

---

# **LittlevGL Documentation**

*Release 6.0*

**Gabor Kiss-Vamosi**

**Jul 16, 2019**

## CONTENTS

English (en) - (zh-CN) Magyar (hu) - Türk (tr)

PDF version: [LittlevGL.pdf](#)



LittlevGL egy ingyenes, nyílt forráskódú grafikus könyvtár, ami mindent biztosít számodra, hogy beágyazott GUI-t készíts könnyen használható grafikus elemekkel, gyönyörű effekttekkel és alacsony memória használat-tal.

[Website](#) · [GitHub](#) · [Forum](#) · [Live demo](#) · [Simulator](#) · [Blog](#)

## **FONTOS FUNKCIÓK**

- Powerful building blocks buttons, charts, lists, sliders, images etc
- Advanced graphics with animations, anti-aliasing, opacity, smooth scrolling
- Various input devices 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 graphical 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 PC without embedded hardware
- Tutorials, examples, themes for rapid GUI design
- Documentation online and offline
- Free and open-source under MIT license

## REQUIREMENTS

- 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: ~8..16 kB depending on the used features and objects types
  - Stack: > 2kB (> 4 kB is recommended)
  - Dynamic data (heap): > 4 KB (> 16 kB is recommended if using several objects). Set by `LV_MEM_SIZE` in `lv_conf.h`.
  - Display buffer: > “*Horizontal resolution*” pixels (>  $10 \times$  “*Horizontal resolution*” is recommended)
- 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.*

### 3.1 Where to get started?

- For a general overview of LittlevGL visit [littlevgl.com](https://littlevgl.com)
- To make some experiments with LittlevGL in a simulator on your PC or in even in your browser see the *Get started* guide.
- To see how you can port LittlevGL to your device go to the *Porting* section.
- To learn how LittlevGL works start to read the *Overview*.
- To read tutorials or share your own experiences go to the *Blog*
- To see the source code of the library go to GitHub: <https://github.com/littlevgl/lvgl/>.

### 3.2 Where can I ask questions?

To ask questions in the Forum: <https://forum.littlevgl.com/>.

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.

### 3.3 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 LittlevGL. 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
- and anything else with a strong enough MCU and a periphery to drive a display

### 3.4 Is my display supported?

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

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

See the *Porting* section to learn more.

### 3.5 Is LittlevGL free? How can I use it in a commercial product?

LittlevGL comes with MIT license which means you can download and use it for any purpose you want without any obligations.

### 3.6 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.

### 3.7 Why the display driver is called only one? Only the upper part of the display is refreshed.

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

### 3.8 Why I see only garbage on the screen?

Probably there a bug in your display driver. Try the following code without using LittlevGL:

```
#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 < BUF_H; y++) {
    lv_color_t c = lv_color_mix(LV_COLOR_BLUE, LV_COLOR_RED, (y * 255) / BUF_H);
    for(x = 0; x < BUF_W; x++){
        (*buf_p) = c;
        buf_p++;
    }
}

lv_area_t a;
```

(continues on next page)

(continued from previous page)

```
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);
```

## 3.9 Why I see non-sense colors on the screen?

Probably LittlevGL's the 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.

## 3.10 How to speed up my UI?

- Turn on compiler optimization
- 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 you 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 external SRAM) because LittlevGL uses it a lot and it should have a small access time

## 3.11 How to reduce flash/ROM usage?

You can disable all the unused feature (like 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. ‘

## 3.12 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.



## 3.13 How to work with an operating system?

To work with an operating system where tasks can interrupt each other you should protect LittlevGL related function calls with a mutex. See the *Operation system* section to learn more.

## 3.14 How to contribute to LittlevGL?

There are several ways to contribute to LittlevGL:

- write a few lines about your project to inspire others
- answer other's questions
- report and/or fix bugs
- suggest and/or implement new features
- improve and/or translate the documentation
- write a blog post about your experiences

To learn more see [Contributing guide](#)

## 3.15 Where can I find the documentation of the previous version (v5.3)?

You can download it here and open offline:

`Docs-v5-3.zip`

---

### 3.15.1 Get started

#### Élő demo

You can see how LittlevGL looks like without installing and downloading anything. There some ready made user interfaces which you can easily try in your browser.

Go to the [Live demo](#) page and choose a demo you are interested in.

#### Micropython

##### What is Micropython?

[Micropython](#) is Python for microcontrollers. With Micropython you can write Python3 code and run it on bare metal architectures with limited resources.

## Micropython highlights

- **Compact** - fit and run within just 256k of code space and 16k of RAM. No OS is needed, although you can also run it with OS, if you want.
  - **Compatible** - strives to be as compatible as possible with normal Python (known as CPython)
  - **Verstile** - 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. 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.)
- 

## Why Micropython + LittlevGL?

Micropython today [does not have a good high-level GUI library](#). LittlevGL is a good high-level GUI library, it's implemented in C and its API is in C. LittlevGL is an [Object Oriented Component Based](#) library, which seems a natural candidate to map into a higher level language, such as Python.

## Here are some advantages of using LittlevGL in Micropython:

- Develop GUI in Python, a very popular high level language. Use paradigms such as Object Oriented Programming.
- 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 + LittlevGL 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 LittlevGL 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.
-

## So how does it look like?

TL;DR: It's very much like the C API, but Object Oriented for LittlevGL 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)
```

## How can I use it?

### Online Simulator

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

[Link to the online simulator](#)

### PC Simulator

Micropython is ported to many platform, one of the is the “unix port”, which allows you to build and run Micropython (+LittlevGL) on a Linux machine. (On a windows machine you might need Virtual Box or WSL or MinGW or Cygwin etc.)

[More information about building and running the unix port](#)

## The real thing

At the end, the goal is to run it all on an embedded platform. Both Micropython and LittlevGL can be used on many embedded architectures, such as stm32, ESP32 etc. You would also need display and input drivers. We have some example 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!

## Where can I find more information?

- [On the Blog Post](#)
- [On lv\\_micropython README](#)
- [On lv\\_binding\\_micropython README](#)

- On LittlevGL forum (Feel free to ask anything!)
- On Micropython [docs](#) and [forum](#)

## Simulator on PC

You can try out the LittlevGL **using only your PC** without any development boards. Write a code, run it on the PC and see the result on the monitor. It is cross-platform: Windows, Linux and OSX are supported. The written code is portable, you can simply copy it when using an embedded hardware.

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](#).

## Select an IDE

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

In followings the set-up guide of Eclipse CDT is described in more details.

## Set-up Eclipse CDT

### Install Eclipse CDT

Eclipse CDT is C/C++ IDE. You can use other IDEs as well but in this tutorial the configuration for Eclipse CDT is shown.

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`

You can download Eclipse's CDT from: <https://eclipse.org/cdt/>. 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 it 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 will use **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 I suggest [this tutorial](#) to get started with SDL.

## Pre-configured project

A pre-configured graphics library project (based on the latest release) is always available. You can find it on [GitHub](#) or on the [Download](#) page. (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 it check that path and copy (and unzip) the downloaded pre-configured project there. Now you can accept the workspace path. Of course you can modify this path but in that case copy the project to that 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

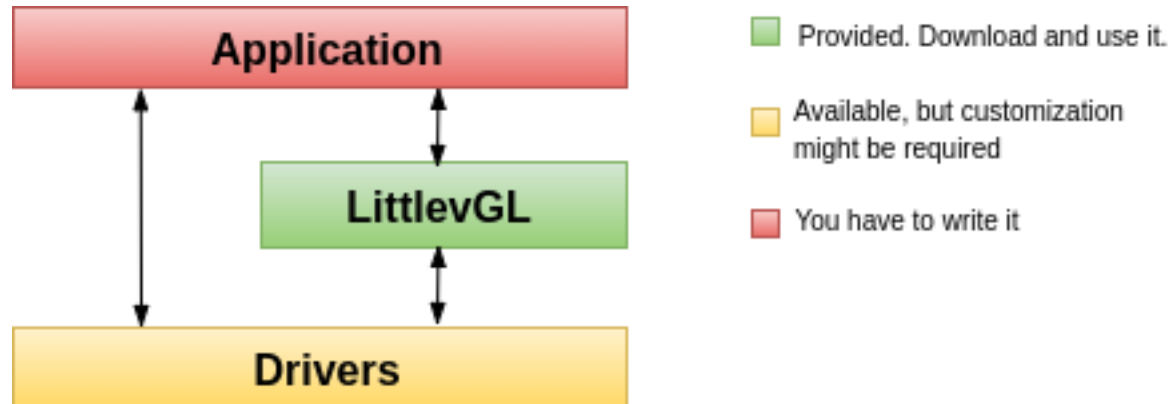
Now you are ready to run the LittlevGL Graphics Library on your PC. Click on the Hammer Icon on the top menu bar to Build the project. If you have done everything right you will not get any errors. Note that on some systems additional steps might be required to "see" SDL 2 from Eclipse but in most of 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 LittlevGL Graphics Library in the practice or begin the development on your PC.

### 3.15.2 Porting

#### System overview



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

**LittlevGL** 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.

There are **two typical hardware set-ups** depending on the MCU has an LCD/TFT driver periphery or not. 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.

#### Set-up a project

##### Get the library

LittlevGL Graphics Library is available on GitHub: <https://github.com/littlevgl/lvgl>.

You can clone it or download the latest version of the library from GitHub or you can use the [Download](#) page as well.

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

## Config file

There is a configuration header file for LittlevGL called **lv\_conf.h**. It sets the library's basic behavior, 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) 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).

## Initialization

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

1. Call *lv\_init()*
2. Initialize your drivers
3. Register the display and input devices drivers in LittlevGL. 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 LittlevGL related tasks. *Learn more.*

## 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.

## 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];
```

(continues on next page)

(continued from previous page)

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

There are there possible configurations regarding the buffer size:

1. **One buffer** LittlevGL 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 LittlevGL 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* LittlevGL 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* LittlevGL 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 LittlevGL. 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.

## 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**. LittlevGL 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 be enabled in `lv_conf.h` To use a GPU the following callbacks can be used:
- **gpu\_fill\_cb** fill an area with colors.
- **gpu\_blend\_cb** blend two buffers using opacity.

Some other optional callbacks to make easier and more optimal to work with monochrome, gray-scale 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 in a more compact way if the display has a special color format. (e.g. 1 bit monochrome, 2 bit gray-scale 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.
- **monitor\_cb** a callback function tell 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 LittlevGL `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)
↪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_mem_fill_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest_buf, const lv_area_t
↪
↪* dest_area, const lv_area_t * fill_area, lv_color_t color);
{
    /*It's an example code which should be done by your GPU*/
    uint32_t x,y;
    for(y = 0; y < length; y++) {
        dest[y] = color;
    }
}

void my_mem_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++) {
```

(continues on next page)

(continued from previous page)

```

        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);
}

```

## API

Display Driver HAL interface header file

## Typedefs

**typedef struct *lv\_disp\_drv\_t* lv\_disp\_drv\_t**  
Display Driver structure to be registered by HAL

**typedef struct *lv\_disp\_t* lv\_disp\_t**  
Display structure. *lv\_disp\_drv\_t* is the first member of the structure.

## 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 LittlevGL 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 LittlevGL 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\_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 uint32_t flushing

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(). LittlevGL 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.

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 LittlevGL. 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 (*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. *lv\_disp\_drv\_t* is 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

### Input device interface

#### 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 LittlevGL 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 are able to 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 = my_input_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 where by 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 = my_input_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*/
}

```

## 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}, ...}`

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

...

bool button_read(lv_indev_drv_t * drv, lv_indev_data_t*data){
    static uint32_t last_btn = 0;    /*Store the last pressed button*/
    int btn_pr = my_btn_read();      /*Get the ID (0,1,2...) of the pressed button*/
    if(btn_pr >= 0) {                 /*Is there a button press? (E.g. -1 indicated no_
    ↪button was pressed)*/
        last_btn = btn_pr;           /*Save the ID of the pressed button*/
        data->state = LV_INDEV_STATE_PR; /*Set the pressed state*/
    } else {
        data->state = LV_INDEV_STATE_REL; /*Set the released state*/
    }

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

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

## 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 gives the opportunity to make 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.

## API

Input Device HAL interface layer header file

## Typedefs

```
typedef uint8_t lv_indev_type_t
```

**typedef** uint8\_t **lv\_indev\_state\_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**

## 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 and 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 device 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**

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**

**uint8\_t drag\_limit\_out**

**uint8\_t drag\_in\_prog**

**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
```

### Tick interface

The LittlevGL needs a system tick to know the elapsed time for animation and other task.

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

`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 operation system (e.g. on Raspberry) `lv_tick_inc` can be called in a thread:

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

### 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()` )

## Task Handler

To handle the tasks of LittlevGL 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.

## 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 these lines to your input device read function if a press 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 press event*/
```

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

## Operating system and interrupts

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

However, in the following case it's valid to call LittlevGL related functions:

- In *events*. Learn more in *Events*.
- In *lv\_tasks*. Learn more in *Tasks*.

## Tasks and threads

If you need to use real tasks or threads you need a mutex which should be taken before the call of `lv_task_handler` and released after it. In addition, you have to use to that mutex in other tasks and threads around every LittlevGL (`lv_...`) related function call and code. This way you can use LittlevGL in a real multitasking environment. Just use a mutex to avoid the concurrent calling of LittlevGL functions.

## Interrupts

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

## Logging

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

### 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 higher level than the set log level will be logged too. E.g. if you `LV_LOG_LEVEL_WARN`, *errors* will be also logged.

## 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`.

## 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()`.

For example:

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

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

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

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

...

lv_log_register_print(my_log_cb);
```

## Add logs

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

## 3.15.3 Overview

### Objects

In the LittlevGL 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.



## Object attributes

### Basic attributes

The objects have basic attributes which are common independently from their type:

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

You can set/get this 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 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`)

## 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 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*/

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

- a pointer the parent object. To create a screen give *NULL* as parent.
- optionally a pointer to an other object with the same type to copy it. Can be *NULL* to not copy an other object.

Independently from the object type a common variable type `lv_obj_t` is used. This pointer can be used later 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 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. is you want to delete the parent of an object in `LV_EVENT_DELETE` signal.

You can delete only the children of an object but leave the object itself “alive”:

```
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);
```

Always there is an active screen on display. By default, the library creates and loads one. To get the currently active screen use the `lv_scr_act()` function to load 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 *default display*. The *default screen* is the lastly registered screen with `lv_disp_drv_register` (if there is only screen then that one) or you can explicitly selected display with `lv_disp_set_default(display)`. `lv_scr_act()` and `lv_scr_load()` operate on the currently default screen.

Visit *Multi display support* to learn more.

## Layers

### Order of creation

The earlier created object (and its children) will be drawn earlier (nearer to the background). In other words, the lastly created object will be on the top among its siblings. It is very important, the order is calculated among the objects on the same level (“siblings”).

Layers can be added easily by creating 2 objects (which can be transparent). Firstly 'A' and secondly 'B'. 'A' and every object on it will be in the background and can be covered by 'B' and its children.



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

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

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

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

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

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

### 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 LittlevGL will

automatically bring the object to the foreground. It works similarly to the windows on PC. When a window in the background is clicked it will come to the foreground automatically.

- Use `lv_obj_move_foreground(obj)` and `lv_obj_move_background(obj)` to explicitly tell the library to bring an object to the foreground or move to the background.
- When `lv_obj_set_parent(obj, new_parent)` is used `obj` will be on the foreground on the new parent.

## Top and sys layer

There are two special layers called `layer_top` and `layer_sys`. Both of them is visible and the same on all screens of a display. `layer_top` is on top of “normal screen” and `layer_sys` is on top of `layer_top` too.

`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);
```

`layer_sys` is used by LittlevGL. For example, it places the mouse cursor there to be sure it’s always visible.

## Events

In LittlevGL events are triggered if something happens which might be interesting to the user. For example an object

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

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

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

...

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

        case LV_EVENT_SHORT_CLICKED:
            printf("Short clicked\n");
            break;

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

        case LV_EVENT_LONG_PRESSED:
```

(continues on next page)

(continued from previous page)

```

        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*.

## Event types

The following event types exist:

### Generic events

Any object can receive these events independently from their type. I.e. these events are sent to Buttons, Labels, Sliders, etc.

### Input device related

Sent when an object is pressed, released, etc by the user. They are used for *Keypad*, *Encoder* and *Button* input devices as well not only for *Pointers*. 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** Still pressing but slid from the objects
- **LV\_EVENT\_SHORT\_CLICKED** Released before `LV_INDEV_LONG_PRESS_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_REPEAT_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.

## Pointer related

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”)

## Keypad and encoder related

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.

To see exactly which events are used by an object type see the particular *Object type’s documentation*.

## Custom data

Some events might contain custom data. For example **LV\_EVENT\_VALUE\_CHANGED** in some cases tells the new value. For more info 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 its a

- single number then it’s `uint32_t *` or `int32_t *`



- text then `char *` or `const char *`

## Send events manually

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

It can be used for example to manually close a message box by simulating a button press:

```
/*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 ask refresh in a generic way.

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

## Styles

*Styles* are used to set the appearance of the objects. A style is a structure variable with attributes like colors, paddings, opacity, font etc.

There is common style type called `lv_style_t` for every object type.

By setting the fields of the `lv_style_t` variables and assigning to to an object you can influence the appearance of the objects.

---

**Important:** The objects store only a pointer to a style so the style cannot be a local variable which is destroyed after the function exists. **You should use static, global or dynamically allocated variables.**

---

```
lv_style_t style_1;           /*OK! Global variables for styles are fine*/
static lv_style_t style_2;    /*OK! Static variables outside the functions are ↵
↵fine*/
void my_screen_create(void)
{
    static lv_style_t style_3; /*OK! Static variables in the functions are fine*/
    lv_style_t style_4;       /*WRONG! Styles can't be local variables*/

    ...
}
```

## Use the styles

The objects have a *Main style* which determines the appearance of their background or main part. However, some object types have additional styles too.

Some object has only one style. E.g.

- Label
- Image
- Line, etc

For example, a slider has 3 styles:

- Background (main style)
- Indicator
- Know

Every object type has its own style set/get functions. For example

```
const lv_style_t * btn_style = lv_btn_get_style(btn, LV_BTN_STYLE_REL);
lv_btn_set_style(btn, LV_BTN_STYLE_REL, &new_style);
```

To see the styles supported by an object type (*LV\_<OBJ\_TYPE>STYLE<STYLE\_TYPE>*) check the documentation of the particular *Object type*.

If you **modify a style which is already used** by one or more objects then the objects have to be notified about the style is changed. You have two options to do that:

```
/*Notify an object about its style is modified*/
void lv_obj_refresh_style(lv_obj_t * obj);

/*Notify all objects with a given style. (NULL to notify all objects)*/
void lv_obj_report_style_mod(void * style);
```

`lv_obj_report_style_mod` can refresh only the *Main styles*.

## Inherit styles

If the *Main style* of an object is **NULL** then its style will be inherited from its parent's style. It makes easier to create a consistent design. Don't forget a style describes a lot of properties at the same time. So for example, if you set a button's style and create a label on it with **NULL** style then the label will be rendered according to the button's style. In other words, the button makes sure its children will look well on it.

Setting the **glass** style property will prevent inheriting that style. You should use it if the style is transparent so that its children use colors and others from its grandparent.

## Style properties

A style has 5 main parts: common, body, text, image and line. An object will use those fields which are relevant to it. For example, *Lines* don't care about the *letter\_space*. To see which fields are used by an object type see their *Documentation*.

The fields of a style structure are the followings:

### Common properties

- **glass** 1: Do not inherit this style

### Body style properties

Used by the rectangle-like objects

- **body.main\_color** Main color (top color)
- **body.grad\_color** Gradient color (bottom color)

- **body.radius** Corner radius. (set to `LV_RADIUS_CIRCLE` to draw circle)
- **body.opa** Opacity (0..255 or `LV_OPA_TRANSP`, `LV_OPA_10`, `LV_OPA_20` ... `LV_OPA_COVER`)
- **body.border.color** Border color
- **body.border.width** Border width
- **body.border.part** Border parts (`LV_BORDER_LEFT/RIGHT/TOP/BOTTOM/FULL` or 'OR'ed values)
- **body.border.opa** Border opacity (0..255 or `LV_OPA_TRANSP`, `LV_OPA_10`, `LV_OPA_20` ... `LV_OPA_COVER`)
- **body.shadow.color** Shadow color
- **body.shadow.width** Shadow width
- **body.shadow.type** Shadow type (`LV_SHADOW_BOTTOM/FULL`)
- **body.padding.top** Top padding
- **body.padding.bottom** Bottom padding
- **body.padding.left** Left padding
- **body.padding.right** Right padding
- **body.padding.inner** Inner padding (between content elements or children)

### Text style properties

Used by the objects which show texts

- **text.color** Text color
- **text.sel\_color** Selected text color
- **text.font** Pointer to a font
- **text.opa** Text opacity (0..255 or `LV_OPA_TRANSP`, `LV_OPA_10`, `LV_OPA_20` ... `LV_OPA_COVER*`)
- **text.letter\_space** Letter space
- **text.line\_space** Line space

### Image style properties

Used by image-like objects or icons on objects

- **image.color** Color for image re-coloring based on the pixels brightness
- **image.intense** Re-color intensity (0..255 or `LV_OPA_TRANSP`, `LV_OPA_10`, `LV_OPA_20` ... `LV_OPA_COVER`)
- **image.opa** Image opacity (0..255 or `LV_OPA_TRANSP`, `LV_OPA_10`, `LV_OPA_20` ... `LV_OPA_COVER`)

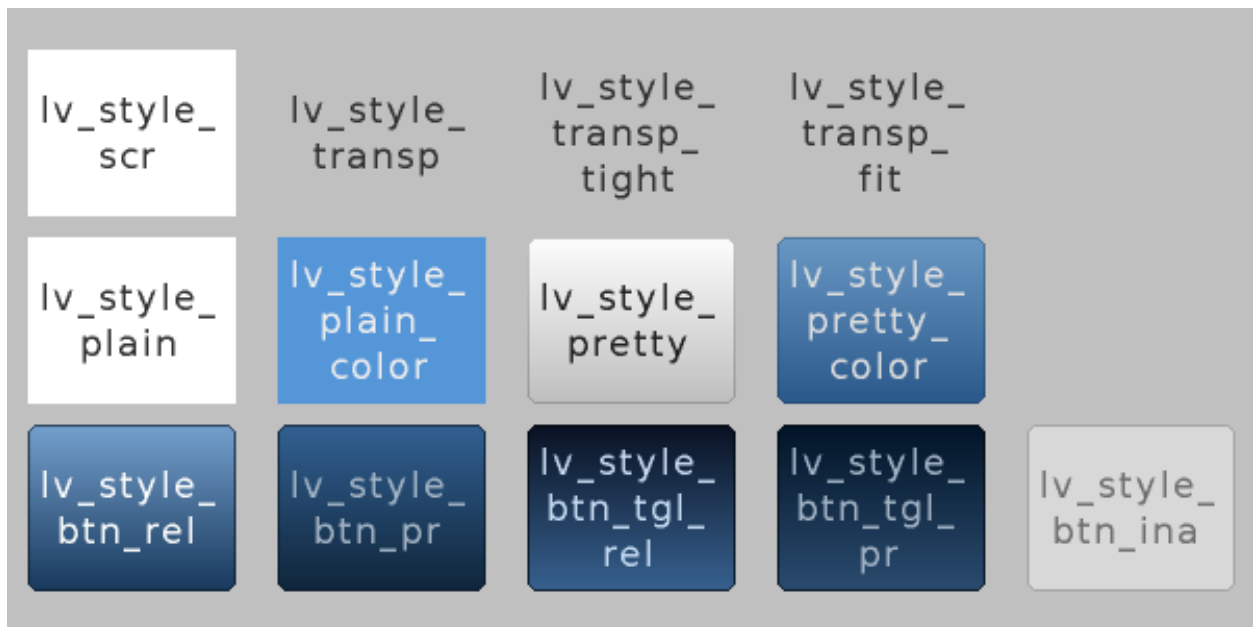
## Line style properties

Used by objects containing lines or line-like elements

- **line.color** Line color
- **line.width** Line width
- **line.opa** Line opacity (0..255 or `LV_OPA_TRANSP`, `LV_OPA_10`, `LV_OPA_20` ... `LV_OPA_COVER`)

## Built-in styles

There are several built-in styles in the library:



As you can see there is a style for screens, for buttons, plain and pretty styles and transparent styles as well.

The `lv_style_transp`, `lv_style_transp_fit` and `lv_style_transp_tight` differ only in paddings: for `lv_style_transp_tight` all paddings are zero, for `lv_style_transp_fit` only hor and ver paddings are zero but has inner padding.

---

**Important:** Transparent built-in styles have `glass = 1` by default which means these styles (e.g. their colors) won't be inherited by children.

---

The built in styles are global `lv_style_t` variables. You can use them like:

```
lv_btn_set_style(obj, LV_BTN_STYLE_REL, &lv_style_btn_rel)
```

## Create new styles

You can modify the built-in styles or you can create new styles.

When creating new styles it's recommended to first copy a built-in style with `lv_style_copy(&dest_style, &src_style)` to be sure all fields are initialized with a proper value.

Do not forget the created style should be **static** or global. For example:

```
static lv_style_t my_red_style;
lv_style_copy(&my_red_style, &lv_style_plain);
my_red_style.body.main_color = LV_COLOR_RED;
my_red_style.body.grad_color = LV_COLOR_RED;
```

## Style animations

You change the styles with animations using `lv_style_anim_...()` function. Two styles are required to represent the *start* and *end* state, and a third style which will be animated. Here is an example to show how it works.

```
lv_anim_t a;
lv_style_anim_init(&a);                                /*A basic
↳initialization*/
lv_style_anim_set_styles(&a, &style_to_anim, &style_start, &style_end); /*Set the
↳styles to use*/
lv_style_anim_set_time(&a, duration, delay);           /*Set the
↳duration and delay*/
lv_style_anim_create(&a);                               /*Create the
↳animation*/
```

To see the whole API of style animations see `lv_core/lv_style.h`.

Here you can learn more about the *Animations*.

## Style example

The example below demonstrates the usage of styles.



```
/*Create a style*/
static lv_style_t style1;
lv_style_copy(&style1, &lv_style_plain); /*Copy a built-in style to initialize the
↳new style*/
style1.body.main_color = LV_COLOR_WHITE;
style1.body.grad_color = LV_COLOR_BLUE;
style1.body.radius = 10;
```

(continues on next page)

(continued from previous page)

```

style1.body.border.color = LV_COLOR_GRAY;
style1.body.border.width = 2;
style1.body.border.opa = LV_OPA_50;
style1.body.padding.left = 5;           /*Horizontal padding, used by the bar_
↪indicator below*/
style1.body.padding.right = 5;
style1.body.padding.top = 5;           /*Vertical padding, used by the bar indicator_
↪below*/
style1.body.padding.bottom = 5;
style1.text.color = LV_COLOR_RED;

/*Create a simple object*/
lv_obj_t *obj1 = lv_obj_create(lv_scr_act(), NULL);
lv_obj_set_style(obj1, &style1);       /*Apply the created style*/
lv_obj_set_pos(obj1, 20, 20);         /*Set the position*/

/*Create a label on the object. The label's style is NULL by default*/
lv_obj_t *label = lv_label_create(obj1, NULL);
lv_obj_align(label, NULL, LV_ALIGN_CENTER, 0, 0); /*Align the label to the_
↪middle*/

/*Create a bar*/
lv_obj_t *bar1 = lv_bar_create(lv_scr_act(), NULL);
lv_bar_set_style(bar1, LV_BAR_STYLE_INDIC, &style1); /*Modify the indicator's_
↪style*/
lv_bar_set_value(bar1, 70);           /*Set the bar's value*/

```

## Themes

To create styles for your GUI is challenging because you need a deeper understanding of the library and you need to have some design skills. In addition, it takes a lot of time to create so many styles.

To speed up the design part themes are introduced. A theme is a style collection which contains the required styles for every object type. For example 5 styles for buttons to describe their 5 possible states. Check the [Existing themes](#) or try some in the [Live demo](#) section.

To be more specific a theme is a structure variable which contains a lot of `lv_style_t *` fields. For buttons:

```

theme.btn.rel    /*Released button style*/
theme.btn.pr     /*Pressed button style*/
theme.btn.tgl_rel /*Toggled released button style*/
theme.btn.tgl_pr  /*Toggled pressed button style*/
theme.btn.ina    /*Inactive button style*/

```

A theme can be initialized by: `lv_theme_<name>_init(hue, font)`. Where `hue` is a Hue value from [HSV color space](#) (0..360) and `font` is the font applied in the theme (NULL to use the `LV_FONT_DEFAULT`)

When a theme is initialized its styles can be used like this:



```

/*Create a default slider*/
lv_obj_t *slider = lv_slider_create(lv_scr_act(), NULL);
lv_slider_set_value(slider, 70);
lv_obj_set_pos(slider, 10, 10);

/*Initialize the alien theme with a reddish hue*/
lv_theme_t *th = lv_theme_alien_init(10, NULL);

/*Create a new slider and apply the themes styles*/
slider = lv_slider_create(lv_scr_act(), NULL);
lv_slider_set_value(slider, 70);
lv_obj_set_pos(slider, 10, 50);
lv_slider_set_style(slider, LV_SLIDER_STYLE_BG, th->slider.bg);
lv_slider_set_style(slider, LV_SLIDER_STYLE_INDIC, th->slider.indic);
lv_slider_set_style(slider, LV_SLIDER_STYLE_KNOB, th->slider.knob);
    
```

You can ask the library to automatically apply the styles from a theme when you create new objects. To do this use `lv_theme_set_current(th)`;

```

/*Initialize the alien theme with a reddish hue*/
lv_theme_t *th = lv_theme_alien_init(10, NULL);
lv_theme_set_current(th);

/*Create a slider. It will use the style from teh current theme.*/
slider = lv_slider_create(lv_scr_act(), NULL);
    
```

Themes can be enabled or disabled one by one in `lv_conf.h`.

## Live update

By default if `lv_theme_set_current(th)` is called again it won't refresh the styles of the existing objects. To enable live update of themes enable `LV_THEME_LIVE_UPDATE` in `lv_conf.h`.

Live update will update only those objects whose style are from the theme, i.e. created after the first call of `lv_theme_set_current(th)` or the styles were set manually

## Input devices

Input devices in general means:

- Pointer-like input devices like touchpad or mouse
- Keypads like a normal keyboard or simple numpad
- 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

---

## 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*/
```

## Keypad and encoder

You can fully control the user interface without touchpad or mouse using a keypad or encoder(s). it works similarly when you press the *TAB* key on 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. Or 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 devices 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

With keypads, there are plenty of keys so it's easy to navigate among the objects and edit them. However, the encoders have a very limited number of “keys”. To effectively support encoders too *Navigate* and *Edit* is created.

