Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

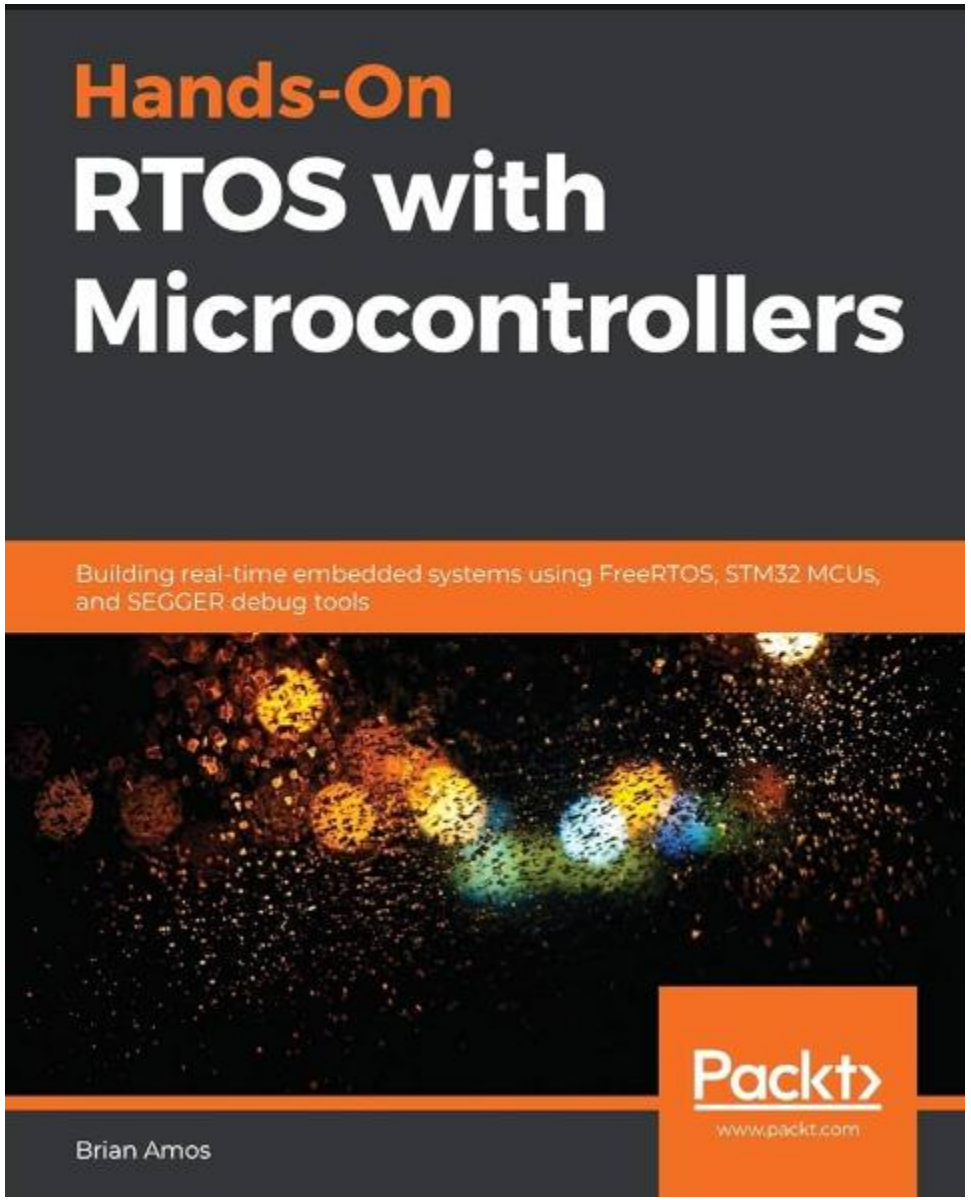
CEG4566/CSI4541/SEG4545

Conception de systèmes informatiques en temps réel

Hiver 2024

Professeur : Mohamed Ali Ibrahim, ing., Ph.D.

