LittlevGL Documentation

Release 6.0

Gabor Kiss-Vamosi

Jun 12, 2019

CONTENTS

Some text and flags to select language.

CHAPTER

ONE

GET STARTED

- lvgl on GihHub
- example projects

1.1 Live demos

See look and feel

1.2 Micropython

play with it in micropython

1.3 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.

1.3.1 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.

1.3.2 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.

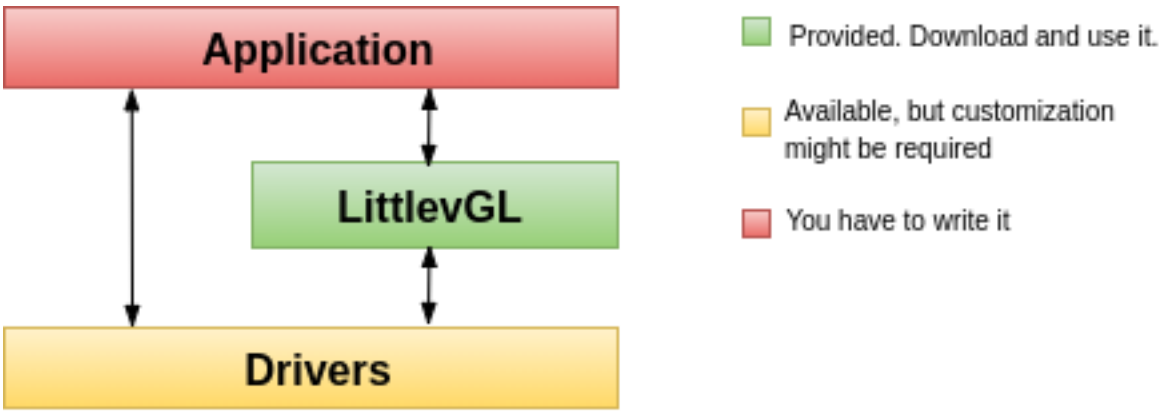
Now everything is ready to use the Littlev Graphics Library in the practice or begin the development on your PC.

CHAPTER

TWO

PORTING

2.1 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.

2.2 Set-up a project

2.2.1 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.

2.2.2 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).

2.2.3 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)

2.3 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.

2.3.1 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.

2.3.2 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);
}
```

2.3.3 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.

2.4 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.

2.4.1 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`.

2.4.2 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.

2.4.3 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

2.4.4 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}, ...}`

2.5 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.

2.6 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);
}
```

2.7 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.

2.8 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.