multitâche en pratique avec l'ESP32 (6)

Groupes d'événements

Warren Gay (Canada)

La synchronisation entre plusieurs tâches est souvent nécessaire lors de la conception d'applications avec *FreeRTOS*. Dans les précédents épisodes de cette série, nous avons examiné les notifications de sémaphores et de tâches. Cependant, celles-ci ne sont capables d'envoyer un événement que d'une tâche ou d'une routine d'interruption de service (ISR) à une seule tâche de réception. Lorsque vous avez besoin de diffuser un événement à plusieurs tâches, la capacité de groupe d'événements est la solution dont vous avez besoin.

Commençons par examiner un cas d'utilisation potentiel. Une boîte à rythmes MIDI doit produire quatre sons simultanés en réponse à une commande reçue par son canal série. Idéalement, toutes les tâches d'émission d'un son devraient dans ce cas commencer exactement en même temps pour éviter que l'oreille ne distingue plusieurs attaques décalées. La coordination pourrait être assurée par l'utilisation de quatre sémaphores distincts et en se fiant à la vitesse de l'unité centrale pour une livraison en temps voulu. Ce serait maladroit, surtout si le nombre de tâches devait augmenter. Les groupes d'événements sont la méthode d'implémentation préférée dans *FreeRTOS* pour coordonner des tâches multiples à partir d'une source unique.

Drapeaux des événements

FreeRTOS définit 24 drapeaux d'événement pour chaque objet de groupe d'événements créé (**fig. 1**). Ces drapeaux sont représentés dans le type de données C `EventBits_t`, un type de données de 32 bits dont 24 bits sont disponibles. Le bit 0 est le bit de poids le plus faible (LSB) des drapeaux disponibles. Les 8 bits de poids le plus fort sont *réservés* à l'usage interne de *FreeRTOS*.

La manière dont ces 24 bits sont attribués et utilisés par l'application est laissée à l'appréciation du programmeur. Dans cette démo, la tâche `loop()` produit un événement toutes les secondes qui se traduit par le démarrage synchrone de deux tâches avec des clignotements différents via les LED associées. Le bit 0 du groupe d'événements (valeur `0b0001`/valeur décimale 1) notifie une tâche nommée `blink2()` qui fait clignoter deux fois la LED1. Le bit 1 du groupe d'événements (valeur `0b0010`/valeur décimale 2) est affecté à la notification d'une tâche nommée `blink3()` qui fait clignoter trois fois la LED2.

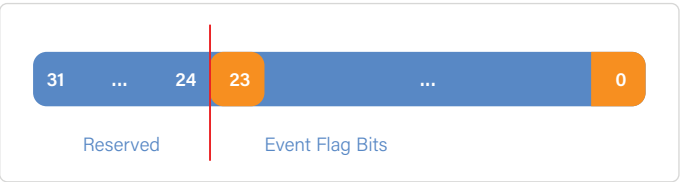

Figure 1. Drapeaux d'événement dans le type de données C EventBits_t.

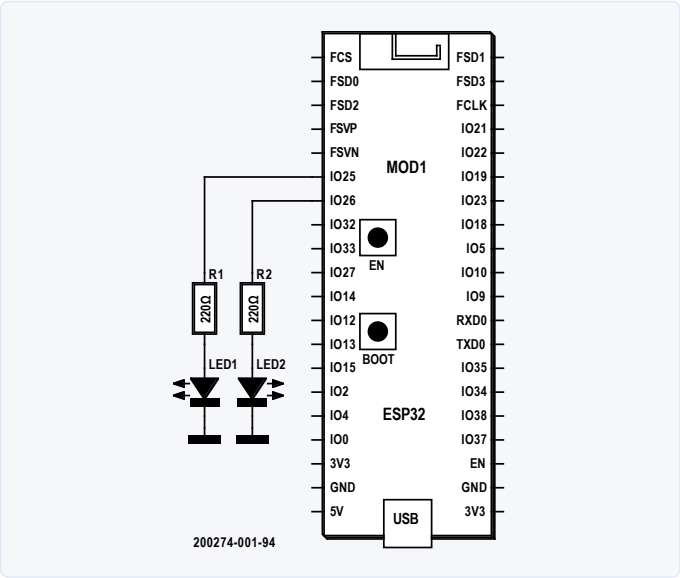

Figure 2. Le schéma du programme de démonstration evtgrp.ino.

