Lab 9

Programmation multitâche avec FreeRTOS pour les Terminaux et Passerelles LoRa

Dans le laboratoire précédent (Lab.8), nous avons étudié la programmation basse consommation avec ESP32 et le développement des terminaux **LoRa** basse consommation. Dans ce laboratoire, nous présentons un certain nombre de mécanismes de programmation nécessaires pour construire des nœuds complexes fonctionnant avec **plusieurs tâches** telles que le traitement, la capture de données (détection), l'affichage des données et les tâches de communication.
ESP32 est fourni avec le système d'exploitation **FreeRTOS** qui permet le déploiement des applications multi-tâches.

Table des matières

9.Introduction à FreeRTOS

FreeRTOS est un **système d'exploitation en temps réel** (RTOS) open source, à faible encombrement et portable. Il fonctionne en **mode préemptif**. Il est aujourd'hui l'un des systèmes d'exploitation en temps réel les plus utilisés sur le marché.

Le **thread contrôlé** par **FreeRTOS** est une **tâche (*task*)**. Le nombre de tâches exécutées simultanément et leur priorité ne sont limités que par le matériel.
La planification est basée sur des **sémaphores**, des mécanismes **mutex** et un système de mise en **file d'attente** (*queue*).

FreeRTOS fonctionne selon le modèle Round-Robin avec gestion des priorités. Conçu pour être très compact, il ne contient que quelques fichiers de langage C et n'implémente aucun pilote matériel.

Les domaines d'application sont assez larges, car les principaux avantages de **FreeRTOS** sont l'exécution en temps réel, le code open source et la très petite taille. Il est donc principalement utilisé pour les systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour les systèmes de traitement vidéo et les applications réseau qui ont des contraintes "temps réel".
Parmi les applications les plus récentes fonctionnant avec des contraintes en temps réel figurent évidemment les **architectures IoT**.

9.1.1 Ordonnanceur de tâches

L'objectif principal du planificateur (ordonnanceur) de tâches est de décider quelles tâches sont à l'état prêt à être exécutées.
Pour faire ce choix, l'ordonnanceur **FreeRTOS** se base uniquement sur la **priorité des tâches**. Les tâches dans FreeRTOS se voient attribuer à leur création un niveau de priorité représenté par un entier.
Le niveau le plus bas est zéro et doit être strictement réservé à la tâche **`Idle`**.
Plusieurs tâches peuvent appartenir au même niveau de priorité. Dans FreeRTOS, il n'y a pas de mécanisme de gestion automatique des priorités (comme pour le système Linux). La priorité d'une tâche ne peut être modifiée qu'à la demande explicite du développeur.
Les tâches sont des fonctions simples qui s'exécutent généralement dans une boucle infinie (**`while(1)`**).

Dans le cas d'un microcontrôleur n'ayant qu'un seul cœur (ESP32 a deux cœurs), il n'y aura à tout moment qu'une seule tâche en cours d'exécution (***run***). Le planificateur s'assurera toujours que la tâche de priorité la plus élevée pouvant être exécutée soit sélectionnée pour entrer dans l'état d'exécution.
Si deux tâches partagent le même niveau de priorité et que les deux peuvent s'exécuter, les deux tâches s'**exécuteront en alternance** en ce qui concerne les réveils du planificateur (***ticks***).
Par défaut, dans ESP32, le planificateur fonctionne sur un noyau (noyau 0) et toutes les autres tâches sont exécutées sur le noyau 1.

Afin de choisir la tâche à exécuter, le planificateur doit lui-même exécuter et préempter la tâche en état d'exécution. Afin d'assurer le réveil de l'ordonnanceur, FreeRTOS définit une interruption périodique appelée "**tick interruption**".
Cette interruption s'exécute indéfiniment à une certaine fréquence qui est définie dans le fichier **`FreeRTOSConfig.h`** par la constante :

```
configTICK_RATE_HZ   // tick frequency interrupt" in Hz
```

FreeRTOS utilise la **planification préemptive** pour gérer les tâches. Cependant, il peut aussi éventuellement utiliser (si la directive lui est donnée) un **ordonnancement coopératif**. Dans ce mode d'ordonnancement, un changement de contexte d'exécution n'a lieu que si la tâche en cours d'exécution **autorise explicitement** l'exécution d'une autre tâche (en appelant un **`yield()`** par exemple) ou en entrant dans un état de blocage.

Les tâches ne sont donc jamais préemptées. Cette méthode de planification simplifie grandement la gestion des tâches, malheureusement elle peut conduire à un système moins efficace et moins sécurisé.

9.1.1.1 Famine (*starvation*)

Dans **FreeRTOS**, la tâche la plus prioritaire prête à être exécutée sera toujours sélectionnée par le planificateur. Cela peut conduire à une situation de **famine**. En effet, si la tâche de priorité supérieure n'est jamais interrompue, toutes les tâches de priorité inférieure ne seront jamais exécutées.

FreeRTOS n'implémente aucun mécanisme automatique pour empêcher le phénomène de famine.
Le développeur doit s'assurer qu'aucune tâche ne monopolise tout le temps d'exécution du microcontrôleur.

Il peut le faire en définissant des **événements** qui interrompront la tâche de priorité supérieure pendant une durée spécifiée ou jusqu'à ce qu'un autre événement se produise, laissant le champ libre pour les tâches de priorité inférieure à exécuter.

Pour éviter la famine, le développeur peut utiliser une **planification de taux monotone**. Il s'agit d'une technique d'attribution de priorité qui donne à chaque tâche une **priorité unique en fonction de sa fréquence d'exécution**.
La priorité la plus élevée est attribuée à la tâche avec la fréquence d'exécution la plus élevée et la priorité la plus faible est accordée à la tâche avec la fréquence la plus basse. L'ordonnancement à des cadences

monotones permet d'optimiser l'ordonnancement des tâches, mais cela reste difficile à réaliser en raison de la nature des tâches qui ne sont pas complètement périodiques.

