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

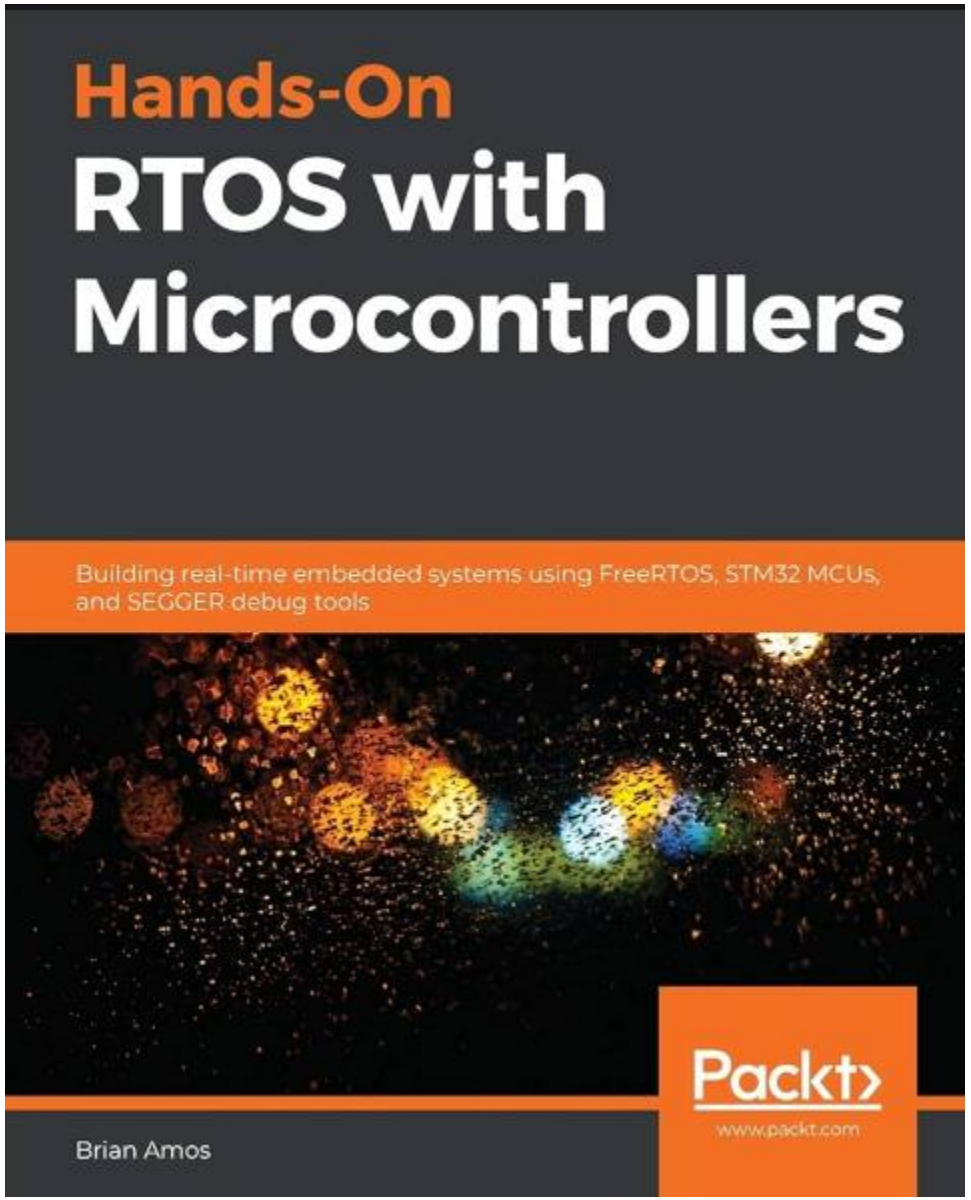
CEG4566/CSI4541/SEG4545

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 11 :
Communication entre tâches

