multitâche en pratique avec l'ESP32 (5)

Notification d'événements de tâches

Warren Gay (Canada)

FreeRTOS offre des fonctions de file d'attente, de sémaphore et de *mutex* qui, pour la synchronisation, peuvent parfois sembler trop compliquées lorsque les besoins sont eux-mêmes simples. La version V8.2.0 de FreeRTOS a introduit le concept de *notification directe des événements* aux tâches, ce qui offre au programmeur une méthode légère de synchronisation avec les tâches.

Chaque tâche comprend une valeur de notification d'événement intégrée de 32 bits, initialisée lors de la création de la tâche. Comme cette valeur est intégrée à chaque tâche, aucune mémoire vive supplémentaire n'est requise, soit une économie considérable par rapport à l'utilisation p. ex. d'objets sémaphores.

Limitations

Il existe toutefois certaines limites à l'utilisation des notifications de tâches directes :

- Vous ne pouvez pas envoyer de notification à un ISR, car un ISR n'est pas une tâche. Cependant, un ISR *peut* notifier une tâche.
- Une seule tâche peut être notifiée par un appel de notification de tâche (les groupes d'événements doivent être utilisés pour notifier plusieurs tâches).
- Les événements de notification ne peuvent pas être mis en mémoire tampon comme les files d'attente.
- La tâche de *notification* (ou ISR) ne bloque pas son exécution pour attendre que la tâche de réception reçoive l'événement. L'événement est simplement affiché et l'appel revient immédiatement, sans entrave.

Si l'une de ces restrictions est en conflit avec vos exigences, vous devrez choisir un autre mécanisme FreeRTOS à la place.

En attente d'un événement

La première étape de l'utilisation des notifications d'événements de tâches consiste à informer la tâche destinataire d'attendre une notification. En d'autres termes, la tâche doit être configurée pour s'interrompre jusqu'à ce qu'un événement soit déclenché. Ceci est mis en œuvre en utilisant l'une des fonctions suivantes :

- `ulTaskNotifyTake()`
- `xTaskNotifyWait()`

La fonction `ulTaskNotifyTake()` est la plus simple des deux, c'est celle que nous examinons dans cet article. La fonction `xTaskNotifyWait()` est plus avancée et est examinée en détail dans mon livre *FreeRTOS for ESP32-Arduino* publié par Elektor.

ulTaskNotifyTake()

La tâche qui reçoit une notification bloque son exécution en appelant `ulTaskNotifyTake()`. L'appel nécessite deux arguments :

```
uint32_t ulTaskNotifyTake(
  BaseType_t xClearCountOnExit, // pdFALSE or pdTRUE
  TickType_t xTicksToWait
);
```

Le premier argument opère sur le mot de notification de tâche de 32 bits, également appelé mot d'événement de tâche (tableau 1). Le deuxième paramètre est la valeur familière de la *temporisation* dans les tics (utilisez la macro `pdMAX_DELAY` si vous ne voulez pas de temporisation). Dans tous les appels à `ulTaskNotifyTake()`, l'exécution de la tâche appelante est bloquée tant que le mot d'événement de la tâche reste à zéro. Dès que la valeur devient non nulle, l'argument `xClearCountOnExit` détermine comment le mot d'événement de la tâche est mis à jour avant de retourner.

Tableau 1. La signification des arguments xClearCountOnExit pour ulTaskNotifyTake().

xClearCountOnExit	Effet sur le mot de notification des tâches de 32 bits
pdFALSE	Décrémentation, blocage de de l'exécution si la valeur était nulle avant la décrémentation, avant de renvoyer la valeur antérieure.
pdTRUE	Effacement de la valeur à 0, blocage de l'exécution si la valeur était déjà nulle, avant de renvoyer la valeur antérieure.

La valeur renvoyée par l'appel de fonction est la valeur du mot de notification de l'événement *avant* qu'il ne soit effacé ou décrémenté . En cas de dépassement de *délai*, la valeur renvoyée sera également égale à zéro.

Notification binaire

Lorsque la valeur de l'argument `xClearCountOnExit` est `pdTRUE`, l'appel à `ulTaskNotifyTake()` fonctionne comme l' opération de

prise de sémaphore binaire. Si le mot d'événement de notification de la tâche est *déjà non nul*, l'appel renvoie immédiatement et efface le mot d'événement (sa valeur antérieure non nulle est cependant renvoyée). Si le mot de notification de la tâche était à zéro au moment de l'appel, l'exécution de la tâche est *bloquée* jusqu'à ce que la tâche soit notifiée (sous réserve d'un délai d'attente). Si un délai d'attente se produit, la valeur de retour sera toujours zéro (reflétant la valeur du mot d'événement au moment du retour). Dans ce mode de fonctionnement, l'appel fonctionne comme un sémaphore binaire. Si la tâche devait être notifiée deux fois avant que la tâche réceptrice n'appelle `ulTaskNotifyTake()`, alors la valeur retournée sera 2. Cependant, le mot d'événement de la tâche est effacé lors du retour. Cela ne fournit effectivement qu'un seul événement de notification dans ce cas particulier, mais le nombre réel de notifications est facilement déterminé en notant la valeur de retour.