9.1.1.2 La tâche inactive (`Idle`)

Un microcontrôleur doit toujours avoir quelque chose à faire. En d'autres termes, il doit toujours y avoir une tâche en cours d'exécution. FreeRTOS gère cette situation en définissant la tâche **`Idle`** qui est créée au démarrage du planificateur (par exemple la tâche de la fonction **`loop()`**). La priorité du noyau la plus basse est attribuée à cette tâche.
Malgré cela, la tâche **`Idle`** peut avoir plusieurs fonctions à exécuter, notamment :

- libérer l'espace mémoire occupé par une tâche supprimée
- mettre le microcontrôleur en veille afin d'économiser l'énergie du système lorsque aucune tâche d'application n'est en cours d'exécution.
- mesurer le taux d'utilisation du processeur
-

9.1.2 Évolution et statut des tâches

Les tâches dans **FreeRTOS** peuvent exister dans 5 états: supprimées, suspendues, prêtes, bloquées ou en cours d'exécution (***deleted, suspended, ready, blocked , in execution***).

Dans **FreeRTOS**, il n'y a pas de variable pour spécifier explicitement l'état d'une tâche, en retour FreeRTOS utilise des **listes d'états** - **TCB**.
La présence de la tâche dans un t**ype de liste** de statuts détermine son statut (prêt, bloqué ou suspendu).
Lorsque les tâches changent d'état, le planificateur déplace la tâche (l'élément **`xListItem`** appartenant à cette même tâche) d'une liste d'états à une autre.

Lors de la création d'une tâche, **FreeRTOS** crée et remplit le **TCB** correspondant à la tâche, puis il insère directement la tâche dans une **liste prête** (***Ready List***) qui est la liste contenant une référence à toutes les tâches à l'état prêt.

FreeRTOS maintient plusieurs listes prêtes - **une liste pour chaque niveau de priorité**. Lors du choix de la prochaine tâche à exécuter, le planificateur analyse les listes prêtes de la priorité la plus élevée à la plus basse.

Plutôt que de définir explicitement un état en cours d'exécution ou une liste qui lui est associée, le noyau **FreeRTOS** décrit une variable **`pxCurrentTCB`** qui identifie le processus en cours d'exécution. Cette variable pointe vers le **TCB** correspondant au processus trouvé dans l'une des **Ready List**.

Une tâche peut se retrouver dans l'**état bloqué** lors de l'accès à une file d'attente de lecture/écriture si la file d'attente est vide/pleine.
Chaque opération d'accès à la file d'attente est configurée avec un délai d'expiration (**`xTicksToWait`**).
Si ce délai est égal à 0, la tâche n'est pas bloquée et l'opération d'accès à la file d'attente est considérée comme ayant échoué. Si le **`timeout`** n'est pas nul, la tâche passe à l'état bloqué jusqu'à ce qu'il y ait une modification de la file d'attente (par une autre tâche par exemple).

Une fois que l'opération d'accès à la file d'attente est possible, la tâche vérifie que son délai d'expiration n'a pas expiré et termine avec succès son opération.

Dans la figure 9.1, nous voyons le **diagramme d'état d'une tâche**. Une tâche peut être intentionnellement placée dans l'état suspendu, puis elle sera complètement ignorée par le planificateur et ne consommera plus de ressources jusqu'à ce qu'elle soit supprimée de cet état et retournée à un **état prêt**.

Le dernier état qu'une tâche peut prendre est l'**état supprim**é, cet état est nécessaire car une tâche supprimée ne libère pas ses ressources instantanément.
Une fois dans l'état supprimé, la tâche est ignorée par le planificateur et une autre tâche nommée **`Idle`** est chargée de libérer les ressources allouées par les tâches étant à l'état supprimé.

La tâche inactive (**`Idle)`** est créée au démarrage du planificateur et reçoit la priorité la plus faible possible, ce qui entraîne une libération retardée des ressources lorsque aucune autre tâche n'est en cours d'exécution.

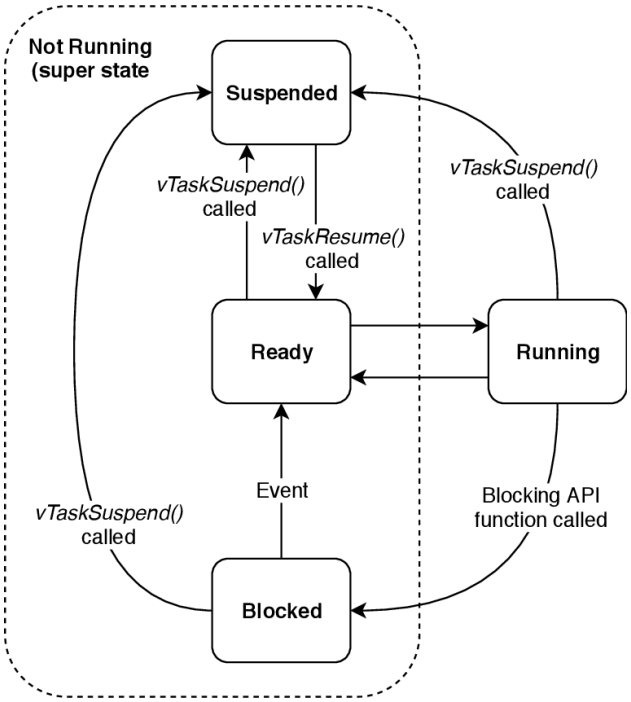


Figure 9.1 Diagramme d'état d'une tâche

9.2 FreeRTOS et programmation en temps réel pour l'IoT sur ESP32

Dans ce laboratoire, nous étudierons comment utiliser **FreeRTOS** fonctionnant sur le **ESP32**. Arduino ESP32 est construit sur FreeRTOS et en fait le programme principal est mis dans une tâche en boucle (**`loop()`**).