Source :


Hands-On
RTOS with
Microcontrollers
Building real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools
Packt
www.packt.com
Brian Amos

Chapitre 7 :
L'ordonnanceur FreeRTOS

Plan

- Création de tâches et lancement de l'ordonnanceur
- Suppression de tâches
- Essai du code
- Allocation de la mémoire des tâches
- Comprendre les états des tâches de FreeRTOS
- Résolution des problèmes de démarrage

Création de tâches et lancement de l'ordonnanceur

Pour qu'une application RTOS soit opérationnelle, plusieurs étapes doivent être franchies :

1. Le matériel de l'UCM doit être initialisé.
2. Les fonctions des tâches doivent être définies.
3. Les tâches RTOS doivent être créées et mises en correspondance avec les fonctions définies à l'étape 2.
4. L’ordonnanceur RTOS doit être lancé.

Initialisation du matériel

- Avant de pouvoir faire quoi que ce soit avec le RTOS, nous devons nous assurer que notre matériel est correctement configuré.
- Il s'agit notamment de s'assurer que les lignes GPIO sont dans leur état correct, de configurer la mémoire vive externe, de configurer les périphériques critiques et les circuits externes, d'effectuer des tests intégrés, etc.
- Dans tous nos exemples, l'initialisation matérielle du MCU peut être effectuée en appelant HAL_Init(), qui effectue toutes les initialisations matérielles de base nécessaires :

```
int main(void)
{
    HAL_Init() ;
```

Définition des fonctions des tâches

- Chacune des tâches, à savoir RedTask, BlueTask et GreenTask, est associée à une fonction.
- N'oubliez pas qu'une tâche n'est en fait qu'une boucle while infinie avec sa propre pile et une priorité. Examinons-les une par une.
- GreenTask dort pendant un petit moment (1,5 seconde) avec la LED verte allumée, puis s'efface. Quelques éléments méritent d'être soulignés, dont certains sont présentés dans la diapositive suivante.
- Normalement, une tâche contient une boucle while infinie afin de ne pas revenir. GreenTask ne revient toujours pas puisqu'elle se supprime elle-même.
- Vous pouvez facilement confirmer que vTaskDelete ne permet pas l'exécution au-delà de l'appel de fonction en regardant la carte Nucleo. Le voyant vert ne reste allumé que pendant 1,5 seconde avant de s'éteindre définitivement. Regardez l'exemple suivant, qui est un extrait de main_taskCreation.c

GreenTask

```
void GreenTask(void *argument)
{
    SEGGER_SYSVIEW_PrintfHost("Task1 running \")
       pendant que le voyant vert est allumé") ;
    GreenLed.On() ;
    vTaskDelay(1500/ portTICK_PERIOD_MS) ;
    GreenLed.Off() ;

    //une tâche peut se supprimer elle-même en passant NULL à vTaskDelete
    vTaskDelete(NULL) ;

    //la tâche n'arrive jamais ici
    GreenLed.On() ;
}
```

Bleu La tâche clignote

```
void BlueTask( void* argument )
{
    while(1)
    {
        SEGGER_SYSVIEW_PrintfHost("BlueTaskRunning\n") ;
        BlueLed.On() ;
        vTaskDelay(200 / portTICK_PERIOD_MS) ;
        BlueLed.Off() ;
        vTaskDelay(200 / portTICK_PERIOD_MS) ;
    }
}
```

Tâche rouge

```
void RedTask( void* argument )
{
    uint8_t firstRun = 1 ;
    while(1)
    {
        lookBusy() ;
        SEGGER_SYSVIEW_PrintfHost("RedTaskRunning\n") ;
        RedLed.On() ;
        vTaskDelay(500/ portTICK_PERIOD_MS) ;
        RedLed.Off() ;
        vTaskDelay(500/ portTICK_PERIOD_MS) ;
        if(firstRun == 1)
        {
            //Les tâches peuvent s'effacer les unes les autres en transmettant
            l'information souhaitée.
            //TaskHandle_t to vTaskDelete
            vTaskDelete(blueTaskHandle) ;
            firstRun = 0 ;
        }
    }
}
```

Création de tâches

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        configSTACK_DEPTH_TYPE usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask ) ;
```

Appel d'une fonction

