LittlevGL Documentation

Release 6.0

Gabor Kiss-Vamosi

Jun 13, 2019

CONTENTS

1 Table of content **3**

1.1 Introduction . . . 3
1.1.1 Key features . . . 3
1.1.2 Requirements . . . 3
1.2 Get started . . . 4
1.2.1 Live demos . . . 4
1.2.2 Micropython . . . 4
1.2.3 Simulator on PC . . . 4
1.3 Porting . . . 7
1.3.1 System overview . . . 7
1.3.2 Set-up a project . . . 7
1.3.3 Display interface . . . 8
1.3.4 Input device interface . . . 11
1.3.5 Tick interface . . . 13
1.3.6 Task handler . . . 14
1.3.7 Sleep management . . . 14
1.3.8 Using with an operating system . . . 14
1.4 Overview . . . 15
1.4.1 Objects . . . 15
1.4.2 Styles . . . 18
1.4.3 Fonts . . . 22
1.4.4 Animations . . . 24

English - Portuguese - Espanol - Turkish

LittlevGL is a free and open-source graphics library providing everything you need to create embedded GUI with easy-to-use graphical elements, beautiful visual effects and low memory footprint.

CHAPTER

ONE

TABLE OF CONTENT

1.1 Introduction

LittlevGL is a free and open-source graphics library providing everything you need to create embedded GUI with easy-to-use graphical elements, beautiful visual effects and low memory footprint.

1.1.1 Key features

- Powerful building blocks buttons, charts, lists, sliders, images etc
- Advanced graphics with animations, anti-aliasing, opacity, smooth scrolling
- Various input devices touch pad, mouse, keyboard, encoder etc
- Multi language support with UTF-8 encoding
- Fully customizable graphical elements
- Hardware independent to use with any microcontroller or display
- Scalable to operate with little memory (80 kB Flash, 10 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 licence

1.1.2 Requirements

- 16, 32 or 64 bit microcontroller or processor
- 16 MHz clock speed
- 8 kB RAM for static data and >2 KB RAM for dynamic data (graphical objects)
- 64 kB program memory (flash)
- Optionally ~1/10 screen sized memory for internal buffering (at 240 × 320, 16 bit colors it means 15 kB)

- C99 or newer compiler

The LittlevGL is designed to be highly portable and to not use any external resources:

- No external RAM required (but supported)
- No float numbers are used
- No GPU needed (but supported)
- Only a single frame buffer is required located in:
  - Internal RAM or
  - External RAM or
  - External display controller's memory

If you would like to reduce the required hardware resources you can:

- Disable the unused object types to save RAM and ROM
- Change the size of the graphical buffer to save RAM (see later)
- Use more simple styles to reduce the rendering time

1.2 Get started

- lvgl on GihHub
- example projects

1.2.1 Live demos

See look and feel

1.2.2 Micropython

play with it in micropython

1.2.3 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 the reproduce a bug in simualtor and use the code snippen in the Forum.

Select an IDE

The simulator is ported to valrious IDEs. Choose your favourite 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. Uncompress 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
- Righ 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

