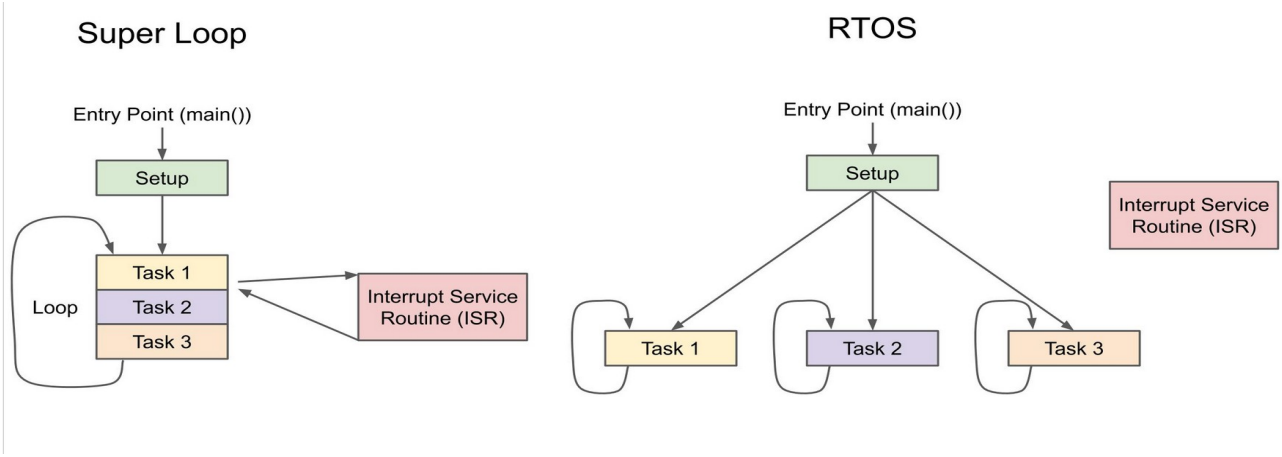
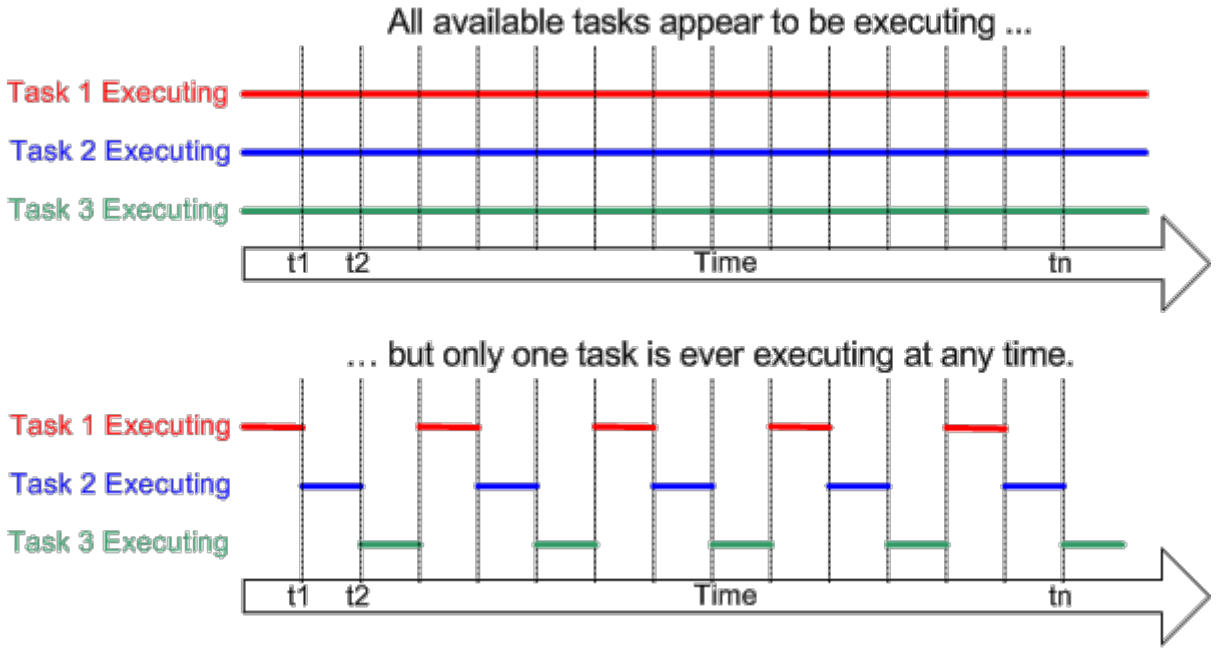
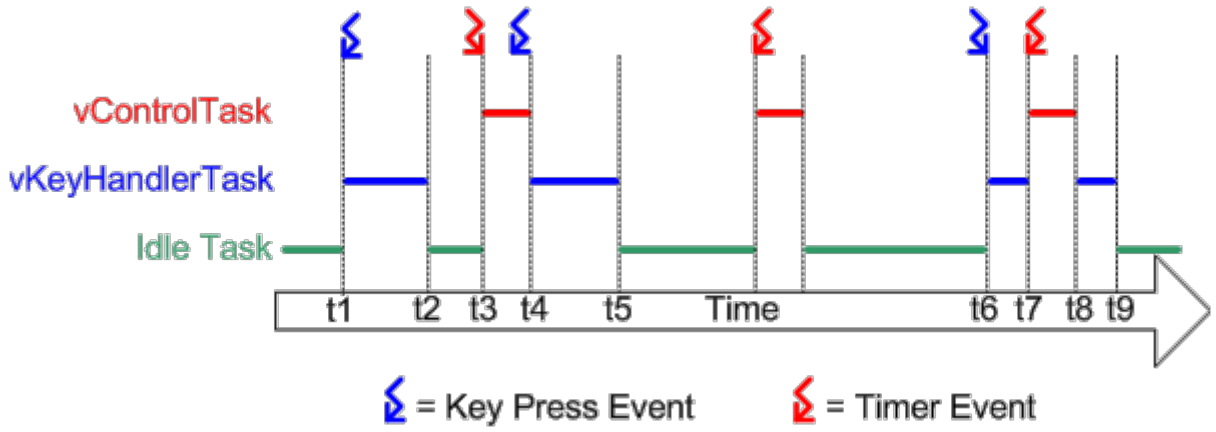
Initiation à FreeRTOS