Mais si nous n'utilisons qu'une seule tâche (même en utilisant **Finite State Machine** pour une application Arduino pure), nous ne profiterons pas pleinement de **FreeRTOS** en **mode multitâche**.

FreeRTOS nous permet de déployer plusieurs tâches travaillant en «parallèle». Ces tâches sont gérées par le planificateur RTOS.

Sur l'ESP32, les tâches peuvent être déployées sur **2 cœur**s ce qui, cela dans le **contexte IoT**, donne la possibilité d'exécuter les tâches associées aux capteurs/actionneurs en parallèle avec les **tâches de communication sur les réseaux WiFi, LoRa, GSM, ..**.

Notre courte présentation de **FreeRTOS pour ESP32** est organisée en 4 parties:

1. création et suppression de tâches
2. mécanismes de communication entre les tâches (variables globales et files d'attente)
3. mécanismes de synchronisation (sémaphores, interruptions, ..)
4. déploiement de tâches sur une architecture dual-core

Après cette présentation, nous allons développer/expérimenter avec les nœuds LoRa: Terminaux et Passerelle exécutant plusieurs tâches.

9.2.1 Créer et supprimer une tâche

Nous créons une tâche avec un appel à la fonction **`xTaskCreate()`**.

Les arguments de cette fonction sont les suivants:

- **`TaskCode`**: dans cet argument, nous devons passer un **pointeur vers la fonction** qui implémentera la tâche.
- **`TaskName`**: le nom de la tâche, dans une chaîne. Par exemple "**`Sensor_Task`**".
- **`StackDepth`**: la taille de la pile de tâches, spécifiée comme le **nombre de variables** qu'elle peut contenir (pas le nombre d'octets). Dans cet exemple simple, nous passerons une valeur assez grande (**`1000`**).
- **`Parameter`**: pointeur vers un paramètre que la fonction de tâche peut recevoir. Il doit être de type (**`void *`**). Dans ce cas, pour simplifier le code, nous ne l'utiliserons pas, nous passons donc **`NULL`** dans l'argument.
- **`Priority`**: priorité de la tâche. Nous allons créer la tâche avec la priorité 1.
- **`TaskHandle`**: renvoie un descripteur qui peut être utilisé pour la dernière référence de la tâche lors des appels de fonction (par exemple, pour supprimer une tâche ou modifier sa priorité). Pour cet exemple simple, nous n'allons pas l'utiliser, donc ce sera **`NULL`**.

La fonction **`xTaskCreate()`** renvoie **`pdPASS`** en cas de succès ou un code d'erreur. Pour l'instant, nous supposerons que la tâche sera créée sans aucun problème, nous n'allons donc pas faire de vérification d'erreur. Bien sûr, pour une application plus réelle, nous aurions besoin de faire ce test pour confirmer que la tâche est bien créée.

Pour supprimer une tâche de son propre code, il vous suffit d'appeler la fonction **`vTaskDelete`**.

```
vTaskDelete (NULL);
```

9.2.2 Création et exécution d'une simple tâche supplémentaire

Dans le premier exemple, à côté de la tâche du fond (**`loop() Task`**), nous ajouterons une tâche supplémentaire à notre application.
Notre application a 2 tâches:
La tâche **`loop()`** imprimera le texte "**`this is main loop Task`**" et la deuxième tâche imprimera "c**`this is an extra task`**" sur le terminal série.

```
void setup() {
  Serial.begin(9600);
  /* we create a new task here */
  xTaskCreate(
      additionalTask,           /* Task function. */
      "additional Task",        /* name of task. */
      10000,                    /* Stack size of task */
      NULL,                     /* parameter of the task */
      1,                        /* priority of the task */
      NULL);                    /* Task handle to keep track of created task */
}
/* the forever loop() function is invoked by ESP32 loopTask */
void loop() {
  Serial.println("this is main loop Task");
  delay(1000);
}
/* this function will be invoked when additionalTask was created */
void additionalTask( void * parameter )
{
  /* loop forever */
  for(;;){
    Serial.println("this is additional Task");
    delay(1000);
  }
  /* delete a task when finish,
  this will never happen because this is infinity loop */
  vTaskDelete( NULL );
}
```

Le résultat affiché :

```
this is this is main loop Task
this is additional Task
this is this is main loop Task
this is additional Task
this is this is main loop Task
this is additional Task
```

9.2.3 Création et exécution de deux tâches

Dans cet exemple, nous allons créer deux tâches avec la même priorité d'exécution (1). La tâche principale (**`loop()`**) reste inactive.

```
void taskOne( void * parameter )
{
    for( int i = 0;i<10;i++ ){
        Serial.println("Hello from task 1");
        delay(1000);
    }
    Serial.println("Ending task 1");
    vTaskDelete( NULL );
}

void taskTwo( void * parameter)
{
    for( int i = 0;i<10;i++ ){
        Serial.println("Hello from task 2");
        delay(1000);
    }
    Serial.println("Ending task 2");
    vTaskDelete( NULL );
}
```

```
void setup() {
  Serial.begin(9600);
  delay(1000);
  xTaskCreate(
                    taskOne,          /* Task function. */
                    "TaskOne",        /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,             /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */

xTaskCreate(
                    taskTwo,          /* Task function. */
                    "TaskTwo",        /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,             /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */
}

void loop() {
  delay(1000);
}
```

Résultat de l'exécution :

```
Hello from task 1
Hello from task 2
Hello from task 1
Hello from task 2
Hello from task 2
Hello from task 1
Hello from task 1
Hello from task 2
```

9.3 Communication entre deux tâches

Il existe plusieurs façons de faire communiquer les tâches entre elles (**variables globales**, **files d'attente**, ..)