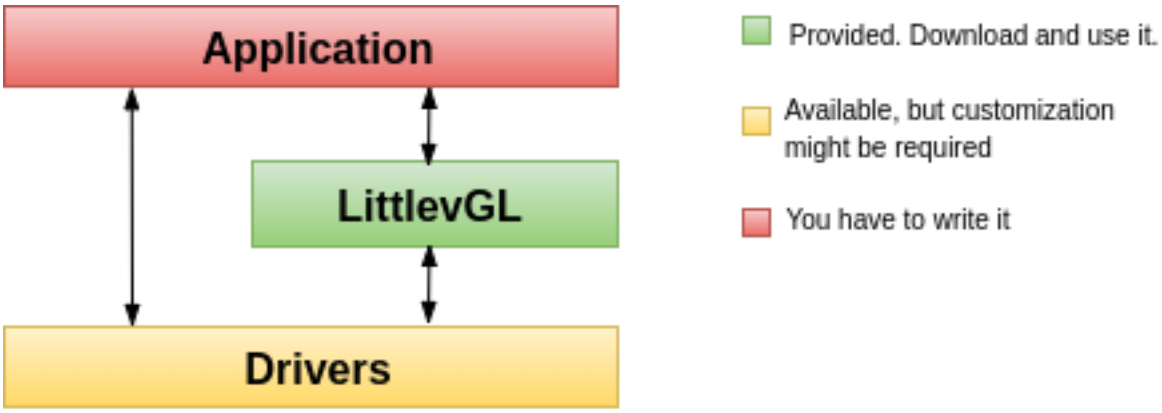
Now you are ready to run the Littlev Graphics Library on your PC. Click on the Hammer Icon on the top menu bar to Build the project. If you have done everything right you will not get any errors. Note that on some systems additional steps might be required to "see" SDL 2 from Eclipse but in most of cases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

1.3 Porting

1.3.1 System overview



System architecture of Littlev Graphics Library

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.

1.3.2 Set-up a project

Get the library

The Littlev Graphics Library is available on GitHub: https://github.com/littlevgl/lvgl. You can clone or download the latest version of the library from here 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, disable unused modules and features, adjust the size of memory buffers in compile time.

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.

In the config file comments explain the meaning of the options. Check at least these three config 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. (see below)
4. Call `lv_tick_inc(x)` in every `x` milliseconds in an interrupt to tell the elapsed time. (see below)
5. Call `lv_task_handler()` periodically in every few milliseconds to handle LittlevGL realted tasks. (see below)

1.3.3 Display interface

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

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

Display buffer

lv_disp_buf_t can bin initalized 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];

/*Initalize `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 to the buffer size:

1. **Only one buffer** this buffer will be used the render the conent of the display. Should enough to hold at least 10 lines. LittlevGL will redraw the screen in chucks whcih fit into the buffer. However if only a smaller area is needs to be redrawn (like buttn when pressed) only the that small area will be redrawn. It can be screen-sized as well.

2. **Two non screen-sized buffers** having two buffer 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 *Only one buffer* LittelvGL will drawn the display's content in chunks which size is at most the size of the buffer.
3. **Two screen-sized buffers** In contast to *Two non screen-sized buffers* LittelvGL 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 inteface and the frambuffer is just a loaction in the RAM.

Display driver

Once the buffer iitialization is ready the display drivers needs to initalized. 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.

And there are some optinal data fields:

- **hor_res** horizontal resolution of teh display. (*LV_HOR_RES_MAX* by default)
- **ver_res** vertical resolution of teh display. (*LV_VER_RES_MAX* by default)
- **color_chroma_key** a color which will be drawn as transparent on CHhrome keyed images. *LV_COLOR_TRANSP* by deafult (lv_conf.h)
- **user_data** custom user data for the driver. Its type acn be modified in lv_conf.h. (Optional)
- **antialiasing** use anti-aliasing (edge smoothing). *LV_ANTIALIAS* by deafult (lv_conf.h)
- **rotated** if `1` swap *hor_res* and *ver_res*. LittelvGL draws in tha same direction in both case (in lines from top to buttom) so the driver also need to be reconfigured to change the display's fill direction.

To use a GPU the following callbacks can be used:

- **mem_fill_cb** fill an area with a colors.
- **mem_blend_cb** blend two buffers using opacity.

Some other optional callbacks to make easier and more optimal to work with monochrome, grayscale or other less standard dislays:

- **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 *buffer*. It can be used to store the pixels in more compact way if the display has a special color format. (e.g. 1 bit monochrome, 2 bit grayscale etc.) 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 intialized with `lv_disp_drv_init(&disp_drv)`. And finally to register a display for LittlevGL the `lv_disp_drv_register(&disp_drv)` needs to be used.

