Lab 9 - Programmation temps réel avec FreeRTOS pour IoT

Contenu

Lab 9 - Programmation temps réel avec FreeRTOS pour IoT

9.1 Introduction au FreeRTOS

FreeRTOS est un système d'exploitation temps réel (**RTOS**) open source de faible empreinte et portable. Il fonctionne en **mode préemptif**. Il a été porté sur dizaines d'architectures différentes. Créé en 2003 par Richard Barry et la **FreeRTOS** Team, il est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel.

FreeRTOS est disponible gratuitement sous une licence GPL modifiable et utilisable sans paiement de redevances, cette licence n'oblige pas les développeurs à publier le code de leurs logiciels applicatifs mais impose de garder le noyau de **FreeRTOS Open Source**.

L'unité d'exécution contrôlée par **FreeRTOS** est une **tâche**. Le nombre de tâches exécutées simultanément et leur priorité ne sont limités que par le matériel.
L'ordonnancement est un système de file d'attente basé sur les **Sémaphores** et les **Mutex**. Il fonctionne selon le modèle Round-Robin avec gestion des priorités. Conçu pour être très compact, il n'est composé que de quelques fichiers en langage C, et n'implémente aucun **driver** matériel.

Les domaines d'applications sont assez larges, car les principaux avantages de **FreeRTOS** sont l'exécution temps réel, un code source ouvert et une taille très faible. Il est donc utilisé principalement pour les systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour des systèmes de traitement vidéo et des applications réseau qui ont des contraintes "temps réel".
Bien évidement parmi les application les plus récentes et le plus exigeantes par rapport à la gestion des activités en temps réel sont les architectures d'**objets connectés** dans l'**IoT (Internet of Things).**

L'objectif de ce laboratoire est d'expérimenter et de développer les applications IoT nécessitant la mise en ouvre de plusieurs mécanismes FreeRTOS.

9.1.1 L'ordonnanceur

L'ordonnanceur (***scheduler***) des tâches a pour but principal de décider parmi les tâches qui sont dans l'état **prêt**, laquelle exécuter. Pour faire ce choix, l'ordonnanceur de **FreeRTOS** se base uniquement sur la **priorité** des tâches. Les tâches en **FreeRTOS** se voient assigner à leur création, un **niveau de priorité** représenté par un nombre entier. Le niveau le plus bas vaut zéro et il doit être strictement réservé pour la tâche **Idle**.
Plusieurs tâches peuvent appartenir à un même niveau de priorité.
Dans **FreeRTOS** Il n'y a aucun mécanisme automatique de gestion des priorités (cas du système Linux) . La priorité d'une tâche ne pourra être changée qu'à la demande explicite du développeur.
Les tâches sont de simples fonctions qui généralement , mais pas exclusivement, s'exécutent en **boucle infinie** .

Dans le cas d'un micro-contrôleur possédant un seul cœur (AVR,ARM-Cortex4,ESP12,..) , il y aura à tout moment **une seule tâche en exécution**. L'ordonnanceur garantira toujours que la tâche de plus haute priorité pouvant s'exécuter sera sélectionnée pour entrer dans l'**état d'exécution**. Si deux tâches partagent le même niveau de priorité et sont toutes les deux capables de s'exécuter, alors les deux tâches s'exécuteront **en alternance** par rapport aux r**éveils de l'ordonnanceur (*tick*)**

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

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

Cette constante décrit alors la période de temps allouée au minimum pour chaque tâche ou expliqué autrement, l'intervalle séparant deux réveils de l'ordonnanceur.

Principalement **FreeRTOS** utilise un **ordonnancement préemptif** pour gérer les tâches.
Néanmoins il peut aussi utiliser optionnellement (si la directive lui est donnée) l'**ordonnancement coopératif**.
Dans ce mode d'ordonnancement, un changement de contexte d'exécution a lieu uniquement si la tâche en

exécution permet explicitement à une autre tâche de s'exécuter (en appelant un **`Yield()`** par exemple) ou alors en entrant dans un état de blocage. Les tâches ne sont donc jamais préemptées. Ce mode d'ordonnancement simplifie beaucoup la gestion des tâches malheureusement il peut mener à un système moins efficace et moins sûr.

FreeRTOS peut aussi utiliser un **ordonnancement hybride**, utilisant l'ordonnancement préemptif et l'ordonnancement coopératif. Dans ce mode, un changement de contexte d'exécution peut aussi avoir lieu lors de l'événement d'une interruption.

9.1.1.1 La famine

Dans **FreeRTOS**, la tâche de plus haute priorité prête à s'exécuter sera toujours sélectionnée par l'ordonnanceur. Ceci peut amener à une situation de **famine**. En effet, si la tâche de plus haute priorité n'est jamais interrompue , toutes les tâches ayant une priorité inférieure ne s'exécuteront jamais.
FreeRTOS n'implémente aucun mécanisme automatique pour prévenir le phénomène de famine. Le développeur doit s'assurer lui-même qu'il n'y ait pas de tâches monopolisant tout le temps d'exécution du micro-contrôleur. Pour cela, il peut placer des événements qui interrompront la tâche de plus haute priorité pour un temps déterminé ou jusqu'à l'avènement d'un autre événement et laissant ainsi le champ libre aux tâches de priorités inférieures pour s'exécuter.
Afin d'éviter la famine, le développeur peut utiliser l'ordonnancement à taux monotones . C'est une technique d'assignation de priorité qui attribue à chaque tâche une unique priorité selon sa fréquence d'exécution. La plus grande priorité est attribuée à la tâche de plus grande fréquence d'exécution et la plus petite priorité est attribuée à la tâche de plus petite fréquence. L'ordonnancement à taux monotones permet d'optimiser l'ordonnancement des tâches mais cela reste difficile à atteindre du fait de la nature des tâches qui ne sont pas totalement périodiques.

La tâche `IDLE`

Un micro-contrôleur doit toujours avoir quelque chose à exécuter. En d'autres termes, il doit toujours y avoir une tâche en exécution. **FreeRTOS** gère cette situation en définissant la tâche **IDLE** qui est créée au lancement de l'ordonnanceur. La plus petite priorité du noyau est attribué à cette tâche. Malgré cela, la tâche **IDLE** peut avoir plusieurs fonctions à remplir dont:

- libérer l'espace occupé par une tâche supprimée
- placer le micro-contrôleur en veille afin d'économiser l'énergie du système lorsque aucune tâche applicative n'est en exécution.
- mesurer le taux d'utilisation du processeur

9.1.2 Evolution et les états des tâches

Les tâches sous **FreeRTOS** peuvent exister sous 5 états : **supprimé**, **suspendu**, **prêt**, **bloqué** ou **en exécution**.

Dans **FreeRTOS**, il n'y a aucune variable pour spécifier explicitement l'état d'une tâche, en contrepartie **FreeRTOS** utilise des **listes d'états**. La présence de la tâche dans un type de listes d'états détermine son état (**prêt**, **bloqué** ou **suspendu**). Les tâches changeant souvent d'état, l'ordonnanceur n'aura alors qu'à déplacer la tâche (l'élément **`xListItem`** appartenant à cette même tâche) d'une liste d'états à une autre.

À la création d'une tâche, **FreeRTOS** crée et remplit la **TCB** (***Task Control Block***) correspondant à la tâche, puis il insère directement la tâche dans une ***Ready List ;*** la liste contenant une référence vers toutes les tâches étant dans l'état **prêt**.
FreeRTOS maintient plusieurs ***Ready List,*** une liste existe pour chaque niveau de priorité. Lors du choix de la prochaine tâche à exécuter, l'ordonnanceur analyse les *Ready List* de la plus haute priorité à la plus basse. Plutôt que de définir explicitement un état **en exécution** ou une liste associée à cet état, le noyau **FreeRTOS** décrit une variable **`pxCurrentTCB`** qui identifie le **processus** en exécution. Cette variable pointe vers la **TCB** correspondant au processus se trouvant dans l'une des *Ready List*.

Une tâche peut se retrouver dans l'état **bloqué** lors de l'accès à une file d'attente (**queue**) en lecture/écriture dans le cas où la file est vide/pleine. Chaque opération d'accès à une file est paramétrée avec un ***timeout*** (**`xTicksToWait`**), Si ce ***timeout*** vaut 0 alors la tâche ne se bloque pas et l'opération d'accès à la file est considérée comme échouée. Dans le cas où le ***timeout*** n'est pas nul, la tâche se met dans l'état **bloqué** jusqu'à ce qu'il y ait une modification de la file (par une autre tâche par exemple). Une fois l'opération d'accès à la file possible, la tâche vérifie que son ***timeout*** n'est pas expiré et termine avec succès son opération.