9.3.1 Variables globales comme arguments

Pour commencer, nous passerons un seul paramètre à partir d'une variable globale.
Les fonctions de tâche reçoivent un paramètre générique **`(void *)`**. Dans notre code, nous l'interpréterons comme un pointeur vers **`int`**, qui correspond à **`(int *)`**.
Donc, la première chose que nous faisons est de convertir (***cast***) en **`(int *)`**.

```
(int *) parameter;
```

Nous avons maintenant un pointeur vers la position mémoire d'un entier. Cependant, nous voulons accéder au contenu actuel de la position mémoire. Nous voulons donc la valeur de la position mémoire vers laquelle pointe notre pointeur. Pour ce faire, nous utilisons l'opérateur de **déférence**, qui est **`*`**.
Donc, ce que nous devons faire est d'utiliser l'opérateur de déférence sur notre pointeur converti et nous devrions pouvoir **accéder à sa valeur**.

```
*((int *)parameter);
```

Une fois que nous y avons accès, nous pouvons simplement l'imprimer en utilisant la fonction **`Serial.println`**, ce que nous ferons dans les deux fonctions.
Le code source complet peut être vu ci-dessous, ainsi que l'implémentation des deux fonctions.
Notez qu'à des fins de débogage, nous imprimons différentes chaînes dans les fonctions.

```
int globalIntVar = 5;
int localIntVar = 9;

void setup()
{
  Serial.begin(9600);
  delay(1000);
  xTaskCreate(
            globalIntTask,             /* Task function. */
            "globalIntTask",           /* String with name of task. */
            10000,                     /* Stack size in words. */
            (void*)&globalIntVar,      /* Parameter passed as input of the task */
            1,                         /* Priority of the task. */
            NULL);                     /* Task handle. */
  xTaskCreate(
            localIntTask,              /* Task function. */
            "localIntTask",            /* String with name of task. */
            10000,                     /* Stack size in words. */
            (void*)&localIntVar,       /* Parameter passed as input of the task */
            1,                         /* Priority of the task. */
            NULL);                     /* Task handle. */
}

void globalIntTask(void *parameter )
{
    Serial.print("globalIntTask: ");
    Serial.println(*((int*)parameter));
    vTaskDelete( NULL );
}

void localIntTask(void *parameter )
{
    Serial.print("localIntTask: ");
    Serial.println(*((int*)parameter));
    vTaskDelete( NULL );
}

void loop() {
  delay(1000);
}
```

Résultat de l'exécution :

```
lobalIntTask: 5
localIntTask: 9
```

9.3.2 Files d'attente (*queues*)

Les files d'attente sont très utiles pour la communication inter-tâches, permettant d'envoyer des messages d'une tâche à une autre. Elles sont généralement organisées en **FIFO** (***First In First Out***), ce qui signifie que de nouvelles données sont insérées à l'arrière de la file d'attente et consommées à l'avant.

Les données placées dans la file d'attente sont copiées plutôt que référencées. Cela signifie que si nous envoyons un entier à la file d'attente, sa valeur sera effectivement copiée et si nous modifions la valeur d'origine par la suite, aucun problème ne devrait se produire.
Un aspect comportemental important est que l'insertion dans une file d'attente complète ou la consommation à partir d'une file d'attente vide peut bloquer les appels pendant une durée donnée (cette **durée est un paramètre** de l'API).

Bien que les files d'attente mentionnées soient généralement utilisées pour la communication entre les tâches, pour cet exemple d'introduction, nous allons insérer et consommer des éléments dans la file d'attente intégrée à la fonction **`loop()`**.

```
QueueHandle_t queue;

void setup() {
  Serial.begin(9600);
  queue = xQueueCreate( 10, sizeof( int ) );
  if(queue == NULL){ Serial.println("Error creating the queue"); }
}
 void loop() {
  if(queue == NULL)return;
  for(int i = 0; i<10; i++){ xQueueSend(queue, &i, portMAX_DELAY); }

  int element;
  for(int i = 0; i<10; i++){
    xQueueReceive(queue, &element, portMAX_DELAY);
    Serial.print(element);
    Serial.print("|");
  }
  Serial.println();
  delay(1000);
}
```

Ci-dessous, le même type de programme est précédé de la création de deux tâches **`ProducerTask`** et **`ConsumerTask`**.

```
QueueHandle_t queue;
int queueSize = 10;

void producerTask( void * parameter )
{
    for( int i = 0;i<queueSize;i++ ){
      xQueueSend(queue, &i, portMAX_DELAY);
    }
    vTaskDelete( NULL );
}

void consumerTask( void * parameter)
{
    int element;
    for( int i = 0;i < queueSize;i++ ){
        xQueueReceive(queue, &element, portMAX_DELAY);
        Serial.print(element);
        Serial.print("|");
    }
    vTaskDelete( NULL );
}

void setup() {
  Serial.begin(9600);
  delay(2000);

  queue = xQueueCreate( queueSize, sizeof( int ) );
  if(queue == NULL){
    Serial.println("Error creating the queue");
  }
```

```
  xTaskCreate(
                    producerTask,     /* Task function. */
                    "Producer",       /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,           /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */

  xTaskCreate(
                    consumerTask,     /* Task function. */
                    "Consumer",       /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,           /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */
}

void loop() {
  Serial.println("in the loop");
  delay(6000);
}
```

9.3.3 A faire

Expérimentez avec l'exemple ci-dessus avec différentes tailles de file d'attente (**128, 1000, ..**) et avec différents types de données (**`char`**, **`float`**, **`double`**, ..) et même des **structures**.

9.4 Une application IoT avec deux tâches et la communication par files d'attente

Dans l'exemple suivant, nous utiliserons un capteur de température/humidité **SHT21**.

