
LittlevGL Documentation

Release 6.0

Gabor Kiss-Vamosi

Jul 31, 2019

CONTENTS

1	Points forts	2
2	Éléments requis	3
3	FAQ	4
3.1	Où commencer ?	4
3.2	Où puis-je poser des questions ?	4
3.3	Est-ce que mon microcontrôleur/matériel est supporté ?	4
3.4	Mon écran est-il supporté?	5
3.5	LittlevGL est-il libre ? Comment puis-je l'utiliser dans un produit commercial ?	5
3.6	Rien ne se passe, mon pilote d'affichage n'est pas appelé. Qu'est-ce que j'ai raté ?	5
3.7	Pourquoi le pilote d'affichage n'est appelé qu'une seule fois ? Seule la partie supérieure de l'écran est actualisée.	5
3.8	Pourquoi je ne vois que des parasites à l'écran?	5
3.9	Pourquoi vois-je des couleurs incorrectes à l'écran ?	6
3.10	Comment accélérer mon interface utilisateur ?	6
3.11	Comment réduire l'utilisation de mémoire flash/MEM ?	6
3.12	Comment réduire l'utilisation de la MEV	7
3.13	Comment travailler avec un système d'exploitation ?	7
3.14	Comment contribuer à LittlevGL ?	7
3.15	Où puis-je trouver la documentation de la version précédente (5.3) ?	7

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

Version PDF: LittlevGL.pdf



LittlevGL est une bibliothèque graphique gratuite et à code source ouvert offrant tout ce dont vous avez besoin pour créer une interface graphique embarquée avec des éléments graphiques faciles à utiliser, de superbes effets visuels et une faible empreinte mémoire.

[Site Internet](#) · [GitHub](#) · [Forum](#) · [Démonstration en ligne](#) · [Simulateur](#) · [Blog](#)

POINTS FORTS

- Éléments de base évolués : boutons, graphiques, listes, curseurs, images, etc.
- Graphiques avancés avec animations, anti-crénelage, opacité, défilement doux
- Périphériques d'entrée variés : pavé tactile, souris, clavier, encodeur, etc.
- Prise en charge multilingue avec encodage UTF-8
- Prise en charge de plusieurs écrans, c-à-d utilisation simultanée d'un écran TFT et d'un écran monochrome
- Éléments graphiques entièrement personnalisables
- Indépendant du matériel : utilisable avec n'importe quel microcontrôleur ou écran
- Adaptable pour fonctionner avec peu de mémoire (64 ko de mémoire Flash, 16 ko de MEV)
- SE, mémoire externe et GPU pris en charge mais non requis
- Fonctionne avec un seul tampon d'affichage même avec des effets graphiques avancés
- Ecrit en C pour une compatibilité maximale (compatible C++)
- Simulateur pour débiter la conception d'interface graphique embarquée sans le matériel embarqué
- Tutoriels, exemples, thèmes pour une conception rapide
- Documentation en ligne et hors ligne
- Gratuit et à code source ouvert, sous licence MIT

ELÉMENTS REQUIS

- Microcontrôleur ou processeur 16, 32 ou 64 bits
- Une vitesse d'horloge $> \text{à } 16 \text{ MHz}$ est recommandée
- Taille de la mémoire Flash/MEM : $> \text{à } 64 \text{ ko}$ pour les composants essentiels (une taille $> \text{à } 180 \text{ ko}$ est recommandée)
- MEV :
 - Utilisation de MEV statique : $\sim 8..16 \text{ ko}$ en fonction des types d'objets et des fonctionnalités utilisés
 - Pile : $> \text{à } 2 \text{ ko}$ (une taille $> \text{à } 4 \text{ ko}$ est recommandée)
 - Données dynamiques (tas) : $> \text{à } 4 \text{ ko}$ (une taille $> \text{à } 16 \text{ ko}$ est recommandée si plusieurs objets sont utilisés). Défini par `LV_MEM_SIZE` dans `lv_conf.h`.
 - Tampon d'affichage : $> \text{à "résolution horizontale" pixels}$ (une taille $> \text{à } 10 \times \text{"résolution horizontale"}$ est recommandée)
- Compilateur conforme à C99 ou plus récent
- Connaissances de bases en C (ou C++) : [pointeurs](#), [structures](#), [fonctions de rappel](#).

Notez que l'utilisation de la mémoire peut varier en fonction de l'architecture, du compilateur et des options de compilation.

3.1 Où commencer ?

- Pour un aperçu général de LittlevGL, visitez littlevgl.com
- Accédez à la section *Démarrer* pour essayer des démonstrations en ligne dans votre navigateur, en savoir plus sur le simulateur et les bases de LittlevGL.
- Vous trouverez un guide de portage détaillé dans la section *Portage*.
- Pour savoir comment LittlevGL fonctionne, accédez à *Vue d'ensemble*.
- Pour lire des tutoriels ou partager vos propres expériences, accédez au *Blog*
- Pour découvrir le code source de la bibliothèque, consultez-le sur GitHub : <https://github.com/littlevgl/lvgl/>.

3.2 Où puis-je poser des questions ?

Pour poser des questions sur le forum : <https://forum.littlevgl.com/>.

Nous utilisons le suivi des problèmes de [GitHub](#) pour les discussions relatives au développement. Vous ne devez donc l'utiliser que si votre question ou votre problème est étroitement lié au développement de la bibliothèque.

3.3 Est-ce que mon microcontrôleur/matériel est supporté ?

Chaque microcontrôleur capable de piloter un affichage via un port parallèle, SPI, une interface RVB ou autre, et conforme aux *éléments requis*, est pris en charge par LittlevGL. Cela inclut

- Les microcontrôleurs “courants” tels que les STM32F, STM32H, NXP Kinetis, LPC, iMX, dsPIC33, PIC32, etc.
- Les modules Bluetooth, GSM, WiFi tels que les Nordic NRF et Espressif ESP32
- Le tampon de trame de Linux comme `/dev/fb0` ce qui inclut également les ordinateurs monocartes comme le Raspberry Pi
- Et tout ce qui possède un microcontrôleur suffisamment puissant et le nécessaire pour piloter un écran

3.4 Mon écran est-il supporté?

LittlevGL nécessite uniquement un simple pilote pour copier un tableau de pixels vers une zone donnée de l'affichage. Si vous pouvez le faire avec votre écran, vous pouvez utiliser cet écran avec LittlevGL. Cela inclut

- Les TFT avec une profondeur de couleur de 16 ou 24 bits
- Les moniteurs avec port HDMI
- Les petits écrans monochromes
- Les écrans à affichages en niveaux de gris
- Les matrices LED
- Ou tout autre affichage où vous pouvez contrôler la couleur/l'état des pixels

Consultez la section *Portage* pour en savoir plus.

3.5 LittlevGL est-il libre ? Comment puis-je l'utiliser dans un produit commercial ?

LittlevGL est fourni avec une licence MIT, ce qui signifie que vous pouvez le télécharger et l'utiliser à vos fins sans obligation.

3.6 Rien ne se passe, mon pilote d'affichage n'est pas appelé. Qu'est-ce que j'ai raté ?

Assurez-vous que vous appelez `lv_tick_inc(x)` dans une interruption et `lv_task_handler ()` dans votre boucle principale `while (1)`.

Apprenez-en plus dans les sections *Tic* et *Gestionnaire de tâche*.

3.7 Pourquoi le pilote d'affichage n'est appelé qu'une seule fois ? Seule la partie supérieure de l'écran est actualisée.

Assurez-vous que vous appelez `lv_disp_flush_ready(drv)` à la fin de votre *fonction de rappel du pilote d'affichage*.

3.8 Pourquoi je ne vois que des parasites à l'écran?

Il y a probablement un bogue dans votre pilote d'affichage. Essayez le code suivant sans utiliser 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++) {
```

(continues on next page)

(continued from previous page)

```

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;
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 Pourquoi vois-je des couleurs incorrectes à l'écran ?

Le format de couleur de LittlevGL n'est probablement pas compatible avec le format de couleur de votre écran. Vérifiez `LV_COLOR_DEPTH` dans `lv_conf.h`.

Si vous utilisez des couleurs 16 bits avec SPI (ou toute autre interface orientée octets), vous devez probablement définir `LV_COLOR_16_SWAP 1` dans `lv_conf.h`. Les octets supérieurs et inférieurs des pixels seront échangés.

3.10 Comment accélérer mon interface utilisateur ?

- Activez les optimisations du compilateur
- Augmentez la taille du tampon d'affichage
- Utilisez 2 tampons d'affichage et transférez le tampon en DMA (ou une technique similaire) en arrière-plan
- Augmentez la vitesse de fonctionnement des ports SPI ou parallèle si vous les utilisez pour piloter l'affichage
- Si votre écran dispose d'un port SPI, envisagez de passer à un modèle avec port parallèle, car son débit est beaucoup plus élevé.
- Conservez le tampon d'affichage dans la MEV interne (pas la SRAM externe) car LittlevGL l'utilise intensivement ce qui implique un temps d'accès minimal.

3.11 Comment réduire l'utilisation de mémoire flash/MEM ?

Vous pouvez désactiver toutes les fonctionnalités (animations, système de fichiers, GPU, etc.) et les types d'objet non utilisés dans `lv_conf.h`.

Si vous utilisez GCC, vous pouvez ajouter

- `-fdata-sections -ffunction-sections` aux options du compilateur
- `--gc-sections` aux options de l'éditeur de liens

pour supprimer les fonctions et variables inutilisées.

3.12 Comment réduire l'utilisation de la MEV

- Réduisez la taille du *tampon d'affichage*
- Réduisez `LV_MEM_SIZE` dans *lv_conf.h*. Cette mémoire est utilisée lorsque vous créez des objets tels que des boutons, des étiquettes, etc.
- Pour travailler avec un `LV_MEM_SIZE` réduit, vous pouvez créer les objets uniquement à l'utilisation et les supprimer lorsqu'ils ne sont plus nécessaires.

3.13 Comment travailler avec un système d'exploitation ?

Pour travailler avec un système d'exploitation où les tâches peuvent s'interrompre, vous devez protéger les appels de fonctions liés à LittlevGL avec un mutex. Consultez la section *Système d'exploitation* pour en savoir plus.

3.14 Comment contribuer à LittlevGL ?

Il y a plusieurs façons de contribuer à LittlevGL :

- Ecrivez quelques lignes sur votre projet pour inspirer les autres
- Répondez aux questions des autres
- Signalez et/ou corrigez des bogues
- Suggérez et/ou implémentez de nouvelles fonctionnalités
- Améliorez et/ou traduisez la documentation
- Ecrivez un article de blog sur vos expériences

Pour en savoir plus, consultez le [Guide de contribution](#)

3.15 Où puis-je trouver la documentation de la version précédente (5.3) ?

Vous pouvez la télécharger ici et l'ouvrir hors ligne :

`Docs-v5-3.zip`

3.15.1 Démarrer

Démonstrations en ligne

Vous pouvez découvrir à quoi ressemble LittlevGL sans installer ou télécharger quoi que ce soit. Il existe des interfaces utilisateurs prêtes à être essayées facilement dans votre navigateur.

Allez à la page [Démonstrations en ligne](#) et choisissez la démonstration qui vous intéresse.

Simulateur sur PC

Vous pouvez essayer LittlevGL **en utilisant uniquement votre PC** sans aucune carte de développement. Ecrivez du code, exécutez-le sur le PC et visualisez le résultat sur le moniteur. C'est multi-plateforme : Windows, Linux et OS X sont supportés. Le code écrit est portable, vous pouvez simplement le copier pour utilisation sur un matériel embarqué.

Le simulateur est également très utile pour rapporter des bogues car c'est une plateforme commune à chaque utilisateur. De ce fait c'est une bonne pratique de reproduire un bogue dans le simulateur et d'utiliser l'extrait de code dans le [forum](#).

Choisir un IDE

Le simulator est porté sur plusieurs IDEs. Choisissez votre IDE préféré, lisez son README sur GitHub, téléchargez le projet, et chargez le dans l'IDE.

Par la suite, le guide de configuration d'Eclipse CDT est décrit plus en détails.

Configurer Eclipse CDT

Installer Eclipse CDT

Eclipse CDT est un IDE C/C++. Vous pouvez utiliser un autre IDE mais ce tutoriel montre la configuration pour Eclipse CDT.

Eclipse est un logiciel écrit en Java de ce fait, soyez certain que l'**environnement d'exécution Java** est installé sur votre système.

Sur les distribution basée sur Debian (p.ex. Ubuntu) : `sudo apt-get install default-jre`

Vous pouvez télécharger Eclipse CDT à partir de : <https://eclipse.org/cdt/>. Démarrez l'installateur et choisissez *Eclipse CDT* dans la liste.

Installer SDL 2

Le simulateur PC utilise la librairie multi-plateforme [SDL 2](#) pour simuler un écran TFT et un pavé tactile.

Linux

Sur **Linux** vous pouvez installer facilement SDL 2 à partir d'un terminal :

1. Trouvez la version actuelle de SDL 2 : `apt-cache search libsdl2` (e.g. `libsdl2-2.0-0`)
2. Installez SDL 2 : `sudo apt-get install libsdl2-2.0-0` (remplacez par la version trouvée)
3. Installez le paquet de développement de SDL 2 : `sudo apt-get install libsdl2-dev`
4. Si les paquets de construction essentiels ne sont pas déjà installés : `sudo apt-get install build-essential`

Windows

Si vous utilisez **Windows** vous devez en premier lieu installer MinGW ([version 64 bits](#)). Après ça, effectuez les étapes suivantes pour ajouter SDL 2 :

1. Téléchargez les libraries de développement de SDL. Allez sur <https://www.libsdl.org/download-2.0.php> et téléchargez *Development Libraries: SDL2-devel-2.0.5-mingw.tar.gz*
2. Décompressez l'archive et allez dans le répertoire *x86_64-w64-mingw32* (pour MinGW 64 bits) ou *i686-w64-mingw32* (pour MinGW 32 bits)
3. Copiez le répertoire *__...mingw32/include/SDL2* vers *C:/MinGW/.../x86_64-w64-mingw32/include*
4. Copiez le contenu de *__...mingw32/lib/* dans *C:/MinGW/.../x86_64-w64-mingw32/lib*
5. Copiez *__...mingw32/bin/SDL2.dll* dans *{eclipse_workspace}/pc_simulator/Debug/*. Faites le plus tard quand Eclipse est installé.

Note : si vous utilisez **Microsoft Visual Studio** à la place d'Eclipse alors vous n'avez pas besoin d'installer MinGW.

OS X

Sur **OS X** vous pouvez facilement installer SDL 2 avec brew : `brew install sdl2`

SI quelque chose ne fonctionne pas je suggère ce [tutoriel](#) pour débiter avec SDL.

Projet pré-configuré

Un projet pré-configuré pour la librairie graphique, basé sur la dernière version publiée, est toujours disponible. Vous pouvez le trouver sur [GitHub](#) ou sur la page de [Téléchargement](#). Le projet est configuré pour Eclipse CDT.

Ajouter le projet pré-configuré à Eclipse CDT.

Lancez Eclipse CDT. Une boîte de dialogue au sujet du **chemin de l'espace de travail** est affichée. Avant de la valider, vérifiez le chemin et copiez à cet emplacement, puis décompressez, le projet pré-configuré préalablement téléchargé. Maintenant vous pouvez accepter le chemin de l'espace de travail. Bien entendu, ce chemin peut être modifié mais dans ce cas il faut copier le projet vers cet emplacement.

Fermez la fenêtre de démarrage et allez à **Fichier->Importer...** et choisissez **Généralités->Projets existants dans l'espace de travail**. Allez au répertoire racine du projet et cliquez **Terminer**

Sur **Windows** vous devez effectuer deux actions additionnelles :

- Copiez le fichier **SDL2.dll** dans le répertoire Debug du projet
- Faites un clic droit sur le projet -> Propriétés -> Génération C/C++ -> Paramètres -> Bibliothèques -> Ajouter... et ajoutez *mingw32* au-dessus de *SDLmain* et *SDL*. L'ordre est important : *mingw32*, *SDLmain*, *SDL*

Compilation et exécution

Vous êtes maintenant prêt à utiliser la librairie graphique LittlevGL sur votre PC. Cliquer sur l'icône Marteau de la barre de menu pour compiler le projet. Si vous avez tout fait correctement aucune erreur ne se produira.

Notez que sur certains systèmes des étapes additionnelles peuvent être requises pour qu'Eclipse prenne en compte SDL 2, mais dans la plupart des cas, la configuration du projet téléchargé est suffisante.

Après avoir compiler avec succès, cliquez sur le bouton Jouer de la barre de menu pour démarrer le projet. Maintenant une fenêtre doit apparaître au milieu de l'écran.

Tout est prêt pour utiliser la librairie graphique LittlevGL pour l'apprentissage ou pour débiter le développement sur votre PC.

Aperçu rapide

Ici, vous pouvez apprendre les points les plus importants sur LittlevGL. Vous devriez le lire en premier pour avoir une impression générale, puis les sections détaillées *Portage* et *Vue d'ensemble* après cela.

Ajouter LittlevGL à votre projet

Les étapes suivantes montrent comment configurer LittlevGL sur un système embarqué avec un écran et un pavé tactile. Vous pouvez utiliser le *Simulateur* pour vous préparer à utiliser des projets pouvant être exécutés sur votre PC.

- Téléchargez ou clonez la librairie
- Copiez le répertoire `lvgl` dans votre projet
- Copiez `lvgl/lv_conf_templ.h` sous le nom `lv_conf.h` au même niveau que le répertoire `lvgl` et définissez au minimum `LV_HOR_RES_MAX`, `LV_VER_RES_MAX` et `LV_COLOR_DEPTH`.
- Incluez `lvgl/lvgl.h` quand vous devez utiliser les fonctions de LittlevGL.
- Appelez `lv_tick_inc(x)` chaque `x` millisecondes à partir d'une horloge ou d'une tâche (`x` doit être compris entre 1 et 10). Ceci est requis pour la synchronisation interne de LittlevGL.
- Appelez `lv_init()`
- Créez un tampon d'affichage pour LittlevGL

```
static lv_disp_buf_t disp_buf;
static lv_color_t buf[LV_HOR_RES_MAX * 10];           /* Déclare un tampon
↳pour 10 lignes */
lv_disp_buf_init(&disp_buf, buf, NULL, LV_HOR_RES_MAX * 10); /* Initialise le
↳tampon d'affichage */
```

- Implémentez et enregistrez une fonction qui copie un tableau de pixels vers une zone de l'écran :

```
lv_disp_drv_t disp_drv;           /* Descripteur du pilote d'affichage */
lv_disp_drv_init(&disp_drv);       /* Initialisation de base */
disp_drv.flush_cb = my_disp_flush; /* Définit la fonction du pilote */
disp_drv.buffer = &disp_buf;      /* Définit le tampon d'affichage */
lv_disp_drv_register(&disp_drv);   /* Finalement, enregistre le pilote */

void my_disp_flush(lv_disp_t * disp, const lv_area_t * area, lv_color_t * color_p)
{
    int32_t x, y;
    for(y = area->y1; y <= area->y2; y++) {
        for(x = area->x1; x <= area->x2; x++) {
            set_pixel(x, y, *color_p); /* Définit la couleur d'un pixel de l'écran */
            color_p++;
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    lv_disp_flush_ready(disp);          /* Indique que les données peuvent être
    ↪ transférées à l'écran */
}

```

- Implémentez et enregistrez une fonction qui peut lire un périphérique d'entrée. P.ex. pour un pavé tactile :

```

lv_indev_drv_init(&indev_drv);          /* Descripteur du pilote du périphérique d
    ↪ entrée */
indev_drv.type = LV_INDEV_TYPE_POINTER; /* Le pavé tactile est un périphérique
    ↪ semblable à un pointeur */
indev_drv.read_cb = my_touchpad_read;   /* Définit la fonction du pilote */
lv_indev_drv_register(&indev_drv);      /* Finalement, enregistre le pilote */

bool my_touchpad_read(lv_indev_t * indev, lv_indev_data_t * data)
{
    static lv_coord_t last_x = 0;
    static lv_coord_t last_y = 0;

    /* Mémorise l'état et les coordonnées, si pressé */
    data->state = touchpad_is_pressed() ? LV_INDEV_STATE_PR : LV_INDEV_STATE_REL;
    if(data->state == LV_INDEV_STATE_PR) touchpad_get_xy(&last_x, &last_y);

    /* Définit les coordonnées (si relâché, les dernières coordonnées quand pressé) */
    data->point.x = last_x;
    data->point.y = last_y;

    return false; /* Retourne `faux` car pas de tampon et plus de données à lire */
}

```

- Appelez `lv_task_handler()` périodiquement, chaque quelques millisecondes, dans la boucle principale `while(1)`, sur interruption d'une horloge ou à partir d'une tâche du système d'exploitation. Cela redessine l'écran si nécessaire, gère les périphériques d'entrée, etc.

Apprendre les bases

Les objets (éléments visuels)

Les éléments graphiques tels que les boutons, les étiquettes, les curseurs, les graphiques, etc. sont appelés des objets dans LittlevGL. Allez à [Types d'objet] (/object-types/index) pour voir la liste complète des types disponibles.

Chaque objet possède un objet parent. L'objet enfant se déplace avec le parent et si vous supprimez le parent, les enfants seront également supprimés. Les enfants ne peuvent être visibles que sur leurs parents.

Les écrans sont les parents "racine". Pour obtenir l'écran actuel, appelez `lv_scr_act()`.

Vous pouvez créer un nouvel objet avec `lv_<type>_create(parent, obj_to_copy)`. Une variable `lv_obj_t *` est retournée qui doit être utilisée comme référence à l'objet pour définir ses paramètres. Le premier paramètre est le *parent* souhaité, le second paramètre peut être un objet à copier (NULL si inutilisé). Par exemple :

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

Pour définir certains attributs de base les fonctions `lv_obj_set_<parameter_name>(obj, <value>)` peuvent être utilisées. Par exemple :

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

Les objets ont également des paramètres spécifiques au type qui peuvent être définis par les fonctions `lv_<type>_set_<parameter_name>(obj, <valeur>)`. Par exemple :

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

Pour voir l'API complète, consultez la documentation des types d'objet ou le fichier d'en-tête associé (p.ex. `lvgl/src/lv_objx/lv_slider.h`).

Styles

Les styles peuvent être affectés aux objets pour changer leur apparence. Un style décrit tout à la fois l'apparence des objets de type rectangle (comme un bouton ou un curseur), des textes, des images et des lignes.

Voici comment créer un nouveau style :

```
static lv_style_t style1; /* Déclare un nouveau style. Devrait être ↵
↳ `static` */
lv_style_copy(&style1, &lv_style_plain); /* Copie un style intégré */
style1.body.main_color = LV_COLOR_RED; /* Couleur principale */
style1.body.grad_color = lv_color_hex(0xffd83c) /* Dégradé de couleur (orange) */
style1.body.radius = 3;
style1.text.color = lv_color_hex3(0x0F0) /* Couleur de texte (vert) */
style1.text.font = &lv_font_dejavu_22; /* Change la police */
...
```

Pour appliquer un nouveau style à un objet, utilisez les fonctions `lv_<type>set_style(obj, LV_<TYPE>_STYLE_<NOM>, &my_style)`. Par exemple :

```
lv_slider_set_style(slider1, LV_SLIDER_STYLE_BG, &slider_bg_style);
lv_slider_set_style(slider1, LV_SLIDER_STYLE_INDIC, &slider_indic_style);
lv_slider_set_style(slider1, LV_SLIDER_STYLE_KNOB, &slider_knob_style);
```

Si le style d'un objet est **NULL**, il héritera du style de son parent. Par exemple, le style des étiquettes est **NULL** par défaut. Si vous les placez sur un bouton, elles utiliseront les propriétés `style.text` du style du bouton.

Apprenez-en plus dans la section [Styles](#).

Événements

Les événements sont utilisés pour informer l'utilisateur si quelque chose s'est passé avec un objet. Vous pouvez affecter une fonction de rappel à un objet qui sera appelée si l'objet est cliqué, relâché, déplacé, en cours de suppression, etc. Voici à quoi cela ressemble :

```
lv_obj_set_event_cb(btn, btn_event_cb);           /* Affecte une fonction de
↳rappel au bouton */

...

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

Apprenez-en plus dans la section *Evénements*.

Exemples

Bouton avec étiquette

```
lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL); /* Ajoute un bouton à l'écran
↳actuel */
lv_obj_set_pos(btn, 10, 10);                       /* Définit sa position */
lv_obj_set_size(btn, 100, 50);                     /* Définit sa taille */
lv_obj_set_event_cb(btn, btn_event_cb);             /* Affecte une fonction de
↳rappel au bouton */

lv_obj_t * label = lv_label_create(btn, NULL);       /* Ajoute une étiquette au
↳bouton */
lv_label_set_text(label, "Button");                 /* Définit le texte de l
↳étiquette */

...

