multitâche en pratique avec l'ESP32

programmation de tâches avec FreeRTOS et l'EDI Arduino

Warren Gay (Canada)

Avec un microcontrôleur comme plaque tournante d'un projet, les développeurs sont souvent confrontés à la nécessité d'exécuter plusieurs tâches à la fois : balayage des valeurs des capteurs, commande des actionneurs, affichage des états du dispositif et/ou attente d'une action de l'utilisateur. Heureusement, on peut résoudre ce problème de façon élégante avec la programmation de tâches basée sur des systèmes d'exploitation embarqués légers. FreeRTOS est un système d'exploitation à code source ouvert, largement diffusé et disponible pour de nombreuses plateformes à microcontrôleur. Le tandem très répandu ESP32 et EDI Arduino facilite particulièrement l'utilisation de FreeRTOS pour la programmation de tâches, car il est déjà intégré dans les bibliothèques de base. En fait, vous avez peut-être toujours utilisé FreeRTOS sans le savoir !

La plateforme ESP32 d'Espressif est un microcontrôleur passionnant pour les projets des *makers*. Avec ses capacités matérielles, son logiciel riche et son prix abordable, beaucoup le placent dans la catégorie « indispensable ».
Espressif fournit à la fois des environnements en modes natif et compatible Arduino. Le mode natif se nomme officiellement ESP-IDF (*Espressif IoT Development Framework*). Pour cet article, nous utiliserons l'environnement familier d'Arduino. La plupart des API (*Application Programming Interface*) sont disponibles pour les deux [1]. Certains de nos lecteurs savent peut-être qu'Espressif utilise le fameux système d'exploitation embarqué FreeRTOS pour implémenter les fonctions de bibliothèques de l'ESP32 pour le Wi-Fi, le Bluetooth et bien d'autres options du microcontrôleur. Dans cet article, nous verrons que FreeRTOS [2] et la programmation de tâches sont aussi employés pour les fonctions élémentaires `setup()` et `loop()` d'Arduino.

Nous allons tout d'abord décrire un simple projet de démonstration de l'ESP32 [3]. Puis nous jetterons un coup d'œil sous le capot ; nous réaliserons ensuite nos propres tâches pour la démonstration.

Application

Pour cet épisode de la série, nous développerons une application qui lit le convertisseur analogique-numérique (CA/N) à partir d'un potentiomètre et affiche la valeur sous forme de graphique à barres sur un écran OLED. De plus, en utilisant la modulation par largeur d'impulsion (MLI), nous ferons varier la luminosité d'une LED.
Cet article repose sur la carte Lolin ESP32 avec écran OLED intégré (**fig. 1**) [4]. Si vous avez une autre carte à ESP32 sans écran OLED compatible SSD1306, vous pouvez désactiver l'affichage OLED dans le programme et vous fier à la luminosité de la LED.

La LED et le potentiomètre seront câblés selon le schéma de la **figure 2**.
Pendant l'exécution de l'application, tourner le potentiomètre fait varier la tension d'entrée vue sur la broche GPIO36. Cette tension sera mesurée par une valeur sur 12 bits, soit un nombre de 0 à 4095. Tourner le potentiomètre (ou « potar ») à fond dans le sens horaire renverra une valeur de 4095 qui provoquera la pleine luminosité de la LED. Si l'inverse se produit, inversez les connexions externes du potar. La connexion centrale est le curseur du potar dont la tension va varier du potentiel de la masse jusqu'à 3,3 V. Faites attention à ne pas connecter le potar à l'alimentation de 5 V, car c'est supérieur à la tension maximale admissible par la broche GPIO.

Configuration du programme

Passons maintenant au logiciel. Le programme définit des macros en C pour vous faciliter sa reconfiguration. Elles figurent dans le **listage 1**. Mettez la macro `CFG_OLED` à zéro si vous n'utilisez pas d'écran (ceci permet d'ignorer les adresses et macros I2C). On a choisi la valeur de la macro `CFG_ADC_GPIO` pour utiliser ADC1 sur le canal 0 (GPIO36). Enfin, la LED a été configurée par `CFG_LED_GPIO` pour utiliser GPIO13.
Le code complet est disponible sur GitHub dans le sous-répertoire *freertos-tasks1* [3].