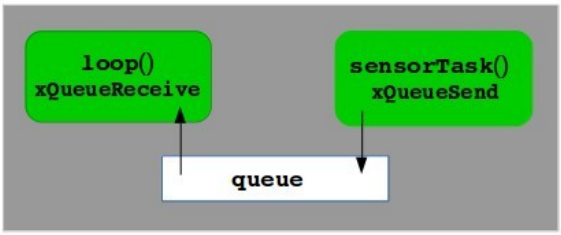


Figure 9.2 `loop()`, `sensorTask` et une file d'attente - `queue`

```
#include <Wire.h>
#include "SHT21.h"
SHT21 SHT21;

QueueHandle_t queue;
int queueSize = 128;

void sensorTask( void * pvParameters ){
float t,h;
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  t=(float) SHT21.getTemperature();
  h=(float) SHT21.getHumidity();
  xQueueSend(queue, &t, portMAX_DELAY);
  xQueueSend(queue, &h, portMAX_DELAY);
  delay(1000);
  }
}

void setup()
{
  Serial.begin(9600);
  delay(1000);
  queue = xQueueCreate( queueSize, sizeof( int ) );
  Serial.println("SHT21 test");

  xTaskCreate(
                    sensorTask,   /* Function to implement the task */
                    "sensorTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL     );  /* Task handle */

  Serial.println("Task created...");
}

void loop() {
   float t,h;
   Serial.println("Starting main loop...");
   while (true){
    xQueueReceive(queue, &t, portMAX_DELAY);
    xQueueReceive(queue, &h, portMAX_DELAY);
    Serial.print("temp:");Serial.println(t);
    Serial.print("humi:");Serial.println(h);
    delay(1000);
    }
}
```

Résultat de l'exécution :

```
Task created...
Starting main loop...
temp:24.11
humi:26.25
temp:24.10
humi:26.53
temp:24.13
humi:26.81
temp:24.13
humi:27.23
```

9.4.1 A faire

1. Testez le même programme avec un capteur différent **`BH1750`** par exemple.
2. Remplacer les variables du capteur par une structure :
```
struct
{
float t;
float h;
} sens;
```
3. Ajouter une deuxième tâche et une deuxième fille pour le deuxième capteur

9.5 Sémaphores

Les sémaphores sont utilisés pour créer des schémas de synchronisation et de protection pour les sections partagées (critiques) du code.
Dans ce paragraphe, nous présenterons plusieurs types de sémaphores:

- **sémaphore binaire**
- **mutex**
- **sémaphore - compteur**

9.5.1 Sémaphore binaire

C'est le moyen le plus simple de contrôler l'accès aux ressources qui n'ont que 2 états: verrouillé/déverrouillé ou indisponible/disponible.
La tâche qui veut accéder à la ressource appellera **`xSemaphoreTake()`**.
Il y a 2 cas:

1. S'il réussit à accéder à la ressource, il la conservera jusqu'à ce qu'elle appelle **`xSemaphoreGive()`** pour libérer la ressource afin que d'autres tâches puissent la récupérer.
2. En cas d'échec, il attendra que la ressource soit libérée par une autre tâche.

Les sémaphores binaires peuvent être appliqués pour interrompre (terminer) le traitement (**Interrupt Sub-Routine ISR**) où la fonction type **ISR** appellera **`xSemaphoreGiveFromISR()`** pour déléguer le traitement d'interruption à la tâche qui attend sur **`xSemaphoreTake()`**.

Lorsque **`xSemaphoreTake()`** est appelé, la tâche sera bloquée et attendra un événement d'interruption.

Remarque :

Les fonctions API appelées depuis l'ISR doivent avoir le préfixe "**`FromISR`**" (**`xSemaphoreGiveFromISR`**).
Elles sont conçus pour les fonctions de l'API ***Interrupt Safe***.

9.5.1.1 Exemple de sémaphore binaire avec `xSemaphoreGiveFromISR`

Nous réutiliserons l'exemple de code suivant en utilisant le style **FreeRTOS** et le sémaphore binaire pour traiter l'interruption.

```
byte ledPin = 22;
byte interruptPin = 0;  // touch pin
/* hold the state of LED when toggling */
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  /* set the interrupt pin as input pullup*/
  pinMode(interruptPin, INPUT_PULLUP);
  /* attach interrupt to the pin
  function blink will be invoked when interrupt occurs
  interrupt occurs whenever the pin change value */
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
}

/* interrupt function toggle the LED */
void blink() {
  state = !state;
  digitalWrite(ledPin, state);
}
```

Le code avec une interruption de traitement de tâche à haute priorité et appelé-activé depuis la **fonction ISR** par:

```
xSemaphoreGiveFromISR(xBinarySemaphore,&xHigherPriorityTaskWoken );
```

9.5.1.2 Code complet

```
byte ledPin = 22;
byte interruptPin = 2;
volatile byte state = LOW;
SemaphoreHandle_t xBinarySemaphore;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  /* set the interrupt pin as input pullup*/
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), ISRcallback, CHANGE);
  /* initialize binary semaphore */
  xBinarySemaphore = xSemaphoreCreateBinary();
  /* this task will process the interrupt event
  which is forwarded by interrupt callback function */
  xTaskCreate(
    ISRprocessing,           /* Task function. */
    "ISRprocessing",         /* name of task. */
    1000,                    /* Stack size of task */
    NULL,                    /* parameter of the task */
    4,                       /* priority of the task */
    NULL);
}

void loop() {}
/* interrupt function callback */
void ISRcallback() {
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  /* un-block the interrupt processing task now */
  xSemaphoreGiveFromISR(xBinarySemaphore,&xHigherPriorityTaskWoken );
}

/* this function will be invoked when additionalTask was created */
void ISRprocessing( void * parameter )
{
  Serial.println((char *)parameter);
  /* loop forever */
  for(;;){
    /* task move to Block state to wait for interrupt event */
    xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
    Serial.println("ISRprocessing is running");
    /* toggle the LED now */
    state = !state;
    digitalWrite(ledPin, state);
  }
  vTaskDelete( NULL );
}
```

9.5.2 Mutex

Un sémaphore de type **mutex** est comme **une clé** associée à la ressource. La tâche contient la clé, verrouille la ressource, la traite, puis déverrouille et renvoie la clé pour d'autres tâches à utiliser.
Ce mécanisme est similaire au sémaphore binaire sauf que la tâche qui prend la clé **doit libérer la clé (mutex)**.
Supposons que nous ayons 2 tâches: une tâche à faible priorité et une tâche à haute priorité.
Ces tâches attendent la clé et la tâche à faible priorité a une chance de capturer la clé, puis bloque la tâche à haute priorité et continue son exécution.