Plan

- Passage des données dans les files d'attente par valeur
- Transmission de données par référence dans les files d'attente
- Notifications directes de tâches

Passage des données dans les files d'attente par valeur

- Comme les sémaphores et les mutex, les files d'attente sont parmi les structures les plus utilisées (et mises en œuvre) lorsqu'il s'agit d'opérer sur plusieurs tâches exécutées de manière asynchrone.
- On les trouve dans presque tous les systèmes d'exploitation, et il est donc très utile de savoir comment les utiliser.
- Nous allons examiner plusieurs façons d'utiliser les files d'attente et d'interagir avec elles pour affecter l'état d'une tâche.

- Dans les exemples suivants, nous allons apprendre à utiliser les files d'attente comme moyen d'envoyer des commandes à une machine à états LED.
- Tout d'abord, nous examinerons un cas d'utilisation très simple, en passant une valeur unique d'un octet à une file d'attente et en l'exploitant.

Passage d'un octet par valeur (1/3)

- Dans cet exemple, un seul uint8_t est configuré pour transmettre des énumérations individuelles (LED_CMDS), définissant l'état d'une LED à la fois ou de toutes les LED (allumées/éteintes).
- Voici un résumé de ce qui est couvert dans cet exemple :
  - ledCmdQueue : Une file de valeurs d'un octet (uint8_t) représentant une énumération définissant les états des DEL.
  - recvTask : Cette tâche reçoit un octet de la file d'attente, exécute l'action souhaitée et tente immédiatement de recevoir l'octet suivant de la file d'attente.
  - sendingTask : Cette tâche envoie des valeurs énumérées à la file d'attente à l'aide d'une simple boucle, avec un délai de 200 ms entre chaque envoi (pour que les DEL qui s'allument et s'éteignent soient visibles).

Passage d'un octet par valeur (2/3)

Commençons donc :

1. Mettre en place un enum pour nous aider à décrire les valeurs qui sont passées dans la file d'attente :

Voici un extrait du fichier mainQueueSimplePassByValue.c :

```
typedef enum
{
        ALL_OFF = 0,
        RED_ON = 1,
        RED_OFF = 2,
        BLUE_ON = 3,
        BLUE_OFF= 4,
        GREEN_ON = 5,
        GREEN_OFF = 6,
        ALL_ON = 7
}LED_CMDS ;
```

Passage d'un octet par valeur (3/3)

```
static QueueHandle_t ledCmdQueue = NULL;
```

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize ) ;
```

uxQueueueLength : Le nombre maximum d'éléments que la file d'attente peut contenir
uxItemSize : Taille (en octets) de chaque élément de la file d'attente
Valeur de retour : Un handle vers la file d'attente créée (ou NULL en cas d'erreur)

xQueueCreate

```
ledCmdQueue = xQueueCreate(2, sizeof(uint8_t)) ;
assert_param(ledCmdQueue != NULL) ;
```

- La file d'attente peut contenir jusqu'à 2 éléments.
- Chaque élément est dimensionné pour contenir uint8_t (un seul octet est suffisamment grand pour stocker la valeur de n'importe quelle énumération que nous avons explicitement définie).
- xQueueCreate renvoie un handle à la file d'attente créée, qui est stocké dans ledCmdQueue. Ce "handle" est un identifiant global qui sera utilisé par les différentes tâches pour accéder à la file d'attente.

recvTask()

```
void recvTask( void* NotUsed )
{
  uint8_t nextCmd = 0 ;

