LVGL Documentation v7.0.2

Contributors of LVGL

Jun 17, 2020

CONTENTS

Documentation of V6

PDF version: `LVGL.pdf`

Website • **GitHub** • **Forum** • **Live demo** • ***Simulator*** • **Blog**

CHAPTER

ONE

INTRODUCTION

LVGL (Light and Versatile Graphic 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:

- 16, 32 or 64 bit microcontroller or processor
- > 16 MHz clock speed is recommended

- Flash/ROM: > 64 kB for the very essential components (> 180 kB is recommended)
- RAM: - Static RAM usage: ~2 kB depending on the used features and objects types - Stack: > 2kB (> 8 kB is recommended) - Dynamic data (heap): > 2 KB (> 16 kB is recommended if using several objects). Set by `LV_MEM_SIZE` in *lv_conf.h*. - Display buffer: > *"Horizontal resolution"* pixels (> 10 × *"Horizontal resolution"* is recommended)
- One frame buffer in the MCU or in external display controller
- C99 or newer compiler
- Basic C (or C++) knowledge: pointers, structs, callbacks

Note that the memory usage might vary depending on the architecture, compiler and build options.

1.3 License

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

The only thing you need to do is to add a notice about you are using LVGL in your product. This notice can be placed in the user guide, on your website, on an about screen, or anywhere else where users might see it.

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 a monthly periodic release cycle.

- **1st Tuesday of the month** - Make a major, minor, or patch release from `master` depending on the new features. - After that merge only patches into `master` and add new features into the `dev`.
- **3rd Tuesday of the month** - Make a patch release from `master`. - After that merge the new features from the `dev` to `master` branch. - In the rest of the month merge only patches into `master` and new features into `dev` branch.

In other words, patches are merged directly into `master` and new features into `dev`. `dev` is merged to `master` in the middle of the month and the new features are released at the beginning of the next month.

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

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_templ.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.
- 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)
{
    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 is automatically get the `LV_STATE_PRESSED` state and when you release is it will be removed.

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

Styles

Styles can be assigned to the parts objects to changed 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 is you can set 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 nt 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

Styling buttons

Slider and alignment

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.

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

TODO

2.5 Espressif (ESP32)

2.6 Arduino

TODO

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: examples don't work with v7 yet, so v6 is used.

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?

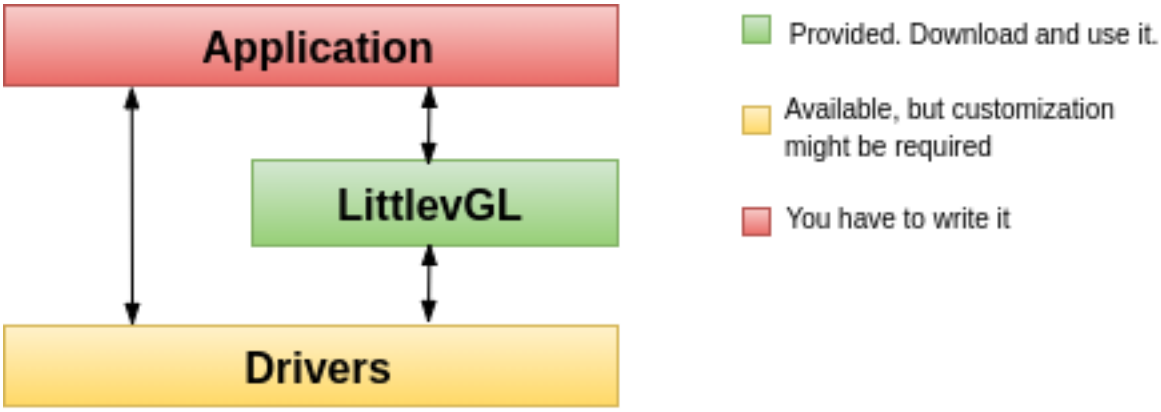
- On the Blog Post
- On `lv_micropython` README
- On `lv_binding_micropython` README
- On LVGL forum (Feel free to ask anything!)
- On Micropython docs and forum

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.

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.

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** if `1` swap `hor_res` and `ver_res`. LVGL draws in the same direction in both cases (in lines from top to bottom) so the driver also needs to be reconfigured to change the display's fill direction.
- **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.

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.

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)
{
    /*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);
}

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);
{
```

(continues on next page)

(continued from previous page)

```
    /*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));
}

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);
}
```

3.3.3 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_size_t

Values:

LV_DISP_SIZE_SMALL

LV_DISP_SIZE_MEDIUM

LV_DISP_SIZE_LARGE

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.

Return pointer to the new display or NULL on error

Parameters

- `driver`: pointer to an initialized 'lv_disp_drv_t' variable (can be local variable)

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

Return pointer to the default display

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

Return the horizontal resolution of the display

Parameters

- `disp`: pointer to a display (NULL to use the default display)

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

Return the vertical resolution of the display

Parameters

- `disp`: pointer to a display (NULL to use the default display)

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

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

Parameters

- `disp`: pointer to a display (NULL to use the default display)

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

Return dpi of the display

Parameters

- `disp`: pointer to a display (NULL to use the default 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.

Return LV_DISP_SIZE_SMALL/MEDIUM/LARGE/EXTRA_LARGE

Parameters

- `disp`: pointer to a display (NULL to use the default display)

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

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

Parameters

- `disp`: pointer to the current display. NULL to initialize.

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

Return pointer to the internal buffers

Parameters

- `disp`: pointer to a display

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

Return 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)

Return true: double buffered; false: not double buffered

Parameters

- `disp`: pointer to to display to check

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)

Return true: double buffered; false: not double buffered

Parameters

- `disp`: pointer to to display to check

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**

volatile int **flushing**

volatile int **flushing_last**

volatile uint32_t **last_area**

volatile 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**
1: turn the display by 90 degree.

Warning Does not update coordinates for you!

uint32_t **screen_transp**
Handle if the 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** *_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** *_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** *_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** *_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** *_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 (***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 *top_layer

See *lv_disp_get_layer_top*

struct _lv_obj_t *sys_layer

See *lv_disp_get_layer_sys*

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 standar encoder behavior, you can also utilise its logic to navigate(focus) and edit widgets using buttons. This is especialy 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*/
```

(continues on next page)

(continued from previous page)

```
    } 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:

LV_INDEV_TYPE_NONE
Uninitialized state

LV_INDEV_TYPE_POINTER
Touch pad, mouse, external button

LV_INDEV_TYPE_KEYPAD
Keypad or keyboard

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

LV_INDEV_TYPE_ENCODER
Encoder with only Left, Right turn and a Button

enum [anonymous]
States for input devices

Values:

LV_INDEV_STATE_REL = 0

LV_INDEV_STATE_PR

enum [anonymous]
Values:

LV_DRAG_DIR_HOR = 0x1
Object can be dragged horizontally.

LV_DRAG_DIR_VER = 0x2
Object can be dragged vertically.

LV_DRAG_DIR_BOTH = 0x3
Object can be dragged in all directions.

LV_DRAG_DIR_ONE = 0x4
Object can be dragged only one direction (the first move).

enum [anonymous]
Values:

LV_GESTURE_DIR_TOP
Gesture dir up.

LV_GESTURE_DIR_BOTTOM
Gesture dir down.

LV_GESTURE_DIR_LEFT
Gesture dir left.

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.

Return pointer to the new input device or NULL on error

Parameters

- `driver`: pointer to an initialized 'lv_indev_drv_t' variable (can be local variable)

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.

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

Parameters

- `indev`: pointer to the current input device. NULL to initialize.

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

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

Parameters

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

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

Return the elapsed milliseconds

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

Get the elapsed milliseconds since a previous time stamp

Return the elapsed milliseconds since 'prev_tick'

Parameters

- `prev_tick`: a previous time stamp (return value of systick_get())

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, int 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);
```

(continues on next page)

(continued from previous page)

```
  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 Object 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 Object's 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.

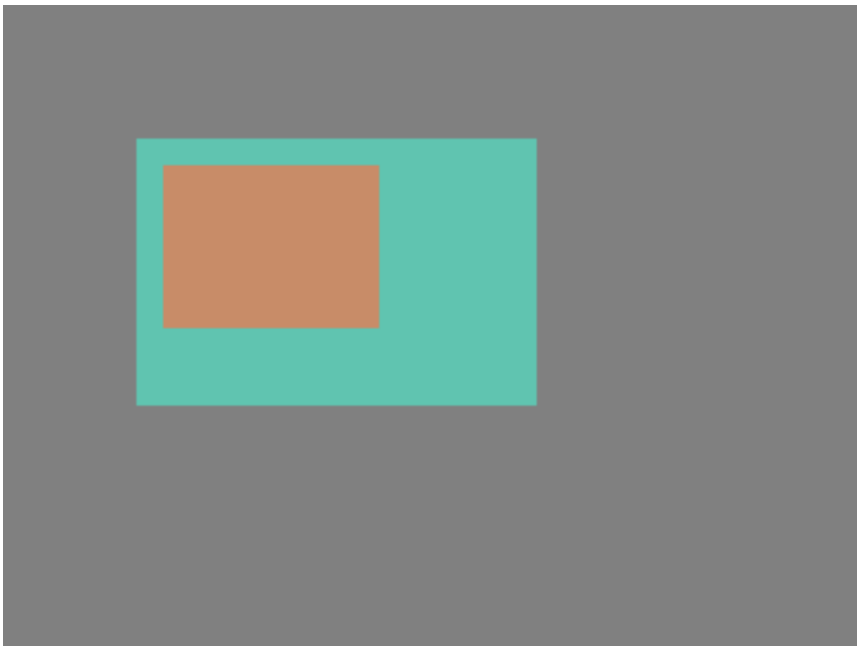
```
lv_obj_t * par = lv_obj_create(lv_scr_act(), NULL); /*Create a parent object on the␣
↪current screen*/
lv_obj_set_size(par, 100, 80);                          /*Set the size of the␣
↪parent*/
```

(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);
```

Screen – the most basic parent

The screens are special objects which have no parent object. So it is created like:

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

There is always an active screen on each display. By default, the library creates and loads a "Base object" as the screen for each display.To get the currently active screen use the `lv_scr_act()` function. To load a new one, use `lv_scr_load(scr1)`.

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

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

Visit *Multi-display support* to learn more.

4.1.3 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.

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)`.

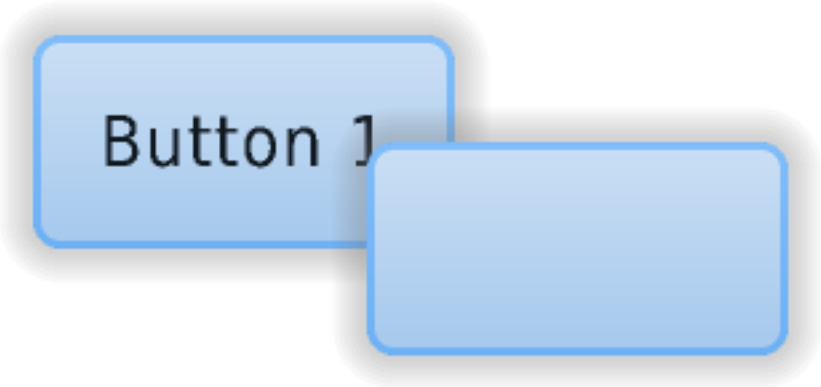
To learn more about the states read the related section of the Style overview.

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*/
```

(continues on next page)

(continued from previous page)

```
/*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 using for similar purpose on LVGL. For example, it places the mouse cursor there 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)
{
```

(continues on next page)

(continued from previous page)

```
    switch(event) {
        case LV_EVENT_PRESSED:
            printf("Pressed\n");
            break;

        case LV_EVENT_SHORT_CLICKED:
            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
- **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

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);
```

Or to perform refresh generically:

```
lv_event_send(label, LV_EVENT_REFRESH, NULL);
```

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

![page_small.png](A scrollable page in lvgl)

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 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 state functions with the following prototype are available: `lv_style_get_<prperty_name>(&style, <state>, <result poiner>);`. The the best matching property will be selected and it's precedence will be returned. `-1` will be returned if the property is not found. For example:

```
lv_color_t color;
int16_t res;
res = lv_style_get_bg_color(&style1, LV_STATE_PRESSED, &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)`

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.
- **clip_corner** (`bool`): `true`: enable to clip the overflowed content on the rounded (radius > 0) corners.
- **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.
- **transform_width** (`lv_style_int_t`): Make the object wider on both sides with this value.
- **transform_height** (`lv_style_int_t`) Make the object higher on both sides with this value.

- **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.

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

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.
- **bg_grad_stop** (`uint8_t`): Specifies where should the gradient start. 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`.

Border properties

The border in drawn on to of the *background*. It has `radius` rounding.

- **border_color** (`lv_color_t`) Specifies the color of the border.
- **border_opa** (`lv_opa_t`) Specifies opacity of the border.
- **border_width** (`lv_style_int_t`): Set the width of the border.

- **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 all children has been drawn.
- **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`.

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.
- **outline_opa** (`lv_opa_t`) Specifies opacity of the outline.
- **outline_width** (`lv_style_int_t`): Set the width of the outline.
- **outline_pad** (`lv_style_int_t`) Set the space between the object and the outline.
- **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`.

Shadow properties

The shadow is a blurred area under the object.

- **shadow_color** (`lv_color_t`) Specifies the color of the shadow.
- **shadow_opa** (`lv_opa_t`) Specifies opacity of the shadow.
- **shadow_width** (`lv_style_int_t`): Set the width (blur size) of the outline.
- **shadow_ofs_x** (`lv_style_int_t`): Set the an X offset for the shadow.
- **shadow_ofs_y** (`lv_style_int_t`): Set the an Y offset for the shadow.
- **shadow_spread** (`lv_style_int_t`): ake the shadow larger than the background in every direction by this value.
- **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`.

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.
- **pattern_opa** (`lv_opa_t`): Specifies opacity of the pattern.
- **pattern_recolor** (`lv_color_t`): Mix this color to the pattern image. In case of symbols (texts) it will be the text color.
- **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.
- **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`.

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.
- **value_color** (`lv_color_t`): Color of the text.
- **value_opa** (`lv_opa_t`): Opacity of the text.
- **value_font** (`const lv_font_t *`): Pointer to font of the text.
- **value_letter_space** (`lv_style_int_t`): Letter space of the text.
- **value_line_space** (`lv_style_int_t`): Line space of the text.
- **value_align** (`lv_align_t`): Alignment of the text. Can be `LV_ALIGN_...`.
- **value_ofs_x** (`lv_style_int_t`): X offset from the original position of the alignment.
- **value_ofs_y** (`lv_style_int_t`): Y offset from the original position of the alignment.
- **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`.

Text properties

Properties for textual object.

- **text_color** (`lv_color_t`): Color of the text.
- **text_opa** (`lv_opa_t`): Opacity of the text.
- **text_font** (`const lv_font_t *`): Pointer to font of the text.
- **text_letter_space** (`lv_style_int_t`): Letter space of the text.
- **text_line_space** (`lv_style_int_t`): Line space of the text.
- **text_decor** (`lv_text_decor_t`): Add text decoration. Can be `LV_TEXT_DECOR_NONE/UNDERLINE/STRIKETHROUGH`.
- **text_sel_color** (`lv_color_t`): Set background color of text selection.
- **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`.

Line properties

Properties of lines.

- **line_color** (`lv_color_t`): Color of the line.
- **line_opa** (`lv_opa_t`): Opacity of the line.
- **line_width** (`lv_style_int_t`): Width of the line.
- **line_dash_width** (`lv_style_int_t`): Width of dash. Dashing is drawn only for horizontal or vertical lines. 0: disable dash.
- **line_dash_gap** (`lv_style_int_t`): Gap between two dash line. Dashing is drawn only for horizontal or vertical lines. 0: disable dash.
- **line_rounded** (`bool`): `true`: draw rounded line endings.
- **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`.

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.
- **image_recolor_opa** (`lv_opa_t`): Intensity of recoloring. Default value: `LV_OPA_TRANSP` (no recoloring).
- **image_opa** (`lv_opa_t`): Opacity of the image.
- **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`.

Transition properties

Properties to describe state change animations.

- **transition_time** (`lv_style_int_t`): Time of the transition.
- **transition_delay** (`lv_style_int_t`): Delay before the transition.
- **transition_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`
- **transition_2** (`property name`): Same as *transition_1* just for an other property.
- **transition_3** (`property name`): Same as *transition_1* just for an other property.
- **transition_4** (`property name`): Same as *transition_1* just for an other property.
- **transition_5** (`property name`): Same as *transition_1* just for an other property.
- **transition_6** (`property name`): Same as *transition_1* just for an other property.
- **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).

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.
- **scale_end_color** (`lv_color_t`): Color of the scale lines in the end region.
- **scale_width** (`lv_style_int_t`): Width of the scale. 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.
- **scale_end_border_width** (`lv_style_int_t`): Width of a border drawn on the outer side of the scale in the end region.
- **scale_end_line_width** (`lv_style_int_t`): Width of a scale lines in the end region.

It 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

4.4.11 Example

Styling a button

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)`.

The 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 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, 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 object where short click has meaning (e.g. *Button*), 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 te object goes to edit mode it goes to `LV_STATE_FOCUSED | LV_STATE_EDITED` state so these style properties will be shown.

For a moew 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.

Return 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

Return the type of the input device from `lv_hal_indev_type_t` (`LV_INDEV_TYPE_...`)

Parameters

- `indev`: pointer to an input device

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

Return current gesture direct

Parameters

- `indev`: pointer to an input device

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)

Return the last pressed key (0 on error)

Parameters

- `indev`: pointer to an input device

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)

Return true: drag is in progress

Parameters

- `indev`: pointer to an input device

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.

Return `LV_RES_INV` if the object being dragged was deleted. Else `LV_RES_OK`.

Parameters

- `indev`: pointer to an input device

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.

Return 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

Return pointer to the found object or NULL if there was no suitable object

Parameters

- `obj`: pointer to a start object, typically the screen
- `point`: pointer to a point for searching the most top child

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.

Return pointer to the indev read refresher task. (NULL on error)

Parameters

- `indev`: pointer to an inout device

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:

LV_KEY_UP = 17

LV_KEY_DOWN = 18

LV_KEY_RIGHT = 19

LV_KEY_LEFT = 20

LV_KEY_ESC = 27

LV_KEY_DEL = 127

LV_KEY_BACKSPACE = 8

LV_KEY_ENTER = 10

LV_KEY_NEXT = 9

LV_KEY_PREV = 11

LV_KEY_HOME = 2

LV_KEY_END = 3

enum [anonymous]

Values:

LV_GROUP_REFOCUS_POLICY_NEXT = 0

LV_GROUP_REFOCUS_POLICY_PREV = 1

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

Return 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

Return result of focused object in group.

Parameters

- `group`: pointer to a group
- `c`: a character (use LV_KEY_.. to navigate)

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

Return pointer to the focused object

Parameters

- `group`: pointer to a group

lv_group_user_data_t ***`lv_group_get_user_data`**(*lv_group_t *group*)
Get a pointer to the group's user data

Return pointer to the user data

Parameters

- `group`: pointer to an group

lv_group_focus_cb_t **lv_group_get_focus_cb(const** *lv_group_t *group***)**
Get the focus callback function of a group

Return the call back function or NULL if not set

Parameters

- `group`: pointer to a group

bool **lv_group_get_editing(const** *lv_group_t *group***)**
Get the current mode (edit or navigate).

Return true: edit mode; false: navigate mode

Parameters

- `group`: pointer to group

bool **lv_group_get_click_focus(const** *lv_group_t *group***)**
Get the `click_focus` attribute.

Return true: `click_focus` is enabled; false: disabled

Parameters

- `group`: pointer to group

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.

In LVGL, you can have multiple displays, each with their own driver and objects.

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 which display to create objects on.

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.

4.6.1 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*).

4.6.2 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.

4.6.3 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.

4.6.4 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)`.

Screens can be deleted with `lv_obj_del(scr)`, but ensure that you do not delete the currently loaded screen.

Opaque screen

Usually, the opacity of the screen is `LV_OPA_COVER` to provide a solid background for its children.

However, in some special cases, you might want a transparent screen. For example, if you have a video player that renders video frames on a lower layer, you want to create an OSD menu on the upper layer (over the video) using LVGL.

To do this, the screen should have a style that sets `body.opa` or `image.opa` to `LV_OPA_TRANSP` (or another non-opaque value) to make the screen opaque.

Also, `LV_COLOR_SCREEN_TRANSP` needs to be enabled. Please note that it only works with `LV_COLOR_DEPTH = 32`.

The Alpha channel of 32-bit colors will be 0 where there are no objects and will be 255 where there are solid objects.

4.6.5 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**).

4.6.6 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

The color module defines the most basic colors such as:

- #FFFFFF `LV_COLOR_WHITE`
- #000000 `LV_COLOR_BLACK`
- #808080 `LV_COLOR_GRAY`
- #c0c0c0 `LV_COLOR_SILVER`
- #ff0000 `LV_COLOR_RED`
- #800000 `LV_COLOR_MAROON`
- #00ff00 `LV_COLOR_LIME`

- #008000 `LV_COLOR_GREEN`
- #808000 `LV_COLOR_OLIVE`
- #0000ff `LV_COLOR_BLUE`
- #000080 `LV_COLOR_NAVY`
- #008080 `LV_COLOR_TEAL`
- #00ffff `LV_COLOR_CYAN`
- #00ffff `LV_COLOR_AQUA`
- #800080 `LV_COLOR_PURPLE`
- #ff00ff `LV_COLOR_MAGENTA`
- #ffa500 `LV_COLOR_ORANGE`
- #ffff00 `LV_COLOR_YELLOW`

as well as `LV_COLOR_WHITE` (fully white).

4.6.7 API

Display

Functions

lv_obj_t ***lv_disp_get_scr_act**(*lv_disp_t *disp*)

Return with a pointer to the active screen

Return pointer to the active screen object (loaded by 'lv_scr_load()')

Parameters

- `disp`: pointer to display which active screen should be get. (NULL to use the default screen)

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)

Return pointer to the top layer object (transparent screen sized lv_obj)

Parameters

- `disp`: pointer to display which top layer should be get. (NULL to use the default screen)

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)

Return pointer to the sys layer object (transparent screen sized lv_obj)

Parameters

- `disp`: pointer to display which sys. layer should be get. (NULL to use the default screen)

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

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)

Return elapsed ticks (milliseconds) since the last activity

Parameters

- `disp`: pointer to an display (NULL to get the overall smallest inactivity)

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)

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.

Return pointer to the display refresher task. (NULL on error)

Parameters

- `disp`: pointer to a display

static *lv_obj_t* ***lv_scr_act**(void)
Get the active screen of the default display

Return pointer to the active screen

static *lv_obj_t* ***lv_layer_top**(void)
Get the top layer of the default display

Return pointer to the top layer

static *lv_obj_t* ***lv_layer_sys**(void)
Get the active screen of the default display

Return pointer to the sys layer

static void **lv_scr_load**(*lv_obj_t *scr*)

Colors

Typedefs

typedef uint32_t **lv_color_int_t**

typedef *lv_color32_t* **lv_color_t**

Enums

enum [anonymous]
Opacity percentages.

Values:

LV_OPA_TRANSP = 0

LV_OPA_0 = 0

LV_OPA_10 = 25

LV_OPA_20 = 51

LV_OPA_30 = 76

LV_OPA_40 = 102

LV_OPA_50 = 127

LV_OPA_60 = 153

LV_OPA_70 = 178

LV_OPA_80 = 204

LV_OPA_90 = 229

LV_OPA_100 = 255

LV_OPA_COVER = 255

Functions

static uint8_t **lv_color_to1**(*lv_color_t color*)

static uint8_t **lv_color_to8**(*lv_color_t color*)

static uint16_t **lv_color_to16**(*lv_color_t color*)

static uint32_t **lv_color_to32**(*lv_color_t color*)

static uint8_t **lv_color_brightness**(*lv_color_t color*)
Get the brightness of a color

Return the brightness [0..255]

Parameters

- `color`: a color

static *lv_color_t* **lv_color_make**(uint8_t *r*, uint8_t *g*, uint8_t *b*)

static *lv_color_t* **lv_color_hex**(uint32_t *c*)

static *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

Return the given RGB color in RGB (with LV_COLOR_DEPTH depth)

Parameters

- `h`: hue [0..359]
- `s`: saturation [0..100]
- `v`: value [0..100]

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

Return the given RGB color in HSV

Parameters

- `r8`: 8-bit red
- `g8`: 8-bit green
- `b8`: 8-bit blue

lv_color_hsv_t **`lv_color_to_hsv`**(*lv_color_t color*)

Convert a color to HSV

Return the given color in HSV

Parameters

- `color`: color

union `lv_color1_t`

Public Members

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:

- `LV_FONT_MONTSERRAT_12` 12 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_14` 14 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_16` 16 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_18` 18 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_20` 20 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_22` 22 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_24` 24 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_26` 26 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_28` 28 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_30` 30 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_32` 32 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_34` 34 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_36` 36 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_38` 38 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_40` 40 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_42` 42 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_44` 44 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_46` 46 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_48` 48 px ASCII + built-in symbol
- `LV_FONT_MONTSERRAT_12_SUBPX` 12 px font with *subpixel rendering*
- `LV_FONT_MONTSERRAT_28_COMPRESSED` 28 px *compressed font* with 3 bpp
- `LV_FONT_DEJAVU_16_PERSIAN_HEBREW` 16 px Hebrew, Arabic, Perisan letters and all their forms
- `LV_FONT_SIMSUN_16_CJK` 16 px 1000 most common CJK radicals
- `LV_FONT_UNSCII_8` 8 px pixel perfect font

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.

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

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:

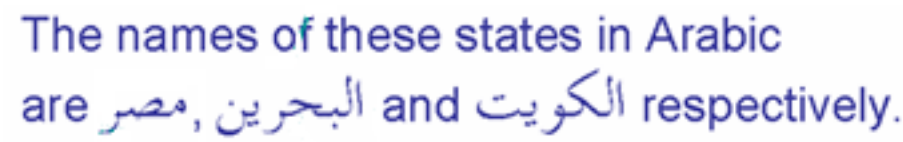
```
lv_label_set_text(my_label, LV_SYMBOL_OK LV_SYMBOL_WIFI LV_SYMBOL_PLAY);
```

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 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 raedy 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 ...
     */
    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` */
  ...

  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)
```

(continues on next page)

(continued from previous page)

```
 * @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` */

}

/**
 * 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_img_decoder_open()`.

```
lv_res_t res;
lv_img_decoder_dsc_t dsc;
res = lv_img_decoder_open(&dsc, &my_img_dsc, &lv_style_plain);

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`

Return LV_RES_OK: info written correctly; LV_RES_INV: failed

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

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`, `style` 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.

Return LV_RES_OK: ok; LV_RES_INV: failed

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

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:

LV_IMG_SRC_VARIABLE

LV_IMG_SRC_FILE

Binary/C variable

LV_IMG_SRC_SYMBOL

File in filesystem

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.

Return LV_RES_OK: success; LV_RES_INV: wasn't able to get info about the image

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

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`

Return 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.

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`
- `style`: the style of 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

Return LV_RES_OK: success; LV_RES_INV: an error occurred

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

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

Return 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

Return LV_RES_OK: the info is successfully stored in `header`; LV_RES_INV: unknown format or other error.

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

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

Return LV_RES_OK: the info is successfully stored in `header`; LV_RES_INV: unknown format or other error.

Parameters

- `decoder`: the decoder where this function belongs
- `dsc`: pointer to decoder descriptor. `src`, `style` are already initialized in it.

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.

Return LV_RES_OK: ok; LV_RES_INV: failed

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

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.

Return pointer to the cache entry or NULL if can open the image

Parameters

- `src`: source of the image. Path to file or pointer to an `lv_img_dsc_t` variable
- `style`: style of 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_tio_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

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 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;
```

(continues on next page)

(continued from previous page)

```
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:

LV_FS_RES_OK = 0

LV_FS_RES_HW_ERR

LV_FS_RES_FS_ERR

LV_FS_RES_NOT_EX

LV_FS_RES_FULL

LV_FS_RES_LOCKED

LV_FS_RES_DENIED

LV_FS_RES_BUSY

LV_FS_RES_TOUT

LV_FS_RES_NOT_IMP

LV_FS_RES_OUT_OF_MEM

LV_FS_RES_INV_PARAM

LV_FS_RES_UNKNOWN

enum [anonymous]

Filesystem mode.

Values:

LV_FS_MODE_WR = 0x01

LV_FS_MODE_RD = 0x02

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

Return pointer to a driver or NULL if not found

Parameters

- `letter`: the driver letter

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.

Return true: drive is ready; false: drive is not ready

Parameters

- `letter`: letter of the drive

lv_fs_res_t **lv_fs_open**(*lv_fs_file_t* **file_p*, **const** char **path*, *lv_fs_mode_t mode*)
Open a file

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

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

lv_fs_res_t **lv_fs_close**(*lv_fs_file_t* **file_p*)
Close an already opened file

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `file_p`: pointer to a *lv_fs_file_t* variable

lv_fs_res_t **lv_fs_remove**(**const** char **path*)
Delete a file

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `path`: path of the file to delete

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

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

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.

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

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

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.

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

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `file_p`: pointer to a *lv_fs_file_t* variable
- `pos`: the new position expressed in bytes index (0: start of file)

lv_fs_res_t **lv_fs_tell(***lv_fs_file_t* ****file_p***, uint32_t ****pos***)**
Give the position of the read write pointer

Return LV_FS_RES_OK or any error from 'fs_res_t'

Parameters

- `file_p`: pointer to a *lv_fs_file_t* variable
- `pos_p`: pointer to store the position of the read write pointer

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

Return LV_FS_RES_OK: no error, the file is read any error from lv_fs_res_t enum

Parameters

- `file_p`: pointer to an 'ufs_file_t' variable. (opened with lv_fs_open)

lv_fs_res_t **lv_fs_size(***lv_fs_file_t* ****file_p***, uint32_t ****size***)**
Give the size of a file bytes

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `file_p`: pointer to a *lv_fs_file_t* variable
- `size`: pointer to a variable to store the size

lv_fs_res_t **lv_fs_rename(const** char ****oldname***, **const** char ****newname***)**
Rename a file

Return LV_FS_RES_OK or any error from 'fs_res_t'

Parameters

- `oldname`: path to the file
- `newname`: path with the new name

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

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `rddir_p`: pointer to a 'fs_read_dir_t' variable
- `path`: path to a directory

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 '/'

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `rddir_p`: pointer to an initialized 'fs_rdir_t' variable
- `fn`: pointer to a buffer to store the filename

lv_fs_res_t **lv_fs_dir_close**(*lv_fs_dir_t *rddir_p*)
Close the directory reading

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `rddir_p`: pointer to an initialized 'fs_dir_t' variable

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

Return LV_FS_RES_OK or any error from lv_fs_res_t enum

Parameters

- `letter`: the driver letter
- `total_p`: pointer to store the total size [kB]
- `free_p`: pointer to store the free size [kB]

char ***lv_fs_get_letters**(char **buf*)
Fill a buffer with the letters of existing drivers

Return the buffer

Parameters

- `buf`: buffer to store the letters ('\0' added after the last letter)

const char ***lv_fs_get_ext**(**const** char **fn*)
Return with the extension of the filename

Return pointer to the beginning extension or empty string if no extension

Parameters

- `fn`: string with a filename

char ***lv_fs_up**(char **path*)
Step up one level

Return the truncated file name

Parameters

- `path`: pointer to a file name

const char ***lv_fs_get_last(const** char **path*)

Get the last element of a path (e.g. U:/folder/file -> file)

Return pointer to the beginning of the last element in the path

Parameters

- `buf`: buffer to store the letters ('\0' added after the last letter)

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 and variable to animate*/
lv_anim_set_exec_cb(&a, btn1, (lv_anim_exec_xcb_t) lv_obj_set_x);

/*Length of the animation [ms]*/
lv_anim_set_time(&a, duration);

/*Set start and end values. E.g. 0, 150 [ms]*/
lv_anim_set_values(&a, start, end);

/* OPTIONAL SETTINGS
 *------------------*/
```

(continues on next page)