All togother it looks like this:

```
    lv_disp_drv_t disp_drv;                 /*A variable to hold the drivers. Can be␣
↪local varaible*/
    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 direver and save the␣
↪created display objects*/
```

Here some simple examples of the callbacks:

```
void my_flush_cb(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t * color_
↪p)
{
    /*The most simple case (but also the slowest) to put all pixels to the screen one-
↪by-one*/
    int32_t x, y;
    for(y = area->y1; y <= area->y2; y++) {
        for(x = area->x1; x <= area->x2; x++) {
            put_px(x, y, *color_p)
            color_p++;
        }
    }

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

void my_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; i < length; i++) {
        dest[i] = 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++) {
        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 hegiht:*/
   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);
}
```

Multi-display support

In LittlevGL multiple displays can be used. Just initlaizes multiple drivers and buffer and register them. Each display have its own screens and objects on the screens. To get curently active scrren of a dsplay use `lv_disp_get_scr_act(disp)` (where `disp` is the return value of `lv_disp_drv_register`). To set a new screen as active on a display use `lv_disp_set_scr_act(screen1)`.

Or in a shorter form set a default display with `lv_disp_set_default(disp)` and get/set the active screen with `lv_scr_act()` and `lv_scr_load()`.

Learn morea bout screens in the Objects section.

1.3.4 Input device interface

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 =.../*See below.*/
lv_indev_drv_register(&indev_drv);  /*Register the driver in LittlevGL*/
```

type can be

- LV_INDEV_TYPE_POINTER: touchpad or mouse
- LV_INDEV_TYPE_KEYPAD: keyboard
- LV_INDEV_TYPE_ENCODER: left, right, push
- LV_INDEV_TYPE_BUTTON: external buttons pressing the screen

read 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.

Touchpad, mouse or any pointer

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

The read function should look like this:

```
bool my_input_read(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 so no more data read*/
}
```

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

Keypad or keyboard

```
indev_drv.type = LV_INDEV_TYPE_KEYPAD;
indev_drv.read = my_input_read;
```

The read function should look like this:

```
bool keyboard_read(lv_indev_data_t*data){
  data->key = last_key();            /*Set 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 so no more data read*/
}
```

To use a keyboard:

- Register a *read* function (like above) with *LV_INDEV_TYPE_KEYPAD* type.
- *USE_LV_GROUP* has to be enabled in *lv_conf.h*
- An object group has to be created: *lv_group_create()* and objects have to be added: *lv_group_add_obj()*
- The created group has to be assigned to an input device: *lv_indev_set_group(my_indev, group1);*
- Use *LV_GROUP_KEY...* to navigate among the objects in the group

Visit Touchpad-less navigation to learn more.

Encoder

With an encoder you can do 4 things:

1. press its button
2. long press its button
3. turn left
4. turn right

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.

```
indev_drv.type = LV_INDEV_TYPE_ENCODER;
indev_drv.read = my_input_read;
```

The read function should look like this:

```
bool encoder_read(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 so no more data read*/
}
```

- To use an `ENCODER`, similarly to the `KEYPAD`, the objects should be added to groups

Button

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

The read function should look like this:

```
bool button_read(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?*/
       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;            /*Set the last button*/

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

- The buttons need to be assigned to pixels on the screen using `lv_indev_set_button_points(indev, points_array)`. Where *points_array* look like `const lv_point_t points_array[] = { {12, 30},{60,90}, ...}`

1.3.5 Tick interface

The LittlevGL uses a system tick. Call the `lv_tick_inc(tick_period)` function periodically and tell the call period in milliseconds. For example if called in every milliseconds: `lv_tick_inc(1)`. It is required for LittlevGL to know the elapsed time. Therefore `lv_tick_inc` should be called in a higher priority then `lv_task_handler()`, for example in an interrupt.