Initialisation

Ignorons les tâches pour le moment, le **listage 2** montre la fonction `setup()` de notre dispositif. Il s'agit d'une initialisation typique dans le cadre d'une d'application Arduino.
L'écran n'est initialisé que lorsqu'il est compilé (configuré). Après cela, le CA/N est configuré pour lire des valeurs sur 12 bits avec la fonction Arduino `analogReadResolution()`. On appelle la fonction `analogSetAttenuation()` pour lire une valeur entre 0

Listage 1. Options de configuration.

```
// Mettre à zéro si vous n'utilisez pas d'écran SSD1306
#define CFG_OLED 1

// adresse I2C de l'écran SSD1306
#define CFG_OLED_ADDRESS 0x3C

// port GPIO de la ligne SDA I2C de l'écran
#define CFG_OLED_SDA 5

// port GPIO de la ligne SCL I2C de l'écran
#define CFG_OLED_SCL 4

// largeur de l'écran en pixels
#define CFG_OLED_WIDTH 128

// hauteur de l'écran en pixels
#define CFG_OLED_HEIGHT 64

// port GPIO pour l'entrée CA/N
#define CFG_ADC_GPIO 36

// port GPIO pour la LED à MLI
#define CFG_LED_GPIO 13
```

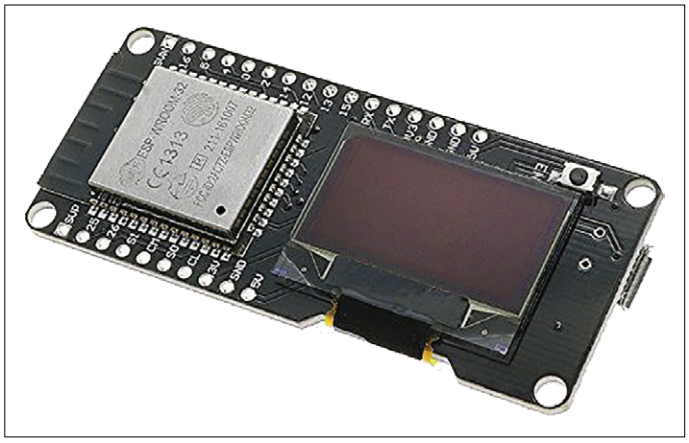

Figure 1. Lolin ESP32 avec écran OLED.

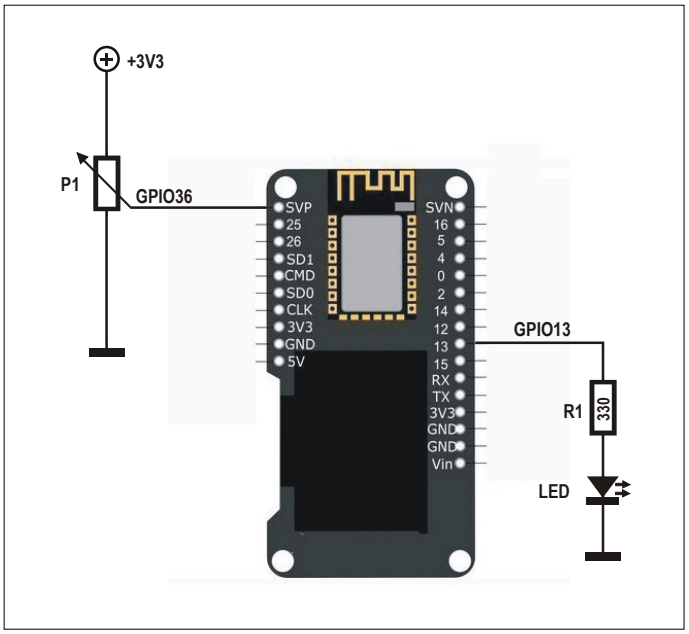

Figure 2. Module Lolin ESP32 équipé avec OLED et LED.

