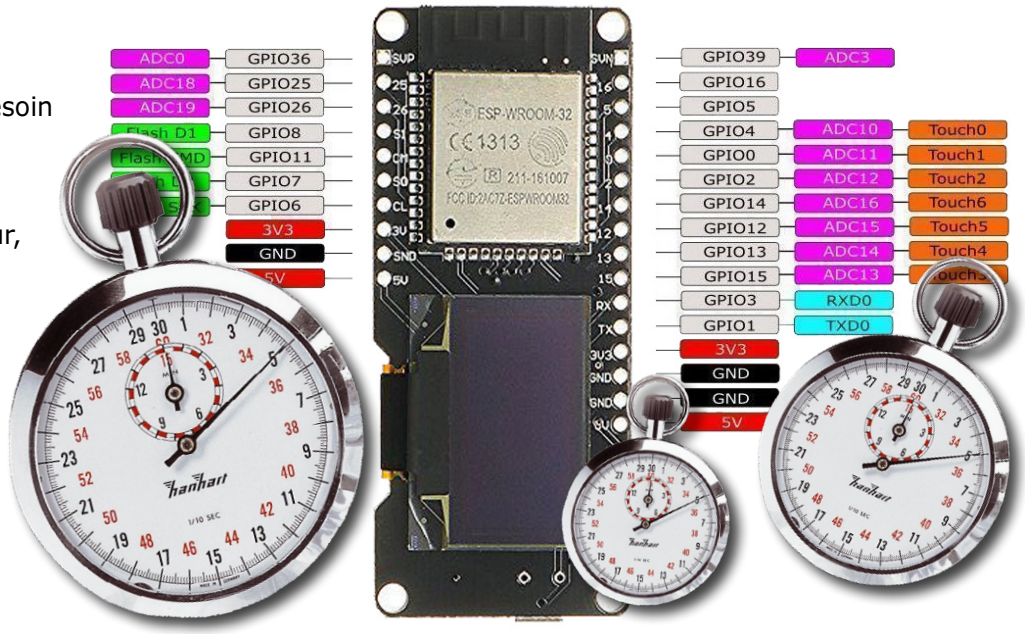
multitâche en pratique avec l'ESP32 (3)

Temporisateurs logiciels

Warren Gay (Canada)

Les développeurs sur microcontrôleur ont souvent besoin de temporisateurs, pour les retransmissions et autres procédures de reprise sur erreur, la suppression des rebonds des poussoirs, ou un simple clignotement. L'ESP32 et l'EDI Arduino fournissent des temporisateurs logiciels qui utilisent la bibliothèque FreeRTOS intégrée. Intéressons-nous à la fonction de temporisation.



Pour montrer comment fonctionnent les temporisateurs de FreeRTOS, notre démonstration visera à construire une classe de LED d'alerte. Il ne s'agit néanmoins pas d'une LED clignotante ordinaire. Lorsqu'elle est activée, elle clignote rapidement cinq fois puis s'éteint pour le reste de la période et recommence. Visuellement, cette alerte attirera l'attention comme le ferait une sonnerie de téléphone.

La classe `AlertLED` va utiliser en interne un temporisateur logiciel FreeRTOS pour chaque pilote de LED. S'agissant d'une classe, vous pourrez en ajouter autant que nécessaire, sans surcharger la mémoire de l'ESP32. Le **listage 1** montre le programme complet [2]. La ligne 100 montre la création d'une instance de LED d'alerte avec une simple instruction de déclaration C++ :

```
static AlertLED alert1(GPIO_LED,1000);
```

Le nom de l'instance de classe est ici `alert1`, elle utilise la LED spécifiée (GPIO 12 en ligne 5), et la période de clignotement est de 1000 ms. Nous pourrions facilement en ajouter une autre pour le GPIO 13, juste en ajoutant la ligne :

```
static AlertLED alert2(13,1000);
```

Nous consacrerons cette démonstration à un seul indicateur, mais la structure de classe C++ en facilite l'usage.