```
retVal = xTaskCreate(Task1, "task1", StackSizeWords, NULL, tskIDLE_PRIORITY + 2, tskHandlePtr) ;
```

Les paramètres de la fonction

- Task1 : Le nom de la fonction qui implémente la boucle infinie while qui constitue la tâche.
- "task1" : Il s'agit d'un nom convivial utilisé pour référencer la tâche pendant le débogage.
- StackSizeWords : Le nombre de mots réservés à la pile de la tâche.
- NULL : Un pointeur qui peut être transmis à la fonction sous-jacente. Assurez-vous que le pointeur est toujours valide lorsque la tâche s'exécute finalement après le démarrage de l'ordonnanceur.
- tskIDLE_PRIORITY + 2 : Il s'agit de la priorité de la tâche en cours de création. Cet appel particulier fixe la priorité à deux niveaux plus élevés que la priorité de la tâche IDLE (qui s'exécute lorsqu'aucune autre tâche n'est en cours d'exécution).
- TaskHandlePtr : Il s'agit d'un pointeur vers un type de données TaskHandle_t (il s'agit d'une poignée qui peut être transmise à d'autres tâches pour faire référence à la tâche de manière programmatique).

Valeur de retour de la fonction

Valeur de retour :

- Le préfixe x de xTaskCreation signifie qu'il renvoie quelque chose.
- Dans ce cas, pdPASS ou errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY est renvoyé, selon que l'espace du tas a été alloué avec succès ou non.
- Vous devez vérifier cette valeur de retour !

L’ordonnanceur de démarrage

vTaskStartScheduler() est appelé lorsque le programme passe d'une super boucle traditionnelle à un RTOS multitâche.

```c
int main(void)
{
   HAL_Init() ;
   SystemClock_Config() ;
   MX_GPIO_Init() ;
   MX_USART2_UART_Init() ;

    //utilisation d'une instruction if soulignée avec une boucle while infinie pour s'arrêter au cas où
    //la tâche n'a pas été créée avec succès
    if (xTaskCreate(GreenTask, "GreenTask", STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL) != pdPASS){ while(1) ; }

    //utilisation d'un assert pour assurer la création correcte d'une tâche
    assert_param(xTaskCreate(BlueTask, "BlueTask", STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, &blueTaskHandle) == pdPASS) ;

    //xTaskCreateStatic renvoie le manuel de la tâche
    //se passe toujours puisque la mémoire a été allouée statiquement
    xTaskCreateStatic(RedTask, "RedTask", STACK_SIZE, NULL,
                tskIDLE_PRIORITY + 1,
                RedTaskStack, &RedTaskTCB) ;

    //démarre le programmateur - ne devrait pas revenir sauf en cas de problème
    vTaskStartScheduler() ;

    //si vous vous retrouvez ici, il y a probablement un problème de dépassement du tas de freeRTOS
    while(1)
    {
    }
}
```

LED verte Tâche

```
void GreenTask(void *argument)
{
   GreenLed.On() ;
   vTaskDelay(1500/ portTICK_PERIOD_MS) ;
   GreenLed.Off() ;

   //une tâche peut se supprimer elle-même en passant NULL à vTaskDelete
   vTaskDelete(NULL) ;

   //la tâche n'arrive jamais ici
   GreenLed.On() ;
}
```

Supprimer une tâche d'une autre tâche

- Pour supprimer une tâche d'une autre tâche, blueTaskHandle doit être transmis à xTaskCreate et sa valeur doit être renseignée.
- blueTaskHandle peut ensuite être utilisé par d'autres tâches pour supprimer BlueTask, comme illustré ici :

```
TaskHandle_t blueTaskHandle ;
int main(void)
{
    HWInit() ;
    assert_param( xTaskCreate(BlueTask, "BlueTask", STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, &blueTaskHandle) == pdPASS) ;
    xTaskCreateStatic( RedTask, "RedTask", STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, RedTaskStack, &RedTaskTCB) ;
    vTaskStartScheduler() ;
    while(1) ;
}

void RedTask( void* argument )
{
    vTaskDelete(blueTaskHandle) ;
}
```

Tâches allouées au tas

L'appel du début de cette section utilise le tas pour stocker la pile :