  while(1)
  {
    if(xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY) == pdTRUE)
    {
      switch(nextCmd)
      {
        case ALL_OFF :
                RedLed.Off() ;
                GreenLed.Off() ;
                BlueLed.Off() ;
        break ;
        case GREEN_ON :
              GreenLed.On() ;
          break ;
```

if(xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY) == pdTRUE)

- La poignée (Handle) ledCmdQueue est utilisée pour accéder à la file d'attente.
- Un uint8_t local, nextCmd, est défini sur la pile. L'adresse de cette variable (un pointeur) est transmise. xQueueReceive copie la prochaine énumération LED_CMD (stockée sous forme d'octet dans la file d'attente) dans nextCmd.
- Un délai infini est utilisé pour cet accès, c'est-à-dire que cette fonction ne reviendra jamais si rien n'est ajouté à la file d'attente (de la même manière que les délais des appels de l'API mutex et sémaphore).

sendingTask

La sendingTask est une simple boucle while qui utilise la connaissance préalable des valeurs de l'énumération pour transmettre différentes valeurs de LED_CMDS à ledCmdQueue :

```
void sendingTask( void* NotUsed )
{
  while(1)
  {
    for(int i = 0 ; i < 8 ; i++)
    {
      uint8_t ledCmd = (LED_CMDS) i ;
      xQueueSend(ledCmdQueue, &ledCmd, portMAX_DELAY) ;
      vTaskDelay(200/portTICK_PERIOD_MS) ;
    }
  }
}
```

xQueueSend()

La sendingTask est une simple boucle while qui utilise la connaissance préalable des valeurs de l'énumération pour transmettre différentes valeurs de LED_CMDS à ledCmdQueue :

Les arguments de la fonction xQueueSend() du côté de l'envoi sont presque identiques à ceux de la fonction xQueueReceive() du côté de la réception, la seule différence étant que nous envoyons cette fois des données à la file d'attente :

```
xQueueSend(ledCmdQueue, &ledCmd, portMAX_DELAY) ;
```

- ledCmdQueue : L'identifiant de la file d'attente à laquelle envoyer les données
- &ledCmd : L'adresse des données à transmettre à la file d'attente
- portMax_DELAY : Le nombre de ticks RTOS à attendre pour que l'espace de la file d'attente devienne disponible (si la file d'attente est pleine).

Transmission d'un type de données composite par valeur

- Les files d'attente FreeRTOS (et la plupart des autres fonctions de l'API FreeRTOS) prennent en argument des void* pour les différents types de données utilisés.

- Il s'agit d'offrir une certaine flexibilité au rédacteur de l'application de la manière la plus efficace possible.

- Étant donné que void* est simplement un pointeur sur n'importe quoi et que la taille des éléments de la file d'attente est définie lors de sa création, les files d'attente peuvent être utilisées pour transmettre n'importe quoi entre les tâches.

Type de données composite

```
typedef struct
{
  uint8_t redLEDState : 1 ; //spécifier que cette variable a une largeur de 1 bit
  uint8_t blueLEDState : 1 ; //spécifier que cette variable a une largeur de 1 bit
  uint8_t greenLEDState : 1 ; //spécifier que cette variable a une largeur de 1 bit
  uint32_t msDelayTime ; //nombre de mS pour rester dans cet état
}LedStates_t ;
```

Nous allons également créer une file d'attente capable de contenir huit copies de la structure LedStates_t entière :

```
ledCmdQueue = xQueueCreate(8, sizeof(LedStates_t)) ;
```

recvTask attend qu'un élément soit disponible dans la file d'attente ledCmdQueue et l'exploite (en activant ou désactivant les DEL selon les besoins) :

mainQueueCompositePassByValue.c recvTask :

```
if(xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY) == pdTRUE)
{
   if(nextCmd.redLEDState == 1)
      RedLed.On() ;
   else
      RedLed.Off() ;
   if(nextCmd.blueLEDState == 1)
      BlueLed.On() ;
   else
      BlueLed.Off() ;
   if(nextCmd.greenLEDState == 1)
      GreenLed.On() ;
   else
      GreenLed.Off() ;
}
vTaskDelay(nextCmd.msDelayTime/portTICK_PERIOD_MS) ;
```

Voici les responsabilités de la boucle primaire de recvTask

- Chaque fois qu'un élément est disponible dans la file d'attente, chaque champ de la structure est évalué et l'action appropriée est entreprise. Les trois DEL sont mises à jour avec une seule commande, envoyée à la file d'attente.

- Le champ msDelayTime nouvellement créé est également évalué (il est utilisé pour ajouter un délai avant que la tâche ne tente à nouveau de recevoir des informations de la file d'attente). C'est ce qui ralentit suffisamment le système pour que les états des DEL soient visibles.

mainQueueCompositePassByValue.c sendingTask

