project, and load it to the IDE.

- Eclipse with SDL driver: Recommended on Linux and Mac
- CodeBlocks: Recommended on Windows
- VisualStudio with SDL driver: For Windows
- VSCode with SDL driver: Recommended on Linux and Mac
- PlatformIO with SDL driver: Recommended on Linux and Mac

You can use any IDEs for the development but, for simplicity, the configuration for Eclipse CDT is focused in this tutorial. The following section describes the set-up guide of Eclipse CDT in more details.

Note: If you are on Windows, it's usually better to use the Visual Studio or CodeBlocks projects instead. They work out of the box without requiring extra steps.

2.2.2 Set-up Eclipse CDT

Install Eclipse CDT

Eclipse CDT is a C/C++ IDE.

Eclipse is a Java based software therefore be sure **Java Runtime Environment** is installed on your system.

On Debian-based distros (e.g. Ubuntu): `sudo apt-get install default-jre`

Note: If you are using other distros, then please refer and install 'Java Runtime Environment' suitable to your distro. Note: If you are using macOS and get a "Failed to create the Java Virtual Machine" error, uninstall any other Java JDK installs and install Java JDK 8u. This should fix the problem.

You can download Eclipse's CDT from: https://www.eclipse.org/cdt/downloads.php. Start the installer and choose *Eclipse CDT* from the list.

Install SDL 2

The PC simulator uses the SDL 2 cross platform library to simulate a TFT display and a touch pad.

Linux

On **Linux** you can easily install SDL2 using a terminal:

1. Find the current version of SDL2: `apt-cache search libsdl2 (e.g. libsdl2-2.0-0)`
2. Install SDL2: `sudo apt-get install libsdl2-2.0-0` (replace with the found version)
3. Install SDL2 development package: `sudo apt-get install libsdl2-dev`
4. If build essentials are not installed yet: `sudo apt-get install build-essential`

Windows

If you are using **Windows** firstly you need to install MinGW (64 bit version). After installing MinGW, do the following steps to add SDL2:

1. Download the development libraries of SDL.Go to https://www.libsdl.org/download-2.0.php and download *Development Libraries: SDL2-devel-2.0.5-mingw.tar.gz*
2. Decompress the file and go to *x86_64-w64-mingw32* directory (for 64 bit MinGW) or to *i686-w64-mingw32* (for 32 bit MinGW)
3. Copy *_...mingw32/include/SDL2* folder to *C:/MinGW/.../x86_64-w64-mingw32/include*
4. Copy *_...mingw32/lib/* content to *C:/MinGW/.../x86_64-w64-mingw32/lib*
5. Copy *_...mingw32/bin/SDL2.dll* to *{eclipse_worksapce}/pc_simulator/Debug/*. Do it later when Eclipse is installed.

Note: If you are using **Microsoft Visual Studio** instead of Eclipse then you don't have to install MinGW.

OSX

On **OSX** you can easily install SDL2 with brew: `brew install sdl2`

If something is not working, then please refer this tutorial to get started with SDL.

Pre-configured project

A pre-configured graphics library project (based on the latest release) is always available to get started easily. You can find the latest one on GitHub. (Please note that, the project is configured for Eclipse CDT).

Add the pre-configured project to Eclipse CDT

Run Eclipse CDT. It will show a dialogue about the **workspace path**. Before accepting the path, check that path and copy (and unzip) the downloaded pre-configured project there. After that, you can accept the workspace path. Of course you can modify this path but, in that case copy the project to the corresponding location.

Close the start up window and go to **File->Import** and choose **General->Existing project into Workspace**. **Browse the root directory** of the project and click **Finish**

On **Windows** you have to do two additional things:

- Copy the **SDL2.dll** into the project's Debug folder
- Right click on the project -> Project properties -> C/C++ Build -> Settings -> Libraries -> Add ... and add *mingw32* above SDLmain and SDL. (The order is important: mingw32, SDLmain, SDL)

Compile and Run

Now you are ready to run the LVGL Graphics Library on your PC. Click on the Hammer Icon on the top menu bar to Build the project. If you have done everything right, then you will not get any errors. Note that on some systems additional steps might be required to "see" SDL 2 from Eclipse but, in most of cases the configurations in the downloaded project is enough.

After a success build, click on the Play button on the top menu bar to run the project. Now a window should appear in the middle of your screen.

Now everything is ready to use the LVGL in the practice or begin the development on your PC.

2.3 STM32

TODO

2.4 NXP

NXP has integrated LVGL into the MCUXpresso SDK packages for several of their general purpose and crossover microcontrollers, allowing easy evaluation and migration into your product design. Download an SDK for a supported board today and get started with your next GUI application.

2.4.1 Creating new project with LVGL

Downloading the MCU SDK example project is recommended as a starting point. It comes fully configured with LVGL (and with PXP support if module is present), no additional integration work is required.

2.4.2 Adding HW acceleration for NXP iMX RT platforms using PXP (PiXel Pipeline) engine for existing projects

Several drawing features in LVGL can be offloaded to PXP engine. In order to use CPU time while PXP is running, RTOS is required to block the LVGL drawing thread and switch to another task, or simply to idle task, where CPU could be suspended to save power.

Features supported:

- RGB565 color format
- Area fill + optional transparency
- BLIT (BLock Image Transfer) + optional transparency
- Color keying + optional transparency
- Recoloring (color tint) + optional transparency
- RTOS integration layer
- Default FreeRTOS and bare metal code provided

Basic configuration:

- Select NXP PXP engine in lv_conf.h: Set `LV_USE_GPU_NXP_PXP` to 1
- Enable default implementation for interrupt handling, PXP start function and automatic initialization: Set `LV_USE_GPU_NXP_PXP_AUTO_INIT` to 1
- If `FSL_RTOS_FREE_RTOS` symbol is defined, FreeRTOS implementation will be used, otherwise bare metal code will be included

Basic initialization:

- If `LV_USE_GPU_NXP_PXP_AUTO_INIT` is enabled, no user code is required; PXP is initialized automatically in `lv_init()`
- For manual PXP initialization, default configuration structure for callbacks can be used. Initialize PXP before calling `lv_init()`

```
#if LV_USE_GPU_NXP_PXP
  #include "lv_gpu/lv_gpu_nxp_pxp.h"
  #include "lv_gpu/lv_gpu_nxp_pxp_osa.h"
#endif
. . .
#if LV_USE_GPU_NXP_PXP
  if (lv_gpu_nxp_pxp_init(&pxp_default_cfg) != LV_RES_OK) {
      PRINTF("PXP init error. STOP.\n");
      for ( ; ; ) ;
  }
#endif
```

Project setup:

- Add PXP related files to project:
  - lv_gpu/lv_gpu_nxp.c, lv_gpu/lv_gpu_nxp.h: low level drawing calls for LVGL
  - lv_gpu/lv_gpu_nxp_osa.c, lv_gpu/lv_gpu_osa.h: default implementation of OS-specific functions (bare metal and FreeRTOS only)
    - optional, required only if `LV_USE_GPU_NXP_PXP_AUTO_INIT` is set to 1
- PXP related code depends on two drivers provided by MCU SDK. These drivers need to be added to project:
  - fsl_pxp.c, fsl_pxp.h: PXP driver
  - fsl_cache.c, fsl_cache.h: CPU cache handling functions

Advanced configuration:

- Implementation depends on multiple OS-specific functions. Structure `lv_nxp_pxp_cfg_t` with callback pointers is used as a parameter for `lv_gpu_nxp_pxp_init()` function. Default implementation for FreeRTOS and baremetal is provided in lv_gpu_nxp_osa.c
  - `pxp_interrupt_init()`: Initialize PXP interrupt (HW setup, OS setup)
  - `pxp_interrupt_deinit()`: Deinitialize PXP interrupt (HW setup, OS setup)
  - `pxp_run()`: Start PXP job. Use OS-specific mechanism to block drawing thread. PXP must finish drawing before leaving this function.
- There are configurable area thresholds which are used to decide whether the area will be processed by CPU, or by PXP. Areas smaller than defined value will be processed by CPU, areas bigger than the threshold will be processed by PXP. These thresholds may be defined as a preprocessor variables. Default values are defined lv_gpu/lv_gpu_nxp_pxp.h
  - `GPU_NXP_PXP_BLIT_SIZE_LIMIT`: size threshold for image BLIT, BLIT with color keying, and BLIT with recolor (OPA > LV_OPA_MAX)

- `GPU_NXP_PXP_BLIT_OPA_SIZE_LIMIT`: size threshold for image BLIT and BLIT with color keying with transparency (OPA < LV_OPA_MAX)
- `GPU_NXP_PXP_FILL_SIZE_LIMIT`: size threshold for fill operation (OPA > LV_OPA_MAX)
- `GPU_NXP_PXP_FILL_OPA_SIZE_LIMIT`: size threshold for fill operation with transparency (OPA < LV_OPA_MAX)

2.5 Espressif (ESP32)

Since v7.7.1 LVGL includes a Kconfig file, so LVGL can be used as an ESP-IDF v4 component.

2.5.1 Get the LVGL demo project for ESP32

We've created lv_port_esp32, a project using ESP-IDF and LVGL to show one of the demos from lv_examples. You are able to configure the project to use one of the many supported display controllers, see lvgl_esp32_drivers for a complete list of supported display and indev (touch) controllers.

2.5.2 Use LVGL in your ESP32 project

Prerequisites

ESP-IDF v4 framework is the suggested version to use.

Get LVGL

You are suggested to add LVGL as a "component". This component can be located inside a directory named "components" on your project root directory.

When your project is a git repository you can include LVGL as a git submodule:

```
git submodule add https://github.com/lvgl/lvgl.git components/lvgl
```

The above command will clone LVGL's main repository into the `components/lvgl` directory. LVGL includes a `CMakeLists.txt` file that sets some configuration options so you can use LVGL right away.

When you are ready to configure LVGL launch the configuration menu with `idf.py menuconfig` on your project root directory, go to `Component config` and then `LVGL configuration`.

2.5.3 Use lvgl_esp32_drivers in your project

You are suggested to add `lvgl_esp32_drivers` as a "component". This component can be located inside a directory named "components" on your project root directory.

When your project is a git repository you can include `lvgl_esp32_drivers` as a git submodule:

```
git submodule add https://github.com/lvgl/lvgl_esp32_drivers.git components/lvgl_
↪esp32_drivers
```

Support for ESP32-S2

Basic support for ESP32-S2 has been added into the `lvgl_esp32_drivers` repository.

2.6 Arduino

The core LVGL library and the examples are directly available as Arduino libraries.

Note that you need to choose a powerful enough board to run LVGL and your GUI. See the requirements of LVGL.

For example ESP32 is a good candidate to create your UI with LVGL.

2.6.1 Get the LVGL Arduino library

LVGL can be installed via Arduino IDE Library Manager or as an .ZIP library. It will also install lv_exmaples which contains a lot of examples and demos to try LVGL.

2.6.2 Set up drivers

To get started it's recommended to use TFT_eSPI library as a TFT driver to simplify testing. To make it work setup `TFT_eSPI` according to your TFT display type via editing either

- `User_Setup.h`
- or by selecting a configuration in the `User_Setup_Select.h`

Both files are located in `TFT_eSPI` library's folder.

2.6.3 Configure LVGL

LVGL has its own configuration file called `lv_conf.h`. When LVGL is installed the followings needs to be done to configure it:

1. Go to directory of the installed Arduino libraries
2. Go to `lvgl` and copy `lv_conf_template.h` as `lv_conf.h` into the Arduino Libraries directory next to the `lvgl` library folder.
3. Open `lv_conf.h` and change the first `#if 0` to `#if 1`
4. Set the resolution of your display in `LV_HOR_RES_MAX` and `LV_VER_RES_MAX`
5. Set the color depth of you display in `LV_COLOR_DEPTH`
6. Set `LV_TICK_CUSTOM 1`

2.6.4 Configure the examples

`lv_examples` can be configures similarly to LVGL but it's configuration file is called `lv_ex_conf.h`.

1. Go to directory of the installed Arduino libraries
2. Go to `lv_examples` and copy `lv_ex_template.h` as `lv_ex_conf.h` next to the `lv_examples` folder.
3. Open `lv_ex_conf.h` and change the first `#if 0` to `#if 1`
4. Enable the demos you want to use. (The small examples starting with `lv_ex_...()` are always enabled.)

2.6.5 Initialize LVGL and run an example

Take a look at LVGL_Arduino.ino to see how to initialize LVGL. It also uses TFT_eSPI as driver.

In the INO file you can see how to register a display and a touch pad for LVGL and call an example.

Note that, there is no dedicated INO file for every example but you can call functions like `lv_ex_btn1()` or `lv_ex_slider1()` to run an example. For the full list of examples see the README of lv_examples.

2.6.6 Debugging and logging

In case of trouble there are debug information inside LVGL. In the `LVGL_Arduino.ino` example there is `my_print` method, which allow to send this debug information to the serial interface. To enable this feature you have to edit `lv_conf.h` file and enable logging in section `log settings`:

```
/*Log settings*/
#define USE_LV_LOG      1   /*Enable/disable the log module*/
#if 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␣
↪problem
 * LV_LOG_LEVEL_ERROR       Only critical issue, when the system may fail
 * LV_LOG_LEVEL_NONE        Do not log anything
 */
#  define LV_LOG_LEVEL    LV_LOG_LEVEL_WARN
```

After enabling log module and setting LV_LOG_LEVEL accordingly the output log is sent to the `Serial` port @ 115200 Baud rate.

2.7 Micropython

2.7.1 What is Micropython?

Micropython is Python for microcontrollers.Using Micropython, you can write Python3 code and run it even on a bare metal architecture with limited resources.

Highlights of Micropython

- **Compact** - Fits and runs within just 256k of code space and 16k of RAM. No OS is needed, although you can also run it with an OS, if you want.
- **Compatible** - Strives to be as compatible as possible with normal Python (known as CPython).
- **Versatile** - Supports many architectures (x86, x86-64, ARM, ARM Thumb, Xtensa).
- **Interactive** - No need for the compile-flash-boot cycle. With the REPL (interactive prompt) you can type commands and execute them immediately, run scripts etc.
- **Popular** - Many platforms are supported. The user base is growing bigger. Notable forks: MicroPython, CircuitPython, MicroPython_ESP32_psRAM_LoBo
- **Embedded Oriented** - Comes with modules specifically for embedded systems, such as the machine module for accessing low-level hardware (I/O pins, ADC, UART, SPI, I2C, RTC, Timers etc.)

2.7.2 Why Micropython + LVGL?

Currently, Micropython does not have a good high-level GUI library by default. LVGL is an Object Oriented Component Based high-level GUI library, which seems to be a natural candidate to map into a higher level language, such as Python. LVGL is implemented in C and its APIs are in C.

Here are some advantages of using LVGL in Micropython:

- Develop GUI in Python, a very popular high level language. Use paradigms such as Object Oriented Programming.
- Usually, GUI development requires multiple iterations to get things right. With C, each iteration consists of **`Change code`** > **`Build`** > **`Flash`** > **`Run`**.In Micropython it's just **`Change code`** > **`Run`** ! You can even run commands interactively using the REPL (the interactive prompt)

Micropython + LVGL could be used for:

- Fast prototyping GUI.
- Shorten the cycle of changing and fine-tuning the GUI.
- Model the GUI in a more abstract way by defining reusable composite objects, taking advantage of Python's language features such as Inheritance, Closures, List Comprehension, Generators, Exception Handling, Arbitrary Precision Integers and others.
- Make LVGL accessible to a larger audience. No need to know C in order to create a nice GUI on an embedded system.This goes well with CircuitPython vision. CircuitPython was designed with education in mind, to make it easier for new or unexperienced users to get started with embedded development.
- Creating tools to work with LVGL at a higher level (e.g. drag-and-drop designer).

2.7.3 So what does it look like?

TL;DR: It's very much like the C API, but Object Oriented for LVGL components.

Let's dive right into an example!

A simple example

```
import lvgl as lv
lv.init()
scr = lv.obj()
btn = lv.btn(scr)
btn.align(lv.scr_act(), lv.ALIGN.CENTER, 0, 0)
label = lv.label(btn)
label.set_text("Button")
lv.scr_load(scr)
```

2.7.4 How can I use it?

Online Simulator

If you want to experiment with LVGL + Micropython without downloading anything - you can use our online simulator!It's a fully functional LVGL + Micropython that runs entirely in the browser and allows you to edit a python script and run it.

Click here to experiment on the online simulator

Hello World

Note: the online simulator is available for lvgl v6 and v7.

PC Simulator

Micropython is ported to many platforms. One notable port is "unix", which allows you to build and run Micropython (+LVGL) on a Linux machine. (On a Windows machine you might need Virtual Box or WSL or MinGW or Cygwin etc.)

Click here to know more information about building and running the unix port

Embedded platform

At the end, the goal is to run it all on an embedded platform.Both Micropython and LVGL can be used on many embedded architectures, such as stm32, ESP32 etc.You would also need display and input drivers. We have some sample drivers (ESP32+ILI9341, as well as some other examples), but most chances are you would want to create your own input/display drivers for your specific purposes.Drivers can be implemented either in C as Micropython module, or in pure Micropython!

2.7.5 Where can I find more information?

- On the Blog Post
- On `lv_micropython` README
- On `lv_binding_micropython` README
- On LVGL forum (Feel free to ask anything!)
- On Micropython docs and forum

2.8 NuttX RTOS

2.8.1 What is NuttX?

NuttX is a mature and secure real-time operating system (RTOS) with an emphasis on technical standards compliance and small size. It is scalable from 8-bit to 64-bit microcontroller and microprocessors. Complaint with the Portable Operating System Interface (POSIX) and the American National Standards Institute (ANSI) standards and with many Linux-like subsystems. The best way to think about NuttX is thinking about a small Unix/Linux for microcontrollers.

Highlights of NuttX

- **Small** - Fits and runs within small microcontroller as small was 32KB Flash and 8KB of RAM.
- **Compliant** - Strives to be as compatible as possible with POSIX and Linux.
- **Versatile** - Supports many architectures (ARM, ARM Thumb, AVR, MIPS, OpenRISC, RISC-V 32-bit and 64-bit, RX65N, x86-64, Xtensa, Z80/Z180, etc).
- **Modular** - Its modular design allow developers to select only what really matters and use modules to include new features.
- **Popular** - NuttX is used by many companies around the world. Probably you already used a product with NuttX without knowing it was running NuttX.
- **Predictable** - NuttX is a preemptible Realtime kernel, then you can use it to create predictable applications for realtime control.

2.8.2 Why NuttX + LVGL?

Although NuttX has its own graphic library called NX, LVGL is a good alternative because users could find more eyes-candy demos and reuse it from previous projects. LVGL is an Object Oriented Component Based high-level GUI library, that could fit very well for a RTOS with advanced features like NuttX. LVGL is implemented in C and its APIs are in C.

Here are some advantages of using LVGL in NuttX

- Develop GUI in Linux first and when it is done just compile it for NuttX, nothing more, no wasting of time.
- Usually, GUI development for low level RTOS requires multiple iterations to get things right. Where each iteration consists of **Change code** > **Build** > **Flash** > **Run**. Using LVGL, Linux and NuttX you can reduce this process and just test everything on your computer and when it is done, compile it on NuttX and that is it.

NuttX + LVGL could be used for

- GUI demos to demonstrate your board graphics capacities.
- Fast prototyping GUI for MVP (Minimum Viable Product) presentation.
- Easy way to visualize sensors data directly on the board without using a computer.
- Final products GUI without touchscreen (i.e. 3D Printer Interface using Rotary Encoder to Input data).
- Final products interface with touchscren (and bells and whistles).

2.8.3 How to get started with NuttX and LVGL?

There are many boards in the NuttX mainline (https://github.com/apache/incubator-nuttx) with support for LVGL. Let's to use the STM32F429IDISCOVERY as example because it is a very popular board.

First you need to install the pre-requisite on your system

Let's to use Linux and example, for Windows

```
$ sudo apt-get install automake bison build-essential flex gcc-arm-none-eabi gperf␣
↪git libncurses5-dev libtool libusb-dev libusb-1.0.0-dev pkg-config kconfig-
↪frontends openocd
```

Now let's to create a workspace to save our files

```
$ mkdir ~/nuttxspace
$ cd ~/nuttxspace
```

Clone the NuttX and Apps repositories:

```
$ git clone https://github.com/apache/incubator-nuttx nuttx
$ git clone https://github.com/apache/incubator-nuttx-apps apps
```

Configure NuttX to use the stm32f429i-disco board and the LVGL Demo

```
$ ./tools/configure.sh stm32f429i-disco:lvgl
$ make
```

If everything went fine you should have now the file `nuttx.bin` to flash on your board:

```
$ ls -l nuttx.bin
-rwxrwxr-x 1 alan alan 287144 Jun 27 09:26 nuttx.bin
```

Flashing the firmware in the board using OpenOCD:

```
$ sudo openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c init -c "reset␣
↪halt" -c "flash write_image erase nuttx.bin 0x08000000"
```

Reset the board and using the 'NSH>' terminal start the LVGL demo:

```
nsh> lvgldemo
```

2.8.4 Where can I find more information?

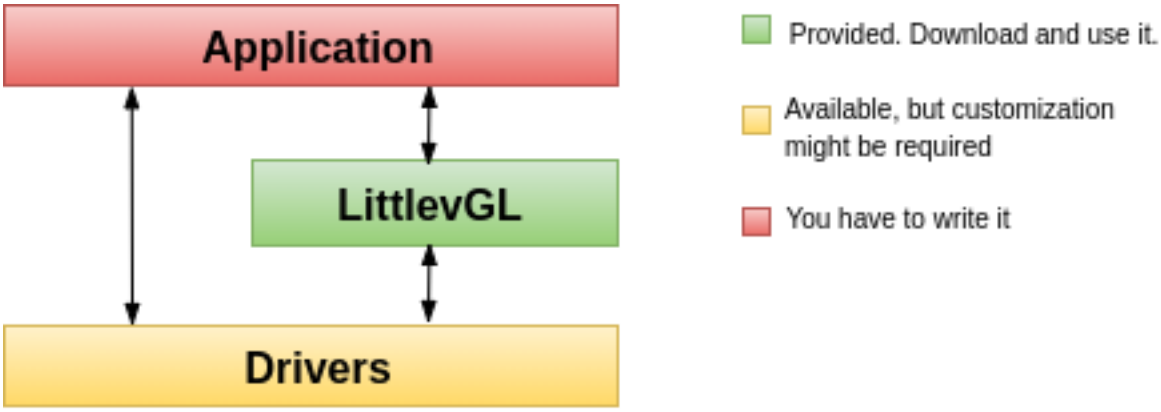
- On the LVGL on LPCXpresso54628
- NuttX mailing list Apache NuttX Mailing List

CHAPTER

THREE

PORTING

3.1 System overview



Application Your application which creates the GUI and handles the specific tasks.

LVGL The graphics library itself. Your application can communicate with the library to create a GUI. It contains a HAL (Hardware Abstraction Layer) interface to register your display and input device drivers.

Driver Besides your specific drivers, it contains functions to drive your display, optionally to a GPU and to read the touchpad or buttons.

Depending on the MCU, there are two typical hardware set-ups. One with built-in LCD/TFT driver periphery and another without it. In both cases, a frame buffer will be required to store the current image of the screen.

1. **MCU with TFT/LCD driver** If your MCU has a TFT/LCD driver periphery then you can connect a display directly via RGB interface. In this case, the frame buffer can be in the internal RAM (if the MCU has enough RAM) or in the external RAM (if the MCU has a memory interface).
2. **External display controller** If the MCU doesn't have TFT/LCD driver interface then an external display controller (E.g. SSD1963, SSD1306, ILI9341) has to be used. In this case, the MCU can communicate with the display controller via Parallel port, SPI or sometimes I2C. The frame buffer is usually located in the display controller which saves a lot of RAM for the MCU.

3.2 Set-up a project

3.2.1 Get the library

LVGL Graphics Library is available on GitHub: https://github.com/lvgl/lvgl.

You can clone it or download the latest version of the library from GitHub.

The graphics library is the **lvgl** directory which should be copied into your project.

3.2.2 Configuration file

There is a configuration header file for LVGL called **lv_conf.h**. It sets the library's basic behaviour, disables unused modules and features, adjusts the size of memory buffers in compile-time, etc.

Copy **lvgl/lv_conf_template.h** next to the *lvgl* directory and rename it to *lv_conf.h*. Open the file and change the `#if 0` at the beginning to `#if 1` to enable its content.

lv_conf.h can be copied other places as well but then you should add `LV_CONF_INCLUDE_SIMPLE` define to your compiler options (e.g. `-DLV_CONF_INCLUDE_SIMPLE` for gcc compiler) and set the include path manually.

In the config file comments explain the meaning of the options. Check at least these three configuration options and modify them according to your hardware:

1. **LV_HOR_RES_MAX** Your display's horizontal resolution.
2. **LV_VER_RES_MAX** Your display's vertical resolution.
3. **LV_COLOR_DEPTH** 8 for (RG332), 16 for (RGB565) or 32 for (RGB888 and ARGB8888).

3.2.3 Initialization

To use the graphics library you have to initialize it and the other components too. The order of the initialization is:

1. Call *lv_init()*.
2. Initialize your drivers.
3. Register the display and input devices drivers in LVGL. More about *Display* and *Input device* registration.
4. Call `lv_tick_inc(x)` in every `x` milliseconds in an interrupt to tell the elapsed time. *Learn more*.
5. Call `lv_task_handler()` periodically in every few milliseconds to handle LVGL related tasks. *Learn more*.

3.3 Display interface

To set up a display an `lv_disp_buf_t` and an `lv_disp_drv_t` variable has to be initialized.

- **lv_disp_buf_t** contains internal graphics buffer(s).
- **lv_disp_drv_t** contains callback functions to interact with the display and manipulate drawing related things.

3.3.1 Display buffer

`lv_disp_buf_t` can be initialized like this:

```
/*A static or global variable to store the buffers*/
static lv_disp_buf_t disp_buf;

/*Static or global buffer(s). The second buffer is optional*/
static lv_color_t buf_1[MY_DISP_HOR_RES * 10];
static lv_color_t buf_2[MY_DISP_HOR_RES * 10];

/*Initialize `disp_buf` with the buffer(s) */
lv_disp_buf_init(&disp_buf, buf_1, buf_2, MY_DISP_HOR_RES*10);
```

There are 3 possible configurations regarding the buffer size:

1. **One buffer** LVGL draws the content of the screen into a buffer and sends it to the display. The buffer can be smaller than the screen. In this case, the larger areas will be redrawn in multiple parts. If only small areas changes (e.g. button press) then only those areas will be refreshed.
2. **Two non-screen-sized buffers** having two buffers LVGL can draw into one buffer while the content of the other buffer is sent to display in the background. DMA or other hardware should be used to transfer the data to the display to let the CPU draw meanwhile. This way the rendering and refreshing of the display become parallel. Similarly to the *One buffer*, LVGL will draw the display's content in chunks if the buffer is smaller than the area to refresh.
3. **Two screen-sized buffers**. In contrast to *Two non-screen-sized buffers* LVGL will always provide the whole screen's content not only chunks. This way the driver can simply change the address of the frame buffer to the buffer received from LVGL. Therefore this method works the best when the MCU has an LCD/TFT interface and the frame buffer is just a location in the RAM.

You can measure the performance of your display configuration using the benchmark example.

3.3.2 Display driver

Once the buffer initialization is ready the display drivers need to be initialized. In the most simple case only the following two fields of `lv_disp_drv_t` needs to be set:

- **buffer** pointer to an initialized `lv_disp_buf_t` variable.
- **flush_cb** a callback function to copy a buffer's content to a specific area of the display. `lv_disp_flush_ready()` needs to be called when flushing is ready. LVGL might render the screen in multiple chunks and therefore call `flush_cb` multiple times. To see which is the last chunk of rendering use `lv_disp_flush_is_last()`.

There are some optional data fields:

- **hor_res** horizontal resolution of the display. (`LV_HOR_RES_MAX` by default from *lv_conf.h*).

- **ver_res** vertical resolution of the display. (`LV_VER_RES_MAX` by default from *lv_conf.h*).
- **color_chroma_key** a color which will be drawn as transparent on chrome keyed images. `LV_COLOR_TRANSP` by default from *lv_conf.h*).
- **user_data** custom user data for the driver. Its type can be modified in lv_conf.h.
- **anti-aliasing** use anti-aliasing (edge smoothing). `LV_ANTIALIAS` by default from *lv_conf.h*.
- **rotated** and **sw_rotate** See the *rotation* section below.
- **screen_transp** if `1` the screen can have transparent or opaque style. `LV_COLOR_SCREEN_TRANSP` needs to enabled in *lv_conf.h*.

To use a GPU the following callbacks can be used:

- **gpu_fill_cb** fill an area in memory with colors.
- **gpu_blend_cb** blend two memory buffers using opacity.
- **gpu_wait_cb** if any GPU function return, while the GPU is still working LVGL, will use this function when required the be sure GPU rendering is ready.

Note that, these functions need to draw to the memory (RAM) and not your display directly.

Some other optional callbacks to make easier and more optimal to work with monochrome, grayscale or other non-standard RGB displays:

- **rounder_cb** round the coordinates of areas to redraw. E.g. a 2x2 px can be converted to 2x8. It can be used if the display controller can refresh only areas with specific height or width (usually 8 px height with monochrome displays).
- **set_px_cb** a custom function to write the *display buffer*. It can be used to store the pixels more compactly if the display has a special color format. (e.g. 1-bit monochrome, 2-bit grayscale etc.) This way the buffers used in `lv_disp_buf_t` can be smaller to hold only the required number of bits for the given area size. `set_px_cb` is not working with `Two screen-sized buffers` display buffer configuration.
- **monitor_cb** a callback function tells how many pixels were refreshed in how much time.
- **clean_dcache_cb** a callback for cleaning any caches related to the display

To set the fields of *lv_disp_drv_t* variable it needs to be initialized with `lv_disp_drv_init(&disp_drv)`. And finally to register a display for LVGL `lv_disp_drv_register(&disp_drv)` needs to be called.

All together it looks like this:

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

Here some simple examples of the callbacks:

```
void my_flush_cb(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t * color_
↪p)
{
```

(continues on next page)

(continued from previous page)

```
    /*The most simple case (but also the slowest) to put all pixels to the screen one-
↪by-one*/
    int32_t x, y;
    for(y = area->y1; y <= area->y2; y++) {
        for(x = area->x1; x <= area->x2; x++) {
            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_drv);
}

void my_gpu_fill_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest_buf, const lv_area_t␣
↪* dest_area, const lv_area_t * fill_area, lv_color_t color);
{
    /*It's an example code which should be done by your GPU*/
    uint32_t x, y;
    dest_buf += dest_width * fill_area->y1; /*Go to the first line*/

    for(y = fill_area->y1; y < fill_area->y2; y++) {
        for(x = fill_area->x1; x < fill_area->x2; x++) {
            dest_buf[x] = color;
        }
        dest_buf+=dest_width;    /*Go to the next line*/
    }
}

void my_gpu_blend_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest, const lv_color_t *␣
↪src, uint32_t length, lv_opa_t opa)
{
    /*It's an example code which should be done by your GPU*/
    uint32_t i;
    for(i = 0; i < length; i++) {
        dest[i] = lv_color_mix(dest[i], src[i], opa);
    }
}

void my_rounder_cb(lv_disp_drv_t * disp_drv, lv_area_t * area)
{
  /* Update the areas as needed. Can be only larger.
   * For example to always have lines 8 px height:*/
   area->y1 = area->y1 & 0x07;
   area->y2 = (area->y2 & 0x07) + 8;
}

void my_set_px_cb(lv_disp_drv_t * disp_drv, uint8_t * buf, lv_coord_t buf_w, lv_coord_
↪t x, lv_coord_t y, lv_color_t color, lv_opa_t opa)
{
    /* Write to the buffer as required for the display.
     * Write only 1-bit for monochrome displays mapped vertically:*/
 buf += buf_w * (y >> 3) + x;
 if(lv_color_brightness(color) > 128) (*buf) |= (1 << (y % 8));
 else (*buf) &= ~(1 << (y % 8));
```

(continues on next page)

(continued from previous page)

```
}

void my_monitor_cb(lv_disp_drv_t * disp_drv, uint32_t time, uint32_t px)
{
  printf("%d px refreshed in %d ms\n", time, ms);
}

void my_clean_dcache_cb(lv_disp_drv_t * disp_drv, uint32)
{
  /* Example for Cortex-M (CMSIS) */
  SCB_CleanInvalidateDCache();
}
```

3.3.3 Rotation

LVGL supports rotation of the display in 90 degree increments. You can select whether you'd like software rotation or hardware rotation.

If you select software rotation (`sw_rotate` flag set to 1), LVGL will perform the rotation for you. Your driver can and should assume that the screen width and height have not changed. Simply flush pixels to the display as normal. Software rotation requires no additional logic in your `flush_cb` callback.

There is a noticeable amount of overhead to performing rotation in software, which is why hardware rotation is also available. In this mode, LVGL draws into the buffer as though your screen now has the width and height inverted. You are responsible for rotating the provided pixels yourself.