(continued from previous page)

```
/*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);

/*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);
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:

LV_ANIM_OFF

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 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 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 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 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 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 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 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_cb`: a function the get the current value of the animation. The built in functions starts with `lv_anim_path_...`

static 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 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 void **lv_anim_set_playback_time**(*lv_anim_t *a*, uint16_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 void **lv_anim_set_playback_delay**(*lv_anim_t *a*, uint16_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 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 void **lv_anim_set_repeat_delay**(*lv_anim_t *a*, uint16_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 void **lv_anim_path_init**(*lv_anim_path_t *path*)
Initialize an animation path

Parameters

- `path`: pointer to path

static 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 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 int32_t **lv_anim_get_delay**(*lv_anim_t *a*)
Get a delay before starting the animation

Return delay before the animation in milliseconds

Parameters

- `a`: pointer to an initialized `lv_anim_t` variable

bool **lv_anim_del**(void **var*, *lv_anim_exec_xcb_t exec_cb*)
Delete an animation of a variable with a given animator function

Return true: at least 1 animation is deleted, false: no animation is deleted

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

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`.

Return pointer to the animation.

Parameters

- `var`: pointer to variable
- `exec_cb`: a function pointer which is animating 'var', or NULL to delete all the animations of 'var'

static 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.

Return true: at least 1 animation is deleted, false: no animation is deleted

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

uint16_t **lv_anim_count_running**(void)

Get the number of currently running animations

Return the number of running animations

uint16_t **lv_anim_speed_to_time**(uint16_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

Return the required time [ms] for the animation with the given parameters

Parameters

- `speed`: speed of animation in unit/sec
- `start`: start value of the animation
- `end`: end value of the animation

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

Return the current value to set

Parameters

- `a`: pointer to an animation

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

Return the current value to set

Parameters

- a: pointer to an animation

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

Return the current value to set

Parameters

- a: pointer to an animation

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)

Return the current value to set

Parameters

- a: pointer to an animation

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

Return the current value to set

Parameters

- a: pointer to an animation

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

Return the current value to set

Parameters

- a: pointer to an animation

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)

Return the current value to set

Parameters

- a: pointer to an animation

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_path_t **path**
Describe the path (curve) of animations

int32_t **start**
Start 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

uint32_t **time_orig**

uint8_t **playback_now**
Play back is in progress

uint32_t **has_run**
Indicates the animation has run in this round

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

Typedefs

typedef void (*__lv_task_cb_t)(**struct** *__lv_task_t* *)
Tasks execute this type 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:

LV_TASK_PRIO_OFF = 0

LV_TASK_PRIO_LOWEST

LV_TASK_PRIO_LOW

LV_TASK_PRIO_MID

LV_TASK_PRIO_HIGH

LV_TASK_PRIO_HIGHEST

_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`

Return 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

Return pointer to the new 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

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

Return the lv_task idle in percentage

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

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.
3. **Two screen-sized buffers** - After calling `flush_cb`, the first buffer, if being displayed as frame buffer. Its content is copied to the second buffer and all the changes are drawn on top of it.

4.12.3 Masking

Masking is the basic concept of LVGL's drawing engine. To use LVGL it's not required to know about the mechanisms described here, but you might find interesting to know how the drawing works under hood.

To learn masking let's learn the steps of drawing first:

1. Create a draw descriptor from an object's styles (e.g. `lv_draw_rect_dsc_t`). It tells the parameters of drawing, for example the colors, widths, opacity, fonts, radius, etc.
2. Call the draw function with the initialized descriptor and some other parameters. It renders the primitive shape to the current draw buffer.
3. If the shape is very simple and doesn't require masks go to #5. Else create the required masks (e.g. a rounded rectangle mask)
4. Apply all the created mask(s) for one or a few lines. It create 0..255 values into a *mask buffer* with the "shape" of the created masks. E.g. in case of a "line mask" according to the parameters of the mask, keep one side of the buffer as it is (255 by default) and set the rest to 0 to indicate that the latter side should be removed.
5. Blend the image or rectangle to the screen. During blending masks (make some pixels transparent or opaque), blending modes (additive, subtractive, etc), opacity are handled.
6. Repeat from #4.

Masks are used the create almost every basic primitives:

- **letters** create a mask from the letter and draw a "letter-colored" rectangle using the mask.

- **line** created from 4 l"ine masks", to mask out the left, right, top and bottom part of the line to get perfectly perpendicular line ending
- **rounded rectangle** a mask is created real-time for each line of a rounded rectangle and a normal filled rectangle is drawn according to the mask.
- **clip corner** to clip to overflowing content on the rounded corners also a rounded rectangle mask is applied.
- **rectangle border** same as a rounded rectangle, but inner part is masked out too
- **arc drawing** a circle border is drawn, but an arc mask is applied.
- **ARGB images** the alpha channel is separated into a mask and the image is drawn as a normal RGB image.

As mentioned in #3 above in some cases no mask is required:

- a mono colored, not rounded rectangles
- RGB images

LVGL has the following built-in mask types which can be calculated and applied real-time:

- **LV_DRAW_MASK_TYPE_LINE** Removes a side of a line (top, bottom, left or right). `lv_draw_line` uses 4 of it. Essentially, every (skew) line is bounded with 4 line masks by forming a rectangle.
- **LV_DRAW_MASK_TYPE_RADIUS** Removes the inner or outer parts of a rectangle which can have radius too. It's also used to create circles by setting the radius to large value (`LV_RADIUS_CIRCLE`)
- **LV_DRAW_MASK_TYPE_ANGLE** Removes a circle sector. It is used by `lv_draw_arc` to remove the "empty" sector.
- **LV_DRAW_MASK_TYPE_FADE** Create a vertical fade (change opacity)
- **LV_DRAW_MASK_TYPE_MAP** The mask is stored in an array and the necessary parts are applied

Masks are create and removed automatically during drawing but the *lv_objmask* allows the user to add masks. Here is an example:

CHAPTER

FIVE

WIDGETS

5.1 Base object (lv_obj)

5.1.1 Overview

The 'Base Object' implements the basic properties of widgets on a screen, such as:

- coordinates
- parent object
- children
- main style
- attributes like *Click enable*, *Drag enable*, etc.

In object-oriented thinking, it is the base class from which all other objects in LVGL are inherited. This, among another things, helps reduce code duplication.

The functions and functionalities of Base object can be used with other widgets too. For example `lv_obj_set_width(slider, 100)`

The Base object can be directly used as a simple widgets. It nothing else then a rectangle.

Coordinates

Size

The object size can be modified on individual axes with `lv_obj_set_width(obj, new_width)` and `lv_obj_set_height(obj, new_height)`, or both axes can be modified at the same time with `lv_obj_set_size(obj, new_width, new_height)`.

Styles can add Margin to the objects. Margin tells that "I want this space around me". To set width or height reduced by the margin `lv_obj_set_width_margin(obj, new_width)` or `lv_obj_set_height_margin(obj, new_height)`. In more exact way: `new_width = left_margin + object_width + right_margin`.

To get the width or height which includes the margins use `lv_obj_get_width/height_margin(obj)`.

Styles can add Padding to the object as well. Padding means "I don't want my children too close to my sides, so keep this space". To set width or height reduced by the padding `lv_obj_set_width_fit(obj, new_width)` or `lv_obj_set_height_fit(obj, new_height)`.In a more exact way: `new_width = left_pad + object_width + right_pad` To get the width or height which is REDUCED by padding use `lv_obj_get_width/height_fit(obj)`. It can be considered the "useful size of the object".

Margin and padding gets important when Layout or Auto-fit is used by other widgets.

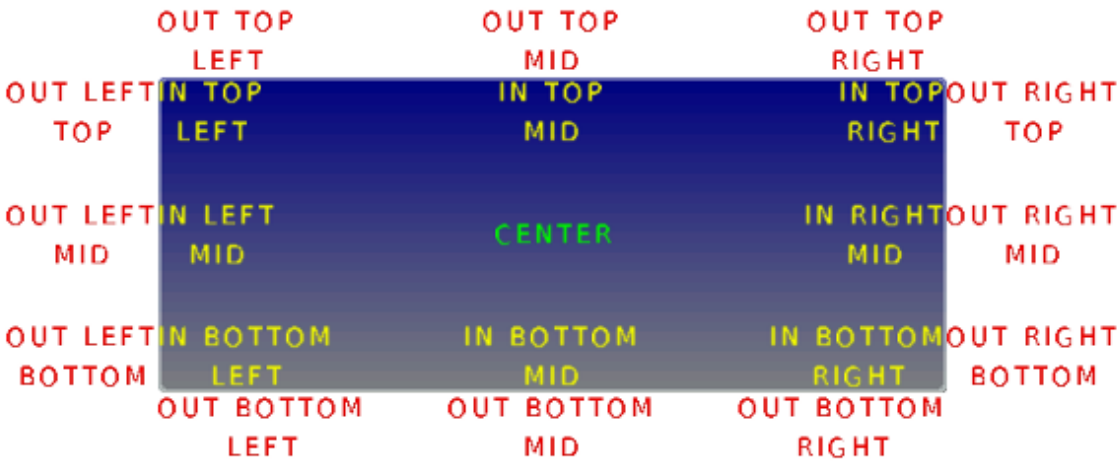
Position

You can set the x and y coordinates relative to the parent with `lv_obj_set_x(obj, new_x)` and `lv_obj_set_y(obj, new_y)`, or both at the same time with `lv_obj_set_pos(obj, new_x, new_y)`.

Alignment

You can align the object to another with `lv_obj_align(obj, obj_ref, LV_ALIGN_..., x_ofs, y_ofs)`.

- `obj` is the object to align.
- `obj_ref` is a reference object. `obj` will be aligned to it. If `obj_ref = NULL`, then the parent of `obj` will be used.
- The third argument is the *type* of alignment. These are the possible options:



  The alignment types build like `LV_ALIGN_OUT_TOP_MID`.
- The last two arguments allow you to shift the object by a specified number of pixels after aligning it.

For example, to align a text below an image: `lv_obj_align(text, image, LV_ALIGN_OUT_BOTTOM_MID, 0, 10)`. Or to align a text in the middle of its parent: `lv_obj_align(text, NULL, LV_ALIGN_CENTER, 0, 0)`.

`lv_obj_align_origo` works similarly to `lv_obj_align` but it aligns the center of the object.

For example, `lv_obj_align_origo(btn, image, LV_ALIGN_OUT_BOTTOM_MID, 0, 0)` will align the center of the button the bottom of the image.

The parameters of the alignment will be saved in the object if `LV_USE_OBJ_REALIGN` is enabled in *lv_conf.h*. You can then realign the objects simply by calling `lv_obj_realign(obj)`. It's equivalent to calling `lv_obj_align` again with the same parameters.

If the alignment happened with `lv_obj_align_origo`, then it will be used when the object is realigned.

If `lv_obj_set_auto_realign(obj, true)` is used the object will be realigned automatically, if its size changes in `lv_obj_set_width/height/size()` functions. It's very useful when size animations are applied to the object and the original position needs to be kept.

Note that the coordinates of screens can't be changed. Attempting to use these functions on screens will result in undefined behavior.

Parents and children

You can set a new parent for an object with `lv_obj_set_parent(obj, new_parent)`. To get the current parent, use `lv_obj_get_parent(obj)`.

To get the children of an object, use `lv_obj_get_child(obj, child_prev)` (from last to first) or `lv_obj_get_child_back(obj, child_prev)` (from first to last). To get the first child, pass `NULL` as the second parameter and use the return value to iterate through the children. The function will return `NULL` if there are no more children. For example:

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

`lv_obj_count_children(obj)` tells the number of children on an object. `lv_obj_count_children_recursive(obj)` also tells the number of children but counts children of children recursively.

Screens

When you have created a screen like `lv_obj_t * screen = lv_obj_create(NULL, NULL)`, you can load it with `lv_scr_load(screen)`. The `lv_scr_act()` function gives you a pointer to the current screen.

If you have more display then it's important to know that these functions operate on the lastly created or the explicitly selected (with `lv_disp_set_default`) display.

To get an object's screen use the `lv_obj_get_screen(obj)` function.

Layers

There are two automatically generated layers:

- top layer
- system layer

They are independent of the screens and they will be shown on every screen. The *top layer* is above every object on the screen and the *system layer* is above the *top layer* too. You can add any pop-up windows to the *top layer* freely. But, the *system layer* is restricted to system-level things (e.g. mouse cursor will be placed here in `lv_indev_set_cursor()`).

The `lv_layer_top()` and `lv_layer_sys()` functions gives a pointer to the top or system layer.

You can bring an object to the foreground or send it to the background with `lv_obj_move_foreground(obj)` and `lv_obj_move_background(obj)`.

Read the *Layer overview* section to learn more about layers.

Events

To set an event callback for an object, use `lv_obj_set_event_cb(obj, event_cb)`,

To manually send an event to an object, use `lv_event_send(obj, LV_EVENT_..., data)`

Read the *Event overview* to learn more about the events.

5.1.2 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.

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.

Style

Be sure to read the *Style overview* first.

To add a style to an object use `lv_obj_add_style(obj, part, &new_style)` function. The Base object use all the rectangle-like style properties.

To remove all styles from an object use `lv_obj_reset_style_list(obj, part)`

If you modify a style, which is already used by objects, in order to refresh the affected objects you can use either `lv_obj_refresh_style(obj)` on each object using it or to notify all objects with a given style use `lv_obj_report_style_mod(&style)`. If the parameter of `lv_obj_report_style_mod` is `NULL`, all objects will be notified.

Attributes

There are some attributes which can be enabled/disabled by `lv_obj_set_...(obj, true/false)`:

- **hidden** - Hide the object. It will not be drawn and will be considered by input devices as if it doesn't exist., Its children will be hidden too.
- **click** - Allows you to click the object via input devices. If disabled, then click events are passed to the object behind this one. (E.g. *Labels* are not clickable by default)
- **top** - If enabled then when this object or any of its children is clicked then this object comes to the foreground.
- **drag** - Enable dragging (moving by an input device)
- **drag_dir** - Enable dragging only in specific directions. Can be `LV_DRAG_DIR_HOR/VER/ALL`.
- **drag_throw** - Enable "throwing" with dragging as if the object would have momentum
- **drag_parent** - If enabled then the object's parent will be moved during dragging. It will look like as if the parent is dragged. Checked recursively, so can propagate to grandparents too.
- **parent_event** - Propagate the events to the parents too. Checked recursively, so can propagate to grandparents too.
- **opa_scale_enable** - Enable opacity scaling. See the [#opa-scale](Opa scale) section.

Protect

There are some specific actions which happen automatically in the library. To prevent one or more that kind of actions, you can protect the object against them. The following protections exists:

- **LV_PROTECT_NONE** No protection
- **LV_PROTECT_POS** Prevent automatic positioning (e.g. Layout in *Containers*)
- **LV_PROTECT_FOLLOW** Prevent the object be followed (make a "line break") in automatic ordering (e.g. Layout in *Containers*)
- **LV_PROTECT_PARENT** Prevent automatic parent change. (e.g. *Page* moves the children created on the background to the scrollable)
- **LV_PROTECT_PRESS_LOST** Prevent losing press when the press is slid out of the objects. (E.g. a *Button* can be released out of it if it was being pressed)
- **LV_PROTECT_CLICK_FOCUS** Prevent automatically focusing the object if it's in a *Group* and click focus is enabled.
- **LV_PROTECT_CHILD_CHG** Disable the child change signal. Used internally by the library

The `lv_obj_set/clear_protect(obj, LV_PROTECT_...)` sets/clears the protection. You can use *'OR'ed* values of protection types too.

Groups

Once, an object is added to *group* with `lv_group_add_obj(group, obj)` the object's current group can be get with `lv_obj_get_group(obj)`.

`lv_obj_is_focused(obj)` tells if the object is currently focused on its group or not. If the object is not added to a group, `false` will be returned.

Read the *Input devices overview* to learn more about the *Groups*.

Extended click area

By default, the objects can be clicked only on their coordinates, however, this area can be extended with `lv_obj_set_ext_click_area(obj, left, right, top, bottom)`. `left/right/top/bottom` describes how far the clickable area should extend past the default in each direction.

This feature needs to enabled in *lv_conf.h* with `LV_USE_EXT_CLICK_AREA`. The possible values are:

- **LV_EXT_CLICK_AREA_FULL** store all 4 coordinates as `lv_coord_t`
- **LV_EXT_CLICK_AREA_TINY** store only horizontal and vertical coordinates (use the greater value of left/right and top/bottom) as `uint8_t`
- **LV_EXT_CLICK_AREA_OFF** Disable this feature

5.1.3 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.1.4 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.1.5 Example

5.1.6 API

Typedefs

typedef uint8_t **lv_design_mode_t**

typedef uint8_t **lv_design_res_t**

typedef *lv_design_res_t* (*__lv_design_cb_t__)(**struct** *__lv_obj_t* *obj, **const** lv_area_t *clip_area, *lv_design_mode_t* mode)

The design callback is used to draw the object on the screen. It accepts the object, a mask area, and the mode in which to draw the object.

typedef uint8_t **lv_event_t**

Type of event being sent to the object.

typedef void (*__lv_event_cb_t__)(**struct** *__lv_obj_t* *obj, *lv_event_t* event)

Event callback. Events are used to notify the user of some action being taken on the object. For details, see *lv_event_t*.

typedef uint8_t **lv_signal_t**

typedef lv_res_t (*__lv_signal_cb_t__)(**struct** *__lv_obj_t* *obj, *lv_signal_t* sign, void *param)

typedef uint8_t **lv_protect_t**

typedef uint8_t **lv_state_t**

typedef struct *__lv_obj_t* **lv_obj_t**

typedef uint8_t **lv_obj_part_t**

Enums

enum [anonymous]
Design modes

Values:

LV_DESIGN_DRAW_MAIN
Draw the main portion of the object

LV_DESIGN_DRAW_POST
Draw extras on the object

LV_DESIGN_COVER_CHK
Check if the object fully covers the 'mask_p' area

enum [anonymous]
Design results

Values:

LV_DESIGN_RES_OK
Draw ready

LV_DESIGN_RES_COVER
Returned on `LV_DESIGN_COVER_CHK` if the areas is fully covered

LV_DESIGN_RES_NOT_COVER
Returned on `LV_DESIGN_COVER_CHK` if the areas is not covered

LV_DESIGN_RES_MASKED
Returned on `LV_DESIGN_COVER_CHK` if the areas is masked out (children also not cover)

enum [anonymous]
Values:

LV_EVENT_PRESSED
The object has been pressed

LV_EVENT_PRESSING
The object is being pressed (called continuously while pressing)

LV_EVENT_PRESS_LOST
User is still pressing but slid cursor/finger off of the object

LV_EVENT_SHORT_CLICKED
User pressed object for a short period of time, then released it. Not called if dragged.

LV_EVENT_LONG_PRESSED
Object has been pressed for at least `LV_INDEV_LONG_PRESS_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 cases when the object has been released

LV_EVENT_DRAG_BEGIN

LV_EVENT_DRAG_END

LV_EVENT_DRAG_THROW_BEGIN

LV_EVENT_GESTURE
The object has been gesture

LV_EVENT_KEY

LV_EVENT_FOCUSED

LV_EVENT_DEFOCUSED

LV_EVENT_LEAVE

LV_EVENT_VALUE_CHANGED
The object's value has changed (i.e. slider moved)

LV_EVENT_INSERT

LV_EVENT_REFRESH

LV_EVENT_APPLY
"Ok", "Apply" or similar specific button has clicked

LV_EVENT_CANCEL
"Close", "Cancel" or similar specific button has clicked

LV_EVENT_DELETE
Object is being deleted

enum [**anonymous**]
Signals are for use by the object itself or to extend the object's functionality. Applications should use *lv_obj_set_event_cb* to be notified of events that occur on the object.

Values:

LV_SIGNAL_CLEANUP
Object is being deleted

LV_SIGNAL_CHILD_CHG
Child was removed/added

LV_SIGNAL_COORD_CHG
Object coordinates/size have changed

LV_SIGNAL_PARENT_SIZE_CHG
Parent's size has changed

LV_SIGNAL_STYLE_CHG
Object's style has changed

LV_SIGNAL_BASE_DIR_CHG
The base dir has changed

LV_SIGNAL_REFR_EXT_DRAW_PAD
Object's extra padding has changed

LV_SIGNAL_GET_TYPE
LVGL needs to retrieve the object's type

LV_SIGNAL_GET_STYLE
Get the style of an object

LV_SIGNAL_GET_STATE_DSC
Get the state of the object

LV_SIGNAL_HIT_TEST
Advanced hit-testing

LV_SIGNAL_PRESSED
The object has been pressed

LV_SIGNAL_PRESSING
The object is being pressed (called continuously while pressing)

LV_SIGNAL_PRESS_LOST
User is still pressing but slid cursor/finger off of the object

LV_SIGNAL_RELEASED
User pressed object for a short period of time, then released it. Not called if dragged.

LV_SIGNAL_LONG_PRESS
Object has been pressed for at least `LV_INDEV_LONG_PRESS_TIME`. Not called if dragged.

LV_SIGNAL_LONG_PRESS_REP
Called after `LV_INDEV_LONG_PRESS_TIME` in every `LV_INDEV_LONG_PRESS_REP_TIME` ms. Not called if dragged.

LV_SIGNAL_DRAG_BEGIN

LV_SIGNAL_DRAG_THROW_BEGIN

LV_SIGNAL_DRAG_END

LV_SIGNAL_GESTURE
The object has been gesture

LV_SIGNAL_LEAVE
Another object is clicked or chosen via an input device

LV_SIGNAL_FOCUS

LV_SIGNAL_DEFOCUS

LV_SIGNAL_CONTROL

LV_SIGNAL_GET_EDITABLE

enum [**anonymous**]
Values:

LV_PROTECT_NONE = 0x00

LV_PROTECT_CHILD_CHG = 0x01
Disable the child change signal. Used by the library

LV_PROTECT_PARENT = 0x02
Prevent automatic parent change (e.g. in lv_page)

LV_PROTECT_POS = 0x04
Prevent automatic positioning (e.g. in lv_cont layout)

LV_PROTECT_FOLLOW = 0x08
Prevent the object be followed in automatic ordering (e.g. in lv_cont PRETTY layout)

LV_PROTECT_PRESS_LOST = 0x10
If the `indev` was pressing this object but swiped out while pressing do not search other object.

LV_PROTECT_CLICK_FOCUS = 0x20
Prevent focusing the object by clicking on it

enum [anonymous]
Values:

LV_STATE_DEFAULT = 0x00

LV_STATE_CHECKED = 0x01

LV_STATE_FOCUSED = 0x02

LV_STATE_EDITED = 0x04

LV_STATE_HOVERED = 0x08

LV_STATE_PRESSED = 0x10

LV_STATE_DISABLED = 0x20

enum [anonymous]
Values:

LV_OBJ_PART_MAIN

_LV_OBJ_PART_VIRTUAL_LAST = 0x01

_LV_OBJ_PART_REAL_LAST = 0x40

LV_OBJ_PART_ALL = 0xFF

Functions

void **lv_init**(void)
Init. the 'lv' library.

void **lv_deinit**(void)
Deinit the 'lv' library Currently only implemented when not using custom allocators, or GC is enabled.

lv_obj_t ***lv_obj_create**(*lv_obj_t *parent*, **const** *lv_obj_t *copy*)
Create a basic object

Return pointer to the new object

Parameters

- `parent`: pointer to a parent object. If NULL then a screen will be created
- `copy`: pointer to a base object, if not NULL then the new object will be copied from it

lv_res_t **lv_obj_del**(*lv_obj_t *obj*)
Delete 'obj' and all of its children

Return LV_RES_INV because the object is deleted

Parameters

- `obj`: pointer to an object to delete

void **lv_obj_del_anim_ready_cb**(*lv_anim_t *a*)
A function to be easily used in animation ready callback to delete an object when the animation is ready

Parameters

- `a`: pointer to the animation

void **lv_obj_del_async(struct** *_lv_obj_t *obj***)**
Helper function for asynchronously deleting objects. Useful for cases where you can't delete an object directly in an `LV_EVENT_DELETE` handler (i.e. parent).

See lv_async_call

Parameters

- `obj`: object to delete

void **lv_obj_clean(***lv_obj_t *obj***)**
Delete all children of an object

Parameters

- `obj`: pointer to an object

void **lv_obj_invalidate_area(const** *lv_obj_t *obj*, **const** lv_area_t **area***)**
Mark an area of an object as invalid. This area will be redrawn by 'lv_refr_task'

Parameters

- `obj`: pointer to an object
- `area`: the area to redraw

void **lv_obj_invalidate(const** *lv_obj_t *obj***)**
Mark the object as invalid therefore its current position will be redrawn by 'lv_refr_task'

Parameters

- `obj`: pointer to an object

void **lv_obj_set_parent(***lv_obj_t *obj*, *lv_obj_t *parent***)**
Set a new parent for an object. Its relative position will be the same.

Parameters

- `obj`: pointer to an object. Can't be a screen.
- `parent`: pointer to the new parent object. (Can't be NULL)

void **lv_obj_move_foreground(***lv_obj_t *obj***)**
Move and object to the foreground

Parameters

- `obj`: pointer to an object

void **lv_obj_move_background(***lv_obj_t *obj***)**
Move and object to the background

Parameters

- `obj`: pointer to an object

void **lv_obj_set_pos(***lv_obj_t *obj*, lv_coord_t *x*, lv_coord_t *y***)**
Set relative the position of an object (relative to the parent)

Parameters

- `obj`: pointer to an object
- `x`: new distance from the left side of the parent
- `y`: new distance from the top of the parent

void **lv_obj_set_x**(*lv_obj_t* **obj*, lv_coord_t *x*)
Set the x coordinate of a object

Parameters

- `obj`: pointer to an object
- `x`: new distance from the left side from the parent

void **lv_obj_set_y**(*lv_obj_t* **obj*, lv_coord_t *y*)
Set the y coordinate of a object

Parameters

- `obj`: pointer to an object
- `y`: new distance from the top of the parent

void **lv_obj_set_size**(*lv_obj_t* **obj*, lv_coord_t *w*, lv_coord_t *h*)
Set the size of an object

Parameters

- `obj`: pointer to an object
- `w`: new width
- `h`: new height

void **lv_obj_set_width**(*lv_obj_t* **obj*, lv_coord_t *w*)
Set the width of an object

Parameters

- `obj`: pointer to an object
- `w`: new width

void **lv_obj_set_height**(*lv_obj_t* **obj*, lv_coord_t *h*)
Set the height of an object

Parameters

- `obj`: pointer to an object
- `h`: new height

void **lv_obj_set_width_fit**(*lv_obj_t* **obj*, lv_coord_t *w*)
Set the width reduced by the left and right padding.

Parameters

- `obj`: pointer to an object
- `w`: the width without paddings

void **lv_obj_set_height_fit**(*lv_obj_t* **obj*, lv_coord_t *h*)
Set the height reduced by the top and bottom padding.

Parameters

- `obj`: pointer to an object
- `h`: the height without paddings

void **lv_obj_set_width_margin**(*lv_obj_t* **obj*, lv_coord_t *w*)
Set the width of an object by taking the left and right margin into account. The object width will be `obj_w = w - margin_left - margin_right`

Parameters

- `obj`: pointer to an object
- `w`: new height including margins

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

Set the height of an object by taking the top and bottom margin into account. The object height will be `obj_h = h - margin_top - margin_bottom`

Parameters

- `obj`: pointer to an object
- `h`: new height including margins

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

Align an object to an other object.

Parameters

- `obj`: pointer to an object to align
- `base`: pointer to an object (if NULL the parent is used). 'obj' will be aligned to it.
- `align`: type of alignment (see 'lv_align_t' enum)
- `x_ofs`: x coordinate offset after alignment
- `y_ofs`: y coordinate offset after alignment

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

Align an object to an other object.

Parameters

- `obj`: pointer to an object to align
- `base`: pointer to an object (if NULL the parent is used). 'obj' will be aligned to it.
- `align`: type of alignment (see 'lv_align_t' enum)
- `x_ofs`: x coordinate offset after alignment
- `y_ofs`: y coordinate offset after alignment

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

Realign the object based on the last `lv_obj_align` parameters.

Parameters

- `obj`: pointer to an object

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

Enable the automatic realign of the object when its size has changed based on the last `lv_obj_align` parameters.

Parameters

- `obj`: pointer to an object
- `en`: true: enable auto realign; false: disable auto realign

void **`lv_obj_set_ext_click_area`**(*lv_obj_t *obj*, lv_coord_t *left*, lv_coord_t *right*, lv_coord_t *top*, lv_coord_t *bottom*)

Set the size of an extended clickable area

Parameters

- `obj`: pointer to an object
- `left`: extended clickable are on the left [px]
- `right`: extended clickable are on the right [px]
- `top`: extended clickable are on the top [px]
- `bottom`: extended clickable are on the bottom [px]

void **`lv_obj_add_style(`***lv_obj_t *obj*, uint8_t *part*, lv_style_t **style***`)**
Add a new style to the style list of an object.

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `style`: pointer to a style to add (Only its pointer will be saved)

void **`lv_obj_remove_style(`***lv_obj_t *obj*, uint8_t *part*, lv_style_t **style***`)**
Remove a style from the style list of an object.

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `style`: pointer to a style to remove

void **`lv_obj_clean_style_list(`***lv_obj_t *obj*, uint8_t *part***`)**
Reset a style to the default (empty) state. Release all used memories and cancel pending related transitions. Typically used in 'LV_SIGN_CLEAN_UP.

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style list should be reseted. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`

void **`lv_obj_reset_style_list(`***lv_obj_t *obj*, uint8_t *part***`)**
Reset a style to the default (empty) state. Release all used memories and cancel pending related transitions. Also notifies the object about the style change.

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style list should be reseted. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`

void **`lv_obj_refresh_style(`***lv_obj_t *obj*, lv_style_property_t *prop***`)**
Notify an object (and its children) about its style is modified

Parameters

- `obj`: pointer to an object
- `prop`: `LV_STYLE_PROP_ALL` or an `LV_STYLE_...` property. It is used to optimize what needs to be refreshed.

void **lv_obj_report_style_mod**(lv_style_t *style)
Notify all object if a style is modified

Parameters

- `style`: pointer to a style. Only the objects with this style will be notified (NULL to notify all objects)

void **_lv_obj_set_style_local_color**(*lv_obj_t* *obj, uint8_t *type*, lv_style_property_t *prop*, *lv_color_t color*)
Set a local style property of a part of an object in a given state.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: a style property ORed with a state. E.g. `LV_STYLE_BORDER_COLOR | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`: value to set

void **_lv_obj_set_style_local_int**(*lv_obj_t* *obj, uint8_t *type*, lv_style_property_t *prop*, lv_style_int_t *value*)
Set a local style property of a part of an object in a given state.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: a style property ORed with a state. E.g. `LV_STYLE_BORDER_WIDTH | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`: value to set

void **_lv_obj_set_style_local_opa**(*lv_obj_t* *obj, uint8_t *type*, lv_style_property_t *prop*, lv_opa_t *opa*)
Set a local style property of a part of an object in a given state.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`

- `prop`: a style property ORed with a state. E.g. `LV_STYLE_BORDER_OPA | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`: value to set

void **_lv_obj_set_style_local_ptr**(*lv_obj_t* **obj*, uint8_t *type*, lv_style_property_t *prop*, **const** void **value*)

Set a local style property of a part of an object in a given state.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: a style property ORed with a state. E.g. `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`
- `the`: value to set

bool **lv_obj_remove_style_local_prop**(*lv_obj_t* **obj*, uint8_t *part*, lv_style_property_t *prop*)

Remove a local style property from a part of an object with a given state.

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

Return true: the property was found and removed; false: the property was not found

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be removed. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: a style property ORed with a state. E.g. `LV_STYLE_TEXT_FONT | (LV_STATE_PRESSED << LV_STYLE_STATE_POS)`

void **lv_obj_set_hidden**(*lv_obj_t* **obj*, bool *en*)

Hide an object. It won't be visible and clickable.

Parameters

- `obj`: pointer to an object
- `en`: true: hide the object

void **lv_obj_set_adv_hittest**(*lv_obj_t* **obj*, bool *en*)

Set whether advanced hit-testing is enabled on an object

Parameters

- `obj`: pointer to an object
- `en`: true: advanced hit-testing is enabled

void **lv_obj_set_click**(*lv_obj_t* **obj*, bool *en*)

Enable or disable the clicking of an object

Parameters

- `obj`: pointer to an object
- `en`: true: make the object clickable

void **lv_obj_set_top**(*lv_obj_t *obj*, bool *en*)
Enable to bring this object to the foreground if it or any of its children is clicked

Parameters

- `obj`: pointer to an object
- `en`: true: enable the auto top feature

void **lv_obj_set_drag**(*lv_obj_t *obj*, bool *en*)
Enable the dragging of an object

Parameters

- `obj`: pointer to an object
- `en`: true: make the object draggable

void **lv_obj_set_drag_dir**(*lv_obj_t *obj*, *lv_drag_dir_t drag_dir*)
Set the directions an object can be dragged in

Parameters

- `obj`: pointer to an object
- `drag_dir`: bitwise OR of allowed drag directions

void **lv_obj_set_drag_throw**(*lv_obj_t *obj*, bool *en*)
Enable the throwing of an object after is is dragged

Parameters

- `obj`: pointer to an object
- `en`: true: enable the drag throw

void **lv_obj_set_drag_parent**(*lv_obj_t *obj*, bool *en*)
Enable to use parent for drag related operations. If trying to drag the object the parent will be moved instead

Parameters

- `obj`: pointer to an object
- `en`: true: enable the 'drag parent' for the object

void **lv_obj_set_gesture_parent**(*lv_obj_t *obj*, bool *en*)
Enable to use parent for gesture related operations. If trying to gesture the object the parent will be moved instead

Parameters

- `obj`: pointer to an object
- `en`: true: enable the 'gesture parent' for the object

void **lv_obj_set_parent_event**(*lv_obj_t *obj*, bool *en*)
Propagate the events to the parent too

Parameters

- `obj`: pointer to an object
- `en`: true: enable the event propagation

void **lv_obj_set_base_dir**(*lv_obj_t* *obj, lv_bidi_dir_t *dir*)
Set the base direction of the object

Parameters

- `obj`: pointer to an object
- `dir`: the new base direction. `LV_BIDI_DIR_LTR/RTL/AUTO/INHERIT`

void **lv_obj_add_protect**(*lv_obj_t* *obj, uint8_t *prot*)
Set a bit or bits in the protect filed

Parameters

- `obj`: pointer to an object
- `prot`: 'OR'-ed values from `lv_protect_t`

void **lv_obj_clear_protect**(*lv_obj_t* *obj, uint8_t *prot*)
Clear a bit or bits in the protect filed

Parameters

- `obj`: pointer to an object
- `prot`: 'OR'-ed values from `lv_protect_t`

void **lv_obj_set_state**(*lv_obj_t* *obj, *lv_state_t state*)
Set the state (fully overwrite) of an object. If specified in the styles a transition animation will be started from the previous state to the current

Parameters

- `obj`: pointer to an object
- `state`: the new state

void **lv_obj_add_state**(*lv_obj_t* *obj, *lv_state_t state*)
Add a given state or states to the object. The other state bits will remain unchanged. If specified in the styles a transition animation will be started from the previous state to the current

Parameters

- `obj`: pointer to an object
- `state`: the state bits to add. E.g `LV_STATE_PRESSED | LV_STATE_FOCUSED`

void **lv_obj_clear_state**(*lv_obj_t* *obj, *lv_state_t state*)
Remove a given state or states to the object. The other state bits will remain unchanged. If specified in the styles a transition animation will be started from the previous state to the current

Parameters

- `obj`: pointer to an object
- `state`: the state bits to remove. E.g `LV_STATE_PRESSED | LV_STATE_FOCUSED`

void **lv_obj_finish_transitions**(*lv_obj_t* *obj, uint8_t *part*)
Finish all pending transitions on a part of an object

Parameters

- `obj`: pointer to an object
- `part`: part of the object, e.g `LV_BRN_PART_MAIN` or `LV_OBJ_PART_ALL` for all parts

void **lv_obj_set_event_cb**(*lv_obj_t *obj*, *lv_event_cb_t event_cb*)

Set a an event handler function for an object. Used by the user to react on event which happens with the object.

Parameters

- `obj`: pointer to an object
- `event_cb`: the new event function

lv_res_t **lv_event_send**(*lv_obj_t *obj*, *lv_event_t event*, **const** void **data*)

Send an event to the object

Return LV_RES_OK: `obj` was not deleted in the event; LV_RES_INV: `obj` was deleted in the event

Parameters

- `obj`: pointer to an object
- `event`: the type of the event from `lv_event_t`.
- `data`: arbitrary data depending on the object type and the event. (Usually `NULL`)

lv_res_t **lv_event_send_func**(*lv_event_cb_t event_xcb*, *lv_obj_t *obj*, *lv_event_t event*, **const** void **data*)

Call an event function with an object, event, and data.

Return LV_RES_OK: `obj` was not deleted in the event; LV_RES_INV: `obj` was deleted in the event

Parameters

- `event_xcb`: an event callback function. If `NULL` `LV_RES_OK` will return without any actions. (the 'x' in the argument name indicates that its not a fully generic function because it not follows the `func_name(object, callback, ...)` convention)
- `obj`: pointer to an object to associate with the event (can be `NULL` to simply call the `event_cb`)
- `event`: an event
- `data`: pointer to a custom data

const void ***lv_event_get_data**(void)

Get the `data` parameter of the current event

Return the `data` parameter

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

Set the a signal function of an object. Used internally by the library. Always call the previous signal function in the new.

Parameters

- `obj`: pointer to an object
- `signal_cb`: the new signal function

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

Send an event to the object

Return LV_RES_OK or LV_RES_INV

Parameters

- `obj`: pointer to an object

- `event`: the type of the event from `lv_event_t`.

void **`lv_obj_set_design_cb`**(*lv_obj_t *obj*, *lv_design_cb_t design_cb*)
Set a new design function for an object

Parameters

- `obj`: pointer to an object
- `design_cb`: the new design function

void ***`lv_obj_allocate_ext_attr`**(*lv_obj_t *obj*, uint16_t *ext_size*)
Allocate a new ext. data for an object

Return pointer to the allocated ext

Parameters

- `obj`: pointer to an object
- `ext_size`: the size of the new ext. data

void **`lv_obj_refresh_ext_draw_pad`**(*lv_obj_t *obj*)
Send a 'LV_SIGNAL_REFR_EXT_SIZE' signal to the object to refresh the extended draw area. he object needs to be invalidated by `lv_obj_invalidate(obj)` manually after this function.

Parameters

- `obj`: pointer to an object

lv_obj_t ***`lv_obj_get_screen`**(**const** *lv_obj_t *obj*)
Return with the screen of an object

Return pointer to a screen

Parameters

- `obj`: pointer to an object

lv_disp_t ***`lv_obj_get_disp`**(**const** *lv_obj_t *obj*)
Get the display of an object

Return pointer the object's display

lv_obj_t ***`lv_obj_get_parent`**(**const** *lv_obj_t *obj*)
Returns with the parent of an object

Return pointer to the parent of 'obj'

Parameters

- `obj`: pointer to an object

lv_obj_t ***`lv_obj_get_child`**(**const** *lv_obj_t *obj*, **const** *lv_obj_t *child*)
Iterate through the children of an object (start from the "youngest, lastly created")

Return the child after 'act_child' or NULL if no more child

Parameters

- `obj`: pointer to an object
- `child`: NULL at first call to get the next children and the previous return value later

lv_obj_t ***`lv_obj_get_child_back`**(**const** *lv_obj_t *obj*, **const** *lv_obj_t *child*)
Iterate through the children of an object (start from the "oldest", firstly created)

Return the child after 'act_child' or NULL if no more child

Parameters

- `obj`: pointer to an object
- `child`: NULL at first call to get the next children and the previous return value later

uint16_t **`lv_obj_count_children(const`** *lv_obj_t *obj*`)`
Count the children of an object (only children directly on 'obj')

Return children number of 'obj'

Parameters

- `obj`: pointer to an object

uint16_t **`lv_obj_count_children_recursive(const`** *lv_obj_t *obj*`)`
Recursively count the children of an object

Return children number of 'obj'

Parameters

- `obj`: pointer to an object

void **`lv_obj_get_coords(const`** *lv_obj_t *obj*, lv_area_t **cords_p*`)`
Copy the coordinates of an object to an area

Parameters

- `obj`: pointer to an object
- `cords_p`: pointer to an area to store the coordinates

void **`lv_obj_get_inner_coords(const`** *lv_obj_t *obj*, lv_area_t **coords_p*`)`
Reduce area retried by *`lv_obj_get_coords()`* the get graphically usable area of an object. (Without the size of the border or other extra graphical elements)

Parameters

- `coords_p`: store the result area here

lv_coord_t **`lv_obj_get_x(const`** *lv_obj_t *obj*`)`
Get the x coordinate of object

Return distance of 'obj' from the left side of its parent

Parameters

- `obj`: pointer to an object

lv_coord_t **`lv_obj_get_y(const`** *lv_obj_t *obj*`)`
Get the y coordinate of object

Return distance of 'obj' from the top of its parent

Parameters

- `obj`: pointer to an object

lv_coord_t **`lv_obj_get_width(const`** *lv_obj_t *obj*`)`
Get the width of an object

Return the width

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_height(const** *lv_obj_t *obj***)**
Get the height of an object

Return the height

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_width_fit(const** *lv_obj_t *obj***)**
Get that width reduced by the left and right padding.

Return the width which still fits into the container

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_height_fit(const** *lv_obj_t *obj***)**
Get that height reduced by the top an bottom padding.

Return the height which still fits into the container

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_height_margin(***lv_obj_t *obj***)**
Get the height of an object by taking the top and bottom margin into account. The returned height will be `obj_h + margin_top + margin_bottom`

Return the height including thee margins

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_width_margin(***lv_obj_t *obj***)**
Get the width of an object by taking the left and right margin into account. The returned width will be `obj_w + margin_left + margin_right`

Return the height including thee margins