```
xTaskCreate(Task1, "task1", StackSizeWords, TaskHandlePtr, tskIDLE_PRIORITY + 2, NULL) ;
```

- xTaskCreate() est la plus simple des deux méthodes à appeler.
- Il utilisera la mémoire du tas FreeRTOS pour la pile de Task1 et le bloc de contrôle des tâches (TCB).

Tâches attribuées de manière statique

- Le prototype FreeRTOS pour xTaskCreateStatic() est le suivant :

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
                                const char * const pcName,
                                const uint32_t ulStackDepth,
                                void * const pvParameters,
                                UBaseType_t uxPriority,
                                StackType_t * const puxStackBuffer,
                                StaticTask_t * const pxTaskBuffer ) ;
```

- Voyons comment cela est utilisé dans notre exemple, qui crée une tâche avec une pile allouée de manière statique :

```
static StackType_t RedTaskStack[STACK_SIZE];
static StaticTask_t RedTaskTCB;
xTaskCreateStatic( RedTask, "RedTask", STACK_SIZE, NULL,
                   tskIDLE_PRIORITY + 1,
                   RedTaskStack, &RedTaskTCB);
```

- Contrairement à xTaskCreate(), xTaskCreateStatic() garantit toujours la création de la tâche, à condition que RedTaskStack ou RedTaskTCB ne soit pas NULL. Tant que l'éditeur de liens de votre chaîne d'outils peut trouver de l'espace en RAM pour stocker les variables, la tâche sera créée avec succès.
- configSUPPORT_STATIC_ALLOCATION doit être mis à 1 dans FreeRTOSConfig.h si vous souhaitez utiliser le code précédent.

Création de tâches protégées par la mémoire

- Les tâches peuvent également être créées dans un environnement à mémoire protégée, ce qui garantit qu'une tâche n'accède qu'à la mémoire qui lui est spécifiquement attribuée.
- Il existe des implémentations de FreeRTOS qui tirent parti du matériel MPU embarqué.

Comprendre les états des tâches de FreeRTOS

- Comme expliqué au chapitre 2, Comprendre les tâches du RTOS, tous les changements de contexte entre les tâches s'effectuent en arrière-plan, ce qui est très pratique pour le programmeur responsable de l'implémentation des tâches.

- En effet, cela leur évite d'ajouter du code à chaque tâche qui tente d'équilibrer la charge du système.

- Bien que le code de la tâche n'effectue pas explicitement les transitions de l'état de la tâche, il interagit avec le noyau.