Pourquoi un OS temps réel ?





Exemple d'occupation d'un processeur :



Une tâche

Une tâche est simplement une fonction dont la structure est la même que celle d'un programme classique en informatique embarqué :

```
void maTache(void *parametres)
{

    /* Initialisations de la tâche ici */

    while (1) // boucle infinie
    {

        /* Code de la tâche ici */

    }
}
```

FreeRTOS va gérer l'ordre d'exécution des tâches avec un ordonnanceur (scheduler).

Pour qu'une tâche s'exécute, il faut la créer avec un appel à la fonction `xTaskCreate()` comme ci-dessous :

```
xTaskCreate(
    maTache, /* Fonction de la tâche. */
    "Ma tâche", /* Nom de la tâche. */
    10000, /* Taille de la pile de la tâche */
    NULL, /* Paramètres de la tâche, NULL si pas de paramètre */
    1, /* Priorité de la tâche */
    NULL); /* Pointeur pour récupérer le « handle » de la tâche, optionnel */
```

La fonction retourne `pdPASS` si la tâche est créée, sinon un code d'erreur.

Plus la priorité d'une tâche est petite moins la tâche est prioritaire.
Le niveau de priorité va de `0` à `(configMAX_PRIORITIES-1)`.
La valeur de `configMAX_PRIORITIES` est à 25 pour l'ESP32 (donc priorité de 0 à 24).
Pour connaître la priorité d'une tâche :

```
priorite = uxTaskPriorityGet(handle);
```

Pour changer la priorité d'une tâche :

```
vTaskPrioritySet(handle, priorite);
```

Les tâches sont identifiées par leur descripteur, un « handle ».
Si le « `handle` » vaut `NULL`, les fonctions s'appliquent à la tâche en cours.

Le nombre de tâches n'est limité que par le matériel.

La fonction `loop()` du framework Arduino est une tâche.

1. Testez l'exemple 1 et commentez le fonctionnement obtenu.

2. Retrouvez et affichez dans le terminal la priorité de `maTache` et celle du `loop`.

L'ESP32 possède deux cœurs. La fonction `xPortGetCoreID()` permet de savoir sur lequel s'exécute la tâche.

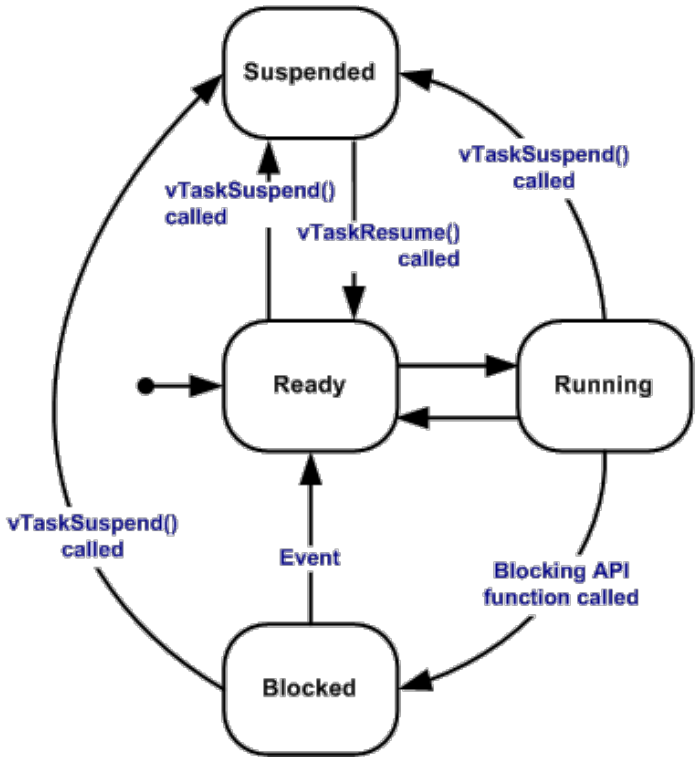
3. Modifiez le code pour afficher sur quel cœur s'exécutent les deux tâches (`maTache` et `loop`).