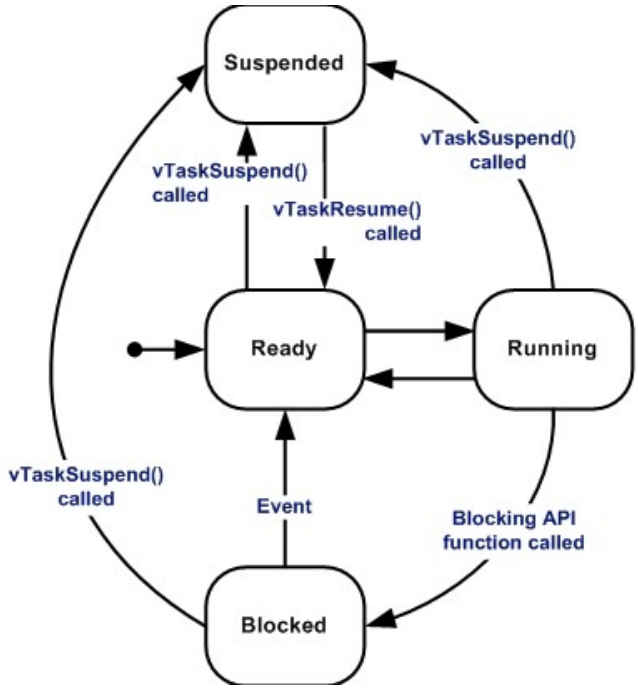

Figure 9.1 Diagramme d'état d'une tâche.

Une tâche peut être volontairement placée dans l'état **suspendu**, elle sera alors totalement ignorée par l'ordonnanceur et ne consommera plus aucune ressource jusqu'à ce qu'elle soit retirée de l'état suspendu et remise dans un état **prêt**.

Le dernier état que peut prendre une tâche est l'état **supprimé**, 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 l'ordonnanceur et la tâche nommée **IDLE** est chargée de libérer les ressources allouées par les tâches étant en état **supprimé**. La tâche '**IDLE**' est créée lors du démarrage de l'ordonnanceur et se voit assigner la plus petite priorité possible ce qui conduit à une libération retardée des ressources lorsque aucune autre tâche est en exécution.

9.2 FreeRTOS et programmation temps réel sur ESP32

Pendant ce laboratoire nous allons étudier comment exploiter **FreeRTOS** implémenté dans la carte **ESP32**. Le code Arduino ESP32 est construit sur **FreeRTOS** et en fait le programme principal qui est mis dans un **`loop Task`** .
Si nous n'utilisons qu'une seule tâche - **`loop Task`** , nous ne profiterons pas pleinement de **FreeRTOS** en mode **multitâche**.

FreeRTOS permet de déployer plusieurs tâches fonctionnant en concurrence et gérées par l'ordonnanceur du RTOS. Sur l'ESP32 les tâches peuvent être déployées sur **2 cœurs** ce qui, dans le contexte IoT donne, par exemple, la possibilité de faire tourner les tâches associées aux capteurs/activateurs en parallèle avec les tâches de la communication radio.

Notre étude est organisée en 5 parties:

- création et suppression des tâches
- les mécanismes de la communication entre les tâches (variables globales et files d'attente)
- les mécanismes de la synchronisation (sémaphores, interruptions, événements)
- déploiement des tâches sur l'architecture multi- cœur (bi-cœur)
- une étude de cas (*case study*) – communication bidirectionnelle avec modem LoRa

9.2.1 Création et suppression d'une tâche

On crée 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 sur la fonction qui implémentera la tâche. Nous allons créer une fonction **additionalTask**, que nous définirons en dernier et qui sera passée dans cet argument.
- **`TaskName`**: Le nom de la tâche, dans une chaîne. Nous allons utiliser « **additionalTask** ».
- **`StackDepth`**: La taille de la pile de la tâche, spécifiée comme le nombre de variables qu'elle peut contenir (pas le nombre d'octets).. Dans cet exemple simple, nous allons passer une valeur qui est 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, donc nous passons **`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 sur les appels aux fonctions (par exemple, pour supprimer une tâche ou modifier sa priorité). Pour cet exemple simple, nous n'allons pas l'utiliser, donc ce sera **`NULL`**.

Cette fonction retourne **`pdPASS`** en cas de succès ou un code d'erreur . Pour l'instant, nous supposerons que la tâche sera créée sans aucun problème, donc nous n'allons pas faire de vérification d'erreurs. Naturellement, 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 suffit d'appeler la fonction **`vTaskDelete(`**.
Cette fonction reçoit en entrée le descripteur de la tâche à supprimer (souvenez-vous de l'argument de **`xTaskCreate`** que nous n'allons pas utiliser. Néanmoins, si nous passons **`NULL`** en entrée, la tâche appelante sera supprimée , ce que nous voulons, puisque nous allons l'appeler à partir du propre code de la tâche.

```
vTaskDelete(NULL);
```

9.2.2 Création et exécution d'une simple tache supplémentaire – code Arduino

Dans le premier exemple, à côté de la tâche Arduino (**loopTask**), nous ajouterons à notre application une tâche supplémentaire. Notre application a 2 tâches: La tâche Arduino imprimera le texte **"ceci est une tâche Arduino"** et la deuxième tâche imprimera **"c'est une tâche supplémentaire"** sur le terminal série.

```
// esp32.HT.freeRTOS.1
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 Arduino ESP32 loopTask */
void loop() {
  Serial.println("this is Arduino 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 Arduino Task
this is additional Task
this is Arduino Task
this is additional Task
this is Arduino Task
this is additional Task
..
```

9.2.3 Création et exécution de deux tâches

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

```
// esp32.HT.freeRTOS.2
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); }

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 );
}
```

Résultat d'exécution :

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

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

A faire :

Testez le même programme avec la priorité 2 pour la tâche **`taskTwo`**. Commentez le résultat.

9.3 Communication entre deux tâches

Il y'a plusieurs manières de faire communiquer les taches entre elles (variables globales, files d'attente (*queues*), ..)

9.3.1 Variables globales comme arguments

Pour commencer nous allons passer un simple paramètre à partir d'une **variable globale.** Les fonctions de tâche reçoivent un paramètre générique (**`void *`**). Dans notre code, nous allons l'interpréter comme un **pointeur** sur **`int,`** ce qui correspond à (**`int *`**).

Donc, la première chose que nous faisons est un **cast** à (**`int *`**).

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

Maintenant, nous avons un pointeur sur la position de la mémoire d'un **entier**. Néanmoins, nous voulons accéder au contenu actuel de la position de la mémoire. Donc, nous voulons la valeur de la position de la mémoire à 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 être en mesure d'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 allons faire dans les deux fonctions.
Le code source complet peut être vu ci-dessous, avec l'implémentation pour les deux fonctions. Notez qu'à des fins de débogage, nous imprimons différentes chaînes dans les fonctions.

```
// esp32.HT.freeRTOS.3
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 d'exécution :

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

9.3.2 Files d'attente

Les files d'attente (**queues**) sont très utiles pour la communication inter-tâches, permettant d'envoyer des messages d'une tâche à une autre en toute sécurité en termes de concurrence. Ils sont généralement utilisés comme **FIFO** (**First In First Out**), ce qui signifie que les nouvelles données sont insérées à l'arrière de la file d'attente et consommées à partir du front.

Les données insérées 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 réellement copiée et si nous changeons la valeur d'origine après cela, aucun problème ne devrait se produire.

Un aspect de comportement important est que l'insertion dans une file d'attente pleine ou la consommation 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 inter-tâches, pour cet exemple d'introduction, nous insérerons et consommerons des éléments dans la file d'attente de la fonction de boucle principale Arduino, afin de nous concentrer sur les appels API de base.

```
// esp32.HT.freeRTOS.4
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);
}
```

Le même type de programme précédé par la création de deux tâches **producerTask** et **consumerTask**.

```
// esp32.HT.freeRTOS.5
QueueHandle_t queue;
int queueSize = 10;

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);
}

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 );
}
```

A faire :

Expérimenter avec différentes taille de file d'attente (128,1000, ..)
et avec différents types de données (**char, float, double,..**) et même structures.

9.3.3 Une application IoT de traitement par tâches et de communication par file d'attente.

Dans l'exemple suivant nous allons exploiter le capteur de la température/humidité **HTU21D**.

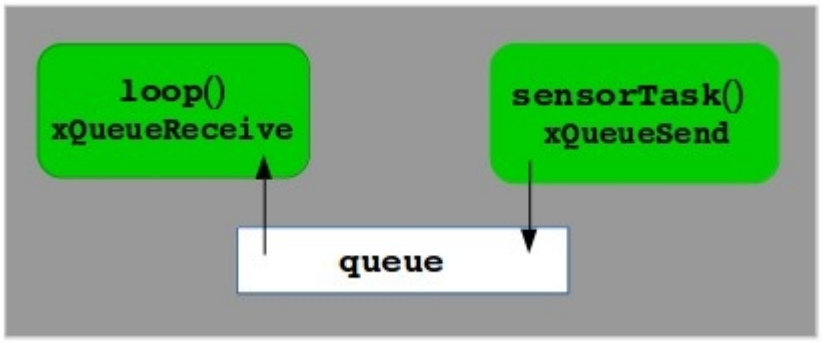

Figure 9.2 Tâches `loop` et `sensorTask` et file d'attente `queue`