9.5.2.1 Exemple de mutex et deux tâches (de priorité basse et haute)

Dans cet exemple, nous créons 2 tâches: une tâche à faible priorité et une tâche à haute priorité. La tâche à faible priorité conservera la clé et bloquera la tâche à haute priorité.

```
SemaphoreHandle_t xMutex;

void setup() {
  Serial.begin(9600);
  /* create Mutex */
  xMutex = xSemaphoreCreateMutex();
  xTaskCreate(
      lowPriorityTask,           /* Task function. */
      "lowPriorityTask",        /* name of task. */
      1000,                    /* Stack size of task */
      NULL,                     /* parameter of the task */
      1,                        /* priority of the task */
      NULL);                    /* Task handle to keep track of created task */
  delay(500);
  /* let lowPriorityTask run first then create highPriorityTask */
  xTaskCreate(
      highPriorityTask,           /* Task function. */
      "highPriorityTask",        /* name of task. */
      1000,                    /* Stack size of task */
      NULL,                     /* parameter of the task */
      4,                        /* priority of the task */
      NULL);                    /* Task handle to keep track of created task */
}

void loop() {
}

void lowPriorityTask( void * parameter )
{
  Serial.println((char *)parameter);
  for(;;){
    Serial.println("lowPriorityTask gains key");
    xSemaphoreTake( xMutex, portMAX_DELAY );
    /* even low priority task delay high priority
    still in Block state */
    delay(4000);
    Serial.println("lowPriorityTask releases key");
    xSemaphoreGive( xMutex );
  }
  vTaskDelete( NULL );
}
void highPriorityTask( void * parameter )
{
  Serial.println((char *)parameter);
  for(;;){
    Serial.println("highPriorityTask gains key");
    /* highPriorityTask wait until lowPriorityTask release key */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    Serial.println("highPriorityTask is running");
    Serial.println("highPriorityTask releases key");
    xSemaphoreGive( xMutex );
    /* delay so that lowPriorityTask has chance to run */
    delay(2000);
  }
  vTaskDelete( NULL );
}
```

9.6 Tâches FreeRTOS sur un processeur multicœur (ESP32)

L'utilisation d'un processeur avec des tâches indépendantes pour les applications IoT permet de séparer les opérations de capture (détection) et de communication (WiFi, LoRa, ..). Pour aller plus loin nous pouvons implémenter ces fonctionnalités sur les tâches exécutées sur **plusieurs processeurs**.
Dans cette section, nous montrerons comment introduire la deuxième unité d'exécution et ensuite comment partager des tâches sur ces deux processeurs.

9.6. Créer une tâche épinglée sur un CPU

Tout d'abord, nous allons déclarer une variable globale qui contiendra le numéro du CPU où la tâche **FreeRTOS** que nous allons lancer sera épinglée. Cela garantit que nous pouvons facilement le modifier tout en testant le code.
Notez que nous allons exécuter le code deux fois, en attribuant les valeurs **`0`** et **`1`** à cette variable.

```
static int taskCore = 0;
```

Au début, nous afficherons le numéro du noyau que nous testons ; il est stocké dans la variable globale précédemment déclarée.
Ensuite, nous lancerons la tâche **FreeRTOS**, en l'attribuant à un processeur spécifique de l'ESP32. Nous utiliserons la fonction **`xTaskCreatePinnedToCore`**. Cette fonction prend exactement les mêmes arguments que **`xTaskCreate`** et un **argument supplémentaire** à la fin pour spécifier le processeur sur lequel la tâche doit s'exécuter.

Nous allons implémenter la tâche dans une fonction appelée **`coreTask`**. Nous devrions donner à la tâche la priorité **`0`**, pour qu'elle soit inférieure par rapport aux - **`setup()`** et main loop - **`loop()`**.
Nous n'avons pas besoin de nous soucier de passer des paramètres d'entrée ou de stocker le descripteur de tâche.
Dans **`loop()`**, nous commençons par imprimer un message sur le port série indiquant que nous lançons la boucle principale
Nous programmons une boucle **`while(1)`** infinie, sans code à l'intérieur. Il est essentiel que nous ne mettions aucun type de fonction d'exécution ou de retard à l'intérieur. La meilleure approche consiste à laisser ce champ vide.
La fonction de la tâche est également très simple. Nous allons simplement imprimer un message indiquant le CPU qui lui est attribué. Il est obtenu avec **`xPortGetCoreID`**. Bien entendu, cela doit correspondre au processeur spécifié dans la variable globale.