La classe AlertLED

La définition de la classe est reproduite ci-dessous par commodité. Les données membres de la classe comprennent le membre `thandle` (ligne 11), qui est un index vers le temporisateur FreeRTOS (type `TimerHandle_t`). Au début il prend la valeur `nullptr` (NULL en langage C). La donnée membre `state` contiendra l'état de la LED (ligne 12), active à l'état haut (`true` = on). La donnée membre `count` (ligne 13) sera utilisée comme un sous-état dans la fonction de retour, comme expliqué plus loin. Finalement, les membres `period_ms` et `gpio` définissent la période de clignotement en ms et le GPIO à piloter pour la LED (lignes 14 et 15).

```
0010: class AlertLED {
0011:   TimerHandle_t     thandle = nullptr;
0012:   volatile bool     state;
0013:   volatile unsigned count;
0014:   unsigned          period_ms;
0015:   int               gpio;
0016:
0017:   void reset(bool s);
0018:
0019: public:
0020:   AlertLED(int gpio,unsigned period_ms=1000);
0021:   void alert();
```

Listage 1 : Le programme alertled.ino [2].

```
0001: // alertled.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // La LED est active à l'état haut
0005: #define GPIO_LED      12
0006:
0007: //
0008: // Classe AlertLED pour pilotage de la LED
0009: //
0010: class AlertLED {
0011:   TimerHandle_t     thandle = nullptr;
0012:   volatile bool     state;
0013:   volatile unsigned count;
0014:   unsigned          period_ms;
0015:   int               gpio;
0016:
0017:   void reset(bool s);
0018:
0019: public:
0020:   AlertLED(int gpio,unsigned period_ms=1000);
0021:   void alert();
0022:   void cancel();
0023:
0024:   static void callback(TimerHandle_t th);
0025: };
0026:
0027: //
0028: // Constructeur :
0029: //  gpio      broche GPIO pour activer la LED
0030: //  period_ms  période complète en ms
0031: //
0032: AlertLED::AlertLED(int gpio,
          unsigned period_ms) {
0033:  this->gpio = gpio;
0034:  this->period_ms = period_ms;
0035:  pinMode(this->gpio,OUTPUT);
0036:  digitalWrite(this->gpio,LOW);
0037: }
0038:
0039: //
0040: // Méthode interne
         pour réinitialiser les valeurs
0041: //
0042: void AlertLED::reset(bool s) {
0043:  state = s;
0044:  count = 0;
0045: digitalWrite(this->gpio,s?HIGH:LOW);
0046: }
0047:
0048: //
0049: // Méthode pour démarrer l'alerte :
0050: //
0051: void AlertLED::alert() {
0052:
0053:  if ( !thandle ) {
0054:    thandle = xTimerCreate(
0055:      "alert_tmr",
0056:      pdMS_TO_TICKS(period_ms/20),
0057:      pdTRUE,
0058:      this,
0059:      AlertLED::callback);
0060:    assert(thandle);
0061:  }
0062:  reset(true);
0063:  xTimerStart(thandle,portMAX_DELAY);
0064: }
0065:
0066: //
0067: // Méthode pour arrêter une alerte :
0068: //
0069: void AlertLED::cancel() {
0070:  if ( thandle ) {
0071:    xTimerStop(thandle,portMAX_DELAY);
0072:    digitalWrite(gpio,LOW);
0073:  }
0074: }
0075:
0076: // Méthode statique, agissant comme callback
0077: // du temporisateur :
0078: //
0079: void AlertLED::callback(TimerHandle_t th) {
0080:  AlertLED *obj = (AlertLED*)
         pvTimerGetTimerID(th);
0081:
0082:  assert(obj->thandle == th);
0083:  obj->state ^= true;
0084:  digitalWrite(obj->gpio,obj->state?HIGH:LOW);
0085:
0086:  if ( ++obj->count >= 5 * 2 ) {
0087:    obj->reset(true);
0088:    xTimerChangePeriod(th,pdMS_TO_TICKS
           (obj->period_ms/20),portMAX_DELAY);
0089:  } else if ( obj->count == 5 * 2 - 1 ) {
0090:    xTimerChangePeriod(th,
0091:      pdMS_TO_TICKS(obj->period_ms/20+
            obj->period_ms/2),
0092:      portMAX_DELAY);
0093:    assert(!obj->state);
0094:  }
0095: }
0096:
0097: //
0098: // Objets globaux
0099: //
0100: static AlertLED alert1(GPIO_LED,1000);
0101: static unsigned loop_count = 0;
0102:
0103: //
0104: // Initialisation:
0105: //
0106: void setup() {
0107:  // delay(2000);
       // pour permettre la connexion USB
0108:  alert1.alert();
0109: }
0110:
0111: void loop() {
0112:  if ( loop_count >= 70 ) {
0113:    alert1.alert();
0114:    loop_count = 0;
0115:  }
0116:
0117:  delay(100);
0118:
0119:  if ( ++loop_count >= 50 )
0120:    alert1.cancel();
0121: }
```

```
0022:   void cancel();
0023:
0024:   static void callback(TimerHandle_t th);
0025: };
```