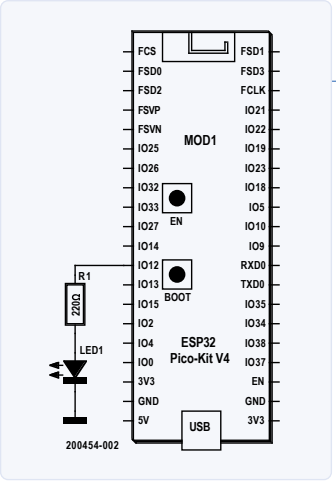

Figure 1. Schéma pour le code ***tasknfy.ino****.*

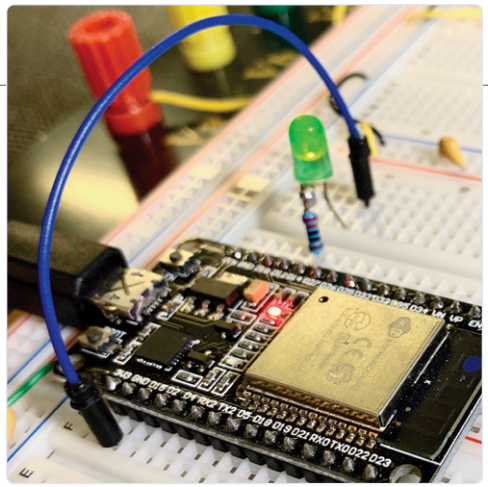

Figure 2. Mise en place du circuit de démonstration final à l'aide de GPIO 12.

Notification de comptage

Si l'exigence est que les notifications de tâches multiples *doivent* provoquer des réveils de tâches multiples , alors l'argument `xClearCountOnExit` doit être fourni avec la valeur `pdFALSE`. Dans un tel cas, la tâche réceptrice se bloque alors que la valeur de l'événement de la tâche est à zéro (comme auparavant). Chaque notification de la tâche incrémente la valeur de l'événement de tâche, mais elle n'est décrémentée que d'*une unité* pour chaque appel à `ulTaskNotifyTake()`. Par exemple, si quatre notifications se produisent avant que la tâche réceptrice n'appelle `ulTaskNotifyTake()`, alors cet appel revient immédiatement sans blocage jusqu'à ce que la valeur de notification de la tâche ait été décrémentée à nouveau à zéro. En d'autres termes, la tâche réceptrice doit effectuer quatre appels à `ulTaskNotifyTake()` avant que le mot d'événement ne soit décrémenté jusqu'à zéro.

Notifier

Alors que la tâche réceptrice utilise la fonction `ulTaskNotifyTake()`, la notification est fournie en utilisant `xTaskNotifyGive()` :

```
BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
```

Cette fonction ne nécessite que le *handle* (ou poignée) de la tâche à notifier et renvoie toujours `pdTRUE` (le manuel de FreeRTOS indique que `xTaskNotifyGive()` est défini comme une macro).

Démonstration

Le programme de démonstration très simple (**liste 1)** [1] utilise le moniteur sériel afin que vous puissiez voir la valeur du mot d'événement rapportée. La **figure 1** donne le câblage d'une LED optionnelle, tandis que la **figure 2** illustre la configuration du tableau de bord. Presque toutes les cartes de développement ESP32 peuvent être utilisées si GPIO 12 est disponible.

Le programme *tasknfy1.ino* utilise le `loopTask()` fourni par Arduino

Liste 1. Programme de démonstration tasknfy1.ino avec `xTaskNotifyGive()` et `ulTaskNotifyTake()` .

```
0001: // tasknfy1.ino
0002:
0003: #define GPIO_LED        12
0004:
0005: static TaskHandle_t htask1;
0006:
0007: static void task1(void *arg) {
0008:  uint32_t rv;
0009:
0010:  for (;;) {
0011:    rv = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
0012:    digitalWrite(GPIO_LED, digitalRead(GPIO_LED)^HIGH);
0013:    printf("Task notified: rv=%u\n", unsigned(rv));
0014:  }
0015: }
0016:
0017: void setup() {
0018:  int app_cpu = 0;
0019:  BaseType_t rc;
0020:
0021:  app_cpu = xPortGetCoreID();
0022:  pinMode(GPIO_LED, OUTPUT);
0023:  digitalWrite(GPIO_LED, LOW);
0024:
0025:  delay(2000);    // Allow USB to connect
0026:   printf("tasknfy1.ino:\n");
0027:
0028:  rc = xTaskCreatePinnedToCore(
0029:    task1,        // Task function
0030:    "task1",      // Name
0031:    3000,         // Stack size
0032:    nullptr,      // Parameters
0033:    1,            // Priority
0034:    &htask1,      // handle
0035:    app_cpu       // CPU
0036:  );
0037:  assert(rc == pdPASS);
0038: }
0039:
0040: void loop() {
0041:  delay(1000);
0042:  xTaskNotifyGive(htask1);
0043: }
```