1.3.6 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);
}
```

1.3.7 Sleep management

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

```
while(1) {
  /*Normal operation 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_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.

1.3.8 Using with an operating system

LittlevGL is **not thread-safe** by deafult. Despite it, it's quite simple to use LittlevGL inside an operating system.

The **simple scenario** is to don't use the operating system's tasks but use `lv_tasks`. An *lv_task* is a function called periodically in `lv_task_handler`. In the *lv_task* you can get the state of the sensors, buffers etc and call LittlevGL functions to refresh the GUI. To create an *lv_task* use: `lv_task_create(my_func, period_ms, LV_TASK_PRIO_LOWEST/LOW/MID/HIGH/HIGHEST, custom_ptr)`

If you need to **use other task or threads** you need one mutex which should be taken before calling `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 code. This way you can use LittlevGL in a real multitasking environment. Just use a mutex to avoid concurrent calling of LittlevGL functions.

1.4 Overview

1.4.1 Objects

In the Littlev Graphics Library the **basic building blocks** of a user interface are the objects. For example:

Click to check all the existing Object types

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. For example a slider have:

- Min. max. values
- Current value
- Callback function for new value set
- 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);                             /*Set the current value␣
↪(position)*/
lv_slider_set_action(slider1, my_action);                     /*Set a callback function*/
```

Object's working mechanisms

Parent-child structure

A parent 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 typically parent (e.g. button) and typical child (e.g. label) objects.

Screen – the most basic parent

The screen is a special object which has no parent object. Always there is an active screen. By default, the library creates and loads one. To get the currently active screen use the `lv_scr_act()` function.

A screen can be created with any object type, for example, a basic object or an image to make a wallpaper.

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. So 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.

Objects are moving together 1

```
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:

Graphical objects are moving togother 2

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

Visibility only on parent

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

A graphical object is visible on its parent

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

Create - delete objects

In the graphics library 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 only when it is required and delete after it is used.

Every objects type has its own **create** function with an unified prototype. It needs two parameters: a pointer the parent object and optionally a pointer to an other object with the same type. If the second parameter is not *NULL* then this objects will be copied to the new one. To create a screen give *NULL* as parent. The return value of the create function is a pointer to the created 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);
```

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

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

Layers

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.

Creating graphical objects in Littlev Graphics Library

```
/*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);
```

1.4.2 Styles

To set the appearance of the objects styles can be used. A style is a structure variable with attributes like colors, paddings, visibility, and others. There is common style type: **lv_style_t**.

By setting the fields of an `lv_style_t` structure you can influence the appearance of the objects using that style.

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_1;            /*WRONG! Styles can't be local variables*/

  ...
}
```

Style properties

A style has 5 main parts: common, body, text, image and line. An object will use that fields which are relevant for 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 (see below)
- **Body style properties** Used by the rectangle-like objects
  - **body.empty** Do not fill the rectangle (just draw border and/or shadow)
  - **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
  - **body.shadow.color** Shadow color
  - **body.shadow.width** Shadow width
  - **body.shadow.type** Shadow type (LV_SHADOW_BOTTOM or LV_SHADOW_FULL)
  - **body.padding.hor** Horizontal padding
  - **body.padding.ver** Vertical padding
  - **body.padding.inner** Inner padding

- **Text style properties** Used by the objects which show texts
  - **text.color** 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)

Using styles

Every object type has a unique function to set its style or styles.

If the object has only one style - like a label - the `lv_label_set_style(label1, &style)` function can be used to set a new style.

If the object has more styles (like a button have 5 styles for each state) `lv_btn_set_style(obj, LV_BTN_STYLE_..., &rel_style)` function can be used to set a new style.

The styles and the style properties used by an object type are described in their documentation.