Pour créer une tâche, on peut aussi utiliser la fonction `xTaskCreatePinnedToCore` qui permet de préciser sur quel cœur la tâche va être exécutée.

4. Modifiez le code de l'exemple pour que `maTache` s'exécute sur le cœur de votre choix.

États d'une tâche

Lorsqu'une tâche est créée, elle passe à l'état « prête » (Ready). L'ordonnanceur évalue la priorité des tâches prêtes et choisit la plus prioritaire pour lui allouer un temps de processeur. Si plusieurs tâches prêtes sont de même priorité, l'ordonnanceur va leur allouer le processeur à tour de rôle (Round-Robin). Avec FreeRTOS, si la tâche de plus haute priorité est toujours prête lors de l'évaluation de l'ordonnanceur, alors les tâches de priorité inférieure ne s'exécuteront jamais. On parle de famine (starvation). C'est au programmeur de s'assurer que toutes les tâches peuvent avoir du temps de processeur.
Une tâche sélectionnée par l'ordonnanceur pour avoir du temps processeur passe à l'état « en exécution » (Running).
Si une tâche exécute une fonction bloquante de l'API de FreeRTOS, un délai ou une attente d'évènement, elle passe à l'état « bloquée » (Blocked). Lors de l'évènement attendu, fin du délai par exemple, elle passe à l'état « prête ».



Valid task state transitions

5. Testez l'exemple 2 et commentez le fonctionnement obtenu.

Une tâche peut être suspendue (suspend) dans n'importe quel état. Elle reprendra (resume) alors à l'état « prête ».
Pour suspendre une tâche :

```
vTaskSuspend(handle);
```

Pour reprendre une tâche :

```
vTaskResume(handle);
```

Pour terminer une tâche :

```
vTaskDelete(handle);
```

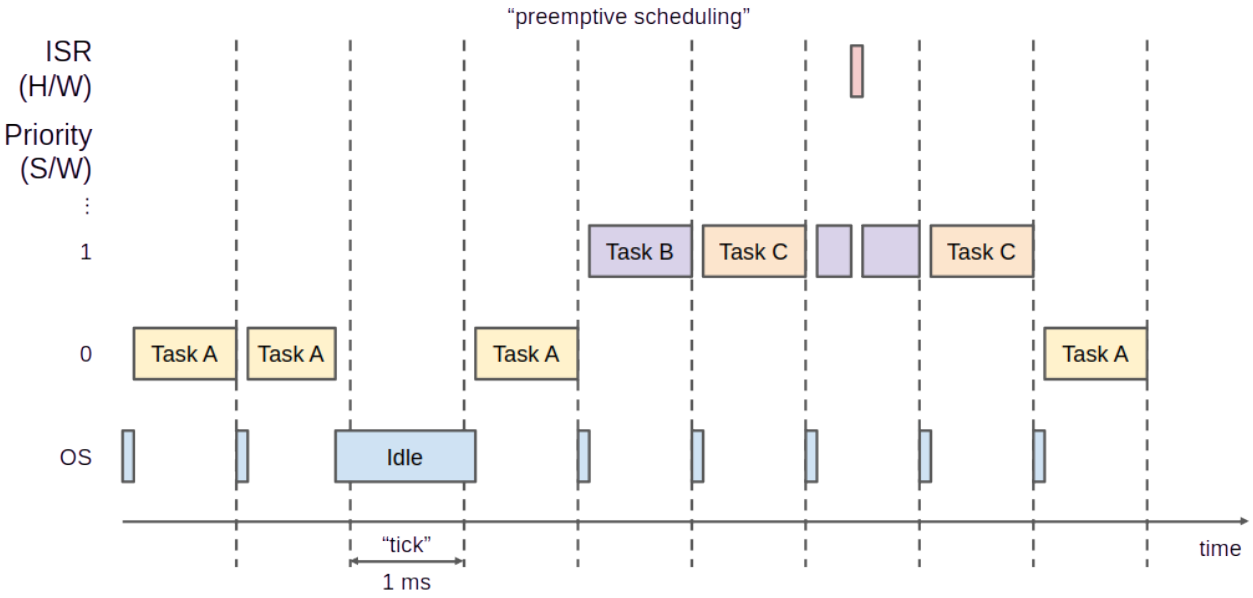
6. Modifiez l'exemple 2 pour :
   - suspendre la tâche 1 quand sa variable i arrive à 5.
   - terminer la tâche 2 quand la variable i du loop arrive à 10.
   - reprendre la tâche 1 quand la variable i du loop arrive à 15.

