interfaces graphiques tactiles pour ESP32, RPi & Co.

avec la bibliothèque LittlevGL

Mathias Claussen (Elektor Labs)

Pratiquement tous les projets à microcontrôleur doivent afficher quelque chose. Si jadis des afficheurs de texte à deux lignes suffisaient à la tâche, de nos jours on utilise de plus en plus des afficheurs graphiques à cristaux liquides (LCD) ou à diodes électroluminescentes organiques (OLED). Pour l'affichage de graphiques attrayants et/ou la commande tactile, il existe toute une série de bibliothèques. *LittlegVL* est une bibliothèque qui, sous la licence MIT très libre, peut être adaptée aux afficheurs et aux contrôleurs les plus variés. Dans cet article, nous allons démontrer tout cela avec une carte ESP32 et un écran LCD tactile qui affiche les données d'une station météorologique.

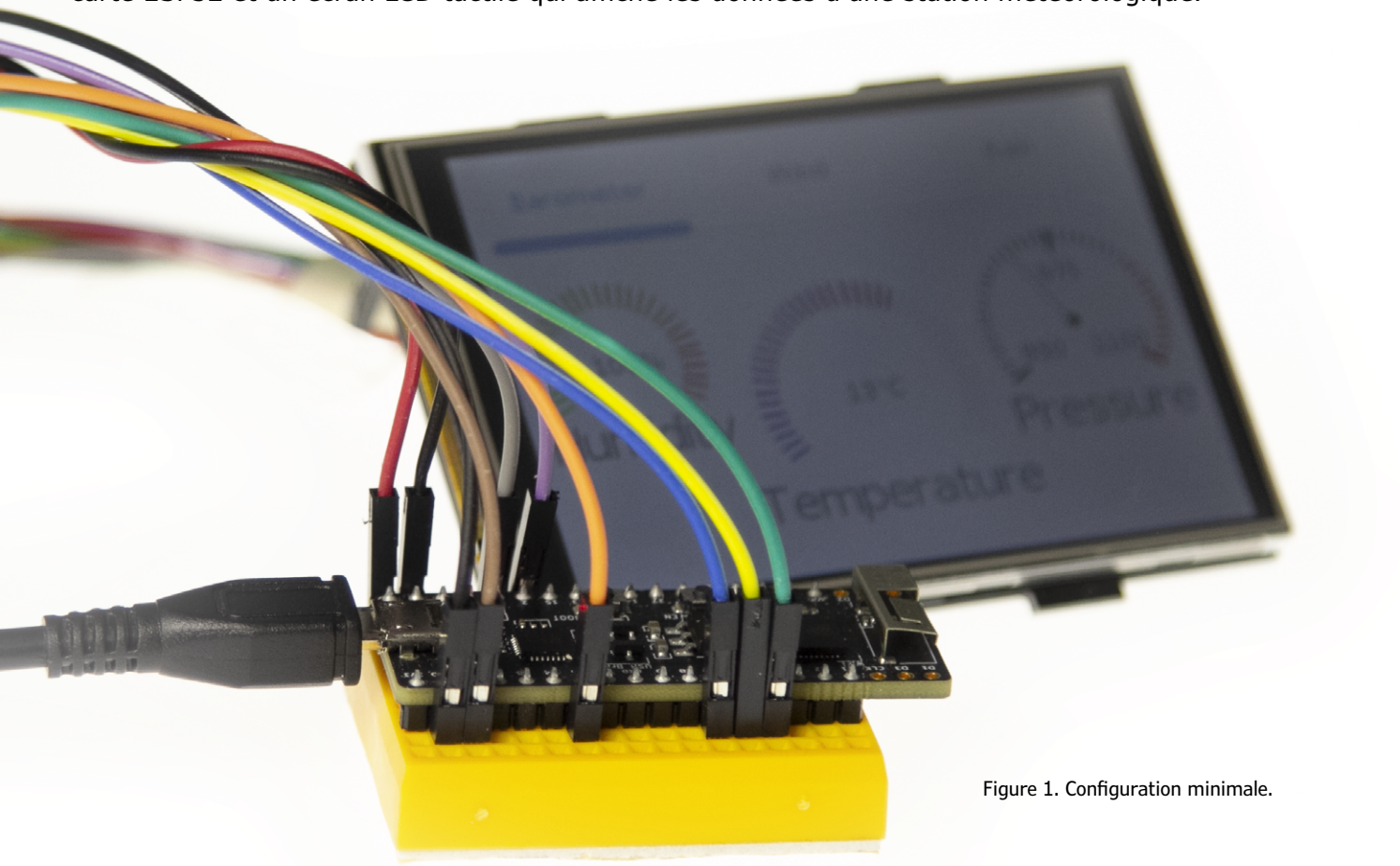


Figure 1. Configuration minimale.

