multitâche en pratique avec l'ESP32 (2)

Priorités des tâches

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. L'ESP32 et l'EDI Arduino facilitent la programmation de tâches, car le fameux FreeRTOS est déjà intégré dans les bibliothèques de base [**1**].
Ici nous nous intéressons aux priorités de tâches.

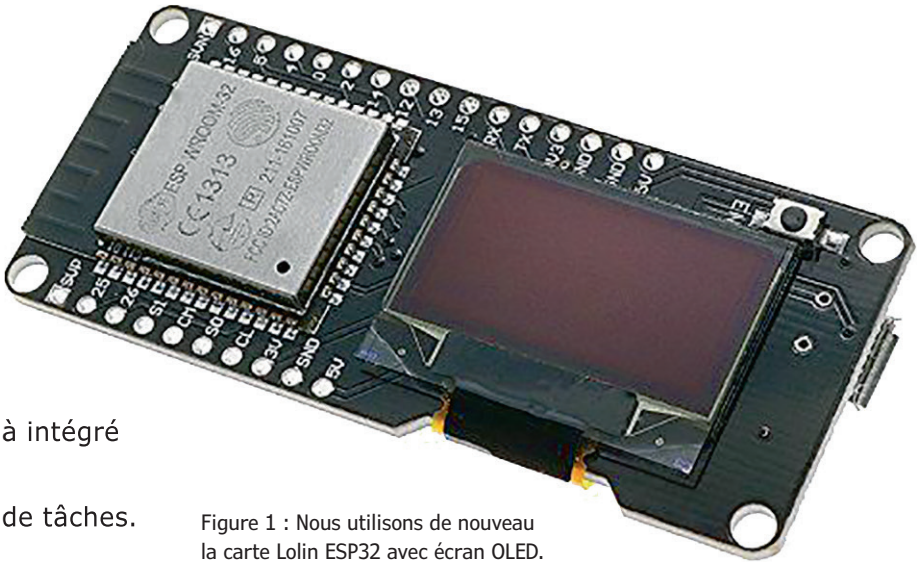

Figure 1 : Nous utilisons de nouveau la carte Lolin ESP32 avec écran OLED.

Dans l'implémentation pour l'ESP32 de l'ordonnanceur de FreeRTOS, les tâches sont exécutées selon leur priorité. La priorité est affectée lorsque la tâche est créée, mais peut être modifiée ensuite. Les tâches de rang élevé sont traitées en premier pour l'Unité Centrale (UC) configurée, tandis que les tâches de priorité nulle sont traitées en dernier. Le concept de priorité d'exécution peut vous être familier, mais l'ordonnanceur en temps réel de FreeRTOS fonctionne différemment de ce que vous connaissez de Linux ou Windows. Cet article étudie la différence au moyen d'une démonstration.
L'implémentation pour l'ESP32 comporte 25 niveaux de priorité, de 0 à 24. Par défaut, les fonctions Arduino `setup()` et `loop()` s'exécutent au niveau de priorité 1 (ces fonctions font partie de la même tâche principale [**1**]).

Vive la différence

Comment l'ordonnancement des tâches peut-il être différent ? Sur un système Linux p. ex., la priorité affecte l'urgence relative du processus ou du *thread*. Même un processus de faible priorité obtient un peu de temps d'UC – ce qui allonge son temps d'exécution. Mais le processus **finit toujours par s'exécuter.** Voilà la différence. Dans un système en temps réel comme FreeRTOS, l'ordonnanceur **ne garantit pas** que les tâches de moindres priorités soient un jour exécutées. Si vous avez des tâches p. ex. de priorité 9 toujours *prêtes à s'exécuter*, aucune tâche de priorité 8 ou moins ne sera séquencée sur cette même UC. Les tâches de priorité 9 privent d'UC toutes les tâches de priorité inférieure.

Prête à s'exécuter

Il est important de comprendre ce que *ready* (prête) signifie pour FreeRTOS. Une tâche est *ready* lorsque qu'elle n'est pas bloquée en attente de quelque chose, que ce soit un évènement, une entrée poussée dans une *file d'attente* ou le déverrouillage d'un *mutex* (nous y reviendrons plus tard dans cette série). Une tâche prête à s'exécuter est insérée selon sa priorité dans la *ready list* de l'ordonnanceur et sera exécutée une fois son tour venu. Comme c'est une liste classée par priorité, les tâches de plus hautes priorités sont traitées en premier.
Les tâches d'égales priorités sont séquencées avec une approche *Round-Robin* (tourniquet). Trois tâches « *ready* » de priorité 9 (a, b et c) vont se succéder :

- tâche9a
- tâche9b
- tâche9c
- tâche9a
- tâche9b
- etc.