Remarque :
Si on ne veut pas utiliser le `loop` du framework Arduino, il suffit de terminer le `setup` par un appel à `vTaskDelete(NULL);`

Gestion du temps

FreeRTOS compte le temps en ticks. Ici le tick dure 1ms mais il est possible que cette valeur change dans le futur. Pour rendre le code indépendant de cette valeur, la fonction `pdMS_TO_TICKS()` convertit les ms en ticks.
Avec le framework Arduino, la fonction `delay(ms)` est adaptée à FreeRTOS car elle correspond à `vTaskDelay(ms / portTICK_PERIOD_MS)`.

Exemple d'occupation d'un processeur :



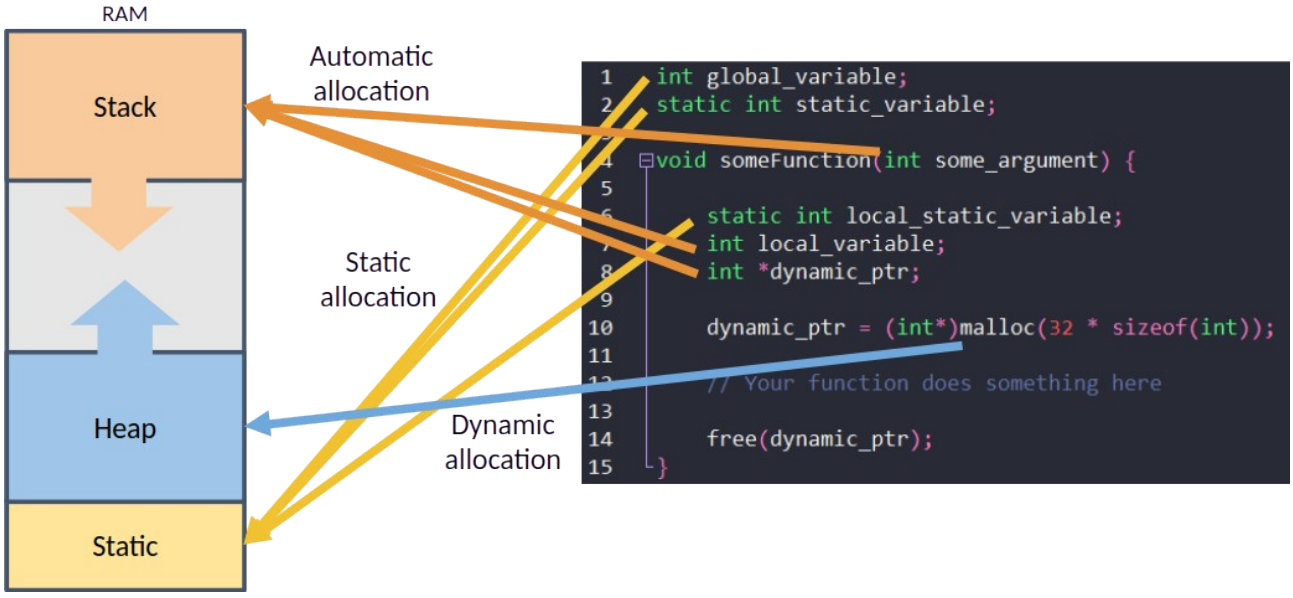
L'ordonnanceur s'exécute tous les « ticks » grâce à une interruption (tick interrupt). Il peut préempter une tâche sans que celle-ci coopère (sans qu'elle rende le processeur).

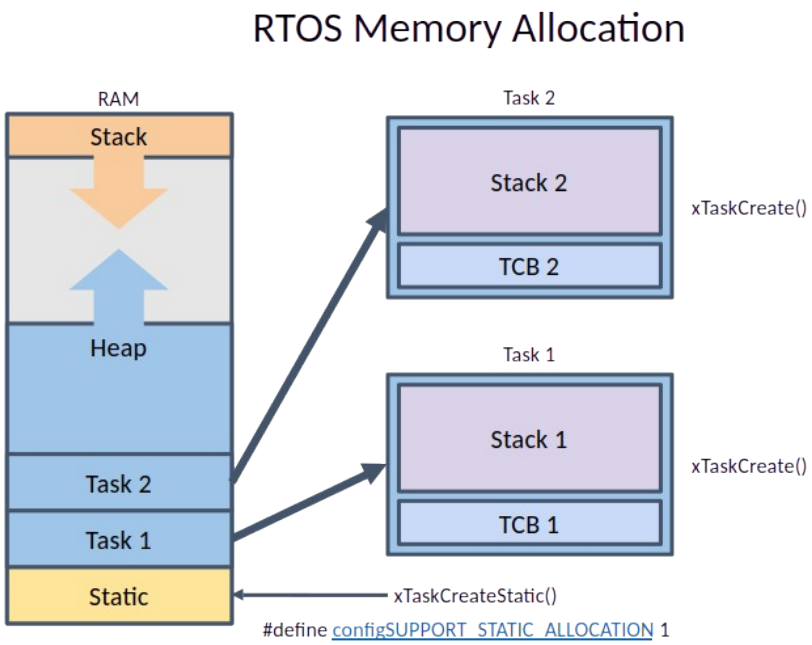
Pour qu'un traitement s'effectue de manière périodique, on peut utiliser la structure de l'exemple 3.