If you **modify a style which is 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:

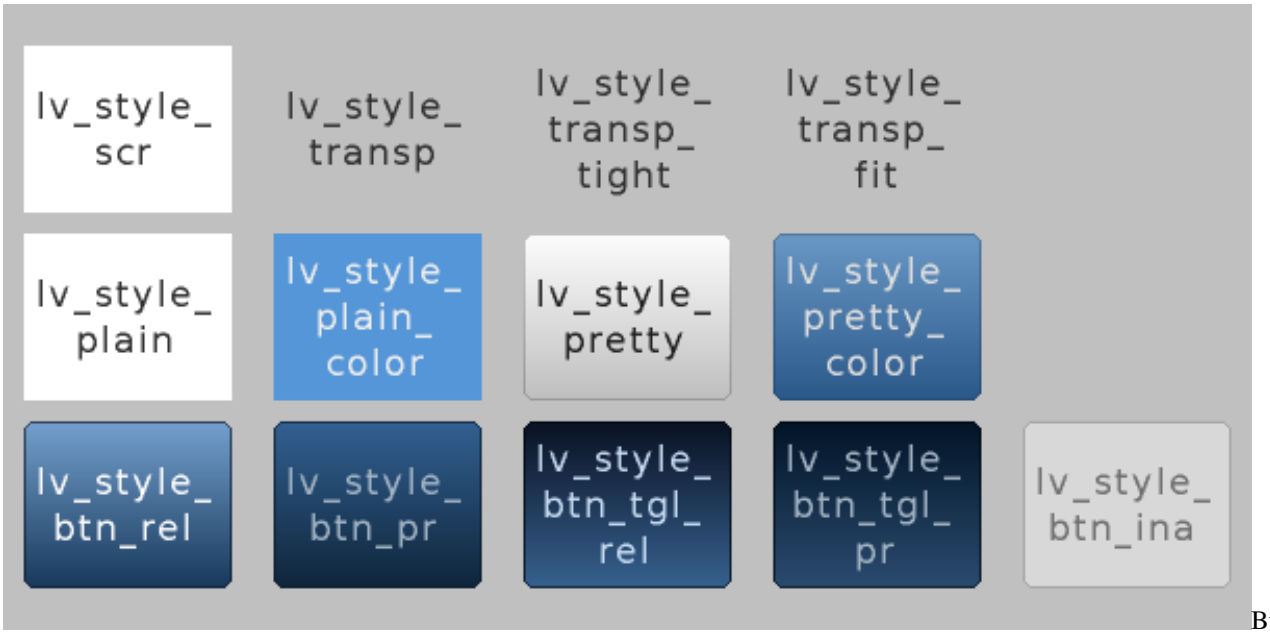
```
void lv_obj_refresh_style(lv_obj_t * obj);                /*Notify an object about␣
↪its style is modified*/
void lv_obj_report_style_mod(void * style);               /*Notify all object if a␣
↪style is modified.(NULL to notify all objects)*/
```

If the **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 buttons styles. 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 parent.

Built-in styles

There are several built-in styles in the library:



Built in styles in LittlevGL Embedded Graphics 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 padings are zero, for `lv_style_transp_fit` only hor and ver paddings are zero.

The built in styles are global lv_style_t variables so you can use them like: lv_btn_set_style(obj, LV_BTN_STYLE_REL, &lv_style_btn_rel)

You can modify the built-in styles or you can create new styles. When creating new styles it is recommended to first copy a built-in style to be sure all fields are initialized with a proper value. The lv_style_copy(&dest_style, &src_style) can be used to copy styles.