Les interfaces graphiques utilisateur (GUI) nécessitent souvent un effort de programmation bien supérieur au « code utile » qui réalise les fonctions importantes du projet. Afin de se concentrer sur les aspects essentiels du projet, il est de plus en plus important et apprécié de pouvoir recourir, pour l'affichage de données sous forme graphique, à des solutions complètes, voire des bibliothèques toutes faites telles que *µGFX*, *emWin* ou *TouchGFX* pour les cartes STM32, pour n'en citer que quelques-unes parmi les plus connues. Elles ont toutes des avantages et des inconvénients, par ex. à cause de leurs licences commerciales ou de leurs liens avec des fabricants particuliers de microcontrôleurs. Bien entendu, il est toujours possible d'écrire sa propre bibliothèque, mais cela représente un travail énorme, bourré de chausse-trapes, sans compter les nombreux bogues tapis dans un code personnel volumineux.
Avec *LittlevGL* de Gábor Kiss-Vámosi [1], on se trouve en terrain

bien plus sympathique, car cette bibliothèque bénéficie d'une licence MIT avec des conditions d'utilisation très amicales. Une GUI réalisée avec cette bibliothèque est bien adaptée aux écrans tactiles, mais peut aussi être utilisée avec une souris, un clavier ou des touches indépendantes. Le code est exécutable sur des microcontrôleurs à 16, 32 et 64 bits. La configuration minimale se contente de 16 MHz, 64 Ko de flash et 16 Ko de RAM. Cela rend cette bibliothèque idéale pour des petites cartes comme l'ESP32 ou l'ESP8266. Espressif l'a d'ailleurs ajoutée à son environnement de développement de l'Internet des Objets (*IoT Development Framework*, IDF). De plus, il existe de l'aide pour débuter et l'assemblage du matériel de test. *LittlevGL* offre aussi la possibilité non négligeable de développer et de tester des GUI sur PC ; le code créé sur PC peut ensuite être porté sur le microcontrôleur cible sans guère de modifications.

Bibliothèques & ESP32

C'est avec des essais pratiques qu'on apprend le plus. C'est pourquoi nous montrerons ici comment utiliser cette bibliothèque avec la station météorologique d'Elektor [2]. L'objectif est une GUI adaptée à un écran tactile. Nous allons même réaliser un affichage de données sur plusieurs pages. Mais pour cela, nous avons besoins de matériel.
Il est facile de se procurer un module ESP32. Sont appropriés un *ESP32-PICO-D4*, un *ESP32-DevKitC* ou une carte dérivée. Pour l'afficheur, on a le choix entre une interface série ou une connexion parallèle, mais qui consommerait presque toutes les entrées/sorties de l'ESP32. Comme il y a aussi la question du prix, nous nous sommes décidés pour un afficheur LCD de 3,5 pouces pour Raspberry Pi très répandu. La plupart des afficheurs bon marché comme le 3,5 pouces de JOY-iT sont connectés par SPI et travaillent avec des niveaux de signaux de 3,3 V. Ils sont donc parfaits pour une connexion économe en broches à une carte ESP32. De plus, ils possèdent déjà un contrôleur de la fonction tactile intégré et connectable par SPI. Les afficheurs SPI destinés au RPi sont limités quant à la vitesse de transfert des données, comme on peut s'en rendre compte à la lenteur de rafraîchissement des images affichées.

Important : *LittlevGL* ne fournit pas de pilote d'afficheur, mais seulement des fonctions « de niveau plus élevé » pour le dessin d'objets. L'écriture des routines de gestion du matériel reste à la charge du développeur. Mais même là, point n'est besoin de réinventer la roue, car il existe des bibliothèques toutes faites pour la plupart des afficheurs. Dans notre cas, il s'agit de la bibliothèque Arduino *TFT_eSPI* [3] qui supporte aussi les afficheurs de 3,5 pouces.

Quincaillerie

Pour le projet (**fig. 1** ci-contre), il vous faut :
- ESP32-DevKitC-32D ou ESP32-PICO-Kit V4
- Afficheur tactile de 3,5 pouces pour Raspberry Pi de JOY-iT
- Petites cartes de prototypage et fils de câblage

Matière grise

Pas de surprise pour le logiciel nécessaire : outre l'inévitable EDI Arduino avec gestion des cartes ESP32, il faut les versions Arduino des bibliothèques *LittlevGL* et *TFT_eSPI*.
Pour installer et gérer confortablement ces deux bibliothèques, il faut insérer les deux lignes suivantes dans l'IDE Arduino sous *Préférences -> URL de gestionnaire de cartes supplémentaires* :

```
https://github.com/littlevgl/lv_arduino/library.json
https://github.com/Bodmer/TFT_eSPI/library.json
```

Ces lignes demandent au gestionnaire de bibliothèques de chercher et d'installer *LittlevGL* et *TFT_eSPI*. Ensuite, on vérifie que ces deux bibliothèques apparaissent bien dans le dossier des bibliothèques d'Arduino.
Comme déjà mentionné, les deux bibliothèques sont nécessaires. *LittlevGL* s'occupe de l'interface utilisateur, donc de l'animation et de l'agencement des objets, de la gestion de multiples scènes et du rendu graphique. Le résultat est une matrice de bits (*bitmap*). Ces données sont alors transférées sous la forme adéquate à l'afficheur par *TFT_eSPI*. Ces bibliothèques réalisent une abstraction de l'afficheur effectivement utilisé.

D'autres d'afficheurs

TFT_eSPI supporte non seulement les afficheurs SPI pour le RPi, mais aussi ceux équipés des contrôleurs suivants : ILI9341, ST7735, ILI9163, S6D02A1, HX8357D, ILI9481, ILI9486, ILI9488, ST7789 et R61581.
C'est là une belle liste d'afficheurs couleurs courants. Si on veut utiliser un contrôleur de marque RAiO, comme le RA8875, on peut recourir à la bibliothèque *RA8875* de Adafruit [4]. Il faut alors prévoir des adaptations pour pouvoir lier *LittlevGL.* Outre des afficheurs couleurs, la bibliothèque *u8g2* [5] permet aussi de piloter des afficheurs monochromes. Le texte qui suit ne concerne toutefois que l'afficheur SPI LCD de 3,5 pouces pour le RPi.