7. Testez le comportement, analysez le code et expliquez la différence avec un simple délai.

8. Modifiez le code pour mettre à 1 une broche de l'ESP32 au début de la boucle de la tâche périodique et pour la remettre à 0 juste avant le délai. Observez la broche à l'oscilloscope. Effectuez des mesures de temps et conclure.

La mémoire





Une tâche est décrite par un TCB : Task Control Block.

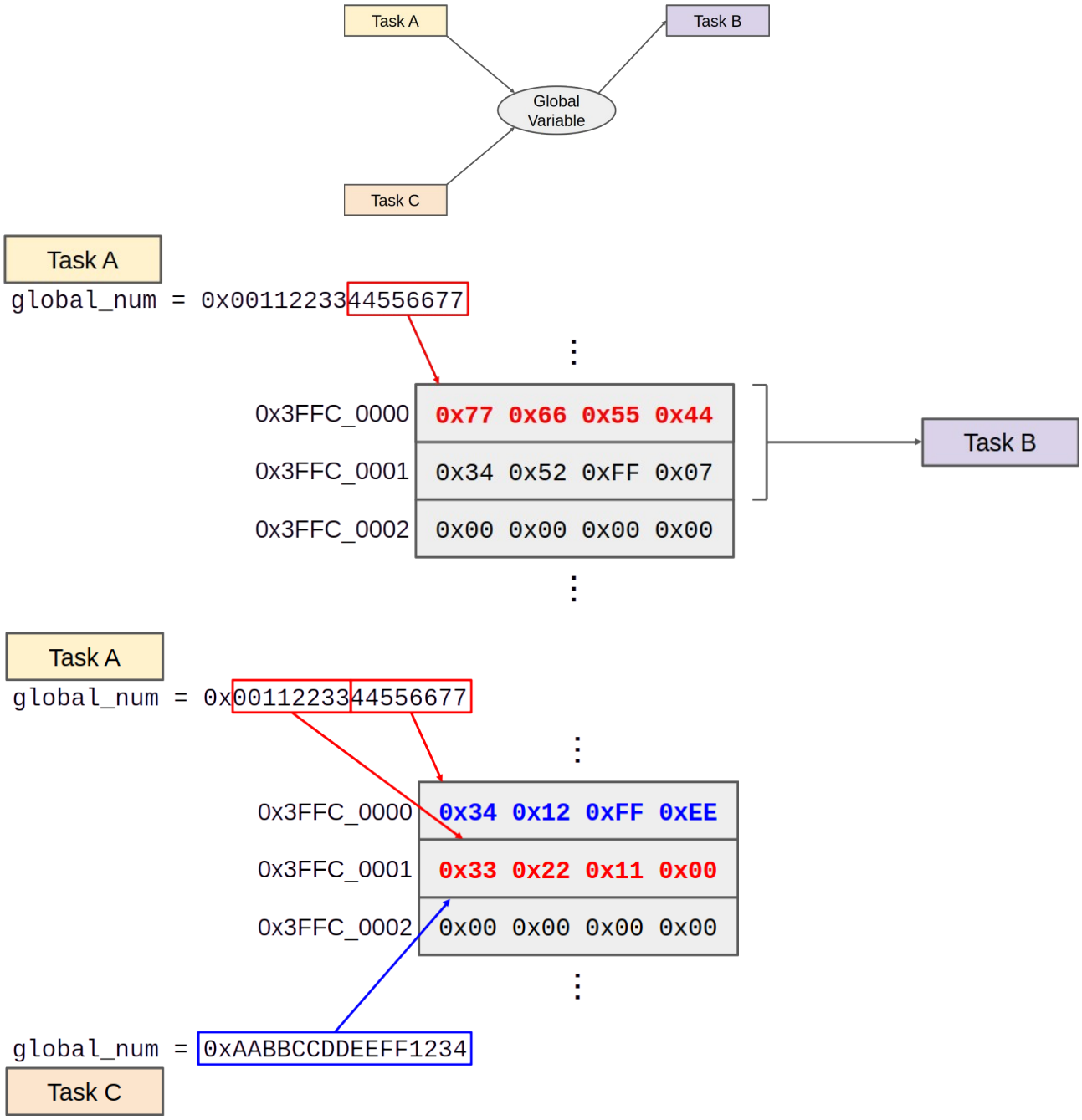
Une même fonction peut servir à la création de plusieurs tâches. Chaque tâche créée avec la même fonction sera une instance de la fonction.