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_width_grid(***lv_obj_t *obj*, uint8_t *div*, uint8_t *span***)**
Divide the width of the object and get the width of a given number of columns. Take paddings into account.

Return the width according to the given parameters

Parameters

- `obj`: pointer to an object
- `div`: indicates how many columns are assumed. If 1 the width will be set the the parent's width If 2 only half parent width - inner padding of the parent If 3 only third parent width - 2 * inner padding of the parent
- `span`: how many columns are combined

lv_coord_t **lv_obj_get_height_grid(***lv_obj_t *obj*, uint8_t *div*, uint8_t *span***)**
Divide the height of the object and get the width of a given number of columns. Take paddings into account.

Return the height according to the given parameters

Parameters

- `obj`: pointer to an object
- `div`: indicates how many rows are assumed. If 1 the height will be set the the parent's height If 2 only half parent height - inner padding of the parent If 3 only third parent height - 2 * inner padding of the parent
- `span`: how many rows are combined

bool **lv_obj_get_auto_realign(const** *lv_obj_t *obj*)
Get the automatic realign property of the object.

Return true: auto realign is enabled; false: auto realign is disabled

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_ext_click_pad_left(const** *lv_obj_t *obj*)
Get the left padding of extended clickable area

Return the extended left padding

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_ext_click_pad_right(const** *lv_obj_t *obj*)
Get the right padding of extended clickable area

Return the extended right padding

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_ext_click_pad_top(const** *lv_obj_t *obj*)
Get the top padding of extended clickable area

Return the extended top padding

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_ext_click_pad_bottom(const** *lv_obj_t *obj*)
Get the bottom padding of extended clickable area

Return the extended bottom padding

Parameters

- `obj`: pointer to an object

lv_coord_t **lv_obj_get_ext_draw_pad(const** *lv_obj_t *obj*)
Get the extended size attribute of an object

Return the extended size attribute

Parameters

- `obj`: pointer to an object

lv_style_list_t ***lv_obj_get_style_list(const** *lv_obj_t *obj*, uint8_t *part*)
Get the style list of an object's part.

Return pointer to the style list. (Can be `NULL`)

Parameters

- `obj`: pointer to an object.
- `part`: part the part of the object which style list should be get. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`

lv_style_int_t **`_lv_obj_get_style_int(const`** *lv_obj_t* **obj*, uint8_t *part*, lv_style_property_t *prop*`)`

Get a style property of a part of an object in the object's current state. If there is a running transitions it is taken into account

Return the value of the property of the given part in the current state. If the property is not found a default value will be returned.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be get. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: the property to get. E.g. `LV_STYLE_BORDER_WIDTH`. The state of the object will be added internally

lv_color_t **`_lv_obj_get_style_color(const`** *lv_obj_t *obj*, uint8_t *part*, lv_style_property_t *prop*`)`

Get a style property of a part of an object in the object's current state. If there is a running transitions it is taken into account

Return the value of the property of the given part in the current state. If the property is not found a default value will be returned.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be get. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: the property to get. E.g. `LV_STYLE_BORDER_COLOR`. The state of the object will be added internally

lv_opa_t **`_lv_obj_get_style_opa(const`** *lv_obj_t *obj*, uint8_t *part*, lv_style_property_t *prop*`)`

Get a style property of a part of an object in the object's current state. If there is a running transitions it is taken into account

Return the value of the property of the given part in the current state. If the property is not found a default value will be returned.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be get. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: the property to get. E.g. `LV_STYLE_BORDER_OPA`. The state of the object will be added internally

const void ***_lv_obj_get_style_ptr(const** *lv_obj_t *obj*, uint8_t *part*, lv_style_property_t *prop*)

Get a style property of a part of an object in the object's current state. If there is a running transitions it is taken into account

Return the value of the property of the given part in the current state. If the property is not found a default value will be returned.

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

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be get. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`
- `prop`: the property to get. E.g. `LV_STYLE_TEXT_FONT`. The state of the object will be added internally

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

Get the local style of a part of an object.

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

Parameters

- `obj`: pointer to an object
- `part`: the part of the object which style property should be set. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_PART_MAIN`, `LV_SLIDER_PART_KNOB`

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

Get the hidden attribute of an object

Return true: the object is hidden

Parameters

- `obj`: pointer to an object

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

Get whether advanced hit-testing is enabled on an object

Return true: advanced hit-testing is enabled

Parameters

- `obj`: pointer to an object

bool **lv_obj_get_click(const** *lv_obj_t *obj*)

Get the click enable attribute of an object

Return true: the object is clickable

Parameters

- `obj`: pointer to an object

bool **`lv_obj_get_top(const`** *lv_obj_t *obj*`)`
Get the top enable attribute of an object

Return true: the auto top feature is enabled

Parameters

- `obj`: pointer to an object

bool **`lv_obj_get_drag(const`** *lv_obj_t *obj*`)`
Get the drag enable attribute of an object

Return true: the object is draggable

Parameters

- `obj`: pointer to an object

lv_drag_dir_t **`lv_obj_get_drag_dir(const`** *lv_obj_t *obj*`)`
Get the directions an object can be dragged

Return bitwise OR of allowed directions an object can be dragged in

Parameters

- `obj`: pointer to an object

bool **`lv_obj_get_drag_throw(const`** *lv_obj_t *obj*`)`
Get the drag throw enable attribute of an object

Return true: drag throw is enabled

Parameters

- `obj`: pointer to an object

bool **`lv_obj_get_drag_parent(const`** *lv_obj_t *obj*`)`
Get the drag parent attribute of an object

Return true: drag parent is enabled

Parameters

- `obj`: pointer to an object

bool **`lv_obj_get_parent_event(const`** *lv_obj_t *obj*`)`
Get the drag parent attribute of an object

Return true: drag parent is enabled

Parameters

- `obj`: pointer to an object

bool **`lv_obj_get_gesture_parent(const`** *lv_obj_t *obj*`)`
Get the gesture parent attribute of an object

Return true: gesture parent is enabled

Parameters

- `obj`: pointer to an object

lv_bidi_dir_t **lv_obj_get_base_dir(const** *lv_obj_t* ***obj*)**

uint8_t **lv_obj_get_protect(const** *lv_obj_t* ***obj*)**

Get the protect field of an object

Return protect field ('OR'ed values of `lv_protect_t`)

Parameters

- `obj`: pointer to an object

bool **lv_obj_is_protected(const** *lv_obj_t* ***obj***, uint8_t *prot*)

Check at least one bit of a given protect bitfield is set

Return false: none of the given bits are set, true: at least one bit is set

Parameters

- `obj`: pointer to an object
- `prot`: protect bits to test ('OR'ed values of `lv_protect_t`)

lv_state_t **lv_obj_get_state(const** *lv_obj_t* ***obj***, uint8_t *part*)

lv_signal_cb_t **lv_obj_get_signal_cb(const** *lv_obj_t* ***obj*)**

Get the signal function of an object

Return the signal function

Parameters

- `obj`: pointer to an object

lv_design_cb_t **lv_obj_get_design_cb(const** *lv_obj_t* ***obj*)**

Get the design function of an object

Return the design function

Parameters

- `obj`: pointer to an object

lv_event_cb_t **lv_obj_get_event_cb(const** *lv_obj_t* ***obj*)**

Get the event function of an object

Return the event function

Parameters

- `obj`: pointer to an object

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

Check if a given screen-space point is on an object's coordinates.

This method is intended to be used mainly by advanced hit testing algorithms to check whether the point is even within the object (as an optimization).

Parameters

- `obj`: object to check
- `point`: screen-space point

bool **lv_obj_hittest(***lv_obj_t* ***obj***, lv_point_t ***point*)**

Hit-test an object given a particular point in screen space.

Return true if the object is considered under the point

Parameters

- `obj`: object to hit-test
- `point`: screen-space point

void *`lv_obj_get_ext_attr(const` *lv_obj_t *obj*`)`
Get the ext pointer

Return the ext pointer but not the dynamic version Use it as ext->data1, and NOT da(ext)->data1

Parameters

- `obj`: pointer to an object

void `lv_obj_get_type(const` *lv_obj_t *obj*, *lv_obj_type_t *buf*`)`
Get object's and its ancestors type. Put their name in `type_buf` starting with the current type. E.g. buf.type[0]="lv_btn", buf.type[1]="lv_cont", buf.type[2]="lv_obj"

Parameters

- `obj`: pointer to an object which type should be get
- `buf`: pointer to an *`lv_obj_type_t`* buffer to store the types

lv_obj_user_data_t `lv_obj_get_user_data(const` *lv_obj_t *obj*`)`
Get the object's user data

Return user data

Parameters

- `obj`: pointer to an object

lv_obj_user_data_t *`lv_obj_get_user_data_ptr(const` *lv_obj_t *obj*`)`
Get a pointer to the object's user data

Return pointer to the user data

Parameters

- `obj`: pointer to an object

void `lv_obj_set_user_data(`*lv_obj_t *obj*, lv_obj_user_data_t *data*`)`
Set the object's user data. The data will be copied.

Parameters

- `obj`: pointer to an object
- `data`: user data

void *`lv_obj_get_group(const` *lv_obj_t *obj*`)`
Get the group of the object

Return the pointer to group of the object

Parameters

- `obj`: pointer to an object

bool `lv_obj_is_focused(const` *lv_obj_t *obj*`)`
Tell whether the object is the focused object of a group or not.

Return true: the object is focused, false: the object is not focused or not in a group

Parameters

- `obj`: pointer to an object

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

Used in the signal callback to handle `LV_SIGNAL_GET_TYPE` signal

Return LV_RES_OK

Parameters

- `buf`: pointer to *`lv_obj_type_t`*. (`param` in the signal callback)
- `name`: name of the object. E.g. "lv_btn". (Only the pointer is saved)

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

Initialize a rectangle descriptor from an object's styles

Note Only the relevant fields will be set. E.g. if `border width == 0` the other border properties won't be evaluated.

Parameters

- `obj`: pointer to an object
- `type`: type of style. E.g. `LV_OBJ_PART_MAIN`, `LV_BTN_SLIDER_KOB`
- `draw_dsc`: the descriptor the initialize

void **lv_obj_init_draw_label_dsc**(*lv_obj_t *obj*, uint8_t *type*, lv_draw_label_dsc_t **draw_dsc*)

void **lv_obj_init_draw_img_dsc**(*lv_obj_t *obj*, uint8_t *part*, lv_draw_img_dsc_t **draw_dsc*)

void **lv_obj_init_draw_line_dsc**(*lv_obj_t *obj*, uint8_t *part*, lv_draw_line_dsc_t **draw_dsc*)

lv_coord_t **lv_obj_get_draw_rect_ext_pad_size**(*lv_obj_t *obj*, uint8_t *part*)

Get the required extra size (around the object's part) to draw shadow, outline, value etc.

Parameters

- `obj`: pointer to an object
- `part`: part of the object

void **lv_obj_fade_in**(*lv_obj_t *obj*, uint32_t *time*, uint32_t *delay*)

Fade in (from transparent to fully cover) an object and all its children using an `opa_scale` animation.

Parameters

- `obj`: the object to fade in
- `time`: duration of the animation [ms]
- `delay`: wait before the animation starts [ms]

void **lv_obj_fade_out**(*lv_obj_t *obj*, uint32_t *time*, uint32_t *delay*)

Fade out (from fully cover to transparent) an object and all its children using an `opa_scale` animation.

Parameters

- `obj`: the object to fade in
- `time`: duration of the animation [ms]
- `delay`: wait before the animation starts [ms]

bool **lv_debug_check_obj_type**(**const** *lv_obj_t *obj*, **const** char **obj_type*)

Check if any object has a given type

Return true: valid

Parameters

- `obj`: pointer to an object
- `obj_type`: type of the object. (e.g. "lv_btn")

bool **`lv_debug_check_obj_valid(const`** *lv_obj_t *obj*`)`
Check if any object is still "alive", and part of the hierarchy

Return true: valid

Parameters

- `obj`: pointer to an object
- `obj_type`: type of the object. (e.g. "lv_btn")

`struct lv_realign_t`

Public Members

`const struct` *_lv_obj_t* ***`base`**

lv_coord_t **`xofs`**

lv_coord_t **`yofs`**

lv_align_t **`align`**

uint8_t **`auto_realign`**

uint8_t **`origo_align`**
1: the origo (center of the object) was aligned with `lv_obj_align_origo`

`struct _lv_obj_t`

Public Members

`struct` *_lv_obj_t* ***`parent`**
Pointer to the parent object

lv_ll_t **`child_ll`**
Linked list to store the children objects

lv_area_t **`coords`**
Coordinates of the object (x1, y1, x2, y2)

lv_event_cb_t **`event_cb`**
Event callback function

lv_signal_cb_t **`signal_cb`**
Object type specific signal function

lv_design_cb_t **`design_cb`**
Object type specific design function

void ***`ext_attr`**
Object type specific extended data

lv_style_list_t **`style_list`**

uint8_t **`ext_click_pad_hor`**
Extra click padding in horizontal direction

uint8_t **ext_click_pad_ver**
Extra click padding in vertical direction

lv_area_t **ext_click_pad**
Extra click padding area.

lv_coord_t **ext_draw_pad**
EXTend the size in every direction for drawing.

uint8_t **click**
1: Can be pressed by an input device

uint8_t **drag**
1: Enable the dragging

uint8_t **drag_throw**
1: Enable throwing with drag

uint8_t **drag_parent**
1: Parent will be dragged instead

uint8_t **hidden**
1: Object is hidden

uint8_t **top**
1: If the object or its children is clicked it goes to the foreground

uint8_t **parent_event**
1: Send the object's events to the parent too.

uint8_t **adv_hittest**
1: Use advanced hit-testing (slower)

uint8_t **gesture_parent**
1: Parent will be gesture instead

lv_drag_dir_t **drag_dir**
Which directions the object can be dragged in

lv_bidi_dir_t **base_dir**
Base direction of texts related to this object

void ***group_p**

uint8_t **protect**
Automatically happening actions can be prevented. 'OR'ed values from `lv_protect_t`

lv_state_t **state**

lv_realign_t **realign**
Information about the last call to *lv_obj_align*.

lv_obj_user_data_t **user_data**
Custom user data for object.

struct lv_obj_type_t
#include <lv_obj.h> Used by *`lv_obj_get_type()`*. The object's and its ancestor types are stored here

Public Members

const char ***type**[**LV_MAX_ANCESTOR_NUM**]
[0]: the actual type, [1]: ancestor, [2] #1's ancestor ... [x]: "lv_obj"

struct lv_hit_test_info_t

Public Members

lv_point_t ***point**

bool **result**

struct lv_get_style_info_t

Public Members

uint8_t **part**

lv_style_list_t ***result**

struct lv_get_state_info_t

Public Members

uint8_t **part**

lv_state_t **result**

5.2 Arc (lv_arc)

5.2.1 Overview

The Arc are consists of a background and a foreground arc. Both can have start and end angles and thickness.

5.2.2 Parts and Styles

The Arc's main part is called `LV_ARC_PART_MAIN`. It draws a background using the typical background style properties and an arc using the *line* style properties. The arc's size and position will respect the *padding* style properties.

`LV_ARC_PART_INDIC` is virtual part and it draws an other arc using the *line* style proeprties. It's padding values are interpreted relative to the background arc. The radius of the indicator arc will be modified according to the greatest padding value.

5.2.3 Usage

Angles

To set the angles of the background, use the `lv_arc_set_bg_angles(arc, start_angle, end_angle)` function or `lv_arc_set_bg_start/end_angle(arc, start_angle)`. Zero degree is at the middle right (3 o'clock) of the object and the degrees are increasing in a clockwise direction. The angles should be in [0;360] range.

Similarly, `lv_arc_set_angles(arc, start_angle, end_angle)` function or `lv_arc_set_start/end_angle(arc, start_angle)` sets the angles of the indicator arc.

5.2.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.2.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.2.6 Example

5.2.7 API

Typedefs

typedef uint8_t **lv_arc_part_t**

Enums

enum [anonymous]

Values:

LV_ARC_PART_BG = *LV_OBJ_PART_MAIN*

LV_ARC_PART_INDIC

_LV_ARC_PART_VIRTUAL_LAST

_LV_ARC_PART_REAL_LAST = *_LV_OBJ_PART_REAL_LAST*

Functions

lv_obj_t ***lv_arc_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a arc objects

Return pointer to the created arc

Parameters

- `par`: pointer to an object, it will be the parent of the new arc
- `copy`: pointer to a arc object, if not NULL then the new object will be copied from it

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

Set the start angle of an arc. 0 deg: right, 90 bottom, etc.

Parameters

- `arc`: pointer to an arc object
- `start`: the start angle

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

Set the start angle of an arc. 0 deg: right, 90 bottom, etc.

Parameters

- `arc`: pointer to an arc object
- `end`: the end angle

void **`lv_arc_set_angles`**(*lv_obj_t *arc*, uint16_t *start*, uint16_t *end*)
Set the start and end angles

Parameters

- `arc`: pointer to an arc object
- `start`: the start angle
- `end`: the end angle

void **`lv_arc_set_bg_start_angle`**(*lv_obj_t *arc*, uint16_t *start*)
Set the start angle of an arc background. 0 deg: right, 90 bottom, etc.

Parameters

- `arc`: pointer to an arc object
- `start`: the start angle

void **`lv_arc_set_bg_end_angle`**(*lv_obj_t *arc*, uint16_t *end*)
Set the start angle of an arc background. 0 deg: right, 90 bottom etc.

Parameters

- `arc`: pointer to an arc object
- `end`: the end angle

void **`lv_arc_set_bg_angles`**(*lv_obj_t *arc*, uint16_t *start*, uint16_t *end*)
Set the start and end angles of the arc background

Parameters

- `arc`: pointer to an arc object
- `start`: the start angle
- `end`: the end angle

void **`lv_arc_set_rotation`**(*lv_obj_t *arc*, uint16_t *rotation_angle*)
Set the rotation for the whole arc

Parameters

- `arc`: pointer to an arc object
- `rotation_angle`: rotation angle

uint16_t **`lv_arc_get_angle_start`**(*lv_obj_t *arc*)
Get the start angle of an arc.

Return the start angle [0..360]

Parameters

- `arc`: pointer to an arc object

uint16_t **`lv_arc_get_angle_end`**(*lv_obj_t *arc*)
Get the end angle of an arc.

Return the end angle [0..360]

Parameters

- `arc`: pointer to an arc object

uint16_t **`lv_arc_get_bg_angle_start`**(*lv_obj_t *arc*)
Get the start angle of an arc background.

Return the start angle [0..360]

Parameters

- `arc`: pointer to an arc object

uint16_t **`lv_arc_get_bg_angle_end`**(*lv_obj_t *arc*)
Get the end angle of an arc background.

Return the end angle [0..360]

Parameters

- `arc`: pointer to an arc object

`struct lv_arc_ext_t`

Public Members

uint16_t **`rotation_angle`**

uint16_t **`arc_angle_start`**

uint16_t **`arc_angle_end`**

uint16_t **`bg_angle_start`**

uint16_t **`bg_angle_end`**

lv_style_list_t **`style_arc`**

5.3 Bar (lv_bar)

5.3.1 Overview

The bar object has a background and an indicator on it. The width of the indicator is set according to the current value of the bar.

Vertical bars can be created if the width of the object is smaller than its height.

Not only end, but the start value of the bar can be set which changes the start position of the indicator.

5.3.2 Parts and Styles

The Bar's main part is called `LV_BAR_PART_BG` and it uses the typical background style properties.

`LV_BAR_PART_INDIC` is a virtual part which also uses all the typical background properties. By default the indicator maximal size is the same as the background's size but setting positive padding values in `LV_BAR_PART_BG` will make the indicator smaller. (negative values will make it larger) If the *value* style property is used on the indicator the alignment will be calculated based on the current size of the indicator. For example a center aligned value is always shown in the middle of the indicator regardless it's current size.

5.3.3 Usage

Value and range

A new value can be set by `lv_bar_set_value(bar, new_value, LV_ANIM_ON/OFF)`. The value is interpreted in a range (minimum and maximum values) which can be modified with `lv_bar_set_range(bar, min, max)`. The default range is 1..100.

The new value in `lv_bar_set_value` can be set with or without an animation depending on the last parameter (`LV_ANIM_ON/OFF`). The time of the animation can be adjusted by `lv_bar_set_anim_time(bar, 100)`. The time is in milliseconds unit.

It's also possible to set the start value of the bar using `lv_bar_set_start_value(bar, new_value, LV_ANIM_ON/OFF)`

Modes

The bar can be drawn symmetrical to zero (drawn from zero, left to right), if it's enabled with `lv_bar_set_type(bar, LV_BAR_TYPE_SYMMETRICAL)`.

5.3.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.3.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.3.6 Example

5.3.7 API

Typedefs

typedef uint8_t **lv_bar_type_t**

typedef uint8_t **lv_bar_part_t**

Enums

enum [anonymous]

Values:

LV_BAR_TYPE_NORMAL

LV_BAR_TYPE_SYMMETRICAL

LV_BAR_TYPE_CUSTOM

enum [anonymous]
Bar parts

Values:

LV_BAR_PART_BG

LV_BAR_PART_INDIC
Bar background style.

_LV_BAR_PART_VIRTUAL_LAST
Bar fill area style.

Functions

lv_obj_t ***lv_bar_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a bar objects

Return pointer to the created bar

Parameters

- `par`: pointer to an object, it will be the parent of the new bar
- `copy`: pointer to a bar object, if not NULL then the new object will be copied from it

void **lv_bar_set_value**(*lv_obj_t *bar*, int16_t *value*, *lv_anim_enable_t anim*)
Set a new value on the bar

Parameters

- `bar`: pointer to a bar object
- `value`: new value
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

void **lv_bar_set_start_value**(*lv_obj_t *bar*, int16_t *start_value*, *lv_anim_enable_t anim*)
Set a new start value on the bar

Parameters

- `bar`: pointer to a bar object
- `value`: new start value
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

void **lv_bar_set_range**(*lv_obj_t *bar*, int16_t *min*, int16_t *max*)
Set minimum and the maximum values of a bar

Parameters

- `bar`: pointer to the bar object
- `min`: minimum value
- `max`: maximum value

void **lv_bar_set_type**(*lv_obj_t *bar*, *lv_bar_type_t type*)
Set the type of bar.

Parameters

- `bar`: pointer to bar object
- `type`: bar type

void **lv_bar_set_anim_time**(*lv_obj_t *bar*, uint16_t *anim_time*)
Set the animation time of the bar

Parameters

- `bar`: pointer to a bar object
- `anim_time`: the animation time in milliseconds.

int16_t **lv_bar_get_value**(**const** *lv_obj_t *bar*)
Get the value of a bar

Return the value of the bar

Parameters

- `bar`: pointer to a bar object

int16_t **lv_bar_get_start_value**(**const** *lv_obj_t *bar*)
Get the start value of a bar

Return the start value of the bar

Parameters

- `bar`: pointer to a bar object

int16_t **lv_bar_get_min_value**(**const** *lv_obj_t *bar*)
Get the minimum value of a bar

Return the minimum value of the bar

Parameters

- `bar`: pointer to a bar object

int16_t **lv_bar_get_max_value**(**const** *lv_obj_t *bar*)
Get the maximum value of a bar

Return the maximum value of the bar

Parameters

- `bar`: pointer to a bar object

lv_bar_type_t **lv_bar_get_type**(*lv_obj_t *bar*)
Get the type of bar.

Return bar type

Parameters

- `bar`: pointer to bar object

uint16_t **lv_bar_get_anim_time**(**const** *lv_obj_t *bar*)
Get the animation time of the bar

Return the animation time in milliseconds.

Parameters

- `bar`: pointer to a bar object

struct lv_bar_anim_t

Public Members

lv_obj_t ***bar**

lv_anim_value_t **anim_start**

lv_anim_value_t **anim_end**

lv_anim_value_t **anim_state**

struct lv_bar_ext_t

#include <lv_bar.h> Data of bar

Public Members

int16_t **cur_value**

int16_t **min_value**

int16_t **max_value**

int16_t **start_value**

lv_area_t **indic_area**

lv_anim_value_t **anim_time**

lv_bar_anim_t **cur_value_anim**

lv_bar_anim_t **start_value_anim**

uint8_t **type**

lv_style_list_t **style_indic**

5.4 Button (lv_btn)

5.4.1 Overview

Buttons are simple rectangle-like objects. They are derived from *Containers* so layout and fit are also available. Besides, it can be enabled to automatically go to checked state on click.

5.4.2 Parts and Styles

The buttons has only a main style called `LV_BTN_PART_MAIN` and it can use all the properties from the following groups:

- background
- border
- outline
- shadow
- value
- pattern
- transitions

It also uses the *padding* properties when *layout* or *fit* is enabled.

5.4.3 Usage

States

To make buttons usage simpler the button's state can be get with `lv_btn_get_state(btn)`. It returns one of the following values:

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

With `lv_btn_get_state(btn, LV_BTN_STATE_...)` the buttons state can be changed manually.

If a more precise description of the state is required (e.g. focused) the general `lv_obj_get_state(btn)` can be used.

Checkable

You can configure the buttons as *toggle button* with `lv_btn_set_checkable(btn, true)`. In this case, on click, the button goes to `LV_STATE_CHECKED` state automatically, or back when clicked again.

Layout and Fit

Similarly to *Containers*, buttons also have layout and fit attributes.

- `lv_btn_set_layout(btn, LV_LAYOUT_...)` set a layout. The default is `LV_LAYOUT_CENTER`. So, if you add a label, then it will be automatically aligned to the middle and can't be moved with `lv_obj_set_pos()`. You can disable the layout with `lv_btn_set_layout(btn, LV_LAYOUT_OFF)`.
- `lv_btn_set_fit/fit2/fit4(btn, LV_FIT_..)` enables to set the button width and/or height automatically according to the children, parent, and fit type.

5.4.4 Events

Besides the Generic events the following Special events are sent by the buttons:

- **LV_EVENT_VALUE_CHANGED** - sent when the button is toggled.

Learn more about *Events*.

5.4.5 Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_RIGHT/UP** - Go to toggled state if toggling is enabled.
- **LV_KEY_LEFT/DOWN** - Go to non-toggled state if toggling is enabled.

Note that, the state of `LV_KEY_ENTER` is translated to `LV_EVENT_PRESSED/PRESSING/RELEASED` etc.

Learn more about *Keys*.

5.4.6 Example

5.4.7 API

Typedefs

typedef uint8_t **lv_btn_state_t**

typedef uint8_t **lv_btn_part_t**

Enums

enum [anonymous]

Possible states of a button. It can be used not only by buttons but other button-like objects too

Values:

LV_BTN_STATE_RELEASED

LV_BTN_STATE_PRESSED

LV_BTN_STATE_DISABLED

LV_BTN_STATE_CHECKED_RELEASED

LV_BTN_STATE_CHECKED_PRESSED

LV_BTN_STATE_CHECKED_DISABLED

_LV_BTN_STATE_LAST

enum [anonymous]

Styles

Values:

LV_BTN_PART_MAIN = *LV_OBJ_PART_MAIN*

_LV_BTN_PART_VIRTUAL_LAST

_LV_BTN_PART_REAL_LAST = *_LV_OBJ_PART_REAL_LAST*

Functions

lv_obj_t ***lv_btn_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a button object

Return pointer to the created button

Parameters

- `par`: pointer to an object, it will be the parent of the new button
- `copy`: pointer to a button object, if not NULL then the new object will be copied from it

void **lv_btn_set_checkable**(*lv_obj_t *btn*, bool *tgl*)

Enable the toggled states. On release the button will change from/to toggled state.

Parameters

- `btn`: pointer to a button object
- `tgl`: true: enable toggled states, false: disable

void **`lv_btn_set_state`**(*lv_obj_t *btn*, *lv_btn_state_t state*)
Set the state of the button

Parameters

- `btn`: pointer to a button object
- `state`: the new state of the button (from lv_btn_state_t enum)

void **`lv_btn_toggle`**(*lv_obj_t *btn*)
Toggle the state of the button (ON->OFF, OFF->ON)

Parameters

- `btn`: pointer to a button object

static void **`lv_btn_set_layout`**(*lv_obj_t *btn*, *lv_layout_t layout*)
Set the layout on a button

Parameters

- `btn`: pointer to a button object
- `layout`: a layout from 'lv_cont_layout_t'

static void **`lv_btn_set_fit4`**(*lv_obj_t *btn*, *lv_fit_t left*, *lv_fit_t right*, *lv_fit_t top*, *lv_fit_t bottom*)
Set the fit policy in all 4 directions separately. It tells how to change the button size automatically.

Parameters

- `btn`: pointer to a button object
- `left`: left fit policy from `lv_fit_t`
- `right`: right fit policy from `lv_fit_t`
- `top`: top fit policy from `lv_fit_t`
- `bottom`: bottom fit policy from `lv_fit_t`

static void **`lv_btn_set_fit2`**(*lv_obj_t *btn*, *lv_fit_t hor*, *lv_fit_t ver*)
Set the fit policy horizontally and vertically separately. It tells how to change the button size automatically.

Parameters

- `btn`: pointer to a button object
- `hor`: horizontal fit policy from `lv_fit_t`
- `ver`: vertical fit policy from `lv_fit_t`

static void **`lv_btn_set_fit`**(*lv_obj_t *btn*, *lv_fit_t fit*)
Set the fit policy in all 4 direction at once. It tells how to change the button size automatically.

Parameters

- `btn`: pointer to a button object
- `fit`: fit policy from `lv_fit_t`

lv_btn_state_t **lv_btn_get_state(const** *lv_obj_t *btn***)**
Get the current state of the button

Return the state of the button (from lv_btn_state_t enum) If the button is in disabled state `LV_BTN_STATE_DISABLED` will be ORed to the other button states.

Parameters

- `btn`: pointer to a button object

bool **lv_btn_get_checkable(const** *lv_obj_t *btn***)**
Get the toggle enable attribute of the button

Return true: checkable enabled, false: disabled

Parameters

- `btn`: pointer to a button object

static *lv_layout_t* **lv_btn_get_layout(const** *lv_obj_t *btn***)**
Get the layout of a button

Return the layout from 'lv_cont_layout_t'

Parameters

- `btn`: pointer to button object

static *lv_fit_t* **lv_btn_get_fit_left(const** *lv_obj_t *btn***)**
Get the left fit mode

Return an element of `lv_fit_t`

Parameters

- `btn`: pointer to a button object

static *lv_fit_t* **lv_btn_get_fit_right(const** *lv_obj_t *btn***)**
Get the right fit mode

Return an element of `lv_fit_t`

Parameters

- `btn`: pointer to a button object

static *lv_fit_t* **lv_btn_get_fit_top(const** *lv_obj_t *btn***)**
Get the top fit mode

Return an element of `lv_fit_t`

Parameters

- `btn`: pointer to a button object

static *lv_fit_t* **lv_btn_get_fit_bottom(const** *lv_obj_t *btn***)**
Get the bottom fit mode

Return an element of `lv_fit_t`

Parameters

- `btn`: pointer to a button object

struct lv_btn_ext_t
#include <lv_btn.h> Extended data of button

Public Members

lv_cont_ext_t **cont**
Ext. of ancestor

uint8_t **checkable**
1: Toggle enabled

5.5 Button matrix (lv_btnmatrix)

5.5.1 Overview

The Button Matrix objects can display **multiple buttons** in rows and columns.

The main reasons for wanting to use a button matrix instead of a container and individual button objects are:

- The button matrix is simpler to use for grid-based button layouts.
- The button matrix consumes a lot less memory per button.

5.5.2 Parts and Styles

The Button matrix's main part is called `LV_BTNMATRIX_PART_BG`. It draws a background using the typical background style properties.

`LV_BTNMATRIX_PART_BTN` is virtual part and it refers to the buttons on the button matrix. It also uses all the typical background properties.

The top/bottom/left/right padding values from the background are used to keep some space on the sides. Inner padding is applied between the buttons.

5.5.3 Usage

Button's text

There is a text on each button. To specify them a descriptor string array, called *map*, needs to be used. The map can be set with `lv_btnmatrix_set_map(btnm, my_map)`. The declaration of a map should look like `const char * map[] = {"btn1", "btn2", "btn3", ""}`. Note that **the last element has to be an empty string**!

Use `"\n"` in the map to make **line break**. E.g. `{"btn1", "btn2", "\n", "btn3", ""}`. Each line's buttons have their width calculated automatically.

Control buttons

The **buttons width** can be set relative to the other button in the same line with `lv_btnmatrix_set_btn_width(btnm, btn_id, width)` E.g. in a line with two buttons: *btnA, width = 1* and *btnB, width = 2*, *btnA* will have 33 % width and *btnB* will have 66 % width. It's similar to how the `flex-grow` property works in CSS.

In addition to width, each button can be customized with the following parameters:

- **LV_BTNMATRIX_CTRL_HIDDEN** - make a button hidden (hidden buttons still take up space in the layout, they are just not visible or clickable)
- **LV_BTNMATRIX_CTRL_NO_REPEAT** - disable repeating when the button is long pressed
- **LV_BTNMATRIX_CTRL_DISABLED** - make a button disabled
- **LV_BTNMATRIX_CTRL_CHECKABLE** - enable toggling of a button
- **LV_BTNMATRIX_CTRL_CHECK_STATE** - set the toggle state
- **LV_BTNMATRIX_CTRL_CLICK_TRIG** - if 0, the button will react on press, if 1, will react on release

The set or clear a button's control attribute, use `lv_btnmatrix_set_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)` and `lv_btnmatrix_clear_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)` respectively. More `LV_BTNM_CTRL_...` values can be *Or*ed

The set/clear the same control attribute for all buttons of a button matrix, use `lv_btnmatrix_set_btn_ctrl_all(btnm, btn_id, LV_BTNM_CTRL_...)` and `lv_btnmatrix_clear_btn_ctrl_all(btnm, btn_id, LV_BTNM_CTRL_...)`.

The set a control map for a button matrix (similarly to the map for the text), use `lv_btnmatrix_set_ctrl_map(btnm, ctrl_map)`. An element of `ctrl_map` should look like `ctrl_map[0] = width | LV_BTNM_CTRL_NO_REPEAT | LV_BTNM_CTRL_TGL_ENABLE`. The number of elements should be equal to the number of buttons (excluding newlines characters).

One check

The "One check" feature can be enabled with `lv_btnmatrix_set_one_check(btnm, true)` to allow only one button to be checked (toggled) at once.

Recolor

The **texts** on the button can be **recolored** similarly to the recolor feature for *Label* object. To enable it, use `lv_btnmatrix_set_recolor(btnm, true)`. After that a button with `#FF0000 Red#` text will be red.

Aligning the button's text

To align the text on the buttons, use `lv_btnmatrix_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`. All text items in the button matrix will conform to the alignment proprty as it is set.

Notes

The Button matrix object is very light weighted because the buttons are not created just virtually drawn on the fly. This way, 1 button use only 8 extra bytes instead of the ~100-150 byte size of a normal *Button* object (plus the size of its container and a label for each button).

The disadvantage of this setup is that the ability to style individual buttons to be different from others is limited (aside from the toggling feature). If you require that ability, using individual buttons is very likely to be a better approach.

5.5.4 Events

Besides the Generic events, the following Special events are sent by the button matrices:

- **LV_EVENT_VALUE_CHANGED** - sent when the button is pressed/released or repeated after long press. The event data is set to the ID of the pressed/released button.

Learn more about *Events*.

5.5.5 Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_RIGHT/UP/LEFT/RIGHT** - To navigate among the buttons to select one
- **LV_KEY_ENTER** - To press/release the selected button

Learn more about *Keys*.

5.5.6 Example

5.5.7 API

Typedefs

typedef uint16_t **lv_btnmatrix_ctrl_t**

typedef uint8_t **lv_btnmatrix_part_t**

Enums

enum [anonymous]

Type to store button control bits (disabled, hidden etc.) The first 3 bits are used to store the width

Values:

LV_BTNMATRIX_CTRL_HIDDEN = 0x0008
Button hidden

LV_BTNMATRIX_CTRL_NO_REPEAT = 0x0010
Do not repeat press this button.

LV_BTNMATRIX_CTRL_DISABLED = 0x0020
Disable this button.

LV_BTNMATRIX_CTRL_CHECKABLE = 0x0040
Button *can* be toggled.

LV_BTNMATRIX_CTRL_CHECK_STATE = 0x0080
Button is currently toggled (e.g. checked).

LV_BTNMATRIX_CTRL_CLICK_TRIG = 0x0100
1: Send LV_EVENT_SELECTED on CLICK, 0: Send LV_EVENT_SELECTED on PRESS

enum [anonymous]

Values:

LV_BTNMATRIX_PART_BG

LV_BTNMATRIX_PART_BTN

Functions

LV_EXPORT_CONST_INT(LV_BTNMATRIX_BTN_NONE)

lv_obj_t ***lv_btnmatrix_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a button matrix objects

Return pointer to the created button matrix

Parameters

- `par`: pointer to an object, it will be the parent of the new button matrix
- `copy`: pointer to a button matrix object, if not NULL then the new object will be copied from it

void **lv_btnmatrix_set_map**(*lv_obj_t *btnm*, **const** char **map*[])

Set a new map. Buttons will be created/deleted according to the map. The button matrix keeps a reference to the map and so the string array must not be deallocated during the life of the matrix.

Parameters

- `btnm`: pointer to a button matrix object
- `map`: pointer a string array. The last string has to be: "". Use "\n" to make a line break.

void **lv_btnmatrix_set_ctrl_map**(*lv_obj_t *btnm*, **const** *lv_btnmatrix_ctrl_t ctrl_map*[])

Set the button control map (hidden, disabled etc.) for a button matrix. The control map array will be copied and so may be deallocated after this function returns.

Parameters

- `btnm`: pointer to a button matrix object
- `ctrl_map`: pointer to an array of `lv_btn_ctrl_t` control bytes. The length of the array and position of the elements must match the number and order of the individual buttons (i.e. excludes newline entries). An element of the map should look like e.g.: `ctrl_map[0] = width | LV_BTNMATRIX_CTRL_NO_REPEAT | LV_BTNMATRIX_CTRL_TGL_ENABLE`

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

Set the focused button i.e. visually highlight it.

Parameters

- `btnm`: pointer to button matrix object
- `id`: index of the button to focus(`LV_BTNMATRIX_BTN_NONE` to remove focus)

void **lv_btnmatrix_set_recolor**(**const** *lv_obj_t *btnm*, bool *en*)

Enable recoloring of button's texts

Parameters

- `btnm`: pointer to button matrix object
- `en`: true: enable recoloring; false: disable

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

Set the attributes of a button of the button matrix

Parameters

- `btnm`: pointer to button matrix object
- `btn_id`: 0 based index of the button to modify. (Not counting new lines)

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

Clear the attributes of a button of the button matrix

Parameters

- `btnm`: pointer to button matrix object
- `btn_id`: 0 based index of the button to modify. (Not counting new lines)

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

Set the attributes of all buttons of a button matrix

Parameters

- `btnm`: pointer to a button matrix object
- `ctrl`: attribute(s) to set from `lv_btnmatrix_ctrl_t`. Values can be ORed.

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

Clear the attributes of all buttons of a button matrix

Parameters

- `btnm`: pointer to a button matrix object
- `ctrl`: attribute(s) to set from `lv_btnmatrix_ctrl_t`. Values can be ORed.
- `en`: true: set the attributes; false: clear the attributes

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

Set a single buttons relative width. This method will cause the matrix be regenerated and is a relatively expensive operation. It is recommended that initial width be specified using `lv_btnmatrix_set_ctrl_map` and this method only be used for dynamic changes.

Parameters

- `btnm`: pointer to button matrix object
- `btn_id`: 0 based index of the button to modify.
- `width`: Relative width compared to the buttons in the same row. [1..7]

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

Make the button matrix like a selector widget (only one button may be toggled at a time). `Checkable` must be enabled on the buttons you want to be selected with `lv_btnmatrix_set_ctrl` or `lv_btnmatrix_set_btn_ctrl_all`.

Parameters

- `btnm`: Button matrix object
- `one_chk`: Whether "one check" mode is enabled

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

Get the current map of a button matrix

Return the current map

Parameters

- `btnm`: pointer to a button matrix object

bool **lv_btnmatrix_get_recolor(const** *lv_obj_t *btnm***)**
Check whether the button's text can use recolor or not

Return true: text recolor enable; false: disabled

Parameters

- `btnm`: pointer to button matrix object

uint16_t **lv_btnmatrix_get_active_btn(const** *lv_obj_t *btnm***)**
Get the index of the lastly "activated" button by the user (pressed, released etc) Useful in the the `event_cb` to get the text of the button, check if hidden etc.

Return index of the last released button (LV_BTNMATRIX_BTN_NONE: if unset)

Parameters

- `btnm`: pointer to button matrix object

const char ***lv_btnmatrix_get_active_btn_text(const** *lv_obj_t *btnm***)**
Get the text of the lastly "activated" button by the user (pressed, released etc) Useful in the the `event_cb`

Return text of the last released button (NULL: if unset)

Parameters

- `btnm`: pointer to button matrix object

uint16_t **lv_btnmatrix_get_focused_btn(const** *lv_obj_t *btnm***)**
Get the focused button's index.

Return index of the focused button (LV_BTNMATRIX_BTN_NONE: if unset)

Parameters

- `btnm`: pointer to button matrix object

const char ***lv_btnmatrix_get_btn_text(const** *lv_obj_t *btnm*, uint16_t *btn_id***)**
Get the button's text

Return text of btn_index' button

Parameters

- `btnm`: pointer to button matrix object
- `btn_id`: the index a button not counting new line characters. (The return value of lv_btnmatrix_get_pressed/released)

bool **lv_btnmatrix_get_btn_ctrl(***lv_obj_t *btnm*, uint16_t *btn_id*, *lv_btnmatrix_ctrl_t ctrl***)**
Get the whether a control value is enabled or disabled for button of a button matrix

Return true: long press repeat is disabled; false: long press repeat enabled

Parameters

- `btnm`: pointer to a button matrix object
- `btn_id`: the index a button not counting new line characters. (E.g. the return value of lv_btnmatrix_get_pressed/released)
- `ctrl`: control values to check (ORed value can be used)

bool **lv_btnmatrix_get_one_check(const** *lv_obj_t *btnm***)**
Find whether "one toggle" mode is enabled.

Return whether "one toggle" mode is enabled

Parameters

- `btnm`: Button matrix object

struct lv_btnmatrix_ext_t

Public Members

const char ****map_p**

lv_area_t ***button_areas**

lv_btnmatrix_ctrl_t ***ctrl_bits**

lv_style_list_t **style_btn**

uint16_t **btn_cnt**

uint16_t **btn_id_pr**

uint16_t **btn_id_focused**

uint16_t **btn_id_act**

uint8_t **recolor**

uint8_t **one_check**

5.6 Calendar (lv_calendar)

5.6.1 Overview

The Calendar object is a classic calendar which can:

- highlight the current day
- highlight any user-defined dates
- display the name of the days
- go the next/previous month by button click
- highlight the clicked day

5.6.2 Parts and Styles

The calendar's main part is called `LV_CALENDAR_PART_BG`. It draws a background using the typical background style properties.