Pilote spécifique

Si l'on utilise un afficheur pour lequel il n'existe pas de pilote Arduino, il faut savoir quelles fonctions doivent être disponibles. Ces informations sont aussi bien utiles en cas de portage et d'édition de liens.
En principe, un pilote spécifique peut se limiter à savoir placer des pixels de couleur donnée sur l'écran. *LittlevGL* s'attend à une fonction de la forme suivante :

```
/*************************************
* Function    : disp_flush_data
* Description : Sends pixels to the display
* Input       : lv_disp_drv_t *disp, int32_t x1,
*               int32_t y1, int32_t x2, int32_t y2,
*               const lv_color_t *color_array
* Output      : none
* Remarks     : none
*************************************/
void disp_flush_data(lv_disp_drv_t *disp,
 const lv_area_t *area, lv_color_t *color_p){
   /* Here: grab the pixel and send it to the display
   */

   /* Inform the library after job is done */
   lv_flush_ready(disp);
}
```

Cette fonction est passée à *LittlevGL* sous la forme d'un pointeur. Elle reçoit comme paramètres les coordonnées de début et de fin de l'espace à remplir, ainsi qu'un pointeur vers les données de l'image. L'action sur les pixels dépend du pilote de l'afficheur concerné. Toutefois il peut arriver que

les couleurs spécifiées par *LittlevGL* doivent être recalculées avant affichage (par ex. de RVB à BVR). La routine de dessin n'a en principe rien d'autre à savoir. Par ailleurs, il existe des solutions d'accélération du matériel, comme le moteur DMA2D de certains contrôleurs STM32.

Touche finale

Nous savons à présent comment l'image est envoyée sur l'écran. Il ne manque que le traitement des données du contrôleur tactile. Pour cela, il y a une fonction de *LittlevGL* qui lit et traite les données si on touche l'écran, en fonction du type de l'afficheur. En général, les afficheurs pour RPi sont équipés d'un *XPT2046* connecté en mode esclave au bus SPI. Malheureusement, on ne peut pas le lire à une vitesse supérieure à 2,5 MHz. Comme l'afficheur et son contrôleur sont cadencés à 20 MHz (et même jusqu'à 26 MHz à température ambiante), la vitesse du bus doit être réglée pour chaque accès et ensuite restaurée à la valeur d'origine. Là encore la bibliothèque *TFT_eSPI* montre son utilité, car non seulement elle supporte le XPT2046, mais elle prend automatiquement en charge la commutation de vitesse.
Sans fonction tactile, on peut toujours utiliser l'interface avec une souris, un clavier, une molette ou des boutons-poussoirs. Bien entendu, ces moyens de saisie nécessitent aussi des pilotes appropriés. Ils doivent être enregistrés de manière adéquate dans *LittlevGL*.

Dessine-moi un bouton

D'abord un mot sur les points forts de *LittlevGL* : il est bien pratique que le code source soit disponible. Cela facilite le débogage de son code personnel. De plus, la bibliothèque est bien documentée et continue d'être activement développée. Grâce aux exemples, même les débutants peuvent rapidement obtenir des résultats lors de la conception d'interfaces. Depuis le simple étiquetage de boutons, tableaux, jusqu'aux listes déroulantes et aux cadrans, on dispose d'une vaste palette d'éléments de contrôle et de commande. De plus, des fenêtres simples ou contextuelles ainsi que des thèmes pour l'écran d'accueil sont proposés afin de pouvoir encore mieux personnaliser la GUI. Tous les éléments bénéficient d'une description détaillée dans la documentation. Les fonctions de la bibliothèque ainsi que leurs interactions sont montrées dans [1]. *LittlevGL* n'offre pas (encore) de fonctions de base d'affichage de pixels ou de tracé de lignes. La raison en est le mode de production de l'image : quand un élément change, il est possible de déterminer le domaine de l'écran à redessiner et de préparer le tampon en RAM interne. L'image est ensuite envoyée à l'écran. De ce fait, il n'y a pas que les lignes qui doivent être disponibles sous forme d'objets, mais aussi les pixels. Cette restriction est compensée par une plus grande facilité de rafraîchissement des domaines sur l'écran.
Venons-en aux détails techniques de la formation et de l'affichage des images : pour obtenir des animations ou des rafraîchissements d'écran sans erreur ni clignotement, on pourrait commencer par préparer l'image complète dans la RAM du microcontrôleur et ensuite l'envoyer vers l'afficheur. Dans notre cas, cela représente 307 Ko de données. Mais on pourrait aussi envoyer directement tous les éléments vers l'afficheur et ainsi occuper moins de RAM. Mais ceci rendrait difficile un affichage sans clignotement et interdirait des effets tels que l'anticrénelage (*antialiasing*), la transparence et les ombres.
On a un compromis en mettant en RAM l'image d'une partie seulement de l'écran. Avec à peine plus de 10% de la mémoire nécessaire pour l'image complète, on dispose déjà de toutes les propriétés ci-dessus. Un afficheur de 480 × 320 pixels avec une palette de couleurs à 16 bits ne consommerait que 30,7 Ko de RAM. Ça reste considérable pour un ESP32, mais encore gérable. Dans la version 6 actuelle de la bibliothèque, la zone mémoire n'est pas communiquée par un `#define` mais doit être préparée dans le code. Cette façon de procéder est particulièrement utile si vous disposez de RAM externe supplémentaire à utiliser. Cette démo simplifie le code en se limitant à l'allocation statique d'une zone dans la mémoire de l'ESP32 :

```
//Memory for the displaybuffer
static lv_disp_buf_t disp_buf;
static lv_color_t buf[LV_HOR_RES_MAX * 10];
```