The default rotation of your display when it is initialized can be set using the `rotated` flag. The available options are `LV_DISP_ROT_NONE`, `LV_DISP_ROT_90`, `LV_DISP_ROT_180`, or `LV_DISP_ROT_270`. The rotation values are relative to how you would rotate the physical display in the clockwise direction. Thus, `LV_DISP_ROT_90` means you rotate the hardware 90 degrees clockwise, and the display rotates 90 degrees counterclockwise to compensate.

(Note for users upgrading from 7.10.0 and older: these new rotation enum values match up with the old 0/1 system for rotating 90 degrees, so legacy code should continue to work as expected. Software rotation is also disabled by default for compatibility.)

Display rotation can also be changed at runtime using the `lv_disp_set_rotation(disp, rot)` API.

Support for software rotation is a new feature, so there may be some glitches/bugs depending on your configuration. If you encounter a problem please open an issue on GitHub.

3.3.4 API

Display Driver HAL interface header file

Typedefs

typedef struct *_disp_drv_t* **lv_disp_drv_t**
Display Driver structure to be registered by HAL

typedef struct *_disp_t* **lv_disp_t**
Display structure.

Note: `lv_disp_drv_t` should be the first member of the structure.

Enums

enum lv_disp_rot_t
Values:

enumerator LV_DISP_ROT_NONE

enumerator LV_DISP_ROT_90

enumerator LV_DISP_ROT_180

enumerator LV_DISP_ROT_270

enum lv_disp_size_t
Values:

enumerator LV_DISP_SIZE_SMALL

enumerator LV_DISP_SIZE_MEDIUM

enumerator LV_DISP_SIZE_LARGE

enumerator LV_DISP_SIZE_EXTRA_LARGE

Functions

void **lv_disp_drv_init**(*lv_disp_drv_t *driver*)
Initialize a display driver with default values. It is used to have known values in the fields and not junk in memory. After it you can safely set only the fields you need.

Parameters **driver** -- pointer to driver variable to initialize

void **lv_disp_buf_init**(*lv_disp_buf_t *disp_buf*, void *buf1*, void *buf2*, uint32_t *size_in_px_cnt*)
Initialize a display buffer

Parameters

- **disp_buf** -- pointer *lv_disp_buf_t* variable to initialize
- **buf1** -- A buffer to be used by LVGL to draw the image. Always has to specified and can't be NULL. Can be an array allocated by the user. E.g. `static lv_color_t disp_buf1[1024 * 10]` Or a memory address e.g. in external SRAM
- **buf2** -- Optionally specify a second buffer to make image rendering and image flushing (sending to the display) parallel. In the `disp_drv->flush` you should use DMA or similar hardware to send the image to the display in the background. It lets LVGL to render next frame into the other buffer while previous is being sent. Set to `NULL` if unused.

- **size_in_px_cnt** -- size of the buf1 and buf2 in pixel count.

lv_disp_t ***lv_disp_drv_register**(*lv_disp_drv_t *driver*)
Register an initialized display driver. Automatically set the first display as active.

Parameters **driver** -- pointer to an initialized 'lv_disp_drv_t' variable (can be local variable)

Returns pointer to the new display or NULL on error

void **lv_disp_drv_update**(*lv_disp_t *disp*, *lv_disp_drv_t *new_drv*)
Update the driver in run time.

Parameters

- **disp** -- pointer to a display. (return value of lv_disp_drv_register)
- **new_drv** -- pointer to the new driver

void **lv_disp_remove**(*lv_disp_t *disp*)
Remove a display

Parameters **disp** -- pointer to display

void **lv_disp_set_default**(*lv_disp_t *disp*)
Set a default screen. The new screens will be created on it by default.

Parameters **disp** -- pointer to a display

lv_disp_t ***lv_disp_get_default**(void)
Get the default display

Returns pointer to the default display

lv_coord_t **lv_disp_get_hor_res**(*lv_disp_t *disp*)
Get the horizontal resolution of a display

Parameters **disp** -- pointer to a display (NULL to use the default display)

Returns the horizontal resolution of the display

lv_coord_t **lv_disp_get_ver_res**(*lv_disp_t *disp*)
Get the vertical resolution of a display

Parameters **disp** -- pointer to a display (NULL to use the default display)

Returns the vertical resolution of the display

bool **lv_disp_get_antialiasing**(*lv_disp_t *disp*)
Get if anti-aliasing is enabled for a display or not

Parameters **disp** -- pointer to a display (NULL to use the default display)

Returns true: anti-aliasing is enabled; false: disabled

lv_coord_t **lv_disp_get_dpi**(*lv_disp_t *disp*)
Get the DPI of the display

Parameters **disp** -- pointer to a display (NULL to use the default display)

Returns dpi of the display

lv_disp_size_t **lv_disp_get_size_category**(*lv_disp_t *disp*)
Get the size category of the display based on it's hor. res. and dpi.

Parameters **disp** -- pointer to a display (NULL to use the default display)

Returns LV_DISP_SIZE_SMALL/MEDIUM/LARGE/EXTRA_LARGE

lv_disp_t ***lv_disp_get_next**(*lv_disp_t *disp*)
Get the next display.

Parameters **disp** -- pointer to the current display. NULL to initialize.

Returns the next display or NULL if no more. Give the first display when the parameter is NULL

lv_disp_buf_t ***lv_disp_get_buf**(*lv_disp_t *disp*)
Get the internal buffer of a display

Parameters **disp** -- pointer to a display

Returns pointer to the internal buffers

uint16_t **lv_disp_get_inv_buf_size**(*lv_disp_t *disp*)
Get the number of areas in the buffer

Returns number of invalid areas

void **_lv_disp_pop_from_inv_buf**(*lv_disp_t *disp*, uint16_t *num*)
Pop (delete) the last 'num' invalidated areas from the buffer

Parameters **num** -- number of areas to delete

bool **lv_disp_is_double_buf**(*lv_disp_t *disp*)
Check the driver configuration if it's double buffered (both `buf1` and `buf2` are set)

Parameters **disp** -- pointer to to display to check

Returns true: double buffered; false: not double buffered

bool **lv_disp_is_true_double_buf**(*lv_disp_t *disp*)
Check the driver configuration if it's TRUE double buffered (both `buf1` and `buf2` are set and `size` is screen sized)

Parameters **disp** -- pointer to to display to check

Returns true: double buffered; false: not double buffered

struct lv_disp_buf_t
#include <lv_hal_disp.h> Structure for holding display buffer information.

Public Members

void ***buf1**
First display buffer.

void ***buf2**
Second display buffer.

void ***buf_act**

uint32_t **size**

lv_area_t **area**

int **flushing**

int **flushing_last**

uint32_t **last_area**

uint32_t **last_part**

struct _disp_drv_t

#include <lv_hal_disp.h> Display Driver structure to be registered by HAL

Public Members

lv_coord_t **hor_res**
Horizontal resolution.

lv_coord_t **ver_res**
Vertical resolution.

lv_disp_buf_t ***buffer**
Pointer to a buffer initialized with ***lv_disp_buf_init()***. LVGL will use this buffer(s) to draw the screens contents

uint32_t **antialiasing**
1: antialiasing is enabled on this display.

uint32_t **rotated**

uint32_t **sw_rotate**
1: use software rotation (slower)

uint32_t **screen_transp**
Handle if the screen doesn't have a solid (opa == LV_OPA_COVER) background. Use only if required because it's slower.

uint32_t **dpi**
DPI (dot per inch) of the display. Set to `LV_DPI` from `lv_Conf.h` by default.

void (***flush_cb**)(**struct** *lv_disp_drv_t* *disp_drv, **const** lv_area_t *area, lv_color_t *color_p)
MANDATORY: Write the internal buffer (VDB) to the display. 'lv_disp_flush_ready()' has to be called when finished

void (***rounder_cb**)(**struct** *lv_disp_drv_t* *disp_drv, lv_area_t *area)
OPTIONAL: Extend the invalidated areas to match with the display drivers requirements E.g. round `y` to, 8, 16 ..) on a monochrome display

void (***set_px_cb**)(**struct** *lv_disp_drv_t* *disp_drv, uint8_t *buf, lv_coord_t buf_w, lv_coord_t x, lv_coord_t y, lv_color_t color, lv_opa_t opa)
OPTIONAL: Set a pixel in a buffer according to the special requirements of the display Can be used for color format not supported in LittelvGL. E.g. 2 bit -> 4 gray scales

---

Note: Much slower then drawing with supported color formats.

---

void (***monitor_cb**)(**struct** *lv_disp_drv_t* *disp_drv, uint32_t time, uint32_t px)
OPTIONAL: Called after every refresh cycle to tell the rendering and flushing time + the number of flushed pixels

void (***wait_cb**)(**struct** *lv_disp_drv_t* *disp_drv)
OPTIONAL: Called periodically while lvgl waits for operation to be completed. For example flushing or GPU User can execute very simple tasks here or yield the task

void (***clean_dcache_cb**)(**struct** *lv_disp_drv_t* *disp_drv)
OPTIONAL: Called when lvgl needs any CPU cache that affects rendering to be cleaned

void (***gpu_wait_cb**)(**struct** *lv_disp_drv_t* *disp_drv)
OPTIONAL: called to wait while the gpu is working

void (*gpu_blend_cb)(struct _disp_drv_t *disp_drv, lv_color_t *dest, const lv_color_t *src, uint32_t length, lv_opa_t opa)
OPTIONAL: Blend two memories using opacity (GPU only)

void (*gpu_fill_cb)(struct _disp_drv_t *disp_drv, lv_color_t *dest_buf, lv_coord_t dest_width, const lv_area_t *fill_area, lv_color_t color)
OPTIONAL: Fill a memory with a color (GPU only)

lv_color_t **color_chroma_key**
On CHROMA_KEYED images this color will be transparent. `LV_COLOR_TRANSP` by default. (lv_conf.h)

lv_disp_drv_user_data_t **user_data**
Custom display driver user data

struct _disp_t

#include <lv_hal_disp.h> Display structure.

Note: `lv_disp_drv_t` should be the first member of the structure.

Public Members

lv_disp_drv_t **driver**
< Driver to the display A task which periodically checks the dirty areas and refreshes them

lv_task_t ***refr_task**

lv_ll_t **scr_ll**
Screens of the display

struct *_lv_obj_t* ***act_scr**
Currently active screen on this display

struct *_lv_obj_t* ***prev_scr**
Previous screen. Used during screen animations

struct *_lv_obj_t* ***scr_to_load**
The screen prepared to load in lv_scr_load_anim

struct *_lv_obj_t* ***top_layer**

See *lv_disp_get_layer_top*

struct *_lv_obj_t* ***sys_layer**

See *lv_disp_get_layer_sys*

uint8_t **del_prev**
1: Automatically delete the previous screen when the screen load animation is ready

lv_color_t **bg_color**
Default display color when screens are transparent

const void ***bg_img**
An image source to display as wallpaper

lv_opa_t **bg_opa**
Opacity of the background color or wallpaper

lv_area_t **inv_areas**[**LV_INV_BUF_SIZE**]
Invalidated (marked to redraw) areas

uint8_t **inv_area_joined**[**LV_INV_BUF_SIZE**]

uint32_t **inv_p**

uint32_t **last_activity_time**
Last time there was activity on this display

3.4 Input device interface

3.4.1 Types of input devices

To set up an input device an `lv_indev_drv_t` variable has to be initialized:

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

type can be

- **LV_INDEV_TYPE_POINTER** touchpad or mouse
- **LV_INDEV_TYPE_KEYPAD** keyboard or keypad
- **LV_INDEV_TYPE_ENCODER** encoder with left, right, push options
- **LV_INDEV_TYPE_BUTTON** external buttons pressing the screen

read_cb is a function pointer which will be called periodically to report the current state of an input device. It can also buffer data and return `false` when no more data to be read or `true` when the buffer is not empty.

Visit *Input devices* to learn more about input devices in general.

Touchpad, mouse or any pointer

Input devices which can click points of the screen belong to this category.

```
indev_drv.type = LV_INDEV_TYPE_POINTER;
indev_drv.read_cb = my_input_read;

...

bool my_input_read(lv_indev_drv_t * drv, lv_indev_data_t*data)
{
    data->point.x = touchpad_x;
    data->point.y = touchpad_y;
    data->state = LV_INDEV_STATE_PR or LV_INDEV_STATE_REL;
    return false; /*No buffering now so no more data read*/
}
```

Important: Touchpad drivers must return the last X/Y coordinates even when the state is *LV_INDEV_STATE_REL*.

To set a mouse cursor use `lv_indev_set_cursor(my_indev, &img_cursor)`. (`my_indev` is the return value of `lv_indev_drv_register`)

Keypad or keyboard

Full keyboards with all the letters or simple keypads with a few navigation buttons belong here.

To use a keyboard/keypad:

- Register a `read_cb` function with `LV_INDEV_TYPE_KEYPAD` type.
- Enable `LV_USE_GROUP` in *lv_conf.h*
- An object group has to be created: `lv_group_t * g = lv_group_create()` and objects have to be added to it with `lv_group_add_obj(g, obj)`
- The created group has to be assigned to an input device: `lv_indev_set_group(my_indev, g)` (`my_indev` is the return value of `lv_indev_drv_register`)
- Use `LV_KEY_...` to navigate among the objects in the group. See `lv_core/lv_group.h` for the available keys.

```
indev_drv.type = LV_INDEV_TYPE_KEYPAD;
indev_drv.read_cb = keyboard_read;

...

bool keyboard_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
  data->key = last_key();            /*Get the last pressed or released key*/

  if(key_pressed()) data->state = LV_INDEV_STATE_PR;
  else data->state = LV_INDEV_STATE_REL;

  return false; /*No buffering now so no more data read*/
}
```

Encoder

With an encoder you can do 4 things:

1. Press its button
2. Long-press its button
3. Turn left
4. Turn right

In short, the Encoder input devices work like this:

- By turning the encoder you can focus on the next/previous object.
- When you press the encoder on a simple object (like a button), it will be clicked.
- If you press the encoder on a complex object (like a list, message box, etc.) the object will go to edit mode whereby turning the encoder you can navigate inside the object.
- To leave edit mode press long the button.

To use an *Encoder* (similarly to the *Keypads*) the objects should be added to groups.

```
indev_drv.type = LV_INDEV_TYPE_ENCODER;
indev_drv.read_cb = encoder_read;

...

bool encoder_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
  data->enc_diff = enc_get_new_moves();

  if(enc_pressed()) data->state = LV_INDEV_STATE_PR;
  else data->state = LV_INDEV_STATE_REL;

  return false; /*No buffering now so no more data read*/
}
```

Using buttons with Encoder logic

In addition to standard encoder behavior, you can also utilise its logic to navigate(focus) and edit widgets using buttons. This is especially handy if you have only few buttons avalible, or you want to use other buttons in addition to encoder wheel.

You need to have 3 buttons avalible:

- **LV_KEY_ENTER** will simulate press or pushing of the encoder button
- **LV_KEY_LEFT** will simulate turnuing encoder left
- **LV_KEY_RIGHT** will simulate turnuing encoder right
- other keys will be passed to the focused widget

If you hold the keys it will simulate encoder click with period specified in `indev_drv.long_press_rep_time`.

```
indev_drv.type = LV_INDEV_TYPE_ENCODER;
indev_drv.read_cb = encoder_with_keys_read;

...

bool encoder_with_keys_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
  data->key = last_key();            /*Get the last pressed or released key*/
                                     /* use LV_KEY_ENTER for encoder press */
  if(key_pressed()) data->state = LV_INDEV_STATE_PR;
  else {
      data->state = LV_INDEV_STATE_REL;
      /* Optionally you can also use enc_diff, if you have encoder*/
      data->enc_diff = enc_get_new_moves();
  }

  return false; /*No buffering now so no more data read*/
}
```

Button

Buttons mean external "hardware" buttons next to the screen which are assigned to specific coordinates of the screen. If a button is pressed it will simulate the pressing on the assigned coordinate. (Similarly to a touchpad)

To assign buttons to coordinates use `lv_indev_set_button_points(my_indev, points_array)`.`points_array` should look like `const lv_point_t points_array[] = { {12,30},{60,90}, ...}`

Important: The points_array can't go out of scope. Either declare it as a global variable or as a static variable inside a function.

```
indev_drv.type = LV_INDEV_TYPE_BUTTON;
indev_drv.read_cb = button_read;

...

bool button_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
    static uint32_t last_btn = 0;   /*Store the last pressed button*/
    int btn_pr = my_btn_read();     /*Get the ID (0,1,2...) of the pressed button*/
    if(btn_pr >= 0) {               /*Is there a button press? (E.g. -1 indicated no␣
↪button was pressed)*/
       last_btn = btn_pr;           /*Save the ID of the pressed button*/
       data->state = LV_INDEV_STATE_PR;  /*Set the pressed state*/
    } else {
       data->state = LV_INDEV_STATE_REL; /*Set the released state*/
    }

    data->btn = last_btn;            /*Save the last button*/

    return false;                    /*No buffering now so no more data read*/
}
```

3.4.2 Other features

Besides `read_cb` a `feedback_cb` callback can be also specified in `lv_indev_drv_t`. `feedback_cb` is called when any type of event is sent by the input devices. (independently from its type). It allows making feedback for the user e.g. to play a sound on `LV_EVENT_CLICK`.

The default value of the following parameters can be set in *lv_conf.h* but the default value can be overwritten in `lv_indev_drv_t`:

- **drag_limit** Number of pixels to slide before actually drag the object
- **drag_throw** Drag throw slow-down in [%]. Greater value means faster slow-down
- **long_press_time** Press time to send `LV_EVENT_LONG_PRESSED` (in milliseconds)
- **long_press_rep_time** Interval of sending `LV_EVENT_LONG_PRESSED_REPEAT` (in milliseconds)
- **read_task** pointer to the `lv_task` which reads the input device. Its parameters can be changed by `lv_task_...()` functions

Every Input device is associated with a display. By default, a new input device is added to the lastly created or the explicitly selected (using `lv_disp_set_default()`) display. The associated display is stored and can be changed in `disp` field of the driver.

3.4.3 API

Input Device HAL interface layer header file

Typedefs

typedef uint8_t **lv_indev_type_t**

typedef uint8_t **lv_indev_state_t**

typedef uint8_t **lv_drag_dir_t**

typedef uint8_t **lv_gesture_dir_t**

typedef struct *_lv_indev_drv_t* **lv_indev_drv_t**
Initialized by the user and registered by 'lv_indev_add()'

typedef struct *_lv_indev_proc_t* **lv_indev_proc_t**
Run time data of input devices Internally used by the library, you should not need to touch it.

typedef struct *_lv_indev_t* **lv_indev_t**
The main input device descriptor with driver, runtime data ('proc') and some additional information

Enums

enum [anonymous]
Possible input device types

Values:

enumerator LV_INDEV_TYPE_NONE
Uninitialized state

enumerator LV_INDEV_TYPE_POINTER
Touch pad, mouse, external button

enumerator LV_INDEV_TYPE_KEYPAD
Keypad or keyboard

enumerator LV_INDEV_TYPE_BUTTON
External (hardware button) which is assigned to a specific point of the screen

enumerator LV_INDEV_TYPE_ENCODER
Encoder with only Left, Right turn and a Button

enum [anonymous]
States for input devices

Values:

enumerator LV_INDEV_STATE_REL

enumerator LV_INDEV_STATE_PR

enum [anonymous]
Values:

enumerator LV_DRAG_DIR_HOR
Object can be dragged horizontally.

enumerator LV_DRAG_DIR_VER
Object can be dragged vertically.

enumerator LV_DRAG_DIR_BOTH
Object can be dragged in all directions.

enumerator LV_DRAG_DIR_ONE
Object can be dragged only one direction (the first move).

enum [anonymous]
Values:

enumerator LV_GESTURE_DIR_TOP
Gesture dir up.

enumerator LV_GESTURE_DIR_BOTTOM
Gesture dir down.

enumerator LV_GESTURE_DIR_LEFT
Gesture dir left.

enumerator LV_GESTURE_DIR_RIGHT
Gesture dir right.

Functions

void **lv_indev_drv_init**(*lv_indev_drv_t *driver*)
Initialize an input device driver with default values. It is used to surly have known values in the fields ant not memory junk. After it you can set the fields.

Parameters **driver** -- pointer to driver variable to initialize

lv_indev_t ***lv_indev_drv_register**(*lv_indev_drv_t *driver*)
Register an initialized input device driver.

Parameters **driver** -- pointer to an initialized 'lv_indev_drv_t' variable (can be local variable)

Returns pointer to the new input device or NULL on error

void **lv_indev_drv_update**(*lv_indev_t *indev*, *lv_indev_drv_t *new_drv*)
Update the driver in run time.

Parameters

- **indev** -- pointer to a input device. (return value of `lv_indev_drv_register`)
- **new_drv** -- pointer to the new driver

lv_indev_t ***lv_indev_get_next**(*lv_indev_t *indev*)
Get the next input device.

Parameters **indev** -- pointer to the current input device. NULL to initialize.

Returns the next input devise or NULL if no more. Give the first input device when the parameter is NULL

bool **_lv_indev_read**(*lv_indev_t *indev*, *lv_indev_data_t *data*)
Read data from an input device.

Parameters

- **indev** -- pointer to an input device
- **data** -- input device will write its data here

Returns false: no more data; true: there more data to read (buffered)

struct lv_indev_data_t

#include <lv_hal_indev.h> Data structure passed to an input driver to fill

Public Members

lv_point_t **point**
For LV_INDEV_TYPE_POINTER the currently pressed point

uint32_t **key**
For LV_INDEV_TYPE_KEYPAD the currently pressed key

uint32_t **btn_id**
For LV_INDEV_TYPE_BUTTON the currently pressed button

int16_t **enc_diff**
For LV_INDEV_TYPE_ENCODER number of steps since the previous read

lv_indev_state_t **state**
LV_INDEV_STATE_REL or LV_INDEV_STATE_PR

struct _lv_indev_drv_t

#include <lv_hal_indev.h> Initialized by the user and registered by 'lv_indev_add()'

Public Members

lv_indev_type_t **type**
< Input device type Function pointer to read input device data. Return 'true' if there is more data to be read (buffered). Most drivers can safely return 'false'

bool (***read_cb**)(**struct** *_lv_indev_drv_t* *indev_drv, *lv_indev_data_t* *data)

void (***feedback_cb**)(**struct** *_lv_indev_drv_t**, uint8_t)
Called when an action happened on the input device. The second parameter is the event from `lv_event_t`

lv_indev_drv_user_data_t **user_data**

struct *_disp_t* ***disp**
< Pointer to the assigned display Task to read the periodically read the input device

lv_task_t ***read_task**
Number of pixels to slide before actually drag the object

uint8_t **drag_limit**
Drag throw slow-down in [%]. Greater value means faster slow-down

uint8_t **drag_throw**
At least this difference should between two points to evaluate as gesture

uint8_t **gesture_min_velocity**
At least this difference should be to send a gesture

uint8_t **gesture_limit**
Long press time in milliseconds

uint16_t **long_press_time**
Repeated trigger period in long press [ms]

uint16_t **long_press_rep_time**

struct _lv_indev_proc_t

#include <lv_hal_indev.h> Run time data of input devices Internally used by the library, you should not need to touch it.

Public Members

lv_indev_state_t **state**
Current state of the input device.

lv_point_t **act_point**
Current point of input device.

lv_point_t **last_point**
Last point of input device.

lv_point_t **vect**
Difference between `act_point` and `last_point`.

lv_point_t **drag_sum**

lv_point_t **drag_throw_vect**

struct *_lv_obj_t* ***act_obj**

struct *_lv_obj_t* ***last_obj**

struct *_lv_obj_t* ***last_pressed**

lv_gesture_dir_t **gesture_dir**

lv_point_t **gesture_sum**

uint8_t **drag_limit_out**

uint8_t **drag_in_prog**

lv_drag_dir_t **drag_dir**

uint8_t **gesture_sent**

struct *_lv_indev_proc_t*::[**anonymous**]::[**anonymous**] **pointer**

lv_indev_state_t **last_state**

uint32_t **last_key**

struct *_lv_indev_proc_t*::[**anonymous**]::[**anonymous**] **keypad**

union *_lv_indev_proc_t*::[**anonymous**] **types**

uint32_t **pr_timestamp**
Pressed time stamp

uint32_t **longpr_rep_timestamp**
Long press repeat time stamp

uint8_t **long_pr_sent**

uint8_t **reset_query**

uint8_t **disabled**

uint8_t **wait_until_release**

struct _lv_indev_t

#include <lv_hal_indev.h> The main input device descriptor with driver, runtime data ('proc') and some additional information

Public Members

lv_indev_drv_t **driver**

lv_indev_proc_t **proc**

struct *_lv_obj_t* ***cursor**
Cursor for LV_INPUT_TYPE_POINTER

struct *_lv_group_t* ***group**
Keypad destination group

const lv_point_t ***btn_points**
Array points assigned to the button ()screen will be pressed here by the buttons

3.5 Tick interface

The LVGL needs a system tick to know the elapsed time for animation and other tasks.

You need to call the `lv_tick_inc(tick_period)` function periodically and tell the call period in milliseconds. For example, `lv_tick_inc(1)` for calling in every millisecond.

`lv_tick_inc` should be called in a higher priority routine than `lv_task_handler()` (e.g. in an interrupt) to precisely know the elapsed milliseconds even if the execution of `lv_task_handler` takes longer time.

With FreeRTOS `lv_tick_inc` can be called in `vApplicationTickHook`.

On Linux based operating system (e.g. on Raspberry Pi) `lv_tick_inc` can be called in a thread as below:

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

3.5.1 API

Provide access to the system tick with 1 millisecond resolution

Functions

uint32_t **lv_tick_get**(void)
Get the elapsed milliseconds since start up

Returns the elapsed milliseconds

uint32_t **lv_tick_elaps**(uint32_t *prev_tick*)
Get the elapsed milliseconds since a previous time stamp

Parameters **prev_tick** -- a previous time stamp (return value of *lv_tick_get()*)

Returns the elapsed milliseconds since 'prev_tick'

3.6 Task Handler

To handle the tasks of LVGL you need to call `lv_task_handler()` periodically in one of the followings:

- *while(1)* of *main()* function
- timer interrupt periodically (low priority then `lv_tick_inc()`)
- an OS task periodically

The timing is not critical but it should be about 5 milliseconds to keep the system responsive.

Example:

```
while(1) {
  lv_task_handler();
  my_delay_ms(5);
}
```

To learn more about task visit the *Tasks* section.

3.7 Sleep management

The MCU can go to sleep when no user input happens. In this case, the main `while(1)` should look like this:

```
while(1) {
  /*Normal operation (no sleep) in < 1 sec inactivity*/
  if(lv_disp_get_inactive_time(NULL) < 1000) {
          lv_task_handler();
  }
  /*Sleep after 1 sec inactivity*/
  else {
          timer_stop();   /*Stop the timer where lv_tick_inc() is called*/
          sleep();                    /*Sleep the MCU*/
  }
  my_delay_ms(5);
}
```

You should also add below lines to your input device read function if a wake-up (press, touch or click etc.) happens:

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

In addition to `lv_disp_get_inactive_time()` you can check `lv_anim_count_running()` to see if every animations are finished.

3.8 Operating system and interrupts

LVGL is **not thread-safe** by default.

However, in the following conditions it's valid to call LVGL related functions:

- In *events*. Learn more in *Events*.
- In *lv_tasks*. Learn more in *Tasks*.

3.8.1 Tasks and threads

If you need to use real tasks or threads, you need a mutex which should be invoked before the call of `lv_task_handler` and released after it. Also, you have to use the same mutex in other tasks and threads around every LVGL (`lv_...`) related function calls and codes. This way you can use LVGL in a real multitasking environment. Just make use of a mutex to avoid the concurrent calling of LVGL functions.

3.8.2 Interrupts

Try to avoid calling LVGL functions from the interrupts (except `lv_tick_inc()` and `lv_disp_flush_ready()`). But, if you need to do this you have to disable the interrupt which uses LVGL functions while `lv_task_handler` is running. It's a better approach to set a flag or some value and periodically check it in an `lv_task`.

3.9 Logging

LVGL has built-in *log* module to inform the user about what is happening in the library.

3.9.1 Log level

To enable logging, set `LV_USE_LOG 1` in *lv_conf.h* and set `LV_LOG_LEVEL` to one of the following values:

- **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 problem
- **LV_LOG_LEVEL_ERROR** Only critical issue, when the system may fail
- **LV_LOG_LEVEL_NONE** Do not log anything

The events which have a higher level than the set log level will be logged too. E.g. if you `LV_LOG_LEVEL_WARN`, *errors* will be also logged.

3.9.2 Logging with printf

If your system supports `printf`, you just need to enable `LV_LOG_PRINTF` in *lv_conf.h* to send the logs with `printf`.

3.9.3 Custom log function

If you can't use `printf` or want to use a custom function to log, you can register a "logger" callback with `lv_log_register_print_cb()`.

For example:

```
void my_log_cb(lv_log_level_t level, const char * file, uint32_t line, const char *␣
↪fn_name, const char * dsc)
{
  /*Send the logs via serial port*/
  if(level == LV_LOG_LEVEL_ERROR) serial_send("ERROR: ");
  if(level == LV_LOG_LEVEL_WARN)  serial_send("WARNING: ");
  if(level == LV_LOG_LEVEL_INFO)  serial_send("INFO: ");
  if(level == LV_LOG_LEVEL_TRACE) serial_send("TRACE: ");

  serial_send("File: ");
  serial_send(file);

  char line_str[8];
  sprintf(line_str,"%d", line);
  serial_send("#");
  serial_send(line_str);

  serial_send(": ");
  serial_send(fn_name);
  serial_send(": ");
  serial_send(dsc);
  serial_send("\n");
}

...


lv_log_register_print_cb(my_log_cb);
```

3.9.4 Add logs

You can also use the log module via the `LV_LOG_TRACE/INFO/WARN/ERROR(description)` functions.

CHAPTER

FOUR

OVERVIEW

4.1 Objects

In the LVGL the **basic building blocks** of a user interface are the objects, also called *Widgets*. For example a *Button*, *Label*, *Image*, *List*, *Chart* or *Text area*.

Check all the *Object types* here.

4.1.1 Attributes

Basic attributes

All object types share some basic attributes:

- Position
- Size
- Parent
- Drag enable
- Click enable etc.

You can set/get these attributes with `lv_obj_set_...` and `lv_obj_get_...` functions. For example:

```
/*Set basic object attributes*/
lv_obj_set_size(btn1, 100, 50);          /*Button size*/
lv_obj_set_pos(btn1, 20,30);      /*Button position*/
```

To see all the available functions visit the Base object's *documentation*.

Specific attributes

The object types have special attributes too. For example, a slider has

- Min. max. values
- Current value
- Custom styles

For these attributes, every object type have unique API functions. For example for a slider:

```
/*Set slider specific attributes*/
lv_slider_set_range(slider1, 0, 100);           /*Set min. and max. values*/
lv_slider_set_value(slider1, 40, LV_ANIM_ON);       /*Set the current value␣
↪(position)*/
lv_slider_set_action(slider1, my_action);     /*Set a callback function*/
```

The API of the object types are described in their *Documentation* but you can also check the respective header files (e.g. *lv_objx/lv_slider.h*)

4.1.2 Working mechanisms

Parent-child structure

A parent object can be considered as the container of its children. Every object has exactly one parent object (except screens), but a parent can have an unlimited number of children. There is no limitation for the type of the parent but, there are typical parent (e.g. button) and typical child (e.g. label) objects.

Moving together

If the position of the parent is changed the children will move with the parent. Therefore all positions are relative to the parent.

The (0;0) coordinates mean the objects will remain in the top left-hand corner of the parent independently from the position of the parent.

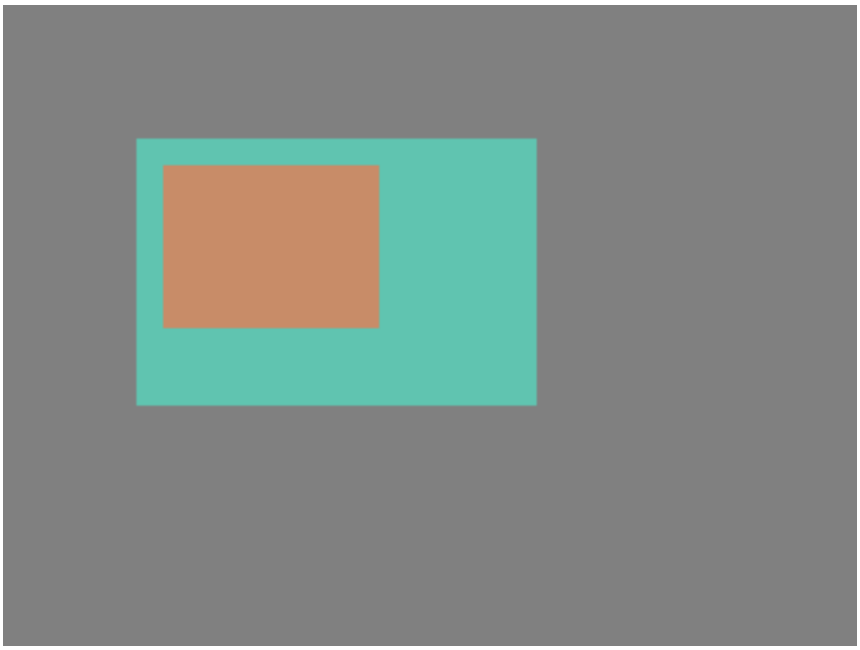
```
lv_obj_t * par = lv_obj_create(lv_scr_act(), NULL); /*Create a parent object on the␣
↪current screen*/
lv_obj_set_size(par, 100, 80);                          /*Set the size of the␣
↪parent*/
```

(continues on next page)

(continued from previous page)

```
lv_obj_t * obj1 = lv_obj_create(par, NULL);                    /*Create an object on the␣
↪previously created parent object*/
lv_obj_set_pos(obj1, 10, 10);                                 /*Set the position of the new␣
↪object*/
```

Modify the position of the parent:



```
lv_obj_set_pos(par, 50, 50);          /*Move the parent. The child will move with it.*/
```

(For simplicity the adjusting of colors of the objects is not shown in the example.)