```
while(1)
 {
  nextStates.redLEDState = 1 ;
  nextStates.greenLEDState = 1 ;
  nextStates.blueLEDState = 1 ;
  nextStates.msDelayTime = 100 ;

  xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY) ;

  nextStates.blueLEDState = 0 ; //Éteindre uniquement la LED bleue
  nextStates.msDelayTime = 1500 ;
  xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY) ;

  nextStates.greenLEDState = 0;//éteindre uniquement la LED verte
  nextStates.msDelayTime = 200 ;
  xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY) ;

  nextStates.redLEDState = 0 ;
  xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY) ;
 }
```

La boucle de sendingTask envoie quelques commandes à ledCmdQueueue

- sendingTask est un peu différente de la précédente. Maintenant, puisqu'une structure est transmise, nous pouvons accéder à chaque champ, en définissant plusieurs champs avant d'envoyer nextStates à la file d'attente.

- Chaque fois que xQueueSend est appelé, le contenu de nextStates est copié dans la file d'attente avant de passer à autre chose. Dès que xQueueSend() est renvoyé avec succès, la valeur de nextStates est copiée dans la file d'attente ; nextStates n'a pas besoin d'être conservé.

xTaskCreate()

- Pour bien faire comprendre que la valeur de nextStates est copiée dans la file d'attente, cet exemple modifie les priorités des tâches de manière à ce que la file d'attente soit entièrement remplie par sendingTask avant d'être vidée par recvTask.