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 used normally 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 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 the focused object

To visually highlight the focused element its **Main style** will be updated. By default, some orange color is mixed to the original colors of the style. A new style modifier callback be set by **lv\_group\_set\_style\_mod\_cb(g, my\_style\_mod\_cb)**. A style modifier callback receives a pointer to a caller group and pointer to a style to modify. The default style modifier looks like this (slightly simplified):

```
static void default_style_mod_cb(lv_group_t * group, lv_style_t * style)
{
    /*Make the bodies a little bit orange*/
    style->body.border.opa    = LV_OPA_COVER;
    style->body.border.color  = LV_COLOR_ORANGE;
    style->body.border.width  = LV_DPI / 20;

    style->body.main_color    = lv_color_mix(style->body.main_color, LV_COLOR_ORANGE,
    ↪ LV_OPA_70);
    style->body.grad_color    = lv_color_mix(style->body.grad_color, LV_COLOR_ORANGE,
    ↪ LV_OPA_70);
    style->body.shadow.color  = lv_color_mix(style->body.shadow.color, LV_COLOR_ORANGE,
    ↪ LV_OPA_60);
}
```

(continues on next page)

(continued from previous page)

```

/*Recolor text*/
style->text.color = lv_color_mix(style->text.color, LV_COLOR_ORANGE, LV_OPA_70);

/*Add some recolor to the images*/
if(style->image.intense < LV_OPA_MIN) {
    style->image.color = LV_COLOR_ORANGE;
    style->image.intense = LV_OPA_40;
}
}

```

This style modifier callback is used for keypads and encoder in *Navigate* mode. For the *Edit* mode and other callback is used which can be set with `lv_group_set_style_mod_edit_cb()`. By default, it has a greenish color.

## Live demo

Try this [Live demo](#) to see how a group and touchpad-less navigation works in the practice.

## 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)

Reset one or all input devices

#### Parameters

- **indev**: pointer to an input device to reset or NULL to reset all of them

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

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

void **lv\_indev\_wait\_release**(*lv\_indev\_t* \**indev*)

Do nothing until the next release

#### Parameters

- **indev**: pointer to an input device

*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

*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

## 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\_style\_mod\_cb**(*lv\_group\_t \*group*, *lv\_group\_style\_mod\_cb\_t style\_mod\_cb*)  
Set a function for a group which will modify the object's style if it is in focus

**Parameters**

- **group**: pointer to a group
- **style\_mod\_cb**: the style modifier function pointer

void **lv\_group\_set\_style\_mod\_edit\_cb**(*lv\_group\_t \*group*, *lv\_group\_style\_mod\_cb\_t style\_mod\_edit\_cb*)  
Set a function for a group which will modify the object's style if it is in focus in edit mode

**Parameters**

- **group**: pointer to a group
- **style\_mod\_edit\_cb**: the style modifier function pointer

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 focussed 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\_style\_t \***lv\_group\_mod\_style**(*lv\_group\_t \*group, const lv\_style\_t \*style*)

Modify a style with the set 'style\_mod' function. The input style remains unchanged.

**Return** a copy of the input style but modified with the 'style\_mod' function

**Parameters**

- **group**: pointer to group
- **style**: pointer to a style to modify

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\_style\_mod\_cb\_t **lv\_group\_get\_style\_mod\_cb**(*const lv\_group\_t \*group*)

Get a the style modifier function of a group

**Return** pointer to the style modifier function

**Parameters**

- **group**: pointer to a group

*lv\_group\_style\_mod\_cb\_t* **lv\_group\_get\_style\_mod\_edit\_cb**(const *lv\_group\_t* \*group)

Get a the style modifier function of a group in edit mode

**Return** pointer to the style modifier function

**Parameters**

- **group**: pointer to a 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

void **lv\_group\_report\_style\_mod**(*lv\_group\_t* \*group)

Notify the group that current theme changed and style modification callbacks need to be refreshed.

**Parameters**

- **group**: pointer to group. If NULL then all groups are notified.

**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_style_mod_cb_t` **style\_mod\_cb**  
 A function to modifies the style of the focused object

`lv_group_style_mod_cb_t` **style\_mod\_edit\_cb**  
 A function which modifies the style of the edited object

`lv_group_focus_cb_t` **focus\_cb**  
 A function to call when a new object is focused (optional)

`lv_style_t` **style\_tmp**  
 Stores the modified style of the focused object

`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.

## Displays

---

**Important:** The basic concept of *Display* in LittlevGL is explained in the [Porting](/porting/display) section. So before reading further, please read that section first.

---

In LittlevGL you can have multiple displays each with its own drivers and objects.

Creating more displays is easy: just initialize display buffers and register the drivers for every display. When you create the UI use `lv_disp_set_deafult disp` to tell the library to which display create the object.

But in which cases can you use the 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 own VNC driver)
- Have a large TFT display and a small monochrome display.
- Have some smaller and simple displays in a large instrument or technology
- Have two large TFT displays: one for a customer and one for the shop assistant

## Using only one display

Using more displays can be useful but in most of the cases, it's not required. Therefore the whole concept of multi-displays is completely hidden if you register only one display. By default, the lastly created (the only one) display is used as default.

`lv_scr_act()`, `lv_scr_load(scr)`, `lv_layer_top()`, `lv_layer_sys()`, `LV_HOR_RES` and `LV_VER_RES` are always applied on the lastly created (default) screen. If you pass `NULL` as `disp` parameter to display related function usually the default display will be used. E.g. `lv_disp_trig_activity(NULL)` will trigger a user activity on the default screen. (See below in *Inactivity*).

### Mirror display

To mirror the image of display to an other display you don't really need to use the multi-display support. Just transfer the buffer received in `drv.flush_cb` to an other display too.

### Split image

You can create a larger display from more smaller ones. You do it like this:

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,

### Screens

Every display has it each set of [Screens](#) and the object on the screens.

Screens can be considered the highest level containers which have no parent. The screen's size is always equal to its display's 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 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)`. The get active screen use `lv_scr_act()`. These functions works on the default display to specify which display you mean use `lv_disp_get_scr_act(disp)` and `lv_disp_load_scr(disp, scr)`.

Screens can be deleted with `lv_obj_del(scr)` but be sure to not delete currently loaded screen.

### Opaque screen

Usually, the opacity of the screen is `LV_OPA_COVER` to provide a solid, folly covering background for its children. However, in some special case, you might want a transparent screen. For example, if you have a video player which renders the video frames on a layer but on an other layer you want to create an OSD menu (over the video) using LittlevGL. In this the style of the screen you should have `body.opa = LV_OPA_TRANSP` or `image.opa = LV_OPA_TRANSP` (or other `LV_OPA...` values) to make the screen opaque. To properly handle the screens opacity `LV_COLOR_SCREEN_TRANSP` needs to be enabled. Not that, it works on 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.

## 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(displ)`. If `NULL` is passed the overall smallest inactivity time will be returned from all displays.

You can manually trigger an activity using `lv_disp_trig_activity(displ)`. If `displ` is `NULL` the default screen will be used.

### 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 (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 an other. 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;
```

(continues on next page)

(continued from previous page)

```
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 c32;
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\_color\_hex3(0x28C)**.

Mixing two colors is possible with **mixed\_color = lv\_color\_mix(color1, color2, ratio)**. Ratio 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 fully 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 fully covers
- **LV\_OPA\_COVER** Value: 255, means the color fully covers

You can also use the **LV\_OPA\_\*** defines in **lv\_color\_mix()** as *ratio*.

## Built-in colors

The color module defines the most basic colors:

-  #000000 LV\_COLOR\_BLACK
-  #808080 LV\_COLOR\_GRAY
-  #c0c0c0 LV\_COLOR\_SILVER
-  #ff0000 LV\_COLOR\_RED
-  #800000 LV\_COLOR\_MARRON
-  #00ff00 LV\_COLOR\_LIME
-  #008000 LV\_COLOR\_GREEN
-  #808000 LV\_COLOR\_OLIVE
-  #0000ff LV\_COLOR\_BLUE
-  #000080 LV\_COLOR\_NAVY
-  #008080 LV\_COLOR\_TAIL
-  #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.

## 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

*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

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)

**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
typedef uint8_t lv_opa_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)
union lv_color1_t
```

#### Public Members

```
uint8_t blue
uint8_t green
uint8_t red
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
```



## Fonts

In LittlevGL 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 it in style's `text.font` field. For example:

```
my_style.text.font = &lv_font_roboto_28; /*Set a larger font*/
```

The fonts have a **bpp (Bit-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. E.g. *bpp* = 4 makes the font ~4 times greater compared to *bpp* = 1.

## Unicode support

LittlevGL supports **UTF-8** encoded Unicode characters. You need to configure your editor to save your code/text as UTF-8 (usually this the default) and be sure `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.

## 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_ROBOTO_12` 12 px
- `LV_FONT_ROBOTO_16` 16 px
- `LV_FONT_ROBOTO_22` 22 px
- `LV_FONT_ROBOTO_28` 28 px

The built-in fonts are **global variables** with names like `lv_font_roboto_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 [Roboto](#) font.

In addition to the ASCII rangle, 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_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_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_BLUETOOTH

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);
```

## Add new font

There are several ways to add a new font to your project:

1. The most simple way is to use the [Online font converter](#). Just set the parameters, click the *Convert* button, copy the font to your project and use it.
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 to font globally available add them to `LV_FONT_CUSTOM_DECLARE` in `lv_conf.h`.

## Add new symbols

The built-in symbols are created from [FontAwesome](#) font. To add new symbols from the [FontAwesome](#) font do the following steps:

1. Search symbol on <https://fontawesome.com>. For example the [USB symbol](#)
2. Open the [Online font converter](#) add `FontAwesome.ttf` and add the Unicode ID of the symbol to the range field. E.g. `0xf287` for the USB symbol. More symbols can be enumerated with `.`
3. Convert the font and copy it to your project.
4. 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`.
5. Create a `define` from the UTF8 values: `#define MY_USB_SYMBOL "\xEF\x8A\x87"`
6. Use the symbol as the built-in symbols. `lv_label_set_text(label, MY_USB_SYMBOL)`

## Add a new font engine

LittlevGL's font interface is designed to be very flexible. You don't need to use LittlevGL'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 need them.

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 glyphs*/
my_font.get_glyph_bitmap = my_get_glyph_bitmap_cb;     /*Set a callback to get bitmap of_
↳a glyph*/
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;                                    /*Bit 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*/
}

```

## Images

An image can be a file or variable which stores the image itself and some metadata.

## 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 has `lv_img_dsc_t` type with the following fields:

- **header**
  - *cf* Color format. See *below*
  - *w* width in pixels ( $\leq 2048$ )
  - *h* height in pixels ( $\leq 2048$ )
  - *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

## Files

To deal with files you need to add a *Drive* to LittlevGL. In short, a *Drive* a collection of functions (*open*, *read*, *close*, etc) registered in LittlevGL to make file operation. You can add an interface to a standard file system (FAT32 on SD card) or you create your own simple file system to read data from an SPI Flash memory. In every cases, a *Drive* is just an abstraction to read and/or write data to a memory. See the *File system* section to learn more.

## Color formats

Various built-in color formats are supported:

- **LV\_IMG\_CF\_TRUE\_COLOR** Simply store the RGB colors
- **LV\_IMG\_CF\_TRUE\_COLOR\_ALPHA** Store the RGB colors but add an Alpha byte too for every pixel
- **LV\_IMG\_CF\_TRUE\_COLOR\_CHROMA\_KEYED** Store the RGB 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** Use palette with 2, 4, 16 or 256 colors and store each pixel on 1, 2, 4 or 8 bit
- **LV\_IMG\_CF\_ALPHA\_1/2/4/8BIT** Store only the Alpha value on 1, 2, 4 or 8 bits. Draw the pixels `style.image.color` and the set opacity.

The bytes of the *True color* 32 bit images are stored in the following order

- 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 *Image decoder* needs to be used to decode the image.

- **`LV_IMG_CF_RAW`** A raw image e.g. a PNG or JPG image
- **`LV_IMG_CF_RAW_ALPHA`** Indicate that the image has alpha, and an Alpha byte is added for every pixel
- **`LV_IMG_CF_RAW_CHROME_KEYED`** Indicate that the image is chrome keyed as described in `LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED` above.

## Add and use images

You can add images to LittlevGL in two ways:

- using the online converter
- manually create images

### Online converter

The Online Image Converter is available here: <https://littlevgl.com/image-to-c-array>

You need to select a *BMP*, *PNG* or *JPG* image, give it a name, select the *Color format*, select the type (file or variable) and hit the *Convert* button and the result file be download.

In the converter C arrays (variables) the image for all the Color depths (1, 8, 16 or 32) are included and the used image will be selected in compile time based on `LV_COLOR_DEPTH` in *lv\_conf.h*.

IN case of files you need to tell which 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 calculate an image run-time you can craft an image variable to display it. 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`.

An other option to create image run-time is to use the *Canvas* object.

## Use images

The most simple way to use an Image in LittlevGL 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 icon in the file where you want to use it.

## Image decoder

As you can see in the *Color formats* section LittlevGL supports several built image formats. However, it doesn't support for example PNG or JPG out of the box. To handle non-built-in image formats you need to use external libraries and attach them to LittlevGL via the *Image decoder* interface.

The image decoder consists of 4 images:

- **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 the 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 find one which is able to open the image, i.e. know that format.

The `LV_IMG_CF_TURE_COLOR...`, `LV_IMG_INDEXED...` and `LV_IMG_ALPHA...` formats are known 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**, **Raw with chrome keyed** format. It will just take the every bytes of selected image and write them as image data. `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 covering image, use alpha channel or use chroma keying.

After decoding, the *raw* formats are considered *True color*. In other words the image decoder should decode the *Raw* images to *True color* according to the format described in `[#color-formats]` (Color formats) section.

If you want to create a really 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

For example, if you want LittlevGL to “understand” PNG images you need to create a new image decoder and set some functions to open/close the PNG files. It should look 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;

    ...

    header->cf = LV_IMG_CF_RAW_ALPHA;
    header->w = width;
    header->h = height;
}

/**
 * Open a PNG image and return the decoded image
 * @param decoder pointer to the decoder where this function belongs
```

(continues on next page)



(continued from previous page)

```

* @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` the `read_line` function will be
    ↳called to get the image data line-by-line*/
    dsc->img_data = my_png_decoder(src);

    /*Change the color format if required. For PNG usually 'Raw' is fine*/
    dsc->header.cf = LV_IMG_CF_...

    /*Call a built in decoder function if required. It's not required if `my_png_
    ↳decoder` opened the image in true color format.*/
    lv_res_t res = lv_img_decoder_built_in_open(decoder, dsc);

    return res;
}

/**
 * Decode `len` pixels starting from the given `x`, `y` coordinates and store them in
    ↳`buf`.
 * Required only if the "open" function can't open the whole decoded pixel array.
    ↳(`dsc->img_data == NULL`)
 * @param decoder pointer to the decoder the function associated with
 * @param dsc pointer to decoder descriptor
 * @param x start x coordinate
 * @param y start y coordinate
 * @param len number of pixels to decode
 * @param buf a buffer to store the decoded pixels
 * @return LV_RES_OK: ok; LV_RES_INV: failed
 */
lv_res_t decoder_built_in_read_line(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t
    ↳* dsc, lv_coord_t x,
    lv_coord_t y, lv_coord_t len, uint8_
    ↳t * buf)
{
    /*With PNG it's usually not required*/

    /*Copy `len` pixels from `x` and `y` coordinates in True color format to `buf` */
}

/**
 * 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*/

```

(continues on next page)

(continued from previous page)

```

/*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**. It's 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 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

LittlevGL will use the registered image decoder automatically 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);
}

```

### 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 effective. Therefore LittlevGL caches a given number of images. Caching means some images will be left open hence LittlevGL can quickly access them from **dsc->img\_data** instead of decoding the again.

### 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 lastly used image will be left open. The size of cache can be changed in run-time with **lv\_img\_cache\_set\_size(entry\_num)**

## Value of images

If you use more images then the cache size LittlevGL can't cache all of the images. Instead, if a new image needs to be opened but there is no place in the cache the library will close an image. To decide which image to close LittlevGL measured how much did it take to open the image. Images which more time consuming to open are considered more valuable and LittlevGL tries to cache them longer. 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 to the image. (Leave it unchanged to let LittlevGL 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 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 change `my_png->data` you need to notify LittlevGL to cache the image again. To do this use `lv_img_cache_invalidate_src(&my_png)`. If `NULL` is passed as parameter the whole cache will be cleaned.

## API

### Image decoder

#### Typedefs

```
typedef uint8_t lv_img_src_t
```

```
typedef uint8_t lv_img_cf_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)

```
enum [anonymous]
    Values:
```

**LV\_IMG\_CF\_UNKNOWN** = 0

#### **LV\_IMG\_CF\_RAW**

Contains the file as it is. Needs custom decoder function

#### **LV\_IMG\_CF\_RAW\_ALPHA**

Contains the file as it is. The image has alpha. Needs custom decoder function

#### **LV\_IMG\_CF\_RAW\_CHROMA\_KEYED**

Contains the file as it is. The image is chroma keyed. Needs custom decoder function

#### **LV\_IMG\_CF\_TRUE\_COLOR**

Color format and depth should match with LV\_COLOR settings

#### **LV\_IMG\_CF\_TRUE\_COLOR\_ALPHA**

Same as LV\_IMG\_CF\_TRUE\_COLOR but every pixel has an alpha byte

#### **LV\_IMG\_CF\_TRUE\_COLOR\_CHROMA\_KEYED**

Same as LV\_IMG\_CF\_TRUE\_COLOR but LV\_COLOR\_TRANSP pixels will be transparent

#### **LV\_IMG\_CF\_INDEXED\_1BIT**

Can have 2 different colors in a palette (always chroma keyed)

#### **LV\_IMG\_CF\_INDEXED\_2BIT**

Can have 4 different colors in a palette (always chroma keyed)

#### **LV\_IMG\_CF\_INDEXED\_4BIT**

Can have 16 different colors in a palette (always chroma keyed)

#### **LV\_IMG\_CF\_INDEXED\_8BIT**

Can have 256 different colors in a palette (always chroma keyed)

#### **LV\_IMG\_CF\_ALPHA\_1BIT**

Can have one color and it can be drawn or not

#### **LV\_IMG\_CF\_ALPHA\_2BIT**

Can have one color but 4 different alpha value

#### **LV\_IMG\_CF\_ALPHA\_4BIT**

Can have one color but 16 different alpha value

#### **LV\_IMG\_CF\_ALPHA\_8BIT**

Can have one color but 256 different alpha value

## 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, const
                             lv_style_t *style)
```

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

```
struct lv_img_header_t
```

*#include <lv\_img\_decoder.h>* LittlevGL image header

### Public Members

uint32\_t **cf**

uint32\_t **always\_zero**

uint32\_t **reserved**

uint32\_t **w**

uint32\_t **h**

```
struct lv_img_dsc_t
```

*#include <lv\_img\_decoder.h>* Image header it is compatible with the result from image converter utility

### Public Members

lv\_img\_header\_t **header**

uint32\_t **data\_size**

const uint8\_t \***data**

```
struct _lv_img_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

**const** lv\_style\_t \***style**

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*, **const** lv\_style\_t \**style*)

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

### File system

LittlevGL has File system abstraction module which enables to attache any type of file system. The file system are identified by a letter. For example if the SD card is associated with letter 'S' a file can be reached like "S:path/to/file.txt.

### Add a driver

To add a driver an `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 uppercased letter to identify teh_
↳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 */
```

(continues on next page)

(continued from previous page)

```
drv.rename_cb = my_size_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_size_cb;       /*Callback to tell free space on the drive
↳*/

drv.user_data = my_user_data;         /*Any custom data if required*/

lv_fs_drv_register(&drv);             /*Finally register the drive*/
```

Any of the callbacks can be **NULL** to indicate that operation is not supported.

## Use drivers for images

Image objects can be open from files too (besides variables stored in the flash)

To initialize the for images the following callbacks are required:

- open
- close
- read
- seek
- tell

## 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 filesystem 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 initied with the corresponding function pointers. The data will be copied so the variable can be local.

bool **lv\_fs\_is\_ready**(char *letter*)  
Test if a drive is rady 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

```

## 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 LittlevGL. For example `lv_obj_set_x(obj, value)` or `lv_obj_set_width(obj, value)`

## Create an animation

To create an animation an `lv_anim_t` variable has to be initialized and configured with `lv_anim_set_...()` functions.

```
lv_anim_t a;
lv_anim_set_exec_cb(&a, btn1, lv_obj_set_x);    /*Set the animator function and
↪variable to animate*/
lv_anim_set_time(&a, duration, delay);
lv_anim_set_values(&a, start, end);             /*Set start and end values. E.g. 0,
↪150*/
lv_anim_set_path_cb(&a, lv_anim_path_linear);    /*Set path from `lv_anim_path_...`
↪functions or a custom one.*/
lv_anim_set_ready_cb(&a, ready_cb);             /*Set a callback to call then
↪animation is ready. (Optional)*/
lv_anim_set_playback(&a, wait_time);            /*Enable playback of teh animation
↪with `wait_time` delay*/
lv_anim_set_repeat(&a, wait_time);              /*Enable repeat of teh animation with
↪`wait_time` delay. Can be compiled with playback*/

lv_anim_create(&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_create()` will delete the already existing variable-function animations.

## 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 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:

- `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)

## 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.



## Delete animations

You can **delete an animation** by `lv_anim_del(var, func)` by providing the animated variable and its animator function.

## 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** 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** *lv\_anim\_value\_t* (\***lv\_anim\_path\_cb\_t**)(const struct *lv\_anim\_t* \*)  
Get the current value during an animation

**typedef** void (\***lv\_anim\_ready\_cb\_t**)(struct *lv\_anim\_t* \*)  
Callback to call when the animation is ready

**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); lv_anim_create(&a);`

**Parameters**

- **a**: pointer to an `lv_anim_t` variable to initialize

**static void lv\_anim\_set\_exec\_cb**(*lv\_anim\_t* \*a, void \*var, *lv\_anim\_exec\_xcb\_t* exec\_cb)  
Set a variable to animate function to execute on **var**

#### Parameters

- **a**: pointer to an initialized `lv_anim_t` variable
- **var**: pointer to a variable to animate
- **exec\_cb**: a function to execute. LittlevGL's built-in functions can be used. E.g. `lv_obj_set_x`

**static void lv\_anim\_set\_time**(*lv\_anim\_t* \*a, *uint16\_t* duration, *uint16\_t* delay)  
Set the duration and delay of an animation

#### Parameters

- **a**: pointer to an initialized `lv_anim_t` variable
- **duration**: duration of the animation in milliseconds
- **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_var_and_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 LittlevGL is binded to other languages because it's more consistent to have `lv_anim_t *` as first parameter.

#### Parameters

- **a**: pointer to an initialized `lv_anim_t` variable
- **exec\_cb**: a function to execute.

**static void lv\_anim\_set\_path\_cb**(*lv\_anim\_t* \*a, *lv\_anim\_path\_cb\_t* path\_cb)  
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\_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**(*lv\_anim\_t \*a*, uint16\_t *wait\_time*)

Make the animation to play back to when the forward direction is ready

**Parameters**

- **a**: pointer to an initialized `lv_anim_t` variable
- **wait\_time**: time in milliseconds to wait before starting the back direction

**static void lv\_anim\_clear\_playback**(*lv\_anim\_t \*a*)

Disable playback. (Disabled after `lv_anim_init()`)

**Parameters**

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

**static void lv\_anim\_set\_repeat**(*lv\_anim\_t \*a*, uint16\_t *wait\_time*)

Make the animation to start again when ready.

**Parameters**

- **a**: pointer to an initialized `lv_anim_t` variable
- **wait\_time**: time in milliseconds to wait before starting the animation again

**static void lv\_anim\_clear\_repeat**(*lv\_anim\_t \*a*)

Disable repeat. (Disabled after `lv_anim_init()`)

**Parameters**

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

**static void lv\_anim\_set\_user\_data**(*lv\_anim\_t \*a*, lv\_anim\_user\_data\_t *user\_data*)

Set a user specific data for the animation

**Parameters**

- **a**: pointer to an initialized `lv_anim_t` variable
- **user\_data**: the user data

**static lv\_anim\_user\_data\_t lv\_anim\_get\_user\_data**(*lv\_anim\_t \*a*)

Get the user data

**Return** the user data

**Parameters**

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

**static lv\_anim\_user\_data\_t \*lv\_anim\_get\_user\_data\_ptr**(*lv\_anim\_t \*a*)

Get pointer to the user data

**Return** pointer to the user data

**Parameters**

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

**void lv\_anim\_create**(*lv\_anim\_t \*a*)

Create an animation

**Parameters**

- **a**: an initialized 'anim\_t' variable. Not required after call.

**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’

**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 exist 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 makes 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

*lv\_anim\_value\_t* **lv\_anim\_path\_linear**(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\_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\_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\_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\_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\_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\_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

**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\_path\_cb\_t* **path\_cb**

Function to get the steps of animations

*lv\_anim\_ready\_cb\_t* **ready\_cb**

Call it when the animation is ready

int32\_t **start**

Start value

int32\_t **end**

End value

uint16\_t **time**

Animation time in ms

`int16_t` **act\_time**  
Current time in animation. Set to negative to make delay.

`uint16_t` **playback\_pause**  
Wait before play back

`uint16_t` **repeat\_pause**  
Wait before repeat

`lv_anim_user_data_t` **user\_data**  
Custom user data

`uint8_t` **playback**  
When the animation is ready play it back

`uint8_t` **repeat**  
Repeat the animation infinitely

`uint8_t` **playback\_now**  
Play back is in progress

`uint32_t` **has\_run**  
Indicates the animation has run in this round

## Tasks

LittlevGL has a built-in task system. You can register a functions to call them periodically. The tasks are handled and called in `lv_task_handler()` which needs to be called periodically in every few milliseconds. See *Porting* for more information.

The tasks are non-preemptive which means a task can interrupt an other. Therefore you can call any LittlevGL related function in a task.

## 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()` also can be used 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 LittlevGL*/
    if(something_happened) {
        something_happened = false;
        lv_btn_create(lv_scr_act(), NULL);
    }
}
...

```

(continues on next page)

(continued from previous page)

```
static uint32_t user_data = 10;
lv_task_t * task = lv_task_create(my_task, 500, LV_TASK_PRIO_MID, &user_data);
```

## 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 the defined period milliseconds later.

## 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)`

## One-shot tasks

You can make a task to run only once by calling `lv_task_once(task)`. The task will be automatically deleted when called for the first time.

## 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 might be misleading if you use an operating system and call `lv_task_handler` in a task.

## 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 still uses 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` to 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);
}
```

(continues on next page)

(continued from previous page)

```
...

/*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*/
```

## 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 craeted 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\_once**(*lv\_task\_t \*task*)  
Delete the lv\_task after one call

### Parameters

- **task:** pointer to a lv\_task.

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

uint8\_t **prio**

Task priority

uint8\_t **once**

1: one shot task

## Drawing

With LittlevGL you don't need to draw anything manually. Just create objects (like buttons and labels), move and change them and LittlevGL will refresh and redraw what is required.

However, it might be useful to have a basic understanding of how drawing happens in LittlevGL.

The basic concept is to not draw directly to 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. E.g. when drawing a *background + button + text* each "stage" would be visible for a short time.
2. **It's faster** because when pixels are redrawn multiple times (e.g. background + button + text) 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).

## Buffering types

As you already might learn in the *Porting* section there are 3 types of buffering:

1. **One buffer** LittlevGL 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 LittlevGL 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* LittlevGL 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* LittlevGL 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 LittlevGL. 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.

### Mechanism of screen refreshing

1. Something happens on the GUI which requires redrawing. E.g. a button has been pressed, a chart has been changed or an animation happened, etc.
2. LittlevGL saves the changed object's old and new area into a buffer, called *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*):
  - LittlevGL checks the invalid areas and joins the adjacent or intersecting areas
  - Takes the first joined area if it's smaller the *display buffer* then simply draws 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** LittlevGL 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** LittlevGL 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 is being displayed as frame buffer. Its content is copied to the second buffer and all the changes are drawn on top of it.

### 3.15.4 Object types (Widgets)

#### Base object (`lv_obj`)

## Overview

The Base Object contains the most basic attributes of the objects:

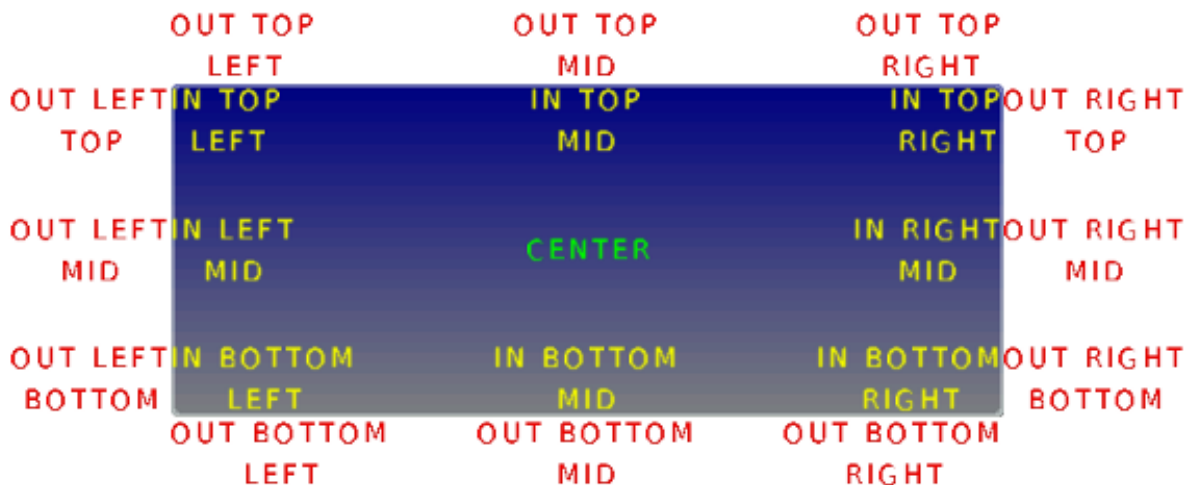
- coordinates
- parent object
- children
- main style
- attributes like *Click enable*, *Drag enable*, etc.