La classe comporte aussi quelques appels à des méthodes publiques comme :

- `alert()` - active l'indicateur à LED (ligne 21)
- `cancel()` - désactive l'indicateur à LED (ligne 22)

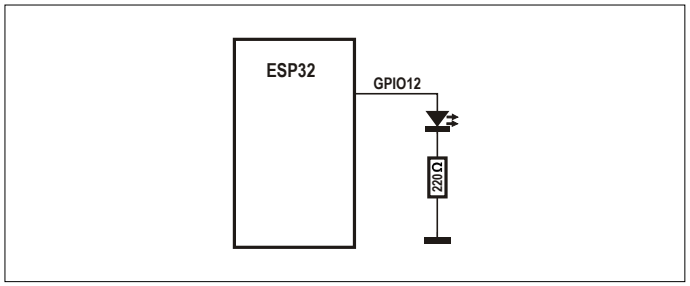


Figure 1. Le câblage du programme de démo *alertled.ino*.

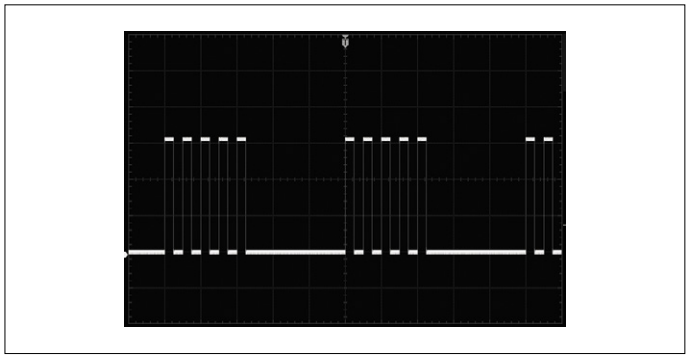


Figure 2. Le signal de pilotage de la LED pour *alertled.ino*, en abscisse 200 ms/div.

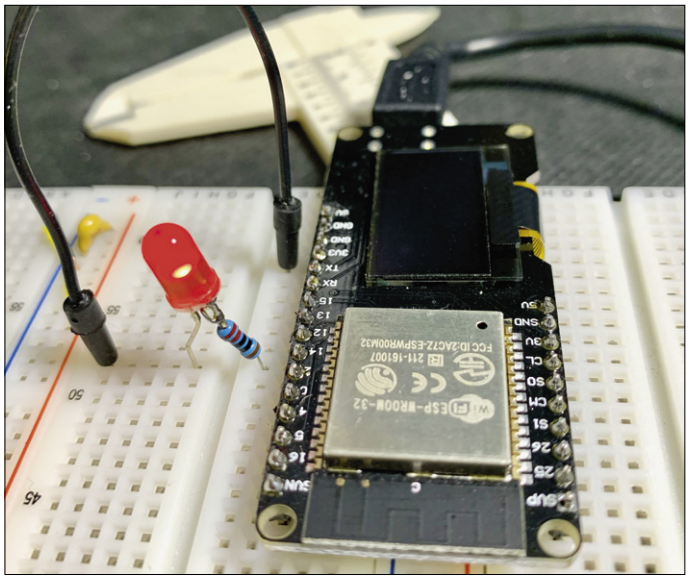


Figure 3. ESP32, LED et résistance de 220 Ω câblés pour la démo *alertled.ino*.

Il y a aussi une méthode C++ de type `static` nommée `callback()` à la ligne 24. Nous y reviendrons.