Visibility only on the parent

If a child is partially or fully out of its parent then the parts outside will not be visible.

```
lv_obj_set_x(obj1, -30);        /*Move the child a little bit of the parent*/
```

Create - delete objects

In LVGL objects can be created and deleted dynamically in run-time. It means only the currently created objects consume RAM. For example, if you need a chart, you can create it when required and delete it when it is not visible or necessary.

Every object type has its own **create** function with a unified prototype. It needs two parameters:

- A pointer to the *parent* object. To create a screen give *NULL* as parent.
- Optionally, a pointer to *copy* object with the same type to copy it. This *copy* object can be *NULL* to avoid the copy operation.

All objects are referenced in C code using an `lv_obj_t` pointer as a handle. This pointer can later be used to set or get the attributes of the object.

The create functions look like this:

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

There is a common **delete** function for all object types. It deletes the object and all of its children.

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

`lv_obj_del` will delete the object immediately. If for any reason you can't delete the object immediately you can use `lv_obj_del_async(obj)`. It is useful e.g. if you want to delete the parent of an object in the child's `LV_EVENT_DELETE` signal.

You can remove all the children of an object (but not the object itself) using `lv_obj_clean`:

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

4.1.3 Screens

Create screens

The screens are special objects which have no parent object. So they can be created like:

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

Screens can be created with any object type. For example, a *Base object* or an image to make a wallpaper.

Get the active screen

There is always an active screen on each display. By default, the library creates and loads a "Base object" as a screen for each display.

To get the currently active screen use the `lv_scr_act()` function.

Load screens

To load a new screen, use `lv_scr_load(scr1)`.

Load screen with animation

A new screen can be loaded with animation too using `lv_scr_load_anim(scr, transition_type, time, delay, auto_del)`. The following transition types exist:

- `LV_SCR_LOAD_ANIM_NONE`: switch immediately after `delay` ms
- `LV_SCR_LOAD_ANIM_OVER_LEFT/RIGHT/TOP/BOTTOM` move the new screen over the other towards the given direction
- `LV_SCR_LOAD_ANIM_MOVE_LEFT/RIGHT/TOP/BOTTOM` move both the old and new screens towards the given direction
- `LV_SCR_LOAD_ANIM_FADE_ON` fade the new screen over the old screen

Setting `auto_del` to `true` will automatically delete the old screen when the animation is finished.

The new screen will become active (returned by `lv_scr_act()`) when the animations starts after `delay` time.

Handling multiple displays

Screens are created on the currently selected *default display*. The *default display* is the last registered display with `lv_disp_drv_register` or you can explicitly select a new default display using `lv_disp_set_default(disp)`.

`lv_scr_act()`, `lv_scr_load()` and `lv_scr_load_anim()` operate on the default screen.

Visit *Multi-display support* to learn more.

4.1.4 Parts

The widgets can have multiple parts. For example a *Button* has only a main part but a *Slider* is built from a background, an indicator and a knob.

The name of the parts is constructed like `LV_ + <TYPE> _PART_ <NAME>`. For example `LV_BTN_PART_MAIN` or `LV_SLIDER_PART_KNOB`. The parts are usually used when styles are add to the objects. Using parts different styles can be assigned to the different parts of the objects.

To learn more about the parts read the related section of the Style overview.

4.1.5 States

The object can be in a combinations of the following states:

- **LV_STATE_DEFAULT** Normal, released
- **LV_STATE_CHECKED** Toggled or checked
- **LV_STATE_FOCUSED** Focused via keypad or encoder or clicked via touchpad/mouse
- **LV_STATE_EDITED** Edit by an encoder
- **LV_STATE_HOVERED** Hovered by mouse (not supported now)
- **LV_STATE_PRESSED** Pressed
- **LV_STATE_DISABLED** Disabled or inactive

The states are usually automatically changed by the library as the user presses, releases, focuses etc an object. However, the states can be changed manually too. To completely overwrite the current state use `lv_obj_set_state(obj, part, LV_STATE...)`. To set or clear given state (but leave to other states untouched) use `lv_obj_add/clear_state(obj, part, LV_STATE_...)` In both cases ORed state values can be used as well. E.g. `lv_obj_set_state(obj, part, LV_STATE_PRESSED | LV_PRESSED_CHECKED)`.

To learn more about the states read the related section of the Style overview.

4.2 Layers

4.2.1 Order of creation

By default, LVGL draws old objects on the background and new objects on the foreground.

For example, assume we added a button to a parent object named button1 and then another button named button2. Then button1 (with its child object(s)) will be in the background and can be covered by button2 and its children.


```
/*Create a screen*/
lv_obj_t * scr = lv_obj_create(NULL, NULL);
lv_scr_load(scr);           /*Load the screen*/

/*Create 2 buttons*/
lv_obj_t * btn1 = lv_btn_create(scr, NULL);         /*Create a button on the screen*/
lv_btn_set_fit(btn1, true, true);                   /*Enable to automatically set the␣
↪size according to the content*/
lv_obj_set_pos(btn1, 60, 40);                         /*Set the position of the␣
↪button*/

lv_obj_t * btn2 = lv_btn_create(scr, btn1);         /*Copy the first button*/
lv_obj_set_pos(btn2, 180, 80);                     /*Set the position of the button*/

/*Add labels to the buttons*/
lv_obj_t * label1 = lv_label_create(btn1, NULL);        /*Create a label on the first␣
↪button*/
lv_label_set_text(label1, "Button 1");                  /*Set the text of the label*/

lv_obj_t * label2 = lv_label_create(btn2, NULL);          /*Create a label on the␣
↪second button*/
lv_label_set_text(label2, "Button 2");                    /*Set the text of the␣
↪label*/

/*Delete the second label*/
lv_obj_del(label2);
```

4.2.2 Bring to the foreground

There are several ways to bring an object to the foreground:

- Use `lv_obj_set_top(obj, true)`. If `obj` or any of its children is clicked, then LVGL will automatically bring the object to the foreground. It works similarly to a typical GUI on a PC. When a window in the background is clicked, it will come to the foreground automatically.
- Use `lv_obj_move_foreground(obj)` to explicitly tell the library to bring an object to the foreground. Similarly, use `lv_obj_move_background(obj)` to move to the background.
- When `lv_obj_set_parent(obj, new_parent)` is used, `obj` will be on the foreground on the `new_parent`.

4.2.3 Top and sys layers

LVGL uses two special layers named as `layer_top` and `layer_sys`. Both are visible and common on all screens of a display. **They are not, however, shared among multiple physical displays.** The `layer_top` is always on top of the default screen (`lv_scr_act()`), and `layer_sys` is on top of `layer_top`.

The `layer_top` can be used by the user to create some content visible everywhere. For example, a menu bar, a pop-up, etc. If the `click` attribute is enabled, then `layer_top` will absorb all user click and acts as a modal.

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

The `layer_sys` is also used for a similar purpose on LVGL. For example, it places the mouse cursor above all layers to be sure it's always visible.

4.3 Events

Events are triggered in LVGL when something happens which might be interesting to the user, e.g. if an object:

- is clicked
- is dragged
- its value has changed, etc.

The user can assign a callback function to an object to see these events. In practice, it looks like this:

```
lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);
lv_obj_set_event_cb(btn, my_event_cb);   /*Assign an event callback*/

...

static void my_event_cb(lv_obj_t * obj, lv_event_t event)
{
    switch(event) {
        case LV_EVENT_PRESSED:
            printf("Pressed\n");
            break;

        case LV_EVENT_SHORT_CLICKED:
```

(continues on next page)

(continued from previous page)

```
            printf("Short clicked\n");
            break;

        case LV_EVENT_CLICKED:
            printf("Clicked\n");
            break;

        case LV_EVENT_LONG_PRESSED:
            printf("Long press\n");
            break;

        case LV_EVENT_LONG_PRESSED_REPEAT:
            printf("Long press repeat\n");
            break;

        case LV_EVENT_RELEASED:
            printf("Released\n");
            break;
    }

    /*Etc.*/
}
```

More objects can use the same *event callback*.

4.3.1 Event types

The following event types exist:

Generic events

All objects (such as Buttons/Labels/Sliders etc.) receive these generic events regardless of their type.

Related to the input devices

These are sent when an object is pressed/released etc. by the user. They are used not only for *Pointers* but can used for *Keypad*, *Encoder* and *Button* input devices as well. Visit the *Overview of input devices* section to learn more about them.

- **LV_EVENT_PRESSED** The object has been pressed
- **LV_EVENT_PRESSING** The object is being pressed (sent continuously while pressing)
- **LV_EVENT_PRESS_LOST** The input device is still being pressed but is no longer on the object
- **LV_EVENT_SHORT_CLICKED** Released before `LV_INDEV_LONG_PRESS_TIME` time. Not called if dragged.
- **LV_EVENT_LONG_PRESSED** Pressing for `LV_INDEV_LONG_PRESS_TIME` time. Not called if dragged.
- **LV_EVENT_LONG_PRESSED_REPEAT** Called after `LV_INDEV_LONG_PRESS_TIME` in every `LV_INDEV_LONG_PRESS_REP_TIME` ms. Not called if dragged.
- **LV_EVENT_CLICKED** Called on release if not dragged (regardless to long press)

- **LV_EVENT_RELEASED** Called in every case when the object has been released even if it was dragged. Not called if slid from the object while pressing and released outside of the object. In this case, `LV_EVENT_PRESS_LOST` is sent.

Related to pointer

These events are sent only by pointer-like input devices (E.g. mouse or touchpad)

- **LV_EVENT_DRAG_BEGIN** Dragging of the object has started
- **LV_EVENT_DRAG_END** Dragging finished (including drag throw)
- **LV_EVENT_DRAG_THROW_BEGIN** Drag throw started (released after drag with "momentum")

Related to keypad and encoder

These events are sent by keypad and encoder input devices. Learn more about *Groups* in [overview/indev](Input devices) section.

- **LV_EVENT_KEY** A *Key* is sent to the object. Typically when it was pressed or repeated after a long press. The key can be retrived by `uint32_t * key = lv_event_get_data()`
- **LV_EVENT_FOCUSED** The object is focused in its group
- **LV_EVENT_DEFOCUSED** The object is defocused in its group

General events

Other general events sent by the library.

- **LV_EVENT_DELETE** The object is being deleted. Free the related user-allocated data.

Special events

These events are specific to a particular object type.

- **LV_EVENT_VALUE_CHANGED** The object value has changed (e.g. for a *Slider*)
- **LV_EVENT_INSERT** Something is inserted to the object. (Typically to a *Text area*)
- **LV_EVENT_APPLY** "Ok", "Apply" or similar specific button has clicked. (Typically from a *Keyboard* object)
- **LV_EVENT_CANCEL** "Close", "Cancel" or similar specific button has clicked. (Typically from a *Keyboard* object)
- **LV_EVENT_REFRESH** Query to refresh the object. Never sent by the library but can be sent by the user.

Visit particular *Object type's documentation* to understand which events are used by an object type.

4.3.2 Custom data

Some events might contain custom data. For example, `LV_EVENT_VALUE_CHANGED` in some cases tells the new value. For more information, see the particular *Object type's documentation*. To get the custom data in the event callback use `lv_event_get_data()`.

The type of the custom data depends on the sending object but if it's a

- single number then it's `uint32_t *` or `int32_t *`
- text then `char *` or `const char *`

4.3.3 Send events manually

Arbitrary events

To manually send events to an object, use `lv_event_send(obj, LV_EVENT_..., &custom_data)`.

For example, it can be used to manually close a message box by simulating a button press (although there are simpler ways of doing this):

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

Refresh event

`LV_EVENT_REFRESH` is special event because it's designed to be used by the user to notify an object to refresh itself. Some examples:

- notify a label to refresh its text according to one or more variables (e.g. current time)
- refresh a label when the language changes
- enable a button if some conditions are met (e.g. the correct PIN is entered)
- add/remove styles to/from an object if a limit is exceeded, etc

To simplest way to handle similar cases is utilizing the following functions.

`lv_event_send_refresh(obj)` is just a wrapper to `lv_event_send(obj, LV_EVENT_REFRESH, NULL)`. So it simply sends an `LV_EVENT_REFRESH` to an object.

`lv_event_send_refresh_recursive(obj)` sends `LV_EVENT_REFRESH` event to an object and all of its children. If `NULL` is passed as parameter all objects of all displays will be refreshed.

4.4 Styles

Styles are used to set the appearance of the objects. Styles in lvgl are heavily inspired by CSS. The concept in nutshell is the following:

- A style is an `lv_style_t` variable which can hold properties, for example border width, text color and so on. It's similar to `class` in CSS.
- Not all properties have to be specified. Unspecified properties will use a default value.
- Styles can be assigned to objects to change their appearance.

- A style can be used by any number of objects.
- Styles can be cascaded which means multiple styles can be assigned to an object and each style can have different properties. For example `style_btn` can result in a default gray button and `style_btn_red` can add only a `background-color=red` to overwrite the background color.
- Later added styles have higher precedence. It means if a property is specified in two styles the later added will be used.
- Some properties (e.g. text color) can be inherited from the parent(s) if it's not specified in the object.
- Objects can have local styles that have higher precedence than "normal" styles.
- Unlike CSS (where pseudo-classes describes different states, e.g. `:hover`), in lvgl a property is assigned to a given state. (I.e. not the "class" is related to state but every single property has a state)
- Transitions can be applied when the object changes state.

4.4.1 States

The objects can be in the following states:

- **LV_STATE_DEFAULT** (0x00): Normal, released
- **LV_STATE_CHECKED** (0x01): Toggled or checked
- **LV_STATE_FOCUSED** (0x02): Focused via keypad or encoder or clicked via touchpad/mouse
- **LV_STATE_EDITED** (0x04): Edit by an encoder
- **LV_STATE_HOVERED** (0x08): Hovered by mouse (not supported now)
- **LV_STATE_PRESSED** (0x10): Pressed
- **LV_STATE_DISABLED** (0x20): Disabled or inactive

Combination of states is also possible, for example `LV_STATE_FOCUSED | LV_STATE_PRESSED`.

The style properties can be defined in every state and state combination. For example, setting a different background color for default and pressed state. If a property is not defined in a state the best matching state's property will be used. Typically it means the property with `LV_STATE_DEFAULT` state.¿ If the property is not set even for the default state the default value will be used. (See later)

But what does the "best matching state's property" really means? States have a precedence which is shown by their value (see in the above list). A higher value means higher precedence. To determine which state's property to use let's use an example. Let's see the background color is defined like this:

- `LV_STATE_DEFAULT`: white
- `LV_STATE_PRESSED`: gray
- `LV_STATE_FOCUSED`: red

1. By the default the object is in default state, so it's a simple case: the property is perfectly defined in the object's current state as white
2. When the object is pressed there are 2 related properties: default with white (default is related to every state) and pressed with gray. The pressed state has 0x10 precedence which is higher than the default state's 0x00 precedence, so gray color will be used.
3. When the object is focused the same thing happens as in pressed state and red color will be used. (Focused state has higher precedence than default state).
4. When the object is focused and pressed both gray and red would work, but the pressed state has higher precedence than focused so gray color will be used.

5. It's possible to set e.g rose color for `LV_STATE_PRESSED | LV_STATE_FOCUSED`. In this case, this combined state has 0x02 + 0x10 = 0x12 precedence, which higher than the pressed states precedence so rose color would be used.
6. When the object is checked there is no property to set the background color for this state. So in lack of a better option, the object remains white from the default state's property.

Some practical notes:

- If you want to set a property for all state (e.g. red background color) just set it for the default state. If the object can't find a property for its current state it will fall back to the default state's property.
- Use ORed states to describe the properties for complex cases. (E.g. pressed + checked + focused)
- It might be a good idea to use different style elements for different states. For example, finding background colors for released, pressed, checked + pressed, focused, focused + pressed, focused + pressed + checked, etc states is quite difficult. Instead, for example, use the background color for pressed and checked states and indicate the focused state with a different border color.

4.4.2 Cascading styles

It's not required to set all the properties in one style. It's possible to add more styles to an object and let the later added style to modify or extend the properties in the other styles. For example, create a general gray button style and create a new for red buttons where only the new background color is set.

It's the same concept when in CSS all the used classes are listed like `<div class=".btn .btn-red">`.

The later added styles have higher precedence over the earlier ones. So in the gray/red button example above, the normal button style should be added first and the red style second. However, the precedence coming from states are still taken into account. So let's examine the following case:

- the basic button style defines dark-gray color for default state and light-gray color pressed state
- the red button style defines the background color as red only in the default state

In this case, when the button is released (it's in default state) it will be red because a perfect match is found in the lastly added style (red style). When the button is pressed the light-gray color is a better match because it describes the current state perfectly, so the button will be light-gray.

4.4.3 Inheritance

Some properties (typically that are related to texts) can be inherited from the parent object's styles. Inheritance is applied only if the given property is not set in the object's styles (even in default state). In this case, if the property is inheritable, the property's value will be searched in the parent too until a part can tell a value for the property. The parents will use their own state to tell the value. So is button is pressed, and text color comes from here, the pressed text color will be used.

4.4.4 Parts

Objects can have *parts* which can have their own style. For example a *page* has four parts:

- Background
- Scrollable
- Scrollbar
- Edge flash

There is three types of object parts **main**, **virtual** and **real**.

The main part is usually the background and largest part of the object. Some object has only a main part. For example, a button has only a background.

The virtual parts are additional parts just drawn on the fly to the main part. There is no "real" object behind them. For example, the page's scrollbar is not a real object, it's just drawn when the page's background is drawn. The virtual parts always have the same state as the main part. If the property can be inherited, the main part will be also considered before going to the parent.

The real parts are real objects created and managed by the main object. For example, the page's scrollable part is real object. Real parts can be in different state than the main part.

To see which parts an object has visit their documentation page.

4.4.5 Initialize styles and set/get properties

Styles are stored in `lv_style_t` variables. Style variables should be `static`, global or dynamically allocated. In other words they can not be local variables in functions which are destroyed when the function exists. Before using a style it should be initialized with `lv_style_init(&my_style)`. After initializing the style properties can be set or added to it. Property set functions looks like this: `lv_style_set_<property_name>(&style, <state>, <value>);` For example the *above mentioned* example looks like this:

```
static lv_style_t style1;
lv_style_set_bg_color(&style1, LV_STATE_DEFAULT, LV_COLOR_WHITE);
lv_style_set_bg_color(&style1, LV_STATE_PRESSED, LV_COLOR_GRAY);
lv_style_set_bg_color(&style1, LV_STATE_FOCUSED, LV_COLOR_RED);
lv_style_set_bg_color(&style1, LV_STATE_FOCUSED | LV_STATE_PRESSED, lv_color_
↪hex(0xf88));
```

It's possible to copy a style with `lv_style_copy(&style_destination, &style_source)`. After copy properties still can be added freely.

To remove a property use:

```
lv_style_remove_prop(&style, LV_STYLE_BG_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_
↪POS));
```

To get the value from style in a given state functions with the following prototype are available: `_lv_style_get_color/int/opa/ptr(&style, <prop>, <result buf>);`. The best matching property will be selected and it's precedence will be returned. `-1` will be returned if the property is not found.

The form of the function (`...color/int/opa/ptr`) should be used according to the type of `<prop>`.

For example:

```
lv_color_t color;
int16_t res;
res = _lv_style_get_color(&style1,  LV_STYLE_BG_COLOR | (LV_STATE_PRESSED << LV_STYLE_
↪STATE_POS), &color);
if(res >= 0) {
  //the bg_color is loaded into `color`
}
```

To reset a style (free all it's data) use

```
lv_style_reset(&style);
```

4.4.6 Managing style list

A style on its own not that useful. It should be assigned to an object to take its effect. Every part of the objects stores a *style list* which is the list of assigned styles.

To add a style to an object use `lv_obj_add_style(obj, <part>, &style)` For example:

```
lv_obj_add_style(btn, LV_BTN_PART_MAIN, &btn);      /*Default button style*/
lv_obj_add_style(btn, LV_BTN_PART_MAIN, &btn_red);  /*Overwrite only a some colors to␣
↪red*/
```

An objects style list can be reset with `lv_obj_reset_style_list(obj, <part>)`

If a style which is already assigned to an object changes (i.e. one of it's property is set to a new value) the objects using that style should be notified with `lv_obj_refresh_style(obj, part, property)`. To refresh all parts and proeprties use `lv_obj_refresh_style(obj, LV_OBJ_PART_ALL, LV_STYLE_PROP_ALL)`.

To get a final value of property, including cascading, inheritance, local styles and transitions (see below), get functions like this can be used: `lv_obj_get_style_<property_name>(obj, <part>)`. These functions uses the object's current state and if no better candidate returns a default value. For example:

```
lv_color_t color = lv_obj_get_style_bg_color(btn, LV_BTN_PART_MAIN);
```

4.4.7 Local styles

In the object's style lists, so-called local properties can be stored as well. It's the same concept than CSS's `<div style="color:red">`. The local style is the same as a normal style, but it belongs only to a given object and can not be shared with other objects. To set a local property use functions like `lv_obj_set_style_local_<property_name>(obj, <part>, <state>, <value>);` For example:

```
lv_obj_set_style_local_bg_color(btn, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, LV_COLOR_
↪RED);
```

4.4.8 Transitions

By default, when an object changes state (e.g. it's pressed) the new properties from the new state are set immediately. However, with transitions it's possible to play an animation on state change. For example, on pressing a button its background color can be animated to the pressed color over 300 ms.

The parameters of the transitions are stored in the styles. It's possible to set

- the time of the transition
- the delay before starting the transition
- the animation path (also known as timing function)
- the properties to animate

The transition properties can be defined for each state. For example, setting 500 ms transition time in default state will mean that when the object goes to default state 500 ms transition time will be applied. Setting 100 ms transition time in the pressed state will mean a 100 ms transition time when going to presses state. So this example configuration will result in fast going to presses state and slow going back to default.

4.4.9 Properties

The following properties can be used in the styles.

Mixed properties

- **radius** (`lv_style_int_t`): Set the radius of the background. 0: no radius, `LV_RADIUS_CIRCLE`: maximal radius. Default value: 0.
- **clip_corner** (`bool`): `true`: enable to clip the overflowed content on the rounded (radius > 0) corners. Default value: `false`.
- **size** (`lv_style_int_t`): Size of internal elements of the widgets. See the documentation of the widgets if this property is used or not. Default value: `LV_DPI / 20`.
- **transform_width** (`lv_style_int_t`): Make the object wider on both sides with this value. Default value: 0.
- **transform_height** (`lv_style_int_t`) Make the object higher on both sides with this value. Default value: 0.
- **transform_angle** (`lv_style_int_t`): Rotate the image-like objects. It's uinit is 0.1 deg, for 45 deg use 450. Default value: 0.
- **transform_zoom** (`lv_style_int_t`) Zoom image-like objects. 256 (or `LV_IMG_ZOOM_NONE`) for normal size, 128 half size, 512 double size, ans so on. Default value: `LV_IMG_ZOOM_NONE`.
- **opa_scale** (`lv_style_int_t`): Inherited. Scale down all opacity values of the object by this factor. As it's inherited the children objects will be affected too. Default value: `LV_OPA_COVER`.

Padding and margin properties

Padding sets the space on the inner sides of the edges. It means "I don't want my children too close to my sides, so keep this space".*Padding inner* set the "gap" between the children. *Margin* sets the space on the outer side of the edges. It means "I want this space around me".

These properties are typically used by *Container* object if layout or auto fit is enabled. However other widgets also use them to set spacing. See the documentation of the widgets for the details.

- **pad_top** (`lv_style_int_t`): Set the padding on the top. Default value: 0.
- **pad_bottom** (`lv_style_int_t`): Set the padding on the bottom. Default value: 0.
- **pad_left** (`lv_style_int_t`): Set the padding on the left. Default value: 0.
- **pad_right** (`lv_style_int_t`): Set the padding on the right. Default value: 0.
- **pad_inner** (`lv_style_int_t`): Set the padding inside the object between children. Default value: 0.
- **margin_top** (`lv_style_int_t`): Set the margin on the top. Default value: 0.
- **margin_bottom** (`lv_style_int_t`): Set the margin on the bottom. Default value: 0.
- **margin_left** (`lv_style_int_t`): Set the margin on the left. Default value: 0.
- **margin_right** (`lv_style_int_t`): Set the margin on the right. Default value: 0.

Background properties

The background is a simple rectangle which can have gradient and `radius` rounding.

- **bg_color** (`lv_color_t`) Specifies the color of the background. Default value: `LV_COLOR_WHITE`.
- **bg_opa** (`lv_opa_t`) Specifies opacity of the background. Default value: `LV_OPA_TRANSP`.
- **bg_grad_color** (`lv_color_t`) Specifies the color of the background's gradient. The color on the right or bottom is `bg_grad_dir != LV_GRAD_DIR_NONE`. Default value: `LV_COLOR_WHITE`.
- **bg_main_stop** (`uint8_t`): Specifies where should the gradient start. 0: at left/top most position, 255: at right/bottom most position. Default value: 0.
- **bg_grad_stop** (`uint8_t`): Specifies where should the gradient stop. 0: at left/top most position, 255: at right/bottom most position. Default value: 255.
- **bg_grad_dir** (`lv_grad_dir_t`) Specifies the direction of the gradient. Can be `LV_GRAD_DIR_NONE/HOR/VER`. Default value: `LV_GRAD_DIR_NONE`.
- **bg_blend_mode** (`lv_blend_mode_t`): Set the blend mode the background. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.

```c
#include "../../lv_examples.h"

/**
 * Using the background style properties
 */
void lv_ex_style_1(void)
{
    static lv_style_t style;
    lv_style_init(&style);
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);

    /*Make a gradient*/
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_bg_grad_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_bg_grad_dir(&style, LV_STATE_DEFAULT, LV_GRAD_DIR_VER);

    /*Shift the gradient to the bottom*/
    lv_style_set_bg_main_stop(&style, LV_STATE_DEFAULT, 128);
    lv_style_set_bg_grad_stop(&style, LV_STATE_DEFAULT, 192);


    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Border properties

The border is drawn on top of the *background*. It has `radius` rounding.

- **border_color** (`lv_color_t`) Specifies the color of the border. Default value: `LV_COLOR_BLACK`.
- **border_opa** (`lv_opa_t`) Specifies opacity of the border. Default value: `LV_OPA_COVER`.
- **border_width** (`lv_style_int_t`): Set the width of the border. Default value: 0.
- **border_side** (`lv_border_side_t`) Specifies which sides of the border to draw. Can be `LV_BORDER_SIDE_NONE/LEFT/RIGHT/TOP/BOTTOM/FULL`. ORed values are also possible. Default value: `LV_BORDER_SIDE_FULL`.
- **border_post** (`bool`): If `true` the border will be drawn after all children have been drawn. Default value: `false`.
- **border_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the border. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.

```c
#include "../../lv_examples.h"

/**
 * Using the border style properties
 */
void lv_ex_style_2(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add border to the bottom+right*/
    lv_style_set_border_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
```

(continues on next page)

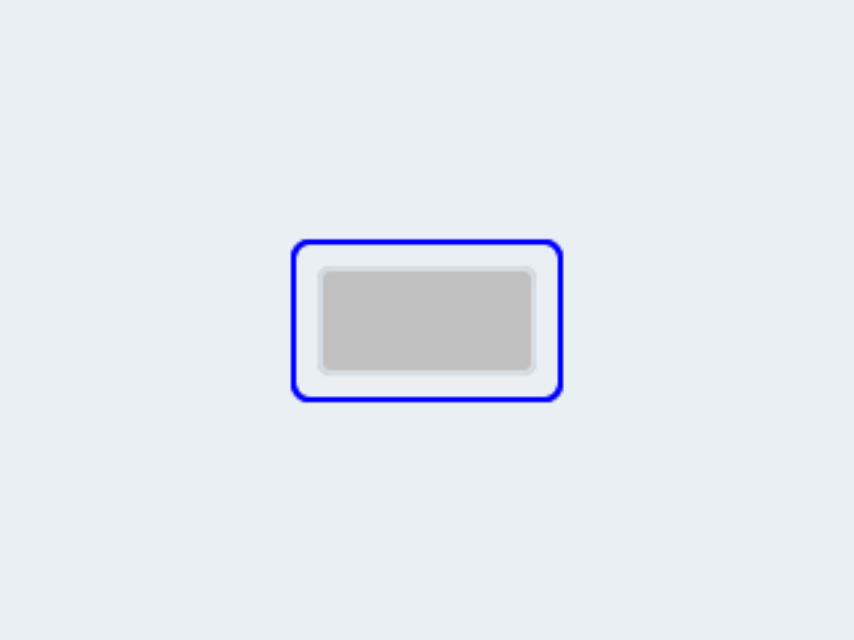
(continued from previous page)

```
    lv_style_set_border_width(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_border_opa(&style, LV_STATE_DEFAULT, LV_OPA_50);
    lv_style_set_border_side(&style, LV_STATE_DEFAULT, LV_BORDER_SIDE_BOTTOM | LV_
↪BORDER_SIDE_RIGHT);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Outline properties

The outline is similar to *border* but is drawn outside of the object.

- **outline_color** (`lv_color_t`) Specifies the color of the outline. Default value: `LV_COLOR_BLACK`.
- **outline_opa** (`lv_opa_t`) Specifies opacity of the outline. Default value: `LV_OPA_COVER`.
- **outline_width** (`lv_style_int_t`): Set the width of the outline. Default value: 0.
- **outline_pad** (`lv_style_int_t`) Set the space between the object and the outline. Default value: 0.
- **outline_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the outline. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.



```
#include "../../lv_examples.h"

/**
 * Using the outline style properties
 */
void lv_ex_style_3(void)
```

(continues on next page)

(continued from previous page)

```
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add outline*/
    lv_style_set_outline_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_outline_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_outline_pad(&style, LV_STATE_DEFAULT, 8);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Shadow properties

The shadow is a blurred area under the object.

- **shadow_color** (`lv_color_t`) Specifies the color of the shadow. Default value: `LV_COLOR_BLACK`.
- **shadow_opa** (`lv_opa_t`) Specifies opacity of the shadow. Default value: `LV_OPA_TRANSP`.
- **shadow_width** (`lv_style_int_t`): Set the width (blur size) of the outline. Default value: 0.
- **shadow_ofs_x** (`lv_style_int_t`): Set the an X offset for the shadow. Default value: 0.
- **shadow_ofs_y** (`lv_style_int_t`): Set the an Y offset for the shadow. Default value: 0.
- **shadow_spread** (`lv_style_int_t`): make the shadow larger than the background in every direction by this value. Default value: 0.
- **shadow_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the shadow. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.

```c
#include "../../lv_examples.h"

/**
 * Using the Shadow style properties
 */
void lv_ex_style_4(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add a shadow*/
    lv_style_set_shadow_width(&style, LV_STATE_DEFAULT, 8);
    lv_style_set_shadow_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_shadow_ofs_x(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_shadow_ofs_y(&style, LV_STATE_DEFAULT, 20);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Pattern properties

The pattern is an image (or symbol) drawn in the middle of the background or repeated to fill the whole background.