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



Bouton avec styles

Ajoutez des styles au bouton de l'exemple précédent :

```
static lv_style_t style_btn_rel;                   /* Une variable pour
↳enregistrer le style relâché */
lv_style_copy(&style_btn_rel, &lv_style_plain);    /* Initialise à partir d'un
↳style intégré */
```

(continues on next page)

(continued from previous page)

```

style_btn_rel.body.border.color = lv_color_hex3(0x269);
style_btn_rel.body.border.width = 1;
style_btn_rel.body.main_color = lv_color_hex3(0xADF);
style_btn_rel.body.grad_color = lv_color_hex3(0x46B);
style_btn_rel.body.shadow.width = 4;
style_btn_rel.body.shadow.type = LV_SHADOW_BOTTOM;
style_btn_rel.body.radius = LV_RADIUS_CIRCLE;
style_btn_rel.text.color = lv_color_hex3(0xDEF);

static lv_style_t style_btn_pr;                                /* Une variable pour ↵
↪enregistrer le style pressé */
lv_style_copy(&style_btn_pr, &style_btn_rel);                  /* Initialise à partir du ↵
↪style relâché */
style_btn_pr.body.border.color = lv_color_hex3(0x46B);
style_btn_pr.body.main_color = lv_color_hex3(0x8BD);
style_btn_pr.body.grad_color = lv_color_hex3(0x24A);
style_btn_pr.body.shadow.width = 2;
style_btn_pr.text.color = lv_color_hex3(0xBCD);

lv_btn_set_style(btn, LV_BTN_STYLE_REL, &style_btn_rel);      /* Définit le style ↵
↪relâché du bouton */
lv_btn_set_style(btn, LV_BTN_STYLE_PR, &style_btn_pr);         /* Définit le style ↵
↪pressé du bouton */

```



Curseur et alignement de l'objet

```

lv_obj_t * label;

...

/* Crée un curseur au centre de l'affichage */
lv_obj_t * slider = lv_slider_create(lv_scr_act(), NULL);
lv_obj_set_width(slider, 200);                                /* Définit la largeur */
lv_obj_align(slider, NULL, LV_ALIGN_CENTER, 0, 0);             /* Aligne au centre du parent ↵
↪(écran) */
lv_obj_set_event_cb(slider, slider_event_cb);                  /* Affecte une fonction de ↵
↪rappel */

/* Crée une étiquette sous le curseur */
label = lv_label_create(lv_scr_act(), NULL);
lv_label_set_text(label, "0");
lv_obj_set_auto_realign(slider, true);
lv_obj_align(label, slider, LV_ALIGN_OUT_BOTTOM_MID, 0, 10);

...

void slider_event_cb(lv_obj_t * slider, lv_event_t event)
{

```

(continues on next page)

(continued from previous page)

```

    if(event == LV_EVENT_VALUE_CHANGED) {
        static char buf[4]; /* Maximum 3 octets pour
↪ le nombre plus 1 octet final nul */
        snprintf(buf, 4, "%u", lv_slider_get_value(slider));
        lv_label_set_text(slider_label, buf); /* Actualise le texte */
    }
}

```



76

Liste et thèmes

```

/* Textes des éléments de la liste */
const char * txts[] = {"First", "Second", "Third", "Fourth", "Fifth", "Sixth", NULL};

/* Initialise et définit un thème. `LV_THEME_NIGHT` doit être activé dans lv_conf.h.
↪ */
lv_theme_t * th = lv_theme_night_init(20, NULL);
lv_theme_set_current(th);

/* Crée une liste */
lv_obj_t * list = lv_list_create(lv_scr_act(), NULL);
lv_obj_set_size(list, 120, 180);
lv_obj_set_pos(list, 10, 10);

/* Ajoute des boutons */
uint8_t i;
for(i = 0; txts[i]; i++) {
    lv_obj_t * btn = lv_list_add_btn(list, LV_SYMBOL_FILE, txts[i]);
    lv_obj_set_event_cb(btn, list_event); /* Affecte une fonction de rappel */
    lv_btn_set_toggle(btn, true); /* Active la fonction de bascule */
}

/* Initialise et définit un autre thème. `LV_THEME_MATERIAL` doit être activé dans lv_
↪ conf.h.
* Si `LV_THEME_LIVE_UPDATE 1` alors le style de la liste précédente sera également
↪ mis à jour. */
th = lv_theme_material_init(210, NULL);
lv_theme_set_current(th);

/* Crée une autre liste */
list = lv_list_create(lv_scr_act(), NULL);
lv_obj_set_size(list, 120, 180);
lv_obj_set_pos(list, 150, 10);

/* Ajoute des boutons avec les mêmes textes */
for(i = 0; txts[i]; i++) {
    lv_obj_t * btn = lv_list_add_btn(list, LV_SYMBOL_FILE, txts[i]);
    lv_obj_set_event_cb(btn, list_event);
}

```

(continues on next page)

(continued from previous page)

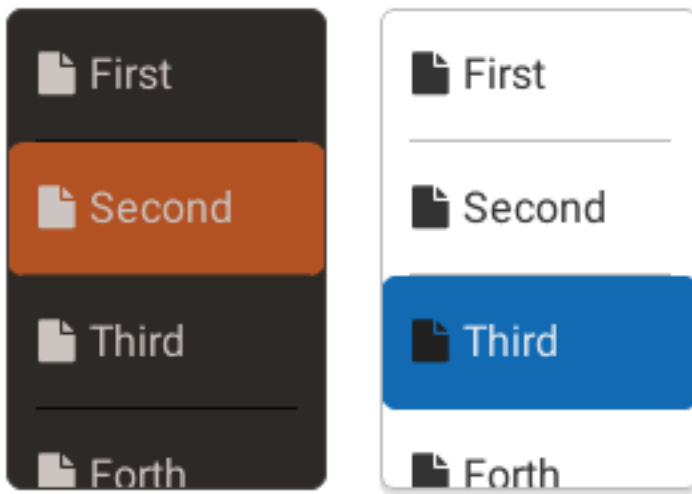
```

    lv_btn_set_toggle(btn, true);
}

...

static void list_event(lv_obj_t * btn, lv_event_t e)
{
    if(e == LV_EVENT_CLICKED) {
        printf("%s\n", lv_list_get_btn_text(btn));
    }
}

```



Utiliser LittlevGL avec Micropython

Apprenez-en plus sur *Micropython*.

```

# Crée un bouton et une étiquette
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")

# Charge l'écran
lv.scr_load(scr)

```

Contribuer

Pour poser des questions s'il vous plaît utilisez le [forum](#). Pour les tâches liées au développement (rapports de bogues, suggestions de fonctionnalités), utilisez le suivi des problèmes de [GitHub].(<https://github.com/littlevgl/lvgl/issues>).

Si vous souhaitez contribuer à LittlevGL, vous pouvez

- Aidez les autres sur le [forum](#).

- **Inspirez les gens** en parlant de votre projet dans la catégorie [Mon projet](#) du forum ou en l'ajoutant à la rubrique [Références](https://blog.littlevgl.com/2018-12-26/references)
- **Améliorez et/ou traduisez la documentation.** Visitez le dépôt [Documentation](#) pour en apprendre plus **Écrivez un article de blog** sur vos expériences. Regardez comment faire dans le dépôt [Blog](https://github.com/littlevgl/blog).
- **Signalez et/ou corrigez des bogues** avec l'outil de suivi des problèmes de [GitHub](#)
- **Aidez au développement.** Vérifiez les [problèmes en cours](#), en particulier ceux avec la mention [Aide demandée](#) et partagez vos idées sur un sujet ou implémentez une fonctionnalité.

Il devrait être utile de lire le

Micropython

Qu'est-ce que Micropython ?

[Micropython](#) est une version de Python destinées aux microcontrôleurs. Avec Micropython vous pouvez écrire du code Python 3 et l'exécuter directement sur des architectures aux ressources limitées.

Points forts de Micropython

- **Compact** - s'exécute dans seulement 256 ko d'espace de code et 16 ko de MEV. Aucun SE n'est nécessaire, bien qu'il soit possible de l'exécuter sur un SE, si vous le souhaitez.
- **Compatible** - s'efforce d'être aussi compatible que possible avec le Python de référence (CPython)
- **Adaptable** - supporte de multiples architectures (x86, x86-64, ARM, ARM Thumb, Xtensa)
- **Interactif** - le cycle compilation-programmation-démarrage n'est pas nécessaire. Avec REPL (l'invite interactive) vous pouvez entrer des commandes et les exécuter immédiatement, lancer des scripts etc.
- **Populaire** - de nombreuses plateformes sont supportées. Le nombre d'utilisateurs est en constante augmentation. Variantes notables : [MicroPython](#), [CircuitPython](#), [MicroPython_ESP32_psRAM_LoBo](#)
- **Orienté embarqué** - fourni avec des modules spécifiques aux systèmes embarqués, comme le module [machine](#) pour accéder au matériel bas-niveau (broches d'E/S, CAN, UART, SPI, I2C, RTC, horloges etc.)

Pourquoi Micropython + LittlevGL ?

Aujourd'hui Micropython ne dispose pas d'une bonne librairie de haut-niveau pour réaliser des interfaces graphiques. LittlevGL est une bonne librairie de haut-niveau, implémentée en C dont l'API est en C. LittlevGL est une librairie basée sur l'utilisation de [composants orientés objet](#), ce qui en fait une candidate idéale pour s'interfacer à un langage de plus haut-niveau tel que Python.

Voici quelques avantages à utiliser LittlevGL avec Micropython :

- Développez des interfaces graphiques en Python, langage de haut-niveau très populaire. Utilisez des paradigmes tels que la programmation orientée objet.

- Le développement d'interfaces graphiques nécessite de multiples itérations pour obtenir un bon résultat. Avec C, chaque itération nécessite de **modifier le code** > **compiler** > **programmer** > **exécuter**. En Micropython il faut seulement **modifier le code** > **exécuter**. Vous pouvez même exécuter des commandes de manière interactive en utilisant [REPL](#) (l'invite interactive)

Micropython + LittlevGL peuvent être utilisés pour :

- Le prototypage rapide d'interface graphique.
- Réduire le cycle de modification et d'optimisation de l'interface graphique.
- Modéliser l'interface graphique d'une manière plus abstraite en définissant des objets composites réutilisables, en tirant avantage des fonctionnalités du langage Python telles que l'héritage, les clôtures, les listes en compréhension, les générateurs, la gestion d'exception, les entiers multiprécision et autres.
- Rendre LittlevGL accessible à une plus large audience. Aucun besoin de connaître le C dans le but de créer une interface graphique fonctionnelle sur un système embarqué. C'est également vrai pour [CircuitPython vision](#). CircuitPython a été conçu avec l'éducation à l'esprit, pour rendre plus facile à des utilisateurs, nouveaux ou inexpérimentés, de débiter avec le développement embarqué.

Alors, à quoi ça ressemble ?

TL;DR : C'est très similaire à l'API C, mais orienté objet pour les composants de LittlevGL.

Plongeons droit dans un exemple !

Un exemple simple

```
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)
```

Comment l'utiliser?

Simulateur en ligne

Si vous souhaitez expérimenter LittlevGL + Micropython sans télécharger quoi que ce soit - vous pouvez utiliser notre simulateur en ligne ! C'est un ensemble LittlevGL + Micropython entièrement fonctionnel qui s'exécute dans le navigateur et permet d'éditer et d'exécuter un script Python.

[Lien vers le simulateur en ligne](#)

Simulateur PC

Micropython est porté sur plusieurs environnements, dont Unix, ce qui permet de compiler et exécuter Micropython (+ LittlevGL) sur une machine Linux (sur une machine Windows, d'autres outils peuvent être nécessaires : VirtualBox ou WSL ou MinGW ou Cygwin etc.).

Plus d'informations pour compiler et exécuter le port Unix

La réalité

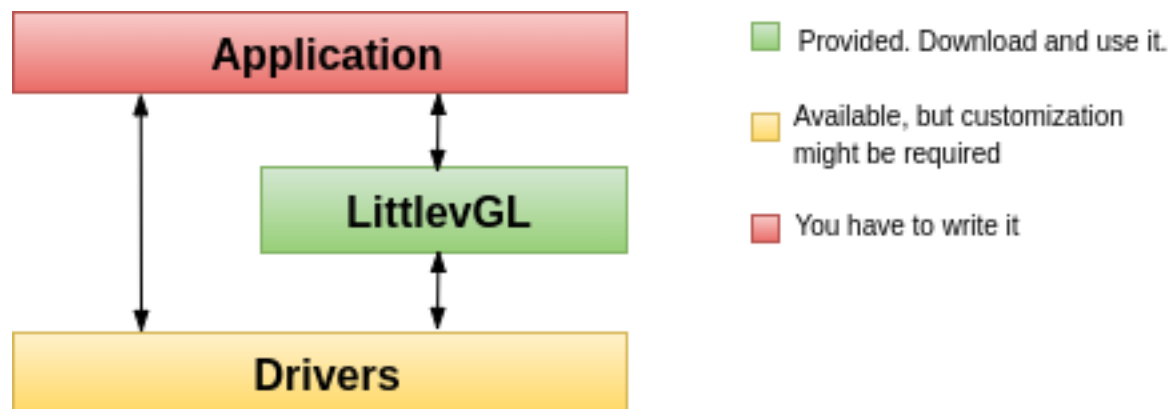
Au final, le but est d'exécuter sur une plateforme embarquée. Micropython et LittlevGL peuvent être utilisés sur de nombreuses architectures embarquées, telles que STM32, ESP32 etc. Vous aurez également besoin de pilotes d'affichage et d'entrée. Nous avons quelques exemples de pilotes (ESP32 + ILI9341, ainsi que d'autres exemples), mais il est fort probable que vous souhaitiez créer vos propres pilotes d'affichage et d'entrée pour vos besoins spécifiques. Les pilotes peuvent être implémentés soit en C en tant que module Micropython, soit en Micropython pur !

Où trouver plus d'informations ?

- Dans le sujet du [Blog](#)
- Dans le [README lv_micropython](#)
- Dans le [README lv_binding_micropython](#)
- Sur le forum LittlevGL (n'hésitez pas à demander quoi que ce soit !)
- Dans la [documentation](#) et sur le [forum](#) Micropython

3.15.2 Portage

Aperçu système



Application Votre application qui crée l'interface graphique et gère les tâches spécifiques.

LittlevGL La bibliothèque graphique elle-même. Votre application peut communiquer avec la bibliothèque pour créer une interface graphique. Elle contient une interface HAL (Hardware Abstraction Layer, couche d'abstraction matérielle) permettant d'enregistrer vos pilotes de périphérique d'affichage et d'entrée.

Pilote Outre vos pilotes spécifiques, il contient des fonctions pour gérer l'écran, éventuellement un GPU (processeur graphique), et lire un pavé tactile ou des boutons.

Il existe ** deux configurations matérielles typiques **, selon que le microcontrôleur dispose ou non d'un contrôleur LCD/TFT. Dans les deux cas, un tampon d'affichage sera nécessaire pour mémoriser l'image actuelle de l'écran.

1. **Microcontrôleur avec contrôleur TFT/LCD** Si votre microcontrôleur dispose d'un contrôleur TFT/LCD, vous pouvez connecter un écran directement via une interface RVB. Dans ce cas, le tampon d'affichage peut résider dans la MEV interne (si le microcontrôleur dispose de suffisamment de MEV) ou dans la MEV externe (si la microcontrôleur a une interface mémoire).
2. **Contrôleur d'affichage externe** Si votre microcontrôleur ne dispose pas d'un contrôleur TFT/LCD alors un contrôleur d'affichage externe (ex. SSD1963, SSD1306, ILI9341) doit être utilisé. Dans ce cas, le microcontrôleur peut communiquer avec le contrôleur d'affichage via un port parallèle, SPI ou parfois I2C. Le tampon d'affichage est généralement situé dans le contrôleur d'affichage, ce qui économise beaucoup de MEV pour le microcontrôleur.

Configurer un projet

Obtenir la librairie

LittlevGL Graphics Library est disponible sur GitHub : <https://github.com/littlevgl/lvgl>.

Vous pouvez la cloner ou télécharger la dernière version de la librairie depuis GitHub ou vous pouvez également utiliser la page [Télécharger] (<https://littlevgl.com/download>).

La librairie graphique est le répertoire **lvgl** qui doit être copié dans votre projet.

Fichier de configuration

Il existe un fichier d'en-tête de configuration pour LittlevGL appelé **lv_conf.h**. Il définit le comportement de base de la bibliothèque, désactive les modules et fonctionnalités inutilisés, ajuste la taille des tampons au moment de la compilation, etc.

Copiez **lvgl/lv_conf_template.h** au même niveau que le répertoire *lvgl* et renommez-le en *lv_conf.h*. Ouvrez le fichier et changez le **#if 0** du début en **#if 1** pour activer son contenu.

- *lv_conf.h* * peut également être copié à d'autres emplacements, mais vous devez ensuite ajouter **LV_CONF_INCLUDE_SIMPLE** à vos options de compilation (p.ex. **-DLV_CONF_INCLUDE_SIMPLE** pour gcc) et définir le chemin d'inclusion manuellement.

Dans le fichier de configuration, les commentaires expliquent la signification des options. Vérifiez au moins ces trois options de configuration et modifiez-les en fonction de votre matériel :

1. **LV_HOR_RES_MAX** la résolution horizontale de votre écran
2. **LV_VER_RES_MAX** la résolution verticale de votre écran
3. **LV_COLOR_DEPTH** 8 pour (RG32), 16 pour (RGB565) ou 32 pour (RGB888 et ARGB8888).

Initialisation

Pour utiliser la librairie graphique, vous devez l'initialiser ainsi que les autres composants. Voici l'ordre d'initialisation :

1. Appelez *lv_init()*

2. Initialisez vos pilotes
3. Enregistrez les pilotes de périphérique d'affichage et d'entrée dans LittlevGL. En savoir plus sur l'enregistrement : [Affichage](#) et [Périphérique d'entrée](#).
4. Appelez `lv_tick_inc(x)` toutes les `x` millisecondes dans une interruption pour indiquer le temps écoulé. [En savoir plus](#).
5. Appelez `lv_task_handler()` périodiquement toutes les quelques millisecondes pour gérer les tâches liées à LittlevGL. [En savoir plus](#).

Interface d'affichage

Pour configurer un affichage, les variables `lv_disp_buf_t` et `lv_disp_drv_t` doivent être initialisées.

- `lv_disp_buf_t` contient le(s) tampon(s) graphique(s) interne(s).
- `** lv_disp_drv_t **` contient les fonctions de rappel pour interagir avec l'affichage et manipuler des éléments liés au dessin.

Tampon d'affichage

`lv_disp_buf_t` peut être initialisé comme ceci :

```
/* Une variable statique ou globale pour mémoriser les tampons */
static lv_disp_buf_t disp_buf;

/* Tampon(s) statique(s) ou global(aux). Le second tampon est optionnel */
static lv_color_t buf_1[MY_DISP_HOR_RES * 10];
static lv_color_t buf_2[MY_DISP_HOR_RES * 10];

/* Initialise `disp_buf` avec le(s) tampon(s) */
lv_disp_buf_init(&disp_buf, buf_1, buf_2, MY_DISP_HOR_RES*10);
```

Voici les configurations possibles concernant la taille de la mémoire tampon :

1. **Un tampon** LittlevGL dessine le contenu de l'écran dans un tampon et l'envoie à l'écran. Le tampon peut être plus petit que l'écran. Dans ce cas, les zones les plus grandes seront redessinées en plusieurs parties. Si seules de petites zones changent (p.ex. appui sur un bouton), seules ces zones seront actualisées.
2. **Deux tampons de taille différente de l'écran** ayant deux tampons LittlevGL peut dessiner dans un tampon tandis que le contenu de l'autre tampon est envoyé à l'écran en arrière-plan. Le DMA ou une autre méthode doit être utilisé pour transférer les données à l'écran afin de permettre au CPU de dessiner dans le même temps. De cette manière, le rendu et le rafraîchissement de l'affichage deviennent parallèles. De même que *Un tampon*, LittlevGL dessine le contenu de l'affichage en fragments si le tampon est plus petit que la zone à actualiser.
3. **Deux tampons de la taille d'un écran.** Contrairement à *Deux tampons de taille différente de l'écran* LittlevGL fournira toujours tout le contenu de l'affichage, pas seulement des fragments. De cette façon, le pilote peut simplement changer l'adresse du tampon d'affichage par celle du tampon préparé par LittlevGL. Par conséquent, cette méthode est la meilleure lorsque le microcontrôleur dispose d'une interface LCD/TFT et que le tampon d'affichage est un emplacement dans la MEV.

Pilote d'affichage

Une fois l'initialisation des tampons effectuée, les pilotes d'affichage doivent être initialisés. Dans le cas le plus simple, seuls les deux champs suivants de `lv_disp_drv_t` doivent être définis :

- **buffer** pointeur sur une variable `lv_disp_buf_t` initialisée.
- **flush_cb** une fonction de rappel permettant de copier le contenu d'un tampon dans une zone spécifique de l'écran.

Il y a quelques champs de données optionnels :

- **hor_res** résolution horizontale de l'écran. (`LV_HOR_RES_MAX` par défaut à partir de `lv_conf.h`)
- **ver_res** résolution verticale de l'écran. (`LV_VER_RES_MAX` par défaut à partir de `lv_conf.h`)
- **color_chroma_key** une couleur qui sera dessinée de manière transparente sur les images incrustées. `LV_COLOR_TRANSP` par défaut à partir de `lv_conf.h`)
- **** user_data **** donnée utilisateur personnalisée pour le pilote. Son type peut être modifié dans `lv_conf.h`.
- **** anti-aliasing **** utilise un anti-crênelage (lissage des bords). `LV_ANTIALIAS` par défaut à partir de `lv_conf.h`
- **rotated** si `1` permute **hor_res** et **ver_res**. LittlevGL dessine dans le même sens dans les deux cas (lignes du haut vers le bas); le pilote doit donc également être reconfiguré pour modifier le sens de remplissage de l'écran.
- **screen_transp** si `1` l'écran peut avoir un style transparent ou opaque. `LV_COLOR_SCREEN_TRANSP` doit être activé dans `lv_conf.h`

Pour utiliser un GPU, les fonctions de rappel suivantes peuvent être utilisées :

- **gpu_fill_cb** remplit une zone en mémoire avec une couleur
- **gpu_blend_cb** combine deux tampons en mémoire avec gestion de l'opacité.

Notez que ces fonctions doivent dessiner en mémoire (MEV) et non directement sur l'affichage.

Certaines autres fonctions de rappel facultatives facilitent et optimisent l'utilisation des écrans monochromes, à niveaux de gris ou autres écrans RVB non standard :

- **rounder_cb** arrondit les coordonnées des zones à redessiner. P.ex. une zone de 2 x 2 px peut être convertie en 2 x 8 px. Utile si la carte graphique ne peut actualiser que les zones ayant une hauteur ou une largeur spécifique (généralement une hauteur de 8 px avec des écrans monochromes).
- **set_px_cb** une fonction personnalisée pour écrire le *tampon d'affichage*. Utile pour enregistrer les pixels de manière plus compacte si l'affichage présente un format de couleur spécial. (par exemple monochrome 1 bit, échelle de gris 2 bit, etc.) De cette façon, les tampons utilisés dans `lv_disp_buf_t` peuvent être plus petits pour ne contenir que le nombre de bits requis pour la taille de zone donnée.
- **monitor_cb** indique combien de pixels ont été actualisés et en combien de temps.

Pour définir les champs de la variable `lv_disp_drv_t`, celle-ci doit être initialisée avec `lv_disp_drv_init(&disp_drv)`. Et enfin, pour enregistrer un affichage pour LittlevGL, `lv_disp_drv_register(&disp_drv)` doit être appelée.