Style animations

You can animate styles using `lv_style_anim_create(&anim)`. Before calling this function you have to initialize an `lv_style_anim_t` variable. The animation will fade a `style_1` to `style_2`.

```
lv_style_anim_t a;     /*Will be copied, can be local variable*/
a.style_anim = & style_to_anim;      /*Pointer to style to animate*/
a.style_start = & style_1;           /*Pointer to the initial style (only pointer␣
↪saved) */
a.style_end = & style_2;             /*Pointer to the target style (only pointer␣
↪saved) */
a.act_time = 0;                      /*Set negative to make a delay*/
a.time = 1000;                       /*Time of animation in milliseconds*/
a.playback = 0;                      /*1: play the animation backward too*/
a.playback_pause = 0;                /*Wait before playback [ms]*/
a.repeat = 0;                        /*1: repeat the animation*/
a.repeat_pause = 0;                  /*Wait before repeat [ms]*/
a.end_cb = NULL;                     /*Call this function when the animation ready*/
```

Style example

The example below demonstrates the above-described style usage

Styles usage example in LittlevGL Embedded Graphics Library

```
/*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;
style1.body.border.color = LV_COLOR_GRAY;
style1.body.border.width = 2;
style1.body.border.opa = LV_OPA_50;
style1.body.padding.hor = 5;            /*Horizontal padding, used by the bar indicator below*/
style1.body.padding.ver = 5;            /*Vertical padding, used by the bar indicator below*/
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.

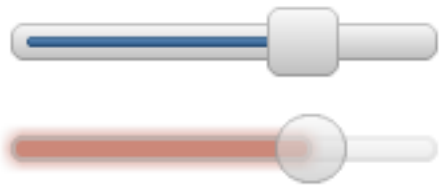
To speed up the design part themes are introduced. A theme is a style collection which contains the required styles for every object type. For example 5 styles for buttons to describe their 5 possible states. Check the Existing themes.

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 initialized by: `lv_theme_xxx_init(hue, font)`. Where xxx is the name of the theme, *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` default font)

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



Theme usage example in Littlev Embedded Graphics Library

```
/*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 redish 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 apply the styles from a theme when you create new objects. To do this use `lv_theme_set_current(th);`

1.4.3 Fonts

In LittlevGL fonts are bitmaps and other descriptors to store the images of the letters (glyph) and some additional information. A font is stored in a **lv_font_t** variable and can be set it in style's *text.font* field.

The fonts have a **bpp (Bit-Per-Pixel)** property. It shows how much bit is used to describe a pixel in the font. The value stored for a pixel determines the pixel's opacity. This way the image of the letters (especially on the edges) can be smooth and even. 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's memory size 4 times greater compared to bpp = 1.

Built-in fonts

There are several built-in fonts which can be enabled in lv_conf.h by *USE_LV_FONT..._* defines. There are built-in fonts in **different sizes**:

- 10 px
- 20 px
- 30 px
- 40 px

You can enable the fonts with 1, 2, 4 or 8 values to set its bpp (e.g. `#define USE_LV_FONT_DEJAVU_20 4` in `lv_conf.h`).

The built-in fonts exist with **multiply character-sets** in each size:

- ASCII (Unicode 32..126)
- Latin supplement (Unicode 160..255)
- Cyrillic (Unicode 1024..1279)

The built-in fonts use the *Dejavu* font.

The built-in fonts are **global variables** with names like:

- `lv_font_dejavu_20` (20 px ASCII font)
- `lv_font_dejavu_20_latin_sup` (20 px Latin supplement font)
- `lv_font_dejavu_20_cyrillic` (20 px Cyrillic font)

Unicode support

The LittlevGL supports Unicode letter from **UTF-8** coded characters. You need to configure your editor to save your code/text as UTF-8 (usually this the default) and enable *LV_TXT_UTF8* in lv_conf.h. Without enabled *LV_TXT_UTF8* only ASCII fonts and symbols can be used (see the symbols below)

After it the texts will be decoded to determine the Unicode values. To display the letters your font needs to contain the image (glyph) of the characters.

You can assign more fonts to create a **larger character-set**. To do this choose a base font (typically the ASCII font) and add the extensions to it: `lv_font_add(child, parent)`. Only fonts with the same height can be assigned.