À moins qu'elles ne se bloquent, cela continue indéfiniment. Seule une tâche de plus haute priorité peut les préempter. Par exemple, la tâche ESP32 de haute priorité nommée *idc1* (pour l'UC 1), peut préempter vos tâches de priorité 9 pour effectuer certaines opérations. Dès qu'*idc1* quitte à nouveau l'état *ready*, les tâches de priorité 9 reprennent de là où elles s'étaient arrêtées.
Voici quelques exemples de sortie de l'état *ready* par une tâche FreeRTOS :

- mise en sommeil ou délai pour un moment (attente d'une temporisation) ;
- attente d'un *mutex* ou d'un sémaphore ;
- attente d'un message depuis une file d'attente vide ;
- attente d'insertion d'un message dans une file d'attente pleine ;
- attente d'un évènement FreeRTOS ou d'un groupe d'évènements ;

- attente de la fin d'une E/S ;
- suspension (soit par la tâche elle-même, soit par une autre tâche).

L'attente de réception d'un message depuis une file vide est une situation de blocage. Si la file est vide, la tâche n'a rien à faire, donc l'ordonnanceur retire la tâche de la *ready list* et en cherche d'autres à exécuter. Seules les tâches de la *ready list* seront traitées. Si aucune tâche n'est trouvée, c'est la tâche inactive du système qui est exécutée à la place.
Notez que l'appel à la fonction `taskYIELD()` n'est pas une des raisons listées. Lorsqu'une tâche cède la main, soit après avoir épuisé sa tranche de temps soit volontairement par appel à `taskYIELD()`, le contrôle repasse à FreeRTOS pour qu'il puisse choisir une autre tâche à exécuter pour la tranche suivante. Céder la main n'est pas bloquant, car ces tâches restent prêtes à s'exécuter et retrouveront l'UC dès la prochaine occasion.

Modifications d'ESP-IDF pour FreeRTOS SMP

FreeRTOS a été conçu pour des microcontrôleurs mono-UC. Comme l'ESP32 est un dispositif à double-UC (sauf l'ESP32-S2), Espressif a adapté le composant ordonnanceur. À l'examen, les UC ESP32 suivantes sont présentes :

- CPU 0 alias PRO_CPU (Protocol CPU) ;
- CPU 1 alias APP_CPU (Application CPU).

Espressif indique que « les deux cœurs sont identiques en pratique et partagent la même mémoire ». Pour prendre en charge le traitement symétrique multiprocesseur (SMP), ils précisent que « l'ordonnanceur sautera des tâches lors de la mise en œuvre d'un ordonnancement *Round-Robin* entre plusieurs tâches à l'état *Ready* de même priorité ». Ceci provient de la limitation à utiliser une *ready list* conçue pour une UC unique sur une plateforme qui en a deux [**2**].

Le problème rencontré était que lorsque l'UC avait besoin d'un changement de contexte de tâche (pour exécuter la tâche suivante), l'UC n'a qu'une *ready list* de tâches à scruter. Alors si l'index courant de la liste pointe sur des tâches *ready* pour l'autre UC, ces entrées doivent être sautées jusqu'à ce qu'une entrée pour l'UC concernée soit trouvée. L'ordonnancement Round-Robin peut donc être loin de la perfection. Le résultat pour le développeur est que l'ordonnancement *Round-Robin* n'est pas complètement équitable dans les ESP32 à double-UC. Pour de nombreux projets, cela ne sera pas perceptible, mais si cela devient problématique, il y a des moyens de le contourner par codage. Pensez-y dans votre planification de tâches.