## Coordinates

The object size can be modified with `lv_obj_set_width(obj, new_width)` and `lv_obj_set_height(obj, new_height)` or in one function with `lv_obj_set_size(obj, new_width, new_height)`.

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 in one function with `lv_obj_set_pos(obj, new_x, new_y)`.

You can align the object to an other with `lv_obj_align(obj, obj_ref, LV_ALIGN_..., x_shift, y_shift)`. The second argument 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 argument means an x and y shift after the alignment.

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 middle point 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 realign the objects manually with `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 is no more children. For example:

```
lv_obj_t * child;
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_create(NULL, NULL)` you can load it with `lv_scr_load(screen1)`. 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 the screen of an object 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 the same layers will be shown on every screen. The *top layer* is above every object on the screen and *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.

## Style

The base object stores the *Main style* of the object. To set a new style use `lv_obj_set_style(obj, &new_style)` function. If `NULL` is set as style then the object will inherit its parent's style.

Note that you shouldn't use `lv_obj_set_style` for "non Base objects". Every object type has its own style set function which should be used for them. E.g. for button `lv_btn_set_style()`

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)` or to notify all object with a given style `lv_obj_report_style_mod(&style)`. If the parameter of `lv_obj_report_style_mod` is `NULL` all objects will be notified.

Read the *Style overview* to learn more about styles.

## 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.

## 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 as if it doesn't exist., Its children will be hidden too.
- **click** Enabled to click the object via input devices. If disabled then object behind this object will be clicked. (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.

## Opa scale

If `lv_obj_set_opa_scale_enable(obj, true)` is set for an object then the object's and all of its children's opacity can be adjusted with `lv_obj_set_opa_scale(obj, LV_OPA_...)`. The opacities stored in the styles will be scaled down by this factor.

It is very useful to fade in/out an object with some children using an *Animation*.

A little bit of technical background: during the rendering process the object and its parents are checked recursively to find a parent with enabled *Opa scale*. If an object has found with enabled *Opa scale* then that *Opa scale* will be used by the rendered object too. Therefore if you want to disable the Opa scaling for an object when the parent has Opa scale just enable Opa scaling for the object and set its value to `LV_OPA_COVER`. It will overwrite the parent's settings.

## 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 in 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` tells extra size the directions respectively.

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

## Styles

Use `lv_obj_set_style(obj, &style)` to set a style for a base object.

All `style.body` properties are used. The default style for screens is `lv_style_scr` and `lv_style_plain_color` for normal objects

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code



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

void lv_ex_obj_1(void)
{
    lv_obj_t * obj1;
    obj1 = lv_obj_create(lv_scr_act(), NULL);
    lv_obj_set_size(obj1, 100, 50);
    lv_obj_set_style(obj1, &lv_style_plain_color);
    lv_obj_align(obj1, NULL, LV_ALIGN_CENTER, -60, -30);

    /*Copy the previous object and enable drag*/
    lv_obj_t * obj2;
    obj2 = lv_obj_create(lv_scr_act(), obj1);
    lv_obj_set_style(obj2, &lv_style_pretty_color);
    lv_obj_align(obj2, NULL, LV_ALIGN_CENTER, 0, 0);

    static lv_style_t style_shadow;
    lv_style_copy(&style_shadow, &lv_style_pretty);
    style_shadow.body.shadow.width = 6;
    style_shadow.body.radius = LV_RADIUS_CIRCLE;

    /*Copy the previous object (drag is already enabled)*/
    lv_obj_t * obj3;
    obj3 = lv_obj_create(lv_scr_act(), obj2);
    lv_obj_set_style(obj3, &style_shadow);
    lv_obj_align(obj3, NULL, LV_ALIGN_CENTER, 60, 30);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_design\_mode\_t**

**typedef** bool (\***lv\_design\_cb\_t**)(**struct** \_\_lv\_obj\_t \*obj, **const** lv\_area\_t \*mask\_p,  
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\_align\_t**

**typedef** uint8\_t **lv\_drag\_dir\_t**

```
typedef struct _lv_obj_t lv_obj_t
typedef uint8_t lv_protect_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]  
*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\_REPEAT\_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\_KEY**

**LV\_EVENT\_FOCUSED**

**LV\_EVENT\_DEFOCUSED**

**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\_CORD\_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\_REFR\_EXT\_DRAW\_PAD**

Object’s extra padding has changed

**LV\_SIGNAL\_GET\_TYPE**

LittlevGL needs to retrieve the object’s type

**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\_END**

**LV\_SIGNAL\_FOCUS**

**LV\_SIGNAL\_DEFOCUS**

**LV\_SIGNAL\_CONTROL**

**LV\_SIGNAL\_GET\_EDITABLE**

**enum** [anonymous]  
Object alignment.

*Values:*

**LV\_ALIGN\_CENTER** = 0

**LV\_ALIGN\_IN\_TOP\_LEFT**

**LV\_ALIGN\_IN\_TOP\_MID**

**LV\_ALIGN\_IN\_TOP\_RIGHT**

**LV\_ALIGN\_IN\_BOTTOM\_LEFT**

**LV\_ALIGN\_IN\_BOTTOM\_MID**

**LV\_ALIGN\_IN\_BOTTOM\_RIGHT**

**LV\_ALIGN\_IN\_LEFT\_MID**

**LV\_ALIGN\_IN\_RIGHT\_MID**

**LV\_ALIGN\_OUT\_TOP\_LEFT**

**LV\_ALIGN\_OUT\_TOP\_MID**

**LV\_ALIGN\_OUT\_TOP\_RIGHT**

**LV\_ALIGN\_OUT\_BOTTOM\_LEFT**

**LV\_ALIGN\_OUT\_BOTTOM\_MID**

**LV\_ALIGN\_OUT\_BOTTOM\_RIGHT**

**LV\_ALIGN\_OUT\_LEFT\_TOP**

**LV\_ALIGN\_OUT\_LEFT\_MID**

**LV\_ALIGN\_OUT\_LEFT\_BOTTOM**

**LV\_ALIGN\_OUT\_RIGHT\_TOP**

**LV\_ALIGN\_OUT\_RIGHT\_MID**

**LV\_ALIGN\_OUT\_RIGHT\_BOTTOM**

**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\_ALL** = 0x3

Object can be dragged in all directions.

**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 `index` 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

## Functions

void **lv\_init**(void)

Init. the 'lv' library.

*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\_clean**(*lv\_obj\_t* \*obj)

Delete all children of an object

### Parameters

- **obj**: pointer to an object

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\_align**(*lv\_obj\_t \*obj*, **const** *lv\_obj\_t \*base*, *lv\_align\_t align*, *lv\_coord\_t x\_mod*,  
                                   *lv\_coord\_t y\_mod*)

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\_mod**: x coordinate shift after alignment
- **y\_mod**: y coordinate shift after alignment

void **lv\_obj\_align\_origo**(*lv\_obj\_t \*obj*, **const** *lv\_obj\_t \*base*, *lv\_align\_t align*, *lv\_coord\_t*  
                                   *x\_mod*, *lv\_coord\_t y\_mod*)

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\_mod**: x coordinate shift after alignment
- **y\_mod**: y coordinate shift 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\_set\_style**(*lv\_obj\_t \*obj*, **const** *lv\_style\_t \*style*)

Set a new style for an object

#### Parameters

- **obj**: pointer to an object
- **style\_p**: pointer to the new style

void **lv\_obj\_refresh\_style**(*lv\_obj\_t \*obj*)

Notify an object about its style is modified

#### Parameters

- **obj**: pointer to an object

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\_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\_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\_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\_opa\_scale\_enable**(*lv\_obj\_t \*obj*, bool *en*)

Set the opa scale enable parameter (required to set opa\_scale with *lv\_obj\_set\_opa\_scale()*)

#### Parameters

- **obj**: pointer to an object
- **en**: true: opa scaling is enabled for this object and all children; false: no opa scaling

void **lv\_obj\_set\_opa\_scale**(*lv\_obj\_t \*obj*, *lv\_opa\_t opa\_scale*)

Set the opa scale of an object

#### Parameters

- **obj**: pointer to an object
- **opa\_scale**: a factor to scale down opacity [0..255]

void **lv\_obj\_set\_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\_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

**void lv\_signal\_send**(*lv\_obj\_t \*obj, lv\_signal\_t signal, void \*param*)  
Send an event to the object

#### 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

**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

**Parameters**

- **scr**: pointer to an object

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 \*coords\_p)

Copy the coordinates of an object to an area

**Parameters**

- **obj**: pointer to an object
- **coords\_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(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(lv_obj_t *obj)`  
 Get that height reduced by the top and bottom padding.

**Return** the height which still fits into the container

**Parameters**

- `obj`: pointer to an object

`bool lv_obj_get_auto_realign(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

**const lv\_style\_t \*lv\_obj\_get\_style(const lv\_obj\_t \*obj)**

Get the style pointer of an object (if NULL get style of the parent)

**Return** pointer to a style

**Parameters**

- **obj**: pointer to an object

**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\_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

lv\_opa\_t **lv\_obj\_get\_opa\_scale\_enable**(const lv\_obj\_t \*obj)

Get the opa scale enable parameter

**Return** true: opa scaling is enabled for this object and all children; false: no opa scaling

#### Parameters

- **obj**: pointer to an object

lv\_opa\_t **lv\_obj\_get\_opa\_scale**(const lv\_obj\_t \*obj)

Get the opa scale parameter of an object

**Return** opa scale [0..255]

#### Parameters

- **obj**: pointer to an object

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\_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

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**(*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**(*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**(*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

**struct lv\_reailgn\_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 \*par**

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

**const** lv\_style\_t \***style\_p**

Pointer to the object's style

void \***group\_p**

Pointer to the group of the object

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.

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 opa_scale_en`  
 1: `opa_scale` is set

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

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

`uint8_t reserved`  
 Reserved for future use

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

`lv_opa_t opa_scale`  
 Scale down the opacity by this factor. Effects all children as well

`lv_coord_t ext_draw_pad`  
 EXTtend the size in every direction for drawing.

`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"

## Arc (lv\_arc)

### Overview

The *Arc* object **draws an arc** within **start and end angles** and with a given **thickness**.

## Angles

To set the angles use the `lv_arc_set_angles(arc, start_angle, end_angle)` function. The zero degree is at the bottom of the object and the degrees are increasing in a counter-clockwise direction. The angles should be in `[0;360]` range.

## Notes

The **width and height** of the *Arc* should be the **same**.

Currently, the *Arc* object **does not support anti-aliasing**.

## Styles

To set the style of an *Arc* object use `lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &style)`

- **line.rounded** make the endpoints rounded (opacity won't work properly if set to 1)
- **line.width** the thickness of the arc
- **line.color** the color of the arc.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

## C



code

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

void lv_ex_arc_1(void)
{
    /*Create style for the Arcs*/
    lv_style_t style;
    lv_style_copy(&style, &lv_style_plain);
    style.line.color = LV_COLOR_BLUE;           /*Arc color*/
    style.line.width = 8;                       /*Arc width*/

    /*Create an Arc*/
    lv_obj_t * arc = lv_arc_create(lv_scr_act(), NULL);
    lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &style); /*Use the new style*/
    lv_arc_set_angles(arc, 90, 60);
    lv_obj_set_size(arc, 150, 150);
    lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_arc_style_t
```

## Enums

**enum** [anonymous]

*Values:*

**LV\_ARC\_STYLE\_MAIN**

## 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\_angles**(*lv\_obj\_t* \*arc, uint16\_t start, uint16\_t end)

Set the start and end angles of an arc. 0 deg: bottom, 90 deg: right etc.

**Parameters**

- **arc**: pointer to an arc object
- **start**: the start angle [0..360]
- **end**: the end angle [0..360]

void **lv\_arc\_set\_style**(*lv\_obj\_t* \*arc, *lv\_arc\_style\_t* type, **const** *lv\_style\_t* \*style)

Set a style of a arc.

**Parameters**

- **arc**: pointer to arc object
- **type**: which style should be set
- **style**: pointer to a style

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

**const** *lv\_style\_t* \***lv\_arc\_get\_style**(**const** *lv\_obj\_t* \*arc, *lv\_arc\_style\_t* type)

Get style of a arc.

**Return** style pointer to the style

**Parameters**

- **arc**: pointer to arc object
- **type**: which style should be get

**struct lv\_arc\_ext\_t**

#### Public Members

lv\_coord\_t **angle\_start**

lv\_coord\_t **angle\_end**

### Bar (lv\_bar)

#### Overview

The Bar objects have got two main parts:

1. a **background** which is the object itself
2. an **indicator** which shape is similar to the background but its width/height can be adjusted.

The orientation of the bar can be vertical or horizontal according to the width/height ratio. Logically on horizontal bars, the indicator's width, on vertical bars the indicator's height can be changed.

#### 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.

#### Symmetrical

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

#### Styles

To set the style of an *Bar* object use `lv_bar_set_style(arc, LV_BAR_STYLE_MAIN, &style)`

- **LV\_BAR\_STYLE\_BG** is an *Base object* therefore it uses its style elements. Its default style is: `lv_style_pretty`.
- **LV\_BAR\_STYLE\_INDIC** is similar to the background. It uses the *left*, *right*, *top* and *bottom* paddings to keep some space from the edges of the background. Its default style is: `lv_style_pretty_color`.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

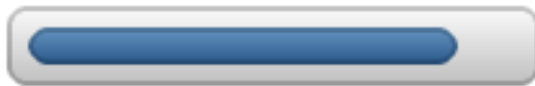
## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

### C

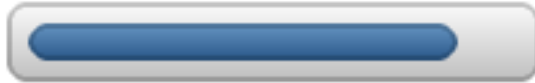


code

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

void lv_ex_bar_1(void)
{
    lv_obj_t * bar1 = lv_bar_create(lv_scr_act(), NULL);
    lv_obj_set_size(bar1, 200, 30);
    lv_obj_align(bar1, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_bar_set_anim_time(bar1, 1000);
    lv_bar_set_value(bar1, 100, LV_ANIM_ON);
}
```

## MicroPython



code

```
bar1 = lv.bar(lv.scr_act())
bar1.set_size(200, 30);
bar1.align(None, lv.ALIGN.CENTER, 0, 0);
bar1.set_anim_time(1000);
bar1.set_value(100, lv.ANIM.ON);
```

## API

### Typedefs

**typedef** uint8\_t **lv\_bar\_style\_t**

### Enums

**enum** [anonymous]

Bar styles.

*Values:*

**LV\_BAR\_STYLE\_BG**

**LV\_BAR\_STYLE\_INDIC**

Bar background 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\_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\_sym**(*lv\_obj\_t \*bar*, bool *en*)

Make the bar symmetric to zero. The indicator will grow from zero instead of the minimum position.

**Parameters**

- **bar**: pointer to a bar object
- **en**: true: enable disable symmetric behavior; false: disable

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.

void **lv\_bar\_set\_style**(*lv\_obj\_t \*bar*, *lv\_bar\_style\_t type*, const *lv\_style\_t \*style*)

Set a style of a bar

**Parameters**

- **bar**: pointer to a bar object
- **type**: which style should be set
- **style**: pointer to a style

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_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

`bool lv_bar_get_sym(lv_obj_t *bar)`

Get whether the bar is symmetric or not.

**Return** true: symmetric is enabled; false: disable

**Parameters**

- **bar**: pointer to a bar object

`uint16_t lv_bar_get_anim_time(lv_obj_t *bar)`

Get the animation time of the bar

**Return** the animation time in milliseconds.

**Parameters**

- **bar**: pointer to a bar object

`const lv_style_t *lv_bar_get_style(const lv_obj_t *bar, lv_bar_style_t type)`

Get a style of a bar

**Return** style pointer to a style

**Parameters**

- **bar**: pointer to a bar object
- **type**: which style should be get

`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`

`lv_anim_value_t anim_start`

`lv_anim_value_t anim_end`

`lv_anim_value_t anim_state`

`lv_anim_value_t anim_time`

`uint8_t sym`

```
const lv_style_t *style_indic
```

## Button (lv\_btn)

### Overview

Buttons are simple rectangle-like objects, but they change their style and state when they are pressed or released.

### States

Buttons can be in one of the 5 possible states:

- **LV\_BTN\_STATE\_REL** Released state
- **LV\_BTN\_STATE\_PR** Pressed state
- **LV\_BTN\_STATE\_TGL\_REL** Toggled released state
- **LV\_BTN\_STATE\_TGL\_PR** Toggled pressed state
- **LV\_BTN\_STATE\_INA** Inactive state

The state from `..._REL` to `..._PR` will be changed automatically when the button is pressed and back when released.

You can set the button's state manually with `lv_btn_set_state(btn, LV_BTN_STATE_TGL_REL)`.

### Toggle

You can configure the buttons as *toggle button* with `lv_btn_set_toggle(btn, true)`. In this case on release, the button goes to *toggled released* state.

### 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.

### Ink effect

You can enable a special animation on buttons: when a button is pressed, the pressed state will be drawn in a growing circle starting from the point of pressing. It's like an ink droplet in the water. When the button is released, the released state will be reverted by fading. It's like the ink is fully mixed with a lot of water and become no visible in it.

To control this animation use the following functions:

- `lv_btn_set_ink_in_time(btn, time_ms)` time of circle growing
- `lv_btn_set_ink_wait_time(btn, time_ms)` minim time to keep the fully covering (pressed) state
- `lv_btn_set_ink_out_time(btn, time_ms)` time fade back to releases state

This feature needs to be enabled with `LV_BTN_INK_EFFECT 1` in `lv_conf.h`.

## Styles

A button can have 5 independent styles for the 5 state. You can set them via: `lv_btn_set_style(btn, LV_BTN_STYLE_..., &style)`. The styles use the `style.body` properties.

- `LV_BTN_STYLE_REL` style of the released state. Default: `lv_style_btn_rel`
- `LV_BTN_STYLE_PR` style of the pressed state. Default: `lv_style_btn_pr`
- `LV_BTN_STYLE_TGL_REL` style of the toggled released state. Default: `lv_style_btn_tgl_rel`
- `LV_BTN_STYLE_TGL_PR` style of the toggled pressed state. Default: `lv_style_btn_tgl_pr`
- `LV_BTN_STYLE_INA` style of the inactive state. Default: `lv_style_btn_ina`

When you create a label on a button, it's a good practice to set the button's `style.text` properties too. Because labels have `style = NULL` by default, they inherit the parent's (button) style. Hence you don't need to create a new style for the label.

## Events

Besides 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*.

## 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*.

## Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_CLICKED) {
        printf("Clicked\n");
    }
    else if(event == LV_EVENT_VALUE_CHANGED) {
        printf("Toggled\n");
    }
}

void lv_ex_btn_1(void)
{
    lv_obj_t * label;

    lv_obj_t * btn1 = lv_btn_create(lv_scr_act(), NULL);
    lv_obj_set_event_cb(btn1, event_handler);
    lv_obj_align(btn1, NULL, LV_ALIGN_CENTER, 0, -40);

    label = lv_label_create(btn1, NULL);
    lv_label_set_text(label, "Button");

    lv_obj_t * btn2 = lv_btn_create(lv_scr_act(), NULL);
    lv_obj_set_event_cb(btn2, event_handler);
    lv_obj_align(btn2, NULL, LV_ALIGN_CENTER, 0, 40);
    lv_btn_set_toggle(btn2, true);
    lv_btn_toggle(btn2);
}
```

(continues on next page)

(continued from previous page)

```
lv_btn_set_fit2(btn2, LV_FIT_NONE, LV_FIT_TIGHT);

label = lv_label_create(btn2, NULL);
lv_label_set_text(label, "Toggled");
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_btn\_state\_t**

**typedef** uint8\_t **lv\_btn\_style\_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\_REL**

Released

**LV\_BTN\_STATE\_PR**

Pressed

**LV\_BTN\_STATE\_TGL\_REL**

Toggled released

**LV\_BTN\_STATE\_TGL\_PR**

Toggled pressed

**LV\_BTN\_STATE\_INA**

Inactive

**\_LV\_BTN\_STATE\_NUM**

Number of states

**enum** [anonymous]

Styles

*Values:*

**LV\_BTN\_STYLE\_REL**

Release style

**LV\_BTN\_STYLE\_PR**

Pressed style

**LV\_BTN\_STYLE\_TGL\_REL**

Toggle released style

## LV\_BTN\_STYLE\_TGL\_PR

Toggle pressed style

## LV\_BTN\_STYLE\_INA

Inactive style

## Functions

*lv\_obj\_t* \***lv\_btn\_create**(*lv\_obj\_t* \*par, **const** *lv\_obj\_t* \*copy)

Create a button objects

**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\_toggle**(*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 tell 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**: bottom 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 tell 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 tell how to change the button size automatically.

**Parameters**

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

**void lv\_btn\_set\_ink\_in\_time**(*lv\_obj\_t \*btn, uint16\_t time*)

Set time of the ink effect (draw a circle on click to animate in the new state)

**Parameters**

- **btn**: pointer to a button object
- **time**: the time of the ink animation

**void lv\_btn\_set\_ink\_wait\_time**(*lv\_obj\_t \*btn, uint16\_t time*)

Set the wait time before the ink disappears

**Parameters**

- **btn**: pointer to a button object
- **time**: the time of the ink animation

**void lv\_btn\_set\_ink\_out\_time**(*lv\_obj\_t \*btn, uint16\_t time*)

Set time of the ink out effect (animate to the released state)

**Parameters**

- **btn**: pointer to a button object
- **time**: the time of the ink animation

**void lv\_btn\_set\_style**(*lv\_obj\_t \*btn, lv\_btn\_style\_t type, const lv\_style\_t \*style*)

Set a style of a button.

**Parameters**

- **btn**: pointer to button object
- **type**: which style should be set
- **style**: pointer to a style

*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)

**Parameters**

- **btn**: pointer to a button object



bool **lv\_btn\_get\_toggle**(const lv\_obj\_t \*btn)

Get the toggle enable attribute of the button

**Return** ture: toggle 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

uint16\_t **lv\_btn\_get\_ink\_in\_time**(const lv\_obj\_t \*btn)

Get time of the ink in effect (draw a circle on click to animate in the new state)

**Return** the time of the ink animation

**Parameters**

- btn: pointer to a button object

uint16\_t **lv\_btn\_get\_ink\_wait\_time**(const lv\_obj\_t \*btn)

Get the wait time before the ink disappears

**Return** the time of the ink animation

**Parameters**

- **btn**: pointer to a button object

**uint16\_t lv\_btn\_get\_ink\_out\_time(const lv\_obj\_t \*btn)**

Get time of the ink out effect (animate to the releases state)

**Return** the time of the ink animation

#### Parameters

- **btn**: pointer to a button object

**const lv\_style\_t \*lv\_btn\_get\_style(const lv\_obj\_t \*btn, lv\_btn\_style\_t type)**

Get style of a button.

**Return** style pointer to the style

#### Parameters

- **btn**: pointer to button object
- **type**: which style should be get

**struct lv\_btn\_ext\_t**

*#include <lv\_btn.h>* Extended data of button

#### Public Members

**lv\_cont\_ext\_t cont**

Ext. of ancestor

**const lv\_style\_t \*styles[\_LV\_BTN\_STATE\_NUM]**

Styles in each state

**uint16\_t ink\_in\_time**

[ms] Time of ink fill effect (0: disable ink effect)

**uint16\_t ink\_wait\_time**

[ms] Wait before the ink disappears

**uint16\_t ink\_out\_time**

[ms] Time of ink disappearing

**lv\_btn\_state\_t state**

Current state of the button from 'lv\_btn\_state\_t' enum

**uint8\_t toggle**

1: Toggle enabled

### Button matrix (lv\_btnm)

#### Overview

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

#### 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\_btnm\_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", ""}`. The button's width is recalculated in every line to will the whole line.

## Control buttons

The **buttons width** can be set relative to the other button in the same line with `lv_btnm_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.

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

- **LV\_BTNM\_CTRL\_HIDDEN** make a button hidden
- **LV\_BTNM\_CTRL\_NO\_REPEAT** disable repeating when the button is long pressed
- **LV\_BTNM\_CTRL\_INACTIVE** make a button inactive
- **LV\_BTNM\_CTRL\_TGL\_ENABLE** enable toggling of a button
- **LV\_BTNM\_CTRL\_TGL\_STATE** set the toggle state
- **LV\_BTNM\_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_btnm_set_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)` and `lv_btnm_clear_btn_ctrl(btnm, btn_id, LV_BTNM_CTRL_...)` respectively. More **LV\_BTNM\_CTRL\_...** values can be *Ored*

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

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

## Recolor

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

## 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.

## Styles

The Button matrix works with 6 styles: a background and 5 button styles for each state. You can set the styles with `lv_btm_set_style(btn, LV_BTNM_STYLE_..., &style)`. The background and the buttons use the `style.body` properties. The labels use the `style.text` properties of the button styles.

- **LV\_BTNM\_STYLE\_BG** Background style. Uses all *style.body* properties including *padding* Default: *lv\_style\_pretty*
- **LV\_BTNM\_STYLE\_BTN\_REL** style of the released buttons. Default: *lv\_style\_btn\_rel*
- **LV\_BTNM\_STYLE\_BTN\_PR** style of the pressed buttons. Default: *lv\_style\_btn\_pr*
- **LV\_BTNM\_STYLE\_BTN\_TGL\_REL** style of the toggled released buttons. Default: *lv\_style\_btn\_tgl\_rel*
- **LV\_BTNM\_STYLE\_BTN\_TGL\_PR** style of the toggled pressed buttons. Default: *lv\_style\_btn\_tgl\_pr*
- **LV\_BTNM\_STYLE\_BTN\_INA** style of the inactive buttons. Default: *lv\_style\_btn\_ina*

## 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 ID of the pressed/released button.

Learn more about *Events*.

### ##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*.

## Example

## C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        const char * txt = lv_btm_get_active_btm_text(obj);

        printf("%s was pressed\n", txt);
    }
}

static const char * btm_map[] = {"1", "2", "3", "4", "5", "\n",
                                  "6", "7", "8", "9", "0", "\n",
                                  "Action1", "Action2", ""};

void lv_ex_btm_1(void)
{
    lv_obj_t * btm1 = lv_btm_create(lv_scr_act(), NULL);
    lv_btm_set_map(btm1, btm_map);
    lv_btm_set_btm_width(btm1, 10, 2);          /*Make "Action1" twice as wide as
    ↪ "Action2"*/
    lv_obj_align(btm1, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_obj_set_event_cb(btm1, event_handler);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint16\_t **lv\_btmn\_ctrl\_t**

**typedef** uint8\_t **lv\_btmn\_style\_t**

### Enums

**enum** [anonymous]

Type to store button control bits (disabled, hidden etc.)

*Values:*

**LV\_BTNM\_CTRL\_HIDDEN** = 0x0008

Button hidden

**LV\_BTNM\_CTRL\_NO\_REPEAT** = 0x0010

Do not repeat press this button.

**LV\_BTNM\_CTRL\_INACTIVE** = 0x0020

Disable this button.

**LV\_BTNM\_CTRL\_TGL\_ENABLE** = 0x0040

Button *can* be toggled.

**LV\_BTNM\_CTRL\_TGL\_STATE** = 0x0080

Button is currently toggled (e.g. checked).

**LV\_BTNM\_CTRL\_CLICK\_TRIG** = 0x0100

1: Send LV\_EVENT\_SELECTED on CLICK, 0: Send LV\_EVENT\_SELECTED on PRESS

**enum** [anonymous]

*Values:*

**LV\_BTNM\_STYLE\_BG**

**LV\_BTNM\_STYLE\_BTN\_REL**

**LV\_BTNM\_STYLE\_BTN\_PR**

**LV\_BTNM\_STYLE\_BTN\_TGL\_REL**

**LV\_BTNM\_STYLE\_BTN\_TGL\_PR**

**LV\_BTNM\_STYLE\_BTN\_INA**

### Functions

*lv\_obj\_t* \***lv\_btmn\_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\_btnm\_set\_map**(const 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\_btnm\_set\_ctrl\_map**(const lv\_obj\_t \*btnm, const lv\_btnm\_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\_BTNM\_CTRL\_NO\_REPEAT | LV\_BTNM\_CTRL\_TGL\_ENABLE

void **lv\_btnm\_set\_pressed**(const lv\_obj\_t \*btnm, uint16\_t id)

Set the pressed button i.e. visually highlight it. Mainly used a when the btnm is in a group to show the selected button

#### Parameters

- **btnm**: pointer to button matrix object
- **id**: index of the currently pressed button (LV\_BTNM\_BTN\_NONE to unpress)

void **lv\_btnm\_set\_style**(lv\_obj\_t \*btnm, lv\_btnm\_style\_t type, const lv\_style\_t \*style)

Set a style of a button matrix

#### Parameters

- **btnm**: pointer to a button matrix object
- **type**: which style should be set
- **style**: pointer to a style

void **lv\_btnm\_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\_btnm\_set\_btn\_ctrl**(const lv\_obj\_t \*btnm, uint16\_t btn\_id, lv\_btnm\_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\_btnm\_clear\_btn\_ctrl**(const lv\_obj\_t \*btnm, uint16\_t btn\_id, lv\_btnm\_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\_btnm\_set\_btn\_ctrl\_all**(*lv\_obj\_t \*btnm, lv\_btnm\_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\_btnm\_ctrl\_t**. Values can be ORed.

void **lv\_btnm\_clear\_btn\_ctrl\_all**(*lv\_obj\_t \*btnm, lv\_btnm\_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\_btnm\_ctrl\_t**. Values can be ORed.
- **en**: true: set the attributes; false: clear the attributes

void **lv\_btnm\_set\_btn\_width**(**const** *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\_btnm\_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\_btnm\_set\_one\_toggle**(*lv\_obj\_t \*btnm, bool one\_toggle*)

Make the button matrix like a selector widget (only one button may be toggled at a time).

Toggling must be enabled on the buttons you want to be selected with **lv\_btnm\_set\_ctrl** or **lv\_btnm\_set\_btn\_ctrl\_all**.