- **pattern_image** (`const void *`): Pointer to an `lv_img_dsc_t` variable, a path to an image file or a symbol. Default value: `NULL`.
- **pattern_opa** (`lv_opa_t`): Specifies opacity of the pattern. Default value: `LV_OPA_COVER`.
- **pattern_recolor** (`lv_color_t`): Mix this color to the pattern image. In case of symbols (texts) it will be the text color. Default value: `LV_COLOR_BLACK`.
- **pattern_recolor_opa** (`lv_opa_t`): Intensity of recoloring. Default value: `LV_OPA_TRANSP` (no recoloring).
- **pattern_repeat** (`bool`): `true`: the pattern will be repeated as a mosaic. `false`: place the pattern in the middle of the background. Default value: `false`.
- **pattern_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the pattern. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.

```c
#include "../../lv_examples.h"

/**
 * Using the pattern style properties
 */
void lv_ex_style_5(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
```

(continues on next page)

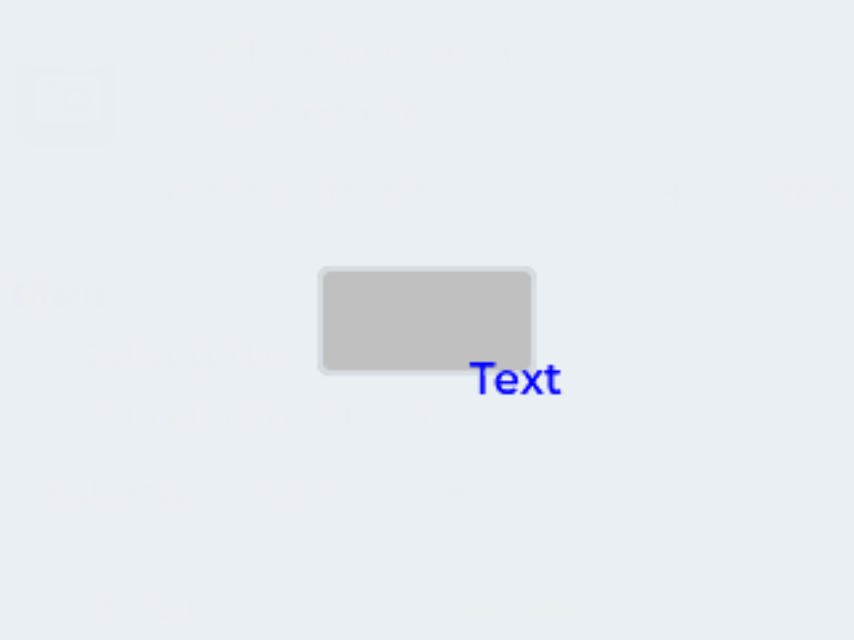
(continued from previous page)

```
    /*Add a repeating pattern*/
    lv_style_set_pattern_image(&style, LV_STATE_DEFAULT, LV_SYMBOL_OK);
    lv_style_set_pattern_recolor(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_pattern_opa(&style, LV_STATE_DEFAULT, LV_OPA_50);
    lv_style_set_pattern_repeat(&style, LV_STATE_DEFAULT, true);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Value properties

Value is an arbitrary text drawn to the background. It can be a lightweighted replacement of creating label objects.

- **value_str** (`const char *`): Pointer to text to display. Only the pointer is saved! (Don't use local variable with lv_style_set_value_str, instead use static, global or dynamically allocated data). Default value: `NULL`.
- **value_color** (`lv_color_t`): Color of the text. Default value: `LV_COLOR_BLACK`.
- **value_opa** (`lv_opa_t`): Opacity of the text. Default value: `LV_OPA_COVER`.
- **value_font** (`const lv_font_t *`): Pointer to font of the text. Default value: `NULL`.
- **value_letter_space** (`lv_style_int_t`): Letter space of the text. Default value: 0.
- **value_line_space** (`lv_style_int_t`): Line space of the text. Default value: 0.
- **value_align** (`lv_align_t`): Alignment of the text. Can be `LV_ALIGN_...`. Default value: `LV_ALIGN_CENTER`.
- **value_ofs_x** (`lv_style_int_t`): X offset from the original position of the alignment. Default value: 0.
- **value_ofs_y** (`lv_style_int_t`): Y offset from the original position of the alignment. Default value: 0.
- **value_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the text. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.




```c
#include "../../lv_examples.h"

/**
 * Using the value style properties
 */
void lv_ex_style_6(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Add a value text properties*/
    lv_style_set_value_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_value_align(&style, LV_STATE_DEFAULT, LV_ALIGN_IN_BOTTOM_RIGHT);
    lv_style_set_value_ofs_x(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_value_ofs_y(&style, LV_STATE_DEFAULT, 10);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);

    /*Add a value text to the local style. This way every object can have different␣
↪text*/
    lv_obj_set_style_local_value_str(obj, LV_OBJ_PART_MAIN, LV_STATE_DEFAULT, "Text");
}
```

Text properties

Properties for textual object.

- **text_color** (`lv_color_t`): Color of the text. Default value: `LV_COLOR_BLACK`.
- **text_opa** (`lv_opa_t`): Opacity of the text. Default value: `LV_OPA_COVER`.
- **text_font** (`const lv_font_t *`): Pointer to font of the text. Default value: `NULL`.
- **text_letter_space** (`lv_style_int_t`): Letter space of the text. Default value: 0.
- **text_line_space** (`lv_style_int_t`): Line space of the text. Default value: 0.
- **text_decor** (`lv_text_decor_t`): Add text decoration. Can be `LV_TEXT_DECOR_NONE/UNDERLINE/STRIKETHROUGH`. Default value: `LV_TEXT_DECOR_NONE`.
- **text_sel_color** (`lv_color_t`): Set color of the text selection. Default value: `LV_COLOR_BLACK`
- **text_sel_bg_color** (`lv_color_t`): Set background color of text selection. Default value: `LV_COLOR_BLUE`
- **text_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the text. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.


```c
#include "../../lv_examples.h"

/**
 * Using the text style properties
 */
void lv_ex_style_7(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
```

(continues on next page)

(continued from previous page)

```
    lv_style_set_border_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_border_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);

    lv_style_set_pad_top(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_bottom(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_left(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_right(&style, LV_STATE_DEFAULT, 10);

    lv_style_set_text_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_text_letter_space(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_text_line_space(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_text_decor(&style, LV_STATE_DEFAULT, LV_TEXT_DECOR_UNDERLINE);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_label_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_LABEL_PART_MAIN, &style);
    lv_label_set_text(obj, "Text of\n"
                           "a label");
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Line properties

Properties of lines.

- **line_color** (`lv_color_t`): Color of the line. Default value: `LV_COLOR_BLACK`
- **line_opa** (`lv_opa_t`): Opacity of the line. Default value: `LV_OPA_COVER`
- **line_width** (`lv_style_int_t`): Width of the line. Default value: 0.
- **line_dash_width** (`lv_style_int_t`): Width of dash. Dashing is drawn only for horizontal or vertical lines. 0: disable dash. Default value: 0.
- **line_dash_gap** (`lv_style_int_t`): Gap between two dash line. Dashing is drawn only for horizontal or vertical lines. 0: disable dash. Default value: 0.
- **line_rounded** (`bool`): `true`: draw rounded line endings. Default value: `false`.
- **line_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the line. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.

```c
#include "../../lv_examples.h"

/**
 * Using the line style properties
 */
void lv_ex_style_8(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    lv_style_set_line_color(&style, LV_STATE_DEFAULT, LV_COLOR_GRAY);
    lv_style_set_line_width(&style, LV_STATE_DEFAULT, 6);
    lv_style_set_line_rounded(&style, LV_STATE_DEFAULT, true);
#if LV_USE_LINE
    /*Create an object with the new style*/
    lv_obj_t * obj = lv_line_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_LINE_PART_MAIN, &style);

    static lv_point_t p[] = {{10, 30}, {30, 50}, {100, 0}};
    lv_line_set_points(obj, p, 3);

    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
#endif
}
```

Image properties

Properties of image.

- **image_recolor** (`lv_color_t`): Mix this color to the pattern image. In case of symbols (texts) it will be the text color. Default value: `LV_COLOR_BLACK`
- **image_recolor_opa** (`lv_opa_t`): Intensity of recoloring. Default value: `LV_OPA_TRANSP` (no recoloring). Default value: `LV_OPA_TRANSP`
- **image_opa** (`lv_opa_t`): Opacity of the image. Default value: `LV_OPA_COVER`
- **image_blend_mode** (`lv_blend_mode_t`): Set the blend mode of the image. Can be `LV_BLEND_MODE_NORMAL/ADDITIVE/SUBTRACTIVE`). Default value: `LV_BLEND_MODE_NORMAL`.

```c
#include "../../lv_examples.h"

/**
 * Using the image style properties
 */
void lv_ex_style_9(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_border_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_border_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);

    lv_style_set_pad_top(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_bottom(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_left(&style, LV_STATE_DEFAULT, 10);
    lv_style_set_pad_right(&style, LV_STATE_DEFAULT, 10);
```

(continues on next page)

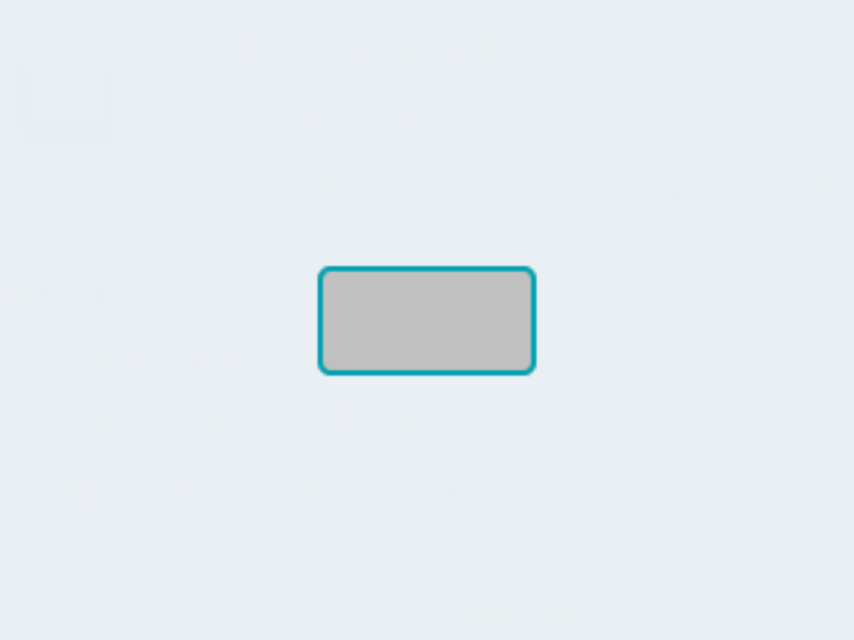
(continued from previous page)

```
    lv_style_set_image_recolor(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_image_recolor_opa(&style, LV_STATE_DEFAULT, LV_OPA_50);

#if LV_USE_IMG
    /*Create an object with the new style*/
    lv_obj_t * obj = lv_img_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_IMG_PART_MAIN, &style);
    LV_IMG_DECLARE(img_cogwheel_argb);
    lv_img_set_src(obj, &img_cogwheel_argb);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
#endif
}
```

Transition properties

Properties to describe state change animations.

- **transition_time** (`lv_style_int_t`): Time of the transition. Default value: 0.
- **transition_delay** (`lv_style_int_t`): Delay before the transition. Default value: 0.
- **transition_prop_1** (`property name`): A property on which transition should be applied. Use the property name with upper case with `LV_STYLE_` prefix, e.g. `LV_STYLE_BG_COLOR`. Default value: 0 (none).
- **transition_prop_2** (`property name`): Same as *transition_1* just for another property. Default value: 0 (none).
- **transition_prop_3** (`property name`): Same as *transition_1* just for another property. Default value: 0 (none).
- **transition_prop_4** (`property name`): Same as *transition_1* just for another property. Default value: 0 (none).
- **transition_prop_5** (`property name`): Same as *transition_1* just for another property. Default value: 0 (none).
- **transition_prop_6** (`property name`): Same as *transition_1* just for another property. Default value: 0 (none).
- **transition_path** (`lv_anim_path_t`): An animation path for the transition. (Needs to be static or global variable because only its pointer is saved). Default value: `lv_anim_path_def` (linear path).



```c
#include "../../lv_examples.h"

/**
 * Using the transitions style properties
 */
void lv_ex_style_10(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Set different background color in pressed state*/
    lv_style_set_bg_color(&style, LV_STATE_PRESSED, LV_COLOR_GRAY);

    /*Set different transition time in default and pressed state
     *fast press, slower revert to default*/
    lv_style_set_transition_time(&style, LV_STATE_DEFAULT, 500);
    lv_style_set_transition_time(&style, LV_STATE_PRESSED, 200);

    /*Small delay to make transition more visible*/
    lv_style_set_transition_delay(&style, LV_STATE_DEFAULT, 100);

    /*Add `bg_color` to transitioned properties*/
    lv_style_set_transition_prop_1(&style, LV_STATE_DEFAULT, LV_STYLE_BG_COLOR);

    /*Create an object with the new style*/
    lv_obj_t * obj = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_OBJ_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

Scale properties

Auxiliary properties for scale-like elements. Scales have a normal and end region. As the name implies the end region is the end of the scale where can be critical values or inactive values. The normal region is before the end region. Both regions could have different properties.

- **scale_grad_color** (`lv_color_t`): In normal region make gradient to this color on the scale lines. Default value: `LV_COLOR_BLACK`.
- **scale_end_color** (`lv_color_t`): Color of the scale lines in the end region. Default value: `LV_COLOR_BLACK`.
- **scale_width** (`lv_style_int_t`): Width of the scale. Default value: `LV_DPI / 8`. Default value: `LV_DPI / 8`.
- **scale_border_width** (`lv_style_int_t`): Width of a border drawn on the outer side of the scale in the normal region. Default value: 0.
- **scale_end_border_width** (`lv_style_int_t`): Width of a border drawn on the outer side of the scale in the end region. Default value: 0.
- **scale_end_line_width** (`lv_style_int_t`): Width of a scale lines in the end region. Default value: 0.


```c
#include "../../lv_examples.h"

/**
 * Using the scale style properties
 */
void lv_ex_style_11(void)
{
    static lv_style_t style;
    lv_style_init(&style);

    /*Set a background color and a radius*/
    lv_style_set_radius(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_bg_opa(&style, LV_STATE_DEFAULT, LV_OPA_COVER);
```

(continues on next page)

(continued from previous page)

```
    lv_style_set_bg_color(&style, LV_STATE_DEFAULT, LV_COLOR_SILVER);

    /*Set some paddings*/
    lv_style_set_pad_inner(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_pad_top(&style, LV_STATE_DEFAULT, 20);
    lv_style_set_pad_left(&style, LV_STATE_DEFAULT, 5);
    lv_style_set_pad_right(&style, LV_STATE_DEFAULT, 5);

    lv_style_set_scale_end_color(&style, LV_STATE_DEFAULT, LV_COLOR_RED);
    lv_style_set_line_color(&style, LV_STATE_DEFAULT, LV_COLOR_WHITE);
    lv_style_set_scale_grad_color(&style, LV_STATE_DEFAULT, LV_COLOR_BLUE);
    lv_style_set_line_width(&style, LV_STATE_DEFAULT, 2);
    lv_style_set_scale_end_line_width(&style, LV_STATE_DEFAULT, 4);
    lv_style_set_scale_end_border_width(&style, LV_STATE_DEFAULT, 4);

    /*Gauge has a needle but for simplicity its style is not initialized here*/
#if LV_USE_GAUGE
    /*Create an object with the new style*/
    lv_obj_t * obj = lv_gauge_create(lv_scr_act(), NULL);
    lv_obj_add_style(obj, LV_GAUGE_PART_MAIN, &style);
    lv_obj_align(obj, NULL, LV_ALIGN_CENTER, 0, 0);
#endif
}
```

In the documentation of the widgets you will see sentences like "The widget use the typical background properties". The "typical background" properties are:

- Background
- Border
- Outline
- Shadow
- Pattern
- Value

4.4.10 Themes

Themes are a collection of styles. There is always an active theme whose styles are automatically applied when an object is created. It gives a default appearance to UI which can be modified by adding further styles.

The default theme is set in `lv_conf.h` with `LV_THEME_...` defines. Every theme has the following properties

- primary color
- secondary color
- small font
- normal font
- subtitle font
- title font

- flags (specific to the given theme)

It up to the theme how to use these properties.

There are 3 built-in themes:

- empty: no default styles are added
- material: an impressive, modern theme - mono: simple black and white theme for monochrome displays
- template: a very simple theme which can be copied to create a custom theme

Extending themes

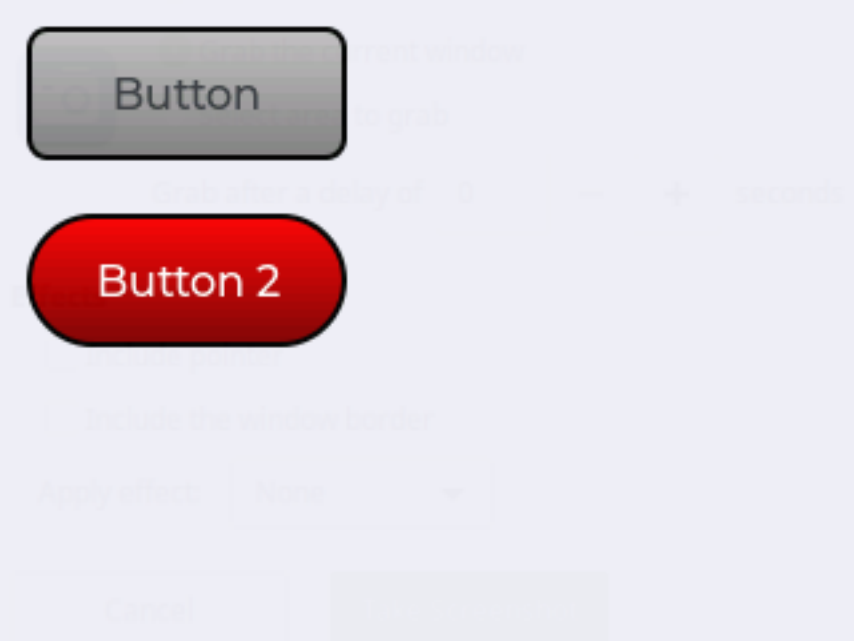
Built-in themes can be extended by custom theme. If a custom theme is created a "base theme" can be selected. The base theme's styles will be added before the custom theme. Any number of themes can be chained this was. E.g. material theme -> custom theme -> dark theme.

Here is an example about how to create a custom theme based on the currently active built-in theme.

```
 /*Get the current theme (e.g. material). It will be the base of the custom theme.*/
lv_theme_t * base_theme = lv_theme_get_act();

/*Initialize a custom theme*/
static lv_theme_t custom_theme;                          /*Declare a theme*/
lv_theme_copy(&custom_theme, base_theme);                /*Initialize the custom theme␣
↪from the base theme*/
lv_theme_set_apply_cb(&custom_theme, custom_apply_cb);  /*Set a custom theme apply␣
↪callback*/
lv_theme_set_base(custom_theme, base_theme);             /*Set the base theme of the␣
↪csutom theme*/

/*Initialize styles for the new theme*/
static lv_style_t style1;
lv_style_init(&style1);
lv_style_set_bg_color(&style1, LV_STATE_DEFAULT, custom_theme.color_primary);

...

/*Add a custom apply callback*/
static void custom_apply_cb(lv_theme_t * th, lv_obj_t * obj, lv_theme_style_t name)
{
    lv_style_list_t * list;

    switch(name) {
        case LV_THEME_BTN:
            list = lv_obj_get_style_list(obj, LV_BTN_PART_MAIN);
            _lv_style_list_add_style(list, &my_style);
            break;
    }
}
```

4.4.11 Example

Styling a button




```
#include "../../lv_examples.h"


/**
 * Create styles from scratch for buttons.
 */
void lv_ex_get_started_2(void)
{
    static lv_style_t style_btn;
    static lv_style_t style_btn_red;

    /*Create a simple button style*/
    lv_style_init(&style_btn);
    lv_style_set_radius(&style_btn, LV_STATE_DEFAULT, 10);
    lv_style_set_bg_opa(&style_btn, LV_STATE_DEFAULT, LV_OPA_COVER);
    lv_style_set_bg_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_SILVER);
    lv_style_set_bg_grad_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_GRAY);
    lv_style_set_bg_grad_dir(&style_btn, LV_STATE_DEFAULT, LV_GRAD_DIR_VER);

    /*Swap the colors in pressed state*/
    lv_style_set_bg_color(&style_btn, LV_STATE_PRESSED, LV_COLOR_GRAY);
    lv_style_set_bg_grad_color(&style_btn, LV_STATE_PRESSED, LV_COLOR_SILVER);

    /*Add a border*/
    lv_style_set_border_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_WHITE);
    lv_style_set_border_opa(&style_btn, LV_STATE_DEFAULT, LV_OPA_70);
    lv_style_set_border_width(&style_btn, LV_STATE_DEFAULT, 2);

    /*Different border color in focused state*/
    lv_style_set_border_color(&style_btn, LV_STATE_FOCUSED, LV_COLOR_BLUE);
```

(continues on next page)

(continued from previous page)

```
    lv_style_set_border_color(&style_btn, LV_STATE_FOCUSED | LV_STATE_PRESSED, LV_
↪COLOR_NAVY);

    /*Set the text style*/
    lv_style_set_text_color(&style_btn, LV_STATE_DEFAULT, LV_COLOR_WHITE);

    /*Make the button smaller when pressed*/
    lv_style_set_transform_height(&style_btn, LV_STATE_PRESSED, -5);
    lv_style_set_transform_width(&style_btn, LV_STATE_PRESSED, -10);
#if LV_USE_ANIMATION
    /*Add a transition to the size change*/
    static lv_anim_path_t path;
    lv_anim_path_init(&path);
    lv_anim_path_set_cb(&path, lv_anim_path_overshoot);

    lv_style_set_transition_prop_1(&style_btn, LV_STATE_DEFAULT, LV_STYLE_TRANSFORM_
↪HEIGHT);
    lv_style_set_transition_prop_2(&style_btn, LV_STATE_DEFAULT, LV_STYLE_TRANSFORM_
↪WIDTH);
    lv_style_set_transition_time(&style_btn, LV_STATE_DEFAULT, 300);
    lv_style_set_transition_path(&style_btn, LV_STATE_DEFAULT, &path);
#endif

    /*Create a red style. Change only some colors.*/
    lv_style_init(&style_btn_red);
    lv_style_set_bg_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_RED);
    lv_style_set_bg_grad_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_MAROON);
    lv_style_set_bg_color(&style_btn_red, LV_STATE_PRESSED, LV_COLOR_MAROON);
    lv_style_set_bg_grad_color(&style_btn_red, LV_STATE_PRESSED, LV_COLOR_RED);
    lv_style_set_text_color(&style_btn_red, LV_STATE_DEFAULT, LV_COLOR_WHITE);
#if LV_USE_BTN
    /*Create buttons and use the new styles*/
    lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);     /*Add a button the␣
↪current screen*/
    lv_obj_set_pos(btn, 10, 10);                            /*Set its position*/
    lv_obj_set_size(btn, 120, 50);                          /*Set its size*/
    lv_obj_reset_style_list(btn, LV_BTN_PART_MAIN);         /*Remove the styles␣
↪coming from the theme*/
    lv_obj_add_style(btn, LV_BTN_PART_MAIN, &style_btn);

    lv_obj_t * label = lv_label_create(btn, NULL);          /*Add a label to the␣
↪button*/
    lv_label_set_text(label, "Button");                     /*Set the labels text*/

    /*Create a new button*/
    lv_obj_t * btn2 = lv_btn_create(lv_scr_act(), btn);
    lv_obj_set_pos(btn2, 10, 80);
    lv_obj_set_size(btn2, 120, 50);                            /*Set its size*/
    lv_obj_reset_style_list(btn2, LV_BTN_PART_MAIN);         /*Remove the styles␣
↪coming from the theme*/
    lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_btn);
    lv_obj_add_style(btn2, LV_BTN_PART_MAIN, &style_btn_red);   /*Add the red style␣
↪on top of the current */
    lv_obj_set_style_local_radius(btn2, LV_BTN_PART_MAIN, LV_STATE_DEFAULT, LV_RADIUS_
↪CIRCLE); /*Add a local style*/
```

(continues on next page)

(continued from previous page)

```
    label = lv_label_create(btn2, NULL);          /*Add a label to the button*/
    lv_label_set_text(label, "Button 2");                    /*Set the labels text*/
#endif
}
```

4.4.12 API

Typedefs

typedef uint8_t **lv_border_side_t**

typedef uint8_t **lv_grad_dir_t**

typedef uint8_t **lv_text_decor_t**

typedef uint8_t **lv_style_attr_t**

typedef uint16_t **lv_style_property_t**

typedef uint16_t **lv_style_state_t**

typedef int16_t **lv_style_int_t**

Enums

enum [anonymous]

Values:

enumerator LV_BORDER_SIDE_NONE

enumerator LV_BORDER_SIDE_BOTTOM

enumerator LV_BORDER_SIDE_TOP

enumerator LV_BORDER_SIDE_LEFT

enumerator LV_BORDER_SIDE_RIGHT

enumerator LV_BORDER_SIDE_FULL

enumerator LV_BORDER_SIDE_INTERNAL

FOR matrix-like objects (e.g. Button matrix)

enumerator _LV_BORDER_SIDE_LAST

enum [anonymous]

Values:

enumerator LV_GRAD_DIR_NONE

enumerator LV_GRAD_DIR_VER

enumerator LV_GRAD_DIR_HOR

enumerator _LV_GRAD_DIR_LAST

enum [anonymous]

Values:

enumerator LV_TEXT_DECOR_NONE

enumerator LV_TEXT_DECOR_UNDERLINE

enumerator LV_TEXT_DECOR_STRIKETHROUGH

enumerator _LV_TEXT_DECOR_LAST

enum [anonymous]
Values:

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

enumerator LV_STYLE_PROP_INIT

Functions

LV_EXPORT_CONST_INT(LV_RADIUS_CIRCLE)

void **lv_style_init**(*lv_style_t *style*)

Initialize a style

Parameters **style** -- pointer to a style to initialize

void **lv_style_copy**(*lv_style_t *style_dest*, **const** *lv_style_t *style_src*)

Copy a style with all its properties

Copy a style to an other

Parameters

- **style_dest** -- pointer to the destination style. (Should be initialized with *lv_style_init()*)
- **style_src** -- pointer to the source (to copy)style
- **dest** -- pointer to the destination style
- **src** -- pointer to the source style

void **lv_style_list_init**(*lv_style_list_t *list*)
Initialize a style list

Parameters **list** -- a style list to initialize

void **lv_style_list_copy**(*lv_style_list_t *list_dest*, **const** *lv_style_list_t *list_src*)
Copy a style list with all its styles and local style properties

Parameters

- **list_dest** -- pointer to the destination style list. (should be initialized with *lv_style_list_init()*)
- **list_src** -- pointer to the source (to copy) style list.

void **_lv_style_list_add_style**(*lv_style_list_t *list*, *lv_style_t *style*)
Add a style to a style list. Only the style pointer will be saved so the shouldn't be a local variable. (It should be static, global or dynamically allocated)

Parameters

- **list** -- pointer to a style list
- **style** -- pointer to a style to add

void **_lv_style_list_remove_style**(*lv_style_list_t *list*, *lv_style_t *style*)
Remove a style from a style list

Parameters

- **style_list** -- pointer to a style list
- **style** -- pointer to a style to remove

void **_lv_style_list_reset**(*lv_style_list_t *style_list*)
Remove all styles added from style list, clear the local style, transition style and free all allocated memories. Leave `ignore_trans` flag as it is.

Parameters **list** -- pointer to a style list.

static inline *lv_style_t* ***lv_style_list_get_style**(*lv_style_list_t *list*, uint8_t *id*)

void **lv_style_reset**(*lv_style_t *style*)
Clear all properties from a style and all allocated memories.

Parameters **style** -- pointer to a style

uint16_t **_lv_style_get_mem_size**(**const** *lv_style_t *style*)
Get the size of the properties in a style in bytes

Parameters **style** -- pointer to a style

Returns size of the properties in bytes

bool **lv_style_remove_prop**(*lv_style_t *style*, *lv_style_property_t prop*)
Remove a property from a style

Parameters

- **style** -- pointer to a style
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`

Returns true: the property was found and removed; false: the property wasn't found

void **_lv_style_set_int**(*lv_style_t *style*, *lv_style_property_t prop*, *lv_style_int_t value*)
Set an integer typed property in a style.

Note: shouldn't be used directly. Use the specific property set functions instead. For example: `lv_style_set_border_width()`

Note: for performance reasons it's not checked if the property really has integer type

---

Parameters

- **style** -- pointer to a style where the property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

void **_lv_style_set_color**(*lv_style_t *style*, *lv_style_property_t prop*, lv_color_t *color*)
Set a color typed property in a style.

Note: shouldn't be used directly. Use the specific property set functions instead. For example: `lv_style_set_border_color()`

Note: for performance reasons it's not checked if the property really has color type

---

Parameters

- **style** -- pointer to a style where the property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

void **_lv_style_set_opa**(*lv_style_t *style*, *lv_style_property_t prop*, lv_opa_t *opa*)
Set an opacity typed property in a style.

Note: shouldn't be used directly. Use the specific property set functions instead. For example: `lv_style_set_border_opa()`

Note: for performance reasons it's not checked if the property really has opacity type

---

Parameters

- **style** -- pointer to a style where the property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

void **_lv_style_set_ptr**(*lv_style_t *style*, *lv_style_property_t prop*, **const** void *p)

Set a pointer typed property in a style.

Note: shouldn't be used directly. Use the specific property set functions instead. For example: `lv_style_set_border_width()`

Note: for performance reasons it's not checked if the property really has pointer type

---

Parameters

- **style** -- pointer to a style where the property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_TEXT_POINTER | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

int16_t **_lv_style_get_int**(**const** *lv_style_t *style*, *lv_style_property_t prop*, *lv_style_int_t *res*)

Get an integer typed property from a style.

Note: shouldn't be used directly. Use the specific property get functions instead. For example: `lv_style_get_border_width()`

Note: for performance reasons it's not checked if the property really has integer type

---

Parameters

- **style** -- pointer to a style from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result value

Returns -1: the property wasn't found in the style. The matching state bits of the desired state (in `prop`) and the best matching property's state Higher value means match in higher precedence state.

int16_t **_lv_style_get_color**(**const** *lv_style_t *style*, *lv_style_property_t prop*, lv_color_t *res*)

Get a color typed property from a style.

Note: shouldn't be used directly. Use the specific property get functions instead. For example: `lv_style_get_border_color()`

Note: for performance reasons it's not checked if the property really has color type

Parameters

- **style** -- pointer to a style from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result value

Returns -1: the property wasn't found in the style. The matching state bits of the desired state (in `prop`) and the best matching property's state Higher value means match in higher precedence state.

int16_t **_lv_style_get_opa(const** *lv_style_t *style*, *lv_style_property_t prop*, lv_opa_t ***res*)
Get an opacity typed property from a style.

Note: shouldn't be used directly. Use the specific property get functions instead. For example: `lv_style_get_border_opa()`

Note: for performance reasons it's not checked if the property really has opacity type

Parameters

- **style** -- pointer to a style from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result value

Returns -1: the property wasn't found in the style. The matching state bits of the desired state (in `prop`) and the best matching property's state Higher value means match in higher precedence state.

int16_t **_lv_style_get_ptr(const** *lv_style_t *style*, *lv_style_property_t prop*, **const** void ****res*)
Get a pointer typed property from a style.

Note: shouldn't be used directly. Use the specific property get functions instead. For example: `lv_style_get_text_font()`

Note: for performance reasons it's not checked if the property really has pointer type

Parameters

- **style** -- pointer to a style from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result value

Returns -1: the property wasn't found in the style. The matching state bits of the desired state (in `prop`) and the best matching property's state Higher value means match in higher precedence state.

lv_style_t ***lv_style_list_get_local_style**(*lv_style_list_t *list*)
Get the local style of a style list

Parameters **list** -- pointer to a style list where the local property should be set

Returns pointer to the local style if exists else `NULL`.

lv_style_t ***_lv_style_list_get_transition_style**(*lv_style_list_t *list*)
Get the transition style of a style list

Parameters **list** -- pointer to a style list where the transition property should be set

Returns pointer to the transition style if exists else `NULL`.

lv_style_t ***_lv_style_list_add_trans_style**(*lv_style_list_t *list*)
Allocate the transition style in a style list. If already exists simply return it.

Parameters **list** -- pointer to a style list

Returns the transition style of a style list

void **_lv_style_list_set_local_int**(*lv_style_list_t *list*, *lv_style_property_t prop*, *lv_style_int_t value*)
Set a local integer typed property in a style list.

Note: for performance reasons it's not checked if the property really has integer type

---

Parameters

- **list** -- pointer to a style list where the local property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

void **_lv_style_list_set_local_color**(*lv_style_list_t *list*, *lv_style_property_t prop*, lv_color_t *value*)
Set a local color typed property in a style list.

Note: for performance reasons it's not checked if the property really has color type

---

Parameters

- **list** -- pointer to a style list where the local property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`

- **value** -- the value to set

void **_lv_style_list_set_local_opa**(*lv_style_list_t *list*, *lv_style_property_t prop*, lv_opa_t *value*)

Set a local opacity typed property in a style list.

Note: for performance reasons it's not checked if the property really has opacity type

---

Parameters

- **list** -- pointer to a style list where the local property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

void **_lv_style_list_set_local_ptr**(*lv_style_list_t *list*, *lv_style_property_t prop*, **const** void *value*)

Set a local pointer typed property in a style list.

Note: for performance reasons it's not checked if the property really has pointer type

---

Parameters

- **list** -- pointer to a style list where the local property should be set
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **value** -- the value to set

lv_res_t **_lv_style_list_get_int**(*lv_style_list_t *list*, *lv_style_property_t prop*, *lv_style_int_t *res*)

Get an integer typed property from a style list. It will return the property which match best with given state.

Note: for performance reasons it's not checked if the property really has integer type

---

Parameters

- **list** -- pointer to a style list from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result

Returns LV_RES_OK: there was a matching property in the list LV_RES_INV: there was NO matching property in the list

lv_res_t **_lv_style_list_get_color**(*lv_style_list_t *list*, *lv_style_property_t prop*, lv_color_t *res*)

Get a color typed property from a style list. It will return the property which match best with given state.

Note: for performance reasons it's not checked if the property really has color type

Parameters

- **list** -- pointer to a style list from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result

Returns LV_RES_OK: there was a matching property in the list LV_RES_INV: there was NO matching property in the list

lv_res_t **_lv_style_list_get_opa**(*lv_style_list_t *list*, *lv_style_property_t prop*, lv_opa_t **res*)

Get an opacity typed property from a style list. It will return the property which match best with given state.

Note: for performance reasons it's not checked if the property really has opacity type

Parameters

- **list** -- pointer to a style list from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result

Returns LV_RES_OK: there was a matching property in the list LV_RES_INV: there was NO matching property in the list

lv_res_t **_lv_style_list_get_ptr**(*lv_style_list_t *list*, *lv_style_property_t prop*, **const** void **res*)

Get a pointer typed property from a style list. It will return the property which match best with given state.

Note: for performance reasons it's not checked if the property really has pointer type

Parameters

- **list** -- pointer to a style list from where the property should be get
- **prop** -- a style property ORed with a state. E.g. `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- **res** -- pointer to a buffer to store the result

Returns LV_RES_OK: there was a matching property in the list LV_RES_INV: there was NO matching property in the list

bool **lv_debug_check_style**(**const** *lv_style_t *style*)

Check whether a style is valid (initialized correctly)

Parameters **style** -- pointer to a style

Returns true: valid

bool **lv_debug_check_style_list**(**const** *lv_style_list_t *list*)
Check whether a style list is valid (initialized correctly)

Parameters **list** -- pointer to a style list

Returns true: valid

struct lv_style_t

Public Members

uint8_t ***map**

uint32_t **sentinel**

struct lv_style_list_t

Public Members

lv_style_t ****style_list**

uint32_t **sentinel**

uint32_t **style_cnt**

uint32_t **has_local**

uint32_t **has_trans**

uint32_t **skip_trans**

uint32_t **ignore_trans**

uint32_t **valid_cache**

uint32_t **ignore_cache**

uint32_t **radius_zero**

uint32_t **opa_scale_cover**

uint32_t **clip_corner_off**

uint32_t **transform_all_zero**

uint32_t **pad_all_zero**

uint32_t **margin_all_zero**

uint32_t **blend_mode_all_normal**

uint32_t **bg_opa_transp**

uint32_t **bg_opa_cover**

uint32_t **border_width_zero**

uint32_t **border_side_full**

uint32_t **border_post_off**

uint32_t **outline_width_zero**

uint32_t **pattern_img_null**

uint32_t **shadow_width_zero**

uint32_t **value_txt_str**

uint32_t **img_recolor_opa_transp**

uint32_t **text_space_zero**

uint32_t **text_decor_none**

uint32_t **text_font_normal**

Typedefs

typedef void (***lv_theme_apply_cb_t**)(**struct** *_lv_theme_t**, *lv_obj_t**, *lv_theme_style_t*)

typedef void (***lv_theme_apply_xcb_t**)(*lv_obj_t**, *lv_theme_style_t*)

typedef struct *_lv_theme_t* **lv_theme_t**

Enums

enum lv_theme_style_t

A theme in LVGL consists of many styles bound together.

There is a style for each object type, as well as a generic style for backgrounds and panels.

Values:

enumerator LV_THEME_NONE

enumerator LV_THEME_SCR

enumerator LV_THEME_OBJ

enumerator LV_THEME_ARC

enumerator LV_THEME_BAR

enumerator LV_THEME_BTN

enumerator LV_THEME_BTNMATRIX

enumerator LV_THEME_CALENDAR

enumerator LV_THEME_CANVAS

enumerator LV_THEME_CHECKBOX

enumerator LV_THEME_CHART

enumerator LV_THEME_CONT

enumerator LV_THEME_CPICKER

enumerator LV_THEME_DROPDOWN

enumerator LV_THEME_GAUGE

enumerator LV_THEME_IMAGE

enumerator LV_THEME_IMGBTN

enumerator LV_THEME_KEYBOARD

enumerator LV_THEME_LABEL

enumerator LV_THEME_LED

enumerator LV_THEME_LINE

enumerator LV_THEME_LIST

enumerator LV_THEME_LIST_BTN

enumerator LV_THEME_LINEMETER

enumerator LV_THEME_MSGBOX

enumerator LV_THEME_MSGBOX_BTNS

enumerator LV_THEME_OBJMASK

enumerator LV_THEME_PAGE

enumerator LV_THEME_ROLLER

enumerator LV_THEME_SLIDER

enumerator LV_THEME_SPINBOX

enumerator LV_THEME_SPINBOX_BTN

enumerator LV_THEME_SPINNER

enumerator LV_THEME_SWITCH

enumerator LV_THEME_TABLE

enumerator LV_THEME_TABVIEW

enumerator LV_THEME_TABVIEW_PAGE

enumerator LV_THEME_TEXTAREA

enumerator LV_THEME_TILEVIEW

enumerator LV_THEME_WIN

enumerator LV_THEME_WIN_BTN

enumerator _LV_THEME_BUILTIN_LAST

enumerator LV_THEME_CUSTOM_START

enumerator _LV_THEME_CUSTOM_LAST

Functions

void **lv_theme_set_act**(*lv_theme_t* **th*)

Set a theme for the system. From now, all the created objects will use styles from this theme by default

Parameters **th** -- pointer to theme (return value of: 'lv_theme_init_xxx()')

lv_theme_t ***lv_theme_get_act**(void)

Get the current system theme.

Returns pointer to the current system theme. NULL if not set.

void **lv_theme_apply**(*lv_obj_t* **obj*, *lv_theme_style_t name*)

Apply the active theme on an object

Parameters

- **obj** -- pointer to an object

- **name** -- the name of the theme element to apply. E.g. LV_THEME_BTN

void **lv_theme_copy**(*lv_theme_t *theme*, **const** *lv_theme_t *copy*)
Copy a theme to an other or initialize a theme

Parameters

- **theme** -- pointer to a theme to initialize
- **copy** -- pointer to a theme to copy or NULL to initialize theme to empty

void **lv_theme_set_base**(*lv_theme_t *new_theme*, *lv_theme_t *base*)
Set a base theme for a theme. The styles from the base them will be added before the styles of the current theme. Arbitrary long chain of themes can be created by setting base themes.

Parameters

- **new_theme** -- pointer to theme which base should be set
- **base** -- pointer to the base theme

void **lv_theme_set_apply_cb**(*lv_theme_t *theme*, *lv_theme_apply_cb_t apply_cb*)
Set an apply callback for a theme. The apply callback is used to add styles to different objects

Parameters

- **theme** -- pointer to theme which callback should be set
- **apply_cb** -- pointer to the callback

const lv_font_t ***lv_theme_get_font_small**(void)
Get the small font of the theme

Returns pointer to the font

const lv_font_t ***lv_theme_get_font_normal**(void)
Get the normal font of the theme

Returns pointer to the font

const lv_font_t ***lv_theme_get_font_subtitle**(void)
Get the subtitle font of the theme

Returns pointer to the font

const lv_font_t ***lv_theme_get_font_title**(void)
Get the title font of the theme

Returns pointer to the font

lv_color_t **lv_theme_get_color_primary**(void)
Get the primary color of the theme

Returns the color

lv_color_t **lv_theme_get_color_secondary**(void)
Get the secondary color of the theme

Returns the color

uint32_t **lv_theme_get_flags**(void)
Get the flags of the theme

Returns the flags

struct _lv_theme_t

Public Members

lv_theme_apply_cb_t **apply_cb**

lv_theme_apply_xcb_t **apply_xcb**

struct *_lv_theme_t* ***base**
Apply the current theme's style on top of this theme.

lv_color_t **color_primary**

lv_color_t **color_secondary**

const lv_font_t ***font_small**

const lv_font_t ***font_normal**

const lv_font_t ***font_subtitle**

const lv_font_t ***font_title**

uint32_t **flags**

void ***user_data**

4.5 Input devices

An input device usually means:

- Pointer-like input device like touchpad or mouse
- Keypads like a normal keyboard or simple numeric keypad
- Encoders with left/right turn and push options
- External hardware buttons which are assigned to specific points on the screen

Important: Before reading further, please read the [Porting](/porting/indev) section of Input devices

4.5.1 Pointers

Pointer input devices can have a cursor. (typically for mouses)