- Pour ce faire, nous donnons à sendingTask une priorité supérieure à celle de revcTask. Voici à quoi ressemblent les définitions de nos tâches (les assertions sont présentes dans le code mais ne sont pas montrées ici pour réduire l'encombrement) :

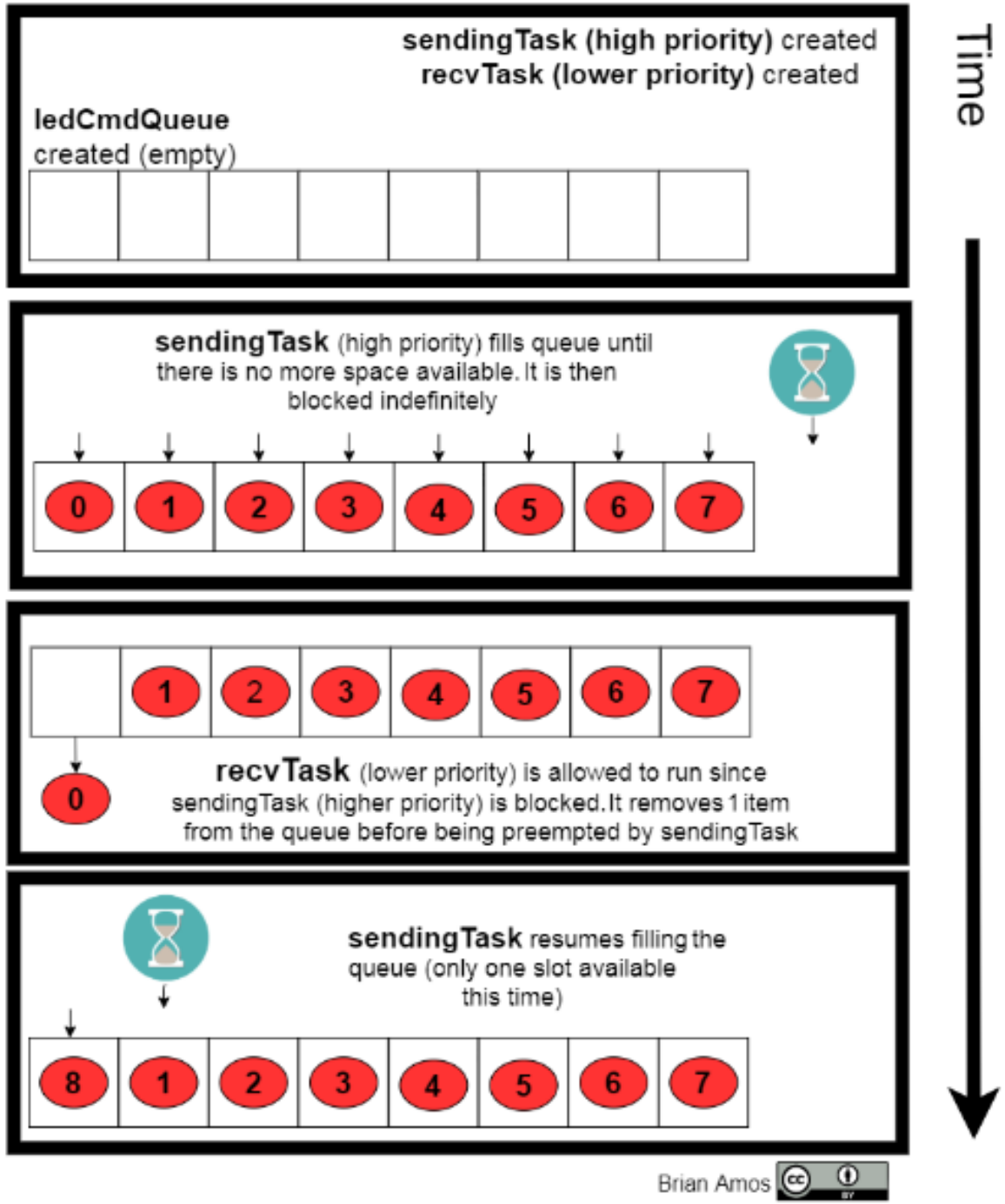
```
xTaskCreate(recvTask, "recvTask", STACK_SIZE, NULL, tskIDLE_PRIORITY + 1,
        NULL) ;
xTaskCreate(sendingTask, "sendingTask", STACK_SIZE, NULL,
        configMAX_PRIORITIES - 1, NULL) ;
```

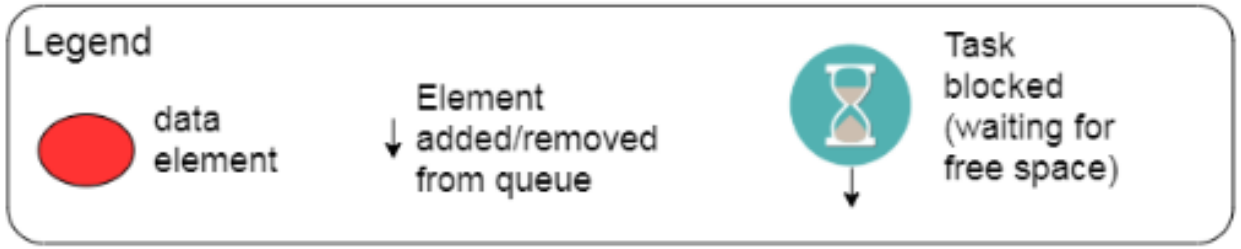
Établissement de priorités

Cette configuration de priorité permet à sendingTask d'envoyer à plusieurs reprises des données à la file d'attente jusqu'à ce qu'elle soit pleine (parce que sendingTask a une priorité plus élevée).

Une fois la file d'attente remplie, sendingTask se bloque et permet à recvTask de retirer un élément de la file d'attente.

Tâche hautement prioritaire envoi dans la file d'attente





Transmission de données par référence dans les files d'attente

- Lorsque de gros éléments doivent être mis en file d'attente, il est judicieux de passer les éléments par référence. Voici un exemple de structure plus importante.
- Une fois que le compilateur a rempli cette structure, sa taille est de 264 octets :

mainQueueCompositePassByReference.c :