Démonstration

Un programme de démonstration Arduino existe pour la carte Lolin ESP32 OLED Display Module (**fig. 1**). En modifiant quelques macros du programme, vous pourrez y changer les priorités de quatre tâches différentes. Le programme est conçu pour afficher trois chenilles qui rampent d'avant en arrière sur toute la longueur de l'OLED. Chaque chenille n'ondule que si la tâche qui la pilote obtient du temps d'UC. Les tâches privées d'UC laisseront la chenille immobile ou lente.
Chaque chenille est pilotée par une tâche qui consomme du temps d'UC puis envoie un message à la quatrième tâche chargée de faire ramper cette chenille et de l'afficher.
Le code pour dessiner et gérer l'état de la chenille est défini dans la classe `InchWorm` (pas présentée ici). Pour cet article, nous allons juste nous concentrer sur l'action sur chaque chenille de la méthode `InchWorm::draw()`. Chaque instance de la classe `InchWorm` gère son propre état et sa progression. L'écran et les instances de chenilles sont déclarés dans le programme comme suit :

```
static Display oled;
static InchWorm worm1(oled,1);
static InchWorm worm2(oled,2);
static InchWorm worm3(oled,3);
```

Chaque instance comporte une référence C++ (comme un pointeur en C) vers la classe de l'écran en premier argument et le numéro de chenille en second. La référence à l'écran permet de futures évolutions comme la prise en charge de plusieurs écrans. Le numéro de chenille détermine sa position sur l'OLED (1, 2 et 3 correspondent à la ligne du haut, du milieu et du bas). La tâche derrière chaque chenille est juste une boucle de consommation de temps d'UC et d'appel à l'envoi d'un message :

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__(«nop»);
    xQueueSendToBack(qh,&worm,0);
    // vTaskDelay(10);
  }
}
```

Il est important de laisser la fonction `vTaskDelay()` commentée pour l'instant. Elle sera utilisée dans une prochaine expérimentation.

La même fonction de tâche sert pour les trois tâches, l'argument nommé `arg` précisant quelle instance de chenille on souhaite faire tortiller. L'adresse de la chenille provient d'un pointeur non typé et est stockée dans la variable locale `worm`. Elle n'est utilisée dans cette tâche que pour l'envoi d'un message à la tâche d'affichage (tâche principale) qui lui indique la chenille à faire tortiller.

Notez que lorsque `xQueueSendToBack()` est appelée dans cette démonstration, le paramètre temps d'attente a été fixé à zéro (3e argument). Ceci indique à FreeRTOS de l'insérer dans la file si possible, mais échoue immédiatement si la file est pleine. C'est volontaire parce que si la file devient pleine, il ne faut pas que cela bloque l'exécution de notre tâche de chenille. La tâche ne doit pas relâcher l'UC pour cette démonstration afin qu'elle puisse vraiment monopoliser l'UC.

La boucle `for` extérieure fait que la tâche effectue ses opérations indéfiniment. La boucle `for` intérieure de consommation de temps d'UC exécute 800 000 fois une instruction *no operation* (`nop`). Le mot-clé `__volatile__` empêche que le compilateur n'optimise cette boucle en la supprimant. Quoiqu'en pense le compilateur, ce gaspillage est utile, nous y tenons.
À la fin de la boucle de consommation de temps, nous envoyons l'adresse de la chenille à faire se tortiller à la file identifiée par le *handle* `qh`. Dès que le message est reçu par la tâche d'affichage, cela fait avancer notre chenille et affiche son déplacement.

La tâche principale `loop()` d'Arduino est utilisée comme tâche d'affichage pour faire se tortiller les chenilles :

```
void loop() {
  InchWorm *worm = nullptr;

  if ( xQueueReceive(qh,&worm,portMAX_DELAY) )
    worm->draw();
}
```

Cette boucle bloque l'exécution jusqu'à ce que l'une des tâches envoie l'adresse de la chenille à dessiner. Dès que ce pointeur de classe est reçu, on invoque la méthode `InchWorm::draw()` pour dessiner la chenille et la faire avancer.
La fonction `setup()` est illustrée dans le **listage 1**, qui montre comment les trois tâches chenilles et la file sont créées.

Listage 1 – La fonction setup().