```
...
lv_indev_t * mouse_indev = lv_indev_drv_register(&indev_drv);

LV_IMG_DECLARE(mouse_cursor_icon);                          /*Declare the image file.
↪*/
lv_obj_t * cursor_obj =  lv_img_create(lv_scr_act(), NULL); /*Create an image object␣
↪for the cursor */
lv_img_set_src(cursor_obj, &mouse_cursor_icon);             /*Set the image source*/
lv_indev_set_cursor(mouse_indev, cursor_obj);               /*Connect the image ␣
↪object to the driver*/
```

Note that the cursor object should have `lv_obj_set_click(cursor_obj, false)`. For images, *clicking* is disabled by default.

4.5.2 Keypad and encoder

You can fully control the user interface without touchpad or mouse using a keypad or encoder(s). It works similar to the *TAB* key on the PC to select the element in an application or a web page.

Groups

The objects, you want to control with keypad or encoder, needs to be added to a *Group*. In every group, there is exactly one focused object which receives the pressed keys or the encoder actions. For example, if a *Text area* is focused and you press some letter on a keyboard, the keys will be sent and inserted into the text area. Similarly, if a *Slider* is focused and you press the left or right arrows, the slider's value will be changed.

You need to associate an input device with a group. An input device can send the keys to only one group but, a group can receive data from more than one input device too.

To create a group use `lv_group_t * g = lv_group_create()` and to add an object to the group use `lv_group_add_obj(g, obj)`.

To associate a group with an input device use `lv_indev_set_group(indev, g)`, where `indev` is the return value of `lv_indev_drv_register()`

Keys

There are some predefined keys which have special meaning:

- **LV_KEY_NEXT** Focus on the next object
- **LV_KEY_PREV** Focus on the previous object
- **LV_KEY_ENTER** Triggers `LV_EVENT_PRESSED/CLICKED/LONG_PRESSED` etc. events
- **LV_KEY_UP** Increase value or move upwards
- **LV_KEY_DOWN** Decrease value or move downwards
- **LV_KEY_RIGHT** Increase value or move the the right
- **LV_KEY_LEFT** Decrease value or move the the left
- **LV_KEY_ESC** Close or exit (E.g. close a *Drop down list*)
- **LV_KEY_DEL** Delete (E.g. a character on the right in a *Text area*)
- **LV_KEY_BACKSPACE** Delete a character on the left (E.g. in a *Text area*)
- **LV_KEY_HOME** Go to the beginning/top (E.g. in a *Text area*)
- **LV_KEY_END** Go to the end (E.g. in a *Text area*))

The most important special keys are `LV_KEY_NEXT/PREV`, `LV_KEY_ENTER` and `LV_KEY_UP/DOWN/LEFT/RIGHT`. In your `read_cb` function, you should translate some of your keys to these special keys to navigate in the group and interact with the selected object.

Usually, it's enough to use only `LV_KEY_LEFT/RIGHT` because most of the objects can be fully controlled with them.

With an encoder, you should use only `LV_KEY_LEFT`, `LV_KEY_RIGHT`, and `LV_KEY_ENTER`.

Edit and navigate mode

Since a keypad has plenty of keys, it's easy to navigate between the objects and edit them using the keypad. But, the encoders have a limited number of "keys" hence, it is difficult to navigate using the default options. *Navigate* and *Edit* are created to avoid this problem with the encoders.

In *Navigate* mode, the encoders `LV_KEY_LEFT/RIGHT` is translated to `LV_KEY_NEXT/PREV`. Therefore the next or previous object will be selected by turning the encoder. Pressing `LV_KEY_ENTER` will change to *Edit* mode.

In *Edit* mode, `LV_KEY_NEXT/PREV` is usually used to edit the object. Depending on the object's type, a short or long press of `LV_KEY_ENTER` changes back to *Navigate* mode. Usually, an object which can not be pressed (like a *Slider*) leaves *Edit* mode on short click. But with objects where short click has meaning (e.g. *Button*), a long press is required.

Styling

If an object is focused either by clicking it via touchpad, or focused via an encoder or keypad it goes to `LV_STATE_FOCUSED`. Hence focused styles will be applied on it.

If the object goes to edit mode it goes to `LV_STATE_FOCUSED | LV_STATE_EDITED` state so these style properties will be shown.

For a more detaild description read the Style section.

4.5.3 API

Input device

Functions

void **_lv_indev_init**(void)
Initialize the display input device subsystem

void **_lv_indev_read_task**(*lv_task_t* **task*)
Called periodically to read the input devices

Parameters **task** -- pointer to the task itself

lv_indev_t ***lv_indev_get_act**(void)
Get the currently processed input device. Can be used in action functions too.

Returns pointer to the currently processed input device or NULL if no input device processing right now

lv_indev_type_t **lv_indev_get_type**(**const** *lv_indev_t* **indev*)
Get the type of an input device

Parameters **indev** -- pointer to an input device

Returns the type of the input device from `lv_hal_indev_type_t` (`LV_INDEV_TYPE_...`)

void **lv_indev_reset**(*lv_indev_t* **indev*, *lv_obj_t* **obj*)
Reset one or all input devices

Parameters

- **indev** -- pointer to an input device to reset or NULL to reset all of them

- **obj** -- pointer to an object which triggers the reset.

void **lv_indev_reset_long_press**(*lv_indev_t *indev*)
Reset the long press state of an input device

Parameters **indev_proc** -- pointer to an input device

void **lv_indev_enable**(*lv_indev_t *indev*, bool *en*)
Enable or disable an input devices

Parameters

- **indev** -- pointer to an input device
- **en** -- true: enable; false: disable

void **lv_indev_set_cursor**(*lv_indev_t *indev*, *lv_obj_t *cur_obj*)
Set a cursor for a pointer input device (for LV_INPUT_TYPE_POINTER and LV_INPUT_TYPE_BUTTON)

Parameters

- **indev** -- pointer to an input device
- **cur_obj** -- pointer to an object to be used as cursor

void **lv_indev_set_group**(*lv_indev_t *indev*, *lv_group_t *group*)
Set a destination group for a keypad input device (for LV_INDEV_TYPE_KEYPAD)

Parameters

- **indev** -- pointer to an input device
- **group** -- point to a group

void **lv_indev_set_button_points**(*lv_indev_t *indev*, **const** lv_point_t *points*[])
Set the an array of points for LV_INDEV_TYPE_BUTTON. These points will be assigned to the buttons to press a specific point on the screen

Parameters

- **indev** -- pointer to an input device
- **group** -- point to a group

void **lv_indev_get_point**(**const** *lv_indev_t *indev*, lv_point_t *point*)
Get the last point of an input device (for LV_INDEV_TYPE_POINTER and LV_INDEV_TYPE_BUTTON)

Parameters

- **indev** -- pointer to an input device
- **point** -- pointer to a point to store the result

lv_gesture_dir_t **lv_indev_get_gesture_dir**(**const** *lv_indev_t *indev*)
Get the current gesture direct

Parameters **indev** -- pointer to an input device

Returns current gesture direct

uint32_t **lv_indev_get_key**(**const** *lv_indev_t *indev*)
Get the last pressed key of an input device (for LV_INDEV_TYPE_KEYPAD)

Parameters **indev** -- pointer to an input device

Returns the last pressed key (0 on error)

bool **lv_indev_is_dragging**(**const** *lv_indev_t *indev*)
Check if there is dragging with an input device or not (for LV_INDEV_TYPE_POINTER and LV_INDEV_TYPE_BUTTON)

Parameters **indev** -- pointer to an input device

Returns true: drag is in progress

void **lv_indev_get_vect**(**const** *lv_indev_t *indev*, lv_point_t *point*)
Get the vector of dragging of an input device (for LV_INDEV_TYPE_POINTER and LV_INDEV_TYPE_BUTTON)

Parameters

- **indev** -- pointer to an input device
- **point** -- pointer to a point to store the vector

lv_res_t **lv_indev_finish_drag**(*lv_indev_t *indev*)
Manually finish dragging. `LV_SIGNAL_DRAG_END` and `LV_EVENT_DRAG_END` will be sent.

Parameters **indev** -- pointer to an input device

Returns `LV_RES_INV` if the object being dragged was deleted. Else `LV_RES_OK`.

void **lv_indev_wait_release**(*lv_indev_t *indev*)
Do nothing until the next release

Parameters **indev** -- pointer to an input device

lv_obj_t ***lv_indev_get_obj_act**(void)
Gets a pointer to the currently active object in indev proc functions. NULL if no object is currently being handled or if groups aren't used.

Returns pointer to currently active object

lv_obj_t ***lv_indev_search_obj**(*lv_obj_t *obj*, lv_point_t *point*)
Search the most top, clickable object by a point

Parameters

- **obj** -- pointer to a start object, typically the screen
- **point** -- pointer to a point for searching the most top child

Returns pointer to the found object or NULL if there was no suitable object

lv_task_t ***lv_indev_get_read_task**(*lv_disp_t *indev*)
Get a pointer to the indev read task to modify its parameters with `lv_task_...` functions.

Parameters **indev** -- pointer to an inout device

Returns pointer to the indev read refresher task. (NULL on error)

Groups

Typedefs

typedef uint8_t **lv_key_t**

typedef void (***lv_group_style_mod_cb_t**)(**struct** *_lv_group_t**, *lv_style_t**)

typedef void (***lv_group_focus_cb_t**)(**struct** *_lv_group_t**)

typedef struct *_lv_group_t* **lv_group_t**
Groups can be used to logically hold objects so that they can be individually focused. They are NOT for laying out objects on a screen (try `lv_cont` for that).

typedef uint8_t **lv_group_refocus_policy_t**

Enums

enum [anonymous]
Values:

enumerator LV_KEY_UP

enumerator LV_KEY_DOWN

enumerator LV_KEY_RIGHT

enumerator LV_KEY_LEFT

enumerator LV_KEY_ESC

enumerator LV_KEY_DEL

enumerator LV_KEY_BACKSPACE

enumerator LV_KEY_ENTER

enumerator LV_KEY_NEXT

enumerator LV_KEY_PREV

enumerator LV_KEY_HOME

enumerator LV_KEY_END

enum [anonymous]
Values:

enumerator LV_GROUP_REFOCUS_POLICY_NEXT

enumerator LV_GROUP_REFOCUS_POLICY_PREV

Functions

void **_lv_group_init**(void)
Init. the group module

Remark Internal function, do not call directly.

lv_group_t ***lv_group_create**(void)
Create a new object group

Returns pointer to the new object group

void **lv_group_del**(*lv_group_t *group*)
Delete a group object

Parameters **group** -- pointer to a group

void **lv_group_add_obj**(*lv_group_t *group*, *lv_obj_t *obj*)
Add an object to a group

Parameters

- **group** -- pointer to a group
- **obj** -- pointer to an object to add

void **lv_group_remove_obj**(*lv_obj_t *obj*)
Remove an object from its group

Parameters **obj** -- pointer to an object to remove

void **lv_group_remove_all_objs**(*lv_group_t *group*)
Remove all objects from a group

Parameters **group** -- pointer to a group

void **lv_group_focus_obj**(*lv_obj_t *obj*)
Focus on an object (defocus the current)

Parameters **obj** -- pointer to an object to focus on

void **lv_group_focus_next**(*lv_group_t *group*)
Focus the next object in a group (defocus the current)

Parameters **group** -- pointer to a group

void **lv_group_focus_prev**(*lv_group_t *group*)
Focus the previous object in a group (defocus the current)

Parameters **group** -- pointer to a group

void **lv_group_focus_freeze**(*lv_group_t *group*, bool *en*)
Do not let to change the focus from the current object

Parameters

- **group** -- pointer to a group
- **en** -- true: freeze, false: release freezing (normal mode)

lv_res_t **lv_group_send_data**(*lv_group_t *group*, uint32_t *c*)
Send a control character to the focuses object of a group

Parameters

- **group** -- pointer to a group
- **c** -- a character (use LV_KEY_.. to navigate)

Returns result of focused object in group.

void **lv_group_set_focus_cb**(*lv_group_t *group*, *lv_group_focus_cb_t focus_cb*)
Set a function for a group which will be called when a new object is focused

Parameters

- **group** -- pointer to a group
- **focus_cb** -- the call back function or NULL if unused

void **lv_group_set_refocus_policy**(*lv_group_t *group*, *lv_group_refocus_policy_t policy*)
Set whether the next or previous item in a group is focused if the currently focused obj is deleted.

Parameters

- **group** -- pointer to a group
- **new** -- refocus policy enum

void **lv_group_set_editing**(*lv_group_t *group*, bool *edit*)
Manually set the current mode (edit or navigate).

Parameters

- **group** -- pointer to group
- **edit** -- true: edit mode; false: navigate mode

void **lv_group_set_click_focus**(*lv_group_t *group*, bool *en*)
Set the `click_focus` attribute. If enabled then the object will be focused then it is clicked.

Parameters

- **group** -- pointer to group
- **en** -- true: enable `click_focus`

void **lv_group_set_wrap**(*lv_group_t *group*, bool *en*)
Set whether focus next/prev will allow wrapping from first->last or last->first object.

Parameters

- **group** -- pointer to group
- **en** -- true: wrapping enabled; false: wrapping disabled

lv_obj_t ***lv_group_get_focused**(**const** *lv_group_t *group*)
Get the focused object or NULL if there isn't one

Parameters **group** -- pointer to a group

Returns pointer to the focused object

lv_group_user_data_t ***lv_group_get_user_data**(*lv_group_t *group*)
Get a pointer to the group's user data

Parameters **group** -- pointer to an group

Returns pointer to the user data

lv_group_focus_cb_t **lv_group_get_focus_cb**(**const** *lv_group_t *group*)
Get the focus callback function of a group

Parameters **group** -- pointer to a group

Returns the call back function or NULL if not set

bool **lv_group_get_editing**(**const** *lv_group_t *group*)
Get the current mode (edit or navigate).

Parameters **group** -- pointer to group

Returns true: edit mode; false: navigate mode

bool **lv_group_get_click_focus**(**const** *lv_group_t *group*)
Get the `click_focus` attribute.

Parameters **group** -- pointer to group

Returns true: `click_focus` is enabled; false: disabled

bool **lv_group_get_wrap**(*lv_group_t *group*)
Get whether focus next/prev will allow wrapping from first->last or last->first object.

Parameters

- **group** -- pointer to group
- **en** -- true: wrapping enabled; false: wrapping disabled

struct _lv_group_t
#include <lv_group.h> Groups can be used to logically hold objects so that they can be individually focused. They are NOT for laying out objects on a screen (try `lv_cont` for that).

Public Members

lv_ll_t **obj_ll**
Linked list to store the objects in the group

lv_obj_t ****obj_focus**
The object in focus

lv_group_focus_cb_t **focus_cb**
A function to call when a new object is focused (optional)

lv_group_user_data_t **user_data**

uint8_t **frozen**
1: can't focus to new object

uint8_t **editing**
1: Edit mode, 0: Navigate mode

uint8_t **click_focus**
1: If an object in a group is clicked by an indev then it will be focused

uint8_t **refocus_policy**
1: Focus prev if focused on deletion. 0: Focus next if focused on deletion.

uint8_t **wrap**
1: Focus next/prev can wrap at end of list. 0: Focus next/prev stops at end of list.

4.6 Displays

Important: The basic concept of *display* in LVGL is explained in the [Porting](/porting/display) section. So before reading further, please read the [Porting](/porting/display) section first.

4.6.1 Multiple display support

In LVGL, you can have multiple displays, each with their own driver and objects. The only limitation is that every display needs to be have same color depth (as defined in `LV_COLOR_DEPTH`). If the displays are different in this regard the rendered image can be converted to the correct format in the drivers `flush_cb`.

Creating more displays is easy: just initialize more display buffers and register another driver for every display. When you create the UI, use `lv_disp_set_default(disp)` to tell the library on which display to create objects.

Why would you want multi-display support? Here are some examples:

- Have a "normal" TFT display with local UI and create "virtual" screens on VNC on demand. (You need to add your VNC driver).
- Have a large TFT display and a small monochrome display.
- Have some smaller and simple displays in a large instrument or technology.
- Have two large TFT displays: one for a customer and one for the shop assistant.

Using only one display

Using more displays can be useful, but in most cases, it's not required. Therefore, the whole concept of multi-display is completely hidden if you register only one display. By default, the lastly created (the only one) display is used as default.

`lv_scr_act()`, `lv_scr_load(scr)`, `lv_layer_top()`, `lv_layer_sys()`, `LV_HOR_RES` and `LV_VER_RES` are always applied on the lastly created (default) screen. If you pass `NULL` as `disp` parameter to display related function, usually the default display will be used. E.g. `lv_disp_trig_activity(NULL)` will trigger a user activity on the default screen. (See below in *Inactivity*).

Mirror display

To mirror the image of the display to another display, you don't need to use the multi-display support. Just transfer the buffer received in `drv.flush_cb` to another display too.

Split image

You can create a larger display from smaller ones. You can create it as below:

1. Set the resolution of the displays to the large display's resolution.
2. In `drv.flush_cb`, truncate and modify the `area` parameter for each display.
3. Send the buffer's content to each display with the truncated area.

4.6.2 Screens

Every display has each set of Screens and the object on the screens.

Be sure not to confuse displays and screens:

- **Displays** are the physical hardware drawing the pixels.
- **Screens** are the high-level root objects associated with a particular display. One display can have multiple screens associated with it, but not vice versa.

Screens can be considered the highest level containers which have no parent. The screen's size is always equal to its display and size their position is (0;0). Therefore, the screens coordinates can't be changed, i.e. `lv_obj_set_pos()`, `lv_obj_set_size()` or similar functions can't be used on screens.

A screen can be created from any object type but, the two most typical types are the *Base object* and the *Image* (to create a wallpaper).

To create a screen, use `lv_obj_t * scr = lv_<type>_create(NULL, copy)`. `copy` can be an other screen to copy it.

To load a screen, use `lv_scr_load(scr)`. To get the active screen, use `lv_scr_act()`. These functions works on the default display. If you want to to specify which display to work on, use `lv_disp_get_scr_act(disp)` and `lv_disp_load_scr(disp, scr)`. Screen can be loaded with animations too. Read more here.

Screens can be deleted with `lv_obj_del(scr)`, but ensure that you do not delete the currently loaded screen.

Transparent screens

Usually, the opacity of the screen is `LV_OPA_COVER` to provide a solid background for its children. If it's not the case (opacity $< 100\%$) the display's background color or image will be visible. See the *Display background* section for more details. If the display's background opacity is also not `LV_OPA_COVER` LVGL has no solid background to draw.

This configuration (transparent screen ans display) could be used to create for example OSD menus where a video is played to lower layer, and menu is created on an upper layer.

To handle transparent displays special (slower) color mixing algorithms needs to be used by LVGL so this feature needs to enabled with `LV_COLOR_SCREEN_TRANSP` n `lv_conf.h`. As this mode operates on the Alpha channel of the pixels `LV_COLOR_DEPTH = 32` is also required. The Alpha channel of 32-bit colors will be 0 where there are no objects and will be 255 where there are solid objects.

In summary, to enable transparent screen and displays to create OSD menu-like UIs:

- Enable `LV_COLOR_SCREEN_TRANSP` in `lv_conf.h`
- Be sure to use `LV_COLOR_DEPTH 32`
- Set the screens opacity to `LV_OPA_TRANSP` e.g. with `lv_obj_set_style_local_bg_opa(lv_scr_act(), LV_OBJMASK_PART_MAIN, LV_STATE_DEFAULT, LV_OPA_TRANSP)`
- Set the display opacity to `LV_OPA_TRANSP` with `lv_disp_set_bg_opa(NULL, LV_OPA_TRANSP);`

4.6.3 Features of displays

Inactivity

The user's inactivity is measured on each display. Every use of an *Input device* (if associated with the display) counts as an activity. To get time elapsed since the last activity, use `lv_disp_get_inactive_time(disp)`. If `NULL` is passed, the overall smallest inactivity time will be returned from all displays (**not the default display**).

You can manually trigger an activity using `lv_disp_trig_activity(disp)`. If `disp` is `NULL`, the default screen will be used (**and not all displays**).

Background

Every display has background color, a background image and background opacity properties. They become visible when the current screen is transparent or not positioned to cover the whole display.

Background color is a simple color to fill the display. It can be adjusted with `lv_disp_set_bg_color(disp, color)`;

Background image is path to file or pointer to an `lv_img_dsc_t` variable (converted image) to be used as wallpaper. It can be set with `lv_disp_set_bg_color(disp, &my_img)`; If the background image is set (not `NULL`) the background won't filled with `bg_color`.

The opacity of the background color or image can be adjusted with `lv_disp_set_bg_opa(disp, opa)`.

The `disp` parameter of these functions can be `NULL` to refer it to the default display.

4.6.4 Colors

The color module handles all color-related functions like changing color depth, creating colors from hex code, converting between color depths, mixing colors, etc.

The following variable types are defined by the color module:

- **lv_color1_t** Store monochrome color. For compatibility, it also has R, G, B fields but they are always the same value (1 byte)
- **lv_color8_t** A structure to store R (3 bit),G (3 bit),B (2 bit) components for 8-bit colors (1 byte)
- **lv_color16_t** A structure to store R (5 bit),G (6 bit),B (5 bit) components for 16-bit colors (2 byte)
- **lv_color32_t** A structure to store R (8 bit),G (8 bit), B (8 bit) components for 24-bit colors (4 byte)
- **lv_color_t** Equal to `lv_color1/8/16/24_t` according to color depth settings
- **lv_color_int_t** `uint8_t`, `uint16_t` or `uint32_t` according to color depth setting. Used to build color arrays from plain numbers.
- **lv_opa_t** A simple `uint8_t` type to describe opacity.

The `lv_color_t`, `lv_color1_t`, `lv_color8_t`, `lv_color16_t` and `lv_color32_t` types have got four fields:

- **ch.red** red channel
- **ch.green** green channel
- **ch.blue** blue channel
- **full** red + green + blue as one number

You can set the current color depth in *lv_conf.h*, by setting the `LV_COLOR_DEPTH` define to 1 (monochrome), 8, 16 or 32.

Convert color

You can convert a color from the current color depth to another. The converter functions return with a number, so you have to use the `full` field:

```
lv_color_t c;
c.red   = 0x38;
c.green = 0x70;
c.blue  = 0xCC;

lv_color1_t c1;
c1.full = lv_color_to1(c);        /*Return 1 for light colors, 0 for dark colors*/

lv_color8_t c8;
c8.full = lv_color_to8(c);        /*Give a 8 bit number with the converted color*/

lv_color16_t c16;
c16.full = lv_color_to16(c); /*Give a 16 bit number with the converted color*/

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

Swap 16 colors

You may set `LV_COLOR_16_SWAP` in *lv_conf.h* to swap the bytes of *RGB565* colors. It's useful if you send the 16-bit colors via a byte-oriented interface like SPI.

As 16-bit numbers are stored in Little Endian format (lower byte on the lower address), the interface will send the lower byte first. However, displays usually need the higher byte first. A mismatch in the byte order will result in highly distorted colors.

Create and mix colors

You can create colors with the current color depth using the LV_COLOR_MAKE macro. It takes 3 arguments (red, green, blue) as 8-bit numbers. For example to create light red color: `my_color = COLOR_MAKE(0xFF,0x80,0x80)`.

Colors can be created from HEX codes too: `my_color = lv_color_hex(0x288ACF)` or `my_color = lv_folro_hex3(0x28C)`.

Mixing two colors is possible with `mixed_color = lv_color_mix(color1, color2, ratio)`. Ration can be 0..255. 0 results fully color2, 255 result fully color1.

Colors can be created with from HSV space too using `lv_color_hsv_to_rgb(hue, saturation, value)` . `hue` should be in 0..360 range, `saturation` and `value` in 0..100 range.

Opacity

To describe opacity the `lv_opa_t` type is created as a wrapper to `uint8_t`. Some defines are also introduced:

- **LV_OPA_TRANSP** Value: 0, means the opacity makes the color completely transparent
- **LV_OPA_10** Value: 25, means the color covers only a little
- **LV_OPA_20 ... OPA_80** come logically
- **LV_OPA_90** Value: 229, means the color near completely covers
- **LV_OPA_COVER** Value: 255, means the color completely covers

You can also use the `LV_OPA_*` defines in `lv_color_mix()` as a *ratio*.

Built-in colors

This section is not available in the PDF build due to bugs with the documentation generator. Please see `lv_misc/lv_color.h`.