#### Parameters

- **btnm**: Button matrix object
- **one\_toggle**: Whether “one toggle” mode is enabled

**const** char \*\***lv\_btnm\_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\_btnm\_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\_btm\_get\_active\_btn**(const lv\_obj\_t \*btm)

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\_BTNM\_BTN\_NONE: if unset)

**Parameters**

- `btm`: pointer to button matrix object

const char \***lv\_btm\_get\_active\_btn\_text**(const lv\_obj\_t \*btm)

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**

- `btm`: pointer to button matrix object

uint16\_t **lv\_btm\_get\_pressed\_btn**(const lv\_obj\_t \*btm)

Get the pressed button’s index. The button be really pressed by the user or manually set to pressed with `lv_btm_set_pressed`

**Return** index of the pressed button (LV\_BTNM\_BTN\_NONE: if unset)

**Parameters**

- `btm`: pointer to button matrix object

const char \***lv\_btm\_get\_btn\_text**(const lv\_obj\_t \*btm, uint16\_t btn\_id)

Get the button’s text

**Return** text of `btn_index`’ button

**Parameters**

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

bool **lv\_btm\_get\_btn\_ctrl**(lv\_obj\_t \*btm, uint16\_t btn\_id, lv\_btm\_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**

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

const lv\_style\_t \***lv\_btm\_get\_style**(const lv\_obj\_t \*btm, lv\_btm\_style\_t type)

Get a style of a button matrix

**Return** style pointer to a style

**Parameters**

- `btm`: pointer to a button matrix object
- `type`: which style should be get

bool **lv\_btm\_get\_one\_toggle**(const lv\_obj\_t \*btm)

Find whether “one toggle” mode is enabled.

**Return** whether “one toggle” mode is enabled

**Parameters**

- **btm**: Button matrix object

**struct lv\_btm\_ext\_t**

**Public Members**

const char \*\*map\_p

lv\_area\_t \*button\_areas

lv\_btm\_ctrl\_t \*ctrl\_bits

const lv\_style\_t \*styles\_btn[\_LV\_BTN\_STATE\_NUM]

uint16\_t btn\_cnt

uint16\_t btn\_id\_pr

uint16\_t btn\_id\_act

uint8\_t recolor

uint8\_t one\_toggle

**Calendar (lv\_calendar)**

**Overview**

The Calendar object is a classic calendar which can:

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

The 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 a 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)`.

## Styles

You can set the styles with `lv_calendar_set_style(btn, LV_CALENDAR_STYLE_..., &style)`.

- **LV\_CALENDAR\_STYLE\_BG** Style of the background using the **body** properties and the style of the date numbers using the **text** properties. **body.padding.left/right/bottom** padding will be added on the edges. around the date numbers.
- **LV\_CALENDAR\_STYLE\_HEADER** Style of the header where the current year and month is displayed. **body** and **text** properties are used.
- **LV\_CALENDAR\_STYLE\_HEADER\_PR** Pressed header style, used when the next/prev. month button is being pressed. **text** properties are used by the arrows.
- **LV\_CALENDAR\_STYLE\_DAY\_NAMES** Style of the day names. **text** properties are used by the day texts and **body.padding.top** determines the space above the day names.
- **LV\_CALENDAR\_STYLE\_HIGHLIGHTED\_DAYS** **text** properties are used to adjust the style of the highlights days
- **LV\_CALENDAR\_STYLE\_INACTIVE\_DAYS** **text** properties are used to adjust the style of the visible days of previous/next month.
- **LV\_CALENDAR\_STYLE\_WEEK\_BOX** **body** properties are used to set the style of the week box
- **LV\_CALENDAR\_STYLE\_TODAY\_BOX** **body** and **text** properties are used to set the style of the today box

## 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.

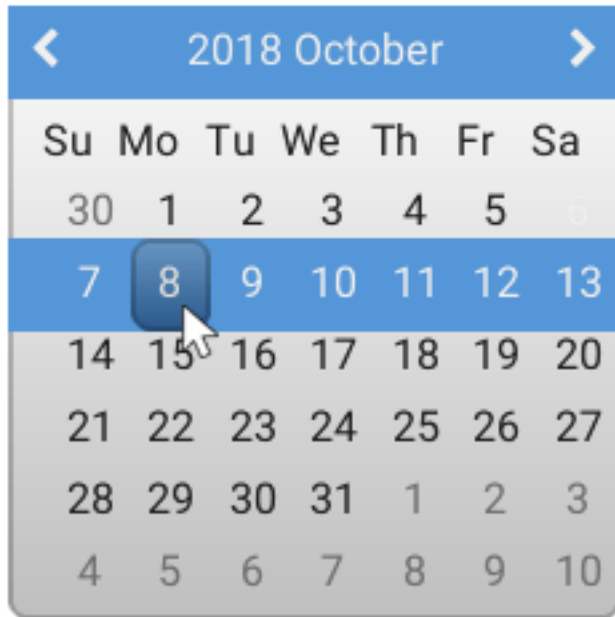
## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code

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

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_CLICKED) {
        lv_calendar_date_t * date = lv_calendar_get_pressed_date(obj);
        if(date) {
            lv_calendar_set_today_date(obj, date);
        }
    }
}

void lv_ex_calendar_1(void)
{
    lv_obj_t * calendar = lv_calendar_create(lv_scr_act(), NULL);
    lv_obj_set_size(calendar, 230, 230);
    lv_obj_align(calendar, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_obj_set_event_cb(calendar, event_handler);

    /*Set the today*/
    lv_calendar_date_t today;
```

(continues on next page)

(continued from previous page)

```

today.year = 2018;
today.month = 10;
today.day = 23;

lv_calendar_set_today_date(calendar, &today);
lv_calendar_set_showed_date(calendar, &today);

/*Highlight some days*/
static lv_calendar_date_t highlighted_days[3];      /*Only it's pointer will be
↳ saved so should be static*/
highlighted_days[0].year = 2018;
highlighted_days[0].month = 10;
highlighted_days[0].day = 6;

highlighted_days[1].year = 2018;
highlighted_days[1].month = 10;
highlighted_days[1].day = 11;

highlighted_days[2].year = 2018;
highlighted_days[2].month = 11;
highlighted_days[2].day = 22;

lv_calendar_set_highlighted_dates(calendar, highlighted_days, 3);
}

```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_calendar\_style\_t**

### Enums

**enum** [anonymous]

Calendar styles

*Values:*

**LV\_CALENDAR\_STYLE\_BG**

Background and “normal” date numbers style

**LV\_CALENDAR\_STYLE\_HEADER**

**LV\_CALENDAR\_STYLE\_HEADER\_PR**

Calendar header style

**LV\_CALENDAR\_STYLE\_DAY\_NAMES**

Calendar header style (when pressed)

## LV\_CALENDAR\_STYLE\_HIGHLIGHTED\_DAYS

Day name style

## LV\_CALENDAR\_STYLE\_INACTIVE\_DAYS

Highlighted day style

## LV\_CALENDAR\_STYLE\_WEEK\_BOX

Inactive day style

## LV\_CALENDAR\_STYLE\_TODAY\_BOX

Week highlight 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 \*\**day\_names*)  
Set the name of the month

#### Parameters

- **calendar**: pointer to a calendar object
- **day\_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.

void **lv\_calendar\_set\_style**(*lv\_obj\_t \*calendar*, *lv\_calendar\_style\_t type*, **const** *lv\_style\_t \*style*)  
Set a style of a calendar.

#### Parameters

- **calendar**: pointer to calendar object
- **type**: which style should be set
- **style**: pointer to a style

*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.

#### 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

**const** lv\_style\_t \***lv\_calendar\_get\_style**(**const** lv\_obj\_t \*calendar, lv\_calendar\_style\_t type)

Get style of a calendar.

**Return** style pointer to the style

#### Parameters

- **calendar**: pointer to calendar object
- **type**: which style should be get

**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**

uint8\_t **highlighted\_dates\_num**

int8\_t **btn\_pressing**

lv\_calendar\_date\_t **pressed\_date**

**const** char \*\***day\_names**

**const** char \*\***month\_names**

**const** lv\_style\_t \***style\_header**



```

const lv_style_t *style_header_pr
const lv_style_t *style_day_names
const lv_style_t *style_highlighted_days
const lv_style_t *style_inactive_days
const lv_style_t *style_week_box
const lv_style_t *style_today_box
    
```

## Canvas (lv\_canvas)

### Overview

A Canvas is like an *Image* where the user can draw anything.

### 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_TRUE_COLOR_ALPHA)`. `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.

### 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)` fills the whole canvas to blue.

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

## Rotate

A rotated image can be added to canvas with `lv_canvas_rotate(canvas, &img_dsc, angle, x, y, pivot_x, pivot_y)`. It will rotate the image shown by `img_dsc` around the given pivot and stores it on the `x, y` coordinates of `canvas`. Instead of `img_dsc` and the buffer of an other canvas also can be used by `lv_canvas_get_img(canvas)`.

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

## Styles

You can set the styles with `lv_canvas_set_style(btn, LV_CANVAS_STYLE_MAIN, &style)`. `style.image.color` is used to tell the base color with `LV_IMG_CF_ALPHA...` color format.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code

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

#define CANVAS_WIDTH 200
#define CANVAS_HEIGHT 150

void lv_ex_canvas_1(void)
{
    static lv_style_t style;
    lv_style_copy(&style, &lv_style_plain);
    style.body.main_color = LV_COLOR_RED;
    style.body.grad_color = LV_COLOR_MAROON;
    style.body.radius = 4;
    style.body.border.width = 2;
    style.body.border.color = LV_COLOR_WHITE;
    style.body.shadow.color = LV_COLOR_WHITE;
    style.body.shadow.width = 4;
    style.line.width = 2;
    style.line.color = LV_COLOR_BLACK;
    style.text.color = LV_COLOR_BLUE;

    static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_TRUE_COLOR(CANVAS_WIDTH, CANVAS_
↪HEIGHT)];

    lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);
    lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH, CANVAS_HEIGHT, LV_IMG_CF_TRUE_
↪COLOR);
    lv_obj_align(canvas, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_canvas_fill_bg(canvas, LV_COLOR_SILVER);

    lv_canvas_draw_rect(canvas, 70, 60, 100, 70, &style);

    lv_canvas_draw_text(canvas, 40, 20, 100, &style, "Some text on text canvas", LV_
↪LABEL_ALIGN_LEFT);

    /* Test the rotation. It requires an other buffer where the original image is
↪stored.
    * So copy the current image to buffer and rotate it to the canvas */
    lv_color_t cbuf_tmp[CANVAS_WIDTH * CANVAS_HEIGHT];
    memcpy(cbuf_tmp, cbuf, sizeof(cbuf_tmp));
    lv_img_dsc_t img;
    img.data = (void *)cbuf_tmp;
    img.header.cf = LV_IMG_CF_TRUE_COLOR;
    img.header.w = CANVAS_WIDTH;
    img.header.h = CANVAS_HEIGHT;

    lv_canvas_fill_bg(canvas, LV_COLOR_SILVER);
    lv_canvas_rotate(canvas, &img, 30, 0, 0, CANVAS_WIDTH / 2, CANVAS_HEIGHT / 2);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_canvas\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_CANVAS\_STYLE\_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

void **lv\_canvas\_set\_style**(*lv\_obj\_t \*canvas*, *lv\_canvas\_style\_t type*, **const** *lv\_style\_t \*style*)  
Set a style of a canvas.

#### Parameters

- **canvas**: pointer to canvas object
- **type**: which style should be set
- **style**: pointer to a style

*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

**const** *lv\_style\_t \****lv\_canvas\_get\_style**(**const** *lv\_obj\_t \*canvas*, *lv\_canvas\_style\_t type*)  
Get style of a canvas.

**Return** style pointer to the style

#### Parameters

- **canvas**: pointer to canvas object
- **type**: which style should be get

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\_rotate**(*lv\_obj\_t \*canvas, lv\_img\_dsc\_t \*img, int16\_t angle, lv\_coord\_t offset\_x, lv\_coord\_t offset\_y, int32\_t pivot\_x, int32\_t pivot\_y*)

Rotate and image and store the result on a canvas.

#### Parameters

- **canvas**: pointer to a canvas object
- **img**: pointer to an image descriptor. Can be the image descriptor of an other canvas too (**lv\_canvas\_get\_img()**).
- **angle**: the angle of rotation (0..360);
- **offset\_x**: offset X to tell where to put the result data on destination canvas
- **offset\_y**: offset Y 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

void **lv\_canvas\_fill\_bg**(*lv\_obj\_t \*canvas, lv\_color\_t color*)

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, const lv\_style\_t \*style*)

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, const lv\_style\_t \*style, 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,
                       const lv_style_t *style)
```

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,
                        const lv_style_t *style)
```

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, const lv_style_t *style)
```

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, const lv_style_t *style)
```

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**

## Check box (*lv\_cb*)

### Overview

The Check Box objects are built from a *Button* background which contains an also Button *bullet* and a *Label* to realize a classical check box.

### Text

The text can be modified by the `lv_cb_set_text(cb, "New text")` function. It will dynamically allocate the text.

To set a static text use `lv_cb_set_static_text(cb, txt)`. This way only a pointer of `txt` will be stored it shouldn't be deallocated while the checkbox exists.

### Check/Uncheck

You can manually check / un-check the Check box via `lv_cb_set_checked(cb, true/false)`.

### Inactive

To make the Check box inactive use `lv_cb_set_inactive(cb, true)`.

### Styles

The Check box styles can be modified with `lv_cb_set_style(cb, LV_CB_STYLE_..., &style)`.

- **LV\_CB\_STYLE\_BG** Background style. Uses all `style.body` properties. The label's style comes from `style.text`. Default: `lv_style_transp`
- **LV\_CB\_STYLE\_BOX\_REL** Style of the released box. Uses the `style.body` properties. Default: `lv_style_btn_rel`
- **LV\_CB\_STYLE\_BOX\_PR** Style of the pressed box. Uses the `style.body` properties. Default: `lv_style_btn_pr`
- **LV\_CB\_STYLE\_BOX\_TGL\_REL** Style of the checked released box. Uses the `style.body` properties. Default: `lv_style_btn_tgl_rel`
- **LV\_CB\_STYLE\_BOX\_TGL\_PR** Style of the checked released box. Uses the `style.body` properties. Default: `lv_style_btn_tgl_pr`
- **LV\_CB\_STYLE\_BOX\_INA** Style of the inactive box. Uses the `style.body` properties. Default: `lv_style_btn_ina`



## Events

Besides the [Generic events](#) the following [Special events](#) are sent by the Check boxes:

- **LV\_EVENT\_VALUE\_CHANGED** sent when the Check box 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 Check boxes.

Learn more about *Events*.

## 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*.

## Example

C

☐ I agree to terms and conditions.

code

```
#include "lvgl/lvgl.h"
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        printf("State: %s\n", lv_cb_is_checked(obj) ? "Checked" : "Unchecked");
    }
}

void lv_ex_cb_1(void)
{
    lv_obj_t * cb = lv_cb_create(lv_scr_act(), NULL);
    lv_cb_set_text(cb, "I agree to terms and conditions.");
    lv_obj_align(cb, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_obj_set_event_cb(cb, event_handler);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_cb\_style\_t**

### Enums

**enum** [anonymous]  
Checkbox styles.

*Values:*

**LV\_CB\_STYLE\_BG**  
Style of object background.

**LV\_CB\_STYLE\_BOX\_REL**  
Style of box (released).

**LV\_CB\_STYLE\_BOX\_PR**  
Style of box (pressed).

**LV\_CB\_STYLE\_BOX\_TGL\_REL**  
Style of box (released but checked).

**LV\_CB\_STYLE\_BOX\_TGL\_PR**  
Style of box (pressed and checked).

**LV\_CB\_STYLE\_BOX\_INA**  
Style of disabled box

### Functions

*lv\_obj\_t \****lv\_cb\_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\_cb\_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\_cb\_set\_static\_text**(*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\_cb\_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\_cb\_set\_inactive**(*lv\_obj\_t* \**cb*)

Make the check box inactive (disabled)

**Parameters**

- **cb**: pointer to a check box object

void **lv\_cb\_set\_style**(*lv\_obj\_t* \**cb*, *lv\_cb\_style\_t* *type*, **const** *lv\_style\_t* \**style*)

Set a style of a check box

**Parameters**

- **cb**: pointer to check box object
- **type**: which style should be set
- **style**: pointer to a style

**const** char \***lv\_cb\_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\_cb\_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\_cb\_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

**const** *lv\_style\_t* \***lv\_cb\_get\_style**(const *lv\_obj\_t* \*cb, *lv\_cb\_style\_t* type)

Get a style of a button

**Return** style pointer to the style

**Parameters**

- **cb**: pointer to check box object
- **type**: which style should be get

**struct** **lv\_cb\_ext\_t**

**Public Members**

*lv\_btn\_ext\_t* **bg\_btn**

*lv\_obj\_t* \***bullet**

*lv\_obj\_t* \***label**

## Chart (lv\_chart)

### Overview

Charts have a rectangle-like background with horizontal and vertical division lines and data series drawn from lines, points columns or areas.

### 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** exists:

- **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\_COL** Draw columns
- **LV\_CHART\_TYPE\_POINT** Draw points
- **LV\_CHART\_TYPE\_AREA** Draw areas (fill the area below the lines)

- **LV\_CHART\_TYPE\_VERTICAL\_LINE** Draw only vertical lines to connect the points. Useful if the chart width is equal to the number of points.

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 | LV_CHART_TYPE_POINT`).

### 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 to not draw that point, column, or line segment.

### Update modes

`lv_chart_set_next` can behave in two way 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** Add the new data in a circular way. (Like an ECG diagram)

To 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 a the min. and max. values in y directions 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.

## Series' appearance

To set the **line width** and **point radius** of the series use the `lv_chart_set_series_width(chart, size)` function. The default value is: 2.

The **opacity of the data lines** can be specified by `lv_chart_set_series_opa(chart, opa)`. The default value is: OPA\_COVER.

You can apply a **dark color fade** on the bottom of columns and points by `lv_chart_set_series_darking(chart, effect)` function. The default dark level is OPA\_50.

## Tick marks and labels

Ticks and texts to ticks can be added.

`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.

Where text are added *major tick lines* are drawn, ot the other places *minor tick lines*. `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`

`lv_chart_set_margin(chart, 20)` needs to be used to add some extra space around the chart for the ticks and texts.

## Styles

You can set the styles with `lv_chart_set_style(btn, LV_CHART_STYLE_MAIN, &style)`.

- **style.body** properties set the background's appearance
- **style.line** properties set the division lines' appearance
- **style.text** properties set the axis labels' appearance

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code

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

void lv_ex_chart_1(void)
{
    /*Create a chart*/
    lv_obj_t * chart;
    chart = lv_chart_create(lv_scr_act(), NULL);
    lv_obj_set_size(chart, 200, 150);
    lv_obj_align(chart, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_chart_set_type(chart, LV_CHART_TYPE_POINT | LV_CHART_TYPE_LINE); /*Show
↪ lines and points too*/
    lv_chart_set_series_opa(chart, LV_OPA_70); /*Opacity
↪ of the data series*/
    lv_chart_set_series_width(chart, 4); /*Line
↪ width and point radius*/

    lv_chart_set_range(chart, 0, 100);

    /*Add two data series*/
    lv_chart_series_t * ser1 = lv_chart_add_series(chart, LV_COLOR_RED);
    lv_chart_series_t * ser2 = lv_chart_add_series(chart, LV_COLOR_GREEN);

    /*Set the next points on 'dl1'*/
    lv_chart_set_next(chart, ser1, 10);
    lv_chart_set_next(chart, ser1, 10);
    lv_chart_set_next(chart, ser1, 10);
    lv_chart_set_next(chart, ser1, 10);
    lv_chart_set_next(chart, ser1, 10);
}
```

(continues on next page)

(continued from previous page)

```
lv_chart_set_next(chart, ser1, 10);
lv_chart_set_next(chart, ser1, 10);
lv_chart_set_next(chart, ser1, 30);
lv_chart_set_next(chart, ser1, 70);
lv_chart_set_next(chart, ser1, 90);

/*Directly set points on 'dl2'*/
ser2->points[0] = 90;
ser2->points[1] = 70;
ser2->points[2] = 65;
ser2->points[3] = 65;
ser2->points[4] = 65;
ser2->points[5] = 65;
ser2->points[6] = 65;
ser2->points[7] = 65;
ser2->points[8] = 65;
ser2->points[9] = 65;

lv_chart_refresh(chart); /*Required after direct set*/
}
```

## MicroPython

No examples yet.

## 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
typedef uint8_t lv_chart_style_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
LV_CHART_TYPE_POINT = 0x04
    Draw circles on the points
```



**LV\_CHART\_TYPE\_VERTICAL\_LINE** = 0x08

Draw vertical lines on points (useful when chart width == point count)

**LV\_CHART\_TYPE\_AREA** = 0x10

Draw area chart

**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

**enum** [anonymous]

*Values:*

**LV\_CHART\_STYLE\_MAIN**

## Functions

`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 serie

**Parameters**

- **chart**: pointer to a chart object

- **serie**: pointer to the chart's serie 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 to chart object
- **point\_cnt**: new number of points on the data lines

void **lv\_chart\_set\_series\_opa**(*lv\_obj\_t \*chart*, lv\_opa\_t *opa*)  
Set the opacity of the data series

#### Parameters

- **chart**: pointer to a chart object
- **opa**: opacity of the data series

void **lv\_chart\_set\_series\_width**(*lv\_obj\_t \*chart*, lv\_coord\_t *width*)  
Set the line width or point radius of the data series

#### Parameters

- **chart**: pointer to a chart object
- **width**: the new width

void **lv\_chart\_set\_series\_darkening**(*lv\_obj\_t \*chart*, lv\_opa\_t *dark\_eff*)  
Set the dark effect on the bottom of the points or columns

#### Parameters

- **chart**: pointer to a chart object
- **dark\_eff**: dark effect level (LV\_OPA\_TRANSP to turn off)

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

**static** void **lv\_chart\_set\_style**(*lv\_obj\_t \*chart, lv\_chart\_style\_t type, const lv\_style\_t \*style*)

Set the style of a chart

#### Parameters

- **chart**: pointer to a chart object
- **type**: which style should be set (can be only LV\_CHART\_STYLE\_MAIN)
- **style**: pointer to a style

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_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_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

```
void lv_chart_set_margin(lv_obj_t *chart, uint16_t margin)
```

Set the margin around the chart, used for axes value and ticks

#### Parameters

- **chart**: pointer to an chart object
- **margin**: value of the margin [px]

```
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_cnt(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

*lv\_opa\_t* **lv\_chart\_get\_series\_opa**(const *lv\_obj\_t* \*chart)

Get the opacity of the data series

**Return** the opacity of the data series

**Parameters**

- **chart**: pointer to chart object

*lv\_coord\_t* **lv\_chart\_get\_series\_width**(const *lv\_obj\_t* \*chart)

Get the data series width

**Return** the width the data series (lines or points)

**Parameters**

- **chart**: pointer to chart object

*lv\_opa\_t* **lv\_chart\_get\_series\_darking**(const *lv\_obj\_t* \*chart)

Get the dark effect level on the bottom of the points or columns

**Return** dark effect level (LV\_OPA\_TRANSP to turn off)

**Parameters**

- **chart**: pointer to chart object

**static const** *lv\_style\_t* \***lv\_chart\_get\_style**(const *lv\_obj\_t* \*chart, *lv\_chart\_style\_t* type)

Get the style of an chart object

**Return** pointer to the chart's style

**Parameters**

- **chart**: pointer to an chart object
- **type**: which style should be get (can be only LV\_CHART\_STYLE\_MAIN)

*uint16\_t* **lv\_chart\_get\_margin**(*lv\_obj\_t* \*chart)

Get the margin around the chart, used for axes value and labels

**Parameters**

- **chart**: pointer to an chart object
- **return**: value of the margin

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_chart_type_t type
lv_chart_axis_cfg_t y_axis
lv_chart_axis_cfg_t x_axis
uint16_t margin
uint8_t update_mode
lv_coord_t width
uint8_t num
lv_opa_t opa
lv_opa_t dark
struct lv_chart_ext_t::[anonymous] series

```

## Container (lv\_cont)

### Overview

The containers are **rectangle-like 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, `padding.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 `padding.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.

## Auto fit

Containers have an auto fit 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** Set the size to involve all children by keeping `padding.top/bottom/left/right` space on the edges.
- **LV\_FIT\_FLOOD** Set the size to the parent's size by keeping `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.

To set the auto fit use `lv_cont_set_fit(cont, LV_FIT_...)`. It will set the same auto fit in every direction. 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)`.

## Styles

You can set the styles with `lv_cont_set_style(btn, LV_CONT_STYLE_MAIN, &style)`.

- `style.body` properties are used.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

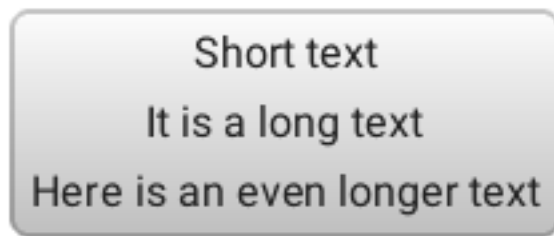
## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code

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

void lv_ex_cont_1(void)
{
    lv_obj_t * cont;

    cont = lv_cont_create(lv_scr_act(), NULL);
    lv_obj_set_auto_realign(cont, true); /*Auto realign when the
↪size changes*/
    lv_obj_align_origo(cont, NULL, LV_ALIGN_CENTER, 0, 0); /*This parametrs will be
↪sued when realigned*/
    lv_cont_set_fit(cont, LV_FIT_TIGHT);
```

(continues on next page)



(continued from previous page)

```
lv_cont_set_layout(cont, LV_LAYOUT_COL_M);

lv_obj_t * label;
label = lv_label_create(cont, NULL);
lv_label_set_text(label, "Short text");

label = lv_label_create(cont, NULL);
lv_label_set_text(label, "It is a long text");

label = lv_label_create(cont, NULL);
lv_label_set_text(label, "Here is an even longer text");
}
```

## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_layout_t
typedef uint8_t lv_fit_t
typedef uint8_t lv_cont_style_t
```

### Enums

```
enum [anonymous]
    Container layout options
    Values:
    LV_LAYOUT_OFF = 0
        No layout
    LV_LAYOUT_CENTER
        Center objects
    LV_LAYOUT_COL_L
        Column left align
    LV_LAYOUT_COL_M
        Column middle align
    LV_LAYOUT_COL_R
        Column right align
    LV_LAYOUT_ROW_T
        Row top align
    LV_LAYOUT_ROW_M
        Row middle align
```

## **LV\_LAYOUT\_ROW\_B**

Row bottom align

## **LV\_LAYOUT\_PRETTY**

Put as many object as possible in row and begin a new row

## **LV\_LAYOUT\_GRID**

Align same-sized object into a grid

## **\_LV\_LAYOUT\_NUM**

**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\_FLOOD**

Align the size to the parent's edge

## **LV\_FIT\_FILL**

Align the size to the parent's edge first but if there is an object out of it then get larger

## **\_LV\_FIT\_NUM**

**enum** [anonymous]

*Values:*

## **LV\_CONT\_STYLE\_MAIN**

## 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**: bottom 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 tell 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 tell how to change the container's size automatically.

#### Parameters

- **cont**: pointer to a container object
- **fit**: fit policy from `lv_fit_t`

**static void lv\_cont\_set\_style**(*lv\_obj\_t \*cont, lv\_cont\_style\_t type, const lv\_style\_t \*style*)

Set the style of a container

#### Parameters

- **cont**: pointer to a container object
- **type**: which style should be set (can be only `LV_CONT_STYLE_MAIN`)
- **style**: pointer to the new style

*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

**static const** `lv_style_t *lv_cont_get_style(const lv_obj_t *cont, lv_cont_style_t type)`

Get the style of a container

**Return** pointer to the container's style

**Parameters**

- **cont**: pointer to a container object
- **type**: which style should be get (can be only `LV_CONT_STYLE_MAIN`)

**struct** `lv_cont_ext_t`

**Public Members**

`uint8_t layout`

`uint8_t fit_left`

`uint8_t fit_right`

`uint8_t fit_top`

`uint8_t fit_bottom`

**Drop down list (`lv_ddlist`)**

**Overview**

Drop Down Lists allow you to simply select one option from more. The Drop Down List is closed by default and show the currently selected text. If you click on it the list opens and all the options are shown.

**Set options**

The options are passed to the Drop Down List as a string with `lv_ddlist_set_options(ddlist, options)`. The options should be separated by `\n`. For example: "First\nSecond\nThird".

You can select an option manually with `lv_ddlist_set_selected(ddlist, id)`, where *id* is the index of an option.

### Get selected option

The get the currently selected option use `lv_ddlist_get_selected(ddlist)` it will return the *index* of the selected option.

`lv_ddlist_get_selected_str(ddlist, buf, buf_size)` copies the name of the selected option to `buf`.

### Align the options

To align the label horizontally use `lv_ddlist_set_align(ddlist, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`.

### Height and width

By default, the list's height is adjusted automatically to show all options. The `lv_ddlist_set_fix_height(ddlist, height)` sets a fixed height for the opened list. `0` means to use auto height.

The width is also adjusted automatically. To prevent this apply `lv_ddlist_set_fix_width(ddlist, width)`. `0` means to use auto width.

### Scrollbars

Similarly to *Page* with fix height the Drop Down List supports various scrollbar display modes. It can be set by `lv_ddlist_set_sb_mode(ddlist, LV_SB_MODE_...)`.

### Animation time

The Drop Down List open/close animation time is adjusted by `lv_ddlist_set_anim_time(ddlist, anim_time)`. Zero animation time means no animation.

### Decoration arrow

A down arrow can be added to the left side of the Drop down list with `lv_ddlist_set_draw_arrow(ddlist, true)`.

### Stay open

You can force the Drop down list to **stay opened** when an option is selected with `lv_ddlist_set_stay_open(ddlist, true)`.

### Styles

The `lv_ddlist_set_style(ddlist, LV_DDLIST_STYLE_..., &style)` set the styles of a Drop Down List.

- **LV\_DDLIST\_STYLE\_BG** Style of the background. All `style.body` properties are used. `style.text` is used for the option's label. Default: `lv_style_pretty`
- **LV\_DDLIST\_STYLE\_SEL** Style of the selected option. The `style.body` properties are used. The selected option will be recolored with `text.color`. Default: `lv_style_plain_color`
- **LV\_DDLIST\_STYLE\_SB** Style of the scrollbar. The `style.body` properties are used. Default: `lv_style_plain_color`

## Events

Besides the [Generic events](#) the following [Special events](#) are sent by the Drop down lists:

- **LV\_EVENT\_VALUE\_CHANGED** sent when the a new option is selected

Learn more about *Events*.