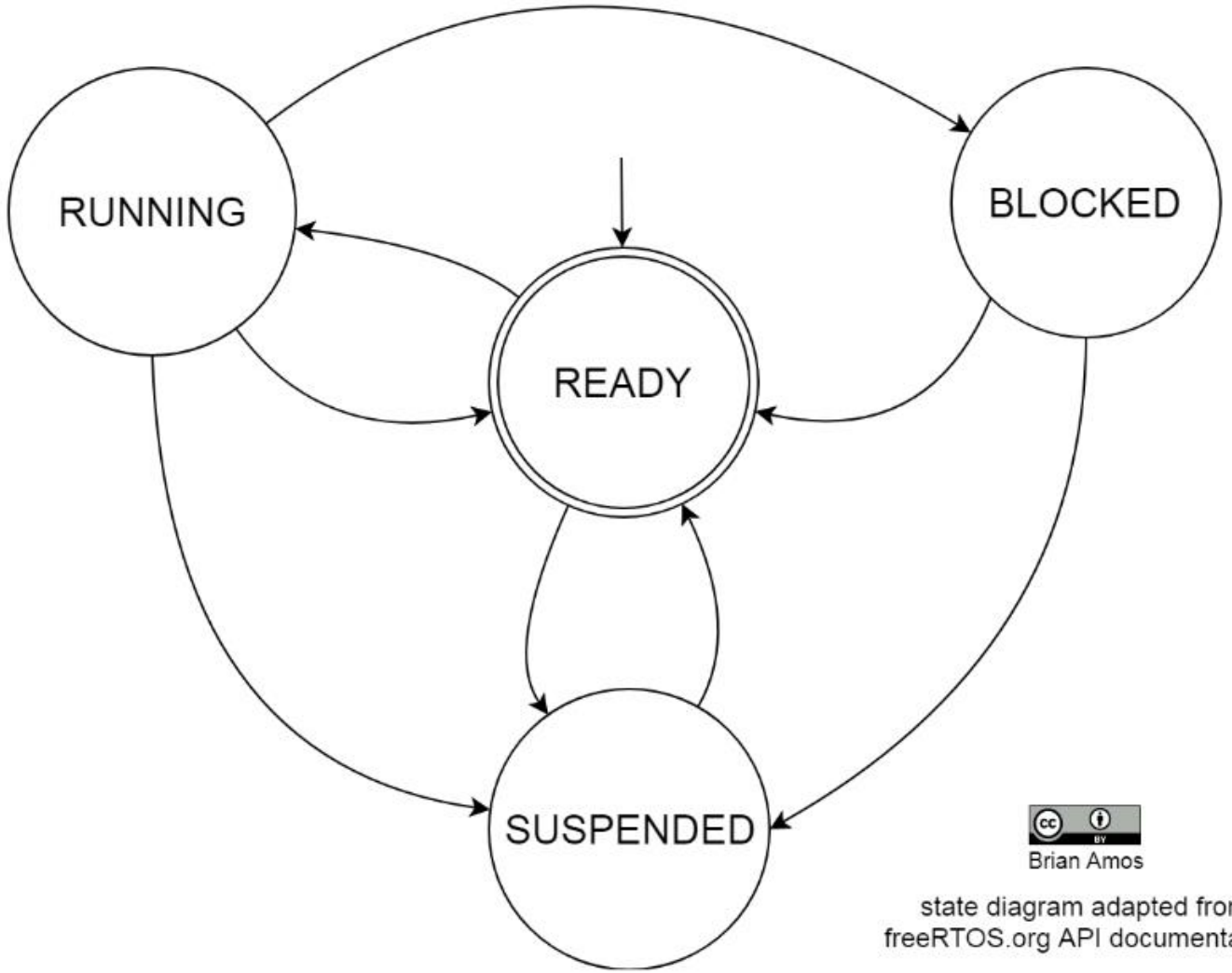
- Les appels à l'API FreeRTOS provoquent l'exécution de l'ordonnanceur du noyau, qui est responsable de la transition des tâches entre les états nécessaires

Comprendre les différents états des tâches

- Chaque transition illustrée dans le diagramme d'état suivant est causée soit par un appel API effectué par votre code, soit par une action entreprise par l’ordonnanceur.

- Voici un aperçu graphique simplifié des états et transitions possibles, accompagné d'une description de chacun d'entre eux (diapositive suivante).

États des tâches sous FreeRTOS



state diagram adapted from
freeRTOS.org API documentation

États des tâches FreeRTOS (1/4)

Une tâche peut se trouver dans l'un des états suivants :

Exécution

- Lorsqu'une tâche est en cours d'exécution, on dit qu'elle est dans l'état "Running".
- Elle utilise actuellement le processeur.
- Si le processeur sur lequel tourne le RTOS ne possède qu'un seul cœur, il ne peut y avoir qu'une seule tâche dans l'état "Running" à un moment donné.

Prêt

- Les tâches prêtes sont celles qui peuvent être exécutées (elles ne sont pas bloquées ou suspendues) mais qui ne le sont pas actuellement parce qu'une autre tâche de priorité égale ou supérieure est déjà en cours d'exécution.

Source : www.freertos.org/

États des tâches du FreeRTOS (2/4)

Bloqué

- Une tâche est dite bloquée si elle est en attente d'un événement temporel ou externe.
- Par exemple, si une tâche appelle vTaskDelay(), elle se bloquera (sera placée dans l'état Bloqué) jusqu'à ce que la période de delai ait expiré - un événement temporel.
- Les tâches peuvent également se bloquer pour attendre une file d'attente, un sémaphore, un groupe d'événements, une notification ou un événement sémaphore.
- Les tâches dans l'état bloqué ont normalement une période de temporisation, après laquelle la tâche est temporisée et débloquée, même si l'événement que la tâche attendait ne s'est pas produit.
- Les tâches en état bloqué n'utilisent pas de temps de traitement et ne peuvent pas être sélectionnées pour passer en état d'exécution.