```
// esp32.HT.freeRTOS.6
#include <Wire.h>
#include "Adafruit_HTU21DF.h"
Adafruit_HTU21DF htu = Adafruit_HTU21DF();
//#include <Wire.h>
//#include <SHT21.h>

QueueHandle_t queue;
int queueSize = 128;

void sensorTask( void * pvParameters ){
    float t,h;
    while(true){
        t=htu.readTemperature();
        h=htu.readHumidity();
         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("HTU21D-F test");
  if (!htu.begin()) {
    Serial.println("Couldn't find sensor!");
    while (1);
}
  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);
    }
}
```

A faire:

Tester le même programme avec un autre capteur (selon la disponibilité): luminosité, CO2,pression atmosphérique, GPS,

9.4 Sémaphores

Les sémaphores permettent de construire les schémas de synchronisation et de protection pour les sections partagées (critiques) du code.

Dans cette section nous allons présenter plusieurs types de **sémaphores** :

- sémaphore binaire
- mutex
- sémaphore de comptage

9.4.1 Sémaphore binaire et une ISR (routine d'interruption)

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 gardera la ressource jusqu'à ce qu'elle appelle **`xSemaphoreGive()`** pour libérer la ressource afin que d'autres tâches puissent la capter.
2. en cas d'échec, elle attendra que la ressource soit libérée par une autre tâche.

Binary sémaphores sera appliquée au traitement d'interruption (ISR) où la fonction de rappel ISR appellera **`xSemaphoreGiveFromISR()`** pour déléguer le traitement d'interruption à la tâche qui appelle **`xSemaphoreTake()`**.

Quand **`xSemaphoreTake()`** est appelée, la tâche sera bloquée et attente d'un événement d'interruption.

Remarque:
Le sémaphore binaire que nous utilisons dans cet exemple est un peu différent du sémaphore « **standard** » car la tâche qui appelle **`xSemaphoreTake()`** ne libère pas le sémaphore.

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

Exemple

Nous allons réutiliser l'exemple suivant du code en utilisant le style **FreeRTOS** et le sémaphore binaire pour traiter l'interruption.

```
// esp32.HT.freeRTOS.7
byte ledPin = 25;   // LED on DevKit ESP32 – Heltec board
byte interruptPin = 0; // button PRG on ESP32 – Heltec board
/* hold the state of LED when toggling */
volatile byte state = LOW;

SemaphoreHandle_t xBinarySemaphore;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);  // set interrupt pin in pull-up mode
  // attach interrupt to the pin
  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 );
}
```

A faire

Afficher le message généré par la fonction ISR - **`ISRprocessing is running`** sur l'écran OLED.

9.4.2 Mutex

Un sémaphore de type **mutex** est comme une clé associée à la ressource. La tâche laquelle contient la clé, verrouille la ressource, la traite, puis déverrouille et restitue la clé pour que d'autres tâches puissent l'utiliser. Ce mécanisme est similaire au sémaphore binaire sauf que la tâche qui prend la clé **doit libérer la clé**.

Remarque:
Supposons que nous ayons 2 tâches: une tâche à priorité basse et une tâche à priorité élevée. Ces tâches attendent la clé et la tâche de faible priorité a la chance de capter la clé, puis elle bloque la tâche de haute priorité et continue l'exécution.

9.4.2.1 Exemple de code

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

```
// esp32.HT.freeRTOS.8
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(2000);
    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(1000);
  }
  vTaskDelete( NULL );
}
```

Le résultat d'exécution :

```
highPriorityTask gains key
lowPriorityTask releases key
highPriorityTask is running
highPriorityTask releases key
lowPriorityTask gains key
highPriorityTask gains key
lowPriorityTask releases key
highPriorityTask is running
highPriorityTask releases key
lowPriorityTask gains key
…
```

9.4.3 Sémaphore de comptage (*counting semaphore*)

Les sémaphores sont généralement utilisés à la fois pour la synchronisation et l'exclusion mutuelle dans l'accès aux ressources. Nous allons développer une application simple où nous utiliserons un **sémaphore de comptage** comme barrière d'exécution.

Dans notre programme, la fonction de **`setup()`** lancera une quantité configurable de tâches, puis elle attendra sur un sémaphore pour que toutes les tâches soient terminées. Après le lancement des tâches, la fonction de **`setup()`** essayera de tenir un sémaphore autant de fois que le nombre des tâches lancées, et continuera seulement l'exécution après ce point.
Notez que lorsqu'une tâche essaie de prendre un sémaphore avec le compteur de sémaphore égal à **`0`**, elle reste bloquée jusqu'à ce que le compte de sémaphore soit incrémenté par une autre tâche ou qu'un délai d'attente défini se produise.
Nous allons commencer notre code en déclarant une **variable globale** pour y enregistrer le nombre de tâches à lancer. Nous allons lancer notre exemple avec 4 tâches, mais vous pouvez le changer en un autre nombre.

Ensuite, nous allons déclarer un ***counting* sémaphore** comme une variable globale. Cette variable peut être consultée à la fois par la fonction de **`setup()`** et les tâches.

Nous créons le sémaphore avec la fonction **`xSemaphoreCreateCounting()`**. Cette fonction reçoit comme arguments d'entrée le nombre maximal que le sémaphore peut atteindre et sa valeur initiale.
Puisque nous l'utilisons à des fins de synchronisation, nous spécifierons un nombre maximum égal au nombre de tâches à lancer et l'initialiserons à **`0`**

```
int nTasks=4;
SemaphoreHandle_t barrierSemaphore = xSemaphoreCreateCounting(nTasks,0);
```

Dans le **`setup()`** nous écrivons une boucle pour lancer nos tâches. Notez que nous pouvons utiliser la même fonction de tâche pour lancer plusieurs tâches, nous n'avons donc pas besoin d'implémenter du code pour chaque nouvelle tâche.
Nous allons créer les tâches avec la priorité **`0`**. La fonction de **`setup()`** s'exécute avec une priorité de **`1`**.

Etant donné que les tâches ont une priorité inférieure, elles ne fonctionneront que si la fonction de **`setup()`** est bloquée et nous nous attendons à ce qu'elle bloque sur notre sémaphore.

Pour différencier les tâches, nous allons passer le numéro de l'itération de la boucle comme un paramètre d'entrée, afin qu'ils puissent l'imprimer dans leur code..

Notez que nous allons passer un pointeur sur la position mémoire de la variable d'itération et non sur sa valeur réelle. Néanmoins, cela ne posera pas de problème puisque les tâches doivent s'exécuter et utiliser cette valeur avant que la fonction de **`setup()`** ne se termine pas , et ainsi, la mémoire de la variable sera toujours valide.

```
void setup()
{
Serial.begin(9600);delay(1000);
Serial.println("Starting to launch tasks..");
for(int i= 0; i< nTasks; i++){
    xTaskCreatePinnedToCore(
                    genericTask,    /* Function to implement the task */
                    "genericTask",  /* Name of the task */
                    10000,          /* Stack size in words */
                    (void *)&i,     /* Task input parameter */
                    0,              /* Priority of the task */
                    NULL,           /* Task handle. */
                    1);             /* Core where the task should run */
}
```

Après le lancement des tâches, nous exécutons une boucle **`for()`** essayant d'obtenir le sémaphore autant de fois que les tâches lancées. Ainsi, la fonction de **`setup()`** ne doit passer à partir de ce point d'exécution que s'il y a autant d'unités dans le sémaphore que de tâches.

Comme les tâches incrémentent le sémaphore, nous devrions garantir que la fonction de **`setup()`** ne se terminera qu'après que toutes les tâches soient terminées.

Pour prendre une unité à partir d'un sémaphore, nous appellerons la fonction **`xSemaphoreTake()`**. Cette fonction reçoit comme premier argument le **`handle`** du sémaphore (la variable globale que nous avons déclarée et initialisée au début) et un nombre maximum de **`ticks`** **FreeRTOS** à attendre.

Puisque nous voulons le bloquer indéfiniment jusqu'à ce qu'il puisse obtenir le sémaphore, nous passons à ce dernier argument la valeur **`portMAX_DELAY`** .

Après cela, nous allons exécuter un message final, indiquant que la fonction de setup() a passé ce point d'exécution.

```
for(i= 0; i< nTasks; i++){
    xSemaphoreTake(barrierSemaphore, portMAX_DELAY);
}
Serial.println("Tasks launched and semaphore passed...");
```

Maintenant, nous allons implémenter la boucle principale où, à titre d'exemple, nous allons utiliser une fonction appelée **`vTaskSuspend()`** pour suspendre la tâche de la boucle principale.

Cela bloquera son exécution quelle que soit sa priorité. Cette fonction reçoit en entrée le **`handle`** de la tâche à suspendre . Dans ce cas, puisque nous voulons suspendre la tâche appelante, nous passons **`NULL`**

```
void loop() {
  vTaskSuspend(NULL);
}
```

Enfin, nous allons implémenter le code pour nos tâches. Ils vont simplement commencer, imprimer leur numéro (à partir de l'argument passé lors de leur lancement), incrémenter le sémaphore et finir.

Donc, nous commençons par construire et imprimer la chaîne indiquant le numéro de la tâche. Ensuite, nous allons **incrémenter** le sémaphore en utilisant la fonction **`xSemaphoreGive()`**.

Cette fonction reçoit en entrée le **`handle`** du sémaphore. Notez qu'il existe également un **`xSemaphoreGiveFromISR`** à utiliser dans le contexte d'une routine de service d'interruption.

Vérifiez le code de la fonction ci-dessous, qui inclut déjà l'appel pour supprimer la tâche après l'exécution.