Dans l'ensemble, cela ressemble à ceci :

```
lv_disp_drv_t disp_drv;           /* Une variable pour contenir les pilotes.
→ Peut être une variable locale */
lv_disp_drv_init(&disp_drv);      /* Initialisation de base */
```

(continues on next page)

(continued from previous page)

```

disp_drv.buffer = &disp_buf;           /* Affecte un tampon initialisé */
disp_drv.flush_cb = my_flush_cb;       /* Définit une fonction de rappel pour
↳ dessiner à l'écran */
lv_disp_t * disp;
disp = lv_disp_drv_register(&disp_drv); /* Enregistre le pilote et sauvegarde les
↳ objets d'affichage créés */

```

Voici quelques exemples simples de fonctions de rappel :

```

void my_flush_cb(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t * color_
↳ p)
{
    /* Le cas le plus simple (mais aussi le plus lent) pour mettre tous les pixels à l
↳ 'écran un par un */
    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 !!!
    * Informe la librairie graphique que vous êtes prêt pour le transfert */
    lv_disp_flush_ready(disp);
}

void my_gpu_fill_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest_buf, const lv_area_t
↳ * dest_area, const lv_area_t * fill_area, lv_color_t color);
{
    /* Cet exemple de code devrait être effectué par un GPU */
    uint32_t x, y;
    dest_buf += dest_width * fill_area->y1; /* Aller à la première ligne */

    for(y = fill_area->y1; y < fill_area->y2; y++) {
        for(x = fill_area->x1; x < fill_area->x2; x++) {
            dest_buf[x] = color;
        }
        dest_buf+=dest_width; /* Aller à la ligne suivante */
    }
}

void my_gpu_blend_cb(lv_disp_drv_t * disp_drv, lv_color_t * dest, const lv_color_t *
↳ src, uint32_t length, lv_opa_t opa)
{
    /* Cet exemple de code devrait être effectué par un 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)
{
    /* Modifie les zones en fonction des besoins. Agrandir uniquement.
    * Par exemple, pour toujours avoir des lignes de 8 px de hauteur : */
}

```

(continues on next page)

(continued from previous page)

```

    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)
{
    /* Écrit dans le tampon comme requis par l'affichage.
    * Écrit seulement 1 bit pour les écrans monochromes orientés verticalement : * /
    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 *__disp_drv_t* lv_disp_drv_t
Display Driver structure to be registered by HAL

typedef struct *__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 be 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

Interface de périphérique d'entrée

Types de périphériques d'entrée

Pour configurer un périphérique d'entrée, une variable `lv_indev_drv_t` doit être initialisée :

```
lv_indev_drv_t indev_drv;
lv_indev_drv_init(&indev_drv);           /* Initialisation de base */
indev_drv.type = ...                     /* Voir ci-dessous. */
indev_drv.read_cb = ...                  /* Voir ci-dessous. */
/* Enregistre le pilote dans LittlevGL et sauvegarde l'objet de périphérique d'entrée.
↳ créé */
lv_indev_t * my_indev = lv_indev_drv_register(&indev_drv);
```

type peut être

- **LV_INDEV_TYPE_POINTER** pavé tactile ou souris
- **LV_INDEV_TYPE_KEYPAD** clavier
- **LV_INDEV_TYPE_ENCODER** encodeur avec options gauche, droite et appui
- **LV_INDEV_TYPE_BUTTON** bouton externe

`read_cb` est une fonction de rappel qui sera appelé périodiquement pour indiquer l'état actuel d'un périphérique d'entrée. Les données peuvent être placées dans un tampon, la fonction retourne **false** lorsqu'il ne reste plus de données à lire ou **true** lorsque le tampon n'est pas vide.

Visitez [Périphériques d'entrée](#) pour en savoir plus sur les périphériques d'entrée en général.

Pavé tactile, souris ou autre pointeur

Les périphériques d'entrée pouvant cliquer sur des points de l'écran appartiennent à cette catégorie.

```
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; /* Pas de tampon donc plus de données à lire */
}
```

Important: Les pilotes de pavé tactile doivent renvoyer les dernières coordonnées X/Y même lorsque l'état est `LV_INDEV_STATE_REL`.

Pour définir un curseur de souris, utilisez `lv_indev_set_cursor(my_indev, &img_cursor)` (`my_indev` est la valeur de retour de `lv_indev_drv_register`).

Clavier

Les claviers complets avec toutes les lettres ou plus simples avec quelques boutons de navigation sont décrits ici.

Pour utiliser un clavier :

- Enregistrez une fonction `read_cb` avec le type `LV_INDEV_TYPE_KEYPAD`.
- Activez `LV_USE_GROUP` dans `lv_conf.h`
- Un groupe d'objets doit être créé : `lv_group_t * g = lv_group_create()` et des objets doivent y être ajoutés avec `lv_group_add_obj(g, obj)`
- Le groupe créé doit être affecté à un périphérique d'entrée : `lv_indev_set_group(my_indev, g)` (`my_indev` est la valeur de retour de `lv_indev_drv_register`)
- Utilisez `LV_KEY_...` pour naviguer parmi les objets du groupe. Voir `lv_core/lv_group.h` pour les touches disponibles.

```
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){
```

(continues on next page)

(continued from previous page)

```
data->key = last_key();           /* Obtient la dernière touche pressée ou ↵
↵relâchée */

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

return false; /* Pas de tampon donc plus de données à lire */
}
```

Encodeur

Avec un encodeur, vous pouvez réaliser 4 actions :

1. Appuyer son bouton
2. Appuyer longuement son bouton
3. Tourner à gauche
4. Tourner à droite

En bref, les encodeurs fonctionnent comme ceci :

- En tournant l’encodeur, vous pouvez sélectionner l’objet suivant/précédent.
- Lorsque vous appuyez sur l’encodeur sur un objet simple (comme un bouton), vous cliquez dessus.
- Si vous appuyez sur l’encodeur sur un objet complexe (comme une liste, une boîte de message, etc.), l’objet passera en mode édition. Vous pouvez alors naviguer dans l’encodeur en le tournant.
- Pour quitter le mode édition, appuyez longuement sur le bouton.

Pour utiliser un *encodeur* (comme un *clavier*), des objets doivent être ajoutés aux groupes.

```
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; /* Pas de tampon donc plus de données à lire */
}
```

Bouton

Bouton signifie bouton “matériel” externe à côté de l’écran, affecté à des coordonnées spécifiques de l’écran. Si un bouton est pressé, il simule l’appui sur la coordonnée attribuée (comme un pavé tactile)

Pour affecter des boutons aux coordonnées, utilisez `lv_indev_set_button_points(my_indev, points_array)`. `points_array` doit ressembler à `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; /* Mémorise le dernier bouton pressé */
    int btn_pr = my_btn_read(); /* Obtient l'ID (0, 1, 2 ...) du bouton pressé */
    if(btn_pr >= 0) { /* Un bouton est-il pressé ? P.ex. -1 indique qu
↳ 'aucun bouton n'est pressé */
        last_btn = btn_pr; /* Sauvegarde l'ID du bouton pressé */
        data->state = LV_INDEV_STATE_PR; /* Définit l'état pressé */
    } else {
        data->state = LV_INDEV_STATE_REL; /* Définit l'état relâché */
    }

    data->btn = last_btn; /* Enregistre le dernier bouton */

    return false; /* Pas de tampon donc plus de données à lire */
}

```

Autres fonctionnalités

Outre `read_cb`, une autre fonction de rappel `feedback_cb` peut également être spécifiée dans `lv_indev_drv_t`. `feedback_cb` est appelée lorsqu'un événement, quel qu'il soit, est envoyé par les périphériques d'entrée. (indépendamment de leur type). Cela permet de faire un retour à l'utilisateur, par exemple. jouer un son sur `LV_EVENT_CLICK`.

La valeur par défaut des paramètres suivants peut être définie dans `lv_conf.h` mais la valeur par défaut peut être surchargée dans `lv_indev_drv_t` :

- **drag_limit** Nombre de pixels à parcourir avant de faire glisser l'objet
- **drag_throw** Ralentissement du glissé après lâché en [%]. Une valeur haute signifie un ralentissement plus rapide
- **long_press_time** Temps d'appui avant de générer `LV_EVENT_LONG_PRESSED` (en millisecondes)
- **long_press_rep_time** Intervalle de temps entre deux envois `LV_EVENT_LONG_PRESSED_REPEAT` (en millisecondes)
- **read_task** pointeur sur l'objet `lv_task` qui lit le périphérique d'entrée. Ses paramètres peuvent être modifiés avec les fonctions `lv_task_...()`

Chaque périphérique d'entrée est associé à un affichage. Par défaut, un nouveau périphérique d'entrée est ajouté à l'affichage créé en dernier ou explicitement sélectionné (à l'aide de `lv_disp_set_default()`). L'affichage associé est sauvegardé et peut être modifié dans le champ `disp` du pilote.

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
```

Interface tic

LittlevGL a besoin d'un tic système pour connaître le temps écoulé pour l'animation et d'autres tâches.

Vous devez appeler la fonction `lv_tick_inc(tick_period)` périodiquement et indiquer la période d'appel en millisecondes. Par exemple, si elle est appelée chaque milliseconde : `lv_tick_inc(1)`.

`lv_tick_inc` doit être appelée dans une routine de priorité supérieure à `lv_task_handler()` (par exemple, dans une interruption) pour connaître avec précision les millisecondes écoulées, même si l'exécution de `lv_task_handler` prend plus de temps.

Sur FreeRTOS, `lv_tick_inc` peut être appelée dans `vApplicationTickHook`.

Sur les systèmes d'exploitation basés sur Linux (par exemple, sur Raspberry), `lv_tick_inc` peut être appelée dans un fil d'exécution :

```
void * tick_thread (void *args)
{
    while(1) {
        usleep(5*1000); /* Dors pendant 5 millisecondes */
        lv_tick_inc(5); /* Indique à LittlevGL que 5 millisecondes se sont
↳ écoulées */
    }
}
```

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()`)

Gestionnaire de tâche

Pour gérer les tâches de LittlevGL, vous devez appeler `lv_task_handler()` régulièrement à partir d'un des éléments suivants :

- boucle `while(1)` de la fonction `main()`
- interruption périodique d'une horloge (priorité plus basse que `lv_tick_inc()`)
- une tâche périodique du SE

Le délai n'est pas critique, mais il faut environ 5 millisecondes pour que le système reste réactif.

Exemple :

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

Pour en savoir plus sur les tâches, visitez la section *Tâches*.

Gestion du sommeil

Le microcontrôleur peut s'endormir lorsque aucune intervention de l'utilisateur n'est effectuée. Dans ce cas, la boucle principale `while (1)` devrait ressembler à ceci :

```
while(1) {
    /* Fonctionnement normal (pas de sommeil) si moins de 1 seconde d'inactivité */
    if(lv_disp_get_inactive_time(NULL) < 1000) {
        lv_task_handler();
    }
    /* Sommeil après une seconde d'inactivité */
    else {
        timer_stop(); /* Arrêt de l'horloge où lv_tick_inc() est appelée */
        sleep();      /* Place le microcontrôleur en sommeil */
    }
    my_delay_ms(5);
}
```

Vous devez également ajouter ces lignes à la fonction de lecture de votre périphérique d'entrée si un appui est effectué :

```
lv_tick_inc(LV_DISP_DEF_REFR_PERIOD); /* Force l'exécution de la tâche au réveil */
timer_start();                       /* Redémarre l'horloge où lv_tick_inc() est
↪appelée */
lv_task_handler();                    /* Appelle `lv_task_handler()` manuellement
↪pour traiter l'appui */
```

En plus de `lv_disp_get_inactive_time()`, vous pouvez vérifier `lv_anim_count_running()` pour voir si toutes les animations sont terminées.

Système d'exploitation et interruptions

LittlevGL n'est **pas compatible avec les fils d'exécution** par défaut.

Cependant, dans les cas suivants, il est valide d'appeler des fonctions liées à LittlevGL :

- Dans les *événements*. Apprenez-en plus dans *Événements*.
- Dans *lv_tasks*. Apprenez-en plus dans *Tâches*.

Tâches et fils d'exécution

Si vous avez besoin d'utiliser de vraies tâches ou des fils d'exécution, vous avez besoin d'un mutex qui doit être acquis avant l'appel de `lv_task_handler` et libéré ensuite. Par ailleurs, vous devez utiliser ce mutex dans les autres tâches et fils d'exécution lors de chaque appel de fonction et code liés à LittlevGL (`lv_..`). De cette façon, vous pouvez utiliser LittlevGL dans un environnement réellement multitâche. Utilisez simplement un mutex pour éviter l'appel concurrent des fonctions LittlevGL.

Interruptions

Essayez d'éviter d'appeler les fonctions LittlevGL à partir d'une interruption (sauf `lv_tick_inc()` et `lv_disp_flush_ready()`). Mais si vous avez vraiment besoin de le faire, vous devez désactiver l'interruption qui utilise les fonctions LittlevGL pendant que `lv_task_handler` est en cours d'exécution. Il est préférable de positionner un indicateur ou toute autre valeur et de le vérifier périodiquement dans une `lv_task`.

Journalisation

LittlevGL a un module *journal* intégré pour informer l'utilisateur de ce qui se passe dans la librairie.

Niveau de détail

Pour activer la journalisation, définissez `LV_USE_LOG 1` dans *lv_conf.h* et définissez `LV_LOG_LEVEL` sur l'une des valeurs suivantes :

- `LV_LOG_LEVEL_TRACE` Beaucoup de messages pour donner des informations détaillées
- `LV_LOG_LEVEL_INFO` Consigne les événements importants
- `LV_LOG_LEVEL_WARN` Journalise si quelque chose d'inattendu s'est produit mais n'a pas causé de problème
- `LV_LOG_LEVEL_ERROR` Uniquement les problèmes critiques, lorsque le système peut planter

- **LV_LOG_LEVEL_NONE** Ne journalise rien

Les événements dont le niveau est supérieur au niveau de journalisation défini seront également consignés. P. ex. si vous activez **LV_LOG_LEVEL_WARN**, les erreurs seront également consignées.

Journalisation avec printf

Si votre système prend en charge **printf**, il vous suffit d'activer **LV_LOG_PRINTF** dans *lv_conf.h* pour traiter les journaux avec **printf**.

Fonction de journalisation personnalisée

Si vous ne pouvez pas utiliser **printf** ou si vous souhaitez utiliser une fonction personnalisée pour journaliser, vous pouvez enregistrer une fonction de rappel "journaliseur" avec **lv_log_register_print()**.

Par exemple :

```
void my_log_cb(lv_log_level_t level, const char * file, int line, const char * dsc)
{
    /* Envoie les messages via le port série */
    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);
```

Ajouter des messages

Vous pouvez également utiliser le module de journalisation via les fonctions **LV_LOG_TRACE/INFO/WARN/ERROR(description)**.

3.15.3 Vue d'ensemble

Objets

Dans LittlevGL, les **éléments de base** d'une interface utilisateur sont les objets, également appelés *éléments visuels*. Par exemple, un *Bouton*, une *Etiquette*, une *Image*, une *Liste*, un *Graphique* ou une *Zone de texte*.

Découvrez tous les *Types d'objet* ici.

Attributs d'objet

Attributs de base

Les objets ont des attributs de base communs indépendamment de leur type :

- Position
- Taille
- Parent
- Autorisation du glissé
- Autorisation du clic etc.

Vous pouvez définir/obtenir ces attributs avec les fonctions `lv_obj_set _...` et `lv_obj_get _....`. Par exemple :

```
/* Définit les attributs de base de l'objet */
lv_obj_set_size(btn1, 100, 50);          /* Taille du bouton */
lv_obj_set_pos(btn1, 20,30);             /* Position du bouton */
```

Pour voir toutes les fonctions disponibles, visitez la *documentation* de l'objet de base.

Attributs spécifiques

Les types d'objet ont aussi des attributs spéciaux. Par exemple, un curseur a

- Des valeurs minimum et maximum
- Une valeur courante
- Des styles personnalisés

Pour ces attributs, chaque type d'objet possède des fonctions API uniques. Par exemple pour un curseur :

```
/* Définit les attributs spécifiques du curseur */
lv_slider_set_range(slider1, 0, 100);      /* Définit les valeurs minimum et ↵
↵maximum */
lv_slider_set_value(slider1, 40, LV_ANIM_ON); /* Définit la valeur courante ↵
↵(position) */
lv_slider_set_action(slider1, my_action);   /* Définit une fonction de rappel */
```

Les API des types d'objet sont décrites dans leur *Documentation* mais vous pouvez également consulter les fichiers d'en-tête respectifs (p.ex. `lv_objx/lv_slider.h`).

Mécanismes de fonctionnement de l'objet

Structure parent-enfant

Un objet parent peut être considéré comme le conteneur de ses enfants. Chaque objet a exactement un objet parent (à l'exception des écrans), mais un parent peut avoir un nombre illimité d'enfants. Il n'y a pas de contrainte pour le type du parent, mais il existe des objets parent typiques (p.ex. un bouton) et enfants (p.ex. une étiquette).

Se déplacer ensemble

Si la position du parent est modifiée, les enfants se déplaceront avec lui. Par conséquent, toutes les positions sont relatives au parent.

Les coordonnées (0, 0) signifient que les objets resteront dans le coin supérieur gauche du parent indépendamment de la position du parent.



```
lv_obj_t * par = lv_obj_create(lv_scr_act(), NULL); /* Crée un objet parent sur l
↳ 'écran actuel */
lv_obj_set_size(par, 100, 80); /* Définit la taille du
↳ parent */

lv_obj_t * obj1 = lv_obj_create(par, NULL); /* Crée un objet sur l
↳ 'objet parent créé précédemment */
lv_obj_set_pos(obj1, 10, 10); /* Définit la position du
↳ nouvel objet */
```

Modifiez la position du parent :



```
lv_obj_set_pos(par, 50, 50);           /* Déplacez le parent. L'enfant va bouger avec. */
```

Pour simplifier, la définition des couleurs des objets n'est pas montrée dans l'exemple.

Visibilité uniquement sur le parent

Si un enfant est partiellement ou complètement hors de son parent, les parties extérieures ne seront pas visibles.



```
lv_obj_set_x(obj1, -30);                /* Déplace l'enfant en partie en dehors du parent */
```

Créer - supprimer des objets

Dans LittlevGL, les objets peuvent être créés et supprimés dynamiquement à l'exécution. Cela signifie que seuls les objets actuellement créés consomment de la MEV. Par exemple, si vous avez besoin d'un graphique, vous pouvez le créer à l'utilisation et le supprimer s'il n'est pas visible ou plus nécessaire.

Chaque type d'objet a sa propre fonction **create** avec une signature unifiée. Deux paramètres sont nécessaires :

- un pointeur sur l'objet parent. Pour créer un écran, donnez *NULL* en tant que parent.
- éventuellement un pointeur sur un autre objet du même type pour copie. Peut être *NULL* pour ne pas copier un autre objet.

Indépendamment du type d'objet, un type de variable commun `lv_obj_t` est utilisé. Ce pointeur peut être utilisé ultérieurement pour définir ou obtenir les attributs de l'objet.

Les fonctions de création ressemblent à ceci :

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

Il existe une fonction commune de **suppression** pour tous les types d'objet. Il supprime l'objet et tous ses enfants.

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

`lv_obj_del` supprimera immédiatement l'objet. Si pour une quelconque raison vous ne pouvez pas supprimer l'objet immédiatement, vous pouvez utiliser `lv_obj_del_async(obj)`. Utile, p.ex. si vous voulez supprimer le parent d'un objet dans le traitement de l'événement `LV_EVENT_DELETE`.

Vous pouvez supprimer uniquement les enfants d'un objet mais laissez l'objet lui-même "vivant" :

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

Ecran - le parent le plus élémentaire

Les écrans sont des objets spéciaux qui n'ont pas d'objet parent. Il est donc créé ainsi :

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

Il y a toujours un écran actif affiché. Par défaut, la librairie en crée et en charge un. Pour obtenir l'écran actuellement actif, utilisez la fonction `lv_scr_act()` pour en charger un nouveau, utilisez `lv_scr_load(scr1)`.

Les écrans peuvent être créés avec n'importe quel type d'objet. Par exemple, un *Objet de base* ou une image pour créer un fond d'écran.

Les écrans sont créés sur l' *affichage par défaut*. L'écran *par défaut* est le dernier écran enregistré avec `lv_disp_drv_register` (s'il n'y a qu'un seul écran, alors c'est celui-ci) ou vous pouvez explicitement sélectionner l'affichage avec `lv_disp_set_default (disp)`. `lv_scr_act()` et `lv_scr_load()` fonctionnent sur l'écran courant par défaut.

Visitez *Support multi-affichage* pour en savoir plus.

Couches

Ordre de création

L'objet créé le plus tôt (et ses enfants) sera dessiné le plus tôt (au plus près de l'arrière-plan). En d'autres termes, le dernier objet créé sera au-dessus de ses frères et sœurs. C'est très important, l'ordre est calculé entre les objets de même niveau ("frères et sœurs").

Des couches peuvent être facilement ajoutées en créant 2 objets (qui peuvent être transparents). D'abord 'A' et ensuite 'B'. 'A' et tous les objets qu'il contient seront à l'arrière-plan et pourront être couverts par 'B' et ses enfants.



```
/* Crée un écran */
lv_obj_t * scr = lv_obj_create(NULL, NULL);
lv_scr_load(scr);          /* Charge l'écran */

/* Crée 2 boutons */
lv_obj_t * btn1 = lv_btn_create(scr, NULL);          /* Crée un bouton sur l'écran */
lv_btn_set_fit(btn1, true, true);                    /* Permet de définir
↳ automatiquement la taille en fonction du contenu */
lv_obj_set_pos(btn1, 60, 40);                        /* Définit la position du
↳ bouton */

lv_obj_t * btn2 = lv_btn_create(scr, btn1);           /* Copie le premier bouton */
lv_obj_set_pos(btn2, 180, 80);                       /* Définit la position du bouton */

/* Ajoute des étiquettes aux boutons */
lv_obj_t * label1 = lv_label_create(btn1, NULL);      /* Crée une étiquette sur le
↳ premier bouton */
lv_label_set_text(label1, "Button 1");                /* Définit le texte de l
↳ étiquette */

lv_obj_t * label2 = lv_label_create(btn2, NULL);      /* Crée une étiquette sur
↳ le deuxième bouton */
lv_label_set_text(label2, "Button 2");                /* Définit le texte de l
↳ étiquette */
```

(continues on next page)

(continued from previous page)

```
/* Supprime la deuxième étiquette */
lv_obj_del(label2);
```

Amener au premier plan

Il y a plusieurs façons d'amener un objet au premier plan :

- Utilisez `lv_obj_set_top(obj, true)`. Si vous cliquez sur `obj` ou l'un de ses enfants, LittlevGL amènera automatiquement l'objet au premier plan. Cela fonctionne de manière similaire aux fenêtres sur PC. Lorsque vous cliquez sur une fenêtre en arrière-plan, elle apparaît automatiquement au premier plan.
- Utilisez `lv_obj_move_foreground(obj)` et `lv_obj_move_background(obj)` pour indiquer explicitement à la librairie de placer un objet au premier plan ou de le passer à l'arrière-plan.
- Quand `lv_obj_set_parent(obj, new_parent)` est utilisé, `obj` sera au premier plan sur le nouveau parent.

Couches supérieure et système

Il y a deux couches spéciales appelées `layer_top` et `layer_sys`. Les deux sont visibles et identiques sur tous les écrans d'un affichage. `layer_top` est au-dessus de "l'écran normal" et `layer_sys` lui-même au-dessus de `layer_top`.

`layer_top` peut être utilisé par l'utilisateur pour créer du contenu visible partout. Par exemple, une barre de menus, une fenêtre contextuelle, etc. Si l'attribut `click` est activé, `layer_top` absorbera tous les clics de l'utilisateur et agira comme modal.

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

`layer_sys` est utilisé par LittlevGL. Par exemple, la librairie y place le curseur de la souris pour s'assurer qu'il est toujours visible.

Événements

Dans LittlevGL, des événements sont déclenchés s'il se produit quelque chose d'intéressant pour l'utilisateur. Par exemple un objet

- est cliqué
- est déplacé
- sa valeur a changé, etc.

L'utilisateur peut affecter une fonction de rappel à un objet pour voir ces événements. Dans la pratique, cela ressemble à ceci :

```
lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);
lv_obj_set_event_cb(btn, my_event_cb); /* Assigne une fonction de rappel */

...

static void my_event_cb(lv_obj_t * obj, lv_event_t event)
{
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