La méthode AlertLED::alert()

Les méthodes `alert()` et `callback()` sont les parties les plus intéressantes de la classe. Nous avons vu que l'index a été initialisé à `nullptr`. Lorsque la méthode `alert()` est appelée pour la première fois, elle va trouver que la valeur est nulle en ligne 53, ce qui entraîne la création du temporisateur.
Le temporisateur logiciel est créé avec `xTimerCreate()` aux lignes 54 à 59, ce qui retourne l'index du temporisateur nouvellement créé. La macro `assert()` à la ligne 60 s'assure du succès de la création. Les arguments de `xTimerCreate()` sont :

1. Un nom convivial pour notre temporisateur (ici c'est juste « `alert_tmr` »). FreeRTOS ne l'utilise que pour quelques fonctions de recueils statistiques. Il doit normalement être unique mais peut ne pas l'être si vous ne faites pas de recherches par nom.
2. Une période de temporisation en tops d'horloge système (convertis en ms à la ligne 56 avec la macro `pdMS_T0_TICKS()`).
3. Une indication si le temporisateur est simple (valeur `pdFALSE`) ou réarmable (valeur `pdTRUE`). Notre exemple crée un temporisateur réarmable (ligne 57).
4. `this` est un « identifiant de temporisateur » dans la terminologie FreeRTOS. On pourrait le voir comme un « paramètre de donnée utilisateur ». C'est un pointeur non typé, qui dans cet exemple est l'adresse de l'objet de classe qui utilise le mot réservé `this` en C++ (ligne 58).
5. L'adresse de la fonction de retour. Nous fournissons à la ligne 59 le nom de la méthode statique `AlertLED::callback()`. C'est la fonction qui sera appelée lorsque le temporisateur déclenche.

Certaines données membres sont réinitialisées par l'appel à la méthode interne `reset()` à la ligne 62, puis vient le démarrage du temporisateur à la ligne 63 avec `xTimerStart()`. Tous les temporisateurs sont créés « inactifs » jusqu'à ce qu'ils soient démarrés/redémarrés. Une fois démarrés, ils passent dans l'état « en cours ».

La méthode statique AlertLED::callback()

La classe `AlertLEd` définit un appel à une méthode statique appelée `callback()`. Pour qui n'est pas familier avec le C++, cela signifie que cette méthode est juste une fonction. Elle n'a donc pas de pointeur d'objet propre (pas de pointeur « this »). C'est la même chose qu'une fonction C excepté son nom bizarre en C++ et le fait qu'elle bénéficie d'un accès spécial aux éléments internes de l'objet de classe.
Le temporisateur logiciel requiert un argument de type `TimerHandle_t` (ligne 79). Cela nous donne accès à l'index du temporisateur qui est déclenché mais il nous faut plus d'information — dans ce cas l'instance de LED qui doit être pilotée. La ligne 80 l'obtient avec la valeur du « Timer ID » - l'adresse de l'instance (`alert1`) de la classe (LED) avec l'appel à `pvTimerGetTimerID()`, qui nécessite l'index du temporisateur. Nous avons en retour la valeur fournie à la ligne 58, lorsque le temporisateur a été créé. Nous pouvons maintenant accéder à l'objet LED et à ses membres par la variable `obj`.
Les lignes 83 et 84 basculent le GPIO spécifié pour la LED.

Lorsque la valeur de `count` est incrémentée (ligne 86), et jusqu'à ce qu'elle atteigne 9, on ressort simplement de la fonction. Ceci produit le clignotement rapide simple pour la première moitié de la période d'alerte.
Lorsque `count` atteint 9, les lignes 90 à 93 sont exécutées. La fonction `xTimerChangePeriod()` modifie la période de temps utilisée par le temporisateur que nous avons créé. Notez l'ajout de period_ms/20 à period_ms/2 pour compenser la dernière moitié d'extinction du cycle de clignotement rapide.
Finalement, au dixième passage dans la fonction, les lignes 87 et 88 sont exécutées. Les données membres sont réinitialisées par appel à la méthode interne `reset()` (ligne 87), et la période du temporisateur rétablie pour un clignotement rapide à la ligne 88. Comme la donnée membre `count` est remise à zéro, le cycle complet recommence. Le caractère réarmable du temporisateur maintiendra la répétition du scénario jusqu'à l'appel de `AlertLED::cancel()`.

La méthose AlertLED::cancel()

On utilise la méthode `AlertLED::cancel()` pour arrêter le clignotement de l'alerte. Pour cela, on appelle `xTimerStop()` (ligne 71), et la LED est désactivée en écrivant LOW sur le GPIO de la LED à la ligne 72.