Listage 2. La fonction setup().

```
void setup() {

#if CFG_OLED
  display.init();
  display.clear();
  display.setColor(WHITE);
  display.display();
#endif

  analogReadResolution(12);
  analogSetAttenuation(ADC_11db);

  pinMode(gpio_led,OUTPUT);
  ledcAttachPin(gpio_led,0);
  ledcSetup(0,5000,8);
}
```

Listage 3. La fonction graphique à barres.

```
#include "SSD1306.h"
...
void barGraph(unsigned v) {
  char buf[20];
  unsigned width, w;

  snprintf(buf,sizeof buf,"ADC %u",v);
  width = disp_width-2;
  w = v * width / 4095;
  display.fillRect(1,38,w,disp_height-2);
  display.setColor(BLACK);
  display.
fillRect(w,38,disp_width-2,disp_height-2);
  display.fillRect(1,1,disp_width-2,37);
  display.setColor(WHITE);
  display.drawLine(0,38,disp_width-2,38);
  display.
drawRect(0,0,disp_width-1,disp_height-1);
  display.setTextAlignment(TEXT_ALIGN_CENTER);
  display.setFont(ArialMT_Plain_24);
  display.drawString(64,5,buf);
  display.display();
}
```

et 3,3 V. Le CA/N comporte un amplificateur interne, il est donc important de le configurer pour utiliser la plage correcte. Ensuite, la broche GPIO pour la LED est configurée en sortie et en mode MLI par des appels à `ledcAttachPin()` et `ledcSetup()`.

Affichage graphique à barres

Avec l'écran OLED SSD1306, l'application va dessiner un graphique à barres et y afficher la valeur du CA/N (voir le **listage 3**).
La fonction `barGraph()` reçoit une valeur d'entrée `v` qui est la valeur du CA/N, et la formate en un message court dans le tableau `buf` avec `snprintf()`. Certains rectangles sont dessinés en noir et d'autres en blanc pour représenter le graphique en barres. L'appel à la méthode `display.drawString()` place aussi le texte formaté sur l'écran. Enfin, l'appel à la méthode `display.display()` recopie l'image mémoire de l'écran dans le contrôleur OLED du dispositif.

Listage 4. Code d'une boucle Arduino typique.

```
void loop() {
  uint adc = analogRead(ADC_CH);

#if CFG_OLED
  barGraph(adc);
#endif

  printf("ADC %u\n",adc);
  ledcWrite(0,adc*255/4095);
  delay(50);
}
```

Tableau 1. Au démarrage de l'Arduino.

nom de la tâche	n° de la tâche	priorité	pile	UC
loopTask	12	1	5188	1
Tmr Svc	8	1	1468	0
IDLE1	7	0	592	1
IDLE0	6	0	396	0
ipc1	3	24	480	1
ipc0	2	24	604	0
esp_timer	1	22	4180	0

La fonction *loop()*

Si nous écrivions un programme Arduino normal, nous coderions une boucle comme celle du **listage 4**.
Dans ce code, les étapes sont simples :

- Lire la valeur sur 12 bits du CA/N (récupération d'une valeur de 0 à 4095).
- L'afficher sur le graphique à barres (s'il est configuré).
- Imprimer « ADC » et la valeur sur le moniteur série.
- Ajuster la luminosité de la LED en changeant le paramètre de MLI.
- Et attendre 50 tops d'horloge.

C'est un programme volontairement simple. Mais si cette application était compliquée, il faudrait que vous la décomposiez en « tâches » plus petites.

Quelles sont les tâches ?