/* Etc. */
}
```

Plusieurs objets peuvent utiliser la même *fonction de rappel*.

Types d'événements

Les types d'événements suivants existent :

Evénements génériques

Tout objet peut recevoir ces événements indépendamment de son type. C-à- d. ces événements sont envoyés aux boutons, aux étiquettes, aux curseurs, etc.

Liés au périphérique d'entrée

Envoyés lorsqu'un objet est pressé, relâché, etc. par l'utilisateur. Ils sont utilisés pour les périphériques d'entrée *clavier*, *encodeur* et *bouton*, ainsi que pour les *pointeurs*. Consultez la section *Périphériques d'entrée* pour en savoir plus à leur sujet.

- **LV_EVENT_PRESSED** L'objet a été pressé
- **LV_EVENT_PRESSING** L'objet est pressé(envoyé continuellement pendant l'appui)
- **LV_EVENT_PRESS_LOST** Toujours pressé mais hors de l'objet
- **LV_EVENT_SHORT_CLICKED** Relâché avant **LV_INDEV_LONG_PRESS_TIME**. Pas généré si l'objet est déplacé.

- **LV_EVENT_LONG_PRESSED** Pressé durant **LV_INDEV_LONG_PRESS_TIME**. Pas généré si l'objet est déplacé.
- **LV_EVENT_LONG_PRESSED_REPEAT** Généré après **LV_INDEV_LONG_PRESS_TIME** à chaque **LV_INDEV_LONG_PRESS_REP_TIME** ms. Pas généré si l'objet est déplacé.
- **LV_EVENT_CLICKED** Généré au relâché si l'objet n'est pas déplacé (indépendamment d'un appui long)
- **LV_EVENT_RELEASED** Généré dans tous les cas lorsque l'objet a été relâché, même s'il a été déplacé. Non généré si il y a eu déplacement pendant l'appui et si le relâché intervient hors de l'objet. Dans ce cas, **LV_EVENT_PRESS_LOST** est généré.

Liés au pointeur

Ces événements sont envoyés uniquement par des périphériques d'entrée de type pointeur (p.ex. souris ou pavé tactile).

- **LV_EVENT_DRAG_BEGIN** Le déplacement de l'objet a débuté
- **LV_EVENT_DRAG_END** Dragging finished (including drag throw)
- **LV_EVENT_DRAG_THROW_BEGIN** Drag throw started (released after drag with “momentum”)

Liés au clavier et à l'encodeur

Ces événements sont envoyés par les périphériques d'entrée clavier et encodeur. En savoir plus sur les *groupes* dans la section [Périphériques d'entrée] (overview/indev).

- **LV_EVENT_KEY** Une touche* est envoyée à l'objet. Typiquement quand elle a été pressée ou répétée après un appui long
- **LV_EVENT_FOCUSED** L'objet est activé dans son groupe
- **LV_EVENT_DEFOCUSED** L'objet est désactivé dans son groupe

Événements généraux

Autres événements généraux envoyés par la librairie.

- **LV_EVENT_DELETE** L'objet est en cours de suppression. Libérez les données associées allouées par l'utilisateur.

Événements spéciaux

Ces événements sont spécifiques à un type particulier d'objet.

- **LV_EVENT_VALUE_CHANGED** La valeur de l'objet a changé (p.ex. pour un *Curseur*)
- **LV_EVENT_INSERT** Quelque chose est inséré dans l'objet (typiquement à une *Zone de texte*)
- **LV_EVENT_APPLY** “Ok”, “Appliquer” ou un bouton spécifique similaire a été cliqué (typiquement à partir d'un objet *Clavier*)
- **LV_EVENT_CANCEL** “Fermer”, “Annuler” ou un bouton spécifique similaire a été cliqué (typiquement à partir d'un objet *Clavier*)

- **LV_EVENT_REFRESH** Demande à actualiser l'objet. Jamais généré par la librarie mais peut l'être par l'utilisateur.

Pour voir exactement quels événements sont utilisés par un type d'objet, voir la documentation des *Types d'objet*.

Données personnalisées

Certains événements peuvent comporter des données personnalisées. Par exemple, **LV_EVENT_VALUE_CHANGED** indique dans certains cas la nouvelle valeur. Pour plus d'informations, voir la documentation des *Types d'objet*. Pour obtenir les données personnalisées dans la fonction de rappel, utilisez `lv_event_get_data()`.

Le type des données personnalisées dépend de l'objet, mais si c'est un

- entier alors c'est un `uint32_t *` ou un `int32_t *`
- texte alors c'est un `char *` ou un `const char *`

Envoyer des événements manuellement

Pour envoyer manuellement des événements à un objet, utilisez `lv_event_send(obj, LV_EVENT_..., &custom_data)`.

Cela peut être utilisé par exemple pour fermer manuellement une boîte de message en simulant un appui sur un bouton :

```
/* Simuler l'appui du premier bouton (les index partent de zéro) */
uint32_t btn_id = 0;
lv_event_send(mbox, LV_EVENT_VALUE_CHANGED, &btn_id);
```

Ou pour demander une actualisation de manière générique.

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

Styles

Les *styles* sont utilisés pour définir l'apparence des objets. Un style est une variable structurée avec des attributs tels que couleurs, marges, opacité, police, etc.

Il existe un type de style commun nommé `lv_style_t` pour chaque type d'objet.

En définissant les champs des variables `lv_style_t` et en les affectant à un objet, vous pouvez modifier l'apparence des objets.

Important: Les objets mémorisent uniquement un pointeur vers un style. Le style ne peut donc pas être une variable locale détruite après la sortie de la fonction. **Vous devez utiliser des variables statiques, globales ou allouées dynamiquement.**

```
lv_style_t style_1;           /* OK ! Les variables globales pour les styles sont ↵
↵adaptées */
static lv_style_t style_2;    /* OK ! Les variables statiques en dehors des ↵
↵fonctions sont adaptées */
```

(continues on next page)

(continued from previous page)

```
void my_screen_create(void)
{
    static lv_style_t style_3;    /* OK ! Les variables statiques dans les fonctions
    ↪ sont adaptées */
    lv_style_t style_4;          /* Non ! Les styles ne peuvent pas être des variables
    ↪ locales */

    ...
}
```

Utiliser les styles

Les objets ont un *style principal* qui détermine l'apparence de leur arrière-plan ou de leur partie principale. Cependant, certains types d'objet ont aussi des styles supplémentaires.

Certains objets ont un seul style. P.ex.

- Etiquette
- Image
- Ligne, etc

Par exemple, un curseur a 3 styles :

- Arrière-plan (style principal)
- Indicateur
- Bouton

Chaque type d'objet a ses propres fonctions de gestion des styles. Par exemple

```
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);
```

Pour voir les styles pris en charge par un type d'objet (*LV_<OBJ_TYPE>STYLE<STYLE_TYPE>*) vérifier la documentation du *Type d'objet* particulier.

Si vous **modifiez un style déjà utilisé** par un ou plusieurs objets, les objets doivent être avertis du changement de style. Vous avez deux possibilités pour le faire :

```
/* Notifie un objet que son style est modifié */
void lv_obj_refresh_style(lv_obj_t * obj);

/* Notifie tous les objets avec un style donné (NULL pour notifier tous les objets) */
void lv_obj_report_style_mod(void * style);
```

`lv_obj_report_style_mod` ne peut actualiser que les *styles principaux*.

Héritage de styles

Si le *style principal* d'un objet est **NULL**, son style sera hérité du style de son parent. Cela facilite la création d'une interface cohérente. N'oubliez pas qu'un style décrit beaucoup de propriétés en même temps. Ainsi, par exemple, si vous définissez le style d'un bouton et créez une étiquette avec le style **NULL**, l'étiquette sera rendue en fonction du style du bouton. En d'autres termes, le bouton garantit à ses enfants une apparence correcte.

La définition de la propriété de style **glass** empêchera d'hériter ce style. Vous devez l'utiliser si le style est transparent pour que les enfants utilisent les couleurs et autres des grands-parents.

Propriétés de style

Un style comporte 5 parties principales : commun, corps, texte, image et ligne. Un objet utilisera les champs qui le concernent. Par exemple, les *lignes* ne se soucient pas de *letter_space*. Pour voir quels champs sont utilisés par un type d'objet, voir la documentation des *Types d'objet*.

Les champs d'une structure de style sont les suivants :

Propriétés communes

- **glass** 1: Ne pas hériter de ce style

Propriétés de style de corps

Utilisé par les objets rectangulaires

- **body.main_color** Couleur principale (couleur du haut)
- **body.grad_color** Dégradé de couleur (couleur de fond)
- **body.radius** Rayon pour arrondir les angles (**LV_RADIUS_CIRCLE** pour dessiner un cercle)
- **body.opa** Opacité (0..255 ou **LV_OPA_TRANSP**, **LV_OPA_10**, **LV_OPA_20** ... **LV_OPA_COVER**)
- **body.border.color** Couleur de bord
- **body.border.width** Largeur de bord
- **body.border.part** Segments de bord (**LV_BORDER_LEFT**/**RIGHT**/**TOP**/**BOTTOM**/**FULL** ou 'OR' de plusieurs valeurs)
- **body.border.opa** Opacité du bord (0..255 ou **LV_OPA_TRANSP**, **LV_OPA_10**, **LV_OPA_20** ... **LV_OPA_COVER**)
- **body.shadow.color** Couleur de l'ombre
- **body.shadow.width** Largeur de l'ombre
- **body.shadow.type** Type d'ombre (**LV_SHADOW_BOTTOM**/**FULL**)
- **body.padding.top** Marge haute
- **body.padding.bottom** Marge basse
- **body.padding.left** Marge gauche
- **body.padding.right** Marge droite
- **body.padding.inner** Marge intérieure (entre les éléments constitutifs ou les enfants)

Propriétés de style de texte

Utilisés par les objets qui affichent du texte

- **text.color** Couleur de texte

- **text.sel_color** Couleur de texte sélectionné
- **text.font** Pointeur vers une police
- **text.opa** Opacité du texte (0..255 ou LV_OPA_TRANSP, LV_OPA_10, LV_OPA_20 ... LV_OPA_COVER*)
- **text.letter_space** Espace de lettre
- **text.line_space** Espace de ligne

Propriétés de style d'image

Utilisé par les objets de type image ou les icônes sur les objets

- **image.color** Couleur pour la re-coloration de l'image en fonction de la luminosité des pixels
- **image.intense** Intensité de re-coloration (0..255 ou LV_OPA_TRANSP, LV_OPA_10, LV_OPA_20 ... LV_OPA_COVER)
- **image.opa** Opacité de l'image (0..255 ou LV_OPA_TRANSP, LV_OPA_10, LV_OPA_20 ... LV_OPA_COVER)

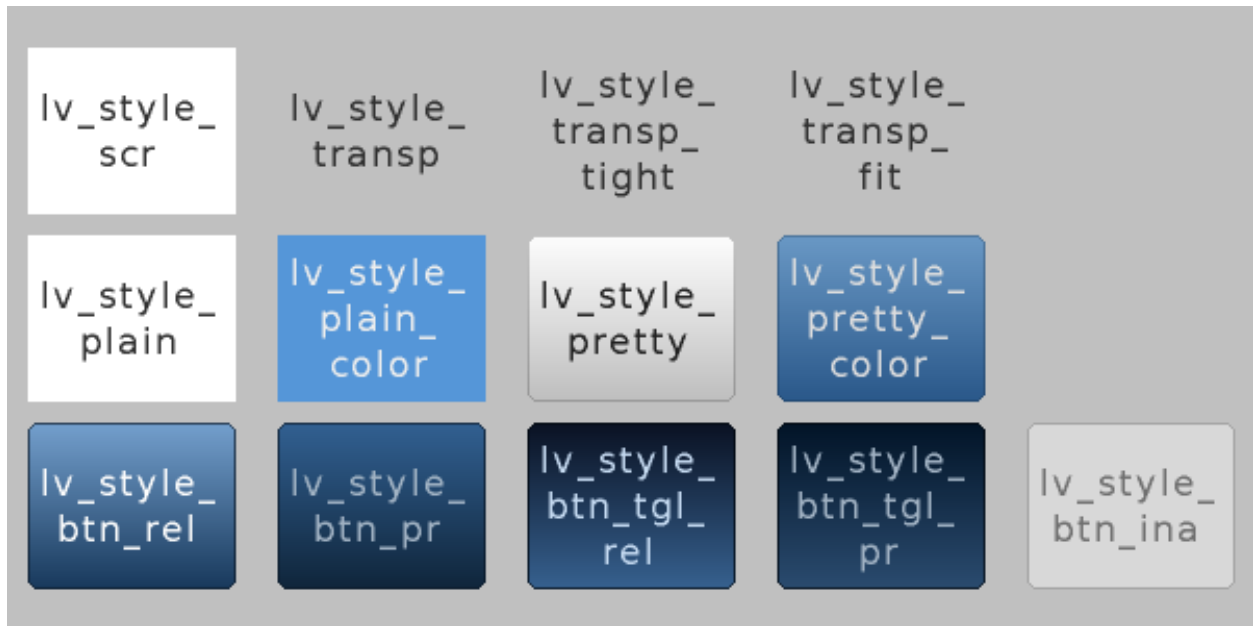
Propriétés de style de ligne

Utilisé par des objets contenant des lignes ou des éléments de type ligne

- **line.color** Couleur de ligne
- **line.width** Largeur de ligne
- **line.opa** Opacité de ligne (0..255 or LV_OPA_TRANSP, LV_OPA_10, LV_OPA_20 ... LV_OPA_COVER)

Styles intégrés

Il existe plusieurs styles intégrés dans la librairie :



Comme vous pouvez le constater, il existe un style pour les écrans, un pour les boutons, des styles simples ou améliorés et des styles transparents.

Les styles `lv_style_transp`, `lv_style_transp_fit` et `lv_style_transp_tight` diffèrent uniquement par les marges : pour `lv_style_transp_tight` les marges sont nulles, pour `lv_style_transp_fit` seules les marges horizontales et verticales sont nulles mais il y a une marge intérieure.

Important: Les styles intégrés transparents ont *glass* = 1 par défaut, ce qui signifie que ces styles (les couleurs, par exemple) ne seront pas hérités par les enfants.

Les styles intégrés sont des variables globales `lv_style_t`. Vous pouvez les utiliser ainsi :

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

Créer de nouveaux styles

Vous pouvez modifier les styles intégrés ou en créer de nouveaux.

Lors de la création de nouveaux styles, il est recommandé de copier d'abord un style intégré avec `lv_style_copy(&dest_style, &src_style)` pour s'assurer que tous les champs sont initialisés avec une valeur appropriée.

N'oubliez pas que le style créé doit être **statique** ou global. Par exemple :

```
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;
```

Animations de style

Vous modifiez les styles avec des animations en utilisant la fonction `lv_style_anim _... ()`. Deux styles sont requis pour représenter les états *initial* et *final*, et un troisième style qui sera animé. Voici un exemple pour montrer comment cela fonctionne.

```
lv_anim_t a;
lv_style_anim_init(&a);                                /*
↳Initialisation de base */
lv_style_anim_set_styles(&a, &style_to_anim, &style_start, &style_end); /* Définit
↳les styles à utiliser */
lv_style_anim_set_time(&a, duration, delay);           /* Définit la
↳durée et le délai */
lv_style_anim_create(&a);                               /* Crée l
↳animation */
```

Pour découvrir l'intégralité de l'API des animations de style, voir `lv_core/lv_style.h`.

Ici, vous pouvez en apprendre plus sur les *Animations*.

Exemple de style

L'exemple ci-dessous illustre l'utilisation des styles.



```
/* Crée un style */
static lv_style_t style1;
lv_style_copy(&style1, &lv_style_plain); /* Copie un style intégré pour
↳initialiser le nouveau 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.left = 5; /* Marge horizontale, utilisée par l
↳indicateur de barre ci-dessous */
style1.body.padding.right = 5;
style1.body.padding.top = 5; /* Marge verticale, utilisée par l'indicateur
↳de barre ci-dessous */
style1.body.padding.bottom = 5;
style1.text.color = LV_COLOR_RED;

/* Crée un simple objet */
```

(continues on next page)

(continued from previous page)

```
lv_obj_t *obj1 = lv_obj_create(lv_scr_act(), NULL);
lv_obj_set_style(obj1, &style1);          /* Applique le style créé */
lv_obj_set_pos(obj1, 20, 20);             /* Définit la position */

/* Crée une étiquette sur l'objet. Le style de l'étiquette est NULL par défaut */
lv_obj_t *label = lv_label_create(obj1, NULL);
lv_obj_align(label, NULL, LV_ALIGN_CENTER, 0, 0); /* Aligne l'étiquette au
↪milieu */

/* Crée une barre */
lv_obj_t *bar1 = lv_bar_create(lv_scr_act(), NULL);
lv_bar_set_style(bar1, LV_BAR_STYLE_INDIC, &style1); /* Modifie le style de l
↪'indicateur */
lv_bar_set_value(bar1, 70);                /* Définit la valeur de la
↪barre */
```

Thèmes

Il est difficile de créer des styles pour votre interface graphique, car vous avez besoin d'une profonde compréhension de la librairie et de compétences en matière de conception. En outre, il faut beaucoup de temps pour créer autant de styles.

Pour accélérer la conception les thèmes sont introduits. Un thème est une collection de styles contenant les styles requis pour chaque type d'objet. Par exemple, 5 styles de boutons décrivant leurs 5 états possibles. Consultez les [Thèmes] (<https://littlevgl.com/themes>) existants ou essayez-les dans la section [Démonstration en ligne] (<https://littlevgl.com/live-demo>).

Pour être plus précis, un thème est une variable structurée qui contient beaucoup de champs `lv_style_t*`. Pour les boutons :

```
theme.btn.rel      /* Style de bouton relâché */
theme.btn.pr       /* Style de bouton pressé */
theme.btn.tgl_rel  /* Style de bouton bascule relâché */
theme.btn.tgl_pr   /* Style de bouton bascule pressé */
theme.btn.ina      /* Style de bouton inactif */
```

Un thème peut être initialisé par : `lv_theme_<nom>_init(hue, font)`. Où `hue` est une valeur de teinte de l'espace colorimétrique HSV (<https://en.wikipedia.org/wiki/Hue>) (0..360) et `font` est la police appliquée dans le thème (NULL utilise `LV_FONT_DEFAULT`)

Quand un thème est initialisé, ses styles peuvent être utilisés comme ceci :



```
/* Crée un curseur par défaut */
lv_obj_t *slider = lv_slider_create(lv_scr_act(), NULL);
lv_slider_set_value(slider, 70);
lv_obj_set_pos(slider, 10, 10);
```

(continues on next page)

(continued from previous page)

```
/* Initialise le thème alien avec une teinte rouge */
lv_theme_t *th = lv_theme_alien_init(10, NULL);

/* Crée un nouveau curseur et applique les styles du thèmes */
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);
```

Vous pouvez demander à la librairie d'appliquer automatiquement les styles d'un thème lorsque vous créez de nouveaux objets. Pour ce faire, utilisez `lv_theme_set_current(th)` :

```
/* Initialise le thème alien avec une teinte rouge */
lv_theme_t *th = lv_theme_alien_init(10, NULL);
lv_theme_set_current(th);

/* Crée un curseur. Il utilisera le style du thème actuel. */
slider = lv_slider_create(lv_scr_act(), NULL);
```

Les thèmes peuvent être activés ou désactivés individuellement dans `lv_conf.h`.

Mise à jour automatique

Par défaut, si `lv_theme_set_current(th)` est appelé à nouveau, les styles des objets existants ne seront pas actualisés. Pour activer la mise à jour automatique des thèmes, activez `LV_THEME_LIVE_UPDATE` dans `lv_conf.h`.

La mise à jour automatique mettra à jour uniquement les objets dont le style provient du thème, c'est-à-dire créés après le premier appel de `lv_theme_set_current(th)` ou dont les styles ont été définis manuellement.

Périphériques d'entrée

En général périphériques d'entrée signifie :

- Périphériques de type pointeur tels que pavé tactile ou souris
- Claviers, normal ou simple pavé numérique
- Encodeurs avec mouvement rotatif à gauche / droite et bouton
- Boutons matériels externes affectés à des points spécifiques de l'écran

Important: Avant de poursuivre votre lecture, veuillez lire la section [Portage](/porting/indev) sur les périphériques d'entrée

Pointeurs

Les périphériques d'entrée de type pointeur peuvent avoir un curseur (typiquement pour les souris).


```
...
lv_indev_t * mouse_indev = lv_indev_drv_register(&indev_drv);

LV_IMG_DECLARE(mouse_cursor_icon);           /* Declare le fichier de l
↪ 'image. */
lv_obj_t * cursor_obj = lv_img_create(lv_scr_act(), NULL); /* Crée un objet image
↪ pour le curseur */
lv_img_set_src(cursor_obj, &mouse_cursor_icon);      /* Définit la source de l
↪ 'image */
lv_indev_set_cursor(mouse_indev, cursor_obj);        /* Connecte l'objet image
↪ au pilote */
```

Notez que l'objet curseur devrait avoir `lv_obj_set_click(cursor_obj, false)`. Pour les images cliquer est désactivé par défaut.

Clavier et encodeur

Vous pouvez contrôler entièrement l'interface utilisateur sans pavé tactile ou souris à l'aide d'un clavier ou d'un ou de plusieurs encodeurs. Cela fonctionne de manière similaire lorsque vous appuyez sur la touche *TAB* sur un PC pour sélectionner l'élément dans une application ou une page Web.

Groupes

Les objets que vous souhaitez contrôler avec un clavier ou un encodeur doivent être ajoutés à un *groupe*. Dans chaque groupe, il y a exactement un seul objet focalisé qui reçoit les notifications de touche pressée ou les actions du codeur. Par exemple, si une *Zone de texte* est sélectionnée et que vous appuyez sur une lettre d'un clavier, les codes sont envoyés et traités par la zone de texte. Ou si un *Curseur* est sélectionné et que vous appuyez sur les flèches gauche ou droite, la valeur du curseur sera modifiée.

Vous devez associer un périphérique d'entrée à un groupe. Un périphérique d'entrée peut envoyer les codes à un seul groupe, mais un groupe peut recevoir des données de plusieurs périphériques d'entrée.

Pour créer un groupe, utilisez `lv_group_t g = lv_group_create()` et pour ajouter un objet au groupe, utilisez `lv_group_add_obj(g, obj)`.

Pour associer un groupe à un périphérique d'entrée, utilisez `lv_indev_set_group(indev, g)`, où `indev` est la valeur de retour de `lv_indev_drv_register()`

Codes

Certains codes prédéfinis ont une signification particulière :

- **LV_KEY_NEXT** Sélectionne l'objet suivant
- **LV_KEY_PREV** Sélectionne l'objet précédant
- **LV_KEY_ENTER** Génère les événements **LV_EVENT_PRESSED/CLICKED/LONG_PRESSED** etc
- **LV_KEY_UP** Augmente la valeur ou se déplace vers le haut
- **LV_KEY_DOWN** Diminue la valeur ou se déplace vers le bas
- **LV_KEY_RIGHT** Augmente la valeur ou se déplace vers la droite
- **LV_KEY_LEFT** Diminue la valeur ou se déplace vers la gauche
- **LV_KEY_ESC** Ferme ou quitte (p.ex. ferme une *Liste déroulante*)

- **LV_KEY_DEL** Supprime (p.ex. le caractère à droite dans une *Zone de texte*)
- **LV_KEY_BACKSPACE** Supprime le caractère à gauche (p.ex. dans une *Zone de texte*)
- **LV_KEY_HOME** Se déplace au début ou en haut (p.ex. dans une *Zone de texte*)
- **LV_KEY_END** Se déplace à la fin (p.ex. dans une *Zone de texte*)

Les codes spéciaux les plus importants sont : **LV_KEY_NEXT/PREV**, **LV_KEY_ENTER** et **LV_KEY_UP/DOWN/LEFT/RIGHT**. Dans votre fonction **read_cb**, vous devez traduire certaines de vos codes en ces codes spéciaux pour naviguer dans le groupe et interagir avec l'objet sélectionné.

Habituellement, il suffit d'utiliser uniquement **LV_KEY_LEFT/RIGHT** car la plupart des objets peuvent être entièrement contrôlés avec eux.

Avec un encodeur, vous devez utiliser uniquement **LV_KEY_LEFT**, **LV_KEY_RIGHT** et **LV_KEY_ENTER**.

Edition et navigation

Avec les claviers, il y a beaucoup de touches, il est donc facile de naviguer entre les objets et de les éditer. Cependant, les encodeurs ont un nombre très limité de "touches". Afin de prendre en charge efficacement les encodeurs également, les modes *navigation* et *édition* sont créés.

En mode *navigation*, les **LV_KEY_LEFT/RIGHT** des encodeurs sont traduits en **LV_KEY_NEXT/PREV**. Par conséquent, l'objet suivant ou précédent sera sélectionné en tournant l'encodeur. Un appui sur **LV_KEY_ENTER** passera en mode *édition*.

En mode *édition*, **LV_KEY_NEXT/PREV** sont utilisés normalement pour éditer l'objet. En fonction du type d'objet, une pression courte ou longue de **LV_KEY_ENTER** repasse en mode *navigation*. Généralement, un objet sur lequel vous ne pouvez pas appuyer (comme un *Curseur*) quitte le mode *édition* en cas de clic bref, mais avec un objet pour lequel un clic court a une signification (par exemple, *Bouton*) un appui long est requis.

Styler l'objet sélectionné

Pour mettre en évidence visuellement l'élément sélectionné, son [Style principal] (/overview/style#utiliser-les-styles) sera mis à jour. Par défaut, de l'orange est mélangé aux couleurs d'origine du style. Une fonction de rappel pour modifier le style est définie par **lv_group_set_style_mod_cb(g, my_style_mod_cb)**. Cette fonction reçoit un pointeur sur un groupe d'objet et un style à modifier. Le modificateur de style par défaut ressemble à ceci (légèrement simplifié) :

```
static void default_style_mod_cb(lv_group_t * group, lv_style_t * style)
{
    /* Rend les corps un peu 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);

    /* Recolore le texte*/
    style->text.color = lv_color_mix(style->text.color, LV_COLOR_ORANGE, LV_OPA_70);
}
```

(continues on next page)

(continued from previous page)