Nécessité de *volatile*

Le lecteur attentif aura noté que nous avons utilisé le mot clé C `volatile` aux lignes 12 et 13 de la définition de classe. Pourquoi est-ce nécessaire ? Ceci informe le compilateur que chaque fois qu'il accède aux valeurs des données membres `state` ou `count`, il doit aller directement à la mémoire de l'instance de classe (`alert1` dans notre démo) plutôt que de se fier à une valeur qui pourrait subsister dans un registre. Il y a deux tâches qui interagissent avec notre classe :

- la classe démon de FreeRTOS (via la fonction de retour)
- votre tâche qui invoque les méthodes `alert()` or `cancel()` de la classe.

Si ces valeurs n'étaient pas marquées comme *volatile*, une des tâches ou les deux pourraient utiliser des valeurs qui sont encore dans un registre, car c'est normalement le plus efficace. Toutefois, les valeurs en mémoire peuvent avoir été modifiées par *l'autre* tâche. Pour cette raison le mot clé *volatile* force le compilateur à ignorer cette possible optimisation et à générer à la place un code qui va directement chercher les valeurs en mémoire.

La démonstration

Pour une simple démo *alertled.ino*, nous pouvons de nouveau utiliser le Lolin ESP32 OLED Display Module. La **fig. 1** montre le câblage. La LED est câblée selon la configuration active-haut. Pour démontrer l'activation et l'annulation de la classe `AlertLED`, la fonction `loop()` maintient une variable de comptage nommée `loop_count` (ligne 101). La fonction `setup()` active initialement la LED, mais lorsque le décompte de boucle atteint 50, l'alerte est annulée à la ligne 120. Lorsque plus tard le décompte atteint 70, l'alerte est réactivée à la ligne 113 et le compteur est remis à zéro.
Une fois le programme flashé et exécuté sur l'ESP32, vous devriez voir la LED d'alerte clignoter de façon à attirer l'attention. Elle s'éteindra quelques instants après, juste pour repartir de nouveau. La **fig. 2** monte le signal de pilotage de la LED. La **fig. 3** montre la configuration, avec une combinaison LED-résistance et un fil Dupont vers GND.

Limitations de FreeRTOS

Nous avons jusqu'ici appliqué l'API (Interface de Programmation d'Application) du temporisateur logiciel, sans trop parler de certaines limitations importantes de la fonction de retour. Parmi ces limitations :

- Ne jamais faire d'appel bloquant à l'intérieur d'une fonction de retour (comme `delay()` ou une mise en file bloquante par exemple).
- Évitez les longues durées d'exécution (cela perturbera l'API).
- Évitez d'utiliser trop d'espace de pile (l'Arduino Esp32 est à priori limité autour de 1500 octets). Des fonctions comme `printf()` et `snprintf()` exigent souvent une taille de pile considérable et devraient donc être évitées dans la fonction de retour.
- Certaines des fonctions API de FreeRTOS fonctionnent au moyen d'une file interne. C'est pourquoi notre programme de démo a utilisé `portMAX_DELAY` aux lignes 88 et 92. La profondeur de la file de temporisateur pour l'Arduino ESP32 étant de 10, il est peu probable qu'elle se remplisse, à moins que de nombreux temporisateurs soient actifs en même temps.

Résumé

Pourquoi ne pas simplement utiliser une tâche pour gérer chaque LED d'alerte ? La raison principale est que chaque tâche nécessite une pile et un bloc de contrôle (TCB). Si vous deviez allouer 32 indicateurs à LED, cela demanderait 32 x (356 + 1500) octets, soit 59392 octets ! Comparez cela avec l'approche à temporisateur FreeRTOS qui prend 40 octets pour chaque temporisateur, pour un total de 1200 octets (l'objet de type `StaticTimer_t` fait 40 octets). La pile est partagée avec la tâche démon FreeRTOS pour tous les temporisateurs (autrefois appelée tâche *timer service*).
Les applications peuvent mélanger les approches pour des besoins spécifiques, mais bien souvent un temporisateur FreeRTOS répond aux exigences. ⏮

(200071-03 VF Denis Lafourcade)

Liens

[1] Multitâche en pratique avec l'ESP32, Elektor Magazine 1/2020 : www.elektormagazine.fr/magazine/190182-03

[2] Code source du projet : https://github.com/ve3wwg/esp32_freertos/blob/master/alertled/alertled.ino