Besides the following virtual parts exist:

- `LV_CALENDAR_PART_HEADER` The upper area where the current year and month's name is shown. It also has buttons to move the next/previous month. It uses typical background properties and padding to keep some distance from the background (top, left, right) and the day names (bottom).
- `LV_CALENDAR_PART_DAY_NAMES` Shows the name of the days below the header. It uses the *text* style properties padding to keep some distance from the background (left, right), header (top) and dates (bottom).

- `LV_CALENDAR_PART_DATES` Show the date numbers from 1..28/29/30/31 (depending on current month). Different "state" of the states are drawn according to the states defined in this part:
  - normal dates: drawn with `LV_STATE_DEFAULT` style
  - pressed date: drawn with `LV_STATE_PRESSED` style
  - today: drawn with `LV_STATE_FOCUSED` style
  - highlighted dates: drawn with `LV_STATE_CHECKED` style

5.6.3 Usage

5.6.4 Overview

To set and get dates in the calendar, the `lv_calendar_date_t` type is used which is a structure with `year`, `month` and `day` fields.

Current date

To set the current date (today), use the `lv_calendar_set_today_date(calendar, &today_date)` function.

Shown date

To set the shown date, use `lv_calendar_set_shown_date(calendar, &shown_date)`;

Highlighted days

The list of highlighted dates should be stored in a `lv_calendar_date_t` array loaded by `lv_calendar_set_highlighted_dates(calendar, &highlighted_dates)`.Only the arrays pointer will be saved so the array should be a static or global variable.

Name of the days

The name of the days can be adjusted with `lv_calendar_set_day_names(calendar, day_names)` where `day_names` looks like `const char * day_names[7] = {"Su", "Mo", ...};`

Name of the months

Similarly to `day_names`, the name of the month can be set with `lv_calendar_set_month_names(calendar, month_names_array)`.

5.6.5 Events

Besides the Generic events, the following Special events are sent by the calendars: **LV_EVENT_VALUE_CHANGED** is sent when the current month has changed.

In *Input device related* events, `lv_calendar_get_pressed_date(calendar)` tells which day is currently being pressed or return `NULL` if no date is pressed.

5.6.6 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.6.7 Example

5.6.8 API

Typedefs

typedef uint8_t **lv_calendar_part_t**

Enums

enum [anonymous]

Calendar parts

Values:

LV_CALENDAR_PART_BG

Background and "normal" date numbers style

LV_CALENDAR_PART_HEADER

LV_CALENDAR_PART_DAY_NAMES

Calendar header style

LV_CALENDAR_PART_DATE

Day name style

Functions

lv_obj_t ***lv_calendar_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a calendar objects

Return pointer to the created calendar

Parameters

- `par`: pointer to an object, it will be the parent of the new calendar
- `copy`: pointer to a calendar object, if not NULL then the new object will be copied from it

void **lv_calendar_set_today_date**(*lv_obj_t *calendar*, *lv_calendar_date_t *today*)

Set the today's date

Parameters

- `calendar`: pointer to a calendar object
- `today`: pointer to an *`lv_calendar_date_t`* variable containing the date of today. The value will be saved it can be local variable too.

void **lv_calendar_set_showed_date**(*lv_obj_t *calendar*, *lv_calendar_date_t *showed*)

Set the currently showed

Parameters

- `calendar`: pointer to a calendar object
- `showed`: pointer to an *`lv_calendar_date_t`* variable containing the date to show. The value will be saved it can be local variable too.

void **`lv_calendar_set_highlighted_dates`**(*lv_obj_t *calendar*, *lv_calendar_date_t highlighted*[], uint16_t *date_num*)

Set the the highlighted dates

Parameters

- `calendar`: pointer to a calendar object
- `highlighted`: pointer to an *`lv_calendar_date_t`* array containing the dates. ONLY A POINTER WILL BE SAVED! CAN'T BE LOCAL ARRAY.
- `date_num`: number of dates in the array

void **`lv_calendar_set_day_names`**(*lv_obj_t *calendar*, **const** char ***day_names*)

Set the name of the days

Parameters

- `calendar`: pointer to a calendar object
- `day_names`: pointer to an array with the names. E.g. `const char * days[7] = {"Sun", "Mon", ...}` Only the pointer will be saved so this variable can't be local which will be destroyed later.

void **`lv_calendar_set_month_names`**(*lv_obj_t *calendar*, **const** char ***month_names*)

Set the name of the month

Parameters

- `calendar`: pointer to a calendar object
- `month_names`: pointer to an array with the names. E.g. `const char * days[12] = {"Jan", "Feb", ...}` Only the pointer will be saved so this variable can't be local which will be destroyed later.

lv_calendar_date_t ***`lv_calendar_get_today_date`**(**const** *lv_obj_t *calendar*)

Get the today's date

Return return pointer to an *`lv_calendar_date_t`* variable containing the date of today.

Parameters

- `calendar`: pointer to a calendar object

lv_calendar_date_t ***`lv_calendar_get_showed_date`**(**const** *lv_obj_t *calendar*)

Get the currently showed

Return pointer to an *`lv_calendar_date_t`* variable containing the date is being shown.

Parameters

- `calendar`: pointer to a calendar object

lv_calendar_date_t ***`lv_calendar_get_pressed_date`**(**const** *lv_obj_t *calendar*)

Get the the pressed date.

Return pointer to an *`lv_calendar_date_t`* variable containing the pressed date. `NULL` if not date pressed (e.g. the header)

Parameters

- `calendar`: pointer to a calendar object

lv_calendar_date_t ***lv_calendar_get_highlighted_dates(const** *lv_obj_t* ***calendar)**
Get the the highlighted dates

Return pointer to an `lv_calendar_date_t` array containing the dates.

Parameters

- `calendar`: pointer to a calendar object

uint16_t **lv_calendar_get_highlighted_dates_num(const** *lv_obj_t* ***calendar)**
Get the number of the highlighted dates

Return number of highlighted days

Parameters

- `calendar`: pointer to a calendar object

const char ****lv_calendar_get_day_names(const** *lv_obj_t* ***calendar)**
Get the name of the days

Return pointer to the array of day names

Parameters

- `calendar`: pointer to a calendar object

const char ****lv_calendar_get_month_names(const** *lv_obj_t* ***calendar)**
Get the name of the month

Return pointer to the array of month names

Parameters

- `calendar`: pointer to a calendar object

struct lv_calendar_date_t
#include <lv_calendar.h> Represents a date on the calendar object (platform-agnostic).

Public Members

uint16_t **year**

int8_t **month**

int8_t **day**

struct lv_calendar_ext_t

Public Members

lv_calendar_date_t **today**

lv_calendar_date_t **showed_date**

lv_calendar_date_t ***highlighted_dates**

int8_t **btn_pressing**

uint16_t **highlighted_dates_num**

lv_calendar_date_t **pressed_date**

const char ****day_names**

const char ****month_names**

lv_style_list_t **style_header**

lv_style_list_t **style_day_names**

lv_style_list_t **style_date_nums**

5.7 Canvas (lv_canvas)

5.7.1 Overview

A Canvas inherites from *Image* where the user can draw anything. Rectangles, texts, images, lines arcs can be drawn here using lvgl's drawing engine. Besides some "effects" can be applied as well like rotation, zoom and blur.

5.7.2 Parts and Styles

The Canvas has on one main part called `LV_CANVAS_PART_MAIN` and only the *image_recolor* property is used to give a color to `LV_IMG_CF_ALPHA_1/2/4/8BIT` images.

5.7.3 Usage

Buffer

The Canvas needs a buffer which stores the drawn image. To assign a buffer to a Canvas, use `lv_canvas_set_buffer(canvas, buffer, width, height, LV_IMG_CF_...)`. Where `buffer` is a static buffer (not just a local variable) to hold the image of the canvas. For example, `static lv_color_t buffer[LV_CANVAS_BUF_SIZE_TRUE_COLOR(width, height)]`. `LV_CANVAS_BUF_SIZE_...` macros help to determine the size of the buffer with different color formats.

The canvas supports all the built-in color formats like `LV_IMG_CF_TRUE_COLOR` or `LV_IMG_CF_INDEXED_2BIT`. See the full list in the Color formats section.

Palette

For `LV_IMG_CF_INDEXED_...` color formats, a palette needs to be initialized with `lv_canvas_set_palette(canvas, 3, LV_COLOR_RED)`. It sets pixels with *index=3* to red.

Drawing

To set a pixel on the canvas, use `lv_canvas_set_px(canvas, x, y, LV_COLOR_RED)`. With `LV_IMG_CF_INDEXED_...` or `LV_IMG_CF_ALPHA_...`, the index of the color or the alpha value needs to be passed as color. E.g. `lv_color_t c; c.full = 3;`

`lv_canvas_fill_bg(canvas, LV_COLOR_BLUE, LV_OPA_50)` fills the whole canvas to blue with 50% opacity. Note that, if the current color format doesn't support colors (e.g. `LV_IMG_CF_ALPHA_2BIT`) teh color will be ignored. Similarly, if opacity is not supported (e.g. `LV_IMG_CF_TRUE_COLOR`) it will be ignored.

An array of pixels can be copied to the canvas with `lv_canvas_copy_buf(canvas, buffer_to_copy, x, y, width, height)`. The color format of the buffer and the canvas need to match.

To draw something to the canvas use

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

`draw_dsc` is a `lv_draw_rect/label/img/line_dsc_t` variable which should be first initialized with `lv_draw_rect/label/img/line_dsc_init()` function and then it's filed should be modified with the desired colors and other values.

The draw function can draw to any color format. For example, it's possible to draw a text to an `LV_IMG_VF_ALPHA_8BIT` canvas and use the result image as a mask in *lv_objmask* later.

Transformations

`lv_canvas_transform()` can be used to rotate and/or scale the image of an image and store the result on the canvas. The function needs the following parameters:

- `canvas` pointer to a canvas object to store the result of the transformation.
- `img pointer` to an image descriptor to transform. Can be the image descriptor of an other canvas too (`lv_canvas_get_img()`).
- `angle` the angle of rotation (0..3600), 0.1 deg resolution
- `zoom` zoom factor (256 no zoom, 512 double size, 128 half size);
- `offset_x` offset X to tell where to put the result data on destination canvas
- `offset_y` offset X to tell where to put the result data on destination canvas
- `pivot_x` pivot X of rotation. Relative to the source canvas. Set to `source width / 2` to rotate around the center
- `pivot_y` pivot Y of rotation. Relative to the source canvas. Set to `source height / 2` to rotate around the center
- `antialias` true: apply anti-aliasing during the transformation. Looks better but slower.

Note that a canvas can't be rotated on itself. You need a source and destination canvas or image.

5.7.4 Blur

A given area of the canvas can be blurred horizontally with `lv_canvas_blur_hor(canvas, &area, r)` ot vertically with `lv_canvas_blur_ver(canvas, &area, r)`. `r` is the radius of the blur (greater value means more intensive burring). `area` is the area where the blur should be applied (interpreted relative to the canvas)

5.7.5 Events

As default the clicking of a canvas is disabled (inherited by *Image*) and therefore no events are generated.

If clicking is enabled (`lv_obj_set_click(canvas, true)`) only the Generic events are sent by the object type.

Learn more about *Events*.

5.7.6 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.7.7 Example

5.7.8 API

Typedefs

typedef uint8_t **lv_canvas_part_t**

Enums

enum [**anonymous**]

Values:

LV_CANVAS_PART_MAIN

Functions

lv_obj_t ***lv_canvas_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a canvas object

Return pointer to the created canvas

Parameters

- `par`: pointer to an object, it will be the parent of the new canvas
- `copy`: pointer to a canvas object, if not NULL then the new object will be copied from it

void **lv_canvas_set_buffer**(*lv_obj_t *canvas*, void *buf*, lv_coord_t *w*, lv_coord_t *h*, lv_img_cf_t *cf*)

Set a buffer for the canvas.

Parameters

- `buf`: a buffer where the content of the canvas will be. The required size is (lv_img_color_format_get_px_size(cf) * w * h) / 8) It can be allocated with `lv_mem_alloc()` or it can be statically allocated array (e.g. static lv_color_t buf[100*50]) or it can be an address in RAM or external SRAM
- `canvas`: pointer to a canvas object
- `w`: width of the canvas

- `h`: height of the canvas
- `cf`: color format. `LV_IMG_CF_...`

void **`lv_canvas_set_px`**(*lv_obj_t *canvas*, lv_coord_t *x*, lv_coord_t *y*, *lv_color_t c*)

Set the color of a pixel on the canvas

Parameters

- `canvas`:
- `x`: x coordinate of the point to set
- `y`: x coordinate of the point to set
- `c`: color of the point

void **`lv_canvas_set_palette`**(*lv_obj_t *canvas*, uint8_t *id*, *lv_color_t c*)

Set the palette color of a canvas with index format. Valid only for `LV_IMG_CF_INDEXED1/2/4/8`

Parameters

- `canvas`: pointer to canvas object
- `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

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

Get the color of a pixel on the canvas

Return color of the point

Parameters

- `canvas`:
- `x`: x coordinate of the point to set
- `y`: x coordinate of the point to set

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

Get the image of the canvas as a pointer to an `lv_img_dsc_t` variable.

Return pointer to the image descriptor.

Parameters

- `canvas`: pointer to a canvas object

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

Copy a buffer to the canvas

Parameters

- `canvas`: pointer to a canvas object
- `to_copy`: buffer to copy. The color format has to match with the canvas's buffer color format
- `x`: left side of the destination position

- `y`: top side of the destination position
- `w`: width of the buffer to copy
- `h`: height of the buffer to copy

void **`lv_canvas_transform`**(*lv_obj_t *canvas*, lv_img_dsc_t *img*, int16_t *angle*, uint16_t *zoom*, lv_coord_t *offset_x*, lv_coord_t *offset_y*, int32_t *pivot_x*, int32_t *pivot_y*, bool *antialias*)

Transform and image and store the result on a canvas.

Parameters

- `canvas`: pointer to a canvas object to store the result of the transformation.
- `img`: pointer to an image descriptor to transform. Can be the image descriptor of an other canvas too (*`lv_canvas_get_img()`*).
- `angle`: the angle of rotation (0..3600), 0.1 deg resolution
- `zoom`: zoom factor (256 no zoom);
- `offset_x`: offset X to tell where to put the result data on destination canvas
- `offset_y`: offset X to tell where to put the result data on destination canvas
- `pivot_x`: pivot X of rotation. Relative to the source canvas Set to `source width / 2` to rotate around the center
- `pivot_y`: pivot Y of rotation. Relative to the source canvas Set to `source height / 2` to rotate around the center
- `antialias`: apply anti-aliasing during the transformation. Looks better but slower.

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

Apply horizontal blur on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `r`: radius of the blur

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

Apply vertical blur on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `area`: the area to blur. If `NULL` the whole canvas will be blurred.
- `r`: radius of the blur

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

Fill the canvas with color

Parameters

- `canvas`: pointer to a canvas
- `color`: the background color

void **`lv_canvas_draw_rect`**(*lv_obj_t *canvas*, lv_coord_t *x*, lv_coord_t *y*, lv_coord_t *w*, lv_coord_t *h*, lv_draw_rect_dsc_t *rect_dsc*)

Draw a rectangle on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `x`: left coordinate of the rectangle
- `y`: top coordinate of the rectangle
- `w`: width of the rectangle
- `h`: height of the rectangle
- `style`: style of the rectangle (`body` properties are used except `padding`)

void **lv_canvas_draw_text**(*lv_obj_t *canvas*, lv_coord_t *x*, lv_coord_t *y*, lv_coord_t *max_w*, lv_draw_label_dsc_t **label_draw_dsc*, **const** char **txt*, *lv_label_align_t align*)

Draw a text on the canvas.

Parameters

- `canvas`: pointer to a canvas object
- `x`: left coordinate of the text
- `y`: top coordinate of the text
- `max_w`: max width of the text. The text will be wrapped to fit into this size
- `style`: style of the text (`text` properties are used)
- `txt`: text to display
- `align`: align of the text (`LV_LABEL_ALIGN_LEFT/RIGHT/CENTER`)

void **lv_canvas_draw_img**(*lv_obj_t *canvas*, lv_coord_t *x*, lv_coord_t *y*, **const** void **src*, lv_draw_img_dsc_t **img_draw_dsc*)

Draw an image on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `src`: image source. Can be a pointer an `lv_img_dsc_t` variable or a path an image.
- `style`: style of the image (`image` properties are used)

void **lv_canvas_draw_line**(*lv_obj_t *canvas*, **const** lv_point_t *points*[], uint32_t *point_cnt*, lv_draw_line_dsc_t **line_draw_dsc*)

Draw a line on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `points`: point of the line
- `point_cnt`: number of points
- `style`: style of the line (`line` properties are used)

void **lv_canvas_draw_polygon**(*lv_obj_t *canvas*, **const** lv_point_t *points*[], uint32_t *point_cnt*, lv_draw_rect_dsc_t **poly_draw_dsc*)

Draw a polygon on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `points`: point of the polygon
- `point_cnt`: number of points

- `style`: style of the polygon (`body.main_color` and `body.opa` is used)

void **`lv_canvas_draw_arc`**(*lv_obj_t *canvas*, lv_coord_t *x*, lv_coord_t *y*, lv_coord_t *r*, int32_t *start_angle*, int32_t *end_angle*, lv_draw_line_dsc_t **arc_draw_dsc*)

Draw an arc on the canvas

Parameters

- `canvas`: pointer to a canvas object
- `x`: origo x of the arc
- `y`: origo y of the arc
- `r`: radius of the arc
- `start_angle`: start angle in degrees
- `end_angle`: end angle in degrees
- `style`: style of the polygon (`body.main_color` and `body.opa` is used)

`struct lv_canvas_ext_t`

Public Members

lv_img_ext_t **`img`**

lv_img_dsc_t **`dsc`**

5.8 Checkbox (lv_cb)

5.8.1 Overview

The Checkbox objects are built from a *Button* background which contains an also Button *bullet* and a *Label* to realize a classical checkbox.

5.8.2 Parts and Styles

The Check box's main part is called `LV_CHECKBOX_PART_BG`. It's a container for a "bullet" and a text next to it. The background uses all the typical background style properties.

The bullet is real *lv_obj* object and can be referred with `LV_CHACKBOX_PART_BULLET`. The bullet automatically inherits the state of the background. So the background is pressed the bullet goes to pressed state as well. The bullet also uses all the typical background style properties.

There is not dedicated part for the label. Its styles can be set in the background's styles because the *text* styles properties are always inherited.

5.8.3 Usage

Text

The text can be modified by the `lv_checkbox_set_text(cb, "New text")` function. It will dynamically allocate the text.

To set a static text, use `lv_checkbox_set_static_text(cb, txt)`. This way, only a pointer of `txt` will be stored and it shouldn't be deallocated while the checkbox exists.

Check/Uncheck

You can manually check / un-check the Checkbox via `lv_checkbox_set_checked(cb, true/false)`. Setting `true` will check the checkbox and `false` will un-check the checkbox.

Disabled

To make the Checkbox disabled, use `lv_checkbox_set_disabled(cb, true)`.

5.8.4 Events

Besides the Generic events the following Special events are sent by the Checkboxes:

- **LV_EVENT_VALUE_CHANGED** - sent when the checkbox is toggled.

Note that, the generic input device-related events (like `LV_EVENT_PRESSED`) are sent in the inactive state too. You need to check the state with `lv_cb_is_inactive(cb)` to ignore the events from inactive Checkboxes.

Learn more about *Events*.

5.8.5 Keys

The following *Keys* are processed by the 'Buttons':

- **LV_KEY_RIGHT/UP** - Go to toggled state if toggling is enabled
- **LV_KEY_LEFT/DOWN** - Go to non-toggled state if toggling is enabled

Note that, as usual, the state of `LV_KEY_ENTER` is translated to `LV_EVENT_PRESSED/PRESSING/RELEASED` etc.

Learn more about *Keys*.

5.8.6 Example

5.8.7 API

Typedefs

typedef uint8_t **lv_checkbox_style_t**

Enums

enum [anonymous]

Checkbox styles.

Values:

LV_CHECKBOX_PART_BG = *LV_BTN_PART_MAIN*
Style of object background.

_LV_CHECKBOX_PART_VIRTUAL_LAST

LV_CHECKBOX_PART_BULLET = *_LV_BTN_PART_REAL_LAST*
Style of box (released).

_LV_CHECKBOX_PART_REAL_LAST

Functions

lv_obj_t ***lv_checkbox_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a check box objects

Return pointer to the created check box

Parameters

- `par`: pointer to an object, it will be the parent of the new check box
- `copy`: pointer to a check box object, if not NULL then the new object will be copied from it

void **lv_checkbox_set_text**(*lv_obj_t *cb*, **const** char **txt*)
Set the text of a check box. `txt` will be copied and may be deallocated after this function returns.

Parameters

- `cb`: pointer to a check box
- `txt`: the text of the check box. NULL to refresh with the current text.

void **lv_checkbox_set_text_static**(*lv_obj_t *cb*, **const** char **txt*)
Set the text of a check box. `txt` must not be deallocated during the life of this checkbox.

Parameters

- `cb`: pointer to a check box
- `txt`: the text of the check box. NULL to refresh with the current text.

static void **lv_checkbox_set_checked**(*lv_obj_t *cb*, bool *checked*)
Set the state of the check box

Parameters

- `cb`: pointer to a check box object
- `checked`: true: make the check box checked; false: make it unchecked

static void **lv_checkbox_set_disabled**(*lv_obj_t *cb*)
Make the check box inactive (disabled)

Parameters

- `cb`: pointer to a check box object

const char ***lv_checkbox_get_text**(**const** *lv_obj_t *cb*)
Get the text of a check box

Return pointer to the text of the check box

Parameters

- `cb`: pointer to check box object

static bool **lv_checkbox_is_checked(const** *lv_obj_t* ***cb)**

Get the current state of the check box

Return true: checked; false: not checked

Parameters

- `cb`: pointer to a check box object

static bool **lv_checkbox_is_inactive(const** *lv_obj_t* ***cb)**

Get whether the check box is inactive or not.

Return true: inactive; false: not inactive

Parameters

- `cb`: pointer to a check box object

struct lv_checkbox_ext_t

Public Members

lv_btn_ext_t **bg_btn**

lv_obj_t ***bullet**

lv_obj_t ***label**

5.9 Chart (lv_chart)

5.9.1 Overview

Charts are basic object to visualize data points. It support *Line* charts (connect points with lines and/or draw points on them) and *Column* chart.

Chart also support division lines, 2 y axis, axis ticks, and texts on ticks.

5.9.2 Parts and Styles

The Chart's main part is called `LV_CHART_PART_BG` and it uses all the typical background properties. The *text* style properties determine the style of the axis texts and the *line* properties determine ticks' style. *Padding* values add some space on the sides thus it makes *series area* smaller. Padding also can be used to make space for axis texts and ticks.

The background of the series is called `LV_CHART_PART_SERIES_BG` and it's placed on the main background. The division lines, and series data is drawn on this part. Besides the typical background style properties the *line* style properties are used by the division lines. The *padding* values tells the space between the this part and the axis texts.

The style of the series can be referenced by `LV_CHART_PART_SERIES`. In case of column type the following properties are used:

- *radius*: radius of the bars
- *padding_inner*: space between the columns of the same x coordinate

In case of Line type these properties are used:

- *line properties* to describe the lines

- *size* radius of the points
- *bg_opa*: the overall opacity of the area below the lines
- *bg_main_stop*: % of *bg_opa* at the top to create an alpha fade (0: transparent at the top, 255: *bg_opa* at the top)
- *bg_grad_stop*: % of *bg_opa* at the bottom to create an alpha fade (0: transparent at the bottom, 255: *bg_opa* at the top)
- *bg_drag_dir*: should be `LV_GRAD_DIR_VER` to allow alpha fading with *bg_main_stop* and *bg_grad_stop*

5.9.3 Usage

Data series

You can add any number of series to the charts by `lv_chart_add_series(chart, color)`. It allocates data for a `lv_chart_series_t` structure which contains the chosen `color` and an array for the data points.

Series' type

The following **data display types** exist:

- **LV_CHART_TYPE_NONE** - Do not display any data. It can be used to hide a series.
- **LV_CHART_TYPE_LINE** - Draw lines between the points.
- **LV_CHART_TYPE_COLUMN** - Draw columns.

You can specify the display type with `lv_chart_set_type(chart, LV_CHART_TYPE_...)`. The types can be 'OR'ed (like `LV_CHART_TYPE_LINE`).

Modify the data

You have several options to set the data of series:

1. Set the values manually in the array like `ser1->points[3] = 7` and refresh the chart with `lv_chart_refresh(chart)`.
2. Use the `lv_chart_set_next(chart, ser, value)`.
3. Initialize all points to a given value with: `lv_chart_init_points(chart, ser, value)`.
4. Set all points from an array with: `lv_chart_set_points(chart, ser, value_array)`.

Use `LV_CHART_POINT_DEF` as value to make the library skip drawing that point, column, or line segment.

Update modes

`lv_chart_set_next` can behave in two ways depending on *update mode*:

- **LV_CHART_UPDATE_MODE_SHIFT** - Shift old data to the left and add the new one o the right.
- **LV_CHART_UPDATE_MODE_CIRCULAR** - Circularly add the new data (Like an ECG diagram).

The update mode can be changed with `lv_chart_set_update_mode(chart, LV_CHART_UPDATE_MODE_...)`.

Number of points

The number of points in the series can be modified by `lv_chart_set_point_count(chart, point_num)`. The default value is 10.

Vertical range

You can specify the minimum and maximum values in y-direction with `lv_chart_set_range(chart, y_min, y_max)`. The value of the points will be scaled proportionally. The default range is: 0..100.

Division lines

The number of horizontal and vertical division lines can be modified by `lv_chart_set_div_line_count(chart, hdiv_num, vdiv_num)`. The default settings are 3 horizontal and 5 vertical division lines.

Tick marks and labels

Ticks and labels can be added to the axis.

`lv_chart_set_x_tick_text(chart, list_of_values, num_tick_marks, LV_CHART_AXIS_...)` set the ticks and texts on x axis. `list_of_values` is a string with `'\n'` terminated text (expect the last) with text for the ticks. E.g. `const char * list_of_values = "first\nsec\nthird"`. `list_of_values` can be `NULL`. If `list_of_values` is set then `num_tick_marks` tells the number of ticks between two labels. If `list_of_values` is `NULL` then it specifies the total number of ticks.

Major tick lines are drawn where text is placed, and *minor tick lines* are drawn elsewhere. `lv_chart_set_x_tick_length(chart, major_tick_len, minor_tick_len)` sets the length of tick lines on the x-axis.

The same functions exists for the y axis too: `lv_chart_set_y_tick_text` and `lv_chart_set_y_tick_length`.

5.9.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.9.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.9.6 Example

5.9.7 API

Typedefs

typedef uint8_t **lv_chart_type_t**

typedef uint8_t **lv_chart_update_mode_t**

typedef uint8_t **lv_chart_axis_options_t**

Enums

enum [**anonymous**]
Chart types

Values:

LV_CHART_TYPE_NONE = 0x00
Don't draw the series

LV_CHART_TYPE_LINE = 0x01
Connect the points with lines

LV_CHART_TYPE_COLUMN = 0x02
Draw columns

enum [**anonymous**]
Chart update mode for `lv_chart_set_next`

Values:

LV_CHART_UPDATE_MODE_SHIFT
Shift old data to the left and add the new one o the right

LV_CHART_UPDATE_MODE_CIRCULAR
Add the new data in a circular way

enum [**anonymous**]
Data of axis

Values:

LV_CHART_AXIS_SKIP_LAST_TICK = 0x00
don't draw the last tick

LV_CHART_AXIS_DRAW_LAST_TICK = 0x01
draw the last tick

LV_CHART_AXIS_INVERSE_LABELS_ORDER = 0x02
draw tick labels in an inversed order

enum [**anonymous**]
Values:

LV_CHART_PART_BG = *LV_OBJ_PART_MAIN*

LV_CHART_PART_SERIES_BG = *_LV_OBJ_PART_VIRTUAL_LAST*

LV_CHART_PART_SERIES

Functions

LV_EXPORT_CONST_INT(LV_CHART_POINT_DEF)

LV_EXPORT_CONST_INT(LV_CHART_TICK_LENGTH_AUTO)

lv_obj_t ***lv_chart_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a chart background objects

Return pointer to the created chart background

Parameters

- `par`: pointer to an object, it will be the parent of the new chart background
- `copy`: pointer to a chart background object, if not NULL then the new object will be copied from it

lv_chart_series_t ***lv_chart_add_series**(*lv_obj_t *chart*, *lv_color_t color*)

Allocate and add a data series to the chart

Return pointer to the allocated data series

Parameters

- `chart`: pointer to a chart object
- `color`: color of the data series

void **lv_chart_clear_serie**(*lv_obj_t *chart*, *lv_chart_series_t *serie*)

Clear the point of a series

Parameters

- `chart`: pointer to a chart object
- `serie`: pointer to the chart's series to clear

void **lv_chart_set_div_line_count**(*lv_obj_t *chart*, uint8_t *hdiv*, uint8_t *vdiv*)

Set the number of horizontal and vertical division lines

Parameters

- `chart`: pointer to a graph background object
- `hdiv`: number of horizontal division lines
- `vdiv`: number of vertical division lines

void **lv_chart_set_range**(*lv_obj_t *chart*, lv_coord_t *ymin*, lv_coord_t *ymax*)

Set the minimal and maximal y values

Parameters

- `chart`: pointer to a graph background object
- `ymin`: y minimum value
- `ymax`: y maximum value

void **lv_chart_set_type**(*lv_obj_t *chart*, *lv_chart_type_t type*)

Set a new type for a chart

Parameters

- `chart`: pointer to a chart object
- `type`: new type of the chart (from 'lv_chart_type_t' enum)

void **lv_chart_set_point_count**(*lv_obj_t *chart*, uint16_t *point_cnt*)
Set the number of points on a data line on a chart

Parameters

- `chart`: pointer r to chart object
- `point_cnt`: new number of points on the data lines

void **lv_chart_init_points**(*lv_obj_t *chart*, *lv_chart_series_t *ser*, lv_coord_t *y*)
Initialize all data points with a value

Parameters

- `chart`: pointer to chart object
- `ser`: pointer to a data series on 'chart'
- `y`: the new value for all points

void **lv_chart_set_points**(*lv_obj_t *chart*, *lv_chart_series_t *ser*, lv_coord_t *y_array*[])
Set the value of points from an array

Parameters

- `chart`: pointer to chart object
- `ser`: pointer to a data series on 'chart'
- `y_array`: array of 'lv_coord_t' points (with 'points count' elements)

void **lv_chart_set_next**(*lv_obj_t *chart*, *lv_chart_series_t *ser*, lv_coord_t *y*)
Shift all data right and set the most right data on a data line

Parameters

- `chart`: pointer to chart object
- `ser`: pointer to a data series on 'chart'
- `y`: the new value of the most right data

void **lv_chart_set_update_mode**(*lv_obj_t *chart*, *lv_chart_update_mode_t update_mode*)
Set update mode of the chart object.

Parameters

- `chart`: pointer to a chart object
- `update`: mode

void **lv_chart_set_x_tick_length**(*lv_obj_t *chart*, uint8_t *major_tick_len*, uint8_t *minor_tick_len*)
Set the length of the tick marks on the x axis

Parameters

- `chart`: pointer to the chart
- `major_tick_len`: the length of the major tick or `LV_CHART_TICK_LENGTH_AUTO` to set automatically (where labels are added)
- `minor_tick_len`: the length of the minor tick, `LV_CHART_TICK_LENGTH_AUTO` to set automatically (where no labels are added)

void **lv_chart_set_y_tick_length**(*lv_obj_t *chart*, uint8_t *major_tick_len*, uint8_t *minor_tick_len*)
Set the length of the tick marks on the y axis

Parameters

- `chart`: pointer to the chart
- `major_tick_len`: the length of the major tick or `LV_CHART_TICK_LENGTH_AUTO` to set automatically (where labels are added)
- `minor_tick_len`: the length of the minor tick, `LV_CHART_TICK_LENGTH_AUTO` to set automatically (where no labels are added)

void **lv_chart_set_secondary_y_tick_length**(*lv_obj_t *chart*, uint8_t *major_tick_len*, uint8_t *minor_tick_len*)

Set the length of the tick marks on the secondary y axis

Parameters

- `chart`: pointer to the chart
- `major_tick_len`: the length of the major tick or `LV_CHART_TICK_LENGTH_AUTO` to set automatically (where labels are added)
- `minor_tick_len`: the length of the minor tick, `LV_CHART_TICK_LENGTH_AUTO` to set automatically (where no labels are added)

void **lv_chart_set_x_tick_texts**(*lv_obj_t *chart*, **const** char **list_of_values*, uint8_t *num_tick_marks*, *lv_chart_axis_options_t options*)

Set the x-axis tick count and labels of a chart

Parameters

- `chart`: pointer to a chart object
- `list_of_values`: list of string values, terminated with , except the last
- `num_tick_marks`: if list_of_values is NULL: total number of ticks per axis else number of ticks between two value labels
- `options`: extra options

void **lv_chart_set_secondary_y_tick_texts**(*lv_obj_t *chart*, **const** char **list_of_values*, uint8_t *num_tick_marks*, *lv_chart_axis_options_t options*)

Set the secondary y-axis tick count and labels of a chart

Parameters

- `chart`: pointer to a chart object
- `list_of_values`: list of string values, terminated with , except the last
- `num_tick_marks`: if list_of_values is NULL: total number of ticks per axis else number of ticks between two value labels
- `options`: extra options

void **lv_chart_set_y_tick_texts**(*lv_obj_t *chart*, **const** char **list_of_values*, uint8_t *num_tick_marks*, *lv_chart_axis_options_t options*)

Set the y-axis tick count and labels of a chart

Parameters

- `chart`: pointer to a chart object
- `list_of_values`: list of string values, terminated with , except the last
- `num_tick_marks`: if list_of_values is NULL: total number of ticks per axis else number of ticks between two value labels

- `options`: extra options

lv_chart_type_t **lv_chart_get_type(const** *lv_obj_t *chart*)
Get the type of a chart

Return type of the chart (from 'lv_chart_t' enum)

Parameters

- `chart`: pointer to chart object

uint16_t **lv_chart_get_point_count(const** *lv_obj_t *chart*)
Get the data point number per data line on chart

Return point number on each data line

Parameters

- `chart`: pointer to chart object

void **lv_chart_refresh(***lv_obj_t *chart*)
Refresh a chart if its data line has changed

Parameters

- `chart`: pointer to chart object

struct lv_chart_series_t

Public Members

lv_coord_t ***points**

lv_color_t **color**

uint16_t **start_point**

struct lv_chart_axis_cfg_t

Public Members

const char ***list_of_values**

lv_chart_axis_options_t **options**

uint8_t **num_tick_marks**

uint8_t **major_tick_len**

uint8_t **minor_tick_len**

struct lv_chart_ext_t

Public Members

lv_ll_t **series_ll**

lv_coord_t **ymin**

lv_coord_t **ymax**

uint8_t **hdiv_cnt**

uint8_t **vdiv_cnt**

uint16_t **point_cnt**

lv_style_list_t **style_series_bg**

lv_style_list_t **style_series**

lv_chart_type_t **type**

lv_chart_axis_cfg_t **y_axis**

lv_chart_axis_cfg_t **x_axis**

lv_chart_axis_cfg_t **secondary_y_axis**

uint8_t **update_mode**

5.10 Container (lv_cont)

5.10.1 Overview

The containers are essentially a **basic object** with some special features.

Layout

You can apply a layout on the containers to automatically order their children. The layout spacing comes from `style.body.padding. ...` properties. The possible layout options:

- **LV_LAYOUT_OFF** - Do not align the children.
- **LV_LAYOUT_CENTER** - Align children to the center in column and keep `padding.inner` space between them.
- **LV_LAYOUT_COL_** - Align children in a left-justified column. Keep `padding.left` space on the left, `pad.top` space on the top and `padding.inner` space between the children.
- **LV_LAYOUT_COL_M** - Align children in centered column. Keep `padding.top` space on the top and `padding.inner` space between the children.
- **LV_LAYOUT_COL_R** - Align children in a right-justified column. Keep `padding.right` space on the right, `padding.top` space on the top and `padding.inner` space between the children.
- **LV_LAYOUT_ROW_T** - Align children in a top justified row. Keep `padding.left` space on the left, `padding.top` space on the top and `padding.inner` space between the children.
- **LV_LAYOUT_ROW_M** - Align children in centered row. Keep `padding.left` space on the left and `padding.inner` space between the children.
- **LV_LAYOUT_ROW_B** - Align children in a bottom justified row. Keep `padding.left` space on the left, `padding.bottom` space on the bottom and `padding.inner` space between the children.
- **LV_LAYOUT_PRETTY** - Put as many objects as possible in a row (with at least `padding.inner` space and `padding.left/right` space on the sides). Divide the space in each line equally between the children. Keep `padding.top` space on the top and `pad.inner` space between the lines.
- **LV_LAYOUT_GRID** - Similar to `LV_LAYOUT_PRETTY` but not divide horizontal space equally just let `padding.left/right` on the edges and `padding.inner` space between the elements.

Autofit

Container have an autofit feature which can automatically change the size of the container according to its children and/or parent. The following options exist:

- **LV_FIT_NONE** - Do not change the size automatically.
- **LV_FIT_TIGHT** - Shrink-wrap the container around all of its children, while keeping `padding.top/bottom/left/right` space on the edges.
- **LV_FIT_FLOOD** - Set the size to the parent's size minus `padding.top/bottom/left/right` (from the parent's style) space.
- **LV_FIT_FILL** - Use `LV_FIT_FLOOD` while smaller than the parent and `LV_FIT_TIGHT` when larger. It will ensure that the container is, at minimum, the size of its parent.

To set the auto fit mode for all directions, use `lv_cont_set_fit(cont, LV_FIT_...)`. To use different auto fit horizontally and vertically, use `lv_cont_set_fit2(cont, hor_fit_type, ver_fit_type)`. To use different auto fit in all 4 directions, use `lv_cont_set_fit4(cont, left_fit_type, right_fit_type, top_fit_type, bottom_fit_type)`.

5.10.2 Styles

You can set the styles with `lv_cont_set_style(btn, LV_CONT_STYLE_MAIN, &style)`.

- **style.body** properties are used.

5.10.3 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.10.4 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.10.5 Example

5.10.6 API

Typedefs

typedef uint8_t **lv_layout_t**

typedef uint8_t **lv_fit_t**

Enums

enum [anonymous]

Container layout options

Values:

LV_LAYOUT_OFF = 0

No layout

LV_LAYOUT_CENTER

Center objects

LV_LAYOUT_COLUMN_LEFT

COULMN:

- Place the object below each other
- Keep `pad_top` space on the top
- Keep `pad_inner` space between the objectsColumn left align

LV_LAYOUT_COLUMN_MID

Column middle align

LV_LAYOUT_COLUMN_RIGHT

Column right align

LV_LAYOUT_ROW_TOP

ROW:

- Place the object next to each other
- Keep `pad_left` space on the left
- Keep `pad_inner` space between the objects
- If the object which applies the layout has `base_dir == LV_BIDI_DIR_RTL` the row will start from the right applying `pad.right` spaceRow top align

LV_LAYOUT_ROW_MID

Row middle align

LV_LAYOUT_ROW_BOTTOM

Row bottom align

LV_LAYOUT_PRETTY_TOP

PRETTY:

- Place the object next to each other
- If there is no more space start a new row
- Respect `pad_left` and `pad_right` when determining the available space in a row
- Keep `pad_inner` space between the objects in the same row
- Keep `pad_inner` space between the objects in rows
- Divide the remaining horizontal space equallyRow top align

LV_LAYOUT_PRETTY_MID

Row middle align

LV_LAYOUT_PRETTY_BOTTOM

Row bottom align

LV_LAYOUT_GRID
GRID

- Place the object next to each other
- If there is no more space start a new row
- Respect `pad_left` and `pad_right` when determining the available space in a row
- Keep `pad_inner` space between the objects in the same row
- Keep `pad_inner` space between the objects in rows
- Unlike `PRETTY`, `GRID` always keep `pad_inner` space horizontally between objects so it doesn't divide the remaining horizontal space equallyAlign same-sized object into a grid

_LV_LAYOUT_LAST

enum [anonymous]
How to resize the container around the children.

Values:

LV_FIT_NONE
Do not change the size automatically

LV_FIT_TIGHT
Shrink wrap around the children

LV_FIT_PARENT
Align the size to the parent's edge

LV_FIT_MAX
Align the size to the parent's edge first but if there is an object out of it then get larger

_LV_FIT_LAST

enum [anonymous]
Values:

LV_CONT_PART_MAIN = *LV_OBJ_PART_MAIN*

_LV_CONT_PART_VIRTUAL_LAST = *_LV_OBJ_PART_VIRTUAL_LAST*

_LV_CONT_PART_REAL_LAST = *_LV_OBJ_PART_REAL_LAST*

Functions

lv_obj_t ***lv_cont_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a container objects

Return pointer to the created container

Parameters

- `par`: pointer to an object, it will be the parent of the new container
- `copy`: pointer to a container object, if not NULL then the new object will be copied from it

void **lv_cont_set_layout**(*lv_obj_t *cont*, *lv_layout_t layout*)
Set a layout on a container

Parameters

- `cont`: pointer to a container object

- `layout`: a layout from 'lv_cont_layout_t'

void **lv_cont_set_fit4**(*lv_obj_t *cont*, *lv_fit_t left*, *lv_fit_t right*, *lv_fit_t top*, *lv_fit_t bottom*)

Set the fit policy in all 4 directions separately. It tell how to change the container's size automatically.

Parameters

- `cont`: pointer to a container object
- `left`: left fit policy from `lv_fit_t`
- `right`: right fit policy from `lv_fit_t`
- `top`: top fit policy from `lv_fit_t`
- `bottom`: bottom fit policy from `lv_fit_t`

static void **lv_cont_set_fit2**(*lv_obj_t *cont*, *lv_fit_t hor*, *lv_fit_t ver*)

Set the fit policy horizontally and vertically separately. It tells how to change the container's size automatically.

Parameters

- `cont`: pointer to a container object
- `hor`: horizontal fit policy from `lv_fit_t`
- `ver`: vertical fit policy from `lv_fit_t`

static void **lv_cont_set_fit**(*lv_obj_t *cont*, *lv_fit_t fit*)

Set the fit policy in all 4 direction at once. It tells how to change the container's size automatically.

Parameters

- `cont`: pointer to a container object
- `fit`: fit policy from `lv_fit_t`

lv_layout_t **lv_cont_get_layout**(**const** *lv_obj_t *cont*)

Get the layout of a container

Return the layout from 'lv_cont_layout_t'

Parameters

- `cont`: pointer to container object

lv_fit_t **lv_cont_get_fit_left**(**const** *lv_obj_t *cont*)

Get left fit mode of a container

Return an element of `lv_fit_t`

Parameters

- `cont`: pointer to a container object

lv_fit_t **lv_cont_get_fit_right**(**const** *lv_obj_t *cont*)

Get right fit mode of a container

Return an element of `lv_fit_t`

Parameters

- `cont`: pointer to a container object

lv_fit_t **lv_cont_get_fit_top**(**const** *lv_obj_t *cont*)

Get top fit mode of a container

Return an element of `lv_fit_t`

Parameters

- `cont`: pointer to a container object

lv_fit_t **`lv_cont_get_fit_bottom(const`** *lv_obj_t *cont*`)`

Get bottom fit mode of a container

Return an element of `lv_fit_t`

Parameters

- `cont`: pointer to a container object

`struct lv_cont_ext_t`

Public Members

lv_layout_t **`layout`**

lv_fit_t **`fit_left`**

lv_fit_t **`fit_right`**

lv_fit_t **`fit_top`**

lv_fit_t **`fit_bottom`**

5.11 color picker (lv_cpicker)

5.11.1 Overview

As its name implies *Color picker* allows to select color. The Hue, Saturation and Value of the color can be selected after each other.