```

/* Colorise les images */
if(style->image.intense < LV_OPA_MIN) {
    style->image.color = LV_COLOR_ORANGE;
    style->image.intense = LV_OPA_40;
}
}

```

Cette fonction de rappel modificateur de style est utilisée pour les claviers et encodeurs en mode *navigation*. En mode *édition*, une autre fonction de rappel est utilisée qui peut être définie avec `lv_group_set_style_mod_edit_cb()`. Par défaut, il utilise la couleur verte.

Démonstration en ligne

Essayez cette [Démonstration en ligne](#) pour voir comment une navigation de groupe sans pavé tactile fonctionne dans la pratique.

API

Périphérique d'entrée

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 input device

lv_obj_t ***lv_indev_get_obj_act**(void)

Gets a pointer to the currently active object in indev proc functions. NULL if no object is currently being handled or if groups aren't used.

Return pointer to currently active object

Groupes

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.

Affichage

Important: Le concept de base d'*affichage* dans LittlevGL est expliqué dans la section [Portage](/porting/display). Donc, avant de continuer à lire, veuillez lire cette section en premier.

Dans LittlevGL, vous pouvez avoir plusieurs affichages, chacun avec ses propres pilotes et objets.

Il est facile de créer plus d'affichages : il suffit d'initialiser les tampons d'affichage et d'enregistrer les pilotes pour chaque affichage. Lorsque vous créez l'interface utilisateur, utilisez `lv_disp_set_default(dispatch)` pour indiquer à la librairie dans quel affichage afficher l'objet.

Mais dans quels cas pouvez-vous utiliser le support multi-affichages ? Voici quelques exemples :

- Avoir un écran TFT "normal" avec une interface utilisateur locale et créer des écrans "virtuels" sur VNC à la demande (vous devez ajouter votre propre pilote VNC).
- Avoir un grand écran TFT et un petit écran monochrome.
- Avoir des écrans plus petits et simples dans un grand appareil
- Avoir deux grands écrans TFT : un pour le client et un pour le vendeur

Utiliser un seul affichage

L'utilisation de plusieurs affichages peut être utile, mais dans la plupart des cas, cela n'est pas nécessaire. Par conséquent, le concept de multi-affichages est complètement masqué si vous ne déclarez qu'un seul affichage. Par défaut, l'affichage créé en dernier (le seul) est utilisé par défaut.

`lv_scr_act()`, `lv_scr_load(scr)`, `lv_layer_top()`, `lv_layer_sys()`, `LV_HOR_RES` et `LV_VER_RES` sont toujours appliqués sur l'affichage créé en dernier (par défaut). Si vous passez `NULL`

en tant que paramètre **disp** de fonctions associées à l’affichage, l’affichage par défaut sera utilisé. P.ex. `lv_disp_trig_activity(NULL)` déclenchera une activité utilisateur sur l’affichage par défaut (voir ci-dessous dans *Inactivité*).

Affichage miroir

Pour refléter l’image de l’affichage sur un autre affichage, vous n’avez pas besoin d’utiliser le support multi-affichages. Transférez simplement le tampon reçu dans `drv.flush_cb` vers un autre affichage.

Division d’image

Vous pouvez créer un affichage plus grand à partir de plus petits. Voici comment faire :

1. Définissez la résolution des affichages à celle désirée
2. Dans `drv.flush_cb`, tronquez et modifiez le paramètre **area** pour chaque affichage
3. Envoyez le contenu du tampon à chaque affichage pour la zone tronquée

Ecrans

Chaque affichage possède son propre ensemble d’ [Ecrans](#) et d’objets à l’écran.

Les écrans peuvent être considérés comme les conteneurs de plus haut niveau sans parent. La taille de l’écran est toujours égale à celle de son affichage et sa position est (0, 0). Par conséquent, les coordonnées des écrans ne peuvent pas être modifiées, c-à-d `lv_obj_set_pos()`, `lv_obj_set_size()` ou les fonctions similaires ne peuvent pas être utilisées sur les écrans.

Un écran peut être créé à partir de n’importe quel type d’objet, mais les deux types les plus courants sont *Objet de base* et *Image* (pour créer un fond d’écran).

Pour créer un écran, utilisez `lv_obj_t * scr = lv_<type>_create(NULL, copy)`. `copy` peut être un autre écran à copier.

Pour charger un écran, utilisez `lv_scr_load(scr)`. Pour obtenir l’écran actif utilisez `lv_scr_act()`. Ces fonctions agissent sur l’affichage par défaut. Pour spécifier l’affichage, utilisez `lv_disp_get_scr_act(disp)` et `lv_disp_load_scr(disp, scr)`.

Les écrans peuvent être supprimés avec `lv_obj_del(scr)` mais assurez-vous de ne pas supprimer l’écran actuellement chargé.

Ecran opaque

Habituellement, l’opacité de l’écran est `LV_OPA_COVER` afin de fournir un fond solide et totalement couvrant pour les enfants. Cependant, dans certains cas particuliers, vous voudrez peut-être un écran transparent. Par exemple, si vous avez un lecteur vidéo qui restitue les images vidéo sur un calque mais sur un autre calque, vous souhaitez créer un menu OSD (au-dessus de la vidéo) à l’aide de LittlevGL. Dans ce cas, le style de l’écran doit avoir `body.opa = LV_OPA_TRANSP` ou `image.opa = LV_OPA_TRANSP` (ou d’autres valeurs `LV_OPA_...`) pour rendre l’écran transparent. Pour gérer correctement l’opacité des écrans, `LV_COLOR_SCREEN_TRANSP` doit être activé. Notez que cela ne fonctionne qu’avec `LV_COLOR_DEPTH = 32`. Le canal alpha des couleurs 32 bits sera 0 lorsqu’il n’y a pas d’objets et 255 en cas d’objets solides.

Fonctionnalités des affichages

Inactivité

L'inactivité de l'utilisateur est surveillée pour chaque affichage. Chaque utilisation d'un *Périphérique d'entrée* (s'il est associé à l'affichage) compte comme une activité. Pour obtenir le temps écoulé depuis la dernière activité, utilisez `lv_disp_get_inactive_time(dispatch)`. Si `NULL` est passé comme paramètre, le temps d'inactivité le plus petit de tous les affichages sera retourné.

Vous pouvez simuler une activité en utilisant `lv_disp_trig_activity(dispatch)`. Si `dispatch` est `NULL`, l'affichage par défaut sera utilisé.

Couleurs

Le module couleur gère toutes les fonctions liées aux couleurs telles que la modification de la profondeur de couleur, la création de couleurs à partir de code hexadécimal, la conversion entre les profondeurs de couleur, le mélange de couleurs, etc.

Les types de variable suivants sont définis par le module couleur :

- **lv_color1_t** Enregistre une couleur monochrome. Pour la compatibilité, il possède également des champs R, G, B mais ils sont tous identiques (1 octet)
- **lv_color8_t** Une structure pour enregistrer R (3 bits), G (3 bits), B (2 bits) composantes d'une couleur 8 bits (1 octet)
- **lv_color16_t** Une structure pour enregistrer R (5 bits), G (6 bits), B (5 bits) composantes d'une couleur 16 bits (2 octets)
- **lv_color32_t** Une structure pour enregistrer R (8 bits), G (8 bits), B (8 bits) composantes d'une couleur 24 bits (4 octets)
- **lv_color_t** Equivaut à `lv_color1/8/16/24_t` selon le paramètre de profondeur de couleur
- **lv_color_int_t** `uint8_t`, `uint16_t` ou `uint32_t` selon le paramètre de profondeur de couleur. Utilisé pour construire des tableaux de couleurs à partir de valeurs numériques.
- **lv_opa_t** Un simple type `uint8_t` pour définir l'opacité.

Les types `lv_color_t`, `lv_color1_t`, `lv_color8_t`, `lv_color16_t` et `lv_color32_t` ont quatre champs :

- **ch.red** canal rouge
- **ch.green** canal vert
- **ch.blue** canal bleu
- **full** rouge + vert + bleu en une seule valeur

Vous pouvez définir la profondeur de couleur actuelle dans `lv_conf.h` en définissant la valeur `LV_COLOR_DEPTH` sur 1 (monochrome), 8, 16 ou 32.

Conversion de couleur

Vous pouvez convertir une couleur de la profondeur de couleur actuelle en une autre. Les fonctions de conversion retournent un nombre, vous devez donc utiliser le champ **full** :

```
lv_color_t c;
c.red   = 0x38;
c.green = 0x70;
c.blue  = 0xCC;

lv_color1_t c1;
c1.full = lv_color_to1(c);      /* Retourne 1 pour les couleurs claires, 0 pour les
↪couleurs sombres */

lv_color8_t c8;
c8.full = lv_color_to8(c);      /* Donne un nombre de 8 bits avec la couleur
↪convertie */

lv_color16_t c16;
c16.full = lv_color_to16(c); /* Donne un nombre de 16 bits avec la couleur convertie
↪*/

lv_color32_t c32;
c32.full = lv_color_to32(c);    /* Donne un nombre de 32 bits avec la couleur
↪convertie */
```

Permutation 16 bits

Vous pouvez définir **LV_COLOR_16_SWAP** dans *lv_conf.h* pour permuter les octets de *couleurs RGB565*. C'est utile si vous transférez les couleurs 16 bits via une interface orientée octet comme SPI. Comme les nombres de 16 bits sont stockés au format Little Endian (octet le moins significatif à l'adresse la plus basse), l'interface envoie d'abord l'octet le moins significatif. Cependant, les écrans ont généralement besoin de l'octet le plus significatif en premier. Une non-concordance dans l'ordre des octets entraînera une altération des couleurs.

Créer et mélanger les couleurs

Vous pouvez créer des couleurs avec la profondeur de couleur actuelle à l'aide de la macro **LV_COLOR_MAKE**. Il faut 3 arguments (rouge, vert, bleu) sous forme de nombres de 8 bits. Par exemple, pour créer une couleur rouge clair : **my_color = COLOR_MAKE(0xFF, 0x80, 0x80)**.

Les couleurs peuvent aussi être créées à partir de codes hexadécimaux : **my_color = lv_color_hex(0x288ACF)** ou **my_color = lv_color_hex(0x28C)**.

Le mélange de deux couleurs est possible avec **mixed_color = lv_color_mix(color1, color2, ratio)**. Le ratio peut être 0..255. 0 donne entièrement color2, 255 donne entièrement color1.

Les couleurs peuvent aussi être créées avec l'espace HSV en utilisant **lv_color_hsv_to_rgb(hue, saturation, value)**. **hue** devrait être dans la plage 0..360, **saturation** et **value** dans la plage 0..100.

Opacité

Pour décrire l'opacité, le type **lv_opa_t** est créé comme équivalent d'un **uint8_t**. Quelques définitions sont également introduites :


- **LV_OPA_TRANSP** Valeur : 0, l'opacité rend la couleur totalement transparente.
- **LV_OPA_10** Valeur : 25, signifie que la couleur est un peu couvrante

- **LV_OPA_20 ... OPA_80** viennent logiquement
- **LV_OPA_90** Valeur : 229 signifie que la couleur est presque entièrement couvrante
- **LV_OPA_COVER** Valeur : 255, signifie que la couleur couvre entièrement

Vous pouvez également utiliser les définitions **LV_OPA_*** dans **lv_color_mix()** en tant que *ratio*.

Couleurs intégrées

Le module couleur définit les couleurs les plus basiques :

-  **#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

ainsi que **LV_COLOR_WHITE**.

API

Affichage

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)
```

Couleurs

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
```

Polices

Dans LittlevGL, les polices sont des collections d'images matricielles et d'autres informations nécessaires au rendu des images des lettres (glyphes). Une police est stockée dans une variable `lv_font_t` et peut être affectée au champ `text.font` du style. Par exemple :

```
my_style.text.font = &lv_font_roboto_28; /* Définit une police plus grande */
```

Les polices ont une propriété **bpp (Bits Par Pixel)**. Il indique combien de bits sont utilisés pour décrire un pixel dans la police. La valeur enregistrée pour un pixel détermine l'opacité du pixel. De cette façon, avec un *bpp* plus important, les bords de la lettre peuvent être lissés. Les valeurs de *bpp* possibles sont 1, 2, 4 et 8 (une valeur plus élevée signifie une meilleure qualité).

Le `* bpp *` affecte également la taille de la mémoire requise pour utiliser la police. P.ex. *bpp* = 4 rend la police environ 4 fois plus volumineuse que *bpp* = 1.

Support Unicode

LittlevGL prend en charge les caractères Unicode codés **UTF-8**. Vous devez configurer votre éditeur pour qu'il enregistre votre code/texte au format UTF-8 (généralement par défaut) et assurez-vous que `LV_TXT_ENC` est défini sur `LV_TXT_ENC_UTF8` dans `lv_conf.h` (ceci est la valeur par défaut).

Pour le vérifier, essayez

```
lv_obj_t * label1 = lv_label_create(lv_scr_act(), NULL);
lv_label_set_text(label1, LV_SYMBOL_OK);
```

Si tout fonctionne bien, un caractère ✓ devrait être affiché.

Polices intégrées



















































Il existe plusieurs polices intégrées de différentes tailles qui peuvent être activées dans `lv_conf.h` par la définition de constantes symboliques `LV_FONT_...` :

- `LV_FONT_ROBOTO_12` 12 px
- `LV_FONT_ROBOTO_16` 16 px
- `LV_FONT_ROBOTO_22` 22 px
- `LV_FONT_ROBOTO_28` 28 px

Les polices intégrées sont des **variables globales** portant des noms tels que `lv_font_roboto_16` pour une police de 16 pixels de haut. Pour les utiliser dans un style, ajoutez simplement un pointeur sur une variable de police, comme indiqué ci-dessus.

Les polices intégrées ont *bpp* = 4, contiennent les caractères ASCII et utilisent la police [Roboto](#).

Outre la plage ASCII, les symboles suivants sont également ajoutés aux polices intégrées à partir de la police [FontAwesome](#).

	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

Les symboles peuvent être utilisés ainsi :

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

Ou avec des chaînes :

```
lv_label_set_text(my_label, LV_SYMBOL_OK "Apply");
```

Ou plusieurs symboles ensemble :

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

Ajouter une nouvelle police

Il y a plusieurs manières d'ajouter une nouvelle police à votre projet :

1. Le moyen le plus simple consiste à utiliser le [Convertisseur de polices en ligne](#). Il suffit de définir les paramètres, de cliquer sur le bouton *Convert*, de copier la police dans votre projet et de l'utiliser.
2. Utilisez le [Convertisseur de polices hors ligne] (https://github.com/littlevgl/lv_font_conv) (nécessite l'installation de Node.js).
3. Si vous voulez créer quelque chose comme les polices intégrées (police Roboto et symboles) mais de tailles et/ou de plages différentes, vous pouvez utiliser le script `built_in_font_gen.py` du dossier `lvgl/scripts/built_in_font`. Il nécessite que Python et `lv_font_conv` soient installés.

Pour déclarer la police dans un fichier, utilisez `LV_FONT_DECLARE(my_font_name)`.

Pour rendre les polices disponibles globalement, ajoutez-les à `LV_FONT_CUSTOM_DECLARE` dans `lv_conf.h`.

Ajouter de nouveaux symboles

Les symboles intégrés sont créés à partir de la police [FontAwesome](#). Pour ajouter de nouveaux symboles à partir de la police FontAwesome, procédez comme suit :

1. Recherchez un symbole sur [<https://fontawesome.com>] (<https://fontawesome.com>). Par exemple le symbole [USB](#)
2. Ouvrez le [Convertisseur de polices en ligne](#), ajoutez [[FontAwesome.ttf](https://fontawesome.com)] (<https://fontawesome.com>) et ajoutez l'ID Unicode du symbole au champ plage. P.ex. `0xf287` pour le symbole USB. Plusieurs symboles peuvent être énumérés séparés par `,.`
3. Convertissez la police et copiez-la dans votre projet.
4. Convertissez la valeur Unicode en UTF8. Vous pouvez le faire, par exemple, sur ce [site] (<http://www.ltg.ed.ac.uk/~richard/utf-8.cgi?input=f287&mode=hex>). Pour `0xf287`, les *octets hexadécimaux UTF-8* sont `EF 8A 87`.
5. Créez une **définition de constante symbolique** à partir des valeurs UTF8 : `#define MY_USB_SYMBOL "\xEF \x8A \x87"`
6. Utilisez le symbole comme les symboles intégrés. `lv_label_set_text (label, MY_USB_SYMBOL)`

Ajouter un nouveau moteur de polices

L'interface de police de LittlevGL est conçue pour être très flexible. Vous n'avez pas besoin d'utiliser le moteur de polices interne de LittlevGL, mais vous pouvez ajouter le vôtre. Par exemple, utilisez [FreeType] (<https://www.freetype.org/>) pour restituer les glyphes en temps réel à partir de polices TTF ou utilisez une mémoire flash externe pour sauvegarder les images matricielles de la police et les lire lorsque la librairie en a besoin.

Pour ce faire, une variable `lv_font_t` personnalisée doit être créée :

```
/* Décrit les propriétés d'une police */
lv_font_t my_font;
my_font.get_glyph_dsc = my_get_glyph_dsc_cb;          /* Définit une fonction de rappel ↵
↳pour obtenir des informations sur les glyphes */
my_font.get_glyph_bitmap = my_get_glyph_bitmap_cb;    /* Définit une fonction de rappel ↵
↳pour obtenir l'image matricielle d'un glyphe */
my_font.line_height = height;                         /* La hauteur réelle de la ligne ↵
↳où le texte s'inscrit */
my_font.base_line = base_line;                       /* La ligne de base mesurée à ↵
↳partir du haut de la ligne */
my_font.dsc = something_required;                     /* Enregistre ici toutes les ↵
↳données spécifiques à l'implémentation */
my_font.user_data = user_data;                       /* Éventuellement des données ↵
↳utilisateur supplémentaires */

...

/* Obtient des informations sur le glyphe de `unicode_letter` dans la police `font`.
 * Enregistre le résultat dans `dsc_out`.
 * La lettre suivante (`unicode_letter_next`) peut être utilisée pour calculer la ↵
↳largeur requise par ce glyphe (crénage)
 */
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)
{
    /* Votre code ici */

    /* Enregistre le résultat.
     * Par exemple ...
     */
    dsc_out->adv_w = 12;                               /* Espace horizontal requis par le glyphe en [px] */
    dsc_out->box_h = 8;                                 /* Hauteur de l'image en [px] */
    dsc_out->box_w = 6;                                 /* Largeur de l'image en [px] */
    dsc_out->ofs_x = 0;                                 /* Déplacement X de l'image en [px] */
    dsc_out->ofs_y = 3;                                 /* Déplacement Y de l'image mesuré depuis la ligne de ↵
↳base */
    dsc_out->bpp = 2;                                   /* Bits par pixel : 1/2/4/8 */

    return true;                                       /* true : glyphe trouvé; false : glyphe non trouvé */
}

/* Obtient l'image matricielle de `unicode_letter` à partir de `font`.*/
const uint8_t * my_get_glyph_bitmap_cb(const lv_font_t * font, uint32_t unicode_
↳letter)
{
    /* Votre code ici */
```

(continues on next page)

(continued from previous page)

```

/* L'image matricielle doit être un flux continu de bits où
 * chaque pixel est représenté par `bpp` bits */

return bitmap;    /* Ou NULL si non trouvé */
}

```

Images

Une image peut être un fichier ou une variable qui enregistre l'image elle-même et des métadonnées.

Enregistrer des images

Vous pouvez enregistrer des images à deux endroits

- en tant que variable en mémoire interne (MEV ou MEM)
- en tant que fichier

Variables

Les images enregistrées dans une variable ont le type `lv_img_dsc_t` avec les champs suivants :

- **header**
 - *cf* Format de couleur. Voir *ci-dessous*
 - *w* largeur en pixels (≤ 2048)
 - *h* hauteur en pixels (≤ 2048)
 - *always zero* 3 bits qui doivent toujours être à zéro
 - *reserved* réservé pour une utilisation future
- **datapointeur** sur un tableau où l'image elle-même est enregistrée
- **data__size** longueur de **data** en octets

Fichiers

Pour traiter les fichiers, vous devez ajouter un *lecteur* à LittlevGL. En bref, un lecteur est une collection de fonctions (*open*, *read*, *close*, etc.) enregistrées dans LittlevGL pour effectuer des opérations sur les fichiers. Vous pouvez ajouter une interface à un système de fichiers standard (FAT32 sur une carte SD) ou créer votre propre système de fichiers pour lire des données à partir d'une mémoire Flash SPI. Dans tous les cas, un lecteur n'est qu'une abstraction pour lire et/ou écrire des données dans une mémoire. Voir la section *Système de fichiers* pour en apprendre plus.

Formats de couleur

Divers formats de couleur intégrés sont pris en charge:

- **LV_IMG_CF_TRUE_COLOR** Enregistre simplement les couleurs RVB

- **LV_IMG_CF_TRUE_COLOR_ALPHA** Enregistre les couleurs RVB mais ajoute également un octet Alpha pour chaque pixel
- **LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED** Enregistre les couleurs RVB, mais si un pixel prend la valeur **LV_COLOR_TRANSP** (défini dans *lv_conf.h*), le pixel sera transparent
- **LV_IMG_CF_INDEXED_1/2/4/8BIT** Utilise une palette avec 2, 4, 16 ou 256 couleurs et enregistre chaque pixel sur 1, 2, 4 ou 8 bits
- **LV_IMG_CF_ALPHA_1/2/4/8BIT** Enregistre uniquement la valeur Alpha sur 1, 2, 4 ou 8 bits. Dessine les pixels avec la couleur **style.image.color** et l'opacité définie.

Les octets des images *Couleurs vraies* 32 bits sont enregistrés dans l'ordre suivant

- Byte 0: Bleu
- Byte 1: Vert
- Byte 2: Rouge
- Byte 3: Alpha

Pour une profondeur de couleur de 16 bits

- Byte 0: Vert 3 bits de poids faible, Bleu 5 bits
- Byte 1: Rouge 5 bits, Vert 3 bits de poids fort
- Byte 2: octet Alpha (seulement avec **LV_IMG_CF_TRUE_COLOR_ALPHA**)

Pour une profondeur de couleur de 8 bits

- Byte 0: Rouge 3 bits, Vert 3 bits, Bleu 2 bits
- Byte 2: octet Alpha (seulement avec **LV_IMG_CF_TRUE_COLOR_ALPHA**)

Vous pouvez enregistrer des images au format *brut* pour indiquer qu'il ne s'agit pas d'un format de couleur intégré et qu'un *Décodeur d'images* doit être utilisé pour décoder l'image.

- **LV_IMG_CF_RAW** Une image brute, p.ex. une image PNG ou JPG
- **LV_IMG_CF_RAW_ALPHA** Indique que l'image a un canal alpha et qu'un octet Alpha est ajouté pour chaque pixel
- **LV_IMG_CF_RAW_CHROME_KEYED** Indique que l'image a une couleur transparente comme indiqué par **LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED** ci-dessus.

Ajouter et utiliser des images

Vous pouvez ajouter des images à LittlevGL de deux manières :

- utiliser le convertisseur en ligne
- créer manuellement des images

Convertisseur en ligne

Le convertisseur d'image en ligne est disponible [ici](#).

Vous devez sélectionner une image *BMP*, *PNG* ou *JPG*, lui donner un nom, sélectionner le *Format de couleur*, sélectionner le type (fichier ou variable) et cliquer sur le bouton *Convert*. Le fichier de résultat est téléchargé.

Dans le cas de tableaux C (variables), les données de l'image sont incluses pour toutes les profondeurs de couleurs (1, 8, 16 ou 32) et les données à utiliser seront sélectionnées lors de la compilation en fonction de `LV_COLOR_DEPTH` dans `lv_conf.h`.

Dans le cas de fichiers, vous devez indiquer le format de couleur souhaité

- Binary RGB332 pour une profondeur de couleur de 8 bits
- Binary RGB565 pour une profondeur de couleur de 16 bits
- Binary RGB565 pour une profondeur de couleur de 16 bits (les deux octets sont permutés)
- Binary RGB888 pour une profondeur de couleur de 32 bits

Créer une image manuellement

Si vous réalisez une image au moment de l'exécution, vous pouvez créer une variable d'image pour l'afficher. Par exemple :

```
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,          /* Définit le format de couleur */
    .data = my_img_data,
};
```

Si le format de couleur est `LV_IMG_CF_TRUE_COLOR_ALPHA`, vous pouvez définir `data_size` comme `80 * 60 * LV_IMG_PX_SIZE_ALPHA_BYTE`.

Une autre option pour créer une image au moment de l'exécution consiste à utiliser l'objet *Canvas*.

Utiliser des images

Le moyen le plus simple d'utiliser une image dans LittlevGL consiste à l'afficher avec un objet *Image*:

```
lv_obj_t * icon = lv_img_create(lv_scr_act(), NULL);

/* A partir d'une variable */
lv_img_set_src(icon, &my_icon_dsc);

/* A partir d'un fichier */
lv_img_set_src(icon, "S:my_icon.bin");
```

Si l'image a été convertie avec le convertisseur en ligne, vous devez utiliser `LV_IMG_DECLARE(my_icon_dsc)` pour déclarer l'icône dans le fichier où vous souhaitez l'utiliser.

Décodeur d'images

Comme vous pouvez le voir dans la section [Formats de couleur] (`#formats-de-couleur`), LittlevGL prend en charge plusieurs formats d'image intégrés. Cependant, il ne supporte pas par exemple les formats PNG ou JPG. Pour gérer les formats d'image non intégrés, vous devez utiliser des librairie externes et les attacher à LittlevGL via l'interface *décodeur d'images*.

Le décodeur d'image comprend 4 fonctions :

- **info** obtient des informations de base sur l'image (largeur, hauteur et format de couleur)
- **open** ouvre l'image : enregistre l'image décodée. **NULL** indique que l'image peut être lue ligne par ligne
- **read** si *open* ne traite pas complètement l'image, cette fonction devrait retourner les données décodées (maximum 1 ligne) à partir d'une position donnée.
- **close** ferme l'image ouverte, libère les ressources allouées.

Vous pouvez ajouter n'importe quel nombre de décodeurs d'image. Quand une image doit être dessinée, la librairie essaiera tout les décodeurs d'images enregistrés jusqu'à en trouver un capable d'ouvrir l'image, c-à-d de manipuler le format.

Les formats `LV_IMG_CF_TRUE_COLOR...`, `LV_IMG_INDEXED...` and `LV_IMG_ALPHA...` sont connus par le décodeur intégré.

Formats d'image personnalisés

Le moyen le plus simple de créer une image personnalisée consiste à utiliser le convertisseur d'image en ligne et à définir le format **Raw**, **Raw with alpha** ou **Raw with chrome keyed**. Le convertisseur prendra tous les octets de l'image sélectionnée et les écrira en tant que données d'image. `header.cf` sera respectivement `LV_IMG_CF_RAW`, `LV_IMG_CF_RAW_ALPHA` ou `LV_IMG_CF_RAW_CHROME_KEYED`. Vous devez choisir le bon format en fonction de vos besoins : image normale, utilisation de canal alpha ou de couleur transparente.

Après décodage, les formats *bruts* sont considérés comme des *couleurs vraies*. En d'autres termes, le décodeur d'image doit décoder les images *brutes* en *vraies couleurs* conformément au format décrit dans la section *Formats de couleur*.

Si vous voulez créer une image vraiment personnalisée, vous devez utiliser les formats de couleur `LV_IMG_CF_USER_ENCODED_0..7`. Cependant, la librairie peut dessiner les images uniquement au format *couleurs vraies* (ou *brut*, mais finalement, elles sont supposées être au format *couleurs vraies*). Donc, les formats `LV_IMG_CF_USER_ENCODED...` ne sont pas connus de la librairie. Ils doivent donc être décodés dans l'un des formats connus de la section *Formats de couleur*. Il est possible de décoder d'abord l'image dans un format de couleur non vraie, par exemple `LV_IMG_INDEXED_4BITS`, puis d'appeler les fonctions du décodeur intégré pour la convertir en *couleurs vraies*.

Pour les formats *encodés par l'utilisateur*, le format de couleur (`dsc-> header.cf`) doit être modifié en conséquence, dans la fonction `open`.

Enregistrer un décodeur d'image

Par exemple, si vous voulez que LittlevGL "comprenne" les images PNG, vous devez créer un nouveau décodeur d'images et définir certaines fonctions pour ouvrir/fermer les fichiers PNG. Voici à quoi cela devrait ressembler :

```
/* Crée un nouveau décodeur et enregistre les fonctions */
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);

/**
```

(continues on next page)

(continued from previous page)

```

* Obtient les informations sur une image PNG
* @param decoder pointeur vers le décodeur auquel cette fonction appartient
* @param src peut être un nom de fichier ou un pointeur sur un tableau C
* @param header enregistre l'information ici
* @return LV_RES_OK : pas d'erreur ; LV_RES_INV : impossible d'obtenir l'information
*/
static lv_res_t decoder_info(lv_img_decoder_t * decoder, const void * src, lv_img_
↳ header_t * header)
{
    /* Vérifie si le type `src` est connu du décodeur */
    if(is_png(src) == false) return LV_RES_INV;

    ...