```
void genericTask( void * parameter ){
    String taskMessage = "Task number:";
    taskMessage = taskMessage + *((int *)parameter);
    Serial.println(taskMessage);
    xSemaphoreGive(barrierSemaphore);
    vTaskDelete(NULL);
}
```

9.4.3.1 Code complet de l'exemple :

```
// esp32.HT.freeRTOS.9
int nTasks=4;
SemaphoreHandle_t barrierSemaphore = xSemaphoreCreateCounting( nTasks, 0 );

void genericTask( void * parameter ){
    String taskMessage = "Task number:";
    taskMessage = taskMessage + *((int *)parameter);
    Serial.println(taskMessage);
    xSemaphoreGive(barrierSemaphore);
    vTaskDelete(NULL);
}

void setup() {
  Serial.begin(9600);
  delay(1000);
  Serial.println("Starting to launch tasks..");
  int i;
  for(i= 0; i< nTasks; i++){
    xTaskCreatePinnedToCore(
                    genericTask,    /* Function to implement the task */
                    "genericTask",  /* Name of the task */
                    10000,          /* Stack size in words */
                    (void *)&i,     /* Task input parameter */
                    0,              /* Priority of the task */
                    NULL,           /* Task handle. */
                    1);             /* Core where the task should run */
  }
  for(i= 0; i< nTasks; i++){
    xSemaphoreTake(barrierSemaphore, portMAX_DELAY);}
  Serial.println("Tasks launched and semaphore passed...");
}

void loop() { vTaskSuspend(NULL); }
```

Résultat d'exécution :

```
Starting to launch tasks..
Task number:0
Task number:1
Task number:2
Task number:3
Tasks launched and semaphore passed…
```

9.5 Gestion des événements

9.4.1 Bits d'événement (indicateurs d'événement)

Les **bits d'événement** sont utilisés pour indiquer si un événement s'est produit ou non. Les bits d'événement sont souvent appelés **indicateurs d'événement**.

Par exemple, une application peut:

- Définir un bit (ou un indicateur) qui signifie "Un **message a été reçu** et est prêt pour le traitement" lorsqu'il est défini sur **1** et "**aucun message** n'est en attente de traitement" lorsqu'il est défini sur **0**.
- Définir un bit (ou un indicateur) qui signifie "L'application a mis en **file d'attente** un message prêt à être **envoyé sur un réseau**" lorsqu'il est défini sur 1 et "aucun message n'est en attente d'être envoyé au réseau". est défini sur 0.
- Définir un bit (ou indicateur) qui signifie "Il est temps d'envoyer un **message de pulsatio**n (***heartbeat***) sur un réseau" lorsqu'il est défini sur 1, et "il n'est pas encore temps d'envoyer un autre message de pulsation" lorsqu'il est défini sur 0.
-

9.5.2 Groupes d'événements

Un **groupe** d'événements est un **ensemble de bits d'événement**. Les bits d'événement individuels dans un groupe d'événements sont référencés par le numéro du bit.

Extension de l'exemple fourni ci-dessus:

- Le bit d'événement qui signifie "Un message a été reçu et est prêt à être traité" peut être le numéro de bit 0 **dans un groupe d'événements**.
- Le bit d'événement qui signifie que "l'application a mis en file d'attente un message prêt à être envoyé à un réseau" peut être le bit numéro 1 dans le même groupe d'événements.
- Le bit d'événement qui signifie "Il est temps d'envoyer un message de pulsation sur un réseau" peut être le numéro de bit 2 dans le même groupe d'événements.

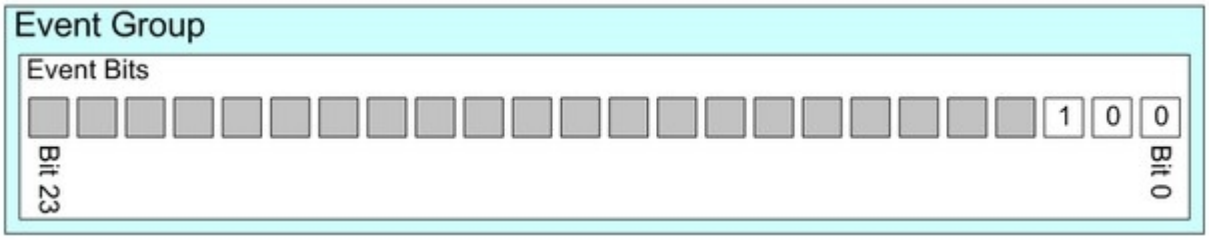

Figure 9.3 Un groupe de bits d'événements

9.5.3 Fonctions de l'API du groupe d'événements RTOS

Des fonctions d'API de groupe d'événements permettent à une tâche, entre autres choses, de définir un ou plusieurs bits d'événement dans un groupe d'événements, d'effacer un ou plusieurs bits d'événement dans un groupe d'événements et de passer à l'état bloqué pour attendre qu'un ensemble d'un ou plusieurs bits d'événement soit activé dans un groupe d'événements.

Les groupes d'événements peuvent également être utilisés pour **synchroniser des tâches**, créant ce que l'on appelle souvent une «**tâche de rendez-vous**». Un point de synchronisation de tâches est une place dans le code de l'application à laquelle une tâche attendra à l'état bloqué (ne consommant aucun temps CPU) jusqu'à ce que toutes les autres tâches participant à la synchronisation aient également atteint leur point de synchronisation.

9.5.3.1 Exemple du code

```
// esp32.HT.freeRTOS.10
#include "freertos/event_groups.h"
// define event bits
#define TASK_1_BIT        ( 1 << 0 ) //1
#define TASK_2_BIT        ( 1 << 1 ) //10
#define TASK_3_BIT        ( 1 << 2 ) //100
#define ALL_SYNC_BITS (TASK_1_BIT | TASK_2_BIT | TASK_3_BIT) //111
hw_timer_t * timer = NULL; // create a hardware timer
EventGroupHandle_t eg;  // create event group
int count = 0;

void IRAM_ATTR onTimer(){    // timer ISR callback
BaseType_t xHigherPriorityTaskWoken; count++;
switch(count == 2)
  {
  case 2: // set event bit of task1
  xEventGroupSetBitsFromISR(eg,TASK_1_BIT, &xHigherPriorityTaskWoken); break;
  case 3: // set event bit of task 2 and 3
    xEventGroupSetBitsFromISR(eg,TASK_2_BIT|TASK_3_BIT,&xHigherPriorityTaskWoken);
    break ;
  case 4: // reset counter to start again
    count = 0; break;
  default: break;
  }
}

void setup() {
  Serial.begin(9600);
  eg = xEventGroupCreate();
  // 1 tick takes 1/(80MHZ/80) → 1 microsecond
  timer = timerBegin(0, 80, true);
  // attach onTimer function to our timer
  timerAttachInterrupt(timer, &onTimer, true);
  // call onTimer function every second  1000000 ticks
  timerAlarmWrite(timer, 1000000, true); // true - repeat the alarm
  timerAlarmEnable(timer); // start timer alarm
  Serial.println("start timer");
  xTaskCreate(
      task1,           /* Task function. */
      "task1",        /* 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 */
  xTaskCreate(
      task2,           /* Task function. */
      "task2",        /* 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 */

xTaskCreate(
```

```
      task3,           /* Task function. */
      "task3",        /* 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 */
}

void loop() {}

void task1( void * parameter )
{
  for(;;){// wait forever until event bit of task 1 is set
  EventBits_t xbit=xEventGroupWaitBits(eg,TASK_1_BIT,pdTRUE,pdTRUE,portMAX_DELAY);
  Serial.print("task1 has event bit: "); Serial.println(xbit);
  }
  vTaskDelete( NULL );
}

void task2( void * parameter )
{
  for(;;){//wait forever until event bit of task 2 is set
  EventBits_t xbit=xEventGroupWaitBits(eg,TASK_2_BIT,pdTRUE,pdTRUE,portMAX_DELAY);
  Serial.print("task2 has event bit: "); Serial.println(xbit);
  }
  vTaskDelete( NULL );
}