4.6.5 API

Display

Enums

enum lv_scr_load_anim_t

Values:

enumerator LV_SCR_LOAD_ANIM_NONE

enumerator LV_SCR_LOAD_ANIM_OVER_LEFT

enumerator LV_SCR_LOAD_ANIM_OVER_RIGHT

enumerator LV_SCR_LOAD_ANIM_OVER_TOP

enumerator LV_SCR_LOAD_ANIM_OVER_BOTTOM

enumerator LV_SCR_LOAD_ANIM_MOVE_LEFT

enumerator LV_SCR_LOAD_ANIM_MOVE_RIGHT

enumerator LV_SCR_LOAD_ANIM_MOVE_TOP

enumerator LV_SCR_LOAD_ANIM_MOVE_BOTTOM

enumerator LV_SCR_LOAD_ANIM_FADE_ON

Functions

lv_obj_t ***lv_disp_get_scr_act**(*lv_disp_t *disp*)
Return with a pointer to the active screen

Parameters **disp** -- pointer to display which active screen should be get. (NULL to use the default screen)

Returns pointer to the active screen object (loaded by 'lv_scr_load()')

lv_obj_t ***lv_disp_get_scr_prev**(*lv_disp_t *disp*)
Return with a pointer to the previous screen. Only used during screen transitions.

Parameters **disp** -- pointer to display which previous screen should be get. (NULL to use the default screen)

Returns pointer to the previous screen object or NULL if not used now

void **lv_disp_load_scr**(*lv_obj_t *scr*)
Make a screen active

Parameters **scr** -- pointer to a screen

lv_obj_t ***lv_disp_get_layer_top**(*lv_disp_t *disp*)
Return with the top layer. (Same on every screen and it is above the normal screen layer)

Parameters **disp** -- pointer to display which top layer should be get. (NULL to use the default screen)

Returns pointer to the top layer object (transparent screen sized lv_obj)

lv_obj_t ***lv_disp_get_layer_sys**(*lv_disp_t *disp*)
Return with the sys. layer. (Same on every screen and it is above the normal screen and the top layer)

Parameters **disp** -- pointer to display which sys. layer should be get. (NULL to use the default screen)

Returns pointer to the sys layer object (transparent screen sized lv_obj)

void **lv_disp_assign_screen**(*lv_disp_t *disp*, *lv_obj_t *scr*)
Assign a screen to a display.

Parameters

- **disp** -- pointer to a display where to assign the screen
- **scr** -- pointer to a screen object to assign

void **lv_disp_set_bg_color**(*lv_disp_t *disp*, lv_color_t *color*)
Set the background color of a display

Parameters

- **disp** -- pointer to a display
- **color** -- color of the background

void **lv_disp_set_bg_image**(*lv_disp_t *disp*, **const** void *img_src*)
Set the background image of a display

Parameters

- **disp** -- pointer to a display
- **img_src** -- path to file or pointer to an *lv_img_dsc_t* variable

void **lv_disp_set_bg_opa**(*lv_disp_t* *disp, lv_opa_t *opa*)
Opacity of the background

Parameters

- **disp** -- pointer to a display
- **opa** -- opacity (0..255)

void **lv_scr_load_anim**(*lv_obj_t* *scr, *lv_scr_load_anim_t anim_type*, uint32_t *time*, uint32_t *delay*, bool *auto_del*)
Switch screen with animation

Parameters

- **scr** -- pointer to the new screen to load
- **anim_type** -- type of the animation from `lv_scr_load_anim_t`. E.g. `LV_SCR_LOAD_ANIM_MOVE_LEFT`
- **time** -- time of the animation
- **delay** -- delay before the transition
- **auto_del** -- true: automatically delete the old screen

uint32_t **lv_disp_get_inactive_time**(**const** *lv_disp_t* *disp)
Get elapsed time since last user activity on a display (e.g. click)

Parameters **disp** -- pointer to an display (NULL to get the overall smallest inactivity)

Returns elapsed ticks (milliseconds) since the last activity

void **lv_disp_trig_activity**(*lv_disp_t* *disp)
Manually trigger an activity on a display

Parameters **disp** -- pointer to an display (NULL to use the default display)

void **lv_disp_clean_dcache**(*lv_disp_t* *disp)
Clean any CPU cache that is related to the display.

Parameters **disp** -- pointer to an display (NULL to use the default display)

lv_task_t ***_lv_disp_get_refr_task**(*lv_disp_t* *disp)
Get a pointer to the screen refresher task to modify its parameters with `lv_task_...` functions.

Parameters **disp** -- pointer to a display

Returns pointer to the display refresher task. (NULL on error)

static inline *lv_obj_t* ***lv_scr_act**(void)
Get the active screen of the default display

Returns pointer to the active screen

static inline *lv_obj_t* ***lv_layer_top**(void)
Get the top layer of the default display

Returns pointer to the top layer

static inline *lv_obj_t* ***lv_layer_sys**(void)
Get the active screen of the default display

Returns pointer to the sys layer

static inline void **lv_scr_load**(*lv_obj_t* *scr)

static inline lv_coord_t **lv_dpx**(lv_coord_t *n*)

Colors

Enums

enum [anonymous]
Opacity percentages.

Values:

enumerator LV_OPA_TRANSP

enumerator LV_OPA_0

enumerator LV_OPA_10

enumerator LV_OPA_20

enumerator LV_OPA_30

enumerator LV_OPA_40

enumerator LV_OPA_50

enumerator LV_OPA_60

enumerator LV_OPA_70

enumerator LV_OPA_80

enumerator LV_OPA_90

enumerator LV_OPA_100

enumerator LV_OPA_COVER

Functions

typedef LV_CONCAT3 (uint, LV_COLOR_SIZE, _t) lv_color_int_t

typedef LV_CONCAT3 (lv_color, LV_COLOR_DEPTH, _t) lv_color_t

static inline uint8_t **lv_color_to1**(lv_color_t *color*)

static inline uint8_t **lv_color_to8**(lv_color_t *color*)

static inline uint16_t **lv_color_to16**(lv_color_t *color*)

static inline uint32_t **lv_color_to32**(lv_color_t *color*)

static inline uint8_t **lv_color_brightness**(lv_color_t *color*)
Get the brightness of a color

Parameters **color** -- a color

Returns the brightness [0..255]

static inline lv_color_t **lv_color_make**(uint8_t *r*, uint8_t *g*, uint8_t *b*)

static inline lv_color_t **lv_color_hex**(uint32_t *c*)

static inline lv_color_t **lv_color_hex3**(uint32_t *c*)

lv_color_t **lv_color_lighten**(lv_color_t *c*, lv_opa_t *lvl*)

lv_color_t **lv_color_darken**(lv_color_t *c*, lv_opa_t *lvl*)

lv_color_t **lv_color_hsv_to_rgb**(uint16_t *h*, uint8_t *s*, uint8_t *v*)
Convert a HSV color to RGB

Parameters

- **h** -- hue [0..359]
- **s** -- saturation [0..100]
- **v** -- value [0..100]

Returns the given RGB color in RGB (with LV_COLOR_DEPTH depth)

lv_color_hsv_t **lv_color_rgb_to_hsv**(uint8_t *r8*, uint8_t *g8*, uint8_t *b8*)
Convert a 32-bit RGB color to HSV

Parameters

- **r8** -- 8-bit red
- **g8** -- 8-bit green
- **b8** -- 8-bit blue

Returns the given RGB color in HSV

lv_color_hsv_t **lv_color_to_hsv**(lv_color_t *color*)
Convert a color to HSV

Parameters **color** -- color

Returns the given color in HSV

union lv_color1_t

Public Members

uint8_t **full**

uint8_t **blue**

uint8_t **green**

uint8_t **red**

union *lv_color1_t*::[**anonymous**] **ch**

uint8_t **full**

union lv_color8_t

Public Members

uint8_t **blue**

uint8_t **green**

uint8_t **red**

struct *lv_color8_t*::[**anonymous**] **ch**

uint8_t **full**

union lv_color16_t

Public Members

uint16_t **blue**

uint16_t **green**

uint16_t **red**

uint16_t **green_h**

uint16_t **green_l**

struct *lv_color16_t*::[**anonymous**] **ch**

uint16_t **full**

union lv_color32_t

Public Members

uint8_t **blue**

uint8_t **green**

uint8_t **red**

uint8_t **alpha**

struct *lv_color32_t*::[**anonymous**] **ch**

uint32_t **full**

struct lv_color_hsv_t

Public Members

uint16_t **h**

uint8_t **s**

uint8_t **v**

4.7 Fonts

In LVGL fonts are collections of bitmaps and other information required to render the images of the letters (glyph). A font is stored in a `lv_font_t` variable and can be set in style's *text_font* field. For example:

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

The fonts have a **bpp (bits per pixel)** property. It shows how many bits are used to describe a pixel in the font. The value stored for a pixel determines the pixel's opacity. This way, with higher *bpp*, the edges of the letter can be smoother. The possible *bpp* values are 1, 2, 4 and 8 (higher value means better quality).

The *bpp* also affects the required memory size to store the font. For example, *bpp = 4* makes the font nearly 4 times greater compared to *bpp = 1*.

4.7.1 Unicode support

LVGL supports **UTF-8** encoded Unicode characters. Your editor needs to be configureed to save your code/text as UTF-8 (usually this the default) and be sure that, `LV_TXT_ENC` is set to `LV_TXT_ENC_UTF8` in *lv_conf.h*. (This is the default value)

To test it try

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

If all works well, a ✓ character should be displayed.

4.7.2 Built-in fonts

There are several built-in fonts in different sizes, which can be enabled in `lv_conf.h` by *LV_FONT_...* defines.

Normal fonts

Containing all the ASCII characters, the degree symbol (U+00B0), the bullet symbol (U+2022) and the build in symbols (see below).

- `LV_FONT_MONTSERRAT_12` 12 px font
- `LV_FONT_MONTSERRAT_14` 14 px font
- `LV_FONT_MONTSERRAT_16` 16 px font
- `LV_FONT_MONTSERRAT_18` 18 px font
- `LV_FONT_MONTSERRAT_20` 20 px font
- `LV_FONT_MONTSERRAT_22` 22 px font
- `LV_FONT_MONTSERRAT_24` 24 px font
- `LV_FONT_MONTSERRAT_26` 26 px font
- `LV_FONT_MONTSERRAT_28` 28 px font
- `LV_FONT_MONTSERRAT_30` 30 px font
- `LV_FONT_MONTSERRAT_32` 32 px font
- `LV_FONT_MONTSERRAT_34` 34 px font
- `LV_FONT_MONTSERRAT_36` 36 px font
- `LV_FONT_MONTSERRAT_38` 38 px font
- `LV_FONT_MONTSERRAT_40` 40 px font
- `LV_FONT_MONTSERRAT_42` 42 px font
- `LV_FONT_MONTSERRAT_44` 44 px font
- `LV_FONT_MONTSERRAT_46` 46 px font
- `LV_FONT_MONTSERRAT_48` 48 px font

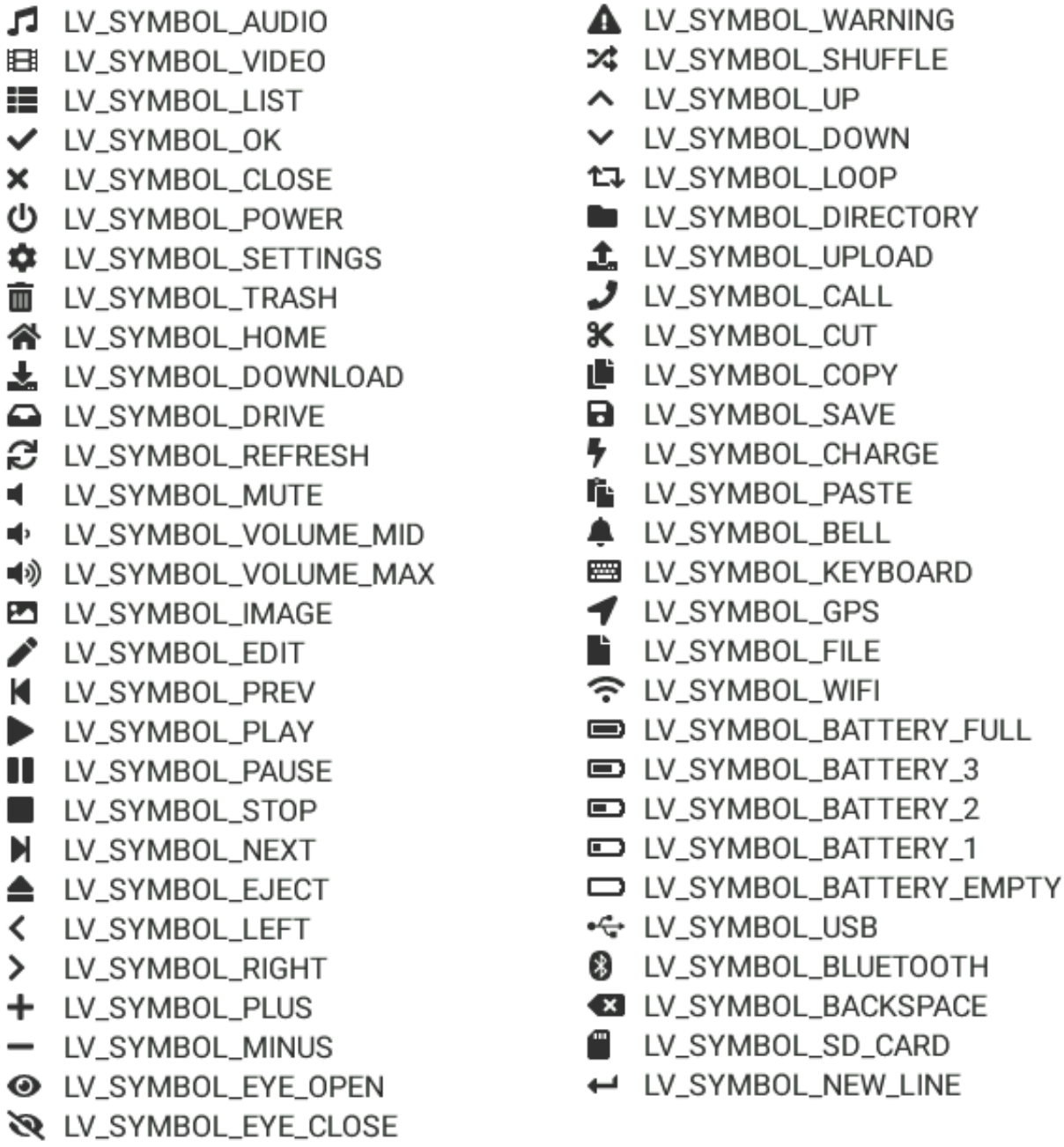
Special fonts

- `LV_FONT_MONTSERRAT_12_SUBPX` Same as normal 12 px font but with *subpixel rendering*
- `LV_FONT_MONTSERRAT_28_COMPRESSED` Same as normal 28 px font but *compressed font* with 3 bpp
- `LV_FONT_DEJAVU_16_PERSIAN_HEBREW` 16 px font with normal range + Hebrew, Arabic, Perisan letters and all their forms
- `LV_FONT_SIMSUN_16_CJK`16 px font with normal range + 1000 most common CJK radicals
- `LV_FONT_UNSCII_8` 8 px pixel perfect font with only ASCII characters
- `LV_FONT_UNSCII_16` 16 px pixel perfect font with only ASCII characters

The built-in fonts are **global variables** with names like `lv_font_montserrat_16` for 16 px hight font. To use them in a style, just add a pointer to a font variable like shown above.

The built-in fonts have *bpp = 4*, contains the ASCII characters and uses the Montserrat font.

In addition to the ASCII range, the following symbols are also added to the built-in fonts from the FontAwesome font.



The symbols can be used as:

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

Or with together with strings:

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

Or more symbols together:

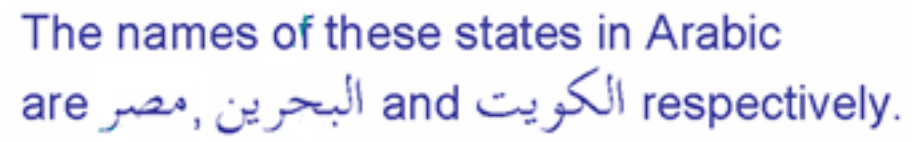
```
lv_label_set_text(my_label, LV_SYMBOL_OK LV_SYMBOL_WIFI LV_SYMBOL_PLAY);
```

4.7.3 Special features

Bidirectional support

Most of the languages use Left-to-Right (LTR for short) writing direction, however some languages (such as Hebrew, Persian or Arabic) uses Right-to-Left (RTL for short) direction.

LVGL not only supports RTL texts but supports mixed (a.k.a. bidirectional, BiDi) text rendering too. Some examples:



The BiDi support can be enabled by `LV_USE_BIDI` in *lv_conf.h*

All texts have a base direction (LTR or RTL) which determines some rendering rules and the default alignment of the text (Left or Right). However, in LVGL, base direction is applied not only for labels. It's a general property which can be set for every object. If unset then it will be inherited from the parent. So it's enough to set the base direction of the screen and every object will inherit it.

The default base direction of screen can be set by `LV_BIDI_BASE_DIR_DEF` in *lv_conf.h* and other objects inherit the base direction from their parent.

To set an object's base direction use `lv_obj_set_base_dir(obj, base_dir)`. The possible base direction are:

- `LV_BIDI_DIR_LTR`: Left to Right base direction
- `LV_BIDI_DIR_RTL`: Right to Left base direction
- `LV_BIDI_DIR_AUTO`: Auto detect base direction
- `LV_BIDI_DIR_INHERIT`: Inherit the base direction from the parent (default for non-screen objects)

This list summarizes the effect of RTL base direction on objects:

- Create objects by default on the right
- `lv_tabview`: displays tabs from right to left
- `lv_checkbox`: Show the box on the right
- `lv_btnmatrix`: Show buttons from right to left
- `lv_list`: Show the icon on the right
- `lv_dropdown`: Align the options to the right
- The texts in `lv_table`, `lv_btnmatrix`, `lv_keyboard`, `lv_tabview`, `lv_dropdown`, `lv_roller` are "BiDi processed" to be displayed correctly

Arabic and Persian support

There are some special rules to display Arabic and Persian characters: the *form* of the character depends on their position in the text. A different form of the same letter needs to be used if it isolated, start, middle or end position. Besides these some conjunction rules also should be taken into account.

LVGL supports to apply these rules if `LV_USE_ARABIC_PERSIAN_CHARS` is enabled.

However, there some limitations:

- Only displaying texts is supported (e.g. on labels), text inputs (e.g. text area) doesn't support this feature
- Static text (i.e. const) are not processed. E.g. texts set by `lv_label_set_text()` will "Arabic processed" but `lv_lable_set_text_static()` won't.
- Text get functions (e.g. `lv_label_get_text()`) will return the processed text.

Subpixel rendering

Subpixel rendering means to triple the horizontal resolution by rendering on Red, Green and Blue channel instead of pixel level. It takes advantage of the position of physical color channels of each pixel. It results in higher quality letter anti-aliasing. Lear more here.

Subpixel rendering requires to generate the fonts with special settings:

- In the online converter tick the `Subpixel` box
- In the command line tool use `--lcd` flag. Note that the generated font needs about 3 times more memory.

Subpixel rendering works only if the color channels of the pixels have a horizontal layout. That is the R, G, B channels are next each other and not above each other. The order of color channels also needs to match with the library settings. By default the LVGL assumes `RGB` order, however it can be swapped by setting `LV_SUBPX_BGR 1` in *lv_conf.h*.

Compress fonts

The bitmaps of the fonts can be compressed by

- ticking the `Compressed` check box in the online converter
- not passing `--no-compress` flag to the offline converter (applies compression by default)

The compression is more effective with larger fonts and higher bpp. However, it's about 30% slower to render the compressed fonts. Therefore it's recommended to compress only the largest fonts of user interface, because

- they need the most memory
- they can be compressed better
- and probably they are used less frequently then the medium sized fonts. (so performance cost is smaller)

4.7.4 Add new font

There are several ways to add a new font to your project:

1. The simplest method is to use the Online font converter. Just set the parameters, click the *Convert* button, copy the font to your project and use it. **Be sure to carefully read the steps provided on that site or you will get an error while converting.**
2. Use the Offline font converter. (Requires Node.js to be installed)
3. If you want to create something like the built-in fonts (Roboto font and symbols) but in different size and/or ranges, you can use the `built_in_font_gen.py` script in `lvgl/scripts/built_in_font` folder. (It requires Python and `lv_font_conv` to be installed)

To declare the font in a file, use `LV_FONT_DECLARE(my_font_name)`.

To make the fonts globally available (like the builtin fonts), add them to `LV_FONT_CUSTOM_DECLARE` in *lv_conf.h*.

4.7.5 Add new symbols

The built-in symbols are created from FontAwesome font.

1. Search symbol on https://fontawesome.com. For example the USB symbol. Copy it's Unicode ID which is `0xf287` in this case.
2. Open the Online font converter. Add Add FontAwesome.woff. .
3. Set the parameters such as Name, Size, BPP. You'll use this name to declare and use the font in your code.
4. Add the Unicode ID of the symbol to the range field. E.g. `0xf287` for the USB symbol. More symbols can be enumerated with `,`.
5. Convert the font and copy it to your project. Make sure to compile the .c file of your font.
6. Declare the font using `extern lv_font_t my_font_name;` or simply `LV_FONT_DECLARE(my_font_name);`.

Using the symbol

1. Convert the Unicode value to UTF8. You can do it e.g on this site. For `0xf287` the *Hex UTF-8 bytes* are `EF 8A 87`.
2. Create a `define` from the UTF8 values: `#define MY_USB_SYMBOL "\xEF\x8A\x87"`
3. Create a label and set the text. Eg. `lv_label_set_text(label, MY_USB_SYMBOL)`

Note - `lv_label_set_text(label, MY_USB_SYMBOL)` searches for this symbol in the font defined in `style.text.font` properties. To use the symbol you may need to change it. Eg `style.text.font = my_font_name`

4.7.6 Load font in run-time

`lv_font_load` can be used to load a font from a file. The font to load needs to have a special binary format. (Not TTF or WOFF). Use lv_font_conv with `--format bin` option to generate an LVGL compatible font file.

Note that to load a font *LVGL's filesystem* needs to be enabled and a driver needs to be added.

Example

```
lv_font_t * my_font;
my_font = lv_font_load(X/path/to/my_font.bin);

/*Use the font*/

/*Free the font if not required anymore*/
lv_font_free(my_font);
```

4.7.7 Add a new font engine

LVGL's font interface is designed to be very flexible. You don't need to use LVGL's internal font engine but, you can add your own. For example, use FreeType to real-time render glyphs from TTF fonts or use an external flash to store the font's bitmap and read them when the library needs them.

A ready to use FreeType can be found in lv_freetype repository.

To do this a custom `lv_font_t` variable needs to be created:

```
/*Describe the properties of a font*/
lv_font_t my_font;
my_font.get_glyph_dsc = my_get_glyph_dsc_cb;        /*Set a callback to get info␣
↪about gylphs*/
my_font.get_glyph_bitmap = my_get_glyph_bitmap_cb;  /*Set a callback to get bitmap of␣
↪a glyp*/
my_font.line_height = height;                       /*The real line height where any␣
↪text fits*/
my_font.base_line = base_line;                      /*Base line measured from the top␣
↪of line_height*/
my_font.dsc = something_required;                   /*Store any implementation␣
↪specific data here*/
my_font.user_data = user_data;                      /*Optionally some extra user␣
↪data*/

...

/* Get info about glyph of `unicode_letter` in `font` font.
 * Store the result in `dsc_out`.
 * The next letter (`unicode_letter_next`) might be used to calculate the width␣
↪required by this glyph (kerning)
 */
bool my_get_glyph_dsc_cb(const lv_font_t * font, lv_font_glyph_dsc_t * dsc_out,␣
↪uint32_t unicode_letter, uint32_t unicode_letter_next)
{
    /*Your code here*/

    /* Store the result.
     * For example ...
```

(continues on next page)

(continued from previous page)

```
     */
    dsc_out->adv_w = 12;        /*Horizontal space required by the glyph in [px]*/
    dsc_out->box_h = 8;         /*Height of the bitmap in [px]*/
    dsc_out->box_w = 6;         /*Width of the bitmap in [px]*/
    dsc_out->ofs_x = 0;         /*X offset of the bitmap in [pf]*/
    dsc_out->ofs_y = 3;         /*Y offset of the bitmap measured from the as line*/
    dsc_out->bpp   = 2;         /*Bits per pixel: 1/2/4/8*/

    return true;                /*true: glyph found; false: glyph was not found*/
}


/* Get the bitmap of `unicode_letter` from `font`. */
const uint8_t * my_get_glyph_bitmap_cb(const lv_font_t * font, uint32_t unicode_
↪letter)
{
    /* Your code here */

    /* The bitmap should be a continuous bitstream where
     * each pixel is represented by `bpp` bits */

    return bitmap;    /*Or NULL if not found*/
}
```

4.8 Images

An image can be a file or variable which stores the bitmap itself and some metadata.

4.8.1 Store images

You can store images in two places

- as a variable in the internal memory (RAM or ROM)
- as a file

Variables

The images stored internally in a variable is composed mainly of an `lv_img_dsc_t` structure with the following fields:

- **header**
  - *cf* Color format. See *below*
  - *w* width in pixels (<= 2048)
  - *h* height in pixels (<= 2048)
  - *always zero* 3 bits which need to be always zero
  - *reserved* reserved for future use
- **data** pointer to an array where the image itself is stored
- **data_size** length of `data` in bytes

These are usually stored within a project as C files. They are linked into the resulting executable like any other constant data.

Files

To deal with files you need to add a *Drive* to LVGL. In short, a *Drive* is a collection of functions (*open*, *read*, *close*, etc.) registered in LVGL to make file operations. You can add an interface to a standard file system (FAT32 on SD card) or you create your simple file system to read data from an SPI Flash memory. In every case, a *Drive* is just an abstraction to read and/or write data to a memory. See the *File system* section to learn more.

Images stored as files are not linked into the resulting executable, and must be read to RAM before being drawn. As a result, they are not as resource-friendly as variable images. However, they are easier to replace without needing to recompile the main program.

4.8.2 Color formats

Various built-in color formats are supported:

- **LV_IMG_CF_TRUE_COLOR** Simply stores the RGB colors (in whatever color depth LVGL is configured for).
- **LV_IMG_CF_TRUE_COLOR_ALPHA** Like `LV_IMG_CF_TRUE_COLOR` but it also adds an alpha (transparency) byte for every pixel.
- **LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED** Like `LV_IMG_CF_TRUE_COLOR` but if a pixel has `LV_COLOR_TRANSP` (set in *lv_conf.h*) color the pixel will be transparent.
- **LV_IMG_CF_INDEXED_1/2/4/8BIT** Uses a palette with 2, 4, 16 or 256 colors and stores each pixel in 1, 2, 4 or 8 bits.
- **LV_IMG_CF_ALPHA_1/2/4/8BIT Only stores the Alpha value on 1, 2, 4 or 8 bits.** The pixels take the color of `style.image.color` and the set opacity. The source image has to be an alpha channel. This is ideal for bitmaps similar to fonts (where the whole image is one color but you'd like to be able to change it).

The bytes of the `LV_IMG_CF_TRUE_COLOR` images are stored in the following order.

For 32-bit color depth:

- Byte 0: Blue
- Byte 1: Green
- Byte 2: Red
- Byte 3: Alpha

For 16-bit color depth:

- Byte 0: Green 3 lower bit, Blue 5 bit
- Byte 1: Red 5 bit, Green 3 higher bit
- Byte 2: Alpha byte (only with LV_IMG_CF_TRUE_COLOR_ALPHA)

For 8-bit color depth:

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

You can store images in a *Raw* format to indicate that, it's not a built-in color format and an external *Image decoder* needs to be used to decode the image.

- **LV_IMG_CF_RAW** Indicates a basic raw image (e.g. a PNG or JPG image).
- **LV_IMG_CF_RAW_ALPHA** Indicates that the image has alpha and an alpha byte is added for every pixel.
- **LV_IMG_CF_RAW_CHROME_KEYED** Indicates that the image is chrome keyed as described in `LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED` above.

4.8.3 Add and use images

You can add images to LVGL in two ways:

- using the online converter
- manually create images

Online converter

The online Image converter is available here: https://lvgl.io/tools/imageconverter

Adding an image to LVGL via online converter is easy.

1. You need to select a *BMP*, *PNG* or *JPG* image first.
2. Give the image a name that will be used within LVGL.
3. Select the *Color format*.
4. Select the type of image you want. Choosing a binary will generate a `.bin` file that must be stored separately and read using the *file support*. Choosing a variable will generate a standard C file that can be linked into your project.
5. Hit the *Convert* button. Once the conversion is finished, your browser will automatically download the resulting file.

In the converter C arrays (variables), the bitmaps for all the color depths (1, 8, 16 or 32) are included in the C file, but only the color depth that matches `LV_COLOR_DEPTH` in *lv_conf.h* will actually be linked into the resulting executable.

In case of binary files, you need to specify the color format you want:

- RGB332 for 8-bit color depth
- RGB565 for 16-bit color depth
- RGB565 Swap for 16-bit color depth (two bytes are swapped)
- RGB888 for 32-bit color depth

Manually create an image

If you are generating an image at run-time, you can craft an image variable to display it using LVGL. For example:

```
uint8_t my_img_data[] = {0x00, 0x01, 0x02, ...};

static lv_img_dsc_t my_img_dsc = {
    .header.always_zero = 0,
    .header.w = 80,
    .header.h = 60,
    .data_size = 80 * 60 * LV_COLOR_DEPTH / 8,
    .header.cf = LV_IMG_CF_TRUE_COLOR,          /*Set the color format*/
    .data = my_img_data,
};
```

If the color format is `LV_IMG_CF_TRUE_COLOR_ALPHA` you can set `data_size` like `80 * 60 * LV_IMG_PX_SIZE_ALPHA_BYTE`.

Another (possibly simpler) option to create and display an image at run-time is to use the *Canvas* object.

Use images

The simplest way to use an image in LVGL is to display it with an *lv_img* object:

```
lv_obj_t * icon = lv_img_create(lv_scr_act(), NULL);

/*From variable*/
lv_img_set_src(icon, &my_icon_dsc);

/*From file*/
lv_img_set_src(icon, "S:my_icon.bin");
```

If the image was converted with the online converter, you should use `LV_IMG_DECLARE(my_icon_dsc)` to declare the image in the file where you want to use it.

4.8.4 Image decoder

As you can see in the *Color formats* section, LVGL supports several built-in image formats. In many cases, these will be all you need. LVGL doesn't directly support, however, generic image formats like PNG or JPG.

To handle non-built-in image formats, you need to use external libraries and attach them to LVGL via the *Image decoder* interface.

The image decoder consists of 4 callbacks:

- **info** get some basic info about the image (width, height and color format).
- **open** open the image: either store the decoded image or set it to `NULL` to indicate the image can be read line-by-line.
- **read** if *open* didn't fully open the image this function should give some decoded data (max 1 line) from a given position.
- **close** close the opened image, free the allocated resources.

You can add any number of image decoders. When an image needs to be drawn, the library will try all the registered image decoder until finding one which can open the image, i.e. knowing that format.

The `LV_IMG_CF_TRUE_COLOR_...`, `LV_IMG_INDEXED_...` and `LV_IMG_ALPHA_...` formats (essentially, all non-`RAW` formats) are understood by the built-in decoder.

Custom image formats

The easiest way to create a custom image is to use the online image converter and set `Raw`, `Raw with alpha` or `Raw with chrome keyed` format. It will just take every byte of the binary file you uploaded and write it as the image "bitmap". You then need to attach an image decoder that will parse that bitmap and generate the real, renderable bitmap.

`header.cf` will be `LV_IMG_CF_RAW`, `LV_IMG_CF_RAW_ALPHA` or `LV_IMG_CF_RAW_CHROME_KEYED` accordingly. You should choose the correct format according to your needs: fully opaque image, use alpha channel or use chroma keying.

After decoding, the *raw* formats are considered *True color* by the library. In other words, the image decoder must decode the *Raw* images to *True color* according to the format described in [#color-formats](Color formats) section.

If you want to create a custom image, you should use `LV_IMG_CF_USER_ENCODED_0..7` color formats. However, the library can draw the images only in *True color* format (or *Raw* but finally it's supposed to be in *True color* format). So the `LV_IMG_CF_USER_ENCODED_...` formats are not known by the library, therefore, they should be decoded to one of the known formats from [#color-formats](Color formats) section. It's possible to decode the image to a non-true color format first, for example, `LV_IMG_INDEXED_4BITS`, and then call the built-in decoder functions to convert it to *True color*.

With *User encoded* formats, the color format in the open function (`dsc->header.cf`) should be changed according to the new format.

Register an image decoder

Here's an example of getting LVGL to work with PNG images.