Avec une unité centrale (UC) ESP32 à double cœur, deux programmes peuvent s'exécuter *simultanément*. C'est formidable avec un microcontrôleur ! Chaque UC fonctionne avec son propre compteur ordinal et autres registres pour exécuter les instructions. Cela représente deux *threads* de commande. Alors qu'une UC à simple cœur n'exécuterait qu'un seul *thread* de commande à un instant donné.
Pour permettre à plus de deux programmes de s'exécuter de façon *concomittante*, on utilise un artifice d'ordonnancement pour suspendre un programme et reprendre les autres. C'est le principal travail de FreeRTOS. Cet ordonnancement a lieu si rapidement qu'il donne l'impression qu'on exécute plusieurs programmes à la fois. Mais à tout moment, il n'y a pas plus de deux programmes qui tournent simultanément dans l'UC à double cœur. Pensez à ce terme de « programme » (utilisé à tort) comme une tâche.
Pour gérer l'exécution simultanée de plusieurs tâches, l'ordonnanceur de FreeRTOS sauvegarde les registres de la tâche courante et restaure ceux de la tâche suivante à exécuter. De cette façon, plusieurs tâches peuvent s'exécuter indépendamment. En plus du compteur ordinal pour chaque tâche, on doit sauvegarder et restaurer son pointeur de pile. C'est indispensable, car les programmes en C/C++ sauvegardent dans la pile des variables et des adresses de retour de fonctions. Chaque tâche doit avoir sa propre pile privée.
Dans l'ESP32, toutes les tâches s'exécutent dans le même espace mémoire (à la différence du Raspberry Pi par exemple). Pour cette raison, les tâches doivent être programmées avec soin pour ne pas corrompre la mémoire utilisée par d'autres tâches. D'autres

considérations s'appliquent pour le fonctionnement d'un noyau à multiples UC, mais gardons cette discussion pour une autre fois.

Tâches Arduino

Lorsque votre programme Arduino pour ESP32 démarre, est-il dans une tâche ? Absolument ! En plus de votre propre tâche au démarrage, il y a d'autres tâches de FreeRTOS qui s'exécutent. Certaines d'entre elles fournissent des services tels que des temporisations, TCP/IP, Bluetooth, etc. Des tâches supplémentaires peuvent être démarrées lorsque vous envoyez une requête de service à l'ESP32.

Le **tableau 1** montre un exemple de tâches FreeRTOS qui s'exécutent lorsque les fonctions Arduino `setup()` et `loop()` sont invoquées. La tâche nommée `loopTask` est votre tâche Arduino principale.

La colonne intitulée *Pile* représente le nombre d'octets de pile inutilisés de FreeRTOS. Nous verrons plus tard comment prévoir les tailles de pile.

Le tableau est trié dans l'ordre chronologique inverse, par numéro de tâche. Votre tâche principale (`loopTask`) est la dernière créée. Les numéros de tâches manquants suggèrent que certaines tâches ont été créées et arrêtées après avoir fini leur travail. La colonne *Priorité* indique les priorités affectées aux tâches, zéro étant la plus faible. Pour l'instant, sachez que les priorités fonctionnent dans FreeRTOS un peu différemment par rapport à Linux par exemple.

Enfin, les tâches sont réparties entre l'UC 0 et l'UC 1. Espressif met les tâches du système dans l'UC 0, tandis que les tâches applicatives utilisent l'UC 1. Ceci permet de ne pas perturber les services pour TCP/IP, Bluetooth, etc., sans que votre application prenne des précautions particulières. Malgré cette convention, il est toujours possible de créer des tâches supplémentaires dans l'une ou l'autre UC.

Démarrage d'un Arduino

Il est bon de savoir comment les programmes Arduino pour ESP32 s'initialisent avant l'appel à *setup()*. Prenez l'extrait de programme simplifié du **listage 5** (les éléments du temporisateur du chien de garde ont été omis).

Sur cet exemple, nous pouvons remarquer plusieurs choses intéressantes :

- Notez qu'il s'agit d'un démarrage en C++ (malgré la déclaration en « C » *extern* de `app_main()`).
- Certaines initialisations d'Arduino sont réalisées par `initArduino()`.
- La tâche `loop` est créée et exécutée par l'appel à `xTaskCreatePinnedToCore()`.

La tâche `loopTask` est créée par l'appel à `xTaskCreatePinnedToCore()` avec de nombreux arguments. Le premier argument est l'adresse de la fonction à exécuter pour la tâche (`loopTask()`). Lorsque la tâche est créée, la fonction `loopTask()` s'exécute alors, invoquant d'abord `setup()`, puis `loop()` depuis une boucle « for » sans fin.