```
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Quelle UC ?
  oled.init();
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
  qh = xQueueCreate(4,sizeof(InchWorm*));

  // Dessiner au moins une chenille :
  worm1.draw();
  worm2.draw();
  worm3.draw();

  xTaskCreatePinnedToCore(
    worm_task,  // Fonction
    «worm1»,    // Nom de la tâche
    3000,       // Taille de la pile
    &worm1,     // Argument
    WORM1_TASK_PRIORITY,
    nullptr,    // Pas de retour d'identifiant
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Fonction
    «worm2»,    // Nom de la tâche
    3000,       // Taille de la pile
    &worm2,     // Argument
    WORM2_TASK_PRIORITY,
    nullptr,    // Pas de retour d'identifiant
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Fonction
    «worm3»,    // Nom de la tâche
    3000,       // Taille de la pile
    &worm3,     // Argument
    WORM3_TASK_PRIORITY,
    nullptr,    // Pas de retour d'identifiant
    app_cpu);
}
```

Changement de priorité

FreeRTOS autorise une tâche à changer sa propre priorité ou celle d'une autre tâche avec la fonction `vTaskPrioritySet()`. Par défaut la tâche qui invoque `setup()` et `loop()` s'exécute au niveau de priorité 1 (ces fonctions sont appelées par la même tâche principale). Pour cette démonstration, il faut que cette priorité soit plus haute que les trois autres tâches chenilles. La fonction `setup()` modifie la priorité de sa propre tâche :

```
static int app_cpu = 0; // Updated by setup()
...
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  ...
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
```

Comme indiqué, la fonction `setup()` obtient son propre identifiant de tâche en appelant `xTaskGetCurrentTaskHandle()` et en le stockant dans `h`. en changeant la priorité de la tâche principale par l'appel à `vTaskPrioritySet()`, la priorité de la tâche utilisée par `loop()` est aussi affectée. C'est un exemple de la façon dont on peut régler les priorités des tâches.
Dans la première expérimentation, on affecte aux tâches chenilles les priorités 9, 8 et 7. Ceci implique que notre tâche (principale) d'affichage soit à une priorité 9 ou supérieure (nous prendrons 10). Si nous ne le faisions pas, la tâche principale `loop()` serait privée d'UC et incapable d'animer les chenilles.

Quelle UC ?

Dans l'extrait de `setup()` présenté, on voit une autre fonction API de l'ESP32 nommée `xPortGetCoreID()` qui cherche sur quelle UC l'application est exécutée. Le résultat est assigné à la variable statique `app_cpu` dans le programme de façon que le code sache pour quelle UC créer de nouvelles tâches. Pour les ESP32 à double-cœur, la valeur de `app_cpu` sera 1 (exécution sur l'UC 1 dans une configuration à double-cœur). Pour les plateformes à un seul cœur, `app_cpu` sera mis à zéro. Le coder ainsi permet normalement la portabilité entre plateformes simples ou doubles.
Toutefois, cette démonstration en particulier ne fonctionnera pas bien sur une plateforme à simple-UC à cause de la façon dont l'UC est monopolisée. Cela déclenchera le temporisateur du chien de garde (*watchdog*) et provoquera une RàZ. Mais la technique d'utilisation de `xPortGetCoreID()`illustre comment permettre la portabilité pour d'autres applications.

Configuration de la démo

Le code source de la démonstration est disponible [**3**]. Au début se trouvent les définitions des macros qui configurent chaque expérimentation :

```
// Worm task priorities
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7

// loop() must have highest priority
#define MAIN_TASK_PRIORITY  10
```

Laissez-les d'abord telles quelles pour la première expérimentation.

Écran OLED personnalisé

Si vous n'utilisez pas la carte recommandée Lolin ESP32 avec son OLED intégré, les réglages de votre écran personnalisé peuvent être reconfigurés ici :

```
Display(
    int width=128,
    int height=64,
    int addr=0x3C,
    int sda=5,
    int scl=4);
```

Si vos réglages sont corrects, l'OLED devrait devenir blanc immédiatement à l'initialisation du programme. Sinon, revoyez les connexions et réglages.

Démonstration 1

Avec le code téléchargé, il suffit de compiler, de graver et d'exécuter l'application. Votre OLED devrait devenir blanc immédiatement, avec trois chenilles noires dessinées (**fig. 2**).
La configuration (encore) pour cette expérimentation est :

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

Avec cette configuration, la chenille du haut va traverser en se tortillant, pas les deux du bas. Pourquoi les chenilles du milieu et du bas ne bougent-elles pas ?