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

/**
* Ouvre une image PNG et retourne l'image décodée
* @param decoder pointeur vers le décodeur auquel cette fonction appartient
* @param dsc pointeur sur le descripteur de cette session de décodage
* @return LV_RES_OK : pas d'erreur ; LV_RES_INV : impossible d'obtenir l'information
*/
static lv_res_t decoder_open(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
{
    /* Vérifie si le type `src` est connu du décodeur */
    if(is_png(src) == false) return LV_RES_INV;

    /* Décode et enregistre l'image. Si `dsc->img_data` est `NULL`, la fonction `read_
↳ line` sera appelée pour obtenir les données de l'image ligne par ligne */
    dsc->img_data = my_png_decoder(src);

    /* Change le format de couleur si nécessaire. Pour le PNG, généralement un format
↳ 'brut' convient */
    dsc->header.cf = LV_IMG_CF_...

    /* Appelle une fonction de décodeur intégré si nécessaire. Ce n'est pas nécessaire,
↳ si `my_png_decoder` a décodé l'image au format couleurs vraies. */
    lv_res_t res = lv_img_decoder_built_in_open(decoder, dsc);

    return res;
}

/**
* Décode `len` pixels à partir des coordonnées fournies `x`, `y` et enregistre-les
↳ dans `buf`.
* Requis uniquement si la fonction "open" ne peut pas décoder l'intégralité du
↳ tableau de pixels (dsc->img_data == NULL).
* @param decoder pointeur vers le décodeur associé à la fonction
* @param dsc pointeur vers le descripteur de décodeur
* @param x coordonnée x de début
* @param y coordonnée y de début
* @param len nombre de pixels à décoder

```

(continues on next page)

(continued from previous page)

```

* @param buf un tampon pour enregistrer les pixels décodés
* @return LV_RES_OK : ok ; LV_RES_INV : échec
*/
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)
{
    /* Avec PNG, ce n'est généralement pas nécessaire */

    /* Copie `len` pixels à partir des coordonnées `x` et `y` au format couleurs
↳ vraies dans `buf` */
}

/**
* Libère les ressources allouées
* @param decoder pointeur vers le décodeur auquel cette fonction appartient
* @param dsc pointeur sur le descripteur de cette session de décodage
*/
static void decoder_close(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t * dsc)
{
    /* Libère toutes les données allouées */

    /* Appelle la fonction intégrée de fermeture si les fonctions intégrées open/read
↳ line ont été utilisées */
    lv_img_decoder_built_in_close(decoder, dsc);
}

```

Donc en résumé :

- Dans `decoder_info`, vous devez collecter les informations de base sur l'image et les mémoriser dans `header`.
- Dans `decoder_open`, vous devez essayer d'ouvrir la source de l'image indiquée par `dsc->src`. Son type est déjà dans `dsc->src_type == LV_IMG_SRC_FILE/VARIABLE`. Si le format/type n'est pas pris en charge par le décodeur, retournez `LV_RES_INV`. Autrement, si vous pouvez ouvrir l'image, un pointeur sur l'image *couleurs vraies* décodée doit être défini dans `dsc->img_data`. Si le format est géré mais que vous ne voulez pas décoder l'ensemble de l'image (p.ex. pas de mémoire), définissez `dsc->img_data = NULL` pour appeler `read_line` afin d'obtenir les pixels.
- Dans `decoder_close`, vous devez libérer toutes les ressources allouées.
- `decoder_read` est optionnel. Le décodage de l'ensemble de l'image nécessite de la mémoire et des calculs supplémentaires. Cependant, si vous pouvez décoder une ligne de l'image sans décoder toute l'image, vous pouvez économiser de la mémoire et du temps. Pour indiquer que la fonction *line read* doit être utilisée, définissez `dsc->img_data = NULL` dans la fonction `open`.

Utiliser manuellement un décodeur d'image

LittlevGL utilisera automatiquement les décodeurs d'images enregistrés, mais vous pouvez également les utiliser manuellement. Créez une variable `lv_img_decoder_dsc_t` pour décrire la session de décodage et appelez `lv_img_decoder_open()` et `lv_img_decoder_close()` .

```
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) {
    /* Faites quelque chose avec `dsc->img_data` */
    lv_img_decoder_close(&dsc);
}
```

Mise en cache des images

Parfois, il faut beaucoup de temps pour ouvrir une image. Continuellement décoder une image PNG ou charger des images à partir d'une mémoire externe lente serait inefficace. LittlevGL place donc en cache un certain nombre d'images. La mise en cache signifie que certaines images resteront ouvertes, ce qui permet à LittlevGL d'y accéder rapidement à partir de **dsc->img_data** au lieu de les décoder à nouveau.

Taille du cache

Le nombre d'entrées du cache peut être défini par **LV_IMG_CACHE_DEF_SIZE** dans *lv_conf.h*. La valeur par défaut est 1, de sorte que seule la dernière image utilisée reste ouverte. La taille du cache peut être modifiée en cours d'exécution avec **lv_img_cache_set_size(entry_num)**

Valeur des images

Si vous utilisez un grand nombre d'images, LittlevGL ne peut pas toutes les mettre en cache. Au lieu de cela, si une nouvelle image doit être ouverte mais qu'il n'y a pas de place dans le cache, la bibliothèque supprimera une image pour libérer de la place. Pour décider quelle image supprimer, LittlevGL a mesuré combien de temps il a fallu pour ouvrir chaque image. Les images dont l'ouverture prend le plus de temps sont favorisées et LittlevGL essaie de les conserver en cache le plus longtemps. Vous pouvez définir manuellement la valeur *temps d'ouverture* dans la fonction d'ouverture du décodeur dans **dsc->time_to_open = temps en ms** pour attribuer une valeur supérieure ou inférieure à l'image (laissez inchangé pour laisser LittlevGL le gérer).

Chaque entrée de cache a une valeur "*durée de vie*". À chaque ouverture d'image par la mémoire cache, la *durée de vie* de toutes les entrées est réduite pour les rendre plus anciennes. Lorsqu'une image en cache est utilisée, sa *durée de vie* est augmentée de la valeur de *temps d'ouverture* pour la rendre plus importante.

S'il n'y a plus d'espace dans la mémoire cache, l'entrée avec la plus petite durée de vie sera supprimée.

Utilisation de la mémoire

Notez que l'image en cache peut consommer de la mémoire en permanence. Par exemple, si 3 images PNG sont mises en cache, elles consomment de la mémoire pendant le temps où elle sont ouvertes. Par conséquent, il incombe à l'utilisateur de s'assurer qu'il y a assez de MEV pour mettre en cache de grandes images en même temps.

Nettoyer le cache

Supposons que vous ayez chargé une image PNG dans une variable **lv_img_dsc_t my_png** et que vous l'utilisiez dans un objet **lv_img**. Si l'image est déjà mise en cache et que vous modifiez **my_png->**

`data`, vous devez avertir LittlevGL de mettre en cache l'image à nouveau. Pour ce faire, utilisez `lv_img_cache_invalidate_src(&my_png)`. Si `NULL` est passé en paramètre, tout le cache sera nettoyé.

API

Décodeur d'image

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

LV_IMG_CF_RESERVED_15
Reserved for further use.

LV_IMG_CF_RESERVED_16
Reserved for further use.

LV_IMG_CF_RESERVED_17
Reserved for further use.

LV_IMG_CF_RESERVED_18
Reserved for further use.

LV_IMG_CF_RESERVED_19
Reserved for further use.

LV_IMG_CF_RESERVED_20
Reserved for further use.

LV_IMG_CF_RESERVED_21
Reserved for further use.

LV_IMG_CF_RESERVED_22
Reserved for further use.

LV_IMG_CF_RESERVED_23
Reserved for further use.

LV_IMG_CF_USER_ENCODED_0
User holder encoding format.

LV_IMG_CF_USER_ENCODED_1
User holder encoding format.

LV_IMG_CF_USER_ENCODED_2
User holder encoding format.

LV_IMG_CF_USER_ENCODED_3
User holder encoding format.

LV_IMG_CF_USER_ENCODED_4
User holder encoding format.

LV_IMG_CF_USER_ENCODED_5
User holder encoding format.

LV_IMG_CF_USER_ENCODED_6
User holder encoding format.

LV_IMG_CF_USER_ENCODED_7
User holder encoding format.

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

lv_res_t **lv_img_decoder_built_in_info**(*lv_img_decoder_t* *decoder, const void *src, *lv_img_header_t* *header)

Get info about a built-in image

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

Parameters

- **decoder**: the decoder where this function belongs
- **src**: the image source: pointer to an *lv_img_dsc_t* variable, a file path or a symbol
- **header**: store the image data here

lv_res_t **lv_img_decoder_built_in_open**(*lv_img_decoder_t* *decoder, *lv_img_decoder_dsc_t* *dsc)

Open a built in image

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

Parameters

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

```
lv_res_t lv_img_decoder_built_in_read_line(lv_img_decoder_t *decoder,
                                           lv_img_decoder_dsc_t *dsc, lv_coord_t
                                           x, lv_coord_t y, lv_coord_t len, uint8_t
                                           *buf)
```

Decode **len** pixels starting from the given **x**, **y** coordinates and store them in **buf**. Required only if the “open” function can’t return with the whole decoded pixel array.

Return LV_RES_OK: ok; LV_RES_INV: failed

Parameters

- **decoder**: pointer to the decoder the function associated with
- **dsc**: pointer to decoder descriptor
- **x**: start x coordinate
- **y**: start y coordinate
- **len**: number of pixels to decode
- **buf**: a buffer to store the decoded pixels

```
void lv_img_decoder_built_in_close(lv_img_decoder_t *decoder, lv_img_decoder_dsc_t
                                  *dsc)
```

Close the pending decoding. Free resources etc.

Parameters

- **decoder**: pointer to the decoder the function associated with
- **dsc**: pointer to decoder descriptor

struct lv_img_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

```

Cache d'images

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

Système de fichiers

LittlevGL a un module d'abstraction de système de fichiers qui permet d'attacher tout type de système de fichiers. Les systèmes de fichiers sont identifiés par une lettre. Par exemple, si la carte SD est associée à la lettre 'S', un fichier peut être localisé par "S:path/to/file.txt".

Ajouter un pilote

Pour ajouter un pilote, un `lv_fs_drv_t` doit être initialisé comme ceci :

```
lv_fs_drv_t drv;
lv_fs_drv_init(&drv);                                /* Initialisation de base */

drv.letter = 'S';                                     /* Une lettre majuscule pour identifier le_
↳lecteur */
drv.file_size = sizeof(my_file_object);              /* Taille requise pour enregistrer un objet_
↳de fichier */
drv.rddir_size = sizeof(my_dir_object);              /* Taille requise pour enregistrer un objet_
↳répertoire (utilisé par dir_open/close/read) */
```

(continues on next page)

(continued from previous page)

```

drv.ready_cb = my_ready_cb;          /* Fonction de rappel pour indiquer si le
↳lecteur est prêt à être utilisé */
drv.open_cb = my_open_cb;            /* Fonction de rappel pour ouvrir un
↳fichier */
drv.close_cb = my_close_cb;          /* Fonction de rappel pour fermer un
↳fichier */
drv.read_cb = my_read_cb;            /* Fonction de rappel pour lire un fichier
↳ */
drv.write_cb = my_write_cb;          /* Fonction de rappel pour écrire un
↳fichier */
drv.seek_cb = my_seek_cb;            /* Fonction de rappel pour se déplacer dans
↳un fichier (déplacer le curseur) */
drv.tell_cb = my_tell_cb;            /* Fonction de rappel pour donner la
↳position du curseur */
drv.trunc_cb = my_trunc_cb;          /* Fonction de rappel pour supprimer un
↳fichier */
drv.size_cb = my_size_cb;            /* Fonction de rappel pour donner la taille
↳d'un fichier */
drv.rename_cb = my_size_cb;          /* Fonction de rappel pour renommer un
↳fichier */

drv.dir_open_cb = my_dir_open_cb;    /* Fonction de rappel pour ouvrir un
↳répertoire et lire son contenu */
drv.dir_read_cb = my_dir_read_cb;    /* Fonction de rappel pour lire le contenu d
↳un répertoire */
drv.dir_close_cb = my_dir_close_cb;  /* Fonction de rappel pour fermer un
↳répertoire */

drv.free_space_cb = my_size_cb;       /* Fonction de rappel pour donner l'espace
↳libre d'un lecteur */

drv.user_data = my_user_data;        /* Toute donnée personnalisée si nécessaire
↳ */

lv_fs_drv_register(&drv);            /* Finalement enregistre le lecteur */

```

N'importe laquelle des fonctions de rappel peut être **NULL** pour indiquer que l'opération n'est pas prise en charge.

Si vous utilisez `lv_fs_open(&file, "S:/folder/file.txt", LV_FS_MODE_WR)` LittlevGL effectuera les opérations suivantes

1. vérifie s'il y a un lecteur avec la lettre 'S'
2. vérifie si `open_cb` est implémentée (pas **NULL**)
3. appelle `open_cb` avec le chemin "folder/file.txt".

Exemple d'utilisation

L'exemple ci-dessous montre comment lire à partir d'un fichier :

```

lv_fs_file_t f;
lv_fs_res_t res;
res = lv_fs_open(&f, "S:folder/file.txt", LV_FS_MODE_RD);

```

(continues on next page)

(continued from previous page)

```
if(res != LV_FS_RES_OK) my_error_handling();

uint32_t read_num;
uint8_t buf[8];
res = lv_fs_read(&f, buf, 8, &read_num);
if(res != LV_FS_RES_OK || read_num != 8) my_error_handling();

lv_fs_close(&f);
```

Le mode dans `lv_fs_open` peut être `LV_FS_MODE_WR` pour ouvrir en écriture ou `LV_FS_MODE_RD` | `LV_FS_MODE_WR` pour lecture/écriture

Cet exemple montre comment lire le contenu d'un répertoire. Il appartient au pilote de marquer les répertoires, mais il peut être judicieux d'insérer un "/" devant le nom du répertoire.

```
lv_fs_dir_t dir;
lv_fs_res_t res;
res = lv_fs_dir_open(&dir, "S:/folder");
if(res != LV_FS_RES_OK) my_error_handling();

char fn[256];
while(1) {
    res = lv_fs_dir_read(&dir, fn);
    if(res != LV_FS_RES_OK) {
        my_error_handling();
        break;
    }

    /* fn est vide s'il n'y a plus d'entrée à lire */
    if(strlen(fn) == 0) {
        break;
    }

    printf("%s\n", fn);
}

lv_fs_dir_close(&dir);
```

Utiliser les pilotes pour les images

Les objets *Image* peuvent également être ouverts à partir de fichiers (en plus des variables stockées dans la mémoire Flash)

Pour initialiser un pilote pour les images, les fonction de rappel suivantes sont requises :

- 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.

lv_fs_drv_t ***lv_fs_get_drv**(char *letter*)

Give a pointer to a driver from its letter

Return pointer to a driver or NULL if not found

Parameters

- **letter**: the driver letter

bool **lv_fs_is_ready**(char *letter*)

Test if a drive is 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

Vous pouvez faire évoluer automatiquement la valeur d’une variable entre une valeur de début et une valeur de fin en utilisant les animations. L’animation est réalisée par l’appel périodique d’une fonction “animateur” avec comme paramètre la valeur correspondante.

La fonction “animateur” a la signature suivante :

```
void func(void * var, lv_anim_var_t value);
```

Cette signature est compatible avec la plupart des fonctions *set* de LittlevGL. Par exemple `lv_obj_set_x(obj, value)` ou `lv_obj_set_width(obj, value)`

Créer une animation

Pour créer une animation, une variable `lv_anim_t` doit être initialisée et configurée avec les fonctions `lv_anim_set_...()`.

```
lv_anim_t a;
lv_anim_set_exec_cb(&a, btn1, lv_obj_set_x);    /* Définit la fonction animateur et
↳ la variable à animer */
lv_anim_set_time(&a, duration, delay);
lv_anim_set_values(&a, start, end);             /* Définit les valeurs de début et de
↳ fin. P. ex. 0, 150 */
lv_anim_set_path_cb(&a, lv_anim_path_linear);    /* Définit le chemin à partir d'une
↳ des fonctions `lv_anim_path_...` ou d'une fonction spécifique. */
lv_anim_set_ready_cb(&a, ready_cb);             /* Définit une fonction de rappel à
↳ exécuter quand l'animation est prête. (Optionnel) */
lv_anim_set_playback(&a, wait_time);           /* Active le déroulé de l'animation
↳ après un délai `wait_time` */
lv_anim_set_repeat(&a, wait_time);             /* Active la répétition d'une
↳ animation après un délai `wait_time` delay. Peut être associé à la fonction `lv_
↳ anim_set_playback` */
lv_anim_create(&a);                            /* Débute l'animation */
```

Vous pouvez appliquer de **multiples différentes animations** à la même variable au même moment. Par exemple animer les coordonnées x et y avec `lv_obj_set_x` et `lv_obj_set_y`. Cependant, une seule animation peut exister avec une variable et une fonction données. Par conséquent, `lv_anim_create()` supprimera les animations déjà existantes de variable et fonction.

Chemin d'animation

Vous pouvez déterminer le **chemin de l'animation**. Dans les cas les plus simples, il est linéaire ce qui veut dire que la valeur entre *start* et *end* évolue linéairement. Un *chemin* est une fonction qui calcule la prochaine valeur à assigner, basée sur le statut actuel de l'animation. Actuellement, les chemins suivants sont prédéfinis :

- `lv_anim_path_linear` animation linéaire
- `lv_anim_path_step` change en une seule fois à la fin
- `lv_anim_path_ease_in` lent au début
- `lv_anim_path_ease_out` lent à la fin
- `lv_anim_path_ease_in_out` lent au début et à la fin
- `lv_anim_path_overshoot` dépasse la valeur de fin
- `lv_anim_path_bounce` rebondit un peu sur la valeur de fin (comme en frappant un mur)

Vitesse vs durée

Par défaut, vous pouvez définir la durée de l'animation. Mais dans certains cas, il est plus pratique d'utiliser la **vitesse d'animation**.

La fonction `lv_anim_speed_to_time(speed, start, end)` calcule la durée requise en millisecondes pour atteindre la valeur de fin à partir de la valeur de départ avec une vitesse donnée. La vitesse est interprétée en *unité/seconde*. Par exemple `lv_anim_speed_to_time(20,0,100)` donnera 5000 millisecondes. Par exemple dans le cas de `lv_obj_set_x` l'*unité* est le pixel donc 20 signifie une vitesse de 20 px/s.

Supprimer des animations

Vous pouvez **supprimer une animation** par `lv_anim_del(var, func)` en indiquant la variable animée et sa fonction animateur.

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

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

Tâches

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);
}
```

(continues on next page)

(continued from previous page)

```

/*Do something with LittlevGL*/
if(something_happened) {
    something_happened = false;
    lv_btn_create(lv_scr_act(), NULL);
}
}

...