```
#define MAX_MSG_LEN 256
typedef struct
{
  uint32_t redLEDState : 1 ;
  uint32_t blueLEDState : 1 ;
  uint32_t greenLEDState : 1 ;
  uint32_t msDelayTime ; //nombre de mS pour rester dans cet état
  //un tableau pour stocker des chaînes de caractères jusqu'à 256 caractères
  char message[MAX_MSG_LEN] ;
}LedStates_t ;
```

ledCmdQueue

- Plutôt que de copier 264 octets à chaque fois qu'un élément est ajouté ou supprimé de ledCmdQueue, nous pouvons définir ledCmdQueue comme contenant un pointeur (4 octets sur Cortex-M) sur LedStates_t :

- ledCmdQueue = xQueueCreate(8, sizeof(LedStates_t*)) ;

Passage par la valeur :

- ledCmdQueue size : ~ 2 KB (264 bytes * 8 elements).
- 264 octets copiés à chaque appel de xQueueSend() ou xQueueReceive().
- La copie originale de LedStates_t qui est ajoutée à la file d'attente peut être éliminée immédiatement (une copie complète est présente dans la file d'attente).

Passage par référence

- taille de ledCmdQueue : 32 octets (4 octets * 8 éléments).
- 4 octets copiés (la taille d'un pointeur) chaque fois que xQueueSend() ou xQueueReceive() est appelé.
- La copie originale de LedStates_t qui est ajoutée à la file d'attente doit être conservée jusqu'à ce qu'elle ne soit plus nécessaire (il s'agit de la seule copie dans le système ; seul un pointeur vers la structure originale a été mis en file d'attente).

Créer un pointeur sur la variable et transmettre l'adresse du pointeur.

```
void sendingTask( void* NotUsed )
{
  LedStates_t* state1Ptr = &ledState1 ;
  LedStates_t* state2Ptr = &ledState2 ;

  while(1)
  {
    xQueueSend(ledCmdQueue, &state1Ptr, portMAX_DELAY) ;
    xQueueSend(ledCmdQueue, &state2Ptr, portMAX_DELAY) ;
  }
}
```

recvTask()

Pour recevoir des éléments, il suffit de définir un pointeur du type de données approprié et de transmettre l'adresse au pointeur.

```
void recvTask( void* NotUsed )
{
   LedStates_t *nextCmd ;

   while(1)
   {
      if(xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY) ==
                                          pdTRUE)
      {
         if(nextCmd->redLEDState == 1)
         RedLed.On() ;
```

Notifications directes de tâches

- Les files d'attente sont un excellent outil de travail pour un RTOS en raison de leur flexibilité.
- Parfois, cette flexibilité n'est pas nécessaire et nous préférons une solution plus légère.
- Les notifications directes de tâches sont similaires aux autres mécanismes de communication évoqués, à ceci près qu'elles ne nécessitent pas l'instanciation préalable de l'objet de communication dans la mémoire vive.
- Ils sont également plus rapides que les sémaphores ou les files d'attente (entre 35% et 45% plus rapides).
- Ils présentent certaines limites, les deux plus importantes étant qu'une seule tâche peut être notifiée à la fois et que les notifications peuvent être envoyées par les BVR mais non reçues.

Notifications directes de tâches (1/2)

- Les notifications de tâches directes ont deux composantes principales :
  - la notification elle-même (qui se comporte de la même manière qu'un sémaphore ou une file d'attente lors du déblocage d'une tâche) et une valeur de notification de 32 bits.

- La valeur de la notification est facultative et peut être utilisée de différentes manières.

- Un notifiant a la possibilité d'écraser la valeur entière ou d'utiliser la valeur de notification comme s'il s'agissait d'un champ de bits et d'activer un seul bit.

- La définition de bits individuels peut s'avérer utile pour signaler différents comportements dont vous souhaitez que la tâche soit informée sans avoir recours à une implémentation plus compliquée basée sur des commandes et des files d'attente.

Notifications directes de tâches (2/2)

- Prenons l'exemple de nos diodes électroluminescentes.

- Si nous voulions créer un simple gestionnaire de DEL qui réponde rapidement à une demande de modification, une file d'attente à plusieurs éléments ne serait pas nécessaire ; nous pouvons utiliser la valeur de notification intégrée de 32 bits de large à la place.

En dehors de main, nous définirons quelques bitmasks et un gestionnaire de tâches