9.6.1.1 Code avec tâche épinglée

```
static int taskCore = 0;

void coreTask( void * pvParameters ){
    Serial.println("task running on core: ");
    while(true){
        Serial.print(xPortGetCoreID());
        delay(1000);
    }
}

void setup() {
  Serial.begin(9600);
  delay(1000);
  Serial.print("Starting to create task on core ");
  Serial.println(taskCore);

  xTaskCreatePinnedToCore(
                    coreTask,   /* Function to implement the task */
                    "coreTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
}
```

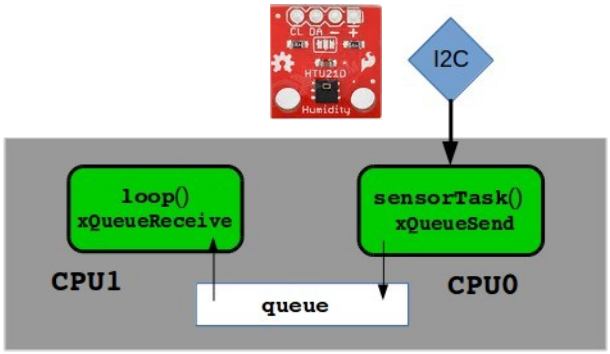
```
void loop() {
  Serial.println("Starting main loop...");
  while (true)
    {
      Serial.print(xPortGetCoreID());
      delay(3000);
     }  // with no delay - CPU is occupied 100%
}
```

Impression du programme :

```
Task created...
Starting main loop...
1task running on core:
00010001000100010001000100
```

9.6.2 Application IoT simple fonctionnant sur 2 cœurs

Dans l'exemple suivant, nous utiliserons 2 cœurs d'exécution, l'un pour lire les données du capteur et l'autre pour afficher ces données sur le terminal.
La communication entre les tâches se fait via une file d'attente.



The code:

```
#include <Wire.h>
#include "SHT21.h"
SHT21 SHT21;
QueueHandle_t queue;
int queueSize = 128;
static int taskCore = 0;

void sensorTask( void * pvParameters ){
float t,h;
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  Serial.printf("In sensor task:%d\n",xPortGetCoreID());
  t=(float) SHT21.getTemperature();
  h=(float) SHT21.getHumidity();
  xQueueSend(queue, &t, portMAX_DELAY);
  xQueueSend(queue, &h, portMAX_DELAY);
  delay(1000);
  }
}

void setup()
{
  Serial.begin(9600);
  delay(1000);
  queue = xQueueCreate( queueSize, sizeof( int ) );
  Serial.println("SHT21 test");
```

```
xTaskCreatePinnedToCore(
                    sensorTask,   /* Function to implement the task */
                    "sensorTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,  /* Task handle */
                    taskCore);  /* Core where the task should run */

  Serial.println("Task created...");
}

void loop() {
   float t,h;
   Serial.println("Starting main loop...");
   while (true){

    xQueueReceive(queue, &t, portMAX_DELAY);
    xQueueReceive(queue, &h, portMAX_DELAY);
    Serial.printf("In main task:%d\n",xPortGetCoreID());
    Serial.print("temp:");Serial.println(t);
    Serial.print("humi:");Serial.println(h);
    delay(1000);
        }
}
```

Résultat d'exécution :

```
SHT21 test
Task created...
Starting main loop...
In sensor task:0
In main task:1
temp:28.09
humi:19.76
In sensor task:0
In main task:1
temp:28.09
humi:19.73
In sensor task:0
In main task:1
temp:28.11
humi:19.69
..
```

9.6.3 A faire

1. Testez les exemples de code présentés
2. Ajoutez une **tâche d'affichage** pour afficher les données reçues dans la file d'attente.
   - Utilisez un écran OLED connecté via le bus I2C, puis
   - Écran IPS connecté via le bus SPI.

9.7 Création de nœuds de terminal et de passerelle LoRa avec plusieurs tâches

Dans cette section, nous continuons le développement des nœuds LoRa introduits dans Lab.7 , y compris les nœuds type Terminal et Gateway.
Commençons par compléter les opérations d'un terminal en ajoutant la **tâche de capteur** (**`sensor_Task`**) - pour capturer les données sur le capteur température/humidité (**SHT21**).

9.7.1 Nœud Terminal avec tâche de capteur

Rappelons le code du terminal qui peut envoyer un certain nombre de données de capteur dans le paquet de **`LoRa_send()`**. Le code ci-dessous n'utilise pas **deep_sleep** afin de permettre à la tâche de capteur de fonctionner à tout moment.
Notez que la tâche de capteur peut capturer les données plusieurs fois avant leur émission dans la trame LoRa. Dans ce cas, la valeur des données capturées peut être lissée et la valeur résultante peut être présentée comme une moyenne des lectures sur le capteur donné.
La communication entre le **`sensor_Task ()`** et la tâche principale dans la boucle **`loop()`** se fait via une file d'attente (**`queue`**).
Dans la fonction **`setup()`**, nous activons la file d'attente et nous commençons à créer la **`sensorTask`** qui exécute la fonction **`sensor_Task()`**.

```
QueueHandle_t queue;
int queueSize = 32;
static int taskCore = 0;

queue = xQueueCreate( queueSize, sizeof(float)*8);
xTaskCreatePinnedToCore(
                    sensor_Task,   /* Function to implement the task */
                    "sensor_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
Serial.println("Task created...");
```

La fonction **`sensor_Task()`** lit les données du capteur et les envoie dans la file d'attente. Le cycle d'exécution de la tâche capteur est quatre fois plus rapide (cycle * 1000/4) que le cycle de la tâche principale dans la **`loop()`**. Cela permet comme toujours de fournir les données. La file d'attente est réinitialisée à chaque fois qu'une nouvelle valeur va être envoyée. Cela empêche la file d'attente de déborder.

```
void sensor_Task( void * pvParameters )
{
float stab[4];
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  stab[0]=(float) SHT21.getTemperature();
  stab[1]=(float) SHT21.getHumidity();
  stab[2]=0.0; stab[3]=0.0;
  Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
  delay(cycle*1000/4);
  xQueueReset(queue);  // reset queue to store only the last values
  xQueueSend(queue, stab, portMAX_DELAY);
  }
}
}
```

Dans la boucle de tâche principale **`loop()`**, le programme attend les nouvelles données dans la file d'attente avec :

```
xQueueReceive(queue, stab, portMAX_DELAY);
```