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

```

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);
}

...

/*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 be initialized with at least `lv_task_set_cb` and `lv_task_set_period`

Return pointer to the created task

lv_task_t ***lv_task_create**(*lv_task_cb_t* task_xcb, uint32_t period, *lv_task_prio_t* prio, void *user_data)

Create a new `lv_task`

Return pointer to the new task

Parameters

- **task_xcb**: a callback which is the task itself. It will be called periodically. (the ‘x’ in the argument name indicates that it's 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 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:

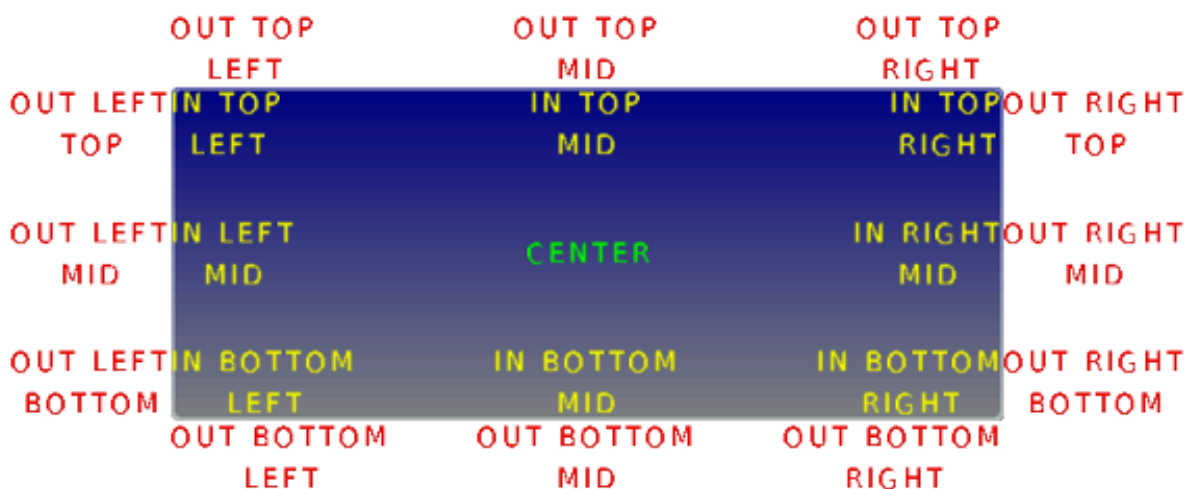
- coordinates
- parent object
- children
- main style
- attributes like *Click enable*, *Drag enable*, etc.

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 another 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 arguments mean 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` is 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 be 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

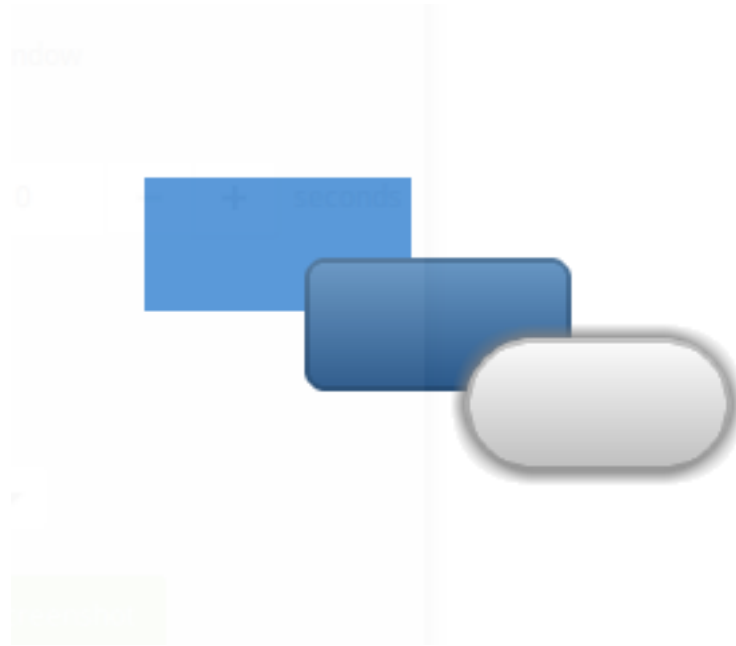
No *Keys* are processed by the object type.

Learn more about [Keys](#).

Example

C

Base objects with custom styles



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 `indev` was pressing this object but swiped out while pressing do not search other object.
LV_PROTECT_CLICK_FOCUS = 0x20
 Prevent focusing the object by clicking on it

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_del_async**(struct _lv_obj_t *obj)

Helper function for asynchronously deleting objects. Useful for cases where you can't delete an object directly in an LV_EVENT_DELETE handler (i.e. parent).

See lv_async_call

Parameters

- **obj**: object to delete

void **lv_obj_clean**(lv_obj_t *obj)

Delete all children of an object

Parameters

- **obj**: pointer to an object

void **lv_obj_invalidate**(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. The opacity of this object and all it's children will be scaled down with this factor. *lv_obj_set_opa_scale_enable(obj, true)* needs to be called to enable it. (not for all children just for the parent where to start the opa scaling)

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* *cords_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 an 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_realign_t

Public Members

const struct `_lv_obj_t *`**base**

`lv_coord_t` **xofs**

`lv_coord_t` **yofs**

`lv_align_t` **align**

`uint8_t` **auto_realign**

`uint8_t` **origo_align**

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

struct _lv_obj_t

Public Members

struct _lv_obj_t *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)

Vue d’ensemble

L’objet *arc* trace un arc entre les angles de début et de fin dans une certaine épaisseur.

Angles

Pour définir les angles, la fonction `lv_arc_set_angles(arc, start_angle, end_angle)` est utilisée. Le degré zéro est en bas de l’objet et les degrés s’incrémentent dans la direction des aiguilles d’une montre. Les angles doivent être compris dans l’intervalle [0;360].

Notes

Les **largeur et hauteur** de l’*arc* doivent être **identiques**.

Actuellement, l’objet *arc* **ne prend pas en charge** l’anticrénelage.

Styles

Pour définir le style d’un objet *arc* la fonction `lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &style)` est utilisée

- **line.rounded** rend les extrémités arrondies (l'opacité ne fonctionnera pas correctement si elle est définie à 1)
- **line.width** l'épaisseur de l'arc
- **line.color** la couleur de l'arc.

Evénements

Les *événements génériques* sont les seuls à être envoyés par ce type d'objet.

Apprenez-en plus sur les *événements*.

Touches

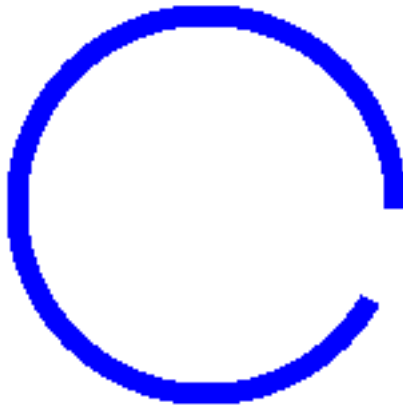
Aucune *touche* n'est traitée par ce type d'objet.

Apprenez-en plus sur les *touches*.

Exemple

C

Simple Arc



code

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

void lv_ex_arc_1(void)
{
```

(continues on next page)

(continued from previous page)

```

/*Create style for the Arcs*/
static 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);
}

```

Loader with Arc



code

```

#include "lvgl/lvgl.h"

/**
 * An `lv_task` to call periodically to set the angles of the arc
 * @param t
 */
static void arc_loader(lv_task_t * t)
{
    static int16_t a = 0;

    a+=5;
    if(a >= 359) a = 359;

    if(a < 180) lv_arc_set_angles(t->user_data, 180-a ,180);
}

```

(continues on next page)

(continued from previous page)

```

else lv_arc_set_angles(t->user_data, 540-a ,180);

if(a == 359) {
    lv_task_del(t);
    return;
}
}

/**
 * Create an arc which acts as a loader.
 */
void lv_ex_arc_2(void)
{
    /*Create style for the Arcs*/
    static lv_style_t style;
    lv_style_copy(&style, &lv_style_plain);
    style.line.color = LV_COLOR_NAVY;           /*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_angles(arc, 180, 180);
    lv_arc_set_style(arc, LV_ARC_STYLE_MAIN, &style);
    lv_obj_align(arc, NULL, LV_ALIGN_CENTER, 0, 0);

    /* Create an `lv_task` to update the arc.
     * Store the `arc` in the user data*/
    lv_task_create(arc_loader, 20, LV_TASK_PRIO_LOWEST, arc);
}
    
```

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**

Barre (lv_bar)

Vue d'ensemble

L'objet barre possède deux parties principales :

1. un **fond**, l'objet lui-même
2. un **indicateur** dont la forme est similaire à celle du fond mais dont les largeur et hauteur peuvent être ajustée.

L'orientation de la barre peut être verticale ou horizontale selon le rapport largeur/hauteur. Logiquement, sur les barres horizontales, la largeur de l'indicateur est modifiable. Sur les barres verticales, c'est la hauteur de l'indicateur qui peut être modifiée.

Valeur et intervalle

Une nouvelle valeur peut être définie par `lv_bar_set_value(bar, new_value, LV_ANIM_ON/OFF)`. La valeur est comprise dans un intervalle (valeurs minimale et maximale) qui peut être modifié avec `lv_bar_set_range(bar, min, max)`. L'intervalle par défaut est 1..100.

La nouvelle valeur définie par `lv_bar_set_value` est affichée avec ou sans animation selon la valeur du dernier paramètre (`LV_ANIM_ON/OFF`). La durée de l'animation peut être ajustée par `lv_bar_set_anim_time(bar, 100)`. L'unité de durée est la milliseconde.

Symétrie

La barre peut être dessinée symétriquement par rapport à zéro (de zéro vers la gauche ou la droite, le haut ou le bas) si cela est activé par `lv_bar_set_sym(bar, true)`

Styles

Pour définir le style d'un objet *barre* `lv_bar_set_style(arc, LV_BAR_STYLE_MAIN, &style)` est utilisée.

- **LV_BAR_STYLE_BG** est un *objet de base* de ce fait, il utilise ses éléments de style. Son style par défaut est: `lv_style_pretty`.
- **LV_BAR_STYLE_INDIC** est similaire au fond. Il utilise les marges *left*, *right*, *top* et *bottom* pour conserver un peu d'espace sur les bords du fond. Son style par défaut est `lv_style_pretty_color`.

Événements

Les événements génériques sont les seuls à être envoyés par ce type d'objet.

Apprenez-en plus sur les *événements*.

Touches

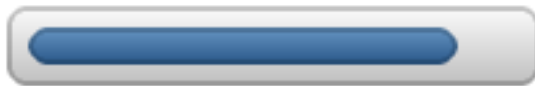
Aucune *touche* n'est traitée par ce type d'objet.

Apprenez-en plus sur les *touches*.

Exemple

C

Simple Bar



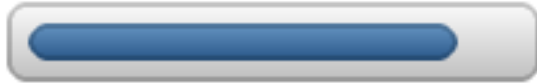
code

```
#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

Simple Bar



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
```

Bouton (lv_btn)

Vue d'ensemble

Les boutons sont de simples objets rectangulaires dont le style et l'état changent quand ils sont pressés ou relâchés.

Etats

Les boutons peuvent prendre l'un des 5 états possibles :

- **LV_BTN_STATE_REL** Etat relâché
- **LV_BTN_STATE_PR** Etat pressé
- **LV_BTN_STATE_TGL_REL** Etat basculé relâché
- **LV_BTN_STATE_TGL_PR** Etat basculé pressé
- **LV_BTN_STATE_INA** Etat inactif

L'état passe automatiquement de `..._REL` à `..._PR` quand le bouton est pressé et inversement quand il est relâché.

L'état peut être défini par programmation avec `lv_btn_set_state(btn, LV_BTN_STATE_TGL_REL)`.

Bascule

Les boutons peuvent être configurés comme *bouton bascule* avec `lv_btn_set_toggle(btn, true)`. Dans ce cas, au relâchement, le bouton prend l'état *basculé relâché*.

Mise en page et remplissage

De la même manière que les *Conteneurs*, les boutons ont des attributs de disposition et de mise en page.

- `lv_btn_set_layout(btn, LV_LAYOUT_...)` définit une mise en page. La valeur par défaut est `LV_LAYOUT_CENTER`. Donc, si vous ajoutez une étiquette, elle sera automatiquement alignée au milieu et ne pourra pas être déplacée avec `lv_obj_set_pos()`. La mise en page peut être désactivée avec `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

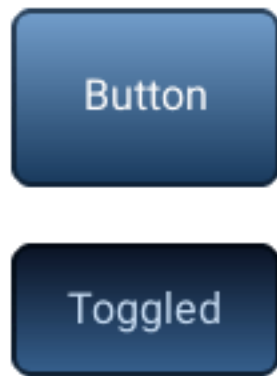
Note that, as usual, the state of `LV_KEY_ENTER` is translated to `LV_EVENT_PRESSED/PRESSING/RELEASED` etc.

Learn more about [Keys](#).

Example

C

Simple Buttons



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);
}
```

(continues on next page)

(continued from previous page)

```
lv_btn_set_toggle(btn2, true);
lv_btn_toggle(btn2);
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 object

Return pointer to the created button

Parameters

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

void **lv_btn_set_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 tells how to change the button size automatically.

Parameters

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

- **right**: right fit policy from `lv_fit_t`
- **top**: top fit policy from `lv_fit_t`
- **bottom**: bottom fit policy from `lv_fit_t`

static void lv_btn_set_fit2(*lv_obj_t* *btn, *lv_fit_t* hor, *lv_fit_t* ver)

Set the fit policy horizontally and vertically separately. It tells how to change the button size automatically.

Parameters

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

static void lv_btn_set_fit(*lv_obj_t* *btn, *lv_fit_t* fit)

Set the fit policy in all 4 direction at once. It tells how to change the button size automatically.

Parameters

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

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 true: 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 **recolored** 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

Simple Button matrix



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_BTNUM_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_BTNUM_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

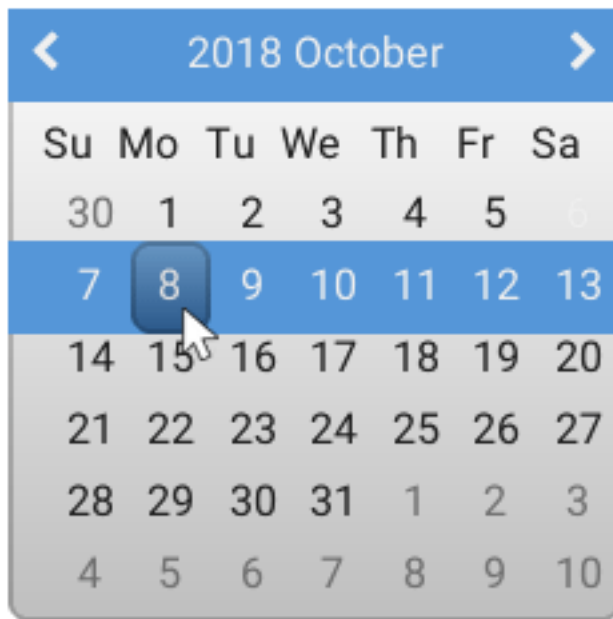
No *Keys* are processed by the object type.

Learn more about *Keys*.

Example

C

Calendar with day select



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);
}
```

(continues on next page)

(continued from previous page)

```

/*Set the today*/
lv_calendar_date_t today;
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. NULL if not date pressed (e.g. the header)

Parameters

- **calendar**: pointer to a calendar object

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

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

Parameters

- **calendar**: pointer to a calendar object

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

Return number of highlighted days

Parameters

- **calendar**: pointer to a calendar object

const char ****lv_calendar_get_day_names**(**const** lv_obj_t *calendar)

Get the name of the days

Return pointer to the array of day names

Parameters

- **calendar**: pointer to a calendar object

const char ****lv_calendar_get_month_names**(**const** lv_obj_t *calendar)

Get the name of the month

Return pointer to the array of month names

Parameters

- **calendar**: pointer to a calendar object

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_...)`. `buffer` is a static buffer (not just a local variable) to hold the image of the canvas. For example `static lv_color_t buffer[LV_CANVAS_BUF_SIZE_TRUE_COLOR(width, height)]`. `LV_CANVAS_BUF_SIZE_...` macros help to determine the size of the buffer with different color formats.

The canvas supports all the built in color formats like `LV_IMG_CF_TRUE_COLOR` or `LV_IMG_CF_INDEXED_2BIT`. See the full list in the [Color formats](#) section.

Palette

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

Drawing

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

`lv_canvas_fill_bg(canvas, LV_COLOR_BLUE)` 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)`

The draw function can draw only to `LV_IMG_CF_TRUE_COLOR`, `LV_IMG_CF_TRUE_COLOR_CHROMA_KEYED` and `LV_IMG_CF_TRUE_COLOR_ALPHA` buffers. `LV_IMG_CF_TRUE_COLOR_ALPHA` is working only with `LV_COLOR_DEPTH 32`.

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

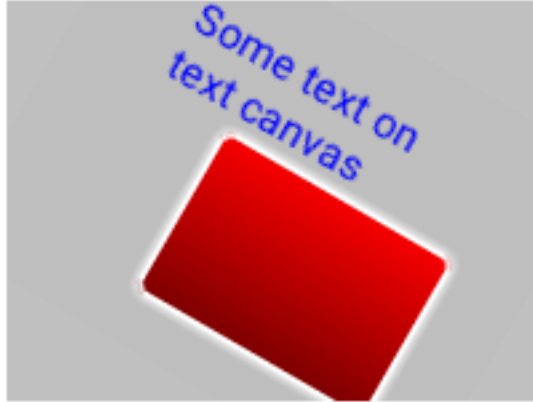
No *Keys* are processed by the object type.

Learn more about [Keys](#).

Example

C

Drawing on the Canvas and rotate



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);
}
```

(continues on next page)

(continued from previous page)

```
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);
}
```

Transparent Canvas with chroma keying



code

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

#define CANVAS_WIDTH 50
#define CANVAS_HEIGHT 50

/**
 * Create a transparent canvas with Chroma keying and indexed color format (palette).
 */
void lv_ex_canvas_2(void)
{
```

(continues on next page)

(continued from previous page)

```

    /*Create a button to better see the transparency*/
    lv_btn_create(lv_scr_act(), NULL);

    /*Create a buffer for the canvas*/
    static lv_color_t cbuf[LV_CANVAS_BUF_SIZE_INDEXED_1BIT(CANVAS_WIDTH, CANVAS_
↪HEIGHT)];

    /*Create a canvas and initialize its the palette*/
    lv_obj_t * canvas = lv_canvas_create(lv_scr_act(), NULL);
    lv_canvas_set_buffer(canvas, cbuf, CANVAS_WIDTH, CANVAS_HEIGHT, LV_IMG_CF_INDEXED_
↪1BIT);
    lv_canvas_set_palette(canvas, 0, LV_COLOR_TRANSP);
    lv_canvas_set_palette(canvas, 1, LV_COLOR_RED);

    /*Create colors with the indices of the palette*/
    lv_color_t c0;
    lv_color_t c1;

    c0.full = 0;
    c1.full = 1;

    /*Transparent background*/
    lv_canvas_fill_bg(canvas, c1);

    /*Create hole on the canvas*/
    uint32_t x;
    uint32_t y;
    for( y = 10; y < 30; y++) {
        for( x = 5; x < 20; x++) {
            lv_canvas_set_px(canvas, x, y, c0);
        }
    }
}

```

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**: y 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

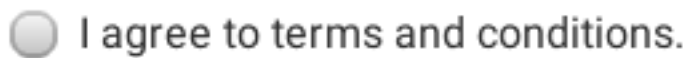
Note that, as usual, the state of **LV_KEY_ENTER** is translated to **LV_EVENT_PRESSED/PRESSING/RELEASED** etc.

Learn more about *Keys*.

Example

C

Simple Checkbox



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_cb_is_checked(obj) ? "Checked" : "Unchecked");
    }
}

void lv_ex_cb_1(void)
{
    lv_obj_t * cb = lv_cb_create(lv_scr_act(), NULL);
```

(continues on next page)

(continued from previous page)

```
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\nseco\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

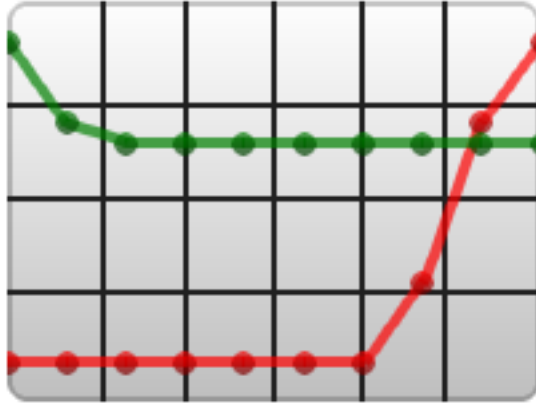
No *Keys* are processed by the object type.

Learn more about [Keys](#).

Example

C

Line Chart



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 'dll'*/
    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);
    lv_chart_set_next(chart, ser1, 10);
    lv_chart_set_next(chart, ser1, 10);
    lv_chart_set_next(chart, ser1, 30);
}
```

(continues on next page)

(continued from previous page)

```
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_darking**(*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 may 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

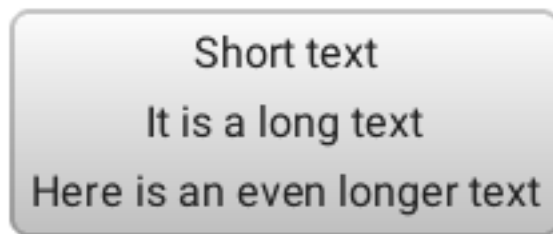
No *Keys* are processed by the object type.

Learn more about [Keys](#).

Example

C

Container with auto-fit



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);
    lv_cont_set_layout(cont, LV_LAYOUT_COL_M);
}
```

(continues on next page)

(continued from previous page)

```
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**: top fit policy from *lv_fit_t*

- **bottom**: bottom fit policy from `lv_fit_t`

static void lv_cont_set_fit2(*lv_obj_t* *cont, *lv_fit_t* hor, *lv_fit_t* ver)

Set the fit policy horizontally and vertically separately. It tells how to change the container's size automatically.

Parameters

- **cont**: pointer to a container object
- **hor**: horizontal fit policy from `lv_fit_t`
- **ver**: vertical fit policy from `lv_fit_t`

static void lv_cont_set_fit(*lv_obj_t* *cont, *lv_fit_t* fit)

Set the fit policy in all 4 direction at once. It tells how to change the container's size automatically.

Parameters

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

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

To 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
- **LY_KEY_ENTER** Apply the selected option (Send **LV_EVENT_VALUE_CHANGED** event and close the Drop down list)

Example

C

Simple Drop down list



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];
```

(continues on next page)

(continued from previous page)

```

        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);
}

```

Drop “up” list



code

```

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

/**
 * Create a drop UP list by applying auto realign

```

(continues on next page)

(continued from previous page)

```

*/
void lv_ex_ddlist_2(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_fix_height(ddlist, 150);
    lv_ddlist_set_draw_arrow(ddlist, true);

    /* Enable auto-realign when the size changes.
     * It will keep the bottom of the ddlist fixed*/
    lv_obj_set_auto_realign(ddlist, true);

    /*It will be called automatically when the size changes*/
    lv_obj_align(ddlist, NULL, LV_ALIGN_IN_BOTTOM_MID, 0, -20);
}

```

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 ‘\n’-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

Simple Gauge



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, $\text{line_cnt} = (\text{sub_div} + 1) * (\text{label_cnt} - 1) + 1$
- **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

Image from variable and symbol



code

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

LV_IMG_DECLARE(cogwheel);

void lv_ex_img_1(void)
{
```

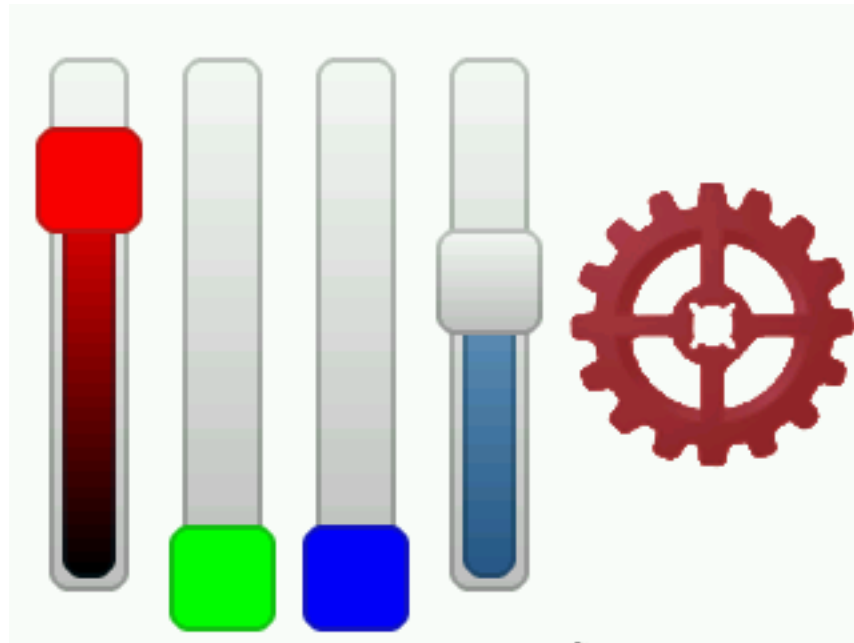
(continues on next page)

(continued from previous page)

```
lv_obj_t * img1 = lv_img_create(lv_scr_act(), NULL);
lv_img_set_src(img1, &cogwheel);
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);
}
```

Image reoloring



code

```
/**
 * @file lv_ex_img_2.c
 *
 */

/*****
 * INCLUDES
 *****/

#include "lvgl/lvgl.h"

/*****
 * DEFINES
 *****/
#define SLIDER_WIDTH 40

/*****
 * TYPEDEFS
 *****/
```

(continues on next page)

(continued from previous page)

```

/*****
 *  STATIC PROTOTYPES
 *****/
static void create_sliders(void);
static void slider_event_cb(lv_obj_t * slider, lv_event_t event);

/*****
 *  STATIC VARIABLES
 *****/
static lv_obj_t * red_slider, * green_slider, * blue_slider, * intense_slider;
static lv_obj_t * img1;
static lv_style_t img_style;
LV_IMG_DECLARE(cogwheel);

/*****
 *  MACROS
 *****/

/*****
 *  GLOBAL FUNCTIONS
 *****/

void lv_ex_img_2(void)
{
    /*Create 4 sliders to adjust RGB color and re-color intensity*/
    create_sliders();

    /* Now create the actual image */
    img1 = lv_img_create(lv_scr_act(), NULL);
    lv_img_set_src(img1, &cogwheel);
    lv_obj_align(img1, intense_slider, LV_ALIGN_OUT_RIGHT_MID, 10, 0);

    /* Create a message box for information */
    static const char * btns[] ={"OK", ""};

    lv_obj_t * mbox = lv_mbox_create(lv_scr_act(), NULL);

    lv_mbox_set_text(mbox, "Welcome to the image recoloring demo!\nThe first three_
↪sliders control the RGB value of the recoloring.\nThe last slider controls the_
↪intensity.");
    lv_mbox_add_btns(mbox, btns);
    lv_obj_align(mbox, NULL, LV_ALIGN_CENTER, 0, 0);

    /* Save the image's style so the sliders can modify it */
    lv_style_copy(&img_style, lv_img_get_style(img1, LV_IMG_STYLE_MAIN));
}

/*****
 *  STATIC FUNCTIONS
 *****/

static void slider_event_cb(lv_obj_t * slider, lv_event_t event)
{
    if(event == LV_EVENT_VALUE_CHANGED) {
        /* Recolor the image based on the sliders' values */
    }
}

```

(continues on next page)