```
#define RED_LED_MASK 0x0001
#define BLUE_LED_MASK 0x0002
#define GREEN_LED_MASK 0x0004
static xTaskHandle recvTaskHandle = NULL ;
```

Dans main, nous créerons la tâche recvTask et lui transmettrons le handle à remplir.

```
retVal = xTaskCreate(recvTask, "recvTask", STACK_SIZE, NULL,
                     tskIDLE_PRIORITY + 2, &recvTaskHandle) ;
assert_param( retVal == pdPASS) ;
assert_param(recvTaskHandle != NULL) ;
```

Tâche recevant la notification

La tâche qui reçoit la notification est configurée pour attendre la prochaine notification entrante et évaluer le masque de chaque DEL, en activant ou désactivant les DEL en conséquence.

```
void recvTask( void* NotUsed )
{
   while(1)
   {
      uint32_t notificationvalue = ulTaskNotifyTake( pdTRUE,
                                   portMAX_DELAY ) ;
      if((notificationvalue & RED_LED_MASK) != 0)
         RedLed.On() ;
      else
         RedLed.Off() ;
```

Tâche pour l’envoi

La tâche d'envoi est configurée pour envoyer une notification, en remplaçant toutes les notifications existantes. Ainsi, xTaskNotify renvoie toujours pdTRUE.

```
void sendingTask( void* NotUsed )
{
   while(1)
   {
      xTaskNotify( recvTaskHandle, RED_LED_MASK,
                  eSetValueWithOverwrite) ;
      vTaskDelay(200) ;
```

Comparaison entre les notifications directes de tâches et les files d'attente

Par rapport aux files d'attente, les notifications de tâches présentent les caractéristiques suivantes :

Ils ont toujours une capacité de stockage d'exactement un entier de 32 bits.

- Ils n'offrent pas de moyen d'attendre pour envoyer une notification à une tâche occupée ; la notification écrasera une notification existante ou reviendra immédiatement sans écrire.
- Elles ne peuvent être utilisées qu'avec un seul récepteur (puisque la valeur de la notification est stockée dans la tâche réceptrice).
- Ils sont plus rapides.

Résumé

- Vous avez maintenant appris les bases de l'utilisation des files d'attente dans divers scénarios, comme le passage d'éléments simples et composites par valeur et par référence.

- Vous connaissez les avantages et les inconvénients de l'utilisation de files d'attente pour stocker des références à des objets et vous savez quand il convient d'utiliser cette méthode.

- Nous avons également abordé certaines interactions détaillées entre les files d'attente, les tâches et les priorités des tâches. Nous avons terminé par un exemple simple et concret de l'utilisation des notifications de tâches pour piloter efficacement une petite machine à états.

- Au fur et à mesure que vous vous habituerez à utiliser les RTOS pour résoudre une grande variété de problèmes, vous trouverez de nouvelles façons créatives d'utiliser les files d'attente et les notifications de tâches.

- Les tâches, les files d'attente, les sémaphores et les mutex sont véritablement les éléments constitutifs des applications basées sur le RTOS et vous aideront à aller loin.

- Nous n'en avons pas encore fini avec ces éléments - il reste encore beaucoup de matériel plus avancé à couvrir en ce qui concerne l'utilisation de toutes ces primitives dans le contexte des ISR, ce qui est la prochaine étape !