The widget has two forms: circle (disc) and rectangle.

In both forms, be long pressing the object, the color picker will change to the next parameter of the color (hue, saturation or value). Besides, double click will reset the current parameter.

5.11.2 Parts and Styles

The Color picker's main part is called `LV_CPICKER_PART_BG`. In circular form it uses *scale_width* to set the the width of the circle and *pad_inner* for padding between the circle and the inner preview circle. In rectangle mode *radius* can be used to apply a radius on the rectangle.

The object has virtual part called `LV_CPICKER_PART_KNOB` which is rectangle (or circle) drawn on the current value. It uses all the rectangle like style properties and padding to make it larger than the width of the circle or rectangle background.

5.11.3 Usage

Type

The type of the Color picker can be changed with `lv_cpicker_set_type(cpicker, LV_CPICKER_TYPE_RECT/DISC)`

Set color

The colro can be set manually with `lv_cpicker_set_hue/saturation/value(cpicker, x)` or all at once with `lv_cpicker_set_hsv(cpicker, hsv)` or `lv_cpicker_set_color(cpicker, rgb)`

Color mode

The current color moed can be manually selected with `lv_cpicker_set_color_mode(cpicker, LV_CPICKER_COLOR_MODE_HUE/SATURATION/VALUE)`.

The color moe be fixed (do not change with long press) using `lv_cpicker_set_color_mode_fixed(cpicker, true)`

Knob color

`lv_cpicker_set_knob_colored(cpicker, true)` make the knob to automatically show the selected color as background color.

5.11.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.11.5 Keys

- **LV_KEY_UP**, **LV_KEY_RIGHT** Increment the current parameter's value by 1
- **LV_KEY_DOWN**, **LV_KEY_LEFT** Decrement the current parameter's by 1
- **LV_KEY_ENTER** By long press the next mode will be shown. By double click the current parameter will be reset.

Learn more about *Keys*.

5.11.6 Example

5.11.7 API

Typedefs

typedef uint8_t **lv_cpicker_type_t**

typedef uint8_t **lv_cpicker_color_mode_t**

Enums

enum [anonymous]

Values:

LV_CPICKER_TYPE_RECT

LV_CPICKER_TYPE_DISC

enum [anonymous]
Values:

LV_CPICKER_COLOR_MODE_HUE

LV_CPICKER_COLOR_MODE_SATURATION

LV_CPICKER_COLOR_MODE_VALUE

enum [anonymous]
Values:

LV_CPICKER_PART_MAIN = *LV_OBJ_PART_MAIN*

LV_CPICKER_PART_KNOB = *_LV_OBJ_PART_VIRTUAL_LAST*

_LV_CPICKER_PART_VIRTUAL_LAST

_LV_CPICKER_PART_REAL_LAST = *_LV_OBJ_PART_REAL_LAST*

Functions

lv_obj_t ***lv_cpicker_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a colorpicker objects

Return pointer to the created colorpicker

Parameters

- `par`: pointer to an object, it will be the parent of the new colorpicker
- `copy`: pointer to a colorpicker object, if not NULL then the new object will be copied from it

void **lv_cpicker_set_type**(*lv_obj_t *cpicker*, *lv_cpicker_type_t type*)
Set a new type for a colorpicker

Parameters

- `cpicker`: pointer to a colorpicker object
- `type`: new type of the colorpicker (from 'lv_cpicker_type_t' enum)

bool **lv_cpicker_set_hue**(*lv_obj_t *cpicker*, uint16_t *hue*)
Set the current hue of a colorpicker.

Return true if changed, otherwise false

Parameters

- `cpicker`: pointer to colorpicker object
- `hue`: current selected hue [0..360]

bool **lv_cpicker_set_saturation**(*lv_obj_t *cpicker*, uint8_t *saturation*)
Set the current saturation of a colorpicker.

Return true if changed, otherwise false

Parameters

- `cpicker`: pointer to colorpicker object
- `saturation`: current selected saturation [0..100]

bool **lv_cpicker_set_value**(*lv_obj_t *cpicker*, uint8_t *val*)
Set the current value of a colorpicker.

Return true if changed, otherwise false

Parameters

- `cpicker`: pointer to colorpicker object
- `val`: current selected value [0..100]

bool **`lv_cpicker_set_hsv`**(*lv_obj_t *cpicker*, *lv_color_hsv_t hsv*)
Set the current hsv of a colorpicker.

Return true if changed, otherwise false

Parameters

- `cpicker`: pointer to colorpicker object
- `hsv`: current selected hsv

bool **`lv_cpicker_set_color`**(*lv_obj_t *cpicker*, *lv_color_t color*)
Set the current color of a colorpicker.

Return true if changed, otherwise false

Parameters

- `cpicker`: pointer to colorpicker object
- `color`: current selected color

void **`lv_cpicker_set_color_mode`**(*lv_obj_t *cpicker*, *lv_cpicker_color_mode_t mode*)
Set the current color mode.

Parameters

- `cpicker`: pointer to colorpicker object
- `mode`: color mode (hue/sat/val)

void **`lv_cpicker_set_color_mode_fixed`**(*lv_obj_t *cpicker*, bool *fixed*)
Set if the color mode is changed on long press on center

Parameters

- `cpicker`: pointer to colorpicker object
- `fixed`: color mode cannot be changed on long press

void **`lv_cpicker_set_knob_colored`**(*lv_obj_t *cpicker*, bool *en*)
Make the knob to be colored to the current color

Parameters

- `cpicker`: pointer to colorpicker object
- `en`: true: color the knob; false: not color the knob

lv_cpicker_color_mode_t **`lv_cpicker_get_color_mode`**(*lv_obj_t *cpicker*)
Get the current color mode.

Return color mode (hue/sat/val)

Parameters

- `cpicker`: pointer to colorpicker object

bool **`lv_cpicker_get_color_mode_fixed`**(*lv_obj_t *cpicker*)
Get if the color mode is changed on long press on center

Return mode cannot be changed on long press

Parameters

- `cpicker`: pointer to colorpicker object

uint16_t **lv_cpicker_get_hue**(*lv_obj_t *cpicker*)
Get the current hue of a colorpicker.

Return current selected hue

Parameters

- `cpicker`: pointer to colorpicker object

uint8_t **lv_cpicker_get_saturation**(*lv_obj_t *cpicker*)
Get the current saturation of a colorpicker.

Return current selected saturation

Parameters

- `cpicker`: pointer to colorpicker object

uint8_t **lv_cpicker_get_value**(*lv_obj_t *cpicker*)
Get the current hue of a colorpicker.

Return current selected value

Parameters

- `cpicker`: pointer to colorpicker object

lv_color_hsv_t **lv_cpicker_get_hsv**(*lv_obj_t *cpicker*)
Get the current selected hsv of a colorpicker.

Return current selected hsv

Parameters

- `cpicker`: pointer to colorpicker object

lv_color_t **lv_cpicker_get_color**(*lv_obj_t *cpicker*)
Get the current selected color of a colorpicker.

Return current selected color

Parameters

- `cpicker`: pointer to colorpicker object

bool **lv_cpicker_get_knob_colored**(*lv_obj_t *cpicker*)
Whether the knob is colored to the current color or not

Return true: color the knob; false: not color the knob

Parameters

- `cpicker`: pointer to color picker object

struct lv_cpicker_ext_t

Public Members

lv_color_hsv_t **hsv**

lv_style_list_t **style_list**

lv_point_t **pos**

uint8_t **colored**

struct *lv_cpicker_ext_t*::[anonymous] **knob**

uint32_t **last_click_time**

uint32_t **last_change_time**

lv_point_t **last_press_point**

lv_cpicker_color_mode_t **color_mode**

uint8_t **color_mode_fixed**

lv_cpicker_type_t **type**

5.12 Drop-down list (lv_dropdown)

5.12.1 Overview

The drop-down list allows the user to select one value from a list.

The drop-down list is closed by default and displays a single value or a predefined text. When activated (by click on the drop-down list), a list is created from which the user may select one option. When the user selects a new value, the list is deleted.

5.12.2 Parts and Styles

The drop-down list's main part is called `LV_DROPDOWN_PART_MAIN` which is a simple *lv_obj* object. It uses all the typical background properties. *Pressed*, *Focused*, *Edited* etc. stiles are also applied as usual.

The list, which is created when the main object is clicked, is an *Page*. Its background part can be referenced with `LV_DROPDOWN_PART_LIST` and uses all the typical background properties for the rectangle itself and text properties for the options. To adjust the space between the options use the *text_line_space* style property. Padding values can be used to make some space on the edges.

The scrollable part of the page is hidden and its styles are always empty (so transparent with no padding).

The scrollbar can be referenced with `LV_DROPDOWN_PART_SCRLBAR` and uses all the typical background properties.

The selected option can be referenced with `LV_DROPDOWN_PART_SELECTED` and uses all the typical background properties. It will used in its default state to draw a rectangle on the selected option, and in pressed state to draw a rectangle on the being pressed option.

5.12.3 Usage

5.12.4 Overview

Set options

The options are passed to the drop-down list as a string with `lv_dropdown_set_options(dropdown, options)`. The options should be separated by `\n`. For example: `"First\nSecond\nThird"`. The string will be saved in the drop-down list, so it can in local variable too.

The `lv_dropdown_add_option(dropdown, "New option", pos)` function inserts a new option to `pos` index.

To save memory the options can set from a static(constant) string too with `lv_dropdown_set_static_options(dropdown, options)`. In this case the options string should be alive while the drop-down list exists and `lv_dropdown_add_option` can't be used

You can select an option manually with `lv_dropdown_set_selected(dropdown, id)`, where *id* is the index of an option.

Get selected option

The get the currently selected option, use `lv_dropdown_get_selected(dropdown)`. It will return the *index* of the selected option.

`lv_dropdown_get_selected_str(dropdown, buf, buf_size)` copies the name of the selected option to a `buf`.

Direction

The list can be created on any side. The default `LV_DROPDOWN_DOWN` can be modified by `lv_dropdown_set_dir(dropdown, LV_DROPDOWN_DIR_LEFT/RIGHT/UP/DOWN)` function.

If the list would be vertically out of the screen, it will aligned to the edge.

Symbol

A symbol (typically an arrow) can be added to the drop down list with `lv_dropdown_set_symbol(dropdown, LV_SYMBOL_...)`

If the direction of the drop-down list is `LV_DROPDOWN_DIR_LEFT` the symbol will be shown on the left, else on the right.

Maximum height

The maximum height of drop-down list can be set via `lv_dropdown_set_max_height(dropdown, height)`. By default it's set to 3/4 vertical resolution.

Show selected

The main part can either show the selected option or a static text. It can controlled with `lv_dropdown_set_show_selected(sropdown, true/false)`.

The static text can be set with `lv_dropdown_set_text(dropdown, "Text")`. Only the pointer of the text is saved.

Animation time

The drop-down list's open/close animation time is adjusted by `lv_dropdown_set_anim_time(ddlist, anim_time)`. Zero animation time means no animation.

Manually open/close

To manually open or close the drop-down list the `lv_dropdown_open/close(dropdown, LV_ANIM_ON/OFF)` function can be used.

5.12.5 Events

Besides the Generic events, the following Special events are sent by the drop-down list:

- **LV_EVENT_VALUE_CHANGED** - Sent when the new option is selected.

Learn more about *Events*.

5.12.6 Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_RIGHT/DOWN** - Select the next option.
- **LV_KEY_LEFT/UP** - Select the previous option.
- **LY_KEY_ENTER** - Apply the selected option (Send `LV_EVENT_VALUE_CHANGED` event and close the drop-down list).

5.12.7 Example

5.12.8 API

Typedefs

typedef uint8_t **lv_dropdown_dir_t**

typedef uint8_t **lv_dropdown_part_t**

Enums

enum [anonymous]

Values:

LV_DROPDOWN_DIR_DOWN

LV_DROPDOWN_DIR_UP

LV_DROPDOWN_DIR_LEFT

LV_DROPDOWN_DIR_RIGHT

enum [anonymous]

Values:

LV_DROPDOWN_PART_MAIN = *LV_OBJ_PART_MAIN*

LV_DROPDOWN_PART_LIST = *_LV_OBJ_PART_REAL_LAST*

LV_DROPDOWN_PART_SCROLLBAR

LV_DROPDOWN_PART_SELECTED

Functions

lv_obj_t ***lv_dropdown_create**(*lv_obj_t* **par*, **const** *lv_obj_t* **copy*)
Create a drop down list objects

Return pointer to the created drop down list

Parameters

- `par`: pointer to an object, it will be the parent of the new drop down list
- `copy`: pointer to a drop down list object, if not NULL then the new object will be copied from it

void **lv_dropdown_set_text**(*lv_obj_t* **ddlist*, **const** char **txt*)
Set text of the ddlist (Displayed on the button if `show_selected = false`)

Parameters

- `ddlist`: pointer to a drop down list object
- `txt`: the text as a string (Only it's pointer is saved)

void **lv_dropdown_clear_options**(*lv_obj_t* **ddlist*)
Clear any options in a drop down list. Static or dynamic.

Parameters

- `ddlist`: pointer to drop down list object

void **lv_dropdown_set_options**(*lv_obj_t* **ddlist*, **const** char **options*)
Set the options in a drop down list from a string

Parameters

- `ddlist`: pointer to drop down list object
- `options`: a string with ' ' separated options. E.g. "One\nTwo\nThree" The options string can be destroyed after calling this function

void **lv_dropdown_set_options_static**(*lv_obj_t* **ddlist*, **const** char **options*)
Set the options in a drop down list from a string

Parameters

- `ddlist`: pointer to drop down list object
- `options`: a static string with ' ' separated options. E.g. "One\nTwo\nThree"

void **lv_dropdown_add_option**(*lv_obj_t* **ddlist*, **const** char **option*, uint32_t *pos*)
Add an options to a drop down list from a string. Only works for dynamic options.

Parameters

- `ddlist`: pointer to drop down list object
- `option`: a string without ' '. E.g. "Four"
- `pos`: the insert position, indexed from 0, LV_DROPDOWN_POS_LAST = end of string

void **lv_dropdown_set_selected**(*lv_obj_t* **ddlist*, uint16_t *sel_opt*)
Set the selected option

Parameters

- `ddlist`: pointer to drop down list object
- `sel_opt`: id of the selected option (0 ... number of option - 1);

void **lv_dropdown_set_dir**(*lv_obj_t *ddlist*, *lv_dropdown_dir_t dir*)
Set the direction of the a drop down list

Parameters

- `ddlist`: pointer to a drop down list object
- `dir`: LV_DROPDOWN_DIR_LEF/RIGHT/TOP/BOTTOM

void **lv_dropdown_set_max_height**(*lv_obj_t *ddlist*, lv_coord_t *h*)
Set the maximal height for the drop down list

Parameters

- `ddlist`: pointer to a drop down list
- `h`: the maximal height

void **lv_dropdown_set_symbol**(*lv_obj_t *ddlist*, **const** char **symbol*)
Set an arrow or other symbol to display when the drop-down list is closed.

Parameters

- `ddlist`: pointer to drop down list object
- `symbol`: a text like `LV_SYMBOL_DOWN` or NULL to not draw icon

void **lv_dropdown_set_show_selected**(*lv_obj_t *ddlist*, bool *show*)
Set whether the ddlist highlight the last selected option and display its text or not

Parameters

- `ddlist`: pointer to a drop down list object
- `show`: true/false

const char ***lv_dropdown_get_text**(*lv_obj_t *ddlist*)
Get text of the ddlist (Displayed on the button if `show_selected = false`)

Return the text string

Parameters

- `ddlist`: pointer to a drop down list object

const char ***lv_dropdown_get_options**(**const** *lv_obj_t *ddlist*)
Get the options of a drop down list

Return the options separated by ' '-s (E.g. "Option1\nOption2\nOption3")

Parameters

- `ddlist`: pointer to drop down list object

uint16_t **lv_dropdown_get_selected**(**const** *lv_obj_t *ddlist*)
Get the selected option

Return id of the selected option (0 ... number of option - 1);

Parameters

- `ddlist`: pointer to drop down list object

uint16_t **lv_dropdown_get_option_cnt**(**const** *lv_obj_t *ddlist*)
Get the total number of options

Return the total number of options in the list

Parameters

- `ddlist`: pointer to drop down list object

void **`lv_dropdown_get_selected_str(const`** *lv_obj_t *ddlist*, char **buf*, uint32_t *buf_size*)
Get the current selected option as a string

Parameters

- `ddlist`: pointer to ddlist object
- `buf`: pointer to an array to store the string
- `buf_size`: size of `buf` in bytes. 0: to ignore it.

lv_coord_t **`lv_dropdown_get_max_height(const`** *lv_obj_t *ddlist*)
Get the fix height value.

Return the height if the ddlist is opened (0: auto size)

Parameters

- `ddlist`: pointer to a drop down list object

`const` char ***`lv_dropdown_get_symbol(`***lv_obj_t *ddlist*)
Get the symbol to draw when the drop-down list is closed

Return the symbol or NULL if not enabled

Parameters

- `ddlist`: pointer to drop down list object

lv_dropdown_dir_t **`lv_dropdown_get_dir(const`** *lv_obj_t *ddlist*)
Get the symbol to draw when the drop-down list is closed

Return the symbol or NULL if not enabled

Parameters

- `ddlist`: pointer to drop down list object

bool **`lv_dropdown_get_show_selected(`***lv_obj_t *ddlist*)
Get whether the ddlist highlight the last selected option and display its text or not

Return true/false

Parameters

- `ddlist`: pointer to a drop down list object

void **`lv_dropdown_open(`***lv_obj_t *ddlist*)
Open the drop down list with or without animation

Parameters

- `ddlist`: pointer to drop down list object

void **`lv_dropdown_close(`***lv_obj_t *ddlist*)
Close (Collapse) the drop down list

Parameters

- `ddlist`: pointer to drop down list object
- `anim_en`: LV_ANIM_ON: use animation; LV_ANOM_OFF: not use animations

`struct lv_dropdown_ext_t`

Public Members

lv_obj_t ***page**

const char ***text**

const char ***symbol**

char ***options**

lv_style_list_t **style_selected**

lv_style_list_t **style_page**

lv_style_list_t **style_scrlbar**

lv_coord_t **max_height**

uint16_t **option_cnt**

uint16_t **sel_opt_id**

uint16_t **sel_opt_id_orig**

uint16_t **pr_opt_id**

lv_dropdown_dir_t **dir**

uint8_t **show_selected**

uint8_t **static_txt**

5.13 Gauge (lv_gauge)

5.13.1 Overview

The gauge is a meter with scale labels and one or more needles.

5.13.2 Parts and Styles

The Gauge's main part is called `LV_GAUGE_PART_MAIN`. It draws a background using the typical background style properties and "minor" scale lines using the *line* and *scale* style properties. It also uses the *text* properties to set the style of the scale labels. *pad_inner* is used to set space between the scale lines and the scale labels.

`LV_GAUGE_PART_MAJOR` is a virtual part which describes the major scale lines (where labels are added) using the *line* and *scale* style properties.

`LV_GAUGE_PART_NEEDLE` is also virtual part and it describes the needle(s) via the *line* style properties. *size* and the typical background properties are used to describe a rectangle (or circle) in the picot point of the needle(s). *pad_inner* is used to to make the needle(s) smaller than the outer radius of the scale lines.

5.13.3 Usage

Set value and needles

The gauge can show more than one needle. Use the `lv_gauge_set_needle_count(gauge, needle_num, color_array)` function to set the number of needles and an array with colors for each needle. The array must be static or global variable because only its pointer is stored.

You can use `lv_gauge_set_value(gauge, needle_id, value)` to set the value of a needle.

Scale

You can use the `lv_gauge_set_scale(gauge, angle, line_num, label_cnt)` function to adjust the scale angle and the number of the scale lines and labels. The default settings are 220 degrees, 6 scale labels, and 21 lines.

The scale of the Gauge can have offset. It can be adjusted with `lv_gauge_set_angle_offset(gauge, angle)`.

Range

The range of the gauge can be specified by `lv_gauge_set_range(gauge, min, max)`. The default range is 0..100.

Critical value

To set a critical value, use `lv_gauge_set_critical_value(gauge, value)`. The scale color will be changed to *scale_end_color* after this value. The default critical value is 80.

5.13.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.13.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.13.6 Example

5.13.7 API

Typedefs

typedef void (*__lv_gauge_format_cb_t__)(*lv_obj_t* *gauge, char *buf, int bufsize, int32_t value)

typedef uint8_t **lv_gauge_style_t**

Enums

enum [anonymous]

Values:

LV_GAUGE_PART_MAIN = *LV_LINEMETER_PART_MAIN*

LV_GAUGE_PART_MAJOR = *_LV_LINEMETER_PART_VIRTUAL_LAST*

LV_GAUGE_PART_NEEDLE

_LV_GAUGE_PART_VIRTUAL_LAST = *_LV_LINEMETER_PART_VIRTUAL_LAST*

_LV_GAUGE_PART_REAL_LAST = *_LV_LINEMETER_PART_REAL_LAST*

Functions

lv_obj_t ***lv_gauge_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a gauge objects

Return pointer to the created gauge

Parameters

- `par`: pointer to an object, it will be the parent of the new gauge
- `copy`: pointer to a gauge object, if not NULL then the new object will be copied from it

void **lv_gauge_set_needle_count**(*lv_obj_t *gauge*, uint8_t *needle_cnt*, **const** *lv_color_t colors*[])

Set the number of needles

Parameters

- `gauge`: pointer to gauge object
- `needle_cnt`: new count of needles
- `colors`: an array of colors for needles (with 'num' elements)

void **lv_gauge_set_value**(*lv_obj_t *gauge*, uint8_t *needle_id*, int32_t *value*)

Set the value of a needle

Parameters

- `gauge`: pointer to a gauge
- `needle_id`: the id of the needle
- `value`: the new value

static void **lv_gauge_set_range**(*lv_obj_t *gauge*, int32_t *min*, int32_t *max*)

Set minimum and the maximum values of a gauge

Parameters

- `gauge`: pointer to he gauge object
- `min`: minimum value
- `max`: maximum value

static void **lv_gauge_set_critical_value**(*lv_obj_t *gauge*, int32_t *value*)

Set a critical value on the scale. After this value 'line.color' scale lines will be drawn

Parameters

- `gauge`: pointer to a gauge object
- `value`: the critical value

void **`lv_gauge_set_scale`**(*lv_obj_t *gauge*, uint16_t *angle*, uint8_t *line_cnt*, uint8_t *label_cnt*)

Set the scale settings of a gauge

Parameters

- `gauge`: pointer to a gauge object
- `angle`: angle of the scale (0..360)
- `line_cnt`: count of scale lines. To get a given "subdivision" lines between labels: `line_cnt = (sub_div + 1) * (label_cnt - 1) + 1`
- `label_cnt`: count of scale labels.

`static` void **`lv_gauge_set_angle_offset`**(*lv_obj_t *gauge*, uint16_t *angle*)

Set the set an offset for the gauge's angles to rotate it.

Parameters

- `gauge`: pointer to a line meter object
- `angle`: angle offset (0..360), rotates clockwise

void **`lv_gauge_set_needle_img`**(*lv_obj_t *gauge*, **`const`** void *img, lv_coord_t *pivot_x*, lv_coord_t *pivot_y*)

Set an image to display as needle(s). The needle image should be horizontal and pointing to the right (`--->`).

Parameters

- `gauge`: pointer to a gauge object
- `img_src`: pointer to an `lv_img_dsc_t` variable or a path to an image (not an `lv_img` object)
- `pivot_x`: the X coordinate of rotation center of the image
- `pivot_y`: the Y coordinate of rotation center of the image

void **`lv_gauge_set_formatter_cb`**(*lv_obj_t *gauge*, *lv_gauge_format_cb_t format_cb*)

Assign a function to format gauge values

Parameters

- `gauge`: pointer to a gauge object
- `format_cb`: pointer to function of lv_gauge_format_cb_t

int32_t **`lv_gauge_get_value`**(**`const`** *lv_obj_t *gauge*, uint8_t *needle*)

Get the value of a needle

Return the value of the needle [min,max]

Parameters

- `gauge`: pointer to gauge object
- `needle`: the id of the needle

uint8_t **`lv_gauge_get_needle_count`**(**`const`** *lv_obj_t *gauge*)

Get the count of needles on a gauge

Return count of needles

Parameters

- `gauge`: pointer to gauge

static int32_t **lv_gauge_get_min_value(const** *lv_obj_t *lmeter*)
Get the minimum value of a gauge

Return the minimum value of the gauge

Parameters

- `gauge`: pointer to a gauge object

static int32_t **lv_gauge_get_max_value(const** *lv_obj_t *lmeter*)
Get the maximum value of a gauge

Return the maximum value of the gauge

Parameters

- `gauge`: pointer to a gauge object

static int32_t **lv_gauge_get_critical_value(const** *lv_obj_t *gauge*)
Get a critical value on the scale.

Return the critical value

Parameters

- `gauge`: pointer to a gauge object

uint8_t **lv_gauge_get_label_count(const** *lv_obj_t *gauge*)
Set the number of labels (and the thicker lines too)

Return count of labels

Parameters

- `gauge`: pointer to a gauge object

static uint16_t **lv_gauge_get_line_count(const** *lv_obj_t *gauge*)
Get the scale number of a gauge

Return number of the scale units

Parameters

- `gauge`: pointer to a gauge object

static uint16_t **lv_gauge_get_scale_angle(const** *lv_obj_t *gauge*)
Get the scale angle of a gauge

Return angle of the scale

Parameters

- `gauge`: pointer to a gauge object

static uint16_t **lv_gauge_get_angle_offset(***lv_obj_t *gauge*)
Get the offset for the gauge.

Return angle offset (0..360)

Parameters

- `gauge`: pointer to a gauge object

const void ***lv_gauge_get_needle_img(***lv_obj_t *gauge*)
Get an image to display as needle(s).

Return pointer to an `lv_img_dsc_t` variable or a path to an image (not an `lv_img` object). `NULL` if not used.

Parameters

- `gauge`: pointer to a gauge object

lv_coord_t **`lv_gauge_get_needle_img_pivot_x`**(*lv_obj_t *gauge*)
Get the X coordinate of the rotation center of the needle image

Return the X coordinate of rotation center of the image

Parameters

- `gauge`: pointer to a gauge object

lv_coord_t **`lv_gauge_get_needle_img_pivot_y`**(*lv_obj_t *gauge*)
Get the Y coordinate of the rotation center of the needle image

Return the X coordinate of rotation center of the image

Parameters

- `gauge`: pointer to a gauge object

`struct lv_gauge_ext_t`

Public Members

lv_linemeter_ext_t **`lmeter`**

int32_t ***`values`**

`const` *lv_color_t* ***`needle_colors`**

`const` void ***`needle_img`**

lv_point_t **`needle_img_pivot`**

lv_style_list_t **`style_needle`**

lv_style_list_t **`style_strong`**

uint8_t **`needle_count`**

uint8_t **`label_count`**

lv_gauge_format_cb_t **`format_cb`**

5.14 Image (lv_img)

5.14.1 Overview

Images are the basic object to display from the flash (as arrays) or externally as files. Images can display symbols (`LV_SYMBOL_...`) too.

Using the Image decoder interface custom image formats can be supported as well.

5.14.2 Parts and Styles

The images has only a main part called `LV_IMG_PART_MAIN` which uses the typical background style propeties to draw a background rectangle and the *image* properties. The padding values are used to make the background virtually larger. (It won't change the image's real size but the size modification is applied only during drawing)

5.14.3 Usage

Image source

To provide maximum flexibility, the source of the image can be:

- a variable in the code (a C array with the pixels).
- a file stored externally (like on an SD card).
- a text with *Symbols*.

To set the source of an image, use `lv_img_set_src(img, src)`.

To generate a **pixel array** from a PNG, JPG or BMP image, use the Online image converter tool and set the converted image with its pointer: `lv_img_set_src(img1, &converted_img_var);` To make the variable visible in the C file, you need to declare it with `LV_IMG_DECLARE(converted_img_var)`.

To use **external files**, you also need to convert the image files using the online converter tool but now you should select the binary Output format. You also need to use LVGL's file system module and register a driver with some functions for the basic file operation. Got to the *File system* to learn more. To set an image sourced from a file, use `lv_img_set_src(img, "S:folder1/my_img.bin")`.

You can set a **symbol** similarly to *Labels*. In this case, the image will be rendered as text according to the *font* specified in the style. It enables to use of light-weighted mono-color "letters" instead of real images. You can set symbol like `lv_img_set_src(img1, LV_SYMBOL_OK)`.

Label as an image

Images and labels are sometimes used to convey the same thing. For example, to describe what a button does. Therefore, images and labels are somewhat interchangeable. To handle these images can even display texts by using `LV_SYMBOL_DUMMY` as the prefix of the text. For example, `lv_img_set_src(img, LV_SYMBOL_DUMMY "Some text")`.

Transparency

The internal (variable) and external images support 2 transparency handling methods:

- **Chrome keying** - Pixels with `LV_COLOR_TRANSP` (*lv_conf.h*) color will be transparent.
- **Alpha byte** - An alpha byte is added to every pixel.

Palette and Alpha index

Besides *True color* (RGB) color format, the following formats are also supported:

- **Indexed** - Image has a palette.
- **Alpha indexed** - Only alpha values are stored.

These options can be selected in the font converter. To learn more about the color formats, read the *Images* section.

Recolor

The images can be re-colored in run-time to any color according to the brightness of the pixels. It is very useful to show different states (selected, inactive, pressed, etc.) of an image without storing more versions of the same image. This feature can be enabled in the style by setting `img.intense` between `LV_OPA_TRANSP` (no recolor, value: 0) and `LV_OPA_COVER` (full recolor, value: 255). The default value is `LV_OPA_TRANSP` so this feature is disabled.

Auto-size

It is possible to automatically set the size of the image object to the image source's width and height if enabled by the `lv_img_set_auto_size(image, true)` function. If *auto-size* is enabled, then when a new file is set, the object size is automatically changed. Later, you can modify the size manually. The *auto-size* is enabled by default if the image is not a screen.

Mosaic

If the object size is greater than the image size in any directions, then the image will be repeated like a mosaic. It's a very useful feature to create a large image from only a very narrow source. For example, you can have a *300 x 1* image with a special gradient and set it as a wallpaper using the mosaic feature.

Offset

With `lv_img_set_offset_x(img, x_ofs)` and `lv_img_set_offset_y(img, y_ofs)`, you can add some offset to the displayed image. It is useful if the object size is smaller than the image source size. Using the offset parameter a Texture atlas or a "running image" effect can be created by *Animating* the x or y offset.

5.14.4 Transformations

Using the `lv_img_set_zoom(img, factor)` the images will be zoomed. Set `factor` to `256` or `LV_IMG_ZOOM_NONE` to disable zooming. A larger value enlarges the images (e.g. `512` double size), a smaller value shrinks it (e.g. `128` half size). Fractional scale works as well. E.g. `281` for 10% enlargement.

To rotate the image use `lv_img_set_angle(img, angle)`. Angle has 0.1 degree precision, so for 45.8° set 458.

By default, the pivot point of the rotation is the center of the image. It can be changed with `lv_img_set_pivot(img, pivot_x, pivot_y)`. `0;0` is the top left corner.

The quality of the transformation can be adjusted with `lv_img_set_antialias(img, true/false)`. With enabled anti-aliasing the transformations has a higher quality but they are slower.

The transformations require the whole image to be available. Therefore indexed images (`LV_IMG_CF_INDEXED_...`), alpha only images (`LV_IMG_CF_ALPHA_...`) or images from files can be transformed. In other words transformations work only on true color images stored as C array, or if a custom Image decoder returns the whole image.

Note that, the real coordinates of image object won't change during transformation. That is `lv_obj_get_width/height/x/y()` will returned the original, non-zoomed coordinates.

5.14.5 Rotate

The images can be rotated with

5.14.6 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.14.7 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.14.8 Example

5.14.9 API

Typedefs

typedef uint8_t **lv_img_part_t**

Enums

enum [anonymous]

Values:

LV_IMG_PART_MAIN

Functions

lv_obj_t ***lv_img_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create an image objects

Return pointer to the created image

Parameters

- `par`: pointer to an object, it will be the parent of the new button
- `copy`: pointer to a image object, if not NULL then the new object will be copied from it

void **lv_img_set_src**(*lv_obj_t *img*, **const** void **src_img*)

Set the pixel map to display by the image

Parameters

- `img`: pointer to an image object
- `data`: the image data

void **lv_img_set_auto_size**(*lv_obj_t *img*, bool *autosize_en*)

Enable the auto size feature. If enabled the object size will be same as the picture size.

Parameters

- `img`: pointer to an image
- `en`: true: auto size enable, false: auto size disable

void **`lv_img_set_offset_x`**(*lv_obj_t *img*, lv_coord_t *x*)
Set an offset for the source of an image. so the image will be displayed from the new origin.

Parameters

- `img`: pointer to an image
- `x`: the new offset along x axis.

void **`lv_img_set_offset_y`**(*lv_obj_t *img*, lv_coord_t *y*)
Set an offset for the source of an image. so the image will be displayed from the new origin.

Parameters

- `img`: pointer to an image
- `y`: the new offset along y axis.

void **`lv_img_set_pivot`**(*lv_obj_t *img*, lv_coord_t *pivot_x*, lv_coord_t *pivot_y*)
Set the rotation center of the image. The image will be rotated around this point

Parameters

- `img`: pointer to an image object
- `pivot_x`: rotation center x of the image
- `pivot_y`: rotation center y of the image

void **`lv_img_set_angle`**(*lv_obj_t *img*, int16_t *angle*)
Set the rotation angle of the image. The image will be rotated around the set pivot set by *`lv_img_set_pivot()`*

Parameters

- `img`: pointer to an image object
- `angle`: rotation angle in degree with 0.1 degree resolution (0..3600: clock wise)

void **`lv_img_set_zoom`**(*lv_obj_t *img*, uint16_t *zoom*)
Set the zoom factor of the image.

Parameters

- `img`: pointer to an image object
- `zoom`: the zoom factor.
  - 256 or LV_ZOOM_IMG_NONE for no zoom
  - <256: scale down
  - >256 scale up
  - 128 half size
  - 512 double size

void **`lv_img_set_antialias`**(*lv_obj_t *img*, bool *antialias*)
Enable/disable anti-aliasing for the transformations (rotate, zoom) or not

Parameters

- `img`: pointer to an image object
- `antialias`: true: anti-aliased; false: not anti-aliased

const void ***lv_img_get_src(***lv_obj_t *img***)**
Get the source of the image

Return the image source (symbol, file name or C array)

Parameters

- `img`: pointer to an image object

const char ***lv_img_get_file_name(const** lv_obj_t *img***)**
Get the name of the file set for an image

Return file name

Parameters

- `img`: pointer to an image

bool **lv_img_get_auto_size(const** lv_obj_t *img***)**
Get the auto size enable attribute

Return true: auto size is enabled, false: auto size is disabled

Parameters

- `img`: pointer to an image

lv_coord_t **lv_img_get_offset_x(***lv_obj_t *img***)**
Get the offset.x attribute of the img object.

Return offset.x value.

Parameters

- `img`: pointer to an image

lv_coord_t **lv_img_get_offset_y(***lv_obj_t *img***)**
Get the offset.y attribute of the img object.

Return offset.y value.

Parameters

- `img`: pointer to an image

uint16_t **lv_img_get_angle(***lv_obj_t *img***)**
Get the rotation angle of the image.

Return rotation angle in degree (0..359)

Parameters

- `img`: pointer to an image object

void **lv_img_get_pivot(***lv_obj_t *img*, lv_point_t *center***)**
Get the rotation center of the image.

Parameters

- `img`: pointer to an image object
- `center`: rotation center of the image

uint16_t **lv_img_get_zoom(***lv_obj_t *img***)**
Get the zoom factor of the image.

Return zoom factor (256: no zoom)

Parameters

- `img`: pointer to an image object

bool **lv_img_get_antialias**(*lv_obj_t *img*)

Get whether the transformations (rotate, zoom) are anti-aliased or not

Return true: anti-aliased; false: not anti-aliased

Parameters

- `img`: pointer to an image object

struct lv_img_ext_t

Public Members

const void ***src**

lv_point_t **offset**

lv_coord_t **w**

lv_coord_t **h**

uint16_t **angle**

lv_point_t **pivot**

uint16_t **zoom**

uint8_t **src_type**

uint8_t **auto_size**

uint8_t **cf**

uint8_t **antialias**

5.15 Image button (lv_imgbtn)

5.15.1 Overview

The Image button is very similar to the simple 'Button' object. The only difference is that, it displays user-defined images in each state instead of drawing a rectangle. Before reading this section, please read the *Button* section for better understanding.

5.15.2 Parts and Styles

The Image button object has only a main part called `LV_IMG_BTN_PART_MAIN` from where all *image* style properties are used. It's possible to recolor the image in each state with *image_recolor* and *image_recolor_opa* proeprties. For example, to make the image darker if it is pressed.

5.15.3 Usage

Image sources

To set the image in a state, use the `lv_imgbtn_set_src(imgbtn, LV_BTN_STATE_..., &img_src)`. The image sources works the same as described in the *Image object* except that, "Symbols" are not supported by the Image button.

If `LV_IMGBTN_TILED` is enabled in *lv_conf.h*, then `lv_imgbtn_set_src_tiled(imgbtn, LV_BTN_STATE_..., &img_src_left, &img_src_mid, &img_src_right)` becomes available. Using the tiled feature the *middle* image will be repeated to fill the width of the object. Therefore with `LV_IMGBTN_TILED`, you can set the width of the Image button using `lv_obj_set_width()`. However, without this option, the width will be always the same as the image source's width.

Button features

Similarly to normal Buttons `lv_imgbtn_set_checkable(imgbtn, true/false)`, `lv_imgbtn_toggle(imgbtn)` and `lv_imgbtn_set_state(imgbtn, LV_BTN_STATE_...)` also works.

5.15.4 Events

Beside the Generic events, the following Special events are sent by the buttons:

- **LV_EVENT_VALUE_CHANGED** - Sent when the button is toggled.

Note that, the generic input device related events (like `LV_EVENT_PRESSED`) are sent in the inactive state too. You need to check the state with `lv_btn_get_state(btn)` to ignore the events from inactive buttons.

Learn more about *Events*.

5.15.5 Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_RIGHT/UP** - Go to toggled state if toggling is enabled.
- **LV_KEY_LEFT/DOWN** - Go to non-toggled state if toggling is enabled.

Note that, as usual, the state of `LV_KEY_ENTER` is translated to `LV_EVENT_PRESSED/PRESSING/RELEASED` etc.

Learn more about *Keys*.

5.15.6 Example

5.15.7 API

Typedefs

typedef uint8_t **lv_imgbtn_part_t**

Enums

enum [anonymous]

Values:

LV_IMGBTN_PART_MAIN = *LV_BTN_PART_MAIN*

Functions

lv_obj_t ***lv_imgbtn_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a image button objects

Return pointer to the created image button

Parameters

- `par`: pointer to an object, it will be the parent of the new image button
- `copy`: pointer to a image button object, if not NULL then the new object will be copied from it

void **lv_imgbtn_set_src**(*lv_obj_t *imgbtn*, *lv_btn_state_t state*, **const** void **src*)

Set images for a state of the image button

Parameters

- `imgbtn`: pointer to an image button object
- `state`: for which state set the new image (from `lv_btn_state_t`) '
- `src`: pointer to an image source (a C array or path to a file)

void **lv_imgbtn_set_src_tiled**(*lv_obj_t *imgbtn*, *lv_btn_state_t state*, **const** void **src_left*, **const** void **src_mid*, **const** void **src_right*)

Set images for a state of the image button

Parameters

- `imgbtn`: pointer to an image button object
- `state`: for which state set the new image (from `lv_btn_state_t`) '
- `src_left`: pointer to an image source for the left side of the button (a C array or path to a file)
- `src_mid`: pointer to an image source for the middle of the button (ideally 1px wide) (a C array or path to a file)
- `src_right`: pointer to an image source for the right side of the button (a C array or path to a file)

static void **lv_imgbtn_set_checkable**(*lv_obj_t *imgbtn*, bool *tgl*)

Enable the toggled states. On release the button will change from/to toggled state.

Parameters

- `imgbtn`: pointer to an image button object
- `tgl`: true: enable toggled states, false: disable

static void **lv_imgbtn_set_state**(*lv_obj_t *imgbtn*, *lv_btn_state_t state*)

Set the state of the image button

Parameters

- `imgbtn`: pointer to an image button object
- `state`: the new state of the button (from lv_btn_state_t enum)

static void **lv_imgbtn_toggle(***lv_obj_t *imgbtn***)**
Toggle the state of the image button (ON->OFF, OFF->ON)

Parameters

- `imgbtn`: pointer to a image button object

const void ***lv_imgbtn_get_src(***lv_obj_t *imgbtn, lv_btn_state_t state***)**
Get the images in a given state

Return pointer to an image source (a C array or path to a file)

Parameters

- `imgbtn`: pointer to an image button object
- `state`: the state where to get the image (from `lv_btn_state_t`) '

const void ***lv_imgbtn_get_src_left(***lv_obj_t *imgbtn, lv_btn_state_t state***)**
Get the left image in a given state

Return pointer to the left image source (a C array or path to a file)

Parameters

- `imgbtn`: pointer to an image button object
- `state`: the state where to get the image (from `lv_btn_state_t`) '

const void ***lv_imgbtn_get_src_middle(***lv_obj_t *imgbtn, lv_btn_state_t state***)**
Get the middle image in a given state

Return pointer to the middle image source (a C array or path to a file)

Parameters

- `imgbtn`: pointer to an image button object
- `state`: the state where to get the image (from `lv_btn_state_t`) '

const void ***lv_imgbtn_get_src_right(***lv_obj_t *imgbtn, lv_btn_state_t state***)**
Get the right image in a given state

Return pointer to the left image source (a C array or path to a file)

Parameters

- `imgbtn`: pointer to an image button object
- `state`: the state where to get the image (from `lv_btn_state_t`) '

static *lv_btn_state_t* **lv_imgbtn_get_state(const** *lv_obj_t *imgbtn***)**
Get the current state of the image button

Return the state of the button (from lv_btn_state_t enum)

Parameters

- `imgbtn`: pointer to a image button object

static bool **lv_imgbtn_get_checkable(const** *lv_obj_t *imgbtn***)**
Get the toggle enable attribute of the image button

Return true: toggle enabled, false: disabled

Parameters

- `imgbtn`: pointer to a image button object

struct lv_imgbtn_ext_t

Public Members

lv_btn_ext_t **btn**

const void ***img_src_mid**[**_LV_BTN_STATE_LAST**]

const void ***img_src_left**[**_LV_BTN_STATE_LAST**]

const void ***img_src_right**[**_LV_BTN_STATE_LAST**]

lv_img_cf_t **act_cf**

uint8_t **tiled**

5.16 Keyboard (lv_keyboard)

5.16.1 Overview

The Keyboard object is a special *Button matrix* with predefined keymaps and other features to realize a virtual keyboard to write text.

5.16.2 Parts and Styles

Similarly to Button matices Keyboards consist of 2 part:

- `LV_KEYBOARD_PART_BG` which is the main part and uses all the typical background properties
- `LV_KEYBOARD_PART_BTN` which is virtual part for the buttons. It also uses all typical backround proeprties and the *text* properties.

5.16.3 Usage

Modes

The Keyboards have thefollowing modes:

- **LV_KEYBOARD_MODE_TEXT_LOWER** - Display lower case letters
- **LV_KEYBOARD_MODE_TEXT_UPPER** - Display upper case letters
- **LV_KEYBOARD_MODE_TEXT_SPECIAL** - Display special characters
- **LV_KEYBOARD_MODE_NUM** - Display numbers, +/- sign, and decimal dot.

The `TEXT` modes' layout contains buttons to change mode.

To set the mode manually, use `lv_keyboard_set_mode(kb, mode)`. The default more is `LV_KEYBOARD_MODE_TEXT_UPPER`.

Assign Text area

You can assign a *Text area* to the Keyboard to automatically put the clicked characters there. To assign the text area, use `lv_keyboard_set_textarea(kb, ta)`.