Le second argument est une chaîne de caractères qui décrit le nom de la tâche avec le type `string` en C (« loopTask »). Le troisième argument, la taille de la pile, est spécifié à 8192 *octets*. Notez que cela diffère du FreeRTOS d'origine sur d'autres plateformes, où elle s'exprime en mots de 4 octets. Lorsque la tâche est créée, l'allocation est faite à partir du *tas* et affectée à la tâche avant l'appel à la fonction. C'est une bonne partie de la SRAM qui serait gaspillée si la tâche principale n'était pas utilisée.

Listage 5. Démarrage simplifié d'un Arduino à ESP32.

```
void loopTask(void *pvParameters) {
  setup();
  for (;;) {
    loop();
  }
}

extern "C" void app_main() {
  initArduino();
  xTaskCreatePinnedToCore(
    loopTask,       // function to run
    "loopTask",     // Name of the task
    8192,           // Stack size (bytes!)
    NULL,           // No parameters
    1,              // Priority
    &loopTaskHandle, // Task Handle
    1);             // ARDUINO_RUNNING_CORE
}
```

Toutes les tâches FreeRTOS acceptent un pointeur vide comme argument. Ici la valeur n'est pas utilisée et fournie comme *NULL*. Le cinquième argument est la priorité FreeRTOS à affecter à la tâche, qui est de 1 dans cet exemple. Cela demande plus de discussion, donc utilisez 1 jusqu'à ce que les priorités de FreeRTOS soient expliquées.

Après l'argument priorité vient un pointeur vers une variable qui recevra l'identificateur de la tâche. Il n'est pas utilisé ici et pourra être fourni comme *NULL*. Le dernier argument spécifie le cœur d'UC d'exécution. Cela peut être 0 ou 1 pour le double ESP32. L'UC 1 est utilisée pour démarrer la tâche principale. Pour les dispositifs à cœur unique, cela ne peut être que 0.

Création d'une tâche

Supposons que notre application soit compliquée et que nous devions scinder le code en deux tâches. Nous avons déjà `loopTask`, qui appelle `loop()` sans cesse. Pour profiter de l'espace de pile qui lui est alloué, nous devrions normalement y coder la partie la plus consommatrice de pile du programme. Pour notre démonstration de tâche, faisons ce qui suit dans la fonction `loop()` :

- Lire la valeur sur 12 bits du CA/N.
- Imprimer la valeur sur le moniteur série.
- Envoyer la valeur du CA/N à la seconde tâche.
- Attendre 50 tops d'horloge et revenir (de `loop()`).

Ce qui reste à faire à la seconde tâche :

- Récupérer la valeur du CA/N depuis la fonction `loop()` (de la tâche `loopTask`).
- Afficher la valeur du CA/N sur le graphique à barres (s'il est configuré).
- Modifier la commande MLI pour la luminosité de la LED.

Le seul endroit pertinent pour créer la seconde tâche est dans la fonction `setup()` car cette création ne doit être faite qu'une fois. Ajoutons donc à notre fonction `setup()` :

```
void setup() {

#if CFG_OLED
  display.init();
  display.clear();
  display.setColor(WHITE);
  display.display();
#endif

  analogReadResolution(12);
  analogSetAttenuation(ADC_11db);

  pinMode (gpio_led,OUTPUT);
  ledcAttachPin(gpio_led,0);
  ledcSetup(0,5000,8);
...
  xTaskCreatePinnedToCore(
    dispTask,   // Display task
    "dispTask", // Task name
    2048,       // Stack size (bytes)
    NULL,       // No parameters
    1,          // Priority
    NULL,       // No handle returned
    1);         // CPU 1
}
```

Cet appel va démarrer une nouvelle tâche nommée `dispTask`, avec une pile de 2048 octets. Nous l'avons affectée à l'UC 1, avec une priorité de 1. Elle va exécuter la fonction `dispTask()` (qui reste à définir). Comme la tâche va s'exécuter sur là même UC et à la même priorité que notre `loopTask`, l'UC sera partagée avec `loopTask` et toutes les autres tâches de priorité 1 sur cette UC.