Programme de démonstration

Avant de nous plonger dans le code, passons en revue ce que le programme de démonstration espère accomplir. Deux LED sont pilotées par les broches 25 et 26 du GPIO (lignes 5 et 6) dans la configuration haute active (**fig. 2**). Ces GPIO sont configurés dans la fonction `setup()` en tant que sorties (lignes 62 à 65). La fonction task `blink2()` commande la LED1 en la faisant clignoter deux fois avant d'attendre le prochain événement (lignes 19 à 25). Task `blink3()` commande la LED2 en la faisant clignoter trois fois avant d'attendre le prochain événement (lignes 41 à 47). Les deux tâches sont identiques, sauf pour le nombre de fois que la boucle de clignotement est exécutée (lignes 27 et 49) et les retards utilisés.

Liste 1. Le programme Arduino evtgrp.ino utilisant des groupes d'événements. [1]

```
0001: // evtgrp.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // LED is active high
0005: #define GPIO_LED1     25
0006: #define GPIO_LED2     26
0007:
0008: #define EVBLK2        0b0001
0009: #define EVBLK3        0b0010
0010: #define EVALL         0b0011
0011:
0012: static EventGroupHandle_t hevt;
0013:
0014: void blink2(void *arg) {
0015:
0016:   for (;;) {
0017:     // Call blocks until EVBLK2 bit set,
0018:     // and same bit is cleared upon return
0019:     xEventGroupWaitBits(
0020:       hevt,
0021:       EVBLK2,
0022:       pdTRUE,
0023:       pdFALSE,
0024:       portMAX_DELAY
0025:     );
0026:     // Blink 2 times
0027:     for ( int x=0; x < 2; ++x ) {
0028:       digitalWrite(GPIO_LED1,HIGH);
0029:       delay(120);
0030:       digitalWrite(GPIO_LED1,LOW);
0031:       delay(120);
0032:     }
0033:   }
0034: }
0035:
0036: void blink3(void *arg) {
0037:
0038:   for (;;) {
0039:     // Call blocks until EVBLK3 bit set,
0040:     // and same bit is cleared upon return
0041:     xEventGroupWaitBits(
0042:       hevt,
0043:       EVBLK3,
0044:       pdTRUE,
0045:       pdFALSE,
0046:       portMAX_DELAY
0047:     );
0048:     // Blink 3 times
0049:     for ( int x=0; x < 3; ++x ) {
0050:       digitalWrite(GPIO_LED2,HIGH);
0051:       delay(75);
0052:       digitalWrite(GPIO_LED2,LOW);
0053:       delay(75);
0054:     }
0055:   }
0056: }
0057:
0058: void setup() {
0059:   int app_cpu = xPortGetCoreID();
0060:   BaseType_t rc;
0061:
0062:   pinMode(GPIO_LED1,OUTPUT);
0063:   pinMode(GPIO_LED2,OUTPUT);
0064:   digitalWrite(GPIO_LED1,LOW);
0065:   digitalWrite(GPIO_LED2,LOW);
0066:
0067:   delay(2000);
0068:
0069:   hevt = xEventGroupCreate();
0070:   assert(hevt);
0071:
0072:   rc = xTaskCreatePinnedToCore(
0073:     blink2,   // func
0074:     "blink2", // name
0075:     1024,     // stack bytes
0076:     nullptr,  // arg ptr
0077:     1,        // priority
0078:     nullptr,  // ptr to task handle
0079:     app_cpu   // cpu#
0080:   );
0081:   assert(rc == pdPASS);
0082:
0083:   rc = xTaskCreatePinnedToCore(
0084:     blink3,   // func
0085:     "blink3", // name
0086:     1024,     // stack bytes
0087:     nullptr,  // arg ptr
0088:     1,        // priority
0089:     nullptr,  // ptr to task handle
0090:     app_cpu   // cpu#
0091:   );
0092:   assert(rc == pdPASS);
0093: }
0094:
0095: void loop() {
0096:
0097:   delay(1000);
0098:   xEventGroupSetBits(
0099:     hevt,
0100:     EVBLK2|EVBLK3
0101:   );
0102: }
0103:
0104: // End evtgrp.ino
```

Synchronisation

La synchronisation est réalisée en faisant bloquer les fonctions de tâche dans un appel à `xEventGroupWaitBits()`. Jusqu'à ce que l'événement soit déclenché, l'exécution est bloquée dans cet appel de fonction. Une fois que le groupe d'événements est notifié, l'exécution des tâches bloquées reprend en revenant de la fonction appelée. L'événement est déclenché toutes les secondes par un appel correspondant à `xEventGroupSetBits()` dans la tâche `loop()` (voir les lignes 98 à 101 de la **liste 1** [1]).