void task3( void * parameter )
{
  for(;;){// wait forever until event bit of task 3 is set
  EventBits_t xbit=xEventGroupWaitBits(eg,TASK_3_BIT,pdTRUE,pdTRUE,portMAX_DELAY);
    Serial.print("task3 has even bit: ");Serial.println(xbit);
  }
  vTaskDelete( NULL );
}
```

9.6 Les tâches FreeRTOS sur un processeur multi-cœur (ESP32)

L'utilisation d'un processeur avec les taches indépendantes pour les applications IoT permet de séparer les fonctionnalités de capture et de communication.

Pour aller plus loin nous pouvons implanter ces fonctionnalités sur les tâches exécutées sur plusieurs processeurs. Dans notre cas nous allons montrer comment introduire la deuxième unité d'exécution puis comment partager les tâches sur deux processeurs.

9.6.1 Création d'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 lancerons sera épinglée. Cela garantit que nous pouvons facilement le modifier lors du test du code. Notez que nous allons exécuter le code deux fois, en affectant les valeurs 0 et 1 à cette variable.

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

Dans la fonction **`setup()`** nous ouvrons une connexion série avec un débit en bauds de 9600, afin d'envoyer des messages informatifs sur le terminal.
Après cela, nous allons imprimer le numéro du coeur que nous testons, qui est stocké dans la variable globale précédemment déclarée.

Ensuite nous allons lancer la tâche **FreeRTOS**, en l'assignant à un CPU spécifique de l'ESP32. Nous allons utiliser la fonction **`xTaskCreatePinnedToCore()`**. Cette fonction prend exactement les mêmes arguments de **`xTaskCreate()`** et un argument supplémentaire à la fin pour spécifier le CPU où la tâche devrait s'exécuter.

Nous allons implémenter la tâche dans une fonction appelée **`coreTask()`**. Nous devrions attribuer la priorité **`0`** à la tâche, donc elle est inférieure à la fois à la configuration – **`setup()`**, et à la boucle principale - **`loop()`**.

Nous n'avons pas besoin de nous soucier de transmettre des paramètres d'entrée ou de stocker le **`handle`** de la tâche.
Dans la boucle principale – **`loop()`**, nous commençons par imprimer un message sur le port série indiquant que nous sommes en train de lancer la boucle principale

Après cela nous programmons une boucle **`while()`** infinie, sans aucun code à l'intérieur. Il est crucial que nous ne mettions aucun type de fonction d'exécution ou de retard à l'intérieur. La meilleure approche est de la laisser vide.

La fonction de la tâche est également très simple. Nous allons juste imprimer un message indiquant le CPU qui lui est assigné. Il est obtenu avec **`xPortGetCoreID()`**.
Naturellement, cela doit correspondre au CPU spécifié dans la variable globale.

```
// esp32.HT.freeRTOS.11
static int taskCore0 = 0;
static int taskCore1 = 1;

void coreTask( void * pvParameters ){
    String taskMessage = "Task running on core ";
    taskMessage = taskMessage + xPortGetCoreID();
    while(true){
        Serial.println(taskMessage);
        delay(1000);
    }
}

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

```
  Serial.println(taskCore0);
  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. */
                    taskCore0);  /* Core where the task should run */
  Serial.println("Task created...");
  Serial.println(taskCore1);
  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. */
                    taskCore1);  /* Core where the task should run */
  Serial.println("Task created...");

}

void loop() {
  Serial.println("Starting main loop...");
  while(true){
    delay(1000);
    }  // no delay - CPU is occupied 100%
}
```

A faire :

Tester le code sans et avec un délai dans la fonction **`loop().`**

9.6.2 Simple application IoT exécutée sur 2 coeurs

Dans l'exemple suivant nous allons exploiter 2 coeurs d'exécution , un pour lire les données du capteur, et l'autre pour afficher ces données sur le terminal et/ou sur l'écran OLED.

La communication entre les tâches est effectué par le biais d'une file d'attente.

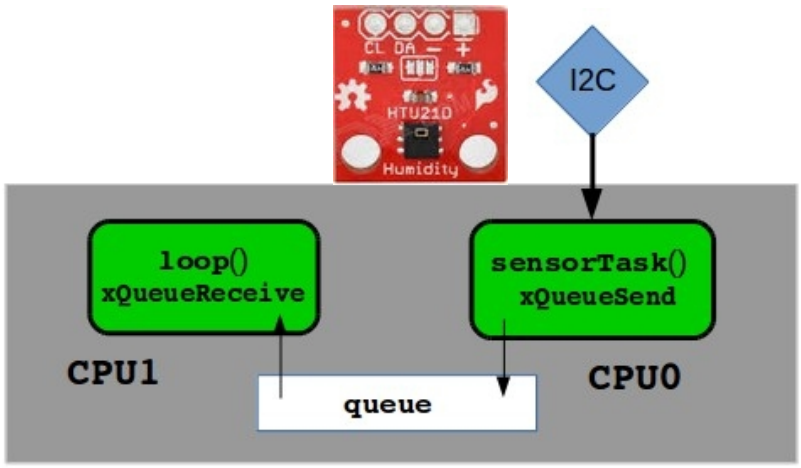

Figure 9.4 Tâches **loop** et **sensorTask** sur 2 processeurs et une file d'attente **queue**

```
// esp32.HT.freeRTOS.12
#include <Wire.h>
#include <SHT21.h>
SHT21 sht;
QueueHandle_t queue;
int queueSize = 128;
static int taskCore = 1;  // or 0

void sensorTask( void * pvParameters ){
    String taskMessage = "Task running on core ";
    taskMessage = taskMessage + xPortGetCoreID();
    float t,h;
    while(true){
        Serial.println(taskMessage);
        t=sht.getTemperature();
        h=sht.getHumidity();
         xQueueSend(queue, &t, portMAX_DELAY);
         xQueueSend(queue, &h, portMAX_DELAY);
        delay(1000);
    }
}

void setup() {
  Serial.begin(9600);
  Wire.begin();
  delay(1000);
    queue = xQueueCreate( queueSize, sizeof( int ) );
  Serial.print("Starting to create task on core ");
  Serial.println(taskCore);
  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){
    Serial.println("main on core 1");
    xQueueReceive(queue, &t, portMAX_DELAY);
    xQueueReceive(queue, &h, portMAX_DELAY);
    Serial.print("temp:");Serial.println(t);
    Serial.print("humi:");Serial.println(h);
    delay(1000);
    }
}
```

A faire :

1. Modifier le numéro du core et tester le même programme sur deux cores différents.
2. Tester le même programme avec différents capteurs , .
3. Ajouter une tâche d'affichage sur l'OLED
4. Dans la tâche principale ajouter une fonction de communication **WiFi** pour envoyer les données sur le serveur **ThingSpeak.fr**

```
// esp32.HT.freeRTOS.12.Oled.Wifi.TS
#include <WiFi.h>
#include "ThingSpeak.h"
#include <U8x8lib.h>
#include <Wire.h>
#include <SHT21.h>
SHT21 sht;
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16); // data, clock, reset
char ssid[] = "Livebox-08B0";          //  your network SSID (name)
char pass[] = "G79ji6dtEptVTPWmZP";   // your network passw
WiFiClient  client;
QueueHandle_t queue;
struct {
  float t;
  float h;
} ssens,rsens;  // send sensors, receive sensors

int queueSize = 128;
static int taskCore = 1;  // or 0

void sensorTask( void * pvParameters ){
    String taskMessage = "Task running on core ";
    taskMessage = taskMessage + xPortGetCoreID();
```

```
    while(true){
        Serial.println(taskMessage);
        ssens.t=sht.getTemperature();
        ssens.h=sht.getHumidity();
        xQueueSend(queue, &ssens, portMAX_DELAY);
        delay(1000);
    }
}

void setup() {
  Serial.begin(9600);
  Wire.begin();
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  WiFi.disconnect(true);delay(1000);
  WiFi.begin(ssid, pass);
  Serial.println(WiFi.getMode());delay(1000);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);Serial.print(".");}
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);
  Serial.println("WiFi setup ok");
  delay(1000);
  Serial.println(WiFi.status());
  ThingSpeak.begin(client);
    delay(1000);
  Serial.println("ThingSpeak begin");;
  delay(1000);
    queue = xQueueCreate(queueSize,8);  // 2 floats
  Serial.print("Starting to create task on core ");
  Serial.println(taskCore);
  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() {
  char dbuff[32];
  Serial.println("Starting main loop...");
  while (true){
    Serial.println("main on core 1");
    xQueueReceive(queue, &rsens, portMAX_DELAY);
    Serial.print("temp:");Serial.println(rsens.t);
    Serial.print("humi:");Serial.println(rsens.h);
    u8x8.clear();
    sprintf(dbuff,"Temp:%4.2f",rsens.t);
    u8x8.drawString(0,1,dbuff);
```

```
    sprintf(dbuff,"Humi:%4.2f",rsens.h);
    u8x8.drawString(0,2,dbuff);
    ThingSpeak.setField(1, rsens.t);
    ThingSpeak.setField(2, rsens.h);
    while (WiFi.status() != WL_CONNECTED) {
      delay(500);Serial.print(".");}
    ThingSpeak.writeFields(1, "HEU64K3PGNWG36C4");  // channel 1 on ThingSpeak.fr
    delay(15000);
    }
}
```