La ligne suivante affecte cette mémoire à l'affichage dans la fonction `hal_init()` :

```
lv_disp_buf_init(&disp_buf, buf, NULL, LV_HOR_RES_MAX
   * 10);
```

Pour d'autres microcontrôleurs, il faut évaluer le possible et le préférable, car il y en a pas mal qui disposent de sensiblement moins de RAM ou qui seraient contraints à des accès acrobatiques à de la RAM externe. Outre la RAM disponible, d'autres paramètres interviennent tels que la puissance de calcul disponible, ou un fonctionnement à fils multiples que *LittlevGL* ne supporte malheureusement pas : tous les accès doivent être exécutés dans le même fil, qui appelle aussi la fonction **lv_task_handler()**. La puissance de calcul nécessaire dépend du nombre d'interactions et de l'activité graphique sur l'écran ainsi que de la présence et du type d'animations. Grâce à son double cœur, un ESP32 dispose de suffisamment de puissance de calcul pour une GUI.

Tableau 1. Câblage afficheur - ESP32.

fonction	broches (afficheur)	broche ESP32	remarque
MISO	21	19	
MOSI	19	23	
SCK	23	18	
DC	18	02	
CS	24	05	
RST	22	EN	économise une broche GPIO
T_CS	26	04	
VCC (5 V)	02	5 V	
GND	14 / 25	GND	
T_IRQ (optionnel)	11	34	broche ESP32 pour entrée uniquement

Expérimentations

Si l'on veut se lancer dans des expérimentations, il faut s'attendre à quelques pièges. Pour une mise en service sans aspérités, nous donnons un exemple de configuration. Une carte ESP32-D4-PICO à laquelle on a ajouté un afficheur présente occasionnellement quelques difficultés au démarrage dues à la charge supplémentaire. Un condensateur additionnel de 10 µF entre le 3,3 V et la masse retarde suffisamment le démarrage pour

que les tensions aient le temps de s'établir à une valeur normale. La connexion d'un afficheur à la carte ESP32 s'effectue selon le **tableau 1**. Le matériel est ainsi fin prêt pour passer à la configuration et au test du logiciel. On commence par la bibliothèque *TFT_eSPI* en tant que pilote de l'afficheur puis on passe à la configuration de *LittlevGL*.
Pour le pilote de l'afficheur, il faut localiser le dossier *TFT_eSPI* dans le répertoire de l'EDI Arduino pour adapter le fichier *User_Setup.h* à l'afficheur utilisé. Il faut que les **#define** ci-dessous, correspondants à l'afficheur employé, soient présents :

```
#define RPI_ILI9486_DRIVER // max. 20 MHz SPI
#define TFT_MISO 19
#define TFT_MOSI 23
#define TFT_SCLK 18
#define TFT_CS   05 // Chip select control
#define TFT_DC   02 // Data Command control
#define TFT_RST  -1 // set TFT_RST to -1 if display
   RESET is connected to ESP32 RST
#define TOUCH_CS 04 // Chip Select (T_CS) of touch
   screen
#define SPI_FREQUENCY  20000000

// An XPT2046 requires a lower SPI clock rate of
   2.5 MHz:
#define SPI_TOUCH_FREQUENCY 2500000
```

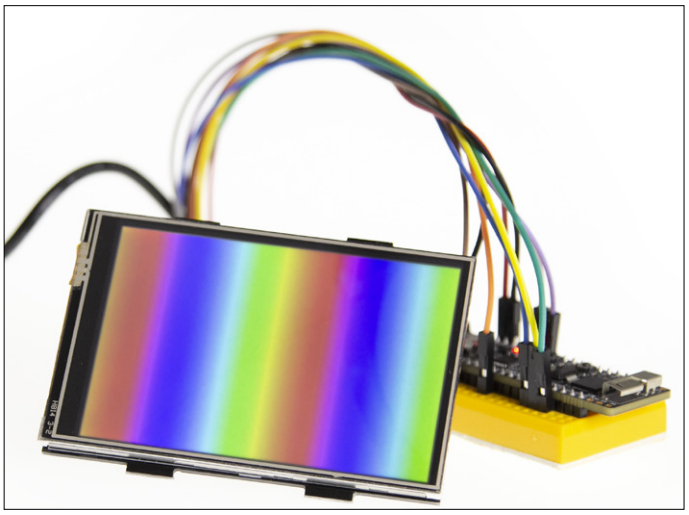
On définit donc les broches GPIO utilisées et une fréquence d'horloge du SPI de 20 MHz, valeur initiale plus sûre pour l'afficheur. Les 2,5 MHz sont mieux adaptés pour le contrôleur tactile. Pour les tests, nous choisissons l'exemple *TFT_eSPI -> 480x320 -> Rainbow480* qui affiche les couleurs de l'arc-en-ciel. Lorsque tout est compilé et connecté correctement, l'afficheur devrait avoir l'aspect de la **figure 2**. Le matériel est alors prêt à l'emploi.
L'étape suivante concerne la liaison de *LittlevGL* au pilote de l'afficheur et l'élaboration d'une interface homme-machine (HMI) personnelle. Pour utiliser *LittlevGL*, il faut commencer par adapter la configuration. Pour cela, on cherche le dossier *littlevGL* dans le dossier des bibliothèques Arduino. Dans le fichier *lv_config.h* qui s'y trouve, on effectue les adaptations pour l'afficheur utilisé et on spécifie les éléments disponibles dans la bibliothèque. Au début du fichier apparaissent les réglages de la gestion de la mémoire. La ligne :

```
#define LV_MEM_SIZE (64U * 1024U)
```