Source : www.freertos.org/

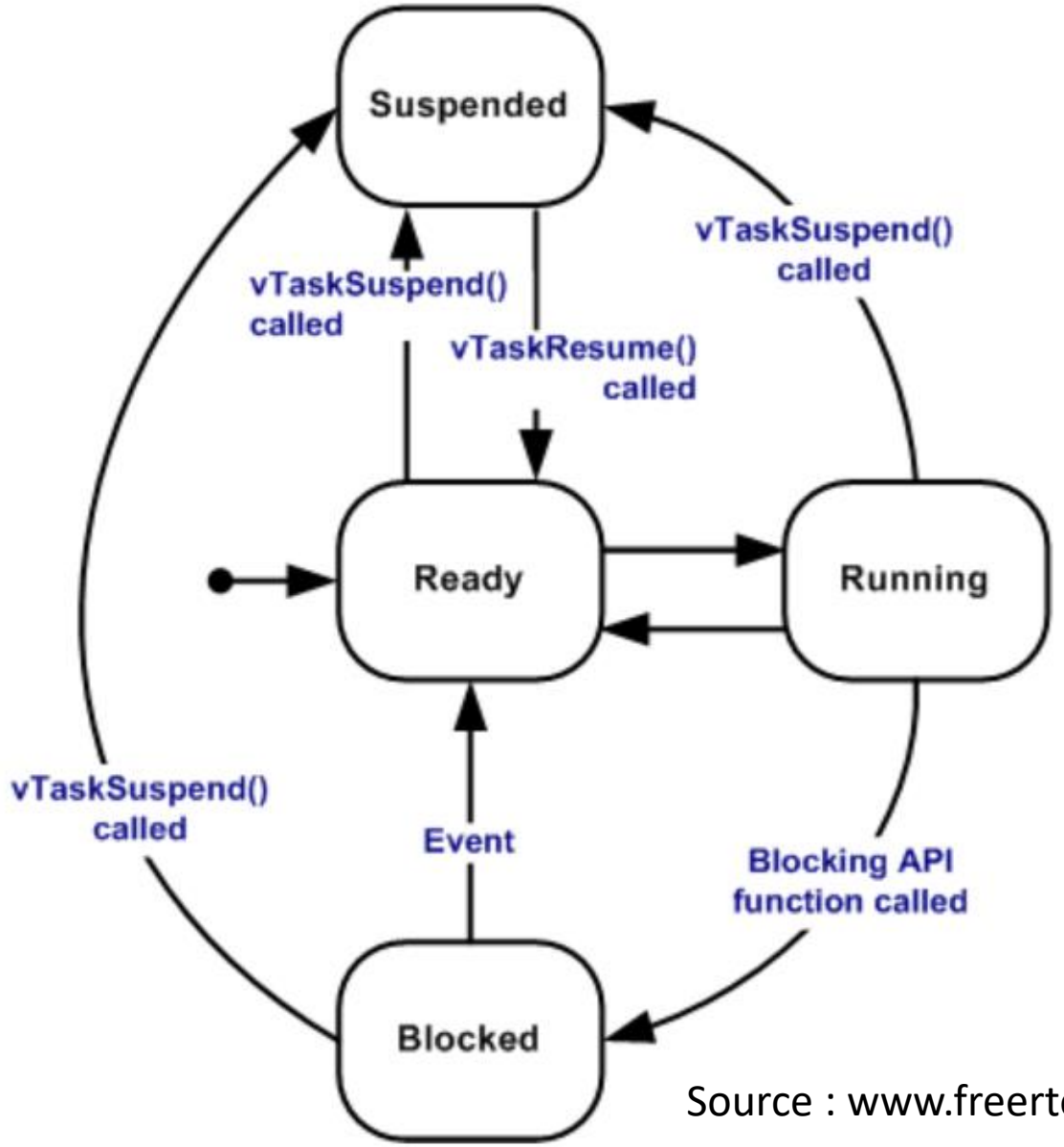
États des tâches du FreeRTOS (3/4)

Suspendu

- Comme les tâches bloquées, les tâches suspendues ne peuvent pas être sélectionnées pour passer à l'état d'exécution, mais les tâches suspendues n'ont pas de délai d'attente.
- Au lieu de cela, les tâches n'entrent ou ne sortent de l'état suspendu que lorsqu'elles en reçoivent l'ordre explicite par le biais des appels API vTaskSuspend() et xTaskResume() respectivement.