The assigned text area's **cursor can be managed** by the keyboard: when the keyboard is assigned, the previous text area's cursor will be hidden and the new one will be shown. When the keyboard is closed by the *Ok* or *Close* buttons, the cursor also will be hidden. The cursor manager feature is enabled by `lv_keyboard_set_cursor_manage(kb, true)`. The default is not managed.

New Keymap

You can specify a new map (layout) for the keyboard with `lv_keyboard_set_map(kb, map)` and `lv_keyboard_set_ctrl_map(kb, ctrl_map)`. Learn more about the *Button matrix* object. Keep in mind that, using following keywords will have the same effect as with the original map:

- *LV_SYMBOL_OK* - Apply.
- *LV_SYMBOL_CLOSE* - Close.
- *LV_SYMBOL_BACKSPACE* - Delete on the left.
- *LV_SYMBOL_LEFT* - Move the cursor left.
- *LV_SYMBOL_RIGHT* - Move the cursor right.
- *"ABC"* - Load the uppercase map.
- *"abc"* - Load the lower case map.
- *"Enter"* - New line.

5.16.4 Events

Besides the Generic events, the following Special events are sent by the keyboards:

- **LV_EVENT_VALUE_CHANGED** - Sent when the button is pressed/released or repeated after long press. The event data is set to the ID of the pressed/released button.
- **LV_EVENT_APPLY** - The *Ok* button is clicked.
- **LV_EVENT_CANCEL** - The *Close* button is clicked.

The keyboard has a **default event handler** callback called `lv_keyboard_def_event_cb`. It handles the button pressing, map changing, the assigned text area, etc. You can completely replace it with your custom event handler however, you can call `lv_keyboard_def_event_cb` at the beginning of your event handler to handle the same things as before.

Learn more about *Events*.

5.16.5 Keys

The following *Keys* are processed by the buttons:

- **LV_KEY_RIGHT/UP/LEFT/RIGHT** - To navigate among the buttons and select one.
- **LV_KEY_ENTER** - To press/release the selected button.

Learn more about *Keys*.

5.16.6 Examples

5.16.7 API

Typedefs

typedef uint8_t **lv_keyboard_mode_t**

typedef uint8_t **lv_keyboard_style_t**

Enums

enum [anonymous]
Current keyboard mode.

Values:

LV_KEYBOARD_MODE_TEXT_LOWER

LV_KEYBOARD_MODE_TEXT_UPPER

LV_KEYBOARD_MODE_SPECIAL

LV_KEYBOARD_MODE_NUM

enum [anonymous]
Values:

LV_KEYBOARD_PART_BG

LV_KEYBOARD_PART_BTN

Functions

lv_obj_t ***lv_keyboard_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a keyboard objects

Return pointer to the created keyboard

Parameters

- `par`: pointer to an object, it will be the parent of the new keyboard
- `copy`: pointer to a keyboard object, if not NULL then the new object will be copied from it

void **lv_keyboard_set_textarea**(*lv_obj_t *kb*, *lv_obj_t *ta*)
Assign a Text Area to the Keyboard. The pressed characters will be put there.

Parameters

- `kb`: pointer to a Keyboard object
- `ta`: pointer to a Text Area object to write there

void **lv_keyboard_set_mode**(*lv_obj_t *kb*, *lv_keyboard_mode_t mode*)
Set a new a mode (text or number map)

Parameters

- `kb`: pointer to a Keyboard object
- `mode`: the mode from 'lv_keyboard_mode_t'

void **lv_keyboard_set_cursor_manage(***lv_obj_t *kb*, bool *en***)**
Automatically hide or show the cursor of the current Text Area

Parameters

- kb: pointer to a Keyboard object
- en: true: show cursor on the current text area, false: hide cursor

void **lv_keyboard_set_map(***lv_obj_t *kb*, *lv_keyboard_mode_t mode*, **const** char **map*[]**)**
Set a new map for the keyboard

Parameters

- kb: pointer to a Keyboard object
- mode: keyboard map to alter 'lv_keyboard_mode_t'
- map: pointer to a string array to describe the map. See 'lv_btnmatrix_set_map()' for more info.

void **lv_keyboard_set_ctrl_map(***lv_obj_t *kb*, *lv_keyboard_mode_t mode*, **const** *lv_btnmatrix_ctrl_t ctrl_map*[]**)**
Set the button control map (hidden, disabled etc.) for the keyboard. The control map array will be copied and so may be deallocated after this function returns.

Parameters

- kb: pointer to a keyboard object
- mode: keyboard ctrl map to alter 'lv_keyboard_mode_t'
- ctrl_map: pointer to an array of lv_btn_ctrl_t control bytes. See: lv_btnmatrix_set_ctrl_map for more details.

lv_obj_t ***lv_keyboard_get_textarea(const** *lv_obj_t *kb***)**
Assign a Text Area to the Keyboard. The pressed characters will be put there.

Return pointer to the assigned Text Area object

Parameters

- kb: pointer to a Keyboard object

lv_keyboard_mode_t **lv_keyboard_get_mode(const** *lv_obj_t *kb***)**
Set a new a mode (text or number map)

Return the current mode from 'lv_keyboard_mode_t'

Parameters

- kb: pointer to a Keyboard object

bool **lv_keyboard_get_cursor_manage(const** *lv_obj_t *kb***)**
Get the current cursor manage mode.

Return true: show cursor on the current text area, false: hide cursor

Parameters

- kb: pointer to a Keyboard object

static const char ****lv_keyboard_get_map_array(const** *lv_obj_t *kb***)**
Get the current map of a keyboard

Return the current map

Parameters

- `kb`: pointer to a keyboard object

void **`lv_keyboard_def_event_cb`**(*lv_obj_t *kb*, *lv_event_t event*)

Default keyboard event to add characters to the Text area and change the map. If a custom `event_cb` is added to the keyboard this function be called from it to handle the button clicks

Parameters

- `kb`: pointer to a keyboard
- `event`: the triggering event

`struct lv_keyboard_ext_t`

Public Members

lv_btnmatrix_ext_t **`btnm`**

lv_obj_t ***`ta`**

lv_keyboard_mode_t **`mode`**

uint8_t **`cursor_mng`**

5.17 Label (lv_label)

5.17.1 Overview

A label is the basic object type that is used to display text.

5.17.2 Parts and Styles

The label has only a main part, called `LV_LABEL_PART_MAIN`. It uses all the typical background properties and the *text* properties. The padding values can be used to make the area for the text small in the related direction.

5.17.3 Usage

Set text

You can set the text on a label at runtime with `lv_label_set_text(label, "New text")`. It will allocate a buffer dynamically, and the provided string will be copied into that buffer. Therefore, you don't need to keep the text you pass to `lv_label_set_text` in scope after that function returns.

With `lv_label_set_text_fmt(label, "Value: %d", 15)` **printf formatting** can be used to set the text.

Labels are able to show text from a **static character buffer** which is `\0`-terminated. To do so, use `lv_label_set_static_text(label, "Text")`. In this case, the text is not stored in the dynamic memory and the given buffer is used directly instead. This means that the array can't be a local variable which goes out of scope when the function exits. Constant strings are safe to use with `lv_label_set_static_text` (except when used with `LV_LABEL_LONG_DOT`, as it modifies the buffer in-place), as they are stored in ROM memory, which is always accessible.

You can also use a **raw array** as label text. The array doesn't have to be `\0` terminated. In this case, the text will be saved to the dynamic memory like with `lv_label_set_text`. To set a raw character array, use the `lv_label_set_array_text(label, char_array, size)` function.

Line break

Line breaks are handled automatically by the label object. You can use `\n` to make a line break. For example: `"line1\nline2\n\nline4"`

Long modes

By default, the width of the label object automatically expands to the text size. Otherwise, the text can be manipulated according to several long mode policies:

- **LV_LABEL_LONG_EXPAND** - Expand the object size to the text size (Default)
- **LV_LABEL_LONG_BREAK** - Keep the object width, break (wrap) the too long lines and expand the object height
- **LV_LABEL_LONG_DOT** - Keep the object size, break the text and write dots in the last line (**not supported when using `lv_label_set_static_text`**)
- **LV_LABEL_LONG_SROLL** - Keep the size and scroll the label back and forth
- **LV_LABEL_LONG_SROLL_CIRC** - Keep the size and scroll the label circularly
- **LV_LABEL_LONG_CROP** - Keep the size and crop the text out of it

You can specify the long mode with `lv_label_set_long_mode(label, LV_LABEL_LONG_...)`

It's important to note that, when a label is created and its text is set, the label's size already expanded to the text size. In addition with the default `LV_LABEL_LONG_EXPAND`, *long mode* `lv_obj_set_width/height/size()` has no effect.

So you need to change the *long mode* first set the new *long mode* and then set the size with `lv_obj_set_width/height/size()`.

Another important note is that **`LV_LABEL_LONG_DOT` manipulates the text buffer in-place** in order to add/remove the dots. When `lv_label_set_text` or `lv_label_set_array_text` are used, a separate buffer is allocated and this implementation detail is unnoticed. This is not the case with `lv_label_set_static_text`! **The buffer you pass to `lv_label_set_static_text` must be writable if you plan to use `LV_LABEL_LONG_DOT`.**

Text align

The lines of the text can be aligned to the left, right or center with `lv_label_set_align(label, LV_LABEL_ALIGN_LEFT/RIGHT/CENTER)`. Note that, it will align only the lines, not the label object itself.

Vertical alignment is not supported by the label itself; you should place the label inside a larger container and align the whole label object instead.

Text recolor

In the text, you can use commands to recolor parts of the text. For example: `"Write a #ff0000 red#word"`. This feature can be enabled individually for each label by `lv_label_set_recolor()` function.

Note that, recoloring work only in a single line. Therefore, `\n` should not use in a recolored text or it should be wrapped by `LV_LABEL_LONG_BREAK` else, the text in the new line won't be recolored.

Very long texts

Lvgl can efficiently handle very long (> 40k characters) by saving some extra data (~12 bytes) to speed up drawing. To enable this feature, set `LV_LABEL_LONG_TXT_HINT 1` in *lv_conf.h*.

Symbols

The labels can display symbols alongside letters (or on their own). Read the *Font* section to learn more about the symbols.

5.17.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.17.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.17.6 Example

5.17.7 API

Typedefs

typedef uint8_t **lv_label_long_mode_t**

typedef uint8_t **lv_label_align_t**

typedef uint8_t **lv_label_part_t**

Enums

enum [**anonymous**]

Long mode behaviors. Used in '*lv_label_ext_t*'

Values:

LV_LABEL_LONG_EXPAND
Expand the object size to the text size

LV_LABEL_LONG_BREAK
Keep the object width, break the too long lines and expand the object height

LV_LABEL_LONG_DOT
Keep the size and write dots at the end if the text is too long

LV_LABEL_LONG_SROLL
Keep the size and roll the text back and forth

LV_LABEL_LONG_SROLL_CIRC
Keep the size and roll the text circularly

LV_LABEL_LONG_CROP
Keep the size and crop the text out of it

enum [**anonymous**]
Label align policy

Values:

LV_LABEL_ALIGN_LEFT
Align text to left

LV_LABEL_ALIGN_CENTER
Align text to center

LV_LABEL_ALIGN_RIGHT
Align text to right

LV_LABEL_ALIGN_AUTO
Use LEFT or RIGHT depending on the direction of the text (LTR/RTL)

enum [**anonymous**]
Label styles

Values:

LV_LABEL_PART_MAIN

Functions

LV_EXPORT_CONST_INT(LV_LABEL_DOT_NUM)

LV_EXPORT_CONST_INT(LV_LABEL_POS_LAST)

LV_EXPORT_CONST_INT(LV_LABEL_TEXT_SEL_OFF)

lv_obj_t ***lv_label_create**(*lv_obj_t* **par*, **const** *lv_obj_t* **copy*)
Create a label objects

Return pointer to the created button

Parameters

- `par`: pointer to an object, it will be the parent of the new label
- `copy`: pointer to a button object, if not NULL then the new object will be copied from it

void **lv_label_set_text**(*lv_obj_t* **label*, **const** char **text*)
Set a new text for a label. Memory will be allocated to store the text by the label.

Parameters

- `label`: pointer to a label object
- `text`: '\0' terminated character string. NULL to refresh with the current text.

void **lv_label_set_text_fmt**(*lv_obj_t* **label*, **const** char **fmt*, ...)
Set a new formatted text for a label. Memory will be allocated to store the text by the label.

Parameters

- `label`: pointer to a label object
- `fmt`: `printf`-like format

void **`lv_label_set_text_static`**(*lv_obj_t *label*, **const** char **text*)
Set a static text. It will not be saved by the label so the 'text' variable has to be 'alive' while the label exist.

Parameters

- `label`: pointer to a label object
- `text`: pointer to a text. NULL to refresh with the current text.

void **`lv_label_set_long_mode`**(*lv_obj_t *label*, *lv_label_long_mode_t long_mode*)
Set the behavior of the label with longer text then the object size

Parameters

- `label`: pointer to a label object
- `long_mode`: the new mode from 'lv_label_long_mode' enum. In LV_LONG_BREAK/LONG/ROLL the size of the label should be set AFTER this function

void **`lv_label_set_align`**(*lv_obj_t *label*, *lv_label_align_t align*)
Set the align of the label (left or center)

Parameters

- `label`: pointer to a label object
- `align`: 'LV_LABEL_ALIGN_LEFT' or 'LV_LABEL_ALIGN_LEFT'

void **`lv_label_set_recolor`**(*lv_obj_t *label*, bool *en*)
Enable the recoloring by in-line commands

Parameters

- `label`: pointer to a label object
- `en`: true: enable recoloring, false: disable

void **`lv_label_set_anim_speed`**(*lv_obj_t *label*, uint16_t *anim_speed*)
Set the label's animation speed in LV_LABEL_LONG_SROLL/SCROLL_CIRC modes

Parameters

- `label`: pointer to a label object
- `anim_speed`: speed of animation in px/sec unit

void **`lv_label_set_text_sel_start`**(*lv_obj_t *label*, uint32_t *index*)
Set the selection start index.

Parameters

- `label`: pointer to a label object.
- `index`: index to set. `LV_LABEL_TXT_SEL_OFF` to select nothing.

void **`lv_label_set_text_sel_end`**(*lv_obj_t *label*, uint32_t *index*)
Set the selection end index.

Parameters

- `label`: pointer to a label object.
- `index`: index to set. `LV_LABEL_TXT_SEL_OFF` to select nothing.

char *`lv_label_get_text(const` *lv_obj_t *label*`)`
Get the text of a label

Return the text of the label

Parameters

- `label`: pointer to a label object

lv_label_long_mode_t `lv_label_get_long_mode(const` *lv_obj_t *label*`)`
Get the long mode of a label

Return the long mode

Parameters

- `label`: pointer to a label object

lv_label_align_t `lv_label_get_align(const` *lv_obj_t *label*`)`
Get the align attribute

Return LV_LABEL_ALIGN_LEFT or LV_LABEL_ALIGN_CENTER

Parameters

- `label`: pointer to a label object

bool `lv_label_get_recolor(const` *lv_obj_t *label*`)`
Get the recoloring attribute

Return true: recoloring is enabled, false: disable

Parameters

- `label`: pointer to a label object

uint16_t `lv_label_get_anim_speed(const` *lv_obj_t *label*`)`
Get the label's animation speed in LV_LABEL_LONG_ROLL and SCROLL modes

Return speed of animation in px/sec unit

Parameters

- `label`: pointer to a label object

void `lv_label_get_letter_pos(const` *lv_obj_t *label*, uint32_t *index*, lv_point_t **pos*`)`
Get the relative x and y coordinates of a letter

Parameters

- `label`: pointer to a label object
- `index`: index of the letter [0 ... text length]. Expressed in character index, not byte index (different in UTF-8)
- `pos`: store the result here (E.g. index = 0 gives 0;0 coordinates)

uint32_t `lv_label_get_letter_on(const` *lv_obj_t *label*, lv_point_t **pos*`)`
Get the index of letter on a relative point of a label

Return the index of the letter on the 'pos_p' point (E.g. on 0;0 is the 0. letter) Expressed in character index and not byte index (different in UTF-8)

Parameters

- `label`: pointer to label object
- `pos`: pointer to point with coordinates on a the label

bool **`lv_label_is_char_under_pos(const`** *lv_obj_t *label*, lv_point_t **pos*)
Check if a character is drawn under a point.

Return whether a character is drawn under the point

Parameters

- `label`: Label object
- `pos`: Point to check for character under

uint32_t **`lv_label_get_text_sel_start(const`** *lv_obj_t *label*)
Get the selection start index.

Return selection start index. `LV_LABEL_TXT_SEL_OFF` if nothing is selected.

Parameters

- `label`: pointer to a label object.

uint32_t **`lv_label_get_text_sel_end(const`** *lv_obj_t *label*)
Get the selection end index.

Return selection end index. `LV_LABEL_TXT_SEL_OFF` if nothing is selected.

Parameters

- `label`: pointer to a label object.

lv_style_list_t ***`lv_label_get_style(`***lv_obj_t *label*, uint8_t *type*)

void **`lv_label_ins_text(`***lv_obj_t *label*, uint32_t *pos*, **`const`** char **txt*)
Insert a text to the label. The label text can not be static.

Parameters

- `label`: pointer to a label object
- `pos`: character index to insert. Expressed in character index and not byte index (Different in UTF-8) 0: before first char. LV_LABEL_POS_LAST: after last char.
- `txt`: pointer to the text to insert

void **`lv_label_cut_text(`***lv_obj_t *label*, uint32_t *pos*, uint32_t *cnt*)
Delete characters from a label. The label text can not be static.

Parameters

- `label`: pointer to a label object
- `pos`: character index to insert. Expressed in character index and not byte index (Different in UTF-8) 0: before first char.
- `cnt`: number of characters to cut

`struct lv_label_ext_t`
#include <lv_label.h> Data of label

Public Members

char *text

char *tmp_ptr

char tmp[LV_LABEL_DOT_NUM + 1]

union *lv_label_ext_t*::[anonymous] dot

uint32_t dot_end

uint16_t anim_speed

lv_point_t offset

lv_draw_label_hint_t hint

uint32_t sel_start

uint32_t sel_end

lv_label_long_mode_t long_mode

uint8_t static_txt

uint8_t align

uint8_t recolor

uint8_t expand

uint8_t dot_tmp_alloc

5.18 LED (lv_led)

5.18.1 Overview

The LEDs are rectangle-like (or circle) object. It's brightness can be adjusted. With lower brightness the the colors of the LED become darker.

5.18.2 Parts and Styles

The LEDs have only one main part, called `LV_LED_PART_MAIN` and it uses all the typical background style properties.

5.18.3 Usage

Brightness

You can set their brightness with `lv_led_set_bright(led, bright)`. The brightness should be between 0 (darkest) and 255 (lightest).

Toggle

Use `lv_led_on(led)` and `lv_led_off(led)` to set the brightness to a predefined ON or OFF value. The `lv_led_toggle(led)` toggles between the ON and OFF state.

5.18.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.18.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.18.6 Example

5.18.7 API

Typedefs

typedef uint8_t **lv_led_part_t**

Enums

enum [anonymous]

Values:

LV_LED_PART_MAIN = *LV_OBJ_PART_MAIN*

Functions

lv_obj_t ***lv_led_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a led objects

Return pointer to the created led

Parameters

- `par`: pointer to an object, it will be the parent of the new led
- `copy`: pointer to a led object, if not NULL then the new object will be copied from it

void **lv_led_set_bright**(*lv_obj_t *led*, uint8_t *bright*)

Set the brightness of a LED object

Parameters

- `led`: pointer to a LED object
- `bright`: LV_LED_BRIGHT_MIN (max. dark) ... LV_LED_BRIGHT_MAX (max. light)

void **lv_led_on**(*lv_obj_t *led*)

Light on a LED

Parameters

- `led`: pointer to a LED object

void **lv_led_off**(*lv_obj_t *led*)
Light off a LED

Parameters

- `led`: pointer to a LED object

void **lv_led_toggle**(*lv_obj_t *led*)
Toggle the state of a LED

Parameters

- `led`: pointer to a LED object

uint8_t **lv_led_get_bright**(**const** *lv_obj_t *led*)
Get the brightness of a LEd object

Return bright 0 (max. dark) ... 255 (max. light)

Parameters

- `led`: pointer to LED object

struct lv_led_ext_t

Public Members

uint8_t **bright**

5.19 Line (lv_line)

5.19.1 Overview

The Line object is capable of drawing straight lines between a set of points.

5.19.2 Parts and Styles

The Line has only a main part, called `LV_LABEL_PART_MAIN`. It uses all the *line* style properties.

5.19.3 Usage

Set points

The points has to be stored in an `lv_point_t` array and passed to the object by the `lv_line_set_points(lines, point_array, point_cnt)` function.

Auto-size

It is possible to automatically set the size of the line object according to its points. It can be enable with the `lv_line_set_auto_size(line, true)` function. If enabled then when the points are set the object's width and height will be changed according to the maximal x and y coordinates among the points. The *auto size* is enabled by default.

Invert y

By deafult, the *y == 0* point is in the top of the object. It might be conter-intuitive in some cases so the y coordinates can be inverted with `lv_line_set_y_invert(line, true)`. In this case, *y == 0* will be the bottom of teh obejct. The *y invert* is disabled by default.

5.19.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.19.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.19.6 Example

5.19.7 API

Typedefs

typedef uint8_t **lv_line_style_t**

Enums

enum [**anonymous**]

Values:

LV_LINE_PART_MAIN

Functions

lv_obj_t ***lv_line_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a line objects

Return pointer to the created line

Parameters

- `par`: pointer to an object, it will be the parent of the new line

void **lv_line_set_points**(*lv_obj_t *line*, **const** lv_point_t *point_a*[], uint16_t *point_num*)

Set an array of points. The line object will connect these points.

Parameters

- `line`: pointer to a line object
- `point_a`: an array of points. Only the address is saved, so the array can NOT be a local variable which will be destroyed
- `point_num`: number of points in 'point_a'

void **lv_line_set_auto_size**(*lv_obj_t *line*, bool *en*)
Enable (or disable) the auto-size option. The size of the object will fit to its points. (set width to x max and height to y max)

Parameters

- `line`: pointer to a line object
- `en`: true: auto size is enabled, false: auto size is disabled

void **lv_line_set_y_invert**(*lv_obj_t *line*, bool *en*)
Enable (or disable) the y coordinate inversion. If enabled then y will be subtracted from the height of the object, therefore the y=0 coordinate will be on the bottom.

Parameters

- `line`: pointer to a line object
- `en`: true: enable the y inversion, false:disable the y inversion

bool **lv_line_get_auto_size**(**const** *lv_obj_t *line*)
Get the auto size attribute

Return true: auto size is enabled, false: disabled

Parameters

- `line`: pointer to a line object

bool **lv_line_get_y_invert**(**const** *lv_obj_t *line*)
Get the y inversion attribute

Return true: y inversion is enabled, false: disabled

Parameters

- `line`: pointer to a line object

struct lv_line_ext_t

Public Members

const lv_point_t ***point_array**

uint16_t **point_num**

uint8_t **auto_size**

uint8_t **y_inv**

5.20 List (lv_list)

5.20.1 Overview

The Lists are built from a background *Page* and *Buttons* on it. The Buttons contain an optional icon-like *Image* (which can be a symbol too) and a *Label*. When the list becomes long enough it can be scrolled.

5.20.2 Parts and Styles

The List has the same parts as the *Page*

- `LV_LIST_PART_BG`
- `LV_LIST_PART_SCRL`
- `LV_LIST_PART_SCRLBAR`
- `LV_LIST_PART_EDGE_FLASH`

Refer to the *Page* documentation for details.

The buttons on the list are treated as normal buttons and they only have a main part called `LV_BTN_PART_MAIN`.

5.20.3 Usage

Add buttons

You can add new list elements (button) with `lv_list_add_btn(list, &icon_img, "Text")` or with symbol `lv_list_add_btn(list, SYMBOL_EDIT, "Edit text")`. If you do not want to add image use `NULL` as image source. The function returns with a pointer to the created button to allow further configurations.

The width of the buttons is set to maximum according to the object width. The height of the buttons are adjusted automatically according to the content. (*content height* + *padding_top* + *padding_bottom*).

The labels are created with `LV_LABEL_LONG_SROLL_CIRC` long mode to automatically scroll the long labels circularly.

`lv_list_get_btn_label(list_btn)` and `lv_list_get_btn_img(list_btn)` can be used to get the label and the image of a list button. The text can be et directly with `lv_list_get_btn_text(list_btn)`.

Delete buttons

To delete a list element just use `lv_obj_del(btn)` on the return value of `lv_list_add_btn()`.

To clean the list (remove all buttons) use `lv_list_clean(list)`

Manual navigation

You can navigate manually in the list with `lv_list_up(list)` and `lv_list_down(list)`.

You can focus on a button directly using `lv_list_focus(btn, LV_ANIM_ON/OFF)`.

The **animation time** of up/down/focus movements can be set via: `lv_list_set_anim_time(list, anim_time)`. Zero animation time means not animations.

Layout

By default the list is vertical. To get a horizontal list use `lv_list_set_layout(list, LV_LAYOUT_ROW_MID)`.

Edge flash

A circle-like effect can be shown when the list reaches the most top or bottom position. `lv_list_set_edge_flash(list, true)` enables this feature.

Scroll propagation

If the list is created on an other scrollable element (like a *Page*) and the list can't be scrolled further the scrolling can be propagated to the parent. This way the scroll will be continued on the parent. It can be enabled with `lv_list_set_scroll_propagation(list, true)`

5.20.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.20.5 Keys

The following *Keys* are processed by the Lists:

- **LV_KEY_RIGHT/DOWN** Select the next button
- **LV_KEY_LEFT/UP** Select the previous button

Note that, as usual, the state of `LV_KEY_ENTER` is translated to `LV_EVENT_PRESSED/PRESSING/RELEASED` etc.

The Selected buttons are in `LV_BTN_STATE_PR/TG_PR` state.

To manually select a button use `lv_list_set_btn_selected(list, btn)`. When the list is defocused and focused again it will restore the last selected button.

Learn more about *Keys*.

5.20.6 Example

5.20.7 API

Typedefs

typedef uint8_t **lv_list_style_t**

Enums

enum [**anonymous**]

List styles.

Values:

LV_LIST_PART_BG = *LV_PAGE_PART_BG*

List background style

LV_LIST_PART_SCROLLBAR = *LV_PAGE_PART_SCROLLBAR*

List scrollbar style.

LV_LIST_PART_EDGE_FLASH = *LV_PAGE_PART_EDGE_FLASH*
List edge flash style.

_LV_LIST_PART_VIRTUAL_LAST = *_LV_PAGE_PART_VIRTUAL_LAST*

LV_LIST_PART_SCROLLABLE = *LV_PAGE_PART_SCROLLABLE*
List scrollable area style.

_LV_LIST_PART_REAL_LAST = *_LV_PAGE_PART_REAL_LAST*

Functions

lv_obj_t ***lv_list_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a list objects

Return pointer to the created list

Parameters

- `par`: pointer to an object, it will be the parent of the new list
- `copy`: pointer to a list object, if not NULL then the new object will be copied from it

void **lv_list_clean**(*lv_obj_t *list*)
Delete all children of the scrl object, without deleting scrl child.

Parameters

- `list`: pointer to an object

lv_obj_t ***lv_list_add_btn**(*lv_obj_t *list*, **const** void **img_src*, **const** char **txt*)
Add a list element to the list

Return pointer to the new list element which can be customized (a button)

Parameters

- `list`: pointer to list object
- `img_fn`: file name of an image before the text (NULL if unused)
- `txt`: text of the list element (NULL if unused)

bool **lv_list_remove**(**const** *lv_obj_t *list*, uint16_t *index*)
Remove the index of the button in the list

Return true: successfully deleted

Parameters

- `list`: pointer to a list object
- `index`: pointer to a the button's index in the list, index must be 0 <= index < lv_list_ext_t.size

void **lv_list_focus_btn**(*lv_obj_t *list*, *lv_obj_t *btn*)
Make a button selected

Parameters

- `list`: pointer to a list object
- `btn`: pointer to a button to select NULL to not select any buttons

static void **lv_list_set_scrollbar_mode**(*lv_obj_t *list*, *lv_scrollbar_mode_t mode*)
Set the scroll bar mode of a list

Parameters

- `list`: pointer to a list object
- `sb_mode`: the new mode from 'lv_page_sb_mode_t' enum

static void **lv_list_set_scroll_propagation(***lv_obj_t *list*, bool *en***)**
Enable the scroll propagation feature. If enabled then the List will move its parent if there is no more space to scroll.

Parameters

- `list`: pointer to a List
- `en`: true or false to enable/disable scroll propagation

static void **lv_list_set_edge_flash(***lv_obj_t *list*, bool *en***)**
Enable the edge flash effect. (Show an arc when the an edge is reached)

Parameters

- `list`: pointer to a List
- `en`: true or false to enable/disable end flash

static void **lv_list_set_anim_time(***lv_obj_t *list*, uint16_t *anim_time***)**
Set scroll animation duration on 'list_up()' 'list_down()' 'list_focus()'

Parameters

- `list`: pointer to a list object
- `anim_time`: duration of animation [ms]

void **lv_list_set_layout(***lv_obj_t *list*, *lv_layout_t layout***)**
Set layout of a list

Parameters

- `list`: pointer to a list object
- `layout`: which layout should be used

const char ***lv_list_get_btn_text(const** *lv_obj_t *btn***)**
Get the text of a list element

Return pointer to the text

Parameters

- `btn`: pointer to list element

lv_obj_t ***lv_list_get_btn_label(const** *lv_obj_t *btn***)**
Get the label object from a list element

Return pointer to the label from the list element or NULL if not found

Parameters

- `btn`: pointer to a list element (button)

lv_obj_t ***lv_list_get_btn_img(const** *lv_obj_t *btn***)**
Get the image object from a list element

Return pointer to the image from the list element or NULL if not found

Parameters

- `btn`: pointer to a list element (button)

lv_obj_t ***lv_list_get_prev_btn(const** *lv_obj_t *list*, *lv_obj_t *prev_btn*)
Get the next button from list. (Starts from the bottom button)

Return pointer to the next button or NULL when no more buttons

Parameters

- `list`: pointer to a list object
- `prev_btn`: pointer to button. Search the next after it.

lv_obj_t ***lv_list_get_next_btn(const** *lv_obj_t *list*, *lv_obj_t *prev_btn*)
Get the previous button from list. (Starts from the top button)

Return pointer to the previous button or NULL when no more buttons

Parameters

- `list`: pointer to a list object
- `prev_btn`: pointer to button. Search the previous before it.

int32_t **lv_list_get_btn_index(const** *lv_obj_t *list*, **const** *lv_obj_t *btn*)
Get the index of the button in the list

Return the index of the button in the list, or -1 of the button not in this list

Parameters

- `list`: pointer to a list object. If NULL, assumes btn is part of a list.
- `btn`: pointer to a list element (button)

uint16_t **lv_list_get_size(const** *lv_obj_t *list*)
Get the number of buttons in the list

Return the number of buttons in the list

Parameters

- `list`: pointer to a list object

lv_obj_t ***lv_list_get_btn_selected(const** *lv_obj_t *list*)
Get the currently selected button. Can be used while navigating in the list with a keypad.

Return pointer to the selected button

Parameters

- `list`: pointer to a list object

lv_layout_t **lv_list_get_layout(***lv_obj_t *list*)
Get layout of a list

Return layout of the list object

Parameters

- `list`: pointer to a list object

static *lv_scrollbar_mode_t* **lv_list_get_scrollbar_mode(const** *lv_obj_t *list*)
Get the scroll bar mode of a list

Return scrollbar mode from 'lv_scrollbar_mode_t' enum

Parameters

- `list`: pointer to a list object

static bool **lv_list_get_scroll_propagation**(*lv_obj_t *list*)
Get the scroll propagation property

Return true or false

Parameters

- `list`: pointer to a List

static bool **lv_list_get_edge_flash**(*lv_obj_t *list*)
Get the scroll propagation property

Return true or false

Parameters

- `list`: pointer to a List

static uint16_t **lv_list_get_anim_time**(**const** *lv_obj_t *list*)
Get scroll animation duration

Return duration of animation [ms]

Parameters

- `list`: pointer to a list object

void **lv_list_up**(**const** *lv_obj_t *list*)
Move the list elements up by one

Parameters

- `list`: pointer a to list object

void **lv_list_down**(**const** *lv_obj_t *list*)
Move the list elements down by one

Parameters

- `list`: pointer to a list object

void **lv_list_focus**(**const** *lv_obj_t *btn*, *lv_anim_enable_t anim*)
Focus on a list button. It ensures that the button will be visible on the list.

Parameters

- `btn`: pointer to a list button to focus
- `anim`: LV_ANOM_ON: scroll with animation, LV_ANIM_OFF: without animation

struct lv_list_ext_t

Public Members

lv_page_ext_t **page**

lv_obj_t ***last_sel_btn**

lv_obj_t ***act_sel_btn**

5.21 Line meter (lv_lmeter)

5.21.1 Overview

The Line meter object consists of some radial lines which draw a scale. Setting a value for the Line meter will change the color of the scale lines proportionally.

5.21.2 Parts and Styles

The Line meter has only a main part, called `LV_LINEMETER_PART_MAIN`. It uses all the typical background properties the draw a rectangle-like or circle background and the *line* and *scale* properties to draw the scale lines. The active lines (which are related to smaller values the the current value) are colored from *line_color* to *scale_grad_color*. The lines in the end (after the current value) are set to *scale_end_color* color.

5.21.3 Usage

Set value

When setting a new value with `lv_linemeter_set_value(linemeter, new_value)` the proportional part of the scale will be recolored.

Range and Angles

The `lv_linemeter_set_range(linemeter, min, max)` function sets the range of the line meter.

You can set the angle of the scale and the number of the lines by: `lv_linemeter_set_scale(linemeter, angle, line_num)`. The default angle is 240 and the default line number is 31.

Angle offset

By default the scale angle is interpreted symmetrically to the y axis. It results in "standing" line meter. With `lv_linemeter_set_angle_offset` an offset can be added the scale angle. It can used e.g to put a quarter line meter into a corner or a half line meter to the right or left side.

Mirror

By default the Line meter's lines are activated clock-wise. It can be changed using `lv_linemeter_set_mirror(linemeter, true/false)`.

5.21.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.21.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.21.6 Example

5.21.7 API

Typedefs

typedef uint8_t **lv_linemeter_part_t**

Enums

enum [anonymous]

Values:

LV_LINEMETER_PART_MAIN

_LV_LINEMETER_PART_VIRTUAL_LAST

_LV_LINEMETER_PART_REAL_LAST = *_LV_OBJ_PART_REAL_LAST*

Functions

lv_obj_t ***lv_linemeter_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a line meter objects

Return pointer to the created line meter

Parameters

- `par`: pointer to an object, it will be the parent of the new line meter
- `copy`: pointer to a line meter object, if not NULL then the new object will be copied from it

void **lv_linemeter_set_value**(*lv_obj_t *lmeter*, int32_t *value*)

Set a new value on the line meter

Parameters

- `lmeter`: pointer to a line meter object
- `value`: new value

void **lv_linemeter_set_range**(*lv_obj_t *lmeter*, int32_t *min*, int32_t *max*)

Set minimum and the maximum values of a line meter

Parameters

- `lmeter`: pointer to he line meter object
- `min`: minimum value
- `max`: maximum value

void **lv_linemeter_set_scale**(*lv_obj_t *lmeter*, uint16_t *angle*, uint16_t *line_cnt*)

Set the scale settings of a line meter

Parameters

- `lmeter`: pointer to a line meter object
- `angle`: angle of the scale (0..360)
- `line_cnt`: number of lines

void **lv_linemeter_set_angle_offset(***lv_obj_t *lmeter*, uint16_t *angle***)**
Set the set an offset for the line meter's angles to rotate it.

Parameters

- `lmeter`: pointer to a line meter object
- `angle`: angle offset (0..360), rotates clockwise

void **lv_linemeter_set_mirror(***lv_obj_t *lmeter*, bool *mirror***)**
Set the orientation of the meter growth, clockwise or counterclockwise (mirrored)

Parameters

- `lmeter`: pointer to a line meter object
- `mirror`: mirror setting

int32_t **lv_linemeter_get_value(const** *lv_obj_t *lmeter***)**
Get the value of a line meter

Return the value of the line meter

Parameters

- `lmeter`: pointer to a line meter object

int32_t **lv_linemeter_get_min_value(const** *lv_obj_t *lmeter***)**
Get the minimum value of a line meter

Return the minimum value of the line meter

Parameters

- `lmeter`: pointer to a line meter object

int32_t **lv_linemeter_get_max_value(const** *lv_obj_t *lmeter***)**
Get the maximum value of a line meter

Return the maximum value of the line meter

Parameters

- `lmeter`: pointer to a line meter object

uint16_t **lv_linemeter_get_line_count(const** *lv_obj_t *lmeter***)**
Get the scale number of a line meter

Return number of the scale units

Parameters

- `lmeter`: pointer to a line meter object

uint16_t **lv_linemeter_get_scale_angle(const** *lv_obj_t *lmeter***)**
Get the scale angle of a line meter

Return angle of the scale

Parameters

- `lmeter`: pointer to a line meter object

uint16_t **lv_linemeter_get_angle_offset**(*lv_obj_t *lmeter*)

Get the offset for the line meter.

Return angle offset (0..360)

Parameters

- `lmeter`: pointer to a line meter object

void **lv_linemeter_draw_scale**(*lv_obj_t *lmeter*, **const** lv_area_t **clip_area*, uint8_t *part*)

bool **lv_linemeter_get_mirror**(*lv_obj_t *lmeter*)

get the mirror setting for the line meter

Return mirror (true or false)

Parameters

- `lmeter`: pointer to a line meter object

struct lv_linemeter_ext_t

Public Members

uint16_t **scale_angle**

uint16_t **angle_ofs**

uint16_t **line_cnt**

int32_t **cur_value**

int32_t **min_value**

int32_t **max_value**

uint8_t **mirrored**

5.22 Message box (lv_msdbox)

5.22.1 Overview

The Message boxes act as pop-ups. They are built from a background *Container*, a *Label* and a *Button matrix* for buttons.

The text will be broken into multiple lines automatically (has `LV_LABEL_LONG_MODE_BREAK`) and the height will be set automatically to involve the text and the buttons (`LV_FIT_TIGHT` fit vertically)-

5.22.2 Parts and Styles

The Message box's main part is called `LV_MSGBOX_PART_MAIN` and it uses all the typical background style properties. Using padding will add space on the sides. *pad_inner* will add space between the text and the buttons. The *label* style properties affect the style of text.

The buttons parts are the same as in case of *Button matrix*:

- `LV_MSGBOX_PART_BTN_BG` the background of the buttons

- `LV_MSGBOX_PART_BTN` the buttons

5.22.3 Usage

Set text

To set the text use the `lv_msgbox_set_text(msgbox, "My text")` function. Not only the pointer of the text will be saved, so the the text can be in a local variable too.

Add buttons

To add buttons use the `lv_msgbox_add_btns(msgbox, btn_str)` function. The button's text needs to be specified like `const char * btn_str[] = {"Apply", "Close", ""}`. For more information visit the *Button matrix* documentation.

The button matrix will be created only when `lv_msgbox_add_btns()` is called for the first time.

Auto-close

With `lv_msgbox_start_auto_close(mbox, delay)` the message box can be closed automatically after `delay` milliseconds with an animation. The `lv_mbox_stop_auto_close(mbox)` function stops a started auto close.

The duration of the close animation can be set by `lv_mbox_set_anim_time(mbox, anim_time)`.

5.22.4 Events

Besides the Generic events the following Special events are sent by the Message boxes:

- **LV_EVENT_VALUE_CHANGED** sent when the button is clicked. The event data is set to ID of the clicked button.

The Message box has a default event callback which closes itself when a button is clicked.

Learn more about *Events*.

##Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_RIGHT/DOWN** Select the next button
- **LV_KEY_LEFT/TOP** Select the previous button
- **LV_KEY_ENTER** Clicks the selected button

Learn more about *Keys*.

5.22.5 Example

5.22.6 API

Typedefs

typedef uint8_t **lv_msgbox_style_t**

Enums

enum [anonymous]
Message box styles.

Values:

LV_MSGBOX_PART_BG = *LV_CONT_PART_MAIN*

LV_MSGBOX_PART_BTN_BG = *_LV_CONT_PART_REAL_LAST*

LV_MSGBOX_PART_BTN

Functions

lv_obj_t ***lv_msgbox_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a message box objects

Return pointer to the created message box

Parameters

- `par`: pointer to an object, it will be the parent of the new message box
- `copy`: pointer to a message box object, if not NULL then the new object will be copied from it

void **lv_msgbox_add_btns**(*lv_obj_t *mbox*, **const** char **btn_mapaction*[])
Add button to the message box

Parameters

- `mbox`: pointer to message box object
- `btn_map`: button descriptor (button matrix map). E.g. a const char *txt[] = {"ok", "close", ""} (Can not be local variable)

void **lv_msgbox_set_text**(*lv_obj_t *mbox*, **const** char **txt*)
Set the text of the message box

Parameters

- `mbox`: pointer to a message box
- `txt`: a '\0' terminated character string which will be the message box text

void **lv_msgbox_set_anim_time**(*lv_obj_t *mbox*, uint16_t *anim_time*)
Set animation duration

Parameters

- `mbox`: pointer to a message box object
- `anim_time`: animation length in milliseconds (0: no animation)

void **lv_msgbox_start_auto_close**(*lv_obj_t *mbox*, uint16_t *delay*)
Automatically delete the message box after a given time

Parameters

- `mbox`: pointer to a message box object
- `delay`: a time (in milliseconds) to wait before delete the message box

void **lv_msgbox_stop_auto_close**(*lv_obj_t *mbox*)
Stop the auto. closing of message box

Parameters

- `mbox`: pointer to a message box object

void **`lv_msgbox_set_recolor(`***lv_obj_t *mbox*, bool *en*`)**
Set whether recoloring is enabled. Must be called after `lv_msgbox_add_btns`.

Parameters

- `mbox`: pointer to message box object
- `en`: whether recoloring is enabled

`const` char ***`lv_msgbox_get_text(const`** *lv_obj_t *mbox*`)**
Get the text of the message box

Return pointer to the text of the message box

Parameters

- `mbox`: pointer to a message box object

uint16_t **`lv_msgbox_get_active_btn(`***lv_obj_t *mbox*`)**
Get the index of the lastly "activated" button by the user (pressed, released etc) Useful in the the `event_cb`.

Return index of the last released button (LV_BTNMATRIX_BTN_NONE: if unset)

Parameters

- `mbox`: pointer to message box object

`const` char ***`lv_msgbox_get_active_btn_text(`***lv_obj_t *mbox*`)**
Get the text of the lastly "activated" button by the user (pressed, released etc) Useful in the the `event_cb`.

Return text of the last released button (NULL: if unset)

Parameters

- `mbox`: pointer to message box object

uint16_t **`lv_msgbox_get_anim_time(const`** *lv_obj_t *mbox*`)**
Get the animation duration (close animation time)

Return animation length in milliseconds (0: no animation)

Parameters

- `mbox`: pointer to a message box object

bool **`lv_msgbox_get_recolor(const`** *lv_obj_t *mbox*`)**
Get whether recoloring is enabled

Return whether recoloring is enabled

Parameters

- `mbox`: pointer to a message box object

lv_obj_t ***`lv_msgbox_get_btnmatrix(`***lv_obj_t *mbox*`)**
Get message box button matrix

Return pointer to button matrix object

Remark return value will be NULL unless `lv_msgbox_add_btns` has been already called

Parameters

- `mbox`: pointer to a message box object

struct lv_msgbox_ext_t

Public Members

lv_cont_ext_t **bg**

lv_obj_t ***text**

lv_obj_t ***btnm**

uint16_t **anim_time**

5.23 Object mask (lv_objmask)

5.23.1 Overview

The *Object mask* is capable of add some mask to drawings when its children is drawn.

5.23.2 Parts and Styles

The Object mask has only a main part called `LV_OBJMASK_PART_BG` and it uses the typical background style properties.

5.23.3 Usage

Adding mask

Before adding a mask to the *Object mask* the mask should be initialized:

```
lv_draw_mask_<type>_param_t mask_param;
lv_draw_mask_<type>_init(&mask_param, ...);
lv_objmask_mask_t * mask_p = lv_objmask_add_mask(objmask, &mask_param);
```

Lvgl supports the following mask types:

- **line** clip the pixels on the top/bottom left/right of a line. Can be initialized from two points or a point and an angle:
- **angle** keep the pixels only between a given start and end angle
- **radius** keep the pixel only inside a rectangle which can have radius (can for a circle too). Can be inverted to keep the pixel outside of the rectangle.
- **fade** fade vertically (change the pixels opacity according to their y position)
- **map** use an alpha mask (a byte array) to describe the pixels opacity.

The coordinates in the mask are relative to the Object. That is if the object moves the masks move with it.

For the details of the mask *init* function see the *API* documentation below.

Update mask

AN existing mask can be updated with `lv_objmask_update_mask(objmask, mask_p, new_param)`, where `mask_p` is return value of `lv_objmask_add_mask`.

Remove mask

A mask can be removed with `lv_objmask_remove_mask(objmask, mask_p)`

5.23.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.23.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.23.6 Example

5.23.7 API

Typedefs

typedef uint8_t **lv_objmask_part_t**

Enums

enum [**anonymous**]

Values:

LV_OBJMASK_PART_MAIN

Functions

lv_obj_t ***lv_objmask_create**(*lv_obj_t* **par*, **const** *lv_obj_t* **copy*)

Create a object mask objects

Return pointer to the created object mask

Parameters

- `par`: pointer to an object, it will be the parent of the new object mask
- `copy`: pointer to a object mask object, if not NULL then the new object will be copied from it

lv_objmask_mask_t ***lv_objmask_add_mask**(*lv_obj_t* **objmask*, void **param*)

Add a mask

Return pointer to the added mask

Parameters

- `objmask`: pointer to an Object mask object
- `param`: an initialized mask parameter

void **`lv_objmask_update_mask`**(*lv_obj_t *objmask*, *lv_objmask_mask_t *mask*, void **param*)

Update an already created mask

Parameters

- `objmask`: pointer to an Object mask object
- `mask`: pointer to created mask (returned by `lv_objmask_add_mask`)
- `param`: an initialized mask parameter (initialized by `lv_draw_mask_line/angle/.../_init`)

void **`lv_objmask_remove_mask`**(*lv_obj_t *objmask*, *lv_objmask_mask_t *mask*)

Remove a mask

Parameters

- `objmask`: pointer to an Object mask object
- `mask`: pointer to created mask (returned by `lv_objmask_add_mask`) If `NULL` passed all masks will be deleted.

`struct lv_objmask_mask_t`

Public Members

void ***param**

`struct lv_objmask_ext_t`

Public Members

lv_cont_ext_t **cont**

lv_ll_t **mask_ll**

Typedefs

typedef uint8_t **lv_draw_mask_res_t**

typedef uint8_t **lv_draw_mask_type_t**

typedef *lv_draw_mask_res_t* (***lv_draw_mask_xcb_t**)(lv_opa_t *mask_buf, lv_coord_t abs_x, lv_coord_t abs_y, lv_coord_t len, void *p)

A common callback type for every mask type. Used internally by the library.

typedef uint8_t **lv_draw_mask_line_side_t**

typedef struct *__lv_draw_mask_map_param_t* **lv_draw_mask_map_param_t**

typedef *__lv_draw_mask_saved_t* **_lv_draw_mask_saved_arr_t**[**_LV_MASK_MAX_NUM**]

Enums

enum [anonymous]
Values:

LV_DRAW_MASK_RES_TRANSP

LV_DRAW_MASK_RES_FULL_COVER

LV_DRAW_MASK_RES_CHANGED

LV_DRAW_MASK_RES_UNKNOWN

enum [anonymous]
Values:

LV_DRAW_MASK_TYPE_LINE

LV_DRAW_MASK_TYPE_ANGLE

LV_DRAW_MASK_TYPE_RADIUS

LV_DRAW_MASK_TYPE_FADE

LV_DRAW_MASK_TYPE_MAP

enum [anonymous]
Values:

LV_DRAW_MASK_LINE_SIDE_LEFT = 0

LV_DRAW_MASK_LINE_SIDE_RIGHT

LV_DRAW_MASK_LINE_SIDE_TOP

LV_DRAW_MASK_LINE_SIDE_BOTTOM

Functions

int16_t **lv_draw_mask_add**(void **param*, void **custom_id*)
Add a draw mask. Everything drawn after it (until removing the mask) will be affected by the mask.

Return the an integer, the ID of the mask. Can be used in `lv_draw_mask_remove_id`.

Parameters

- `param`: an initialized mask parameter. Only the pointer is saved.
- `custom_id`: a custom pointer to identify the mask. Used in `lv_draw_mask_remove_custom`.

void ***lv_draw_mask_remove_id**(int16_t *id*)
Remove a mask with a given ID

Return the parameter of the removed mask. If more masks have `custom_id` ID then the last mask's parameter will be returned

Parameters

- `id`: the ID of the mask. Returned by `lv_draw_mask_add`

void ***lv_draw_mask_remove_custom**(void **custom_id*)
Remove all mask with a given custom ID

Return return the parameter of the removed mask. If more masks have `custom_id` ID then the last mask's parameter will be returned

Parameters

- `custom_id`: a pointer used in `lv_draw_mask_add`

void **`lv_draw_mask_line_points_init`**(*lv_draw_mask_line_param_t *param*, lv_coord_t *p1x*, lv_coord_t *p1y*, lv_coord_t *p2x*, lv_coord_t *p2y*, *lv_draw_mask_line_side_t side*)

Initialize a line mask from two points.

Parameters

- `param`: pointer to a `lv_draw_mask_param_t` to initialize
- `p1x`: X coordinate of the first point of the line
- `p1y`: Y coordinate of the first point of the line
- `p2x`: X coordinate of the second point of the line
- `p2y`: y coordinate of the second point of the line
- `side`: and element of `lv_draw_mask_line_side_t` to describe which side to keep. With `LV_DRAW_MASK_LINE_SIDE_LEFT/RIGHT` and horizontal line all pixels are kept With `LV_DRAW_MASK_LINE_SIDE_TOP/BOTTOM` and vertical line all pixels are kept

void **`lv_draw_mask_line_angle_init`**(*lv_draw_mask_line_param_t *param*, lv_coord_t *p1x*, lv_coord_t *py*, int16_t *angle*, *lv_draw_mask_line_side_t side*)

Initialize a line mask from a point and an angle.

Parameters

- `param`: pointer to a `lv_draw_mask_param_t` to initialize
- `px`: X coordinate of a point of the line
- `py`: X coordinate of a point of the line
- `angle`: right 0 deg, bottom: 90
- `side`: and element of `lv_draw_mask_line_side_t` to describe which side to keep. With `LV_DRAW_MASK_LINE_SIDE_LEFT/RIGHT` and horizontal line all pixels are kept With `LV_DRAW_MASK_LINE_SIDE_TOP/BOTTOM` and vertical line all pixels are kept

void **`lv_draw_mask_angle_init`**(*lv_draw_mask_angle_param_t *param*, lv_coord_t *vertex_x*, lv_coord_t *vertex_y*, lv_coord_t *start_angle*, lv_coord_t *end_angle*)

Initialize an angle mask.

Parameters

- `param`: pointer to a `lv_draw_mask_param_t` to initialize
- `vertex_x`: X coordinate of the angle vertex (absolute coordinates)
- `vertex_y`: Y coordinate of the angle vertex (absolute coordinates)
- `start_angle`: start angle in degrees. 0 deg on the right, 90 deg, on the bottom
- `end_angle`: end angle

void **`lv_draw_mask_radius_init`**(*lv_draw_mask_radius_param_t *param*, **const** lv_area_t *rect*, lv_coord_t *radius*, bool *inv*)

Initialize a fade mask.

Parameters

- `param`: param pointer to a `lv_draw_mask_param_t` to initialize

- `rect`: coordinates of the rectangle to affect (absolute coordinates)
- `radius`: radius of the rectangle
- `inv`: true: keep the pixels inside the rectangle; keep the pixels outside of the rectangle

void **lv_draw_mask_fade_init**(*lv_draw_mask_fade_param_t* **param*, **const** lv_area_t **coords*, lv_opa_t *opa_top*, lv_coord_t *y_top*, lv_opa_t *opa_bottom*, lv_coord_t *y_bottom*)

Initialize a fade mask.

Parameters

- `param`: pointer to a `lv_draw_mask_param_t` to initialize
- `coords`: coordinates of the area to affect (absolute coordinates)
- `opa_top`: opacity on the top
- `y_top`: at which coordinate start to change to opacity to `opa_bottom`
- `opa_bottom`: opacity at the bottom
- `y_bottom`: at which coordinate reach `opa_bottom`.

void **lv_draw_mask_map_init**(*lv_draw_mask_map_param_t* **param*, **const** lv_area_t **coords*, **const** lv_opa_t **map*)

Initialize a map mask.

Parameters

- `param`: pointer to a `lv_draw_mask_param_t` to initialize
- `coords`: coordinates of the map (absolute coordinates)
- `map`: array of bytes with the mask values

struct lv_draw_mask_common_dsc_t

Public Members

lv_draw_mask_xcb_t **cb**

lv_draw_mask_type_t **type**

struct lv_draw_mask_line_param_t

Public Members

lv_draw_mask_common_dsc_t **dsc**

lv_point_t **p1**

lv_point_t **p2**

lv_draw_mask_line_side_t **side**

struct *lv_draw_mask_line_param_t*::[**anonymous**] **cfg**

lv_point_t **origo**

int32_t **xy_steep**

int32_t **yx_steep**

int32_t **steep**

int32_t **spx**

uint8_t **flat**

uint8_t **inv**

struct lv_draw_mask_angle_param_t

Public Members

lv_draw_mask_common_dsc_t **dsc**

lv_point_t **vertex_p**

lv_coord_t **start_angle**

lv_coord_t **end_angle**

struct *lv_draw_mask_angle_param_t*::[**anonymous**] **cfg**

lv_draw_mask_line_param_t **start_line**

lv_draw_mask_line_param_t **end_line**

uint16_t **delta_deg**

struct lv_draw_mask_radius_param_t

Public Members

lv_draw_mask_common_dsc_t **dsc**

lv_area_t **rect**

lv_coord_t **radius**

uint8_t **outer**

struct *lv_draw_mask_radius_param_t*::[**anonymous**] **cfg**

int32_t **y_prev**

lv_sqrt_res_t **y_prev_x**

struct lv_draw_mask_fade_param_t

Public Members

lv_draw_mask_common_dsc_t **dsc**

lv_area_t **coords**

lv_coord_t **y_top**

lv_coord_t **y_bottom**

lv_opa_t **opa_top**

lv_opa_t **opa_bottom**

struct *lv_draw_mask_fade_param_t*::[**anonymous**] **cfg**

struct _lv_draw_mask_map_param_t

Public Members

lv_draw_mask_common_dsc_t **dsc**

lv_area_t **coords**

const lv_opa_t ***map**

struct *_lv_draw_mask_map_param_t*::[**anonymous**] **cfg**

struct _lv_draw_mask_saved_t

Public Members

void ***param**

void ***custom_id**

5.24 Page (lv_page)

5.24.1 Overview

The Page consist of two *Containers* on each other:

- a **background**
- a top which is **scrollable**.

5.24.2 Parts and Styles

The Page's main part is called `LV_PAGE_PART_BG` which is the background of the Page. It uses all the typical background style properties. Using padding will add space on the sides.

The scrollable object can be referenced via the `LV_PAGE_PART_SCRL` part. It also uses all the typical background style properties and padding to add space on the sides.

`LV_LIST_PART_SCROLLBAR` is a virtual part of the background to draw the scroll bars. Uses all the typical background style properties, *size* to set the width of the scroll bars, and *pad_right* and *pad_bottom* to set the spacing.

`LV_LIST_PART_EDGE_FLASH`is also a virtual part of the background to draw a semicircle on the sides when the list can not be scrolled in that direction further. Uses all the typical background properties.

5.24.3 Usage

The background object can be referenced as the page itself like. E.g. to set the page's width: `lv_obj_set_width(page, 100)`.

If a child is created on the page it will be automatically moved to the scrollable container. If the scrollable container becomes larger then the background it can be scrolled by dragging (like the lists on smartphones).

By default, the scrollable's has `LV_FIT_MAX` fit in all directions. It means the scrollable size will be the same as the background's size (minus the padding) while the children are in the background. But when an object is positioned out of the background the scrollable size will be increased to involve it.

Scrollbars

Scrollbars can be shown according to four policies:

- `LV_SCRLBAR_MODE_OFF` Never show scroll bars
- `LV_SCRLBAR_MODE_ON` Always show scroll bars
- `LV_SCRLBAR_MODE_DRAG` Show scroll bars when the page is being dragged
- `LV_SCRLBAR_MODE_AUTO` Show scroll bars when the scrollable container is large enough to be scrolled
- `LV_SCRLBAR_MODE_HIDE` Hide the scroll bar temporally
- `LV_SCRLBAR_MODE_UNHIDE` Unhide the previously hidden scrollbar. Recover the original mode too

The scroll bar show policy can be changed by: `lv_page_set_scrlbar_mode(page, SB_MODE)`. The default value is `LV_SCRLBAR_MODE_AUTO`.

Glue object

A children can be "glued" to the page. In this case, if the page can be scrolled by dragging that object. It can be enabled by the `lv_page_glue_obj(child, true)`.

Focus object

An object on a page can be focused with `lv_page_focus(page, child, LV_ANIM_ON0/FF)`. It will move the scrollable container to show a child. The time of the animation can be set by `lv_page_set_anim_time(page, anim_time)` in milliseconds. `child` doesn't have to be a direct child of the page. This is it works if the scrollable object is the grandparent of the object too.

Manual navigation

You can move the scrollable object manually using `lv_page_scroll_hor(page, dist)` and `lv_page_scroll_ver(page, dist)`

Edge flash

A circle-like effect can be shown if the list reached the most top/bottom/left/right position. `lv_page_set_edge_flash(list, en)` enables this feature.

Scroll propagation

If the list is created on an other scrollable element (like an other page) and the Page can't be scrolled further the scrolling can be propagated to the parent to continue the scrolling on the parent. It can be enabled with `lv_page_set_scroll_propagation(list, true)`

5.24.4 Clean the page

All the object created on the page can be clean with `lv_page_clean(page)`. Note that `lv_obj_clean(page)` doesn't work here because it would delete the scrollable object too.

Scrollable API

There are functions to directly set/get the scrollable's attributes:

- `lv_page_get_scrl()`
- `lv_page_set_scrl_fit/fint2/fit4()`
- `lv_page_set_scrl_width()`
- `lv_page_set_scrl_height()`
- `lv_page_set_scrl_fit_width()`
- `lv_page_set_scrl_fit_height()`
- `lv_page_set_scrl_layout()`

5.24.5 Events

Only the Generic events are sent by the object type.

The scrollable object has a default event callback which propagates the following events to the background object: `LV_EVENT_PRESSED`, `LV_EVENT_PRESSING`, `LV_EVENT_PRESS_LOST`,`LV_EVENT_RELEASED`, `LV_EVENT_SHORT_CLICKED`, `LV_EVENT_CLICKED`, `LV_EVENT_LONG_PRESSED`, `LV_EVENT_LONG_PRESSED_REPEAT`

Learn more about *Events*.

##Keys

The following *Keys* are processed by the Page:

- **LV_KEY_RIGHT/LEFT/UP/DOWN** Scroll the page

Learn more about *Keys*.

5.24.6 Example

5.24.7 API

Typedefs

typedef uint8_t **lv_scrollbar_mode_t**

typedef uint8_t **lv_page_edge_t**

typedef uint8_t **lv_part_style_t**

Enums

enum [anonymous]

Scrollbar modes: shows when should the scrollbars be visible

Values:

LV_SCROLLBAR_MODE_OFF = 0x0

Never show scroll bars

LV_SCROLLBAR_MODE_ON = 0x1
Always show scroll bars

LV_SCROLLBAR_MODE_DRAG = 0x2
Show scroll bars when page is being dragged

LV_SCROLLBAR_MODE_AUTO = 0x3
Show scroll bars when the scrollable container is large enough to be scrolled

LV_SCROLLBAR_MODE_HIDE = 0x4
Hide the scroll bar temporally

LV_SCROLLBAR_MODE_UNHIDE = 0x5
Unhide the previously hidden scroll bar. Recover original mode too

enum [anonymous]
Edges: describes the four edges of the page

Values:

LV_PAGE_EDGE_LEFT = 0x1

LV_PAGE_EDGE_TOP = 0x2

LV_PAGE_EDGE_RIGHT = 0x4

LV_PAGE_EDGE_BOTTOM = 0x8

enum [anonymous]
Values:

LV_PAGE_PART_BG = *LV_CONT_PART_MAIN*

LV_PAGE_PART_SCROLLBAR = *_LV_OBJ_PART_VIRTUAL_LAST*

LV_PAGE_PART_EDGE_FLASH

_LV_PAGE_PART_VIRTUAL_LAST

LV_PAGE_PART_SCROLLABLE = *_LV_OBJ_PART_REAL_LAST*

_LV_PAGE_PART_REAL_LAST

Functions

lv_obj_t ***lv_page_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a page objects

Return pointer to the created page

Parameters

- `par`: pointer to an object, it will be the parent of the new page
- `copy`: pointer to a page object, if not NULL then the new object will be copied from it

void **lv_page_clean**(*lv_obj_t *page*)
Delete all children of the scrl object, without deleting scrl child.

Parameters

- `page`: pointer to an object

lv_obj_t ***lv_page_get_scrollable**(**const** *lv_obj_t *page*)
Get the scrollable object of a page

Return pointer to a container which is the scrollable part of the page

Parameters

- `page`: pointer to a page object

uint16_t **`lv_page_get_anim_time(const`** *lv_obj_t *page*)
Get the animation time

Return the animation time in milliseconds

Parameters

- `page`: pointer to a page object

void **`lv_page_set_scrollbar_mode(`***lv_obj_t *page*, *lv_scrollbar_mode_t sb_mode*)
Set the scroll bar mode on a page

Parameters

- `page`: pointer to a page object
- `sb_mode`: the new mode from 'lv_page_sb.mode_t' enum

void **`lv_page_set_anim_time(`***lv_obj_t *page*, uint16_t *anim_time*)
Set the animation time for the page

Parameters

- `page`: pointer to a page object
- `anim_time`: animation time in milliseconds

void **`lv_page_set_scroll_propagation(`***lv_obj_t *page*, bool *en*)
Enable the scroll propagation feature. If enabled then the page will move its parent if there is no more space to scroll. The page needs to have a page-like parent (e.g. `lv_page`, `lv_tabview` tab, `lv_win` content area etc) If enabled drag direction will be changed `LV_DRAG_DIR_ONE` automatically to allow scrolling only in one direction at one time.

Parameters

- `page`: pointer to a Page
- `en`: true or false to enable/disable scroll propagation

void **`lv_page_set_edge_flash(`***lv_obj_t *page*, bool *en*)
Enable the edge flash effect. (Show an arc when the an edge is reached)

Parameters

- `page`: pointer to a Page
- `en`: true or false to enable/disable end flash

`static` void **`lv_page_set_scrollable_fit4(`***lv_obj_t *page*, *lv_fit_t left*, *lv_fit_t right*, *lv_fit_t top*, *lv_fit_t bottom*)
Set the fit policy in all 4 directions separately. It tell how to change the page size automatically.

Parameters

- `page`: pointer to a page object
- `left`: left fit policy from `lv_fit_t`
- `right`: right fit policy from `lv_fit_t`
- `top`: bottom fit policy from `lv_fit_t`
- `bottom`: bottom fit policy from `lv_fit_t`

static void **lv_page_set_scrollable_fit2**(*lv_obj_t *page*, *lv_fit_t hor*, *lv_fit_t ver*)
Set the fit policy horizontally and vertically separately. It tell how to change the page size automatically.

Parameters

- `page`: pointer to a page object
- `hot`: horizontal fit policy from `lv_fit_t`
- `ver`: vertical fit policy from `lv_fit_t`

static void **lv_page_set_scrollable_fit**(*lv_obj_t *page*, *lv_fit_t fit*)
Set the fit policy in all 4 direction at once. It tell how to change the page size automatically.

Parameters

- `page`: pointer to a button object
- `fit`: fit policy from `lv_fit_t`

static void **lv_page_set_scrl_width**(*lv_obj_t *page*, lv_coord_t *w*)
Set width of the scrollable part of a page

Parameters

- `page`: pointer to a page object
- `w`: the new width of the scrollable (it ha no effect is horizontal fit is enabled)

static void **lv_page_set_scrl_height**(*lv_obj_t *page*, lv_coord_t *h*)
Set height of the scrollable part of a page

Parameters

- `page`: pointer to a page object
- `h`: the new height of the scrollable (it ha no effect is vertical fit is enabled)

static void **lv_page_set_scrl_layout**(*lv_obj_t *page*, *lv_layout_t layout*)
Set the layout of the scrollable part of the page

Parameters

- `page`: pointer to a page object
- `layout`: a layout from 'lv_cont_layout_t'

lv_scrollbar_mode_t **lv_page_get_scrollbar_mode**(**const** *lv_obj_t *page*)
Set the scroll bar mode on a page

Return the mode from 'lv_page_sb.mode_t' enum

Parameters

- `page`: pointer to a page object

bool **lv_page_get_scroll_propagation**(*lv_obj_t *page*)
Get the scroll propagation property

Return true or false

Parameters

- `page`: pointer to a Page

bool **lv_page_get_edge_flash**(*lv_obj_t *page*)
Get the edge flash effect property.

Parameters

- `page`: pointer to a Page return true or false

lv_coord_t **lv_page_get_width_fit(***lv_obj_t *page***)**
Get that width which can be set to the children to still not cause overflow (show scrollbars)

Return the width which still fits into the page

Parameters

- `page`: pointer to a page object

lv_coord_t **lv_page_get_height_fit(***lv_obj_t *page***)**
Get that height which can be set to the children to still not cause overflow (show scrollbars)

Return the height which still fits into the page

Parameters

- `page`: pointer to a page object

lv_coord_t **lv_page_get_width_grid(***lv_obj_t *page*, uint8_t *div*, uint8_t *span***)**
Divide the width of the object and get the width of a given number of columns. Take into account the paddings of the background and scrollable too.

Return the width according to the given parameters

Parameters

- `page`: pointer to an object
- `div`: indicates how many columns are assumed. If 1 the width will be set the the parent's width If 2 only half parent width - inner padding of the parent If 3 only third parent width - 2 * inner padding of the parent
- `span`: how many columns are combined

lv_coord_t **lv_page_get_height_grid(***lv_obj_t *page*, uint8_t *div*, uint8_t *span***)**
Divide the height of the object and get the width of a given number of columns. Take into account the paddings of the background and scrollable too.

Return the height according to the given parameters

Parameters

- `page`: pointer to an object
- `div`: indicates how many rows are assumed. If 1 the height will be set the the parent's height If 2 only half parent height - inner padding of the parent If 3 only third parent height - 2 * inner padding of the parent
- `span`: how many rows are combined

static lv_coord_t **lv_page_get_scrl_width(const** *lv_obj_t *page***)**
Get width of the scrollable part of a page

Return the width of the scrollable

Parameters

- `page`: pointer to a page object

static lv_coord_t **lv_page_get_scrl_height(const** *lv_obj_t *page***)**
Get height of the scrollable part of a page

Return the height of the scrollable

Parameters

- `page`: pointer to a page object

static *lv_layout_t* **lv_page_get_scrl_layout(const** *lv_obj_t *page*)
Get the layout of the scrollable part of a page

Return the layout from 'lv_cont_layout_t'

Parameters

- `page`: pointer to page object

static *lv_fit_t* **lv_page_get_scrl_fit_left(const** *lv_obj_t *page*)
Get the left fit mode

Return an element of `lv_fit_t`

Parameters

- `page`: pointer to a page object

static *lv_fit_t* **lv_page_get_scrl_fit_right(const** *lv_obj_t *page*)
Get the right fit mode

Return an element of `lv_fit_t`

Parameters

- `page`: pointer to a page object

static *lv_fit_t* **lv_page_get_scrl_fit_top(const** *lv_obj_t *page*)
Get the top fit mode

Return an element of `lv_fit_t`

Parameters

- `page`: pointer to a page object

static *lv_fit_t* **lv_page_get_scrl_fit_bottom(const** *lv_obj_t *page*)
Get the bottom fit mode

Return an element of `lv_fit_t`

Parameters

- `page`: pointer to a page object

bool **lv_page_on_edge(***lv_obj_t *page*, *lv_page_edge_t edge*)
Find whether the page has been scrolled to a certain edge.

Return true if the page is on the specified edge

Parameters

- `page`: Page object
- `edge`: Edge to check

void **lv_page_glue_obj(***lv_obj_t *obj*, bool *glue*)
Glue the object to the page. After it the page can be moved (dragged) with this object too.

Parameters

- `obj`: pointer to an object on a page
- `glue`: true: enable glue, false: disable glue

void **lv_page_focus(***lv_obj_t *page*, **const** *lv_obj_t *obj*, *lv_anim_enable_t anim_en*)
Focus on an object. It ensures that the object will be visible on the page.

Parameters

- `page`: pointer to a page object
- `obj`: pointer to an object to focus (must be on the page)
- `anim_en`: LV_ANIM_ON to focus with animation; LV_ANIM_OFF to focus without animation

void **`lv_page_scroll_hor`**(*lv_obj_t *page*, lv_coord_t *dist*)
Scroll the page horizontally

Parameters

- `page`: pointer to a page object
- `dist`: the distance to scroll (< 0: scroll left; > 0 scroll right)

void **`lv_page_scroll_ver`**(*lv_obj_t *page*, lv_coord_t *dist*)
Scroll the page vertically

Parameters

- `page`: pointer to a page object
- `dist`: the distance to scroll (< 0: scroll down; > 0 scroll up)

void **`lv_page_start_edge_flash`**(*lv_obj_t *page*, *lv_page_edge_t edge*)
Not intended to use directly by the user but by other object types internally. Start an edge flash animation.

Parameters

- `page`:
- `edge`: the edge to flash. Can be `LV_PAGE_EDGE_LEFT/RIGHT/TOP/BOTTOM`

`struct lv_page_ext_t`

Public Members

lv_cont_ext_t **`bg`**

lv_obj_t ***`scrl`**

lv_style_list_t **`style`**

lv_area_t **`hor_area`**

lv_area_t **`ver_area`**

uint8_t **`hor_draw`**

uint8_t **`ver_draw`**

lv_scrollbar_mode_t **`mode`**

`struct` *lv_page_ext_t*::[**anonymous**] **`scrlbar`**

lv_anim_value_t **`state`**

uint8_t **`enabled`**

uint8_t **`top_ip`**

uint8_t **`bottom_ip`**

uint8_t **`right_ip`**

uint8_t **left_ip**

struct *lv_page_ext_t*::[**anonymous**] **edge_flash**

uint16_t **anim_time**

lv_obj_t ***scroll_prop_obj**

uint8_t **scroll_prop**

5.25 Roller (lv_roller)

5.25.1 Overview

Roller allows you to simply select one option from more with scrolling.

5.25.2 Parts and Styles

The Roller's main part is called `LV_ROLLER_PART_BG`. It's a rectangle and uses all the typical background properties. The style of the Roller's label is inherited from the *text* style properties of the background. To adjust the space between the options use the *text_line_space* style property. The *padding* style properties set the space on the sides.

The selected option in the middle can be referenced with `LV_ROLLER_PART_SELECTED` virtual part. It also uses all the typical background properties and *text_color* property to change the color of the options in the selected area.

5.25.3 Usage

Set options

The options are passed to the Roller as a string with `lv_roller_set_options(roller, options, LV_ROLLER_MODE_NORMAL/INFINITE)`. The options should be separated by `\n`. For example: `"First\nSecond\nThird"`.

`LV_ROLLER_MODE_INIFINITE` make the roller circular.

You can select an option manually with `lv_roller_set_selected(roller, id, LV_ANIM_ON/OFF)`, where *id* is the index of an option.

Get selected option

The get the currently selected option use `lv_roller_get_selected(roller)` it will return the *index* of the selected option.

`lv_roller_get_selected_str(roller, buf, buf_size)` copy the name of the selected option to `buf`.

Align the options

To align the label horizontally use `lv_roller_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`.

Visible rows

The number of visible rows can be adjusted with `lv_roller_set_visible_row_count(roller, num)`

Animation time

When the Roller is scrolled and doesn't stop exactly on an option it will scroll to the nearest valid option automatically. The time of this scroll animation can be changed by `lv_roller_set_anim_time(roller, anim_time)`. Zero animation time means no animation.

5.25.4 Events

Besides, the Generic events the following Special events are sent by the Drop down lists:

- **LV_EVENT_VALUE_CHANGED** sent when a new option is selected

Learn more about *Events*.

5.25.5 Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_RIGHT/DOWN** Select the next option
- **LV_KEY_LEFT/UP** Select the previous option
- **LY_KEY_ENTER** Apply the selected option (Send `LV_EVENT_VALUE_CHANGED` event)

5.25.6 Example

5.25.7 API

Typedefs

typedef uint8_t **lv_roller_mode_t**

typedef uint8_t **lv_roller_part_t**

Enums

enum [**anonymous**]

Roller mode.

Values:

LV_ROLLER_MODE_NORMAL

Normal mode (roller ends at the end of the options).

LV_ROLLER_MODE_INIFINITE

Infinite mode (roller can be scrolled forever).

enum [**anonymous**]

Values:

LV_ROLLER_PART_BG = *LV_PAGE_PART_BG*

LV_ROLLER_PART_SELECTED = *_LV_PAGE_PART_VIRTUAL_LAST*

_LV_ROLLER_PART_VIRTUAL_LAST

Functions

lv_obj_t ***lv_roller_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a roller object

Return pointer to the created roller

Parameters

- `par`: pointer to an object, it will be the parent of the new roller
- `copy`: pointer to a roller object, if not NULL then the new object will be copied from it

void **lv_roller_set_options**(*lv_obj_t *roller*, **const** char **options*, *lv_roller_mode_t mode*)

Set the options on a roller

Parameters

- `roller`: pointer to roller object
- `options`: a string with ' ' separated options. E.g. "One\nTwo\nThree"
- `mode`: `LV_ROLLER_MODE_NORMAL` or `LV_ROLLER_MODE_INFINITE`

void **lv_roller_set_align**(*lv_obj_t *roller*, *lv_label_align_t align*)

Set the align of the roller's options (left, right or center[default])

Parameters

- `roller`: - pointer to a roller object
- `align`: - one of lv_label_align_t values (left, right, center)

void **lv_roller_set_selected**(*lv_obj_t *roller*, uint16_t *sel_opt*, *lv_anim_enable_t anim*)

Set the selected option

Parameters

- `roller`: pointer to a roller object
- `sel_opt`: id of the selected option (0 ... number of option - 1);
- `anim`: LV_ANOM_ON: set with animation; LV_ANIM_OFF set immediately

void **lv_roller_set_visible_row_count**(*lv_obj_t *roller*, uint8_t *row_cnt*)

Set the height to show the given number of rows (options)

Parameters

- `roller`: pointer to a roller object
- `row_cnt`: number of desired visible rows

void **lv_roller_set_auto_fit**(*lv_obj_t *roller*, bool *auto_fit*)

Allow automatically setting the width of roller according to it's content.

Parameters

- `roller`: pointer to a roller object
- `auto_fit`: true: enable auto fit

static void **lv_roller_set_anim_time(***lv_obj_t *roller*, uint16_t *anim_time***)**
Set the open/close animation time.

Parameters

- `roller`: pointer to a roller object
- `anim_time`: open/close animation time [ms]

uint16_t **lv_roller_get_selected(const** *lv_obj_t *roller***)**
Get the id of the selected option

Return id of the selected option (0 ... number of option - 1);

Parameters

- `roller`: pointer to a roller object

uint16_t **lv_roller_get_option_cnt(const** *lv_obj_t *roller***)**
Get the total number of options

Return the total number of options in the list

Parameters

- `roller`: pointer to a roller object

void **lv_roller_get_selected_str(const** *lv_obj_t *roller*, char **buf*, uint32_t *buf_size***)**
Get the current selected option as a string

Parameters

- `roller`: pointer to roller object
- `buf`: pointer to an array to store the string
- `buf_size`: size of `buf` in bytes. 0: to ignore it.

lv_label_align_t **lv_roller_get_align(const** *lv_obj_t *roller***)**
Get the align attribute. Default alignment after _create is LV_LABEL_ALIGN_CENTER

Return LV_LABEL_ALIGN_LEFT, LV_LABEL_ALIGN_RIGHT or LV_LABEL_ALIGN_CENTER

Parameters

- `roller`: pointer to a roller object

bool **lv_roller_get_auto_fit(***lv_obj_t *roller***)**
Get whether the auto fit option is enabled or not.

Return true: auto fit is enabled

Parameters

- `roller`: pointer to a roller object

const char ***lv_roller_get_options(const** *lv_obj_t *roller***)**
Get the options of a roller

Return the options separated by ' '-s (E.g. "Option1\nOption2\nOption3")

Parameters

- `roller`: pointer to roller object

static uint16_t **lv_roller_get_anim_time(const** *lv_obj_t *roller***)**
Get the open/close animation time.

Return open/close animation time [ms]

Parameters

- `roller`: pointer to a roller

struct lv_roller_ext_t

Public Members

lv_page_ext_t **page**

lv_style_list_t **style_sel**

uint16_t **option_cnt**

uint16_t **sel_opt_id**

uint16_t **sel_opt_id_ori**

lv_roller_mode_t **mode**

uint8_t **auto_fit**

5.26 Slider (lv_slider)

5.26.1 Overview

The Slider object looks like a *Bar* supplemented with a knob. The knob can be dragged to set a value. The Slider also can be vertical or horizontal.

5.26.2 Parts and Styles

The Slider's main part is called `LV_SLIDER_PART_BG` and it uses the typical background style properties.

`LV_SLIDER_PART_INDIC` is a virtual part which also uses all the typical background properties. By default, the indicator maximal size is the same as the background's size but setting positive padding values in `LV_SLIDER_PART_BG` will make the indicator smaller. (negative values will make it larger) If the *value* style property is used on the indicator the alignment will be calculated based on the current size of the indicator. For example a center aligned value is always shown in the middle of the indicator regardless it's current size.

`LV_SLIDER_PART_KNOB` is a virtual part using all the typical background properties to describe the knob(s). Similarly to the *indicator* the *value* text is also aligned to the current position and size of the knob. By default the knob is square (with a radius) with side length equal to the smaller side of the slider. The knob can be made larger with the *padding* values. Padding values can be asymmetric too.

5.26.3 Usage

Value and range

To set an initial value use `lv_slider_set_value(slider, new_value, LV_ANIM_ON/OFF)`. `lv_slider_set_anim_time(slider, anim_time)` sets the animation time in milliseconds.

To specify the range (min, max values) the `lv_slider_set_range(slider, min , max)` can be used.

Symmetrical and Range

Besides the normal type the Slider can be configured in two additional types:

- `LV_SLIDER_TYPE_NORMAL` normal type
- `LV_SLIDER_TYPE_SYMMETRICAL` draw the indicator symmetrical to zero (drawn from zero, left to right)
- `LV_SLIDER_TYPE_RANGE` allow the use of an additional knob for the left (start) value. (Can be used with `lv_slider_set/get_left_value()`)

The type can be changed with `lv_slider_set_type(slider, LV_SLIDER_TYPE_...)`

5.26.4 Events

Besides the Generic events the following Special events are sent by the Slider:

- **LV_EVENT_VALUE_CHANGED** Sent while the slider is being dragged or changed with keys. The event is sent continuously while the slider is dragged and only when it is released. Use `lv_slider_is_dragged` to decide whether is slider is being dragged or just released.

5.26.5 Keys

- **LV_KEY_UP**, **LV_KEY_RIGHT** Increment the slider's value by 1
- **LV_KEY_DOWN**, **LV_KEY_LEFT** Decrement the slider's value by 1

Learn more about *Keys*.

5.26.6 Example

5.26.7 API

Typedefs

typedef uint8_t **lv_slider_type_t**

Enums

enum [anonymous]

Values:

LV_SLIDER_TYPE_NORMAL

LV_SLIDER_TYPE_SYMMETRICAL

LV_SLIDER_TYPE_RANGE

enum [anonymous]

Built-in styles of slider

Values:

LV_SLIDER_PART_BG

LV_SLIDER_PART_INDIC
Slider background style.

LV_SLIDER_PART_KNOB
Slider indicator (filled area) style.

Functions

lv_obj_t ***lv_slider_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a slider objects

Return pointer to the created slider

Parameters

- `par`: pointer to an object, it will be the parent of the new slider
- `copy`: pointer to a slider object, if not NULL then the new object will be copied from it

static void **lv_slider_set_value**(*lv_obj_t *slider*, int16_t *value*, *lv_anim_enable_t anim*)
Set a new value on the slider

Parameters

- `slider`: pointer to a slider object
- `value`: new value
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

static void **lv_slider_set_left_value**(*lv_obj_t *slider*, int16_t *left_value*, *lv_anim_enable_t anim*)
Set a new value for the left knob of a slider

Parameters

- `slider`: pointer to a slider object
- `left_value`: new value
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

static void **lv_slider_set_range**(*lv_obj_t *slider*, int16_t *min*, int16_t *max*)
Set minimum and the maximum values of a bar

Parameters

- `slider`: pointer to the slider object
- `min`: minimum value
- `max`: maximum value

static void **lv_slider_set_anim_time**(*lv_obj_t *slider*, uint16_t *anim_time*)
Set the animation time of the slider

Parameters

- `slider`: pointer to a bar object
- `anim_time`: the animation time in milliseconds.

static void **lv_slider_set_type**(*lv_obj_t *slider*, *lv_slider_type_t type*)
Make the slider symmetric to zero. The indicator will grow from zero instead of the minimum position.

Parameters

- `slider`: pointer to a slider object
- `en`: true: enable disable symmetric behavior; false: disable

int16_t **lv_slider_get_value**(**const** *lv_obj_t *slider*)
Get the value of the main knob of a slider

Return the value of the main knob of the slider

Parameters

- `slider`: pointer to a slider object

static int16_t **lv_slider_get_left_value**(**const** *lv_obj_t *slider*)
Get the value of the left knob of a slider

Return the value of the left knob of the slider

Parameters

- `slider`: pointer to a slider object

static int16_t **lv_slider_get_min_value**(**const** *lv_obj_t *slider*)
Get the minimum value of a slider

Return the minimum value of the slider

Parameters

- `slider`: pointer to a slider object

static int16_t **lv_slider_get_max_value**(**const** *lv_obj_t *slider*)
Get the maximum value of a slider

Return the maximum value of the slider

Parameters

- `slider`: pointer to a slider object

bool **lv_slider_is_dragged**(**const** *lv_obj_t *slider*)
Give the slider is being dragged or not

Return true: drag in progress false: not dragged

Parameters

- `slider`: pointer to a slider object

static uint16_t **lv_slider_get_anim_time**(*lv_obj_t *slider*)
Get the animation time of the slider

Return the animation time in milliseconds.

Parameters

- `slider`: pointer to a slider object

static *lv_slider_type_t* **lv_slider_get_type**(*lv_obj_t *slider*)
Get whether the slider is symmetric or not.

Return true: symmetric is enabled; false: disable

Parameters

- `slider`: pointer to a bar object

struct lv_slider_ext_t

Public Members

lv_bar_ext_t **bar**

lv_style_list_t **style_knob**

lv_area_t **left_knob_area**

lv_area_t **right_knob_area**

int16_t ***value_to_set**

uint8_t **dragging**

5.27 Spinbox (lv_spinbox)

5.27.1 Overview

The Spinbox contains a number as text which can be increased or decreased by *Keys* or API functions. Under the hood the Spinbox is a modified *Text area*.

5.27.2 Parts and Styles

The Spinbox's main part is called `LV_SPINBOX_PART_BG` which is a rectangle-like background using all the typical background style properties. It also describes the style of the label with its *text* style properties.

`LV_SPINBOX_PART_CURSOR` is a virtual part describing the cursor. Read the *Text area* documentation for a detailed description.

Set format

`lv_spinbox_set_digit_format(spinbox, digit_count, separator_position)` set the format of the number. `digit_count` sets the number of digits. Leading zeros are added to fill the space on the left. `separator_position` sets the number of digit before the decimal point. `0` means no decimal point.

`lv_spinbox_set_padding_left(spinbox, cnt)` add `cnt` "space" characters between the sign an the most left digit.

Value and ranges

`lv_spinbox_set_range(spinbox, min, max)` sets the range of the Spinbox.

`lv_spinbox_set_value(spinbox, num)` sets the Spinbox's value manually.

`lv_spinbox_increment(spinbox)` and `lv_spinbox_decrement(spinbox)` increments/decrements the value of the Spinbox.

`lv_spinbox_set_step(spinbox, step)` sets the amount to increment decrement.

5.27.3 Events

Besides the Generic events the following Special events are sent by the Drop down lists:

- **LV_EVENT_VALUE_CHANGED** sent when the value has changed. (the value is set as event data as `int32_t`)
- **LV_EVENT_INSERT** sent by the ancestor Text area but shouldn't be used.

Learn more about *Events*.

5.27.4 Keys

The following *Keys* are processed by the Buttons:

- **LV_KEY_LEFT/RIGHT** With *Keypad* move the cursor left/right. With *Encoder* decrement/increment the selected digit.
- **LY_KEY_ENTER** Apply the selected option (Send `LV_EVENT_VALUE_CHANGED` event and close the Drop down list)
- **LV_KEY_ENTER** With *Encoder* got the net digit. Jump to the first after the last.

5.27.5 Example

5.27.6 API

Typedefs

typedef uint8_t **lv_spinbox_part_t**

Enums

enum [anonymous]

Values:

LV_SPINBOX_PART_BG = *LV_TEXTAREA_PART_BG*

LV_SPINBOX_PART_CURSOR = *LV_TEXTAREA_PART_CURSOR*

_LV_SPINBOX_PART_VIRTUAL_LAST = *_LV_TEXTAREA_PART_VIRTUAL_LAST*

_LV_SPINBOX_PART_REAL_LAST = *_LV_TEXTAREA_PART_REAL_LAST*

Functions

lv_obj_t ***lv_spinbox_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a spinbox objects

Return pointer to the created spinbox

Parameters

- `par`: pointer to an object, it will be the parent of the new spinbox
- `copy`: pointer to a spinbox object, if not NULL then the new object will be copied from it

void **lv_spinbox_set_rollover**(*lv_obj_t *spinbox*, bool *b*)
Set spinbox rollover function

Parameters

- `spinbox`: pointer to spinbox
- `b`: true or false to enable or disable (default)

void **lv_spinbox_set_value**(*lv_obj_t *spinbox*, int32_t *i*)
Set spinbox value

Parameters

- `spinbox`: pointer to spinbox
- `i`: value to be set

void **lv_spinbox_set_digit_format**(*lv_obj_t *spinbox*, uint8_t *digit_count*, uint8_t *separator_position*)
Set spinbox digit format (digit count and decimal format)

Parameters

- `spinbox`: pointer to spinbox
- `digit_count`: number of digit excluding the decimal separator and the sign
- `separator_position`: number of digit before the decimal point. If 0, decimal point is not shown

void **lv_spinbox_set_step**(*lv_obj_t *spinbox*, uint32_t *step*)
Set spinbox step

Parameters

- `spinbox`: pointer to spinbox
- `step`: steps on increment/decrement

void **lv_spinbox_set_range**(*lv_obj_t *spinbox*, int32_t *range_min*, int32_t *range_max*)
Set spinbox value range

Parameters

- `spinbox`: pointer to spinbox
- `range_min`: maximum value, inclusive
- `range_max`: minimum value, inclusive

void **lv_spinbox_set_padding_left**(*lv_obj_t *spinbox*, uint8_t *padding*)
Set spinbox left padding in digits count (added between sign and first digit)

Parameters

- `spinbox`: pointer to spinbox
- `cb`: Callback function called on value change event

bool **lv_spinbox_get_rollover**(*lv_obj_t *spinbox*)
Get spinbox rollover function status

Parameters

- `spinbox`: pointer to spinbox

int32_t **lv_spinbox_get_value**(*lv_obj_t *spinbox*)
Get the spinbox numeral value (user has to convert to float according to its digit format)

Return value integer value of the spinbox

Parameters

- `spinbox`: pointer to spinbox

void **lv_spinbox_step_next**(*lv_obj_t *spinbox*)
Select next lower digit for edition by dividing the step by 10

Parameters

- `spinbox`: pointer to spinbox

void **lv_spinbox_step_prev**(*lv_obj_t *spinbox*)
Select next higher digit for edition by multiplying the step by 10

Parameters

- `spinbox`: pointer to spinbox

void **lv_spinbox_increment**(*lv_obj_t *spinbox*)
Increment spinbox value by one step

Parameters

- `spinbox`: pointer to spinbox

void **lv_spinbox_decrement**(*lv_obj_t *spinbox*)
Decrement spinbox value by one step

Parameters

- `spinbox`: pointer to spinbox

struct lv_spinbox_ext_t

Public Members

lv_textarea_ext_t **ta**

int32_t **value**

int32_t **range_max**

int32_t **range_min**

int32_t **step**

uint8_t **rollover**

uint16_t **digit_count**

uint16_t **dec_point_pos**

uint16_t **digit_padding_left**

5.27.7 Example

5.28 Spinner (lv_spinner)

5.28.1 Overview

The Spinner object is a spinning arc over a border.

5.28.2 Parts and Styles

The Spinner uses the the following parts:

- `LV_SPINNER_PART_BG`: main part
- `LV_SPINNER_PART_INDIC`: the spinning arc (virtual part)

The parts and style works the same as in case of *Arc*. Read its documentation for a details description.

5.28.3 Usage

Arc length

The length of the arc can be adjusted by `lv_spinner_set_arc_length(spinner, deg)`.

Spinning speed

The speed of the spinning can be adjusted by `lv_spinner_set_spin_time(preload, time_ms)`.

Spin types

You can choose from more spin types:

- **LV_SPINNER_TYPE_SPINNING_ARC** spin the arc, slow down on the top
- **LV_SPINNER_TYPE_FILLSPIN_ARC** spin the arc, slow down on the top but also stretch the arc
- **LV_SPINNER_TYPE_CONSTANT_ARC** spin the arc at a constant speed

To apply one if them use `lv_spinner_set_type(preload, LV_SPINNER_TYPE_...)`

Spin direction

The direction of spinning can be changed with `lv_spinner_set_dir(preload, LV_SPINNER_DIR_FORWARD/BACKWARD)`.

5.28.4 Events

Only the Generic events are sent by the object type.

5.28.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.28.6 Example

MicroPython

No examples yet.

5.28.7 API

Typedefs

typedef uint8_t **lv_spinner_type_t**

typedef uint8_t **lv_spinner_dir_t**

typedef uint8_t **lv_spinner_style_t**

Enums

enum [anonymous]

Type of spinner.

Values:

LV_SPINNER_TYPE_SPINNING_ARC

LV_SPINNER_TYPE_FILLSPIN_ARC

LV_SPINNER_TYPE_CONSTANT_ARC

enum [anonymous]

Direction the spinner should spin.

Values:

LV_SPINNER_DIR_FORWARD

LV_SPINNER_DIR_BACKWARD

enum [anonymous]

Values:

LV_SPINNER_PART_BG = *LV_ARC_PART_BG*

LV_SPINNER_PART_INDIC = *LV_ARC_PART_INDIC*

_LV_SPINNER_PART_VIRTUAL_LAST

_LV_SPINNER_PART_REAL_LAST = *_LV_ARC_PART_REAL_LAST*

Functions

lv_obj_t ***lv_spinner_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a spinner object

Return pointer to the created spinner

Parameters

- `par`: pointer to an object, it will be the parent of the new spinner
- `copy`: pointer to a spinner object, if not NULL then the new object will be copied from it

void **lv_spinner_set_arc_length**(*lv_obj_t *spinner*, *lv_anim_value_t deg*)
Set the length of the spinning arc in degrees

Parameters

- `spinner`: pointer to a spinner object
- `deg`: length of the arc

void **lv_spinner_set_spin_time**(*lv_obj_t *spinner*, uint16_t *time*)
Set the spin time of the arc

Parameters

- `spinner`: pointer to a spinner object
- `time`: time of one round in milliseconds

void **lv_spinner_set_type**(*lv_obj_t *spinner*, *lv_spinner_type_t type*)
Set the animation type of a spinner.

Parameters

- `spinner`: pointer to spinner object
- `type`: animation type of the spinner

void **lv_spinner_set_dir**(*lv_obj_t *spinner*, *lv_spinner_dir_t dir*)
Set the animation direction of a spinner

Parameters

- `spinner`: pointer to spinner object
- `direction`: animation direction of the spinner

lv_anim_value_t **lv_spinner_get_arc_length**(**const** *lv_obj_t *spinner*)
Get the arc length [degree] of the a spinner

Parameters

- `spinner`: pointer to a spinner object

uint16_t **lv_spinner_get_spin_time**(**const** *lv_obj_t *spinner*)
Get the spin time of the arc

Parameters

- `spinner`: pointer to a spinner object [milliseconds]

lv_spinner_type_t **lv_spinner_get_type**(*lv_obj_t *spinner*)
Get the animation type of a spinner.

Return animation type

Parameters

- `spinner`: pointer to spinner object

lv_spinner_dir_t **lv_spinner_get_dir**(*lv_obj_t* **spinner*)
Get the animation direction of a spinner

Return animation direction

Parameters

- `spinner`: pointer to spinner object

void **lv_spinner_anim_cb**(void **ptr*, *lv_anim_value_t val*)
Animator function (exec_cb) to rotate the arc of spinner.

Parameters

- `ptr`: pointer to spinner
- `val`: the current desired value [0..360]

struct lv_spinner_ext_t

Public Members

lv_arc_ext_t **arc**

lv_anim_value_t **arc_length**

uint16_t **time**

lv_spinner_type_t **anim_type**

lv_spinner_dir_t **anim_dir**

5.29 Switch (lv_switch)

5.29.1 Overview

The Switch can be used to turn on/off something. It looks like a little slider.

5.29.2 Parts and Styles

The Switch uses the the following parts:

- `LV_SWITCH_PART_BG`: main part
- `LV_SWITCH_PART_INDIC`: the indicator (virtual part)
- `LV_SWITCH_PART_KNOB`: the knob (virtual part)

The parts and style works the same as in case of *Slider*. Read its documentation for a details description.

##Usage

Change state

The state of the Switch can be changed by clicking on it or by `lv_switch_on(switch, LV_ANIM_ON/OFF)`, `lv_switch_off(switch, LV_ANIM_ON/OFF)` or `lv_switch_toggle(switch, LV_ANOM_ON/OFF)` functions

Animation time

The time of animations, when the switch changes state, can be adjusted with `lv_switch_set_anim_time(switch, anim_time)`.

5.29.3 Events

Besides the Generic events the following Special events are sent by the Switch:

- **LV_EVENT_VALUE_CHANGED** Sent when the switch changes state.

5.29.4 Keys

- **LV_KEY_UP**, **LV_KEY_RIGHT** Turn on the slider
- **LV_KEY_DOWN**, **LV_KEY_LEFT** Turn off the slider

Learn more about *Keys*.

5.29.5 Example

5.29.6 API

Typedefs

typedef uint8_t **lv_switch_part_t**

Enums

enum [**anonymous**]

Switch parts.

Values:

LV_SWITCH_PART_BG = *LV_BAR_PART_BG*

Switch background.

LV_SWITCH_PART_INDIC = *LV_BAR_PART_INDIC*

Switch fill area.

LV_SWITCH_PART_KNOB = *_LV_BAR_PART_VIRTUAL_LAST*

Switch knob.

_LV_SWITCH_PART_VIRTUAL_LAST

Functions

lv_obj_t ***lv_switch_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a switch objects

Return pointer to the created switch

Parameters

- `par`: pointer to an object, it will be the parent of the new switch
- `copy`: pointer to a switch object, if not NULL then the new object will be copied from it

void **lv_switch_on**(*lv_obj_t *sw*, *lv_anim_enable_t anim*)
Turn ON the switch

Parameters

- `sw`: pointer to a switch object
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

void **lv_switch_off**(*lv_obj_t *sw*, *lv_anim_enable_t anim*)
Turn OFF the switch

Parameters

- `sw`: pointer to a switch object
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

bool **lv_switch_toggle**(*lv_obj_t *sw*, *lv_anim_enable_t anim*)
Toggle the position of the switch

Return resulting state of the switch.

Parameters

- `sw`: pointer to a switch object
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

static void **lv_switch_set_anim_time**(*lv_obj_t *sw*, uint16_t *anim_time*)
Set the animation time of the switch

Return style pointer to a style

Parameters

- `sw`: pointer to a switch object
- `anim_time`: animation time

static bool **lv_switch_get_state**(**const** *lv_obj_t *sw*)
Get the state of a switch

Return false: OFF; true: ON

Parameters

- `sw`: pointer to a switch object

static uint16_t **lv_switch_get_anim_time**(**const** *lv_obj_t *sw*)
Get the animation time of the switch

Return style pointer to a style

Parameters

- `sw`: pointer to a switch object

struct lv_switch_ext_t

Public Members

lv_bar_ext_t **bar**

lv_style_list_t **style_knob**

uint8_t **state**

5.30 Table (lv_table)

5.30.1 Overview

Tables, as usual, are built from rows, columns, and cells containing texts.

The Table object is very light weighted because only the texts are stored. No real objects are created for cells but they are just drawn on the fly.

5.30.2 Parts and Styles

The main part of the Table is called `LV_TABLE_PART_BG`. It's a rectangle like background and uses all the typical background style properties.

For the cells there are 4 virtual parts. Every cell has type (1, 2, 3 or 4) which tells which part's styles to apply on them. The cell parts are:

- `LV_TABLE_PART_CELL1`
- `LV_TABLE_PART_CELL2`
- `LV_TABLE_PART_CELL3`
- `LV_TABLE_PART_CELL4`

The cells also use all the typical background style properties. If there is a line break (`\n`) in a cell's content then a horizontal division line will drawn after the line break using the *line* style properties.

The style of texts in the cells are inherited from the cell parts or the background part.

5.30.3 Usage

Rows and Columns

To set number of rows and columns use `lv_table_set_row_cnt(table, row_cnt)` and `lv_table_set_col_cnt(table, col_cnt)`

Width and Height

The width of the columns can be set with `lv_table_set_col_width(table, col_id, width)`. The overall width of the Table object will be set to the sum of columns widths.

The height is calculated automatically from the cell styles (font, padding etc) and the number of rows.

Set cell value

The cells can store only texts so numbers needs to be converted to text before displaying them in a table.

`lv_table_set_cell_value(table, row, col, "Content")`. The text is saved by the table so it can be even a local variable.

Line break can be used in the text like `"Value\n60.3"`.

Align

The text alignment in cells can be adjusted individually with `lv_table_set_cell_align(table, row, col, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`.

Cell type

You can use 4 different cell types. Each has its own style.

Cell types can be used to add different style for example to:

- table header
- first column
- highlight a cell
- etc

The type can be selected with `lv_table_set_cell_type(table, row, col, type)` `type` can be 1, 2, 3 or 4.

Merge cells

Cells can be merged horizontally with `lv_table_set_cell_merge_right(table, col, row, true)`. To merge more adjacent cells apply this function for each cell.

Crop text

By default, the texts are word-wrapped to fit into the width of the cell and the height of the cell is set automatically. To disable this and keep the text as it is enable `lv_table_set_cell_crop(table, row, col, true)`.

Scroll

The make the Table scrollable place it on a *Page*

5.30.4 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.30.5 Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

5.30.6 Example

MicroPython

No examples yet.

5.30.7 API

Enums

enum [anonymous]

Values:

LV_TABLE_PART_BG

LV_TABLE_PART_CELL1

LV_TABLE_PART_CELL2

LV_TABLE_PART_CELL3

LV_TABLE_PART_CELL4

Functions

lv_obj_t ***lv_table_create**(*lv_obj_t* **par*, **const** *lv_obj_t* **copy*)

Create a table object

Return pointer to the created table

Parameters

- `par`: pointer to an object, it will be the parent of the new table
- `copy`: pointer to a table object, if not NULL then the new object will be copied from it

void **lv_table_set_cell_value**(*lv_obj_t* **table*, uint16_t *row*, uint16_t *col*, **const** char **txt*)

Set the value of a cell.

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]

- `txt`: text to display in the cell. It will be copied and saved so this variable is not required after this function call.

void **lv_table_set_row_cnt**(*lv_obj_t *table*, uint16_t *row_cnt*)
Set the number of rows

Parameters

- `table`: table pointer to a Table object
- `row_cnt`: number of rows

void **lv_table_set_col_cnt**(*lv_obj_t *table*, uint16_t *col_cnt*)
Set the number of columns

Parameters

- `table`: table pointer to a Table object
- `col_cnt`: number of columns. Must be < LV_TABLE_COL_MAX