The built-in fonts are already added to the same sized ASCII font. For example if *USE_LV_FONT_DEJAVU_20* and *USE_LV_FONT_DEJAVU_20_LATIN_SUP* are enabled in `lv_conf.h` then the *"abcÁÖÜ"* text can be rendered when using *lv_font_dejavu_20*.

Symbol fonts

The symbol fonts are special fonts which contain symbols instead of letters. There are **built-in symbol fonts** as well and they are also assigned to the ASCII font with the same size. In a text, a symbol can be referenced like *SYMBOL_LEFT*, *SYMBOL_RIGHT* etc. You can mix these symbol names with strings:

```
lv_label_set_text(label1,"Right "SYMBOL_RIGHT);
```

The symbols can be used without UTF-8 support as well. (LV_TXT_UTF8 0)

The list below shows the existing symbols:

Basic symbols

Add new font

If you want to **add new fonts to the library** you can use the Online Font Converter Tool. It can create a C array from a TTF file which can be copied copy to your project. You can specify the height, the range of characters and the bpp. Optionally you can enumerate the characters to include only them into the final font. To use the generated font declare it with *LV_FONT_DECLARE(my_font_name)*. After that, the font can be used as the built-in fonts.

Font example

Fonts example

```
/*Create a new style for the label*/
static lv_style_t style;
lv_style_copy(&style, &lv_style_plain);
style.text.color = LV_COLOR_BLUE;
style.text.font = &lv_font_dejavu_40;   /*Unicode and symbol fonts already assigned␣
↪by the library*/

lv_obj_t *label;

/*Use ASCII and Unicode letters*/
label = lv_label_create(lv_scr_act(), NULL);
lv_obj_set_pos(label, 20, 20);
lv_label_set_style(label, &style);
lv_label_set_text(label, "aeuois\n"
                         "äéüöíß");

/*Mix text and symbols*/
label = lv_label_create(lv_scr_act(), NULL);
lv_obj_set_pos(label, 20, 100);
lv_label_set_style(label, &style);
lv_label_set_text(label, "Right "SYMBOL_RIGHT);
```

1.4.4 Animations

You can automatically change the value (animate) of a variable between a start and an end value using an **animator function** with `void func(void* var,int32_t value)` prototype. The animation will happen by the periodical calling of the animator function with the corresponding value parameter.

To **create an animation** you have to initializes an *lv_anim_t* variable (there is a template in lv_anim.h):

```
lv_anim_t a;
a.var = button1;                                                    /*Variable␣
↪to animate*/
a.start = 100;                                                        /
↪*Start value*/
a.end = 300;                                                        /*End␣
↪value*/
a.fp = (lv_anim_fp_t)lv_obj_set_height;                               /
↪*Function to be used to animate*/
a.path = lv_anim_path_linear;                                       /*Path of␣
↪animation*/
a.end_cb = NULL;                                                     /*Callback␣
↪when the animation is ready*/
```

```
a.act_time = 0;                                                    /*Set
↪< 0 to make a delay [ms]*/
a.time = 200;                                                     /
↪*Animation length [ms]*/
a.playback = 0;                                                    /*1:␣
↪animate in reverse direction too when the normal is ready*/
a.playback_pause = 0;                                              /*Wait␣
↪before playback [ms]*/
a.repeat = 0;                                                      /*1:␣
↪Repeat the animation (with or without playback)*/
a.repeat_pause = 0;                                               /*Wait␣
↪before repeat [ms]*/

lv_anim_create(&a);                          /*Start the animation*/
```

The `anim_create(&a)` will register the animation and immediately **applies the *start*** value regardless to the set delay.

You can determinate the **path of animation**. In 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 two built-in paths:

- **lv_anim_path_linear** linear animation
- **lv_anim_path_step** change in one step at the end

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.

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* end *lv_obj_set_y*). But only one animation can exist with a given variable and function pair. Therefore the *lv_anim_create()* function will delete the already existing variable-function animations.

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