définit la mémoire RAM réservée pour les objets de la GUI. Pour la valeur spécifiée de 64 Ko, l'éditeur de liens va plus tard diagnostiquer une impossibilité de réserver une telle quantité de mémoire. Pour une réservation statique (effectuée lors de la phase de compilation), la discontinuité de l'espace mémoire de l'ESP32 se fait sentir. On pourrait réserver des blocs de mémoire appropriés avec **malloc()** et **free()** au moment de l'exécution, mais comme cela comporte d'autres dangers, on s'y prend autrement. On modifie le contenu de la ligne de la manière suivante :

```
#define LV_MEM_SIZE (24U * 1024U)
```

ce qui suffit pour nos premiers pas.



Premier test : les couleurs de l'arc-en-ciel.

L'afficheur a une résolution de 480 × 320 pixels, spécifiée par les **#define** suivants :

```
/* Horizontal and vertical resolution of the library */
#define LV_HOR_RES_MAX (480)
#define LV_VER_RES_MAX (320)
```

On en déduit la résolution en points par pouce (*Dots Per Inch*, DPI) avec la formule suivante :

$$DPI = \frac{\sqrt[2]{(horizontal\ resolution)^2 + (vertical\ resolution)^2}}{screen\ diagonal\ in\ inch}$$

Ce qui donne :

$$DPI = \frac{\sqrt[2]{(480)^2 + (320)^2}}{3{,}5} = 164{,}83$$

Soit en nombre entier :

```
#define LV_DPI 164
```

Voilà pour les réglages de base. Pour les premiers tests, on conserve les autres réglages et on sauvegarde les modifications. Dans l'EDI Arduino, on peut maintenant choisir l'exemple *ESP32_TFT_eSPI* sous *LittlevGL* et l'envoyer sur la carte ESP32. Si tout est correctement configuré, on devrait voir apparaître « Hello Arduino! » sur fond blanc sur l'afficheur.
Le pilote et *LittlevGL* coopèrent donc correctement. Toutefois, on n'a toujours pas lu le contrôleur tactile de l'afficheur, ni passé ses données à la bibliothèque. Nous allons donc nous intéresser aux parties essentielles du code qui nous permettront d'établir un canevas de base pour une application personnelle. Pour cela, examinons de plus près l'exemple *ESP32_TFT_eSPI* de la bibliothèque *LittlevGL* qui vient juste d'être envoyé sur l'ESP32. Dans la fonction `setup()`, après l'initialisation de la bibliothèque à la ligne 63 avec `lv_init()` et celle du TFT aux lignes 69 et 70 avec `tft.begin()` et `tft.setRotation(1)`, on arrive sur les lignes 73 et 74 à celle de la `struct lv_disp_drv_t`. Un

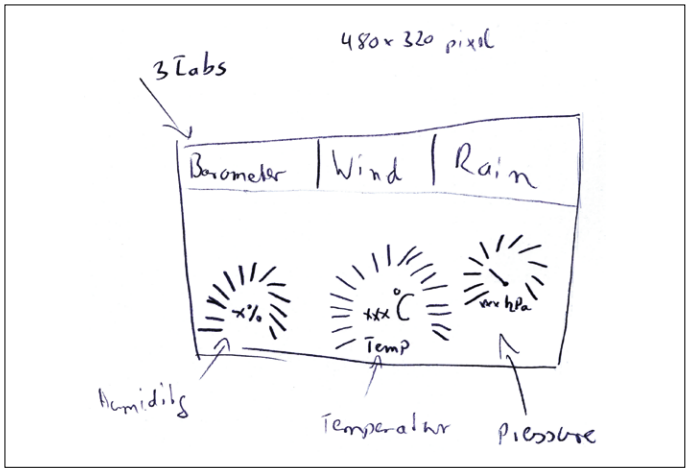


Figure 3. Dessin à la main pour afficher la pression atmosphérique, la température et l'humidité de l'air.

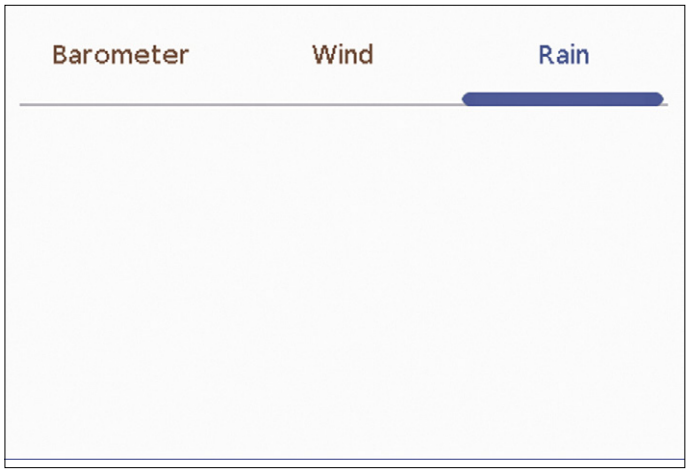


Figure 4. Écran avec trois onglets vides, juste pour voir.

pointeur de fonction est doté de cette `struct` pour l'écriture vers l'afficheur et est ensuite enregistré dans la bibliothèque. Un processus semblable a lieu pour le pilote *Dummy-Touch* sur les lignes 80 à 84. Enfin, une base de temps est mise à la disposition de la bibliothèque au moyen d'un « ticker ». Il s'agit d'une fonction appelée toutes les 20 ms. Un chronomètre est alors incrémenté de 20 ms. Ensuite est créé un bouton auquel est assigné le texte « Hello Arduino! » (lignes 90 à 92).
Dans la fonction `loop()` ne subsiste que l'appel `lv_task_handler()` pour que la GUI puisse réagir à des signaux d'entrée ou rafraîchir l'écran.