void **lv_table_set_col_width**(*lv_obj_t *table*, uint16_t *col_id*, lv_coord_t *w*)
Set the width of a column

Parameters

- `table`: table pointer to a Table object
- `col_id`: id of the column [0 .. LV_TABLE_COL_MAX -1]
- `w`: width of the column

void **lv_table_set_cell_align**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*, *lv_label_align_t align*)
Set the text align in a cell

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]
- `align`: LV_LABEL_ALIGN_LEFT or LV_LABEL_ALIGN_CENTER or LV_LABEL_ALIGN_RIGHT

void **lv_table_set_cell_type**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*, uint8_t *type*)
Set the type of a cell.

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]
- `type`: 1,2,3 or 4. The cell style will be chosen accordingly.

void **lv_table_set_cell_crop**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*, bool *crop*)
Set the cell crop. (Don't adjust the height of the cell according to its content)

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]

- `col`: id of the column [0 .. col_cnt -1]
- `crop`: true: crop the cell content; false: set the cell height to the content.

void **`lv_table_set_cell_merge_right`**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*, bool *en*)
Merge a cell with the right neighbor. The value of the cell to the right won't be displayed.

Parameters

- `table`: table pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]
- `en`: true: merge right; false: don't merge right

`const` char ***`lv_table_get_cell_value`**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*)
Get the value of a cell.

Return text in the cell

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]

uint16_t **`lv_table_get_row_cnt`**(*lv_obj_t *table*)
Get the number of rows.

Return number of rows.

Parameters

- `table`: table pointer to a Table object

uint16_t **`lv_table_get_col_cnt`**(*lv_obj_t *table*)
Get the number of columns.

Return number of columns.

Parameters

- `table`: table pointer to a Table object

lv_coord_t **`lv_table_get_col_width`**(*lv_obj_t *table*, uint16_t *col_id*)
Get the width of a column

Return width of the column

Parameters

- `table`: table pointer to a Table object
- `col_id`: id of the column [0 .. LV_TABLE_COL_MAX -1]

lv_label_align_t **`lv_table_get_cell_align`**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*)
Get the text align of a cell

Return LV_LABEL_ALIGN_LEFT (default in case of error) or LV_LABEL_ALIGN_CENTER or LV_LABEL_ALIGN_RIGHT

Parameters

- `table`: pointer to a Table object

- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]

lv_label_align_t **`lv_table_get_cell_type`**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*)
Get the type of a cell

Return 1,2,3 or 4

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]

lv_label_align_t **`lv_table_get_cell_crop`**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*)
Get the crop property of a cell

Return true: text crop enabled; false: disabled

Parameters

- `table`: pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]

bool **`lv_table_get_cell_merge_right`**(*lv_obj_t *table*, uint16_t *row*, uint16_t *col*)
Get the cell merge attribute.

Return true: merge right; false: don't merge right

Parameters

- `table`: table pointer to a Table object
- `row`: id of the row [0 .. row_cnt -1]
- `col`: id of the column [0 .. col_cnt -1]

lv_res_t **`lv_table_get_pressed_cell`**(*lv_obj_t *table*, uint16_t **row*, uint16_t **col*)
Get the last pressed or being pressed cell

Return LV_RES_OK: a valid pressed cell was found, LV_RES_INV: no valid cell is pressed

Parameters

- `table`: pointer to a table object
- `row`: pointer to variable to store the pressed row
- `col`: pointer to variable to store the pressed column

`union lv_table_cell_format_t`
#include <lv_table.h> Internal table cell format structure.

Use the `lv_table` APIs instead.

Public Members

uint8_t **`align`**

uint8_t **`right_merge`**

uint8_t **`type`**

uint8_t **crop**

struct *lv_table_cell_format_t*::[**anonymous**] **s**

uint8_t **format_byte**

struct lv_table_ext_t

Public Members

uint16_t **col_cnt**

uint16_t **row_cnt**

char ****cell_data**

lv_coord_t ***row_h**

lv_style_list_t **cell_style**[**LV_TABLE_CELL_STYLE_CNT**]

lv_coord_t **col_w**[**LV_TABLE_COL_MAX**]

uint8_t **cell_types**

5.31 Tabview (lv_tabview)

5.31.1 Overview

The Tab view object can be used to organize content in tabs.

5.31.2 Parts and Styles

The Tab view object has several parts. The main is `LV_TABVIEW_PART_BG`. It a rectangle-like container which holds the other parts of the Tab view.

On the background 2 important real parts are created:

- `LV_TABVIEW_PART_BG_SCRL`: it's the scrollable part of *Page*. It holds the content of the tabs next to each other. The background of the Page is always transparent and can't be accessed externally.
- `LV_TABVIEW_PART_TAB_BG`: The tab buttons which is a *Button matrix*. Clicking on a button will scroll `LV_TABVIEW_PART_BG_SCRL` to the related tab's content. The tab buttons can be accessed via `LV_TABVIEW_PART_TAB_BTN`. The height of the tab's button matrix is calculated from the font height plus padding of the background's and the button's style.

All the listed parts supports the typical background style properties and padding.

`LV_TABVIEW_PART_TAB_BG` has an additional real part, an indicator, called `LV_TABVIEW_PART_INDIC`. It's a thin rectangle-like object under the currently selected tab. When the tab view is animated to an other tab the indicator will be animated too. It can be styles using the typical background style properties. The *size* style property will set the its thickness.

When a new tab is added a *Page* is create for them on `LV_TABVIEW_PART_BG_SCRL` and a new button is added to `LV_TABVIEW_PART_TAB_BG` Button matrix. The created Pages can be used as normal Pages and they have the usual Page parts.

5.31.3 Usage

Adding tab

New tabs can be added with `lv_tabview_add_tab(tabview, "Tab name")`. It will return with a pointer to a *Page* object where the tab's content can be created.

Change tab

To select a new tab you can:

- Click on it on the Button matrix part
- Slide
- Use `lv_tabview_set_tab_act(tabview, id, LV_ANIM_ON/OFF)` function

Tab button's position

By default, the tab selector buttons are placed on the top of the Tab view. It can be changed with `lv_tabview_set_btns_pos(tabview, LV_TABVIEW_TAB_POS_TOP/BOTTOM/LEFT/RIGHT/NONE)`

`LV_TABVIEW_TAB_POS_NONE` will hide the tabs.

Note that, you can't change the tab position from top or bottom to left or right when tabs are already added.

Animation time

The animation time is adjusted by `lv_tabview_set_anim_time(tabview, anim_time_ms)`. It is used when the new tab is loaded.

Scroll propagation

As the tabs' content object is a Page it can receive scroll propagation from an other Page-like object. For example, if a text area is created on the tab's content and that Text area is scrolled but it reached the end the scroll can be propagated to the content Page. It can be enabled with `lv_page/textarea_set_scroll_propagation(obj, true)`.

By default the tab's content Pages have enabled scroll propagation, therefore when they are scrolled horizontally the scroll is propagated to `LV_TABVIEW_PART_BG_SCRL` and this way the Pages will be scrolled.

The manual sliding can be disabled with `lv_page_set_scroll_propagation(tab_page, false)`.

5.31.4 Events

Besides the Generic events the following Special events are sent by the Slider:

- **LV_EVENT_VALUE_CHANGED** Sent when a new tab is selected by sliding or clicking the tab button

Learn more about *Events*.

5.31.5 Keys

The following *Keys* are processed by the Tabview:

- **LV_KEY_RIGHT/LEFT** Select a tab
- **LV_KEY_ENTER** Change to the selected tab

Learn more about *Keys*.

5.31.6 Example

5.31.7 API

Typedefs

typedef uint8_t **lv_tabview_btns_pos_t**

typedef uint8_t **lv_tabview_part_t**

Enums

enum [anonymous]

Position of tabview buttons.

Values:

LV_TABVIEW_TAB_POS_NONE

LV_TABVIEW_TAB_POS_TOP

LV_TABVIEW_TAB_POS_BOTTOM

LV_TABVIEW_TAB_POS_LEFT

LV_TABVIEW_TAB_POS_RIGHT

enum [anonymous]

Values:

LV_TABVIEW_PART_BG = *LV_OBJ_PART_MAIN*

_LV_TABVIEW_PART_VIRTUAL_LAST = *_LV_OBJ_PART_VIRTUAL_LAST*

LV_TABVIEW_PART_BG_SCRLLABLE = *_LV_OBJ_PART_REAL_LAST*

LV_TABVIEW_PART_TAB_BG

LV_TABVIEW_PART_TAB_BTN

LV_TABVIEW_PART_INDIC

_LV_TABVIEW_PART_REAL_LAST

Functions

lv_obj_t ***lv_tabview_create**(*lv_obj_t* **par*, **const** *lv_obj_t* **copy*)

Create a Tab view object

Return pointer to the created tab

Parameters

- `par`: pointer to an object, it will be the parent of the new tab
- `copy`: pointer to a tab object, if not NULL then the new object will be copied from it

lv_obj_t ***lv_tabview_add_tab(***lv_obj_t *tabview*, **const** char **name***)
Add a new tab with the given name

Return pointer to the created page object (lv_page). You can create your content here

Parameters

- `tabview`: pointer to Tab view object where to ass the new tab
- `name`: the text on the tab button

void **lv_tabview_clean_tab(***lv_obj_t *tab*)
Delete all children of a tab created by `lv_tabview_add_tab`.

Parameters

- `tab`: pointer to a tab

void **lv_tabview_set_tab_act(***lv_obj_t *tabview*, uint16_t *id*, *lv_anim_enable_t anim*)
Set a new tab

Parameters

- `tabview`: pointer to Tab view object
- `id`: index of a tab to load
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

void **lv_tabview_set_anim_time(***lv_obj_t *tabview*, uint16_t *anim_time*)
Set the animation time of tab view when a new tab is loaded

Parameters

- `tabview`: pointer to Tab view object
- `anim_time`: time of animation in milliseconds

void **lv_tabview_set_btns_pos(***lv_obj_t *tabview*, *lv_tabview_btns_pos_t btns_pos*)
Set the position of tab select buttons

Parameters

- `tabview`: pointer to a tab view object
- `btns_pos`: which button position

uint16_t **lv_tabview_get_tab_act(const** *lv_obj_t *tabview*)
Get the index of the currently active tab

Return the active tab index

Parameters

- `tabview`: pointer to Tab view object

uint16_t **lv_tabview_get_tab_count(const** *lv_obj_t *tabview*)
Get the number of tabs

Return tab count

Parameters

- `tabview`: pointer to Tab view object

lv_obj_t ***lv_tabview_get_tab(const** *lv_obj_t *tabview*, uint16_t *id***)**
Get the page (content area) of a tab

Return pointer to page (lv_page) object

Parameters

- `tabview`: pointer to Tab view object
- `id`: index of the tab (>= 0)

uint16_t **lv_tabview_get_anim_time(const** *lv_obj_t *tabview***)**
Get the animation time of tab view when a new tab is loaded

Return time of animation in milliseconds

Parameters

- `tabview`: pointer to Tab view object

lv_tabview_btns_pos_t **lv_tabview_get_btns_pos(const** *lv_obj_t *tabview***)**
Get position of tab select buttons

Parameters

- `tabview`: pointer to a ab view object

struct lv_tabview_ext_t

Public Members

lv_obj_t ***btns**

lv_obj_t ***indic**

lv_obj_t ***content**

const char ****tab_name_ptr**

lv_point_t **point_last**

uint16_t **tab_cur**

uint16_t **tab_cnt**

uint16_t **anim_time**

lv_tabview_btns_pos_t **btns_pos**

5.32 Text area (lv_textarea)

5.32.1 Overview

The Text Area is a *Page* with a *Label* and a cursor on it. Texts or characters can be added to it. Long lines are wrapped and when the text becomes long enough the Text area can be scrolled.

5.32.2 Parts and Styles

The Text area has the same parts as *Page*. Expect `LV_PAGE_PART_SCRL` because it can't be referenced and it's always transparent. Refer the Page's documentation of details.

Besides the Page parts the virtual `LV_TEXTAREA_PART_CURSOR` part exists to draw the cursor. The cursor's area is always the bounding box of the current character. A block cursor can be created by adding a background color and background opa to `LV_TEXTAREA_PART_CURSOR`'s style. The create line cursor let the cursor transparent and set the *border_side* property.

5.32.3 Usage

Add text

You can insert text or characters to the current cursor's position with:

- `lv_textarea_add_char(textarea, 'c')`
- `lv_textarea_add_text(textarea, "insert this text")`

To add wide characters like `'á'`, `'ß'` or CJK characters use `lv_textarea_add_text(ta, "á")`.

`lv_textarea_set_text(ta, "New text")` changes the whole text.

Placeholder

A placeholder text can be specified - which is displayed when the Text area is empty - with `lv_textarea_set_placeholder_text(ta, "Placeholder text")`

Delete character

To delete a character from the left of the current cursor position use `lv_textarea_del_char(textarea)`. To delete from the right use `lv_textarea_del_char_forward(textarea)`

Move the cursor

The cursor position can be modified directly with `lv_textarea_set_cursor_pos(textarea, 10)`. The `0` position means "before the first characters", `LV_TA_CURSOR_LAST` means "after the last character"

You can step the cursor with

- `lv_textarea_cursor_right(textarea)`
- `lv_textarea_cursor_left(textarea)`
- `lv_textarea_cursor_up(textarea)`
- `lv_textarea_cursor_down(textarea)`

If `lv_textarea_set_cursor_click_pos(textarea, true)` is called the cursor will jump to the position where the Text area was clicked.

Hide the cursor

The cursor can be hidden with `lv_textarea_set_cursor_hidden(textarea, true)`.

Cursor blink time

The blink time of the cursor can be adjusted with `lv_textarea_set_cursor_blink_time(textarea, time_ms)`.

One line mode

The Text area can be configures to be one lined with `lv_ta_set_one_line(ta, true)`. In this mode the height is set automatically to show only one line, line break character are ignored, and word wrap is disabled.

Password mode

The text area supports password mode which can be enabled with `lv_textarea_set_pwd_mode(textarea, true)`.

If the • (Bullet, U+2022) character exists in the font, the entered characters are converted to it after some time or when a new character is entered. If • not exists, * will be used.

In password mode `lv_textarea_get_text(textarea)` gives the real text, not the bullet characters.

The visibility time can be adjusted with `lv_textarea_set_pwd_show_time(textarea, time_ms)`.

Text align

The text can be aligned to the left, center or right with `lv_textarea_set_text_align(textarea, LV_LABEL_ALIGN_LET/CENTER/RIGHT)`.

In one line mode, the text can be scrolled horizontally only if the text is left aligned.

Accepted characters

You can set a list of accepted characters with `lv_textarae_set_accepted_chars(ta, "0123456789.+-")`. Other characters will be ignored.

Max text length

The maximum number of characters can be limited with `lv_textarea_set_max_length(textarea, max_char_num)`

Very long texts

If there is a very long text in the Text area (e. g. > 20k characters) its scrolling and drawing might be slow. However, by enabling `LV_LABEL_LONG_TXT_HINT 1` in *lv_conf.h* it can be hugely improved. It will save some info about the label to speed up its drawing. Using `LV_LABEL_LONG_TXT_HINT` the scrolling and drawing will as fast as with "normal" short texts.

Select text

A part of text can be selected if enabled with `lv_textarea_set_text_sel(textarea, true)`. It works like when you select a text on your PC with your mouse.

Scrollbars

The scrollbars can shown according to different policies set by `lv_textarea_set_sb_mode(textarea, LV_SCRLBAR_MODE_...)`. Learn more at the *Page* object.

Scroll propagation

When the Text area is scrolled on an other scrollable object (like a Page) and the scrolling has reached the edge of the Text area, the scrolling can be propagated to the parent. In other words, when the Text area can be scrolled further, the parent will be scrolled instead.

It can be enabled with `lv_ta_set_scroll_propagation(ta, true)`.

Learn more at the *Page* object.

Edge flash

When the Text area is scrolled to edge a circle like flash animation can be shown if it is enabled with `lv_ta_set_edge_flash(ta, true)`

5.32.4 Events

Besides the Generic events the following Special events are sent by the Slider:

- **LV_EVENT_INSERT** Sent when a character before a character is inserted. The evnet data is the text planned to insert. `lv_ta_set_insert_replace(ta, "New text")` replaces the text to insert. The new text can't be in a local variable which is destroyed when the event callback exists. `""` means do not insert anything.
- **LV_EVENT_VALUE_CHANGED** When the content of the text area has been changed.

5.32.5 Keys

- **LV_KEY_UP/DOWN/LEFT/RIGHT** Move the cursor
- **Any character** Add the character to the current cursor position

Learn more about *Keys*.

5.32.6 Example

5.32.7 API

Typedefs

typedef uint8_t **lv_textarea_style_t**

Enums

enum [anonymous]
Possible text areas styles.

Values:

LV_TEXTAREA_PART_BG = *LV_PAGE_PART_BG*
Text area background style

LV_TEXTAREA_PART_SCROLLBAR = *LV_PAGE_PART_SCROLLBAR*
Scrollbar style

LV_TEXTAREA_PART_EDGE_FLASH = *LV_PAGE_PART_EDGE_FLASH*
Edge flash style

LV_TEXTAREA_PART_CURSOR = *_LV_PAGE_PART_VIRTUAL_LAST*
Cursor style

LV_TEXTAREA_PART_PLACEHOLDER
Placeholder style

_LV_TEXTAREA_PART_VIRTUAL_LAST

_LV_TEXTAREA_PART_REAL_LAST = *_LV_PAGE_PART_REAL_LAST*

Functions

LV_EXPORT_CONST_INT(LV_TEXTAREA_CURSOR_LAST)

lv_obj_t ***lv_textarea_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)
Create a text area objects

Return pointer to the created text area

Parameters

- `par`: pointer to an object, it will be the parent of the new text area
- `copy`: pointer to a text area object, if not NULL then the new object will be copied from it

void **lv_textarea_add_char**(*lv_obj_t *ta*, uint32_t *c*)
Insert a character to the current cursor position. To add a wide char, e.g. 'Á' use 'lv_txt_encoded_conv_wc('Á')'

Parameters

- `ta`: pointer to a text area object
- `c`: a character (e.g. 'a')

void **lv_textarea_add_text**(*lv_obj_t *ta*, **const** char **txt*)
Insert a text to the current cursor position

Parameters

- `ta`: pointer to a text area object
- `txt`: a '\0' terminated string to insert

void **lv_textarea_del_char**(*lv_obj_t *ta*)
Delete a the left character from the current cursor position

Parameters

- `ta`: pointer to a text area object

void **lv_textarea_del_char_forward**(*lv_obj_t *ta*)
Delete the right character from the current cursor position

Parameters

- `ta`: pointer to a text area object

void **lv_textarea_set_text**(*lv_obj_t *ta*, **const** char **txt*)
Set the text of a text area

Parameters

- `ta`: pointer to a text area
- `txt`: pointer to the text

void **lv_textarea_set_placeholder_text**(*lv_obj_t *ta*, **const** char **txt*)
Set the placeholder text of a text area

Parameters

- `ta`: pointer to a text area
- `txt`: pointer to the text

void **lv_textarea_set_cursor_pos**(*lv_obj_t *ta*, int32_t *pos*)
Set the cursor position

Parameters

- `obj`: pointer to a text area object
- `pos`: the new cursor position in character index < 0 : index from the end of the text LV_TEXTAREA_CURSOR_LAST: go after the last character

void **lv_textarea_set_cursor_hidden**(*lv_obj_t *ta*, bool *hide*)
Hide/Unhide the cursor.

Parameters

- `ta`: pointer to a text area object
- `hide`: true: hide the cursor

void **lv_textarea_set_cursor_click_pos**(*lv_obj_t *ta*, bool *en*)
Enable/Disable the positioning of the the cursor by clicking the text on the text area.

Parameters

- `ta`: pointer to a text area object
- `en`: true: enable click positions; false: disable

void **lv_textarea_set_pwd_mode**(*lv_obj_t *ta*, bool *en*)
Enable/Disable password mode

Parameters

- `ta`: pointer to a text area object
- `en`: true: enable, false: disable

void **lv_textarea_set_one_line**(*lv_obj_t *ta*, bool *en*)
Configure the text area to one line or back to normal

Parameters

- `ta`: pointer to a Text area object
- `en`: true: one line, false: normal

void **`lv_textarea_set_text_align`**(*lv_obj_t *ta*, *lv_label_align_t align*)

Set the alignment of the text area. In one line mode the text can be scrolled only with `LV_LABEL_ALIGN_LEFT`. This function should be called if the size of text area changes.

Parameters

- `ta`: pointer to a text are object
- `align`: the desired alignment from `lv_label_align_t`. (LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)

void **`lv_textarea_set_accepted_chars`**(*lv_obj_t *ta*, **const** char **list*)

Set a list of characters. Only these characters will be accepted by the text area

Parameters

- `ta`: pointer to Text Area
- `list`: list of characters. Only the pointer is saved. E.g. "+-.,0123456789"

void **`lv_textarea_set_max_length`**(*lv_obj_t *ta*, uint32_t *num*)

Set max length of a Text Area.

Parameters

- `ta`: pointer to Text Area
- `num`: the maximal number of characters can be added (`lv_textarea_set_text` ignores it)

void **`lv_textarea_set_insert_replace`**(*lv_obj_t *ta*, **const** char **txt*)

In `LV_EVENT_INSERT` the text which planned to be inserted can be replaced by an other text. It can be used to add automatic formatting to the text area.

Parameters

- `ta`: pointer to a text area.
- `txt`: pointer to a new string to insert. If `""` no text will be added. The variable must be live after the `event_cb` exists. (Should be `global` or `static`)

static void **`lv_textarea_set_sscrollbar_mode`**(*lv_obj_t *ta*, *lv_scrollbar_mode_t mode*)

Set the scroll bar mode of a text area

Parameters

- `ta`: pointer to a text area object
- `sb_mode`: the new mode from 'lv_scrollbar_mode_t' enum

static void **`lv_textarea_set_scroll_propagation`**(*lv_obj_t *ta*, bool *en*)

Enable the scroll propagation feature. If enabled then the Text area will move its parent if there is no more space to scroll.

Parameters

- `ta`: pointer to a Text area
- `en`: true or false to enable/disable scroll propagation

static void **`lv_textarea_set_edge_flash`**(*lv_obj_t *ta*, bool *en*)

Enable the edge flash effect. (Show an arc when the an edge is reached)

Parameters

- `page`: pointer to a Text Area
- `en`: true or false to enable/disable end flash

void **`lv_textarea_set_text_sel(`***lv_obj_t *ta*, bool *en*`)**
Enable/disable selection mode.

Parameters

- `ta`: pointer to a text area object
- `en`: true or false to enable/disable selection mode

void **`lv_textarea_set_pwd_show_time(`***lv_obj_t *ta*, uint16_t *time*`)**
Set how long show the password before changing it to '*'

Parameters

- `ta`: pointer to Text area
- `time`: show time in milliseconds. 0: hide immediately.

void **`lv_textarea_set_cursor_blink_time(`***lv_obj_t *ta*, uint16_t *time*`)**
Set cursor blink animation time

Parameters

- `ta`: pointer to Text area
- `time`: blink period. 0: disable blinking

`const` char ***`lv_textarea_get_text(const`** *lv_obj_t *ta*`)**
Get the text of a text area. In password mode it gives the real text (not '*'s).

Return pointer to the text

Parameters

- `ta`: pointer to a text area object

`const` char ***`lv_textarea_get_placeholder_text(`***lv_obj_t *ta*`)**
Get the placeholder text of a text area

Return pointer to the text

Parameters

- `ta`: pointer to a text area object

lv_obj_t ***`lv_textarea_get_label(const`** *lv_obj_t *ta*`)**
Get the label of a text area

Return pointer to the label object

Parameters

- `ta`: pointer to a text area object

uint32_t **`lv_textarea_get_cursor_pos(const`** *lv_obj_t *ta*`)**
Get the current cursor position in character index

Return the cursor position

Parameters

- `ta`: pointer to a text area object

bool **lv_textarea_get_cursor_hidden(const** *lv_obj_t *ta***)**
Get whether the cursor is hidden or not

Return true: the cursor is hidden

Parameters

- `ta`: pointer to a text area object

bool **lv_textarea_get_cursor_click_pos(***lv_obj_t *ta***)**
Get whether the cursor click positioning is enabled or not.

Return true: enable click positions; false: disable

Parameters

- `ta`: pointer to a text area object

bool **lv_textarea_get_pwd_mode(const** *lv_obj_t *ta***)**
Get the password mode attribute

Return true: password mode is enabled, false: disabled

Parameters

- `ta`: pointer to a text area object

bool **lv_textarea_get_one_line(const** *lv_obj_t *ta***)**
Get the one line configuration attribute

Return true: one line configuration is enabled, false: disabled

Parameters

- `ta`: pointer to a text area object

const char ***lv_textarea_get_accepted_chars(***lv_obj_t *ta***)**
Get a list of accepted characters.

Return list of accented characters.

Parameters

- `ta`: pointer to Text Area

uint32_t **lv_textarea_get_max_length(***lv_obj_t *ta***)**
Get max length of a Text Area.

Return the maximal number of characters to be add

Parameters

- `ta`: pointer to Text Area

static *lv_scrollbar_mode_t* **lv_textarea_get_scrollbar_mode(const** *lv_obj_t *ta***)**
Get the scroll bar mode of a text area

Return scrollbar mode from 'lv_scrollbar_mode_t' enum

Parameters

- `ta`: pointer to a text area object

static bool **lv_textarea_get_scroll_propagation(***lv_obj_t *ta***)**
Get the scroll propagation property

Return true or false

Parameters

- `ta`: pointer to a Text area

static bool **lv_textarea_get_edge_flash**(*lv_obj_t *ta*)
Get the scroll propagation property

Return true or false

Parameters

- `ta`: pointer to a Text area

bool **lv_textarea_text_is_selected**(**const** *lv_obj_t *ta*)
Find whether text is selected or not.

Return whether text is selected or not

Parameters

- `ta`: Text area object

bool **lv_textarea_get_text_sel_en**(*lv_obj_t *ta*)
Find whether selection mode is enabled.

Return true: selection mode is enabled, false: disabled

Parameters

- `ta`: pointer to a text area object

uint16_t **lv_textarea_get_pwd_show_time**(*lv_obj_t *ta*)
Set how long show the password before changing it to '*'

Return show time in milliseconds. 0: hide immediately.

Parameters

- `ta`: pointer to Text area

uint16_t **lv_textarea_get_cursor_blink_time**(*lv_obj_t *ta*)
Set cursor blink animation time

Return time blink period. 0: disable blinking

Parameters

- `ta`: pointer to Text area

void **lv_textarea_clear_selection**(*lv_obj_t *ta*)
Clear the selection on the text area.

Parameters

- `ta`: Text area object

void **lv_textarea_cursor_right**(*lv_obj_t *ta*)
Move the cursor one character right

Parameters

- `ta`: pointer to a text area object

void **lv_textarea_cursor_left**(*lv_obj_t *ta*)
Move the cursor one character left

Parameters

- `ta`: pointer to a text area object

void **lv_textarea_cursor_down**(*lv_obj_t* **ta*)
Move the cursor one line down

Parameters

- `ta`: pointer to a text area object

void **lv_textarea_cursor_up**(*lv_obj_t* **ta*)
Move the cursor one line up

Parameters

- `ta`: pointer to a text area object

struct lv_textarea_ext_t

Public Members

lv_page_ext_t **page**

lv_obj_t ***label**

char ***placeholder_txt**

lv_style_list_t **style_placeholder**

char ***pwd_tmp**

const char ***accapted_chars**

uint32_t **max_length**

uint16_t **pwd_show_time**

lv_style_list_t **style**

lv_coord_t **valid_x**

uint32_t **pos**

uint16_t **blink_time**

lv_area_t **area**

uint32_t **txt_byte_pos**

uint8_t **state**

uint8_t **hidden**

uint8_t **click_pos**

struct *lv_textarea_ext_t*::[anonymous] **cursor**

uint32_t **sel_start**

uint32_t **sel_end**

uint8_t **text_sel_in_prog**

uint8_t **text_sel_en**

uint8_t **pwd_mode**

uint8_t **one_line**

5.33 Tile view (lv_tileview)

5.33.1 Overview

The Tileview is a container object where its elements (called *tiles*) can be arranged in a grid form. By swiping the user can navigate between the tiles.

If the Tileview is screen sized it gives a user interface you might have seen on the smartwatches.

5.33.2 Parts and Styles

The Tileview has the same parts as *Page*. Expect `LV_PAGE_PART_SCRL` because it can't be referenced and it's always transparent. Refer the Page's documentation of details.

5.33.3 Usage

Valid positions

The tiles don't have to form a full grid where every element exists. There can be holes in the grid but it has to be continuous, i.e. there can't be an empty rows or columns.

With `lv_tileview_set_valid_positions(tileview, valid_pos_array, array_len)` the valid positions can be set. Scrolling will be possible only to this positions. The `0,0` index means the top left tile. E.g. `lv_point_t valid_pos_array[] = {{0,0}, {0,1}, {1,1}, {{LV_COORD_MIN, LV_COORD_MIN}}` gives a Tile view with "L" shape. It indicates that there is no tile in `{1,1}` therefore the user can't scroll there.

In other words, the `valid_pos_array` tells where the tiles are. It can be changed on the fly to disable some positions on specific tiles. For example, there can be a 2x2 grid where all tiles are added but the first row (y = 0) as a "main row" and the second row (y = 1) contains options for the tile above it. Let's say horizontal scrolling is possible only in the main row and not possible between the options in the second row. In this case the `valid_pos_array` needs to changed when a new main tile is selected:

- for the first main tile: `{0,0}, {0,1}, {1,0}` to disable the `{1,1}` option tile
- for the second main tile `{0,0}, {1,0}, {1,1}` to disable the `{0,1}` option tile

Set tile

To set the currently visible tile use `lv_tileview_set_tile_act(tileview, x_id, y_id, LV_ANIM_ON/OFF)`.

Add element

To add elements just create an object on the Tileview and position it manually to the desired position.

`lv_tileview_add_element(tielview, element)` should be used to make possible to scroll (drag) the Tileview by one its element. For example, if there is a button on a tile, the button needs to be explicitly added to the Tileview to enable the user to scroll the Tileview with the button too.

Scroll propagation

The scroll propagation feature of page-like objects (like *List*) can be used very well here. For example, there can be a full-sized List and when it reaches the top or bottom most position the user will scroll the tile view instead.

Animation time

The animation time of the Tileview can be adjusted with `lv_tileview_set_anim_time(tileview, anim_time)`.

Animations are applied when

- a new tile is selected with `lv_tileview_set_tile_act`
- the current tile is scrolled a little and then released (revert the original title)
- the current tile is scrolled more than half size and then released (move to the next tile)

Edge flash

An "edge flash" effect can be added when the tile view reached hits an invalid position or the end of tile view when scrolled.

Use `lv_tileview_set_edge_flash(tileview, true)` to enable this feature.

5.33.4 Events

Besides the Generic events the following Special events are sent by the Slider:

- **LV_EVENT_VALUE_CHANGED** Sent when a new tile loaded either with scrolling or `lv_tileview_set_act`. The event data is set ti the index of the new tile in `valid_pos_array` (It's type is `uint32_t *`)

5.33.5 Keys

- **LV_KEY_UP**, **LV_KEY_RIGHT** Increment the slider's value by 1
- **LV_KEY_DOWN**, **LV_KEY_LEFT** Decrement the slider's value by 1

Learn more about *Keys*.

5.33.6 Example

5.33.7 API

Enums

enum [anonymous]

Values:

LV_TILEVIEW_PART_BG = *LV_PAGE_PART_BG*

LV_TILEVIEW_PART_SCROLLBAR = *LV_PAGE_PART_SCROLLBAR*

LV_TILEVIEW_PART_EDGE_FLASH = *LV_PAGE_PART_EDGE_FLASH*

_LV_TILEVIEW_PART_VIRTUAL_LAST = *_LV_PAGE_PART_VIRTUAL_LAST*

_LV_TILEVIEW_PART_REAL_LAST = *_LV_PAGE_PART_REAL_LAST*

Functions

lv_obj_t ***lv_tileview_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a tileview objects

Return pointer to the created tileview

Parameters

- `par`: pointer to an object, it will be the parent of the new tileview
- `copy`: pointer to a tileview object, if not NULL then the new object will be copied from it

void **lv_tileview_add_element**(*lv_obj_t *tileview*, *lv_obj_t *element*)

Register an object on the tileview. The register object will able to slide the tileview

Parameters

- `tileview`: pointer to a Tileview object
- `element`: pointer to an object

void **lv_tileview_set_valid_positions**(*lv_obj_t *tileview*, **const** lv_point_t *valid_pos*[], uint16_t *valid_pos_cnt*)

Set the valid position's indices. The scrolling will be possible only to these positions.

Parameters

- `tileview`: pointer to a Tileview object
- `valid_pos`: array width the indices. E.g. `lv_point_t p[] = {{0,0}, {1,0}, {1, 1}}`. Only the pointer is saved so can't be a local variable.
- `valid_pos_cnt`: number of elements in `valid_pos` array

void **lv_tileview_set_tile_act**(*lv_obj_t *tileview*, lv_coord_t *x*, lv_coord_t *y*, *lv_anim_enable_t anim*)

Set the tile to be shown

Parameters

- `tileview`: pointer to a tileview object
- `x`: column id (0, 1, 2...)
- `y`: line id (0, 1, 2...)
- `anim`: LV_ANIM_ON: set the value with an animation; LV_ANIM_OFF: change the value immediately

static void **lv_tileview_set_edge_flash**(*lv_obj_t *tileview*, bool *en*)

Enable the edge flash effect. (Show an arc when the an edge is reached)

Parameters

- `tileview`: pointer to a Tileview
- `en`: true or false to enable/disable end flash

static void **lv_tileview_set_anim_time**(*lv_obj_t *tileview*, uint16_t *anim_time*)

Set the animation time for the Tile view

Parameters

- `tileview`: pointer to a page object
- `anim_time`: animation time in milliseconds

void **lv_tileview_get_tile_act**(*lv_obj_t *tileview*, lv_coord_t **x*, lv_coord_t **y*)
Get the tile to be shown

Parameters

- `tileview`: pointer to a tileview object
- `x`: column id (0, 1, 2...)
- `y`: line id (0, 1, 2...)

static bool **lv_tileview_get_edge_flash**(*lv_obj_t *tileview*)
Get the scroll propagation property

Return true or false

Parameters

- `tileview`: pointer to a Tileview

static uint16_t **lv_tileview_get_anim_time**(*lv_obj_t *tileview*)
Get the animation time for the Tile view

Return animation time in milliseconds

Parameters

- `tileview`: pointer to a page object

struct lv_tileview_ext_t

Public Members

lv_page_ext_t **page**

const lv_point_t ***valid_pos**

uint16_t **valid_pos_cnt**

uint16_t **anim_time**

lv_point_t **act_id**

uint8_t **drag_top_en**

uint8_t **drag_bottom_en**

uint8_t **drag_left_en**

uint8_t **drag_right_en**

5.34 Window (lv_win)

5.34.1 Overview

The Window is container-like objects built from a header with title and button and a content area.

5.34.2 Parts and Styles

The main part is `LV_WIN_PART_BG` which holds the two other real parts:

1. `LV_WIN_PART_HEADER`: a header *Container* on the top with a title and control buttons
2. `LV_WIN_PART_CONTENT_SCRL` the scrollable part of a *Page* for the content below the header.

Besides these, `LV_WIN_PART_CONTENT_SCRL` has a scrollbar part called `LV_WIN_PART_CONTENT_SCRL`. Read the documentation of *Page* for more details on the scrollbars.

All parts supports the typical background properties. The title uses the *Text* properties of the header part.

The height of the control buttons is: *header height - header padding_top - header padding_bottom*.

Title

On the header, there is a title which can be modified by: `lv_win_set_title(win, "New title")`.

Control buttons

Control buttons can be added to the right of the window header with: `lv_win_add_btn_right(win, LV_SYMBOL_CLOSE)`, to add a button to the left side of the window header use `lv_win_add_btn_left(win, LV_SYMBOL_CLOSE)` instead. The second parameter is an *Image* source so it can be a symbol, a pointer to an `lv_img_dsc_t` variable or a path to file.

The width of the buttons can be set with `lv_win_set_btn_width(win, w)`. If `w == 0` the buttons will be square-shaped.

`lv_win_close_event_cb` can be used as an event callback to close the Window.

Scrollbars

The scrollbar behavior can be set by `lv_win_set_scrlbar_mode(win, LV_SCRLBAR_MODE_...)`. See *Page* for details.

Manual scroll and focus

To scroll the Window directly you can use `lv_win_scroll_hor(win, dist_px)` or `lv_win_scroll_ver(win, dist_px)`.

To make the Window show an object on it use `lv_win_focus(win, child, LV_ANIM_ON/OFF)`.

The time of scroll and focus animations can be adjusted with `lv_win_set_anim_time(win, anim_time_ms)`

Layout

To set a layout for the content use `lv_win_set_layout(win, LV_LAYOUT_...)`. See *Container* for details.

5.34.3 Events

Only the Generic events are sent by the object type.

Learn more about *Events*.

5.34.4 Keys

The following *Keys* are processed by the Page:

- **LV_KEY_RIGHT/LEFT/UP/DOWN** Scroll the page

Learn more about *Keys*.

5.34.5 Example

5.34.6 API

Enums

enum [**anonymous**]

Window parts.

Values:

LV_WIN_PART_BG = *LV_OBJ_PART_MAIN*

Window object background style.

_LV_WIN_PART_VIRTUAL_LAST

LV_WIN_PART_HEADER = *_LV_OBJ_PART_REAL_LAST*

Window titlebar background style.

LV_WIN_PART_CONTENT_SCROLLABLE

Window content style.

LV_WIN_PART_SCROLLBAR

Window scrollbar style.

_LV_WIN_PART_REAL_LAST

Functions

lv_obj_t ***lv_win_create**(*lv_obj_t *par*, **const** *lv_obj_t *copy*)

Create a window objects

Return pointer to the created window

Parameters

- `par`: pointer to an object, it will be the parent of the new window
- `copy`: pointer to a window object, if not NULL then the new object will be copied from it

void **lv_win_clean**(*lv_obj_t *win*)

Delete all children of the scrl object, without deleting scrl child.

Parameters

- `win`: pointer to an object

lv_obj_t ***lv_win_add_btn**(*lv_obj_t *win*, **const** void **img_src*)
Add control button to the header of the window

Return pointer to the created button object

Parameters

- `win`: pointer to a window object
- `img_src`: an image source ('lv_img_t' variable, path to file or a symbol)

void **lv_win_close_event_cb**(*lv_obj_t *btn*, *lv_event_t event*)
Can be assigned to a window control button to close the window

Parameters

- `btn`: pointer to the control button on the widows header
- `evet`: the event type

void **lv_win_set_title**(*lv_obj_t *win*, **const** char **title*)
Set the title of a window

Parameters

- `win`: pointer to a window object
- `title`: string of the new title

void **lv_win_set_header_height**(*lv_obj_t *win*, lv_coord_t *size*)
Set the control button size of a window

Return control button size

Parameters

- `win`: pointer to a window object

void **lv_win_set_btn_width**(*lv_obj_t *win*, lv_coord_t *width*)
Set the width of the control buttons on the header

Parameters

- `win`: pointer to a window object
- `width`: width of the control button. 0: to make them square automatically.

void **lv_win_set_content_size**(*lv_obj_t *win*, lv_coord_t *w*, lv_coord_t *h*)
Set the size of the content area.

Parameters

- `win`: pointer to a window object
- `w`: width
- `h`: height (the window will be higher with the height of the header)

void **lv_win_set_layout**(*lv_obj_t *win*, *lv_layout_t layout*)
Set the layout of the window

Parameters

- `win`: pointer to a window object
- `layout`: the layout from 'lv_layout_t'

void **lv_win_set_scrollbar_mode**(*lv_obj_t *win*, *lv_scrollbar_mode_t sb_mode*)
Set the scroll bar mode of a window

Parameters

- `win`: pointer to a window object
- `sb_mode`: the new scroll bar mode from 'lv_scrollbar_mode_t'

void **`lv_win_set_anim_time`**(*lv_obj_t *win*, uint16_t *anim_time*)
Set focus animation duration on *`lv_win_focus()`*

Parameters

- `win`: pointer to a window object
- `anim_time`: duration of animation [ms]

void **`lv_win_set_drag`**(*lv_obj_t *win*, bool *en*)
Set drag status of a window. If set to 'true' window can be dragged like on a PC.

Parameters

- `win`: pointer to a window object
- `en`: whether dragging is enabled

`const` char ***`lv_win_get_title`**(**`const`** *lv_obj_t *win*)
Get the title of a window

Return title string of the window

Parameters

- `win`: pointer to a window object

lv_obj_t ***`lv_win_get_content`**(**`const`** *lv_obj_t *win*)
Get the content holder object of window (`lv_page`) to allow additional customization

Return the Page object where the window's content is

Parameters

- `win`: pointer to a window object

lv_coord_t **`lv_win_get_header_height`**(**`const`** *lv_obj_t *win*)
Get the header height

Return header height

Parameters

- `win`: pointer to a window object

lv_coord_t **`lv_win_get_btn_width`**(*lv_obj_t *win*)
Get the width of the control buttons on the header

Return width of the control button. 0: square.

Parameters

- `win`: pointer to a window object

lv_obj_t ***`lv_win_get_from_btn`**(**`const`** *lv_obj_t *ctrl_btn*)
Get the pointer of a widow from one of its control button. It is useful in the action of the control buttons where only button is known.

Return pointer to the window of 'ctrl_btn'

Parameters

- `ctrl_btn`: pointer to a control button of a window

lv_layout_t **lv_win_get_layout(***lv_obj_t *win***)**
Get the layout of a window

Return the layout of the window (from 'lv_layout_t')

Parameters

- `win`: pointer to a window object

lv_scrollbar_mode_t **lv_win_get_sb_mode(***lv_obj_t *win***)**
Get the scroll bar mode of a window

Return the scroll bar mode of the window (from 'lv_sb_mode_t')

Parameters

- `win`: pointer to a window object

uint16_t **lv_win_get_anim_time(const** *lv_obj_t *win***)**
Get focus animation duration

Return duration of animation [ms]

Parameters

- `win`: pointer to a window object

lv_coord_t **lv_win_get_width(***lv_obj_t *win***)**
Get width of the content area (page scrollable) of the window

Return the width of the content area

Parameters

- `win`: pointer to a window object

static bool **lv_win_get_drag(const** *lv_obj_t *win***)**
Get drag status of a window. If set to 'true' window can be dragged like on a PC.

Return whether window is draggable

Parameters

- `win`: pointer to a window object

void **lv_win_focus(***lv_obj_t *win*, *lv_obj_t *obj*, *lv_anim_enable_t anim_en***)**
Focus on an object. It ensures that the object will be visible in the window.

Parameters

- `win`: pointer to a window object
- `obj`: pointer to an object to focus (must be in the window)
- `anim_en`: LV_ANIM_ON focus with an animation; LV_ANIM_OFF focus without animation

static void **lv_win_scroll_hor(***lv_obj_t *win*, lv_coord_t *dist***)**
Scroll the window horizontally

Parameters

- `win`: pointer to a window object
- `dist`: the distance to scroll (< 0: scroll right; > 0 scroll left)

static void **lv_win_scroll_ver(***lv_obj_t *win*, lv_coord_t *dist***)**
Scroll the window vertically

Parameters

- `win`: pointer to a window object
- `dist`: the distance to scroll (< 0: scroll down; > 0 scroll up)

struct lv_win_ext_t

Public Members

lv_obj_t ***page**

lv_obj_t ***header**

char ***title_txt**

lv_coord_t **btn_w**

CHAPTER

SIX

CONTRIBUTING

6.1 Introduction

Join LVGL's community and leave your footprint in the library!

There are a lot of ways to contribute to LVGL even if you are are new to the library or even new to programming.

It might be scary to make the first step but you have nothing to afraid of. A friendly and helpful community is waiting to know like-minded people and make something great together.

So let's find which contribution option fits you the best and join the development of LVGL!

Before getting started here are some guidelines to make contribution smoother:

- Be kind and friendly.
- Be sure to read the relevant part of the documentation before posting a question.
- Ask questions in Forum and use [GitHub]((https://github.com/lvgl/) for development-related discussions.
- Fill the post or issue templates in the Forum or GitHub. It makes much easier to understand your case and you will get a useful answer faster.
- If possible send an absolute minimal but build-able code example in order to reproduce the issue. Be sure it contains all required variable declarations, constants, and assets (images, fonts).
- Use Markdown to format your posts. You can learn it in 10 minutes.
- Speak about one thing in one issue or topic.
- Give feedback and close the issue or mark the topic as solved if your question is answered.
- For non-trivial fixes and features, it's better to open an issue first to discuss the details instead of sending a pull request directly.
- Please read and follow the Coding style guide.

6.1.1 Pull request

Merging new code into lvgl, documentation, blog, examples, and other repositories happen via *Pull requests* (PR for short). A PR is a notification like "Hey, I made some updates to your project. Here are the changes, you can add them if you want." To do this you need a copy (called fork) of the original project under your account, make some changes there, and notify the original repository about your updates. You can see how it looks like on GitHub for lvgl here: https://github.com/lvgl/lvgl/pulls

To add your changes you can edit files online on GitHub and send a new Pull request from there (recommended for small changes) or add the updates in your favorite editor/IDE and use git to publish the changes (recommended for more complex updates).

From GitHub

1. Navigate to the file you want to edit.
2. Click the Edit button in the top right-hand corner.
3. Add your changes to the file
4. Add a commit message on the bottom of the page
5. Click the *Propose changes* button

From command line

It's about the `lvgl` repository but it works the same way for other repositories too.

1. Fork the lvgl repository. To do this click the "Fork" button in the top right corner. It will "copy" the `lvgl` repository to your GitHub account (`https://github.com/<YOUR_NAME>?tab=repositories`)
2. Clone your forked repository.
3. Add your changes. You can create a *feature branch* from *master* for the updates: `git checkout -b the-new-feature`
4. Commit and push you changed to the forked `lvgl` repository.
5. Create a PR on GitHub from the page of your `lvgl` repository (`https://github.com/<YOUR_NAME>/lvgl`) by clicking the *"New pull request"* button. Don't forget to select the branch where you added your changes.
6. Set the base branch. It means where you want to merge your update. In the `lvgl` repo fixes go to `master`, new features to `dev` branch.
7. Describe what is in the update. An example code is welcome if applicable.
8. Update** your `lvgl` repo with new commits. They will automatically appear in the PR too.

6.2 When you got started with LVGL

Even if you're just getting started with LVGL there plenty of ways to make your feet wet. Most of these options even doesn't require knowing a single line of code of LVGL.

6.2.1 Give LVGL a Star

Show that you like LVGL by giving it star on GitHub!

Star

This simple click makes LVGL more visible on GitHub and makes it more attractive to other people. So with this, you already helped a lot!

6.2.2 Tell what you have achieved

Have you already started LVGL in a *Simulator*, a development board, or your custom hardware? Was it easy or where there some obstacles? Are you happy with the result?

If so why don't you tell it to your friends? You can post it on Twitter, Facebook, LinkedIn, or create a YouTube video.

Any of these helps a lot to spread the word of LVGL and familiarize it with new developers.

Only thing: don't forget to add a link to https://lvg.io or https://github.com/lvgl and #lvgl. Thank you! :)

6.2.3 Write examples

As you learn LVGL probably you will play with the features of widgets. But why don't you publish your experiments?

Every widgets' documentation contains some examples. For example here are the examples of the Drop-down list. The examples are directly loaded from the lv_examples repository.

So all you need to do is sending a Pull request to the lv_examples repository and follow some conventions:

- Name the examples like `lv_ex_<widget_name>_<id>`
- Make the example as short and simple as possible
- Add comments to explain what the example does
- Use 320x240 resolution
- Create a screenshot about the example
- Update `index.rst` in the example's folder with your new example. See how the other examples are added.

6.2.4 Improve the docs

As you read the documentation you might see some typos or unclear sentences. For typos and straightforward fixes, you can simply edit the file on GitHub. There is an `Edit on Github` link on the top right-hand corner of all pages. Click it to see the file on GitHub, hit the Edit button, and add you fixes as described in Pull request - From GitHub section.

Note that, the documentation is also formatted in Markdown.

6.2.5 Translate the docs

If you have more free time you can even translate the documentation. The currently available languages are shown in the locals folder.

If your chosen language is still not added, please write a comment here.

To add your translations:

- Find the *.po* in `<language_code>/LC_MESSAGES/<section_name>.po`. E.g. the widgets translated to German should be in `de/LC_MESSAGES/widgets.po`.
- Open a po file and fill the `msgstr` fields with the translation
- Send a Pull request

To display a translation in the public documentation page at least these sections should be translated:

- Get started: Quick overview
- Overview: Objects, Events, Styles
- Porting: System overview, Set-up a project, Display interface, Input device Interface, Tick interface
- 5 widgets of your choice

6.2.6 Write a blog post

LVGL Blog is free for everyone. It's a good place to talk about a project you created with LVGL, write a tutorial, or share some nice tricks. The latest blog posts are shown on the homepage of LVGL to make your work more visible.

The blog is hosted on GitHub. If you add a post GitHub automatically turns it into a website. See the README of the blog repo to see how to add your post.

6.3 When you already use LVGL

6.3.1 Give feedback

Let us know what you are working on! You can open a new topic in the My projects category of the Forum. Showing your project to others is a win-win situation because it increases your and LVGL's reputation at the same time.

If you don't want to speak about it publicly feel free to use Contact form on lvgl.io to private message to us.

6.3.2 Report bugs

As you use LVGL you might find bugs. Before reporting them be sure to check the relevant parts of the documentation.

If it really seems like a bug feel free to open an issue on GitHub.

When filing the issue be sure to fill the template. It helps a lot to find the root of the problems and helps to avoid a lot of questions.

6.3.3 Send fixes

The beauty of open-source software is you can see how they work and fix or adjust them as you need. If you found a bug and was able to fix it don't hesitate to send a Pull request with the fix.

In your Pull request please also add a line to CHANGELOG.md.

6.3.4 Join the conversations in the Forum

It feels so good when you are not alone if something is not working. But it's even better to help people when they struggle with something.

While you were learning LVGL you might have questions and used the Forum to get answers. Now you already know how LVGL works and have a decent knowledge about it.

It's a nice way of contribution to use the Forum and answer the questions of newcomers - like you were once.

Just read the titles and if you are familiar with the topic don't hesitate to share your thoughts.

Participating in the discussions is one of the best ways to part of the project and know like-minded people!

6.3.5 Add features

We collect the planned features in GitHub issues tracker and mark them with Help wanted label. If you are interested in any of them feel free to tell your remarks, and/or participate in the the implementation.

Other features which are (still) not on the road map are listed in the Feature request category of the Forum. If you have a feature idea for LVGL please use the Forum the share it!

When adding a new features the followings also needs to be updated:

- Add a line to CHANGELOG.md.
- Update the documentation. See this guide.
- Add an example if applicable. See this guide.

6.4 When you are confident with LVGL

6.4.1 Become a maintainer

If you really want to part of the development you can become a maintainer of a repo. By becoming a maintainer

- you get write access to that repo: - add code directly without sending a Pull request - accept Pull request - close/repopen/edit issues
- your name will be added in the credits section of lvgl.io/about (will be added soon) and lvgl' s README.
- you can join the Core_contributor group in the Forum and get the LVGL logo on your avatar.
- your word has higher impact when we make decisions

You can become a maintainer by invitation, however the followings need to met

1. Have > 50 replays in the Forum. You can your stats here
2. Send > 5 non trivial Pull request to the repo where you would like to be maintainer

If you are interested, just send a message (e.g. from the Forum) to the current maintainers of the repository. They will check is the prerequisites are met. Note that, it's not automatic process, i.e. if the conditions are met you won't be automatically a maintainer. It's up to the current maintainers to make the decision.

6.4.2 Garden your repo under LVGL organization

Besides the core `lvgl` repository there are other repos for ports to development boards, IDEs or other environment. If you ported LVGL to a new platform we can host it under the LVGL organization among the other repos.

This way you project will become the part of whole LVGL project and get more visibility. If you are interested in this opportunity just open an issue in lvgl repo and tell what you have!

After that, it all seems good, we open a repo for you project where you will have admin rights. Besides your name will be added in the credits section of lvgl.io/about (will be added soon) and lvgl' s README and you can join the Core_contributor group in the Forum and get the LVGL logo on your avatar.

To make this concept sustainable there a few rules to follow:

- You need to add a README to your repo.
- We expect to maintain the repo some extent:
  - Follow at least major the versions of lvgl
  - respond to the issues (in a reasonable time)
- if there is no activity in a repo for 6 month it will be arcived