9.7 *Case study* 1- Un terminal IoT avec quatre *tasks*

L'exemple suivant est un terminal IoT avec deux ou plus capteurs connecté à un point d'accès WiFi et le serveur **ThingSpeak.fr/.com**.
Les fonctionnalités du terminal sont décomposées en tâches de telle façon que chaque dispositif est indépendant est asynchrone par rapport aux autres.
Les tâches sont suivantes :

- tâche de capteurs
- tâche d'afficheur OLED
- tâche de communication WiFi – **ThingSpeak**
- tâche de fond (à compléter selon le besoin)

9.7.1 Le code de l'application

```
#include <WiFi.h>
#include "ThingSpeak.h"
#include <Adafruit_MLX90614.h>
#include "SHT21.h"
SHT21 SHT21;
Adafruit_MLX90614 mlx = Adafruit_MLX90614();
WiFiClient  client;
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16); // data, clock,
char ssid[] = "...";          //  your network SSID (name)
char pass[] = "..";   // your network passw
uint32_t delayMS;

QueueHandle_t squeue,oledqueue,TSqueue;
int queueSize=4;
SemaphoreHandle_t xSemSensTS;

float sensors[4];

void taskTS( void * parameter)
{
  const TickType_t xDelay = 30000 / portTICK_PERIOD_MS; // portTICK_PERIOD is 1 ms
  float sensors[4];
  while(true)
  {
  xSemaphoreTake(xSemSensTS,20000);
  xQueueReceive(TSqueue,(float *)sensors,10000);
  ThingSpeak.setField(1, sensors[0]);
  ThingSpeak.setField(2, sensors[1]);
  ThingSpeak.setField(3, sensors[2]);
  ThingSpeak.setField(4, sensors[3]);
  while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
      }
    ThingSpeak.writeFields(4, "4M9QG56R7VGG8ONT");
    xSemaphoreGive(xSemSensTS);
    vTaskDelay(xDelay);
  }
}
```

```
void taskOLED( void * parameter)
{
  const TickType_t xDelay = 3000 / portTICK_PERIOD_MS; // portTICK_PERIOD is 1 ms
  //float temp,humi;
  float sensors[4];
  char dbuff[32];
  // Get temperature event and print its value.
  while(true)
  {
  xQueueReceive(oledqueue,(float *)sensors,10000);
  u8x8.clear();
  sprintf(dbuff,"ATemp:%2.2f*C",sensors[0]);
  u8x8.drawString(0,0,dbuff);
  sprintf(dbuff,"OTemp:%2.2f",sensors[1]);
  u8x8.drawString(0,1,dbuff);
  sprintf(dbuff,"Temp:%2.2f*C",sensors[2]);
  u8x8.drawString(0,2,dbuff);
  sprintf(dbuff,"Humi:%2.2f",sensors[3]);
  u8x8.drawString(0,3,dbuff);
  }
}

void taskMLX906( void * parameter)
{
  const TickType_t xDelay = 10000 / portTICK_PERIOD_MS; // portTICK_PERIOD is 1 ms
  //float temp,humi;
  float sensors[4];
  // Get temperature event and print its value.
  while(true)
  {
  xSemaphoreTake(xSemSensTS,20000);
    sensors[0]=(float) mlx.readAmbientTempC();
    Serial.print("Ambient temp ");
    Serial.print(sensors[0]);
    Serial.println(" *C");
    delay(300);
    sensors[1]=(float)mlx.readObjectTempC();
    Serial.print("Object temp ");
    Serial.print(sensors[1]);
    Serial.println("%");
    delay(300);
    sensors[2]=(float)SHT21.getTemperature();
    Serial.print("Temp SHT21 ");
    Serial.print(sensors[2]);
    Serial.println("*C");
    delay(300);
    sensors[3]=(float)SHT21.getHumidity();
    Serial.print("Humi SHT21 ");
    Serial.print(sensors[3]);
    Serial.println("%");

  xQueueReset(squeue);
  xQueueSend(squeue,(float *)sensors,0);
  xQueueReset(oledqueue);
```

```
  xQueueSend(oledqueue,(float *)sensors,0);
  xQueueReset(TSqueue);
  xQueueSend(TSqueue,(float *)sensors,0);
  delay(400);
  xSemaphoreGive(xSemSensTS);
  vTaskDelay(xDelay);
  }
}

void setup() {
  char dbuff[32];
  Serial.begin(9600);
  Wire.begin(21,22);   // SDA, SCL
  delay(100);
  mlx.begin();
  delay(100);
  SHT21.begin();
   WiFi.disconnect(true);
   delay(1000);
   WiFi.begin(ssid, pass);
    while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
    }
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);
  Serial.println("WiFi setup ok");
    ThingSpeak.begin(client);
    delay(1000);
  Serial.println("ThingSpeak begin");
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
  u8x8.drawString(0,0,"IP address");
  sprintf(dbuff,"%d.%d.%d.%d",ip[0],ip[1],ip[2],ip[3]);
  u8x8.drawString(0,1,dbuff);
  delay(100);
  u8x8.drawString(0,3,"Start TS");
  delay(3000);

squeue = xQueueCreate(queueSize,4*sizeof(float));
oledqueue = xQueueCreate(queueSize,4*sizeof(float));
TSqueue = xQueueCreate(queueSize,4*sizeof(float));
xSemSensTS = xSemaphoreCreateMutex();

     xTaskCreate(
                    taskMLX906,          /* Task function. */
                    "taskMLX906",        /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,             /* Parameter passed as input of the task */
                    2,                /* Priority of the task. */
                    NULL);            /* Task handle. */
```

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

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

}

void loop() {
  float sensors[2];
  char dbuff[32];
  Serial.println("in the loop");
  xQueueReceive(squeue,(float *)sensors,10000);
  Serial.print("Temp:");Serial.println(sensors[0]);
  Serial.print("Humi:");Serial.println(sensors[1]);
}
```

A faire :

Choisir les capteurs selon leur disponibilité : température, humidité, luminosité, qualité-d'air, température_d'objet, pression atmosphérique, UV, geste, distance, etc (**max 4**) et modifier le code pour les intégrer dans l'application.

9.8 *Case study* 2 : Un terminal configurable IoT avec la communication LoRa bidirectionnelle

L'exemple présenté ci-dessous permet d'implémenter sur le IoT DevKit une liaison radio configurable LoRa. L'organisation de plusieurs opérations concurrentes telles que la transmission bi-directionnelle par le modem loRa, l'affichage des données en émission et en réception, la lecture des capteurs, .. nécessite la mise en œuvre des mécanismes freeRTOS tels que les **mutex** et les filles d'attente.

Pendant la phase de **`setup()`** l'utilisateur peut définir les paramètres du modem LoRa tels que:

- bande ISM (433 MHz, 868 MHz)
- facteur d'étalement,
- la bande passante du signal,
- la fréquence de base
- et le numéro identificateur du terminal

Le tableau suivant est intégré au code pour calculer la valeur de sensibilité pour les paramètres fournis (SF et SB)

Bandwidth vs Spreading Factor – Sensitivity and Data Rate

		7800	10400	15600	20800	31200	41700	62500	125000	250000	500000
SF6	Sensitivity	-132	-131	-129	-128	-125	-124	-121	-118	-115	-112
	Data Rate	585	780	1170	1560	2344	3128	4688	9375	18750	37500
SF7	Sensitivity	-135	-134	-132	-131	-129	-128	-126	-123	-120	-117
	Data Rate	341	455	683	910	1367	1824	2734	5469	10938	21875
SF8	Sensitivity	-139	-138	-136	-135	-133	-131	-129	-126	-123	-120
	Data Rate	195	260	390	520	781	1367	1563	3125	6250	12500
SF9	Sensitivity	-142	-141	-139	-138	-136	-134	-132	-129	-126	-123
	Data Rate	110	146	219	292	439	586	879	1758	3516	7031
SF10	Sensitivity	-145	-143	-142	-140	-138	-137	-135	-132	-129	-126
	Data Rate	61	81	122	163	244	326	488	977	1953	3906
SF11	Sensitivity	-147	-145	-144	-142	-141	-139	-138	-135	-132	-129
	Data Rate	34	45	67	89	134	179	269	537	1074	2148
SF12	Sensitivity	-149	-148	-146	-145	-143	-142	-140	-137	-134	-131
	Data Rate	18	24	37	49	73	98	146	293	586	1172

L'opération de configuration s'effectue à l'aide d'un simple bouton suivant les messages sur l'écran OLED.
Le code fonctionne avec deux tâches (y compris la tâche principale dans **`loop()`** et une routine d'interruption (ISR) **`onReceive()`**.

L'ISR **`onReceive()`** est réveillée quand une trame LoRa arrive au récepteur. Cette fonction suspend l'exécution des tâches « normales ». Les données reçues par ISR sont envoyées à la file d'attente de messages (**`dqueue`**). Cette *queue* lue par la tâche principale. La valeur RSSI reçue est également envoyée dans une file d'attente séparée. (**`rssiqueue`**) .

La tâche d'envoi qui réçoit les données des capteurs et l'ISR **`onReceive()`** utilisent un sémaphore type **mutex** afin de protéger les transactions sur le bus **I2C**.

Les transactions de la capture des données du capteur ne doivent pas être interrompues par une interruption **`onReceive()`**.

La tâche de fond dans la **`loop()`** lit les messages de **`dqueue`** et de **`rssiqueue`** et les envoie à la fonction d'affichage.
La même fonction d'affichage **`dispData()`** est utilisée pour l'affichage des donnés envoyées par les capteurs.

Afin de ne pas interrompre le processus d'affichage la section critique de cette fonction est également protégée par un sémaphore de type **mutex**.

9.7.1 Les remarques concernant le code principal du terminal

Le code ci-dessous implémente les fonctionnalités principales du terminal.

Il contient:

- 2 tâches - **`loop()`** et **`LoRaSendTask()`**
- 1 routine ISR - **`onReceive()`**
- 2 files d'attente - **`dqueue`** et **`rssiqueue`**
- 2 sémaphores type mutex - **`xSemRecv`** et **`xSemOLED`**

Les données transmisses par le lien radio LoRa son en format suivant :