## 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
- **LV\_KEY\_ENTER** Apply the selected option (Send `LV_EVENT_VALUE_CHANGED` event and close the Drop down list)

## Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        char buf[32];
        lv_ddlist_get_selected_str(obj, buf, sizeof(buf));
        printf("Option: %s\n", buf);
    }
}

void lv_ex_ddlist_1(void)
{
    /*Create a drop down list*/
    lv_obj_t * ddlist = lv_ddlist_create(lv_scr_act(), NULL);
    lv_ddlist_set_options(ddlist, "Apple\n"
        "Banana\n"
        "Orange\n"
        "Melon\n"
        "Grape\n"
        "Raspberry");

    lv_ddlist_set_fix_width(ddlist, 150);
    lv_ddlist_set_draw_arrow(ddlist, true);
    lv_obj_align(ddlist, NULL, LV_ALIGN_IN_TOP_MID, 0, 20);
    lv_obj_set_event_cb(ddlist, event_handler);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_ddlist\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_DDLIST\_STYLE\_BG**  
**LV\_DDLIST\_STYLE\_SEL**  
**LV\_DDLIST\_STYLE\_SB**

## Functions

*lv\_obj\_t* \***lv\_ddlist\_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\_ddlist\_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"

void **lv\_ddlist\_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\_ddlist\_set\_fix\_height**(*lv\_obj\_t* \**ddlist*, lv\_coord\_t *h*)

Set a fix height for the drop down list If 0 then the opened ddlist will be auto. sized else the set height will be applied.

### Parameters

- **ddlist**: pointer to a drop down list
- **h**: the height when the list is opened (0: auto size)

void **lv\_ddlist\_set\_fix\_width**(*lv\_obj\_t* \**ddlist*, lv\_coord\_t *w*)

Set a fix width for the drop down list

### Parameters

- **ddlist**: pointer to a drop down list
- **w**: the width when the list is opened (0: auto size)

void **lv\_ddlist\_set\_draw\_arrow**(*lv\_obj\_t* \**ddlist*, bool *en*)

Set arrow draw in a drop down list

### Parameters

- **ddlist**: pointer to drop down list object
- **en**: enable/disable a arrow draw. E.g. "true" for draw.

void **lv\_ddlist\_set\_stay\_open**(*lv\_obj\_t* \**ddlist*, bool *en*)

Leave the list opened when a new value is selected

### Parameters

- **ddlist**: pointer to drop down list object
- **en**: enable/disable "stay open" feature



**static void lv\_ddlist\_set\_sb\_mode**(*lv\_obj\_t \*ddlist, lv\_sb\_mode\_t mode*)  
Set the scroll bar mode of a drop down list

**Parameters**

- **ddlist**: pointer to a drop down list object
- **sb\_mode**: the new mode from 'lv\_page\_sb\_mode\_t' enum

**static void lv\_ddlist\_set\_anim\_time**(*lv\_obj\_t \*ddlist, uint16\_t anim\_time*)  
Set the open/close animation time.

**Parameters**

- **ddlist**: pointer to a drop down list
- **anim\_time**: open/close animation time [ms]

**void lv\_ddlist\_set\_style**(*lv\_obj\_t \*ddlist, lv\_ddlist\_style\_t type, const lv\_style\_t \*style*)  
Set a style of a drop down list

**Parameters**

- **ddlist**: pointer to a drop down list object
- **type**: which style should be set
- **style**: pointer to a style

**void lv\_ddlist\_set\_align**(*lv\_obj\_t \*ddlist, lv\_label\_align\_t align*)  
Set the alignment of the labels in a drop down list

**Parameters**

- **ddlist**: pointer to a drop down list object
- **align**: alignment of labels

**const char \*lv\_ddlist\_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\_ddlist\_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

**void lv\_ddlist\_get\_selected\_str**(**const** *lv\_obj\_t \*ddlist, char \*buf, uint16\_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\_ddlist\_get\_fix\_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

bool **lv\_ddlist\_get\_draw\_arrow**(lv\_obj\_t \*ddlist)

Get arrow draw in a drop down list

**Parameters**

- **ddlist**: pointer to drop down list object

bool **lv\_ddlist\_get\_stay\_open**(lv\_obj\_t \*ddlist)

Get whether the drop down list stay open after selecting a value or not

**Parameters**

- **ddlist**: pointer to drop down list object

static lv\_sb\_mode\_t **lv\_ddlist\_get\_sb\_mode**(const lv\_obj\_t \*ddlist)

Get the scroll bar mode of a drop down list

**Return** scrollbar mode from 'lv\_page\_sb\_mode\_t' enum

**Parameters**

- **ddlist**: pointer to a drop down list object

static uint16\_t **lv\_ddlist\_get\_anim\_time**(const lv\_obj\_t \*ddlist)

Get the open/close animation time.

**Return** open/close animation time [ms]

**Parameters**

- **ddlist**: pointer to a drop down list

const lv\_style\_t \***lv\_ddlist\_get\_style**(const lv\_obj\_t \*ddlist, lv\_ddlist\_style\_t type)

Get a style of a drop down list

**Return** style pointer to a style

**Parameters**

- **ddlist**: pointer to a drop down list object
- **type**: which style should be get

lv\_label\_align\_t **lv\_ddlist\_get\_align**(const lv\_obj\_t \*ddlist)

Get the alignment of the labels in a drop down list

**Return** alignment of labels

**Parameters**

- **ddlist**: pointer to a drop down list object

void **lv\_ddlist\_open**(lv\_obj\_t \*ddlist, lv\_anim\_enable\_t anim)

Open the drop down list with or without animation

**Parameters**

- **ddlist**: pointer to drop down list object
- **anim\_en**: LV\_ANIM\_ON: use animation; LV\_ANOM\_OFF: not use animations

void **lv\_ddlist\_close**(lv\_obj\_t \*ddlist, lv\_anim\_enable\_t anim)

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\_ddlist\_ext\_t**

### Public Members

```

lv_page_ext_t page
lv_obj_t *label
const lv_style_t *sel_style
uint16_t option_cnt
uint16_t sel_opt_id
uint16_t sel_opt_id_ori
uint8_t opened
uint8_t force_sel
uint8_t draw_arrow
uint8_t stay_open
lv_coord_t fix_height

```

## Gauge (lv\_gauge)

### Overview

The gauge is a meter with scale labels and needles.

### 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.

### 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.

### 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 `line.color` after this value. (default: 80)

## Styles

The gauge uses one style which can be set by `lv_gauge_set_style(gauge, LV_GAUGE_STYLE_MAIN, &style)`. The gauge's properties are derived from the following style attributes:

- **body.main\_color** line's color at the beginning of the scale
- **body.grad\_color** line's color at the end of the scale (gradient with main color)
- **body.padding.left** line length
- **body.padding.inner** label distance from the scale lines
- **body.radius** radius of needle origin circle
- **line.width** line width
- **line.color** line's color after the critical value
- **text.font/color/letter\_space** label attributes

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

# C



code

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

void lv_ex_gauge_1(void)
{
    /*Create a style*/
    static lv_style_t style;
    lv_style_copy(&style, &lv_style_pretty_color);
    style.body.main_color = lv_color_hex3(0x666); /*Line color at the beginning*/
    style.body.grad_color = lv_color_hex3(0x666); /*Line color at the end*/
    style.body.padding.left = 10; /*Scale line length*/
    style.body.padding.inner = 8; /*Scale label padding*/
    style.body.border.color = lv_color_hex3(0x333); /*Needle middle circle color*/
    style.line.width = 3;
    style.text.color = lv_color_hex3(0x333);
    style.line.color = LV_COLOR_RED; /*Line color after the critical
    ↪value*/

    /*Describe the color for the needles*/
    static lv_color_t needle_colors[] = {LV_COLOR_BLUE, LV_COLOR_ORANGE, LV_COLOR_
    ↪PURPLE};

    /*Create a gauge*/
    lv_obj_t * gauge1 = lv_gauge_create(lv_scr_act(), NULL);
    lv_gauge_set_style(gauge1, LV_GAUGE_STYLE_MAIN, &style);
    lv_gauge_set_needle_count(gauge1, 3, needle_colors);
    lv_obj_set_size(gauge1, 150, 150);
    lv_obj_align(gauge1, NULL, LV_ALIGN_CENTER, 0, 20);

    /*Set the values*/
    lv_gauge_set_value(gauge1, 0, 10);
    lv_gauge_set_value(gauge1, 1, 20);
}
```

(continues on next page)

(continued from previous page)

```
    lv_gauge_set_value(gauge1, 2, 30);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_gauge\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_GAUGE\_STYLE\_MAIN**

### 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, int16\_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*, int16\_t *min*, int16\_t *max*)

Set minimum and the maximum values of a gauge

**Parameters**

- **gauge**: pointer to the gauge object
- **min**: minimum value
- **max**: maximum value

**static void lv\_gauge\_set\_critical\_value**(*lv\_obj\_t \*gauge*, int16\_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. The get a given "subdivision" lines between label, **line\_cnt** = (sub\_div + 1) \* (label\_cnt - 1) + 1
- **label\_cnt**: count of scale labels.

**static void lv\_gauge\_set\_style**(*lv\_obj\_t \*gauge*, *lv\_gauge\_style\_t type*, *lv\_style\_t \*style*)

Set the styles of a gauge

**Parameters**

- **gauge**: pointer to a gauge object
- **type**: which style should be set (can be only LV\_GAUGE\_STYLE\_MAIN)
- **style**: set the style of the gauge

**int16\_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 int16\_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** int16\_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** int16\_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** uint8\_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** const lv\_style\_t \***lv\_gauge\_get\_style**(const lv\_obj\_t \*gauge, lv\_gauge\_style\_t type)

Get the style of a gauge

**Return** pointer to the gauge's style

#### Parameters

- **gauge**: pointer to a gauge object
- **type**: which style should be get (can be only LV\_GAUGE\_STYLE\_MAIN)

**struct** lv\_gauge\_ext\_t



## Public Members

```

lv_lmeter_ext_t lmeter
int16_t *values
const lv_color_t *needle_colors
uint8_t needle_count
uint8_t label_count

```

## Image (lv\_img)

### Overview

The Images are the basic object to display images.

### 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 LittlevGL'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 source form 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 for the same thing. E.g.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 then 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.

## Styles

The images uses one style which can be set by `lv_img_set_style(lmeter, LV_IMG_STYLE_MAIN, &style)`. All the `style.image` properties are used:

- **image.intense** intensity of recoloring (0..255 or *LV\_OPA\_...*)
- **image.color** color for recoloring or color of the alpha indexed images
- **image.opa** overall opacity of image

When the Image object displays a text then `style.text` properties are used. See *Label* for more information.

The images' default style is *NULL* so they **inherit the parent's style**.

## Events

Only the *Generic events* are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code

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

LV_IMG_DECLARE(cogwheel);

void lv_ex_img_1(void)
{
    lv_obj_t * img1 = lv_img_create(lv_scr_act(), NULL);
    lv_img_set_src(img1, &cogwheel);
}
```

(continues on next page)

(continued from previous page)

```
lv_obj_align(img1, NULL, LV_ALIGN_CENTER, 0, -20);

lv_obj_t * img2 = lv_img_create(lv_scr_act(), NULL);
lv_img_set_src(img2, LV_SYMBOL_OK "Accept");
lv_obj_align(img2, img1, LV_ALIGN_OUT_BOTTOM_MID, 0, 20);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_img\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_IMG\_STYLE\_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.

**static** void **lv\_img\_set\_style**(*lv\_obj\_t \*img*, *lv\_img\_style\_t type*, **const** *lv\_style\_t \*style*)

Set the style of an image

**Parameters**

- **img**: pointer to an image object
- **type**: which style should be set (can be only LV\_IMG\_STYLE\_MAIN)
- **style**: pointer to a style

**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

**static const** `lv_style_t *lv_img_get_style(const lv_obj_t *img, lv_img_style_t type)`

Get the style of an image object

**Return** pointer to the image's style

### Parameters

- `img`: pointer to an image object
- `type`: which style should be get (can be only `LV_IMG_STYLE_MAIN`)

**struct** `lv_img_ext_t`

### Public Members

**const** void \*`src`

`lv_point_t` **offset**

`lv_coord_t` **w**

`lv_coord_t` **h**

`uint8_t` **src\_type**

`uint8_t` **auto\_size**

`uint8_t` **cf**

### Image button (`lv_imgbtn`)

#### Overview

The Image button is very similar to the simple Button object. The only difference is it displays user-defined images in each state instead of drawing a button. Before reading this please read the *Button* section too.

#### Image sources

To set the image in a state the `lv_imgbtn_set_src(imgbtn, LV_BTN_STATE_..., &img_src)` The image sources works the same as described in the *Image object*.

If `LV_IMGBTN_TILED` is enabled in `lv_conf.h` three sources can be set for each state:

- left
- center
- right

The *center* 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 while without it the width will be always the same as the image source's width.

## States

The states also work like with Button object. It can be set with `lv_imgbtn_set_state(imgbtn, LV_BTN_STATE_...)`.

## Toggle

The toggle feature can be enabled with `lv_imgbtn_set_toggle(imgbtn, true)`

## Style usage

Similarly to normal Buttons, Image buttons also have 5 independent styles for the 5 state. You can set them via: `lv_imgbtn_set_style(btn, LV_IMGBTN_STYLE_..., &style)`. The styles use the `style.image` properties.

- **LV\_IMGBTN\_STYLE\_REL** style of the released state. Default: `lv_style_btn_rel`
- **LV\_IMGBTN\_STYLE\_PR** style of the pressed state. Default: `lv_style_btn_pr`
- **LV\_IMGBTN\_STYLE\_TGL\_REL** style of the toggled released state. Default: `lv_style_btn_tgl_rel`
- **LV\_IMGBTN\_STYLE\_TGL\_PR** style of the toggled pressed state. Default: `lv_style_btn_tgl_pr`
- **LV\_IMGBTN\_STYLE\_INA** style of the inactive state. Default: `lv_style_btn_ina`

When labels are created on a button, it's a good practice to set the image button's `style.text` properties too. Because labels have `style = NULL` by default they inherit the parent's (image button) style. Hence you don't need to create a new style for the label.

## Events

Besided 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*.

## 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*.

## Example

C



code

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

void lv_ex_imgbtn_1(void)
{
    lv_style_t style_pr;
    lv_style_copy(&style_pr, &lv_style_plain);
    style_pr.image.color = LV_COLOR_BLACK;
    style_pr.image.intense = LV_OPA_50;
    style_pr.text.color = lv_color_hex3(0xaa);

    LV_IMG_DECLARE(imgbtn_green);
    LV_IMG_DECLARE(imgbtn_blue);

    /*Create an Image button*/
    lv_obj_t * imgbtn1 = lv_imgbtn_create(lv_scr_act(), NULL);
    lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_REL, &imgbtn_green);
    lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_PR, &imgbtn_green);
    lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_TGL_REL, &imgbtn_blue);
    lv_imgbtn_set_src(imgbtn1, LV_BTN_STATE_TGL_PR, &imgbtn_blue);
    lv_imgbtn_set_style(imgbtn1, LV_BTN_STATE_PR, &style_pr); /*Use the darker
↪ style in the pressed state*/
    lv_imgbtn_set_style(imgbtn1, LV_BTN_STATE_TGL_PR, &style_pr);
    lv_imgbtn_set_toggle(imgbtn1, true);
    lv_obj_align(imgbtn1, NULL, LV_ALIGN_CENTER, 0, -40);

    /*Create a label on the Image button*/
    lv_obj_t * label = lv_label_create(imgbtn1, NULL);
    lv_label_set_text(label, "Button");
}
```



## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_imgbtn_style_t
```

### Enums

**enum** [anonymous]

*Values:*

**LV\_IMGBTN\_STYLE\_REL**

Same meaning as ordinary button styles.

**LV\_IMGBTN\_STYLE\_PR**

**LV\_IMGBTN\_STYLE\_TGL\_REL**

**LV\_IMGBTN\_STYLE\_TGL\_PR**

**LV\_IMGBTN\_STYLE\_INA**

### 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(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\_toggle**(*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

**void lv\_imgbtn\_set\_style**(*lv\_obj\_t \*imgbtn, lv\_imgbtn\_style\_t type, const lv\_style\_t \*style*)

Set a style of a image button.

**Parameters**

- **imgbtn**: pointer to image button object
- **type**: which style should be set
- **style**: pointer to a style

**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\_toggle**(const lv\_obj\_t \*imgbtn)

Get the toggle enable attribute of the image button

**Return** ture: toggle enabled, false: disabled

**Parameters**

- **imgbtn**: pointer to a image button object

**const** lv\_style\_t \***lv\_imgbtn\_get\_style**(const lv\_obj\_t \*imgbtn, lv\_imgbtn\_style\_t type)

Get style of a image button.

**Return** style pointer to the style

**Parameters**

- **imgbtn**: pointer to image button object
- **type**: which style should be get

**struct** **lv\_imgbtn\_ext\_t**

**Public Members**

lv\_btn\_ext\_t **btn**

**const** void \***img\_src**[\_LV\_BTN\_STATE\_NUM]

**const** void \***img\_src\_left**[\_LV\_BTN\_STATE\_NUM]

**const** void \***img\_src\_mid**[\_LV\_BTN\_STATE\_NUM]

**const** void \***img\_src\_right**[\_LV\_BTN\_STATE\_NUM]

lv\_img\_cf\_t **act\_cf**

## Keyboard (lv\_kb)

### Overview

The Keyboard object is a special *Button matrix* with predefined keymaps and other features to realize a virtual keyboard to write text.

### Modes

The Keyboards have two modes:

- **LV\_KB\_MODE\_TEXT** display letters, number, and special characters
- **LV\_KB\_MODE\_NUM** display numbers, +/- sign and decimal dot

To set the mode use `lv_kb_set_mode(kb, mode)`. The default is `LV_KB_MODE_TEXT`

### 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_kb_set_ta(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's 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_kb_set_cursor_manage(kb, true)`. The default is not managed.

### New key map

You can specify a new map (layout) for the keyboard with `lv_kb_set_map(kb, map)`. and `lv_kb_set_ctrl_map(kb, ctrl_map)`. Learn more about in the *Button matrix* object. Keep in mind using following keywords will have the same effect as with the original map:

- `LV_SYMBOL_OK` Apply
- `SYMBOL_CLOSE` Close
- `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
- `"Bkps"` Delete on the left

### Styles

The Keyboards work with 6 styles: a background and 5 button styles for each state. You can set the styles with `lv_kb_set_style(btn, LV_KB_STYLE_..., &style)`. The background and the buttons use the `style.body` properties. The labels use the `style.text` properties of the buttons' styles.

- **LV\_KB\_STYLE\_BG** Background style. Uses all **style.body** properties including **padding**. Default: **lv\_style\_pretty**
- **LV\_KB\_STYLE\_BTN\_REL** style of the released buttons. Default: **lv\_style\_btn\_rel**
- **LV\_KB\_STYLE\_BTN\_PR** style of the pressed buttons. Default: **lv\_style\_btn\_pr**
- **LV\_KB\_STYLE\_BTN\_TGL\_REL** style of the toggled released buttons. Default: **lv\_style\_btn\_tgl\_rel**
- **LV\_KB\_STYLE\_BTN\_TGL\_PR** style of the toggled pressed buttons. Default: **lv\_style\_btn\_tgl\_pr**
- **LV\_KB\_STYLE\_BTN\_INA** style of the inactive buttons. Default: **lv\_style\_btn\_ina**

## 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 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\_kb\_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 but you can call **lv\_kb\_def\_event\_cb** at the beginning of your event handler to handle the same things as before.

Learn more about *Events*.

## Keys

The following *Keys* are processed by the Buttons:

- **LV\_KEY\_RIGHT/UP/LEFT/RIGHT** To navigate among the buttons and elect one
- **LV\_KEY\_ENTER** To press/release the selected button

Learn more about *Keys*.

## Examples

C



code

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

void lv_ex_kb_1(void)
{
    /*Create styles for the keyboard*/
    static lv_style_t rel_style, pr_style;

    lv_style_copy(&rel_style, &lv_style_btn_rel);
    rel_style.body.radius = 0;
    rel_style.body.border.width = 1;

    lv_style_copy(&pr_style, &lv_style_btn_pr);
    pr_style.body.radius = 0;
    pr_style.body.border.width = 1;

    /*Create a keyboard and apply the styles*/
    lv_obj_t *kb = lv_kb_create(lv_scr_act(), NULL);
    lv_kb_set_cursor_manage(kb, true);
    lv_kb_set_style(kb, LV_KB_STYLE_BG, &lv_style_transp_tight);
    lv_kb_set_style(kb, LV_KB_STYLE_BTN_REL, &rel_style);
    lv_kb_set_style(kb, LV_KB_STYLE_BTN_PR, &pr_style);

    /*Create a text area. The keyboard will write here*/
    lv_obj_t *ta = lv_ta_create(lv_scr_act(), NULL);
    lv_obj_align(ta, NULL, LV_ALIGN_IN_TOP_MID, 0, 10);
    lv_ta_set_text(ta, "");

    /*Assign the text area to the keyboard*/
    lv_kb_set_ta(kb, ta);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_kb_mode_t
typedef uint8_t lv_kb_style_t
```

### Enums

```
enum [anonymous]
    Current keyboard mode.
```

*Values:*

```
LV_KB_MODE_TEXT
LV_KB_MODE_NUM
```

```
enum [anonymous]
```

*Values:*

```
LV_KB_STYLE_BG
LV_KB_STYLE_BTN_REL
LV_KB_STYLE_BTN_PR
LV_KB_STYLE_BTN_TGL_REL
LV_KB_STYLE_BTN_TGL_PR
LV_KB_STYLE_BTN_INA
```

### Functions

```
lv_obj_t *lv_kb_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_kb_set_ta(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\_kb\_set\_mode**(*lv\_obj\_t \*kb*, *lv\_kb\_mode\_t mode*)

Set a new a mode (text or number map)

**Parameters**

- **kb**: pointer to a Keyboard object
- **mode**: the mode from 'lv\_kb\_mode\_t'

void **lv\_kb\_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

**static** void **lv\_kb\_set\_map**(*lv\_obj\_t \*kb*, **const** char *\*map*[])

Set a new map for the keyboard

**Parameters**

- **kb**: pointer to a Keyboard object
- **map**: pointer to a string array to describe the map. See 'lv\_btnm\_set\_map()' for more info.

**static** void **lv\_kb\_set\_ctrl\_map**(*lv\_obj\_t \*kb*, **const** *lv\_btnm\_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
- **ctrl\_map**: pointer to an array of *lv\_btn\_ctrl\_t* control bytes. See: *lv\_btnm\_set\_ctrl\_map* for more details.

void **lv\_kb\_set\_style**(*lv\_obj\_t \*kb*, *lv\_kb\_style\_t type*, **const** *lv\_style\_t \*style*)

Set a style of a keyboard

**Parameters**

- **kb**: pointer to a keyboard object
- **type**: which style should be set
- **style**: pointer to a style

*lv\_obj\_t \****lv\_kb\_get\_ta**(**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\_kb\_mode\_t* **lv\_kb\_get\_mode**(**const** *lv\_obj\_t \*kb*)

Set a new a mode (text or number map)

**Return** the current mode from 'lv\_kb\_mode\_t'

**Parameters**

- **kb**: pointer to a Keyboard object



bool **lv\_kb\_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\_kb\_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

const *lv\_style\_t* \***lv\_kb\_get\_style**(const *lv\_obj\_t* \*kb, *lv\_kb\_style\_t* type)

Get a style of a keyboard

**Return** style pointer to a style

**Parameters**

- **kb**: pointer to a keyboard object
- **type**: which style should be get

void **lv\_kb\_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\_kb\_ext\_t**

**Public Members**

*lv\_btnm\_ext\_t* **btnm**

*lv\_obj\_t* \***ta**

*lv\_kb\_mode\_t* **mode**

uint8\_t **cursor\_mng**

**Label (lv\_label)**

**Overview**

The Labels are the basic objects to display text.

## Set text

You can modify the text in run-time at any time with `lv_label_set_text(label, "New text")`. It will allocate the text dynamically.

Labels are able to show text from a **static array**. Use: `lv_label_set_static_text(label, char_array)`. In this case, the text is not stored in the dynamic memory but the given array is used directly instead. Keep in my the array can't be a local variable which destroys when the function exits.

You can also use a **raw character 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. To set a raw character array use the `lv_label_set_array_text(label, char_array)` function.

## Line break

You can use `\n` to make line break. For example: `"line1\nline2\n\nline4"`

## Long modes

The size of the label object can be automatically expanded to the text size or 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\_DOTS** Keep the object size, break the text and write dots in the last line
- **LV\_LABEL\_LONG\_ROLL** Keep the size and scroll the label back and forth
- **LV\_LABEL\_LONG\_ROLL\_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 and then set the size with `lv_obj_set_width/height/size()`.

## Text align

The label's text can be aligned to the left, right or middle with `lv_label_set_align(label, LV_LABEL_ALIGN_LEFT/RIGHT/CENTER)`

## Draw background

You can enable to draw a background for the label with `lv_label_set_body_draw(label, draw)`

## Text recolor

In the text, you can use commands to re-color 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. I.e. there can't be `\n` in a recolored text or it can be wrapped by `LV_LABEL_LONG_BREAK` else the text in the new line won't be recolored.

## Very long texts

LittlevGL can effectively 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 besides letters. Read the *Font* section to learn more about the symbols.

## Styles

The Label uses one style which can be set by `lv_label_set_style(label, LV_LABEL_STYLE_MAIN, &style)`. From the style the following properties are used:

- all properties from `style.text`
- for background drawing `style.body` properties. `padding` will increase the size only visually, the real object's size won't be changed.

The labels' default style is `NULL` so they inherit the parent's style.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

## C

Re-color words of a  
label and wrap long  
text automatically.

. It is a circularly scr

code

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

void lv_ex_label_1(void)
{
    lv_obj_t * label1 = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_long_mode(label1, LV_LABEL_LONG_BREAK); /*Break the long lines*/
    lv_label_set_recolor(label1, true); /*Enable re-coloring by ↵
↵commands in the text*/
    lv_label_set_align(label1, LV_LABEL_ALIGN_CENTER); /*Center aligned lines*/
    lv_label_set_text(label1, "#000080 Re-color# #0000ff words# #6666ff of a# label "
        "and wrap long text automatically.");
    lv_obj_set_width(label1, 150);
    lv_obj_align(label1, NULL, LV_ALIGN_CENTER, 0, -30);

    lv_obj_t * label2 = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_long_mode(label2, LV_LABEL_LONG_SCROLL_CIRC); /*Circular scroll*/
    lv_obj_set_width(label2, 150);
    lv_label_set_text(label2, "It is a circularly scrolling text. ");
    lv_obj_align(label2, NULL, LV_ALIGN_CENTER, 0, 30);
}
```

## MicroPython

No examples yet.

## API

## Typedefs

**typedef** uint8\_t **lv\_label\_long\_mode\_t**

**typedef** uint8\_t **lv\_label\_align\_t**

**typedef** uint8\_t **lv\_label\_style\_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\_SCROLL**

Keep the size and roll the text back and forth

**LV\_LABEL\_LONG\_SCROLL\_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

**enum** [anonymous]

Label styles

*Values:*

**LV\_LABEL\_STYLE\_MAIN**

## Functions

*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\_array\_text**(*lv\_obj\_t \*label*, **const** char *\*array*, uint16\_t *size*)

Set a new text for a label from a character array. The array don't has to be '\0' terminated. Memory will be allocated to store the array by the label.

#### Parameters

- **label**: pointer to a label object
- **array**: array of characters or NULL to refresh the label
- **size**: the size of 'array' in bytes

void **lv\_label\_set\_static\_text**(*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\_RIGHT'

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\_body\_draw**(*lv\_obj\_t \*label*, bool *en*)

Set the label to draw (or not draw) background specified in its style's body

#### Parameters

- **label**: pointer to a label object
- **en**: true: draw body; false: don't draw body

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

**static** void **lv\_label\_set\_style**(*lv\_obj\_t \*label*, *lv\_label\_style\_t type*, **const** *lv\_style\_t \*style*)

Set the style of an label

#### Parameters

- **label**: pointer to an label object
- **type**: which style should be get (can be only LV\_LABEL\_STYLE\_MAIN)
- **style**: pointer to a style

void **lv\_label\_set\_text\_sel\_start**(*lv\_obj\_t \*label*, uint16\_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*, uint16\_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

bool **lv\_label\_get\_body\_draw**(const lv\_obj\_t \*label)

Get the body draw attribute

**Return** true: draw body; false: don't draw body

**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, uint16\_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)

uint16\_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 characte under

static const lv\_style\_t \***lv\_label\_get\_style**(const lv\_obj\_t \*label, lv\_label\_style\_t type)

Get the style of an label object

**Return** pointer to the label's style



#### Parameters

- **label**: pointer to an label object
- **type**: which style should be get (can be only LV\_LABEL\_STYLE\_MAIN)

uint16\_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.

uint16\_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.

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**[sizeof(char \*)]

**union** lv\_label\_ext\_t::[anonymous] **dot**

uint16\_t **dot\_end**

lv\_point\_t **offset**

```

lv_draw_label_hint_t hint
uint16_t anim_speed
uint16_t txt_sel_start
uint16_t txt_sel_end
lv_label_long_mode_t long_mode
uint8_t static_txt
uint8_t align
uint8_t recolor
uint8_t expand
uint8_t body_draw
uint8_t dot_tmp_alloc

```

## LED (`lv_led`)

### Overview

The LEDs are rectangle-like (or circle) object.

### 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.

### Styles

The LED uses one style which can be set by `lv_led_set_style(led, LV_LED_STYLE_MAIN, &style)`. To determine the appearance the `style.body` properties are used.

The colors are darkened and shadow width is reduced at a lower brightness and gains its original value at brightness 255 to show a lighting effect.

The default style is: `lv_style_pretty_color`. Not that, the LED doesn't really look like a LED with the default style so you should create your own style. See the example below.

### Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

### C



code

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

void lv_ex_led_1(void)
{
    /*Create a style for the LED*/
    static lv_style_t style_led;
    lv_style_copy(&style_led, &lv_style_pretty_color);
    style_led.body.radius = LV_RADIUS_CIRCLE;
    style_led.body.main_color = LV_COLOR_MAKE(0xb5, 0x0f, 0x04);
    style_led.body.grad_color = LV_COLOR_MAKE(0x50, 0x07, 0x02);
    style_led.body.border.color = LV_COLOR_MAKE(0xfa, 0x0f, 0x00);
    style_led.body.border.width = 3;
    style_led.body.border.opa = LV_OPA_30;
    style_led.body.shadow.color = LV_COLOR_MAKE(0xb5, 0x0f, 0x04);
    style_led.body.shadow.width = 5;

    /*Create a LED and switch it ON*/
    lv_obj_t * led1 = lv_led_create(lv_scr_act(), NULL);
    lv_obj_set_style(led1, &style_led);
    lv_obj_align(led1, NULL, LV_ALIGN_CENTER, -80, 0);
    lv_led_on(led1);
}
```

(continues on next page)

(continued from previous page)

```

/*Copy the previous LED and set a brightness*/
lv_obj_t * led2 = lv_led_create(lv_scr_act(), led1);
lv_obj_align(led2, NULL, LV_ALIGN_CENTER, 0, 0);
lv_led_set_bright(led2, 190);

/*Copy the previous LED and switch it OFF*/
lv_obj_t * led3 = lv_led_create(lv_scr_act(), led1);
lv_obj_align(led3, NULL, LV_ALIGN_CENTER, 80, 0);
lv_led_on(led3);
}

```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_led\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_LED\_STYLE\_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**: 0 (max. dark) ... 255 (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

**static** void **lv\_led\_set\_style**(*lv\_obj\_t \*led*, *lv\_led\_style\_t type*, **const** *lv\_style\_t \*style*)  
Set the style of a led

#### Parameters

- **led**: pointer to a led object
- **type**: which style should be set (can be only LV\_LED\_STYLE\_MAIN)
- **style**: pointer to a style

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

**static const** *lv\_style\_t \****lv\_led\_get\_style**(**const** *lv\_obj\_t \*led*, *lv\_led\_style\_t type*)  
Get the style of an led object

**Return** pointer to the led's style

#### Parameters

- **led**: pointer to an led object
- **type**: which style should be get (can be only LV\_CHART\_STYLE\_MAIN)

**struct lv\_led\_ext\_t**

#### Public Members

uint8\_t **bright**

## Line (lv\_line)

### Overview

The Line object is capable of drawing straight lines between a set of points.