pour notifier de manière répétée la `task1()` (ligne 42) à intervalles d'une seconde. La fonction `task1()` se bloque alors (ligne 11) jusqu'à ce qu'elle soit notifiée. Une fois notifiée, elle bascule ensuite l'état de la LED (ligne 12). La fonction `task1()` est notifiée par l'utilisation du handle `htask1` (ligne 42). Ce *handle* a été créé dans la fonction `setup()` à la ligne 34.

```
0040: void loop() {
0041:   delay(1000);
0042:   xTaskNotifyGive(htask1);
0043: }
```

Le code de `task1` est presque aussi banal, c'est une boucle sans fin. Il bloque son exécution lors de l'appel à `ulTaskNotifyTake()` dans la ligne 11 pour attendre un événement de notification :

```
0007: static void task1(void *arg) {
0008:  uint32_t rv;
0009:
0010:  for (;;) {
0011:   rv = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
0012:   digitalWrite(GPIO_LED, digitalRead(GPIO_LED)^HIGH);
0013:   printf("Task notified: rv=%u\n", unsigned(rv));
0014:   }
0015: }
```

La poursuite de l'exécution de `task1` reste bloquée jusqu'à l'arrivée de l'événement de notification de la tâche. La valeur attribuée à `rv` (ligne 11) est la valeur du mot de notification d'événement *avant* qu' il ne soit effacé (en raison de l'argument `pdTRUE`). C'est propre et simple - il n'y a pas d'autres objets sémaphores impliqués et, donc, pas de *handle* supplémentaires.
La session de surveillance en série qui en résulte devrait ressembler à ceci :

```
tasknfy1.ino:
Task notified: rv=1
Task notified: rv=1
...
```

Si vous souhaitez notifier une tâche à partir d'un ISR, utilisez la fonction `xTaskNotifyGiveFromISR()`. La convention du suffixe FreeRTOS «FromISR» permet de travailler en toute sécurité à partir d'un ISR, qui implique souvent une manipulation particulière.

```
void vTaskNotifyGiveFromISR(
  TaskHandle_t xTaskToNotify,
  BaseType_t *pxHigherPriorityTaskWoken
);
```

Le deuxième argument est l'adresse du drapeau de réveil de l'ISR, indiquant si l'ordonnanceur doit être invoqué ou non (fournir `NULL/nullptr` lorsque ce n'est pas nécessaire). Cette fonction de notification de tâche constitue un moyen très pratique pour une routine ISR de notifier une tâche.

Notification préalable

Dans cet article, nous n'avons abordé que la paire de fonctions la plus simple, `xTaskNotifyGive()` (ou `xTaskNotifyGiveFive fromISR()` et `ulTaskNotifyTake()`.
Des fonctions supplémentaires et avancées sont disponibles, notamment les suivantes :

- `xTaskNotify()` (or `xTaskNotifyFromISR()`)
- `xTaskNotifyAndQuery()` (or `xTaskNotifyAndQueryFromISR()`)
- `xTaskNotifyWait()`
- `xTaskNotifyStateClear()`
- `xTaskNotifyValueClear()`

Ces fonctions sont similaires, mais supportent le traitement des événements au niveau du bit. Elles fournissent également des formes plus sélectives de test et de compensation des événements. Vous pouvez vous informer à ce sujet dans le manuel de référence de FreeRTOS [2] ou en ligne [3].

Conclusion

L'API de notification de tâches FreeRTOS fournit au développeur d'applications un mécanisme léger de notification d'événements. Ce mécanisme est particulièrement utile pour mettre en œuvre la notification de tâches à partir d'ISR, car il permet de réduire le temps de traitement des interruptions, tout en permettant d'effectuer le travail difficile dans la tâche notifiée. Même sans utiliser les interruptions, elle économise de la mémoire vive et simplifie l'application en ne nécessitant pas d'objets de synchronisation supplémentaires tels que des sémaphores.

200454-03

Votre avis, s'il vous plaît…
Vous pouvez adresser vos questions ou vos commentaires à l'auteur (en anglais) à ve3wwg@gmail.com ou (en français) à redaction@elektor.fr

On participé à cet article
Idée et texte : **Warren Gay** Rédaction : **Stuart Cording** Maquette : **Giel Dols** Traduction : **Frédéric Handle**

LIENS

[1] **Code de tasknfy.ino** : https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/tasknfy1/tasknfy1.ino
[2] **Documentation de FreeRTOS** : www.freertos.org/Documentation/RTOS_book.html
[3] **Documentation pour la notification de tâches FreeRTOS** : www.freertos.org/RTOS-task-notification-API.html