Source : www.freertos.org/

États des tâches du FreeRTOS (4/4)



Source : www.freertos.org/

Optimiser les états des tâches

- Des optimisations judicieuses peuvent être réalisées pour réduire au minimum le temps pendant lequel les tâches restent en état d'exécution.
- Étant donné qu'une tâche ne consomme une part importante du temps de l'unité centrale que lorsqu'elle est en cours d'exécution, il est généralement judicieux de minimiser le temps consacré à des tâches légitimes.

- Comme vous le verrez, l'interrogation des événements fonctionne, mais constitue généralement un gaspillage inutile des cycles de l'unité centrale. S'il est correctement équilibré avec les priorités des tâches, le système peut être conçu pour répondre aux événements importants tout en minimisant le temps d'utilisation de l'unité centrale.
- Il peut y avoir plusieurs raisons d'optimiser une application de cette manière.

Optimiser pour réduire le temps de travail de l'unité centrale (1/2)

- Souvent, un RTOS est utilisé parce que de nombreuses activités différentes doivent se dérouler presque simultanément.
- Lorsqu'une tâche doit agir à la suite d'un événement, il existe plusieurs façons de surveiller l'événement.
- On parle d'interrogation lorsqu'une valeur est lue en continu afin de capturer une transition.
- Un exemple serait l'attente d'une nouvelle lecture de l'ADC.
- Un sondage pourrait ressembler à ceci :

```
uint_fast8_t freshAdcReading = 0 ;
while(!freshAdcReading)
{
    freshAdcReading = checkAdc() ;
}
```

Optimiser pour réduire le temps de travail de l'unité centrale (2/2)

- Pour minimiser le temps qu'une tâche passe dans l'état "Running" (en demandant continuellement un changement), nous pouvons utiliser le matériel inclus dans le MCU pour effectuer le même contrôle sans l'intervention du CPU.

- Par exemple, les **routines de service d'interruption (ISR)** et l'**accès direct à la mémoire (DMA)** peuvent tous deux être utilisés pour décharger une partie du travail de l'unité centrale sur différents périphériques matériels inclus dans le MCU.

- Un ISR peut être interfacé avec les primitives RTOS pour notifier une tâche lorsqu'il y a un travail important à effectuer, éliminant ainsi la nécessité d'une interrogation intensive de l'unité centrale.

Optimiser pour augmenter les performances

- Parfois, les exigences en matière de synchronisation sont très strictes et nécessitent une faible gigue.
- Dans d'autres cas, il peut s'avérer nécessaire d'utiliser un périphérique nécessitant un débit important.
- Bien qu'il soit possible de respecter ces exigences temporelles en interrogeant une tâche hautement prioritaire, il est souvent plus fiable (et plus efficace) d'implémenter la fonctionnalité nécessaire dans un ISR.
- Il est également possible de ne pas impliquer le processeur du tout en utilisant le DMA.
- Ces deux options empêchent les tâches de dépenser des cycles de CPU inutiles dans des boucles d'interrogation et leur permettent de consacrer plus de temps à des tâches utiles.
- Comme les interruptions et les DMA peuvent fonctionner complètement en dessous du RTOS (sans intervention du noyau), ils peuvent avoir un effet positif considérable sur la création d'un système déterministe.

Optimiser pour minimiser la consommation d'énergie

- Avec la prédominance des applications alimentées par des batteries et la récolte d'énergie, les programmeurs ont une raison supplémentaire de s'assurer que le système utilise le moins possible de cycles d'unité centrale.
- Des idées similaires sont présentes dans la création de solutions économes en énergie, mais au lieu de maximiser le déterminisme, l'accent est souvent mis sur l'économie de cycles de l'unité centrale et le fonctionnement avec des fréquences d'horloge plus lentes.
- Il existe une fonctionnalité supplémentaire dans FreeRTOS qui est disponible pour l'expérimentation dans cet espace - la tâche IDLE sans tic-tac.
- Cela permet d'échanger la précision du timing contre une réduction de la fréquence d'exécution du noyau.
- Normalement, si le noyau est configuré pour une fréquence de 1 ms (attente toutes les millisecondes pour vérifier l'activité suivante), il se réveille et exécute le code à 1 kHz.
- Dans le cas d'une tâche IDLE sans tic-tac, le noyau ne se réveille que lorsque c'est nécessaire.
- Maintenant que nous avons abordé quelques points de départ sur la manière d'améliorer un système qui fonctionne déjà, intéressons-nous à quelque chose de plus grave : un système qui ne démarre pas du tout!