## 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. You can enable it 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 default, the  $y == 0$  point is in the top of the object but you can invert the y coordinates with `lv_line_set_y_invert(line, true)`. The *y invert* is disabled by default.

## Styles

The Line uses one style which can be set by `lv_line_set_style(lcd, LV_LINE_STYLE_MAIN, &style)` and it uses all `style.line` properties.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

## C



code

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

void lv_ex_line_1(void)
{
    /*Create an array for the points of the line*/
    static lv_point_t line_points[] = { {5, 5}, {70, 70}, {120, 10}, {180, 60}, {240, 10} };

    /*Create new style (thick dark blue)*/
    static lv_style_t style_line;
    lv_style_copy(&style_line, &lv_style_plain);
    style_line.line.color = LV_COLOR_MAKE(0x00, 0x3b, 0x75);
    style_line.line.width = 3;
    style_line.line.rounded = 1;

    /*Copy the previous line and apply the new style*/
    lv_obj_t * line1;
    line1 = lv_line_create(lv_scr_act(), NULL);
    lv_line_set_points(line1, line_points, 5); /*Set the points*/
    lv_line_set_style(line1, LV_LINE_STYLE_MAIN, &style_line);
    lv_obj_align(line1, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

## MicroPython

No examples yet.

## API

## Typedefs

```
typedef uint8_t lv_line_style_t
```

## Enums

```
enum [anonymous]  
  Values:
```

```
    LV_LINE_STYLE_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

```
static void lv_line_set_style(lv_obj_t *line, lv_line_style_t type, const lv_style_t *style)  
  Set the style of a line
```

### Parameters

- **line**: pointer to a line object
- **type**: which style should be set (can be only LV\_LINE\_STYLE\_MAIN)



- **style**: pointer to a style

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

static const lv\_style\_t \***lv\_line\_get\_style**(const lv\_obj\_t \*line, lv\_line\_style\_t type)

Get the style of an line object

**Return** pointer to the line's style

**Parameters**

- **line**: pointer to an line object
- **type**: which style should be get (can be only LV\_LINE\_STYLE\_MAIN)

**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**

## List (lv\_list)

### 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.

### Add buttons

You can add new list elements 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_SCROLL_CIRC` long mode to automatically scroll the long labels circularly.

You can use `lv_list_get_btn_label(list_btn)` and `lv_list_get_btn_img(list_btn)` to get the label and the image of a list button. You can get the text 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.

### 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, en)` 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)`

If the buttons have `lv_btn_set_toggle` enabled then `lv_list_set_single_mode(list, true)` can be used to ensure that only one button can be in toggled state at the same time.

### Style usage

The `lv_list_set_style(list, LV_LIST_STYLE_..., &style)` function sets the style of a list.

- **LV\_LIST\_STYLE\_BG** list background style. Default: `lv_style_transp_fit`
- **LV\_LIST\_STYLE\_SCRL** scrollable part's style. Default: `lv_style_pretty`
- **LV\_LIST\_STYLE\_SB** scrollbars' style. Default: `lv_style_pretty_color`. For details see *Page*
- **LV\_LIST\_STYLE\_BTN\_REL** button released style. Default: `lv_style_btn_rel`
- **LV\_LIST\_STYLE\_BTN\_PR** button pressed style. Default: `lv_style_btn_pr`
- **LV\_LIST\_STYLE\_BTN\_TGL\_REL** button toggled released style. Default: `lv_style_btn_tgl_rel`
- **LV\_LIST\_STYLE\_BTN\_TGL\_PR** button toggled pressed style. Default: `lv_style_btn_tgl_pr`
- **LV\_LIST\_STYLE\_BTN\_INA** button inactive style. Default: `lv_style_btn_ina`

Because *BG* has a transparent style by default if there is only a few buttons the list will look shorter but become scrollable when more list elements are added.

To modify the height of the buttons adjust the `body.padding.top/bottom` fields of the corresponding styles (`LV_LIST_STYLE_BTN_REL/PR/...`)

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## 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.

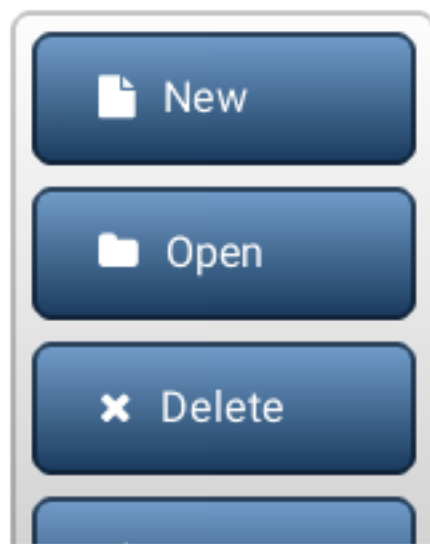
The Selected buttons are in `LV_BTN_STATE_PR/TG_PR` state.

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*.

## Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_CLICKED) {
        printf("Clicked: %s\n", lv_list_get_btn_text(obj));
    }
}

void lv_ex_list_1(void)
{
    /*Create a list*/
    lv_obj_t * list1 = lv_list_create(lv_scr_act(), NULL);
    lv_obj_set_size(list1, 160, 200);
    lv_obj_align(list1, NULL, LV_ALIGN_CENTER, 0, 0);

    /*Add buttons to the list*/

    lv_obj_t * list_btn;

    list_btn = lv_list_add_btn(list1, LV_SYMBOL_FILE, "New");
    lv_obj_set_event_cb(list_btn, event_handler);

    list_btn = lv_list_add_btn(list1, LV_SYMBOL_DIRECTORY, "Open");
    lv_obj_set_event_cb(list_btn, event_handler);

    list_btn = lv_list_add_btn(list1, LV_SYMBOL_CLOSE, "Delete");
    lv_obj_set_event_cb(list_btn, event_handler);

    list_btn = lv_list_add_btn(list1, LV_SYMBOL_EDIT, "Edit");
    lv_obj_set_event_cb(list_btn, event_handler);

    list_btn = lv_list_add_btn(list1, LV_SYMBOL_SAVE, "Save");
    lv_obj_set_event_cb(list_btn, event_handler);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_list\_style\_t**

### Enums

**enum** [anonymous]

List styles.

*Values:*

## LV\_LIST\_STYLE\_BG

List background style

## LV\_LIST\_STYLE\_SCRL

List scrollable area style.

## LV\_LIST\_STYLE\_SB

List scrollbar style.

## LV\_LIST\_STYLE\_EDGE\_FLASH

List edge flash style.

## LV\_LIST\_STYLE\_BTN\_REL

Same meaning as the ordinary button styles.

## LV\_LIST\_STYLE\_BTN\_PR

## LV\_LIST\_STYLE\_BTN\_TGL\_REL

## LV\_LIST\_STYLE\_BTN\_TGL\_PR

## LV\_LIST\_STYLE\_BTN\_INA

## 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* \**obj*)

Delete all children of the scrl object, without deleting scrl child.

### Parameters

- **obj**: 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  $0 \leq \text{index} < \text{lv\_list\_ext\_t.size}$

void **lv\_list\_set\_single\_mode**(*lv\_obj\_t \*list*, bool *mode*)

Set single button selected mode, only one button will be selected if enabled.

#### Parameters

- **list:** pointer to the currently pressed list object
- **mode:** enable(true)/disable(false) single selected mode.

void **lv\_list\_set\_btn\_selected**(*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\_sb\_mode**(*lv\_obj\_t \*list*, *lv\_sb\_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\_style**(*lv\_obj\_t \*list*, *lv\_list\_style\_t type*, **const** *lv\_style\_t \*style*)

Set a style of a list

#### Parameters

- **list:** pointer to a list object
- **type:** which style should be set
- **style:** pointer to a style

bool **lv\_list\_get\_single\_mode**(lv\_obj\_t \*list)

Get single button selected mode.

**Parameters**

- **list**: pointer to the currently pressed list object.

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

static *lv\_sb\_mode\_t* **lv\_list\_get\_sb\_mode**(const *lv\_obj\_t* \*list)

Get the scroll bar mode of a list

**Return** scrollbar mode from 'lv\_page\_sb\_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

const *lv\_style\_t* \***lv\_list\_get\_style**(const *lv\_obj\_t* \*list, *lv\_list\_style\_t* type)

Get a style of a list

**Return** style pointer to a style

**Parameters**

- **list**: pointer to a list object
- **type**: which style should be get

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\_ANIM\_ON: scroll with animation, LV\_ANIM\_OFF: without animation

**struct lv\_list\_ext\_t**

#### Public Members

*lv\_page\_ext\_t* **page**

const *lv\_style\_t* \***styles\_btn**[LV\_BTN\_STATE\_NUM]

const *lv\_style\_t* \***style\_img**

uint16\_t **size**

uint8\_t **single\_mode**

*lv\_obj\_t* \***last\_sel**

*lv\_obj\_t* \***selected\_btn**

## Line meter (lv\_lmeter)

### Overview

The Line Meter object consists of some radial lines which draw a scale.

### Set value

When setting a new value with `lv_lmeter_set_value(lmeter, new_value)` the proportional part of the scale will be recolored.

### Range and Angles

The `lv_lmeter_set_range(lmeter, 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_lmeter_set_scale(lmeter, angle, line_num)`. The default angle is 240 and the default line number is 31.

## Styles

The line meter uses one style which can be set by `lv_lmeter_set_style(lmeter, LV_LMETER_STYLE_MAIN, &style)`. The line meter's properties are derived from the following style attributes:

- **line.color** “inactive line's” color which are greater then the current value
- **body.main\_color** “active line's” color at the beginning of the scale
- **body.grad\_color** “active line's” color at the end of the scale (gradient with main color)
- **body.padding.hor** line length
- **line.width** line width

The default style is `lv_style_pretty_color`.

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

C



code

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

void lv_ex_lmeter_1(void)
{
    /*Create a style for the line meter*/
    static lv_style_t style_lmeter;
    lv_style_copy(&style_lmeter, &lv_style_pretty_color);
    style_lmeter.line.width = 2;
    style_lmeter.line.color = LV_COLOR_SILVER;
    style_lmeter.body.main_color = lv_color_hex(0x91bfed);           /*Light blue*/
    style_lmeter.body.grad_color = lv_color_hex(0x04386c);          /*Dark blue*/
    style_lmeter.body.padding.left = 16;                             /*Line length*/

    /*Create a line meter */
    lv_obj_t * lmeter;
    lmeter = lv_lmeter_create(lv_scr_act(), NULL);
    lv_lmeter_set_range(lmeter, 0, 100);                             /*Set the range*/
    lv_lmeter_set_value(lmeter, 80);                                  /*Set the current value*/
    lv_lmeter_set_scale(lmeter, 240, 31);                             /*Set the angle and number
↳ of lines*/
    lv_lmeter_set_style(lmeter, LV_LMETER_STYLE_MAIN, &style_lmeter); /
↳ *Apply the new style*/
    lv_obj_set_size(lmeter, 150, 150);
    lv_obj_align(lmeter, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_lmeter\_style\_t**

### Enums

**enum** [anonymous]

Values:

**LV\_LMETER\_STYLE\_MAIN**

### Functions

*lv\_obj\_t* \***lv\_lmeter\_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\_lmeter\_set\_value**(*lv\_obj\_t \*lmeter*, int16\_t *value*)  
Set a new value on the line meter

#### Parameters

- **lmeter**: pointer to a line meter object
- **value**: new value

void **lv\_lmeter\_set\_range**(*lv\_obj\_t \*lmeter*, int16\_t *min*, int16\_t *max*)  
Set minimum and the maximum values of a line meter

#### Parameters

- **lmeter**: pointer to the line meter object
- **min**: minimum value
- **max**: maximum value

void **lv\_lmeter\_set\_scale**(*lv\_obj\_t \*lmeter*, uint16\_t *angle*, uint8\_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

static void **lv\_lmeter\_set\_style**(*lv\_obj\_t \*lmeter*, *lv\_lmeter\_style\_t type*, *lv\_style\_t \*style*)  
Set the styles of a line meter

#### Parameters

- **lmeter**: pointer to a line meter object
- **type**: which style should be set (can be only LV\_LMETER\_STYLE\_MAIN)
- **style**: set the style of the line meter

int16\_t **lv\_lmeter\_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

int16\_t **lv\_lmeter\_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

int16\_t **lv\_lmeter\_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

`uint8_t lv_lmeter_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_lmeter_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

`static const lv_style_t *lv_lmeter_get_style(const lv_obj_t *lmeter, lv_lmeter_style_t type)`

Get the style of a line meter

**Return** pointer to the line meter's style

#### Parameters

- `lmeter`: pointer to a line meter object
- `type`: which style should be get (can be only `LV_LMETER_STYLE_MAIN`)

`struct lv_lmeter_ext_t`

#### Public Members

`uint16_t scale_angle`

`uint8_t line_cnt`

`int16_t cur_value`

`int16_t min_value`

`int16_t max_value`

### Message box (`lv_mbox`)

#### 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` auto fit vertically)-

#### Set text

To set the text use the `lv_mbox_set_text(mbox, "My text")` function.

## Add buttons

To add buttons use the `lv_mbox_add_btns(mbox, btn_str)` function. You need specify the button's text like `const char * btn_str[] = {"Apply", "Close", ""}`. For more information visit the *Button matrix* documentation.

## Auto-close

With `lv_mbox_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)`.

## Styles

Use `lv_mbox_set_style(mbox, LV_MBOX_STYLE_..., &style)` to set a new style for an element of the Message box:

- **LV\_MBOX\_STYLE\_BG** specifies the background container's style. `style.body` sets the background and `style.label` sets the text appearance. Default: `lv_style_pretty`
- **LV\_MBOX\_STYLE\_BTN\_BG** style of the Button matrix background. Default: `lv_style_trans`
- **LV\_MBOX\_STYLE\_BTN\_REL** style of the released buttons. Default: `lv_style_btn_rel`
- **LV\_MBOX\_STYLE\_BTN\_PR** style of the pressed buttons. Default: `lv_style_btn_pr`
- **LV\_MBOX\_STYLE\_BTN\_TGL\_REL** style of the toggled released buttons. Default: `lv_style_btn_tgl_rel`
- **LV\_MBOX\_STYLE\_BTN\_TGL\_PR** style of the toggled pressed buttons. Default: `lv_style_btn_tgl_pr`
- **LV\_MBOX\_STYLE\_BTN\_INA** style of the inactive buttons. Default: `lv_style_btn_ina`

The height of the button area comes from *font height + padding.top + padding.bottom* of `LV_MBOX_STYLE_BTN_REL`.

## 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:

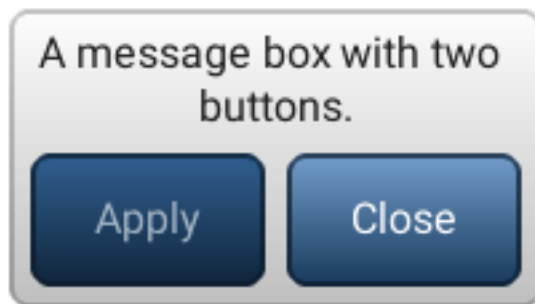
- **LV\_KEY\_RIGHT/DOWN** Select the next button
- **LV\_KEY\_LEFT/TOP** Select the previous button

- **LV\_KEY\_ENTER** Clicks the selected button

Learn more about *Keys*.

### Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        printf("Button: %s\n", lv_mbox_get_active_btn_text(obj));
    }
}

void lv_ex_mbox_1(void)
{
    static const char * btns[] ={"Apply", "Close", ""};

    lv_obj_t * mbox1 = lv_mbox_create(lv_scr_act(), NULL);
    lv_mbox_set_text(mbox1, "A message box with two buttons.");
    lv_mbox_add_btns(mbox1, btns);
    lv_obj_set_width(mbox1, 200);
    lv_obj_set_event_cb(mbox1, event_handler);
    lv_obj_align(mbox1, NULL, LV_ALIGN_CENTER, 0, 0); /*Align to the corner*/
}
```



code

```
/**
 * @file lv_ex_mbox_2.c
 *
 */

/*****
 * INCLUDES
 *****/

#include "lvgl/lvgl.h"

/*****
 * STATIC PROTOTYPES
 *****/

static void mbox_event_cb(lv_obj_t *obj, lv_event_t evt);
static void btn_event_cb(lv_obj_t *btn, lv_event_t evt);

/*****
 * STATIC VARIABLES
 *****/

static lv_obj_t *mbox, *info;

static const char welcome_info[] = "Welcome to the modal message box demo!\n"
    "Press the button to display a message box.";

static const char in_msg_info[] = "Notice that you cannot touch "
    "the button again while the message box is open.";

/*****
 * GLOBAL FUNCTIONS
 *****/
```

(continues on next page)



(continued from previous page)

```

*****/

void lv_ex_mbox_2(void)
{
    /* Create a button, then set its position and event callback */
    lv_obj_t *btn = lv_btn_create(lv_scr_act(), NULL);
    lv_obj_set_size(btn, 200, 60);
    lv_obj_set_event_cb(btn, btn_event_cb);
    lv_obj_align(btn, NULL, LV_ALIGN_IN_TOP_LEFT, 20, 20);

    /* Create a label on the button */
    lv_obj_t *label = lv_label_create(btn, NULL);
    lv_label_set_text(label, "Display a message box!");

    /* Create an informative label on the screen */
    info = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(info, welcome_info);
    lv_label_set_long_mode(info, LV_LABEL_LONG_BREAK); /* Make sure text will
↳wrap */
    lv_obj_set_width(info, LV_HOR_RES - 10);
    lv_obj_align(info, NULL, LV_ALIGN_IN_BOTTOM_LEFT, 5, -5);
}

/*****
 *   STATIC FUNCTIONS
 *****/

static void mbox_event_cb(lv_obj_t *obj, lv_event_t evt)
{
    if(evt == LV_EVENT_DELETE && obj == mbox) {
        /* Delete the parent modal background */
        lv_obj_del_async(lv_obj_get_parent(mbox));
        mbox = NULL; /* happens before object is actually deleted! */
        lv_label_set_text(info, welcome_info);
    } else if(evt == LV_EVENT_VALUE_CHANGED) {
        /* A button was clicked */
        lv_mbox_start_auto_close(mbox, 0);
    }
}

static void btn_event_cb(lv_obj_t *btn, lv_event_t evt)
{
    if(evt == LV_EVENT_CLICKED) {
        static lv_style_t modal_style;
        /* Create a full-screen background */
        lv_style_copy(&modal_style, &lv_style_plain_color);

        /* Set the background's style */
        modal_style.body.main_color = modal_style.body.grad_color = LV_COLOR_
↳BLACK;

        modal_style.body.opa = LV_OPA_50;

        /* Create a base object for the modal background */
        lv_obj_t *obj = lv_obj_create(lv_scr_act(), NULL);
        lv_obj_set_style(obj, &modal_style);
    }
}

```

(continues on next page)

(continued from previous page)

```

        lv_obj_set_pos(obj, 0, 0);
        lv_obj_set_size(obj, LV_HOR_RES, LV_VER_RES);
        lv_obj_set_opa_scale_enable(obj, true); /* Enable opacity scaling for
↪the animation */

        static const char * btns2[] = {"Ok", "Cancel", ""};

        /* Create the message box as a child of the modal background */
        mbox = lv_mbox_create(obj, NULL);
        lv_mbox_add_btns(mbox, btns2);
        lv_mbox_set_text(mbox, "Hello world!");
        lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0);
        lv_obj_set_event_cb(mbox, mbox_event_cb);

        /* Fade the message box in with an animation */
        lv_anim_t a;
        lv_anim_init(&a);
        lv_anim_set_time(&a, 500, 0);
        lv_anim_set_values(&a, LV_OPA_TRANSP, LV_OPA_COVER);
        lv_anim_set_exec_cb(&a, obj, (lv_anim_exec_xcb_t)lv_obj_set_opa_
↪scale);

        lv_anim_create(&a);

        lv_label_set_text(info, in_msg_info);
        lv_obj_align(info, NULL, LV_ALIGN_IN_BOTTOM_LEFT, 5, -5);
    }
}

```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_mbox\_style\_t**

### Enums

**enum** [anonymous]  
Message box styles.

*Values:*

**LV\_MBOX\_STYLE\_BG**

**LV\_MBOX\_STYLE\_BTN\_BG**

Same meaning as ordinary button styles.

**LV\_MBOX\_STYLE\_BTN\_REL**

**LV\_MBOX\_STYLE\_BTN\_PR**

**LV\_MBOX\_STYLE\_BTN\_TGL\_REL**

**LV\_MBOX\_STYLE\_BTN\_TGL\_PR**

**LV\_MBOX\_STYLE\_BTN\_INA**

## Functions

*lv\_obj\_t* \***lv\_mbox\_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\_mbox\_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\_mbox\_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\_mbox\_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\_mbox\_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\_mbox\_stop\_auto\_close**(*lv\_obj\_t* \**mbox*)

Stop the auto. closing of message box

### Parameters

- **mbox**: pointer to a message box object

void **lv\_mbox\_set\_style**(*lv\_obj\_t* \**mbox*, *lv\_mbox\_style\_t* *type*, **const** *lv\_style\_t* \**style*)

Set a style of a message box

### Parameters

- **mbx**: pointer to a message box object
- **type**: which style should be set
- **style**: pointer to a style

void **lv\_mbox\_set\_recolor**(*lv\_obj\_t \*mbx*, bool *en*)

Set whether recoloring is enabled. Must be called after **lv\_mbox\_add\_btns**.

**Parameters**

- **btnm**: pointer to button matrix object
- **en**: whether recoloring is enabled

const char \***lv\_mbox\_get\_text**(const *lv\_obj\_t \*mbx*)

Get the text of the message box

**Return** pointer to the text of the message box

**Parameters**

- **mbx**: pointer to a message box object

uint16\_t **lv\_mbox\_get\_active\_btn**(*lv\_obj\_t \*mbx*)

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\_BTNM\_BTN\_NONE: if unset)

**Parameters**

- **btnm**: pointer to button matrix object

const char \***lv\_mbox\_get\_active\_btn\_text**(*lv\_obj\_t \*mbx*)

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\_mbox\_get\_anim\_time**(const *lv\_obj\_t \*mbx*)

Get the animation duration (close animation time)

**Return** animation length in milliseconds (0: no animation)

**Parameters**

- **mbx**: pointer to a message box object

const lv\_style\_t \***lv\_mbox\_get\_style**(const *lv\_obj\_t \*mbx*, *lv\_mbox\_style\_t type*)

Get a style of a message box

**Return** style pointer to a style

**Parameters**

- **mbx**: pointer to a message box object
- **type**: which style should be get

bool **lv\_mbox\_get\_recolor**(const *lv\_obj\_t \*mbx*)

Get whether recoloring is enabled

**Return** whether recoloring is enabled

### Parameters

- **mbox**: pointer to a message box object

*lv\_obj\_t* \***lv\_mbox\_get\_btnm**(*lv\_obj\_t* \**mbox*)

Get message box button matrix

**Return** pointer to button matrix object

**Remark** return value will be NULL unless **lv\_mbox\_add\_btns** has been already called

### Parameters

- **mbox**: pointer to a message box object

**struct lv\_mbox\_ext\_t**

### Public Members

*lv\_cont\_ext\_t* **bg**

*lv\_obj\_t* \***text**

*lv\_obj\_t* \***btnm**

uint16\_t **anim\_time**

## Page (lv\_page)

### Overview

The Page consist of two *Containers* on each other:

- a **background** (or base)
- a top which is **scrollable**.

The background object can be referenced as the page itself like: **lv\_obj\_set\_width**(page, 100).

If you create a child 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\_FILL** auto fit in all directions. It means the scrollable size will be the same as the background's size (minus the paddings) 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\_SB\_MODE\_OFF** Never show scrollbars
- **LV\_SB\_MODE\_ON** Always show scrollbars
- **LV\_SB\_MODE\_DRAG** Show scrollbars when the page is being dragged
- **LV\_SB\_MODE\_AUTO** Show scrollbars when the scrollable container is large enough to be scrolled

You can set scroll bar show policy by: **lv\_page\_set\_sb\_mode**(page, **SB\_MODE**). The default value is **LV\_SB\_MODE\_AUTO**.

## Glue object

You can glue children to the page. In this case, you can scroll the page by dragging the child object. It can be enabled by the `lv_page_glue_obj(child, true)`.

## Focus object

You can focus on an object on a page with `lv_page_focus(page, child, LV_ANIM_ON/OFF)`. 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.

## 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)`

## Scrollable API

There are functions to directly set/get the scrollable's attributes:

- `lv_page_get_scl()`
- `lv_page_set_scl_fit/fint2/fit4()`
- `lv_page_set_scl_width()`
- `lv_page_set_scl_height()`
- `lv_page_set_scl_layout()`

## Notes

The background draws its border when the scrollable is drawn. It ensures that the page always will have a closed shape even if the scrollable has the same color as the Page's parent.

## Styles

Use `lv_page_set_style(page, LV_PAGE_STYLE_..., &style)` to set a new style for an element of the page:

- **LV\_PAGE\_STYLE\_BG** background's style which uses all `style.body` properties (default: `lv_style_pretty_color`)
- **LV\_PAGE\_STYLE\_SCRL** scrollable's style which uses all `style.body` properties (default: `lv_style_pretty`)
- **LV\_PAGE\_STYLE\_SB** scrollbar's style which uses all `style.body` properties. `padding.right/bottom` sets horizontal and vertical the scrollbars' padding respectively and the `padding.inner` sets the scrollbar's width. (default: `lv_style_pretty_color`)

## 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*.

## Example

## C



code

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

void lv_ex_page_1(void)
{
    /*Create a scroll bar style*/
    static lv_style_t style_sb;
    lv_style_copy(&style_sb, &lv_style_plain);
    style_sb.body.main_color = LV_COLOR_BLACK;
    style_sb.body.grad_color = LV_COLOR_BLACK;
    style_sb.body.border.color = LV_COLOR_WHITE;
    style_sb.body.border.width = 1;
    style_sb.body.border.opa = LV_OPA_70;
    style_sb.body.radius = LV_RADIUS_CIRCLE;
    style_sb.body.opa = LV_OPA_60;
    style_sb.body.padding.right = 3;
    style_sb.body.padding.bottom = 3;
    style_sb.body.padding.inner = 8;           /*Scrollbar width*/

    /*Create a page*/
    lv_obj_t * page = lv_page_create(lv_scr_act(), NULL);
    lv_obj_set_size(page, 150, 200);
    lv_obj_align(page, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_page_set_style(page, LV_PAGE_STYLE_SB, &style_sb);           /*Set the
↪ scrollbar style*/

    /*Create a label on the page*/
    lv_obj_t * label = lv_label_create(page, NULL);
    lv_label_set_long_mode(label, LV_LABEL_LONG_BREAK);           /*Automatically
↪ break long lines*/
    lv_obj_set_width(label, lv_page_get_fit_width(page));           /*Set the label
↪ width to max value to not show hor. scroll bars*/
}
```

(continues on next page)



(continued from previous page)

```

    lv_label_set_text(label, "Lorem ipsum dolor sit amet, consectetur adipiscing elit,
↪\n"
                                "sed do eiusmod tempor incididunt ut labore et dolore_
↪magna aliqua.\n"
                                "Ut enim ad minim veniam, quis nostrud exercitation_
↪ullamco\n"
                                "laboris nisi ut aliquip ex ea commodo consequat. Duis_
↪aute irure\n"
                                "dolor in reprehenderit in voluptate velit esse cillum_
↪dolore\n"
                                "eu fugiat nulla pariat.\n"
                                "Excepteur sint occaecat cupidatat non proident, sunt in_
↪culpa\n"
                                "qui officia deserunt mollit anim id est laborum.");
}

```

## MicroPython

No examples yet.

## API

### Typedefs

```

typedef uint8_t lv_sb_mode_t
typedef uint8_t lv_page_edge_t
typedef uint8_t lv_page_style_t

```

### Enums

```

enum [anonymous]
    Scrollbar modes: shows when should the scrollbars be visible

    Values:

    LV_SB_MODE_OFF = 0x0
        Never show scrollbars

    LV_SB_MODE_ON = 0x1
        Always show scrollbars

    LV_SB_MODE_DRAG = 0x2
        Show scrollbars when page is being dragged

    LV_SB_MODE_AUTO = 0x3
        Show scrollbars when the scrollable container is large enough to be scrolled

    LV_SB_MODE_HIDE = 0x4
        Hide the scroll bar temporally

    LV_SB_MODE_UNHIDE = 0x5
        Unhide the previously hidden scrollbar. Recover it's type 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\_STYLE\_BG**

**LV\_PAGE\_STYLE\_SCROLL**

**LV\_PAGE\_STYLE\_SB**

**LV\_PAGE\_STYLE\_EDGE\_FLASH**

## 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* \*obj)

Delete all children of the scroll object, without deleting scroll child.