Pour ne pas avoir à recommencer depuis le début à chaque projet, l'auteur a conçu un projet de base dans lequel sont effectués les réglages de l'afficheur de JOY-iT et de son contrôleur tactile ainsi que les initialisations des composants. L'orientation de l'afficheur est réglée par l'appel de `tft.setRotation(3)` sur la ligne 139 du croquis. L'image est ainsi tournée de 270° par rapport à la position de sortie. Si un autre afficheur nécessite une autre orientation, par ex. une rotation de 180°, le paramètre doit être mis à 1.

Réalisation de la GUI

Avec ce canevas de base, on peut se mettre à réaliser sa propre GUI. On peut le faire directement sur le matériel de l'ESP32, mais la compilation, le téléversement et les tests prennent du temps. L'alternative est un simulateur sur PC. Son installation, décrite sous [5], exige d'être familier avec *Eclipse*. Elle est un peu plus difficile sous Windows que sous Linux ou OS X. Le simulateur ne permet de tester que les premières étapes sans avoir à recharger à chaque fois le code modifié sur l'ESP32.
On commence par la conception de la couche supérieure pour laquelle le mieux est encore le crayon et le papier (ou la tablette et le stylet), car on devrait avoir des esquisses avant d'écrire la première ligne de code. La **figure 3** montre un exemple d'ébauche tracée à la main. De cette manière, le placement des objets et leurs interactions deviennent clairs.
Comme il s'agit d'une station météorologique, on a choisi pour l'affichages des données une couche supérieure simple avec trois onglets. Pour que le croquis Arduino reste clair, les fonctions et les composants de la GUI sont regroupés dans un fichier séparé.
D'abord on prépare la scène et on crée un élément de visualisation (tabview) auquel on ajoute les trois onglets *Barometer*, *Wind* et *Rain*. Le code suivant se charge de la préparation de la scène :

```
lv_theme_set_current(th);

/* Next step: create a screen */
lv_obj_t * scr = lv_cont_create(NULL, NULL);
lv_scr_load(scr);
```

On commence par charger le thème, passé en paramètre. Pui on prépare et on charge une scène vide. La taille entière de l'écran est assignée à l'élément *tabview*. La copie d'écran (**fig. 4**) montre trois onglets vides avec les titres définis dans le thème. Si l'on clique sur le titre de l'onglet, le changement d'onglet est indiqué par le marqueur bleu. Comme les onglets sont vides, on n'en voit pas plus.
Les cinq lignes de code suivantes :

```
/* And now populate the four tabs */
lv_obj_t * tv = lv_tabview_create(scr, NULL);
lv_obj_set_size(tv, LV_HOR_RES_MAX, LV_VER_RES_MAX);
lv_obj_t * tab0 = lv_tabview_add_tab(tv, "Barometer");
lv_obj_t * tab1 = lv_tabview_add_tab(tv, "Wind");
lv_obj_t * tab2 = lv_tabview_add_tab(tv, "Rain");
```

créent les trois premiers onglets. Pour le moment, ils sont encore vides, mais ils ont déjà un titre.

Affichage de la météo

Allons-y avec le baromètre : nous avons à afficher trois valeurs (humidité de l'air, température et pression). Pour l'humidité de l'air et la température, on utilise **lv_lmeter** et une étiquette (label)qui indiquent la valeur et le nom de la grandeur mesurée. Pour l'humidité de l'air, on utilise **lv_gauge**. Confort supplémentaire : lors de l'exécution, il est possible de modifier les éléments par des styles et ainsi individualiser chaque élément. Lors de l'organisation des éléments, il faut prendre en considé-

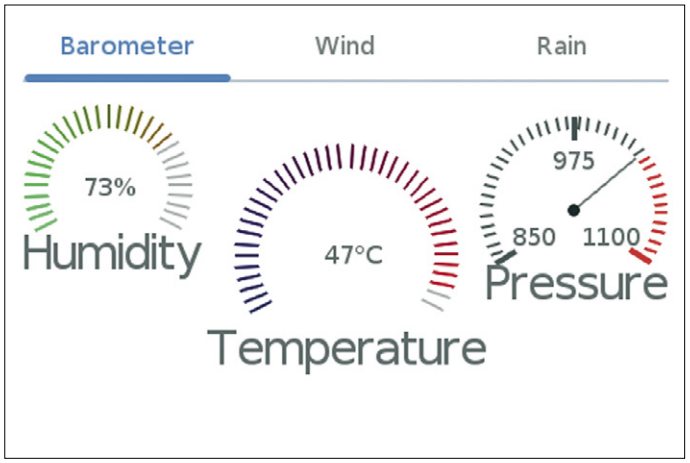


Figure 5. Copie d'écran du premier onglet avec des valeurs du baromètre.