(continued from previous page)

```

        img_style.image.color = lv_color_make(lv_slider_get_value(red_slider), lv_
↪ slider_get_value(green_slider), lv_slider_get_value(blue_slider));
        img_style.image.intense = lv_slider_get_value(intense_slider);
        lv_img_set_style(img1, LV_IMG_STYLE_MAIN, &img_style);
    }
}

static void create_sliders(void)
{
    /* Create a set of RGB sliders */
    /* Use the red one as a base for all the settings */
    red_slider = lv_slider_create(lv_scr_act(), NULL);
    lv_slider_set_range(red_slider, 0, 255);
    lv_obj_set_size(red_slider, SLIDER_WIDTH, 200); /* Be sure it's a vertical slider_
↪ */

    lv_obj_set_event_cb(red_slider, slider_event_cb);

    /* Create the intensity slider first, as it does not use any custom styles */
    intense_slider = lv_slider_create(lv_scr_act(), red_slider);
    lv_slider_set_range(intense_slider, LV_OPA_TRANSP, LV_OPA_COVER);

    /* Create the slider knob and fill styles */
    /* Fill styles are initialized with a gradient between black and the slider's_
↪ respective color. */
    /* Knob styles are simply filled with the slider's respective color. */
    static lv_style_t slider_red_fill_style, slider_red_knob_style;

    lv_style_copy(&slider_red_fill_style, lv_slider_get_style(red_slider, LV_SLIDER_
↪ STYLE_INDIC));
    lv_style_copy(&slider_red_knob_style, lv_slider_get_style(red_slider, LV_SLIDER_
↪ STYLE_KNOB));

    slider_red_fill_style.body.main_color = lv_color_make(255, 0, 0);
    slider_red_fill_style.body.grad_color = LV_COLOR_BLACK;

    slider_red_knob_style.body.main_color = slider_red_knob_style.body.grad_color =_
↪ slider_red_fill_style.body.main_color;

    static lv_style_t slider_green_fill_style, slider_green_knob_style;
    lv_style_copy(&slider_green_fill_style, &slider_red_fill_style);
    lv_style_copy(&slider_green_knob_style, &slider_red_knob_style);

    slider_green_fill_style.body.main_color = lv_color_make(0, 255, 0);

    slider_green_knob_style.body.main_color = slider_green_knob_style.body.grad_color_
↪ = slider_green_fill_style.body.main_color;

    static lv_style_t slider_blue_fill_style, slider_blue_knob_style;
    lv_style_copy(&slider_blue_fill_style, &slider_red_fill_style);
    lv_style_copy(&slider_blue_knob_style, &slider_red_knob_style);

    slider_blue_fill_style.body.main_color = lv_color_make(0, 0, 255);

    slider_blue_knob_style.body.main_color = slider_blue_knob_style.body.grad_color =_
↪ slider_blue_fill_style.body.main_color;

```

(continues on next page)

(continued from previous page)

```

/* Setup the red slider */
lv_slider_set_style(red_slider, LV_SLIDER_STYLE_INDIC, &slider_red_fill_style);
lv_slider_set_style(red_slider, LV_SLIDER_STYLE_KNOB, &slider_red_knob_style);

/* Copy it for the other two sliders */
green_slider = lv_slider_create(lv_scr_act(), red_slider);
lv_slider_set_style(green_slider, LV_SLIDER_STYLE_INDIC, &slider_green_fill_
↪style);
lv_slider_set_style(green_slider, LV_SLIDER_STYLE_KNOB, &slider_green_knob_style);

blue_slider = lv_slider_create(lv_scr_act(), red_slider);
lv_slider_set_style(blue_slider, LV_SLIDER_STYLE_INDIC, &slider_blue_fill_style);
lv_slider_set_style(blue_slider, LV_SLIDER_STYLE_KNOB, &slider_blue_knob_style);

lv_obj_align(red_slider, NULL, LV_ALIGN_IN_LEFT_MID, 10, 0);

lv_obj_align(green_slider, red_slider, LV_ALIGN_OUT_RIGHT_MID, 10, 0);

lv_obj_align(blue_slider, green_slider, LV_ALIGN_OUT_RIGHT_MID, 10, 0);

lv_obj_align(intense_slider, blue_slider, LV_ALIGN_OUT_RIGHT_MID, 10, 0);
}

```

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

Simple Image button



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;
```

(continues on next page)

(continued from previous page)

```

style_pr.image.intense = LV_OPA_50;
style_pr.text.color = lv_color_hex3(0xaaa);

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

Keyboard with text area



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*/
```

(continues on next page)

(continued from previous page)

```
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_SCROLL** Keep the size and scroll the label back and forth
- **LV_LABEL_LONG_SCROLL_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)`

The background will be larger in every direction with `body.padding.top/bottom/left/right` values. However, the background is drawn only “virtually” and doesn't make the label really larger. There for when the label is positioned the label's coordinates will be taken into account and not background's.

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)`. Form 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

Label recoloring and scrolling

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);
}
```

Text shadow

A simple method to create
shadows on text
It even works with

newlines and spaces.

code

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

void lv_ex_label_2(void)
{
    /* Create a style for the shadow*/
    static lv_style_t label_style;
    lv_style_copy(&label_style, &lv_style_plain);
    label_style.text.opa = LV_OPA_50;

    /*Create a label for the shadow first (it's in the background) */
    lv_obj_t * shadow_label = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_style(shadow_label, LV_LABEL_STYLE_MAIN, &label_style);

    /* Create the main label */
    lv_obj_t * main_label = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(main_label, "A simple method to create\n"
                                "shadows on text\n"
                                "It even works with\n\n"
                                "newlines    and spaces.");

    /*Set the same text for the shadow label*/
    lv_label_set_text(shadow_label, lv_label_get_text(main_label));

    /* Position the main label */
    lv_obj_align(main_label, NULL, LV_ALIGN_CENTER, 0, 0);

    /* Shift the second label down and to the right by 1 pixel */
    lv_obj_align(shadow_label, main_label, LV_ALIGN_IN_TOP_LEFT, 1, 1);
}
```

Align labels

A text with
multiple
lines

A text with
multiple
lines

A text with
multiple
lines

code

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

static void text_changer(lv_task_t * t);

lv_obj_t * labels[3];

/**
 * Create three labels to demonstrate the alignments.
 */
void lv_ex_label_3(void)
{
    /*`lv_label_set_align` is not required to align the object itself.
     * It's used only when the text has multiple lines*/

    /* Create a label on the top.
     * No additional alignment so it will be the reference*/
    labels[0] = lv_label_create(lv_scr_act(), NULL);
    lv_obj_align(labels[0], NULL, LV_ALIGN_IN_TOP_MID, 0, 5);
    lv_label_set_align(labels[0], LV_LABEL_ALIGN_CENTER);

    /* Create a label in the middle.
     * `lv_obj_align` will be called every time the text changes
     * to keep the middle position */
    labels[1] = lv_label_create(lv_scr_act(), NULL);
    lv_obj_align(labels[1], NULL, LV_ALIGN_CENTER, 0, 0);
    lv_label_set_align(labels[1], LV_LABEL_ALIGN_CENTER);

    /* Create a label in the bottom.
     * Enable auto realign. */
    labels[2] = lv_label_create(lv_scr_act(), NULL);
    lv_obj_set_auto_realign(labels[2], true);
}
```

(continues on next page)

(continued from previous page)

```

lv_obj_align(labels[2], NULL, LV_ALIGN_IN_BOTTOM_MID, 0, -5);
lv_label_set_align(labels[2], LV_LABEL_ALIGN_CENTER);

lv_task_t * t = lv_task_create(text_changer, 1000, LV_TASK_PRIO_MID, NULL);
lv_task_ready(t);
}

static void text_changer(lv_task_t * t)
{
    const char * texts[] = {"Text", "A very long text", "A text with\nmultiple\nlines", NULL};
    static uint8_t i = 0;

    lv_label_set_text(labels[0], texts[i]);
    lv_label_set_text(labels[1], texts[i]);
    lv_label_set_text(labels[2], texts[i]);

    /*Manually realign `labels[1]`*/
    lv_obj_align(labels[1], NULL, LV_ALIGN_CENTER, 0, 0);

    i++;
    if(texts[i] == NULL) i = 0;
}

```

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_SCROLL/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

LED with custom style



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);

    /*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_off(led3);
}
```

(continues on next page)

(continued from previous page)

```
}

```

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

Simple Line



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.

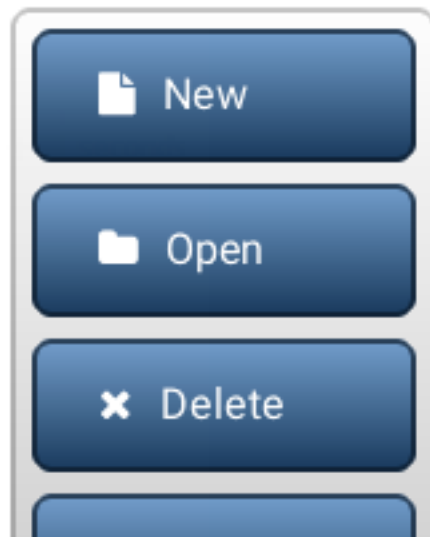
To manually select a button use `lv_list_set_btn_selected(list, btn)`. When the list is defocused and focused again it will restore the last selected button.

Learn more about [Keys](#).

Example

C

Simple List



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);
}
```

(continues on next page)

(continued from previous page)

```
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_SCROLL

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 scr object, without deleting scr 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 <= index < 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 if the button is 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_ANOM_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 is greater than 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

Simple Line meter



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*/
}
```

(continues on next page)

(continued from previous page)

```

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:

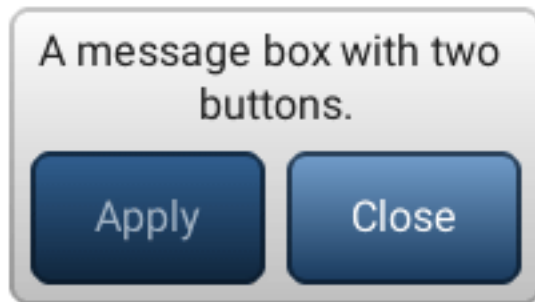
- **LV_KEY_RIGHT/DOWN** Select the next button
- **LV_KEY_LEFT/TOP** Select the previous button
- **LV_KEY_ENTER** Clicks the selected button

Learn more about [Keys](#).

Example

C

Simple Message box



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*/
}
```

Modal



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.";
```

(continues on next page)

(continued from previous page)

```

/*****
 *   GLOBAL FUNCTIONS
 *****/

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 */
    }
}

```

(continues on next page)

(continued from previous page)

```

        lv_obj_t *obj = lv_obj_create(lv_scr_act(), NULL);
        lv_obj_set_style(obj, &modal_style);
        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

- **mbox**: 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 *mbox*, 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 *mbox*)
Get the text of the message box

Return pointer to the text of the message box

Parameters

- **mbox**: pointer to a message box object

uint16_t **lv_mbox_get_active_btn**(*lv_obj_t *mbox*)
Get the index of the lastly “activated” button by the user (pressed, released etc) Useful in the the **event_cb**.

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

Parameters

- **btnm**: pointer to button matrix object

const char ***lv_mbox_get_active_btn_text**(*lv_obj_t *mbox*)
Get the text of the lastly “activated” button by the user (pressed, released etc) Useful in the the **event_cb**.

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

Parameters

- **btnm**: pointer to button matrix object

uint16_t **lv_mbox_get_anim_time**(**const** *lv_obj_t *mbox*)
Get the animation duration (close animation time)

Return animation length in milliseconds (0: no animation)

Parameters

- **mbox**: pointer to a message box object

const *lv_style_t* ***lv_mbox_get_style**(**const** *lv_obj_t *mbox*, *lv_mbox_style_t type*)
Get a style of a message box

Return style pointer to a style

Parameters

- **mbox**: pointer to a message box object
- **type**: which style should be get

bool **lv_mbox_get_recolor**(const lv_obj_t *mbox)

Get whether recoloring is enabled

Return whether recoloring is enabled

Parameters

- **mbox**: pointer to a message box object

lv_obj_t ***lv_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_scr1()`
- `lv_page_set_scr1_fit/fint2/fit4()`
- `lv_page_set_scr1_width()`
- `lv_page_set_scr1_height()`
- `lv_page_set_scr1_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

Page with scrollbar



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 a 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

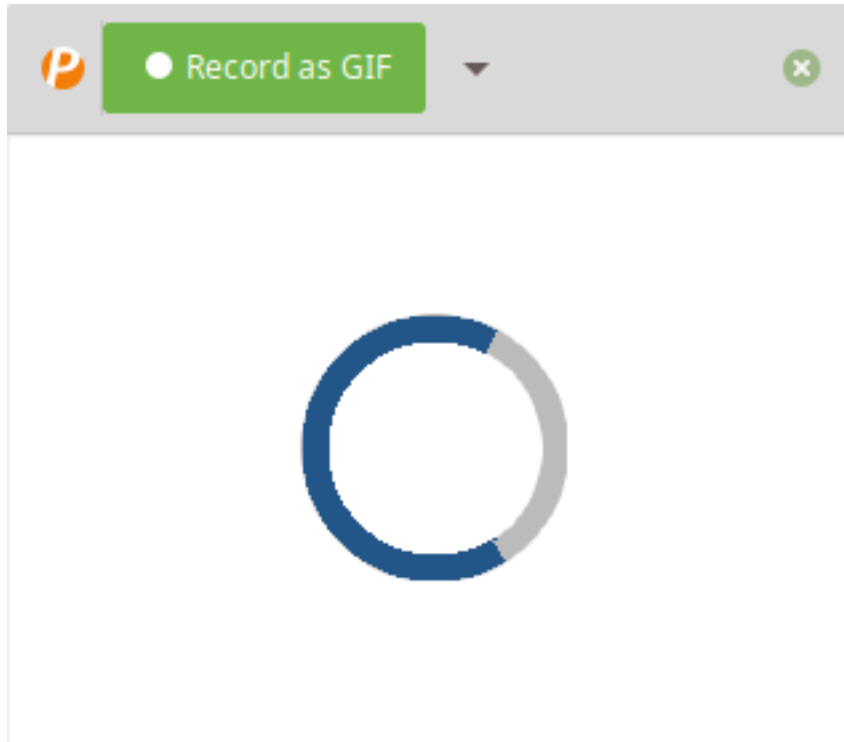
No *Keys* are processed by the object type.

Learn more about [Keys](#).

Example

C

Preloader with custom style



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

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

`lv_roller_get_selected_str(roller, buf, buf_size)` copy the name of the selected option to `buf`.

Align the options

To align the label horizontally use `lv_roller_set_align(roller, LV_LABEL_ALIGN_LEFT/CENTER/RIGHT)`.

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:

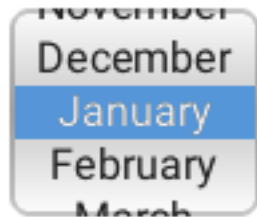
- **LV_KEY_RIGHT/DOWN** Select the next option
- **LV_KEY_LEFT/UP** Select the previous option

- **LV_KEY_ENTER** Apply the selected option (Send LV_EVENT_VALUE_CHANGED event)

Example

C

Simple Roller



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"
```

(continues on next page)

(continued from previous page)

```

        "July\n"
        "August\n"
        "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 ‘ ’-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

Slider with custo mstyle



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));
    }
}
```

(continues on next page)

(continued from previous page)

```
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);
    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);
}
```

Set value with slider

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
 *
 */

/*****
 *   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;
```

(continues on next page)

(continued from previous page)

```

/*****
 *      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_obj_set_auto_realign(slider_label, true);
    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"
                           "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);
    }
}

```

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

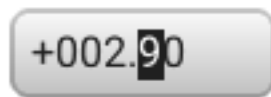
The following *Keys* are processed by the Buttons:

- **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 next digit. Jump to the first after the last.

Example

C

Simple Spinbox



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

Simple Switch



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;
```

(continues on next page)

(continued from previous page)

```

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;

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

Simple table

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:

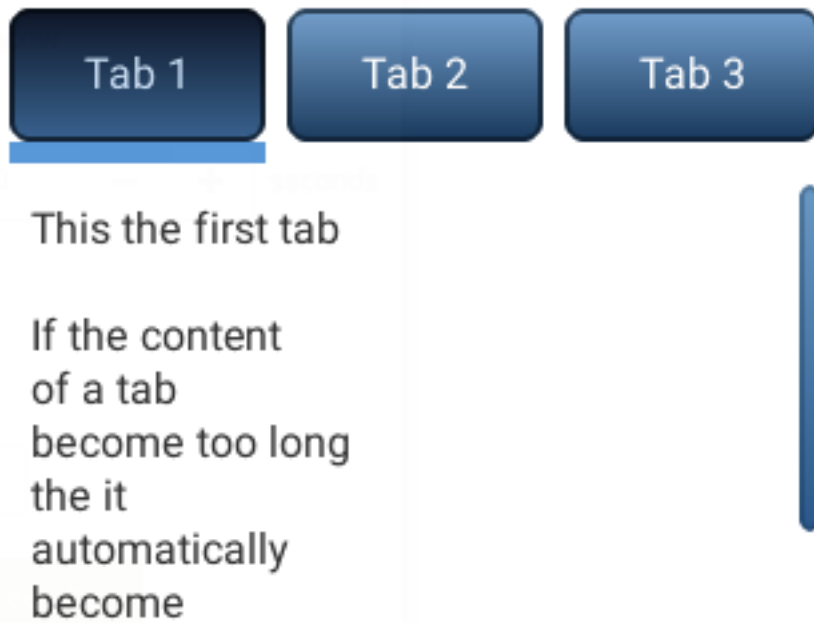
- **LV_KEY_RIGHT/LEFT** Select a tab
- **LV_KEY_ENTER** Change to the selected tab

Learn more about [Keys](#).

Example

C

Simple Tabview



code

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

void lv_ex_tabview_1(void)
{
    /*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 scr object, without deleting scr 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 add 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 (≥ 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 tab 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 tab 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 dragging
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 be 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 be 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 another 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

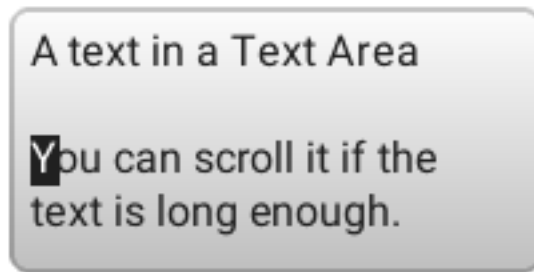
- **LV_KEY_UP/DOWN/LEFT/RIGHT** Move the cursor
- **Any character** Add the character to the current cursor position

Learn more about [Keys](#).

Example

C

Simple Text area



code

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

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);
}
```

Text are with password field

Password:

*****t

Text:

Hello



code

```
/**
 * @file lv_ex_tmpl.c
 *
 */

/*****
 * INCLUDES
 *****/
#include "lvgl/lvgl.h"
#include <stdio.h>

/*****
 * DEFINES
 *****/

/*****
 * TYPEDEFS
 *****/

/*****
 * STATIC PROTOTYPES
 *****/
static void kb_event_cb(lv_obj_t * event_kb, lv_event_t event);
static void ta_event_cb(lv_obj_t * ta, lv_event_t event);

/*****
 * STATIC VARIABLES
 *****/

static lv_obj_t * kb;
/*****/
```

(continues on next page)

(continued from previous page)

```

*      MACROS
*****//

/*****

*      GLOBAL FUNCTIONS
*****//

void lv_ex_ta_2(void)
{
    /* Create the password box */
    lv_obj_t * pwd_ta = lv_ta_create(lv_scr_act(), NULL);
    lv_ta_set_text(pwd_ta, "");
    lv_ta_set_pwd_mode(pwd_ta, true);
    lv_ta_set_one_line(pwd_ta, true);
    lv_obj_set_width(pwd_ta, LV_HOR_RES / 2 - 20);
    lv_obj_set_pos(pwd_ta, 5, 20);
    lv_obj_set_event_cb(pwd_ta, ta_event_cb);

    /* Create a label and position it above the text box */
    lv_obj_t * pwd_label = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(pwd_label, "Password:");
    lv_obj_align(pwd_label, pwd_ta, LV_ALIGN_OUT_TOP_LEFT, 0, 0);

    /* Create the one-line mode text area */
    lv_obj_t * oneline_ta = lv_ta_create(lv_scr_act(), pwd_ta);
    lv_ta_set_pwd_mode(oneline_ta, false);
    lv_ta_set_cursor_type(oneline_ta, LV_CURSOR_LINE | LV_CURSOR_HIDDEN);
    lv_obj_align(oneline_ta, NULL, LV_ALIGN_IN_TOP_RIGHT, -5, 20);

    /* Create a label and position it above the text box */
    lv_obj_t * oneline_label = lv_label_create(lv_scr_act(), NULL);
    lv_label_set_text(oneline_label, "Text:");
    lv_obj_align(oneline_label, oneline_ta, LV_ALIGN_OUT_TOP_LEFT, 0, 0);

    /* Create a keyboard and make it fill the width of the above text areas */
    kb = lv_kb_create(lv_scr_act(), NULL);
    lv_obj_set_pos(kb, 5, 90);
    lv_obj_set_event_cb(kb, kb_event_cb); /* Setting a custom event handler stops the
↪ keyboard from closing automatically */
    lv_obj_set_size(kb, LV_HOR_RES - 10, 140);

    lv_kb_set_ta(kb, pwd_ta); /* Focus it on one of the text areas to start */
    lv_kb_set_cursor_manage(kb, true); /* Automatically show/hide cursors on text
↪ areas */
}

/*****

*      STATIC FUNCTIONS
*****//

static void kb_event_cb(lv_obj_t * event_kb, lv_event_t event)
{
    /* Just call the regular event handler */
    lv_kb_def_event_cb(event_kb, event);
}

```

(continues on next page)

(continued from previous page)

```

}
static void ta_event_cb(lv_obj_t * ta, lv_event_t event)
{
    if(event == LV_EVENT_CLICKED) {
        /* Focus on the clicked text area */
        if(kb != NULL)
            lv_kb_set_ta(kb, ta);
    }

    else if(event == LV_EVENT_INSERT) {
        const char * str = lv_event_get_data();
        if(str[0] == '\n') {
            printf("Ready\n");
        }
    }
}
}

```

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 area 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 '*'s).

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(tileview, 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 can be changed 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 to 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

Tileview with content



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);

    /*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(tileview, list);

    lv_obj_t * list_btn;
    list_btn = lv_list_add_btn(list, NULL, "One");
    lv_tileview_add_element(tileview, list_btn);
}
```

(continues on next page)

(continued from previous page)

```
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);

list_btn = lv_list_add_btn(list, NULL, "Six");
lv_tileview_add_element(tileview, list_btn);

list_btn = lv_list_add_btn(list, NULL, "Seven");
lv_tileview_add_element(tileview, list_btn);

list_btn = lv_list_add_btn(list, NULL, "Eight");
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_STYBE_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**)

The height of the header is set to the greater value from *buttons' height* (set by **lv_win_set_btn_size**) and *title height* (comes from **header_style.text.font**) plus the **body.padding.top** and **body.padding.bottom** of the header style.

Events

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

Learn more about [Events](#).

Keys

The following *Keys* are processed by the Page:

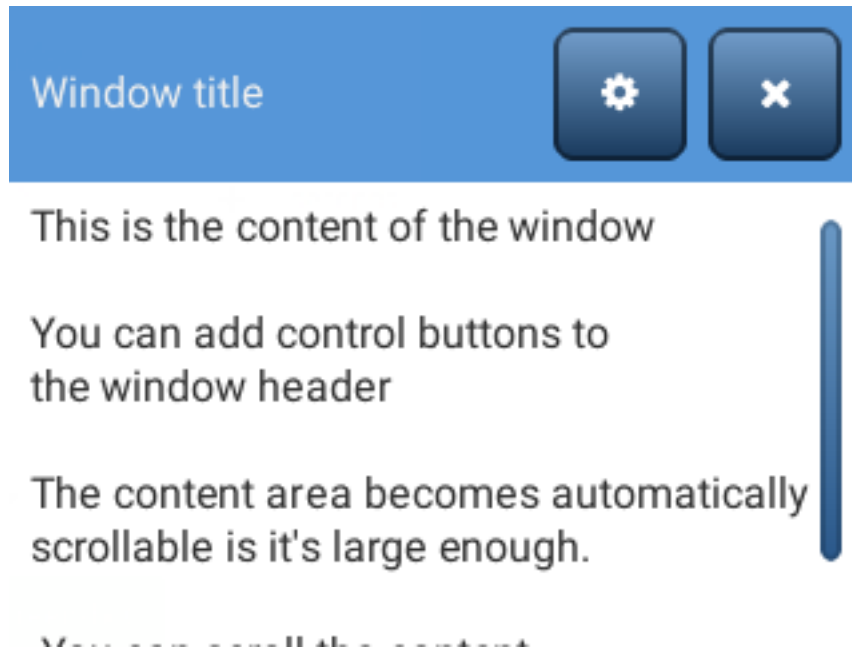
- **LV_KEY_RIGHT/LEFT/UP/DOWN** Scroll the page

Learn more about [Keys](#).

Example

C

Simple window



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 scr object, without deleting scr 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

```