```
union tspack
  {
    uint8_t frame[128];
    struct packet
      {
        uint8_t head[4];//head[3] length: sensors or text, text takes 1 on MSbit
        uint16_t num;
        uint16_t tout;
        union {
              float sensor[8];
              char text[32];
              } tsdata;
      } pack;
  } sdf,sbf,rdf,rbf;  // data frame and beacon frame
```

9.7.2 Code complet

```
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include <Sodaq_SHT2x.h>
#include <BH1750.h>
BH1750 lightMeter;
// sensitivity table
int senst[5][7] = { // 6     7    8    9   10   11   12   - spreading factor
                    {-125,-129,-133,-136,-138,-142,-143}, //  31250 Hz
                    {-121,-126,-129,-132,-135,-138,-140}, //  62500 Hz
                    {-118,-123,-126,-129,-132,-135,-137}, // 125000 Hz
                    {-115,-120,-123,-126,-129,-132,-134}, // 250000 Hz
                    {-112,-117,-120,-123,-126,-129,-131}, // 500000 Hz
};

#define SCK     5    // GPIO5  -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    27   // GPIO27 -- SX127x's MOSI
#define SS      18   // GPIO18 -- SX127x's CS
#define RST     14   // GPIO14 -- SX127x's RESET
#define DI0     26   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16); // data, clock, reset
#include "introd.h"  // configuration function
int rssi;
QueueHandle_t bqueue, dqueue, rssiqueue;  // queues for beacons and data frames
SemaphoreHandle_t xSemRecv,xSemOLED;
int sdcount=0, rdcount=0;
```

```
union tspack
  {
    uint8_t frame[128];
    struct packet
      {
        uint8_t head[4];//head[3] length : sensors or text, text takes 1 on MSbit
        uint16_t num;
        uint16_t tout;
        union {
              float sensor[8];
              char text[32];
              } tsdata;
      } pack;
  } sdf,sbf,rdf,rbf;  // data frame and beacon frame

 union {
   struct
     {
     int rssi;
     uint8_t dev;
     } pack;
   uint8_t buff_rssi[5];
} rurssi,wurssi;

void dispLabel()
{
  u8x8.clear();
  u8x8.draw2x2String(0,0,"Hightech");
  delay(1000);
  u8x8.draw2x2String(3,2,"SPIE");
  delay(1000);
  u8x8.draw2x2String(4,4,"LoRa");
  delay(1000);
  u8x8.draw2x2String(5,6,"link");
  delay(3000);
  u8x8.clear();
}

void dispDatarate(int dr,int sf, int sb)
{
  char dbuf[16];int sbind=0;int sensitivity=0;
  u8x8.clear();
  sprintf(dbuf,"DR=%5.5d bps",dr);
  u8x8.drawString(0,1,dbuf);
  delay(1000);
  u8x8.drawString(0,2,"CR=4/8");
  delay(1000);
  u8x8.drawString(0,3,"CRC check set");
  switch (sb)
    {
    case 31250: sbind=0;break;
    case 62500: sbind=1;break;
    case 125000: sbind=2;break;
    case 250000: sbind=3;break;
```

```
    case 500000: sbind=4;break;
    default: sbind=0;break;
    }
  Serial.println(sf-6);Serial.println(sbind);
  delay(1000);
  sensitivity=senst[sbind][sf-6];
  sprintf(dbuf,"Sens=%3.3d dBm",sensitivity);
  u8x8.drawString(0,4,dbuf);
  delay(1000);
  u8x8.drawString(0,6,"Packet size 24B");
  delay(3000);
  u8x8.clear();
}

int bitrate(int psf, int psb)  // by default cr=4
{
  int res;
  res= (int) ((float)psf*(float)psb/(2.0*pow(2,psf)));
  return res;
}

void dispData(int sr, int rssi, uint8_t dev)  // sr=1 - send, sr=0 - receive
{
  char dbuf[32];
  xSemaphoreTake(xSemOLED,2000);
  u8x8.clear();memset(dbuf,0x00,32);
  if(sr) sprintf(dbuf,"SPacket:%5.5d",sdcount);
  else sprintf(dbuf,"RPacket:%5.5d",rdcount);
  u8x8.drawString(0, 0,dbuf);memset(dbuf,0x00,32);
  if(sr) sprintf(dbuf,"SDevice:%5.5d",(int)sdf.pack.head[1]);
  else sprintf(dbuf,"RDevice:%5.5d",(int)rdf.pack.head[1]);
  u8x8.drawString(0, 1,dbuf); memset(dbuf,0x00,32);
  if(sr) sprintf(dbuf,"Temp:%2.2f",sdf.pack.tsdata.sensor[0]);
  else sprintf(dbuf,"Temp:%2.2f",rdf.pack.tsdata.sensor[0]);
  u8x8.drawString(0, 3,dbuf);memset(dbuf,0x00,32);
  if(sr) sprintf(dbuf,"Humi:%2.2f",sdf.pack.tsdata.sensor[1]);
  else sprintf(dbuf,"Humi:%2.2f",rdf.pack.tsdata.sensor[1]);
  u8x8.drawString(0, 4,dbuf);memset(dbuf,0x00,32);
  if(sr) sprintf(dbuf,"Lumi:%5.5d",(int)sdf.pack.tsdata.sensor[2]);
  else sprintf(dbuf,"Lumi:%5.5d",(int)rdf.pack.tsdata.sensor[2]);
  u8x8.drawString(0, 5,dbuf);memset(dbuf,0x00,32);
  if(!sr) sprintf(dbuf,"Srssi:%3.3d/%2.2d",(int)rdf.pack.tsdata.sensor[3],
(int)rdf.pack.tout);
  u8x8.drawString(0, 6,dbuf); memset(dbuf,0x00,32);
  if(!sr) sprintf(dbuf,"Rrssi:%3.3d/%2.2d",rssi,dev);
  else sprintf(dbuf,"                ");
  u8x8.drawString(0, 7,dbuf);
  if(!sr) delay(3000);
  xSemaphoreGive(xSemOLED);
}
```

```
void LoRaSendTask( void * pvParameters )
{
int i=0;
while(1)
  {
  xSemaphoreTake(xSemRecv,2000);
  LoRa.beginPacket();
  sdf.pack.head[0]= 0xff;  // destination term - broadcast
  sdf.pack.head[1]= devID; // source term 3
  sdf.pack.head[2]= 0xf0;  // field mask - filed1,field2, field3
  sdf.pack.head[3]= 0x10;  // data size 16 bytes
  sdf.pack.num= (uint16_t) sdcount;
  sdf.pack.tout= (uint16_t) rurssi.pack.dev; // timeout 0 - here dev of rssi
  sdf.pack.tsdata.sensor[2] = (float)lightMeter.readLightLevel();
  sdf.pack.tsdata.sensor[0] = SHT2x.GetTemperature();
  sdf.pack.tsdata.sensor[1] = SHT2x.GetHumidity();
  sdf.pack.tsdata.sensor[3]=(float)rurssi.pack.rssi;
  LoRa.write(sdf.frame,24);
  LoRa.endPacket();
  Serial.print("Temp:");Serial.println(sdf.pack.tsdata.sensor[0]);
  Serial.print("Humi:");Serial.println(sdf.pack.tsdata.sensor[1]);
  Serial.print("Lumi:");Serial.println(sdf.pack.tsdata.sensor[2]);
  Serial.print("RSSI:");Serial.println(sdf.pack.tsdata.sensor[3]);
  if(sdcount<64000) sdcount++; else sdcount=0;
  dispData(1,0,0);
  xSemaphoreGive(xSemRecv);
  LoRa.receive();
  delay(10000+random(10000+5000*(int)devID));
  }
}

void onReceive(int packetSize)
{
int i=0,rssi=0;
uint8_t rdbuff[24], rbbuff[8];
 xSemaphoreTake(xSemRecv,2000);
  if (packetSize == 0) return;   // if there's no packet, return
  i=0;
  if (packetSize==24)
    {
    while (LoRa.available()) {
      rdbuff[i]=LoRa.read();i++;
      }
      wurssi.pack.rssi=LoRa.packetRssi();
      wurssi.pack.dev=rdbuff[1];
      xQueueReset(rssiqueue); // to keep only the last element
      xQueueSend(rssiqueue, wurssi.buff_rssi, portMAX_DELAY);
      if(rdcount<64000) rdcount++; else rdcount=0;
      xQueueReset(dqueue); // to keep only the last element
      xQueueSend(dqueue, rdbuff, portMAX_DELAY);
    }
   xSemaphoreGive(xSemRecv);
}
```