**Parameters**

- **obj**: pointer to an object

*lv\_obj\_t* \***lv\_page\_get\_scroll**(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\_sb\_mode**(*lv\_obj\_t* \*page, *lv\_sb\_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.

#### 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\_scrl\_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\_scrl\_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
- **hor**: horizontal fit policy from **lv\_fit\_t**
- **ver**: vertical fit policy from **lv\_fit\_t**

**static** void **lv\_page\_set\_scrl\_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 has no effect if 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 has no effect if 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'

**void lv\_page\_set\_style**(*lv\_obj\_t \*page*, *lv\_page\_style\_t type*, **const** *lv\_style\_t \*style*)  
Set a style of a page

**Parameters**

- **page**: pointer to a page object
- **type**: which style should be set
- **style**: pointer to a style

*lv\_sb\_mode\_t* **lv\_page\_get\_sb\_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\_fit\_width**(*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\_fit\_height**(*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

**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

**const** lv\_style\_t \***lv\_page\_get\_style**(**const** lv\_obj\_t \*page, lv\_page\_style\_t type)

Get a style of a page

**Return** style pointer to a style

#### Parameters

- **page**: pointer to page object
- **type**: which style should be get

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)

Not intended to use directly by the user but by other object types internally. Start an edge flash animation. Exactly one **ext->edge\_flash.xxx\_ip** should be set

### Parameters

- page:

**struct lv\_page\_ext\_t**

### Public Members

```

lv_cont_ext_t bg
lv_obj_t *scr1
const lv_style_t *style
lv_area_t hor_area
lv_area_t ver_area
uint8_t hor_draw
uint8_t ver_draw
lv_sb_mode_t mode
struct lv_page_ext_t::[anonymous] sb
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
uint8_t scroll_prop
uint8_t scroll_prop_ip

```

### Preloader (lv\_preload)

#### Overview

The preloader object is a spinning arc over a border.

#### Arc length

The length of the arc can be adjusted by `lv_preload_set_arc_length(preload, deg)`.

#### Spinning speed

The speed of the spinning can be adjusted by `lv_preload_set_spin_time(preload, time_ms)`.

## Spin types

You can choose from more spin types:

- **LV\_PRELOAD\_TYPE\_SPINNING\_ARC** spin the arc, slow down on the top
- **LV\_PRELOAD\_TYPE\_FILLSPIN\_ARC** spin the arc, slow down on the top but also stretch the arc

To apply one if them use `lv_preload_set_type(preload, LV_PRELOAD_TYPE_...)`

## Spin direction

The direction of spinning can be changed with `lv_preload_set_dir(preload, LV_PRELOAD_DIR_FORWARD/BACKWARD)`.

## Styles

You can set the styles with `lv_preload_set_style(btn, LV_PRELOAD_STYLE_MAIN, &style)`. It describes both the arc and the border style:

- **arc** is described by the **line** properties
- **border** is described by the **body.border** properties including **body.padding.left/top** (the smaller is used) to give a smaller radius for the border.

## Events

Only the [Generic events](#) are sent by the object type.

## Keys

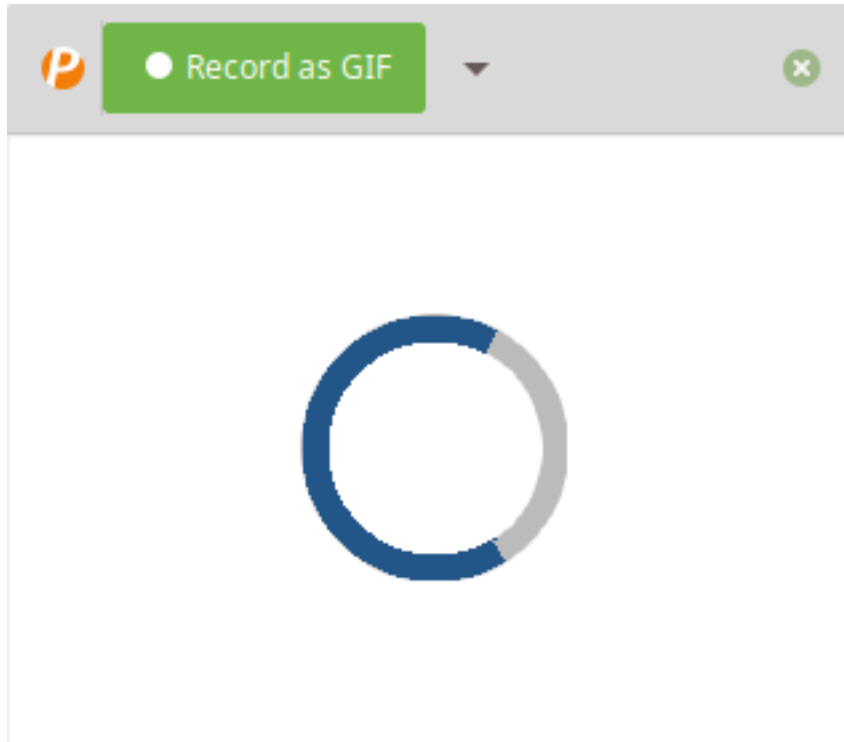
No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example



## C



code

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

void lv_ex_preload_1(void)
{
    /*Create a style for the Preloader*/
    static lv_style_t style;
    lv_style_copy(&style, &lv_style_plain);
    style.line.width = 10; /*10 px thick arc*/
    style.line.color = lv_color_hex3(0x258); /*Blueish arc color*/

    style.body.border.color = lv_color_hex3(0xBBB); /*Gray background color*/
    style.body.border.width = 10;
    style.body.padding.left = 0;

    /*Create a Preloader object*/
    lv_obj_t * preload = lv_preload_create(lv_scr_act(), NULL);
    lv_obj_set_size(preload, 100, 100);
    lv_obj_align(preload, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_preload_set_style(preload, LV_PRELOAD_STYLE_MAIN, &style);
}
```

## MicroPython

No examples yet.

## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_preload_type_t
typedef uint8_t lv_preload_dir_t
typedef uint8_t lv_preload_style_t
```

### Enums

```
enum [anonymous]
    Type of preloader.

    Values:

    LV_PRELOAD_TYPE_SPINNING_ARC
    LV_PRELOAD_TYPE_FILLSPIN_ARC
```

```
enum [anonymous]
    Direction the preloader should spin.

    Values:

    LV_PRELOAD_DIR_FORWARD
    LV_PRELOAD_DIR_BACKWARD
```

```
enum [anonymous]
    Values:

    LV_PRELOAD_STYLE_MAIN
```

### Functions

```
lv_obj_t *lv_preload_create(lv_obj_t *par, const lv_obj_t *copy)
```

Create a pre loader objects

**Return** pointer to the created pre loader

#### Parameters

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

```
void lv_preload_set_arc_length(lv_obj_t *preload, lv_anim_value_t deg)
```

Set the length of the spinning arc in degrees

#### Parameters

- **preload**: pointer to a preload object
- **deg**: length of the arc

void **lv\_preload\_set\_spin\_time**(*lv\_obj\_t \*preload*, *uint16\_t time*)  
 Set the spin time of the arc

**Parameters**

- **preload**: pointer to a preload object
- **time**: time of one round in milliseconds

void **lv\_preload\_set\_style**(*lv\_obj\_t \*preload*, *lv\_preload\_style\_t type*, **const** *lv\_style\_t \*style*)

Set a style of a pre loader.

**Parameters**

- **preload**: pointer to pre loader object
- **type**: which style should be set
- **style**: pointer to a style

void **lv\_preload\_set\_type**(*lv\_obj\_t \*preload*, *lv\_preload\_type\_t type*)  
 Set the animation type of a preloader.

**Parameters**

- **preload**: pointer to pre loader object
- **type**: animation type of the preload

void **lv\_preload\_set\_dir**(*lv\_obj\_t \*preload*, *lv\_preload\_dir\_t dir*)  
 Set the animation direction of a preloader

**Parameters**

- **preload**: pointer to pre loader object
- **direction**: animation direction of the preload

*lv\_anim\_value\_t* **lv\_preload\_get\_arc\_length**(**const** *lv\_obj\_t \*preload*)  
 Get the arc length [degree] of the a pre loader

**Parameters**

- **preload**: pointer to a pre loader object

*uint16\_t* **lv\_preload\_get\_spin\_time**(**const** *lv\_obj\_t \*preload*)  
 Get the spin time of the arc

**Parameters**

- **preload**: pointer to a pre loader object [milliseconds]

**const** *lv\_style\_t \****lv\_preload\_get\_style**(**const** *lv\_obj\_t \*preload*, *lv\_preload\_style\_t type*)  
 Get style of a pre loader.

**Return** style pointer to the style

**Parameters**

- **preload**: pointer to pre loader object
- **type**: which style should be get

*lv\_preload\_type\_t* **lv\_preload\_get\_type**(*lv\_obj\_t \*preload*)  
 Get the animation type of a preloader.

**Return** animation type

### Parameters

- **preload**: pointer to pre loader object

*lv\_preload\_dir\_t* **lv\_preload\_get\_dir**(*lv\_obj\_t \*preload*)

Get the animation direction of a preloader

**Return** animation direction

### Parameters

- **preload**: pointer to pre loader object

void **lv\_preload\_spinner\_anim**(void \*ptr, *lv\_anim\_value\_t* val)

Animator function (exec\_cb) to rotate the arc of spinner.

### Parameters

- **ptr**: pointer to preloader
- **val**: the current desired value [0..360]

**struct lv\_preload\_ext\_t**

### Public Members

*lv\_arc\_ext\_t* **arc**

*lv\_anim\_value\_t* **arc\_length**

uint16\_t **time**

*lv\_preload\_type\_t* **anim\_type**

*lv\_preload\_dir\_t* **anim\_dir**

## Roller (lv\_roller)

### Overview

Roller allows you to simply select one option from more with scrolling. Its functionalities are similar to *Drop down list*.

### 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), where *id* is the index of an option.

### Get selected option

To 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)`.

### Height and width

You can set the number of visible rows with `lv_roller_set_visible_row_count(roller, num)`

The width is adjusted automatically according to the width of the options. To prevent this apply `lv_roller_set_fix_width(roller, width)`. 0 means to use auto width.

### 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.

### Styles

The `lv_roller_set_style(roller, LV_ROLLER_STYLE_..., &style)` set the styles of a Roller.

- **LV\_ROLLER\_STYLE\_BG** Style of the background. All `style.body` properties are used. `style.text` is used for the option's label. Default: `lv_style_pretty`
- **LV\_ROLLER\_STYLE\_SEL** Style of the selected option. The `style.body` properties are used. The selected option will be recolored with `text.color`. Default: `lv_style_plain_color`

### 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*.

### 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)

### Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        char buf[32];
        lv_roller_get_selected_str(obj, buf, sizeof(buf));
        printf("Selected month: %s\n", buf);
    }
}

void lv_ex_roller_1(void)
{
    lv_obj_t *roller1 = lv_roller_create(lv_scr_act(), NULL);
    lv_roller_set_options(roller1,
        "January\n"
        "February\n"
        "March\n"
        "April\n"
        "May\n"
        "June\n"
        "July\n"
        "August\n"
```

(continues on next page)

(continued from previous page)

```

        "September\n"
        "October\n"
        "November\n"
        "December",
        LV_ROLLER_MODE_INIFINITE);

lv_roller_set_visible_row_count(roller1, 4);
lv_obj_align(roller1, NULL, LV_ALIGN_CENTER, 0, 0);
lv_obj_set_event_cb(roller1, event_handler);
}

```

## MicroPython

No examples yet.

## API

### Typedefs

```

typedef uint8_t lv_roller_mode_t
typedef uint8_t lv_roller_style_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\_STYLE\_BG**

**LV\_ROLLER\_STYLE\_SEL**

### 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

**static** void **lv\_roller\_set\_fix\_width**(*lv\_obj\_t \*roller*, *lv\_coord\_t w*)  
 Set a fix width for the drop down list

#### Parameters

- **roller**: pointer to a roller object
- **w**: the width when the list is opened (0: auto size)

**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]

void **lv\_roller\_set\_style**(*lv\_obj\_t \*roller*, *lv\_roller\_style\_t type*, **const** *lv\_style\_t \*style*)  
 Set a style of a roller

#### Parameters

- **roller**: pointer to a roller object
- **type**: which style should be set
- **style**: pointer to a style



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

**static void lv\_roller\_get\_selected\_str**(const lv\_obj\_t \*roller, char \*buf, uint16\_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

**static const char \*lv\_roller\_get\_options**(const lv\_obj\_t \*roller)

Get the options of a roller

**Return** the options separated by ‘\n’-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

bool **lv\_roller\_get\_hor\_fit**(const lv\_obj\_t \*roller)

Get the auto width set attribute

**Return** true: auto size enabled; false: manual width settings enabled

**Parameters**

- **roller**: pointer to a roller object

**const lv\_style\_t \*lv\_roller\_get\_style**(const lv\_obj\_t \*roller, lv\_roller\_style\_t type)

Get a style of a roller

**Return** style pointer to a style

**Parameters**

- **roller**: pointer to a roller object
- **type**: which style should be get

## struct lv\_roller\_ext\_t

### Public Members

*lv\_ddlist\_ext\_t* **ddlist**

*lv\_roller\_mode\_t* **mode**

## Slider (lv\_slider)

### 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.

### 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.

### Knob placement

The knob can be placed in two ways:

- inside the background
- on the edges on min/max values

Use the `lv_slider_set_knob_in(slider, true/false)` to choose between the modes. (*knob\_in = false* is the default)

### Styles

You can modify the slider's styles with `lv_slider_set_style(slider, LV_SLIDER_STYLE_..., &style)`.

- **LV\_SLIDER\_STYLE\_BG** Style of the background. All `style.body` properties are used. The `padding` values make the knob larger than the background. (negative value makes is larger)
- **LV\_SLIDER\_STYLE\_INDIC** Style of the indicator. All `style.body` properties are used. The `padding` values make the indicator smaller than the background.
- **LV\_SLIDER\_STYLE\_KNOB** Style of the knob. All `style.body` properties are used except `padding`.

### 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.

## 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*.

## Example

### C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        printf("Value: %d\n", lv_slider_get_value(obj));
    }
}

void lv_ex_slider_1(void)
{
    /*Create styles*/
    static lv_style_t style_bg;
    static lv_style_t style_indic;
    static lv_style_t style_knob;

    lv_style_copy(&style_bg, &lv_style_pretty);
```

(continues on next page)

(continued from previous page)

```

style_bg.body.main_color = LV_COLOR_BLACK;
style_bg.body.grad_color = LV_COLOR_GRAY;
style_bg.body.radius = LV_RADIUS_CIRCLE;
style_bg.body.border.color = LV_COLOR_WHITE;

lv_style_copy(&style_indic, &lv_style_pretty_color);
style_indic.body.radius = LV_RADIUS_CIRCLE;
style_indic.body.shadow.width = 8;
style_indic.body.shadow.color = style_indic.body.main_color;
style_indic.body.padding.left = 3;
style_indic.body.padding.right = 3;
style_indic.body.padding.top = 3;
style_indic.body.padding.bottom = 3;

lv_style_copy(&style_knob, &lv_style_pretty);
style_knob.body.radius = LV_RADIUS_CIRCLE;
style_knob.body.opa = LV_OPA_70;
style_knob.body.padding.top = 10 ;
style_knob.body.padding.bottom = 10 ;

/*Create a slider*/
lv_obj_t * slider = lv_slider_create(lv_scr_act(), NULL);
lv_slider_set_style(slider, LV_SLIDER_STYLE_BG, &style_bg);
lv_slider_set_style(slider, LV_SLIDER_STYLE_INDIC, &style_indic);
lv_slider_set_style(slider, LV_SLIDER_STYLE_KNOB, &style_knob);
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0);
lv_obj_set_event_cb(slider, event_handler);
}

```

Welcome to the slider+label demo!  
Move the slider and see that the label  
updates to match it.



code

```

/**
 * @file lv_ex_slider_2.c
 *

```

(continues on next page)

(continued from previous page)

```

*/

/*****
 *      INCLUDES
 *****/

#include "lvgl/lvgl.h"
#include <stdio.h>

/*****
 *      DEFINES
 *****/

/*****
 *      TYPEDEFS
 *****/

/*****
 *      STATIC PROTOTYPES
 *****/

static void slider_event_cb(lv_obj_t * slider, lv_event_t event);

/*****
 *      STATIC VARIABLES
 *****/

static lv_obj_t * slider_label;

/*****
 *      MACROS
 *****/

/*****
 *      GLOBAL FUNCTIONS
 *****/

void lv_ex_slider_2(void)
{
    /* Create a slider in the center of the display */
    lv_obj_t * slider = lv_slider_create(lv_scr_act(), NULL);
    lv_obj_set_width(slider, LV_DPI * 2);
    lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_obj_set_event_cb(slider, slider_event_cb);
    lv_slider_set_range(slider, 0, 100);

    /* Create a label below the slider */
    slider_label = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(slider_label, "0");
    lv_label_set_align(slider_label, LV_LABEL_ALIGN_CENTER);
    lv_obj_align(slider_label, slider, LV_ALIGN_OUT_BOTTOM_MID, 0, 10);

    /* Create an informative label */
    lv_obj_t * info = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(info, "Welcome to the slider+label demo!\n"
                           "Move the slider and see that the label\n"

```

(continues on next page)

(continued from previous page)

```

        "updates to match it.");
    lv_obj_align(info, NULL, LV_ALIGN_IN_TOP_LEFT, 10, 10);
}

/*****
 *   STATIC FUNCTIONS
 *****/

static void slider_event_cb(lv_obj_t * slider, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        static char buf[4]; /* max 3 bytes for number plus 1 null terminating byte */
        snprintf(buf, 4, "%u", lv_slider_get_value(slider));
        lv_label_set_text(slider_label, buf);
        lv_obj_align(slider_label, slider, LV_ALIGN_OUT_BOTTOM_MID, 0, 10);
    }
}

```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_slider\_style\_t**

### Enums

**enum** [anonymous]

Built-in styles of slider

*Values:*

**LV\_SLIDER\_STYLE\_BG**

**LV\_SLIDER\_STYLE\_INDIC**

Slider background style.

**LV\_SLIDER\_STYLE\_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\_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.

**void lv\_slider\_set\_knob\_in**(*lv\_obj\_t \*slider*, *bool in*)  
Set the 'knob in' attribute of a slider

**Parameters**

- **slider**: pointer to slider object
- **in**: true: the knob is drawn always in the slider; false: the knob can be out on the edges

**void lv\_slider\_set\_style**(*lv\_obj\_t \*slider*, *lv\_slider\_style\_t type*, **const** *lv\_style\_t \*style*)  
Set a style of a slider

**Parameters**

- **slider**: pointer to a slider object
- **type**: which style should be set
- **style**: pointer to a style

**int16\_t lv\_slider\_get\_value**(**const** *lv\_obj\_t \*slider*)  
Get the value of a slider

**Return** the value 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

bool **lv\_slider\_get\_knob\_in**(const lv\_obj\_t \*slider)

Get the 'knob in' attribute of a slider

**Return** true: the knob is drawn always in the slider; false: the knob can be out on the edges

**Parameters**

- `slider`: pointer to slider object

const lv\_style\_t \***lv\_slider\_get\_style**(const lv\_obj\_t \*slider, lv\_slider\_style\_t type)

Get a style of a slider

**Return** style pointer to a style

**Parameters**

- `slider`: pointer to a slider object
- `type`: which style should be get

**struct lv\_slider\_ext\_t**

**Public Members**

lv\_bar\_ext\_t **bar**

const lv\_style\_t \***style\_knob**

int16\_t **drag\_value**

uint8\_t **knob\_in**

**Spinbox (lv\_spinbox)**

**Overview**

The Spinbox contains a number as text which can be increased or decreased by *Keys* or API functions. The Spinbox is a modified *Text area*.



## 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 and 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.

## Style usage

The `lv_spinbox_set_style(roller, LV_SPINBOX_STYLE_..., &style)` set the styles of a Spinbox.

- **LV\_SPINBOX\_STYLE\_BG** Style of the background. All `style.body` properties are used. `style.text` is used for label. Default: `lv_style_pretty`
- **LV\_SPINBOX\_STYLE\_SB** Scrollbar’s style which uses all `style.body` properties. `padding.right/bottom` sets horizontal and vertical the scrollbars’ padding respectively and the `padding.inner` sets the scrollbar’s width. (default: `lv_style_pretty_color`)
- **LV\_SPINBOX\_STYLE\_CURSOR** Style of the cursor which uses all `style.body` properties including `padding` to make the cursor larger than the digits.

## 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*.

## 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.
- **LV\_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.

## Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        printf("Value: %d\n", lv_spinbox_get_value(obj));
    }
    else if(event == LV_EVENT_CLICKED) {
        /*For simple test: Click the spinbox to increment its value*/
        lv_spinbox_increment(obj);
    }
}

void lv_ex_spinbox_1(void)
{
    lv_obj_t * spinbox;
    spinbox = lv_spinbox_create(lv_scr_act(), NULL);
    lv_spinbox_set_digit_format(spinbox, 5, 3);
    lv_spinbox_step_prev(spinbox);
    lv_obj_set_width(spinbox, 100);
    lv_obj_align(spinbox, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_obj_set_event_cb(spinbox, event_handler);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_spinbox_style_t
```

### Enums

```
enum [anonymous]
```

*Values:*

```
LV_SPINBOX_STYLE_BG
```

```
LV_SPINBOX_STYLE_SB
```

```
LV_SPINBOX_STYLE_CURSOR
```

### 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

```
static void lv_spinbox_set_style(lv_obj_t *spinbox, lv_spinbox_style_t type, lv_style_t *style)
```

Set a style of a spinbox.

**Parameters**

- **templ**: pointer to template object
- **type**: which style should be set
- **style**: pointer to a style

```
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

**static const** lv\_style\_t \***lv\_spinbox\_get\_style**(*lv\_obj\_t \*spinbox*, *lv\_spinbox\_style\_t type*)

Get style of a spinbox.

**Return** style pointer to the style

**Parameters**

- **templ**: pointer to template object
- **type**: which style should be get

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\_ta\_ext\_t* **ta**  
 int32\_t **value**  
 int32\_t **range\_max**  
 int32\_t **range\_min**  
 int32\_t **step**  
 uint16\_t **digit\_count**  
 uint16\_t **dec\_point\_pos**  
 uint16\_t **digit\_padding\_left**

### Example

#### Switch (lv\_sw)

#### Overview

The Switch can be used to turn on/off something. The look like a little slider.

#### Change state

The state of the switch can be changed by

- Clicking on it
- Sliding it
- Using `lv_sw_on(sw, LV_ANIM_ON/OFF)`, `lv_sw_off(sw, LV_ANIM_ON/OFF)` or `lv_sw_toggle(sw, LV_ANOM_ON/OFF)` functions

#### Animation time

The time of animations, when the switch changes state, can be adjusted with `lv_sw_set_anim_time(sw, anim_time)`.

## Styles

You can modify the Switch's styles with `lv_sw_set_style(sw, LV_SW_STYLE_..., &style)`.

- **LV\_SW\_STYLE\_BG** Style of the background. All `style.body` properties are used. The `padding` values make the Switch smaller than the knob. (negative value makes is larger)
- **LV\_SW\_STYLE\_INDIC** Style of the indicator. All `style.body` properties are used. The `padding` values make the indicator smaller than the background.
- **LV\_SW\_STYLE\_KNOB\_OFF** Style of the knob when the switch is off. The `style.body` properties are used except padding.
- **LV\_SW\_STYLE\_KNOB\_ON** Style of the knob when the switch is on. The `style.body` properties are used except padding.

## Events

Besides the [Generic events](#) the following [Special events](#) are sent by the Switch:

- **LV\_EVENT\_VALUE\_CHANGED** Sent when the switch changes state.

## 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*.

## Example

## C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        printf("State: %s\n", lv_sw_get_state(obj) ? "On" : "Off");
    }
}

void lv_ex_sw_1(void)
{
    /*Create styles for the switch*/
    static lv_style_t bg_style;
    static lv_style_t indic_style;
    static lv_style_t knob_on_style;
    static lv_style_t knob_off_style;

    lv_style_copy(&bg_style, &lv_style_pretty);
    bg_style.body.radius = LV_RADIUS_CIRCLE;
    bg_style.body.padding.top = 6;
    bg_style.body.padding.bottom = 6;

    lv_style_copy(&indic_style, &lv_style_pretty_color);
    indic_style.body.radius = LV_RADIUS_CIRCLE;
    indic_style.body.main_color = lv_color_hex(0x9fc8ef);
    indic_style.body.grad_color = lv_color_hex(0x9fc8ef);
    indic_style.body.padding.left = 0;
    indic_style.body.padding.right = 0;
    indic_style.body.padding.top = 0;
    indic_style.body.padding.bottom = 0;
}
```

(continues on next page)

(continued from previous page)

```

lv_style_copy(&knob_off_style, &lv_style_pretty);
knob_off_style.body.radius = LV_RADIUS_CIRCLE;
knob_off_style.body.shadow.width = 4;
knob_off_style.body.shadow.type = LV_SHADOW_BOTTOM;

lv_style_copy(&knob_on_style, &lv_style_pretty_color);
knob_on_style.body.radius = LV_RADIUS_CIRCLE;
knob_on_style.body.shadow.width = 4;
knob_on_style.body.shadow.type = LV_SHADOW_BOTTOM;

/*Create a switch and apply the styles*/
lv_obj_t *sw1 = lv_sw_create(lv_scr_act(), NULL);
lv_sw_set_style(sw1, LV_SW_STYLE_BG, &bg_style);
lv_sw_set_style(sw1, LV_SW_STYLE_INDIC, &indic_style);
lv_sw_set_style(sw1, LV_SW_STYLE_KNOB_ON, &knob_on_style);
lv_sw_set_style(sw1, LV_SW_STYLE_KNOB_OFF, &knob_off_style);
lv_obj_align(sw1, NULL, LV_ALIGN_CENTER, 0, -50);
lv_obj_set_event_cb(sw1, event_handler);

/*Copy the first switch and turn it ON*/
lv_obj_t *sw2 = lv_sw_create(lv_scr_act(), sw1);
lv_sw_on(sw2, LV_ANIM_ON);
lv_obj_align(sw2, NULL, LV_ALIGN_CENTER, 0, 50);
}

```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_sw\_style\_t**

### Enums

**enum** [anonymous]

Switch styles.

*Values:*

**LV\_SW\_STYLE\_BG**

Switch background.

**LV\_SW\_STYLE\_INDIC**

Switch fill area.

**LV\_SW\_STYLE\_KNOB\_OFF**

Switch knob (when off).

**LV\_SW\_STYLE\_KNOB\_ON**

Switch knob (when on).



## Functions

*lv\_obj\_t* \***lv\_sw\_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\_sw\_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\_sw\_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\_sw\_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

void **lv\_sw\_set\_style**(*lv\_obj\_t* \**sw*, *lv\_sw\_style\_t* *type*, **const** *lv\_style\_t* \**style*)

Set a style of a switch

### Parameters

- **sw**: pointer to a switch object
- **type**: which style should be set
- **style**: pointer to a style

void **lv\_sw\_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\_sw\_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

const *lv\_style\_t* \***lv\_sw\_get\_style**(const *lv\_obj\_t* \*sw, *lv\_sw\_style\_t* type)

Get a style of a switch

**Return** style pointer to a style

**Parameters**

- **SW**: pointer to a switch object
- **type**: which style should be get

uint16\_t **lv\_sw\_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\_sw\_ext\_t**

**Public Members**

*lv\_slider\_ext\_t* **slider**

const *lv\_style\_t* \***style\_knob\_off**

Style of the knob when the switch is OFF

const *lv\_style\_t* \***style\_knob\_on**

Style of the knob when the switch is ON (NULL to use the same as OFF)

*lv\_coord\_t* **start\_x**

uint8\_t **changed**

uint8\_t **slided**

uint16\_t **anim\_time**

**Table (lv\_table)**

**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.

## 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 on texts so need to convert numbers 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

To make the Table scrollable place it on a *Page*

## Styles

Use `lv_table_set_style(page, LV_TABLE_STYLE_..., &style)` to set a new style for an element of the page:

- **LV\_PAGE\_STYLE\_BG** background's style which uses all `style.body` properties (default: `lv_style_plain_color`)
- **LV\_PAGE\_STYLE\_CELL1/2/3/4** 4 for styles for the 4 cell types. All `style.body` properties are used. (default: `lv_style_plain`)

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

No *Keys* are processed by the object type.

Learn more about *Keys*.