First, you need to create a new image decoder and set some functions to open/close the PNG files. It should looks like this:

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


/**
 * Get info about a PNG image
 * @param decoder pointer to the decoder where this function belongs
 * @param src can be file name or pointer to a C array
 * @param header store the info here
 * @return LV_RES_OK: no error; LV_RES_INV: can't get the info
 */
static lv_res_t decoder_info(lv_img_decoder_t * decoder, const void * src, lv_img_
↪header_t * header)
{
  /*Check whether the type `src` is known by the decoder*/
  if(is_png(src) == false) return LV_RES_INV;

  /* Read the PNG header and find `width` and `height` */
```

(continues on next page)

(continued from previous page)

```
  ...

  header->cf = LV_IMG_CF_RAW_ALPHA;
  header->w = width;
  header->h = height;
}

/**
 * Open a PNG image and return the decided image
 * @param decoder pointer to the decoder where this function belongs
 * @param dsc pointer to a descriptor which describes this decoding session
 * @return LV_RES_OK: no error; LV_RES_INV: can't get the info
 */
static lv_res_t decoder_open(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
{

  /*Check whether the type `src` is known by the decoder*/
  if(is_png(src) == false) return LV_RES_INV;

  /*Decode and store the image. If `dsc->img_data` is `NULL`, the `read_line`␣
↪function will be called to get the image data line-by-line*/
  dsc->img_data = my_png_decoder(src);

  /*Change the color format if required. For PNG usually 'Raw' is fine*/
  dsc->header.cf = LV_IMG_CF_...

  /*Call a built in decoder function if required. It's not required if`my_png_
↪decoder` opened the image in true color format.*/
  lv_res_t res = lv_img_decoder_built_in_open(decoder, dsc);

  return res;
}

/**
 * Decode `len` pixels starting from the given `x`, `y` coordinates and store them in␣
↪`buf`.
 * Required only if the "open" function can't open the whole decoded pixel array.␣
↪(dsc->img_data == NULL)
 * @param decoder pointer to the decoder the function associated with
 * @param dsc pointer to decoder descriptor
 * @param x start x coordinate
 * @param y start y coordinate
 * @param len number of pixels to decode
 * @param buf a buffer to store the decoded pixels
 * @return LV_RES_OK: ok; LV_RES_INV: failed
 */
lv_res_t decoder_built_in_read_line(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t␣
↪* dsc, lv_coord_t x,
                                                    lv_coord_t y, lv_coord_t len, uint8_
↪t * buf)
{
   /*With PNG it's usually not required*/

   /*Copy `len` pixels from `x` and `y` coordinates in True color format to `buf` */

}
```

(continues on next page)

(continued from previous page)

```
/**
 * Free the allocated resources
 * @param decoder pointer to the decoder where this function belongs
 * @param dsc pointer to a descriptor which describes this decoding session
 */
static void decoder_close(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
{
  /*Free all allocated data*/

  /*Call the built-in close function if the built-in open/read_line was used*/
  lv_img_decoder_built_in_close(decoder, dsc);

}
```

So in summary:

- In `decoder_info`, you should collect some basic information about the image and store it in `header`.
- In `decoder_open`, you should try to open the image source pointed by `dsc->src`. Its type is already in `dsc->src_type == LV_IMG_SRC_FILE/VARIABLE`. If this format/type is not supported by the decoder, return `LV_RES_INV`. However, if you can open the image, a pointer to the decoded *True color* image should be set in `dsc->img_data`. If the format is known but, you don't want to decode while image (e.g. no memory for it) set `dsc->img_data = NULL` to call `read_line` to get the pixels.
- In `decoder_close` you should free all the allocated resources.
- `decoder_read` is optional. Decoding the whole image requires extra memory and some computational overhead. However, if can decode one line of the image without decoding the whole image, you can save memory and time. To indicate that, the *line read* function should be used, set `dsc->img_data = NULL` in the open function.

Manually use an image decoder

LVGL will use the registered image decoder automatically if you try and draw a raw image (i.e. using the `lv_img` object) but you can use them manually too. Create a `lv_img_decoder_dsc_t` variable to describe the decoding session and call `lv_img_decoder_open()`.

```
lv_res_t res;
lv_img_decoder_dsc_t dsc;
res = lv_img_decoder_open(&dsc, &my_img_dsc, LV_COLOR_WHITE);

if(res == LV_RES_OK) {
  /*Do something with `dsc->img_data`*/
  lv_img_decoder_close(&dsc);
}
```

4.8.5 Image caching

Sometimes it takes a lot of time to open an image. Continuously decoding a PNG image or loading images from a slow external memory would be inefficient and detrimental to the user experience.

Therefore, LVGL caches a given number of images. Caching means some images will be left open, hence LVGL can quickly access them from `dsc->img_data` instead of needing to decode them again.

Of course, caching images is resource-intensive as it uses more RAM (to store the decoded image). LVGL tries to optimize the process as much as possible (see below), but you will still need to evaluate if this would be beneficial for your platform or not. If you have a deeply embedded target which decodes small images from a relatively fast storage medium, image caching may not be worth it.

Cache size

The number of cache entries can be defined in `LV_IMG_CACHE_DEF_SIZE` in *lv_conf.h*. The default value is 1 so only the most recently used image will be left open.

The size of the cache can be changed at run-time with `lv_img_cache_set_size(entry_num)`.

Value of images

When you use more images than cache entries, LVGL can't cache all of the images. Instead, the library will close one of the cached images (to free space).

To decide which image to close, LVGL uses a measurement it previously made of how long it took to open the image. Cache entries that hold slower-to-open images are considered more valuable and are kept in the cache as long as possible.

If you want or need to override LVGL's measurement, you can manually set the *time to open* value in the decoder open function in `dsc->time_to_open = time_ms` to give a higher or lower value. (Leave it unchanged to let LVGL set it.)

Every cache entry has a *"life"* value. Every time an image opening happens through the cache, the *life* of all entries are decreased to make them older. When a cached image is used, its *life* is increased by the *time to open* value to make it more alive.

If there is no more space in the cache, always the entry with the smallest life will be closed.

Memory usage

Note that, the cached image might continuously consume memory. For example, if 3 PNG images are cached, they will consume memory while they are opened.

Therefore, it's the user's responsibility to be sure there is enough RAM to cache, even the largest images at the same time.

Clean the cache

Let's say you have loaded a PNG image into a `lv_img_dsc_t my_png` variable and use it in an `lv_img` object. If the image is already cached and you then change the underlying PNG file, you need to notify LVGL to cache the image again. Otherwise, there is no easy way of detecting that the underlying file changed and LVGL will still draw the old image.

To do this, use `lv_img_cache_invalidate_src(&my_png)`. If `NULL` is passed as a parameter, the whole cache will be cleaned.

4.8.6 API

Image decoder

Typedefs

typedef uint8_t **lv_img_src_t**

typedef lv_res_t (***lv_img_decoder_info_f_t**)(**struct** *_lv_img_decoder* *decoder, **const** void *src, *lv_img_header_t* *header)

Get info from an image and store in the `header`

Parameters

- **src** -- the image source. Can be a pointer to a C array or a file name (Use `lv_img_src_get_type` to determine the type)
- **header** -- store the info here

Returns LV_RES_OK: info written correctly; LV_RES_INV: failed

typedef lv_res_t (***lv_img_decoder_open_f_t**)(**struct** *_lv_img_decoder* *decoder, **struct** *_lv_img_decoder_dsc* *dsc)

Open an image for decoding. Prepare it as it is required to read it later

Parameters

- **decoder** -- pointer to the decoder the function associated with
- **dsc** -- pointer to decoder descriptor. `src`, `color` are already initialized in it.

typedef lv_res_t (***lv_img_decoder_read_line_f_t**)(**struct** *_lv_img_decoder* *decoder, **struct** *_lv_img_decoder_dsc* *dsc, lv_coord_t x, lv_coord_t y, lv_coord_t len, uint8_t *buf)

Decode `len` pixels starting from the given `x`, `y` coordinates and store them in `buf`. Required only if the "open" function can't return with the whole decoded pixel array.

Parameters

- **decoder** -- pointer to the decoder the function associated with
- **dsc** -- pointer to decoder descriptor
- **x** -- start x coordinate
- **y** -- start y coordinate
- **len** -- number of pixels to decode
- **buf** -- a buffer to store the decoded pixels

Returns LV_RES_OK: ok; LV_RES_INV: failed

typedef void (***lv_img_decoder_close_f_t**)(**struct** *_lv_img_decoder* *decoder, **struct** *_lv_img_decoder_dsc* *dsc)

Close the pending decoding. Free resources etc.

Parameters

- **decoder** -- pointer to the decoder the function associated with
- **dsc** -- pointer to decoder descriptor

typedef struct *_lv_img_decoder* **lv_img_decoder_t**

typedef struct *_lv_img_decoder_dsc* **lv_img_decoder_dsc_t**

Describe an image decoding session. Stores data about the decoding

Enums

enum [anonymous]

Source of image.

Values:

enumerator LV_IMG_SRC_VARIABLE

enumerator LV_IMG_SRC_FILE

Binary/C variable

enumerator LV_IMG_SRC_SYMBOL

File in filesystem

enumerator LV_IMG_SRC_UNKNOWN

Symbol (lv_symbol_def.h)

Functions

void **_lv_img_decoder_init**(void)

Initialize the image decoder module

lv_res_t **lv_img_decoder_get_info**(**const** char *src, *lv_img_header_t* *header)

Get information about an image. Try the created image decoder one by one. Once one is able to get info that info will be used.

Parameters

- **src** -- the image source. Can be 1) File name: E.g. "S:folder/img1.png" (The drivers needs to registered via `lv_fs_add_drv()`) 2) Variable: Pointer to an *`lv_img_dsc_t`* variable 3) Symbol: E.g. `LV_SYMBOL_OK`
- **header** -- the image info will be stored here

Returns LV_RES_OK: success; LV_RES_INV: wasn't able to get info about the image

lv_res_t **lv_img_decoder_open**(*lv_img_decoder_dsc_t* *dsc, **const** void *src, lv_color_t color)

Open an image. Try the created image decoder one by one. Once one is able to open the image that decoder is save in `dsc`

Parameters

- **dsc** -- describe a decoding session. Simply a pointer to an `lv_img_decoder_dsc_t` variable.

- **src** -- the image source. Can be 1) File name: E.g. "S:folder/img1.png" (The drivers needs to registered via `lv_fs_add_drv()`) 2) Variable: Pointer to an *`lv_img_dsc_t`* variable 3) Symbol: E.g. `LV_SYMBOL_OK`
- **color** -- The color of the image with `LV_IMG_CF_ALPHA_...`

Returns LV_RES_OK: opened the image. `dsc->img_data` and `dsc->header` are set. LV_RES_INV: none of the registered image decoders were able to open the image.

lv_res_t **lv_img_decoder_read_line**(*lv_img_decoder_dsc_t *dsc*, lv_coord_t *x*, lv_coord_t *y*, lv_coord_t *len*, uint8_t *buf*)

Read a line from an opened image

Parameters

- **dsc** -- pointer to `lv_img_decoder_dsc_t` used in `lv_img_decoder_open`
- **x** -- start X coordinate (from left)
- **y** -- start Y coordinate (from top)
- **len** -- number of pixels to read
- **buf** -- store the data here

Returns LV_RES_OK: success; LV_RES_INV: an error occurred

void **lv_img_decoder_close**(*lv_img_decoder_dsc_t *dsc*)

Close a decoding session

Parameters **dsc** -- pointer to `lv_img_decoder_dsc_t` used in `lv_img_decoder_open`

lv_img_decoder_t ***lv_img_decoder_create**(void)

Create a new image decoder

Returns pointer to the new image decoder

void **lv_img_decoder_delete**(*lv_img_decoder_t *decoder*)

Delete an image decoder

Parameters **decoder** -- pointer to an image decoder

void **lv_img_decoder_set_info_cb**(*lv_img_decoder_t *decoder*, *lv_img_decoder_info_f_t info_cb*)

Set a callback to get information about the image

Parameters

- **decoder** -- pointer to an image decoder
- **info_cb** -- a function to collect info about an image (fill an `lv_img_header_t` struct)

void **lv_img_decoder_set_open_cb**(*lv_img_decoder_t *decoder*, *lv_img_decoder_open_f_t open_cb*)

Set a callback to open an image

Parameters

- **decoder** -- pointer to an image decoder
- **open_cb** -- a function to open an image

void **lv_img_decoder_set_read_line_cb**(*lv_img_decoder_t *decoder*, *lv_img_decoder_read_line_f_t read_line_cb*)

Set a callback to a decoded line of an image

Parameters

- **decoder** -- pointer to an image decoder
- **read_line_cb** -- a function to read a line of an image

void **lv_img_decoder_set_close_cb**(*lv_img_decoder_t *decoder*, *lv_img_decoder_close_f_t close_cb*)

Set a callback to close a decoding session. E.g. close files and free other resources.

Parameters

- **decoder** -- pointer to an image decoder
- **close_cb** -- a function to close a decoding session

lv_res_t **lv_img_decoder_built_in_info**(*lv_img_decoder_t *decoder*, **const** void *src, *lv_img_header_t *header*)

Get info about a built-in image

Parameters

- **decoder** -- the decoder where this function belongs
- **src** -- the image source: pointer to an *lv_img_dsc_t* variable, a file path or a symbol
- **header** -- store the image data here

Returns LV_RES_OK: the info is successfully stored in `header`; LV_RES_INV: unknown format or other error.

lv_res_t **lv_img_decoder_built_in_open**(*lv_img_decoder_t *decoder*, *lv_img_decoder_dsc_t *dsc*)

Open a built in image

Parameters

- **decoder** -- the decoder where this function belongs
- **dsc** -- pointer to decoder descriptor. `src`, `style` are already initialized in it.

Returns LV_RES_OK: the info is successfully stored in `header`; LV_RES_INV: unknown format or other error.

lv_res_t **lv_img_decoder_built_in_read_line**(*lv_img_decoder_t *decoder*, *lv_img_decoder_dsc_t *dsc*, lv_coord_t *x*, lv_coord_t *y*, lv_coord_t *len*, uint8_t *buf*)

Decode `len` pixels starting from the given `x`, `y` coordinates and store them in `buf`. Required only if the "open" function can't return with the whole decoded pixel array.

Parameters

- **decoder** -- pointer to the decoder the function associated with
- **dsc** -- pointer to decoder descriptor
- **x** -- start x coordinate
- **y** -- start y coordinate
- **len** -- number of pixels to decode
- **buf** -- a buffer to store the decoded pixels

Returns LV_RES_OK: ok; LV_RES_INV: failed

void **lv_img_decoder_built_in_close**(*lv_img_decoder_t* *decoder, *lv_img_decoder_dsc_t* *dsc)

Close the pending decoding. Free resources etc.

Parameters

- **decoder** -- pointer to the decoder the function associated with
- **dsc** -- pointer to decoder descriptor

struct _lv_img_decoder

Public Members

lv_img_decoder_info_f_t **info_cb**

lv_img_decoder_open_f_t **open_cb**

lv_img_decoder_read_line_f_t **read_line_cb**

lv_img_decoder_close_f_t **close_cb**

lv_img_decoder_user_data_t **user_data**

struct _lv_img_decoder_dsc

#include <lv_img_decoder.h> Describe an image decoding session. Stores data about the decoding

Public Members

lv_img_decoder_t ***decoder**

The decoder which was able to open the image source

const void ***src**

The image source. A file path like "S:my_img.png" or pointer to an *lv_img_dsc_t* variable

lv_color_t **color**

Style to draw the image.

lv_img_src_t **src_type**

Type of the source: file or variable. Can be set in `open` function if required

lv_img_header_t **header**

Info about the opened image: color format, size, etc. MUST be set in `open` function

const uint8_t ***img_data**

Pointer to a buffer where the image's data (pixels) are stored in a decoded, plain format. MUST be set in `open` function

uint32_t **time_to_open**

How much time did it take to open the image. [ms] If not set `lv_img_cache` will measure and set the time to open

const char ***error_msg**

A text to display instead of the image when the image can't be opened. Can be set in `open` function or set NULL.

void ***user_data**

Store any custom data here is required

Image cache

Functions

lv_img_cache_entry_t ***_lv_img_cache_open(const** void **src*, lv_color_t *color*)

Open an image using the image decoder interface and cache it. The image will be left open meaning if the image decoder open callback allocated memory then it will remain. The image is closed if a new image is opened and the new image takes its place in the cache.

Parameters

- **src** -- source of the image. Path to file or pointer to an *lv_img_dsc_t* variable
- **color** -- The color of the image with LV_IMG_CF_ALPHA_...

Returns pointer to the cache entry or NULL if can open the image

void **lv_img_cache_set_size(**uint16_t *new_slot_num*)

Set the number of images to be cached. More cached images mean more opened image at same time which might mean more memory usage. E.g. if 20 PNG or JPG images are open in the RAM they consume memory while opened in the cache.

Parameters **new_entry_cnt** -- number of image to cache

void **lv_img_cache_invalidate_src(const** void **src*)

Invalidate an image source in the cache. Useful if the image source is updated therefore it needs to be cached again.

Parameters **src** -- an image source path to a file or pointer to an *lv_img_dsc_t* variable.

struct lv_img_cache_entry_t

#include <lv_img_cache.h> When loading images from the network it can take a long time to download and decode the image.

To avoid repeating this heavy load images can be cached.

Public Members

lv_img_decoder_dsc_t **dec_dsc**

Image information

int32_t **life**

Count the cache entries's life. Add time_to_open to life when the entry is used. Decrement all lifes by one every in every ::lv_img_cache_open. If life == 0 the entry can be reused

Image buffer

Typedefs

typedef uint8_t **lv_img_cf_t**

Enums

enum [anonymous]
Values:

enumerator LV_IMG_CF_UNKNOWN

enumerator LV_IMG_CF_RAW
Contains the file as it is. Needs custom decoder function

enumerator LV_IMG_CF_RAW_ALPHA
Contains the file as it is. The image has alpha. Needs custom decoder function

enumerator LV_IMG_CF_RAW_CHROMA_KEYED
Contains the file as it is. The image is chroma keyed. Needs custom decoder function

enumerator LV_IMG_CF_TRUE_COLOR
Color format and depth should match with LV_COLOR settings

enumerator LV_IMG_CF_TRUE_COLOR_ALPHA
Same as `LV_IMG_CF_TRUE_COLOR` but every pixel has an alpha byte

enumerator LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED
Same as `LV_IMG_CF_TRUE_COLOR` but LV_COLOR_TRANSP pixels will be transparent

enumerator LV_IMG_CF_INDEXED_1BIT
Can have 2 different colors in a palette (always chroma keyed)

enumerator LV_IMG_CF_INDEXED_2BIT
Can have 4 different colors in a palette (always chroma keyed)

enumerator LV_IMG_CF_INDEXED_4BIT
Can have 16 different colors in a palette (always chroma keyed)

enumerator LV_IMG_CF_INDEXED_8BIT
Can have 256 different colors in a palette (always chroma keyed)

enumerator LV_IMG_CF_ALPHA_1BIT
Can have one color and it can be drawn or not

enumerator LV_IMG_CF_ALPHA_2BIT
Can have one color but 4 different alpha value

enumerator LV_IMG_CF_ALPHA_4BIT
Can have one color but 16 different alpha value

enumerator LV_IMG_CF_ALPHA_8BIT
Can have one color but 256 different alpha value

enumerator LV_IMG_CF_RESERVED_15
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_16
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_17
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_18
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_19
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_20
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_21
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_22
Reserved for further use.

enumerator LV_IMG_CF_RESERVED_23
Reserved for further use.

enumerator LV_IMG_CF_USER_ENCODED_0
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_1
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_2
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_3
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_4
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_5
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_6
User holder encoding format.

enumerator LV_IMG_CF_USER_ENCODED_7
User holder encoding format.

Functions

lv_img_dsc_t ***lv_img_buf_alloc**(lv_coord_t *w*, lv_coord_t *h*, *lv_img_cf_t cf*)
Allocate an image buffer in RAM

Parameters

- **w** -- width of image
- **h** -- height of image
- **cf** -- a color format (`LV_IMG_CF_...`)

Returns an allocated image, or NULL on failure

lv_color_t **lv_img_buf_get_px_color**(*lv_img_dsc_t* **dsc*, lv_coord_t *x*, lv_coord_t *y*, lv_color_t *color*)
Get the color of an image's pixel

Parameters

- **dsc** -- an image descriptor
- **x** -- x coordinate of the point to get
- **y** -- x coordinate of the point to get

- **color** -- the color of the image. In case of LV_IMG_CF_ALPHA_1/2/4/8 this color is used. Not used in other cases.
- **safe** -- true: check out of bounds

Returns color of the point

lv_opa_t **lv_img_buf_get_px_alpha**(*lv_img_dsc_t* *dsc, lv_coord_t *x*, lv_coord_t *y*)
Get the alpha value of an image's pixel

Parameters

- **dsc** -- pointer to an image descriptor
- **x** -- x coordinate of the point to set
- **y** -- x coordinate of the point to set
- **safe** -- true: check out of bounds

Returns alpha value of the point

void **lv_img_buf_set_px_color**(*lv_img_dsc_t* *dsc, lv_coord_t *x*, lv_coord_t *y*, lv_color_t *c*)
Set the color of a pixel of an image. The alpha channel won't be affected.

Parameters

- **dsc** -- pointer to an image descriptor
- **x** -- x coordinate of the point to set
- **y** -- x coordinate of the point to set
- **c** -- color of the point
- **safe** -- true: check out of bounds

void **lv_img_buf_set_px_alpha**(*lv_img_dsc_t* *dsc, lv_coord_t *x*, lv_coord_t *y*, lv_opa_t *opa*)
Set the alpha value of a pixel of an image. The color won't be affected

Parameters

- **dsc** -- pointer to an image descriptor
- **x** -- x coordinate of the point to set
- **y** -- x coordinate of the point to set
- **opa** -- the desired opacity
- **safe** -- true: check out of bounds

void **lv_img_buf_set_palette**(*lv_img_dsc_t* *dsc, uint8_t *id*, lv_color_t *c*)
Set the palette color of an indexed image. Valid only for LV_IMG_CF_INDEXED1/2/4/8

Parameters

- **dsc** -- pointer to an image descriptor
- **id** -- the palette color to set:
  - for LV_IMG_CF_INDEXED1: 0..1
  - for LV_IMG_CF_INDEXED2: 0..3
  - for LV_IMG_CF_INDEXED4: 0..15
  - for LV_IMG_CF_INDEXED8: 0..255

- **c** -- the color to set

void **lv_img_buf_free**(*lv_img_dsc_t *dsc*)
Free an allocated image buffer

Parameters **dsc** -- image buffer to free

uint32_t **lv_img_buf_get_img_size**(lv_coord_t *w*, lv_coord_t *h*, *lv_img_cf_t cf*)
Get the memory consumption of a raw bitmap, given color format and dimensions.

Parameters

- **w** -- width
- **h** -- height
- **cf** -- color format

Returns size in bytes

void **_lv_img_buf_transform_init**(*lv_img_transform_dsc_t *dsc*)
Initialize a descriptor to rotate an image

Parameters **dsc** -- pointer to an *lv_img_transform_dsc_t* variable whose `cfg` field is initialized

bool **_lv_img_buf_transform_anti_alias**(*lv_img_transform_dsc_t *dsc*)
Continue transformation by taking the neighbors into account

Parameters **dsc** -- pointer to the transformation descriptor

static inline bool **_lv_img_buf_transform**(*lv_img_transform_dsc_t *dsc*, lv_coord_t *x*, lv_coord_t *y*)
Get which color and opa would come to a pixel if it were rotated

---

Note: the result is written back to `dsc->res_color` and `dsc->res_opa`

Parameters

- **dsc** -- a descriptor initialized by `lv_img_buf_rotate_init`
- **x** -- the coordinate which color and opa should be get
- **y** -- the coordinate which color and opa should be get

Returns true: there is valid pixel on these x/y coordinates; false: the rotated pixel was out of the image

void **_lv_img_buf_get_transformed_area**(lv_area_t **res*, lv_coord_t *w*, lv_coord_t *h*, int16_t *angle*, uint16_t *zoom*, **const** lv_point_t **pivot*)
Get the area of a rectangle if its rotated and scaled

Parameters

- **res** -- store the coordinates here
- **w** -- width of the rectangle to transform
- **h** -- height of the rectangle to transform
- **angle** -- angle of rotation
- **zoom** -- zoom, (256 no zoom)

- **pivot** -- x,y pivot coordinates of rotation

struct lv_img_header_t

#include <lv_img_buf.h> LVGL image header

Public Members

uint32_t **h**

uint32_t **w**

uint32_t **reserved**

uint32_t **always_zero**

uint32_t **cf**

struct lv_img_header_t

#include <lv_img_buf.h> LVGL image header

Public Members

uint32_t **h**

uint32_t **w**

uint32_t **reserved**

uint32_t **always_zero**

uint32_t **cf**

struct lv_img_dsc_t

#include <lv_img_buf.h> Image header it is compatible with the result from image converter utility

Public Members

lv_img_header_t **header**

uint32_t **data_size**

const uint8_t ***data**

struct lv_img_transform_dsc_t

Public Members

const void ***src**

lv_coord_t **src_w**

lv_coord_t **src_h**

lv_coord_t **pivot_x**

lv_coord_t **pivot_y**

int16_t **angle**

uint16_t **zoom**

lv_color_t **color**

lv_img_cf_t **cf**

bool **antialias**

struct *lv_img_transform_dsc_t*::[**anonymous**] **cfg**

lv_opa_t **opa**

struct *lv_img_transform_dsc_t*::[**anonymous**] **res**

lv_img_dsc_t **img_dsc**

int32_t **pivot_x_256**

int32_t **pivot_y_256**

int32_t **sinma**

int32_t **cosma**

uint8_t **chroma_keyed**

uint8_t **has_alpha**

uint8_t **native_color**

uint32_t **zoom_inv**

lv_coord_t **xs**

lv_coord_t **ys**

lv_coord_t **xs_int**

lv_coord_t **ys_int**

uint32_t **pxi**

uint8_t **px_size**

struct *lv_img_transform_dsc_t*::[**anonymous**] **tmp**

Image draw

Functions

void **lv_draw_img_dsc_init**(*lv_draw_img_dsc_t* *dsc)

void **lv_draw_img**(**const** lv_area_t *coords, **const** lv_area_t *mask, **const** void *src, **const** *lv_draw_img_dsc_t* *dsc)

Draw an image

Parameters

- **coords** -- the coordinates of the image
- **mask** -- the image will be drawn only in this area
- **src** -- pointer to a lv_color_t array which contains the pixels of the image
- **dsc** -- pointer to an initialized `lv_draw_img_dsc_t` variable

lv_img_src_t **lv_img_src_get_type**(**const** void *src)

Get the type of an image source

Parameters **src** -- pointer to an image source:

- pointer to an 'lv_img_t' variable (image stored internally and compiled into the code)
- a path to a file (e.g. "S:/folder/image.bin")
- or a symbol (e.g. LV_SYMBOL_CLOSE)

Returns type of the image source LV_IMG_SRC_VARIABLE/FILE/SYMBOL/UNKNOWN

uint8_t **lv_img_cf_get_px_size**(*lv_img_cf_t cf*)
Get the pixel size of a color format in bits

Parameters **cf** -- a color format (`LV_IMG_CF_...`)

Returns the pixel size in bits

bool **lv_img_cf_is_chroma_keyed**(*lv_img_cf_t cf*)
Check if a color format is chroma keyed or not

Parameters **cf** -- a color format (`LV_IMG_CF_...`)

Returns true: chroma keyed; false: not chroma keyed

bool **lv_img_cf_has_alpha**(*lv_img_cf_t cf*)
Check if a color format has alpha channel or not

Parameters **cf** -- a color format (`LV_IMG_CF_...`)

Returns true: has alpha channel; false: doesn't have alpha channel

struct lv_draw_img_dsc_t

Public Members

lv_opa_t **opa**

uint16_t **angle**

lv_point_t **pivot**

uint16_t **zoom**

lv_opa_t **recolor_opa**

lv_color_t **recolor**

lv_blend_mode_t **blend_mode**

uint8_t **antialias**

4.9 File system

LVGL has a 'File system' abstraction module that enables you to attach any type of file systems. The file system is identified by a drive letter. For example, if the SD card is associated with the letter `'S'`, a file can be reached like `"S:path/to/file.txt"`.

4.9.1 Add a driver

To add a driver, `lv_fs_drv_t` needs to be initialized like this:

```
lv_fs_drv_t drv;
lv_fs_drv_init(&drv);                     /*Basic initialization*/

drv.letter = 'S';                         /*An uppercase letter to identify the drive␣
↪*/
drv.file_size = sizeof(my_file_object);   /*Size required to store a file object*/
drv.rddir_size = sizeof(my_dir_object);   /*Size required to store a directory object␣
↪(used by dir_open/close/read)*/
drv.ready_cb = my_ready_cb;               /*Callback to tell if the drive is ready to␣
↪use */
drv.open_cb = my_open_cb;                 /*Callback to open a file */
drv.close_cb = my_close_cb;               /*Callback to close a file */
drv.read_cb = my_read_cb;                 /*Callback to read a file */
drv.write_cb = my_write_cb;               /*Callback to write a file */
drv.seek_cb = my_seek_cb;                 /*Callback to seek in a file (Move cursor)␣
↪*/
drv.tell_cb = my_tell_cb;                 /*Callback to tell the cursor position  */
drv.trunc_cb = my_trunc_cb;               /*Callback to delete a file */
drv.size_cb = my_size_cb;                 /*Callback to tell a file's size */
drv.rename_cb = my_rename_cb;             /*Callback to rename a file */


drv.dir_open_cb = my_dir_open_cb;         /*Callback to open directory to read its␣
↪content */
drv.dir_read_cb = my_dir_read_cb;         /*Callback to read a directory's content */
drv.dir_close_cb = my_dir_close_cb;       /*Callback to close a directory */

drv.free_space_cb = my_free_space_cb;     /*Callback to tell free space on the drive␣
↪*/

drv.user_data = my_user_data;             /*Any custom data if required*/

lv_fs_drv_register(&drv);                 /*Finally register the drive*/
```

Any of the callbacks can be `NULL` to indicate that operation is not supported.

As an example of how the callbacks are used, if you use `lv_fs_open(&file, "S:/folder/file.txt", LV_FS_MODE_WR)`, LVGL:

1. Verifies that a registered drive exists with the letter `'S'`.
2. Checks if it's `open_cb` is implemented (not `NULL`).
3. Calls the set `open_cb` with `"folder/file.txt"` path.

4.9.2 Usage example

The example below shows how to read from a file:

```
lv_fs_file_t f;
lv_fs_res_t res;
res = lv_fs_open(&f, "S:folder/file.txt", LV_FS_MODE_RD);
if(res != LV_FS_RES_OK) my_error_handling();

uint32_t read_num;
uint8_t buf[8];
res = lv_fs_read(&f, buf, 8, &read_num);
if(res != LV_FS_RES_OK || read_num != 8) my_error_handling();

lv_fs_close(&f);
```

The mode in `lv_fs_open` *can be* `LV_FS_MODE_WR` *to open for write or* `LV_FS_MODE_RD | LV_FS_MODE_WR` *for both*

This example shows how to read a directory's content. It's up to the driver how to mark the directories, but it can be a good practice to insert a `'/'` in front of the directory name.