9.7.2 Code complet du Terminal LoRa avec la tâche du capteur

Dans le code suivant nous utilisons les éléments de paramétrage fournis dans le fichier **`LoRa_Para.h`**.

```
#define TERMINAL
#define SHT21_SLEEP
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "SHT21.h"
SHT21 SHT21;

QueueHandle_t queue;
int queueSize = 32;
static int taskCore = 0;

typedef union
{
  uint8_t frame[32];    // data frame to send/receive data
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    float   sens[4];     // max 4 values - fields
    uint32_t  tout;      // optional timeout
    uint8_t pad[4];      // padding
  } pack;                // data packet
} DT_t;                  // DaTa type

RTC_DATA_ATTR uint32_t gwID=0, termID;  // gateway identifier stored in RTC memory
int cycle=16, rcv_cycle, rcv_ack=0;      // main cycle in millis

void LoRa_send(uint8_t *frame) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.write(frame,32);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  DT_t p;int i=0;
  if(packetSize==32)
    {
    while (LoRa.available())
      {
      p.frame[i]= LoRa.read(); i++;
      }
    Serial.print("Terminal Received Packet:");
    Serial.println(p.pack.tout);
    gwID=p.pack.sid;
    rcv_cycle=(int)p.pack.tout; rcv_ack=1;
    }
}

void sensor_Task( void * pvParameters )
{
float stab[4];
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  stab[0]=(float) SHT21.getTemperature();
  stab[1]=(float) SHT21.getHumidity();
  stab[2]=0.0; stab[3]=0.0;
  Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
  delay(cycle*1000/4);
  xQueueReset(queue);  // reset queue to store only the last values
  xQueueSend(queue, stab, portMAX_DELAY);
  }
}
```

```
void setup() {
  Serial.begin(9600);
  termID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  queue = xQueueCreate( queueSize, sizeof(float)*4);
  xTaskCreatePinnedToCore(
                    sensor_Task,   /* Function to implement the task */
                    "sensor_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  delay(333);
}

void loop() {

    DT_t p; float stab[8];
    p.pack.did=(uint32_t)gwID;  // in the first cycle unknown and set to 0
    p.pack.sid=(uint32_t)termID;
    p.pack.sens[0]=stab[0]; p.pack.sens[1]=stab[1];
    p.pack.sens[2]=0; p.pack.sens[3]=0;
    p.pack.tout=(uint32_t)millis();
    xQueueReceive(queue, stab, portMAX_DELAY);
    LoRa_send(p.frame); // send a packet
    Serial.println("Send Packet!");
    delay(2000);
    if(rcv_ack)
      {
      cycle=rcv_cycle; rcv_ack=0;
      }
    Serial.printf("Cycle=%d\n",cycle);
    delay(1000*cycle);
  }
```

9.7.3 Le code complet du nœud de passerelle avec la tâche d'affichage

```
#define GATEWAY
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c,12,14);
QueueHandle_t queue;
int queueSize = 32;
static int taskCore = 0;

typedef union
{
  uint8_t frame[32];    // data frame to send/receive data
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    float   sens[4];     // max 4 values - fields
    uint32_t  tout;      // optional timeout
    uint8_t pad[4];      // padding
  } pack;                // data packet
} DT_t;                  // DaTa type

RTC_DATA_ATTR uint32_t gwID=0, termID;  // gateway identifier stored in RTC memory
RTC_DATA_ATTR uint32_t rcv_pack=0;

void LoRa_send(uint8_t *frame) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.write(frame,32);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}
```

```
void onReceive(int packetSize) {
  DT_t p;int i=0;
  if(packetSize==32)
    {
    while (LoRa.available())
      {
      p.frame[i]= LoRa.read(); i++;
      }
    Serial.println("Gateway Received Packet ");
    Serial.printf("T:%2.2f, H:%2.2f\n",p.pack.sens[0],p.pack.sens[1]);
    rcv_pack=1;
    xQueueReset(queue);  // reset queue to store only the last values
    xQueueSend(queue, &p, portMAX_DELAY);
    }
}

void disp_Task(void *pvParameters)
{
float stab[4];
DT_t p;
while(true)
  {
  char buff[32];
  xQueueReceive(queue, &p, portMAX_DELAY);
  Wire.begin(12,14); delay(200);
  display.init();delay(200);
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Gateway - disp_Task");  // first 16 lines are yellow
  sprintf(buff,"T:%2.2f,H:%2.2f\n",p.pack.sens[0],p.pack.sens[1]);
  display.drawString(0,16,buff);
  Serial.println(buff);
  display.display();
  }
}


void setup() {
  Serial.begin(9600);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  queue = xQueueCreate(queueSize, sizeof(float)*4);
  xTaskCreatePinnedToCore(
                    disp_Task,   /* Function to implement the task */
                    "disp_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
}

void loop()
{
if(runEvery(1000))
  { // repeat every 5000 millis
  DT_t p;
  p.pack.did=(uint32_t)termID;
  p.pack.sid=(uint32_t)gwID;
  p.pack.sens[0]=23.67; p.pack.sens[1]=67.67;
  p.pack.sens[2]=0; p.pack.sens[3]=0;
  p.pack.tout=(uint32_t)(10 + random(10,40));
  if(rcv_pack)
    {
    LoRa_send(p.frame);
    rcv_pack=0;
    Serial.printf("Send Packet with timeout=%d, sec=%d\n",p.pack.tout,millis()/1000);
    }
  }
}
```

9.8 A faire

1. Testez le code des nœuds Terminal et Gateway
2. Modifiez la tâche du capteur en ajoutant une troisième valeur de capteur (luminosité) et testez les deux nœuds.
3. Modifiez le code du noeud Gateway en remplacant la boucle principale **`loop()`** par une tâche (**`LoRa_Task`**)

Table des matières