Une variable définie dans la fonction d'une tâche est créée dans la pile de la tâche. Elle est donc différente pour chaque instance de la tâche car chaque tâche possède sa propre pile.
Si une variable est définie `static` dans la fonction d'une tâche alors elle est créée en dehors de la pile de la tâche. C'est une variable commune à toutes les instances.

9. Avec l'exemple 4, créez deux instances de même priorité de `maTache()` avec deux appels à `xTaskCreate` dans le `setup`.

10. Expliquez le comportement.

Partager une variable globale



Ceci est encore plus critique avec plusieurs processeurs car les « ticks » ne sont pas forcément synchrones entre processeurs.
Donc il faut des mécanismes pour gérer les ressources partagées.

Sécuriser les échanges de données

Pour sécuriser les échanges de données entre tâches, FreeRTOS propose des files (queue). Une file fonctionne comme une pile, par défaut comme une FIFO.
Pour créer une file, on donne le nombre d'éléments et la taille d'un élément et on récupère le handle de la file, un `xQueueHandle` :

```
queue = xQueueCreate(nb, sizeof(element));
```

Pour envoyer une donnée dans la file, on précise le handle de la file, la donnée et le temps d'attente si l'envoi n'est pas possible immédiatement car la file est pleine. La valeur de retour indique si l'envoi a réussi. La macro `portMAX_DELAY` permet d'attendre tant que l'envoi n'est pas possible :

```
ok = xQueueSend(queue, data, attente);
```

Pour recevoir une donnée de la file, on précise le handle de la file, un pointeur sur la donnée à remplir et le temps d'attente si la réception n'est pas possible immédiatement car la file est vide. La valeur de retour indique si la réception a réussie. La macro `portMAX_DELAY` permet d'attendre tant que la réception n'est pas possible :

```
ok = xQueueReceive(queue, &data, attente);
```

11. Testez l'exemple 5 et commentez le fonctionnement obtenu.

12. Passez la taille de la file à 10 éléments. Commentez le fonctionnement obtenu.

13. Inversez les valeurs de délai de la tâche d'envoi et de la tâche de réception et commentez le fonctionnement obtenu.

14. Mettre un délai de 2000ms dans la tâche d'envoi et de 1000ms dans la tâche de réception. Dans la tâche de réception, remplacez `portMAX_DELAY` par 0 pour que la réception soit immédiate. Commentez le fonctionnement.

Si les données à transmettre sont volumineuses, on utilise une allocation dynamique. Les données dans la file sont des pointeurs. On envoie (et on réceptionne) dans la file le pointeur vers l'espace mémoire. La tâche émettrice alloue l'espace mémoire dynamique et la tâche réceptrice libère l'espace mémoire dynamique.
Pour allouer un espace mémoire, on donne la taille de l'espace à réserver et on récupère un pointeur vers l'espace alloué :

```
pointeur = pvPortMalloc(nb * sizeof(element));
```

Pour libérer un espace mémoire alloué précédemment, il suffit de donner le pointeur :

```
vPortFree(pointeur);
```

Protéger une ressource

Pour protéger une ressource qui peut être utilisée par plusieurs tâches mais par une seule à la fois, on utilise des mutex (pour exclusion mutuelle). FreeRTOS utilise des sémaphores que nous n'avons pas encore vu, pour implémenter des mutex. Les sémaphores sont plus sophistiqués que les mutex. Une tâche va signaler qu'elle prend la ressource en prenant le mutex. Quand elle a terminé, elle rend le mutex. Si le mutex n'est pas disponible, la tâche ne peut pas utiliser la ressource. Elle doit attendre que le mutex soit disponible.
Pour créer un mutex, on récupère un handle du mutex, un `SemaphoreHandle_t` :

```
mutex = xSemaphoreCreateMutex();
```

Pour prendre un mutex, on précise le handle du mutex et le temps d'attente si le mutex n'est pas disponible. La valeur de retour indique si le mutex a été pris :

```
ok = xSemaphoreTake(mutex, attente);
```

Pour rendre un mutex, on précise seulement le handle du mutex. La valeur de retour indique si le mutex a été rendu :

```
ok = xSemaphoreGive(mutex);
```

15. Testez l'exemple 6 et commentez le fonctionnement obtenu.

16. Sans modifier les délais, utilisez un mutex pour obtenir des affichages corrects.

Exercice

Créer un projet et utiliser GitHub pour gérer les versions.
Ecrire un programme qui comporte trois tâches :

- reception
- clignotant
- affichage

La tâche « reception » peut recevoir les commandes suivantes depuis la liaison série avec le terminal de l'ordinateur :

- on suivi d'un appui sur la touche « entrée »
- off suivi d'un appui sur la touche « entrée »
- time suivi d'un espace, d'un nombre entier et d'un appui sur la touche « entrée », exemple : time 258

La tâche « reception » envoie les commandes valides à une tâche « clignotant » qui fait clignoter une led à la cadence réglée par la commande « time » de la tâche « reception ». Elle envoie également les commandes valides à la tâche d'affichage ainsi qu'un message pour indiquer une commande invalide.
La tâche « clignotant » envoie des messages à la tâche « affichage » tous les 100 clignotements.
La tâche « affichage » affiche les messages qu'on lui envoie.