```
lv_fs_dir_t dir;
lv_fs_res_t res;
res = lv_fs_dir_open(&dir, "S:/folder");
if(res != LV_FS_RES_OK) my_error_handling();

char fn[256];
while(1) {
    res = lv_fs_dir_read(&dir, fn);
    if(res != LV_FS_RES_OK) {
        my_error_handling();
        break;
    }

    /*fn is empty, if not more files to read*/
    if(strlen(fn) == 0) {
        break;
    }

    printf("%s\n", fn);
}

lv_fs_dir_close(&dir);
```

4.9.3 Use drivers for images

Image objects can be opened from files too (besides variables stored in the flash).

To initialize the image, the following callbacks are required:

- open
- close
- read
- seek

- tell

4.9.4 API

Typedefs

typedef uint8_t **lv_fs_res_t**

typedef uint8_t **lv_fs_mode_t**

typedef struct *_lv_fs_drv_t* **lv_fs_drv_t**

Enums

enum [anonymous]
Errors in the file system module.

Values:

enumerator LV_FS_RES_OK

enumerator LV_FS_RES_HW_ERR

enumerator LV_FS_RES_FS_ERR

enumerator LV_FS_RES_NOT_EX

enumerator LV_FS_RES_FULL

enumerator LV_FS_RES_LOCKED

enumerator LV_FS_RES_DENIED

enumerator LV_FS_RES_BUSY

enumerator LV_FS_RES_TOUT

enumerator LV_FS_RES_NOT_IMP

enumerator LV_FS_RES_OUT_OF_MEM

enumerator LV_FS_RES_INV_PARAM

enumerator LV_FS_RES_UNKNOWN

enum [anonymous]
Filesystem mode.

Values:

enumerator LV_FS_MODE_WR

enumerator LV_FS_MODE_RD

Functions

void **_lv_fs_init**(void)
Initialize the File system interface

void **lv_fs_drv_init**(*lv_fs_drv_t* **drv*)
Initialize a file system driver with default values. It is used to surly have known values in the fields ant not memory junk. After it you can set the fields.

Parameters **drv** -- pointer to driver variable to initialize

void **lv_fs_drv_register**(*lv_fs_drv_t* **drv_p*)
Add a new drive

Parameters **drv_p** -- pointer to an lv_fs_drv_t structure which is inited with the corresponding function pointers. The data will be copied so the variable can be local.

lv_fs_drv_t ***lv_fs_get_drv**(char *letter*)
Give a pointer to a driver from its letter

Parameters **letter** -- the driver letter

Returns pointer to a driver or NULL if not found

bool **lv_fs_is_ready**(char *letter*)
Test if a drive is ready or not. If the `ready` function was not initialized `true` will be returned.

Parameters **letter** -- letter of the drive

Returns true: drive is ready; false: drive is not ready

lv_fs_res_t **lv_fs_open**(*lv_fs_file_t* **file_p*, **const** char **path*, *lv_fs_mode_t mode*)
Open a file

Parameters

- **file_p** -- pointer to a *lv_fs_file_t* variable
- **path** -- path to the file beginning with the driver letter (e.g. S:/folder/file.txt)
- **mode** -- read: FS_MODE_RD, write: FS_MODE_WR, both: FS_MODE_RD | FS_MODE_WR

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_close**(*lv_fs_file_t* **file_p*)
Close an already opened file

Parameters **file_p** -- pointer to a *lv_fs_file_t* variable

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_remove**(**const** char **path*)
Delete a file

Parameters **path** -- path of the file to delete

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_read**(*lv_fs_file_t* **file_p*, void **buf*, uint32_t *btr*, uint32_t **br*)
Read from a file

Parameters

- **file_p** -- pointer to a *lv_fs_file_t* variable
- **buf** -- pointer to a buffer where the read bytes are stored

- **btr** -- Bytes To Read
- **br** -- the number of real read bytes (Bytes Read). NULL if unused.

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_write(***lv_fs_file_t *file_p*, **const** void **buf*, uint32_t *btw*, uint32_t **bw***)
Write into a file

Parameters

- **file_p** -- pointer to a *lv_fs_file_t* variable
- **buf** -- pointer to a buffer with the bytes to write
- **btr** -- Bytes To Write
- **br** -- the number of real written bytes (Bytes Written). NULL if unused.

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_seek(***lv_fs_file_t *file_p*, uint32_t *pos***)**
Set the position of the 'cursor' (read write pointer) in a file

Parameters

- **file_p** -- pointer to a *lv_fs_file_t* variable
- **pos** -- the new position expressed in bytes index (0: start of file)

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_tell(***lv_fs_file_t *file_p*, uint32_t **pos***)
Give the position of the read write pointer

Parameters

- **file_p** -- pointer to a *lv_fs_file_t* variable
- **pos_p** -- pointer to store the position of the read write pointer

Returns LV_FS_RES_OK or any error from 'fs_res_t'

lv_fs_res_t **lv_fs_trunc(***lv_fs_file_t *file_p***)**
Truncate the file size to the current position of the read write pointer

Parameters **file_p** -- pointer to an 'ufs_file_t' variable. (opened with lv_fs_open)

Returns LV_FS_RES_OK: no error, the file is read any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_size(***lv_fs_file_t *file_p*, uint32_t **size***)
Give the size of a file bytes

Parameters

- **file_p** -- pointer to a *lv_fs_file_t* variable
- **size** -- pointer to a variable to store the size

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_rename(const** char **oldname*, **const** char **newname***)**
Rename a file

Parameters

- **oldname** -- path to the file
- **newname** -- path with the new name

Returns LV_FS_RES_OK or any error from 'fs_res_t'

lv_fs_res_t **lv_fs_dir_open**(*lv_fs_dir_t *rddir_p*, **const** char **path*)
Initialize a 'fs_dir_t' variable for directory reading

Parameters

- **rddir_p** -- pointer to a '*lv_fs_dir_t*' variable
- **path** -- path to a directory

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_dir_read**(*lv_fs_dir_t *rddir_p*, char **fn*)
Read the next filename form a directory. The name of the directories will begin with '/'

Parameters

- **rddir_p** -- pointer to an initialized 'fs_dir_t' variable
- **fn** -- pointer to a buffer to store the filename

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_dir_close**(*lv_fs_dir_t *rddir_p*)
Close the directory reading

Parameters **rddir_p** -- pointer to an initialized 'fs_dir_t' variable

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

lv_fs_res_t **lv_fs_free_space**(char *letter*, uint32_t **total_p*, uint32_t **free_p*)
Get the free and total size of a driver in kB

Parameters

- **letter** -- the driver letter
- **total_p** -- pointer to store the total size [kB]
- **free_p** -- pointer to store the free size [kB]

Returns LV_FS_RES_OK or any error from lv_fs_res_t enum

char ***lv_fs_get_letters**(char **buf*)
Fill a buffer with the letters of existing drivers

Parameters **buf** -- buffer to store the letters ('\0' added after the last letter)

Returns the buffer

const char ***lv_fs_get_ext**(**const** char **fn*)
Return with the extension of the filename

Parameters **fn** -- string with a filename

Returns pointer to the beginning extension or empty string if no extension

char ***lv_fs_up**(char **path*)
Step up one level

Parameters **path** -- pointer to a file name

Returns the truncated file name

const char ***lv_fs_get_last**(**const** char **path*)
Get the last element of a path (e.g. U:/folder/file -> file)

Parameters **path** -- pointer to a file name

Returns pointer to the beginning of the last element in the path

struct _lv_fs_drv_t

Public Members

char **letter**

uint16_t **file_size**

uint16_t **rddir_size**

bool (***ready_cb**)(**struct** *_lv_fs_drv_t* *drv)

lv_fs_res_t (***open_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p, **const** char *path, *lv_fs_mode_t* mode)

lv_fs_res_t (***close_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p)

lv_fs_res_t (***remove_cb**)(**struct** *_lv_fs_drv_t* *drv, **const** char *fn)

lv_fs_res_t (***read_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p, void *buf, uint32_t btr, uint32_t *br)

lv_fs_res_t (***write_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p, **const** void *buf, uint32_t btw, uint32_t *bw)

lv_fs_res_t (***seek_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p, uint32_t pos)

lv_fs_res_t (***tell_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p, uint32_t *pos_p)

lv_fs_res_t (***trunc_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p)

lv_fs_res_t (***size_cb**)(**struct** *_lv_fs_drv_t* *drv, void *file_p, uint32_t *size_p)

lv_fs_res_t (***rename_cb**)(**struct** *_lv_fs_drv_t* *drv, **const** char *oldname, **const** char *newname)

lv_fs_res_t (***free_space_cb**)(**struct** *_lv_fs_drv_t* *drv, uint32_t *total_p, uint32_t *free_p)

lv_fs_res_t (***dir_open_cb**)(**struct** *_lv_fs_drv_t* *drv, void *rddir_p, **const** char *path)

lv_fs_res_t (***dir_read_cb**)(**struct** *_lv_fs_drv_t* *drv, void *rddir_p, char *fn)

lv_fs_res_t (***dir_close_cb**)(**struct** *_lv_fs_drv_t* *drv, void *rddir_p)

lv_fs_drv_user_data_t **user_data**
Custom file user data

struct lv_fs_file_t

Public Members

void ***file_d**

lv_fs_drv_t ***drv**

struct lv_fs_dir_t

Public Members

void *__dir_d__

lv_fs_drv_t *__drv__

4.10 Animations

You can automatically change the value of a variable between a start and an end value using animations. The animation will happen by the periodical call of an "animator" function with the corresponding value parameter.

The *animator* functions has the following prototype:

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

This prototype is compatible with the majority of the *set* function of LVGL. For example `lv_obj_set_x(obj, value)` or `lv_obj_set_width(obj, value)`

4.10.1 Create an animation

To create an animation an `lv_anim_t` variable has to be initialized and configured with `lv_anim_set_...()` functions.

```
/* INITIALIZE AN ANIMATION
 *----------------------*/

lv_anim_t a;
lv_anim_init(&a);

/* MANDATORY SETTINGS
 *------------------*/

/*Set the "animator" function*/
lv_anim_set_exec_cb(&a, (lv_anim_exec_xcb_t) lv_obj_set_x);

/*Set the "animator" function*/
lv_anim_set_var(&a, obj);

/*Length of the animation [ms]*/
lv_anim_set_time(&a, duration);

/*Set start and end values. E.g. 0, 150*/
lv_anim_set_values(&a, start, end);

/* OPTIONAL SETTINGS
 *------------------*/

/*Time to wait before starting the animation [ms]*/
lv_anim_set_delay(&a, delay);

/*Set path (curve). Default is linear*/
lv_anim_set_path(&a, &path);
```

(continues on next page)

(continued from previous page)

```
/*Set a callback to call when animation is ready.*/
lv_anim_set_ready_cb(&a, ready_cb);

/*Set a callback to call when animation is started (after delay).*/
lv_anim_set_start_cb(&a, start_cb);

/*Play the animation backward too with this duration. Default is 0 (disabled) [ms]*/
lv_anim_set_playback_time(&a, wait_time);

/*Delay before playback. Default is 0 (disabled) [ms]*/
lv_anim_set_playback_delay(&a, wait_time);

/*Number of repetitions. Default is 1.  LV_ANIM_REPEAT_INFINIT for infinite␣
↪repetition*/
lv_anim_set_repeat_count(&a, wait_time);

/*Delay before repeat. Default is 0 (disabled) [ms]*/
lv_anim_set_repeat_delay(&a, wait_time);

/*true (default): apply the start vale immediately, false: apply start vale after␣
↪delay when then anim. really starts. */
lv_anim_set_early_apply(&a, true/false);

/* START THE ANIMATION
 *------------------*/
lv_anim_start(&a);                                  /*Start the animation*/
```

You can apply **multiple different animations** on the same variable at the same time. For example, animate the x and y coordinates with `lv_obj_set_x` and `lv_obj_set_y`. However, only one animation can exist with a given variable and function pair. Therefore `lv_anim_start()` will delete the already existing variable-function animations.

4.10.2 Animation path

You can determinate the **path of animation**. In the most simple case, it is linear, which means the current value between *start* and *end* is changed linearly. A *path* is mainly a function which calculates the next value to set based on the current state of the animation. Currently, there are the following built-in paths functions:

- **lv_anim_path_linear** linear animation
- **lv_anim_path_step** change in one step at the end
- **lv_anim_path_ease_in** slow at the beginning
- **lv_anim_path_ease_out** slow at the end
- **lv_anim_path_ease_in_out** slow at the beginning and end too
- **lv_anim_path_overshoot** overshoot the end value
- **lv_anim_path_bounce** bounce back a little from the end value (like hitting a wall)

A path can be initialized like this:

```
lv_anim_path_t path;
lv_anim_path_init(&path);
lv_anim_path_set_cb(&path, lv_anim_path_overshoot);
```

(continues on next page)

(continued from previous page)

```
lv_anim_path_set_user_data(&path, &foo); /*Optional for custom functions*/

/*Set the path in an animation*/
lv_anim_set_path(&a, &path);
```

4.10.3 Speed vs time

By default, you can set the animation time. But, in some cases, the **animation speed** is more practical.

The `lv_anim_speed_to_time(speed, start, end)` function calculates the required time in milliseconds to reach the end value from a start value with the given speed. The speed is interpreted in *unit/sec* dimension. For example, `lv_anim_speed_to_time(20,0,100)` will give 5000 milliseconds. For example, in case of `lv_obj_set_x` *unit* is pixels so *20* means *20 px/sec* speed.

4.10.4 Delete animations

You can **delete an animation** by `lv_anim_del(var, func)` by providing the animated variable and its animator function.

4.10.5 API

Input device

Typedefs

typedef uint8_t **lv_anim_enable_t**

typedef lv_coord_t **lv_anim_value_t**
Type of the animated value

typedef *lv_anim_value_t* (***lv_anim_path_cb_t**)(**const struct** *_lv_anim_path_t**, **const struct** *_lv_anim_t**)
Get the current value during an animation

typedef struct *_lv_anim_path_t* **lv_anim_path_t**

typedef void (***lv_anim_exec_xcb_t**)(void*, *lv_anim_value_t*)
Generic prototype of "animator" functions. First parameter is the variable to animate. Second parameter is the value to set. Compatible with `lv_xxx_set_yyy(obj, value)` functions The `x` in `_xcb_t` means its not a fully generic prototype because it doesn't receive `lv_anim_t *` as its first argument

typedef void (***lv_anim_custom_exec_cb_t**)(**struct** *_lv_anim_t**, *lv_anim_value_t*)
Same as `lv_anim_exec_xcb_t` but receives `lv_anim_t *` as the first parameter. It's more consistent but less convenient. Might be used by binding generator functions.

typedef void (***lv_anim_ready_cb_t**)(**struct** *_lv_anim_t**)
Callback to call when the animation is ready

typedef void (***lv_anim_start_cb_t**)(**struct** *_lv_anim_t**)
Callback to call when the animation really stars (considering `delay`)

typedef struct *_lv_anim_t* **lv_anim_t**
Describes an animation

Enums

enum [anonymous]
Can be used to indicate if animations are enabled or disabled in a case

Values:

enumerator LV_ANIM_OFF

enumerator LV_ANIM_ON

Functions

void **_lv_anim_core_init**(void)
Init. the animation module

void **lv_anim_init**(*lv_anim_t* *a)
Initialize an animation variable. E.g.: lv_anim_t a; lv_anim_init(&a); lv_anim_set_...(&a);

Parameters **a** -- pointer to an `lv_anim_t` variable to initialize

static inline void **lv_anim_set_var**(*lv_anim_t* *a, void *var)
Set a variable to animate

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **var** -- pointer to a variable to animate

static inline void **lv_anim_set_exec_cb**(*lv_anim_t* *a, *lv_anim_exec_xcb_t* *exec_cb*)
Set a function to animate `var`

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **exec_cb** -- a function to execute during animation LittelvGL's built-in functions can be used. E.g. lv_obj_set_x

static inline void **lv_anim_set_time**(*lv_anim_t* *a, uint32_t *duration*)
Set the duration of an animation

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **duration** -- duration of the animation in milliseconds

static inline void **lv_anim_set_delay**(*lv_anim_t* *a, uint32_t *delay*)
Set a delay before starting the animation

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **delay** -- delay before the animation in milliseconds

static inline void **lv_anim_set_values**(*lv_anim_t* *a, *lv_anim_value_t* *start*, *lv_anim_value_t* *end*)
Set the start and end values of an animation

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable

- **start** -- the start value
- **end** -- the end value

static inline void **lv_anim_set_custom_exec_cb**(*lv_anim_t *a, lv_anim_custom_exec_cb_t exec_cb*)

Similar to `lv_anim_set_exec_cb` but `lv_anim_custom_exec_cb_t` receives `lv_anim_t *` as its first parameter instead of `void *`. This function might be used when LVGL is binded to other languages because it's more consistent to have `lv_anim_t *` as first parameter. The variable to animate can be stored in the animation's `user_sata`

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **exec_cb** -- a function to execute.

static inline void **lv_anim_set_path**(*lv_anim_t *a*, **const** *lv_anim_path_t *path*)

Set the path (curve) of the animation.

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **path** -- a function the get the current value of the animation. The built in functions starts with `lv_anim_path_...`

static inline void **lv_anim_set_start_cb**(*lv_anim_t *a*, *lv_anim_ready_cb_t start_cb*)

Set a function call when the animation really starts (considering `delay`)

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **start_cb** -- a function call when the animation starts

static inline void **lv_anim_set_ready_cb**(*lv_anim_t *a*, *lv_anim_ready_cb_t ready_cb*)

Set a function call when the animation is ready

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **ready_cb** -- a function call when the animation is ready

static inline void **lv_anim_set_playback_time**(*lv_anim_t *a*, uint32_t *time*)

Make the animation to play back to when the forward direction is ready

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **time** -- the duration of the playback animation in in milliseconds. 0: disable playback

static inline void **lv_anim_set_playback_delay**(*lv_anim_t *a*, uint32_t *delay*)

Make the animation to play back to when the forward direction is ready

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **delay** -- delay in milliseconds before starting the playback animation.

static inline void **lv_anim_set_repeat_count**(*lv_anim_t *a*, uint16_t *cnt*)

Make the animation repeat itself.

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **cnt** -- repeat count or `LV_ANIM_REPEAT_INFINITE` for infinite repetition. 0: to disable repetition.

static inline void **lv_anim_set_repeat_delay**(*lv_anim_t* *a, uint32_t *delay*)
Set a delay before repeating the animation.

Parameters

- **a** -- pointer to an initialized `lv_anim_t` variable
- **delay** -- delay in milliseconds before repeating the animation.

void **lv_anim_start**(*lv_anim_t* *a)
Create an animation

Parameters **a** -- an initialized 'anim_t' variable. Not required after call.

static inline void **lv_anim_path_init**(*lv_anim_path_t* *path)
Initialize an animation path

Parameters **path** -- pointer to path

static inline void **lv_anim_path_set_cb**(*lv_anim_path_t* *path, *lv_anim_path_cb_t* cb)
Set a callback for a path

Parameters

- **path** -- pointer to an initialized path
- **cb** -- the callback

static inline void **lv_anim_path_set_user_data**(*lv_anim_path_t* *path, void *user_data)
Set a user data for a path

Parameters

- **path** -- pointer to an initialized path
- **user_data** -- pointer to the user data

static inline uint32_t **lv_anim_get_delay**(*lv_anim_t* *a)
Get a delay before starting the animation

Parameters **a** -- pointer to an initialized `lv_anim_t` variable

Returns delay before the animation in milliseconds

bool **lv_anim_del**(void *var, *lv_anim_exec_xcb_t exec_cb*)
Delete an animation of a variable with a given animator function

Parameters

- **var** -- pointer to variable
- **exec_cb** -- a function pointer which is animating 'var', or NULL to ignore it and delete all the animations of 'var

Returns true: at least 1 animation is deleted, false: no animation is deleted

void **lv_anim_del_all**(void)
Delete all the animations animation

lv_anim_t ***lv_anim_get**(void *var, *lv_anim_exec_xcb_t exec_cb*)
Get the animation of a variable and its `exec_cb`.

Parameters

- **var** -- pointer to variable
- **exec_cb** -- a function pointer which is animating 'var', or NULL to delete all the animations of 'var'

Returns pointer to the animation.

static inline bool **lv_anim_custom_del**(*lv_anim_t *a, lv_anim_custom_exec_cb_t exec_cb*)

Delete an animation by getting the animated variable from `a`. Only animations with `exec_cb` will be deleted. This function exists because it's logical that all anim. functions receives an `lv_anim_t` as their first parameter. It's not practical in C but might make the API more consequent and makes easier to generate bindings.

Parameters

- **a** -- pointer to an animation.
- **exec_cb** -- a function pointer which is animating 'var', or NULL to ignore it and delete all the animations of 'var

Returns true: at least 1 animation is deleted, false: no animation is deleted

uint16_t **lv_anim_count_running**(void)

Get the number of currently running animations

Returns the number of running animations

uint32_t **lv_anim_speed_to_time**(uint32_t *speed*, *lv_anim_value_t start*, *lv_anim_value_t end*)

Calculate the time of an animation with a given speed and the start and end values

Parameters

- **speed** -- speed of animation in unit/sec
- **start** -- start value of the animation
- **end** -- end value of the animation

Returns the required time [ms] for the animation with the given parameters

void **lv_anim_refr_now**(void)

Manually refresh the state of the animations. Useful to make the animations running in a blocking process where `lv_task_handler` can't run for a while. Shouldn't be used directly because it is called in `lv_refr_now()`.

lv_anim_value_t **lv_anim_path_linear**(**const** *lv_anim_path_t *path*, **const** *lv_anim_t *a*)

Calculate the current value of an animation applying linear characteristic

Parameters **a** -- pointer to an animation

Returns the current value to set

lv_anim_value_t **lv_anim_path_ease_in**(**const** *lv_anim_path_t *path*, **const** *lv_anim_t *a*)

Calculate the current value of an animation slowing down the start phase

Parameters **a** -- pointer to an animation

Returns the current value to set

lv_anim_value_t **lv_anim_path_ease_out**(**const** *lv_anim_path_t *path*, **const** *lv_anim_t *a*)

Calculate the current value of an animation slowing down the end phase

Parameters **a** -- pointer to an animation

Returns the current value to set

lv_anim_value_t **lv_anim_path_ease_in_out(const** *lv_anim_path_t* *path, **const** *lv_anim_t* *a**)**

Calculate the current value of an animation applying an "S" characteristic (cosine)

Parameters **a** -- pointer to an animation

Returns the current value to set

lv_anim_value_t **lv_anim_path_overshoot(const** *lv_anim_path_t* *path, **const** *lv_anim_t* *a**)**

Calculate the current value of an animation with overshoot at the end

Parameters **a** -- pointer to an animation

Returns the current value to set

lv_anim_value_t **lv_anim_path_bounce(const** *lv_anim_path_t* *path, **const** *lv_anim_t* *a**)**

Calculate the current value of an animation with 3 bounces

Parameters **a** -- pointer to an animation

Returns the current value to set

lv_anim_value_t **lv_anim_path_step(const** *lv_anim_path_t* *path, **const** *lv_anim_t* *a**)**

Calculate the current value of an animation applying step characteristic. (Set end value on the end of the animation)

Parameters **a** -- pointer to an animation

Returns the current value to set

Variables

const *lv_anim_path_t* **lv_anim_path_def**

struct _lv_anim_path_t

Public Members

lv_anim_path_cb_t **cb**

void ***user_data**

struct _lv_anim_t

#include <lv_anim.h> Describes an animation

Public Members

void ***var**

Variable to animate

lv_anim_exec_xcb_t **exec_cb**

Function to execute to animate

lv_anim_start_cb_t **start_cb**

Call it when the animation is starts (considering `delay`)

lv_anim_ready_cb_t **ready_cb**

Call it when the animation is ready

lv_anim_user_data_t **user_data**
Custom user data

lv_anim_path_t **path**
Describe the path (curve) of animations

int32_t **start**
Start value

int32_t **current**
Current value

int32_t **end**
End value

int32_t **time**
Animation time in ms

int32_t **act_time**
Current time in animation. Set to negative to make delay.

uint32_t **playback_delay**
Wait before play back

uint32_t **playback_time**
Duration of playback animation

uint32_t **repeat_delay**
Wait before repeat

uint16_t **repeat_cnt**
Repeat count for the animation

uint8_t **early_apply**
1: Apply start value immediately even is there is `delay`

lv_anim_user_data_t **user_data**
Custom user data

uint8_t **playback_now**
Play back is in progress

uint8_t **run_round**
Indicates the animation has run in this round

uint32_t **time_orig**

4.11 Tasks

LVGL has a built-in task system. You can register a function to have it be called periodically. The tasks are handled and called in `lv_task_handler()`, which needs to be called periodically every few milliseconds. See *Porting* for more information.

The tasks are non-preemptive, which means a task cannot interrupt another task. Therefore, you can call any LVGL related function in a task.

4.11.1 Create a task

To create a new task, use `lv_task_create(task_cb, period_ms, LV_TASK_PRIO_OFF/LOWEST/LOW/MID/HIGH/HIGHEST, user_data)`. It will create an `lv_task_t *` variable, which can be used later to modify the parameters of the task. `lv_task_create_basic()` can also be used. It allows you to create a new task without specifying any parameters.

A task callback should have `void (*lv_task_cb_t)(lv_task_t *);` prototype.

For example:

```
void my_task(lv_task_t * task)
{
  /*Use the user_data*/
  uint32_t * user_data = task->user_data;
  printf("my_task called with user data: %d\n", *user_data);

  /*Do something with LVGL*/
  if(something_happened) {
    something_happened = false;
    lv_btn_create(lv_scr_act(), NULL);
  }
}

...

static uint32_t user_data = 10;
lv_task_t * task = lv_task_create(my_task, 500, LV_TASK_PRIO_MID, &user_data);
```

4.11.2 Ready and Reset

`lv_task_ready(task)` makes the task run on the next call of `lv_task_handler()`.

`lv_task_reset(task)` resets the period of a task. It will be called again after the defined period of milliseconds has elapsed.

4.11.3 Set parameters

You can modify some parameters of the tasks later:

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

4.11.4 One-shot tasks

You can make a task to run only once by calling`lv_task_once(task)`. The task will automatically be deleted after being called for the first time.

4.11.5 Measure idle time

You can get the idle percentage time `lv_task_handler` with `lv_task_get_idle()`. Note that, it doesn't measure the idle time of the overall system, only `lv_task_handler`. It can be misleading if you use an operating system and call `lv_task_handler` in an task, as it won't actually measure the time the OS spends in an idle thread.

4.11.6 Asynchronous calls

In some cases, you can't do an action immediately. For example, you can't delete an object right now because something else is still using it or you don't want to block the execution now. For these cases, you can use the `lv_async_call(my_function, data_p)` to make `my_function` be called on the next call of `lv_task_handler`. `data_p` will be passed to function when it's called. Note that, only the pointer of the data is saved so you need to ensure that the variable will be "alive" while the function is called. You can use *static*, global or dynamically allocated data.

For example:

```
void my_screen_clean_up(void * scr)
{
  /*Free some resources related to `scr`*/

  /*Finally delete the screen*/
  lv_obj_del(scr);
}

...

/*Do somethings with the object on the current screen*/

/*Delete screen on next call of `lv_task_handler`. So not now.*/
lv_async_call(my_screen_clean_up, lv_scr_act());

/*The screen is still valid so you can do other things with it*/
```

If you just want to delete an object, and don't need to clean anything up in `my_screen_cleanup`, you could just use `lv_obj_del_async`, which will delete the object on the next call to `lv_task_handler`.

4.11.7 API

An 'lv_task' is a void *(fp) (struct* _lv_task_t param) type function which will be called periodically. A priority (5 levels + disable) can be assigned to lv_tasks.

Typedefs

typedef void (***lv_task_cb_t**)(**struct** *_lv_task_t**)
Tasks execute this type of functions.

typedef uint8_t **lv_task_prio_t**

typedef struct *_lv_task_t* **lv_task_t**
Descriptor of a lv_task

Enums

enum [anonymous]
Possible priorities for lv_tasks

Values:

enumerator LV_TASK_PRIO_OFF

enumerator LV_TASK_PRIO_LOWEST

enumerator LV_TASK_PRIO_LOW

enumerator LV_TASK_PRIO_MID

enumerator LV_TASK_PRIO_HIGH

enumerator LV_TASK_PRIO_HIGHEST

enumerator _LV_TASK_PRIO_NUM

Functions

void **_lv_task_core_init**(void)
Init the lv_task module

lv_task_t ***lv_task_create_basic**(void)
Create an "empty" task. It needs to initialized with at least `lv_task_set_cb` and `lv_task_set_period`

Returns pointer to the created task

lv_task_t ***lv_task_create**(*lv_task_cb_t task_xcb*, uint32_t *period*, *lv_task_prio_t prio*, void **user_data*)
Create a new lv_task

Parameters

- **task_xcb** -- a callback which is the task itself. It will be called periodically. (the 'x' in the argument name indicates that its not a fully generic function because it not follows the `func_name(object, callback, ...)` convention)
- **period** -- call period in ms unit
- **prio** -- priority of the task (LV_TASK_PRIO_OFF means the task is stopped)
- **user_data** -- custom parameter

Returns pointer to the new task

void **lv_task_del**(*lv_task_t *task*)
Delete a lv_task

Parameters **task** -- pointer to task_cb created by task

void **lv_task_set_cb**(*lv_task_t *task*, *lv_task_cb_t task_cb*)
Set the callback the task (the function to call periodically)

Parameters

- **task** -- pointer to a task
- **task_cb** -- the function to call periodically

void **lv_task_set_prio**(*lv_task_t *task*, *lv_task_prio_t prio*)
Set new priority for a lv_task

Parameters

- **task** -- pointer to a lv_task
- **prio** -- the new priority

void **lv_task_set_period**(*lv_task_t *task*, uint32_t *period*)
Set new period for a lv_task

Parameters

- **task** -- pointer to a lv_task
- **period** -- the new period

void **lv_task_ready**(*lv_task_t *task*)
Make a lv_task ready. It will not wait its period.

Parameters **task** -- pointer to a lv_task.

void **lv_task_set_repeat_count**(*lv_task_t *task*, int32_t *repeat_count*)
Set the number of times a task will repeat.

Parameters

- **task** -- pointer to a lv_task.
- **repeat_count** -- -1 : infinity; 0 : stop ; n>0: residual times

void **lv_task_reset**(*lv_task_t *task*)
Reset a lv_task. It will be called the previously set period milliseconds later.

Parameters **task** -- pointer to a lv_task.

void **lv_task_enable**(bool *en*)
Enable or disable the whole lv_task handling

Parameters **en** -- true: lv_task handling is running, false: lv_task handling is suspended

uint8_t **lv_task_get_idle**(void)
Get idle percentage

Returns the lv_task idle in percentage

lv_task_t ***lv_task_get_next**(*lv_task_t *task*)
Iterate through the tasks

Parameters **task** -- NULL to start iteration or the previous return value to get the next task

Returns the next task or NULL if there is no more task

struct _lv_task_t
#include <lv_task.h> Descriptor of a lv_task

Public Members

uint32_t **period**
How often the task should run

uint32_t **last_run**
Last time the task ran

lv_task_cb_t **task_cb**
Task function

void ***user_data**
Custom user data

int32_t **repeat_count**
1: Task times; -1 : infinity; 0 : stop ; n>0: residual times

uint8_t **prio**
Task priority

Typedefs

typedef void (***lv_async_cb_t**)(void*)
Type for async callback.

typedef struct _lv_async_info_t **lv_async_info_t**

Functions

lv_res_t **lv_async_call**(*lv_async_cb_t async_xcb*, void **user_data*)
Call an asynchronous function the next time lv_task_handler() is run. This function is likely to return **before** the call actually happens!

Parameters

- **async_xcb** -- a callback which is the task itself. (the 'x' in the argument name indicates that its not a fully generic function because it not follows the `func_name(object, callback, ...)` convention)
- **user_data** -- custom parameter

4.12 Drawing

With LVGL, you don't need to draw anything manually. Just create objects (like buttons and labels), move and change them and LVGL will refresh and redraw what is required.

However, it might be useful to have a basic understanding of how drawing happens in LVGL.

The basic concept is to not draw directly to the screen, but draw to an internal buffer first and then copy that buffer to screen when the rendering is ready. It has two main advantages:

1. **Avoids flickering** while layers of the UI are drawn. For example, when drawing a *background + button + text*, each "stage" would be visible for a short time.
2. **It's faster** to modify a buffer in RAM and finally write one pixel once than read/write a display directly on each pixel access. (e.g. via a display controller with SPI interface). Hence, it's suitable for pixels that are redrawn multiple times (e.g. background + button + text).

4.12.1 Buffering types

As you already might learn in the *Porting* section, there are 3 types of buffers:

1. **One buffer** - LVGL draws the content of the screen into a buffer and sends it to the display. The buffer can be smaller than the screen. In this case, the larger areas will be redrawn in multiple parts. If only small areas changes (e.g. button press), then only those areas will be refreshed.
2. **Two non-screen-sized buffers** - having two buffers, LVGL can draw into one buffer while the content of the other buffer is sent to display in the background. DMA or other hardware should be used to transfer the data to the display to let the CPU draw meanwhile. This way, the rendering and refreshing of the display become parallel. If the buffer is smaller than the area to refresh, LVGL will draw the display's content in chunks similar to the *One buffer*.
3. **Two screen-sized buffers** - In contrast to *Two non-screen-sized buffers*, LVGL will always provide the whole screen's content, not only chunks. This way, the driver can simply change the address of the frame buffer to the buffer received from LVGL. Therefore, this method works best when the MCU has an LCD/TFT interface and the frame buffer is just a location in the RAM.

4.12.2 Mechanism of screen refreshing

1. Something happens on the GUI which requires redrawing. For example, a button has been pressed, a chart has been changed or an animation happened, etc.
2. LVGL saves the changed object's old and new area into a buffer, called an *Invalid area buffer*. For optimization, in some cases, objects are not added to the buffer:
   - Hidden objects are not added.
   - Objects completely out of their parent are not added.
   - Areas out of the parent are cropped to the parent's area.
   - The object on other screens are not added.
3. In every `LV_DISP_DEF_REFR_PERIOD` (set in *lv_conf.h*):
   - LVGL checks the invalid areas and joins the adjacent or intersecting areas.
   - Takes the first joined area, if it's smaller than the *display buffer*, then simply draw the areas' content to the *display buffer*. If the area doesn't fit into the buffer, draw as many lines as possible to the *display buffer*.
   - When the area is drawn, call `flush_cb` from the display driver to refresh the display.
   - If the area was larger than the buffer, redraw the remaining parts too.
   - Do the same with all the joined areas.

While an area is redrawn, the library searches the most top object which covers the area to redraw, and starts to draw from that object. For example, if a button's label has changed, the library will see that it's enough to draw the button under the text, and it's not required to draw the background too.

The difference between buffer types regarding the drawing mechanism is the following:

1. **One buffer** - LVGL needs to wait for `lv_disp_flush_ready()` (called at the end of `flush_cb`) before starting to redraw the next part.
2. **Two non-screen-sized buffers** - LVGL can immediately draw to the second buffer when the first is sent to `flush_cb` because the flushing should be done by DMA (or similar hardware) in the background.