## Example

## C

Name	Price
Apple	\$7
Banana	\$4
Citron	\$6

code

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

void lv_ex_table_1(void)
{
    /*Create a normal cell style*/
    static lv_style_t style_cell1;
    lv_style_copy(&style_cell1, &lv_style_plain);
    style_cell1.body.border.width = 1;
    style_cell1.body.border.color = LV_COLOR_BLACK;

    /*Create a header cell style*/
    static lv_style_t style_cell2;
    lv_style_copy(&style_cell2, &lv_style_plain);
    style_cell2.body.border.width = 1;
    style_cell2.body.border.color = LV_COLOR_BLACK;
    style_cell2.body.main_color = LV_COLOR_SILVER;
    style_cell2.body.grad_color = LV_COLOR_SILVER;

    lv_obj_t * table = lv_table_create(lv_scr_act(), NULL);
    lv_table_set_style(table, LV_TABLE_STYLE_CELL1, &style_cell1);
    lv_table_set_style(table, LV_TABLE_STYLE_CELL2, &style_cell2);
    lv_table_set_style(table, LV_TABLE_STYLE_BG, &lv_style_transp_tight);
    lv_table_set_col_cnt(table, 2);
    lv_table_set_row_cnt(table, 4);
    lv_obj_align(table, NULL, LV_ALIGN_CENTER, 0, 0);

    /*Make the cells of the first row center aligned */
    lv_table_set_cell_align(table, 0, 0, LV_LABEL_ALIGN_CENTER);
    lv_table_set_cell_align(table, 0, 1, LV_LABEL_ALIGN_CENTER);

    /*Make the cells of the first row TYPE = 2 (use `style_cell2`) */
}
```

(continues on next page)

(continued from previous page)

```
lv_table_set_cell_type(table, 0, 0, 2);
lv_table_set_cell_type(table, 0, 1, 2);

/*Fill the first column*/
lv_table_set_cell_value(table, 0, 0, "Name");
lv_table_set_cell_value(table, 1, 0, "Apple");
lv_table_set_cell_value(table, 2, 0, "Banana");
lv_table_set_cell_value(table, 3, 0, "Citron");

/*Fill the second column*/
lv_table_set_cell_value(table, 0, 1, "Price");
lv_table_set_cell_value(table, 1, 1, "$7");
lv_table_set_cell_value(table, 2, 1, "$4");
lv_table_set_cell_value(table, 3, 1, "$6");
}
```

## MicroPython

No examples yet.

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_table\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_TABLE\_STYLE\_BG**  
**LV\_TABLE\_STYLE\_CELL1**  
**LV\_TABLE\_STYLE\_CELL2**  
**LV\_TABLE\_STYLE\_CELL3**  
**LV\_TABLE\_STYLE\_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

void **lv\_table\_set\_style**(*lv\_obj\_t \*table*, *lv\_table\_style\_t type*, const *lv\_style\_t \*style*)  
Set a style of a table.

#### Parameters

- **table**: pointer to table object
- **type**: which style should be set
- **style**: pointer to a style

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]

```
const lv_style_t *lv_table_get_style(const lv_obj_t *table, lv_table_style_t type)
```

Get style of a table.

**Return** style pointer to the style

**Parameters**

- **table**: pointer to table object
- **type**: which style should be get

```
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**

**const** lv\_style\_t \***cell\_style**[LV\_TABLE\_CELL\_STYLE\_CNT]

lv\_coord\_t **col\_w**[LV\_TABLE\_COL\_MAX]

### Tabview (lv\_tabview)

#### Overview

The Tab view object can be used to organize content in tabs.

#### Adding tab

You can add a new tabs with `lv_tabview_add_tab(tabview, "Tab name")`. It will return with a pointer to a *Page* object where you can add the tab's content.

## Change tab

To select a new tab you can:

- Click on it on the header part
- Slide horizontally
- Use `lv_tabview_set_tab_act(tabview, id, LV_ANIM_ON/OFF)` function

The manual sliding can be disabled with `lv_tabview_set_sliding(tabview, false)`.

## Tab button's position

By default, the tab selector buttons are placed on the top of the Tabview. It can be changed with `lv_tabview_set_btns_pos(tabview, LV_TABVIEW_BTNS_POS_TOP/BOTTOM/LEFT/RIGHT)`

Note that, you can't change the tab position from top or bottom to left or right when tabs are already added.

## Hide the tabs

The tab buttons can be hidden by `lv_tabview_set_btns_hidden(tabview, true)`

## 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.

## Style usage

Use `lv_tabview_set_style(tabview, LV_TABVIEW_STYLE_..., &style)` to set a new style for an element of the Tabview:

- **LV\_TABVIEW\_STYLE\_BG** main background which uses all `style.body` properties (default: `lv_style_plain`)
- **LV\_TABVIEW\_STYLE\_INDIC** a thin rectangle on indicating the current tab. Uses all `style.body` properties. Its height comes from `body.padding.inner` (default: `lv_style_plain_color`)
- **LV\_TABVIEW\_STYLE\_BTN\_BG** style of the tab buttons' background. Uses all `style.body` properties. The header height will be set automatically considering `body.padding.top/bottom` (default: `lv_style_transp`)
- **LV\_TABVIEW\_STYLE\_BTN\_REL** style of released tab buttons. Uses all `style.body` properties. (default: `lv_style_tbn_rel`)
- **LV\_TABVIEW\_STYLE\_BTN\_PR** style of released tab buttons. Uses all `style.body` properties except `padding`. (default: `lv_style_tbn_rel`)
- **LV\_TABVIEW\_STYLE\_BTN\_TGL\_REL** style of selected released tab buttons. Uses all `style.body` properties except `padding`. (default: `lv_style_tbn_rel`)
- **LV\_TABVIEW\_STYLE\_BTN\_TGL\_PR** style of selected pressed tab buttons. Uses all `style.body` properties except `padding`. (default: `lv_style_btn_tgl_pr`)

The height of the header is calculated like: *font height and padding.top and padding.bottom from LV\_TABVIEW\_STYLE\_BTN\_REL + padding.top and padding bottom from LV\_TABVIEW\_STYLE\_BTN\_BG*

## 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*.

## 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*.

## Example

C



code

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

void lv_ex_tabview_1(void)
```

(continues on next page)

(continued from previous page)

```
{
    /*Create a Tab view object*/
    lv_obj_t *tabview;
    tabview = lv_tabview_create(lv_scr_act(), NULL);

    /*Add 3 tabs (the tabs are page (lv_page) and can be scrolled*/
    lv_obj_t *tab1 = lv_tabview_add_tab(tabview, "Tab 1");
    lv_obj_t *tab2 = lv_tabview_add_tab(tabview, "Tab 2");
    lv_obj_t *tab3 = lv_tabview_add_tab(tabview, "Tab 3");

    /*Add content to the tabs*/
    lv_obj_t * label = lv_label_create(tab1, NULL);
    lv_label_set_text(label, "This the first tab\n\n"
        "If the content\n"
        "of a tab\n"
        "become too long\n"
        "the it \n"
        "automatically\n"
        "become\n"
        "scrollable.");

    label = lv_label_create(tab2, NULL);
    lv_label_set_text(label, "Second tab");

    label = lv_label_create(tab3, NULL);
    lv_label_set_text(label, "Third tab");
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t lv\_tabview\_btns\_pos\_t

**typedef** uint8\_t lv\_tabview\_style\_t

### Enums

**enum** [anonymous]

Position of tabview buttons.

*Values:*

LV\_TABVIEW\_BTNS\_POS\_TOP

LV\_TABVIEW\_BTNS\_POS\_BOTTOM

LV\_TABVIEW\_BTNS\_POS\_LEFT

LV\_TABVIEW\_BTNS\_POS\_RIGHT

**enum** [anonymous]

*Values:*

LV\_TABVIEW\_STYLE\_BG  
 LV\_TABVIEW\_STYLE\_INDIC  
 LV\_TABVIEW\_STYLE\_BTN\_BG  
 LV\_TABVIEW\_STYLE\_BTN\_REL  
 LV\_TABVIEW\_STYLE\_BTN\_PR  
 LV\_TABVIEW\_STYLE\_BTN\_TGL\_REL  
 LV\_TABVIEW\_STYLE\_BTN\_TGL\_PR

## 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

void **lv\_tabview\_clean**(*lv\_obj\_t* \*obj)

Delete all children of the scr1 object, without deleting scr1 child.

**Parameters**

- **obj**: pointer to an object

*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\_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\_sliding**(*lv\_obj\_t* \*tabview, bool en)

Enable horizontal sliding with touch pad

**Parameters**

- **tabview**: pointer to Tab view object

- **en**: true: enable sliding; false: disable sliding

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\_style**(*lv\_obj\_t* \**tabview*, *lv\_tabview\_style\_t* *type*, **const** *lv\_style\_t* \**style*)  
Set the style of a tab view

**Parameters**

- **tabview**: pointer to a tan view object
- **type**: which style should be set
- **style**: pointer to the new style

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

void **lv\_tabview\_set\_btns\_hidden**(*lv\_obj\_t* \**tabview*, *bool* *en*)  
Set whether tab buttons are hidden

**Parameters**

- **tabview**: pointer to a tab view object
- **en**: whether tab buttons are hidden

*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 ( $\geq 0$ )

bool **lv\_tabview\_get\_sliding**(const lv\_obj\_t \*tabview)

Get horizontal sliding is enabled or not

**Return** true: enable sliding; false: disable sliding

**Parameters**

- tabview: pointer to Tab view object

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

const lv\_style\_t \***lv\_tabview\_get\_style**(const lv\_obj\_t \*tabview, lv\_tabview\_style\_t type)

Get a style of a tab view

**Return** style pointer to a style

**Parameters**

- tabview: pointer to a ab view object
- type: which style should be get

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

bool **lv\_tabview\_get\_btns\_hidden**(const lv\_obj\_t \*tabview)

Get whether tab buttons are hidden

**Return** whether tab buttons are hidden

**Parameters**

- tabview: pointer to a tab 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**

uint8\_t **slide\_enable**

uint8\_t **draging**



```
uint8_t drag_hor
uint8_t scroll_ver
uint8_t btns_hide
lv_tabview_btns_pos_t btns_pos
```

## Text area (`lv_ta`)

### 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-

### Add text

You can insert text or characters to the current cursor's position with:

- `lv_ta_add_char(ta, 'c')`
- `lv_ta_add_text(ta, "insert this text")`

To add wide characters like 'á', 'ß' or CJK characters use `lv_ta_add_text(ta, "á")`.

`lv_ta_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_ta_set_placeholder_text(ta, "Placeholder text")`

### Delete character

To delete a character from the left of the current cursor position use `lv_ta_del_char(ta)`. The delete from the right use `lv_ta_del_char_forward(ta)`

### Move the cursor

The cursor position can be modified directly with `lv_ta_set_cursor_pos(ta, 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_ta_cursor_right(ta)`
- `lv_ta_cursor_left(ta)`
- `lv_ta_cursor_up(ta)`
- `lv_ta_cursor_down(ta)`

If `lv_ta_set_cursor_click_pos(ta, true)` is called the cursor will jump to the position where the Text area was clicked.

## Cursor types

There are several cursor types. You can set one of them with: `lv_ta_set_cursor_type(ta, LV_CURSOR_...)`

- **LV\_CURSOR\_NONE** No cursor
- **LV\_CURSOR\_LINE** A simple vertical line
- **LV\_CURSOR\_BLOCK** A filled rectangle on the current character
- **LV\_CURSOR\_OUTLINE** A rectangle border around the current character
- **LV\_CURSOR\_UNDERLINE** Underline the current character

You can 'OR' `LV_CURSOR_HIDDEN` to any type to temporarily hide the cursor.

The blink time of the cursor can be adjusted with `lv_ta_set_cursor_blink_time(ta, time_ms)`.

## One line mode

The Text area can be configured 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_ta_set_pwd_mode(ta, true)`. In password mode, the entered characters are converted to \* after some time or when a new character is entered.

In password mode `lv_ta_get_text(ta)` gives the real text and not the asterisk characters

The visibility time can be adjusted with `lv_ta_set_pwd_show_time(ta, time_ms)`.

## Text align

The text can be aligned to the left, center or right with `lv_ta_set_text_align(ta, LV_LABEL_ALIGN_LEFT/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_ta_set_accepted_chars(ta, "0123456789.+ -")`. Other characters will be ignored.

## Max text length

The maximum number of characters can be limited with `lv_ta_set_max_length(ta, max_char_num)`

## Very long texts

If there is a very long text in the Text area (> 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_ta_set_text_sel(ta, 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_ta_set_sb_mode(ta, LV_SB_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)`

## Style usage

Use `lv_ta_set_style(page, LV_TA_STYLE_..., &style)` to set a new style for an element of the text area:

- **LV\_TA\_STYLE\_BG** background’s style which uses all `style.body` properties. The label uses `style.label` from this style. (default: `lv_style_pretty`)
- **LV\_TA\_STYLE\_SB** scrollbar’s style which uses all `style.body` properties (default: `lv_style_pretty_color`)
- **LV\_TA\_STYLE\_CURSOR** cursor style. If `NULL` then the library sets a style automatically according to the label’s color and font
  - *LV\_CURSOR\_LINE*: a `style.line.width` wide line but drawn as a rectangle as `style.body.padding.top/left` makes an offset on the cursor
  - *LV\_CURSOR\_BLOCK*: a rectangle as `style.body.padding` makes the rectangle larger
  - *LV\_CURSOR\_OUTLINE*: an empty rectangle (just a border) as `style.body.padding` makes the rectangle larger

- `LV_CURSOR_UNDERLINE`: a `style.line.width` wide line but drawn as a rectangle as `style.body.padding.top/left` makes an offset on the cursor

## 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 event 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.

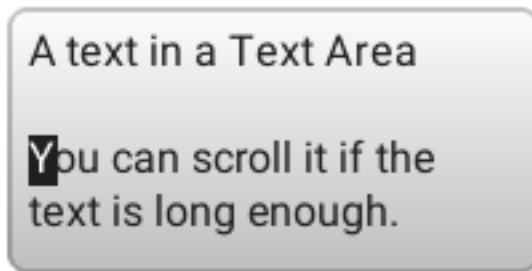
## Keys

- **LV\_KEY\_UP/DOWN/LEFT/RIGHT** Move the cursor
- **Any character** Add the character to the current cursor position

Learn more about *Keys*.

## Example

C



code

```
#include "lvgl/lvgl.h"
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
lv_obj_t * ta1;

static void event_handler(lv_obj_t * obj, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        printf("Value: %s\n", lv_ta_get_text(obj));
    }
    else if(event == LV_EVENT_LONG_PRESSED_REPEAT) {
        /*For simple test: Long press the Text are to add the text below*/
        const char * txt = "\n\nYou can scroll it if the text is long enough.\n";
        static uint16_t i = 0;
        if(txt[i] != '\0') {
            lv_ta_add_char(ta1, txt[i]);
            i++;
        }
    }
}

void lv_ex_ta_1(void)
{
    ta1 = lv_ta_create(lv_scr_act(), NULL);
    lv_obj_set_size(ta1, 200, 100);
    lv_obj_align(ta1, NULL, LV_ALIGN_CENTER, 0, 0);
    lv_ta_set_cursor_type(ta1, LV_CURSOR_BLOCK);
    lv_ta_set_text(ta1, "A text in a Text Area"); /*Set an initial text*/
    lv_obj_set_event_cb(ta1, event_handler);
}
```

## MicroPython

No examples yet.

## API

### Typedefs

```
typedef uint8_t lv_cursor_type_t
```

```
typedef uint8_t lv_ta_style_t
```

### Enums

```
enum [anonymous]
    Style of text area's cursor.
```

*Values:*

```
LV_CURSOR_NONE
```

No cursor

```
LV_CURSOR_LINE
```

Vertical line

## LV\_CURSOR\_BLOCK

Rectangle

## LV\_CURSOR\_OUTLINE

Outline around character

## LV\_CURSOR\_UNDERLINE

Horizontal line under character

## LV\_CURSOR\_HIDDEN = 0x08

This flag can be ORed to any of the other values to temporarily hide the cursor

## enum [anonymous]

Possible text areas tyles.

*Values:*

## LV\_TA\_STYLE\_BG

Text area background style

## LV\_TA\_STYLE\_SB

Scrollbar style

## LV\_TA\_STYLE\_CURSOR

Cursor style

## LV\_TA\_STYLE\_EDGE\_FLASH

Edge flash style

## LV\_TA\_STYLE\_PLACEHOLDER

Placeholder style

## Functions

*lv\_obj\_t* \***lv\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_set\_cursor\_pos**(*lv\_obj\_t \*ta, int16\_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\_TA\_CURSOR\_LAST: go after the last character

void **lv\_ta\_set\_cursor\_type**(*lv\_obj\_t \*ta, lv\_cursor\_type\_t cur\_type*)

Set the cursor type.

**Parameters**

- **ta**: pointer to a text area object
- **cur\_type**: element of 'lv\_cursor\_type\_t'

void **lv\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_set\_max\_length**(*lv\_obj\_t \*ta*, uint16\_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\_ta\_set\_text* ignores it)

void **lv\_ta\_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\_ta\_set\_sb\_mode**(*lv\_obj\_t \*ta*, *lv\_sb\_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\_page\_sb\_mode\_t*’ enum

**static** void **lv\_ta\_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\_ta\_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\_ta\_set\_style**(*lv\_obj\_t \*ta*, *lv\_ta\_style\_t type*, **const** *lv\_style\_t \*style*)  
 Set a style of a text area

**Parameters**

- **ta**: pointer to a text area object
- **type**: which style should be set
- **style**: pointer to a style

void **lv\_ta\_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\_ta\_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\_ta\_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\_ta\_get\_text**(**const** *lv\_obj\_t \*ta*)  
 Get the text of a text area. In password mode it gives the real text (not '\*').

**Return** pointer to the text

**Parameters**

- **ta**: pointer to a text area object

**const** char \***lv\_ta\_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\_ta\_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

uint16\_t **lv\_ta\_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

lv\_cursor\_type\_t **lv\_ta\_get\_cursor\_type**(const lv\_obj\_t \*ta)

Get the current cursor type.

**Return** element of 'lv\_cursor\_type\_t'

#### Parameters

- **ta**: pointer to a text area object

bool **lv\_ta\_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\_ta\_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\_ta\_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\_ta\_get\_accepted\_chars**(lv\_obj\_t \*ta)

Get a list of accepted characters.

**Return** list of accented characters.

#### Parameters

- **ta**: pointer to Text Area

uint16\_t **lv\_ta\_get\_max\_length**(lv\_obj\_t \*ta)

Set max length of a Text Area.

**Return** the maximal number of characters to be add

#### Parameters

- **ta**: pointer to Text Area

static lv\_sb\_mode\_t **lv\_ta\_get\_sb\_mode**(const lv\_obj\_t \*ta)

Get the scroll bar mode of a text area

**Return** scrollbar mode from 'lv\_page\_sb\_mode\_t' enum

**Parameters**

- **ta**: pointer to a text area object

**static** bool **lv\_ta\_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\_ta\_get\_edge\_flash**(lv\_obj\_t \*ta)

Get the scroll propagation property

**Return** true or false

**Parameters**

- **ta**: pointer to a Text area

**const** lv\_style\_t \***lv\_ta\_get\_style**(const lv\_obj\_t \*ta, lv\_ta\_style\_t type)

Get a style of a text area

**Return** style pointer to a style

**Parameters**

- **ta**: pointer to a text area object
- **type**: which style should be get

bool **lv\_ta\_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\_ta\_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\_ta\_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\_ta\_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\_ta\_clear\_selection**(*lv\_obj\_t \*ta*)  
Clear the selection on the text area.

#### Parameters

- **ta**: Text area object

void **lv\_ta\_cursor\_right**(*lv\_obj\_t \*ta*)  
Move the cursor one character right

#### Parameters

- **ta**: pointer to a text area object

void **lv\_ta\_cursor\_left**(*lv\_obj\_t \*ta*)  
Move the cursor one character left

#### Parameters

- **ta**: pointer to a text area object

void **lv\_ta\_cursor\_down**(*lv\_obj\_t \*ta*)  
Move the cursor one line down

#### Parameters

- **ta**: pointer to a text area object

void **lv\_ta\_cursor\_up**(*lv\_obj\_t \*ta*)  
Move the cursor one line up

#### Parameters

- **ta**: pointer to a text area object

**struct lv\_ta\_ext\_t**

#### Public Members

*lv\_page\_ext\_t* **page**

*lv\_obj\_t* \***label**

*lv\_obj\_t* \***placeholder**

char \***pwd\_tmp**

**const** char \***accapted\_chars**

uint16\_t **max\_length**

uint16\_t **pwd\_show\_time**

**const** lv\_style\_t \***style**

lv\_coord\_t **valid\_x**

uint16\_t **pos**

uint16\_t **blink\_time**

lv\_area\_t **area**

uint16\_t **txt\_byte\_pos**

*lv\_cursor\_type\_t* **type**

```

uint8_t state
uint8_t click_pos
struct lv_ta_ext_t::[anonymous] cursor
uint16_t tmp_sel_start
uint16_t tmp_sel_end
uint8_t text_sel_in_prog
uint8_t text_sel_en
uint8_t pwd_mode
uint8_t one_line
    
```

## Tile view (lv\_tileview)

### Overview

The Tileview 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.

### 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 be an empty row or column.

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 be 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

### Add element

To add elements just create an object on the Tileview and call `lv_tileview_add_element(tileview, element)`.

The element should have the same size than the Tile view and needs to be positioned manually to the desired position.

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.

`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.

It true for the buttons on a *List* as well. Every list button and the list itself needs to be added with `lv_tileview_add_element`.

## Set tile

To set the currently visible tile use `lv_tileview_set_tile_act(tileview, x_id, y_id, LV_ANIM_ON/OFF)`.

## Animation time

The animation time when a tile

- is selected with `lv_tileview_set_tile_act`
- is scrolled a little and then released (revert the original title)
- is scrolled more than half size and then release (move to the next tile)

can be set with `lv_tileview_set_anim_time(tileview, anim_time)`.

## 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.

## Styles

The Tileview has on one style which van be changes with `lv_tileview_set_style(slider, LV_TILEVIEW_STYLE_MAIN, &style)`.

- **LV\_TILEVIEW\_STYLE\_MAIN** Style of the background. All `style.body` properties are used.

## 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 *`)

## 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*.

## Example

### C



code

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

void lv_ex_tileview_1(void)
{
    static lv_point_t valid_pos[] = {{0,0}, {0, 1}, {1,1}};
    lv_obj_t *tileview;
    tileview = lv_tileview_create(lv_scr_act(), NULL);
    lv_tileview_set_valid_positions(tileview, valid_pos, 3);
    lv_tileview_set_edge_flash(tileview, true);

    lv_obj_t * tile1 = lv_obj_create(tileview, NULL);
    lv_obj_set_size(tile1, LV_HOR_RES, LV_VER_RES);
    lv_obj_set_style(tile1, &lv_style_pretty);
    lv_tileview_add_element(tileview, tile1);

    /*Tile1: just a label*/
    lv_obj_t * label = lv_label_create(tile1, NULL);
    lv_label_set_text(label, "Tile 1");
    lv_obj_align(label, NULL, LV_ALIGN_CENTER, 0, 0);
}
```

(continues on next page)

(continued from previous page)

```

/*Tile2: a list*/
lv_obj_t * list = lv_list_create(tileview, NULL);
lv_obj_set_size(list, LV_HOR_RES, LV_VER_RES);
lv_obj_set_pos(list, 0, LV_VER_RES);
lv_list_set_scroll_propagation(list, true);
lv_list_set_sb_mode(list, LV_SB_MODE_OFF);
lv_tileview_add_element(list, list);

lv_obj_t * list_btn;
list_btn = lv_list_add_btn(list, NULL, "One");
lv_tileview_add_element(tileview, list_btn);

list_btn = lv_list_add_btn(list, NULL, "Two");
lv_tileview_add_element(tileview, list_btn);

list_btn = lv_list_add_btn(list, NULL, "Three");
lv_tileview_add_element(tileview, list_btn);

list_btn = lv_list_add_btn(list, NULL, "Four");
lv_tileview_add_element(tileview, list_btn);

list_btn = lv_list_add_btn(list, NULL, "Five");
lv_tileview_add_element(tileview, list_btn);

/*Tile3: a button*/
lv_obj_t * tile3 = lv_obj_create(tileview, tile1);
lv_obj_set_pos(tile3, LV_HOR_RES, LV_VER_RES);
lv_tileview_add_element(tileview, tile3);

lv_obj_t * btn = lv_btn_create(tile3, NULL);
lv_obj_align(btn, NULL, LV_ALIGN_CENTER, 0, 0);

label = lv_label_create(btn, NULL);
lv_label_set_text(label, "Button");
}

```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_tileview\_style\_t**

### Enums

**enum** [anonymous]

*Values:*

**LV\_TILEVIEW\_STYLE\_MAIN**



## 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**: numner 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_set_style(lv_obj_t *tileview, lv_tileview_style_t type, const lv_style_t
                          *style)
```

Set a style of a tileview.

#### Parameters

- **tileview**: pointer to tileview object
- **type**: which style should be set
- **style**: pointer to a style

```
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

```
const lv_style_t *lv_tileview_get_style(const lv_obj_t *tileview, lv_tileview_style_t
                                       type)
```

Get style of a tileview.

**Return** style pointer to the style

#### Parameters

- **tileview**: pointer to tileview object
- **type**: which style should be get

```
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**

uint8\_t **drag\_hor**

uint8\_t **drag\_ver**

## Window (lv\_win)

### Overview

The windows are one of the most complex container-like objects. They are built from two main parts:

1. a header *Container* on the top
2. a *Page* for the content below the header.

### Title

On the header, there is a title which can be modified by: `lv_win_set_title(win, "New title")`. The title always inherits the style of the header.

### Control buttons

You can add control buttons to the right side of the header with: `lv_win_add_btn(win, LV_SYMBOL_CLOSE)`. The second parameter is an *Image* source.

`lv_win_close_event_cb` can be used as an event callback to close the Window.

You can modify the size of the control buttons with the `lv_win_set_btn_size(win, new_size)` function.

### Scrollbars

The scrollbar behavior can be set by `lv_win_set_sb_mode(win, LV_SB_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.

### Style usage

Use `lv_win_set_style(win, LV_WIN_STYLE_..., &style)` to set a new style for an element of the Window:

- **LV\_WIN\_STYE\_BG** main background which uses all **style.body** properties (header and content page are placed on it) (default: **lv\_style\_plain**)
- **LV\_WIN\_STYLE\_CONTENT** content page's scrollable part which uses all **style.body** properties (default: **lv\_style\_transp**)
- **LV\_WIN\_STYLE\_SB** scroll bar's style which uses all **style.body** properties. **left/top** padding sets the scrollbars' padding respectively and the inner padding sets the scrollbar's width. (default: **lv\_style\_pretty\_color**)
- **LV\_WIN\_STYLE\_HEADER** header's style which uses all **style.body** properties (default: **lv\_style\_plain\_color**)
- **LV\_WIN\_STYLE\_BTN\_REL** released button's style (on header) which uses all **style.body** properties (default: **lv\_style\_btn\_rel**)
- **LV\_WIN\_STYLE\_BTN\_PR** pressed button's style (on header) which uses all **style.body** properties (default: **lv\_style\_btn\_pr**)

## Events

Only the [Generic events](#) are sent by the object type.

Learn more about *Events*.

## Keys

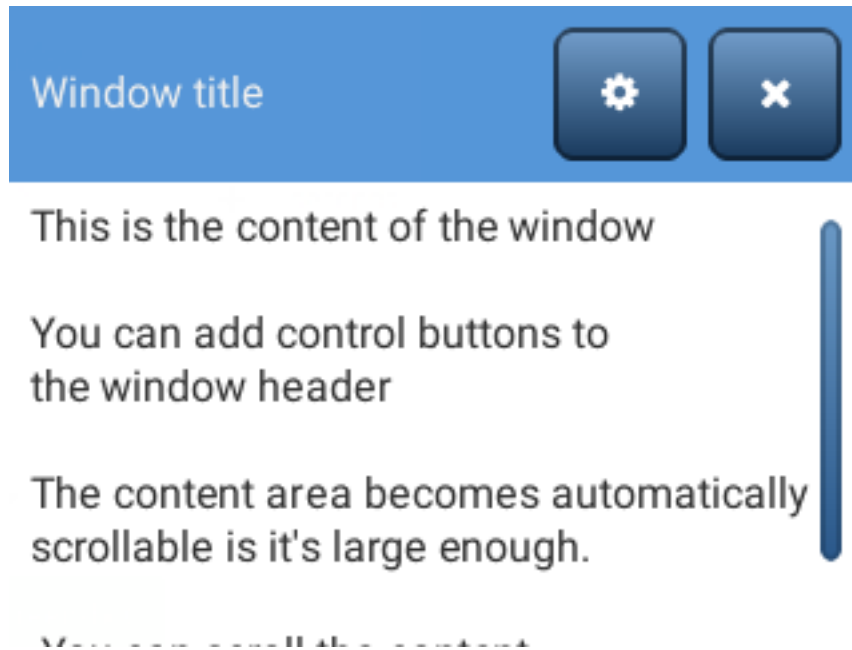
The following *Keys* are processed by the Page:

- **LV\_KEY\_RIGHT/LEFT/UP/DOWN** Scroll the page

Learn more about *Keys*.

## Example

## C



code

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

void lv_ex_win_1(void)
{
    /*Create a window*/
    lv_obj_t * win = lv_win_create(lv_scr_act(), NULL);
    lv_win_set_title(win, "Window title"); /*Set the title*/

    /*Add control button to the header*/
    lv_obj_t * close_btn = lv_win_add_btn(win, LV_SYMBOL_CLOSE); /*Add
↪close button and use built-in close action*/
    lv_obj_set_event_cb(close_btn, lv_win_close_event_cb);
    lv_win_add_btn(win, LV_SYMBOL_SETTINGS); /*Add a setup button*/

    /*Add some dummy content*/
    lv_obj_t * txt = lv_label_create(win, NULL);
    lv_label_set_text(txt, "This is the content of the window\n\n"
        "You can add control buttons to\n"
        "the window header\n\n"
        "The content area becomes automatically\n"
        "scrollable is it's large enough.\n\n"
        " You can scroll the content\n"
        "See the scroll bar on the right!");
}
```

## MicroPython

No examples yet.

## API

### Typedefs

**typedef** uint8\_t **lv\_win\_style\_t**

### Enums

**enum** [anonymous]

Window styles.

*Values:*

**LV\_WIN\_STYLE\_BG**

Window object background style.

**LV\_WIN\_STYLE\_CONTENT**

Window content style.

**LV\_WIN\_STYLE\_SB**

Window scrollbar style.

**LV\_WIN\_STYLE\_HEADER**

Window titlebar background style.

**LV\_WIN\_STYLE\_BTN\_REL**

Same meaning as ordinary button styles.

**LV\_WIN\_STYLE\_BTN\_PR**

### 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* \**obj*)

Delete all children of the scr1 object, without deleting scr1 child.

**Parameters**

- **obj**: 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 window's header
- **event**: 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\_btn\_size**(*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\_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\_sb\_mode**(*lv\_obj\_t \*win*, *lv\_sb\_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\_sb\_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\_style**(*lv\_obj\_t \*win*, *lv\_win\_style\_t type*, **const** *lv\_style\_t \*style*)

Set a style of a window

#### Parameters

- **win**: pointer to a window object
- **type**: which style should be set
- **style**: pointer to a style

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\_btn\_size**(**const** lv\_obj\_t \*win)

Get the control button size of a window

**Return** control button size

**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\_sb\_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

`const lv_style_t *lv_win_get_style(const lv_obj_t *win, lv_win_style_t type)`

Get a style of a window

**Return** style pointer to a style

**Parameters**

- **win**: pointer to a button object
- **type**: which style window be get

`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`

```

lv_obj_t *title
const lv_style_t *style_btn_rel
const lv_style_t *style_btn_pr
lv_coord_t btn_size

```