S’y retrouver dans l’API de FreeRTOS.

Les types particuliers :

- `TickType_t` : type pour les mesures de temps en « ticks ». Implémentation sur ESP32 comme `uint32_t`.
- `BaseType_t` : type de données le plus efficace pour l’architecture. Implémentation sur ESP32 comme `int`. Par exemple pour les retours de fonctions avec un booléen, `pdTRUE` et `pdFALSE`.

Les principaux préfixes de variables ou de fonctions :

- v pour `void`
- c pour `char`
- s pour `short`
- l pour `long`
- x pour `TickType_t` ou `BaseType_t`
- p pour pointeur

Les noms de fonctions sont précédés d’un préfixe pour la valeur de retour et d’une indication du fichier de définition de FreeRTOS. Quelques exemples :

- `v`**`Task`**`PrioritySet()` : retourne un void et se trouve dans tasks.c
- `x`**`Task`**`Create()` : retourne un BaseType_t et se trouve dans tasks.c
- `pv`**`Timer`**`GetTimerID()` : retourne un pointeur vers un `void` et se trouve dans timers.c

Les macros sont en majuscules et préfixées en minuscule par une indication du fichier dans lequel elles se trouvent. Quelques exemples :

- `pdTRUE` : préfixe pd pour projdefs.h
- `portMAXDELAY` : préfixe port pour portmacro.h
- `taskENTER_CRITICAL()` : préfixe task pour task.h

Pour pouvoir utiliser une fonction de l’API de FreeRTOS dans une routine d’interruption, il faut que la fonction existe post-fixée par FromISR. Si elle n’existe pas, alors la fonction ne peut pas être utilisée dans une routine d’interruption. Quelques exemples :

- `xQueueSend()` devient `xQueueSendFromISR()`
- `xTaskResume()` devient `xTaskResumeFromISR()`

Pour en savoir plus sur FreeRTOS (sources de ce document) :

https://fr.wikipedia.org/wiki/FreeRTOS

https://www.youtube.com/playlist?list=PLEBQazB0HUyQ4hAPU1cJED6t3DU0h34bz

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos_idf.html

https://www.freertos.org/

Exemple 1

```
#include <Arduino.h>

void maTache(void *parametres)
{
    while (1) // boucle infinie
    {
      Serial.printf("%s %08X\n", pcTaskGetName(NULL),
                    (int)xTaskGetCurrentTaskHandle());
      delay(500);
    }
}

void setup()
{
  Serial.begin(115200);
  while (!Serial);
  Serial.printf("Départ\n");

  xTaskCreate(
    maTache, /* Fonction de la tâche. */
    "Ma tâche", /* Nom de la tâche. */
    10000, /* Taille de la pile de la tâche */
    NULL, /* Paramètres de la tâche, NULL si pas de paramètre */
    1, /* Priorité de la tâche */
    NULL); /* Pointeur pour récupérer le « handle » de la tâche, optionnel */
}

void loop()
{
  Serial.printf("%s %08X\n", pcTaskGetName(NULL),
                (int)xTaskGetCurrentTaskHandle());
  delay(1000);
}
```

Exemple 2

```
#include <Arduino.h>

void maTache1(void *parametres)
{
  int i = 0;
  while (1) // boucle infinie
  {
    Serial.printf("maTache1 %4d\n", i++);
    delay(500);
  }
}

void maTache2(void *parametres)
{
  int i = 0;
  while (1) // boucle infinie
  {
    Serial.printf("maTache2 %4d\n", i++);
    delay(2000);
  }
}

void setup()
{
  Serial.begin(115200);
  while (!Serial)
    ;
  Serial.printf("Départ\n");

  xTaskCreate(
      maTache1,     /* Fonction de la tâche. */
      "Ma tâche 1", /* Nom de la tâche. */
      10000,        /* Taille de la pile de la tâche */
      NULL,         /* Paramètres de la tâche, NULL si pas de paramètre */
      1,            /* Priorité de la tâche */
      NULL);        /* Pointeur pour récupérer le « handle » de la tâche,
optionnel */

  xTaskCreate(
      maTache2,     /* Fonction de la tâche. */
      "Ma tâche 2", /* Nom de la tâche. */
      10000,        /* Taille de la pile de la tâche */
      NULL,         /* Paramètres de la tâche, NULL si pas de paramètre */
      1,            /* Priorité de la tâche */
      NULL);        /* Pointeur pour récupérer le « handle » de la tâche,
optionnel */
}

void loop()
{
  static int i = 0;
  Serial.printf("Loop %4d\n", i++);
  delay(1000);
}
```