Résolution des problèmes de démarrage

- Disons que vous travaillez sur un projet et que les choses ne se sont pas déroulées comme prévu.
- Au lieu d'être récompensé par des lumières clignotantes, vous vous retrouvez face à un matériel qui ne l'est pas du tout.
- À ce stade, il est généralement préférable de faire fonctionner le débogueur, plutôt que de faire des suppositions aléatoires sur ce qui ne va pas et de modifier sporadiquement des sections du code.

Aucune de mes tâches n'est en cours d'exécution!

- Le plus souvent, les problèmes de démarrage dans les premières phases de développement seront causés par une allocation insuffisante d'espace dans le tas de FreeRTOS.
- Il en résulte généralement deux symptômes.

Échec de la création d'une tâche (1/3)

- Dans le cas suivant, le code se bloque avant l'exécution de l'ordonnanceur (aucune lumière ne clignote).
- Effectuez les étapes suivantes pour en déterminer la raison :
  1. À l'aide d'un débogueur, procédez à la création des tâches jusqu'à ce que vous trouviez la tâche incriminée. C'est facile à faire parce que toutes nos tentatives de création de tâches ne progresseront que si les tâches ont été créées avec succès.
  2. Dans ce cas, vous verrez que xTaskCreate ne renvoie pas pdPASS lors de la création de BlueTask. Le code suivant demande une pile de 50 Ko pour BlueTask (diapositive suivante).

La création de la tâche a échoué (2/3)

```
int main(void)
{
  HWInit() ;

  if (xTaskCreate(GreenTask, "GreenTask",
                  TAILLE DE LA PILE, NULL,
                  tskIDLE_PRIORITY + 2, NULL) != pdPASS)
      { while(1) ; }

  //code ne progressera pas au-delà de assert_failed (appelé par
  //assert_param sur les assertions échouées)
  retval = (xTaskCreate(BlueTask, "BlueTask",
                STACK_SIZE*100,NULL,
                tskIDLE_PRIORITY + 1, &blueTaskHandle) ;
  assert_param(retVal == pdPASS) ;
```

Échec de la création d'une tâche (3/3)

- Voici le code pour assert_failed.
- La boucle infinie while permet de retrouver très facilement la ligne incriminée à l'aide d'une sonde de débogage et d'un examen de la pile d'appels :

```
void assert_failed(uint8_t *file, uint32_t line)
{
  SEGGER_SYSVIEW_PrintfHost("Échec de l'assertion : fichier %s")
                            sur la ligne %d\r\n", file, line) ;
  while(1) ;
}
```

Remarques importantes

- La mémoire vive des systèmes embarqués à base de MCU est généralement une ressource rare.
- En augmentant l'espace de tas disponible pour FreeRTOS (configTOTAL_HEAP_SIZE), vous réduirez la quantité de RAM disponible pour le code non-RTOS.

Résumé

- Dans ce chapitre, nous avons abordé les différentes manières de définir les tâches et comment démarrer l'ordonnanceur FreeRTOS.

- En cours de route, nous avons abordé d'autres exemples d'utilisation d'Ozone, de SystemView et de STM32CubeIDE (ou de tout autre IDE basé sur Eclipse CDT).

- Toutes ces informations ont été utilisées pour créer une démonstration en direct qui associe tous les concepts RTOS relatifs à la création de tâches aux mécanismes de chargement et d'analyse de code fonctionnant sur du matériel embarqué.

- Des suggestions ont également été faites sur la manière de ne pas surveiller les événements (polling).

- Dans le chapitre suivant, nous présenterons ce que vous devriez utiliser pour la surveillance des événements. Nous aborderons les différentes manières de mettre en œuvre la signalisation et la synchronisation entre les tâches, le tout à l'aide d'exemples.

- Il y aura BEAUCOUP de code et d'analyses pratiques utilisant la carte Nucleo.