Création de groupes d'événements

Le groupe d'événements est créé en appelant `xEventGroupCreate()` (ligne 69). Il n'y a pas d'arguments à fournir et la fonction renvoie une poignée (ou *handle*) à l'objet de groupe d'événements alloué. Le type de données renvoyé est `EventGroupHandle_t`. Si la valeur retournée est égale à zéro, votre pile est à court de mémoire (d'où le contrôle d'assertion à la ligne 70).

```
0069:   hevt = xEventGroupCreate();
0070:   assert(hevt);
```

Notification

Il existe différents choix pour manipuler les groupes d'événements. Dans cet article, nous utilisons une des fonctions les plus simples appelée `xEventGroupSetBits()`.

```
EventBits_t xEventGroupSetBits(
  EventGroupHandle_t xEventGroup,    // Handle
  const EventBits_t uxBitsToSet      // Bits to set
);
```

La poignée `xEventGroup` spécifie le groupe d'événements à exploiter, tandis que l'argument `uxBitsToSet` spécifie les bits d'événements que vous souhaitez définir de manière atomique**. La valeur retournée sera les bits d'événement qui étaient en vigueur *après* l'appel à `xEventGroupSetBits()`. Cependant, sachez que la valeur retournée peut ne pas toujours inclure les bits que vous venez de définir car les tâches de réception peuvent les avoir déjà effacés à la réception. La tâche `loop()` appelle la fonction `xEventGroupSetBits()` en fixant les bits 0 et 1 à l'aide de la macro expression `EVBLK2|EVBLK3`.

```
0008: #define EVBLK2        0b0001
0009: #define EVBLK3        0b0010

0098:   xEventGroupSetBits(
0099:     hevt,
0100:     EVBLK2|EVBLK3
0101:   );
```

De cet appel, on peut voir que les bits 0 et 1 sont réglés atomiquement** par cet appel.

La tâche loop()

La tâche `loop()` retarde d'une seconde (ligne 97) et envoie ensuite une notification d'événement (lignes 98 à 101). Ensuite, suivant la tradition d'Arduino, la fonction `loop()` quitte et se répète. Les événements sont alors déclenchés à environ une seconde d'intervalle.

Réception des tâches

Les tâches `blink2()` et `blink3()` sont notifiées par le groupe d'événements à l'aide de l'identifiant enregistré (lignes 12 et 69). Par exemple, `blink2()` voit son exécution suspendue aux lignes 19 à 25 jusqu'à ce qu'un événement soit envoyé. De même, l'exécution de `blink3()` `est suspendue aux` lignes 41 à 47 jusqu'à ce qu'un événement soit envoyé. Comme les événements sont déclenchés atomiquement** par les lignes 98 à 101, les deux tâches reprendront en même temps. Si les deux tâches sont assignées à la même unité centrale, elles seront alors programmées pour s'exécuter l'une après l'autre. Si les deux tâches sont assignées à des CPU différentes, il est alors possible qu'elles reprennent et s'exécutent simultanément.
Une fois que l'exécution reprend dans chaque tâche, ils peuvent alors faire clignoter leur LED de manière unique. Une fois ce clignotement terminé, l'exécution reprend au sommet de la boucle de tâches en attendant l'arrivée de l'événement suivant.

Compensation des événements

L'appel de la fonction `xEventGroupWaitBits()` est délicat et la flexibilité qu'il offre ajoute de la complexité. Décomposons le tout :

```
EventBits_t xEventGroupWaitBits(
  const EventGroupHandle_t xEventGroup, // Event group handle
  const EventBits_t uxBitsToWaitFor,
  const BaseType_t xClearOnExit,
  const BaseType_t xWaitForAllBits,
  TickType_t xTicksToWait
);
```