Exemple 3

```
#include <Arduino.h>

void tachePeriodique(void *pvParameters)
{
  TickType_t xLastWakeTime;
  double x = 0, y = 0;
  // Lecture du nombre de ticks quand la tâche débute
  xLastWakeTime = xTaskGetTickCount();
  while (1)
  {

    TickType_t debCalcul = xTaskGetTickCount();
    // Des calculs pour que la tâche occupe le processeur
    int nbTour = 3000 + rand()%3000;
    for (int i = 0; i < nbTour; i++) {
      double xn = sin(x) + cos(y);
      double yn = cos(x) + sin(y);
      double d = sqrt(xn * xn + yn * yn);
      if (d == 0) {
        x = 0;
        y = 0;
      } else {
        x = xn / d;
        y = yn / d;
      }
    }
    TickType_t finCalcul = xTaskGetTickCount();
    Serial.printf("Temps de calcul = %u\n", finCalcul - debCalcul);

    // Endort la tâche pendant le temps restant par rapport au réveil,
    // ici 200ms, donc la tâche s'effectue toutes les 200ms
    vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(200)); // toutes les 200 ms
  }
}

void setup()
{
  Serial.begin(115200);
  Serial.printf("Initialisation\n");

  // Création de la tâche périodique
  xTaskCreate(tachePeriodique, "Tâche périodique", 10000, NULL, 2, NULL);
}

void loop()
{
  static int i = 0;
  Serial.printf("Boucle principale : %d\n", i++);
  delay(1000);
}
```

Exemple 4

```
void maTache(void *parametres)
{
    int v1 = 0;
    static int v2 = 0;

    while (1) // boucle infinie
    {
      Serial.printf("%s : v1=%d v2=%d\n", pcTaskGetName(NULL), v1, v2);
      v1++;
      v2++;
      delay(500);
    }
}
```

Exemple 5

```
#include <Arduino.h>

// Handle de la queue
xQueueHandle queue;

void tacheEnvoi(void *parametres)
{
  int i = 100;
  while (1) {
    if (xQueueSend(queue, &i, portMAX_DELAY) == pdPASS) {
      Serial.printf("Envoi %d\n", i);
      i++;
    } else {
      Serial.printf("Envoi échec\n");
    }
    delay(1000);
  }
}

void tacheReception(void *parametres)
{
  int i;
  while (1) {
    if (xQueueReceive(queue, &i, portMAX_DELAY) != pdTRUE) {
      Serial.printf("Réception échec\n");
    } else {
      Serial.printf("Réception %d\n", i);
    }
    delay(2000);
  }
}

void setup()
{
  Serial.begin(115200);
  while (!Serial);
  Serial.printf("Départ\n");

  // Création de la file
  queue = xQueueCreate(5, sizeof(int));

  xTaskCreate(
      tacheEnvoi, /* Fonction de la tâche. */
      "Envoi",    /* Nom de la tâche. */
      10000,      /* Taille de la pile de la tâche */
      NULL,       /* Paramètres de la tâche, NULL si pas de paramètre */
      1,          /* Priorité de la tâche */
      NULL);  /* Pointeur pour récupérer le « handle » de la tâche, optionnel */

  xTaskCreate(
      tacheReception, /* Fonction de la tâche. */
      "Réception",    /* Nom de la tâche. */
      10000,          /* Taille de la pile de la tâche */
      NULL,           /* Paramètres de la tâche, NULL si pas de paramètre */
      1,              /* Priorité de la tâche */
      NULL);  /* Pointeur pour récupérer le « handle » de la tâche, optionnel */

  vTaskDelete(NULL);
}
```

```
void loop()
{
  // Ne s'exécute pas
}
```

Exemple 6

```
#include <Arduino.h>

void tache1(void *parametres)
{
  int i = 0;
  while (1)
  {
    Serial.printf("Dans la tâche 1 : ", i);
    delay(1);
    Serial.printf("%d\n", i);
    i++;
    delay(1000);
  }
}

void tache2(void *parametres)
{
  int i = 100;
  while (1)
  {
    Serial.printf("Dans la tâche 2 : ", i);
    delay(1);
    Serial.printf("%d\n", i);
    i++;
    delay(1000);
  }
}

void setup()
{
  Serial.begin(115200);
  while (!Serial)
    ;
  Serial.printf("Départ\n");

  xTaskCreate(
      tache1,    /* Fonction de la tâche. */
      "Tâche 1", /* Nom de la tâche. */
      10000,     /* Taille de la pile de la tâche */
      NULL,      /* Paramètres de la tâche, NULL si pas de paramètre */
      1,         /* Priorité de la tâche */
      NULL);  /* Pointeur pour récupérer le « handle » de la tâche, optionnel */

  xTaskCreate(
      tache2,    /* Fonction de la tâche. */
      "Tâche 2", /* Nom de la tâche. */
      10000,     /* Taille de la pile de la tâche */
      NULL,      /* Paramètres de la tâche, NULL si pas de paramètre */
      1,         /* Priorité de la tâche */
      NULL);  /* Pointeur pour récupérer le « handle » de la tâche, optionnel */

  vTaskDelete(NULL);
}

void loop()
{
  // Ne s'exécute pas
}
```