Files

Il manque encore un ingrédient – comment envoyer la valeur du CA/N d'une tâche à une autre ? Vous pourriez essayer de passer par la mémoire, ce qui pourrait fonctionner pour des tâches sur la même UC. Mais les choses se compliquent pour passer de l'information d'une UC à une autre. La file de FreeRTOS est la solution à notre problème.
La file permet à une tâche d'y « pousser » un élément de façon « atomique ». De la même façon, elle permet au récepteur de récupérer cet élément de façon atomique. Atomique signifie que l'élément ne peut pas être poussé ou récupéré partiellement. Avec une UC à double cœur qui exécute simultanément deux instructions, il y a des problèmes de synchronisation et d'accès concurrentiel. Le mécanisme de file a des dispositions spéciales pour que cela se produise au niveau atomique.
Pour créer une file FreeRTOS, examinons le listage ci-dessous :

```
static QueueHandle_t qh = 0;
...
  qh = xQueueCreate(8,sizeof(uint));
```

Le premier argument est la profondeur de la file (le nombre maximum d'entrées possibles dans la file). Le second argument spécifie la taille de chaque élément de la file. Ici c'est la taille d'une valeur `uint` en octets. La valeur retournée est l'identificateur de la file. Cette étape doit avoir lieu juste avant la création de `dispTask` dans `setup()`, de sorte qu'elle puisse immédiatement en disposer.
Intéressons-nous maintenant au remplissage de cette file :

```
void loop() {
  uint adc = analogRead(ADC_CH);

  printf("ADC %u\n",adc);
  xQueueSendToBack(qh,&adc,portMAX_DELAY);
  delay(50);
}
```

La mise à jour du graphique à barres et de la LED aura lieu dans notre nouvelle fonction `dispTask()`, qui reste encore à définir. Toutefois, la fonction `loopTask()` lit la valeur du CA/N dans `adc` puis l'imprime sur le moniteur série. Ensuite, la valeur est poussée au *fond* de la file avec la fonction `xQueueSendToBack()`. Nous avons fourni l'identificateur de la file en premier argument. La valeur à mettre en file est fournie par un pointeur comme second argument. Le dernier argument indique le temps d'attente en cas de file pleine. En spécifiant la macro `portMAX_DELAY`, nous indiquons que nous voulons bloquer l'exécution jusqu'à ce qu'il y ait de la place dans la file.

Tâche d'affichage

Examinons maintenant la tâche d'affichage dans le code qui suit :

```
void dispTask(void *arg) {
  uint adc;

  for (;;) {
    xQueueReceive(qh,&adc,portMAX_DELAY);
    barGraph(adc);
    ledcWrite(0,adc*255/4095);
  }
}
```

Cette fonction, comme toutes les fonctions de tâche, reçoit un argument de type pointeur lorsqu'elle démarre. Il n'est pas utilisé ici, et vaut *NULL* du fait de la façon dont la tâche a été créée.

Le corps principal de la tâche est une boucle `for` de réception depuis la file. Ici `xQueueReceive()` renvoie la valeur de la donnée en file, ou attend indéfiniment si la file est vide (la

Liens

[1] API de référence pour ESP32 : https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/
[2] Page d'accueil FreeRTOS : http://www.freertos.org
[3] Code source du projet : https://github.com/ve3wwg/esp32_freertos/blob/master/freertos-tasks1/freertos-tasks1.ino
[4] Carte ESP32 Lolin avec OLED : www.elektor.com/lolin-esp32-oled-module-with-wifi

valeur du troisième argument est `portMAX_DELAY`). Une fois qu'une valeur est reçue depuis la file, l'affichage du graphique à barres est mis à jour et la valeur MLI de la LED est modifiée.
Remarquez qu'il n'y a pas besoin d'appeler `delay()` dans cette tâche d'affichage. La raison en est que la tâche va se bloquer en attente dans la fonction `xQueueReceive()` lorsque la file est vide. C'est la tâche émettrice qui fixe le rythme d'exécution.