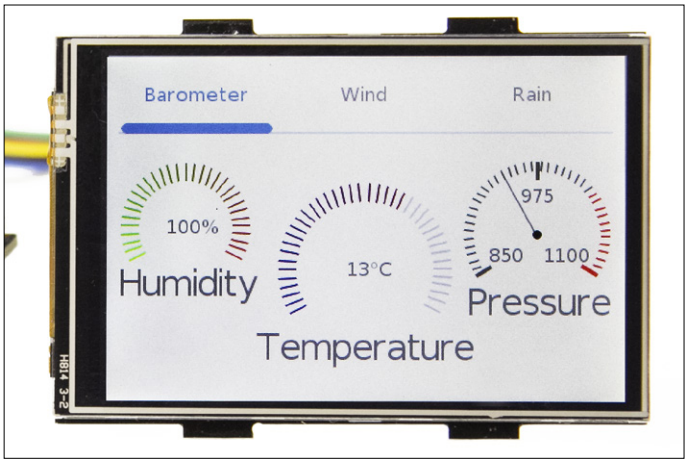


Figure 6. Baromètre sur l'afficheur réel.

ration la fonction **autofit** de la bibliothèque. Il faut donc agencer convenablement les éléments ou bien désactiver **autofit**. On peut positionner les éléments par rapport à plusieurs origines de coordonnées, voir l'aperçu sous [**7**]. Les divers éléments peuvent avoir des *parents*, c'est-à-dire des objets dont dépend leur position. De cette manière, on obtient une interdépendance élégante des positions, où le déplacement d'un parent provoque une réorganisation de tous les enfants (*childs*). Après leur création, on ne peut plus accéder directement aux éléments avec le code, c'est pourquoi, pour *LMeter* et *Gauges,* nous utilisons des pointeurs à accès global. L'exemple de code

```
lv_obj_t* humidity_lmeter
lv_obj_t* humidity_label
lv_obj_t* temp_lmeter
lv_obj_t* temp_label
lv_obj_t* air_pressure_gauge
```

montre que des fonctions comme **lv_lmeter_create** ne retournent que des pointeurs. Se pose la question de savoir où la mémoire est allouée. C'est dissimulé un peu plus profondément dans la bibliothèque. L'expression :

```
# define LV_MEM_SIZE (24U * 1024U)
```

alloue une zone de mémoire statique pour les éléments graphiques. À chaque appel d'une fonction **create**, une portion de mémoire est prélevée dans cette zone et attribuée à l'objet graphique. Le résultat de l'opération est un pointeur vers cette portion, qui sert à modifier les caractéristiques de l'objet. Si jamais cette zone s'épuise, ce qui peut arriver avec des interfaces dynamiques, la bibliothèque signale une erreur et le programme s'enferme dans une boucle sans fin, ce qui paralyse l'ESP32.

Au début, les pointeurs ne sont valides que dans la fonction où l'on se trouve. Si, plus tard, on veut accéder directement à un élément, les pointeurs doivent être sauvegardés en dehors de la fonction. Pour simplifier, nous utilisons pour cela quelques variables globales, ce qui est toutefois déconseillé pour des applications sérieuses.

En passant par les pointeurs, on peut écrire de nouvelles valeurs dans les affichages. La fonction UpdateTemperature en est un bon exemple. Pour l'élément d'affichage *Lmeter*, on attend une valeur comprise entre 0 et 100, mais le domaine de valeurs est de ±50°. On doit donc affecter la température d'un décalage de 50. De sorte que 0° corresponde à une valeur *Lmeter* de 50. La température courante est affichée sous forme de texte, au moyen de **snprintf()** et d'un petit tampon local, dont le contenu sert à rafraîchir le champ de texte. Si la longueur du texte change, son alignement n'est pas automatique. Cet

Liens

[1] Bibliothèque LittlevGL : https://github.com/littlevgl/lv_arduino

[2] « station météo à ESP32 », Elektor 01-02/2019 : www.elektormagazine.fr/180468-04

[3] Bibliothèque TFT_eSPI : https://github.com/Bodmer/TFT_eSPI

[4] Bibliothèque RA8875 : https://github.com/adafruit/Adafruit_RA8875

[5] Bibliothèque u8g2 : https://github.com/olikraus/u8g2

[6] Simulateur pour PC : https://docs.littlevgl.com/en/html/get-started/pc-simulator.html

[7] Positionnement des objets : https://docs.littlevgl.com/en/html/overview/object.html#object-s-working-mechanisms

[8] « horloge à LED géante avec Wi-Fi et mesures météo », Elektor 05-06/2019 : www.elektormagazine.de/180254-01

[9] La page de cet article : www.elektormagazine.fr/190295-03

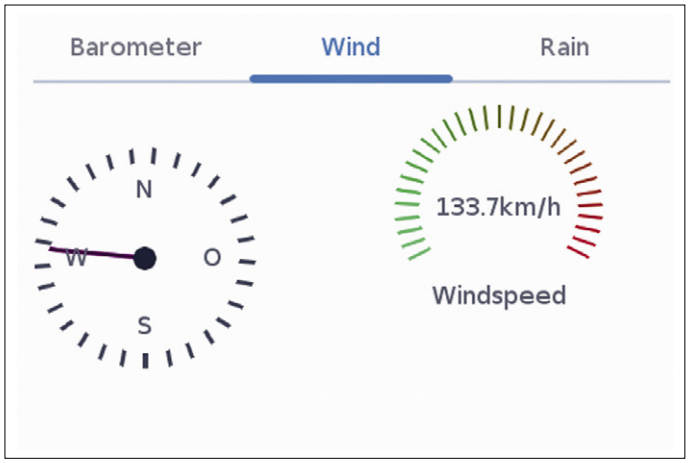


Figure 7. Onglet avec la direction et la vitesse du vent.