```
                    _¯_
_¯_
_¯_
```

Nous avions laissé la tâche principale à la priorité 10. Elle profite donc de la priorité la plus haute du jeu de tâches de notre application. La première chenille, qui s'affiche sur la ligne du haut, pouvait se déplacer parce que c'était la seule tâche consommatrice d'UC capable de s'exécuter. Cette tâche de priorité 9 est capable de s'exécuter parce que la tâche d'affichage de priorité 10 fait des E/S vers l'OLED puis attend l'arrivée de messages dans la file des messages (devient bloquée). Lorsque la tâche d'affichage est bloquée, les autres tâches de priorité inférieures peuvent être séquencées.

Les tâches de priorité 8 et 7 (pour les chenilles du milieu et du bas) sont privées d'UC et ne sont jamais exécutées parce que la tâche de priorité 9 monopolise complètement l'UC. C'est le propre de l'ordonnancement en temps réel dans FreeRTOS. À la différence de Linux ou Windows, les tâches de priorité inférieures n'ont aucune chance de s'exécuter.

Démonstration 2

Modifions la configuration pour donner aux trois chenilles la même priorité, mais laissons la tâche principale d'affichage à la priorité 10. Mettons-les toutes trois à la même priorité 9 (ou 8 ou 7) :

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  10
```

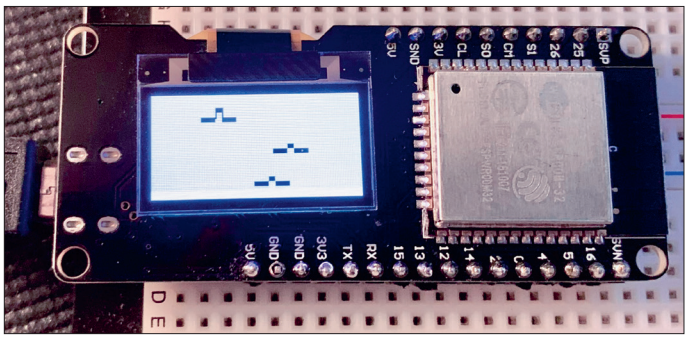

Figure 2 : Lorsque les tâches sont exécutées, les chenilles de la démo se déplacent.

Lorsque vous recompilez et regravez l'ESP32, que constatez-vous ?

```
        _¯_
        _¯_
        _¯_
```

Elles vont marcher au même rythme (ou presque). Quand on laisse tourner la démonstration assez longtemps, une chenille peut prendre un peu d'avance sur les autres.

Démonstration 3

Dans cette expérimentation, modifiez la configuration pour donner aux trois chenilles la même priorité (comme dans la précédente) et donnez à la tâche principale d'affichage cette même priorité. Je vais utiliser la priorité 9 pour toutes ces tâches :

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  9
```

Après recompilation, regravure et exécution du code, que constatez-vous ? Y a-t-il une différence ? Pourquoi avancent-elles à des vitesses différentes ?

```
      _¯_
            _¯_
                        _¯_
```

Lorsque j'exécute cela, la chenille du bas semble obtenir le plus d'UC (c.-à-d. qu'elle se tortille le plus vite). La chenille du haut est la plus lente. C'est encore la faute à l'irrégularité du *Round-Robin* signalé par *Espressif*. Idéalement, la tâche d'affichage ne devrait prendre qu'un peu d'UC pendant le dessin de la chenille. Par ailleurs, le temps d'UC restant devrait être réparti également entre les trois autres tâches qui pilotent les chenilles. Pourtant l'ordonnancement est déséquilibré. Les deux UC répondent à des temporisateurs et autres interruptions. Ce sont les imperfections du code de l'ordonnanceur qui perturbent l'ordonnancement *Round-Robin*.

Démonstration 4

Jusqu'ici, dans chaque démonstration, chaque tâche chenille a consommé tout le temps d'UC qu'elle a pu en obtenir. Que se passe-t-il si nous introduisons un petit retard (blocage) dans la boucle ?

Réinitialisez la configuration pour que la tâche principale d'affichage ait une priorité 10, et chacune des tâches chenilles respectivement une priorité 9, 8 et 7 :

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

Puis ôtez le commentaire de la ligne où `vTaskDelay()` est appelée de sorte que la boucle de la tâche devienne :

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    ...
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__(«nop»);
    xQueueSendToBack(qh,&worm,0);
    vTaskDelay(10); // Uncommented
  }
}
```