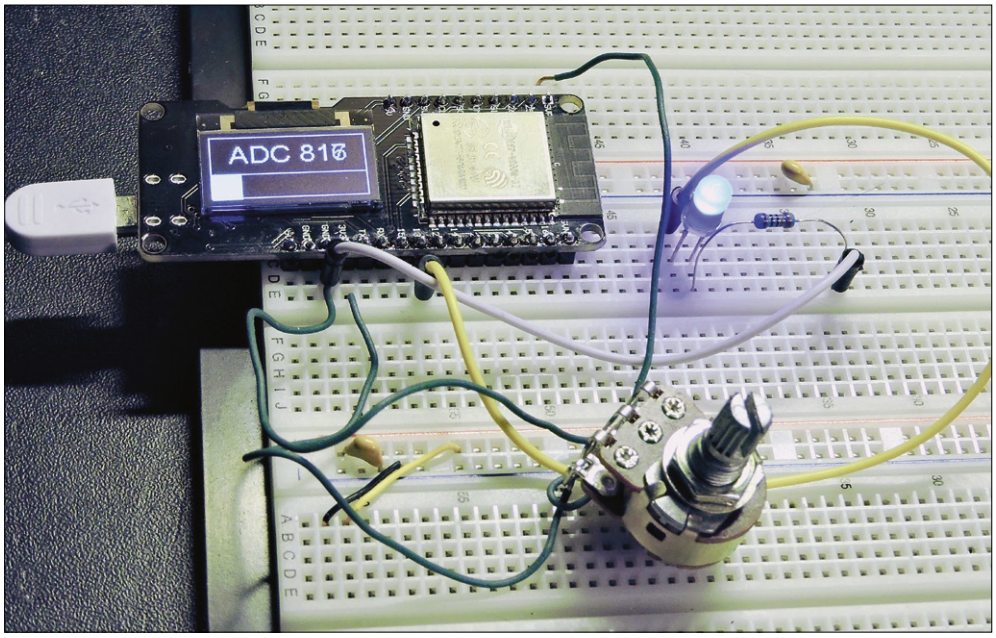

Figure 3. Exécution de la démo.

Exécution de la démo

Une fois câblé et programmé, vous devriez voir l'affichage du graphique à barres et la LED s'allumer (voir la **fig. 3**). Si la LED est éteinte, tourner le potar devrait l'allumer. Si vous avez compilé le programme sans l'écran, alors démarrez le moniteur série et cherchez des lignes de sortie de la forme « ADC 3420 » etc. Tourner le potar dans le sens horaire devrait augmenter la luminosité de la LED à MLI et la diminuer dans l'autre sens. Simultanément les valeurs du CA/N devraient augmenter avec une rotation dans le sens horaire et diminuer dans l'autre sens.

En résumé

Nous avons parcouru beaucoup de chemin dans cet article. Vous avez vu la procédure de démarrage de l'ESP32, depuis `app_main()`, jusqu'à `setup()` *et* `loop()`. La création de la tâche principale nommée `loopTask` a été couverte. Dans la démo, nous avons codé notre propre tâche supplémentaire dans le but de commander l'écran OLED et la LED à MLI, et de communiquer des données à une file. Ce code s'exécute bien indépendamment de la tâche principale.
De plus, nous nous sommes servis de la tâche principale en sachant qu'une bonne quantité d'espace de pile lui était attribuée (depuis le tas). Dans un prochain épisode, nous discuterons de la façon de déterminer l'utilisation de la pile pour les nouvelles tâches que vous allez créer.

Cet exemple était simpliste en termes de complexité. Mais reconnaissez que les tâches rendent les applications complexes plus simples en décomposant le problème. Imaginez comment un contrôleur MIDI pourrait être découpé en tâches d'émission, de réception et de commande. La tâche de réception est alors disponible pour recevoir des données sur la ligne série et les décoder sous forme d'événements à exécuter par la tâche de commande. Certains événements de commande peuvent à leur tour déclencher l'émission de données série par la tâche émettrice. C'est un élégant partage du travail. Non seulement cela fait de la maintenance du code un plaisir, mais cela améliore l'exactitude du programme.

(190182-03 – version française : Denis Lafourcade)