Le premier argument est la poignée du groupe d'événements référencés. C'est assez simple. Le deuxième argument spécifie les bits d'événement que la fonction veut attendre. Dans le programme de démonstration, nous avons spécifié la macro `EVBLK2` (ligne 21) pour la tâche `blink2()` et la macro `EVBLK3` pour la tâche `blink3()`. La raison de l'utilisation de bits séparés apparaîtra bientôt.
Le troisième argument `xClearOnExit` est une version en langage C d'une valeur booléenne (`pdTRUE` a été fourni aux lignes 22 et 44). Cela informe la fonction d'effacer les bits d'événement reçus avant de revenir. Ainsi, la tâche `blink2()` attend que le bit 0 se mette en place (macro `EVBLK2`), et le troisième argument `pdTRUE` l'informe d'effacer ce bit à la réception et au retour. Cette opération d'attente et d'effacement est effectuée de manière atomique**.
Dans notre cas particulier, l'argument quatre est académique mais examinons-le. L'argument `xWaitForAllBits` est également booléen et précise quand la fonction doit revenir :

- lorsque l'*un* des bits attendus est activé (`pdFALSE`), *ou*
- seulement une fois que *tous les* bits attendus sont réglés (`pdTRUE`).

Notre exemple n'attend qu'un seul bit (le bit 0 pour `blink2()` et le bit 1 pour `blink3()`), ce qui signifie que l'indication d'un ou de tous n'a pas d'effet matériel. Cependant, si vous souhaitez attendre une combinaison de bits, cette possibilité peut être très utile.

Contrôle des événements

La raison pour laquelle il était nécessaire de disposer de bits d'événements séparés peut maintenant être évidente. Chaque tâche de réception attend et efface ensuite son propre bit d'événement par un

*** Ces opérations dites atomiques assurent une synchronisation non bloquante pour résoudre les problèmes de performance causés par des opérations asynchrones qui agissent sur la mémoire partagée.*

LIENS

[1] **Code pour evtgrp.ino** : https://bit.ly/35YrjCN
[2] **Livre : *FreeRTOS for ESP32-Arduino*, W. Gay, Elektor 2020** : https://bit.ly/2U2Yhg1
[3] **Documentation FreeRTOS** : https://bit.ly/386HZL6

appel à `xEventGroupWaitBits()`. Il doit donc y avoir un bit séparé pour chaque tâche. Dans la fonction `loop()`, les deux tâches sont notifiées simultanément en réglant leurs bits d'événement spécifiques en même temps (ligne 100).
Comme on peut le voir, les bits du groupe d'événements peuvent être utilisés pour créer un code d'application très complexe et créatif. Pour plus de détails sur les fonctions supplémentaires, voir le livre *FreeRTOS for ESP32-Arduino* [2] et la documentation de *FreeRTOS* [3].

Lancer la démo

La démo n'utilise pas le moniteur sériel et peut être exécutée sur n'importe quel ESP32 qui rend disponible les GPIO 25 et 26 (ou changez les lignes 5 et 6 de la **liste 1** pour utiliser d'autres sorties). Faites clignoter le programme *evtgrp.ino* dans l'ESP32 avec deux LED et des résistances de 220 Ω comme sur la **fig. 2**. Une fois que l'ESP32 commence l'exécution, les LED devraient clignoter - une fois par seconde, la LED1 devrait clignoter deux fois et ensuite attendre tandis que la LED2 devrait clignoter trois fois (rapidement) puis attendre. Le processus se répétera toutes les secondes.

Synchronisation totale

Cette démo illustre comment deux tâches complètement indépendantes, `blink2()` et `blink3()`, peuvent être synchronisées en utilisant un seul objet de groupe d'événements. La tâche `loop()` diffuse son événement des lignes 98 à 101. Cette approche peut être étendue jusqu'à 24 tâches si nécessaire (la limite est définie par le nombre de drapeaux d'événement disponibles dans un seul groupe d'événements). Cet article a montré une façon d'utiliser les groupes d'événements, mais il existe bien d'autres façons plus subtiles de coordonner des événements en utilisant d'autres fonctions de *FreeRTOS*.

200528-03

Votre avis, s'il vous plaît
Vos questions et vos commentaires sont bienvenus chez l'auteur ve3wwg@gmail.com en anglais, ou à la rédaction redaction@elektor.fr en français.

Ont contribué à cet article

Auteur : **Warren Gay**
Schéma : **Patrick Wielders**
Illustration : **Jack Jamar**
Rédaction : **Stuart Cording**
Maquette : **Giel Dols**
Traduction : **Léo Helba**

PRODUITS

> **Livre : *FreeRTOS for ESP32-Arduino*, W. Gay, Elektor 2020**
https://bit.ly/2U2Yhg1