```
void setup()
{
int sf,sb; long freq; int br=0;
Serial.begin(9600);
pinMode(buttonPin, INPUT_PULLUP);
u8x8.begin(); u8x8.setFont(u8x8_font_chroma48medium8_r);
dispLabel();
introd(&sf,&sb,&freq); // configuration function
br=bitrate(sf,sb); dispDatarate(br,sf,sb);
Serial.println(sf);Serial.println(sb);Serial.println(freq);Serial.println(br);
pinMode(26, INPUT);  // recv interrupt
SPI.begin(SCK,MISO,MOSI,SS);
LoRa.setPins(SS,RST,DI0);
if (!LoRa.begin(freq)) {
  Serial.println("Starting LoRa failed!");
  u8x8.drawString(0, 1, "Starting LoRa failed!");
  while (1);
  }
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
LoRa.setCodingRate4(8);
LoRa.enableCrc();
Wire.begin();
lightMeter.begin();
dqueue = xQueueCreate(8, 24); // queue for 4 LoRaTS data frames
rssiqueue = xQueueCreate(4, 5);  // queue for rssi and devID
xSemRecv = xSemaphoreCreateMutex();
xSemOLED = xSemaphoreCreateMutex();
xTaskCreate(
                    LoRaSendTask,   /* 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);
  Serial.println("LoRaSendTask created...");
  LoRa.onReceive(onReceive);
  LoRa.receive();
  delay(3000);
}

void loop() {
 int rec_rssi;
 xQueuePeek(dqueue, rdf.frame, 30000); //portMAX_DELAY)
 xQueuePeek(rssiqueue, rurssi.buff_rssi, 10000);
 Serial.print("RSSI from queue:");Serial.println(rurssi.pack.rssi);
 dispData(0,rurssi.pack.rssi,rurssi.pack.dev);
 Serial.println(rdf.pack.head[2]);
 Serial.println(rdf.pack.tsdata.sensor[0]);
 Serial.println(rdf.pack.tsdata.sensor[1]);
 Serial.println(rdf.pack.tsdata.sensor[2]);
 Serial.println(rdf.pack.tsdata.sensor[3]);Serial.println();
 delay(16000);
}
```

9.7.3 Fonction de configuration : `introd(&sf,&sb,&freq);`

```
const int buttonPin = 0;    // 17 for wemos shield, 0 for PRG
long ISMband;
uint8_t devID;

uint8_t setdevID()
{
int did=1;
int  buttonState;
u8x8.clear();
char dbuf[32];
u8x8.drawString(0, 0,"Setting Dev ID");
while(1)
  {
  buttonState = digitalRead(buttonPin);
  if(buttonState) { did++; if(did==24) did=1;
                    Serial.println(did);
                    sprintf(dbuf,"devID=%2.2d",did);
                    u8x8.drawString(0, 2,dbuf);
                  }
  else
     {
      Serial.println(did);
      u8x8.drawString(0, 4,"Device ID");
      sprintf(dbuf,"set to %2.2d",did);
             u8x8.drawString(0, 5,dbuf);
      return did;
      }
  delay(3000);
  }
}

long setISMband()
{
int count=0;
int  buttonState;
char dbuf[32];
u8x8.clear();
u8x8.drawString(0, 0,"Set ISM band");
while(1)
  {
  buttonState = digitalRead(buttonPin);
  if(buttonState)
    {
      if(!count)count=1;else count=0;
      Serial.println(count);
      if(count) u8x8.drawString(4, 3,"868 MHz");
      else u8x8.drawString(4, 3,"433 MHz");
    }
  else
    {
    if(count) u8x8.drawString(0, 3,"Set to 868 MHz");
    else u8x8.drawString(0, 3,"Set to 433 MHz");
    if(count) return 868E6;
    else return 433E6;
    }
  delay(3000);
  }
}

int setparam()
{
int count=0;
```

```
int  buttonState;
char dbuf[32];
u8x8.clear();
u8x8.drawString(0, 0,"Set Parameters");
while(1)
  {
  buttonState = digitalRead(buttonPin);
  if(buttonState)
    {
      if(!count)count=1;else count=0;
      Serial.println(count);
      if(count) u8x8.drawString(0, 1,"YES");
      else u8x8.drawString(0, 1,"NO ");
  }
  else { Serial.println(count);
        if(count)
          {
            u8x8.drawString(0, 2,"Starting to set");
            u8x8.drawString(0, 4,"Spreading Factor");
            u8x8.drawString(0, 5,"Signal bandwidth");
            u8x8.drawString(0, 6,"Frequency");
          }
        else
            {
            u8x8.drawString(0, 2,"Setting defaults");
            u8x8.drawString(0, 4,"SF=8");
            u8x8.drawString(0, 5,"SB=125 KHz");
            u8x8.drawString(0, 6,"freq=433 MHz");
            }
  return count;
  }
  delay(3000);
  }
}

int setsf()
{
int sf=5;
int  buttonState;
u8x8.clear();
char dbuf[32];
u8x8.drawString(0, 0,"Setting SF");
while(1)
  {
  buttonState = digitalRead(buttonPin);
  if(buttonState) { sf++; if(sf==13) sf=6;
                    Serial.println(sf);
                    sprintf(dbuf,"SF=%2.2d",sf);
                    u8x8.drawString(0, 2,dbuf);
                  }
  else
     {
      Serial.println(sf);
      u8x8.drawString(0, 4,"Spreading Factor");
      sprintf(dbuf,"set to %2.2d",sf);
             u8x8.drawString(0, 5,dbuf);
      return sf;
      }
  delay(3000);
  }
}
```

```
int setsb()
{
const int inval=31250;
int sb=inval; int i=0;
int  buttonState;
u8x8.clear();
char dbuf[32];
u8x8.drawString(0, 0,"Setting SB");
while(1)
  {
  buttonState = digitalRead(buttonPin);
  if(buttonState)
  {
    sb=inval*pow(2,i);i++;
    Serial.println(sb);
    sprintf(dbuf,"SB=%d Hz",sb);
    u8x8.drawString(0, 2,dbuf);
    if(sb==500000) i=0;
  }
  else { Serial.println(sb);
           u8x8.drawString(0, 4,"Signal Band set");
           sprintf(dbuf,"to %d Hz",sb);
           u8x8.drawString(0, 5,dbuf);return sb;
           }
  delay(3000);
  }
}

long setfreq()
{
long inval=ISMband;
int freq=inval; int i=0;
int  buttonState;
u8x8.clear();
char dbuf[32];
u8x8.drawString(0, 0,"Setting Freq");
while(1)
  {
  buttonState = digitalRead(buttonPin);
  if(buttonState)
  {
    freq=inval+250E3*i;i++;
    Serial.println(freq);
    sprintf(dbuf,"Freq=%d KHz",freq/1000);
    u8x8.drawString(0, 2,dbuf);
    if(freq==435E6) i=0;
  }
  else { Serial.println(freq);
         u8x8.drawString(0, 4,"Frequency set");
         sprintf(dbuf,"to %d KHz",freq/1000);
         u8x8.drawString(0, 5,dbuf);
             return freq;}
  delay(3000);
  }
}

void introd(int *psf,int *psb, long *pfreq)
{
int sf,sb; long freq;
int par=0; // default no configuration
ISMband=setISMband();
delay(5000);
devID=setdevID();
delay(5000);
par=setparam();
```

```
delay(5000);
Serial.println(par);
if(par)
{
sf=setsf();
delay(5000);
Serial.println(sf);
sb=setsb();
Serial.println(sb);
delay(5000);
freq=setfreq();
Serial.println(freq);
delay(5000);
}
else
  Serial.println("defaults");
if(!par)
  {
    sf=8;sb=125000;freq=ISMband;
  }
*psf= sf; *psb=sb; *pfreq=ISMband;
}
```

A faire

- Analyser le code ci-dessus.
- Dans le code principal adapter les capteurs disponibles.
- Dans la fonction de configuration intégrer les paramètres **`devID`** et **`ISMband`**

Réseaux LoRa en mode *mesh*

- Tester ensemble avec plusieurs terminaux et les mêmes paramètres LoRa - r

Table of Contents