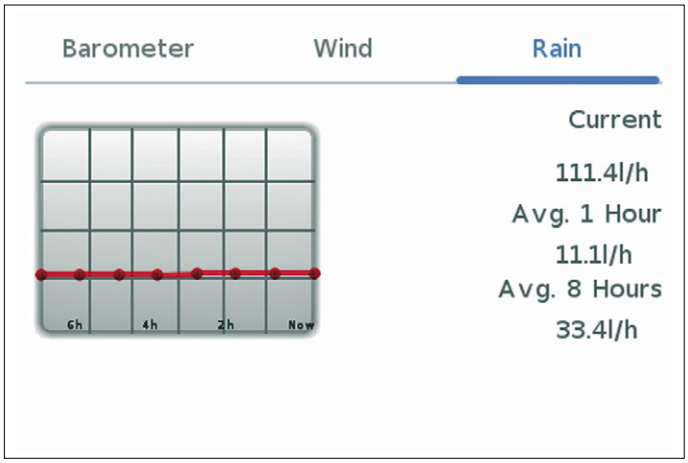


Figure 8. L'onglet Pluie avec courbe et valeurs des précipitations.

alignement doit être réeffectué après écriture du texte. Pour cela, on rappelle **lv_obj_align()**avec les paramètres de l'étiquette. L'humidité et la pression de l'air sont traitées de manière analogue. La **figure 5** montre une copie d'écran des onglets terminés et la **figure 6** l'aspect « réel » de l'affichage sur l'écran LCD.

Nous avons donc rempli le premier onglet avec des données. Nous procéderons de manière analogue avec le second onglet, sauf que l'affichage de la direction du vent sous forme de boussole exigera un peu plus de moyens. Un cadran (*gauge*) sert de parent à quatre étiquettes. Dans le code on crée un cadran gradué de 0 à 359°, suivi de quatre étiquettes (*label*) qui reçoivent pour parent la boussole. Les étiquettes sont définies par rapport au centre de la boussole et indiquent ainsi les quatre points cardinaux. L'aiguille indique la direction d'où vient le vent. Sur le cadran, ce n'est pas la valeur 0 qui donne 0°, mais la valeur 180. Pour l'indication de la vitesse du vent, on utilise un *Lmeter* analogue à celui du baromètre. On voit que, lors de la création des éléments, les mêmes étapes se succèdent : on commence par attribuer un style à l'objet, puis on crée l'objet, et enfin, on lui attribue ses propriétés. Le résultat est visible sur la **figure 7**.

Pour la pluie ou les précipitations, l'affichage des valeurs prend un autre aspect. Les valeurs sont représentées sous forme de texte et d'un diagramme de l'évolution. Les textes sont réalisés de la manière déjà décrite : création des styles et des objets, puis assignation des valeurs. Pour l'évolution des précipitations, on utilise un diagramme linéaire qui, depuis la version 6, ne nécessite plus de bidouillage pour l'étiquetage des axes. Pour l'actualisation des valeurs, il n'est pas nécessaire de déplacer individuellement chaque point, **lv_chart_set_next** s'en charge. Une nouvelle valeur est communiquée au diagramme une fois par heure. La mise à jour des précipitations est effectuée, comme pour d'autres textes, par une fonction spécifique. La **figure 8** présente une copie d'écran de données fictives pour la courbe et les précipitations.

Pour la liaison des données de l'afficheur, nous réutilisons le code du projet d'horloge à LED géantes [**8**], qui traite des données envoyées par un agent (*broker*) via MQTT. Le code s'attend à ce que l'agent envoie un message JSON contenant l'humidité de l'air, la température, la pression atmosphérique, la direction du vent et les précipitations. Quand l'agent envoie de nouvelles données, elles sont réparties dans leurs éléments associés. Il faut veiller à ne pas le faire faire par des fils différents. Excepté l'absence du réglage de l'heure, il n'y a pas de différence notable avec le projet de l'horloge à LED géantes, même pour la configuration. Les réglages du Wi-Fi et de MQTT sont repris tels quels, il n'y a qu'à régler la station météo et l'affichage sur le même sujet (*topic*). À partir de là, les valeurs parviennent directement à l'afficheur. Seule la pluie fait provisoirement exception, car seule la quantité actuelle est disponible. Il manque encore dans la station météo le calcul des quantités horaires et de l'évolution. Dès que ce sera disponible, ces valeurs seront actualisées sur l'afficheur.

Conclusion

Le code de cet exemple pratique de quelques-unes des fonctions fondamentales de *LittlevGL* est téléchargeable gratuitement [**9**]. *LittlevGL* offre bien davantage de fonctions et d'animations et de possibilités d'agencement de tableaux de bord, de liste et de menus déroulants.

Cette bibliothèque est facile à utiliser. Essayez *LittlevGL* dans vos propres projets. Un module ESP32 associé à un afficheur forme, pour un prix raisonnable, une plateforme universelle et puissante.

Vous trouverez ailleurs dans ce numéro un entretien de la rédaction et de labo d'Elektor avec Gábor Kiss-Vámosi. le géniteur de cette remarquable bibliothèque ⏮

(190295-03 – VF : Helmut Müller)