Maintenant, chaque tâche chenille va consommer de l'UC, essayer d'insérer une chenille dans la file puis se bloquer pendant 10 ms. Compilez, gravez et exécutez cet exemple. Que constatez-vous ?
La chenille du haut est la plus rapide et celle du bas la plus lente. Celle du haut avec la priorité 9 est la première à obtenir l'UC grâce à sa haute priorité (pendant que la tâche d'affichage est bloquée). Lorsque la tâche chenille est bloquée dans l'appel à `vTaskDelay(10)`, la tâche de priorité immédiatement inférieure (la chenille du milieu) obtient un peu d'UC à consommer et finit par appeler `vTaskDelay(10)`. Ceci permet à la tâche de priorité 7, encore plus basse, d'obtenir quelques cycles. Cet effet de cascade distribue le temps d'UC des niveaux les plus élevés aux plus bas.
Notez que les tâches de priorité 8 et 7 sont préemptées dès que la tâche de plus haute priorité 9 redevient *ready*. C'est pourquoi la chenille du haut est la plus rapide. Quelquefois celle du milieu préempte la tâche de priorité 7 : elle tend à être plus rapide que la chenille du bas.

Plus d'expérimentations

Que se passe-t-il si vous allongez le délai de `vTaskDelay()` à beaucoup plus que 10 ms ? Essayez d'intuiter la réponse puis exécutez-le. Pourquoi obtenez-vous ce résultat ? Que se passe-t-il si vous réduisez le délai à 1 ms ? À vous d'explorer…

Configuration de priorité

Nous n'avons pas encore parlé des interruptions dans l'ESP32, mais il est bon d'examiner le fichier d'entête **FreeRTOSConfig.h**, qui configure les priorités pour la plateforme et se trouve ici :

```
$IDF_PATH/components/freertos/include/freertos/
FreeRTOSConfig.h
```

L'entête définit les valeurs de macro de priorité suivantes. Les valeurs compilées sont indiquées :

```
configMAX_PRIORITIES = 25
configKERNEL_INTERRUPT_PRIORITY = 1
configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
```

La première macro définit le nombre maximum de priorités disponibles. Cela signifie que les niveaux de priorités valides vont de 0 à 24.
La seconde macro définit la priorité utilisée par le noyau lui-même pour les **interruptions**, tandis que la troisième macro fixe la priorité la plus haute utilisée par les interruptions du noyau. Tout appel d'une API FreeRTOS lancé depuis une *Interrupt Service Routine* (ISR) ne concernera que des fonctions API FreeRTOS dont le nom se termine par `FromISR()`. De plus, avec les valeurs indiquées, ces fonctions ne peuvent être appelées que par des tâches d'interruption de priorités 1 à 3. Si aucun appel `FromISR()` n'est fait, l'ISR peut librement opérer à des niveaux de priorité de 4 à 24.

Résumé

Que conclure de ces expérimentations ? Un concept de priorité, simple a priori, ne l'est pas. De sorte que si vos priorités de tâches ne sont pas bien planifiées, vous aurez des surprises – certaines tâches peuvent être privées d'UC. Nous n'avons pas encore parlé de l'impact des temporisateurs du chien de garde. Si p. ex. le temporisateur du chien de garde se déclenche dans l'UC 0, votre ESP32 redémarre !
Pour les ESP32 à double-cœur, il y a aussi le problème de l'ordonnancement *Round-Robin* des niveaux de priorité identiques qui se traduit par des inégalités de durée d'exécution, ce qui peut être problématique dans certaines applications, pas dans d'autres. Tout dépend de votre « système ».

Pour beaucoup d'applications, vous pouvez simplement créer des tâches qui fonctionnent en priorité 1. C'est la priorité configurée pour les tâches `setup()` et `loop()` d'Arduino. Des tâches de priorité supérieure peuvent être utilisées sans risque si elles se bloquent sur une file, un sémaphore ou un autre évènement. Lorsqu'une tâche se bloque ou est suspendue, l'UC est partagée entre les autres tâches de priorité égale ou inférieure. Une application avec des priorités de tâches correctement configurées fonctionnera comme une machine bien huilée. ⏮

(191195-02 VF Denis Lafourcade)

Liens

[1] Multitâche en pratique avec l'ESP32, Elektor janvier 2020 p. 42 : www.elektormagazine.fr/190182-03
[2] Traitement symétrique multiprocesseur : https://thc420.xyz/esp-idf/file/docs/en/api-guides/freertos-smp.rst.html
[3] Code source du projet : https://github.com/ve3wwg/esp32_freertos/blob/master/priority-worms1/priority-worms1.ino